dylan0514sina.cn

BeanDefinition数据流

博客分类: spring3.x

BeanDefinition是Spring配置文件中bean定义的内存表现形式,我们先来看看bean的创建。在下图中不同的元素的解析路线用不同的颜色标注

主线部分

1. 实例化xmlApplicationContext
2. 实例化XmlBeanDefinitionReader
3. ResourceLoader定位bean文件
4. XmlBeanDefinitionReader读入Resource
5. DefaultBeanDefinitionDocumentReader解析Document
6. 默认命名空间判断
7. import元素判断
8. alias元素判断
9. beans元素递归

非默认bean命名空间处理,如AOP

1. 读取多个jar包中的META-INF/spring.handlers
2. DefaultNamespaceResolver解析spring.handlers文件,取得元素命名空间对应的NamespaceHandler
3. NamespaceHandler解析Element元素
4. 根据不同的Handler会生成或ComponentDefinition或BeanDefinition
5. 注册BeanDefinition

import元素处理部分

1. 重复主线中的3,4,5步

alias元素处理部分

1. 注册标志和别名

bean元素处理部分

1. 通过BeanfinitionParserDelegate解析bean元素
2. 注册BeanDefinitionHolder

- 2013-12 (3)
- 2013-11 (1)
- 2013-09 (1)
- 更多存档...

最新评论

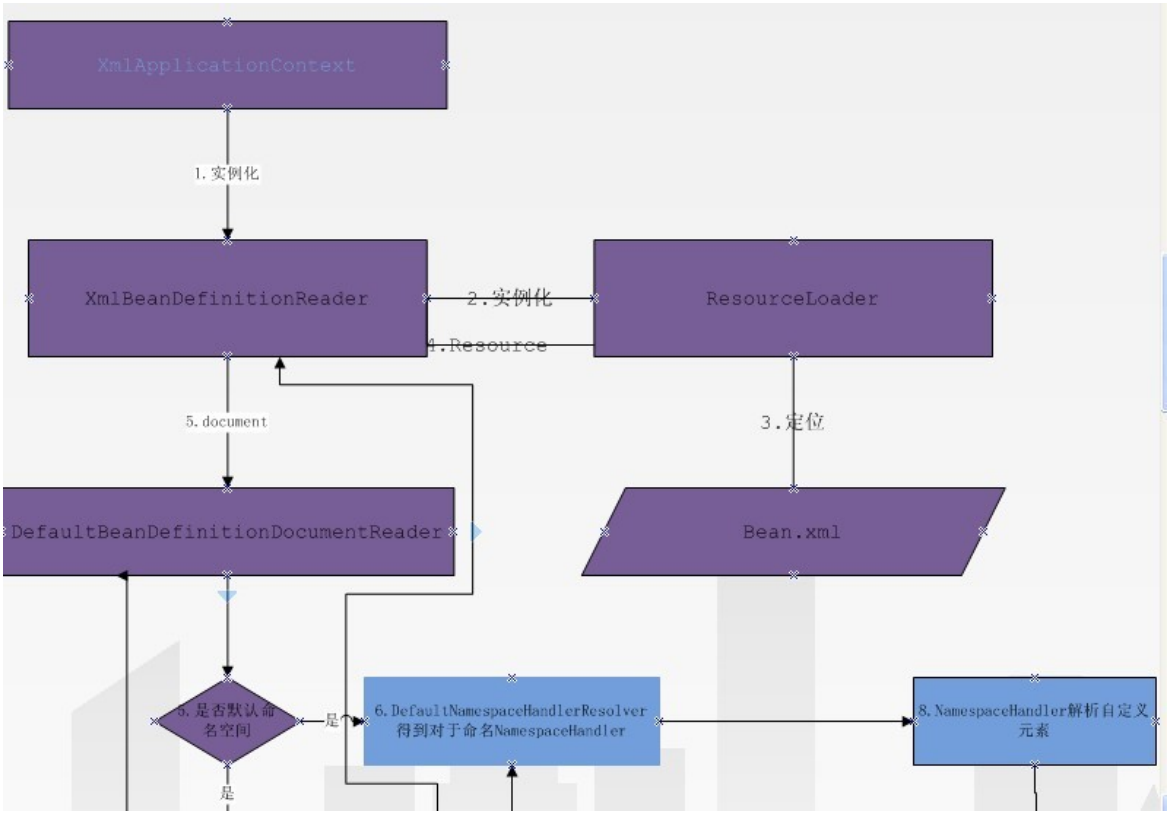
dylan0514sina.cn: youjianbo_han_87 写道dylan0514sin ...
方法缓存

