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 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),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. 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。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 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**>

在第4章中,我们看到如果一个bean实现了 Spring的 ApplicationContextAware接口,它可以使用ApplicationContext的 getBean方法以编程方式获取bean实例.实现 Application ContextAware接口会使应用程序代码与 Spring耦合,因此,建议不要实现 Application ContextAware接口。我们将介绍一个 BeanPostProcessor实现,它为bean提供一个包装 ApplicationContext实例的对象,从而使得应用程序代码不直接依赖于 Spring的 ApplicationContextAware和 ApplicationContext接口。

首先定义接口

**public interface** DependencyResolver {  
 **void** resolveDependency(MyApplicationContext myApplicationContext);  
}

DependencyResolver定义了一个 resolveDependency方法,它接受一个 MyApplicationContext对象——一个 ApplicationContext对象的包装器。

实现 DependencyResolver接口的 FixedDepositServicelmpl类。

@Override  
**public void** resolveDependency(MyApplicationContext myApplicationContext) {  
 *logger*.info(**"Resolving dependencies of FixedDepositServiceImpl instance"**);  
 **fixedDepositDao** = myApplicationContext.getBean(FixedDepositDao.**class**);  
}

DependencyResolutionBeanPostProcessor实现了 Spring的 BeanPostProcessor和 Ordered接口。 myApplicationContext字段(类型为 MyApplicationContext)表示一个 DependencyResolutionBeanPostProcessor的依赖项。postProcess Beforelnitialization方法调用实现 DependencyResolver接口的bean的 resolveDependency方法, 将 MyApplicationContext对象作为参数传递。

**public class** DependencyResolutionBeanPostProcessor **implements** BeanPostProcessor, Ordered {

**private** MyApplicationContext **myApplicationContext**;

**public void** setMyApplicationContext(  
 MyApplicationContext myApplicationContext) {  
 **this**.**myApplicationContext** = myApplicationContext;  
}

@Override  
**public** Object postProcessBeforeInitialization(Object bean, String beanName)  
 **throws** BeansException {  
 *logger*.info(**"DependencyResolutionBeanPostProcessor's postProcessBeforeInitialization method invoked for bean "** + beanName + **" of type "** + bean.getClass());  
 **if** (bean **instanceof** DependencyResolver) {  
 ((DependencyResolver) bean).resolveDependency(**myApplicationContext**);  
 }  
 **return** bean;  
}

}

作为 Spring的 ApplicationContext对象的包装器的 My ApplicationContext类。

**public class** MyApplicationContext **implements** ApplicationContextAware {  
 **private** ApplicationContext **applicationContext**;  
   
 @Override  
 **public void** setApplicationContext(ApplicationContext applicationContext)  
 **throws** BeansException {  
 **this**.**applicationContext** = applicationContext;  
 }  
   
 **public** <T> T getBean(Class<T> klass) {  
 **return applicationContext**.getBean(klass);  
 }  
}

<**bean  
 class="sample.spring.chapter05.bankapp.postprocessor.DependencyResolutionBeanPostProcessor"**>  
 <**property name="myApplicationContext" ref="myApplicationContext"** />  
 <**property name="order" value="0"** />  
</**bean**>

<**bean id="myApplicationContext"  
 class="sample.spring.chapter05.bankapp.common.MyApplicationContext"** />

Spring容器不会将一个 BcanPostProcessor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。

一个实现了 Spring的 FactoryBean接口的bean代表一个用于创建bean实例的工厂。当 Spring容器创建一个 FactoryBean实例时,将调用 BeanPostProcessor的postProcessBeforeInitialization和 postProccssAfterlnitialization方法。而且,对于由 Factory Bean创建的bean实例,只调用 postProcessAfterlnitialization方法。

4. RequiredAnnotationBeanPostProcessor

如果在bean类字段的setter方法上使用 Sprin的@Required注释, Spring的 RequiredAnnotationBeanPostProcessor(一个 BeanPostProcessor实现)将检查bean属性是否已在应用程序上下文XML文件中配置

public class FixedDepositServiceImpl implements FixedDepositService (

private FixedDepositDao fixedDepositDao;

