Spring第一天

**整体课程安排：**

第一天：Spring框架入门、IoC控制反转的xml配置管理

第二天: IoC控制反转的注解配置管理、Spring Web集成、Spring Junit集成, Spring AOP面向切面编程底层原理

第三天：AspectJ的集成配置、JdbcTemplate工具类。

第四天：Spring声明式事务管理,xml和注解配置

**第一天的主要内容（IoC相关）：**

Spring的概述

Spring IoC快速入门（工程环境构建、IoC和DI概念）

Spring IoC容器装配Bean的配置（XML方式）

**学习目标：**

掌握：什么是IoC，什么是Di，怎么通过spring装配Bean

# Spring的概述

## 什么是Spring

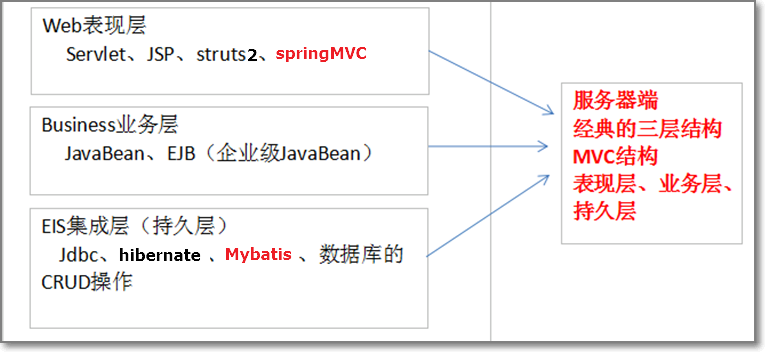
Spring是分层的、JavaSE/EE一站式(full-stack)、轻量级开源框架。

* JavaEE分层

JavaEE规范的三层结构体系：

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

如图：



* 一站式

Spring提供了JavaEE各层的解决方案：

表现层：struts1、struts2、Spring MVC

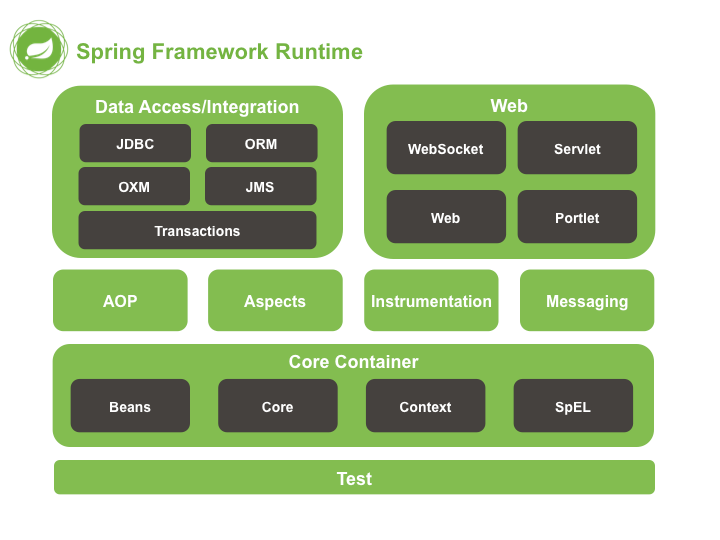
业务层：Ioc、AOP、事务控制

持久层：JdbcTemplate、mybatis、hibernate、springDataJpa框架（对象关系映射）整合

* 轻量级：Spring的出现取代了EJB的臃肿、低效、繁琐复杂、脱离现实的情况. 而且使用spring编程是非侵入式的。

## Spring的体系结构

Spring框架是一个分层架构，它包含一系列的功能要素并被分为大约20个模块。这些模块分为Core Container、Data Access/Integration、Web、AOP（Aspect Oriented Programming）、Instrumentation和测试部分，如图：



**核心容器(Core Container)** 包括Core、Beans、Context、EL模块。

1：Core和Beans模块提供了Spring最基础的功能，提供IoC和依赖注入特性。这里的基础概念是BeanFactory，它提供对Factory模式的经典实现来消除对程序性单例模式的需要，并真正地允许你从程序逻辑中分离出依赖关系和配置。

2：Context模块基于Core和Beans来构建，它提供了用一种框架风格的方式来访问对象，有些像JNDI注册表。Context封装包继承了beans包的功能，还增加了国际化（I18N）,事件传播，资源装载，以及透明创建上下文，例如通过servlet容器，以及对大量JavaEE特性的支持，如EJB、JMX。核心接口是ApplicationContext。

3：Expression Language，表达式语言模块，提供了在运行期间查询和操作对象图的强大能力。支持访问和修改属性值，方法调用，支持访问及修改数组、容器和索引器，命名变量，支持算数和逻辑运算，支持从Spring 容器获取Bean，它也支持列表投影、选择和一般的列表聚合等。

**数据访问/集成部分(Data Access/Integration)**

1：JDBC模块，提供对JDBC的抽象，它可消除冗长的JDBC编码和解析数据库厂商特有的错误代码。

2：ORM模块，提供了常用的"对象/关系"映射APIs的集成层。 其中包括JPA、JDO、Hibernate 和 iBatis 。利用ORM封装包，可以混合使用所有Spring提供的特性进行"对象/关系"映射，如简单声明事务管理 。

3：OXM模块，提供一个支持Object和XML进行映射的抽象层，其中包括JAXB、Castor、XMLBeans、JiBX和XStream。

4：JMS模块，提供一套"消息生产者、消费者"模板用于更加简单的使用JMS，JMS用于用于在两个应用程序之间，或分布式系统中发送消息，进行异步通信。

5：Transaction模块，支持程序通过简单声明性 事务管理，只要是Spring管理对象都能得到Spring管理事务的好处，即使是POJO，也可以为他们提供事务。

**Web**

1：Web模块，提供了基础的web功能。例如多文件上传、集成IoC容器、远程过程访问、以及Web Service支持，并提供一个RestTemplate类来提供方便的Restful services访问

2：Web-Servlet模块，提供了Web应用的Model-View-Controller（MVC）实现。Spring MVC框架提供了基于注解的请求资源注入、更简单的数据绑定、数据验证等及一套非常易用的JSP标签，完全无缝与Spring其他技术协作。