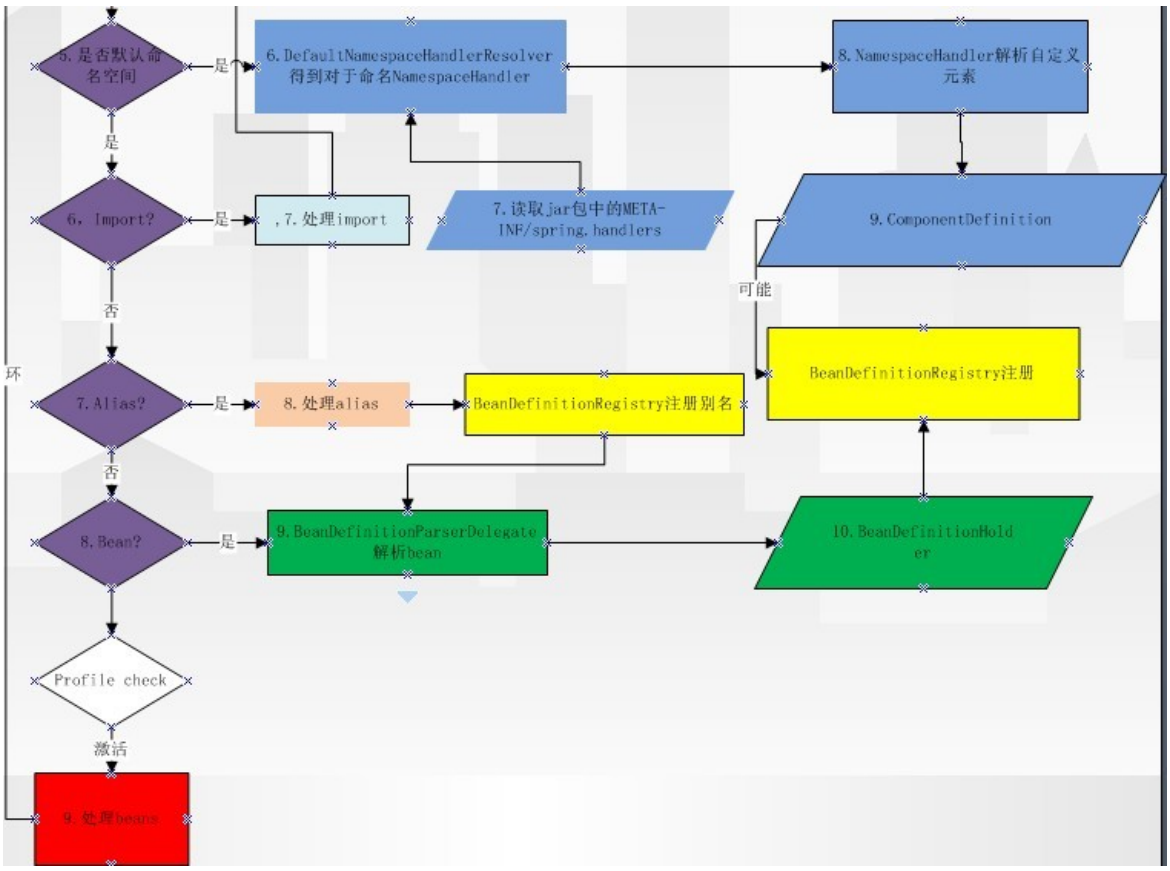
youjianbo_han_87: dylan0514sina.cn 写道youjianbo_ha ...
方法缓存

dylan0514sina.cn: youjianbo_han_87 写道缓存方法有意义吗,方法+ ...
方法缓存

youjianbo_han_87: 缓存方法有意义吗,方法+调用从缓存中取内容的方法 换成 方法+ ...
方法缓存

dylan0514sina.cn: Shen.Yiyang 写道剔除策略只有方法执行的时候指定ke ...
方法缓存





在解析bean.xml的过程中,用"<property....."作为类的属性值,与之对应的值的定义方式不同反应不同的数据结构,

Xml代码

```
1.<property name="foo" ref="bean"/>
2.<property name="foo" value="beanString"/>
3.<property name="foo">
  <bean class=""/>
</property>
....
```

beanDefinition必须存储这些信息,那BeanDefinition定义中,可以看到属性值设置信息在MutablePropertyValues 中,当然<constructor-arg/>类似,存储在ConstructorArgumentValues

