Spring Bean管理

# Spring容器

## 使用Spring容器

Spring容器最基本的接口就是BeanFactor。BeanFactory负责配置、创建、管理Bean，他有一个子接口：ApplicationContext，因此也称之为Spring上下文。Spring容器负责管理Bean与Bean之间的依赖关系。

BeanFactory接口包含以下几个基本方法：

1. Boolean containBean(String name):判断Spring容器是否包含id为name的Bean实例。
2. <T> getBean(Class<T> requiredTypr):获取Spring容器中属于requiredType类型的唯一的Bean实例。
3. Object getBean(String name)：返回Sprin容器中id为name的Bean实例。
4. <T> T getBean(String name,Class requiredType)：返回容器中id为name,并且类型为requiredType的Bean
5. Class <?> getType(String name)：返回容器中指定Bean实例的类型。

调用者只需使用getBean()方法即可获得指定Bean引用，无须关心Bean的实例化过程。即Bean实例的创建过程完全透明。

在使用BeanFactory接口时，我们一般都是使用这个实现类：org.spring  
framework.beans.factory.xml.XmlBeanFactory。然而ApplicationContext作为BeanFactory的子接口，使用它作为Spring容器会更加方便。它的实现类有：FileSystemXmlApplicationContext、ClassPathXmlApplication  
Context、AnnotationConfigApplicationContext。

创建Spring容器实例时，必须提供Spring容器管理的Bean的详细配置信息。Spring的配置信息通常采用xml配置文件来设置，因此，创建BeanFactory实例时，应该提供XML配置文件作为参数。

XML配置文件通常使用Resource对象传入。Resource接口是Spring提供的资源访问接口，通过使用该接口，Spring能够以简单、透明的方式访问磁盘、类路径以及网络上的资源。对于Java EE应用而言，可在启动Web应用时自动加载ApplicationContext实例，接受Spring管理的Bean无须知道ApplicationContext的存在。一般使用如下方式实例化BeanFactory。

|  |
| --- |
| //搜索当前文件路径下的bean.xml文件创建Resource对象  InputStreamSource isr = new FileSystemResource("bean.xml");  //以Resource对象作为参数创建BeanFactory实例  XmlBeanFactory factory = new XmlBeanFactory((Resource) isr); |
| ClassPathResource res = new ClassPathResource("bean.xml");  //以Resource对象作为参数创建BeanFactory实例  XmlBeanFactory factory = new XmlBeanFactory(res); |
| 但是如果应用里面有多个属性配置文件，则应该采用BeanFactory的子接口ApplicationContext来创建BeanFactory的实例。ApplicationContext通常使用如下两个实现类：  FileSystemXmlApplicationContext：以基于文件系统的XML配置文件创建ApplicationContext实例。  ClassPathXmlApplicationContext：以类加载路径下的XML配置文件创建的ApplicationContext实例。  如果需要同时加载多个XML配置文件，采用如下方式：  //搜索CLASSPATH路径，以classpath路径下的bean.xml、service.xml文件创建applicationContext  ApplicationContext ctx = new ClassPathXmlApplicationContext(new String[]{"bean.xml","service.xml"});  //以指定路径下的bean.xml、service.xml文件创建applicationContext  ApplicationContext ctx1 = new FileSystemXmlApplicationContext(new String[]{"bean.xml","service.xml"}); |

## Bean获取Spring容器

在Spring中我们可以使用Spring容器中getBean()方法来获取Spring容器中的Bean实例。在这样的访问模式下，程序中总是持有Spring容器的引用。但是在实际的应用中，Spring容器通常是采用声明式方式配置产生：开发者只要在web.xml文件中配置一个Listener，该Listener将会负责初始化Spring容器。在这种情况下，容器中Bean处于容器管理下，无须主动访问容器，只需要接受容器的注入管理即可。同时Bean实例的依赖关系通常也是由容器主动注入，无须Bean实例主动请求。

