Spring框架简介

Spring框架的核心是提供了依赖注入( Dependency Injection,DI)机制的控制翻转( Inversion of Control,IoC)容器。

可以使用Spring开发Web应用程序、访问数据库、管理事务、创建单元和集成测试等。Spring使用不同的模块实现这些功能,开发应用时只需要引入应用程序所需要的模块。在 Spring框架中,JAR文件的命名惯例如下

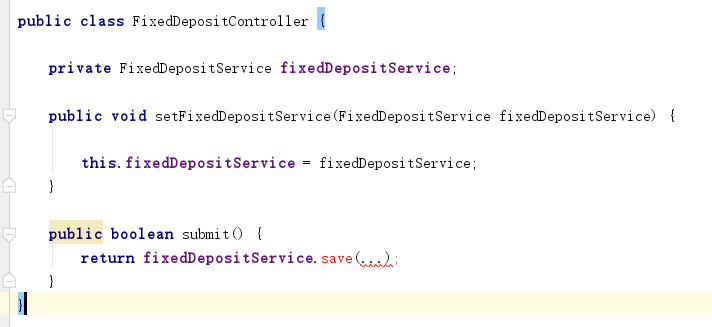
spring-<short-module-name>-<spring-version>.jar

<short-module-name>是spring模块的简称

Java应用程序由互相调用的一组对象组成。如果X对象调用Y对象，那么Y就是X的依赖项，X依赖于Y。

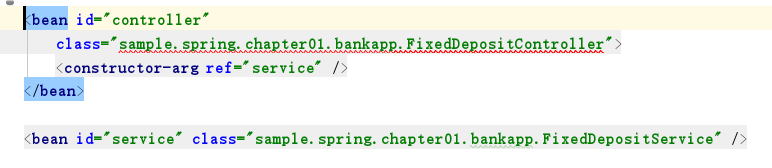
Java的DI设计模式中。对象的依赖项作为对象的构造函数参数或setter方法参数被注入。这些依赖项在对象创建时注入该对象中。

在 Spring应用程序中, Spring loC容器(也称为 Spring容器)负责创建应用程序对象并注入它们的依赖项。 Spring容器创建和管理的应用对象称为bean。 对象及其依赖项的信息是由配置元数据来指定，配置元数据可以通过XMl、java注释，或java代码来指定。spring容器使用java反射API创建对象并注入其依赖项



submit方法中，因为 FixedDepositController调用了 FixedDepositService,所以FixedDepositService就是 FixedDepositController的一个依赖项。

把FixedDepositController和FixedDepositService配置成spring管理的bean



每个<bean>元素定义了一个由 Spring容器管理的应用对象, <constructor-arg>元素指定 service实例作为一个构造函数的参数传递给 controller

每个<bean>元素配置一个由 Spring容器管理的应用程序对象。 id特性指定bean的唯一名称, class特性指定bean的完全限定类名。还可以使用元素的name特性来指定bean的别名。

<property>元素指定由<bean>元素配置的bean的依赖项。<property>元素对应于bean类中的 JavaBean风格的 setter方法,该方法由 Spring容器调用以设置bean的依赖关系

特性name的值**fixedDepositService对应FixedDepositController的**setFixedDepositService()方法,

Ref特性指定的另一个bean的引用将作为setFixedDepositService()方法的参数

<**bean id="controller"  
 class="sample.spring.chapter01.bankapp.FixedDepositController"**>  
 <**property name="fixedDepositService" ref="service"** />  
</**bean**>

创建一个spring容器的实例

Spring的ApplicationContext对象表示spring容器的一个实例。Spring提供了ApplicationContext接口的内置实现，如ClassPathXmlApplicationContext。FileSystemXmlApplicationContext、XmlWebApplicationContext、XmlPortletApplicationContext.。使用哪种实现取决于配置元数据的格式和应用程序类型。

由于 Spring容器管理着由<bean>元素配置的应用程序对象, Spring容器也就需要承担创建并注入它们的依赖关系的责任。不需要直接创建由<bean>元素定义的应用程序对象实例,而是应该从 Spring容器中获取它们。

从spring容器访问bean

调用ApplicationContext接口的getBean方法，参数是bean的id或name特性的值。需要类型转换

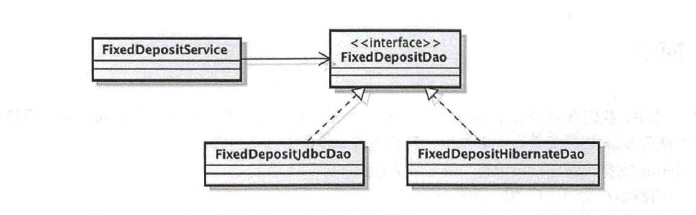
FixedDepositController fixedDepositController = (FixedDepositController) context  
 .getBean(**"controller"**);

**面向接口编程的设计方法**

类依赖于具体实现会导致类之间紧密耦合。如果要替换类的依赖项的其他实现，则需要更改这个依赖于其他类的类本身。

Java接口定义了其实现类应遵循的契约。当类依赖于接口时，替换不同的依赖项实现时，不需要修改类本身。面向接口的设计使类和它的依赖项之间松耦合

提高依赖类的可测试性



Spring对面向接口编程设计方法的支持

创建依赖接口的类

为bean类注入依赖接口的具体实现

**提高依赖类的可测试性**

为了简化FixedDepositService的单元测试，可以模拟一个它依赖接口的实现。

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**

Spring容器可以创建和管理任何类的实例,而不管类是否提供无参数构造函数。

静态工厂方法getfixedDepositDao根据传入的参数创建不同的实例。

<**bean id="factoryDao" class="sample.spring.chapter02.bankapp.FixedDepositDaoFactory"  
 factory-method="getFixedDepositDao"**>  
 <**constructor-arg index="0" value="jdbc"** />  
</**bean**>

将fixedDepositDao注入FixedDepositServiceImpl类中

<bean id=”service” class=”sample.spring.bankapp.FixedDepositServiceImpl”>

<property name=”fixedDepositDao” ref=” **factoryDao**”>

</bean>

2.通过实例工厂方法实例化bean

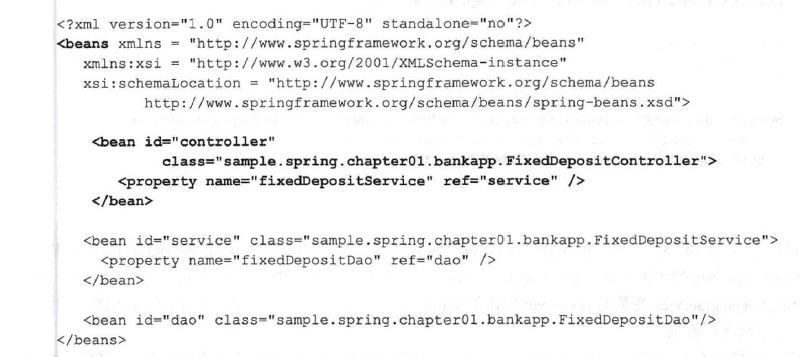
*<!--、先要有一个工厂方法类的实例bean  
 然后使用工厂类实例bean构造需要的bean  
-->*

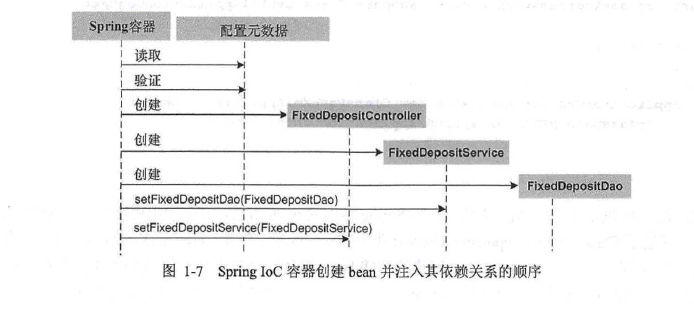
**为工厂方法创建的bean注入依赖**

在 getFixedDepositDao方法返回的 FixedDepositJdbcDao实例上执行基于setter的DI如下

<**bean id="factoryDao" class="sample.spring.chapter02.bankapp.FixedDepositDaoFactory"  
 factory-method="getFixedDepositDao"**>  
 <**constructor-arg index="0" value="jdbc"** />

<property name=”dep” ref=”beanRef” />  
</**bean**>





Spring容器创建bean的顺序取决于他们在applicationContext.xml文件中定义的顺序。还要确保在调用setter方法之前完全配置了一个bean的依赖关系（创建了依赖项bean它的依赖项关系）。创建controller后，在调用setfixedDepositService方法之前（为controller注入依赖。）会先创建fixedDepositService

Bean实例的作用域。

Bean的作用域范围决定了bean实例的生命周期，由bean元素的scope特性定义，默认是singleton。可以是request、sesson、websocket、application和globalSession

Singleton范围bean实例在创建spring容器时创建，容器销毁时销毁。Singleton只创建唯一实例，该实例由依赖于他的所有bean共享

Singleton范围实例，仅在spring容器实例中单例。分别从相同的配置创建出的两个spring容器实例获取同一id的bean得到的是不同的bean实例

需要维护会话状态的bean应创建成prototype。

**scope="prototype"**

**scope="singleton"**

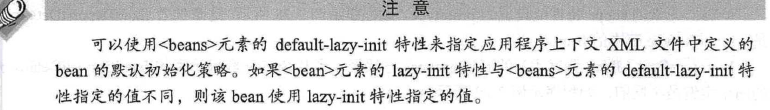
何时实例化bean。

容器启动时

访问时（延迟初始化）lazy-init=true

默认情况下, singleton范围的bean是预实例化的,

**Prototype范围**总是被**延迟初始化。**



**Bean配置**

在某些场景下,为了使bean定义不那么冗长,你可能希望bean定义从另一个bean定义继承配置信息。Bean配置的继承仅仅是bean代码的继承，并不是bean表示的类的继承

**可以继承的配置信息如下**

属性,通过<property>元素指定

构造函数参数,通过< constructor-arg>元素指定;

方法覆盖(见4.5节);

初始化和销毁方法(见第5章)

