singleton和 prototype范围的bean的依赖项

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

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

<**bean id="customerRequestDetails"  
 class="sample.spring.chapter04.bankapp.domain.CustomerRequestDetails"  
 scope="prototype"** />

<**bean id="customerRequestDao" class="sample.spring.chapter04.bankapp.dao.CustomerRequestDaoImpl"** />

**customerRequestService是一个singleton bean，它依赖的customerRequestDetails是prototype bean。**

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:

使 singleton bean的类实现 Spring的 Application Context Aware接口;

使用 Spring的bean模式的<lookup-method:>元素

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。 Application ContextAware接口定义了单一的方法 setApplication Context,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。Application ContextAware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。

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 Application ContextAware接口的bean可以通过调用 Application Context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。

@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 Application ContextAware接口的缺点在于它将bean类与 Spring Framework相耦合

<lookup-method>元素

如果一个bean类定义了一个 bean lookup方法,其返回类型表示一个bean,那么<lookup-method>元素

将指示 Spring为此方法提供实现。 Spring提供的方法实现负责从 Spring容器获取bean实例并返回它。< lookup- method>元素的bean特性指定要查找的bean的名称,而名称特性指定 Spring实现方法的名称。注意,bean类定义的bean查找方法可以是抽象的或具体的方法。

**public abstract** CustomerRequestDetails getCustomerRequestDetails();

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

……

}

CustomerRequestServicelmpl类的变体,它定义了一个抽象bean查找方法 getCustomerRequestDetails,其返回类型为 CustomerRequestDetails.。submitRequest方法调用 getCustomerRequestDetails方法来获取一个新的CustomerRequestDetails实例。

应该注意的是,可以将 get CustomerRequestDetails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。一下是bean定义

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

自动装配依赖项

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。

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。

CustomerRegistrationServicelmpl类定义了 CustomerRegistration Details和 CustomerRegistration Dao对象作为其依赖项

<**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 Application Context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。例如,如果 bean A的构造函数接受X和Y类型的参数, Spring在 ApplicationContext中寻找X和Y类型的bean,并将它们作为参数注入 bean A的构造函数中。

如果Spring在 ApplicationContext中找不到任何类型与构造函数参数类型匹配的bean。构造函数参数不会被设置。 如果 Spring在Application Context中找到多个类型匹配的bean,则会抛出异常。

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。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。由于 Spring的默认行为是不使用bean的自动装配,因此将 autowire特性的值指定为 default或者no意味着不会对该bean使用。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和bean定义

5.2自定义bean的初始化和销毁逻辑

Spring容器负责创建一个bean实例并注入其依赖项。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 Isetter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),请将初始化方法的名称指定为元素的init-method特性的值。同样,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

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

**public class** FixedDepositDaoImpl **implements** FixedDepositDao {  
 **private** DatabaseConnection **connection**;  
  
 **public void** initializeDbConnection() {  
 *logger*.info(**"FixedDepositDaoImpl's initializeDbConnection method invoked"**);  
 **connection** = DatabaseConnection.*getInstance*();  
 }

}

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

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

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(由 destroy- method特性指定)的必要逻辑。独立应用程序需要由用户自己实现调用cleanup方法的逻辑

Spring的 ConfigurableApplication Context( 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 Configurable pplication Context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. Initializing Bean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。Bean在afterPropertiesSet方法中执行初始化工作,例如获取到数据库的连接、打开平面文件进行读取等。

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。

1. JSR250s@ 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@ 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"**/>

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

BeanPostProcessor用于在 Spring容器调用新创建的bean实例的初始化方法之前和/或之后,与其进行交互

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接口定义了以下方法

Object postProcess Beforelnitialization( Object bean, String beanName),此方法在bean实例的初始化方法被调用之前被调用

Object postProcessAfterlnitialization( Object bean, String beanName),此方法在bean实例的初始化方法被调用之后被调用。

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

可以像任何其他 Spring bean一样在应用程序上下文XML文件中配置一个 BeanPostProcessor实现。 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 postProcess Beforelnitialization和 postProcessAfterinitialization方法。

实现 BeanFactoryPostProcessor接口的bean是在实现 BeanPostProcessor接口的bean之前创建的。

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