3：Web-Portlet模块，提供了在Portlet环境下的MVC实现

**AOP**

1：AOP模块，提供了符合AOP 联盟规范的面向方面的编程实现，让你可以定义如方法拦截器和切入点，从逻辑上讲，可以减弱代码的功能耦合，清晰的被分离开。而且，利用源码级的元数据功能，还可以将各种行为信息合并到你的代码中 。

2：Aspects模块，提供了对AspectJ的集成。

3：Instrumentation模块， 提供一些类级的工具支持和ClassLoader级的实现，可以在一些特定的应用服务器中使用。

**Test**

1：Test模块，提供对使用JUnit和TestNG来测试Spring组件的支持，它提供一致的ApplicationContexts并缓存这些上下文，它还能提供一些mock对象，使得你可以独立的测试代码。

## Spring的核心

IoC（Inverse of Control 控制反转）： 将对象创建权利交给Spring工厂进行管理。 比如说 Book book = new Book();

现在: Book book = Spring工厂.getBook();

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

今天主要学习IoC

## Spring的优点

Spring 出现为了解决JavaEE 实际问题

（1）方便解耦，简化开发

Spring就是一个大工厂，它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

（3）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

关于框架的特性，我们也会俗称Spring为开发架构的粘合剂。

# Spring IoC快速入门

Spring核心内容的基本开发步骤：

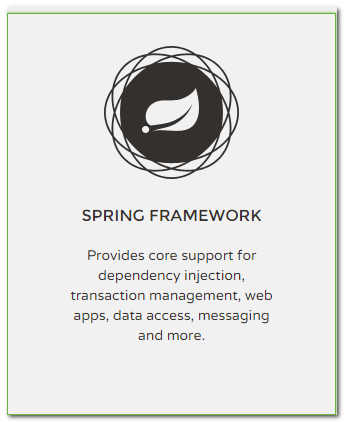
* 下载开发包，导入jar包
* 编写代码（基础代码和调用代码）
* 编写配置文件（XML）

## Spring的开发包

开发包的下载

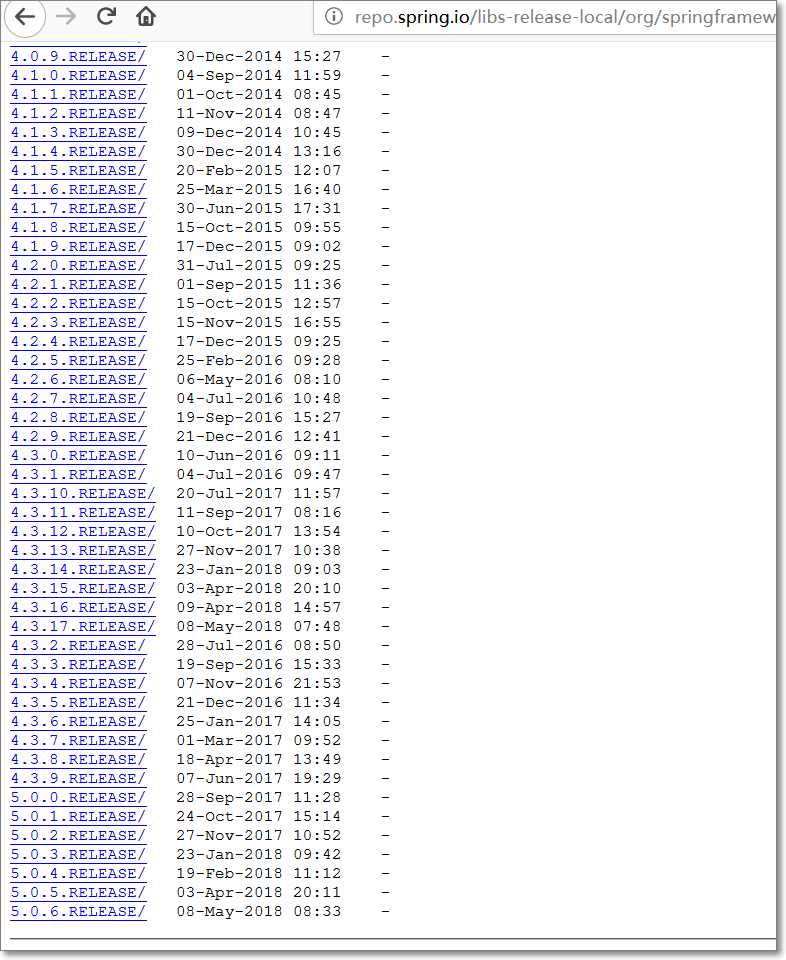


Spring官方：http://spring.io/

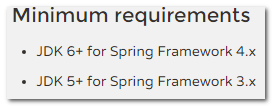


下载网址：http://repo.spring.io/libs-release-local/org/springframework/spring/

官方最新版本：



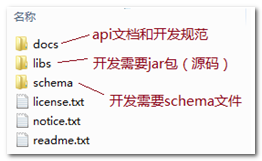
不同系列版本对开发环境的最低需求：



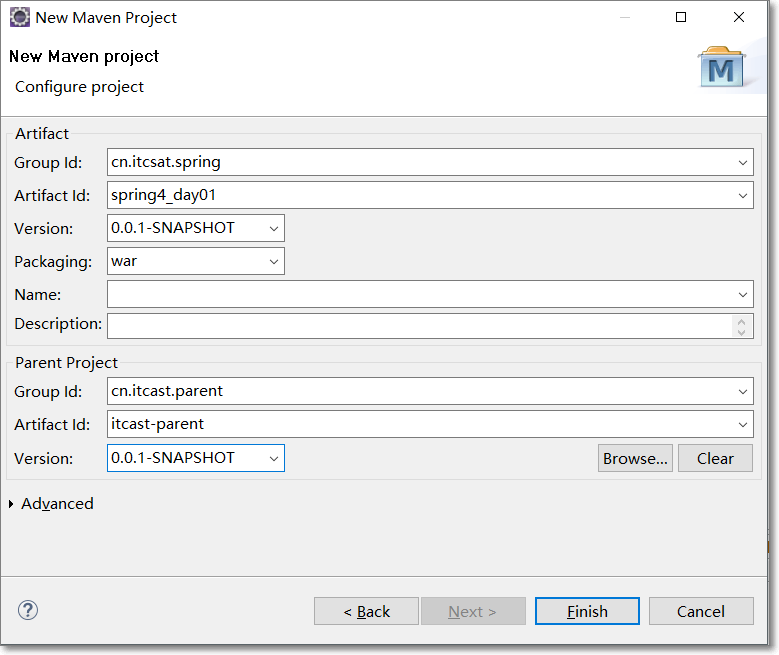
本次课程中我们采用的版本是：4.2.x的版本（企业主流版本，框架整合也需要对应版本jar）：