Java代码

```
ConstructorArgumentValues getConstructorArgumentValues();
MutablePropertyValues getPropertyValues();
```

不同的Element或Attribute对应不同的属性值的结构列表如下

bean	BeanDefinitionHolder
ref	RuntimeBeanReference
value,null	TypedStringValue
Idref	RuntimeBeanNameReference
array	ManagedArray
list	ManagedList
set	ManagedSet
map	ManagedMap
props	ManagedProperties

这些数据类型会在getBean初始化设值时转化为必要值引用类型,对于值是beanDefinitionHolder的将生成为一个prototype的bean。

Java代码

```
public Object resolveValueIfNecessary(Object argName, Object value) {
                // We must check each value to see whether it requires a runtime reference
                // to another bean to be resolved.
                if (value instanceof RuntimeBeanReference) {
                        RuntimeBeanReference ref = (RuntimeBeanReference) value;
                        return resolveReference(argName, ref);
                }
                else if (value instanceof RuntimeBeanNameReference) {
                        String refName = ((RuntimeBeanNameReference) value).getBeanName();
                        refName = String.valueOf(evaluate(refName));
                        if (!this.beanFactory.containsBean(refName)) {
                                throw new BeanDefinitionStoreException(
                                                "Invalid bean name '" + refName + "' in be
an reference for " + argName);
                        }
                        return refName;
                }
                else if (value instanceof BeanDefinitionHolder) {
                        // Resolve BeanDefinitionHolder: contains BeanDefinition with name a
nd aliases.
                        BeanDefinitionHolder bdHolder = (BeanDefinitionHolder) value;
                        return resolveInnerBean(argName, bdHolder.getBeanName(), bdHolder.get
BeanDefinition());
```

```
21.                 }
22.                 else if (value instanceof BeanDefinition) {
23.                         // Resolve plain BeanDefinition, without contained name: use dummy n
    ame.
24.                         BeanDefinition bd = (BeanDefinition) value;
25.                         return resolveInnerBean(argName, "(inner bean)", bd);
26.                 }
27.                 else if (value instanceof ManagedArray) {
28.                         // May need to resolve contained runtime references.
29.                         ManagedArray array = (ManagedArray) value;
30.                         Class elementType = array.resolvedElementType;
31.                         if (elementType == null) {
32.                                 String elementTypeName = array.getElementTypeName();
33.                                 if (StringUtils.hasText(elementTypeName)) {
34.                                         try {
35.                                                 elementType = ClassUtils.forName(elementType
    Name, this.beanFactory.getBeanClassLoader());
36.                                                 array.resolvedElementType = elementType;
37.                                         }
38.                                         catch (Throwable ex) {
39.                                                 // Improve the message by showing the conte
    xt.
40.                                                 throw new BeanCreationException(
41.                                                                 this.beanDefinition.getReso
    urceDescription(), this.beanName,
42.                                                                 "Error resolving array typ
    e for " + argName, ex);
43.                                         }
44.                                 }
45.                                 else {
46.                                         elementType = Object.class;
47.                                 }
48.                         }
49.                         return resolveManagedArray(argName, (List<?>) value, elementType);
50.                 }
51.                 else if (value instanceof ManagedList) {
52.                         // May need to resolve contained runtime references.
53.                         return resolveManagedList(argName, (List<?>) value);
54.                 }
55.                 else if (value instanceof ManagedSet) {
56.                         // May need to resolve contained runtime references.
57.                         return resolveManagedSet(argName, (Set<?>) value);
58.                 }
59.                 else if (value instanceof ManagedMap) {
60.                         // May need to resolve contained runtime references.
61.                         return resolveManagedMap(argName, (Map<?, ?>) value);
62.                 }
63.                 else if (value instanceof ManagedProperties) {
64.                         Properties original = (Properties) value;
65.                         Properties copy = new Properties();
66.                         for (Map.Entry propEntry : original.entrySet()) {
67.                                 Object propKey = propEntry.getKey();
68.                                 Object propValue = propEntry.getValue();
69.                                 if (propKey instanceof TypedStringValue) {
70.                                         propKey = evaluate((TypedStringValue) propKey);
71.                                 }
72.                                 if (propValue instanceof TypedStringValue) {
73.                                         propValue = evaluate((TypedStringValue) propValue);

74.                                 }
75.                                 copy.put(propKey, propValue);
76.                         }
77.                         return copy;
78.                 }
79.                 else if (value instanceof TypedStringValue) {
```

