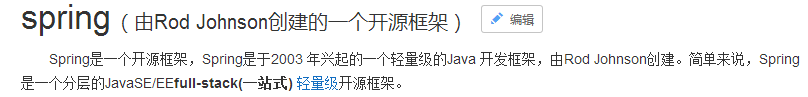
# Spring第一天

# Spring概述

## Spring是什么



那什么是“一站式”框架呢？

JavaEE开发规范规定我们的程序应该要分为三层：WEB层、业务层、持久层。每一层的含的含义如下：

* WEB层（页面数据显示、页面跳转调度）-jsp/servlet
* 业务层（业务处理和功能逻辑、事务控制）-serivice
* 持久层（数据存取和封装、和数据库打交道）-dao

JavaEE三层架构中每一层都有对应的解决方案：WEB层可以采用Struts2；持久层可以采用Hibernate；而业务层是需要处理事务的，利用Spring中的AOP可以很好的管理事务。每一层所对应的框架如下图所示：



那什么是“一站式”呢？“一站式”指的是Spring框架提供了JavaEE三层架构中每一层的解决方案。通过前面的分析我们知道，WEB层可以用Struts2，持久层可以用Hibernate。现在WEB层和持久层也可以采用其他技术，WEB层可以用Spring中的MVC模块，持久层可以用Spring中的JdbcTemplate技术。这样一来的话，JavaEE三层中的架构中的每一层都可以采用Spring来解决，所以我们称Spring是“一站式”框架，意思是采用Spring这一个框架就足够了，Spring给我们提供“一条龙”服务。JavaEE三层每一层都采用Spring来解决的示意图如下：



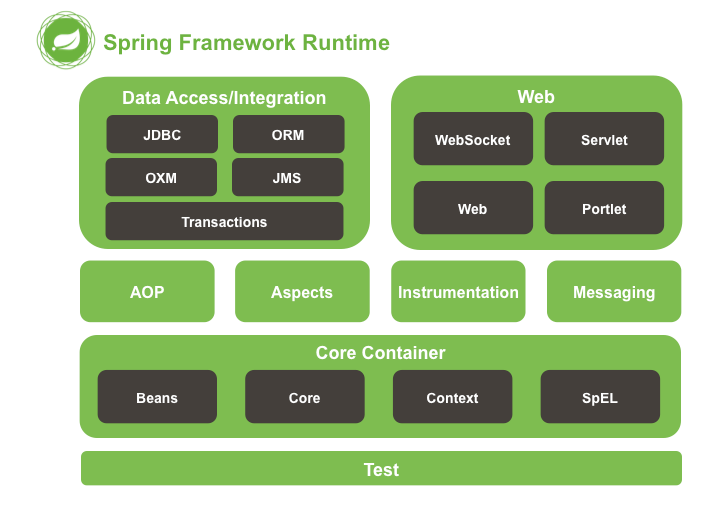
那什么又是“轻量级”呢？“轻量级”指的是Spring的出现取代了EJB的臃肿、低效、繁琐复杂。

**总结一句话：Spring是一个分层的“一站式”轻量级开源框架。**

**Spring“全家桶“**

## Spring体系架构

Spring是一个很庞大的体系，其中包含了很多个模块。Spring完整体系架构图如下：



在本课程中，我们主要学习Spring中的两大核心技术：IoC和AOP：

IoC（Inverse of Control 反转控制）： 将对象创建权利交给Spring工厂进行管理。

AOP（Aspect Oriented Programming 面向切面编程），基于动态代理功能增强。

## Spring的优势

**方便解耦，简化开发**

通过Spring提供的IoC容器，可以将对象间的依赖关系交由Spring进行控制，避免硬编码所造成的过度程序耦合。用户也不必再为单例模式类、属性文件解析等这些很底层的需求编写代码，可以更专注于上层的应用。

**AOP编程的支持**

通过Spring的AOP功能，方便进行面向切面的编程，许多不容易用传统OOP实现的功能可以通过AOP轻松应付。

**声明式事务的支持**

可以将我们从单调烦闷的事务管理代码中解脱出来，通过声明式方式灵活的进行事务的管理，提高开发效率和质量。

**方便程序的测试**

可以用非容器依赖的编程方式进行几乎所有的测试工作，测试不再是昂贵的操作，而是随手可做的事情。

**方便集成各种优秀框架**

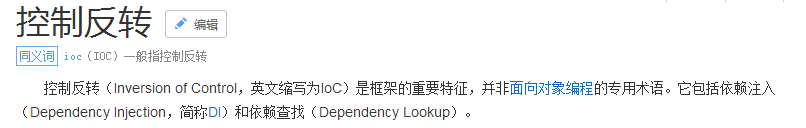
Spring可以降低各种框架的使用难度，提供了对各种优秀框架（Struts、Hibernate、Hessian、Quartz等）的直接支持。

**降低JavaEE API的使用难度**

Spring对JavaEE API（如JDBC、JavaMail、远程调用等）进行了薄薄的封装层，使这些API的使用难度大为降低。

# Spring IOC快速入门

## 什么是IOC



IOC，它是Inverse of Control的缩写，中文含义是**控制反转，表示将对象的创建权力反转给Spring框架！！**

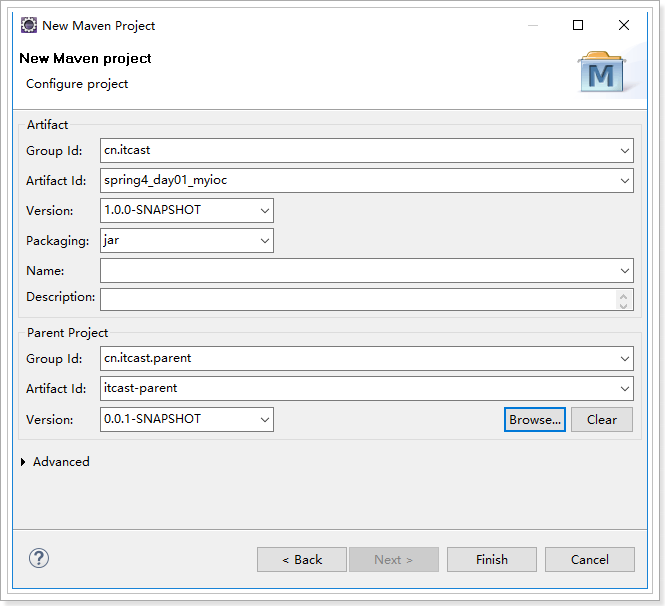
IOC解决的问题：使用IOC可以解决的程序耦合性高的问题！！

那么什么是程序的耦合呢？

## 什么是程序的耦合

我们在开发中，会写很多的类，而有些类之间不可避免的产生依赖关系，这种依赖关系称之为耦合。有些依赖关系是必须的，有些依赖关系可以通过优化代码来解除的。请看下面的示例代码：

* 首先，创建maven工程：spring4\_day01\_myioc



* 因为我们需要做单元测试，所以需要在pom.xml中添加junit的依赖

<project xmlns=*"http://maven.apache.org/POM/4.0.0"* xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*

xsi:schemaLocation=*"http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd"*>

<modelVersion>4.0.0</modelVersion>

<parent>

<groupId>cn.itcast.parent</groupId>

<artifactId>itcast-parent</artifactId>

<version>0.0.1-SNAPSHOT</version>

</parent>

<groupId>cn.itcast</groupId>

<artifactId>spring4\_day01\_myioc</artifactId>

<version>1.0.0-SNAPSHOT</version>

<dependencies>

**<dependency>**

**<groupId>junit</groupId>**

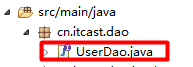
**<artifactId>junit</artifactId>**

**</dependency>**

</dependencies>

</project>

* 创建UserDao接口

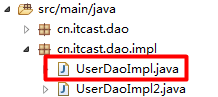


**public** **interface** UserDao {

**public** **void** save();

}

* 创建UserDao接口的实现类UserDaoImpl



**public** **class** UserDaoImpl **implements** UserDao {

@Override

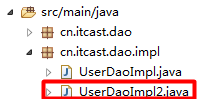
**public** **void** save() {

System.***out***.println("持久层：用户保存...");

}

}

* 创建UserDao接口的第二个实现类UserDaoImpl2



**public** **class** UserDaoImpl2 **implements** UserDao {

@Override

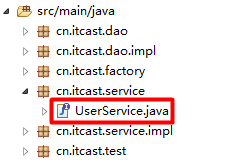
**public** **void** save() {

System.***out***.println("持久层：用户保存22222.....");