在这种情况下，Sprig容器中Bean通常不会需要访问容器中其他Bean—采用依赖注入，让Spring把被依赖的Bean注入到依赖的Bean中即可。

实现BeanFactoryAware接口的Bean，拥有访问的BeanFactory容器的能力，实现BeanFactoryAware接口的Bean实例将会拥有对容器的访问能力。BeanFactoryAware接口仅有如下一个方法：

|  |
| --- |
| SetBeanFactory(BeanFactory beanFactory)：该方法有一个参数beanFactory，该参数指向创建它的BeanFactory。该方法将由Spring调动，当Spring调用该方法时会将Spring容器作为参数传入该方法。 |
| public class Chinese implements ApplicationContextAware{  //将BeanFactory容器以成员变量保存  private ApplicationContext ctx;  /\*\*  \* 实现ApplicationContextAware接口实现的方法  \*/  public void setApplicationContext(ApplicationContext cyx)  throws BeansException {  this.ctx = ctx;  }    //获取ApplicationContext的测试方法  public ApplicationContext getContext(){  return ctx;  }  } |
| public class ChineseTest {  public static void main(String[] args) {  ApplicationContext ctx = new ClassPathXmlApplicationContext("bean.xml");  Chinese c = ctx.getBean("chinese",Chinese.class);  System.out.println(c.getContext());    System.out.println(c.getContext()==ctx);  }  } //true  上面的代码虽然实现了ApplicationContextAware接口让Bean拥有了访问容器的能力，但是污染了代码，导致代码与Spring接口耦合在一起。所以，如果不是特别需要，一般不建议直接访问容器。 |

## ApplicationContext的使用

前面介绍了，我们一般不会使用BeanFactory实例作为spring容器，而是使用ApplicationContext实例作为容器，它增强了BeanFactory的功能。

ApplicationContext允许以声明式方式操作容器，无须手动创建它。在Web应用启动时自动创建ApplicationContext。当然，也可以采用编程方式创建ApplicationContext。除了提供BeanFactory所支持的全部功能外，ApplicationContext还有如下功能：

1. ApplicationContext继承MessageSource接口，因此提供国际化支持。
2. 资源访问
3. 事件机制。
4. 载入多个配置文件。
5. 以声明式的方式启动，并创建Spring容器。

当系统创建ApplicationContext容器时，默认会预初始化所有的single  
ton Bean。也就是说，当ApplicationContext容器初始化完成后，容器中所有singleton Bean也实例化完成，这就意味着：系统前期创建Application  
Context时将有较大的系统开销，但一旦ApplicationContext初始化完成，程序后面获取singleton Bean实例时将拥有较好的性能。

### ApplicationContext的国际化支持

ApplicationContext接口继承MessageSource接口，因此具备国际化功能。MessageSource接口中定义了两个个方法用于国际化功能。

* String getMessage(Stringcode,Object[] args,Locale loc);
* StringgetMessage(String code,Object[] args,String default,Locale loc);

ApplicationContext正是通过这两个方法来实现国际化的。当程序创建ApplicationContext容器时，Spring会自动查找在配置文件中名为messageSource的bean实例，一旦找到这个Bean实例，上述两个方法的调用被委托给MessageSource Bean。如果没有该Bean，ApplicationContext会查找其父定义中的messagesource Bean，如果找到，它会作为messageSource Bean使用。但是如果无法找到messageSource，系统将会创建一个空的staticMessageSource Bean，该Bean的能接受上述两个方法的调用。

Spring中配置messagesourceBean使用ResourceBundleMessageSource.

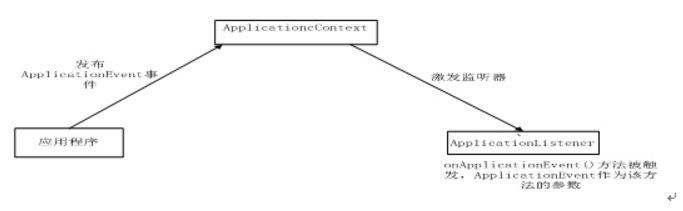
|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <beans xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xmlns="http://www.springframework.org/schema/beans"  xsi:schemaLocation="http://www.springframework.org/schema/beans  http://www.springframework.org/schema/beans/spring-beans-4.0.xsd">  <bean id="messsageSource" class="org.springframework.context.support.ResourceBundleMessageSource">  <property name="basenames">  <list>  <value>.....</value>  <value>.....</value>  <value>.....</value>  </list>  </property>  </bean>  </beans> |

### ApplicationContext的事件机制

ApplicationContext的事件机制是观察者设计模式的实现，通过Applica  
tionEvent类和ApplicationListener接口，可以实现ApplicationContext的事件处理。如果容器中有一个ApplicationListener Bean，每当Application  
Context发布ApplicationEvent时ApplicationListenerBean就会自动触发。Spring的事件框架有如下两个重要成员：

* ApplicationEvent：容器事件，必须由ApplicationContext发布。
* ApplicationListener：监听器，可由容器中的任何监听器Bean担任。

