http://daveztong.github.io/2016/11/01/Spring%E5%8F%AF%E6%89%A9%E5%B1%95%E7%9A%84XML%E9%85%8D%E7%BD%AE/

**Spring可扩展的XML配置**

Nov 01, 201635Hits

**Spring 可扩展的XML配置**

Spring 自从2.0开始就为基础的xml格式提供了一个基于xml schema的扩展机制，用于定义和配置beans。本文基于此简单讲解如果定义自己的BeanDefinitionParser和如何将定义好的parsers集成到Spring Icon container中。

创建一个xml配置扩展可以通过以下4步完成：

1. 创建一个xml schema来描述你自定的xml元素。
2. 编写一个Namespace Handler的具体实现。
3. 编写一个或者多个BeanDefinitionParser的实现。主要的工作都在此步骤完成。
4. 关联xsd,NamespaceHandler。

下面依照以上四个步骤并附一个示例详细讲解。完整代码放在码云上:<https://git.oschina.net/android-speeder/springcustomxml.git>

**创建schema文件**

创建一个spring可以用的xml配置扩展首先需要定义一个xml schema来描述这个扩展。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | <?xml version="1.0" encoding="UTF-8"?>  <xsd:schema xmlns="http://daveztong.github.io/schema/dz"  xmlns:xsd="http://www.w3.org/2001/XMLSchema"  xmlns:beans="http://www.springframework.org/schema/beans"  targetNamespace="http://daveztong.github.io/schema/dz"  elementFormDefault="qualified"  attributeFormDefault="unqualified">  <xsd:import namespace="http://www.springframework.org/schema/beans"/>  <xsd:element name="person">  <xsd:complexType>  <xsd:complexContent>  <xsd:extension base="beans:identifiedType">  <xsd:attribute name="name" type="xsd:string" use="required"/>  <xsd:attribute name="age" type="xsd:int"/>  </xsd:extension>  </xsd:complexContent>  </xsd:complexType>  </xsd:element>  </xsd:schema> |

注意<xsd:extension base="beans:identifiedType">表示该tag含有一个id属性并且会被spring container用于识别该bean。使用该属性的前提是需要导入<xsd:import namespace="http://www.springframework.org/schema/beans"/>。

**实现Namespace Handler**

当spring在解析xml时遇到该namespace就需要使用自定义的Namespacehandler去解析所有该名称空间下的元素。

Namespace Handler只包含三个方法:

* init(): 用于初始化Namespace Handler。
* BeanDefinition parse(Element, ParserContext)：当spring遇到顶级的元素时才会调用。该方法可以直接返回一个bean definition或者自己注册一个。
* BeanDefinitionHolder decorate(Node, BeanDefinitionHolder, ParserContext):当spring遇到属性定义或者嵌套在不同名称空间下的元素时才会被调用。

我们可以自己实现Namespace Handler,但是基于spring的惯例，一般都会提供一个基础类简化开发人员的工作，所以我们可以直接继承spring提供的NamespaceHandlerSupport。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | package io.github.daveztong;  import org.springframework.beans.factory.xml.NamespaceHandlerSupport;  /\*\*  \* Created by tangwei on 2016/10/31.  \*/  public class DZNamespaceHandler extends NamespaceHandlerSupport{  @Override  public void init() {  registerBeanDefinitionParser("person",new PersonBeanDefinitionParser());  }  } |

可以看到DZNamespaceHandler的代码量很少，因为真正的解析工作都委托给了BeanDefinitionParser。 Namespace Handler支持注册任意多个BeanDefinitionParser,当其需要解析其所负责的名称空间下的元素时就会将解析工作委托给注册的BeanDefinitionParser。

**实现BeanDefinitionParser**

当Namespace Handler遇到一个与注册列表中key匹配的xml元素时就会将该元素的解析任务交给与key对应的BeanDefinitionParser。在本例中则是PersonBeanDefinitionParser。也就是说一个BeanDefinitionParser仅负责解析定义在指定schema中的一个唯一顶级的xml元素。在parser中我们可以访问到其负责解析的元素和子元素的内容。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29 | package io.github.daveztong;  import org.springframework.beans.factory.support.BeanDefinitionBuilder;  import org.springframework.beans.factory.xml.AbstractSingleBeanDefinitionParser;  import org.springframework.util.StringUtils;  import org.w3c.dom.Element;  /\*\*  \* Created by tangwei on 2016/10/31.  \*/  public class PersonBeanDefinitionParser extends AbstractSingleBeanDefinitionParser {  @Override  protected Class<?> getBeanClass(Element element) {  return Person.class;  }  @Override  protected void doParse(Element element, BeanDefinitionBuilder builder) {  String name = element.getAttribute("name");  if (StringUtils.hasText(name)) {  builder.addPropertyValue("name", name);  }  String age = element.getAttribute("age");  if (StringUtils.hasText(age)) {  builder.addPropertyValue("age", Integer.parseInt(age));  }  }  } |