}

}

* 创建UserService接口

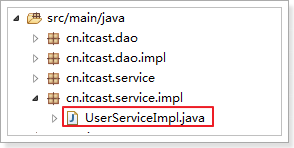


**public** **interface** UserService {

**public** **void** saveUser();

}

* 创建UserService接口的实现类UserServiceImpl



**public** **class** UserServiceImpl **implements** UserService {

**private** UserDao userDao = **new** UserDaoImpl();

@Override

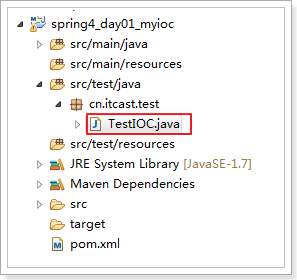
**public** **void** saveUser() {

userDao.save();

}

}

* 创建单元测试类TestIOC,在其中创建test1单元测试方法，测试UserServiceImpl能否调用UserDaoImpl



**public** **class** TestIOC {

@Test

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

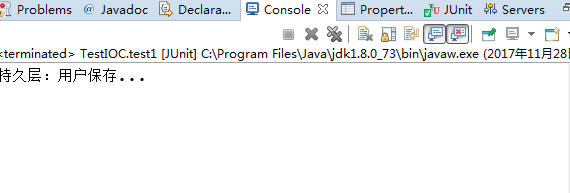
UserService userService = **new** UserServiceImpl();

userService.saveUser();

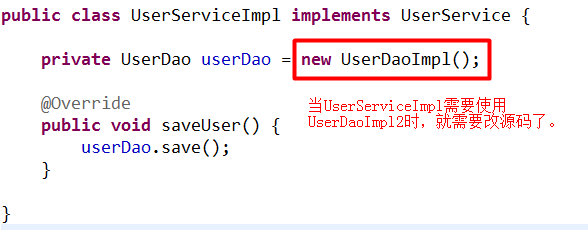
}

}

* 运行test1方法，测试结果如下：



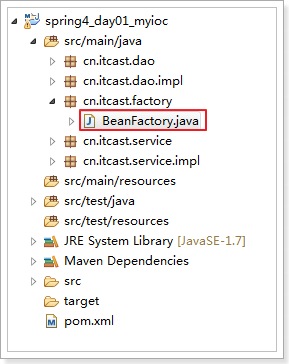
以上程序，UserServiceImpl和UserDaoImpl就耦合了，因为在UserServiceImpl中直接实例化了UserDaoImpl，UserServiceImpl和UserDaoImpl联系太过紧密了。耦合性过高的弊端就在于，当在UserServiceImpl中需要更换为UserDaoImpl2时，就需要改源码了，违背了软件开发领域的“开闭”原则。



那怎么样才能让UserServiceImpl需要更换UserDao的实现时，不用修改UserServiceImpl的源码？也就是让UserServiceImpl和DAO解耦。首先，可以采用工厂的方式来解耦。下面，我们来看下采用工厂的方式该如何解耦？

## 创建工厂类解决耦合性的问题

* 创建工厂类BeanFactory



/\*\*

\* 专门用来生产bean的工厂类

\* **@author** kevin

\*/

**public** **class** BeanFactory {

/\*\*

\* 获取bean对象的工厂方法

\* **@return**

\*/

**public** **static** UserDao getBean(){

**return** **new** UserDaoImpl();

}

}

* 修改UserServiceImpl类，不用再手动实例化DAO了，而是通过工厂类的方式来获取DAO

**public** **class** UserServiceImpl **implements** UserService {

//手动实例DAO，Service严重耦合DAO

//private UserDao userDao = new UserDaoImpl();

//通过工厂的方式来获取DAO对象

**private** UserDao userDao = BeanFactory.*getBean*();

@Override

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

userDao.save();

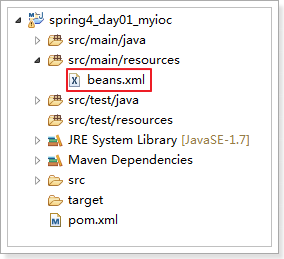
}

}

分析以上的程序我们发现：UserServiceImpl类和具体的UserDao的实现类解耦了，但是工厂类BeanFactory又和具体的UserDao的实现类耦合了，程序并没有实现真正意义上的解耦。那怎么办呢？可以创建配置文件彻底解耦耦合性的问题。

## 创建配置文件彻底解决耦合性的问题

* 在src/main/resources下创建配置文件beans.xml



beans.xml的内容如下：

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

<bean id=*"userDao"* class=*"cn.itcast.dao.impl.UserDaoImpl"*></bean>

* 修改工厂类BeanFactory，从配置文件读取bean信息，读取配置文件时需要用到dom4j,故先引入dom4j的依赖，坐标如下：

<project xmlns=*"http://maven.apache.org/POM/4.0.0"* xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*

xsi:schemaLocation=*"http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd"*>

<modelVersion>4.0.0</modelVersion>

<parent>

<groupId>cn.itcast.parent</groupId>

<artifactId>itcast-parent</artifactId>

<version>0.0.1-SNAPSHOT</version>

</parent>

<groupId>cn.itcast</groupId>

<artifactId>spring4\_day01\_myioc</artifactId>

<version>1.0.0-SNAPSHOT</version>

<dependencies>

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

</dependency>

**<dependency>**

**<groupId>dom4j</groupId>**

**<artifactId>dom4j</artifactId>**

**<version>1.6.1</version>**

**</dependency>**

</dependencies>

</project>

/\*\*

\* 专门用来生产bean的工厂类

\*

\* **@author** kevin

\*/

**public** **class** BeanFactory {

/\*

\* public static UserDao getBean(){ return new UserDaoImpl(); }

\*/

// 从xml文件中解析bean，通过反射得到的对象存放到map中

**private** **static** Map<String, Object> *map* = **new** HashMap<String, Object>();

/\*\*

\* 解析xml文件

\*/

**static** {

SAXReader saxReader = **new** SAXReader();

**try** {

Document document = saxReader.read(BeanFactory.**class**.getClassLoader().getResourceAsStream("beans.xml"));

Element root = document.getRootElement();

String id = root.attributeValue("id");

// 类的全路径名

String clazz = root.attributeValue("class");

// 通过反射生成对象

Object obj = Class.*forName*(clazz).newInstance();

*map*.put(id, obj);

} **catch** (Exception e) {

e.printStackTrace();

}

}

/\*\*

\* 获取bean对象的工厂方法

\*

\* **@return**

\*/

**public** **static** Object getBean(String id) {

**return** *map*.get(id);

}

}

* 修改UserServiceImpl类，改为通过工厂类获取UserDao对象

**public** **class** UserServiceImpl **implements** UserService {

//手动实例DAO，Service严重耦合DAO

//private UserDao userDao = new UserDaoImpl();

//通过工厂的方式来获取DAO对象

**private** UserDao userDao = (UserDao) BeanFactory.*getBean*("userDao");

@Override

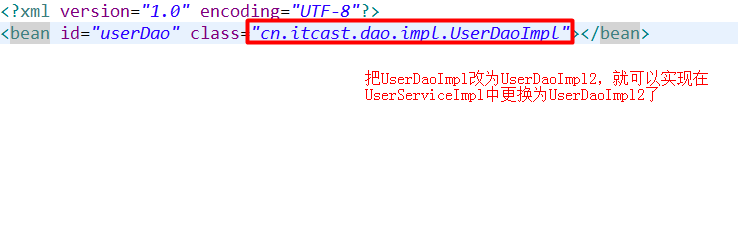
**public** **void** saveUser() {

userDao.save();