Spring的事件机制需要事件源、事件和事件监听器组成。只是此处的事件是ApplicationContext，且事件必须由Java程序显示触发。下图简单示范了ApplicationContext的事件流程。



|  |
| --- |
| #1定义一个ApplicationEvent类，其对象就是Spring容器事件。  **public** **class** EmailEvent **extends** ApplicationEvent {  **private** **static** **final** **long** ***serialVersionUID*** = 1L;    **private** String address;  **private** String text;    // 定义一个带参的构造函数  **public** EmailEvent(Object source) {  **super**(source);  }  //set,get |
| #2容器事件的监听器类必须实现ApplicationListener接口，它的实现方法如下：  onAPplicationEvent(ApplicationEventevent):每当容器内发生任何事件时，此方法都会被触发。  public class EmailNotifier implements ApplicationListener{  //该方法会在容器发生事件时触发  public void onApplicationEvent(ApplicationEvent event) {  if(event instanceof EmailEvent){  //只处理EmailEvent，发送email通知  EmailEvent emailEvent = (EmailEvent) event;  System.out.println("需要发送邮件的接收地址为:"+emailEvent.getAddress());    System.out.println("需要发送邮件的邮件正文是:"+emailEvent.getText());  }  else {  //容器内置事件不作任何处理  System.out.println("容器本身的事件:"+event);  }  }  } |
| #3将监听器类配置在容器中。  <!-- 配置监听器 -->  <bean class=*"com.imooc.test.EmailNotifier"* /> |
| #4 通过上面的3个步骤就可以实现Spring容器的事件。当系统创建Spring容器，加载Spring容器时会自动触发容器事件，容器事件监听器可以监听到这些事件。同时我们也可以调用ApplicationContext的pulishEvent()方法来主动触发容器事件。  ApplicationContext ctx = **new** ClassPathXmlApplicationContext ("spring-core.xml");  //创建一个ApplicationEvent对象  EmailEvent emailEvent = **new** EmailEvent("test", "spring\_test@163.com","this is a test");    //主动触发容器事件  ctx.publishEvent(emailEvent); |

如果Bean想发布事件，则Bean必须获得其容器的引用。如果程序中没有直接获取容器的引用，则应该让Bean实现ApplicationContextAware或BeanFactoryAware接口，从而获得容器的引用。除了我们可以自己实现Spring容器的事件外，Spring也提供了几个内置事件：

* ContextRefreshedEvent：ApplicationContext容器初始化或者刷新时触发该事件。
* ContextStartedEvent：当使用ConfigurableApplicationContext接口的start()方法启动ApplicationContext容器时触发该事件。
* ContextClosedEvent：当使用ConfigurableApplicationContext接口的close()方法关闭ApplicationContext容器时触发该事件。
* ContextStopedEvent: 当使用ConfigurableApplicationContext接口的stop()方法停止ApplicationContext容器时触发该事件。

## 资源访问 - Resource

对于Java来说，资源访问通常有java,net.URL和文件IO流来完成，如果需要访问来自网络的资源时，通常选择URL类，可是URL类目前还无法在类加载路径或者相对于ServletContext的路径中访问资源。虽然可以使用特定的URL前缀注册新的处理类，但是这样做通常比较复杂，而且URL接口还缺少一些有用的功能，比如检查所指向的资源是否存在等。spring提供了一个Resource接口，该接口是所有资源访问类所实现的接口，该接口主要提供了如下几个方法：

|  |
| --- |
| getInputStream()：定位并且打开资源，返回资源对应的输入流，每次调用都会返回新的输入流，调用者必须负责关闭输入流  isOpen()：返回资源文件是否打开，如果资源文件不能多次读取，每次读取结束时应该显示关闭，以防止资源泄露  exists()：返回Resource所指向的资源是否存在  getDescription()：返回资源的描述信息，用于资源处理出错时输出该信息，通常是全限定文件名或者实际URL  getFile()：返回资源对应的File对象  getURL()：返回资源对应的URL对象 |

Resource接口是Spring资源访问的接口，具体的资源访问由该接口的实现类完成。Spring提供了Resource接口的大量实现类:

|  |
| --- |
| UrlReource：访问网络资源的实现类  ClassPathResource：访问类加载路径里资源的实现类  FileSystemResource: 访问文件系统里的资源的实现类  ServletContextResource：访问相对于ServletContext路径下的资源的实现类，通常在WEB应用中使用  InputStreamResource：访问输入流资源的实现类  ByteArrayResource：访问字节数组资源的实现类 |
| Spring提供如下两个标识性接口:  ResourceLoader：该接口实现类的实例可以获得一个Resource实例，该接口里有如下方法： Resource getResource(String location)：该接口仅包含这个方法，该方法用于返回一个Resource实例。ApplicationContext的实现类都实现了ResourceLoader接口，因此ApplicationContext直接获取Resource实例  ResourceLoaderAware：该接口实现类的实例将获得一个ResourceLoader的引用。该接口包含一个方法：void setResourceLoader(ResourceLoader resourceLoader)；该方法用于设置一个ResourceLoader实例。实现该接口的类之后，可以配置成Bean实例，我们不需要为该setter方法显式<property />设置参数传递，Spring容器会自动检测到实现ResourceLoaderAware接口的Bean类然后自动将自动传递给该参数作为ResourceLoader引用。 |