编码部分就这么多，是不是so easy! 可能有人会疑问，为什么没有看到BeanDefiniiton的创建，那是因为创建的工作由AbstractSingleBeanDefinitionParser#parseInternal完成了。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29 | @Override  protected final AbstractBeanDefinition parseInternal(Element element, ParserContext parserContext) {  BeanDefinitionBuilder builder = BeanDefinitionBuilder.genericBeanDefinition();  String parentName = getParentName(element);  if (parentName != null) {  builder.getRawBeanDefinition().setParentName(parentName);  }  Class<?> beanClass = getBeanClass(element);  if (beanClass != null) {  builder.getRawBeanDefinition().setBeanClass(beanClass);  }  else {  String beanClassName = getBeanClassName(element);  if (beanClassName != null) {  builder.getRawBeanDefinition().setBeanClassName(beanClassName);  }  }  builder.getRawBeanDefinition().setSource(parserContext.extractSource(element));  if (parserContext.isNested()) {  // Inner bean definition must receive same scope as containing bean.  builder.setScope(parserContext.getContainingBeanDefinition().getScope());  }  if (parserContext.isDefaultLazyInit()) {  // Default-lazy-init applies to custom bean definitions as well.  builder.setLazyInit(true);  }  doParse(element, parserContext, builder);  return builder.getBeanDefinition();  } |

The last statement builder.getBeanDefinition();返回了我们需要的bean. 继续看builder.getBeanDefinition()的代码:

|  |  |
| --- | --- |
| 1  2  3  4 | public AbstractBeanDefinition getBeanDefinition() {  this.beanDefinition.validate();  return this.beanDefinition;  } |

That’s our bean!

编码部分是完成了，但是spring还不知道他们的存在，接下来就需要让spring aware of them!(aware是个很关键的词啊，在spring的代码里随处可见^\_^)

**注册handler and schema**

要让spring aware of out handler and xsd schema,我们需要在两个特殊的properties文件中注册他们。这些properties文件需要放在META-INF目录下面,并且可以和jar包一起发布。spring的解析框架通过这些特殊的properties文件就能pick up our handler and schema!

**META-INF/spring.handlers**

spring.handlers包含了 xml schema uri 和namespace handler之间的k-v映射关系，如下:

|  |  |
| --- | --- |
| 1 | http\://daveztong.github.io/schema/dz=io.github.daveztong.DZNamespaceHandler |

注意，因为:在properties文件中是个合法的分隔符，所以需要escape以下！

其中URI部分是自定义的namespace扩展，必须与xsd中的targetNamespace完全匹配。

**META-INF/spring.schemas**

spring.schemas中包含了xml schema uri(对应xsi:schemaLocation的值)与classpath resources的映射关系。如果这个文件不存在，spring默认会从网上加载xsi:schemaLocation中所定义的schema。有个这个映射文件，spring就会从classpath中加载而不再联网查找。

|  |  |
| --- | --- |
| 1 | http\://daveztong.github.io/schema/dz/dz.xsd=io/github/daveztong/schema/dz/dz.xsd |

**Now,time to test**

使用xml定义person bean: spring-beans.xml

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | <?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:dz="http://daveztong.github.io/schema/dz"  xsi:schemaLocation="  http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd  http://daveztong.github.io/schema/dz http://daveztong.github.io/schema/dz/dz.xsd">  <!-- as a top-level bean -->  <dz:person age="22" id="person" name="tangwei"/>  </beans> |

为了演示效果，这里使用spring boot来快速测试: App.java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32 | package io.github.daveztong;  import org.springframework.boot.SpringApplication;  import org.springframework.boot.autoconfigure.EnableAutoConfiguration;  import org.springframework.context.annotation.ImportResource;  import org.springframework.stereotype.Controller;  import org.springframework.web.bind.annotation.RequestMapping;  import org.springframework.web.bind.annotation.ResponseBody;  import javax.annotation.Resource;  /\*\*  \* Created by tangwei on 2016/11/1.  \*/  @EnableAutoConfiguration  @Controller  @ImportResource("classpath:spring-beans.xml")  public class App {  @Resource  private Person person;  public static void main(String[] args) {  SpringApplication.run(App.class);  }  @RequestMapping("/")  @ResponseBody  public String showPerson() {  return person.toString();  }  } |

访问<http://localhost:8080/> ,dada:

|  |  |
| --- | --- |
| 1 | Person{age=22, name='tangwei'} |

说了这么多，到底谁在用这个，能干啥呢！

**Dubbo xml 配置扩展**

Dubbo 在国内开源RPC界算是比较知名的，她其实采用的就是这种方式解析自定义的配置。dubbo jar 包结构:



spring.handlers:

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

spring.schemas:

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

Namespace Handler实现为:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | public class DubboNamespaceHandler extends NamespaceHandlerSupport {  static {  Version.checkDuplicate(DubboNamespaceHandler.class);  }  public void init() {  registerBeanDefinitionParser("application", new DubboBeanDefinitionParser(ApplicationConfig.class, true));  registerBeanDefinitionParser("module", new DubboBeanDefinitionParser(ModuleConfig.class, true));  registerBeanDefinitionParser("registry", new DubboBeanDefinitionParser(RegistryConfig.class, true));  registerBeanDefinitionParser("monitor", new DubboBeanDefinitionParser(MonitorConfig.class, true));  registerBeanDefinitionParser("provider", new DubboBeanDefinitionParser(ProviderConfig.class, true));  registerBeanDefinitionParser("consumer", new DubboBeanDefinitionParser(ConsumerConfig.class, true));  registerBeanDefinitionParser("protocol", new DubboBeanDefinitionParser(ProtocolConfig.class, true));  registerBeanDefinitionParser("service", new DubboBeanDefinitionParser(ServiceBean.class, true));  registerBeanDefinitionParser("reference", new DubboBeanDefinitionParser(ReferenceBean.class, false));  registerBeanDefinitionParser("annotation", new DubboBeanDefinitionParser(AnnotationBean.class, true));  }  } |

该handler注册了所有的bean definition parser!

了解这种机制后，我们也可以根据自己需要做任何配置扩展。So cool!