工厂方法,通过元素的工厂方法特性指定(见2.3节,了解静态和实例工厂方法如何用于创建bean)。

parent特性用于指定父bean的名称。daoTemplate的abstract特性值是true，表示bean定义是抽象的，spring不会创建一个抽象bean。没有指定class特性的bean必须是抽象

<**bean id="daoTemplate" abstract="true"**>  
 <**property name="databaseOperations" ref="databaseOperations"** />  
</**bean**>

<**bean id="fixedDepositDao" parent="daoTemplate"  
 class="sample.spring.chapter03.bankapp.dao.FixedDepositDaoImpl"** />

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

继承工厂方法配置

//定义实例工厂方法类的bean

<**bean id="controllerFactory"  
 class="sample.spring.chapter03.bankapp.controller.ControllerFactory"** />

//指定工厂类的**getController**方法用于创建bean实例，作为父bean配置

<**bean id="controllerTemplate" factory-bean="controllerFactory"  
 factory-method="getController" abstract="true"**>  
</**bean**>

//继承

<**bean id="fixedDepositController" parent="controllerTemplate"**>  
 <**constructor-arg index="0" value="fixedDepositController"** />  
 <**property name="fixedDepositService" ref="fixedDepositService"** />  
</**bean**>

构造函数参数匹配

Spring容器为bean类注入构造函数参数时，使用<constructor-arg>元素指定的构造函数参数和在bean类的构造函数中指定的构造函数参数进行匹配。<constructor-arg>元素使用index特性时，index为0的<constructor-arg>元素将用于构造函数的第一个参数。。。

Spring根据<constructor-arg>元素设置的匹配策略，选择将<constructor-arg>元素注入到构造函数的哪一个参数。

如果构造函数参数是简单的Java类型(如int、 String等),则< constructor-arg>元素的 value特性用于指定构造函数参数的值。如果构造函数参数是对bean的引用,则使用<constructo-arg>元素的ref特性指定bean的名称。

**基于类型的构造方法参数匹配**

如果未指定<constructor-arg>元素的 Index特性,则 Spring容器通过将<constructor-arg>元素引用的类型与bean类的构造函数中指定的参数类型进行匹配。如果构造函数的参数间没有继承关系，即使<constructor-arg>元素的顺序和bean类构造函数参数顺序不一样，spring也能正确地将参数注入

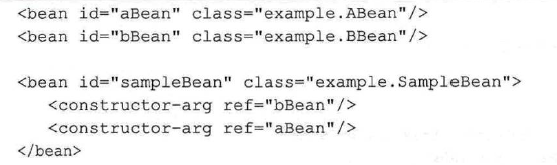
SampleBean构造函数有两个参数ABean和BBean。BBean是 ABean 的子类

public class SampleBean{

public SampleBean(ABean aBean, BBean bEan) {……}

}

Bean配置如下，由于ABean和BBean有继承关系，spring选择以<constructor-arg>元素定义的顺序注入构造函数参数。导致Bbean被注入到aBean中。



解决办法是为元素使用type特性，标识构造函数参数类型  
<**bean id="sampleBean" class="example.SampleBean"**>  
 <**constructor-arg type="sample.spring.chapter03.bankapp.controller.BBean" ref="bBean"**/>   
 <**constructor-arg type="sample.spring.chapterO3.bankapp.controller.ABean" ref="aBean"**/>  
</**bean**>

**构造函数参数表示标准Java类型和自定义类型**

如果构造函数参数的类型是原始类型(如int、long、 boolean等)或 String类型或自定义类型(如 Address), 则<constructor-arg>元素的 value特性用于指定值。如果有两个或多个构造函数参数,由 value属性指定的字符串值可以互相转换,则 Spring容器无法获取构造函数参数的类型(例如,该值表示int、long或 String)。spring选择以<constructor-arg>元素定义的顺序注入构造函数参数。在这种情况下,也需要使用type特性显式指定构造函数参数的类型。

3基于名称的构造函数参数匹配

略

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) {

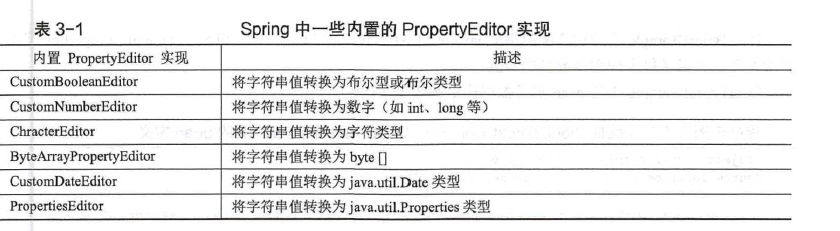
}

3.4配置不同类型的bean属性和构造函数参数

Spring bean的属性和构造函数参数的取值范围可以是 String类型，对另一个bean的引用、其他标准类型(如java.uti.Date、java.ui.Map)或自定义类型(例如 Address)。bean引用使用ref特性指定。String类型和其他标准类型使用value特性指定

为bean属性和构造函数参数指定value特性时，spring使用JavaBeans PropertyEditors将value特性指定的字符串值转换为实际的Java类型的属性或构造函数。

Spring带有几个内置的 PropertyEditor实现,它们的功能是将应用程序上下文XML文件中指定的值转换为bean属性或构造函数参数的对应的Java类型。



注意,默认情况下,并非 Spring中的所有内置 Property Editor实现都注册到 Spring容器中。例如,你需要明确注册 CustomDateEditor以允许 Spring容器执行从字符串值到 java util. Date类型的转换

**指定不同集合类型的值**

<property>和<constructor-arg>元素的<list>、<map>和<set>子元素分别用于设置类型为java.util.List、java.util.Map和java.util.Set的属性和构造方法参数。

使用< constructor-arg>元素的<props>子元素指定 anotherPropertiesType( javautil. Properties类型)的值。在 <props>元素内,每个<prop>子元素指定一个键值对,key特性指定该键,而<prop>元素的内容是该键的值, 还可以使用<constructor-arg>元素的<value>子元素取代<props>元素来指定

<**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**>

<set>子元素指定 setType构造函数参数(类型为 java.util.set)的值。<set> 的每个< value>子元素指定一个set中包含的元素。如果构造函数参数被定义为参数化Set(如Set< Integer>),则 Spring容器使用注册了的属性编辑器CustomNumberEditor来执行将值转换为参数化Set接受的类型。

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

<**constructor-arg name="listType"**>  
 <**list**>  
 <**value** >1</**value**>  
 <**value**>2</**value**>  
 </**list**>  
</**constructor-arg**>

<map>子元素指定类型为java.util.Map的值的<map>的<entry>子元素指定Map中包含的键值对,<key>元素指定键,< value>元素指定键的值。如果构造函数参数被定义为参数化的Map(如Map<Integer,Integer> ),那么 Spring容器使用注册了的属性编辑器来执行键和值的转换,使其被参数化的Map接受。

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

map的key和value特性对应key和value子元素

<**util:map id="mapType" map-class="java.util.TreeMap"**>  
 <**entry key="map key 1" value="map key 1’s value"** />  
</**util:map**>

**嵌套集合**

在应用程序中,集合可能包含Map、Set、Class、 Properties或任何其他Java类型的元素。集合中包含的元素也可以是bean引用。为了解决这种情况, Spring允许以<map>,<set>,<list>和< props>、<ref>等元素作为<map>,<set>,<list>的子元素。

为List<List>类型参数指定值

<**constructor-arg name="nestedList"**>  
 <**list**>  
 <**list**>  
 <**value**>A simple String value in the nested list</**value**>  
 <**value**>Another simple String value in nested list</**value**>  
 </**list**>  
 </**list**>  
</**constructor-arg**>

为Map<List,Set>类型参数指定值

<**constructor-arg name="nestedListAndSetMap"**>  
 <**map**>  
 <**entry**>  
 <**key**>  
 <**list**>  
 <**value**>a List element</**value**>  
 </**list**>  
 </**key**>  
 <**set**>  
 <**value**>a Set element</**value**>  
 </**set**>  
 </**entry**>  
 </**map**>  
</**constructor-arg**>

<**value** > <**set**> <**list**> <ref > 可以互换？

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 />

指定数组的值

<property name="numbersProperty">

<array>

<value>l</value>

<value>2</value>

</array>

</property>



**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

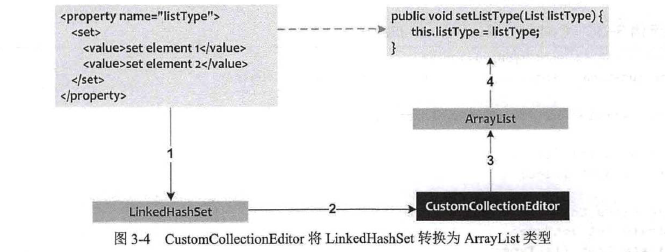
Bean类定义如下**listType是List集合**

**public class** CollectionTypesExample {  
 **private** List **listType**;  
 **private** Set **setType**;  
 **private** Map **mapType**;

bean配置中<set>元素用于初始化CollectionTypesExample 类的**listType字段**

<**bean class="sample.spring.chapter03.beans.CollectionTypesExample"**>  
 <**property name="listType"**>  
 <**set**>  
 <**value**>set element 1</**value**>  
 <**value**>set element 2</**value**>  
 </**set**>  
 </**property**>

首先, Spring创建一个对应于 <set>元素的 LinkedHashSet实例。由于 listType字段的类型为List,spring使用CustomCollectioneditor来设置listType，之后CustomCollectioneditor会创建一个 ArrayList实例,并使用 LinkedHashSet中的元素进行填充。最后, listype字段的值被设置为由 CustomCollectionEditor创建的 ArrayList 实现



**向spring容器注册属性编辑器**

使用 Spring中 CustomEditorConfigurer这个特定的bean。CustomEditorConfigurer类实现了 Spring的 BeanFactoryPostProcessor接口,它由 Spring容器自动检测并执行

