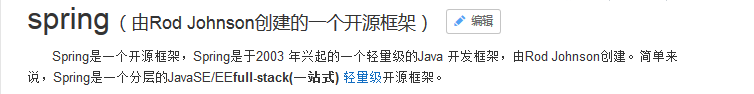
# Spring的概述

## 什么是Spring :



Spring是一个开源框架，Spring是于2003 年兴起的一个轻量级的Java 开发框架，由[**Rod Johnson**](http://baike.baidu.com/view/2192255.htm)在其著作《Expert one-on-one J2EE Development without EJB》，中阐述的部分理念和原型衍生而来。它是为了解决企业应用开发的复杂性而创建的。框架的主要优势之一就是其分层架构，分层架构允许使用者选择使用哪一个组件，同时为 [J2EE](http://baike.baidu.com/view/1507.htm) 应用程序开发提供集成的框架。Spring使用基本的[JavaBean](http://baike.baidu.com/view/28155.htm)来完成以前只可能由EJB完成的事情。然而，Spring的用途不仅限于[服务器](http://baike.baidu.com/view/899.htm)端的开发。从简单性、可测试性和松耦合的角度而言，任何Java应用都可以从Spring中受益。Spring的核心是[控制反转](http://baike.baidu.com/view/1486379.htm)（[IoC](http://baike.baidu.com/view/146665.htm)）和面向切面（[AOP](http://baike.baidu.com/view/73626.htm)）。简单来说，**Spring是一个分层的JavaSE/EEfull-stack(一站式)** [**轻量级**](http://baike.baidu.com/subview/1318763/16205192.htm)**开源框架。**Rod Johnson成为一个改变Java世界的大师级人物。

**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.

Expert One-to-One J2EE Design and Development :J2EE的设计和开发:(2002.EJB)

对Java EE 系统框架臃肿、低效、脱离现实的种种现状提出了质疑，并积极寻求探索革新

Expert One-to-One J2EE Development without EJB :J2EE不使用EJB的开发.

## Spring的版本

**Spring 3.X 和 Spring4.X**

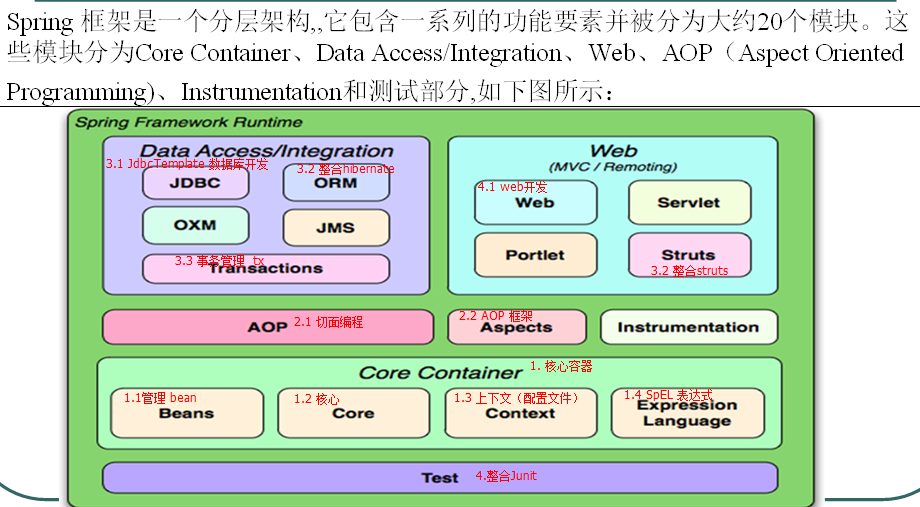
## spring核心

* Spring的核心是**控制反转（IoC）**和**面向切面（AOP）**

## spring优点

|  |
| --- |
| * 方便解耦，简化开发 （高内聚低耦合）   + Spring就是一个大工厂（容器），可以将所有对象创建和依赖关系维护，交给Spring管理   + spring工厂是用于生成bean * 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 * 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 * 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 * 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 * 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体系结构