Spring4.2版本开发包目录结构：



第一步：新建Web工程Spring4\_day01，



第二步：导入jar包

1．Spring项目的核心容器的最基本Jar包（4个）：



Pom.xml:

<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.itcsat.spring</groupId>

<artifactId>spring4\_day01</artifactId>

<packaging>war</packaging>

<dependencies>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-context</artifactId>

</dependency>

<!-- junit -->

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<scope>test</scope>

</dependency>

</dependencies>

</project>

添加日志：

<dependency>

<groupId>org.slf4j</groupId>

<artifactId>slf4j-log4j12</artifactId>

</dependency>

添加log4j.properties文件放置到src下。

log4j.rootLogger=INFO,A1

log4j.logger.org.apache=INFO

log4j.appender.A1.Target=System.err

log4j.appender.A1=org.apache.log4j.ConsoleAppender

log4j.appender.A1.layout=org.apache.log4j.PatternLayout

log4j.appender.A1.layout.ConversionPattern=%-d{yyyy-MM-dd HH:mm:ss,SSS} [%t] [%c]-[%p] %m%n

## 传统方式业务代码编写（业务层、数据持久层）

采用的示例业务是模拟用户登录操作。

第一步：创建包cn.itcast.spring.a\_quickstart

第二步：创建dao

1：创建接口IUserDao

**package** cn.itcast.spring.a\_quickstart;

//用户的dao层

**public** **interface** IUserDao {

//向数据查询数据，根据用户名和密码

**public** **void** findByUsernameAndPassword();

}

2：创建IUserDao接口的实现类UserDaompl

**package** cn.itcast.spring.a\_quickstart;

//dao的实现类

**public** **class** UserDaoImpl **implements** IUserDao {

@Override

**public** **void** findByUsernameAndPassword() {

System.*out*.println("UserDaoImpl-dao层被调用了");

}

}

第三步：创建service

1：创建接口IUserService

**package** cn.itcast.spring.a\_quickstart;

//业务层

**public** **interface** IUserService {

//登录

**public** **void** login();

}

2：创建IUserService接口的实现类UserServiceImpl

**package** cn.itcast.spring.a\_quickstart;

//业务层实现

**public** **class** UserServiceImpl **implements** IUserService{

**public** **void** login() {

System.*out*.println("UserServiceImpl-service层被调用了。。。");

//实例化dao层

//**传统方式**

IUserDao userDao = **new** UserDaoImpl();

userDao.findByUsernameAndPassword();

}

}

第四步：测试

创建SpringTest类进行测试：

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

//测试

@Test

**public** **void** test(){

//创建service的示例

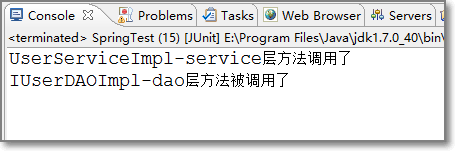
IUserService userService = **new** UserServiceImpl();

userService.login();

}

}

控制台：



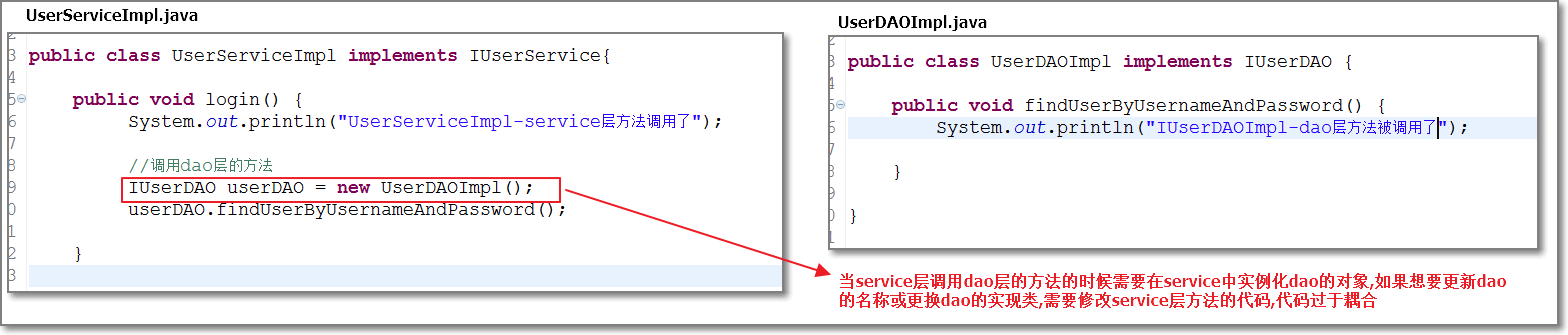
【思考分析】

存在问题：代码过于耦合，上层代码过度依赖于下一层代码的实现：

例如：UserDao userDao = new UserDaoImpl();

如果要更换实现类，或者实现类换一个名字，此时代码会报错，必须要修改原来的业务代码！

传统方式:



怎么解决类与类之间如此密切的耦合问题呢?

