**Spring课件**

# Spring简介

spring （由Rod Johnson创建的一个开源框架）

Spring是一个开放源代码的设计层面框架，他解决的是业务逻辑层和其他各层的松耦合问题，因此它将面向接口的编程思想贯穿整个系统应用。Spring是于2003 年兴起的一个**轻量级**的Java 开发框架，由Rod Johnson创建。简单来说，Spring是一个分层的JavaSE/Eef ull-stack(**一站式**) 轻量级开源框架。

**一站式**

传统EE开发分成三层结构:

\* WEB层:Spring MVC.

\* 业务层:Bean管理:(IOC)

\* 持久层:Spring的JDBC模板.ORM模板用于整合其他的持久层框架.

## Spring的版本

Spring 3.X 和 Spring4.X 和Spring5.X

## 为什么学习Spring

①:方便解耦，简化开发

Spring就是一个大工厂，可以将所有对象创建和依赖关系维护，交给Spring管理

②:AOP编程的支持

Spring提供面向切面编程，可以方便的实现对程序进行权限拦截、运行监控等功能

③:声明式事务的支持

只需要通过配置就可以完成对事务的管理，而无需手动编程

④:方便程序的测试

Spring对Junit4支持，可以通过注解方便的测试Spring程序

⑤:方便集成各种优秀框架

Spring不排斥各种优秀的开源框架，其内部提供了对各种优秀框架（如：Struts、Hibernate、MyBatis、Quartz等）的直接支持

⑥:降低JavaEE API的使用难度

Spring 对JavaEE开发中非常难用的一些API（JDBC、JavaMail、远程调用等），都提供了封装，使这些API应用难度大大降低

## spring的特征

轻量

控制反转

面向切面

容器

框架

MVC

所有Spring的这些特征使你能够编写更干净、更可管理、并且更易于测试的代码。它们也为Spring中的各种模块提供了基础支持。

## Spring的好处

1.轻量级,非[低]侵入式设计，代码污染极低

2.方便解耦合,简化开发.

3.声明事务的支持

4.方便程序测试

5.方便继承各种优秀框架

6.降低JavaEE API的使用难度

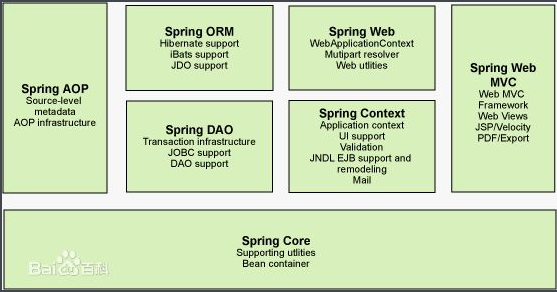
2.独立于各种[应用服务器](https://baike.baidu.com/item/%E5%BA%94%E7%94%A8%E6%9C%8D%E5%8A%A1%E5%99%A8)，基于Spring框架的应用，可以真正实现Write Once,Run Anywhere的承诺

3.Spring的AOP支持允许将一些通用任务如安全、事务、日志等进行集中式管理，从而提供了更好的复用

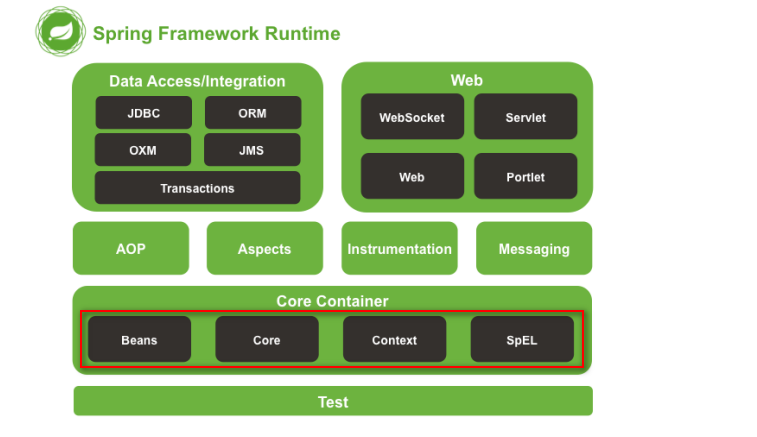
4.Spring的ORM和DAO提供了与第三方持久层框架的良好整合，并简化了底层的数据库访问

5.Spring并不强制应用完全依赖于Spring，开发者可自由选用Spring框架的部分或全部

## Spring模块图示



## Spring体系结构图



# Spring入门-IoC

## IoC是什么?

Inverse of Control，控制反转，它是将对象的创建权反转给Spring:通过容器来控制对象的创建及维护包括对象中成员变量的创建及维护。反转就是将对象的控制权由开发者转移给容器处理，目的是获得更好的扩展性和可维护性。这就是控制反转。

使用IOC可以解决的程序耦合性高的问题。

## 入门程序

**一:下载Spring的jar包**

**二:创建Web项目,导入所需要的jar包**

通过体系结构图,可以看到,Core Container就是IoC容器,spring所有的功能都依赖与它,而它所需要的jar包通过体系图可以看到分别是Beans,Core,Context,SpEL,所以要把这四个jar包导入到lib文件夹下.

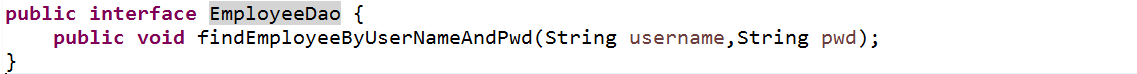
在下载的Spring包中,仅仅只有Spring自己的,它自己所依赖的jar包并不包含在里边,启动的时候就需要日志包org.apache.commons.logging-1.1.1.jar

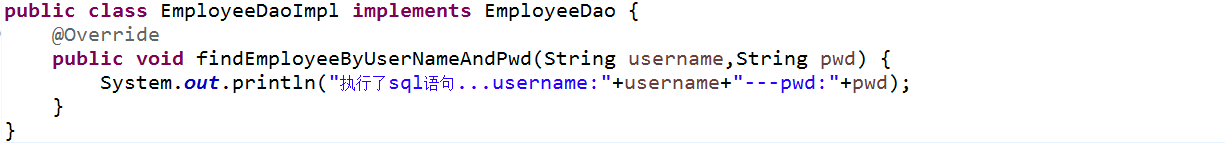
com.springsource.org.apache.log4j-1.2.15.jar

所以需要把该包也添加到lib目录中



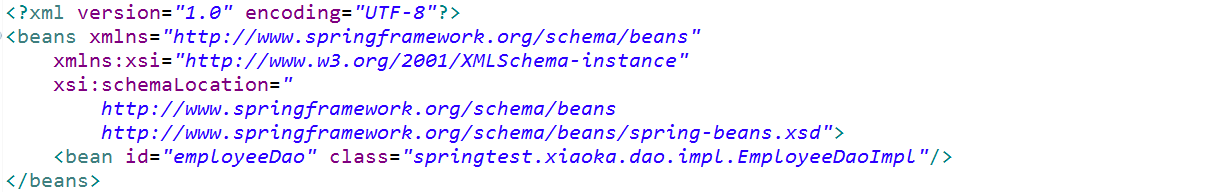
**三:创建接口EmployeeDao.java和实现类EmployeeDaoImpl.java**





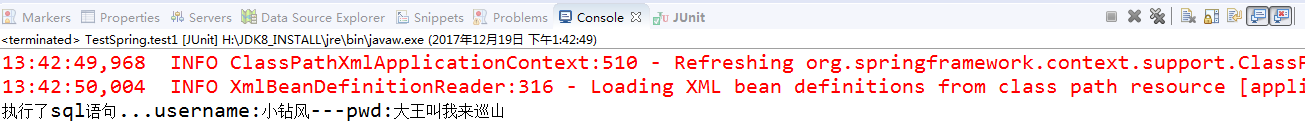
**四:创建配置文件applicationContext.xml**

在类路径下创建默认的配置文件applicationContext.xml。注意,配置文件的名称是任意的,但是一般使用默认的名称。

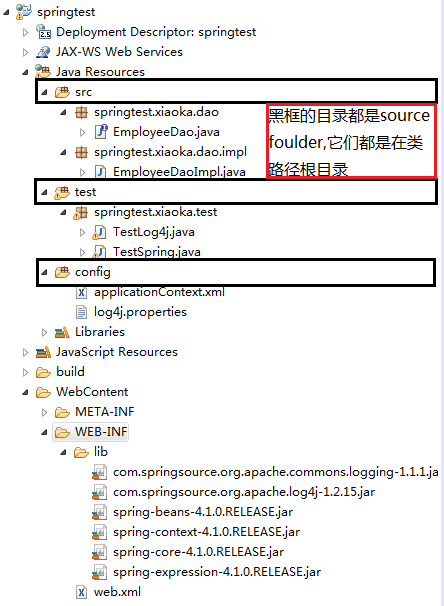


**五:测试代码并查看运行结果**



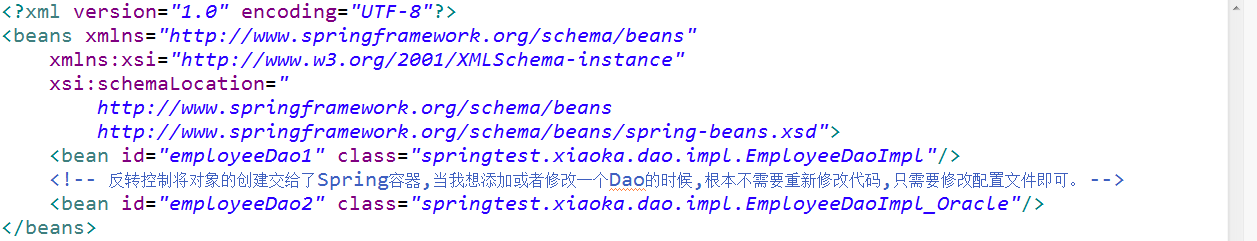


**六:项目完整目录示意图**



**小结**:至此,我们的spring入门程序就结束了。现在我们回头来看一下使用IoC容器的好处:它最大的程度的提高了程序的可扩展性以及最大程度的解绑了代码的耦合性。例如,假设我的Dao实现类目前是操作MySql数据库,当我想操作Oracle数据库的时候,只需要在applicationConext.xml中重新配置一个Bean节点即可实现,根本不用再去重新修改代码。

示例:添加一个Dao



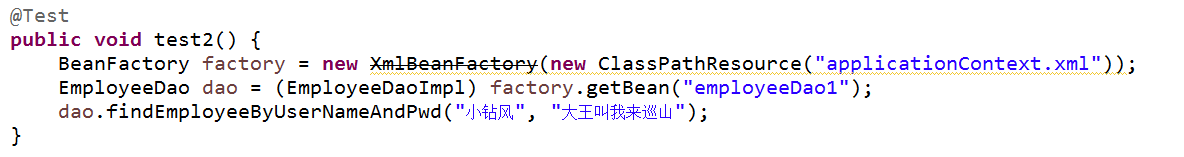
## Spring中的工厂

ApplicationContext是工厂接口,它有两个实现类

ClassPathXmlApplicationContext :加载类路径下的Spring配置文件

FileSystemXmlApplicationContext :加载本地磁盘下的Spring配置文件,很少使用

BeanFactory :Spring早期版本中用于创建Bean对象的工厂接口,已经不推荐使用



**这两个工厂接口的区别是:**

BeanFactory --采取延迟加载，在第一次getBean时才会初始化Bean

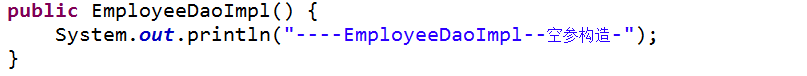
ApplicationContext -- 在加载applicationContext.xml时候就会创建具体的Bean对象的实 例,此外还提供了一些其他的功能,例如有事件传递、Bean自动装配、各种不同应用层的Context实现等。

## Bean的实例化

通过入门程序可以体会到Spring的IoC的优势,现在我们来介绍以下Spring创建Bean的三种方式

### 构造器方式

**①:spring默认调用JavaBean的空参构造来创建实例对象。**



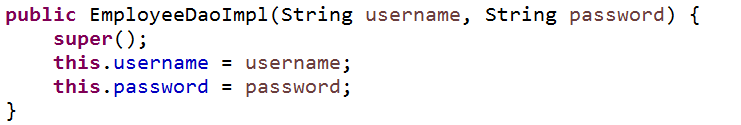
**测试**





**②:指定使用JavaBean的带参构造**

**EmployeeDaoImpl.java**



**配置文件**

<!-- 指定使用Bean的带参构造创建对象 -->

<bean id=*"employeeDao2"* class=*"springtest.xiaoka.dao.impl.EmployeeDaoImpl"*>

<!-- 直接通过属性名称为属性赋值 -->

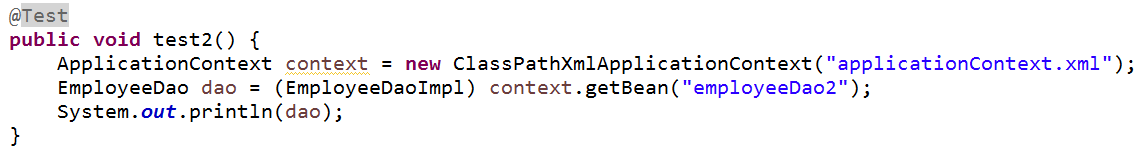
<constructor-arg name=*"username"* value=*"scott"* />