# Spring的Bean之Bean的基本概念

## Bean的定义

<beans.../>元素是Spring配置文件的根元素，该元素可以指定如下属性：

|  |  |
| --- | --- |
| **beans元素的属性** | |
| 属性名 | 说明 |
| **deafult-lazy-init** | 指定该beans元素下配置的所有Bean默认的延迟初始化行为 |
| **default-merge** | 指定该beans元素下配置的所有Bean默认的merge行为 |
| **default-autowire** | 指定该beans元素下配置的所有Bean默认的自动装配行为 |
| **default-autowire-candidates** | 指定该beans元素下配置的所有Bean默认是否作为自动装配的候选Bean |
| **default-init-method** | 指定该beans元素下配置的所有Bean默认的初始化方法 |
| **default-destroy-method** | 指定该beans元素下配置的所有Bean默认的回收方法 |

<bean.../>元素是<beans.../>元素的子元素，beans元素可以包含多个bean子元素，每个bean子元素定义一个Bean，每个Bean对应Spring容器里的一个Java实例。定义Bean时，通常需要指定两个属性：id和class。id属性在Spring容器中应该是唯一的，class属性不能是接口，必须是实现类。

除了可以为bean元素指定一个id属性外，还可以为bean元素指定name属性，用于为Bean实例指定别名。如果需要为Bean实例指定多个别名，可以在name属性中使用逗号、冒号或空格来分隔多个别名，后面通过任一别名即可访问该Bean实例。在一些极端情况下，程序无法在定义Bean时就指定所有的别名，而是需要在其他地方为一个已经存在的Bean实例指定别名，则可使用<alias.../>元素来完成，该元素可指定如下两个属性：

① name：指定一个Bean实例的标识名，表明将为该Bean实例指定别名。

② alias：指定一个别名。

1. **<alias** name="person" alias="jack"**/>**
2. **<alias** name="jack" alias="jackee"**/>**

## 容器中Bean的作用域

|  |  |
| --- | --- |
| **Spring支持的作用域** | |
| 作用域 | 说明 |
| **singleton** | 单例模式，在整个Spring IOC容器中，使用singleton定义的Bean将只有一个实例。 |
| **prototype** | 原型模式，每次通过容器的getBean方法获取prototype定义的Bean时，都将产生一个新的Bean实例。 |
| **request** | 对于每次HTTP请求，使用request定义的Bean都将产生一个新实例，即每次HTTP请求将会产生不同的 Bean实例。只有在Web应用中使用Spring时，该作用域才真正有效。 |
| **session** | 对于每次HTTP Session，使用session定义的Bean都将产生一个新实例，即每次HTTP Session都将产生不同的Bean实例。只有在Web应用中使用Spring时，该作用域才真正有效。 |
| **global session** | 每个全局的HTTP Session对应一个Bean实例。典型的情况下，仅在使用portlet context的时候有效。 只有在Web应用中使用Spring时，该作用域才真正有效。 |

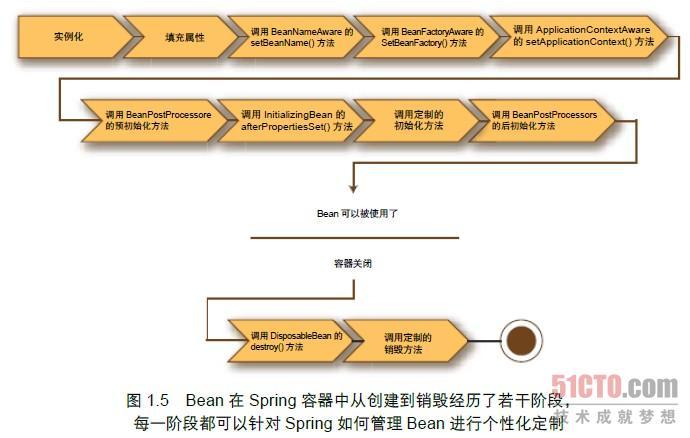
## bean的自动装配

在spring容器中一个Bean依赖注入另一个Bean，我们需要在该bean中添加等方式来配置，但是有时候会出现某种极端情况：我们不清楚该依赖的bean的name，因为并没有配置该bean，这时候我们可以使用Spring的自动装配来解决问题。