**核心容器：beans、core、context、expression**

# 入门案例：IoC（）【掌握】

IOC:Inversion of Control的缩写

#### 步骤一:下载Spring的开发包:

官网：http://spring.io/

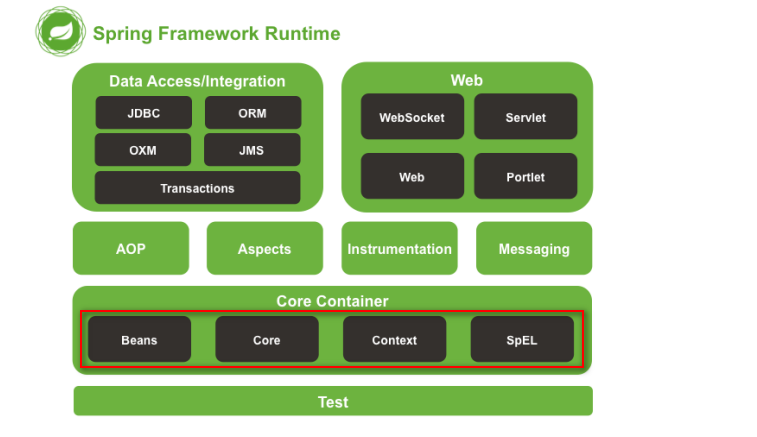
下载地址：http://repo.springsource.org/libs-release-local/org/springframework/spring解压:(Spring目录结构:)

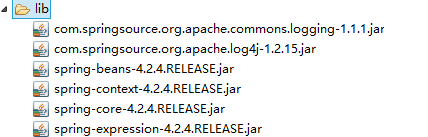
\* docs :API和开发规范.

\* libs :jar包和源码.

\* schema :约束.

#### 步骤二:创建web项目,引入Spring的开发包:





4个核心包2个日志包

#### 步骤三创建目标类

* 提供UserService接口和实现类
* 获得UserService实现类的实例

之前开发中，直接new一个对象即可。

学习spring之后，将由Spring创建对象实例--> IoC 控制反转（Inverse of Control）

之后需要实例对象时，从spring工厂（容器）中获得，需要将实现类的全限定名称配置到xml文件中

|  |
| --- |
| **public** **interface** UserService {    **public** **void** addUser();  } |
| **public** **class** UserServiceImpl **implements** UserService {  @Override  **public** **void** addUser() {  System.*out*.println("add user");  }  } |

#### 步骤四编写配置文件

**引入相关配置文件: applicationContext.xml和log4j.properties日志配置文件**

**Spring的配置文件：applicationContext.xml：**

* 位置：任意，开发中一般在classpath下（src）
* 名称：任意，开发中常用applicationContext.xml
* 内容：添加schema约束

|  |
| --- |
| <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=*"userServiceId"* class=*"com.qf.ioc.userServiceImpl"></bean>*  </beans> |

#### 步骤五编写测试类

**ApplicationContext接口（了解）**

\* 使用ApplicationContext工厂的接口，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

@Test

// Spring的方式：

**public** **void** demo2(){

// 创建Spring的工厂类：

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

//获得内容 --不需要自己new，都是从spring容器获得

UserService userService = (UserService) applicationContext.getBean("userServiceId");

userService.addUser();

}

注意：Spring工厂默认调用无参数的构造函数来实例化对象。

# 入门案例：DI【掌握】

* DI Dependency Injection ,依赖注入

is a ：是一个，继承。

has a：有一个，成员变量，依赖。

class B {

private A a; //B类依赖A类

}

依赖：一个对象需要使用另一个对象

注入：通过setter方法进行另一个对象实例设置。

* 例如：

class BookServiceImpl{

//之前开发：接口 = 实现类 （service和dao耦合）

//private BookDao bookDao = new BookDaoImpl();

//spring之后 （解耦：service实现类使用dao接口，不知道具体的实现类）

private BookDao bookDao;

setter方法

}

模拟spring执行过程

创建service实例：BookService bookService = new BookServiceImpl() -->IoC

创建dao实例：BookDao bookDao = new BookDaoImple() -->IoC

将dao设置给service：bookService.setBookDao(bookDao); -->DI

### dao

|  |
| --- |
| **public** **interface** BookDao {    **public** **void** addBook();  } |
| **public** **class** BookDaoImpl **implements** BookDao {  @Override  **public** **void** addBook() {  System.*out*.println(" add book");  }  } |

### Service

|  |
| --- |
| **public** **interface** BookService {  **public** **abstract** **void** addBook();  } |
| **public** **class** BookServiceImpl **implements** BookService {    // 方式1：之前，接口=实现类  // private BookDao bookDao = new BookDaoImpl();  // 方式2：接口 + setter  **private** BookDao bookDao;  **public** **void** setBookDao(BookDao bookDao) {  **this**.bookDao = bookDao;  }    @Override  **public** **void** addBook(){  **this**.bookDao.addBook();  }  } |

### 配置文件

|  |
| --- |
| <!-- 创建service -->  <bean id=*"bookServiceId"* class=*"com.qf.BookServiceImpl"*>  <property name=*"bookDao"* ref=*"bookDaoId"*></property>  </bean>    <!-- 创建dao实例 -->  <bean id=*"bookDaoId"* class=*"com.qf. di.BookDaoImpl"></bean>* |