}

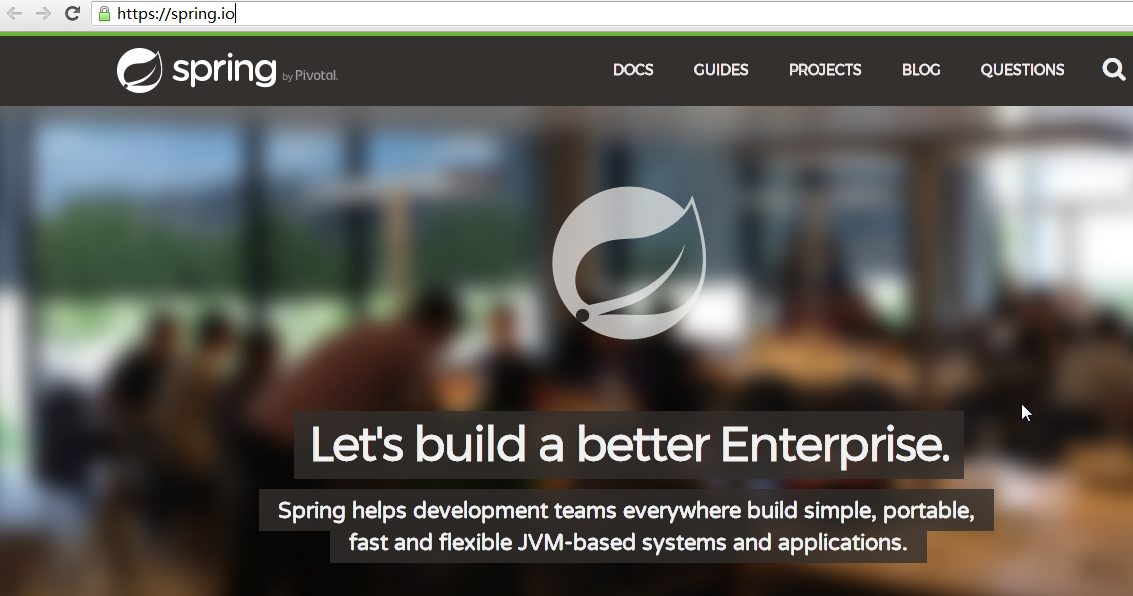
}

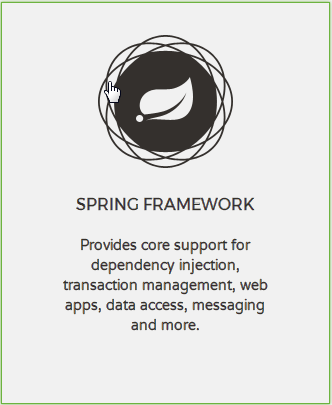
此时，我们的程序已经真正解耦了，当UserServiceImpl需要更换DAO时，只需要修改beans.xml文件即可，不需要改源代码了。



## Spring开发包

* 官网地址：<https://spring.io/>





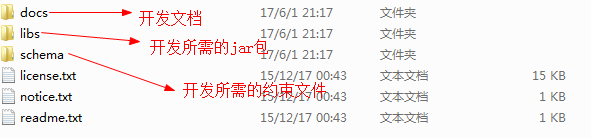
* 下载网址：<http://repo.spring.io/libs-release-local/org/springframework/spring/>



* 本课程采用4.2.4这个版本

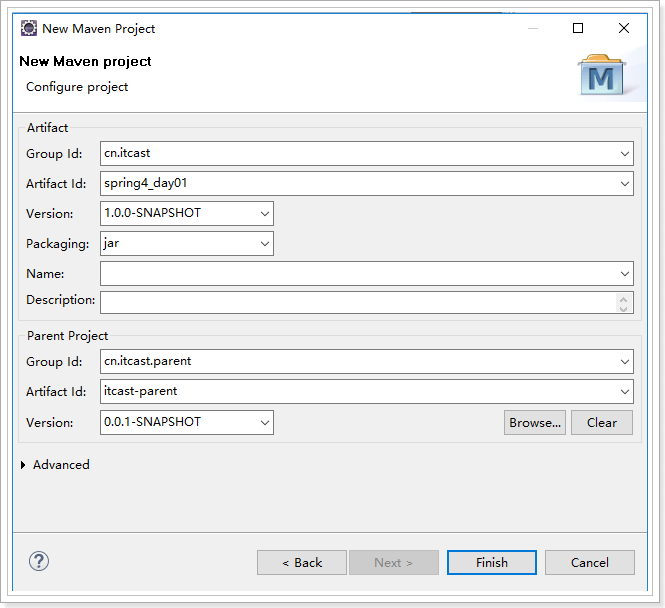


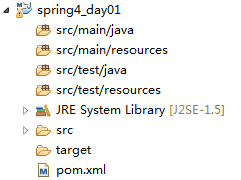
* 把spring-framework.jar包解压，开发包目录结构如下：



## Spring IOC入门

### 创建工程并引入依赖





* 引入spring ioc依赖的jar包，坐标如下：

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-context</artifactId>

</dependency>

* Spring运行还需要日志环境，这里我们引入log4j的jar包，坐标如下：

<dependency>

<groupId>org.slf4j</groupId>

<artifactId>slf4j-log4j12</artifactId>

</dependency>

* 后面还需要做单元测试，引入junit的依赖，坐标如下：

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

</dependency>

* 完整的pom.xml内容如下：

<project xmlns=*"http://maven.apache.org/POM/4.0.0"* xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*

xsi:schemaLocation=*"http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd"*>

<modelVersion>4.0.0</modelVersion>

<parent>

<groupId>cn.itcast.parent</groupId>

<artifactId>itcast-parent</artifactId>

<version>0.0.1-SNAPSHOT</version>

</parent>

<groupId>cn.itcast</groupId>

<artifactId>spring4\_day01</artifactId>

<version>1.0.0-SNAPSHOT</version>

<dependencies>

**<dependency>**

**<groupId>org.springframework</groupId>**

**<artifactId>spring-context</artifactId>**

**</dependency>**

**<dependency>**

**<groupId>org.slf4j</groupId>**

**<artifactId>slf4j-log4j12</artifactId>**

**</dependency>**

**<dependency>**

**<groupId>junit</groupId>**

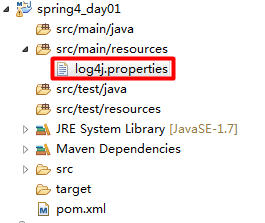
**<artifactId>junit</artifactId>**

**</dependency>**

</dependencies>

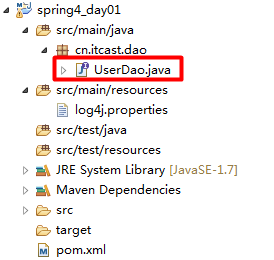
</project>

* Spring框架在运行时需要有日志环境的支持，前面我们已经导入了log4j的jar包，还需要引入log4j的配置文件log4j.properties:



### 创建一个接口和实现类

* 创建UserDao接口：

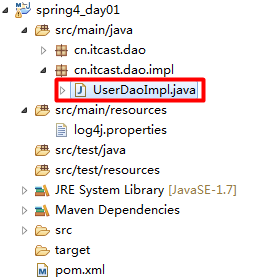


**public** **interface** UserDao {

**public** **void** save();

}

* 创建UserDao接口的实现类UserDaoImpl：



**public** **class** UserDaoImpl **implements** UserDao {

@Override

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

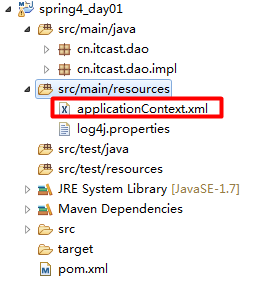
System.***out***.println("持久层：用户保存...");

}

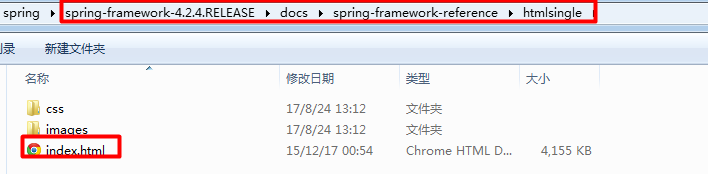
}

### 创建Spring的核心配置文件

* 在src/main/resources下新建Spring核心配置文件applicationContext.xml



* 在applicationContext.xml中引入约束，约束的内容可以从spring的官方文档中拷贝



打开官方文档index.html，找到6.2.1这一节，可以找到约束：



引入约束后的applicationContext.xml内容如下：

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

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

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

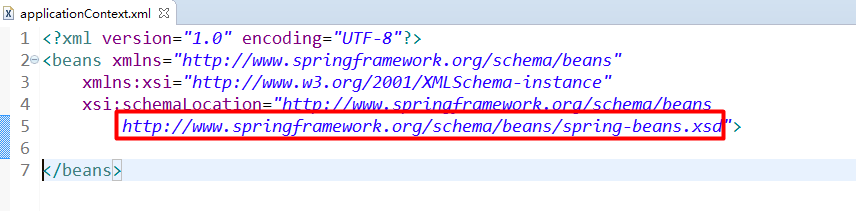
xsi:schemaLocation=*"http://www.springframework.org/schema/beans*

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

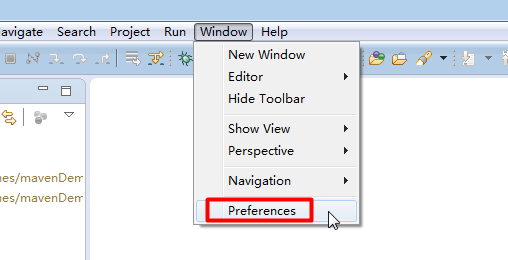
</beans>

* 如何配置XML自动提示？
* 如果电脑可以联网，自动从网上下载 DTD，在编写xml文件时就有了提示。
* 如果电脑没有联网，在本地配置DTD文件位置，提供本地提示。
* 配置XML自动提示步骤：