Spring容器的自动装配开启方式有两种：beans的default-autowire属性和bean的autowire属性，分别是全局开启和指定bean。该属性有五个值，常用的有三种：

* no : 默认，不使用自动装配
* byType : 根据依赖注入的setter方法返回的值的类型来自动装配Bean。如果Spring容器里只有一个bean返回值类型与之相同，那么就自动装配依赖注入；如果有多个，就会抛出异常；如果没有，Spring容器就不会执行任何操作。
* byName : 根据依赖注入的setter方法名来自动装配Bean。

## spring bean的生命周期



详细描述如下：

第一步：Spring对bean进行实例化。

第二步：Spring将值和bean的引用注入到bean对应的属性中。

第三步：如果bean实现了BeanNameAware接口，Spring将bean的ID传递给setBeanName()方法。

第四步：如果bean实现了BeanFactoryAware接口，Spring将调用setBeanFactory()方法，将BeanFactory容器实例传入。

第五步：如果bean实现了ApplicationContextAware接口，Spring将调用setApplicationContext()方法，将bean所在的上下文引用传入进来。

第六步：如果bean实现了BeanPostProcessor接口，Spring将调用他们的postProcessBeforeInitialization (bean,name)方法

第七步：如果bean实现了InitializingBean接口，Spring将调用它们的afterPropertiesSet()方法，类似的，如果使用init-method声明了初始化方法，该方法也会被调用(没有定义初始化方法，才会使用默认初始化方法)。

第八步：如果bean实现了BeanPostProcessor接口，Spring将调用他们的postProcessAfterInitialization (bean,name)方法。

第九步：此时，bean已经准备就绪，就可以被应用程序使用了，它们将一直驻留在应用上下文中，直到该应用上下文被销毁。

第十步：如果bean实现了DisposableBean接口，Spring将调用它的destroy()接口方法。同样的，如果bean使用了destroy-method声明了销毁方法，该方法也会被调用(没有定义销毁方法，才会使用默认销毁方法)。

## 容器后处理

容器后处理器必须实现 BeanFactoryPostProcessor 接口，该接口有一个方法：void postProcessBeanFactory(ConfigurableListableBeanFactory beanFactory) throws BeansException实现该方法的方法体就是对Spring容器进行处理，这种处理可以对Spring容器进行自定义扩展，当然也可以对Spring容器不进行任何处理。Spring已经提供了几个常用的容器后处理器：

① PropertyPlaceholderConfigurer：属性占位符配置器。

② PropertyOverrideConfigurer：重写占位符配置器。

③ CustomAutowireConfigurer：自定义自动装配的配置器。

④ CustomScopeConfigurer：自定义作用域的配置器。

|  |
| --- |
| #属性占位符配置器的使用  <bean id="propertyConfigurer" class="org.springframework.beans.factory. config.PropertyPlaceholderConfigurer">  <!--使用location属性定义单个配置文件  <property name="location">  <value>classpath:jdbc.properties</value>  </property>  -->    <!-- 使用locations属性定义多个配置文件 -->  <property name="locations">  <list>  <value>classpath:jdbc.properties</value>  </list>  </property>  </bean>  <bean id="dataSource" class="org.springframework.jdbc.datasource .DriverManagerDataSource">  <property name="url">  <value>${database.url}</value>  </property>  <property name="driverClassName">  <value>${database.driver}</value>  </property>  <property name="username">  <value>${database.user}</value>  </property>  <property name="password">  <value>${database.password}</value>  </property>  </bean>  </beans> |
| #创建jdbc.properties文件  database.driver=com.mysql.jdbc.Driver  database.url=jdbc:mysql://localhost:3306/right?useUnicode=true&autoReconnect=true&characterEncoding=UTF-8  database.user=root  database.password=root  jdbc.pool.c3p0.acquire\_increment=2  jdbc.pool.c3p0.max\_size=20  jdbc.pool.c3p0.min\_size=2  jdbc.pool.c3p0.preferred\_test\_query='SELECT 1'  jdbc.pool.c3p0.idle\_connection\_test\_period=18000  jdbc.pool.c3p0.max\_idle\_time=25000 |
| #也可以使用context命名空间  <context:property-placeholder location= "classpath:jdbc.properties"/> |