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

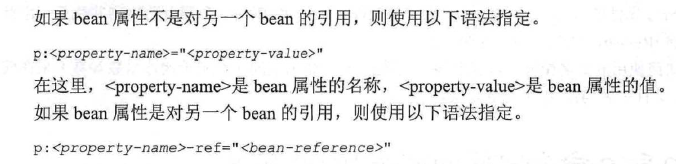
<**bean class="org.springframework.beans.factory.config.CustomEditorConfigurer"**>  
 <**property name="propertyEditorRegistrars"**>  
 <**list**>  
 <**ref bean="myPropertyEditorRegistrar"**/>  
 </**list**>  
 </**property**>  
</**bean**>

**CustomEditorConfigurer的propertyEditorRegistrars属性指定了PropertyEditorRegistrar实现的列表**

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

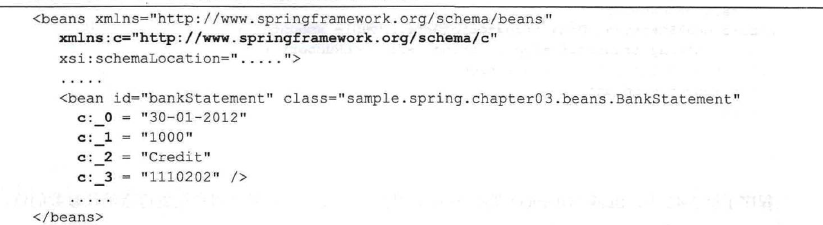
使用p命名空间设置bean属性,请将bean属性指定为bean元素的特性,

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



<entry value-ref="">

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**

Spring的util模式提供了一种简洁的方式来执行常见的配置任务,以此来简化配置bean。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

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。默认情况下,暴露的bean是 singleton范围的。

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**>

list-class特性指定创建的list是java.util.ArrayList的实现，如果未指定list-class特性则默认情况下会创建一个java.util.ArrayList实例。

由于util模式的<list>元素将list实例作为bean暴露,可以将暴露的list实例作为依赖项注入到任何其他bean中

<**constructor-arg name="listType" ref="listType"** />



util模式的< properties>元素使用 properties文件中定义的属性(由 location特性指定)创建了一个java.util.Properties实例,并将这个实例暴露为名为 branchAddresses

<**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"** />

util模式的<property-path>元素用于将bean属性值暴露为一个bean

在名称为dataTypes 的bean中定义了属性dateType。

<**bean id="dataTypes" class="sample.spring.chapter03.beans.DataTypesExample"**>  
 <**property name="dateType" value="30-01-2012"** />  
</**bean**>

<property-path>元素检索dataTypes中 的 dateType属性，并暴露为名称为dateType的bean

<**util:property-path id="dateType" path="dataTypes.dateType"** />

**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。FactoryBean适合用于根据复杂条件决定创建bean的类型以及为创建的bean执行复杂的初始化逻辑

2

FactoryBean接口定义了以下需要实现的方法。

getObjectType:返回由该 FactoryBean实现创建的对象的类型。

getObject:返回由该 FactoryBean要创建的对象。

isSingleton:如果该 FactoryBean实现是 singleton范围对象的工厂,则返回true。

*<!--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**>

*<!--将FactoryBean创建的bean注入-->*

<**bean id="service"  
 class="sample.spring.chapter03.bankapp.service.FixedDepositServiceImpl"**>  
 <**property name="fixedDepositDao" ref="dao"** />  
 <**property name="eventSender" ref="eventSenderFactory"** /></**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。例如, 可以定义一个定义应用程序的数据访问对象(DAO)的 myapp-dao.xml文件,定义服务的 myapp- service.xml。以及定义应用程序控制器的 myapp- controller.xml。在这种情况下,你可以将所有配置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。内部bean在 <property>或< constructor-arg>元素内定义。Property元素不再指定ref特性。FixedDepositDaoImpl的bean不再具有id特性。

<bean id=”service” class="sample. spring. chapter04 bankapp service. FixedDepositServiceImpl">

<property name="fixedDepositDao>

<bean class=”sample.spring.chapter04.bankapp.dao.FixedDepositDaoImpl” />

</property>

<bean>

内部bean未向 Spring容器注册。如果为内部bean定义指定了一个id特性,则它将被 Spring容器忽略。内部bean总是 prototype范围。在本质上是匿名的,并且对于 Spring容器中的其他bean(包含该内部bean定义的bean除外)是不可访问

**使用 depends-on特性控制bean的初始化顺序**

spring按照应用程序上下文XML文件中定义的顺序创建bean。bean的创建顺序也基于bean的相互依赖关系决定。bean的依赖项都通过<property>和< constructor-arg>元素显式指定时，spring容器可以根据这些依赖关系保证创建bean顺序的正确性。如果bean的依赖项是隐式的,则可以使用元素的 depends-on特性来控制由 Spring容器创建bean的顺序，以免在未创建依赖项的情况下创建需要此依赖项的bean

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**

singleton bean依赖于 prototype bean,则会有些复杂。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

1. ApplicationContextAware接口

Spring的 ApplicationContextAware接口由需要访问它们正在运行的 ApplicationContext实例的bean来实现。 ApplicationContextAware接口定义了单一的方法 setApplicationContext,它为实现bean提供了一个Application Context对象的实例。在创建bean时,由 Spring容器调用 setApplicationContext方法

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。（在初始化方法调用之前被调用）

Spring容器调用。

通常我们认为一个bean实例是在它的初始化方法被 Spring容器调用之后才被完全初始化的。只有在bean实例被完全初始化之后,它才被 Spring容器注入依赖的bean

实现 ApplicationContextAware接口的bean可以通过调用 ApplicationContext的 getBean方法来访问通过ApplicationContext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。

@Override  
**public void** setApplicationContext(ApplicationContext applicationContext)  
 **throws** BeansException {  
 **this**.**applicationContext** = applicationContext;  
}

@Override  
**public void** submitRequest(String requestType, String requestDescription) {  
 *// -- populate CustomerRequestDetails object and save it* CustomerRequestDetails customerRequestDetails = **applicationContext** .getBean(CustomerRequestDetails.**class**);  
 customerRequestDetails.setType(requestType);  
 customerRequestDetails.setDescription(requestDescription);  
 **customerRequestDao**.submitRequest(customerRequestDetails);  
}

实现 ApplicationContextAware接口的缺点在于它将bean类与 Spring Framework相耦合

**<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**>

**public abstract** CustomerRequestDetails getCustomerRequestDetails();

@Override  
**public void** submitRequest(String requestType, String requestDescription) {  
 *// -- populate CustomerRequestDetails object and save it* CustomerRequestDetails customerRequestDetails = getCustomerRequestDetails();

……

}

应该注意的是,可以将 getCustomerRequestDetails方法定义为具体方法。由于 getCustomerRequestDetails方法被 Spring覆盖,因此不管在方法中执行任何操作或将其保持为空,都无关紧要

**<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”。 <bean>元素的 autowire特性指定了如何由 Spring自动解决一个bean的依赖项。 autowire特性可以使用以下任何值: default、 byName、 byType、 constructor和no。现在我们来详细分析这些特性值。

1 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

<**bean id="customerRegistrationService"  
 class="sample.spring.chapter04.bankapp.service.CustomerRegistrationServiceImpl"  
 scope="prototype" autowire="byType"** />

Spring有可能找不到任何在 ApplicationContext中注册且其类型能够与属性类型匹配的bean。在这种情况下,不会抛出异常,并且bean属性不会被设置。 如果 Spring在 ApplicationContext中找到多个与属性类型匹配的bean,则会抛出异常。在这种情况下, 不要使用自动装配功能,而是应该使用< property>元素来明确标识bean的依赖项,或者通过将其元素的 primary特性值设置为true,将bean设置为自动装配的主要候选者。

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。抛出异常的情况和上面相同

3. byName

如果将 autowire特性的值指定为 byName,那么 Spring将根据名称自动选择bean属性。例如,如果一个 beanA类定义了一个名为x的字段, Spring会在 ApplicationContext中寻找一个名为x的bean,并将其注入 beanA

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。Spring的默认行为是不使用bean的自动装配。可以通过设置< beans>元素的 default- autowire特性来更改所有bean的默认自动装配行为。

5.使bean无法用于自动装配

Spring容器的默认行为是使bean可用于自动装配。可以通过将元素的 autowire- candidate特性的值设置为 false,使bean不能用于其他bean的自动装配。

6.自动装配的局限性

不能使用自动装配来设置简单Java类型的属性或构造函数参数(如int、long、 boolean、 String、Date等),如果 autowire特性的值设置为 byType或 constructor,则可以自动装配数组、类型集合和映射。

由于bean的依赖项由 Spring自动解析,因此隐藏了应用程序的整体结构。因此,不推荐在大型应用中使用自动装配

**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(如打开文件、创建数据库连接等)，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**destroy-method**特性的值

ApplicationContextAware接口的setApplicationContext方法在init-method之前调用

调用顺序

<**bean id="fixedDepositService"  
 class="sample.spring.chapter05.bankapp.service.FixedDepositServiceImpl"**>  
 <**property name="myFixedDepositDao" ref="myFixedDepositDao"** />  
</**bean**>  
  
<**bean id="myFixedDepositDao"  
 class="sample.spring.chapter05.bankapp.dao.FixedDepositDaoImpl"  
 init-method="initializeDbConnection" destroy-method="releaseDbConnection"** />

FixedDepositServiceImpl's constructor method invoked

FixedDepositDaoImpl's constructor invoked

FixedDepositDaoImpl's initializeDbConnection method invoked

FixedDepositServiceImpl's setMyFixedDepositDao method invoked

//FixedDepositDaoImpl's createFixedDeposit method invoked

1. 使 Spring调用由 destroy- method特性指定的 cleanup方法

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的 ConfigurableApplicationContext( Application Context的子接口)定义了一个 registerShutdownHook方法,该方法向JVM注册了一个关闭钩子。这个关闭钩子负责在JVM停止时关闭 ApplicationContext。在ApplicationContext关闭时会清理所有singleton bean

一个替代使用 registerShutdownHook方法的方案是使用 ConfigurableApplicationContext的close方法,可以调用它来显式关闭 ApplicationContext

1. 清理方法和 prototype bean

prototype范围的bean的中的 destroy- method特性会被 Spring容器忽略。Spring容器期望依赖于prototype bean实例的对象能自己显示调用 prototype bean实例的清除方法