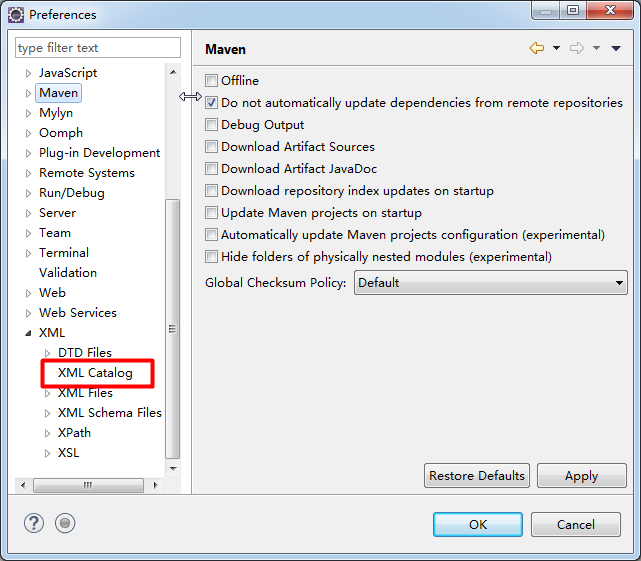
**第一步**：从applicationContext.xml中复制dtd网址：



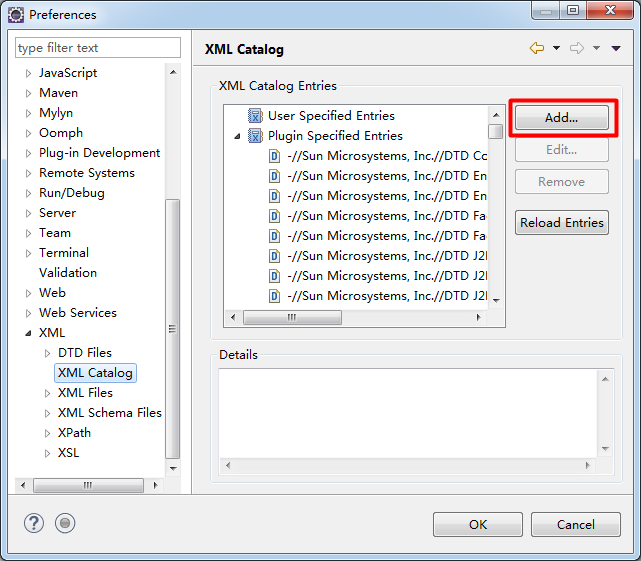
**第二步**：选择window-Preferences

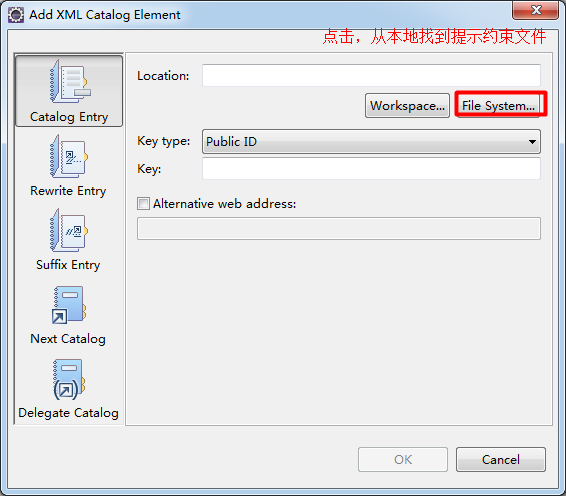


选择Xml Catalog

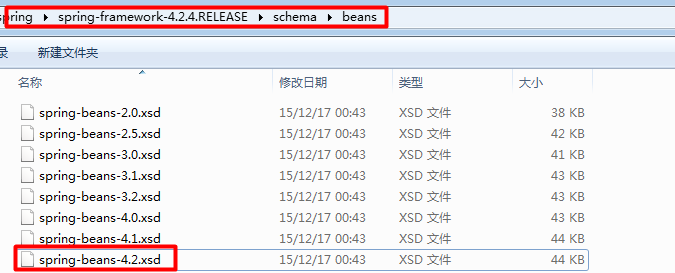


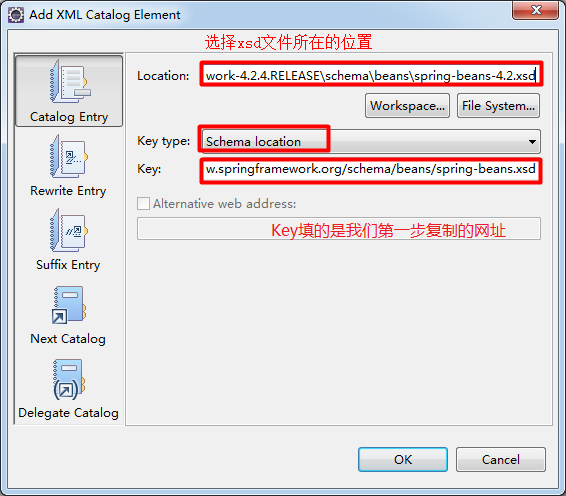
点击“Add”





* 提示约束文件在官方包的schema目录中：





### 配置UserDaoImpl类

在applicationContext.xml配置UserDaoImpl

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

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

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

xsi:schemaLocation=*"http://www.springframework.org/schema/beans*

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

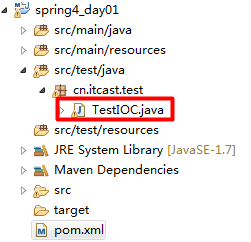
<bean id=*"userDao"* class=*"cn.itcast.dao.impl.UserDaoImpl"*></bean>

</beans>

id属性指定bean的唯一标识；class属性指定bean的全路径名。

### 编写测试类

在src/test/java中创建单元测试类TestIOC



**public** **class** TestIOC {

@Test

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

//创建Spring工厂（创建IOC容器）

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

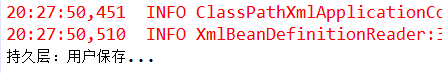
UserDao userDao = (UserDao) ac.getBean("userDao");

userDao.save();

}

}

测试发现：**我们可以从spring容器中获取对象。**



# Spring中的工厂

在spring中提供了两个工厂接口：

1. ApplicationContext
2. BeanFactory

## ApplicationContext接口

* 使用该接口可以获取到具体的Bean对象
* 该接口下有两个具体的实现类

\* **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

下面演示FileSystemXmlApplicationContext的用法：

把src下的applicationContext.xml拷贝到你电脑的某个目录，例如：c:/spring，可以通过FileSystemXmlApplicationContext加载本地磁盘下的spring配置文件

**public** **class** TestIOC {

@Test

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

//创建Spring工厂（创建IOC容器）

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

UserDao userDao = (UserDao) ac.getBean("userDao");

userDao.save();

}

@Test

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

//创建Spring工厂（创建IOC容器）

ApplicationContext ac = **new** FileSystemXmlApplicationContext("C:/spring/applicationContext.xml");

UserDao userDao = (UserDao) ac.getBean("userDao");

userDao.save();

}

}

## BeanFactory工厂

BeanFactory是Spring框架早期的创建Bean对象的工厂接口。在TestIOC中创建test3测试方法：

/\*\*

\* 早期版本的spring工厂：BeanFactory

\*/

@Test

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

BeanFactory factory = **new** ~~XmlBeanFactory~~(**new** ClassPathResource("applicationContext.xml"));

UserDao userDao = (UserDao) factory.getBean("userDao");

userDao.save();

}

* BeanFactory和ApplicationContext的区别  
              \* BeanFactory                -- BeanFactory采取延迟加载，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

修改UserDaoImpl，增加一个无参构造方法，运行test1和test3两个单元测试方法，验证ApplicationContext和BeanFactory这两个工厂到底什么时候创建bean对象？？

**public** **class** UserDaoImpl **implements** UserDao {

**public** UserDaoImpl() {

System.***out***.println("调用了无参构造方法...");

}

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

System.***out***.println("持久层：用户保存...");

}

}

修改test1单元测试方法：

/\*\*

\* 创建ioc容器时就已经把对象创建好了

\*/

@Test

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

//创建Spring工厂（创建IOC容器）

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

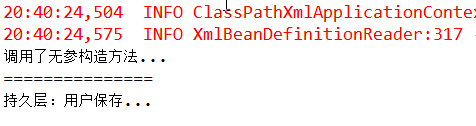
System.***out***.println("===============");