# Spring bean的配置方式

## XML配置方式

|  |
| --- |
| #1配置bean  <bean id="chinese" class="com.imooc.beans.Chinese"></bean> |
| #2配置依赖  Bean的 依赖注入 通常表现为如下两种形式：  ① 属性：通过<property.../>元素配置，对应设值注入。  ② 构造器参数：通过<constructor-arg.../>元素指定，对应构造注入。  由于Java实例的属性值可以是各种数据类型，除了基本类型、字符串类型等，还可以是其他Java实例，也可以是容器中其他Bean的实例，甚至是Java集合、数组等，所以Spring允许通过如下元素为Bean实例的属性指定值：  ① value：用于指定字符串类型，基本类型的属性值。  ② ref：属性值是容器中的另一个Bean实例。  ③ bean：注入嵌套Bean。  ④ list、set、map和props：注入集合值。 |
| #基本类型及其包装类型，字符串类型可以使用value配置依赖   1. **<bean** id="p" class="com.bean.Person"**>** 2. **<property** name="name" value="tom"**/>** 3. **<property** name="age" value="23"**/>** 4. **</bean>**   #如果需要为Bean设置的属性值是容器中的另一个Bean实例，则使用<ref.../>元素。使用ref元素时可指定如下两个属性：  ① bean：引用不在同一份XML配置文件中的其他Bean实例的id属性值。  ② local：引用同一份XML配置文件中的其他Bean实例的id属性值。   1. **<bean** id="steelAxe" class="com.bean.SteelAxe"**/>** 2. **<bean** id="chinese" class="com.bean.Chinese"**>** 3. **<property** name="axe"**>** 4. **<ref** bean="steelAxe"**/>** 5. **</property>** 6. **</bean>**   简写   1. **<bean** id="steelAxe" class="com.bean.SteelAxe"**/>** 2. **<bean** id="chinese" class="com.bean.Chinese"**>** 3. **<property** name="axe" ref="steelAxe"**/>** 4. **</bean>** |
| #注入集合值，如果Bean的属性是个集合，则可以使用集合元素<list.../> 、<set.../> 、<map.../>和<props.../>元素分别用来设置类型为List、Set、Map和Properties的集合属性值。   1. **<bean** id="chinese" class="com.bean.Chinese"**>** 3. **<property** name="schools"**>** 4. **<list>** 5. **<value>**小学**</value>** 6. **<value>**中学**</value>** 7. **<value>**大学**</value>** 8. **</list>** 9. **</property>** 11. **<property** name="scores"**>** 12. **<map>** 13. **<entry** key="数学" value="87"**/>** 14. **<entry** key="英语" value="89"**/>** 15. **<entry** key="语文" value="82"**/>** 16. **</map>** 17. **</property>** 19. **<property** name="phaseAxes"**>** 20. **<map>** 21. **<entry** key="原始社会" value-ref="stoneAxe"**/>** 22. **<entry** key="农业社会" value-ref="steelAxe"**/>** 23. **</map>** 24. **</property>** 26. **<property** name="health"**>** 27. **<props>** 28. **<prop** key="血压"**>**正常**</prop>** 29. **<prop** key="身高"**>**175**</prop>** 30. **</props>** 31. **</property>** 33. **<property** name="axes"**>** 34. **<set>** 35. <!-- 每个value，bean，ref都配置一个set元素 --> 36. **<value>**普通的字符串**</value>** 37. **<bean** class="com.bean.SteelAxe"**/>** 39. **</set>** 40. **</property>** 42. **<property** name="books"**>** 43. **<list>** 44. **<value>**疯狂Java讲义**</value>** 45. **<value>**JavaWeb与Tomcat开发技术详解**</value>** 46. **<value>**轻量级JavaEE企业应用实战**</value>** 47. **</list>** 48. **</property>** 50. **</bean>** 52. **<bean** id="stoneAxe" class="com.bean.StoneAxe"**/>** 53. **<bean** id="steelAxe" class="com.bean.SteelAxe"**/>**   由于集合元素又可以是基本类型值、引用容器中其他Bean、嵌套Bean或集合属性等，所以<list.../> 、<key.../> 和<set.../>元素又可接受如下子元素：  ① value：集合元素是基本类型或字符串类型。  ② ref：集合元素是容器中另一个Bean实例。  ③ bean：集合元素是一个嵌套Bean。  ④ list、set、map及props：集合元素又是集合。  <props.../>元素用于配置Properties类型的属性，Properties类型是一种特殊的类型，其key和value都只能是字符串。  使用<map.../>元素配置Map属性时比较复杂，因为Map集合的每个元素由key、value两个部分组成，所以配置文件中每个<entry.../>配置一个Map元素，其中entry支持如下4个属性：  ① key：Map的key是基本类型或字符串。  ② key-ref：Map的key是容器中另一个Bean实例。  ③ value：Map的value是基本类型或字符串。  ④ value-ref：Map的value是容器中另一个Bean实例。  ③静态/实例工厂的方法注入：通过调用静态工厂的方法来获取自己需要的对象，为了让spring管理所有对象，我们不能直接通过"工程类.静态方法()"来获取对象，而是依然通过spring注入的形式获取：。   1. <!--(此处获取对象的方式是**从工厂类中**获取方法--> 2. **<bean** name="staticFactoryDao" class="com.DaoFactory" factory-method="getStaticFactoryDaoImpl"**>** 3. <!-- 配置实例工厂方法的参数 4. <constructor-arg value="hello" /> 5. **</bean>** |

