Spring容器实例化对象，注入对象到需要它们的地方

配置容器管理的bean

使用静态工厂方法实例化bean

使用实例工厂方法实例化bean

注入bean

通过构造函数参数和setter参数指定类的依赖项。通常使用property或**constructor-arg的ref特性指定**

配置bean实例的原始类型字段（int，string）初始化值（）

使用property或**constructor-arg的value特性指定**

Bean实例的作用域。需要维护会话状态的bean应创建成prototype。

**scope="prototype"**

**scope="singleton"**

何时实例化bean。**Prototype范围**总是被**延迟初始化。**

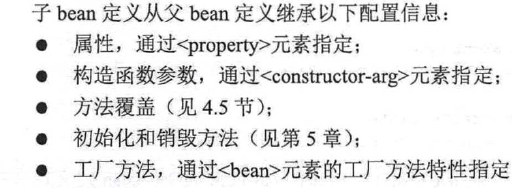
容器启动时

访问时（延迟初始化）lazy-init=true

Bean配置

Bean定义的继承。只用来继承，不实例化的bean配置，设置abstract=true

子bean继承父bean定义的属性，所以在子bean类中要有相应的setter方法。父bean如果有class特性，可以被实例化，作为其他bean的依赖项。子bean继承父bean的配置，如果父bean是非抽象。子bean类不必须extends父bean的class



继承工厂方法

<**bean id="controllerFactory"  
 class="sample.spring.chapter03.bankapp.controller.ControllerFactory"** />  
  
<**bean id="controllerTemplate" factory-bean="controllerFactory"  
 factory-method="getController" abstract="true"**>  
</**bean**>  
  
<**bean id="fixedDepositController" parent="controllerTemplate"**>  
 <**constructor-arg index="0" value="fixedDepositController"** />  
 <**property name="fixedDepositService" ref="fixedDepositService"** />  
</**bean**>

Bean配置的<**constructor-arg>和bean类的构造函数参数匹配**

1使用<**constructor-arg>元素的**index特性

2没有index特性时，spring使用bean类的构造函数参数类型匹配，当继构造函数参数有继承关系时，必须使用type指定类型

3当构造函数参数是原始类型，使用**constructor-arg元素的value特性，当有两个或多个构造函数参数，由value特性指定的字符串值可以互相转换时，需使用type特性。**

4,基于名称的构造函数参数匹配。使用c**onstructor-arg元素的**name特性。需要编译时启用调试标志（javac 的-g选项）或启用了参数名称发现标志（javac 的-parameters选项），当启用调试标志或参数名称发现标志时，构造函数参数和方法参数的名称将保留在生成的.class文件中。还可以使用@ConstructorProperties注解

@ConstructorProperties({**"jmsMessageSender"**,**"emailMessageSender"**,**"webServiceInvoker"**})  
**public** ServiceTemplate(JmsMessageSender jmsMessageSender,  
 EmailMessageSender emailMessageSender,  
 WebServiceInvoker webServiceInvoker) {

}

Bean配置的字符串值转换成对应的类型值，使用PropertyEditor

<**constructor-arg name="propertiesType"**>  
 <**value**>  
 x = y  
 a = b  
 </**value**>  
</**constructor-arg**>  
<**constructor-arg name="anotherPropertiesType"**>  
 <**props**>  
 <**prop key="book"**>Getting started with the Spring Framework</**prop**>  
 </**props**>  
</**constructor-arg**>

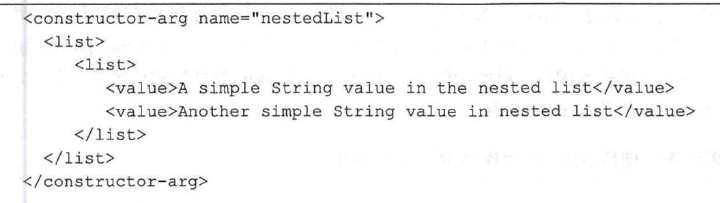
使用map，set，list子元素设置集合类型

<**constructor-arg name="mapType"**>  
 <**map**>  
 <**entry**>  
 <**key**>  
 <**value**>map key 1</**value**>  
 </**key**>  
 <**value**>map key 1’s value</**value**>  
 </**entry**>  
 </**map**>  
</**constructor-arg**>

<**constructor-arg name="setType"**>  
 <**set**>  
 <**value**>Element 1</**value**>  
 <**value**>Element 2</**value**>  
 </**set**>  
</**constructor-arg**>