## IoC控制反转的实现

采用IoC（Inverse of Control，控制反转）的思想解决代码耦合问题。

简单的说就是引入工厂（第三者），将原来在程序中手动创建管理的依赖的UserDaoImpl对象，交给工厂来创建管理。

IoC方式:

步骤一:提供userDAO实例对象的工厂

UserDAOFactory.java:

**public** **class** UserDAOFactory {

//提供获取对象的方法

**public** UserDAOImpl getUserDAO(){

//返回实例对象

**return** **new** UserDAOImpl ();

}

}

步骤二: 修改UserServiceImpl中获得对象的方式

UserServiceImpl.java:

**public** **class** UserServiceImpl **implements** IUserService{

**public** **void** login() {

System.*out*.println("UserServiceImpl-service层方法调用了");

//调用dao层的方法

// IUserDAO userDAO = new UserDAOImpl();

// userDAO.findUserByUsernameAndPassword();

//ioc方式:

//创建工厂,利用工厂提供依赖的对象

UserDAOFactory userDAOFactory = **new** UserDAOFactory();

UserDAOImpl userDAO = userDAOFactory.getUserDAO();

userDAO.findUserByUsernameAndPassword();

}

}

发现问题:工厂方法仍然需要返回具体类型的实例对象,存在代码耦合

解决方案:使用反射技术传入具体类型的类字符串生产对象的实例:

UserDAOFactory.java:

//利用反射技术生产具体类型的实例对象

**public** Object getBean(){

Object bean = **null**;

**try** {

//传入类字符串,生产对象实例

bean = Class.*forName*("cn.itcast.spring.a\_quickstart.UserDAOImpl").newInstance();

} **catch** (Exception e) {

e.printStackTrace();

}

//返回具体类型的对象类型实例

**return** bean;

}

UserServiceImpl.java:

//使用反射方法获取对象

IUserDAO userDAO = (IUserDAO) userDAOFactory.getBean();

userDAO.findUserByUsernameAndPassword();

发现问题:类字符串是固定的,怎么动态的传入不同的类字符串呢?

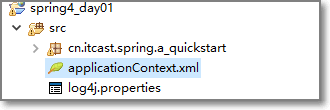
解决方案: 使用xml配置文件动态传入类字符串

IoC底层实现：工厂（设计模式）+反射（机制） + 配置文件（xml）。

### Spring核心配置文件的编写

IoC控制反转的理解和实现

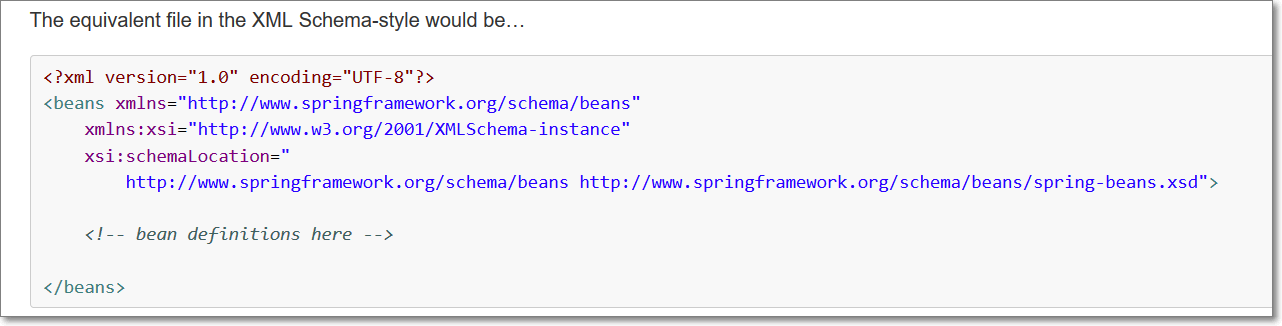
步骤一：在src下建立applicationContext.xml （位置：applicationContext.xml文件放置到任何目录都可以，习惯上放在src目录或者 WEB-INF目录）



步骤二:参考规范文档配置xml的头信息:bean schema

文档位置: / spring-framework-4.2.4.RELEASE-dist/spring-framework-4.2.4.RELEASE/docs/spring-framework-reference/html/xsd-configuration.html

找到下列章节的示例，拷贝到工程中即可：



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"*

xsi:schemaLocation=*"*

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

<!-- bean definitions here -->

</beans>

步骤三:配置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"*

xsi:schemaLocation=*"*

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

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

<!-- bean: spring工厂创建的一个对象(反射机制)

id/name:对象的名字,可以用来引用或者获取对象, 一般为类名或接口名称的首字母小写

class:要创建的对象类型的类字符串,类名全路径

-->

<bean id=*"userDAO"* class=*"cn.itcast.spring.a\_quickstart.UserDAOImpl"* />

</beans>

### 通过Spring的工厂获取Bean完成相关操作

在程序中创建spring工厂对象, 通过工厂对象加载spring的xml配置文件,生产配置文件中配置 的bean对应的对象

UserServiceImpl.java：

//spring配置方式,创建spring工厂,加载spring配置文件

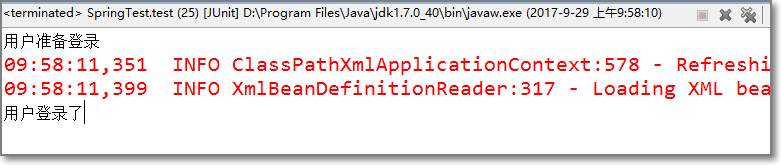
ApplicationContext ac = **new** ClassPathXmlApplicationContext("applicationContext.xml");

//从spring工厂中获取对象,通过bean的id/name

IUserDAO userDAO = (IUserDAO) ac.getBean("userDAO");

userDAO.findUserByUsernameAndPassword();

运行测试:



发现问题:该方式虽然解决了类与类之间的耦合关系,但却需要在获取对象的时候创建spring工厂,有没有更方便获取对象的依赖的方法呢?