1. 为所有bean指定默认的bean初始化和销毁方法

可以使用<beans>元素的 default-init-method和 default-destroy-method特性来指定bean的默认初始化和

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DisposableBean接口定义了在销毁bean实例时由 Spring容器调用的 destroy方法。与 ApplicationContextAware生命周期接口一样,bean应避免实现 InitializingBean和DisposableBean接口,因为它将应用程序代码与 Spring相耦合。

**3 @ PostConstruct和@ PreDestroy注释**

Spring中的一个bean类可以通过使用@ PostConstruct注释来设置一个方法作为初始化方法,并通过使用@ PreDestroy注释将其注释为清理方法。@PostConstruct和@PreDestroy注释不是 Spring特有的,它们是 Java SE的一部分。要在 Spring应用程序中使用@ PostConstruct和@ PreDestroy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

**5.3使用 BeanPostProcessor与新创建的bean实例进行交互**

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。

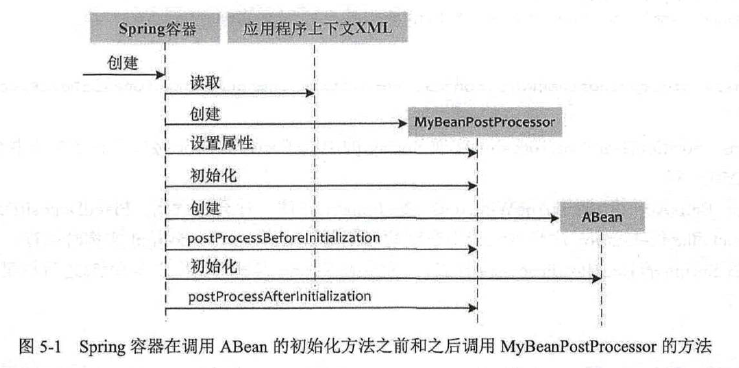
postProcessAfterinitialization方法调用之后bean才会被完全初始化

实现 Spring的 BeanPostProcessor接口的bean是一个特殊的bean类型, Spring容器会自动检测并执行一个 Bean PostProcessor bean.。BeanPostProcessor的方法接受新创建的bean实例及其名称作为参数,它们可能返回相同或修改的bean实例

Object postProcess Beforelnitialization( Object bean, String beanName),

Object postProcessAfterlnitialization( Object bean, String beanName),

由于BeanPostProcessor 的实现方式与任何其他bean的配置方式相同,如果MyBeanPostProcessor 定义了一个初始化方法,容器将调用MyBeanPostProcessor 实例的初始化方法。



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。当 Spring容器创建一个 FactoryBean实例时,将调用 BeanPostProcessor的postProcessBeforeInitialization和 postProccssAfterlnitialization方法。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

Spring容器不会将一个 BeanPostProcessor实现应用于其他 BeanPostProcessor实现例如, 在 MyBank应用程序中,当创建一个 InstanceValidationBeanPostProcessor的实例时, Spring容器不会调用DependencyResolutionPostProcessor的 postProcessBeforelnitialization和 postProcessAfterinitialization方法。

**BeanPostProcessor用例**

**验证bean实例**

在 Spring应用程序中,可能需要在将bean实例注入依赖bean中或在应用程序中被其他对象访问之前验证bean实例是否正确配置。

**public interface** InstanceValidator {  
 **void** validateInstance();  
}

InstanceValidator接口定义了一个 validatelnstance方法,用于验证bean实例是否已正确初始化。validatelnstance方法由 BeanPostProcessor实现调用。

FixedDepositDaoImpl类实现接口

**public class** FixedDepositDaoImpl **implements** FixedDepositDao, InstanceValidator {  
**private** DatabaseConnection **connection**;

**public void** initializeDbConnection() {  
 *logger*.info(**"FixedDepositDaoImpl's initializeDbConnection method invoked"**);  
 **connection** = DatabaseConnection.*getInstance*();  
}  
 @Override  
 **public void** validateInstance() {  
 *logger*.info(**"Validating FixedDepositDaoImpl instance"**);  
 **if**(**connection** == **null**) {  
 *logger*.error(**"Failed to obtain DatabaseConnection instance"**);  
 }  
 }

initializeConnection方法是通过调用 DatabaseConnection类中的 getInstance静态方法实例化**connection。**

由于 initialize Connection初始化方法设置了 connection的值,因此必须在 initializeDbConnection方法之后调用 validateInstance方法。

InstanceValidationBeanPostProcessor类实现了 Spring的 BeanPostProcessor和 Ordered接口。 postProcess Beforelnitialization方法只返回传递给该方法的bean实例。在 postProcess Afterlnitialization方法中,如果发现bean实例的类型为 InstanceValidator,则调用bean实例的 validatelnstance方法。这意味着如果一个bean实现 Instance Validator接口, InstanceValidationBeanPostProcessor在 Spring容器调用bean实例的初始化方法之后调用bean实例的 validateInstance方法。

Ordered接口定义了返回整数值的 getOrder方法。 getOrder方法返回的整数值决定了一个 BeanPostProcessor实现与应用程序上下文XMI文件中配置的其他 BeanPostProcessor实现的优先级。具有较高顺序值的 BeanPostProcessor的优先级较低,并会在具有较低顺序值的 BeanPostProcessor实现之后执行。

<**bean id="fixedDepositDao" class="sample.spring.chapter05.bankapp.dao.FixedDepositDaoImpl"  
 init-method="initializeDbConnection" destroy-method="releaseDbConnection"** />

<**bean  
 class="sample.spring.chapter05.bankapp.postprocessor.InstanceValidationBeanPostProcessor"**>  
 <**property name="order" value="1"** />  
</**bean**>

**RequiredAnnotationBeanPostProcessor**

如果在bean类字段的setter方法上使用 Spring的@Required注释, Spring的 RequiredAnnotationBeanPostProcessor(一个 BeanPostProcessor实现)将检查是否指定了一个< property>元素(或使用p命名空间)来设置 fixed DepositDao属性的值。如果尚未在应用程序上下文XML文件中配置 fixedDepositDao属性,则会导致异常

**DestructionAwareBeanPostProcessor**

有些场景下,可能还需要在一个bean实例被销毁之前与其进行交互。要在bean实例销毁之前与bean实例进行交互,请在应用程序上下文XML文件中配置一个实现 Spring的 DestructionAwareBeanPostProcessor接口的bean。 DestructionAwareBeanPostProcessor是 BeanPostProcessor接口的子接口,并定义了以下方法。

void postProcessBeforeDestruction (Object bean, String beanName)

方法接受即将由 Spring容器销毁的bean实例以及它的名称作为参数。在bean实例被 Spring容器销毁之前, Spring容器会为每个 singleton bean的实例调用 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>**

**第6章使用 Spring进行注释驱动开发**

**使用注解代替xml配置bean**

在前面的章节中,我们看到 Spring容器使用应用程序上下文XML文件中包含的bean定义作为创建bean实例的蓝图。bean的定义指定了关于bean的依赖项、bean的初始化和销毁方法、bean实例的延迟或即刻初始化策略、bean的作用范围等信息。在本章中,我们介绍可以用来在bean类本身中指定相同信息的注释,从而节省在应用程序上下文XML文件中显式配置bean的工作。我们还将介绍 Spring表达

用@ Component标识 Spring bean

Spring的@Component注释是一个类型级的注释,它能标识表示一个 Spring bean(也称为 Spring组件)的类

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。

@Service注释接受一个 value特性,该特性指定了bean在 Spring容器注册的名称。 value特性的作用与元素的id特性相同。

@Service(value = **"customerRequestService"**

可以指定bean名称而不显式指定 value属性,@ Service( value=" fixed DepositService")与@Service("fixedDepositService")的效果是相同的。如果不指定bean名称,则 Spring假定bean名称与以小写字母开头的类的名称相同。

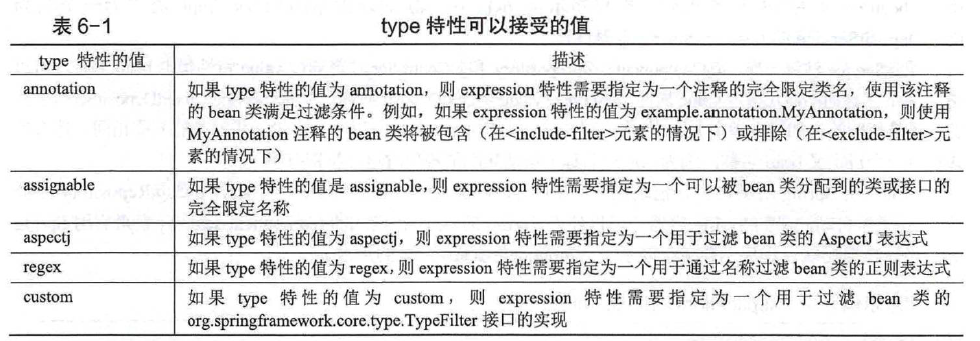
使用 Spring的 context模式的<component-scan>元素来启用 Spring的类路径扫描功能，把注解标注的类自动注册到spring容器

<**beans xmlns="http://www.springframework.org/schema/beans"  
 xmlns:context=**[**http://www.springframework.org/schema/context**](http://www.springframework.org/schema/context)

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

此Xml配置中引用了 Sprig的 context模式,以便其元素可访问。<component-scan>元素的base-package特性指定了一个用于搜索 Spring bean的包列表,该列表以逗号分隔。 <component-scan>元素的< include-fillter>和< exclude-filter>子元素提供了一种简洁的方法来指定用于自动注册的组件类,以及应忽略的类。

<exclude-filter.>和< include-filter>元素定义了一个type特性,它指定了用于过滤bean类的策略,而expression特性指定了相应的过滤器表达式。



因为没有在应用程序上下文XML文件中定义注释的bean类,所以无法使用<property>或<constructor-arg>元素来指定它们的依赖项。因此,需要使用@ Autowired、@ Inject等注释来指定被注释的bean类的依赖项。

**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。AutowiredAnnotationBeanPostProcessor从 Spring容器获取对依赖性类型bean的引用,并将其分配给 bean类