<**list**>  
 <**value** >1</**value**>  
 <**value**>2</**value**>  
</**list**>

嵌套集合



Bean引用添加到集合

<list>

<ref bean=”aBean />

<ref bean=”bBean” />

</list>

Bean名称添加到集合,list[0]的值是字符串aBean。这里也可以使用<value> aBean </value>。Idref使spring在部署应用程序时验证有无名称是aBean 的bean

<list>

<idref bean=”aBean” />

</list>

添加null到集合

<null />



内置的属性编辑器

CustomCollectionEditor，CustomMapEditor, CustomDateEditor

负责将源集合类型转换为目标集合类型。

1. spring创建xml中定义的<list>等类型对应的实例
2. 根据bean类的属性对应的类型，选择合适的属性编辑器
3. 属性编辑器创建bean类的属性类型的实例，用原类型实例的元素填充它

向spring容器注册属性编辑器

1. 创建实现PropertyEditorRegistrar接口的类，该类负责注册属性编辑器
2. 在应用上下文xml中配置PropertyEditorRegistrar实现

<**bean id="myPropertyEditorRegistrar" class="sample.spring.chapter03.beans.MyPropertyEditorRegistrar"** />

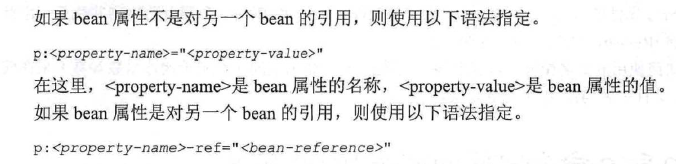
3.配置CustomEditorConfigurer类。CustomEditorConfigurer类实现了BeanFactoryPostProcessor接口

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

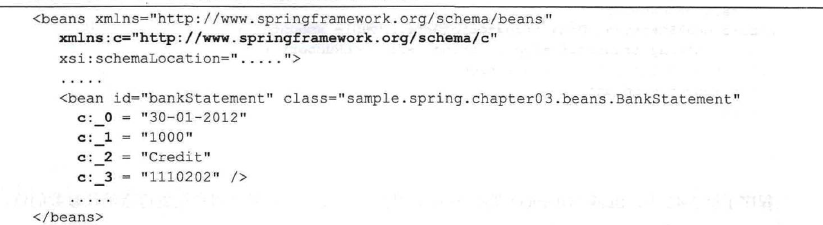
为了使应用程序上下文XML文件中的bean定义不那么冗长, Spring提供了p和c命名空间来分别指定bean属性和构造函数参数的值

使用p命名空间设置bean属性,请将bean属性指定为bean元素的特性,

使用c命名空间设置构造函数参数,请将构造函数参数指定为bean元素的特性



如果类编译未启用debug或“参数名称发现”标志，可以使用索引为构造函数参数提供值



Spring的util模式定义bean

Spring的util模式中所有元素都接受一个 scope特性,该特性指定暴露的bean的范围。

将<list>,<map>等元素定义在外部，需要list，map作为value特性的值的bean属性和构造函数参数

通过ref引用它们



util模式的集合可以指定集合的实现类型。

<**util:list id="listType" list-class="java.util.ArrayList"**>  
 <**value**>A simple String value in list</**value**>  
</**util:list**>

<**util:properties id="branchAddresses" location="classpath:META-INF/addresses.properties"** />

util模式的<constant>元素用于将类的 public static字段暴露为 Spring bean。

<**util:constant id="booleanTrue" static-field="java.lang.Boolean.TRUE"** />

编程的方式创建bean。FactoryBean接口

实现了FactoryBean接口的类可以作为创建bean实例的工厂。在xml配置文件中配置的FactoryBean bean将被spring容器调用bean类的getObject方法，方法返回的bean实例作为其他bean的ref引用。

*<!--配置FactoryBean要创建的bean实例-->*  
<**bean id="eventSenderFactory" class="sample.spring.chapter03.bankapp.event.EventSenderFactoryBean"**>  
 <**property name="databasePropertiesFile" value="META-INF/config/database.properties"**/>  
</**bean**>

访问 FactoryBean实例

使用getBean（）方法访问或使用ref引用配置*FactoryBean的名称将得到执行FactoryBean的getObject返回的对象，如果想访问*FactoryBean实例本身，在*FactoryBean的名称前加上&前缀*

