BeanDefiniton+默认标签+自定义标签解析

1. BeanDefintion是什么
2. 默认标签解析
3. 自定义标签解析
4. 装饰器设计模式

---

id:Bean的唯一标识名。它必须是合法的XMLID，在整个XML文档中唯一。

（2）、name:用来为id创建一个或多个别名。它可以是任意的字母符合。多个别名之间用逗号或空格分开。

（3）、class:用来定义类的全限定名（包名＋类名）。只有子类Bean不用定义该属性。

（4）、parent:子类Bean定义它所引用它的父类Bean。这时前面的class属性失效。子类Bean会继承父类Bean的所有属性，子类Bean也可以覆盖父类Bean的属性。注意：子类Bean和父类Bean是同一个Java类。

（5）、abstract（默认为”false”）：用来定义Bean是否为抽象Bean。它表示这个Bean将不会被实例化，一般用于父类Bean，因为父类Bean主要是供子类Bean继承使用。

（6）、singleton（默认为“true”）：定义Bean是否是Singleton（单例）。如果设为“true”,则在BeanFactory作用范围内，只维护此Bean的一个实例。如果设为“flase”，Bean将是Prototype（原型）状态，BeanFactory将为每次Bean请求创建一个新的Bean实例。

（7）、lazy-init（默认为“default”）：用来定义这个Bean是否实现懒初始化。如果为“true”，它将在BeanFactory启动时初始化所有的SingletonBean。反之，如果为“false”,它只在Bean请求时才开始创建SingletonBean。

（8）、autowire（自动装配，默认为“default”）：它定义了Bean的自动装载方式。

1、“no”:不使用自动装配功能。

2、“byName”:通过Bean的属性名实现自动装配。

3、“byType”:通过Bean的类型实现自动装配。

4、“constructor”:类似于byType，但它是用于构造函数的参数的自动组装。

5、“autodetect”:通过Bean类的反省机制（introspection）决定是使用“constructor”

还是使用“byType”。

（9）、dependency-check（依赖检查，默认为“default”）：它用来确保Bean组件通过JavaBean描述的所以依赖关系都得到满足。在与自动装配功能一起使用时，它特别有用。

1、none：不进行依赖检查。

2、objects：只做对象间依赖的检查。

3、simple：只做原始类型和String类型依赖的检查

4、all：对所有类型的依赖进行检查。它包括了前面的objects和simple。

（10）、depends-on（依赖对象）：这个Bean在初始化时依赖的对象，这个对象会在这个Bean初始化之前创建。

（11）、init-method:用来定义Bean的初始化方法，它会在Bean组装之后调用。它必须是一个无参数的方法。

（12）、destroy-method：用来定义Bean的销毁方法，它在BeanFactory关闭时调用。同样，它也必须是一个无参数的方法。它只能应用于singletonBean。

（13）、factory-method：定义创建该Bean对象的工厂方法。它用于下面的“factory-bean”，表示这个Bean是通过工厂方法创建。此时，“class”属性失效。

（14）、factory-bean:定义创建该Bean对象的工厂类。如果使用了“factory-bean”则“class”属性失效。

（15）、autowire-candidate：采用xml格式配置bean时，将<bean/>元素的autowire-candidate属性设置为false，这样容器在查找自动装配对象时，将不考虑该bean，即它不会被考虑作为其它bean自动装配的候选者，但是该bean本身还是可以使用自动装配来注入其它bean的。

```
public abstract class ShowSixClass<T> {

    public void showsix() { getPeople().showsix(); }

    public T getT() { return null; }

    //不一定是抽象的
    public abstract People getPeople();
}
```

```
<!--    <bean id="woman" class="com.enjoy.jack.bean.Woman" init-method="init"/>-->
<!--    <bean id="people" class="com.enjoy.jack.bean.ShowSixClass">-->
<!--        <lookup-method name="getPeople" bean="woman"></lookup-method>-->
<!--    </bean>-->
```

