



当前位置: Java 技术驿站 (<http://cmsblogs.com>) > 死磕Java (<http://cmsblogs.com/?cat=189>) > 死磕 Spring (<http://cmsblogs.com/?cat=206>) > 正文

## 【死磕 Spring】—— IOC 之解析自定义标签 (<http://cmsblogs.com/?p=2841>)

2018-09-27 分类: 死磕 Spring (<http://cmsblogs.com/?cat=206>) 阅读(6539) 评论(1)

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

在博客【死磕Spring】----- IOC 之 注册 BeanDefinition (<http://cmsblogs.com/?p=2697>) 中提到: 获取 Document 对象后, 会根据该对象和 Resource 资源对象调用 registerBeanDefinitions() 方法, 开始注册 BeanDefinitions 之旅。在注册 BeanDefinitions 过程中会调用 parseBeanDefinitions() 开启 BeanDefinition 的解析过程。在该方法中, 它会根据命名空间的不同调用不同的方法进行解析, 如果是默认的命名空间, 则调用 parseDefaultElement() 进行默认标签解析, 否则调用 parseCustomElement() 方法进行自定义标签解析。前面 6 篇博客都是分析默认标签的解析工作, 这篇博客分析自定义标签的解析过 默认标签的解析博客如下:

1. 【死磕 Spring】—— IOC 之解析Bean: 解析 import 标签 (<http://cmsblogs.com/?p=2724>)
2. 【死磕 Spring】—— IOC 之解析 bean 标签: 开启解析进程 (<http://cmsblogs.com/?p=2731>)
3. 【死磕 Spring】—— IOC 之解析 bean 标签: BeanDefinition (<http://cmsblogs.com/?p=2734>)
4. 【死磕 Spring】—— IOC 之解析 bean 标签: meta、lookup-method、replace-method (<http://cmsblogs.com/?p=2736>)
5. 【死磕 Spring】—— IOC 之解析 bean 标签: constructor-arg、property 子元素 (<http://cmsblogs.com/?p=2754>)
6. 【死磕 Spring】—— IOC 之解析 bean 标签: 解析自定义标签 (<http://cmsblogs.com/?p=2756>)
7. 【死磕 Spring】—— IOC 之注册解析的 BeanDefinition (<http://cmsblogs.com/?p=2763>)

在分析自定义标签的解析之前, 我们有必要了解自定义标签的使用。

### 使用自定义标签

扩展 Spring 自定义标签配置一般需要以下几个步骤:

1. 创建一个需要扩展的组件
2. 定义一个 XSD 文件, 用于描述组件内容
3. 创建一个实现 AbstractSingleBeanDefinitionParser 接口的类, 用来解析 XSD 文件中的定义和组件定义
4. 创建一个 Handler, 继承 NamespaceHandlerSupport, 用于将组件注册到 Spring 容器
5. 编写 Spring.handlers 和 Spring.schemas 文件

下面就按照上面的步骤来实现一个自定义标签组件。 **创建组件** 该组件就是一个普通的 JavaBean, 没有任何特别之处。

```
public class User {  
    private String id;  
  
    private String userName;  
  
    private String email;  
}
```





## 定义 XSD 文件

```
<?xml version="1.0" encoding="UTF-8"?>  
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"  
    xmlns="http://www.cmsblogs.com/schema/user" targetNamespace="http://www.cmsblogs.com/schema/u  
ser"  
    elementFormDefault="qualified">  
    <xsd:element name="user">  
        <xsd:complexType>  
            <xsd:attribute name="id" type="xsd:string" />  
            <xsd:attribute name="userName" type="xsd:string" />  
            <xsd:attribute name="email" type="xsd:string" />  
        </xsd:complexType>  
    </xsd:element>  
</xsd:schema>
```

上面除了对 User 这个 JavaBean 进行了描述外，还定义了 `xmlns="http://www.cmsblogs.com/schema/user"` `targetNamespace="http://www.cmsblogs.com/schema/user"` 这两个值，这两个值在后面是有大作用的。