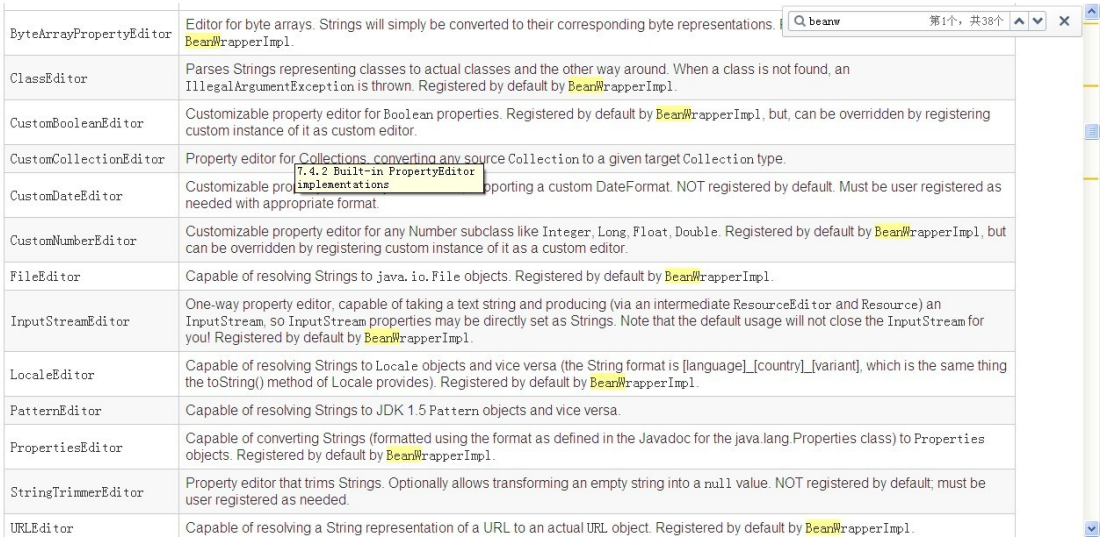
```
80.                         // Convert value to target type here.
81.                         TypedStringValue typedStringValue = (TypedStringValue) value;
82.                         Object valueObject = evaluate(typedStringValue);
83.                         try {
84.                                 Class<?> resolvedTargetType = resolveTargetType(typedStringV
    alue);
85.                                 if (resolvedTargetType != null) {
86.                                         return this.typeConverter.convertIfNecessary(valueOb
    ject, resolvedTargetType);
87.                                 }
88.                                 else {
89.                                         return valueObject;
90.                                 }
91.                         }
92.                         catch (Throwable ex) {
93.                                 // Improve the message by showing the context.
94.                                 throw new BeanCreationException(
95.                                                 this.beanDefinition.getResourceDescription()
    , this.beanName,
96.                                                 "Error converting typed String value for "
    + argName, ex);
97.                         }
98.                 }
99.                 else {
100.                        return evaluate(value);
101.                }
102.        }
```

转化规则如下

BeanDefinitionHolder	忽略 scope="singleton" 请求一次，ID 不唯一 无循环引用可能
RuntimeBeanReference	ID 可以是 SPEL 表达式 请求创建之前，计算表达式
TypedStringValue	计算表达式 目标类型可以用 type 属性指定 如有 typeconverter,则调用转化方法
RuntimeBeanNameReference	返回计算表达式之后的 ID 值
ManagedArray	目标类型可以用 type 属性指定，默认 Object 根据指定类型动态创建数组 元素值进行递归转化
ManagedList	元素值进行递归转化
ManagedSet	元素值进行递归转化
ManagedMap	元素值进行递归转化
ManagedProperties	元素 key 和 value 都进行表达式计算

当实例都准备好之后,在匹配之前支持做类型转换

Spring使用java bean规范中的PropertyEditors解决中Object和String之间的转换 。我们可以自定义java.beans.PropertyEditor实现,然后注册到Spring容器中。比如经常使用的场景是解析bean.xml时。org.springframework.beans.propertyeditors包下的所有编辑器被Spring支持,Spring通过java.bean.PropertyEditorManager设置这种搜索路径。经典实现中,将会查找class bean包路径+"Editor"