UserDao userDao = (UserDao) ac.getBean("userDao");

userDao.save();

}

运行test1方法，发现：ApplicationContext是在创建ioc容器时就已经把对象创建好了



修改test3单元测试方法：

/\*\*

\* 创建ioc容器时并没有创建对象，而是在第一次getBean时再创建对象

\*/

@Test

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

//创建Spring工厂（创建IOC容器）

BeanFactory factory = **new** ~~XmlBeanFactory~~(**new** ClassPathResource("applicationContext.xml"));

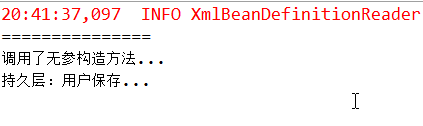
System.***out***.println("===============");

UserDao userDao = (UserDao) factory.getBean("userDao");

userDao.save();

}

运行test3方法，发现：BeanFactory是创建ioc容器时并没有创建对象，而是在第一次getBean时再创建对象



# Spring配置文件

## id属性

id属性是bean的唯一标识

## class属性

bean的全路径名

<bean id=*"userDao"* class=*"cn.itcast.dao.impl.UserDaoImpl"*/>

## scope属性

scope属性代表Bean的作用范围

|  |
| --- |
| singleton:单例（默认值）  prototype:多例，在Spring框架整合Struts2框架的时候，Action类也需要交给Spring做管理，配置把Action类配置成多例！！  request:应用在web工程中，将创建的对象存入到request域中。  session:应用在web工程中，将创建的对象存入到session域中  globalsession:应用在porlet环境下使用。将创建的对象存入到全局的session中。 |

示例：<bean id=*"userDao"* class=*"cn.itcast.dao.impl.UserDaoImpl"* scope=*"prototype"*/>

修改applicationContext.xml，把UserDaoImpl的作用域改为prototype

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

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

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

xsi:schemaLocation=*"http://www.springframework.org/schema/beans*

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

<bean id=*"userDao"* class=*"cn.itcast.dao.impl.UserDaoImpl"* scope=*"prototype"*></bean>

</beans>

修改test1单元测试方法：

@Test

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

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

System.***out***.println("===============");

UserDao userDao = (UserDao) ac.getBean("userDao");

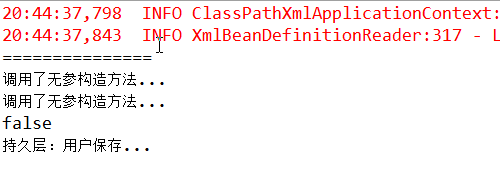
UserDao userDao1 = (UserDao) ac.getBean("userDao");

System.***out***.println(userDao==userDao1);

userDao.save();

}

测试发现：当scope为prototype时，每次获取bean，都会重新实例化



## init-method属性

当bean被载入到容器的时候调用init-method属性指定的方法

修改UserDaoImpl,在其中提供init方法

**public** **class** UserDaoImpl **implements** UserDao {

**public** UserDaoImpl() {

System.***out***.println("调用了无参构造方法...");

}

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

System.***out***.println("调用了init方法...");

}

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

System.***out***.println("持久层：用户保存...");

}

}

修改applicationContext.xml，为UserDaoImpl指定初始化方法

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

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

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

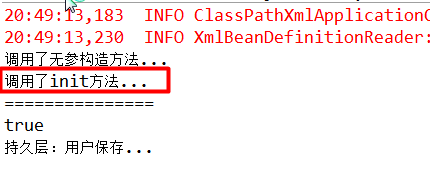
xsi:schemaLocation=*"http://www.springframework.org/schema/beans*

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

<bean id=*"userDao"* class=*"cn.itcast.dao.impl.UserDaoImpl"* init-method=*"init"*></bean>

</beans>

* 运行test1单元测试方法，测试结果：



## destroy-method属性

当bean从容器中删除的时候调用destroy-method属性指定的方法

想查看destroy-method的效果，有如下条件：

scope= singleton有效

web容器中会自动调用，但是main函数或测试用例需要手动调用（需要使用ClassPathXmlApplicationContext的close()方法）

修改UserDaoImpl，在其中提供destroy方法

**public** **class** UserDaoImpl **implements** UserDao {

**public** UserDaoImpl() {

System.***out***.println("调用了无参构造方法...");

}

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

System.***out***.println("调用了init方法...");

}

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

System.***out***.println("持久层：用户保存...");

}

**public** **void** destroy(){

System.***out***.println("调用了销毁方法...");

}

}

在applicationContext.xml中为UserDaoImpl指定销毁方法

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

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

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

xsi:schemaLocation=*"http://www.springframework.org/schema/beans*

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

<bean id=*"userDao"* class=*"cn.itcast.dao.impl.UserDaoImpl"* init-method=*"init"* destroy-method=*"destroy"*></bean>

</beans>

修改test1单元测试方法，显示关闭ioc容器：

@Test

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

//创建Spring工厂（创建IOC容器）

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

System.***out***.println("==============");

UserDao userDao1 = (UserDao) ac.getBean("userDao");

UserDao userDao2 = (UserDao) ac.getBean("userDao");

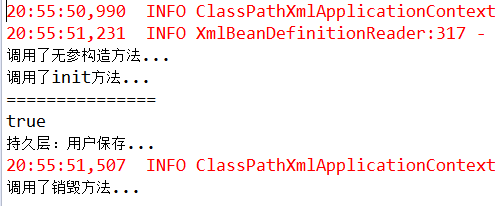
System.***out***.println(userDao1 == userDao2);

//显示关闭ioc+bean一定要是单例的

((ClassPathXmlApplicationContext)ac).close();

}

测试结果如下：



# Spring生成bean的三种方式

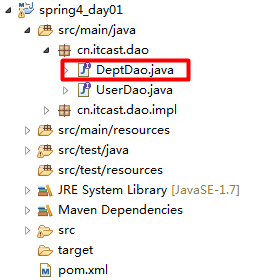
## 无参构造方法

默认调用无参构造方法实例化bean。在此之前，都是调用无参构造来实例化的。

## 静态工厂实例化方式

通过调用工厂类的静态方法来生成bean

* 编写DeptDao接口



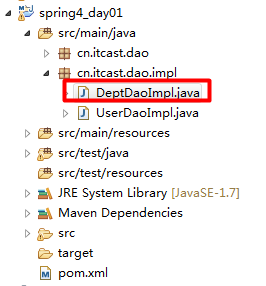
**package** cn.itcast.dao;

**public** **interface** DeptDao {

**public** **void** save();

}

* 编写DeptDaoImpl实现类



**public** **class** DeptDaoImpl **implements** DeptDao{

@Override

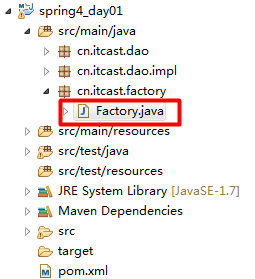
**public** **void** save() {

System.***out***.println("持久层：部门保存...");

}

}

* 编写工厂类，在其中创建静态工厂方法



**public** **class** Factory {

/\*\*

\* 静态工厂方法

\*/

**public** **static** DeptDao create(){

System.***out***.println("调用了静态工厂方法");

**return** **new** DeptDaoImpl();

}

}

* 编写applicationContext.xml配置文件，采用静态工厂方式配置DeptDaoImpl类

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

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

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

xsi:schemaLocation=*"http://www.springframework.org/schema/beans*

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

<bean id=*"userDao"* class=*"cn.itcast.dao.impl.UserDaoImpl"* init-method=*"init"* destroy-method=*"destroy"*></bean>

<bean id=*"deptDao"* class=*"cn.itcast.factory.Factory"* factory-method=*"create"*></bean>

</beans>

在配置DeptDaoImpl这个bean时,class属性写的不是DeptDaoImpl的全路径名，而是工厂类的全路径名；

factory-method：指定工厂类中静态方法的名字

在TestIOC类中编写测试方法test4

@Test

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

//创建Spring工厂（创建IOC容器）

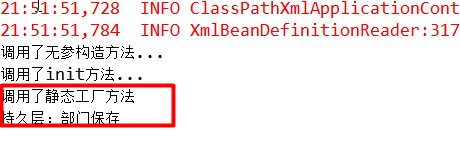
ApplicationContext ac = **new** ClassPathXmlApplicationContext("applicationContext.xml");

DeptDao deptDao = (DeptDao) ac.getBean("deptDao");

deptDao.save();

}

测试结果如下:



