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

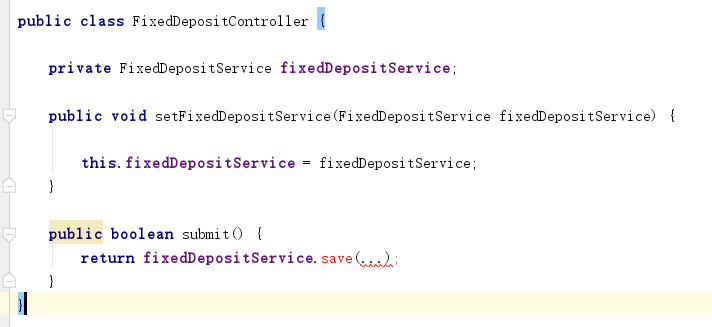
spring-<short-module-name>-<spring-version>.jar

<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的依赖性，X依赖于Y。

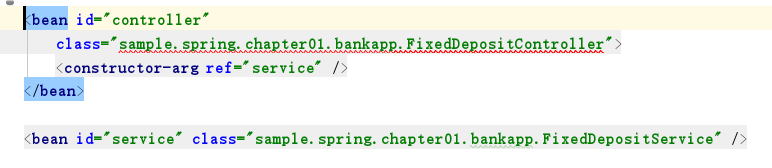
Java的DI设计模式中。对象的依赖项作为对象的构造函数参数或setter方法参数被注入。这些依赖性在对象创建时注入该对象中。

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

在一个独立java应用程序中使用 Spring的DI功能,需要遵循以下步骤

1. 确定应用程序对象及其依赖关系;
2. 根据步骤1中确定的应用程序对象创建POJO类
3. 创建描述应用程序对象及其依赖项的配置元数据
4. 创建一个 Spring loC容器的实例并将配置元数据传递给它;
5. 从Spring IoC容器实例中访问应用程序对象。

FixedDepositController对象依赖于FixedDepositService对象，而 FixedDepositService对象依赖于FixedDepositDao对象

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。

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。

<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。

一个独立的应用程序,因此可以使用 ClassPathXmlApplication Context或FileSystemXmlApplicationContext类来创建一个 Spring容器的实例。应该注意到, ClassPathXmlApplicationContext类从指定的类路径位置加载应用程序上下文XML文件

ApplicationContext context = **new** ClassPathXmlApplicationContext(  
 **"classpath:META-INF/spring/applicationContext.xml"**);

从spring中访问bean

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

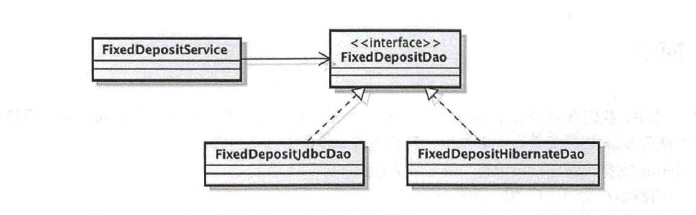
FixedDepositController fixedDepositController = (FixedDepositController) context  
 .getBean(**"controller"**);

**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



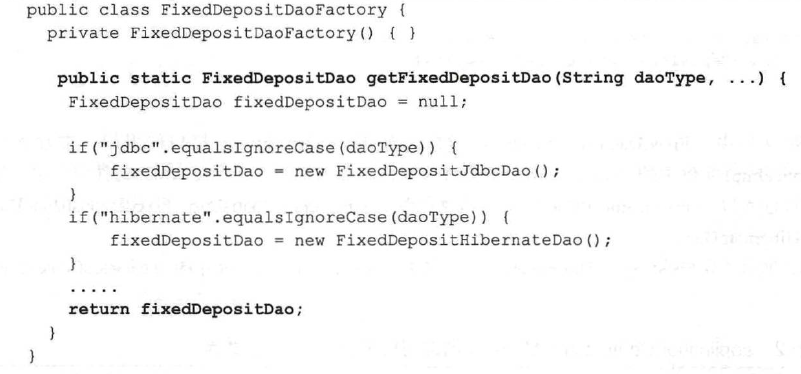
**提高依赖类的可测试性**

为了简化FixedDepositService 的单元测试,我们可以把原来对具体数据库操作的实现去掉,用一个实现了 FixedDepositDao接口但是不需要数据库的代码来代替。通过对依赖接口的模拟依赖类实现,你可以减少针对单元测试的基础设施设置的工作量