ByteArrayPropertyEditor	Editor for byte arrays. Strings will simply be converted to their corresponding byte representations. Registered by default by BeanWrapperImpl.
ClassEditor	Parses Strings representing classes to actual classes and the other way around. When a class is not found, an IllegalArgumentException is thrown. Registered by default by BeanWrapperImpl.
CustomBooleanEditor	Customizable property editor for Boolean properties. Registered by default by BeanWrapperImpl, but, can be overridden by registering custom instance of it as custom editor.
CustomCollectionEditor	Property editor for Collections, converting any source Collection to a given target Collection type.
CustomDateEditor	Customizable property editor for java.util.Date, supporting a custom DateFormat. NOT registered by default. Must be user registered as needed with appropriate format.
CustomNumberEditor	Customizable property editor for any Number subclass like Integer, Long, Float, Double. Registered by default by BeanWrapperImpl, but can be overridden by registering custom instance of it as a custom editor.
FileEditor	Capable of resolving Strings to java.io.File objects. Registered by default by BeanWrapperImpl.
InputStreamEditor	One-way property editor, capable of taking a text string and producing (via an intermediate ResourceEditor and Resource) an InputStream, so InputStream properties may be directly set as Strings. Note that the default usage will not close the InputStream for you! Registered by default by BeanWrapperImpl.
LocaleEditor	Capable of resolving Strings to Locale objects and vice versa (the String format is [language]_[country]_[variant], which is the same thing the toString() method of Locale provides). Registered by default by BeanWrapperImpl.
PatternEditor	Capable of resolving Strings to JDK 1.5 Pattern objects and vice versa.
PropertiesEditor	Capable of converting Strings (formatted using the format as defined in the Javadoc for the java.lang.Properties class) to Properties objects. Registered by default by BeanWrapperImpl.
StringTrimmerEditor	Property editor that trims Strings. Optionally allows transforming an empty string into a null value. NOT registered by default; must be user registered as needed.
URLEditor	Capable of resolving a String representation of a URL to an actual URL object. Registered by default by BeanWrapperImpl.

Foo类同包下面有个FooEditor

Xml代码

```
com
  chank
    pop
      Foo
      FooEditor   // the PropertyEditor for the Foo class
```

在使用JAVA BEAN自省机制之前,可以定义Foo类的BeanInfo

Xml代码

```
com
  chank
    pop
      Foo
      FooBeanInfo   // the BeanInfo for the Foo class
```

Java代码

```
public class FooBeanInfo extends SimpleBeanInfo {

    public PropertyDescriptor[] getPropertyDescriptors() {
        try {
            final PropertyEditor numberPE = new CustomNumberEditor(Integer.class, true);
            PropertyDescriptor ageDescriptor = new PropertyDescriptor("age", Foo.class) {
                public PropertyEditor createPropertyEditor(Object bean) {
                    return numberPE;
                };
            };
            return new PropertyDescriptor[] { ageDescriptor };
        }
        catch (IntrospectionException ex) {
            throw new Error(ex.toString());
        }
    }
}
```

接下来进行注册

将ExoticType 实例注入到DependsOnExoticType 中,很显然,我们需要ExoticType对应的Editor将字符串转化为ExoticType 以注入

Java代码

```java
package example;

public class ExoticType {

    private String name;

    public ExoticType(String name) {
        this.name = name;
    }
}

public class DependsOnExoticType {

    private ExoticType type;

    public void setType(ExoticType type) {
        this.type = type;
    }
}
```

只是一个ExoticType 名字

Xml代码

```xml
<bean id="sample" class="example.DependsOnExoticType">
    <property name="type" value="aNameForExoticType"/>
</bean>
```

PropertyEditor实现

Java代码

```java
// converts string representation to ExoticType object
package example;

public class ExoticTypeEditor extends PropertyEditorSupport {

    public void setAsText(String text) {
        setValue(new ExoticType(text.toUpperCase()));
    }
}
```

我们将使用CustomEditorConfigurer注册,这是一个BeanFactoryPostProcessor Bean,在ApplicationContext生命周期中自动调用

Xml代码

```xml
<bean class="org.springframework.beans.factory.config.CustomEditorConfigurer">
  <property name="customEditors">
    <map>
      <entry key="example.ExoticType" value="example.ExoticTypeEditor"/>
    </map>
  </property>
</bean>
```