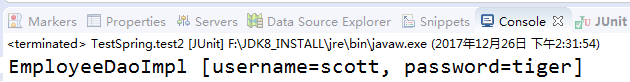
<!-- 通过参数索引为指定属性赋值 -->

<constructor-arg index=*"1"* value=*"tiger"* />

</bean>

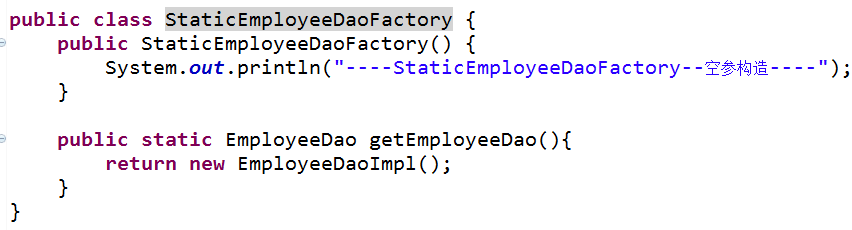
**测试运行,可以发现成员变量已经赋值**





### 静态工厂方式

一:创建静态工厂类，里边有一个静态方法用来获得JavaBean



二：配置文件中配置静态工厂类，指定调用工厂方法

<!-- 使用静态工厂类创建Bean -->

<!-- 只需要配置静态工厂类并指定静态方法即可得到想要的dao对象 -->

<!-- 注意:spring并不会创建静态工厂类的实例 -->

<bean id=*"staticEmployeeDaoFactory"* class=*"springtest.xiaoka.factory.StaticEmployeeDaoFactory"* factory-method=*"getEmployeeDao"*/>

三：测试：

@Test

**public** **void** test3() {

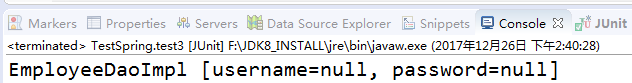
ApplicationContext context = **new** ClassPathXmlApplicationContext("applicationContext.xml");

//这里虽然是获取的静态工厂类的id值,但是却是返回静态工厂类的静态方法返回的对象

EmployeeDao dao = (EmployeeDaoImpl) context.getBean("staticEmployeeDaoFactory");

System.***out***.println(dao);

}



### 普通工厂方式

第一步：创建普通工厂类

**public** **class** CommonEmployeeDaoFactory {

**public** **void** CommonEmployeeDaoFactory (){

System.***out***.println("----CommonEmployeeDaoFactory----空参构造");

}

**public** EmployeeDao getEmployeeDao(){

System.***out***.println("----getEmployeeDao()方法被调用----");

**return** **new** EmployeeDaoImpl();

}

}

第二步：配置文件

<!-- 配置普通工厂类 -->

<bean id=*"comonFactory"* class=*"springtest.xiaoka.factory.CommonEmployeeDaoFactory"*/>

<!-- 配置一个employeeDao3的Bean节点,并不是直接配置EmployeeDaoImpl类,通过获取该节点bean来指定comonFactory来调用getEmployeeDao()方法创建对象-->

<bean id=*"employeeDao3"* factory-bean=*"comonFactory"* factory-method=*"getEmployeeDao"*/>

第三步：测试

@Test

**public** **void** test4() {

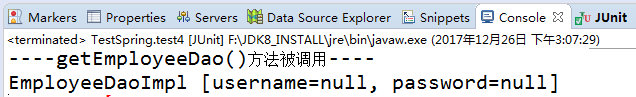
ApplicationContext context = **new** ClassPathXmlApplicationContext("applicationContext.xml");

//使用普通工厂类,普通工厂类会调用指定的函数来创建JavaBean

EmployeeDao dao = (EmployeeDaoImpl) context.getBean("employeeDao3");

System.***out***.println(dao);

}



## Bean标签详解

**id**:bean实例对象的唯一约束值,不允许出现诸如斜杠等特殊字符

**name**:和id属性用法差不多,允许出现特殊字符

**class**:对象的全限定类名

**scope**:对象的作用范围

singleton:指定当前bean为单例模式

prototype:指定当前bean为多例,每次都会创建一个新的bean

request:应用在Web项目中,每次请求都会创建一个新的bean

session:应用在Web项目中,同一个Session共享一个bean

globalsession:用在Web项目中,多服务器间的session共享一个bean,应用场景,例如单点登录

**lazy-init:**初始化对象的时机,仅当scope为singleton的时候才生效

lazy-init为ture,第一次获取对象时候创建Bean.

lazy-init为false，容器加载时候创建对象。

lazy-init默认值defalut相当于false.

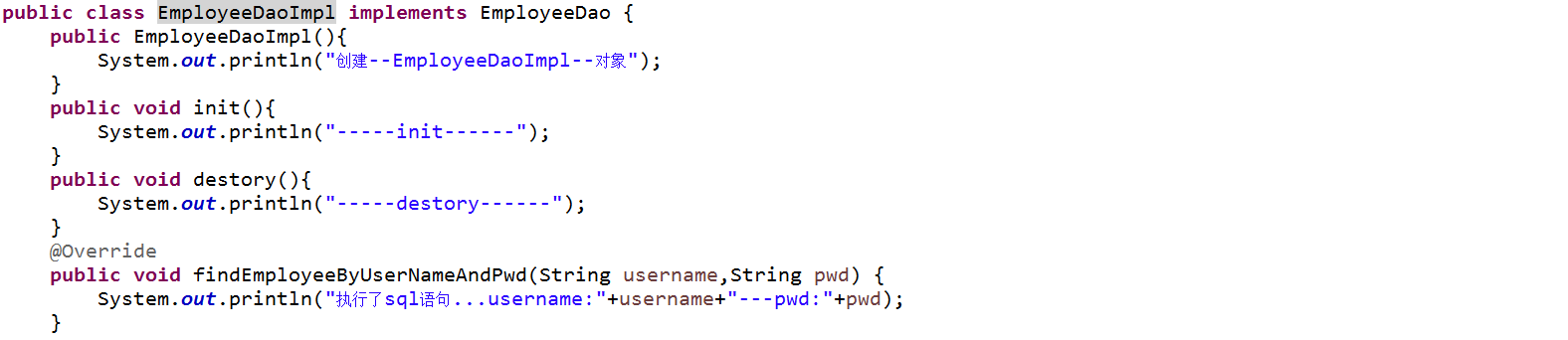
**init-method**:对象创建完毕以后调用的初始化方法

**destroy-method**:对象销毁调用的销毁方法

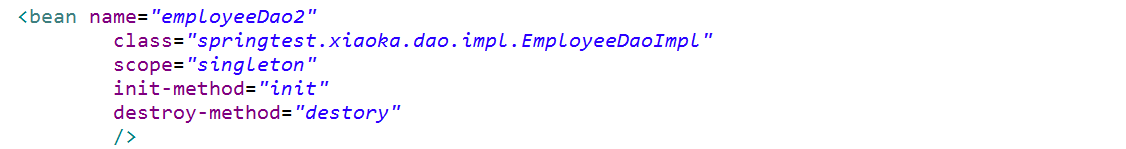
注意:init-method和destory-method对应的方法就是Bean的生命周期.

**示例**

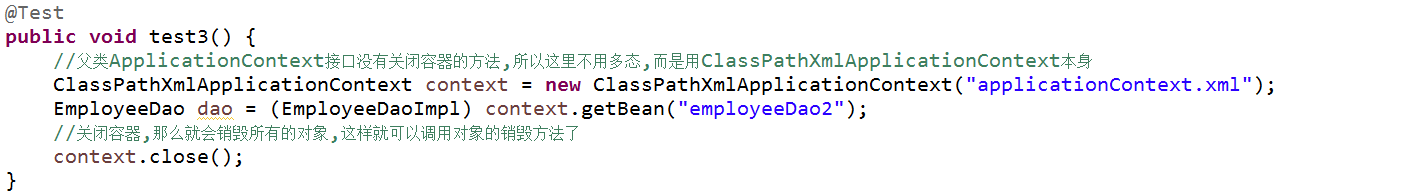
**一: EmployeeDaoImpl.java**



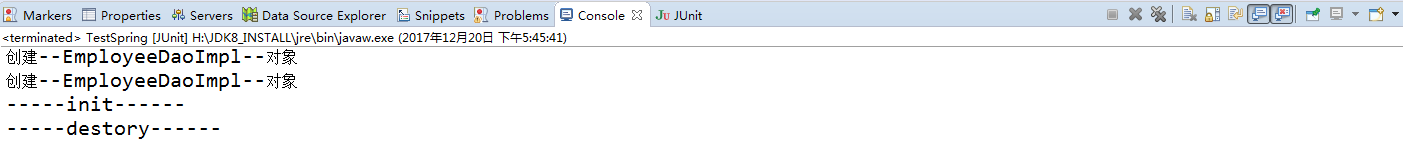
**二配置文件applicationContext.xml**



**三测试**



**四:结果**



## 依赖注入

依赖注入,英文名称Dependency Injection，简称[DI](https://baike.baidu.com/item/DI)：由外部容器动态地将依赖对象注入到另一个对象的组件中。spring采用这种方式为bean的属性赋值。 通俗的说spring容器不仅可以初始化对象，也可以为对象当中的成员变量赋值，初始化成员变量对象以及赋值的过程，不需手动编码，而是由spring容器去做。这种依赖容器初始化对象，并赋值给bean全局变量的方式叫依赖注入。依赖注入分为两种方式，一种是手动注入，一种是自动注入。

### 手动注入

手动注入又分为setter注入和构造方法注入两种方式

#### Setter注入

这种方式是调用JavaBean的set方法为属性赋值

setter方法注入：spring容器调用bean组件中属性对应的setter方法完成属性的注入（赋值）。

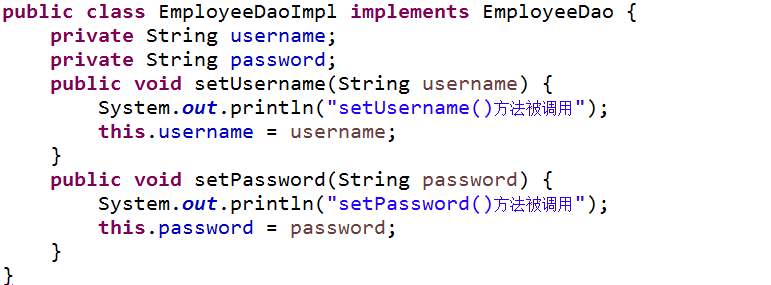
根据属性的不同类型，可分为如下三种注入情况。

**1 基本类型注入。2 spring组件类型注入。3集合类型注入。**

##### 一：基本类型注入

泛指：当前bean组件的属性为基本类型、基本类型的包装类型、String、StringBuffer类型，容器可以读取配置文件对当前属性进行赋值。

**一: EmployeeDaoImpl.java**



**二配置文件applicationContext.xml**

在<bean>中使用property标签为JavaBean属性赋值

<!-- 依赖注入:通过setter方法为Bean对象属性赋值 -->

<bean id=*"employeeDao4"* class=*"springtest.xiaoka.dao.impl.EmployeeDaoImpl"*>

<property name=*"username"* value=*"scott"*/>

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

</bean>

**测试**

@Test

**public** **void** test5() {

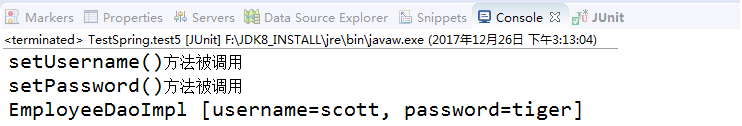
ApplicationContext context = **new** ClassPathXmlApplicationContext("applicationContext.xml");

//使用普通工厂类,普通工厂类会调用指定的函数来创建JavaBean

EmployeeDao dao = (EmployeeDaoImpl) context.getBean("employeeDao4");

System.***out***.println(dao);