使用静态和实例工厂方法创建spring bean

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。

FixedDepositDaoFactory类中的静态工厂方法getFixedDepositDao（）如下



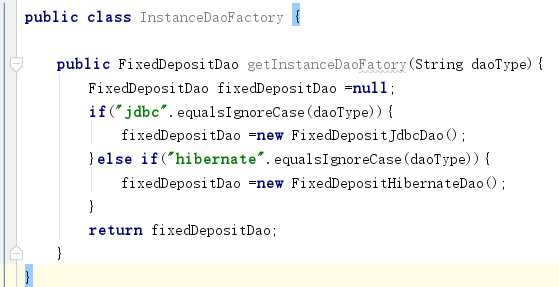
使用静态方法创建的bean。如果需要静态工厂方法创建的多个实例，按下面的格式定义<bean>元素，传入不同的构造函数参数值。

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

class特性指定了定义静态工厂方法的类的完全限定名称。factory-method特性指定了Spring容器调用的获取 FixedDepositDao对象实例的静态工厂方法的名称。<constructor-arg>元素用于传递构造函数的参数。index特性值为0意味着< constructor-arg>元素为 getFixedDepositDao工厂方法的第一个参数(即dao'Type)

2.通过实例工厂方法实例化bean

InstanceDaoFactory类中的实例工厂方法getInstanceDaoFatory（）如下



*<!--、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  
-->*<**bean id="instanceDaoFactoryBean" class="sample.spring.chapter02.bankapp.InstanceDaoFactory"** />  
<**bean id="instanceFactoryDao" factory-bean="instanceDaoFactoryBean" factory-method="getInstanceDaoFatory"**>  
 <**constructor-arg index="0" value="hibernate"** />  
</**bean**>

**为工厂方法创建的bean注入依赖**

可以将bean依赖项作为参数传递给工厂方法,也可以使用基于 setter的DI来注入由静态或实例工厂。

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

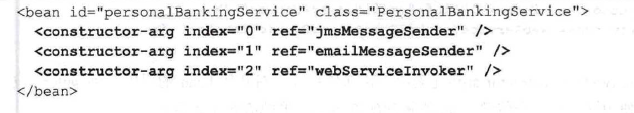
<**bean id="factoryDao" class="sample.spring.chapter02.bankapp.FixedDepositDaoFactory"  
 factory-method="getFixedDepositDao"**>  
 <**constructor-arg index="0" value="jdbc"** />

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

基于构造函数的DI

在 Spring中,依赖注入是通过将参数传递给bean的构造函数和setter方法来实现的。可以混合使用构造函数和setter方法的依赖注入

<constructor-arg>元素指定了传递给personalBankingService实例的构造函数参数的详细信息。Index特性值为0，表示<constructor-arg>元素对应于第一个构造函数参数。



Bean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

默认情况下, singleton范围的bean是预实例化的,这意味着在创建 Spring容器的实例时, 将创建一个 singleton范围bean的实例。延迟初始化singleton范围的bean。只有第一次请求时，才创建bean

<bean id=”lazy” class=”lazy” lazy-init=”true”>

Prototype范围的bean，总是被延迟初始化。需要维护会话状态的bean应创建成prototype。Spring容器总是返回一个 prototype范围bean的新实例

**bean的配置**

bean定义的继承

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

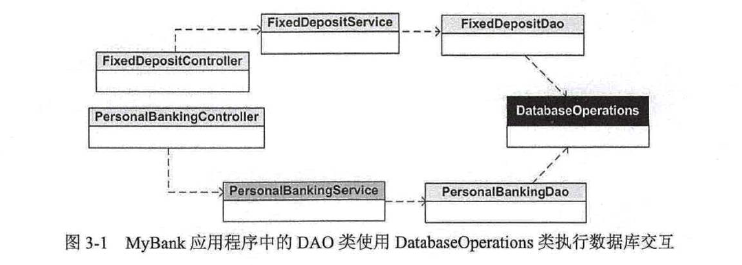
构造函数参数,通过< constructor-arg>元素指定;

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)。

假设应用定义了一个可以简化与数据库交互的 DatabaseOperations类,因此应用中的所有DAO都依赖于 DatabaseOperations



不使用继承的bean配置

<**bean id=" fixedDepositDao "  
 class="sample.spring.chapter03.bankapp.dao.FixedDepositDaoImpl "**>  
 <**property name="databaseOperations" ref="databaseOperations"** />

</**bean**>

如果应用程序中的多个bean共享一组公共的配置(属性、构造函数参数等),则可以创建一个bean定义,作为其他bean定义的父定义。

fixedDepositDao，personalBankingDao 都使用属性databaseOperations，可以将属性databaseOperations写成公共的bean配置