<**property name="eventSender**FactoryBean**" ref="&amp;eventSenderFactory"** />

context.getBean(**"&eventSenderFactory")**

模块化bean配置

可以在多个应用程序上下文XML文件中定义bean,以将应用程序配置进行模块化或结构化。将所有配置XML文件传递给ClassPathXmlApplication Context的构造函数,也可以将所有XML文件导入一个XML文件中,并将该文件传递给 ClassPathXmlApplication Context的构造函数

<**import resource="bankapp-dao.xml"** />  
<**import resource="bankapp-service.xml"** />

依赖注入

内部bean

如果一个bean的依赖项没有被多个bean共享的情况,那么可以考虑将该依赖项定义为内部bean。

使用 depends-on特性控制bean的初始化顺序

可以通过以下两种方式解决隐含依赖问题

1. 更改在应用程序上下文XML文件中bean定义的顺序
2. 使用元素的 depends-on特性显式指定

多个隐式依赖项,逗号分隔

<**bean id="abean" depends-on="bBean, cBean"**>  
 .....  
</**bean**>

singleton和 prototype范围的bean的依赖项

由于 Spring容器仅创建一个 singleton bean的实例,因此 singleton bean在其整个生命周期中保持对同一prototype bean实例的引用。 singleton bean的方法可以使用以下任一方法从 Spring容器获取其 prototype范围的依赖项的新实例:

1.使 singleton bean的类实现 Spring的 ApplicationContextAware接口;

2.使用 Spring的bean模式的<lookup-method:>元素

3.使用 Spring的bean模式的<replaced-method>元素

ApplicationContextAware接口是一个生命周期接口setApplicationContext方法在创建bean实例之后,但在bean实例完全初始化之前由 Spring容器调用。

实现 ApplicationContextAware接口的bean可以通过调用 ApplicationContext的 getBean方法来访问通过Application Context实例注册的其他的bean，也就可以通过调用getBean方法来显式地获取其 prototype范围的依赖项。

<lookup-method>元素

在配置文件中通过元素的name特性指定bean类的lookup方法，方法返回类型表示一个bean, Spring提供方法实现负责从 Spring容器获取bean实例并返回它。Bean特性指定从容器中查找的bean 的id。包含lookup-method>元素的bean，在运行时由spring子类化

<**bean id="customerRequestService"  
 class="sample.spring.chapter04.bankapp.service.CustomerRequestServiceImpl"**>  
 <**constructor-arg name="customerRequestDao" ref="customerRequestDao"** />  
 <**lookup-method bean="customerRequestDetails" name="getCustomerRequestDetails"**/>  
</**bean**>

<replaced-method>元素允许使用不同的实现替换bean类中的任何方法。Name特性表示bean类的方法名称。**Replacer特性表示实现MethodReplacer接口的bean引用。MethodReplacer的接口方法会覆盖getMyBean方法。**

<**bean id="customerRequestService"  
 class="sample.spring.chapter04.bankapp.service.CustomerRequestServiceImpl"**>  
 <**constructor-arg name="customerRequestDao" ref="customerRequestDao"** />  
 <**replaced-method name="getMyBean" replacer="methodReplacer"** />  
</**bean**>

自动注入依赖项（自动装配）

在 Spring中,可以选择使用<property>和<constructor-arg>元素显式指定bean依赖项,或者让 Spring自动解析bean依赖项。 Spring中自动解析依赖项的过程称为“自动装配”。

byType

如果将 autowIre特性的值指定为 byType,那么 Spring将根据其类型自动装配bean属性

2. constructor

如果将 autowire特性的值指定为 constructor, Spring将根据其类型自动装配bean类的构造函数参数。

3. byName

如果将 autowire特性的值指定为 byName,那么 Sprin将根据名称自动选择bean属性。

4. default /no

如果将 autowire 特性的值指定为 default 或no,则对该bean禁用自动装配功能

禁止bean被自动装配到其他bean。可以通过将元素的 autowire- candidate特性的值设置为 false,

自定义bean的初始化和销毁逻辑

1 通过调用bean类的构造函数创建bean实例后, Spring容器通过调用bean的 setter方法来设置bean属性。如果要在设置bean属性之后又在 Spring容器完全初始化bean之前执行自定义初始化逻辑，将初始化方法的名称指定为元素的init-method特性的值。,如果要在包含bean实例的 Spring容器被销毁之前执行自定义清理逻辑,则可以将 cleanup方法的名称指定为元素的

