目录

[写在前面的话： 1](#_Toc485925047)

[一、Spring框架概述 1](#_Toc485925048)

[1.依赖注入（DI）和控制反转（IoC） 2](#_Toc485925049)

[2.面向方面编程（AOP） 4](#_Toc485925050)

[3.Bean 4](#_Toc485925051)

[4.Spring容器 6](#_Toc485925052)

[二、Spring容器装配/配置 6](#_Toc485925053)

[1. 基于XML配置的自动装配 6](#_Toc485925054)

[2. 基于注解的配置 7](#_Toc485925055)

[3. 基于Java的配置 7](#_Toc485925056)

[三、Spring中事件处理 7](#_Toc485925057)

[1. 监听上下文 8](#_Toc485925058)

[2. 自定义事件 8](#_Toc485925059)

[四、Spring AOP 8](#_Toc485925060)

[1. AOP术语 8](#_Toc485925061)

[2. Spring中基于AOP的XML架构 9](#_Toc485925062)

[3. Spring中基于AOP的@AspectJ 10](#_Toc485925063)

写在前面的话：

想要通过本笔记学习Spring或者是通过一般的阅读材料来学习都要面临一个问题，不知道从哪个地方切入作为入门点。尤其是对于只有Java语言经验，没有任何框架使用经验的人来说更是如此。

具体方法是：可以先读（并反复读）每个章节和小节的前几句概述（显然教程不一定按照合理的顺序介绍，也没有人知道什么是合理的顺序），对于整体框架有个简单的印象。这样的印象也许是不确切的，但是这么做便于入门和理解并且符合人们认识事物的习惯。在这期间如果遇到一些不懂的技术和知识，可以适当的扩展学习（最好按照需求学习，不需要的暂时不学）。对整体有印象后，可以从头到尾通读教程或笔记。

给人的更多启示：很多教程的概述真的就是概括，使用极简的词汇和大量的术语，所以太抽象。对于需要看概述的新手来说很难理解，对于不需要看概述的老手来说完全没意义。所以对于学习的人必须找到（上网简单搜搜是最好的方法）最能易于理解的概述；对教授知识的人来说，必须创造（举例子或比对学生已经掌握的知识，哪怕是不确切的也行）最能易于理解的概述。

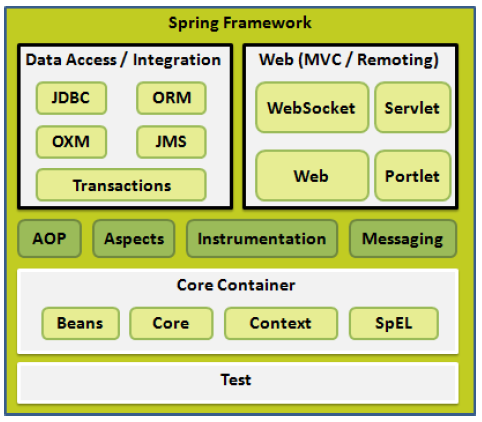
一、Spring框架概述

•Spring框架的核心功能有两个：

①Spring容器作为超级大工厂，负责创建、管理所有的Java对象，这些Java对象被称为Bean。

②Spring容器管理容器中Bean之间的依赖关系，Spring使用一种被称为"依赖注入"的方式来管理Bean之间的依赖关系。

•Spring框架是模块化的，可以根据需要来使用或不使用



1.依赖注入（DI）和控制反转（IoC）

•控制反转（Inverse of Control）和依赖注入（Dependency Injection）：二者含义完全相同，说白了就是一种构造对象的过程，构造过程中需要用到其他类的对象。为什么特别？是因为构造对象的过程是由Spring控制的，而不是一般性的构造。构造的什么对象？Bean。

•特点：使用依赖注入，不仅可以为Bean注入普通的属性值，还可以注入其他Bean的引用。依赖注入是一种优秀的解耦方式，其可以让Bean以配置文件组织在一起，而不是以硬编码的方式耦合在一起。

•原理：当某个Java对象（调用者）需要调用另一个Java对象（被依赖对象）的方法时，在传统模式下通常有两种做法：

①原始做法: 调用者主动创建被依赖对象，然后再调用被依赖对象的方法。

②简单工厂模式: 调用者先找到被依赖对象的工厂，然后主动通过工厂去获取被依赖对象，最后再调用被依赖对象的方法。

这必然会导致调用者与被依赖对象实现类的硬编码耦合，非常不利于项目升级的维护。使用Spring框架之后，调用者无需主动获取被依赖对象，调用者只要被动接受Spring容器为调用者的成员变量赋值即可，由此可见，使用Spring后，调用者获取被依赖对象的方式由原来的主动获取，变成了被动接受；另外从Spring容器的角度来看，Spring容器负责将被依赖对象赋值给调用者的成员变量——相当于为调用者注入它依赖的实例。