注意,使用@ Autowired注释的字段不需要一定是公有的或具有相应的公有 setter方法。

@Autowired  
**public void** obtainCustomerRegistrationDetails(  
 CustomerRegistrationDetails customerRegistrationDetails) {  
 **this**.**customerRegistrationDetails** = customerRegistrationDetails;  
}

在程序中, getCustomerRegistrationDetails方法使用了@ Autowired注释。如果一个方法使用了@ Autowired注释,则该方法的参数是自动装配的。 请注意,@ Autowired注释方法不需要一定是公有的

如果一个构造函数使用了@ Autowired注释,则构造函数的参数是自动装配的。@ Autowired注释的构造函数不需要是公有的

使用@ Autowired注释时,如果找不到匹配所需类型的bean,则抛出异常。@ Autowired的 required特性可以指定一个依赖项是必选的还是可选的。如果将 required特性的值设置为 false,则即使在 Spring容器中找不到匹配所需类型的bean时,也不会抛出异常。

@Autowired(required=false)

public CustomerRequestServiceImpl(CustomerRequestDetails customerRequestDetails) {}

如果一个bean类定义了一个带@ Autowired注释的构造函数,其中required属性的值设置为true,则该bean类不能有另一个用@Autowired注释的构造函数。

一个bean类可以定义多个@ Autowired注释的构造函数,其中required特性的值需要设置为false.。

一个bean类，它定义了两个使用@ Autowired( required= false)注释的构造函数以及没有注释的构造函数。

@Service(value="customerRequestService")

public class CustomerRequestServiceImpl implements CustomerRequestService {

public CustomerRequestServiceImpl() {

}

@Autowired(required=false)

public CustomerRequestServiceImpl(CustomerRequestDetails customerRequestDetails) {

)

@Autowired(required=false)

public CustomerRequestServiceImpl(CustomerRequestDetails customerRequestDetails,

CustomerRequestDao customerRequestDao) {}

}

如果类型是CustomerRequestDetails和CustomerRequestDao注册到容器中，将调用最后一个构造函数，如果容器中只注册了CustomerRequestDetails类型的bean 则调用倒数第二个，最后调用第一个

@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bean

@Autowired  
@Qualifier(value=**"myFixedDepositDao"**)  
**private** FixedDepositDao **myFixedDepositDao**;

程序示例6-11在方法参数级和构造函数参数级中使用@Qualifier注释

public class Sample {

@Autowired

public Sample(@Qualifier("aBean") ABean bean) {。。。。}

@Autowired

public void doSomething(@Qualifier("bBean") BBean bean, CBeancBean){。。。。}

}

@Qualifier注解类型，为类型取别名，并使用别名装配。

@Qualifier注释在bean类上，用于为bean起一个别名，在依赖于这个bean类的类字段上再次使用@Qualifier注释和起的别名，可以把@Qualifier注释的bean类注入到这个字段

@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）

**JSR330的****@ Inject和@ Named注释将Java平台的依赖注入注释标准化**

JSR330(Java的依赖注入)将Java平台的依赖注入注释标准化。JSR330分别定义了与 Spring的@ )Autowired和@ Qualifier注释类似的@ Inject和@ Named注释。

要使用@ Named和@ Inject注释,需要在项目中包含JSR330JAR文件。

<dependency>

<groupId>javax.inject</groupId>

<artifactId>javax.inject</artifactId>

<version>l</version>

</dependency>

@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 、@Inject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。

@value 注释注释的用途与<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@Inject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), SpEL是一种表达式语言,可用于在运行时查询和操作对象。可以使用SpEL从其他bean中获取配置信息。

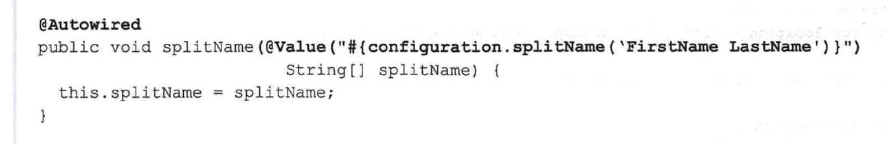
@ value ( " # {configuration.environment} " )

private string environment

表达式可以使用格式<beanName>.<field或property或method>。例如# {configuration.environment}表示获取configuration的bean的environment属性值，# {configuration. getCountry()}表示调用名为configuration的bean的getCountry方法

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, SpEL的使用不限于@vaue注释,可以在应用程序上下文XML文件中包含的bean定义中使用SpEL

<**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"** />

使用 Spring的Validator接口验证对象

Spring的 Validator接口是 Spring Validation API的一部分,它允许执行对象的验证。可以使用 Validator接口来执行任何应用程序层中对象的验证。

要验证 FixedDepositDetails对象的 depositAmount属性值不能为0,我们需要创建一个 Spring的 Validator接口的实现

Validator接口定义了 supports和 validate方法。 supports方法检查提供的对象实例(由 clazz特性表示) 是否可以验证。如果 supports方法返回true,则使用 validate方法验证该对象。validate方法接受要验证的对象实例和一个 Errors实例。 Errors实例的 reject方法用于存储验证期间发生的错误。

@Override  
**public boolean** supports(Class<?> clazz) {  
 **return** FixedDepositDetails.**class**.isAssignableFrom(clazz);  
}  
  
@Override  
**public void** validate(Object target, Errors errors) {  
 FixedDepositDetails fdd = (FixedDepositDetails) target;  
 **if** (fdd.getDepositAmount() == 0) {  
 errors.reject(**"zeroDepositAmount"**);  
 }  
}

FixedDepositServicelmpl的 createFixedDeposit方法在 FixedDepositDao保存到数据存储之前验证FixedDepositDetails对象，

1 创建一个 FixedDepositValidator和 Spring的 BeanPropertyBindingResult 的实例,由 Spring提供的Errors接口的开箱即用的默认实现。

2 调用 FixedDepositValidator的 validate方法,传递 FixedDepositDetails对象和 BeanPropertyBindingResult实例

3 调用 BeanPropertyBindingResult的 getErrorCount方法来检查是否报告了任何验证错误。如果没有

**public void** createFixedDeposit(FixedDepositDetails fixedDepositDetails) **throws** Exception {  
 *// -- create fixed deposit* BeanPropertyBindingResult bindingResult = **new** BeanPropertyBindingResult(  
 fixedDepositDetails, **"Errors"**);  
 FixedDepositValidator validator = **new** FixedDepositValidator();  
 validator.validate(fixedDepositDetails, bindingResult);  
 **if** (bindingResult.getErrorCount() > 0) {  
 *logger*.error(**"Errors were found while validating FixedDepositDetails instance"**);  
 } **else** {  
 **myFixedDepositDao**.createFixedDeposit(fixedDepositDetails);  
 *logger*.info(**"Created fixed deposit"**);  
 }  
}

**使用JsR349注释指定约束**

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

@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LocalvalidatorFactoryBean 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。LocalvalidatorFactoryBean实现了JSR349的 Validator和 ValidatorFactory接口以及 Spring的 Validator接口。

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

import org. springframework.validation.Validator;

@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"**);  
}

程序示例6-45展示了被 validator字段引用的 Spring的 Validator实现。由于 LocalValidatorFactoryBean实现了 Spring的 Validator接口, LocalValidatorFactoryBean实例被分配给 validator字段。

由于 LocalValidatorFactoryBean还实现了JSR349的 Validator和 ValidatorFactory接口,因此可以选择使用JSR349 API来执行 FixedDepositDetails对象的验证。

import javax.validation.Constraintviolation

import javax.validation.Validator

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

**public void** createFixedDeposit(FixedDepositDetails fdd) **throws** Exception {  
 Set<ConstraintViolation<FixedDepositDetails>> violations = **validator**.validate(fdd);  
 Iterator<ConstraintViolation<FixedDepositDetails>> itr = violations.iterator();  
 **if** (itr.hasNext()) {  
 *logger*.error(**"Errors were found while validating FixedDepositDetails instance"**);  
 } **else** {  
 **myFixedDepositDao**.createFixedDeposit(fdd);  
 *logger*.info(**"Created fixed deposit"**);  
 }  
}

validate方法返回一个包含由JSR349提供者报告的约束冲突的 java.util.Set对象。可以通过检查 validate方法返回的 Java uti!set对象来了解是否报告了任何约束冲突。

**验证方法**

JSR349支持验证方法——它们的参数和返回值。要启用方法验证,需要配置 Spring的 MethodValidationPostProcessor，一个将方法验证委托给可用的JSR349提供者的 BeanPostProcessor。

<bean class="org. springframework.validation. beanvalidation.MethodValidationPostProcessor"/>

默认情况下, MethodValidationPostProcessor查找使用 Spring的@ Validated注释的bean类,并为使用JSR349约束注释的方法添加验证支持

@Validated  
**public interface** CustomerRequestService {  
 @Future  
 Calendar submitRequest(@NotBlank String type, @Size(min=20, max=100) String description, @Past Calendar accountOpeningDate);  
}

@Past注释指定了作为 accountOpeningDate参数的值传递的日期必须在过去。 submitRequest方法上的@ Future注释指定了submitRequest方法返回的日期必须在将来。在 CustomerRequestService接口上的@ validated注释表明它包含具有JSR349约束的方法。

**try** {  
 Calendar futureDate = Calendar.*getInstance*();  
 futureDate.add(Calendar.***MONTH***, 1);  
 customerRequestService.submitRequest(**"request type"**, **"description size > 20"**, futureDate);  
} **catch** (ConstraintViolationException ex) {  
 *printValidationErrors*(ex);  
}

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

在xml中定义分组

<**beans profile="dev, default"**>  
 <**util:properties id="dbProps"  
 location="classpath:META-INF/devDB.properties"** />  
</**beans**>  
  
<**beans profile="production"**>  
 <**util:properties id="dbProps"  
 location="classpath:META-INF/productionDB.properties"** />  
</**beans**>

