



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 bean 标签: meta、lookup-method、replace-method (<http://cmsblogs.com/?p=2736>)

2018-09-19 分类: 死磕 Spring (<http://cmsblogs.com/?cat=206>) 阅读(8682) 评论(0)

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

在上篇博客【死磕Spring】----- IOC 之解析 Bean 标签: BeanDefinition (<http://cmsblogs.com/?p=2734>) 中已经完成了对 Bean 标签属性的解析工作, 这篇博文开始分析子元素的解析。完成 Bean 标签基本属性解析后, 会依次调用 `parseMetaElements()`、`parseLookupOverrideSubElements()`、`parseReplacedMethodSubElements()` 对子元素 meta、lookup-method、replace-method 完成解析。三个子元素的作用如下:

- meta: 元数据。
- lookup-method: Spring 动态改变 bean 里方法的实现。方法执行返回的对象, 使用 Spring 内原有的这类对象替换, 通过改变方法返回值来动态改变方法。内部实现为使用 cglib 方法, 重新生成子类, 重写配置的方法和返回对象, 达到动态改变的效果。
- replace-method: Spring 动态改变 bean 里方法的实现。需要改变的方法, 使用 Spring 内原有其他类 (需要继承接口 `org.springframework.beans.factory.support.MethodReplacer`) 的逻辑, 替换这个方法。通过改变方法执行逻辑来动态改变方法。

meta 子元素

meta : 元数据。当需要使用里面的信息时可以通过key获取

meta 所声明的 key 并不会在 Bean 中体现, 只是一个额外的声明, 当我们需要使用里面的信息时, 通过 BeanDefinition 的 `getAttribute()` 获取。该子元素的解析过程如下:



```

public void parseMetaElements(Element ele, BeanMetadataAttributeAccessor attributeAccessor) {
    NodeList nl = ele.getChildNodes();
    for (int i = 0; i < nl.getLength(); i++) {
        Node node = nl.item(i);
        if (isCandidateElement(node) && nodeNameEquals(node, META_ELEMENT)) {
            Element metaElement = (Element) node;
            String key = metaElement.getAttribute(KEY_ATTRIBUTE);
            String value = metaElement.getAttribute(VALUE_ATTRIBUTE);
            BeanMetadataAttribute attribute = new BeanMetadataAttribute(key, value);
            attribute.setSource(extractSource(metaElement));
            attributeAccessor.addMetadataAttribute(attribute);
        }
    }
}

```



解析过程较为简单，获取相应的 key - value 构建 BeanMetadataAttribute 对象，然后通过 addMetadataAttribute() 加入到 AbstractBeanDefinition 中。”如下：

```

public void addMetadataAttribute(BeanMetadataAttribute attribute) {
    super.setAttribute(attribute.getName(), attribute);
}

```

委托 AttributeAccessorSupport 实现，如下：

```

public void setAttribute(String name, @Nullable Object value) {
    Assert.notNull(name, "Name must not be null");
    if (value != null) {
        this.attributes.put(name, value);
    }
    else {
        removeAttribute(name);
    }
}

```

AttributeAccessorSupport 是接口 AttributeAccessor 的实现者。AttributeAccessor 接口定义了与其他对象的元数据进行连接和访问的约定，可以通过该接口对属性进行获取、设置、删除操作。设置元数据后，则可以通过 getAttribute() 获取如下：

```

public Object getAttribute(String name) {
    BeanMetadataAttribute attribute = (BeanMetadataAttribute) super.getAttribute(name);
    return (attribute != null ? attribute.getValue() : null);
}

```

lookup-method 子元素

lookup-method：获取器注入，是把一个方法声明为返回某种类型的 bean 但实际要返回的 bean 是在配置文件里面配置的。该方法可以用于设计一些可插拔的功能上，解除程序依赖。



直接上例子：**BeanDefinition中有属性methodOverrides封装了abstract方法名=getCar和返回指向的bean的id=hongqi**

```
public interface Car {

    void display();

}

public class Bmw implements Car{

    @Override
    public void display() {
        System.out.println("我是 BMW");
    }

}

public class Hongqi implements Car{

    @Override
    public void display() {
        System.out.println("我是 hongqi");
    }

}

public abstract class Display {

    public void display(){
        getCar().display();
    }

    public abstract Car getCar();

}

public static void main(String[] args) {

    ApplicationContext context = new ClassPathXmlApplicationContext("classpath:spring.xml");

    Display display = (Display) context.getBean("display");
    display.display();

}
```

配置内容如下：

```
<bean id="display" class="org.springframework.core.test1.Display">
    <lookup-method name="getCar" bean="hongqi"/>
</bean>
```