## 实例工厂实例化方式

* 修改Factory工厂类，创建实例工厂方法:

**public** **class** Factory {

/\*\*

\* 静态工厂方法

\*/

/\*public static DeptDao create(){

System.out.println("调用了静态工厂方法");

return new DeptDaoImpl();

}\*/

/\*\*

\* 实例工厂方法

\* **@return**

\*/

**public** DeptDao create(){

System.***out***.println("调用了实例工厂方法");

**return** **new** DeptDaoImpl();

}

}

* 编写applicationContext.xml，采用实例工厂方式重写配置DeptDaoImpl

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

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

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

xsi:schemaLocation=*"http://www.springframework.org/schema/beans*

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

<bean id=*"userDao"* class=*"cn.itcast.dao.impl.UserDaoImpl"* init-method=*"init"* destroy-method=*"destroy"*></bean>

<!-- <bean id="deptDao" class="cn.itcast.factory.Factory" factory-method="create"></bean> -->

<!-- 实例工厂方法来实例化 -->

<bean id=*"factory"* class=*"cn.itcast.factory.Factory"*></bean>

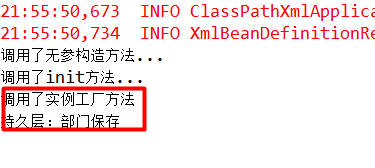
<bean id=*"deptDao"* factory-bean=*"factory"* factory-method=*"create"*></bean>

</beans>

factory-bean：指定工厂bean的id;

Factory-method：指定工厂bean的实例工厂方法的名字

运行test4测试方法，测试结果如下：



# 依赖注入

## 什么是依赖注入

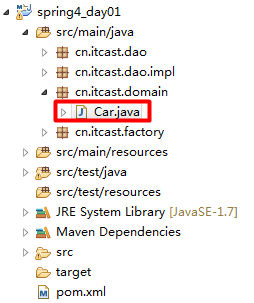
IOC和DI的概念：  
        \* IOC        -- Inverse of Control，控制反转，将对象的创建权反转给Spring！！  
        \* DI        -- **Dependency Injection**，依赖注入，在Spring框架负责创建Bean对象时，动态的将依赖对象注入到Bean组件中！！

如果UserServiceImpl的实现类中有一个属性，那么使用Spring框架的IOC功能时，可以通过依赖注入把该属性的值传入进来！！

## 构造方法注入

什么是构造方法注入？构造方法注入就是利用bean的构造方法完成对bean中属性的赋值。

* 创建Car实体类，**提供有参构造方法**



**package** cn.itcast.domain;

**public** **class** Car **implements** Serializable{

**private** **static** **final** **long** ***serialVersionUID*** = 1L;

**private** String name;

**private** Double price;

**public** Car(String name, Double price) {

**this**.name = name;

**this**.price = price;

}

**public** String getName() {

**return** name;

}

**public** **void** setName(String name) {

**this**.name = name;

}

**public** Double getPrice() {

**return** price;

}

**public** **void** setPrice(Double price) {

**this**.price = price;

}

@Override

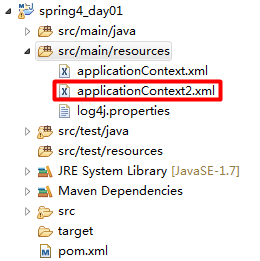
**public** String toString() {

**return** "Car [name=" + name + ", price=" + price + "]";

}

}

* 创建applicationContext2.xml,在applicationContext2.xml配置Car这个bean



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

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

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

xsi:schemaLocation=*"http://www.springframework.org/schema/beans*

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

<!-- 构造方法注入 -->

<bean id=*"car"* class=*"cn.itcast.domain.Car"*>

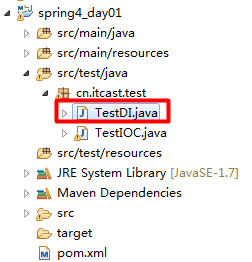
<constructor-arg name=*"name"* value=*"奥迪A6"*></constructor-arg>

<constructor-arg name=*"price"* value=*"57.3"*></constructor-arg>

</bean>

</beans>

* 创建单元测试类TestDI,在其中创建单元测试方法test1测试，注意此处创建IOC容器时，应该加载applicationContext2.xml



**public** **class** TestDI {

@Test

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

//创建Spring工厂（创建IOC容器）

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

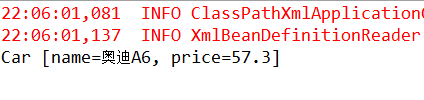
Car car = (Car) ac.getBean("car");

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

}

}

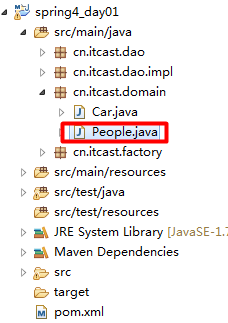
* 运行结果：



## set方法注入

什么是set方法注入？set方法注入就是利用bean中属性的set方法对属性赋值。

* 创建People实体类，提供属性的set方法，不需要提供有参构造方法。



**public** **class** People **implements** Serializable {

**private** **static** **final** **long** ***serialVersionUID*** = 1L;

**private** String name;//要提供属性所对应的set方法

**private** String address;

**private** Car car;//对象属性

**public** **void** setName(String name) {

**this**.name = name;

}

**public** **void** setAddress(String address) {

**this**.address = address;

}

**public** **void** setCar(Car car) {

**this**.car = car;

}

@Override

**public** String toString() {

**return** "People [name=" + name + ", address=" + address + ", car=" + car + "]";

}

}

* 在applicationContext.xml2中配置People实体类，普通属性用value指定值，对象属性用ref指定需要注入的bean的id.

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

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

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

xsi:schemaLocation=*"http://www.springframework.org/schema/beans*

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

<!-- 构造方法注入 -->

<bean id=*"car"* class=*"cn.itcast.domain.Car"*>

<constructor-arg name=*"name"* value=*"奥迪A6"*></constructor-arg>

<constructor-arg name=*"price"* value=*"57.3"*></constructor-arg>

</bean>

<!-- 第二种注入形式：set方法注入 -->

<bean id=*"people"* class=*"cn.itcast.domain.People"*>

<property name=*"name"* value=*"小明"*></property>

<property name=*"address"* value=*"上海"*></property>

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

</bean>

</beans>

* 在TestIOC中创建test2方法

@Test

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

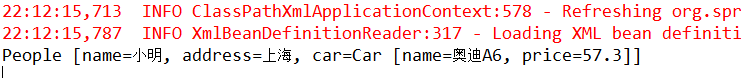
//创建Spring工厂（创建IOC容器）

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

People people = (People) ac.getBean("people");

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

}



**注意：采用set方法注入时，类中一定要有无参构造方法，因为spring会先调用无参构造方法实例化对象。**

## set方法其它注入写法

通过set方法注入还有其它两种写法，这两种写法都是spring在新的版本中提供的写法：

1. p命名空间的写法
2. SpEL的写法

### p命名空间的写法

* 在applicationContext2.xml中引入p命名空间



* 修改applicationContext2.xml，采用p命名空间的写法为People注入值。语法是：

p:简单属性名=”值”

p:对象属性名-ref=”bean的id”

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

xsi:schemaLocation=*"http://www.springframework.org/schema/beans*

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

<!-- 构造方法注入 -->

<bean id=*"car"* class=*"cn.itcast.domain.Car"*>

<constructor-arg name=*"name"* value=*"奥迪A6"*></constructor-arg>

<constructor-arg name=*"price"* value=*"57.3"*></constructor-arg>

</bean>

<!-- 第二种注入形式：set方法注入 -->

<!-- <bean id="people" class="cn.itcast.domain.People">

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

<property name="address" value="上海"></property>

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

</bean> -->

<bean id=*"people"* class=*"cn.itcast.domain.People"* p:name=*"小刚"* p:address=*"北京"* p:car-ref=*"car"*></bean>

</beans>

### SpEL的写法（spring3.0提供）

什么是SpEl:Spring Expression Language

* 修改applicationContext2.xml，采用SpEL的写法为People注入值

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

xsi:schemaLocation=*"http://www.springframework.org/schema/beans*

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

<!-- 构造方法注入 -->

<bean id=*"car"* class=*"cn.itcast.domain.Car"*>

<constructor-arg name=*"name"* value=*"奥迪A6"*></constructor-arg>

<constructor-arg name=*"price"* value=*"57.3"*></constructor-arg>

</bean>

<!-- 第二种注入形式：set方法注入 -->

<!-- <bean id="people" class="cn.itcast.domain.People">

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

<property name="address" value="上海"></property>

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

</bean> -->

<!-- <bean id="people" class="cn.itcast.domain.People" p:name="小刚" p:address="北京" p:car-ref="car"></bean> -->