## java配置方式

|  |
| --- |
| @Configuration：用于修饰一个java配置类  @Bean ：用于修饰一个方法，该方法返回值定义成容器中的一个Bean。id默认为方法名。否则指定为name属性值  @Value：用于修饰一个Filed值，用于配置属性值。相当于配置的一个变量。  @Import：修饰一个java配置类，用于当前java配置类中导入其他java配置类  @Scope：用于修饰方法，表示对应Bean的作用范围。  @Lazy：用于修饰方法，表示该Bean是否进行延迟初始化。  @DependsOn：用于修饰一个方法，表示初始化该方法对应的Bean之前初始化指定的Bean。  #xml中加载java类配置  <bean class="com.imooc.config.AppConfig"/>  #在java类中加载xml配置  @Configuration  @ImportResource("classpath:spring-core.xml")  public class AppConfig {  ... |
| @Configuration  public class AppConfig {    @Value("18") Integer age;    @Bean(name="chinese")  public Person person(){  Person p =new Person();  p.setUser(getUser());  p.setAge(age);  return p;    }    @Bean(name="user")  public User getUser(){  return new User("root","root");  }  } |
| @Test  public void test02(){  ApplicationContext context =  new AnnotationConfigApplicationContext(AppConfig.class);  Person p = context.getBean(Person.class);  System.out.println(p.getUser().getUsername());  System.out.println(p.getAge());  } |

## 注解方式

|  |
| --- |
| #1装配Bean。xml配置：  <context:component-scan base-package="com.imooc"> </context:component-scan>  在默认情况下，Spring在package对应路径自动搜索所有以@Component、@ Controller 、@Service 和 @Repository 标注的Java类，并将他们当作Spring Bean来处理。默认bean的id为类名首字母小写。  我们还可以通过为<component-scan .. /> 元素添加 <include-filter .. /> 或 <exclude-filter ../>子元素来指定Spring Bean类 ， 只要位于指定的路径下的Java类满足这种条件，即使Java类没有使用Annotation 标注，Spring一样可以把它们当成 Bean类来处理。  <include-filter ... /> 元素使用与指定满足该规则的Java类 会被当成Bean类，而<exclude-filter .../>相反。  　　> type : 指定过滤器类型。  　　> expression : 指定过滤器所需要的表达式。  Spring 内建支持如下过滤器：  　　> annotation : Annotation 过滤器 ，该过滤器需要指定一个Annotation名.  　　> assignable : 类名过滤器， 该过滤器直接指定一个Java类  　　> regex : 正则表达式过滤器，该过滤器指定一个正则表达式 ， 匹配该正则表达式的Java类将满足过滤规则。  　　> aspectj : AspectJ过滤器。  Spring通过使用特殊的Annotation来标注Bean类。  　　>@Component :标注一个普通的Spring Bean类。  　　>@Controller : 标注一个控制器组件类。  　　>@Service: 标注一个业务逻辑组件类。  　　>@Repository : 标注一个DAO组件类。  指定Bean的作用域：@Scope Annotation 在指定Bean 的作用域。如下：  @Scope(value=ConfigurableBeanFactory.SCOPE\_SINGLETON)  @Scope(value="singleton") |
| #2自动装配@Autowired(通过类型自动装配。默认(required=true))  @Autowired  private Axe axe;  注意：如果注入的类型不存在(会报NoSuchBeanDefinitionException异常)，可以使用@Autowired(required=false)，在使用的时候进行需要为空判断。如果接口皆有多个实现方式(会报NoUniqueBeanDefinitionException异常），这个时候我们可以使用@Qualifier("steelAxe")指定具体实现 |