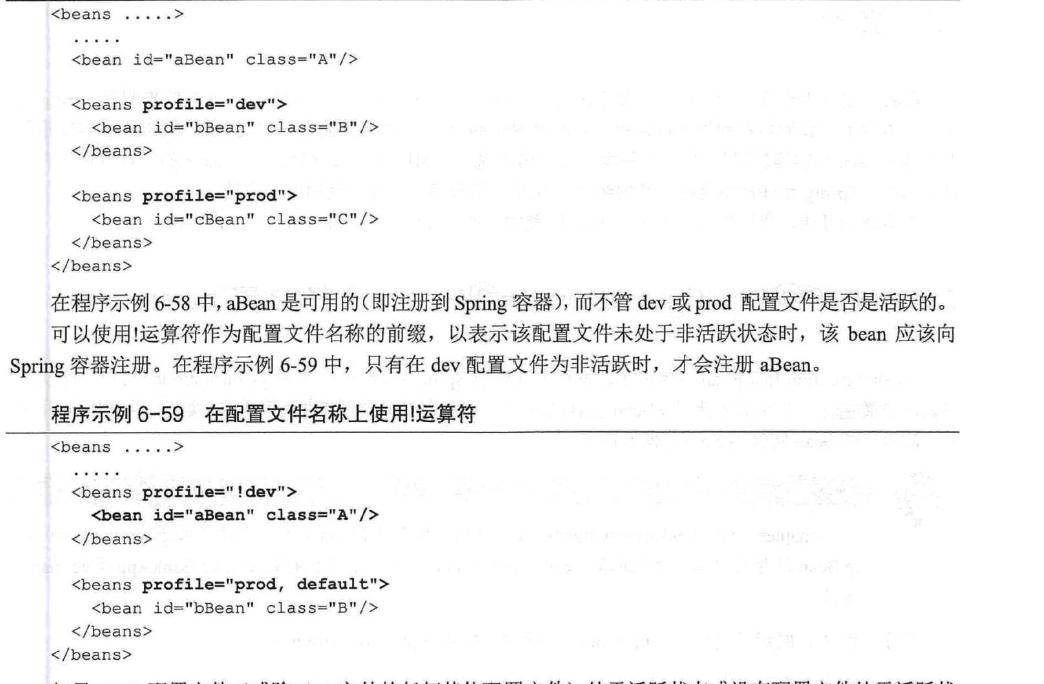
通过注解定义分组

@Repository  
@Profile(**"mybatis"**)  
**public class** FixedDepositMyBatisDao{}

设置活跃的分组

System.*setProperty*(**"spring.profiles.active"**, **"mybatis, production"**);

**spring.profiles.active可以在系统属性，环境变量，jvm系统属性，servlet上下文参数中设置**



**以编程的方式配置容器和bean**

**@Configuration注解标注的类表示类中有@Bean标注的方法，这些方法创建并返回bean实例，返回的bean由spring容器管理**

**Name特性是注册bean用的id。省略name时使用小写字母开头的方法名**

@Bean(name = **"accountStatementDao"**)  
**public** AccountStatementDao accountStatementDao()

@Bean注解还可以包含Autowired、init-method、destroy-method特性，这些和xml中bean上定义的特性等价。Autowired特性默认关闭，禁用bean的自动装配

@Lazy、@DependOn、@Primary、@Scope注解可以跟@Bean一起使用，定义这个bean

@Bean(name = **"customerRegistrationService"**)  
@Scope(value = ConfigurableBeanFactory.***SCOPE\_PROTOTYPE***)  
**public** CustomerRegistrationService customerRegistrationService()

默认情况下@Bean注解的方法返回的bean是singleton

还可以在@Component和@Name注解的bean类中定义@Bean方法，最好不这样做

@Configuration注释使用@Component元注释

**@Bean方法创建的bean的依赖项和注入**

**1通过显式调用创建和返回依赖项的@Bean方法来获取依赖项;**

@Bean(name = **"accountStatementService"**)  
**public** AccountStatementService accountStatementService() {  
 AccountStatementServiceImpl accountStatementServiceImpl = **new** AccountStatementServiceImpl();  
 accountStatementServiceImpl.setAccountStatementDao(accountStatementDao());  
 **return** accountStatementServiceImpl;  
}  
  
@Bean(name = **"accountStatementDao"**)  
**public** AccountStatementDao accountStatementDao() {  
 **return new** AccountStatementDaoImpl();  
}

@Bean方法的行为符合指定的bean配置。例如,如果多次调用 accountStatementService方法, 并不会导致创建 AccountStatementServicelmpl bean的多个实例。因为 AccountStatement Servicelmpl是 singleton范围的,因此 accountStatementService方法将返回同一个 AccountStatementServicelmpl bean实例。这种行为之所以会发生是因为 @Configuration类被子类化,并且它们的@Bean方法被覆盖

**2将bean依赖项指定为@Bean方法的参数,Spring 容器负责调用与依赖项相对应的@bean方法,**

需要在新建的bean上调用setter方法

**3通过在bean类中使用@Autowired 、@Inject 和@Resource 注释来实现自动装配依赖项。**

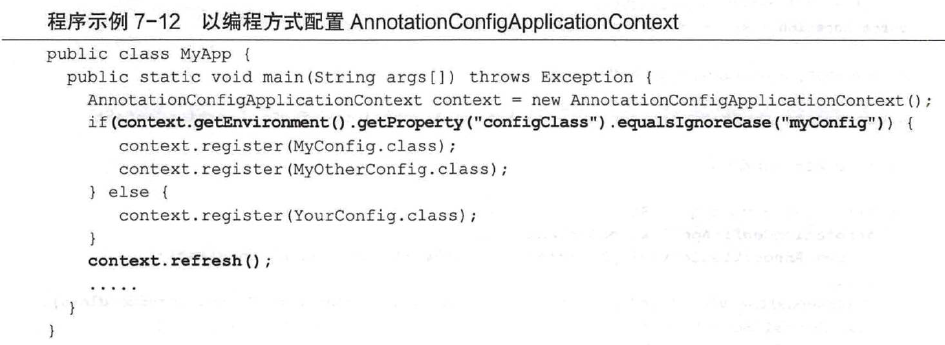
不需要调用setter方法

使用@Configuration注释的类作为bean源，需要使用AnnotationConfigApplicationContext

类的实例表示spring容器

AnnotationConfigApplicationContext context = **new** AnnotationConfigApplicationContext(BankAppConfiguration.**class**);

如果已经在多个@ Configuration类中定义了@Bean方法,那么将所有@Configuration类传递给 AnnotationConfigApplicationContext的构造函数。



可以使用AnnotationConfigApplicationContext 的scan方法来指定要扫描的包,以此取代将Configuration 类显式地添加到AnnotationConfigApplicationContext 中

AnnotationConfigApplicationContext context = **new** AnnotationConfigApplicationContext();

Context.scan(“sampmle.spring”,””);

Context.refresh();

模块化bean配置

可以在多个@Configuration 文件中定义bean.，要组合一个或多个@Configuration文件,可以使用@Import 注释。@ Import注释的作用与 beans模式的 Import元素相同。

@Configuration  
@Import({BankDaosConfig.**class**, BankOtherObjects.**class**})  
**public class** BankServicesConfig{}

**解决依赖关系**

可以使用以下任何方法来处理在不同@Configuration 文件中定义的bean之间的相互依赖关系

1将bean依赖项指定为@Bean方法的参数

2将导入的@Configuration 类自动装配为bean,并调用其@Bean方法来获取依赖项

@Autowired  
**private** BankDaosConfig **bankAppDao**;

@Bean(name = **"fixedDepositService"**)  
**public** FixedDepositService fixedDepositService() {  
 **return new** FixedDepositServiceImpl(**bankAppDao**.fixedDepositDao());  
}

@Bean方法覆盖

AnnotationConfigApplicationContext构造函数参数列表或register后出现的configuration类将覆盖前面的同名@Bean方法

AnnotationConfigApplicationContext context = **new** AnnotationConfigApplicationContext();  
context.register(BankServicesConfig.**class**);  
context.register(BankHibernateDaosConfig.**class**);

使用@Bean方法配置BeanPostProcessors 和BeanFactory PostProcessors . @bean方法必须定义为static

将xml中的bean导入@configuration类

@Configuration  
@ImportResource(locations = **"classpath:META-INF/spring/applicationContext.xml"**)  
**public class** BankHibernateDaosConfig {}

@Profile注释指定了只有在dev或 default配置文件处于活跃状态时,DevDBConfiguration类才会被 Spring处理@PropertySource注释从 devDB properties文件读取数据库配置, 将其添加到 Spring的 Environment对象中。propertySourcesPlaceholderConfigurer方法配置PropertySourcesPlaceholderConfigurer(一个 BeanFactoryPostProcessor),它根据 Environment对象解析由Value注释指定的${…}占位符。例如, driverClass字段上的@vaue("S{ driver ClassName}")注释将driverClass字段的值设置为 Environment对象中的 driverClassName属性值

@Configuration  
@Profile({ **"dev"**, **"default"** })  
@PropertySource(**"classpath:/META-INF/devDB.properties"**)  
**public class** DevDBConfiguration {  
 **private static** Logger *logger* = Logger.*getLogger*(DevDBConfiguration.**class**);  
  
 @Value(**"${driverClassName}"**)  
 **private** String **driverClass**;

}

不使用PropertySourcesPlaceholderConfigurer来配置bean

@Autowired  
**private** Environment **env**;  
  
**public** ProdDBConfiguration() {  
 *logger*.info(**"initializing"**);  
}  
  
@Bean  
**public** DataSource dataSource() {  
 **return new** DataSource(**env**.getProperty(**"driverClass"**), **env**.getProperty(**"url"**), **env**.getProperty(**"username"**),  
 **env**.getProperty(**"password"**));  
}

Profile可以用在bean级别

@Bean  
@Profile({ **"hibernate"**, **"default"** })  
**public** FixedDepositDao fixedDepositHibernateDao(DataSource dataSource) {  
 *logger*.info(**"creating FixedDepositHibernateDao. Database URL is - "** + dataSource.getUrl());  
 **return new** FixedDepositHibernateDao(dataSource);  
}  
设置环境变量中的active

AnnotationConfigApplicationContext context = **new** AnnotationConfigApplicationContext();  
context.getEnvironment().setActiveProfiles(**"mybatis"**, **"production"**);