**Parser 类** 定义一个 Parser 类，该类继承 `AbstractSingleBeanDefinitionParser`，并实现 `getBeanClass()` 和 `doParse()` 两个方法。主要是用于解析 XSD 文件中的定义和组件定义。


```

public class UserDefinitionParser extends AbstractSingleBeanDefinitionParser {

    @Override
    protected Class<?> getBeanClass(Element element) {
        return User.class;
    }

    @Override
    protected void doParse(Element element, BeanDefinitionBuilder builder) {
        String id = element.getAttribute("id");
        String userName=element.getAttribute("userName");
        String email=element.getAttribute("email");
        if(StringUtils.hasText(id)){
            builder.addPropertyValue("id",id);
        }
        if(StringUtils.hasText(userName)){
            builder.addPropertyValue("userName", userName);
        }
        if(StringUtils.hasText(email)){
            builder.addPropertyValue("email", email);
        }
    }
}

```

**Handler 类** 定义 Handler 类，继承 NamespaceHandlerSupport ,主要目的是将组件注册到 Spring 容器中。

```

public class UserNamespaceHandler extends NamespaceHandlerSupport {

    @Override
    public void init() {
        registerBeanDefinitionParser("user",new UserDefinitionParser());
    }
}

```




## Spring.handlers

<http://www.cmsblogs.com/schema/user=org.springframework.core.customelement.UserNamespaceHandler>

## Spring.schemas

<http://www.cmsblogs.com/schema/user.xsd=user.xsd>

经过上面几个步骤，就可以使用自定义的标签了。在 xml 配置文件中使用如下：



```
<?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:myTag="http://www.cmsblogs.com/schema/user"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
http://www.cmsblogs.com/schema/user http://www.cmsblogs.com/schema/user.xsd">

    <myTag:user id="user" email="12233445566@qq.com" userName="chenssy" />

</beans>
```

测试:

```
public static void main(String[] args){
    ApplicationContext context = new ClassPathXmlApplicationContext("spring.xml");

    User user = (User) context.getBean("user");

    System.out.println(user.getUserName() + "----" + user.getEmail());
}
}
```

运行结果:

(<https://gitee.com/chenssy/blog-home/raw/master/image/201810/spring-201807110001.png>)

## 解析自定义标签

上面已经演示了 Spring 自定义标签的使用，下面就来分析自定义标签的解析过程。DefaultBeanDefinitionDocumentReader.parseBeanDefinitions() 负责标签的解析工作，其中它根据命名空间的不同进行不同标签的解析，其中自定义标签由 delegate.parseCustomElement() 实现。如下：

```
public BeanDefinition parseCustomElement(Element ele) {
    return parseCustomElement(ele, null);
}
```

调用 parseCustomElement() 方法，如下：



```

public BeanDefinition parseCustomElement(Element ele, @Nullable BeanDefinition containingBd) {
    // 获取 namespaceUri
    String namespaceUri = getNamespaceURI(ele);
    if (namespaceUri == null) {
        return null;
    }

    // 根据 namespaceUri 获取相应的 Handler
    NamespaceHandler handler = this.readerContext.getNamespaceHandlerResolver().resolve(namespaceUri
);
    if (handler == null) {
        error("Unable to locate Spring NamespaceHandler for XML schema namespace [" + namespaceUri +
        "]", ele);
        return null;
    }

    // 调用自定义的 Handler 处理
    return handler.parse(ele, new ParserContext(this.readerContext, this, containingBd));
}

```



处理过程分为三步：

1. 获取 namespaceUri
2. 根据 namespaceUri 获取相应的 Handler
3. 调用自定义的 Handler 处理

这个处理过程很简单明了，根据 namespaceUri 获取 Handler，这个映射关系我们在 Spring.handlers 中已经定义了，所以只需要找到该类，然后初始化返回，最后调用该 Handler 对象的 parse() 方法处理，该方法我们也提供了实现。所以上面的核心就在于怎么找到该 Handler 类。调用方法为：

```
this.readerContext.getNamespaceHandlerResolver().resolve(namespaceUri)
```

getNamespaceHandlerResolver() 方法返回的命名空间的解析器，该解析定义在 XmlReaderContext 中，如下：

```

public final NamespaceHandlerResolver getNamespaceHandlerResolver() {
    return this.namespaceHandlerResolver;
}