运行结果为：

如果将 `bean="hognqi"` 替换为 `bean="bmw"`，则运行结果变成：

我是 BMW

看了这个示例，我们初步了解了 `looku-method` 子元素提供的功能了，其解析过程如下：

```
public void parseLookupOverrideSubElements(Element beanEle, MethodOverrides overrides) {
    NodeList nl = beanEle.getChildNodes();
    for (int i = 0; i < nl.getLength(); i++) {
        Node node = nl.item(i);
        if (isCandidateElement(node) && nodeNameEquals(node, LOOKUP_METHOD_ELEMENT)) {
            Element ele = (Element) node;
            String methodName = ele.getAttribute(NAME_ATTRIBUTE);
            String beanRef = ele.getAttribute(BEAN_ELEMENT);
            LookupOverride override = new LookupOverride(methodName, beanRef);
            override.setSource(extractSource(ele));
            overrides.addOverride(override);
        }
    }
}
```

解析过程和 `meta` 子元素没有多大区别，同样是解析 `methodName`、`beanRef` 构造一个 `LookupOverride` 对象，然后覆盖即可。在实例化 `Bean` 的时候，再详细阐述具体的实现过程，这里仅仅只是一个标记作用。

replace-method 子元素

replaced-method：可以在运行时调用新的方法替换现有的方法，还能动态的更新原有方法的逻辑

该标签使用方法和 `lookup-method` 标签差不多，只不过替代方法的类需要实现 `MethodReplacer` 接口。如下：



```
public class Method {  
    public void display(){  
        System.out.println("我是原始方法");  
    }  
}  
  
public class MethodReplace implements MethodReplacer {  
  
    @Override  
    public Object reimplement(Object obj, Method method, Object[] args) throws Throwable {  
        System.out.println("我是替换方法");  
  
        return null;  
    }  
}  
  
public static void main(String[] args) {  
    ApplicationContext context = new ClassPathXmlApplicationContext("classpath:spring.xml");  
  
    Method method = (Method) context.getBean("method");  
    method.display();  
}
```

如果 spring.xml 文件如下:

```
<bean id="methodReplace" class="org.springframework.core.test1.MethodReplace"/>  
  
<bean id="method" class="org.springframework.core.test1.Method"/>
```

则运行结果为:

我是原始方法


增加 replaced-method 子元素:

```
<bean id="methodReplace" class="org.springframework.core.test1.MethodReplace"/>  
  
<bean id="method" class="org.springframework.core.test1.Method">  
    <replaced-method name="display" replacer="methodReplace"/>  
</bean>
```

运行结果为:

我是替换方法

上面代码演示了 replaced-method 子元素的用法, 下面再看看该子元素的解析过程。


```


public void parseReplacedMethodSubElements(Element beanEle, MethodOverrides overrides) {
    NodeList nl = beanEle.getChildNodes();
    for (int i = 0; i < nl.getLength(); i++) {
        Node node = nl.item(i);
        if (isCandidateElement(node) && nodeNameEquals(node, REPLACED_METHOD_ELEMENT)) {
            Element replacedMethodEle = (Element) node;
            String name = replacedMethodEle.getAttribute(NAME_ATTRIBUTE);
            String callback = replacedMethodEle.getAttribute(REPLACER_ATTRIBUTE);
            ReplaceOverride replaceOverride = new ReplaceOverride(name, callback);
            // Look for arg-type match elements.
            List<Element> argTypeEles = DomUtils.getChildElementsByTagName(replacedMethodEle, ARG_TY
E_ELEMENT);
            for (Element argTypeEle : argTypeEles) {
                String match = argTypeEle.getAttribute(ARG_TYPE_MATCH_ATTRIBUTE);
                match = (StringUtils.hasText(match) ? match : DomUtils.getTextValue(argTypeEle));
                if (StringUtils.hasText(match)) {
                    replaceOverride.addTypeIdentifier(match);
                }
            }
            replaceOverride.setSource(extractSource(replacedMethodEle));
            overrides.addOverride(replaceOverride);
        }
    }
}

```

该子元素和 lookup-method 资源的解析过程差不多，同样是提取 name 和 replacer 属性构建 ReplaceOverride 对象，然后记录到 AbstractBeanDefinition 中的 methodOverrides 属性中。对于 lookup-method 和 replaced-method 两个子元素是如何使用以完成他们所提供的功能，在后续实例化 Bean 的时候会做详细说明。

- 【死磕 Spring】—— IOC 之解析Bean：解析 import 标签 (<http://cmsblogs.com/?p=2724>)
- 【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>)

 赞(9)

 打赏

【公告】版权声明 (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:
BeanDefinition (<http://cmsblogs.com/?p=2734>)

设计模式六大原则, 你真的懂了吗?
(<http://cmsblogs.com/?p=2739>)

- 【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>)