1.1 基于构造函数的依赖注入

•同前理，说白了就是调用Java类的构造函数创建Bean

public class Foo {

public Foo(int year, String name) { // ... }}

•方式1：

<beans>

<bean id="exampleBean" class="examples.ExampleBean">

<constructor-arg type="int" value="2001"/>

<constructor-arg type="java.lang.String" value="Zara"/>

</bean>

</beans>

•方式2：

<beans>

<bean id="exampleBean" class="examples.ExampleBean">

<constructor-arg index="0" value="2001"/>

<constructor-arg index="1" value="Zara"/>

</bean>

</beans>

•上述两种方式都是采用值传递参数，适合基本数据类型和String，如果要采用引用传递（即传递其他的bean对象引用），用ref代替value即可，属性值取所需bean id即可。其实采用引用传递其他对象才算真正用到了依赖注入技术。

•Spring框架使用的xml文件的具体格式存在很大灵活性，上述两种方式不是唯二的，有时候可以省略type或index，只用value或ref属性。也可以将value或ref作为constructor-arg的子标签，具体用法应见到即通。

1.2 基于设值函数的依赖注入

•如果一个对象中不包含其他对象成员，或只有基本类型成员，那么不需要定义设值函数，即也谈不上用到依赖注入技术。

•设置函数一般称为setter，其实就是一种可自定义的成员函数，专门用来设定对象的数据成员。这些成员函数名一般为setDataname(…)。有点类似于C#中的属性。

•例：

public class TextEditor {

private SpellChecker spellChecker;

// a setter method to inject the dependency.

public void setSpellChecker(SpellChecker spellChecker) {

this.spellChecker = spellChecker;}

// a getter method to return spellChecker

public SpellChecker getSpellChecker() {

return spellChecker;}

}

<bean id="textEditor" class="com.tutorialspoint.TextEditor">

<property name="spellChecker" ref="spellChecker"/>

</bean>

<bean id="spellChecker" class="com.tutorialspoint.SpellChecker"/>

•在xml文件中，除了标签名称不一样，道理上和基于构造函数的方法类似，都是传递其他bean的id

•基于设值函数的依赖注入，xml文件和一般的情况下类似，只是传递的是ref。而Java代码中增加了新的成员函数。

1.3 使用内部bean依赖注入

•内部bean即bean中定义的bean，虽然类似内部类，但是没有直接关系。因此并非带有内部类的类才能创建内部bean。

•1.2中例子可以改为，可见xml中的bean和Java中的类其逻辑关系是有联系却也相互独立的。

<bean id="textEditor" class="com.tutorialspoint.TextEditor">

<property name="spellChecker">

<bean id="spellChecker" class="com.tutorialspoint.SpellChecker"/>

</property>

</bean>

1.4 使用p-namespace

•这种方法只是一个简便写法，下面的第一个例子可以改写为第二个例子：

•<bean id="john-classic" class="com.example.Person">

<property name="name" value="John Doe"/>

<property name="spouse" ref="jane"/>

</bean>

<bean name="jane" class="com.example.Person">

<property name="name" value="John Doe"/>

</bean>

•<bean id="john-classic" class="com.example.Person" p:name="John Doe" p:spouse-ref="jane"/>

<bean name="jane" class="com.example.Person" p:name="John Doe"/>

1.5 注入集合和注入空字符

见p70

2.面向方面编程（AOP）

•原理：它是可以通过预编译方式和运行期动态代理实现在不修改源代码的情况下给程序动态统一添加功能的一种技术。即使用代理类包装原来的类，来为各个不同的类增加统一的通用功能，比如日志记录和输入验证功能。可以避免在每个独立类中重复地实现类似的代码。

•Spring AOP模块使用一个拦截器来拦截调用命令，以在执行方法的前后统一增添功能。

## 3.Bean

3.1 Bean定义

•简单说来，Bean是对象。Spring框架创建的程序是由各种Bean组成的，每个Bean都是Spring根据组成程序的Java类创建的对象，Spring框架管理程序是通过管理Bean来间接实现的。一般由Java类直接实例化出来的对象称为POJO（Plain Old Java Object），而Bean是由Spring从Java类实例化出来的“加强版”对象。