}



##### 二 spring组件类型注入

spring组件类型注入：当前bean组件的属性为spring容器的其他bean组件，容器可以读取配置文件对当前属性进行赋值。

**一:创建EmployeeService.java**

**public** **class** EmployeeService {

//成员属性是spirng管理的bean组件

**private** EmployeeDao employeeDao;

**public** **void** setEmployeeDao(EmployeeDao employeeDao) {

**this**.employeeDao = employeeDao;

}

**public** **void** findEmployeeBuUserNameAndPwd(String username, String pwd) {

employeeDao.findEmployeeBuUserNameAndPwd(username, pwd);

}

}

**二配置文件applicationContext.xml**

<bean id=*"employeeDao4"* class=*"springtest.xiaoka.dao.impl.EmployeeDaoImpl"*>

<property name=*"username"* value=*"scott"*/>

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

</bean>

<bean id=*"employeeService"* class=*"springtest.xiaoka.service.EmployeeService"*>

<!-- 依赖注入:当Bean的属性是spring组件的时候,要用ref引用bean配置文件中的bean来赋值 -->

<property name=*"employeeDao"* ref=*"employeeDao4"*/>

</bean>

**测试**

@Test

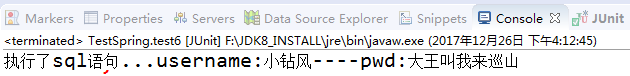
**public** **void** test6() {

ApplicationContext context = **new** ClassPathXmlApplicationContext("applicationContext.xml");

EmployeeService service = (EmployeeService) context.getBean("employeeService");

service.findEmployeeBuUserNameAndPwd("小钻风", "大王叫我来巡山");

}



##### 三集合类型注入

集合类型注入泛指：当前bean组件的属性为List Set Map、数组、Properties等。

环境准备【本例将三个集合都写进去了】

**一: EmployeeDaoImpl.java**

**public** **class** EmployeeDaoImpl **implements** EmployeeDao {

**private** String username;

**private** String password;

**private** List driverClass;

**private** Set users;

**private** Map urls;

}

**二配置文件applicationContext.xml**

注意：由于集合中会放入spring的组件bean,这个时候要**防止**属性关联，否则会不停的创建对象，导致内存溢出。

属性关联：本例中配置集合属性的JavaBean是EmployeeDaoImpl类，那么就不能在EmployeeDaoImpl的bean配置中再创建EmployeeDaoImpl对象。

<bean id=*"employeeDao5"* class=*"springtest.xiaoka.dao.impl.EmployeeDaoImpl"*>

<property name=*"username"* value=*"scott"* />

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

<!-- 为List集合属性赋值 -->

<property name=*"driverClass"*>

<list>

<!-- 元素存到集合中的顺序就是配置的顺序,基本类型和spring组建类型都可以使用 -->

<!-- 放到集合的第一个位置[0号角标,这是基本类型] -->

<value type=*"java.lang.String"*>example1.driverClass</value>

<!-- 放到集合的第二个位置[1号角标,这是spring组建类型] -->

<ref bean=*"employeeDao4"* />

</list>

</property>

<!-- 为set集合属性赋值 -->

<property name=*"users"*>

<set>

<!-- 基本类型和spring组建类型都可以使用 -->

<value type=*"java.lang.String"*>example1.user</value>

<ref bean=*"employeeDao4"* />

</set>

</property>

<!-- 为Map集合属性赋值 -->

<property name=*"urls"*>

<map>

<!-- 无论key还是value,基本类型和spring组建类型都可以使用 -->

<entry key=*"url1"* value=*"example.url1"*/>

<entry key-ref=*"employeeDao4"* value-ref=*"employeeDao4"*></entry>

</map>

</property>

</bean>

**测试**

@Test

**public** **void** test7() {

ApplicationContext context = **new** ClassPathXmlApplicationContext("applicationContext.xml");

EmployeeDao dao = (EmployeeDaoImpl) context.getBean("employeeDao5");

System.***out***.println(dao);

}

#### 构造器方式注入

构造器方法注入：spring在初始化bean组件时，调用含参数的构造器对全局变量进行复制，参数类型可以为基本类型、bean组件类型、集合类型，赋值方式与setter方法赋值方式一致。[其实在介绍bean的实例化的时候已经使用过了该方式，就是使用constructor-arg标签]

**一: EmployeeDaoImpl.java**

**public** **class** EmployeeDaoImpl **implements** EmployeeDao {

**private** String username;

**private** String password;

**private** List driverClass;

**private** Set users;

**private** Map urls;

**public** EmployeeDaoImpl(String username, String password, List driverClass, Set users, Map urls) {

**super**();

**this**.username = username;

**this**.password = password;

**this**.driverClass = driverClass;

**this**.users = users;

**this**.urls = urls;

}

}

**二配置文件applicationContext.xml**

<bean id=*"employeeDao6"* class=*"springtest.xiaoka.dao.impl.EmployeeDaoImpl"*>

<constructor-arg name=*"username"* value=*"scott"* />

<constructor-arg name=*"password"* value=*"tiger"* />

<!-- 为List集合属性赋值 -->

<constructor-arg name=*"driverClass"*>

<list>

<!-- 元素存到集合中的顺序就是配置的顺序,基本类型和spring组建类型都可以使用 -->

<!-- 放到集合的第一个位置[0号角标,这是基本类型] -->

<value type=*"java.lang.String"*>example1.driverClass</value>

<!-- 放到集合的第二个位置[1号角标,这是spring组建类型] -->

<ref bean=*"employeeDao4"* />

</list>

</constructor-arg>

<!-- 为set集合属性赋值 -->

<constructor-arg name=*"users"*>

<set>

<!-- 基本类型和spring组建类型都可以使用 -->

<value type=*"java.lang.String"*>example1.user</value>

<ref bean=*"employeeDao4"* />

</set>

</constructor-arg>

<!-- 为Map集合属性赋值 -->

<constructor-arg name=*"urls"*>

<map>

<!-- 无论key还是value,基本类型和spring组建类型都可以使用 -->

<entry key=*"url1"* value=*"example.url1"* />

<entry key-ref=*"employeeDao4"* value-ref=*"employeeDao4"*></entry>

</map>

</constructor-arg>

</bean>

**测试**

@Test

**public** **void** test8() {

ApplicationContext context = **new** ClassPathXmlApplicationContext("applicationContext.xml");

//使用普通工厂类,普通工厂类会调用指定的函数来创建JavaBean

EmployeeDao dao = (EmployeeDaoImpl) context.getBean("employeeDao6");

System.***out***.println(dao);

}

#### 命名空间P属性注入[了解]

Spring2.x版本后提供的方式.

**配置文件,要引入命名空间p**

<beans xmlns=*"http://www.springframework.org/schema/beans"*

xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*

xmlns:p=*"http://www.springframework.org/schema/p"*

xsi:schemaLocation=*"*

*http://www.springframework.org/schema/beans*

[*http://www.springframework.org/schema/beans/spring-beans.xsd">*](http://www.springframework.org/schema/beans/spring-beans.xsd%22%3e)

**配置文件applicationContext.xml**

*<!-- 使用P名称空间的方式为属性注入值 -->*

<bean id=*"employeeDao7"* class=*"springtest.xiaoka.dao.impl.EmployeeDaoImpl"* p:username=*"scott"* p:password=*"tiger"*/>

**测试**

@Test

**public** **void** test9() {

ApplicationContext context = **new** ClassPathXmlApplicationContext("applicationContext.xml");

// 使用普通工厂类,普通工厂类会调用指定的函数来创建JavaBean

EmployeeDao dao = (EmployeeDaoImpl) context.getBean("employeeDao7");

System.***out***.println(dao);

}

#### SPEL表达式属性注入[了解]

SpEL的方式的属性注入:Spring3.x版本后提供的方式.

**配置文件applicationContext.xml**

<!-- 使用SPEL表达式为属性注入值 -->

<bean id=*"employeeDao8"* class=*"springtest.xiaoka.dao.impl.EmployeeDaoImpl"*>

<!-- 字符串要使用单引号 -->

<property name=*"username"* value=*"#{'小钻风'}"* />

<!-- 数字直接写 -->

<property name=*"password"* value=*"#{123}"* />

</bean>

**测试**

@Test

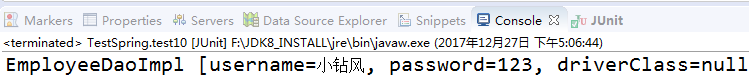
**public** **void** test10() {

ApplicationContext context = **new** ClassPathXmlApplicationContext("applicationContext.xml");

EmployeeDao dao = (EmployeeDaoImpl) context.getBean("employeeDao8");

System.***out***.println(dao);

}



#### 属性文件注入[了解]

**一: EmployeeDaoImpl.java**

**public** **class** EmployeeDaoImpl **implements** EmployeeDao {

**private** Properties pro;

**private** String username;

**private** String password;

**private** List driverClass;

**private** Set users;

**private** Map urls;

}

**二:配置文件applicationContext.xml**

<!-- 当成员属性是Properties类型的时候 -->

<bean id=*"employeeDao9"* class=*"springtest.xiaoka.dao.impl.EmployeeDaoImpl"*>

<property name=*"pro"*>

<props>

<prop key=*"username"*>钻风</prop>

<prop key=*"password"*>大王叫我来巡山</prop>

</props>

</property>

</bean>

**三:测试**

@Test

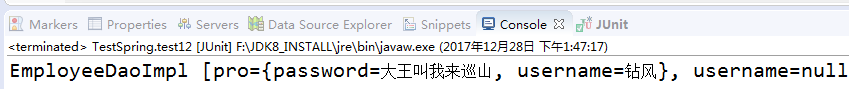
**public** **void** test12() {

ApplicationContext context = **new** ClassPathXmlApplicationContext("applicationContext.xml");

EmployeeDao dao = (EmployeeDaoImpl) context.getBean("employeeDao9");

System.***out***.println(dao);

}



### 自动注入

自动注入也叫自动装配，这种方式是根据autowire的属性值来执行对应的匹配规则为属性赋值，自动装配可以省略<property></property>

autowire属性取值的匹配规则

**\* byType （默认值）** **：**按类型装配，可以根据属性的类型，在容器中寻找跟该类型匹配的bean。如果发现多个，那么将会抛出异常。如果没有找到，即属性值为null。

**\* byName**：按名称装配，可以根据属性的名称，在容器中寻找跟该属性名相同的bean，如果没有找到，即属性值为null。

**\* constructor**与byType的方式类似，不同之处在于它应用于构造器参数。如果在容器中没有找到与构造器参数类型一致的bean，否则将会抛出异常。

示例

**配置文件**

<!--根据类型为当前bean的属性赋值,如果没找到则属性值为null,如果发现多个则抛异常 -->

<bean id=*"employeeService2"* class=*"springtest.xiaoka.service.EmployeeService"* autowire=*"byType"* />

<!--根据属性名称为当前bean的属性赋值,如果没找到则属性值为null -->

<bean id=*"employeeService2"* class=*"springtest.xiaoka.service.EmployeeService"* autowire=*"byName"* />

**测试**

@Test

**public** **void** test11(){

ApplicationContext context = **new** ClassPathXmlApplicationContext("applicationContext.xml");

EmployeeService employeeService1 = (EmployeeService) context.getBean("employeeService1");

EmployeeService employeeService2 = (EmployeeService) context.getBean("employeeService2");

employeeService1.findEmployeeBuUserNameAndPwd("小钻风", "大王叫我来巡山");

employeeService2.findEmployeeBuUserNameAndPwd("小钻风", "大王叫我来巡山");

}

# 注解完成IOC配置

Spring3.0后为我们引入了组件自动扫描机制，它可以在类路径底下寻找标注了@Component、@Service、@Controller、@Repository注解的类，并把这些类纳入进spring容器中管理。

也就是说配置了注解以后的作用和在spring的配置文件中使用bean节点配置组件是一样的。

注解是为了替代传统的xml配置方式而生。

**要使用自动扫描机制，我们需要打开以下配置信息:**

1、引入AOP的jar包（之前的版本不用）。

2、spring配置文件增加context命名空间与xsd的引用，使用注解，需要context组件解析配置文件。

3、增加context的动态扫描

## 注解实现反转控制

**各个注解的含义**

@Service：用于标注业务层组件、

@Controller：用于标注控制层组件（如struts中的action）、

@Repository：用于标注数据访问组件，即DAO组件。

@Component：泛指组件，例如java类

注意：这几个注解标注的javabean在使用时并没有什么不同。但预计spring之后的版本会区分这几个注解的特殊性，因此建议在目前版本开发时按照此规则标注注解。

示例代码

**一:导入AOP的jar包**



**二:创建配置文件applicationContext2.xml**

<beans xmlns=*"http://www.springframework.org/schema/beans"*

xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*

xmlns:p=*"http://www.springframework.org/schema/p"*

xmlns:context=*"http://www.springframework.org/schema/context"*

xsi:schemaLocation=*"*

*http://www.springframework.org/schema/beans*

*http://www.springframework.org/schema/beans/spring-beans.xsd*

*http://www.springframework.org/schema/context*

*http://www.springframework.org/schema/context/spring-context.xsd"*>

<!--

开启动态扫描,需要引入上下文约束

spring在加载的时候就会扫描base-package下的包以及其子包里所有的java类,

凡是使用了spring注解的都会交给spring来管理

-->

<context:component-scan base-package=*"springtest.xiaoka"*/>

</beans>

**三: EmployeeDaoImpl.java和EmployeeService.java**

/\*这相当于配置<bean id="employeeDao" class="springtest.xiaoka.dao.impl.EmployeeDaoImpl" />\*/