```

这里直接返回，那是在什么时候初始化的呢？这里需要回退到博文：【死磕Spring】----- IOC 之 注册 BeanDefinition ()，在这篇博客中提到在注册 BeanDefinition 时，首先是通过 createBeanDefinitionDocumentReader() 获取 Document 解析器 BeanDefinitionDocumentReader 实例，然后调用该实例 registerBeanDefinitions() 方法进行注册。registerBeanDefinitions() 方法需要提供两个参数，一个是 Document 实例 doc，一个是 XmlReaderContext 实例 readerContext，readerContext 实例对象由 createReaderContext() 方法提供。namespaceHandlerResolver 实例对象就是在这个时候初始化的。如下：



```
public XmlReaderContext createReaderContext(Resource resource) {  
    return new XmlReaderContext(resource, this.problemReporter, this.eventListener,  
        this.sourceExtractor, this, getNamespaceHandlerResolver());  
}
```



XmlReaderContext 构造函数中最后一个参数就是 NamespaceHandlerResolver 对象，该对象由 getNamespaceHandlerResolver() 提供，如下：

```
public NamespaceHandlerResolver getNamespaceHandlerResolver() {  
    if (this.namespaceHandlerResolver == null) {  
        this.namespaceHandlerResolver = createDefaultNamespaceHandlerResolver();  
    }  
    return this.namespaceHandlerResolver;  
}  
  
protected NamespaceHandlerResolver createDefaultNamespaceHandlerResolver() {  
    ClassLoader cl = (getResourceLoader() != null ? getResourceLoader().getClassLoader() : getBeanClassLoader());  
    return new DefaultNamespaceHandlerResolver(cl);  
}
```

所以 getNamespaceHandlerResolver().resolve(namespaceUri) 调用的就是 DefaultNamespaceHandlerResolver 的 resolve()。如下：



```

public NamespaceHandler resolve(String namespaceUri) {
    // 获取所有已经配置的 Handler 映射
    Map<String, Object> handlerMappings = getHandlerMappings();

    // 根据 namespaceUri 获取 handler 的信息：这里一般都是类路径
    Object handlerOrClassName = handlerMappings.get(namespaceUri);
    if (handlerOrClassName == null) {
        return null;
    }
    else if (handlerOrClassName instanceof NamespaceHandler) {
        // 如果已经做过解析，直接返回
        return (NamespaceHandler) handlerOrClassName;
    }
    else {
        String className = (String) handlerOrClassName;
        try {

            Class<?> handlerClass = ClassUtils.forName(className, this.classLoader);
            if (!NamespaceHandler.class.isAssignableFrom(handlerClass)) {
                throw new FatalBeanException("Class [" + className + "] for namespace [" + namespaceU
ri +
                "] does not implement the [" + NamespaceHandler.class.getName() + "] interfac
e");
            }

            // 初始化类
            NamespaceHandler namespaceHandler = (NamespaceHandler) BeanUtils.instantiateClass(handler
Class);

            // 调用 init() 方法
            namespaceHandler.init();

            // 记录在缓存
            handlerMappings.put(namespaceUri, namespaceHandler);
            return namespaceHandler;
        }
        catch (ClassNotFoundException ex) {
            throw new FatalBeanException("Could not find NamespaceHandler class [" + className +
                "] for namespace [" + namespaceUri + "]", ex);
        }
        catch (LinkageError err) {
            throw new FatalBeanException("Unresolvable class definition for NamespaceHandler class ["
+
                className + "] for namespace [" + namespaceUri + "]", err);
        }
    }
}

```

首先调用 `getHandlerMappings()` 获取所有配置文件中的映射关系 `handlerMappings`，该关系为 <命名空间, 类路径>，然后根据命名空间 `namespaceUri` 从映射关系中获取相应的信息，如果为空或者已经初始化了就直

接返回，否则根据反射对其进行初始化，同时调用其 `init()` 方法，最后将该 `Handler` 对象缓存。 `init()` 方法主要是将自定义标签解析器进行注册，如我们自定义的 `init()`：

```
@Override
public void init() {
    registerBeanDefinitionParser("user", new UserDefinitionParser());
}
```

直接调用父类的 `registerBeanDefinitionParser()` 方法进行注册：

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

其实就是将映射关系放在一个 `Map` 结构的 `parsers` 对象中：`private final Map<String, BeanDefinitionParser> parsers`。完成后返回 `NamespaceHandler` 对象，然后调用其 `parse()` 方法开始自定义标签的解析，如下：