<bean id=*"people"* class=*"cn.itcast.domain.People"*>

<property name=*"name"* value=*"#{'小明'}"*></property>

<property name=*"address"* value=*"#{'上海'}"*></property>

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

</bean>

</beans>

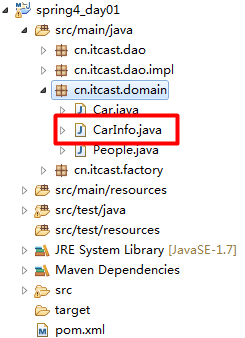
SpEL注入的语法是：

注入字符串：#{‘字符串’}

注入数字：#{数字}

注入其它对象：#{对象id}

* SpEL还可以注入其它对象的属性或方法的返回值，创建CarInfo类，存储Car的信息：



**public** **class** CarInfo {

**public** String getName(){

**return** "宝骏560";

}

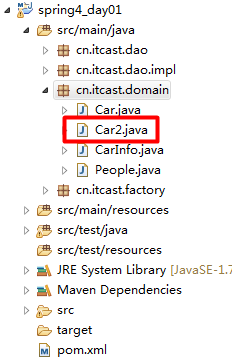
**public** **double** calculatePrice(){

**return** Math.*random*() \* 10000;

}

}

* 创建Car2实体类



**public** **class** Car2 **implements** Serializable {

**private** **static** **final** **long** ***serialVersionUID*** = 1L;

**private** String name;

**private** Double price;

**public** **void** setName(String name) {

**this**.name = name;

}

**public** **void** setPrice(Double price) {

**this**.price = price;

}

@Override

**public** String toString() {

**return** "Car [name=" + name + ", price=" + price + "]";

}

}

在applicationContext2.xml配置Car2

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

xsi:schemaLocation=*"http://www.springframework.org/schema/beans*

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

<!-- 构造方法注入 -->

<bean id=*"car"* class=*"cn.itcast.domain.Car"*>

<constructor-arg name=*"name"* value=*"奥迪A6"*></constructor-arg>

<constructor-arg name=*"price"* value=*"57.3"*></constructor-arg>

</bean>

<!-- 第二种注入形式：set方法注入 -->

<!-- <bean id="people" class="cn.itcast.domain.People">

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

<property name="address" value="上海"></property>

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

</bean> -->

<!-- <bean id="people" class="cn.itcast.domain.People" p:name="小刚" p:address="北京" p:car-ref="car"></bean> -->

<bean id=*"people"* class=*"cn.itcast.domain.People"*>

<property name=*"name"* value=*"#{'小明'}"*></property>

<property name=*"address"* value=*"#{'上海'}"*></property>

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

</bean>

<!-- SpEL的写法 -->

<bean id=*"carInfo"* class=*"cn.itcast.domain.CarInfo"*></bean>

<bean id=*"car2"* class=*"cn.itcast.domain.Car2"*>

<property name=*"name"* value=*"#{carInfo.name}"*></property>

<property name=*"price"* value=*"#{carInfo.calculatePrice()}"*></property>

</bean>

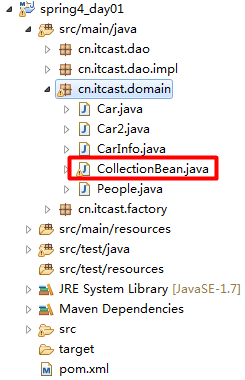
</beans>

**注意：#{carInfo.carName}会调用getCarName方法获取汽车的名称；而#{carInfo.calculatePrice()}会直接调用calculatePrice方法获取汽车价格。**

## 数组或list注入（重点）

有的时候，bean中的属性是List或数组类型。那么该怎么给List或数组注入值呢？数组和list注入的写法是一样的。

* 新建bean类：CollectionBean



**public** **class** CollectionBean **implements** Serializable{

**private** **static** **final** **long** ***serialVersionUID*** = 1L;

**private** List<String> list;//也可以是数组

**public** **void** setList(List<String> list) {

**this**.list = list;

}

@Override

**public** String toString() {

**return** "CollectionBean [list=" + list + "]";

}

}

* 在applicationContext2.xml中配置CollectionBean

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

xsi:schemaLocation=*"http://www.springframework.org/schema/beans*

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

<!-- 构造方法注入 -->

<bean id=*"car"* class=*"cn.itcast.domain.Car"*>

<constructor-arg name=*"name"* value=*"奥迪A6"*></constructor-arg>

<constructor-arg name=*"price"* value=*"57.3"*></constructor-arg>

</bean>

<!-- 第二种注入形式：set方法注入 -->

<!-- <bean id="people" class="cn.itcast.domain.People">

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

<property name="address" value="上海"></property>

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

</bean> -->

<!-- <bean id="people" class="cn.itcast.domain.People" p:name="小刚" p:address="北京" p:car-ref="car"></bean> -->

<bean id=*"people"* class=*"cn.itcast.domain.People"*>

<property name=*"name"* value=*"#{'小明'}"*></property>

<property name=*"address"* value=*"#{'上海'}"*></property>

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

</bean>

<!-- SpEL的写法 -->

<bean id=*"carInfo"* class=*"cn.itcast.domain.CarInfo"*></bean>

<bean id=*"car2"* class=*"cn.itcast.domain.Car2"*>

<property name=*"name"* value=*"#{carInfo.name}"*></property>

<property name=*"price"* value=*"#{carInfo.calculatePrice()}"*></property>

</bean>

<!-- 特殊类型的注入 -->

<bean id=*"cb"* class=*"cn.itcast.domain.CollectionBean"*>

**<property name=*"list"*>**

**<list>**

**<value>乔峰</value>**

**<value>段誉</value>**

**<value>虚竹</value>**

**</list>**

**</property>**

</bean>

</beans>

在TestDI中创建单元测试方法test3:

@Test

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

//创建Spring工厂（创建IOC容器）

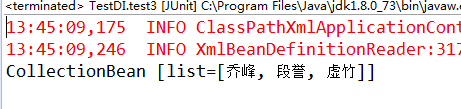
ApplicationContext ac = **new** ClassPathXmlApplicationContext("applicationContext.xml");

CollectionBean cb = (CollectionBean) ac.getBean("cb");

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

}

测试结果如下：



## Set集合的注入

有的时候，bean中的属性是Set类型。那么该怎么给Set类型的属性注入值呢？

修改CollectionBean，在其中添加一个Set类型的属性并提供set方法：

**public** **class** CollectionBean **implements** Serializable{

**private** **static** **final** **long** ***serialVersionUID*** = 1L;

**private** List<String> list;//也可以也成数组

**private** Set<String> set;

**public** **void** setList(List<String> list) {

**this**.list = list;

}

**public** **void** setSet(Set<String> set) {

**this**.set = set;

}

@Override

**public** String toString() {

**return** "CollectionBean [list=" + list + ", set=" + set + "]";

}

}

在applicaitonContext2.xml中为Set类型的属性注入值：

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

xsi:schemaLocation=*"http://www.springframework.org/schema/beans*

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

<!-- 构造方法注入 -->

<bean id=*"car"* class=*"cn.itcast.domain.Car"*>

<constructor-arg name=*"name"* value=*"奥迪A6"*></constructor-arg>

<constructor-arg name=*"price"* value=*"57.3"*></constructor-arg>

</bean>

<!-- 第二种注入形式：set方法注入 -->

<!-- <bean id="people" class="cn.itcast.domain.People">

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

<property name="address" value="上海"></property>

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

</bean> -->

<!-- <bean id="people" class="cn.itcast.domain.People" p:name="小刚" p:address="北京" p:car-ref="car"></bean> -->