@Repository(value="employeeDao")

**public** **class** EmployeeDaoImpl **implements** EmployeeDao {}

/\*

\* 这相当于配置<bean id="employeeService"

\*class="springtest.xiaoka.dao.impl.EmployeeService" />

\* 也就是说默认id值是类名首字母小写

\*/

@Service

**public** **class** EmployeeService {}

**四:测试**

@Test

**public** **void** test13(){

ApplicationContext context = **new** ClassPathXmlApplicationContext("applicationContext2.xml");

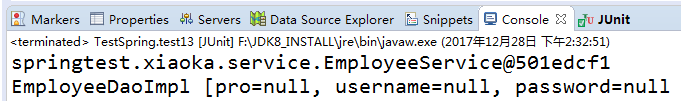
EmployeeDao dao = (EmployeeDaoImpl) context.getBean("employeeDao");

EmployeeService service = (EmployeeService) context.getBean("employeeService");

System.***out***.println(service);

System.***out***.println(dao);

}



可以看到,新的配置文件仅仅是使用了动态扫描,那么凡是使用了注解的类都成为了spring的组建。

## 注解完成依赖注入

使用注解注入的方式,可以不用提供set方法

|  |  |  |  |
| --- | --- | --- | --- |
| 注解名称 | 作用对象 | 解释 | 分类 |
| @Value(value=””) | 成员属性 | 用于注入基本类型数据,value可以省略 | 属性注入的注解 |
| @Autowired | 成员属性 | 自动装配,默认按类型进行装配  @Qualifier:强制使用名称注入. |
| @Resource(name=””) | 成员属性 | 等同于@Autowired和@Qualifier一起使用.name不可以省略 |
| @Scope(value=””) | 类 | singleton:单例  prototype:多例 | Bean作用范围的注解 |
| @PostConstruct | 成员函数 | 配置Bean的初始化函数 | Bean生命周期的注解 |
| @PreDestroy | 成员函数 | 配置Bean的销毁函数 |
| 注意:  ①:一般不会使用注解为集合进行依赖注入,当出现集合属性的时候,更多的使用xml进行配置  ②:@Resource注解是java的,并不是spring的,只不过spring也支持它而已 | | | |

示例代码

一:**配置文件applicationContext2.xml**

<beans xmlns=*"http://www.springframework.org/schema/beans"*

xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*

xmlns:p=*"http://www.springframework.org/schema/p"*

xmlns:context=*"http://www.springframework.org/schema/context"*

xsi:schemaLocation=*"*

*http://www.springframework.org/schema/beans*

*http://www.springframework.org/schema/beans/spring-beans.xsd*

*http://www.springframework.org/schema/context*

*http://www.springframework.org/schema/context/spring-context.xsd"*>

<context:component-scan base-package=*"springtest.xiaoka"*/>

</beans>

**二:EmployeeDaoImpl.java**

@Repository(value="employeeDao")

**public** **class** EmployeeDaoImpl **implements** EmployeeDao {

@Value(value="小钻风")//使用注解完成依赖注入:对基本类型进行赋值

**private** String username;

@Value("大王叫我来巡山")//使用注解完成依赖注入:对基本类型进行赋值[省略value]

**private** String password;

@PostConstruct//使用注解完成初始化函数

**public** **void** init(){

System.***out***.println("----初始化函数执行-----");

}

@PreDestroy//使用注解完成销毁函数

**public** **void** destory(){

System.***out***.println("----销毁函数执行-----");

}

**public** EmployeeDaoImpl() {

System.***out***.println("----EmployeeDaoImpl--空参构造-");

}

@Override

**public** String toString() {

**return** "EmployeeDaoImpl [username=" + username + ", password=" + password + "]";

}

@Override

**public** **void** findEmployeeBuUserNameAndPwd(String username, String pwd) {

System.***out***.println("执行了sql语句...username:"+username+"----pwd:"+pwd);

}

}

**三:EmployeeService.java**

@Service

**public** **class** EmployeeService {

@Autowired//使用注解完成依赖注入:为spring组建赋值

**private** EmployeeDao employeeDao;

**public** **void** findEmployeeBuUserNameAndPwd(String username, String pwd) {

employeeDao.findEmployeeBuUserNameAndPwd(username, pwd);

}

}

**四:测试**

@Test

**public** **void** test14(){

ClassPathXmlApplicationContext context = **new** ClassPathXmlApplicationContext("applicationContext2.xml");

EmployeeDao dao = (EmployeeDaoImpl) context.getBean("employeeDao");

System.***out***.println(dao);

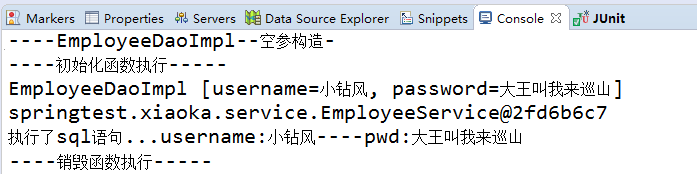
EmployeeService service = (EmployeeService) context.getBean("employeeService");

System.***out***.println(service);

service.findEmployeeBuUserNameAndPwd("小钻风", "大王叫我来巡山");

context.close();

}



## 注解和xml配置的比较

|  |  |  |
| --- | --- | --- |
|  | 基础XML配置 | 基于注解配置 |
| Bean定义 | <bean id=”xxx” class=”…”/> | @Component  @Controller  @Service  @Repository |
| Bean名称 | 通过id或者name属性指定 | @Component(value=”xxx”) |
| Bean注入 | <property>或者p命名空间 | @Autowired按类型注入  @Qualifier按名称注入 |
| 生命过程，  Bean作用范围 | Init-method、destory-method  范围scope属性 | @PostConstruct初始化  @PreDestory销毁  @Scope设置作用范围 |
| 适合场景 | Bean来自第三方 | Bean的实现类由用户自己开发 |

XML和注解:

\* XML :结构清晰.

\* 注解 :开发方便.(属性注入.)

实际开发中还有一种XML和注解整合开发:

\* Bean有XML配置.但是使用的属性使用注解注入.

# Spring的AOP机制

## AOP简介

**什么是AOP**

在软件业，AOP为Aspect Oriented Programming的缩写，意为：面向切面编程

AOP是一种编程范式，隶属于软工范畴，指导开发者如何组织程序结构

AOP最早由AOP联盟的组织提出的,制定了一套规范.Spring将AOP思想引入到框架中,必须遵守AOP联盟的规范

通过预编译方式和运行期动态代理实现程序功能的统一维护的一种技术

AOP是OOP的延续，是软件开发中的一个热点，也是Spring框架中的一个重要内容，是函数式编程的一种衍生范型

利用AOP可以对业务逻辑的各个部分进行隔离，从而使得业务逻辑各部分之间的耦合度降低，提高程序的可重用性，同时提高了开发的效率

AOP采取横向抽取机制，取代了传统纵向继承体系重复性代码（性能监视、事务管理、安全检查、缓存）

**为什么要学习AOP**

可以在不修改源代码的前提下，对程序进行增强！！！它意图将日志记录，性能统计，安全控制，事务处理，[异常处理](https://baike.baidu.com/item/%E5%BC%82%E5%B8%B8%E5%A4%84%E7%90%86)等代码从业务逻辑代码中划分出来，通过对这些行为的分离，我们希望可以将它们独立到非指导业务逻辑的方法中，进而改变这些行为的时候不影响业务逻辑的代码。

**Spring框架的AOP的底层实现**

Srping框架的AOP技术底层也是采用的代理技术，代理的方式提供了JDK和CGLIB两种.

Spring的传统AOP中根据类是否实现接口，来采用不同的代理方式

1.如果实现类接口，使用JDK动态代理完成AOP

2.如果没有实现接口，采用CGLIB动态代理完成AOP

一：基于JDK的动态代理

必须是面向接口的，只有实现了具体接口的类才能生成代理对象

二：基于CGLIB动态代理【附录】

对于没有实现了接口的类，也可以产生代理，产生这个类的子类的方式

## AOP入门程序

AOP相关术语

Joinpoint(连接点): 所谓连接点是指那些被拦截到的点。在spring中,这些点指的是方法,因为spring只支持方法类型的连接点.

Pointcut(切入点): 所谓切入点是指我们要对哪些Joinpoint进行拦截的定义.

Advice(通知/增强):

所谓通知是指拦截到Joinpoint之后所要做的事情就是通知.通知分为前置通知,后置通知,异常通知,最终通知,环绕通知(切面要完成的功能)

Introduction(引介): 引介是一种特殊的通知在不修改类代码的前提下, Introduction可以在运行期为类动态地添加一些方法或Field.

Target(目标对象): 代理的目标对象

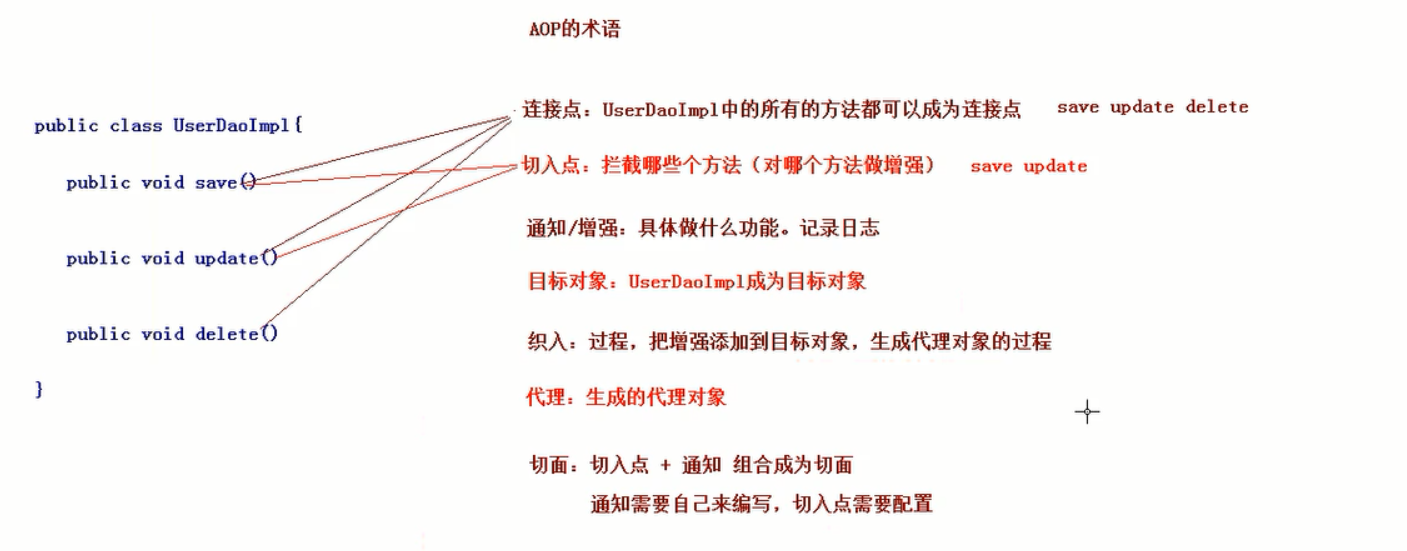
Weaving(织入):

是指把增强应用到目标对象来创建新的代理对象的过程.

spring采用动态代理织入，而AspectJ采用编译期织入和类装在期织入

Proxy（代理）: 一个类被AOP织入增强后，就产生一个结果代理类

Aspect(切面): 是切入点和通知（引介）的结合

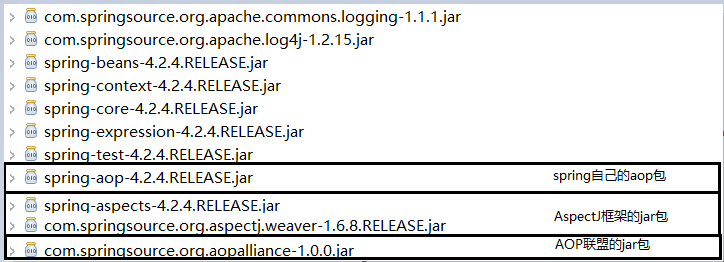


示例代码

**一：导包**

Spring引入AOP思想，所以要导入AOP联盟包

Spring的AOP是基于AspectJ的AOP的开发，所以要导入AspectJ的框架包



**二：创建切面类和EmployeeDaoImpl.java**

**public** **class** EmployeeAspect {

**public** **void** log(){

System.***out***.println("------记录日志------");