如果要使用多个Editor,则还可以实现PropertyEditorRegistrar 接口,在实现中编程注册;显然多了一层包装。

Java代码

```java
public final class CustomPropertyEditorRegistrar implements PropertyEditorRegistrar {

    public void registerCustomEditors(PropertyEditorRegistry registry) {

```

```
        // it is expected that new PropertyEditor instances are created
        registry.registerCustomEditor(ExoticType.class, new ExoticTypeEditor());

        // you could register as many custom property editors as are required here...
    }
}
```

Xml代码

```
<bean class="org.springframework.beans.factory.config.CustomEditorConfigurer">
    <property name="propertyEditorRegistrars">
        <list>
            <ref bean="customPropertyEditorRegistrar"/>
        </list>
    </property>
</bean>

<bean id="customPropertyEditorRegistrar"
      class="com.foo.editors.spring.CustomPropertyEditorRegistrar"/>
```

java bean风格的类型转化只在String与Object之间,那如果Object和Object,则java bean标准转化机制显得不够用。spring 3引入了core.convert包提供更加一般化的类型转换。使用泛型机制,强制类型。

Java代码

```
public interface Converter<S, T> {

    T convert(S source);

}
//简单实现StringToInteger
final class StringToInteger implements Converter<String, Integer> {

    public Integer convert(String source) {
        return Integer.valueOf(source);
    }

}
```

当仅仅关注某一类型的转化器时,ConverterFactory有了用武之地。

Java代码

```
public interface ConverterFactory<S, R> {

    <T extends R> Converter<S, T> getConverter(Class<T> targetType);

}
//将字符串转化为一个枚举类型的factory
final class StringToEnumConverterFactory implements ConverterFactory<String, Enum> {

    public <T extends Enum> Converter<String, T> getConverter(Class<T> targetType) {
        return new StringToEnumConverter(targetType);
    }

    private final class StringToEnumConverter<T extends Enum> implements Converter<String, T>
 {

        private Class<T> enumType;

        public StringToEnumConverter(Class<T> enumType) {
            this.enumType = enumType;
        }
```

```
20.
21.         public T convert(String source) {
22.             return (T) Enum.valueOf(this.enumType, source.trim());
23.         }
24.     }
25. }
```

GenericConverter没有泛型化

Java代码

```
1.  package org.springframework.core.convert.converter;
2.
3.  public interface GenericConverter {
4.
5.      public Set<ConvertiblePair> getConvertibleTypes();
6.
7.      Object convert(Object source, TypeDescriptor sourceType, TypeDescriptor targetType);
8.
9.  }
```

ConditionalGenericConverter想执行转换之前知道是否能够转换

Java代码

```
1.  public interface ConditionalGenericConverter extends GenericConverter {
2.
3.      boolean matches(TypeDescriptor sourceType, TypeDescriptor targetType);
4.
5.  }
```

ConversionService 实现具体的转换逻辑。spring中大多数现象了ConversionService 的同时也实现了ConverterRegistry,ConverterRegistry用于注册转换器,一般来说,ConversionService 实现需要使用转换器了解源对象和目标对象的类型。GenericConversionService就是这样的一个实现。

Java代码

```
1.  package org.springframework.core.convert;
2.
3.  public interface ConversionService {
4.
5.      boolean canConvert(Class<?> sourceType, Class<?> targetType);
6.
7.      <T> T convert(Object source, Class<T> targetType);
8.
9.      boolean canConvert(TypeDescriptor sourceType, TypeDescriptor targetType);
10.
11.     Object convert(Object source, TypeDescriptor sourceType, TypeDescriptor targetType);
12.
13. }
```

我们将使用ConversionServiceFactoryBean注册转换器和GenericConversionService

Java代码

```
1.  <bean id="conversionService"
2.        class="org.springframework.context.support.ConversionServiceFactoryBean">
3.      <property name="converters">
4.          <list>
5.              <bean class="example.MyCustomConverter"/>
6.          </list>
7.      </property>
```

```
8.  </bean>
```

查看图片附件

分享到:

基于xml schema的扩展标签 | bean的创建周期回调

2013-07-08 19:41 | 浏览 216 | 评论(0) | 分类:开源软件 | 相关推荐 MORE

评论

发表评论

您还没有登录,请您登录后再发表评论