•进一步描述Spring和Bean的工作方式：程序所需的各种类仍然由Java写出，指定程序操作的主类或其主函数中将调用Spring框架来完成这些Java类的实例化，实例化后的对象都是Bean。而之后对于这些对象的操作使用仍然和POJO一样。

•Bean是由配置文件提供的配置元数据创建的，Bean包含以下的属性和方法

|  |  |
| --- | --- |
| class | 这个属性是强制性的，并且指定用来创建bean的Java类。 |
| name | 这个属性指定唯一的bean标识符。在基于XML的配置元数据中，你可以使用ID 和/或name属性来指定bean标识符。 |
| scope | 这个属性指定由特定的 bean 定义创建的对象的作用域，它将会在bean作用域的章节中进行讨论。 |
| constructor-arg | 它是用来注入依赖关系的，并会在接下来的章节中进行讨论。 |
| properties | 它是用来注入依赖关系的，并会在接下来的章节中进行讨论。 |
| autowiring mode | 它是用来注入依赖关系的，并会在接下来的章节中进行讨论。 |
| lazy-initialization mode | 延迟初始化的bean告诉IoC容器在它第一次被请求时，而不是在启动时去创建一个 bean 实例。 |
| initialization方法 | 在bean的所有必需的属性被容器设置之后，调用回调方法。它将会在bean的生命周期章节中进行讨论。 |
| destruction方法 | 当包含该bean的容器被销毁时，使用回调方法。它将会在bean的生命周期章节中进行讨论。 |

**3.2 Bean的作用域**

•Bean带有作用域属性，由起始标签中scope属性定义，取值可有以下几种

|  |  |
| --- | --- |
| singleton | 该作用域将bean的定义的限制在每一个Spring IoC容器中的一个单一实例(默认)。即每次需要时返回同一个bean实例。 |
| prototype | 该作用域将单一bean的定义限制在任意数量的对象实例。即每次需要时都产生一个新的bean实例。 |
| request | 该作用域将bean的定义限制为HTTP请求。只在web-aware Spring ApplicationContext 的上下文中有效。 |
| session | 该作用域将 bean 的定义限制为 HTTP 会话。 只在web-aware Spring ApplicationContext的上下文中有效。 |
| global-session | 该作用域将 bean 的定义限制为全局 HTTP 会话。只在 web-aware Spring ApplicationContext 的上下文中有效。 |

•用法举例：

<bean id="..." class="..." scope="singleton">

<!-- collaborators and configuration for this bean go here -->

</bean>

**3.3 Bean的生命周期**

•Bean也是对象，因此可以设定初始化和销毁回调方法，使用被用来创建bean的Java类中的方法来做到这一点。为单一的bean指定初始化和销毁方法如本节末尾的例子。可以在xml文件beans标签设定默认初始化和销毁回调函数，之后凡省略该项属性的bean均采用该方法：

<beans xmlns = “...” xmls:xsi= “...”xsi:schemaLocation= “...”

default-init-method = “int”

default-destroy-method = “destroy”>

•特别注意：初始化回调和销毁回调方法不是bean的构造方法和析构方法，应该是构造和析构行为触发的事件回调方法。而二者只是用来初始化和处理后续工作的方法。

•一般例子：

<!-- A bean definition with lazy init set on -->

<bean id="..." class="..." lazy-init="true">

<!-- collaborators and configuration for this bean go here -->

</bean>

<!-- A bean definition with initialization method -->

<bean id="..." class="..." init-method="..." destroy-method= “...”>

//init-method的值只写方法名（无括号）即可，但是必须是class指定的类中的void型无参方法

//destroy-method使用类似init-method

<!-- collaborators and configuration for this bean go here -->

</bean>

3.4 Bean后置处理器

•bean后置处理器是一种功能，可以让用户自定义实例化逻辑和依赖解析逻辑。BeanPostProcessor便是支持这种功能的接口。如果要使用，应当创建一个Java类实现这个接口，然后IoC容器将这个类转化为一个特殊的bean。

•BeanPostProcessor接口有以下几个方法和属性

①Object postProcessBeforeInitialization(Object bean, String beanName) throws BeansException

②Object postProcessAfterInitialization(Object bean, String beanName)