}

}

**public** **class** EmployeeDaoImpl **implements** EmployeeDao {

@Override

**public** **void** save() {

System.***out***.println("--------保存员工------");

}

}

**三:配置文件applicationContext.xml**

<beans xmlns=*"http://www.springframework.org/schema/beans"*

xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*

xmlns:context=*"http://www.springframework.org/schema/context"*

xmlns:aop=*"http://www.springframework.org/schema/aop"*

xsi:schemaLocation=*"*

*http://www.springframework.org/schema/beans*

*http://www.springframework.org/schema/beans/spring-beans.xsd*

*http://www.springframework.org/schema/context*

*http://www.springframework.org/schema/context/spring-context.xsd*

*http://www.springframework.org/schema/aop*

*http://www.springframework.org/schema/aop/spring-aop.xsd"*>

<bean id=*"employeeDao"* class=*"springtest.xiaoka.dao.impl.EmployeeDaoImpl"* />

<!-- 配置切面类 -->

<bean id=*"employeeAspect"* class=*"springtest.xiaoka.aspect.EmployeeAspect"* />

<!-- 配置AOP -->

<aop:config>

<!--配置切面：切入点+通知 -->

<aop:aspect ref=*"employeeAspect"*>

<!-- 配置前置通知，log函数会在save函数执行前执行 -->

<aop:before method=*"log"*

pointcut=*"execution(public void springtest.xiaoka.dao.impl.EmployeeDaoImpl.save())"* />

</aop:aspect>

</aop:config>

</beans>

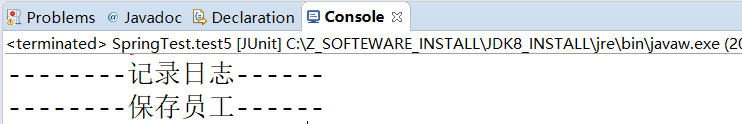
**四：测试**

@Test

**public** **void** test5(){

employeeDao.save();

}



通过测试可以看到在不改变源码的情况下对dao的save方法进行了功能增强。

## AOP的通知类型

1. 前置通知

切入点执行之前执行。

配置文件信息：<aop:before method="before" pointcut-ref="myPointcut"/>

应用：可以对方法的参数来做校验

2. 最终通知

切入点执行之后执行，如果程序出现了异常，最终通知也会执行。

在配置文件中编写具体的配置：<aop:after method="after" pointcut-ref=" myPointcut"/>

应用：例如像释放资源

3. 后置通知

切入点执行之后执行，如果程序出现了异常，后置通知不会执行。

在配置文件中编写具体的配置：<aop:after-returning method="afterReturning" pointcut-ref=" myPointcut"/>

应用：可以修改方法的返回值

4. 异常抛出通知

切入点抛出异常后通知

在配置文件中编写具体的配置：<aop:after-throwing method="afterThrowException" pointcut-ref=" myPointcut"/>

应用：包装异常的信息

5. 环绕通知

切入点的执行前后执行。

在配置文件中编写具体的配置：<aop:around method="around" pointcut-ref=" myPointcut"/>

要注意：**目标的方法默认不执行**，需要使用ProceedingJoinPoint对来让目标对象的方法执行。

示例代码

**一:切面类和EmployeeDaoImpl.java**

**public** **class** EmployeeAspect {

**public** **void** log(){

System.***out***.println("记录日志");

}

**public** **void** round(ProceedingJoinPoint pjp){

**try** {

System.***out***.println("--------sql执行前日志------");

pjp.proceed();

} **catch** (Throwable e) {

e.printStackTrace();

}

System.***out***.println("--------sql执行后日志------");

}

}

**public** **class** EmployeeDaoImpl **implements** EmployeeDao {

**public** **void** before() {

System.***out***.println("dao:执行Sql语句");

}

**public** **void** after() {

System.***out***.println("dao:执行Sql语句");

}

**public** **void** afterReturning() {

System.***out***.println("dao:执行Sql语句");

}

**public** **void** round() {

System.***out***.println("dao:执行Sql语句");

}

**public** **void** afterThrowException() {

System.***out***.println("dao:执行Sql语句");

}

}

**二:配置文件applicationContext.xml**

<?xml version=*"1.0"* encoding=*"UTF-8"*?>

<beans xmlns=*"http://www.springframework.org/schema/beans"*

xmlns:xsi=*"*[*ttp://www.w3.org/2001/XMLSchema-instance"*](ttp://www.w3.org/2001/XMLSchema-instance%22)

xmlns:context=*"http://www.springframework.org/schema/context"*

xmlns:aop=*"http://www.springframework.org/schema/aop"*

xsi:schemaLocation=*"*

*http://www.springframework.org/schema/beans*

*http://www.springframework.org/schema/beans/spring-beans.xsd*

*http://www.springframework.org/schema/context*

*http://www.springframework.org/schema/context/spring-context.xsd*

*http://www.springframework.org/schema/aop*

*http://www.springframework.org/schema/aop/spring-aop.xsd"*>

<bean id=*"employeeDao"* class=*"springtest.xiaoka.dao.impl.EmployeeDaoImpl"* />

<!-- 配置切面类 -->

<bean id=*"employeeAspect"* class=*"springtest.xiaoka.aspect.EmployeeAspect"* />

<!-- 配置AOP -->

<aop:config>

<!--配置切面：切入点+通知 -->

<aop:aspect ref=*"employeeAspect"*>

<!-- 配置前置通知，log函数会在save函数执行前执行 -->

<aop:before method=*"log"*

pointcut=*"execution(public void springtest.xiaoka.dao.impl.EmployeeDaoImpl.before())"* />

<!-- 配置后置通知 -->

<aop:after-returning method=*"log"* pointcut=*"execution(public void springtest.xiaoka.dao.impl.EmployeeDaoImpl.after())"*/>

<!-- 配置最终通知 -->

<aop:after method=*"log"* pointcut=*"execution(public void springtest.xiaoka.dao.impl.EmployeeDaoImpl.afterReturning())"*/>

<!-- 配置环绕通知 -->

<aop:around method=*"round"* pointcut=*"execution(public void springtest.xiaoka.dao.impl.EmployeeDaoImpl.round())"*/>

<!-- 配置异常通知 -->

<aop:after-throwing method=*"log"* pointcut=*"execution(public void springtest.xiaoka.dao.impl.EmployeeDaoImpl.afterThrowException())"*/>

</aop:aspect>

</aop:config>

</beans>

**三：测试**

@Test

**public** **void** test5(){

employeeDao.before();

}

@Test

**public** **void** test6(){

employeeDao.after();

}

@Test

**public** **void** test7(){

employeeDao.afterReturning();

}

@Test

**public** **void** test8(){

employeeDao.round();

}

@Test

**public** **void** test9(){

employeeDao.afterThrowException();

}

## 切入点表达式

切入点表达式：切入点表达式用于声明spring容器中哪些组件的函数是目标函数，也就是切面程序要作用到哪些组件的哪些函数上。

* 常用的切入点表达式分为：
  + 按类匹配：匹配的java类中全部函数作为目标函数，使用within关键字。
  + 按函数匹配：匹配的函数作为目标函数，使用execution关键字。
  + 按bean的id匹配：匹配的bean中全部函数作为目标组件。使用bean关键字

按类匹配

|  |
| --- |
| 匹配到具体的类，那么该类下所有的方法都将是切入点  <aop:pointcut id=”targetPointcut” expression=”within(com.xiaoka.target)”/> |
| 匹配到包下的类，那么该包下所有的类的所有方法都将是切入点  <aop:pointcut id=”targetPointcut” expression=”within(com.xiaoka.\*)”/> |
| 匹配到包下及其子包下的类，那么该包已经该包的子包里边所有的类的所有方法都将是切入点  <aop:pointcut id=”targetPointCut” expression=”within(com..\*)”> |

按函数匹配

|  |  |
| --- | --- |
| 完整写法：只有指定类的指定函数是切入点程序  返回值类型限制，方法名称限制，方法参数限制 | 只有Target类的save(String)方法是切入点程序，返回值类型必须是String  Execution (String com.xiaoka.Target.save(String)) |
| 返回值类型不限制：指定类的指定函数是切入点程序  方法名称限制，方法参数限制 | Target类的save(String)方法是切入点程序，返回值类型不限制  Execution (\* com.xiaoka.Target.save(String)) |
| 类名称不限制：指定包下的所有类，任意返回值类型作为切入点程序  方法名称限制，方法参数限制 | Com.xiaoka包下的所有类，只要函数名是save且参数是String,那么它就是切入点程序  Execution (\* com.xiaoka.\*.save(String)) |
| 类名称和方法名称都不限制：指定包下的所有类，任意返回值类型，任意函数作为切入点程序  方法参数限制 | Com.xiaoka包下的所有类所有函数,只要参数是Sting 类型，就作为切入点程序  Execution (\* com.xiaoka.\*.\*(String)) |
| 全部不限制 | Com包及其子包下所有的类的所有函数都作为切入点程序  Execttion (\* com..\*.\*(..)) |

按bean的ID匹配

|  |
| --- |
| 根据bean组件名称匹配，那么target组件下所有的函数都是切入点程序  <aop:pointcut id=”targetPoint” expression=”bean(target)”> |
| 根据bean组件名称(含通配符)匹配，target1,target2…下的函数都是切入点程序  <aop:pointcut id=”targetPoint” expression=”bean(target\*)”> |

## 注解完成AOP

与IoC一样，Spring同样提供了AOP的注解开发方式。使用AOP注解需要开启ASPECT自动代理

**各个通知类型注解**

前置通知

@org.aspectj.lang.annotation.Before("execution(\* com..\*.\*(..))")

后置通知

@AfterReturning(value="execution(\* com..\*.\*(..))")

异常通知

@AfterThrowing(value="execution(\* com..\*.\*(..))")

最终通知

@After(value="execution(\* com..\*.\*(..))")

环绕通知

@org.aspectj.lang.annotation.Around("execution(\* com..\*.\*(..))")

自定义切入点

@Pointcut(value = "execution(\* springtest..\*.\*())")

public void myPoint() {

}

**示例代码**

**一：切面类**

@Component

@Aspect

**public** **class** EmployeeAspect {

//使用自定义切入点

@Before(value = "EmployeeAspect.myPoint1()")

**public** **void** log1() {

System.***out***.println("前置通知记录日志");

}

@AfterReturning(value = "EmployeeAspect.myPoint2()")

**public** **void** log2() {

System.***out***.println("后置通知记录日志");

}

@AfterThrowing(value = "EmployeeAspect.myPoint3()")

**public** **void** log3() {

System.***out***.println("异常通知记录日志");

}

@After(value = "EmployeeAspect.myPoint4()")

**public** **void** log4() {

System.***out***.println("最终通知记录日志");

}

@Around(value = "execution(public void springtest.xiaoka.dao.impl.EmployeeDaoImpl.round())")

**public** **void** log5(ProceedingJoinPoint pjp) {

**try** {

System.***out***.println("--------环绕通知sql执行前日志------");

pjp.proceed();

} **catch** (Throwable e) {

e.printStackTrace();

}

System.***out***.println("--------环绕通知sql执行后日志------");

}

@Pointcut(value = "execution(\* springtest..\*.add\*())")

**public** **void** myPoint1() {

}

@Pointcut(value = "execution(\* springtest..\*.update\*())")

**public** **void** myPoint2() {

}

@Pointcut(value = "execution(\* springtest..\*.delete\*())")

**public** **void** myPoint3() {

}

@Pointcut(value = "execution(\* springtest..\*.query\*())")

**public** **void** myPoint4() {

}

}

**二:配置文件applicationContext.xml**

<?xml version=*"1.0"* encoding=*"UTF-8"*?>

<beans xmlns=*"http://www.springframework.org/schema/beans"*

xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"* xmlns:context=*"http://www.springframework.org/schema/context"*

xmlns:aop=*"http://www.springframework.org/schema/aop"*

xsi:schemaLocation=*"*

*http://www.springframework.org/schema/beans*

*http://www.springframework.org/schema/beans/spring-beans.xsd*

*http://www.springframework.org/schema/context*

*http://www.springframework.org/schema/context/spring-context.xsd*

*http://www.springframework.org/schema/aop*

*http://www.springframework.org/schema/aop/spring-aop.xsd"*>

<context:component-scan base-package=*"springtest.xiaoka"* />

<!-- 开启aspect自动代理 -->

<aop:aspectj-autoproxy />

</beans>

**三：EmployeeDaoImpl.java**

**略**

**四：测试**

@Test

**public** **void** test1(){

employeeDao.addEmployee();