使用spring jdbc和数据库交互

1配置一个标识数据源的javax.sql.DataSource对象

2实现使用spring jdbc模块类进行数据库交互的DAO

配置DataSource，配置使用DataSource的jdbc模块，注入jdbc模块到dao

使用hibernate

配置DataSource，配置使用DataSource的SessionFactory，注入SessionFactory到dao，从SessionFactory生成Session

使用事务

编程式事务

配置DataSource，配置使用DataSource的transactionManager，配置使用transactionManager的TransactionTemplate，注入TransactionTemplate到dao

声明式事务

配置DataSource，配置使用DataSource的transactionManager，使用spring的tx模式的<annotation-driben>元素启用声明式事务管理，配置<annotation-driben>元素使用transactionManager

配置Hibernate的事务管理

配置DataSource，配置使用DataSource的transactionManager（HibernateTransactionManager），使用spring的tx模式的<annotation-driben>元素启用声明式事务管理，配置<annotation-driben>元素使用transactionManager

创建使用spring的jdbc模块类的DAO

JdbcTemplate

<**bean id="jdbcTemplate" class="org.springframework.jdbc.core.JdbcTemplate"**>  
 <**property name="dataSource" ref="dataSource"** />  
</**bean**>

NamedParameterJdbcTemplate

SimpleJdbcInsert类用元数据来简化sql插入语句的创建

Spring继承hibernate

配置SessionFactory对象，SessionFactory用于创建hibernate的session对象

<**bean id="sessionFactory"  
 class="org.springframework.orm.hibernate5.LocalSessionFactoryBean"**>  
 <**property name="dataSource" ref="dataSource"** />  
 <**property name="packagesToScan" value="sample.spring"** />  
</**bean**>

**packagesToScan配置扫描路径，扫描使用jpa的@Entity注释的类**

**DAO中注入**SessionFactory

@Autowired  
**private** SessionFactory **sessionFactory**;

s**essionFactory**.getCurrentSession().save(bankAccountDetails);

spring的事务管理

编程式事务管理，手动启动、结束和提交事务

声明式事务管理。使用@Transaction注释

使用spring的tx模式的<annotation-driven>元素启用声明式事务管理

<**tx:annotation-driven transaction-manager="txManager"** />

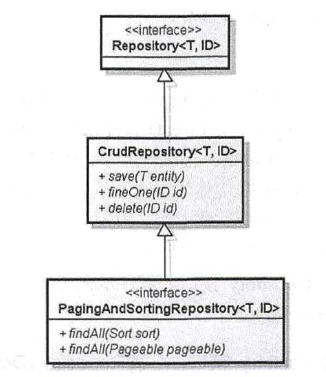
**transaction-manager特性指定事务管理器**

配置transactionManager，不同的数据访问技术使用不同的transactionManager实现

<**bean id="txManager"  
 class="org.springframework.jdbc.datasource.DataSourceTransactionManager"**>  
 <**property name="dataSource" ref="dataSource"** />  
</**bean**>

Spring Data 提供了一个抽象层,减少了实现数据访问层所需的样板代

使用Spring Data时,可以在应用程序中定义与每个域实体对应的存储库接口。存储库包含对实体执行CRUD操作的方法,并对实体进行分页和排序。可以通过继承Repository 、CrudRepository 或Paging AndSortingRepository 接口来创建与域实体对应的存储库。



Spring Data 使用Repository 接口(一个标记接口)来发现应用程序中定义的存储库。存储库接口接受实体类(由T类型指定)及其主键类型(由ID类型指定)作为类型参数。CrudRepository 继承了Repository 并声明了对该实体执行官crud操作的方法

Spring Data 使用Repository 接口(一个标记接口)来发现应用程序中定义的存储库。存储库接口接受实体。

不需要实现这些方法，spring data将提供实现

可以使用count和delete方法的变体

CountBy<field-name>

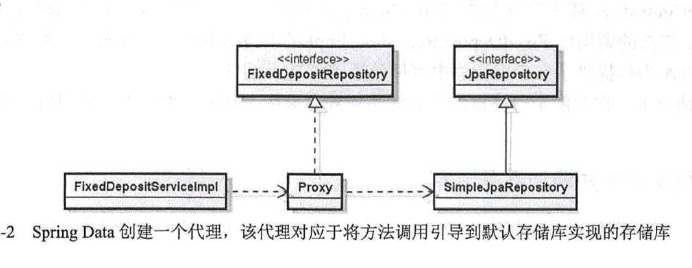
deleteBy<field-name>

spring data的查询构建器为 格式名如findByTenure的方法构建特定于数据存储的查询

对于不同的数据存储，有不同的spring data项目。每个spring data项目提供特性于存储库的接口，使用这些接口可以使用特定于存储库的功能

spring data的工作原理。

为开发者创建的存储库接口创建对应的代理。代理持有spring data默认存储库实现的引用。对FixedDepositRepository接口的调用被代理拦截并委派给SimpleJpaRepository



使用自定义实现代替存储库方法

在类中定义同名的方法代替，使用类名<your-repository-interface>Impl

添加自定义方法

配置内嵌式ActiveMQ代理

*<!-- embedded ActiveMQ broker configuration -->*<**amq:broker**>  
 <**amq:transportConnectors**>  
 <**amq:transportConnector uri="tcp://localhost:61616"** />  
 </**amq:transportConnectors**>  
</**amq:broker**>

配置与内嵌式ActiveMQ实例的连接

Xml中配置

*<!-- JMS ConnectionFactory configuration -->*<**amq:connectionFactory brokerURL="vm://localhost"  
 id="jmsFactory"**>  
 <**amq:trustedPackages**>  
 <**value**>sample.spring.chapter10.bankapp.domain</**value**>  
 <**value**>java.util</**value**>  
 </**amq:trustedPackages**>  
</**amq:connectionFactory**>

配置适配器引用上面的factory

<**bean class="org.springframework.jms.connection.CachingConnectionFactory"  
 id="cachingConnectionFactory"**>  
 <**property name="targetConnectionFactory" ref="jmsFactory"** />  
</**bean**>

使用jmstemplate发送jms消息

<**bean class="org.springframework.jms.core.JmsTemplate" id="jmsTemplate"**>  
 <**property name="connectionFactory" ref="cachingConnectionFactory"** />  
 <**property name="defaultDestination" ref="fixedDepositDestination"** />  
</**bean**>

面向切面编程(AOP)是一种编程方法,其中分布在多个类中的职责被封装到单独的类中,称为“切面”。 跨多个类分配的职责被称为“横切关注点”，也叫通知

触发通知方法调用的方法叫做通知连接点。

使用注解定义切面

@Aspect  
@Component  
**public class** LoggingAspect {

@Before(value = **"execution(\* sample.spring.chapter11.bankapp.service.\*Service.\*(..))"**)  
**public void** log(JoinPoint joinPoint) {}

}

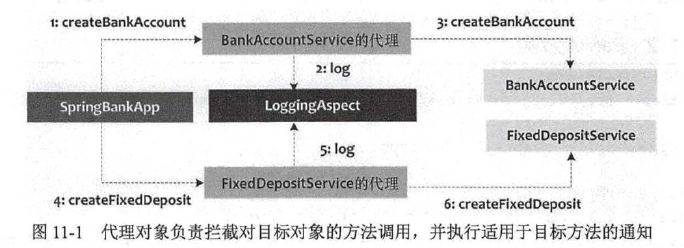
@Before标注通知。Value特性指定试用于此通知的通知连接点表达式

在xml中启用对AspectJ注释的支持

<**aop:aspectj-autoproxy proxy-target-class="false" expose-proxy="true"**/>

@Aspect定义的切面类需要使用@Component，让容器注册这个切面

Spring aop框架是基于代理的。将为通知的连接点（通知的目标对象）创建代理对象，作为调用对象和目标对象的中间对象。



执行通知的时间取决于通知的类型。@Before 注释指定在调用目标方法之前执行该通知,@After 注释指定在调用目标方法后执行该通知,@Around 注释指定在执行目标方法之前和之后都要执行该通知

在使用Spring AOP 时,可以选择通过的ProxyFactoryBean显式创建AOP代理,也可以让Spring 自动创建AOP代理。<**aop:aspectj-autoproxy>元素指示spring aop框架自动为目标对象创建代理，它的expose-proxy特性指定目标对象的方法是否可以通过调用AopContext的currentProxy静态方法来访问aop代理**

**目标方法中调用的方法不会触发通知方法调用除非。** ((BankAccountService)AopContext.currentProxy()).isDuplicataAccount(BankAccountDetaaAs);

切入点表达式标识一个要应用通知的连接点，切入点表达式使用切入点指示符：execution、args、within和this等



前置通知@Before

如果前置的通知不会引发异常，则将始终调用目标方法

返回后通知@AfterReturning

如果目标方法引发异常，将不会执行返回后通知

抛出后通知@AfterThrowing

后置通知@After

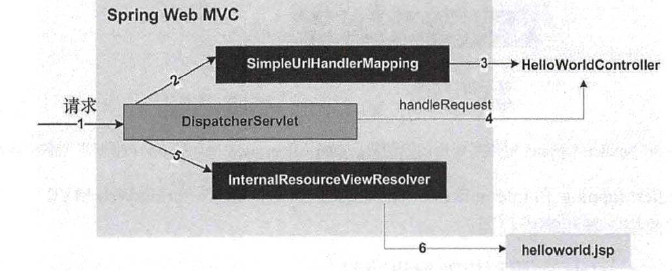
不管目标方法正常完成还是引发异常，都会调用后置通知

围绕通知@Around

可控制目标方法是否执行

使用xml配置aop

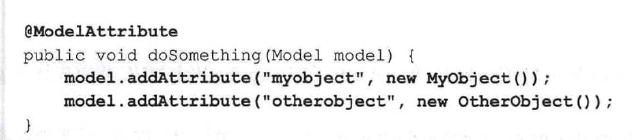
<**bean id="sampleAspect"  
 class="sample.spring.chapter11.bankapp.aspects.SampleAspect"** />  
   