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"** />

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，则可以被实例化，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

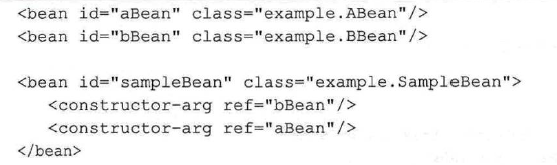
SampleBean构造函数有两个参数ABean和BBean。BBean是 ABea 的子类

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

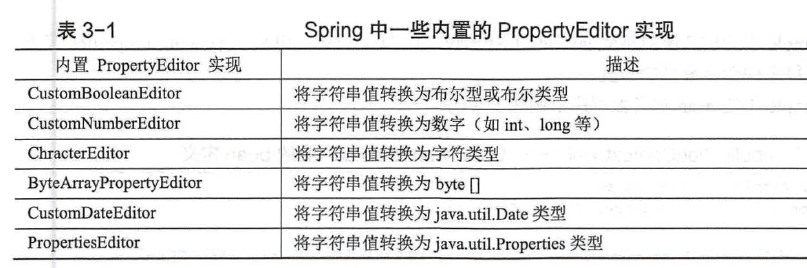
略

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。



Spring容器使用注册的 Property Editors转换，内置的PropertyEditor实现只有部分已经注册过。

java.util.Properties的属性(或构造函数参数)的值也可以使用<property> (或<constructor-arg>)元素的<props>子元素来指定。

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。

<**constructor-arg name="anotherPropertiesType"**>  
 <**props**>  
 <**prop key="book"**>Getting started with the Spring Framework</**prop**>  
 </**props**>  
</**constructor-arg**>

<**constructor-arg name="anotherPropertiesType "**>  
 <**value**>  
 x = y  
 a = b  
 </**value**>  
</**constructor-arg**>

<props>子元素指定java.util. Properties类型的值。在<props>元素内,每个子元素指定一个键值对,key特性指定该键,而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="listType"**>  
 <**list**>  
 <**value** >1</**value**>  
 <**value**>2</**value**>  
 </**list**>  
</**constructor-arg**>

<list>子元素指定java.uti.List<Integer>的值。元素的<value>子元素指定列表中包含的项目。由于 listType构造函数参数的类型为List<Integer>, Spring容器使用 CustomNumberEditor将由<value>元素指定的字符串值转换为java.lang.Integer类型

<**constructor-arg name="mapType"**>  
 <**map**>  
 <**entry**>  
 <**key**>  
 <**value**>map key 1</**value**>  
 </**key**>  
 <**value**>map key 1’s value</**value**>  
 </**entry**>  
 <**entry**>  
 <**key**>  
 <**value**>map2</**value**>  
 </**key**>  
 <**value**>map key 2’s value</**value**>  
 </**entry**>  
 </**map**>

<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。

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**>

将bean引用添加的集合类型

<bean .....>

<constructor-arg name="myList">

<list>

<ref bean="aBean" />

<ref bean="bBean\*' />

</list>

</constructor-arg>

</bean>

<bean id="aBean\*' class="somepackage.ABean" />

<bean id="bBean" class="somepackage.BBean" />

指定数组的值

<property name="numbersProperty">

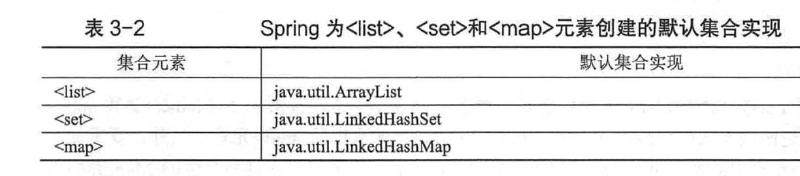
<array>

<value>l</value>

<value>2</value>

</array>

</property>



3.5内置属性编辑器

CustomCollectionEditor属性编辑器负责将源集合类型转换为目标集合(如java.util. LinkedList ->java.util.ArrayList)类型。默认情况下, CustomCollectionEditor会对Set、 SortedSet和List类型进行注册