//每一个bean的初始化前后都要调用前两个方法，而用户可以根据bean的名称来进行针对性或普遍性操作

③order //用来控制多个BeanPostProcessor的执行顺序

•对于BeanPostProcessor这个特殊的bean在xml中的定义

<bean class = “实现BeanPostProcessor的类”/>

3.5 Bean定义继承

•bean继承和Java继承没有直接关系，但是道理是一样的。而且一般能够有继承关系的bean都是由有继承关系的Java类创造的，但是不必须是这样。

•定义继承的方法是：xml中，在子bean的起始标签定义parent属性，赋给其其他bean的id即可指定为父bean。子bean继承父bean标签元素中的所有内容（如property），当然在子bean中定义重复名称的标签即可重写父bean继承下来的内容。

3.6 Bean定义模板

•bean也有类似于抽象类一样抽象bean或bean模板，只能用来继承，不能用来实例化，定义方法如下：

<bean id = “...” abstract = “true”> //抽象bean不能定义class属性

...

</bean>

## 4.Spring容器

•Spring有两种不同类型的容器，BeanFactory和ApplicationContext。这两个名称即两个容器的接口名称。容器可以理解为装载Bean的容器，因此也可以理解为Spring的程序对象。当遇到请求时，也可以分配Bean。

•ApplicationContext容器比BeanFactory高级，同时包含后者的功能，但是后者仍然可以用于轻量级的应用中，比如移动设备和基于applet（微型app只包含一个功能和若干个函数）的应用程序。

•将容器实例化的过程即根据配置文件和Java POJO类来创建Bean并整合为容器（程序）的过程。配置元数据信息由用户提供，一般有三种形式，XML，Java注释和Java代码。

•BeanFactory容器使用XML配置的实现类是XmlBeanFactory

例：

XmlBeanFactory factory = new XmlBeanFactory(new ClassPathResource(“Beans.xml”));

•ApplicationContext接口的实现种类

①FileSystemXmlApplicationContext：该容器从XML文件中加载已被定义的bean。在这里，你需要提供给构造器 XML 文件的完整路径

②ClassPathXmlApplicationContext：该容器从XML文件中加载已被定义的bean。在这里，你不需要提供XML文件的完整路径，只需正确配置CLASSPATH环境变量即可，因为，容器会从 CLASSPATH 中搜索bean配置文件。

③WebXmlApplicationContext：该容器会在一个web应用程序的范围内加载在XML文件中已被定义的bean。

④AbstractApplicationContext：该容器定义了一个registerShutdownHook()非静态方法，用来正常关闭容器对象并调用bean销毁回调方法。

二、Spring容器装配/配置

## 1. 基于XML配置的自动装配

•自动装配完全可以被手动装配替代，当然后者显然比前者精确可靠。自动装配其实就是自动根据名称或者类型来对有关系的bean进行依赖注入，在<property>或<constructor-arg>标签即可省略手动写入

<bean id = “…” class = “…” autowire = “byName”>

<bean id = “…” class = “…” autowire = “byType”> //前两个是对基于设值函数DI的

<bean id = “ …” class = “…” autowire = “constructor”> //这个是对基于构造函数DI的

•自动装配或者自动布线只能针对一部分bean属性（property），即那些需要传递其他bean引用的属性，对于基本数据类型不能用

## 2. 基于注解的配置

•注解配置和xml配置可以同时使用，实际上xml配置文件本身是spring的配置文件，不是只是用来配置bean的。另外注解配置在xml配置之前进行，后者可以重写前者

例：在<beans>标签第一个元素中添加

<context:annotation-config/>

①@Requried

•这个是一个校验注释，添加在在java类文件中，各种setter方法之前。而此后如果在xml配置中不提供对应的property标签内容，会抛出BeanInitializationException异常

②@Autowired

•此注释可用于setter方法也可用于java类中的属性，被注释的属性将采用byType自动连接而不需xml中显式配置，第二种注释方式也不需要定义setter。实际上两种注释方式都是同一结果，相当于隐藏或除去了setter。

•此注释也可用于构造函数，道理类似

•同前所述，autowired只对对象注入有效，基本数据类型也可以使用@Autowired，但是并不起任何自动装配效果，产生的效果如同@Required，因为基本数据必须依靠xml显式注入。但是使用@Autowired(required=false)可以关闭这种校验行为。