1. InitializingBean和 DisposableBean生命周期接口

一个实现生命周期接口的bean将收到 Spring容器的回调,如 ApplicationContextAware。

InitializingBean接口定义了在设置了bean属性后由 Spring容器调用的 afterPropertiesSet方法

3 @ PostConstruct和@ PreDestroy注释

要在 Spring应用程序中使用@ PostConstruct和@ Pre Destroy注释,需要在应用程序上下文XML文件中配置 Spring的 CommonAnnotationBeanPostProcessor类,

<**bean class="org.springframework.context.annotation.CommonAnnotationBeanPostProcessor"**/>

CommonAnnotationBeanPostProcessor实现了spring的BeanPostProcessor接口

独立应用程序的init-method指定的方法不会在容器关闭时执行，独立应用程序需要由用户自己实现调用cleanup方法的逻辑。ApplicationContext实现的Web版本由 Spring的 WebApplicationContext对象表示。在Web应用程序关闭之前, WebApplicationContext的实现具有调用 singleton bean实例的 cleanup方法的必要逻辑

Spring期望获取Prototype范围对象的对象能显示调用它的清理方法，spring不负责调用

BeanPostProcessor接口提供在bean初始化方法调用前或后调用的方法。Spring将应在用程序上下文XML文件中定义的任何其他bean的实例之前创建BeanPostProcessor的实例。一旦创建了 BeanPostProcessor bean, Spring容器将为其创建的每个bean实例调用BeanPostProcessor的 postProcessBeforelnitialization和 postProcessAfterinitialization方法。

postProcessAfterinitialization方法调用之后bean才会被完全初始化

bean同时实现Spring的 BeanPostProcessor和 Ordered接口。可以使用bean的order字段控制多个BeanPostProcessor接口的调用顺序

BeanPostProcessor的bean可以为其他bean注入依赖，它本身也可以配置依赖其他bean

BeanPostProcessor方法同样作用于FactoryBean。初始化FactoryBean后，从factorybean获取bean时，获取的bean只会调用BeanPostProcessor接口的postProcessAfterinitialization方法

BeanPostProcessor用例

RequireAnnotationBeanPostProcessor：检查标注@require直接的setter方法使用的bean有没有在xml中配置

DestructionAwareBeanPostProcessor是一个BeanPostProcessor的子接口。在bean实例被容器销毁前，spring为每个bean调用postProcessBeforeDestruction方法。Prototype bean不会调用postProcessBeforeDestruction方法。

使用 BeanFactoryPostProcessor修改bean定义

实现 BeanFactoryPostProcessor接口的bean是在实现 BeanPostProcessor接口的bean之前创建的。在 Spring容器加载bean定义之后且在任何bean实例尚未创建之前执行

BeanFactoryPostProcessor不会对每个bean执行接口方法。而是只执行一次。

BeanFactoryPostProcessor示例

**PropertySourcesPlaceholderConfigurer：允许在bean定义中使用占位符。当spring容器加载bean定义时，它从属性文件中提取实际值，并替换**

<**bean id="datasource" class="sample.spring.chapter05.domain.DataSource"**>  
 <**property name="url" value="${database.url}"** />  
 <**property name="username" value="${database.username}"** />  
 <**property name="password" value="${database.password}"** />  
 <**property name="driverClass" value="${database.driverClass}"** />  
</**bean**>

**PropertyOverrideConfigurer：允许在外部属性文件中指定一个bean属性值**

**<bean-name>.<bean-property-name>=<value>**

**使用注解代替xml配置bean**

用@ Component标识 Spring bean

建议使用@ Component注释的特殊形式来注释应用程序的控制器、服务和数据访问对象(DAO)。应该注意的是,@Service、@Controller和@Repository注释使用@ Component注释进行元注释,也就是说,它们本身使用@Component注释进行注释。

Value特性表示bean的id

@Service(value = **"customerRequestService"**)，不指定名称时，spring使用类小写字母开头的类名作为bean名称

使用spring类路径扫描，把注解标注的类自动注册到spring容器

<**context:component-scan base-package="sample.spring"**/>

**base-package指定的包列表用逗号分隔**

**在类中使用注解定义bean的依赖**

@Autowired注释用于通过类型“自动装配依赖项”。 @ Autowired注释可以在构造函数级、方法级和字段级使用。

Spring的 AutowiredAnnotationBeanPostProcessor（一个BeanPostProcessor实现）对使用Spring的@Autowired 或@Inject注释的字段,方法和构造函数进行自动装配。

@Autowired的require特性指定依赖是必须的还是可选的

@Qualifier按名称自动装配依赖项

使用 Spring的@ Qualifier注释以及@ Autowired注释来按名称自动连接依赖项。@Qualifier注释可以在字段级、方法参数级和构造函数参数级来按名称自动装配依赖项。