## DI依赖注入的实现

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（简单的说，可以将另外一个bean对象动态的注入到另外一个bean中。）

回顾之前的代码:



Di的做法是:由Spring容器创建了Service、Dao对象，并且在配置中将Dao传入Servcie，那么Service对象就包含了Dao对象的引用。

步骤一:将service对象也交给spring容器管理

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"*

xsi:schemaLocation=*"*

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

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

<!-- bean: spring工厂创建的一个对象(反射机制)

id/name:对象的名字,可以用来引用或者获取对象, 一般为类名或接口名称的首字母小写

class:要创建的对象类型的类字符串,类名全路径

-->

<bean id=*"userDAO"* class=*"cn.itcast.spring.a\_quickstart.UserDAOImpl"* />

<bean id =*"userService"* class=*"cn.itcast.spring.a\_quickstart.UserServiceImpl"*>

<!-- 注入对象 -->

<!-- property 根据类中的setter方法进行属性注入 -->

<!-- name:setter方法的后缀小写,比如setXxx 对应的name为xxx -->

<!-- ref:引用哪一个bean(对象),值为bean的id/name -->

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

</bean>

</beans>

步骤二:在程序中定义属性提供setter方法：

UserServiceImpl.java

**public** **class** UserServiceImpl **implements** IUserService{

//定义属性

**private** IUserDAO userDAO;

**public** **void** setUserDAO(IUserDAO userDAO) {

**this**.userDAO = userDAO;

}

**public** **void** login() {

System.*out*.println("UserServiceImpl-service层方法调用了");

//ioc:依赖注入

userDAO.findUserByUsernameAndPassword();

}

步骤三:测试运行,此时获取对象必须从spring工厂获取(在spring容器配置中才有依赖注入,自己创建的对象没有注入依赖关系)

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

//测试

@Test

**public** **void** test(){

//创建service的示例

//IUserService userService = new UserServiceImpl();

//userService.login();

//创建spring工厂,获取spring管理的对象

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

IUserService userService = (IUserService) ac.getBean("userService");

userService.login();

}

}

运行结果:



小结:

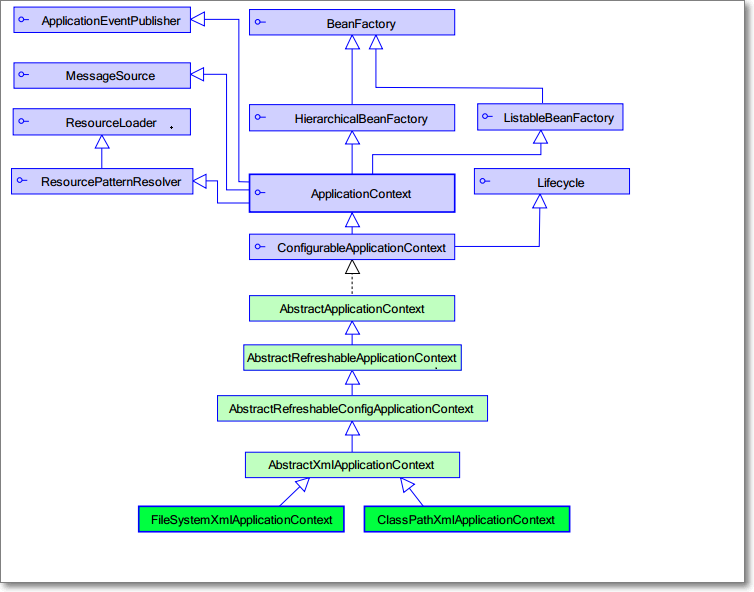
IOC:控制反转,将对象创建管理的权利交给spring容器,获取对象通过spring工厂创建

DI:在spring容器中创建管理多个对象,通过 property标签将对象注入到需要依赖的对象中

## Spring的工厂(了解)

ApplicationContext直译为应用上下文，是用来加载Spring框架配置文件，来构建Spring的工厂对象，它也称之为Spring容器的上下文对象，也称之为Spring的容器。

ApplicationContext 只是BeanFactory（Bean工厂，Bean就是一个java对象） 一个子接口：



为什么不直接使用顶层接口对象来操作呢？

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

\* Beanfactory的用法：

BeanFactory ac = new XmlBeanFactory(new FileSystemResource("D:\\applicationContext.xml"));

\* ApplicationContext是对BeanFactory扩展，提供了更多功能

国际化处理

事件传递

Bean自动装配

各种不同应用层的Context实现

ApplicationContext 更加强大， 所以现在开发基本没人使用BeanFactory。

【扩展】

Bean获取的两种方式：

@Test

**public** **void** getBean(){

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

//获取bean的两种方式

//1.通过spring容器中bean的id/name获取

//IUserService userService = (IUserService) ac.getBean("userService");

//2.根据bean的类型或者bean接口的类型获取,一般使用接口类型

IUserService userService = (IUserService) ac.getBean(IUserService.**class**);

userService.login();

}

常用根据名称获取（id/name）,即第一种方式，使用spring容器中的标识获取对象

如果根据类型获取，配置了多个类型的话，则抛出异常：

applicationContext.xml：

<bean id =*"userService1"* class=*"cn.itcast.spring.a\_quickstart.UserServiceImpl"*>

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

</bean>

<bean id =*"userService"* class=*"cn.itcast.spring.a\_quickstart.UserServiceImpl"*>

<!-- 注入对象 -->

<!-- property 根据类中的setter方法进行属性注入 -->

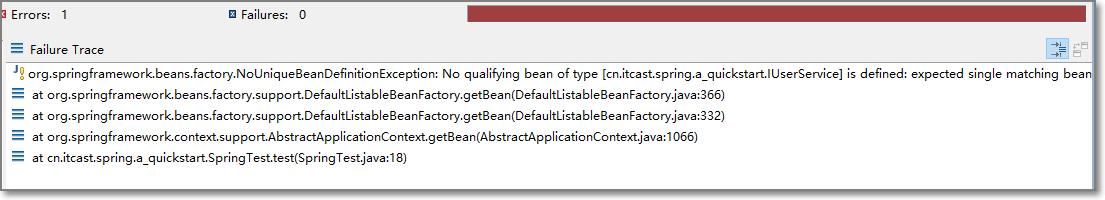
<!-- name:setter方法的后缀小写,比如setXxx 对应的name为xxx -->

<!-- ref:引用哪一个bean(对象),值为bean的id/name -->

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

</bean>

抛出异常



# IoC容器装配Bean\_基于XML配置方式

## 实例化Bean的三种方式 （了解）

创建包：cn.itcast.spring.b\_xmlnewbean

第一种方式 无参数构造器 （最常用）

第一步：创建Bean1.java

//1。默认构造器(spring在创建bean的时候自动调用无参构造器来实例化，相当于new Bean1())