@Required

public void setFixedDepositDao(FixedDepositDao fixedDepositDao) {

this.fixedDepositDao = fixedDepositDao;

}

}

如果在应用程序上下文XML文件中定义了 RequiredAnnotationBeanPostProcessor,则RequiredAnnotationBeanPostProcessor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

5. DestructionAwareBeanPostProcessor

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。

5.4使用 Bean FactoryPostProcessor修改bean定义

Spring的 BeanFactoryPostProcessor接口由那些希望对bean定义进行修改的类实现。BeanFactory PostProcessor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。BeanFactory PostProcessor会在应用程序上下文XML文件中定义的任何其他bean被创建之前就被创建,这使得 BeanFactory PostProcessor有机会对其他bean的bean定义进行修改。可以在应用程序上下文XML文件中配置 BeanFactory PostProcessor实现,就像任何其他 Spring bean一样。

第6章使用 Spring进行注释驱动开发

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

6.2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。

由于 Fixed DepositServicelmpl类使用了@Service注释,因此 FixedDepositServicelmpl类表示一个 Spring bean。@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=**"fixedDepositService"**)  
**public class** FixedDepositServiceImpl **implements** FixedDepositService {…}

可以指定定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。

如果启用 Spring的类路径扫描功能,则使用@ Component、@ Controller，@ Service或@ Repository注释的bean类将自动注册到 Spring容器。可以使用 Spring的 context模式的<component-scan>元素来启用 Spring的类路径扫描功能。

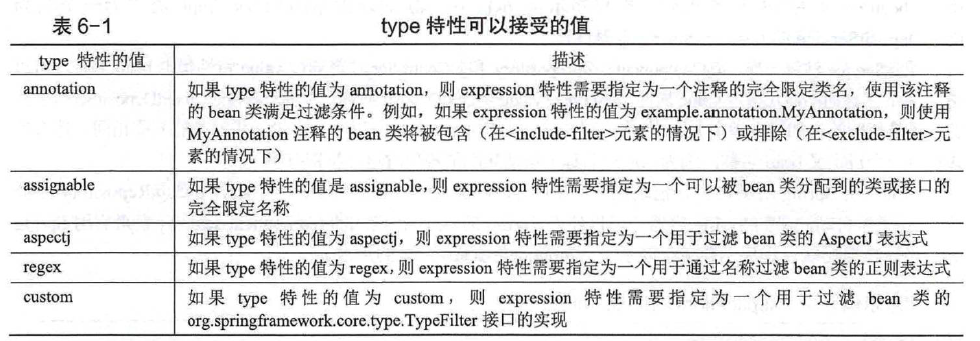
<**beans xmlns="http://www.springframework.org/schema/beans"  
 xmlns:context=**[**http://www.springframework.org/schema/context**](http://www.springframework.org/schema/context)

**……**>  
 <**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。

动注册。< 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。

6.3@Autowired通过类型自动装配依赖项

@ 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)负责自动装配 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

6.4@Qualifier按名称自动装配依赖项

可以使用 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。

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

在程序示例6-10中, myFixedDepositDao字段使用了@ Autowired和@Qualifier两个注释。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@ Qualifier i注释指定的bean名称来定位自动装配候选列表中唯一的bean。

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){。。。。}

}

使用@Qualifier注释bean类

前面的@Qualifier注释字段、构造函数参数和方法参数是用于自动装配依赖项。@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

使用@Qualifier注解TxDaoImpl类，并起别名**myTx。**

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

}

使用刚才注解的bean注入TxServiceImpl类的TxDao字段

@Service(**"txService"**)  
@Qualifier(**"service"**)  
**public class** TxServiceImpl **implements** TxService {  
 @Autowired  
 @Qualifier(**"myTx"**)  
 **private** TxDao **txDao**;

}

创建自定义限定符注释

6.5JSR330的@ Inject和@ Named注释

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。

@ Autowired和@ Inject释具有相同的语义,它们用于按类型自动装配依赖项

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。

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

<dependency>

<groupId>j avax.inject</groupId>

<artifactId>javax.inject</artifactId>

<version>l</version>

</dependency>