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

调用 `findParserForElement()` 方法获取 `BeanDefinitionParser` 实例，其实就是获取在 `init()` 方法里面注册的实例对象。如下：

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

获取 `localName`，在上面的例子中就是：`user`，然后从 `Map` 实例 `parsers` 中获取 `BeanDefinitionParser` 对象。返回 `BeanDefinitionParser` 对象后，调用其 `parse()`，该方法在 `AbstractBeanDefinitionParser` 中实现：





```

public final BeanDefinition parse(Element element, ParserContext parserContext) {
    AbstractBeanDefinition definition = parseInternal(element, parserContext);
    if (definition != null && !parserContext.isNested()) {
        try {
            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()) {
                String name = element.getAttribute(NAME_ATTRIBUTE);
                if (StringUtils.hasLength(name)) {
                    aliases = StringUtils.trimArrayElements(StringUtils.commaDelimitedListToStringArray(name));
                }
            }
            BeanDefinitionHolder holder = new BeanDefinitionHolder(definition, id, aliases);
            registerBeanDefinition(holder, parserContext.getRegistry());
            if (shouldFireEvents()) {
                BeanComponentDefinition componentDefinition = new BeanComponentDefinition(holder);
                postProcessComponentDefinition(componentDefinition);
                parserContext.registerComponent(componentDefinition);
            }
        } catch (BeanDefinitionStoreException ex) {
            String msg = ex.getMessage();
            parserContext.getReaderContext().error((msg != null ? msg : ex.toString()), element);
            return null;
        }
    }
    return definition;
}

```

核心在方法 `parseInternal()` 为什么这么说，以为该方法返回的是 `AbstractBeanDefinition` 对象，从前面默认标签的解析工作中我们就可以判断该方法就是将标签解析为 `AbstractBeanDefinition`，且后续代码都是将 `AbstractBeanDefinition` 转换为 `BeanDefinitionHolder`，所以真正的解析工作都交由 `parseInternal()` 实现，如下：





```
protected final AbstractBeanDefinition parseInternal(Element element, ParserContext parserContext) {
    // 获取
    BeanDefinitionBuilder builder = BeanDefinitionBuilder.genericBeanDefinition();

    // 获取父类元素
    String parentName = getParentName(element);
    if (parentName != null) {
        builder.getRawBeanDefinition().setParentName(parentName);
    }

    // 获取自定义标签中的 class, 这个时候会去调用自定义解析中的 getBeanClass()
    Class<?> beanClass = getBeanClass(element);
    if (beanClass != null) {
        builder.getRawBeanDefinition().setBeanClass(beanClass);
    }
    else {
        // beanClass 为 null, 意味着子类并没有重写 getBeanClass() 方法, 则尝试去判断是否重写了 getBeanClassName()
        String beanClassName = getBeanClassName(element);
        if (beanClassName != null) {
            builder.getRawBeanDefinition().setBeanClassName(beanClassName);
        }
    }
    builder.getRawBeanDefinition().setSource(parserContext.extractSource(element));
    BeanDefinition containingBd = parserContext.getContainingBeanDefinition();
    if (containingBd != null) {
        // Inner bean definition must receive same scope as containing bean.
        builder.setScope(containingBd.getScope());
    }
    if (parserContext.isDefaultLazyInit()) {
        // Default-lazy-init applies to custom bean definitions as well.
        builder.setLazyInit(true);
    }

    // 调用子类的 doParse() 进行解析
    doParse(element, parserContext, builder);
    return builder.getBeanDefinition();
}
```

在该方法中我们主要关注两个方法： `getBeanClass()` 、 `doParse()` 。对于 `getBeanClass()` 方法， `AbstractSingleBeanDefinitionParser` 类并没有提供具体实现，而是直接返回 `null`，意味着它希望子类能够重写该方法，当然如果没有重写该方法，这会去调用 `getBeanClassName()`，判断子类是否已经重写了该方法。对于 `doParse()` 则是直接空实现。所以对于 `parseInternal()` 而言它总是期待它的子类能够实现 `getBeanClass()`、`doParse()`，其中 `doParse()` 尤为重要，如果你不提供实现，怎么来解析自定义标签呢？最后将自定义的解析器： `UserDefinitionParser` 再次回观。

```
public class UserDefinitionParser extends AbstractSingleBeanDefinitionParser {

    @Override
    protected Class<?> getBeanClass(Element element) {
        return User.class;
    }

    @Override
    protected void doParse(Element element, BeanDefinitionBuilder builder) {
        String id = element.getAttribute("id");
        String userName=element.getAttribute("userName");
        String email=element.getAttribute("email");
        if(StringUtils.hasText(id)){
            builder.addPropertyValue("id",id);
        }
        if(StringUtils.hasText(userName)){
            builder.addPropertyValue("userName", userName);
        }
        if(StringUtils.hasText(email)){
            builder.addPropertyValue("email", email);
        }
    }
}
```