}

@Test

**public** **void** test2(){

employeeDao. updateEmployee ();

}

@Test

**public** **void** test3(){

employeeDao. deleteEmployee ();

}

@Test

**public** **void** test4(){

employeeDao. queryEmployee ();

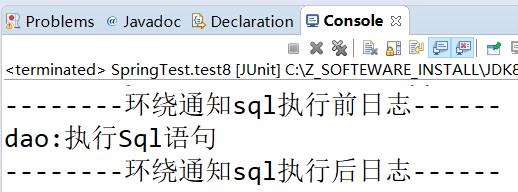
}

@Test

**public** **void** test5(){

employeeDao. round ();

}



# Spring的数据库操作

Spring操作数据库提供了很多持久层技术的模板类，用来简化编程

|  |  |
| --- | --- |
| ORM持久化技术 | 模板类 |
| JDBC | Org.springframework.jdbc.core.JdbcTemplate |
| Hibernate3.0 | Org.springframework.orm.hibernate3.HibernateTemplate |
| Ibatis(Mybatis) | Org.springframework.orm.ibatis.SqlMapClientTemplate |
| JPA | Org.springframework.orm.jpa.JpaTemplate |

## spring集成jdbc概述

* Spring提供了一个工具类JdbcTemplate，该类对jdbc的操作进行了轻量级别的封装。

**优点如下：**

* 直接使用sql语句操作数据库，效率很高。
* 支持数据库的分区，这是数据量大的项目的实现方案，hibernate无法实现。
* 使用java语言拼sql语句，效率很高，这是ibatis无法达到的。

**缺点如下：**

* sql语句直接写在java程序中，很难管理，但部分企业进行了封装，实现了sql语句的简单管理。
* 编码量大。

JdbcTemplate简介

JdbcTemplate封装了操作数据库的各种方法，改类包含一个dataSource[数据源]属性，只有在初始化数据源以后才能使用JdbcTemplate。

## 初始化JdbcTemplate

### 使用spring内置连接池

【导入spring的相关方jar包】



一：使用代码初始化数据源

@Test

**public** **void** test10() {

// spring内置连接池

DriverManagerDataSource dataSource = **new** DriverManagerDataSource();

dataSource.setDriverClassName("oracle.jdbc.OracleDriver");

dataSource.setUrl("jdbc:oracle:thin:@127.0.0.1:1521:orcl");

dataSource.setUsername("scott");

dataSource.setPassword("tiger");

JdbcTemplate jdbcTemplate = **new** JdbcTemplate();

jdbcTemplate.setDataSource(dataSource);

jdbcTemplate.update(

"insert into employees (employee\_id,last\_name,email,hire\_date,job\_id) values(207,'小卡','15810210702@163.com',sysdate,'IT\_PROG')");

}

**测试**

数据已经添加到数据库中



二：使用IoC初始化数据源[将数据库连接池交给spring管理]

<?xml version=*"1.0"* encoding=*"UTF-8"*?>

<beans xmlns=*"http://www.springframework.org/schema/beans"*

xmlns:xsi=*"*[*ttp://www.w3.org/2001/XMLSchema-instance"*](ttp://www.w3.org/2001/XMLSchema-instance%22)

xmlns:context=*"http://www.springframework.org/schema/context"*

xmlns:aop=*"http://www.springframework.org/schema/aop"*

xsi:schemaLocation=*"*

*http://www.springframework.org/schema/beans*

*http://www.springframework.org/schema/beans/spring-beans.xsd*

*http://www.springframework.org/schema/context*

*http://www.springframework.org/schema/context/spring-context.xsd*

*http://www.springframework.org/schema/aop*

*http://www.springframework.org/schema/aop/spring-aop.xsd"*>

<!-- 使用spring内置连接池 -->

<bean id=*"dataSource"*

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

<property name=*"driverClassName"* value=*"oracle.jdbc.OracleDriver"* />

<property name=*"url"* value=*"jdbc:oracle:thin:@127.0.0.1:1521:orcl"* />

<property name=*"username"* value=*"scott"* />

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

</bean>

<bean id=*"jdbcTemplate"* class=*"org.springframework.jdbc.core.JdbcTemplate"*>

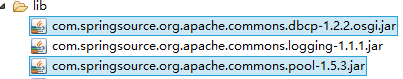
<property name=*"dataSource"* ref=*"dataSource"* />

</bean>

</beans>

### 配置DBCP连接池

【引入dbcp连接池的jar包】



【配置连接池】

<!-- 配置DBCP连接池 -->

<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource">

<property name="driverClassName" value="oracle.jdbc.OracleDriver"/>

<property name="url" value="jdbc:oracle:thin:@127.0.0.1:1521:orcl"/>

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

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

</bean>

### 配置C3P0连接池

【引入相应的jar包】

com.springsource.com.mchange.v2.c3p0-0.9.1.2.jar



【配置连接池】

<!-- 配置C3P0连接池 -->

<bean id="dataSource" class="com.mchange.v2.c3p0.ComboPooledDataSource">

<property name="driverClass" value="oracle.jdbc.OracleDriver"/>

<property name="jdbcUrl" value=" jdbc:oracle:thin:@127.0.0.1:1521:orcl"/>

<property name="user" value="scott"/>

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

</bean>

## JdbcTemplate的CRUD操作

JdbcTemplate的增，删，修全部使用update函数，其有重载的可变参数用于传入参数

查询使用query\*()函数

// 增加

@Test

**public** **void** test12() {

String sql = "insert into employees (employee\_id,last\_name,email,hire\_date,job\_id) values(209,'王小卡','1049191300@qq.com',sysdate,'IT\_PROG')";

jdbcTemplate.update(sql);

}



// 删除

@Test

**public** **void** test13() {

String sql = "delete from employees where employee\_id=209";

jdbcTemplate.update(sql);

}

// 修改

@Test

**public** **void** test14() {

String sql = "update employees set last\_name='王小卡' where employee\_id=207";

jdbcTemplate.update(sql);

}

// 查询[返回一条记录]

@Test

**public** **void** test15() {

String sql = "SELECT \* FROM employees WHERE employee\_id=?";

Employees employees = jdbcTemplate.queryForObject(sql, **new** RowMapper<Employees>() {

@Override

**public** Employees mapRow(ResultSet rs, **int** index) **throws** SQLException {

System.***out***.println(index);

**int** employee\_id = rs.getInt("employee\_id");

String last\_name = rs.getString("last\_name");

**return** **new** Employees(employee\_id, last\_name);

}

}, 207);

System.***out***.println(employees);

}

// 查询[返回多条记录]

@Test

**public** **void** test16() {

String sql = "SELECT \* FROM employees WHERE employee\_id=? or employee\_id=?";

// query函数自动把每条结果添加到了list集合当中

List<Employees> list = jdbcTemplate.query(sql, **new** RowMapper<Employees>() {

@Override

**public** Employees mapRow(ResultSet rs, **int** index) **throws** SQLException {

// index:迭代的行号

System.***out***.println(index);

//rs:已经迭代到的行，所以直接取值就可以

**int** employee\_id = rs.getInt("employee\_id");

String last\_name = rs.getString("last\_name");

**return** **new** Employees(employee\_id, last\_name);

}

},207,208);

System.***out***.println(list);

}

## Spring的事物控制

### 事物概述

* 数据库的事物：

事务是一组操作的执行单元，相对于数据库操作来讲，事务管理的是一组SQL指令，比如增加，修改，删除等。事务的一致性，要求，这个事务内的操作必须全部执行成功，如果在此过程种出现了差错，比如有一条SQL语句没有执行成功，那么这一组操作都将全部回滚

* 事物的四大特性：
* Atomic (原子性) :强调事物的不可分割[要么一起成功，要么一起失败]。
* consistent(一致性) :事物的执行前后数据的完整性保持一致。
* Isolate (隔离性) :一个事物执行的过程中，不应该受到其它事物的干扰。
* Durable (持久性) :一旦事物结束，数据就永久保存到数据库
* 安全问题：

如果不考虑事物的隔离性，会引发如下安全问题

脏读 :一个事务读到了另一个事务的未提交的数据

不可重复读 :一个事务读到了另一个事务已经提交的update的数据导致多次查询结果不一致.

幻/虚读 :一个事务读到了另一个事务已经提交的insert的数据导致多次查询结果不一致.

解决方案：设置事物的隔离级别

|  |  |
| --- | --- |
| 隔离级别 | 含义 |
| DEFAULT | 使用后端数据库默认的隔离级别(spring中的的选择项) |
| READ\_UNCOMMITED  未提交读 | 允许你读取还未提交的改变了的数据。可能导致脏、幻、不可重复读 |
| READ\_COMMITTED  已提交读 | 允许在并发事务已经提交后读取。可防止脏读，但幻读和 不可重复读仍可发生 |
| REPEATABLE\_READ  可重复读取 | 对相同字段的多次读取是一致的，除非数据被事务本身改变。可防止脏、不可重复读，但幻读仍可能发生。 |
| SERIALIZABLE  串行化 | 完全服从ACID的隔离级别，确保不发生脏、幻、不可重复读。这在所有的隔离级别中是最慢的，它是典型的通过完全锁定在事务中涉及的数据表来完成的。 |

* 事物的传播策略：

如两个bean组件都由spring控制事物，且组件的函数之间存在调用关系，即（bean1 函数a 调用了 bean2 函数b），spring提供了一组配置方式供开发者选择，这些配置方式称为事物的传播策略。

事务的传播策略，见下表

|  |  |
| --- | --- |
| **传播行为** | **意义** |
| **REQUIRED** | 业务方法需要在一个事务中运行。如果方法运行时，已经处在一个事务中，那么加入到该事务，否则为自己创建一个新的事务 |
| **NOT\_SUPPORTED** | 声明方法不需要事务。如果方法没有关联到一个事务，容器不会为它开启事务。如果方法在一个事务中被调用，该事务会被挂起，在方法调用结束后，原先的事务便会恢复执行 |
| **REQUIRESNEW** | 属性表明不管是否存在事务，业务方法总会为自己发起一个新的事务。如果方法已经运行在一个事务中，则原有事务会被挂起，新的事务会被创建，直到方法执行结束，新事务才算结束，原先的事务才会恢复执行 |
| **MANDATORY** | 该属性指定业务方法只能在一个已经存在的事务中执行，业务方法不能发起自己的事务。如果业务方法在没有事务的环境下调用，容器就会抛出例外。 |
| **SUPPORTS** | 这一事务属性表明，如果业务方法在某个事务范围内被调用，则方法成为该事务的一部分。如果业务方法在事务范围外被调用，则方法在没有事务的环境下执行 |
| **Never** | 指定业务方法绝对不能在事务范围内执行。如果业务方法在某个事务中执行，容器会抛出例外，只有业务方法没有关联到任何事务，才能正常执行 |
| **NESTED** | 如果一个活动的事务存在，则运行在一个嵌套的事务中. 如果没有活动事务, 则按REQUIRED属性执行.它使用了一个单独的事务， 这个事务拥有多个可以回滚的保存点。内部事务的回滚不会对外部事务造成影响。它只对DataSourceTransactionManager事务管理器起效 |

* 只读事物与读写事物：

与所访问的数据库以及数据库驱动程序相关，并不一定是一个强制选项（例如在只读事物中去更新事物时允许的）。但若在事物中声明了只读事物，将会暗示数据库驱动程序和数据库系统，这个事务并不包含更改数据的操作，那么驱动程序和数据库就有可能根据这种情况对该事务进行一些特定的优化，比方说不安排相应的数据库锁，不记录回滚日志等，以减轻事务对数据库的压力，毕竟事务也是要消耗数据库的资源的。

* 使用场景：

只读事物：单纯的数据库查询

读写事物：对数据进行修改的操作。

* Spring提供了两种事务管理方式
* **编程式事务管理**

通过程序代码来控制你的事务何时开始，何时结束等，结果是控制的颗粒度更细，但需要手写程序，另外在团队合作开发时，会出现事物管理混乱。【了解】

* **声明式事务管理**

在Spring中，你只需要在Spring配置文件中做一些配置，即可将数据库的访问纳入到事务管理中，解除了和代码的耦合， 这是对应用代码影响最小的选择。当你不需要事务管理的时候，可以直接从Spring配置文件中移除该设置

