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。**

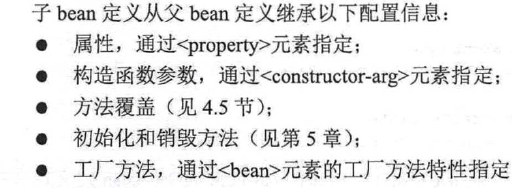
容器启动时

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**>

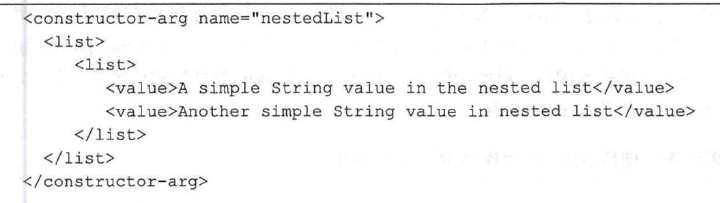
使用map，set，list子元素设置集合类型

<**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"** />

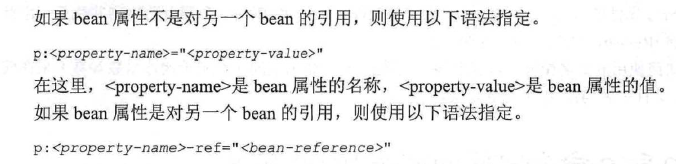
3.配置CustomEditorConfigurer类。CustomEditorConfigurer类实现了BeanFactoryPostProcessor接口

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

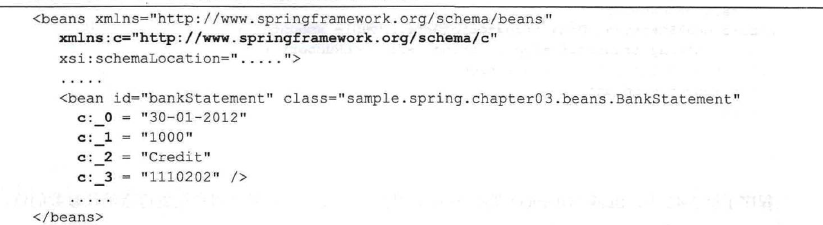
为了使应用程序上下文XML文件中的bean定义不那么冗长, Spring提供了p和c命名空间来分别指定bean属性和构造函数参数的值

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

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;

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

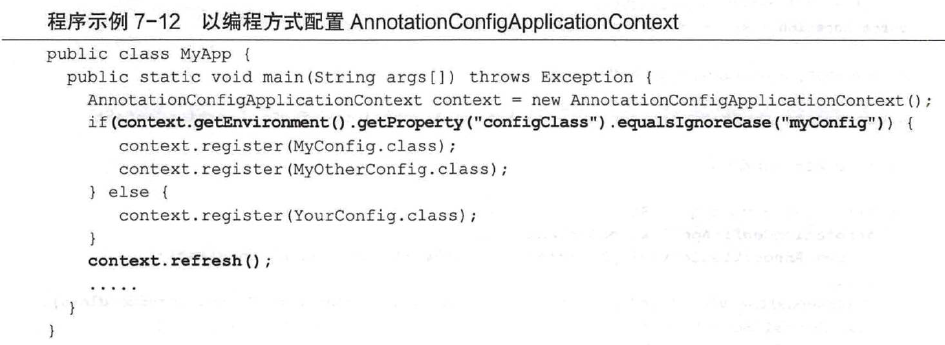
3通过在bean类中使用@Autowired 、@Inject 和@Resource 注释来实现自动装配依赖项。

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**);



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

@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 {}

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"** />

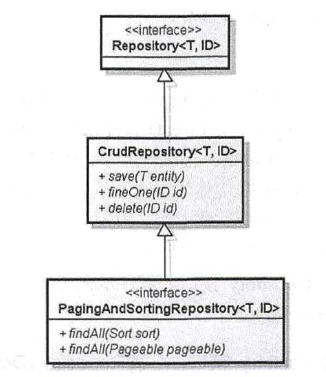
**transaction-manager特性指定事务管理器**

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>

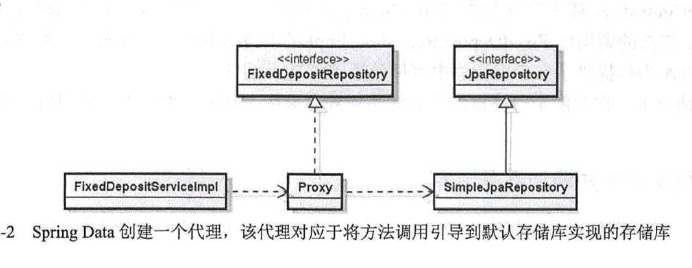
deleteBy<field-name>

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) {}

}

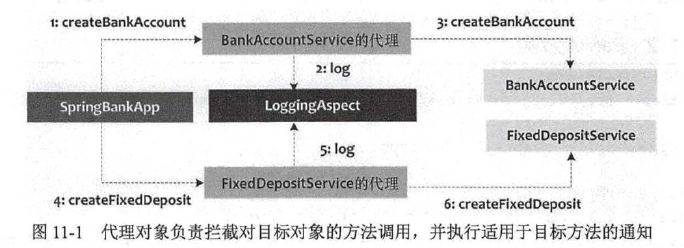
@Before标注通知。Value特性指定试用于此通知的通知连接点表达式

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

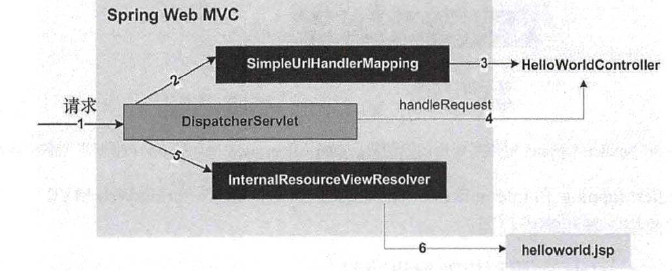
不管目标方法正常完成还是引发异常，都会调用后置通知

围绕通知@Around

可控制目标方法是否执行

使用xml配置aop

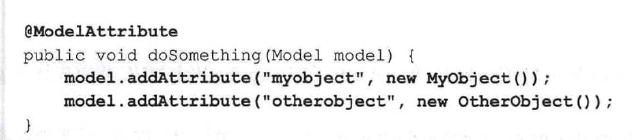
<**bean id="sampleAspect"  
 class="sample.spring.chapter11.bankapp.aspects.SampleAspect"** />  
   
<**aop:config proxy-target-class="false" expose-proxy="true"**>  
   
 <**aop:pointcut expression="execution(\* sample.spring..\*Service.\*(..))"  
 id="services"** />  
 <**aop:aspect id="sampleAspect" 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。