至此，自定义标签的解析过程已经分析完成了。其实整个过程还是较为简单：首先会加载 handlers 文件，将其中内容进行一个解析，形成 <namespaceUri,类路径> 这样的一个映射，然后根据获取的 namespaceUri 就可以得到相应的类路径，对其进行初始化等到相应的 Handler 对象，调用 parse() 方法，在该方法中根据标签的 localName 得到相应的 BeanDefinitionParser 实例对象，调用 parse()，该方法定义在 AbstractBeanDefinitionParser 抽象类中，核心逻辑封装在其 parseInternal() 中，该方法返回一个 AbstractBeanDefinition 实例对象，其主要是在 AbstractSingleBeanDefinitionParser 中实现，对于自定义的 Parser 类，其需要实现 getBeanClass() 或者 getBeanClassName() 和 doParse()。

(<https://gitee.com/chenssy/blog-home/raw/master/image/201810/spring-201807151001.png>)

👍 赞(14)

¥ 打赏

【公告】版权声明 ([http://cmsblogs.com/?page\\_id=1908](http://cmsblogs.com/?page_id=1908))

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Java (<http://cmsblogs.com/?tag=%E6%AD%BB%E7%A3%95java>)

死磕Spring (<http://cmsblogs.com/?tag=%E6%AD%BB%E7%A3%95spring>)

👤 chenssy (<http://cmsblogs.com/?author=1>)

不想当厨师的程序员不是好的架构师....

[上一篇](#)[下一篇](#)

【死磕 Spring】—— IOC 之解析 bean 标签：解析自定义标签 (<http://cmsblogs.com/?p=2756>)

一个致命的 Redis 命令，导致公司损失 400 万! (<http://cmsblogs.com/?p=2765>)

- 【死磕 Redis】—— 如何排查 Redis 中的慢查询 (<http://cmsblogs.com/?p=18352>)
- 【死磕 Redis】—— 发布与订阅 (<http://cmsblogs.com/?p=18348>)
- 【死磕 Redis】—— 布隆过滤器 (<http://cmsblogs.com/?p=18346>)
- 【死磕 Redis】—— 理解 pipeline 管道 (<http://cmsblogs.com/?p=18344>)
- 【死磕 Redis】—— 事务 (<http://cmsblogs.com/?p=18340>)
- 【死磕 Redis】—— Redis 的线程模型 (<http://cmsblogs.com/?p=18337>)
- 【死磕 Redis】—— Redis 通信协议 RESP (<http://cmsblogs.com/?p=18334>)
- 【死磕 Redis】—— 开篇 (<http://cmsblogs.com/?p=18332>)
- 【死磕 Spring】—— IOC 总结 (<http://cmsblogs.com/?p=4047>)
- 【死磕 Spring】—— 4 张图带你读懂 Spring IOC 的世界 (<http://cmsblogs.com/?p=4045>)
- 【死磕 Spring】—— 深入分析 ApplicationContext 的 refresh() (<http://cmsblogs.com/?p=4043>)
- 【死磕 Spring】—— ApplicationContext 相关接口架构分析 (<http://cmsblogs.com/?p=4036>)
- 【死磕 Spring】—— IOC 之 分析 bean 的生命周期 (<http://cmsblogs.com/?p=4034>)
- 【死磕 Spring】—— Spring 的环境&属性: PropertySource、Environment、Profile (<http://cmsblogs.com/?p=4032>)
- 【死磕 Spring】—— IOC 之 BeanDefinition 注册机: BeanDefinitionRegistry (<http://cmsblogs.com/?p=4026>)