**public** **class** Bean1 {

}

第二步：在spring容器applicationContext.xml中配置

<!-- 实例化 bean的四种方式 -->

<!-- 1.默认构造器实例化对象 -->

<bean id =*"bean1"* class=*"cn.itcast.spring.b\_xmlnewbean.Bean1"* />

第三步:创建测试类获取bean对象

SpringTest.java:

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

@Test

**public** **void** test(){

//创建spring工厂

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

//1.默认构造器获取bean对象

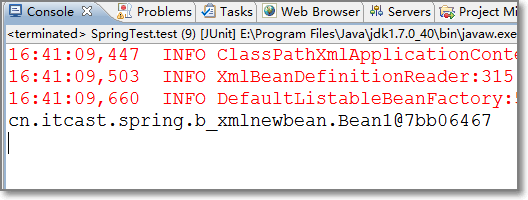
Bean1 bean1 = (Bean1) ac.getBean("bean1");

System.*out*.println(bean1);

}

}

运行结果：



【错误演示】：

**public** **class** Bean1 {

//错误演示

**private** String name;

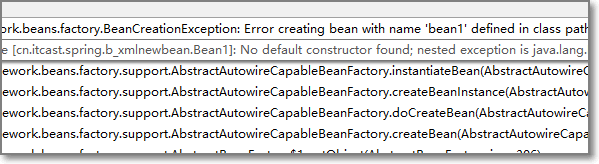
**public** Bean1(String name) {

**this**.name = name;

}

}

运行报错：



第二种方式： 静态工厂方法

第一步：创建Bean2.java

//1.静态工厂方法构造：用来在初始化bean2的时候，可以初始化其他的东西

**public** **class** Bean2 {

}

第二步：创建Bean2Factory.java类

//静态工厂

**public** **class** Bean2Factory {

//静态方法，用来返回对象的实例

**public** **static** Bean2 getBean2(){

//在做实例化的时候，可以做其他的事情，即可以在这里写初始化其他对象的代码

//Connection conn....

**return** **new** Bean2();

}

}

第三步：Spring的容器applicationContext.xml

<!-- 2.静态工厂获取实例化对象 -->

<!-- class:直接指定到静态工厂类, factory-method: 指定生产实例的方法, spring容器在实例化工厂类的时候会自动调用该方法并返回实例对象 -->

<bean id = *"bean2"* class=*"cn.itcast.spring.b\_xmlnewbean.Bean2Factory"* factory-method=*"getBean2"* />

第四步：测试类进行测试

@Test

**public** **void** test(){

//先构建实例化获取spring的容器（工厂、上下文）

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

//2.静态工厂

Bean2 bean2=(Bean2) applicationContext.getBean("bean2");

System.*out*.println(bean2);

}

第三种方式： 实例工厂方法

第一步：创建Bean3.java

//第三种bean，实例工厂方式创建

**public** **class** Bean3 {

}

第二步：创建实例工厂Bean3Factory类

//实例工厂:必须new工厂--》bean

**public** **class** Bean3Factory {

//普通的方法，非静态方法

**public** Bean3 getBean3(){

//初始化实例对象返回

**return** **new** Bean3();

}

}

第三步：Spring容器的配置：applicationContext.xml

<!-- 3：实例工厂的方式实例化bean -->

<bean id=*"bean3Factory"* class=*"cn.itcast.spring.b\_xmlnewbean.Bean3Factory"*/>

<!-- factory-bean相当于ref：引用一个bean对象 -->

<bean id=*"bean3"* factory-bean=*"bean3Factory"* factory-method=*"getBean3"*/>

第四步：使用测试代码，进行测试：

@Test

**public** **void** test(){

//先构建实例化获取spring的容器（工厂、上下文）

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

//3.实例工厂

Bean3 bean3=(Bean3) applicationContext.getBean("bean3");

System.*out*.println(bean3);

}

三种方式：

第一种：最常用

1. 第三种：一些框架初始化的时候用的多。

## Bean的作用域

由spring创建的bean对象在什么情况下有效。



项目开发中通常会使用：singleton 单例、 prototype多例

Singleton： 在一个spring容器中，对象只有一个实例。（默认值）

Prototype： 在一个spring容器中，存在多个实例，每次getBean 返回一个新的实例。

建立包：cn.itcast.spring.c\_xmlscope

第一步：创建类SingletonBean.java和PrototypeBean.java

创建类SingletonBean.java类

//单例bean

**public** **class** SingletonBean {

**public** SingletonBean() {

System.*out*.println("SingletonBean:初始化了单例");

}

}

创建类PrototypeBean.java类

//多例bean

**public** **class** PrototypeBean {

**public** PrototypeBean() {

System.*out*.println("--PrototypeBean初始化了多例的");

}

}

第二步：定义spring容器，applicationContext.xml:

<!--

bean的作用范围

scope:配置作用范围的，默认值就是singleton单例

-->

<!-- 单例 -->

<!-- <bean id="singletonBean" class="cn.itcast.spring.c\_xmlscope.SingletonBean" scope="singleton"/> -->

<bean id=*"singletonBean"* class=*"cn.itcast.spring.c\_xmlscope.SingletonBean"*/>

<!-- 多例 -->

<bean id=*"prototypeBean"* class=*"cn.itcast.spring.c\_xmlscope.PrototypeBean"* scope=*"prototype"*/>

第三步：测试代码，创建SpringTest.java：

//newbean的方式

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

@Test

**public** **void** test(){

//先构建实例化获取spring的容器（工厂、上下文）

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

//目标1：看看多次获取bean的时候，是不是同一个

//目标2：看看bean什么时候初始化的

//获取单例的bean：应该是同一个

//单例：每次从spring容器中获取的对象，是同一个对象

//单例初始化：是在spring容器初始化的时候，就初始化了

SingletonBean singletonBean1=(SingletonBean)applicationContext.getBean("singletonBean");

SingletonBean singletonBean2=(SingletonBean)applicationContext.getBean("singletonBean");

System.*out*.println(singletonBean1);

System.*out*.println(singletonBean2);

//获取多例的bean：

//多例：每次从spring容器中获取的对象，不是同一个对象

//多例初始化：是在getBean的时候初始化，相当于每次getbean就是在new Bean（）

PrototypeBean prototypeBean1=(PrototypeBean)applicationContext.getBean("prototypeBean");