* spring的事务管理器：

spring没有直接管理事务，只是开发了事物管理器调用第三方组件完成事物控制。

* Spring控制事物的方式：

spring控制事物是以bean组件的函数为单位的，如果一个函数正常执行完毕，该函数内的全部数据库操作按照一次事物提交，如果抛出异常，全部回滚。

|  |  |
| --- | --- |
| 事务管理器实现 | 目标 |
| org.springframework.jdbc.datasource.**DataSourceTransactionManager** | 在单一的JDBC Datasource中的管理事务 |
| org.springframework.orm.hibernate3.  **HibernateTransactionManager** | 当持久化机制是hibernate时，用它来管理事务 |
| org.springframework.jdo.  **JdoTransactionManager** | 当持久化机制是Jdo时，用它来管理事务。 |
| org.springframework.transaction.jta.  **JtaTransactionManager** | 使用一个JTA实现来管理事务。在一个事务跨越多个资源时必须使用 |
| org.springframework.orm.ojb.  **PersistenceBrokerTransactionManager** | 当apache的ojb用作持久化机制时，用它来管理事务。 |

### 示例代码

#### 搭建转账的环境

**创建业务层和Dao类**

**public** **class** AccountService {

**private** AccountDao accountDao;

//setter方法...略

**public** **void** transfer(String src,String des,**double** money){

accountDao.minMoney(src, money);

accountDao.addMoney(des, money);

}

}

**public** **class** AccountDao **extends** JdbcDaoSupport {

**public** **void** minMoney(String src, Double money) {

String sql = "UPDATE EMPLOYEES SET SALARY = SALARY-? WHERE EMPLOYEE\_ID = ?";

**this**.getJdbcTemplate().update(sql, money, src);

}

**public** **void** addMoney(String des,Double money){

String sql = "UPDATE EMPLOYEES SET SALARY = SALARY+? WHERE EMPLOYEE\_ID = ?";

**this**.getJdbcTemplate().update(sql, money, des);

}

}

JdbcDaoSupport类是spring提供给开发者用于简化开发的。因为它里边封装了JdbcTemplate，这样开发者不需要再自己声明JdbcTemplate，只需要继承JdbcDaoSupport这个类即可。

**二配置文件applicationContext.xml**

<?xml version=*"1.0"* encoding=*"UTF-8"*?>

<beans xmlns=*"http://www.springframework.org/schema/beans"*

xmlns:xsi=*"*[*ttp://www.w3.org/2001/XMLSchema-instance"*](ttp://www.w3.org/2001/XMLSchema-instance%22)

xmlns:context=*"http://www.springframework.org/schema/context"*

xmlns:aop=*"http://www.springframework.org/schema/aop"*

xsi:schemaLocation=*"*

*http://www.springframework.org/schema/beans*

*http://www.springframework.org/schema/beans/spring-beans.xsd*

*http://www.springframework.org/schema/context*

*http://www.springframework.org/schema/context/spring-context.xsd*

*http://www.springframework.org/schema/aop*

*http://www.springframework.org/schema/aop/spring-aop.xsd"*>

<!-- 使用spring内置连接池 -->

<bean id=*"dataSource"*

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

<property name=*"driverClassName"* value=*"oracle.jdbc.OracleDriver"* />

<property name=*"url"* value=*"jdbc:oracle:thin:@127.0.0.1:1521:orcl"* />

<property name=*"username"* value=*"scott"* />

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

</bean>

<bean id=*"accountDao"* class=*"springtest.xiaoka.dao.AccountDao"*>

<!-- 该属性是继承自JdbcDaoSupport类 -->

<property name=*"dataSource"* ref=*"dataSource"* />

</bean>

<bean id=*"accountService"* class=*"springtest.xiaoka.service.AccountService"*>

<property name=*"accountDao"* ref=*"accountDao"* />

<property name=*"transactionTemplate"* ref=*"transactionTemplate"* />

</bean>

</beans>

**测试**

@RunWith(SpringJUnit4ClassRunner.**class**)

@ContextConfiguration("classpath:applicationContext.xml")

**public** **class** SpringTest2 {

@Resource(name = "accountService")

**private** AccountService accountService;

@Test

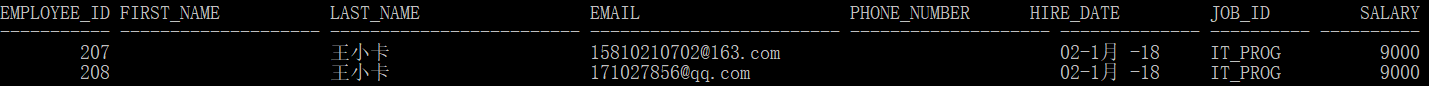
**public** **void** test17() {

accountService.transfer("207", "208", 1000);

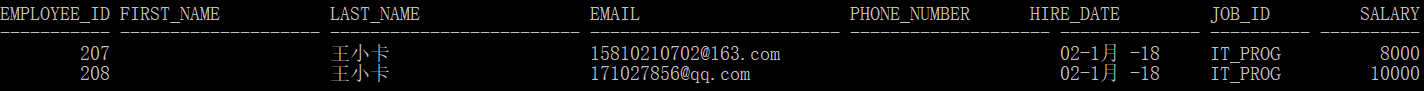
}

}

运行测试前，207和208号员工工资都是9000.



**运行测试代码以后207号减去1000，208号增加1000**



**然后，在service的转账方法中加入异常信息**

**public** **class** AccountService {

**private** AccountDao accountDao;

//setter方法...略

**public** **void** transfer(String src,String des,**double** money){

accountDao.minMoney(src, money);

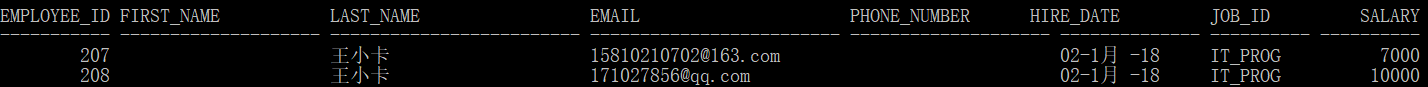
**int** k = 10/0;//模拟异常信息

accountDao.addMoney(des, money);

}

}

再次执行测试代码，可以发现207号员工扣除了1000，但是208号员工并没有相应的收到1000元



**至此，转账环境代码搭建完毕**

#### 一：编程式事物控制【了解】

**配置事务管理器**

<!-- 配置事物管理器 -->

<bean id=*"transactionManager"*

class=*"org.springframework.jdbc.datasource.DataSourceTransactionManager"*>

<property name=*"dataSource"* ref=*"dataSource"* />

</bean>

**配置事物管理模板**

<bean id=*"transactionTemplate"*

class=*"org.springframework.transaction.support.TransactionTemplate"*>

<property name=*"transactionManager"* ref=*"transactionManager"* />

</bean>

**在业务层注入事物管理模板**

<bean id=*"accountService"* class=*"springtest.xiaoka.service.AccountService"*>

<property name=*"accountDao"* ref=*"accountDao"* />

<property name=*"transactionTemplate"* ref=*"transactionTemplate"* />

</bean>

**手动编写代码实现事物管理**

**public** **class** AccountService {

**private** AccountDao accountDao;

**private** TransactionTemplate transactionTemplate;

//setter方法...略

**public** **void** transfer(String src,String des,**double** money){

transactionTemplate.execute(**new** TransactionCallbackWithoutResult() {

@Override

**protected** **void** doInTransactionWithoutResult(TransactionStatus arg0) {

accountDao.minMoney(src, money);

**int** k = 10/0;//模拟异常信息

accountDao.addMoney(des, money);

}

});

}

**测试**

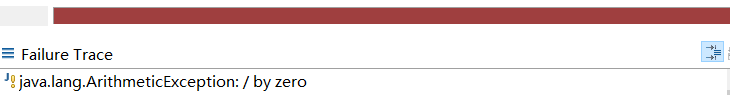
@Test

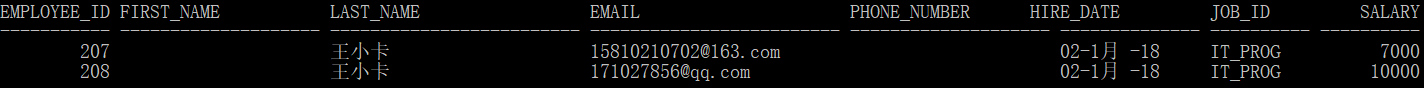
**public** **void** test17() {

accountService.transfer("207", "208", 1000);

}

收到异常信息by zero,事物回滚，207未减去1000，208也未增加1000.





#### 二：声明式事物控制

* 使用spring声明式事物控制分为如下几步：
* 在配置文件头信息上增加tx命名空间与tx.xsd：声明式事物控制需要引入tx标签，因此要引入此xsd文件。
* 配置事物控制管理器：jdbc的事物控制管理器为DataSourceTransactionManager
* 配置事物通知：配置哪些函数委托spring进行事物管理，以及事物管理的隔离级别、传播行为、是否只读事物属性。建议大家在需要更新数据的函数上配置隔离级别为数据库默认级别，传播行为采用required，读写事物。而只是查询的函数使用只读事物，效率更高
* 配置事物的切入点：也就是spring的哪些组件要配置事物通知。

在spring的三层架构中，建议把事物控制放在service层。

配置文件applicationContext.xml

<beans xmlns=*"http://www.springframework.org/schema/beans"*

xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"* xmlns:context=*"http://www.springframework.org/schema/context"*

xmlns:aop=*"http://www.springframework.org/schema/aop"*

xmlns:tx=*"http://www.springframework.org/schema/tx"*

xsi:schemaLocation=*"*

*http://www.springframework.org/schema/beans*

*http://www.springframework.org/schema/beans/spring-beans.xsd*

*http://www.springframework.org/schema/context*

*http://www.springframework.org/schema/context/spring-context.xsd*

*http://www.springframework.org/schema/aop*

*http://www.springframework.org/schema/aop/spring-aop.xsd*

*http://www.springframework.org/schema/tx*

*http://www.springframework.org/schema/tx/spring-tx.xsd"*>

<!-- 使用spring内置连接池 -->

<bean id=*"dataSource"*

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

<property name=*"driverClassName"* value=*"oracle.jdbc.OracleDriver"* />

<property name=*"url"* value=*"jdbc:oracle:thin:@127.0.0.1:1521:orcl"* />

<property name=*"username"* value=*"scott"* />

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

</bean>

<!-- 配置事物管理器 -->

<bean id=*"transactionManager"*

class=*"org.springframework.jdbc.datasource.DataSourceTransactionManager"*>

<property name=*"dataSource"* ref=*"dataSource"* />

</bean>

<!-- 配置事物通知，它相当于AOP的切面类-->

<tx:advice id=*"txAdvice"* transaction-manager=*"transactionManager"*>

<tx:attributes>

<!-- 当执行transferXML函数是，发起事物：默认隔离级别，传播策略是REQUIRED，读写事物 -->

<tx:method name=*"transferXML"* isolation=*"DEFAULT"*

propagation=*"REQUIRED"* read-only=*"false"* />

</tx:attributes>

</tx:advice>

<aop:config>

<!-- 配置切入点 -->

<aop:pointcut expression=*"execution(void springtest..AccountService.\*(..))"*

id=*"txPointCut"* />

<!-- 配置通知 -->

<aop:advisor advice-ref=*"txAdvice"* pointcut-ref=*"txPointCut"* />

</aop:config>

<bean id=*"accountDao"* class=*"springtest.xiaoka.dao.AccountDao"*>

<property name=*"dataSource"* ref=*"dataSource"* />

</bean>

<bean id=*"accountService"* class=*"springtest.xiaoka.service.AccountService"*>

<property name=*"accountDao"* ref=*"accountDao"* />

</bean>

</beans>

**二：Service类**

**public** **class** AccountService {

**private** AccountDao accountDao;

**public** **void** transferXML(String src,String des,**double** money){

accountDao.minMoney(src, money);

**int** k = 10/0;//模拟异常信息

accountDao.addMoney(des, money);

}

}

**三：测试**

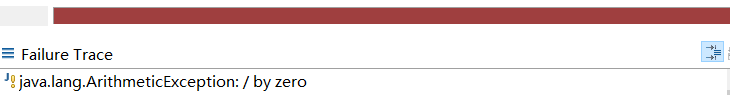
@Test

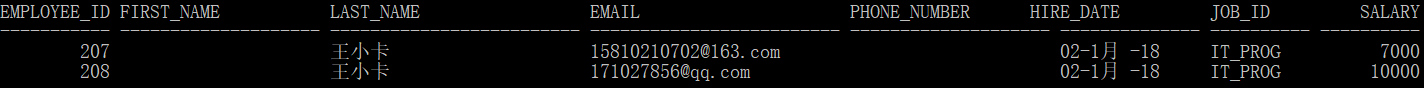
**public** **void** test18() {

accountService.transferXML("100", "101", 1000);

}

收到异常信息by zero,事物回滚，207未减去1000，208也未增加1000.