<bean id=*"people"* class=*"cn.itcast.domain.People"*>

<property name=*"name"* value=*"#{'小明'}"*></property>

<property name=*"address"* value=*"#{'上海'}"*></property>

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

</bean>

<!-- SpEL的写法 -->

<bean id=*"carInfo"* class=*"cn.itcast.domain.CarInfo"*></bean>

<bean id=*"car2"* class=*"cn.itcast.domain.Car2"*>

<property name=*"name"* value=*"#{carInfo.name}"*></property>

<property name=*"price"* value=*"#{carInfo.calculatePrice()}"*></property>

</bean>

<!-- 特殊类型的注入 -->

<bean id=*"cb"* class=*"cn.itcast.domain.CollectionBean"*>

<property name=*"list"*>

<list>

<value>乔峰</value>

<value>段誉</value>

<value>虚竹</value>

</list>

</property>

**<property name=*"set"*>**

**<set>**

**<value>鸠摩智</value>**

**<value>天山童姥</value>**

**<value>无崖子</value>**

**</set>**

**</property>**

</bean>

</beans>

提示：spring在注入set的时，给我们注入的是一个LinkedHashSet，所以在输入set集合中的元素时，是按照我们注入的顺序来的，并不是无序的。

## Map集合的注入

有的时候，bean中的属性是Map类型。那么该怎么给Map类型的属性注入值呢？

修改CollectionBean，在其中添加一个Map类型的属性并提供set方法：

**public** **class** CollectionBean **implements** Serializable{

/\*\*

\*

\*/

**private** **static** **final** **long** ***serialVersionUID*** = 1L;

**private** List<String> list;//也可以也成数组

**private** Set<String> set;

**private Map<String,String> map;**

**public** **void** setList(List<String> list) {

**this**.list = list;

}

**public** **void** setSet(Set<String> set) {

**this**.set = set;

}

**public void setMap(Map<String, String> map) {**

**this.map = map;**

**}**

@Override

**public** String toString() {

**return** "CollectionBean [list=" + list + ", set=" + set + ", map=" + map + "]";

}

}

在applicationContext2.xml中为Map输入注入值：

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

xsi:schemaLocation=*"http://www.springframework.org/schema/beans*

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

<!-- 构造方法注入 -->

<bean id=*"car"* class=*"cn.itcast.domain.Car"*>

<constructor-arg name=*"name"* value=*"奥迪A6"*></constructor-arg>

<constructor-arg name=*"price"* value=*"57.3"*></constructor-arg>

</bean>

<!-- 第二种注入形式：set方法注入 -->

<!-- <bean id="people" class="cn.itcast.domain.People">

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

<property name="address" value="上海"></property>

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

</bean> -->

<!-- <bean id="people" class="cn.itcast.domain.People" p:name="小刚" p:address="北京" p:car-ref="car"></bean> -->

<bean id=*"people"* class=*"cn.itcast.domain.People"*>

<property name=*"name"* value=*"#{'小明'}"*></property>

<property name=*"address"* value=*"#{'上海'}"*></property>

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

</bean>

<!-- SpEL的写法 -->

<bean id=*"carInfo"* class=*"cn.itcast.domain.CarInfo"*></bean>

<bean id=*"car2"* class=*"cn.itcast.domain.Car2"*>

<property name=*"name"* value=*"#{carInfo.name}"*></property>

<property name=*"price"* value=*"#{carInfo.calculatePrice()}"*></property>

</bean>

<!-- 特殊类型的注入 -->

<bean id=*"cb"* class=*"cn.itcast.domain.CollectionBean"*>

<property name=*"list"*>

<list>

<value>乔峰</value>

<value>段誉</value>

<value>虚竹</value>

</list>

</property>

<property name=*"set"*>

<set>

<value>鸠摩智</value>

<value>天山童姥</value>

<value>无崖子</value>

</set>

</property>

**<property name=*"map"*>**

**<map>**

**<entry key=*"id"* value=*"1"*></entry>**

**<entry key=*"username"* value=*"张三"*></entry>**

**</map>**

**</property>**

</bean>

</beans>

说明：

<entry>表示map中的一个键值对；

<entry>的key表示键，value表示值；

## Properties的注入（重点）

有的时候，bean中的属性是Properties类型。那么该怎么给Properties类型的属性注入值呢？

修改CollectionBean，在其中添加一个Properties类型的属性并提供set方法：

**public** **class** CollectionBean **implements** Serializable{

/\*\*

\*

\*/

**private** **static** **final** **long** ***serialVersionUID*** = 1L;

**private** List<String> list;//也可以也成数组

**private** Set<String> set;

**private** Map<String,String> map;

**private Properties properties;**

**public** **void** setList(List<String> list) {

**this**.list = list;

}

**public** **void** setSet(Set<String> set) {

**this**.set = set;

}

**public** **void** setMap(Map<String, String> map) {

**this**.map = map;

}

**public void setProperties(Properties properties) {**

**this.properties = properties;**

**}**

@Override

**public** String toString() {

**return** "CollectionBean [list=" + list + ", set=" + set + ", map=" + map + ", properties=" + properties + "]";

}

}

在applicationContext2.xml中给Properties类型的属性注入值：

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

xsi:schemaLocation=*"http://www.springframework.org/schema/beans*

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

<!-- 构造方法注入 -->

<bean id=*"car"* class=*"cn.itcast.domain.Car"*>

<constructor-arg name=*"name"* value=*"奥迪A6"*></constructor-arg>

<constructor-arg name=*"price"* value=*"57.3"*></constructor-arg>

</bean>

<!-- 第二种注入形式：set方法注入 -->

<!-- <bean id="people" class="cn.itcast.domain.People">

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

<property name="address" value="上海"></property>

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

</bean> -->

<!-- <bean id="people" class="cn.itcast.domain.People" p:name="小刚" p:address="北京" p:car-ref="car"></bean> -->

<bean id=*"people"* class=*"cn.itcast.domain.People"*>

<property name=*"name"* value=*"#{'小明'}"*></property>

<property name=*"address"* value=*"#{'上海'}"*></property>

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

</bean>

<!-- SpEL的写法 -->

<bean id=*"carInfo"* class=*"cn.itcast.domain.CarInfo"*></bean>

<bean id=*"car2"* class=*"cn.itcast.domain.Car2"*>

<property name=*"name"* value=*"#{carInfo.name}"*></property>

<property name=*"price"* value=*"#{carInfo.calculatePrice()}"*></property>

</bean>

<!-- 特殊类型的注入 -->

<bean id=*"cb"* class=*"cn.itcast.domain.CollectionBean"*>

<property name=*"list"*>

<list>

<value>乔峰</value>

<value>段誉</value>

<value>虚竹</value>

</list>

</property>

<property name=*"set"*>

<set>

<value>鸠摩智</value>

<value>天山童姥</value>

<value>无崖子</value>

</set>

</property>

<property name=*"map"*>

<map>

<entry key=*"id"* value=*"1"*></entry>

<entry key=*"username"* value=*"张三"*></entry>

</map>

</property>

**<property name=*"properties"*>**

**<props>**

**<prop key=*"id"*>2</prop>**

**<prop key=*"name"*>小明</prop>**

**</props>**

**</property>**

</bean>

</beans>

说明：

<prop>：表示一个键值对；

<prop>中的key表示键，在<prop></prop>中写的是值；

## 配置文件的分离

在实际开发时，会有很多个bean都需要配置在spring中。但是，如果把所有的bean都配置在applicationContext.xml中，applicationContext.xml就会变得很庞大，不便于修改与维护。可以把spring的配置文件进行分离：

Spring配置文件的分离有两种方式：

方式一：在applicationContext.xml中采用import标签导入另一个配置文件

现在，在整个工程中有applicationContext.xml和applicationContext2.xml两个配置文件。可以把aplicationContext.xml看成一个总的配置文件，在其中包含applicaitonContext2.xml文件。修改applicationContext.xml内容如下：

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

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

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

xsi:schemaLocation=*"http://www.springframework.org/schema/beans*

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

<bean id=*"userDao"* class=*"cn.itcast.dao.impl.UserDaoImpl"*

init-method=*"init"* destroy-method=*"destroy"*></bean>

<!-- <bean id="deptDao" class="cn.itcast.factory.Factory" factory-method="create"></bean> -->

<!-- 实例工厂方法来实例化 -->

<bean id=*"factory"* class=*"cn.itcast.factory.Factory"*></bean>

<bean id=*"deptDao"* factory-bean=*"factory"* factory-method=*"create"*></bean>

<!-- 构造方法注入 -->

<bean id=*"car"* class=*"cn.itcast.domain.Car"*>

<constructor-arg name=*"name"* value=*"奥迪A6"*></constructor-arg>

<constructor-arg name=*"price"* value=*"57.3"*></constructor-arg>

</bean>

**<import resource=*"applicationContext2.xml"*/>**

</beans>

此时，创建IOC容器时，只需要加载applicationContext.xml即可，因为在applicationContext.xml引入了applicationContext2.xml文件，所以，applicationContext2.xml也会加载。

方式二：在实例化ApplicationContext的时候，指定多个配置文件。修改TestIOC中的test4方法，创建ApplicationContext时同时加载多个文件：

@Test

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

//创建Spring工厂（创建IOC容器）

ApplicationContext ac = **new** ClassPathXmlApplicationContext("applicationContext.xml","applicationContext2.xml");

DeptDao deptDao = (DeptDao) ac.getBean("deptDao");

deptDao.save();

}

# 附录：开闭原则