PrototypeBean prototypeBean2=(PrototypeBean)applicationContext.getBean("prototypeBean");

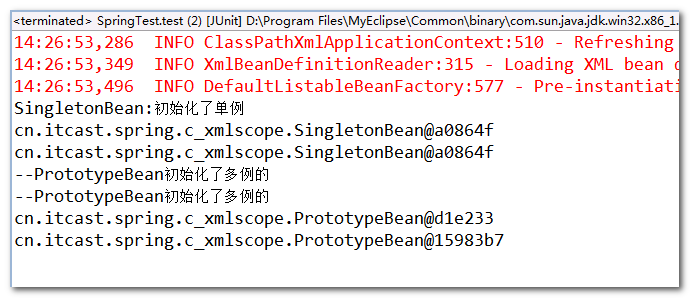
System.*out*.println(prototypeBean1);

System.*out*.println(prototypeBean2);

}

}

运行查看，测试结果：



【注意】

单例是默认值，如果需要单例对象，则不需要配置scope。

## Bean的生命周期

通过spring工厂，可以控制bean的生命周期。

### 在xml配置Bean的初始化和销毁方法

通过 init-method属性 指定实例化后的调用方法

通过 destroy-method属性 指定销毁对象前的方法



创建包cn.itcast.spring.d\_xmllifecycle

第一步：创建LifeCycleBean，指定一个init的方法，和一个destroy的方法。

//测试生命周期过程中的初始化和销毁bean

**public** **class** LifeCycleBean {

//定义构造方法

**public** LifeCycleBean() {

System.*out*.println("LifeCycleBean构造器调用了");

}

//初始化后自动调用方法：方法名随意，但也不能太随便，一会要配置

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

System.*out*.println("LifeCycleBean-init初始化时调用");

}

//bean销毁时调用的方法

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

System.*out*.println("LifeCycleBean-destroy销毁时调用");

}

}

第二步：Spring的核心容器，applicationContext.xml的配置

<!-- 生命周期调用的两个方法

init-method:初始化时（后）调用的，bean中的共有方法即可

destroy-method:销毁时（前）被调用的。

-->

<bean id=*"lifeCycleBean"* class=*"cn.itcast.spring.d\_xmllifecycle.LifeCycleBean"* init-method=*"init"* destroy-method=*"destroy"* />

第三步：SpringTest.java测试代码：

@Test

**public** **void** test(){

//先获取spring的容器，工厂，上下文

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

//对于单例此时已经被初始化

//获取bean

LifeCycleBean lifeCycleBean=(LifeCycleBean) applicationContext.getBean("lifeCycleBean");

System.*out*.println(lifeCycleBean);

//为什么没有销毁方法调用。

//原因是：使用debug模式jvm直接就关了，spring容器还没有来得及销毁对象。

//解决：手动关闭销毁spring容器，自动销毁单例的对象

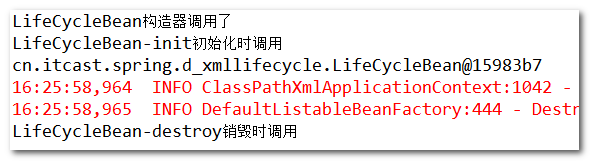
applicationContext**.close();**

}

测试时查看控制台打印，发现销毁方法没有执行。

提示：销毁方法的执行必须满足两个条件：

1. 单例（singleton）的bean才会可以手动销毁。
2. 必须手动关闭容器（调用close的方法）时，才会执行手动销毁的方法。



## Bean属性的依赖注入

### 属性依赖注入的两种种方式

什么是Bean属性的注入？就是对一个对象的属性赋值。有两种方式：

* 第一种：构造器参数注入 new Book(“金瓶梅”,15.8)
* 第二种：setter方法属性注入(setter方法的规范需要符合JavaBean规范)

创建包cn.itcast.spring.e\_xmlpropertydi

### 构造器参数注入 constructor-arg

【示例】

第一步：构造器参数注入属性值。

创建Car类，定义构造方法

//目标，构造器参数注入，new car直接将参数的值直接赋值

**public** **class** Car {

**private** Integer id;

**private** String name;

**private** Double price;

//有参构造

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

**this**.id = id;

**this**.name = name;

**this**.price = price;

}

@Override

**public** String toString() {

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

}

}

第二步：配置applicationContext.xml

<!-- 构造器注入属性的值 -->

<bean id=*"car"* class=*"cn.itcast.spring.e\_xmlpropertydi.Car"*>

<!--constructor-arg：告诉spring容器，要调用有参构造方法了，不再调用默认的构造方法了

new Car(1,"宝马",99999d)

参数第一组：定位属性

\* index:根据索引定位属性，0表示第一个位置

\* name：根据属性参数名称定位属性

\* type:根据属性数据类型定位属性

参数第二组：值

\* value:简单的值，字符串

\* ref:复杂的（由spring容器创建的bean对象）

-->

<!-- <constructor-arg index="0" value="1"/> -->

<constructor-arg index=*"0"* name=*"id"* value=*"1"*/>

<!-- <constructor-arg name="name" value="宝马1代"/> -->

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

<value>宝马2代</value>

</constructor-arg>

<constructor-arg type=*"java.lang.Double"* value=*"99999d"*/>

</bean>

第三步：使用SpringTest.java测试：

@Test

**public** **void** test(){

//spring容器

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

//获取car

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

System.*out*.println(car);

}

【补充】

1．定位属性的标签，可以混用

<constructor-arg index=*"0"* name=*"id"* value=*"1"*/>

2．自标签的属性赋值问题，可以使用子标签的value，效果和value属性一样

<constructor-arg name="name" value="宝马1代"/>

等同于

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

<value>宝马2代</value>

</constructor-arg>

### setter方法属性注入 property

使用的默认的构造器(new Bean())，但必须提供属性的setter方法，使用setter方法也是企业经常使用的属性注入方式。

两步：在类中加入setter方法，在配置文件中使用property

【示例】

第一步：创建Person.java，定义id、name、car属性

/\*\*

\* 定义人类

\* setter方法属性注入

\* 相当于new Person();