③@Qualifier

•@Qualifier一般和@Autowired一起使用，因为后者相当于byType自动装配，因此面对多个同类型bean时，使用Qualifier指定需要注入的bean

•使用时在@Autowired之后一行，在需要注释的属性前再添加一行@Qualifier(…)，括号中指定bean的id即可，需要加引号。

④JSR-250注释

(1)@PostConstruct和@PreDestroy

•功能和指定init-method与destroy-method相同，只是通过在java类中对应方法之前加注释的方式完成

(2)@Resource

•用于java属性或setter方法，功能类似于byName自动装配。使用@Resource(name = “…”)来指定bean id。如果不指定name，在属性上时使用属性名，在setter上时使用bean属性名

## 3. 基于Java的配置

p103

三、Spring中事件处理

•Spring中的事件处理靠bean，这些bean的源java类被定义成实现ApplicationListener接口。这个接口是一个泛型，可用多种标准事件类型实例化。这些标准事件类型都是ApplicationEvent类的子类

•标准事件：

|  |  |
| --- | --- |
| ContextRefreshedEvent | ApplicationContext被初始化或刷新时发布 |
| ContextStartedEvent | ApplicationContext启动时发布 |
| ContextStoppedEvent | ApplicationContext停止时发布 |
| ContextClosedEvent | ApplicationContext关闭时发布，已关闭的不能刷新或重启 |
| RequestHandledEvent | web-specific事件，所有bean HTTP请求已经被服务 |

注：Spring的事件处理是单线程的

## 1. 监听上下文

•ApplicationListener<T>接口只有一个方法void onApplicationEvent(T event)，用一个java类实现这个接口并实例化为一个bean即可针对某种事件建立一个响应机制

## 2. 自定义事件

•如上文所述，事件是一种类的对象，继承自ApplicationEvent

public class CustomEvent extends ApplicationEvent{

public CustomEvent(Object source){

super(source);}

…} //自定义事件必须继承默认的父类构造函数，构造函数参数可以是事件发布器

•有了自定义事件还必须有相关的事件发布器，其需实现ApplicationEventPublisherAware接口。自定义事件发布器类中应包含已经定义的ApplicationEventPublisher，并用其发布自定义事件。ApplicationEventPublisherAware接口的作用是让程序知道这个类是会发布事件的类。

例：

import org.springframework.context.ApplicationEventPublisher;

import org.springframework.context.ApplicationEventPublisherAware;

public class CustomEventPublisher implements ApplicationEventPublisherAware {

private ApplicationEventPublisher publisher;

public void setApplicationEventPublisher(ApplicationEventPublisher publisher){

this.publisher = publisher;}

public void publish() {

CustomEvent ce = new CustomEvent(this);

publisher.publishEvent(ce);}

}

四、Spring AOP

•关注点：AOP把程序逻辑分解成为不同的部分，每个部分称为一个关注点

•横切关注点：跨一个应用的多个点的功能被称为此，往往这些横切关注点在概念上独立于应用程序的逻辑，比如log，审计，security和buffer等

•Spring AOP模块提供拦截器来拦截一个应用程序，例如，当执行一个方法时，你可以在方法执行之前或之后添加额外的功能。

## 1. AOP术语

•是AOP，不一定针对Spring

|  |  |
| --- | --- |
| Aspect | 一个模块具有一组提供横切需求的APIs。例如，一个日志模块为了记录日志将被AOP方面调用。应用程序可以拥有任意数量的方面，这取决于需求。 |
| Join point | 在你的应用程序中它代表一个点，你可以在插件AOP方面。你也能说，它是在实际的应用程序中，其中一个操作将使用Spring AOP框架。 |
| Advice | 这是实际行动之前或之后执行的方法。这是在程序执行期间通过Spring AOP框架实际被调用的代码。也就是AOP拦截器的具体作为。 |
| Pointcut | 这是一个或多个切入点，advice应该被执行。你可以使用表达式或模式指定切入点。 |
| Introduction | 引用允许你添加新方法或属性到现有的类中。 |
| Target object | 被一个或者多个方面所通知的对象，这个对象永远是一个被代理对象。也称为被通知对象。 |
| Weaving | Weaving 把方面连接到其它的应用程序类型或者对象上，并创建一个被通知的对象。这些可以在编译时，类加载时和运行时完成。 |