# 基于XML schema的简化配置方式

## 使用p：命名空间简化配置

|  |
| --- |
| //Person Bean  public class Person {  private String username ;  private String password ;  private int age ;  //omit setter and getter methods  public Person(String username,String password,int age) {  this.username = username ;  this.password = password ;  this.age = age ;  } |
| //之前的设值注入代码  <beans ...>  <bean id="person" class="Person">  <property name="username" value="chengxi" />  <property name="password" value="123456" />  <property name="age" value="20" />  </bean>  </beans> |
| //使用p:命名空间简化设值注入  <beans xmlns:p="http://www.springframework.org/schema/c"...>  <bean id="person" class="Person" p:username="chengxi" p:password="123456" p:age="20" />  注意：设置注入要引入另一个bean，需要在属性后面加上-ref后缀(axe-ref) |

## 使用c:命名空间简化构造注入

|  |
| --- |
| //之前的构造注入代码  <beans...>  <bean id="person" class="Person">  <constructor-arg name="username" value="chengxi" />  <constructor-arg name="password" value="123456" />  <constructor-arg name="age" value="20" />  </bean>  </beans> |
| //使用c:命名空间简化构造注入  <beans xmlns:c="http://www.springframework.org/schema/c"...>  <bean id="person" class="Person" c:username="chengxi" c:password="123456" c:age="20" />  </beans> |

## 使用util:命名空间简化XML配置

|  |
| --- |
| util:constant：该元素用于获取指定类的静态Field的值，用于简化FieldRetrievingFactoryBean。  util：property-path：该元素用于获取指定对象的getter方法的返回值，用于简化PropertyPathFactoryBean。  util:list：该元素用于定义一个List Bean，支持使用<value>/<ref>/<bean>等子元素来定义List集合元素，该元素有三个属性:  1>id：该属性指定定义一个名为List Bean的实例  2>list-class：该属性指定Spring使用哪个List实现类来创建Bean实例，默认使用ArrayList作为实现类  3>scope：指定该List Bean实例的作用域  util:properties：该元素用于加载资源文件，并根据加载的资源文件创建一个Properties Bean实例，该属性有三个属性:  1>id:该属性指定定义一个名为id的Properties Bean实例  2>location:指定资源文件的位置  3>scope:指定该Peroperties Bean实例的作用域 |

# Spring 表达式语言（SpEL）

Spring3中引入了spring表达式语言：SpEL装配Bean方式，它通过运行期执行的表达式将值装配到Bean的属性或构造器参数中。

格式：#{...}

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| #1表示字面量值：用来表示浮点数，String值，以及Boolean值  <property name="count" value="{5}"/>  < property name="age" value="The value is #{22}"/>  #3 SpEL运算符：   |  |  | | --- | --- | | 运算符类型 | 运算符 | | 算数运算 | +, -, \*, /, %, ^ | | 比较运算 | <, >, ==, <=, >=, lt, gt, eq, le, ge | | 逻辑运算 | and, or, not, | | | 条件运算 | ?: (ternary), ?:(Elvis) | | 正则表达式 | matches |   <property name="num0" value="#{T(java.lang.Math).PI+3.0}"/>  <property name="falg" value="#{T(java.lang.Math).random() == 1.000}"/>  <property name="falg" value="#{T(java.lang.Math).random() == 1.000 and T(java.lang.Math).PI > 3.0}"/>  <property name="instrument" value="#{piano.song == 'beijing'? piano:violino}"></property>  //正则表达式计算  #{admin.email matches '[a-zA-Z0-9.\_%+-}+@[a-zA-Z0-9.-]+\\.com}  #3引用bean，属性和方法：通过id来引用其他bean  <property name="instrument" value="#{piano}"/>  //属性可以直接访问，访问方法的时候安全访问运算符?.  <property name="song"value="#{piano.song?.toUpperCase()}"></property>  #4类操作（SpEL通过T()来访问类作用域的方法和常量）  <property name="num0" value="#{T(java.lang.Math).PI}"/>  <property name="num1" value="#{T(java.lang.Math).random() }"/>  #5集合操作(访问集合元素[index]，对集合进行过滤.?[]，.^[]，.$[]用来在集合中查询第一个匹配项和最后一个匹配项，.![]会将集合中每个成员中特定的属性放到另个集合中去)  //songs第五个元素  #{jukebox.songs[4].title}  //查询songs中属性为'Aerosmith'的所有元素集合  #{jukebox.songs.?[artist eq 'Aerosmith']  //查询songs中属性为'Aerosmith'的第一个元素 #{jukebox.songs.^ [artist eq 'Aerosmith']  //将title属性投影到一个新的String集合去  #{jukebox.songs.?[artist eq 'Aerosmith'].![title]} |