\*/

**public** **class** Person {

**private** Integer id;

**private** String name;

**private** Car car;

//必须提供setter属性方法

**public** **void** setId(Integer id) {

**this**.id = id;

}

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

**this**.name = name;

}

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

**this**.car = car;

}

@Override

**public** String toString() {

**return** "Person [id=" + id + ", name=" + name + ", car=" + car + "]";

}

}

第二步：配置spring容器applicationContext.xml

<!-- setter方法属性注入:调用默认构造器，相当于new Person() -->

<bean id=*"person"* class=*"cn.itcast.spring.e\_xmlpropertydi.Person"*>

<!--

property：专门进行setter属性注入用的标签 。

\* name:setter方法的属性的名字,例如SetXxx-那么name的属性值为xxx。

\* value:简单的值

\* ref：bean的名字，对象的引用

-->

<property name=*"id"* value=*"1001"*/>

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

<!-- <property name="car" ref="car"/> --><!--等同于-->

<property name=*"car"*>

<ref bean=*"car"*/>

</property>

</bean>

第三步：使用SpringTest.java测试：

@Test

**public** **void** test(){

//spring容器

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

//获取人

Person person=(Person)applicationContext.getBean("person");

System.*out*.println(person);

}

【扩展】

1. <ref>标签的用法：

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

<!--等同于-->

<property name=*"car"*>

<ref bean=*"car"*/>

</property>

### p名称空间的使用-了解

什么是名称空间？

作用：Schema区分同名元素。（有点类似于java的包）



回顾：Xmlns没有前缀是默认的名称空间。

为简化XML文件的配置，Spring2.5版本开始引入了一个新的p名称空间。简单的说，它的作用是为了简化setter方法属性依赖注入配置的，它不是真正的名称空间。

它的使用方法：

p:<属性名>="xxx" 引入常量值

p:<属性名>\_ref="xxx" 引用其它Bean对象

操作步骤：

第一步：引入p名称空间

|  |
| --- |
| <beans xmlns="http://www.springframework.org/schema/beans"  **xmlns:p="http://www.springframework.org/schema/p"**  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"> |

第二步：将<property> 子元素 简化为 元素的属性注入

<!-- 使用p名称空间简化setter方法属性注入 -->

<!--

p:name：简单数据类型的属性注入

P:car-ref：复杂数据类型（bean）的属性注入

-->

<bean id=*"person2"* class=*"cn.itcast.spring.e\_xmlpropertydi.Person"* p:id=*"1002"* p:name=*"关羽"* p:car-ref=*"car"*/>

**第三步：测试**

@Test

**public** **void** test(){

//spring容器

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

Person person2=(Person)applicationContext.getBean("person2");

System.*out*.println(person2);

}

配置时不需要<property > 子元素，简化了配置 .

### spEL表达式的使用 –会用即可

spEL（Spring Expression Language）是一种表达式语言，它是**spring3.x版本的新特性。**

它的作用是：支持在运行时操作和查询对象，其语法类似统一的EL语言，但是SpEL提供了额外的功能，功能更强大。

什么是EL、OGNL、spEL？

EL：操作servlet相关的一些对象和相关的值

OGNL：主要操作struts2值栈

spEL：操作bean相关的

语法： #{…} , 引用另一个Bean 、属性、 方法 , 运算

SpEL表达式的使用功能比较多，Bean操作相关的通常有：

* #{beanid} 引用Bean(具体对象)
* #{beanId.属性} 引用Bean的属性
* #{beanId.方法(参数)} 调用Bean的方法

案例一：配置applicationContext.xml

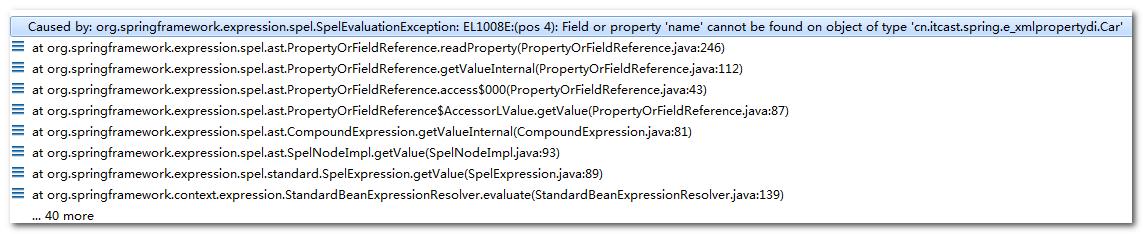
<!-- spEL的使用 -->

<!-- #{person.id} 相当于调用了person的getId()方法 -->

<bean id=*"person3"* class=*"cn.itcast.spring.e\_xmlpropertydi.Person"*

p:id=*"#{1+1}"* p:name=*"#{person.name.toUpperCase()}"* p:car=*"#{car}"*></bean>

如果抛出异常：



需要在Person对象中调用get方法，获取属性值，然后赋值到Person对象name的属性中。

**public** Integer getId() {

**return** id;

}

**public** String getName() {

**return** name;

}

**public** Car getCar() {

**return** car;

}

@Override

**public** String toString() {

**return** "Person [id=" + id + ", name=" + name + ", car=" + car + "]";

}案例二：配置applicationContext.xml

<!-- spEL表达式 -->

<!-- car.id相当于car.getId() -->

<bean id=*"person3"* class=*"cn.itcast.spring.e\_xmlpropertydi.Person"* p:id=*"#{1+1}"* p:name=*"#{'Jack'.toUpperCase()}"* p:car=*"#{car}"*/>

更多参考 ： \spring4\_day1\_课前资料\参考图书\Spring\_表达式语言.pdf

知识点梳理：

1. 复习Spring学习路线 （第一节）
2. IoC和DI概念区分
3. XML配置

实例化Bean方式 ,区分BeanFactory和FactoryBean

作用域 singleton和prototype

初始化和销毁 ------ 了解 BeanPostProcessor 后处理Bean

依赖注入（2种）：构造器注入 <constructor-arg> 、 属性setter注入 <property>

了解 p名称空间 spEL 表达式