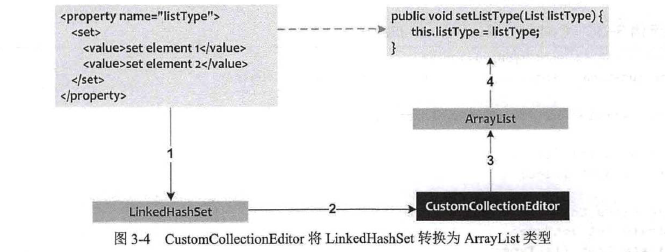
Bean类定义如下**listType是List集合**

**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



2. CustomMapEditor

CustomMapEditor属性编辑器将源Map类型转换为目标Map类型(如HashMap->TreeMap)。默认情况下, CustomMapEditor只对 SortedMap类型进行注册。

3. CustomDateEditor

CustomDateEditor是java.util.Date类型的bean属性和构造函数参数的属性编辑器。 CustomDateEditor支持自定义 java.text.DateFormat,用于将日期时间字符串格式化为java.util.Date类型对象,并将java.util.Date类型对象解析为日期/时间字符串。

**3.6 向spring容器注册属性编辑器**

Spring的 BeanWrapperlmpl类会向 Spring容器注册一些内置的属性编辑器。但是,默认情况下不会向 Spring容器注册 CustomDateEditor属性编辑器。要向 Spring容器注册属性编辑器,可以使用 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

要向 Spring容器注册属性编辑器,需要执行以下操作:

1. 创建一个实现 Spring的 PropertyEditorRegistrar接口的类,该类负责向 Spring容器注册属性编辑器
2. 将 PropertyEditorRegistrar实现配置为应用程序上下文XM文件中的 Spring Bean
3. 在应用程序上下文XML文件中配置 Spring中 CustomEditorConfigurer这个特定的bean,并为其提供步骤2配置的PropertyEditorRegistrar bean的引用

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

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

在bankDetails中使用< property-path>元素暴露的bean

<**bean id="bankDetails" class="sample.spring.chapter03.beans.BankDetails"**>  
 <**property name="dateOfInception" ref="dateType"** />  
 <**property name="branchAddresses" ref="branchAddresses"** />  
</**bean**>

3.9 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

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**>

3.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")**

3.10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"** />  
  
<**bean id="controller"  
 class="sample.spring.chapter03.bankapp.controller.FixedDepositControllerImpl"**>  
 <**property name="fixedDepositService" ref="service"** />  
</**bean**>

第4章依赖注入

4.2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。

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

将bean定义在property内部作为property的值，property不用再使用ref指定它的值

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

<property name="fixedDepositDao>

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

</property>

<bean>

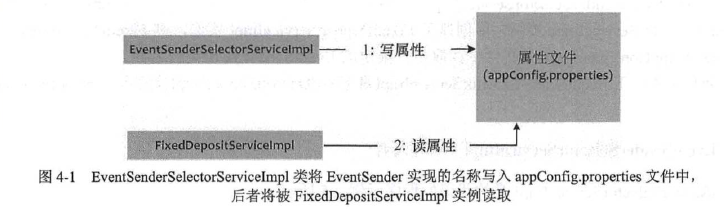
内部bean的概念使得可以在<property>和constructor-arg>元素中使用 Spring的util模式元素

<**bean**>  
 <**constructor-arg name="listType"**>  
 <**util:list list-class="java.util.ArrayList"**>  
 <**value**>A simple String value in list</**value**>  
 <**value**>Another simple String value in list</**value**>  
 </**util:list**>  
 </**constructor-arg**>  
</**bean**>

4.3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

假设有以下隐式依赖关系，程序启动时由EventSenderSelectorServicelmpl将实现类的名称写入属性文件。FixedDepositServicelmpl读取属性文件，根据名称创建实现类。



此时，需保证EventSenderSelectorServicelmpl bean在FixedDepositServicelmpl bean之前实例化

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文件中定义 EventSenderSelectorServicelmpl和 FixedDepositServiceImpl类的bean定义的顺序。使 EventSenderSelectorServicelmp1类的bean定义出现在 FixedDepositervicelmpl类的bean定义之前

2使用元素的 depends-on特性显式指定 service bean(对应于 Fixed DepositServicelmpl类)依赖于 eventSenderSelectorService bean(对应于 EventSenderSelectorServicelmpl类)。

<**bean id="service"  
 class="sample.spring.chapterO4.bankapp.service.FixedDepositServiceImpl"  
depends-on="eventSenderSelectorService"**>   
</**bean**>  
  
<**bean id="eventSenderSelectorService"  
 class="sample.spring.chapterO4.bankapp.service.EventSenderSelectorServiceImpl"**>   
</**bean**>

如果bean具有多个隐式依赖项,则可以将所有这些依赖项的id或name名称指定为 depends-on特性的值

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

singleton和 prototype范围的bean的依赖项