#### 三：使用注解完成事物控制【了解】

日常开发中很少使用注解来完成事务控制，因为它的代码量太大了且容易书写错误疏漏等。

**一：在配置文件中配置事物管理器并开启事物管理注解**

<?xml version=*"1.0"* encoding=*"UTF-8"*?>

<beans xmlns=*"http://www.springframework.org/schema/beans"*

xmlns:xsi=*"*[*ttp://www.w3.org/2001/XMLSchema-instance"*](ttp://www.w3.org/2001/XMLSchema-instance%22)

xmlns:context=*"http://www.springframework.org/schema/context"*

xmlns:aop=*"http://www.springframework.org/schema/aop"*

xmlns:tx=*"http://www.springframework.org/schema/tx"*

xsi:schemaLocation=*"*

*http://www.springframework.org/schema/beans*

*http://www.springframework.org/schema/beans/spring-beans.xsd*

*http://www.springframework.org/schema/context*

*http://www.springframework.org/schema/context/spring-context.xsd*

*http://www.springframework.org/schema/aop*

*http://www.springframework.org/schema/aop/spring-aop.xsd*

*http://www.springframework.org/schema/tx*

*http://www.springframework.org/schema/tx/spring-tx.xsd"*>

<!-- 使用spring内置连接池 -->

<bean id=*"dataSource"*

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

<property name=*"driverClassName"* value=*"oracle.jdbc.OracleDriver"* />

<property name=*"url"* value=*"jdbc:oracle:thin:@127.0.0.1:1521:orcl"* />

<property name=*"username"* value=*"scott"* />

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

</bean>

<!-- 配置事物管理器 -->

<bean id=*"transactionManager"*

class=*"org.springframework.jdbc.datasource.DataSourceTransactionManager"*>

<property name=*"dataSource"* ref=*"dataSource"* />

</bean>

<!-- 开启注解事物管理 -->

<tx:annotation-driven transaction-manager=*"transactionManager"* />

<bean id=*"accountDao"* class=*"springtest.xiaoka.dao.AccountDao"*>

<property name=*"dataSource"* ref=*"dataSource"* />

</bean>

<bean id=*"accountService"* class=*"springtest.xiaoka.service.AccountService"*>

<property name=*"accountDao"* ref=*"accountDao"* />

</bean>

</beans>

**二：添加注解@ Transactional**

* **添加到类上，将作用于该类所有的方法**
* **添加到方法上，只作用于该方法**

@Transactional(isolation = Isolation.***DEFAULT***, propagation = Propagation.***REQUIRED***, readOnly = **true**)

**public** **class** AccountService {

**private** AccountDao accountDao;

**public** **void** transferXML(String src,String des,**double** money){

accountDao.minMoney(src, money);

**int** k = 10/0;//模拟异常信息

accountDao.addMoney(des, money);

}

**public** **void** setAccountDao(AccountDao accountDao) {

**this**.accountDao = accountDao;

}

}

**public** **class** AccountDao **extends** JdbcDaoSupport {

**public** **void** minMoney(String src, Double money) {

String sql = "UPDATE EMPLOYEES SET SALARY = SALARY-? WHERE EMPLOYEE\_ID = ?";

**this**.getJdbcTemplate().update(sql, money, src);

System.***out***.println("----减钱成功---");

}

**public** **void** addMoney(String des,Double money){

String sql = "UPDATE EMPLOYEES SET SALARY = SALARY+? WHERE EMPLOYEE\_ID = ?";

**this**.getJdbcTemplate().update(sql, money, des);

}

}

**三：测试**

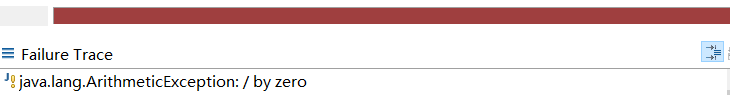
@Test

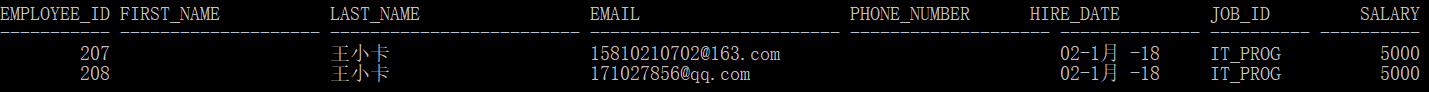
**public** **void** test18() {

accountService.transferXML("100", "101", 1000);

}

收到异常信息by zero,事物回滚，207未减去1000，208也未增加1000.





# 附录

### 配置文件的引入

当一个配置文件内容过多过大的时候不方便维护管理,这个时候可以考虑”分块”管理--写成多个配置文件,然后在主配置文件中引入其它配置文件。

<import resource=*"其它配置文件.xml"*/>

也可以在创建对象的时候指定引入多个配置文件

@Test

**public** **void** test11() {

//spring提供了可变参数的构造方法,它可以引入多个配置文件

ApplicationContext context = **new** ClassPathXmlApplicationContext("applicationContext.xml","applicationContext1.xml");

System.***out***.println(context);

}

### Log4j

Log4j是一个日志输出管理的包,它通过配置文件的方式来管理日志的输出,方便了程序的测试和开发

**一:所需要的jar包**



第一个org.apache.commons.logging-1.1.1.jar是日志输出规范

第二个com.springsource.org.apache.log4j-1.2.15.jar是日志输出实现

**二:配置文件**

log4j默认使用log4j.properties属性文件,把它放在类路径下即可.

### direct log messages to stdout ###

log4j.appender.stdout=org.apache.log4j.ConsoleAppender

log4j.appender.stdout.Target=System.err

log4j.appender.stdout.layout=org.apache.log4j.PatternLayout

log4j.appender.stdout.layout.ConversionPattern=%d{ABSOLUTE} %5p %c{1}:%L - %m%n

### direct messages to file mylog.log ###

log4j.appender.file=org.apache.log4j.FileAppender

log4j.appender.file.File=D\:mylog.log

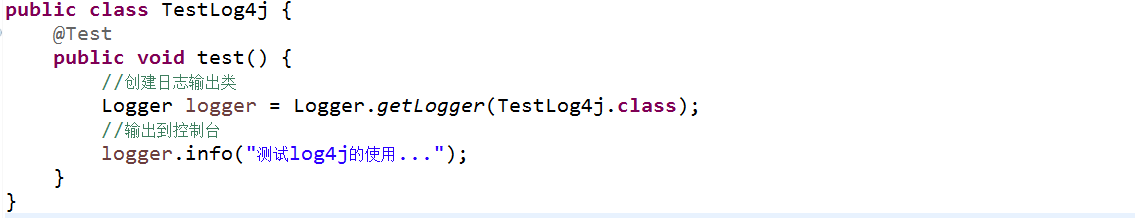
log4j.appender.file.layout=org.apache.log4j.PatternLayout

log4j.appender.file.layout.ConversionPattern=%d{ABSOLUTE} %5p %c{1}:%L - %m%n

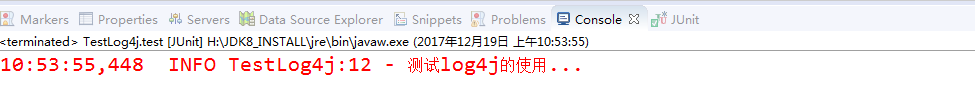
### set log levels - for more verbose logging change 'info' to 'debug' ###

log4j.rootLogger=info, stdout

**三:log4j的代码**



**四:查看结果,发现日志已经输出到控制台**

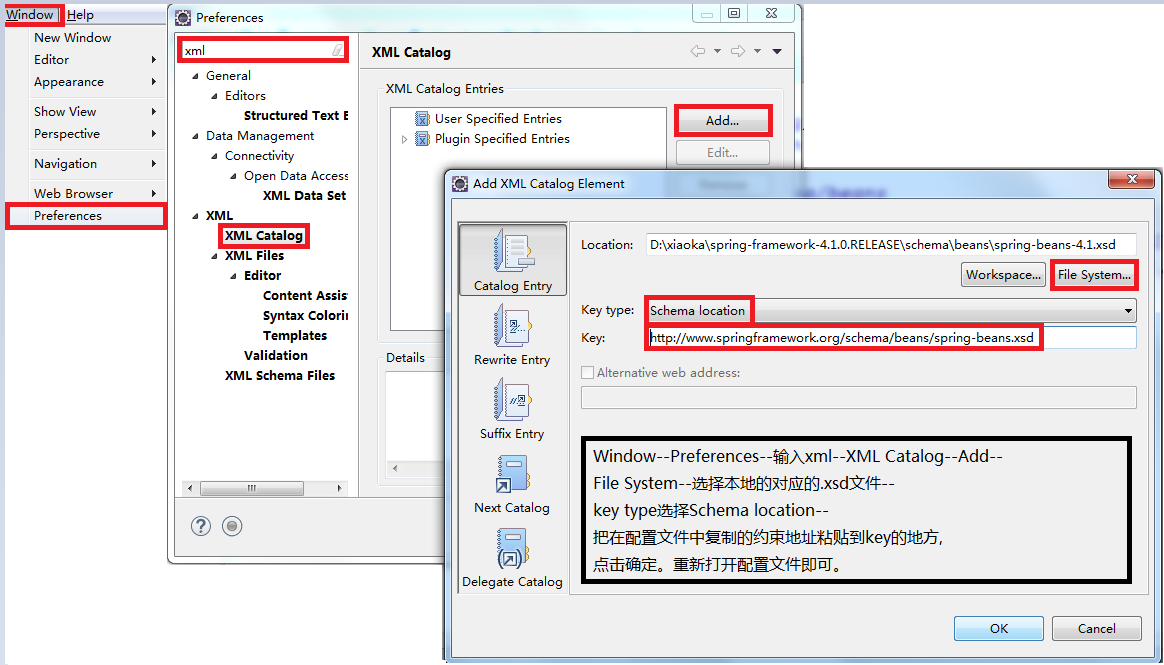


### Spring配置文件编辑不提示

一:复制xml约束中的location地址,比如beans

http://www.springframework.org/schema/beans/spring-beans.xsd

二:图示



### WebApplicationContext

它是一个web层的上下文,可以使用它来加载配置文件,使用下面的代码获取上下文

Web.xml

<listener>

<listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>

</listener>

<context-param>

<param-name>contextConfigLocation</param-name>

<param-value>classpath:applicationContext.xml</param-value>

</context-param>

WebApplicationContext context = WebApplicationContextUtils.*getWebApplicationContext*(ServletContext);

### Spring测试

传统的junit测试步骤太繁琐，为了方便测试，spring提供了对junit测试的整合，用来方便程序的测试

一：导入junit测试包

二：导入spring-test.jar包



三：创建测试类，在改测试类上添加如下注解即可

**package** com.xiaoka.test;

//这个类用来运行测试类

@RunWith(SpringJUnit4ClassRunner.**class**)

//加载类路径下的配置文件

@ContextConfiguration("classpath:applicationContext.xml")

**public** **class** SpringTest {

//将需要测试的类通过依赖的方式添加进来即可

@Resource(name="employeeService")

**private** EmployeeService employeeService;

@Test

**public** **void** test1(){

//直接运行junit测试就行

employeeService.findEmployeeBuUserNameAndPwd("小钻风", "大王叫我来巡山");

}

}

### CGLIB动态代理

传统的JDK动态代理要求目标对象实现了接口，而当目标对象没有实现接口的时候，就可以考虑使用CGLIB动态代理来生成代理对象

如果想使用CGLIB的技术来生成代理对象，那么需要引入CGLIB的开发的jar包，在Spring框架核心包中已经引入了CGLIB的开发包了。所以直接引入Spring核心开发包即可！

**public** **class** UserDaoImpl {//没有实现任何接口

**public** **void** saveUser(){

System.***out***.println("---保存用户---");

}

}

@Test

**public** **void** test3() {

//目标对象

UserDaoImpl userDaoImpl = **new** UserDaoImpl();

//创建CGLIB核心对象

Enhancer enhancer = **new** Enhancer();

//设置父类

enhancer.setSuperclass(UserDaoImpl.**class**);

//设置回调函数

enhancer.setCallback(**new** MethodInterceptor() {

@Override

**public** Object intercept(Object obj, Method method, Object[] args, MethodProxy methodProxy)

**throws** Throwable {

if ("saveUser".equals(method.getName())) {

System.out.println("-------代理对象函数执行-----");

}

System.***out***.println("-------代理对象函数执行-----");

**return** methodProxy.invokeSuper(obj, args);

}

});

//生成代理对象

UserDaoImpl serDaoImplPorxy = (UserDaoImpl) enhancer.create();

serDaoImplPorxy.saveUser();

//打印代理对象，可以发现它是CGLIB生成的目标对象的代理对象

System.***out***.println(serDaoImplPorxy);

}