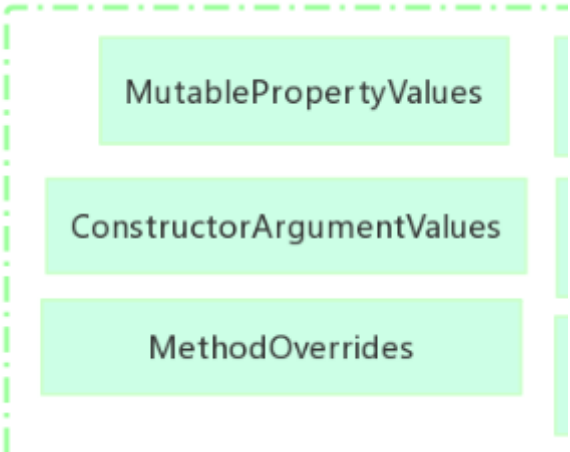
```
        //创建GenericBeanDefinition对象
        AbstractBeanDefinition bd = createBeanDefinition(className, parent

        //解析bean标签的属性，并把解析出来的属性设置到BeanDefinition对象中
        parseBeanDefinitionAttributes(ele, beanName, containingBean, bd);
        bd.setDescription(DomUtils.getChildElementValueByTagName(ele, DESCI

        //解析bean中的meta标签
        parseMetaElements(ele, bd);
        //解析bean中的lookup-method标签  重要程度：2，可看可不看
        parseLookupOverrideSubElements(ele, bd.getMethodOverrides());
        //解析bean中的replaced-method标签  重要程度：2，可看可不看
        parseReplacedMethodSubElements(ele, bd.getMethodOverrides());

        //解析bean中的constructor-arg标签  重要程度：2，可看可不看
        parseConstructorArgElements(ele, bd);
        //解析bean中的property标签  重要程度：2，可看可不看
        parsePropertyElements(ele, bd);
        //可以不看，用不到
        parseQualifierElements(ele, bd);
```

可以是实现

```
public class ReplaceClass implements MethodReplacer {
    @Override
    public Object reimplement(Object obj, Method method, Object[] args) throws
        System.out.println("I am replace method-->reimplement!");
        return null;
    }
}
```

```
<bean id="repalceClass" class="com.enjoy.jack.bean.ReplaceClass"/>
<bean id="originClass" class="com.enjoy.jack.bean.OriginClass">
    <replaced-method name="method" replacer="repalceClass">
        <arg-type match="java.lang.String"/>
    </replaced-method>
</bean>
```

Show help Disable...



```
http\://www.springframework.org/schema/context=org.springframework.context.config.ContextNamespa
http\://www.springframework.org/schema/jee=org.springframework.ejb.config.JeeNamespaceHandler
```

```
@Nullable
public BeanDefinition parseCustomElement(Element ele, @Nullable BeanDefinition containingBd) {
    String namespaceUri = getNamespaceURI(ele);
    if (namespaceUri == null) {
        return null;
    }
    NamespaceHandler handler = this.readerContext.getNamespaceHandlerResolver().resolve(namespaceUr
    if (handler == null) {
        error( message: "Unable to locate Spring NamespaceHandler for XML schema namespace [" + name
        return null;
    }
    return handler.parse(ele, new ParserContext(this.readerContext, delegate: this, [redacted]ngBd));
}
```

code-release-notes-spi

spring.handlers

```
/*
   重要程度：5
  1、创建BeanFactory对象
* 2、xml解析
*   传统标签解析：bean、import等
*   自定义标签解析 如：<context:component-scan base-package="com.xiangxue.jack"/>
*   自定义标签解析流程：
*       a、根据当前解析标签的头信息找到对应的namespaceUri
*       b、加载spring所有jar中的spring.handlers文件。并建立映射关系
*       c、根据namespaceUri从映射关系中找到对应的实现了NamespaceHandler接口的类
*       d、调用类的init方法，init方法是注册了各种自定义标签的解析类
*       e、根据namespaceUri找到对应的解析类，然后调用paser方法完成标签解析
*
* 3、把解析出来的xml标签封装成BeanDefinition对象
* */
// Tell the subclass to refresh the internal bean factory.
ConfigurableListableBeanFactory beanFactory = obtainFreshBeanFactory();
```