•5种类型Advice，有的地方译作“通知”

|  |  |
| --- | --- |
| before advice | 一个方法执行前执行advice |
| after advice | 一个方法执行后，无论结果，执行advice |
| after-returning advice | 一个方法执行并成功完成后，执行advice |
| after-throwing advice | 一个方法执行并抛出异常时，执行advice |
| around advice | 在方法调用的之前和之后，执行advice |

## 2. Spring中基于AOP的XML架构

•aop功能需要用到aop命名空间，需要在xml beans标签导入

<beans … xmlns:aop = “http://www.springframework.org/schema/aop" …>

(1)声明一个方面aspect

例：道理类似一个被其他bean注入的bean，见(3)

•准确的说一个拦截器对应一个Java类，也必须实例化为bean，之后这个bean注入到一个方面中发挥作用。这个拦截器类中显然需要提供所有的advice方法以供选择。

(2)声明切入点pointcut

•切入点即被拦截器代理的方法，显然一个方面可以设定多种切入点元素，而每个切入点也不一定只包含一个方法

例：见(3)

(3)声明建议advice

例：共有五种类型的advice

•注意：after和after-returning/after-throwing建议同时生效时，前者会先执行，因为前者不论结果为何。

<aop:config>

<aop:aspect id="myAspect" ref="aBean">

<aop:pointcut id="businessService" expression="execution(\*com.xyz.myapp.\*.\*(..)) "/>

//这是指定了该包内的所有类的所有方法，如果要指明具体方法，写入具体名称即可，但括号内参数不必提供，仍保留(..)形式即可

<!-- a before advice definition -->

<aop:before pointcut-ref="businessService" method="doRequiredTask"/>

<!-- an after advice definition -->

<aop:after pointcut-ref="businessService" method="doRequiredTask"/>

<!-- an after-returning advice definition -->

<!--The doRequiredTask method must have parameter named retVal -->

<aop:after-returning pointcut-ref="businessService" returning="retVal"

method="doRequiredTask"/>

<!-- an after-throwing advice definition -->

<!--The doRequiredTask method must have parameter named ex -->

<aop:after-throwing pointcut-ref="businessService" throwing="ex"

method="doRequiredTask"/>

<!-- an around advice definition -->

<aop:around pointcut-ref="businessService" method="doRequiredTask"/>

...

</aop:aspect>

</aop:config>

<bean id="aBean" class="...">

...

</bean>

## 3. Spring中基于AOP的@AspectJ

•这其实就是通过注释来声明AOP功能的方式，作用结果同XML架构，要支持@AspectJ功能，仍需在xml配置中添加如下语句，类似自动装配

<aop:aspectj-autoproxy/>

(1)声明Aspect

•在拦截器Java文件中，类声明之上添加@Aspect，注意拦截器仍需实例化为bean才能生效

(2)声明pointcut

例：需要在拦截器中添加空方法签名，方法名作为切入点名

import org.aspectj.lang.annotation.Pointcut;

@Pointcut("execution(\* com.xyz.myapp.service.\*.\*(..))") // expression

private void businessService() {} // signature

(3)声明advice

例：

@Before("businessService()")

public void doBeforeTask(){...}

@AfterReturning(pointcut = "businessService()", returning="retVal")

public void doAfterReturnningTask(Object retVal){

// you can intercept retVal here....}

@AfterThrowing(pointcut = "businessService()", throwing="ex")

public void doAfterThrowingTask(Exception ex){

// you can intercept thrown exception here....}

@Around("businessService()")

public void doAroundTask(){…}

五、Spring JDBC框架概述

## 1.配置数据源

例：

<bean id="dataSource"

class="org.springframework.jdbc.datasource.DriverManagerDataSource">

<property name="driverClassName" value="com.mysql.jdbc.Driver"/>

<property name="url" value="jdbc:mysql://localhost:3306/TEST"/>

<property name="username" value="root"/>

<property name="password" value="password"/>

</bean>

## 2.JdbcTemplate类

•是Spring框架下的一个类，有点类似Java自带JDBC中的statement诸类的功能。可以执行具体的SQL指令。

•具体的SQL语句使用JdbcTemplate类方法的实现见《Spring教程》极客学院p137/138

## 3.数据访问对象（DAO）

•一般是一个自定义的接口，使用JdbcTemplate类直接处理SQL语句仍然十分麻烦，需要使用SQL指令字符串为参数。可以将JdbcTemplate的各种处理SQL语句的机制封装成为一个易调用的可以处理数据库的类或对象（当然还是根据实际需要选择性封装），这就是DAO。