Spring首先通过使用@Autowired注释的字段、构造函数参数和方法参数“按类型”找到自动装配候选项。然后, Spring使用@ Qualifieri注释指定的bean名称来定位自动装配候选列表中唯一的bea

@Qualifier注解类型，为类型取别名，并使用别名装配。

@Repository(value = **"txDao"**)  
@Qualifier(**"myTx"**)  
**public class** TxDaoImpl **implements** TxDao {}

@Autowired  
@Qualifier(**"myTx"**)  
**private** TxDao **txDao**;

自定义限定符注释（略）

@Inject和@ Named注释

将Java平台的依赖注入注释标准化

@Inject等价@Autowired

如果在类型级别使用@ Named注释,它的作用就像 Spring的@ Component注释。如果在方法参数级或构造函数参数级使用@Named注解,它的作用就像 Spring的@Qualifier注释。

@Inject没有等同于@ Autowired注释的 required特性,但是可以使用Java8的 Optional类型来实现相同的行为。

Spring 支持对Optional 类型的字段、构造函数参数和方法参数的自动装配

@Inject

private Optional<ExternalService> externalServiceHolder;

Spring 通过jSR250的@Resource 注释支持按字段和setter方法的名称自动装配。

@Resource 由CommonAnnotationBeanPostProcessor (一个BeanPostProcesso实现)处理。@Resource 注释的name特性指定要自动装配的bean名称。不能注释构造函数参数和接受多个参数的方法

Bean的其他注解



@Lazy还可以用在依赖项上

可以使用@Lazy注释以及自动装配注释(如@Autowired 、@和@Resource )来延迟自动装配依赖项(即依赖项在被依赖bean访问时自动装配)。

使用@Value 简化注释的bean类的配置

通过<property>和<constructor-arg>元素的value 特性指定了bean所需的配置信息的示例。

注释的用途与<property>和<constructor-arg>元素的value 特性相同。@value 注释可以在字段级、方法级、方法参数级和构造函数参数级使用。Spring中处理@Autowired 和@注释的AutowiredAnnotationBeanPostProcessor 也会负责处理@value注释。

在@Value 注释中使用Spring 表达式语言(spel)

@ value ( " # ( configuration environment ] " )

private string environment

表达式可以使用格式<beanName>.<field或property或method>

SpEL表达式的值由AutowiredAnnotationBeanPostProcessor处理

只有当一个方法使用了@Autowired 或@Resource 或@Inject 注释时,该方法可以在方法级和方法参数级使用@ value注释

在基于xml的bean定义中指定SpEL表达式

<**bean id="sample" class="sample.spring.chapter06.beans.Sample"**>  
 <**property name="environment" value="#{configuration.environment}"** />  
 <**property name="currency" value="Some currency"** />  
 <**property name="country" value="#{configuration.getCountry()}"** />  
 <**property name="state" value="#{configuration.state}"** />  
</**bean**>  
  
<**bean id="configuration" class="sample.spring.chapter06.beans.Configuration"** />

JSR349(Bean Validation API 1.1)允许使用注释来指定JavaBeans 组件的约束。当把Spring 和JSR349结合使用时,可以使用JSR349来注释bean属性和方法,而Spring 将负责验证bean并提供验证结果Spring

@NotNull  
**private long id**;  
  
@Min(1000)  
@Max(500000)  
**private float depositAmount**;

Spring 支持使用JSR349约束验证对象Spring 的LocalvalidatorFactory Bean 类负责在应用程序的类路径中检测JSR349提供程序(如Hibernate Validator )的存在并对其进行初始化,

<**bean id="validator"  
 class="org.springframework.validation.beanvalidation.LocalValidatorFactoryBean"** />

@Autowired  
**private** Validator **validator**;

BeanPropertyBindingResult bindingResult = **new** BeanPropertyBindingResult(fdd, **"Errors"**);  
**validator**.validate(fdd, bindingResult);  
**if**(bindingResult.getErrorCount() > 0) {  
 *logger*.error(**"Errors were found while validating FixedDepositDetails instance"**);  
} **else** {  
 **myFixedDepositDao**.createFixedDeposit(fdd);  
 *logger*.info(**"Created fixed deposit"**);  
}

验证方法，验证方法的参数和返回值

将bean分组。根据不同条件使用不用分组的bean

例如开发和生产环境的数据库