<**aop:config proxy-target-class="false" expose-proxy="true"**>  
   
 <**aop:pointcut expression="execution(\* sample.spring..\*Service.\*(..))"  
 id="services"** />  
 <**aop:aspect id="sampleAspect" ref="sampleAspect"**>  
 <**aop:after-returning method="afterReturningAdvice"  
 returning="aValue"  
 pointcut="execution(\* sample.spring..BankAccountService.createBankAccount(..))"** />  
 <**aop:after-throwing method="afterThrowingAdvice"  
 throwing="exception" pointcut="execution(\* sample.spring..FixedDepositService.\*(..))"** />  
 <**aop:after method="afterAdvice"  
 pointcut="execution(\* sample.spring..BankAccountService.createBankAccount(..)) || execution(\* sample.spring..FixedDepositService.\*(..))"** />  
 <**aop:around method="aroundAdvice"  
 pointcut-ref="services"**/>  
 </**aop:aspect**>  
   
</**aop:config**>



@EXceptionHandler 注释用于一个已经使用了注释的控制器,以标识负责处理控制器抛出的异常的方法。Spring 的HandlerExceptionResolver 负责将异常映射到一个适当的处理异常的控制器方法。你应该注意到, Spring 的mvc模式的元素配置了一个ExceptionHandlerExceptionResolver (一个HandlerExceptionResolver 实现)实例,将异常映射到适当的@ExceptionHandler 注释方法。

在上一章中,我们看到@RequestMapping 方法将模型特性存储在HashMap (或ModelMap )实例中, 并通过ModelAndView对象返回这些模型特性。由@RequestMapping 方法返回的模型特性存储在Spring的Model 对象中。（HashMap中的键值对存储在model对象中？）

@Modelattribute 注释的方法，可以在方法内部显示添加模型特性到model对象中，否则使用方法返回值作为模型特性



可以在控制器中定义多个使用@Modelattribute 注释的方法。当请求被派发到一个控制器的@RequestMapping 注释的方法时,在该方法调用之前会先调用该控制器的所有@Modelattribute 注释的方法。

@Modelattribute 可以用在方法级和方法参数级，在方法参数级使用时可以从model对象获取模型特性。从model中获取不到时，spring调用类型的无参构造函数创建一个此类型的实例

@RequestMapping 注释的方法返回void、org.springframework .ui.Model 或java ,util.Map,则RequestToViewNameTranslator 对象决定要呈现的视图。DefaultRequestToViewNameTranslator 是一个RequestToViewNameTranslator 的实现, RequestToViewNameTranslator 使用请求URI来决定要呈现的逻辑视图的名称.DefaultRequestToTranslator

使用@SessionAttributes 注释缓存模型特性

为了不重复调用多次@Modelattribute注释的方法，使用@SessionAttributes注释控制器类，该注释指定了存储在请求之间的HttpSession 中的模型特性。

如果使用@SessionAttributes 注释,则只有在HttpSession 中找不到由@ModelAttribute 注释指定的模型特性时才调用@ModelAttribute 注释的方法

@Controller  
@RequestMapping(path = **"/fixedDeposit"**)  
@SessionAttributes(names = { **"newFixedDepositDetails"**,  
 **"editableFixedDepositDetails"** }, types = {FixedDepositDetails.**class**})  
**public class** FixedDepositController {}

控制器通过@ModelAttribute 注释的方法、@RequestMapping 方法(返回ModelAndView、Model 或Map)以及通过直接向Model 对象添加等方式来提供模型特性。控制器通过任何方法提供的模型特性都可以通过@SessionAttributes 注释存储在HttpSession 中

删除httpSession中不需要的模型特性。在@RequestMapping注释的方法上添加SessionStatus类型的参数。Spring会为方法提供SessionStatus类型的实例，sessionStatus.setComplete()调用会指示spring从Session对象中删除当前控制器的模型特性

RESTful Web 服务与基于SOAP的Web服务相比,实现起来更简单、更具可扩展性。在基于SOAP的Web服务中,请求和响应始终为XML格式在RESTful Web 服务中,可以使用JSON(JavaScript X 对象表示法)XML、纯文本等作为请求和响应。

1. 使用ResponseEntity指定HTTP响应

ResponseEntity 表示一个由响应头、响应体和状态码组成的HTTP响应。在ResponseEntity 对象上的响应体对象由Spring Web MVC 写入HTTP响应体。Spring Web 使用适当的HttpMessageConverter将对象转换为客户端应用程序所期望的格式（根据请求头accept）

@RequestMapping(method = RequestMethod.***GET***)  
**public** ResponseEntity<List<FixedDepositDetails>> getFixedDepositList() {  
 *logger*.info(**"listFixedDeposits() method: Getting list of fixed deposits"**);  
 **return new** ResponseEntity<List<FixedDepositDetails>>(  
 **fixedDepositService**.getFixedDeposits(), HttpStatus.***OK***);  
}

如果不需要在响应中发送HTTP状态代码,则可以使用Spring 的HttpEntity 类代替ResponseEntity。HttpEntity 表示包含头和体的HTTP请求或响应。RequestEntity 和ResponseEntity 是HttpEntity的子类

2在 HttpServletResponse 对象上设置响应体和响应头

RequestMapping(method = RequestMethod.GET)

public void doSomething(HttpServletResponse response) throws IOException (

response.setHeader("some-header", "some-value");

response.setStatus(200) ;

response.getWriter().write("Hello world !");

}

3使用@ ResponseBody将方法的返回值作为http响应体

Spring 使用适当的HttpMessageConverter 实现将@ResponseBody 注释方法返回的值写入了HTTP响应体

异步处理

由于Spring Web 中的异步请求处理基于Servlet 3,所以web.xml必须引用Servlet 3 XML模式，此外,必须将<async-supported>元素添加到web.xml文件的DispatcherServlet定义中以指示它支持异步请求

如果@RequestMapping 注释方法返回Callable或DeferredResult ,则Spring Web 负责在一个应用程序线程(而不是Servlet 容器线程)中处理Callable 以产生结果。

Spring 中的类型转换和格式化支持

Spring 的Converter接口简化了一个对象类型到另一种对象类型的转换。

Spring 的Formatter 接口将对象类型转换为本地化的String 表示形式，反之亦然

创建自定义转换器

配置和使用自定义转换器

要使用自定义转换器,需要使用Spring 的ConversionService 注册自定义转换器。ConversionService 充当转换器和格式化器的注册表,Spring 将类型转换的责任委托给注册的ConversionService .

默认情况下,Spring的mvc模式的<annotation-driven >元素会自动向容器注册一个Spring 的FormattingConversionService (一个ConversionService的实现) 。FormattingConversionService会注册几个spring内置的转换器和格式化器,

要使用FormattingConversionService 实例注册自定义转换器,请配置Spring 的FormattingConversionServiceFactoryBean (一个用于创建和配置FormattingConversionService 实例的FactoryBean 实现),

<**mvc:annotation-driven conversion-service="myConversionService"** />  
  
<**bean id="myConversionService"  
 class="org.springframework.format.support.FormattingConversionServiceFactoryBean"**>  
 <**property name="converters"**>  
 <**set**>  
 <**bean  
 class="sample.spring.chapter15.converter.IdToFixedDepositDetailsConverter"** />  
 </**set**>  
 </**property**>

</bean>

Spring 容器在数据绑定期间使用FormattingConversionService 注册的转换器和格式化器执行类型转换。也就是说将请求参数绑定到控制器参数时可以执行转换？

*显示转换  
String id="";  
conversionService.convert(id,FixedDepositDetails.class);*

格式化器

**public class** AmountFormatter **implements** Formatter<Long>{

需实现方法

**public** String print(Long object, Locale locale)

**public** Long parse(String text, Locale locale) **throws** ParseException {

向FormattingConversionService 注册的AmountFormatter 在数据绑定和渲染期间应用于所有Long类型字段

**${**fixedDeposit.depositAmount**}**

EL表达式不会转换，要使用

<**spring:eval expression="fixedDeposit.id"** />

可以使用Spring 的AnnotationFormatterFactory 控制格式化应用的字段。一个AnnotationFormatterFactory的实现为使用特定注释的字段创建格式化器。

使用AnnotationFormatterFactory 来格式化仅使用@AmountFormat 注释的Long类型字段

**public class** AmountFormatAnnotationFormatterFactory **implements** AnnotationFormatterFactory<AmountFormat> {

spring security

配置拦截web请求，2种方式

1. Web.xml中配置spring的DelegatingFilterProxy过滤器。

过滤器的<**filtername**>springSecurityFilterChain</**filter-name**>有特殊意义。DelegatingFilterProxy过滤器接收到的web请求，会委派给根应用上下文中名为springSecurityFilterChain的spring bean。springSecurityFilterChain bean是由spring security框架创建的。

配置过滤器，按filter的name将请求委派到spring security创建的bean处理

1. 在根应用上下文中，使用security模式的<http>元素配置

Spring Security 框架解析<http>元素,并向Spring容器注册一个名为springSecurityFilterChain 的bean .springSecurityFilterChain bean 负责处理Web请求的安全

springSecurityFilterChain bean 表示FilterChainProxy bean 的一个实例，它包含由<http>元素的子元素添加到链中的Servlet过滤器链

身份验证

Spring Security 的AuthenticationManager 负责处理身份验证

AuthenticationManager可以配置一个或多个Authentication Provider ,以尝试对用户进行身份验证

<**authentication-manager**>  
 <**authentication-provider**>  
 <**user-service**>  
 <**user name="admin" password="admin" authorities="ROLE\_ADMIN"** />  
 </**user-service**>  
 </**authentication-provider**>  
</**authentication-manager**>

<**authentication-manager**>元素配置了一个AuthenticationManager实例

<**authentication-provider**>元素配置了一个DaoAuthenticationProvider实例（一个AuthenticationProvider的实现）

DaoAuthenticationProvider使用配置的UserDetailService（相当于DAO）根据提供的用户名从用户存储库加载用户详细信息

还可以使用<**jdbc-user-service**>、<**ldap-user-service**>元素

Security 的Authentication对象包含已认证的用户的信息

要为应用程序添加方法级安全,需要执行以下操作:

1通过使用security 模式的<global-method-security >元素为应用程序配置方法级安全

2将@Secured 注释添加到你希望防止未经授权访问的方法中。