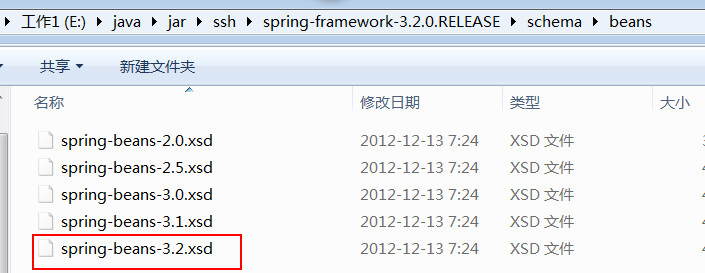
### 测试

|  |
| --- |
| @Test  **public** **void** demo01(){  //从spring容器获得  String xmlPath = "com/qf/di/beans.xml";  ApplicationContext applicationContext = **new** ClassPathXmlApplicationContext(xmlPath);  BookService bookService = (BookService) applicationContext.getBean("bookServiceId");    bookService.addBook();    } |

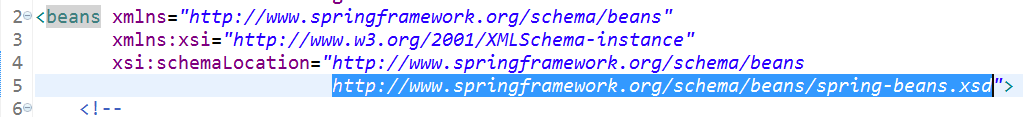
# schema xml提示

## 步骤一：确定xsd文件位置

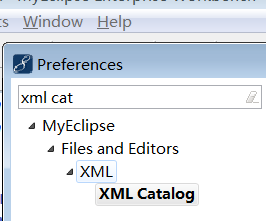
spring-framework-3.2.0.RELEASE\schema\beans



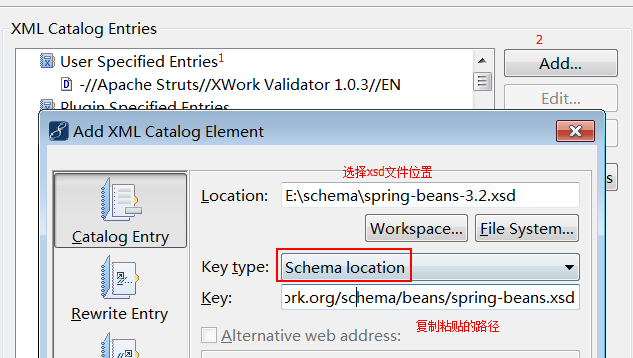
## 步骤二：复制路径



## 步骤三：搜索“xml catalog”

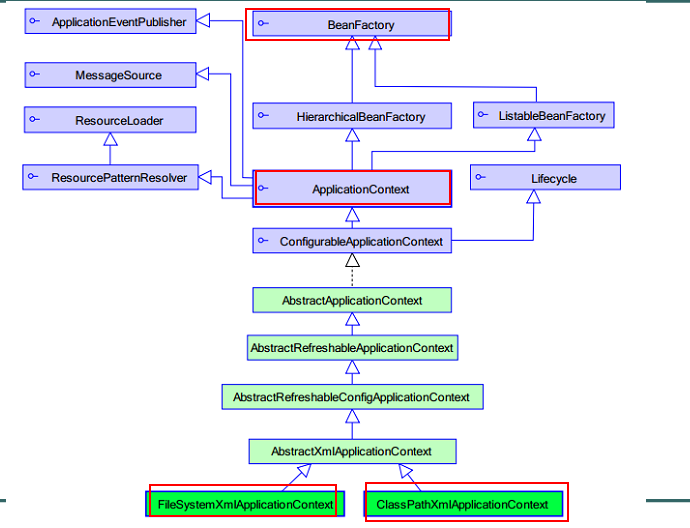


## 步骤四：添加约束提示



# 核心API

* api整体了解，之后不使用，在学习过程需要。



* BeanFactory ：这是一个工厂，用于生成任意bean。

采取延迟加载，第一次getBean时才会初始化Bean

* ApplicationContext：是BeanFactory的子接口，功能更强大。（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）。当配置文件被加载，就进行对象实例化。

ClassPathXmlApplicationContext 用于加载classpath（类路径、src）下的xml

加载xml运行时位置 --> /WEB-INF/classes/...xml

FileSystemXmlApplicationContext 用于加载指定盘符下的xml

加载xml运行时位置 --> /WEB-INF/...xml

通过java web ServletContext.getRealPath() 获得具体盘符

# Spring的相关配置:

id :Bean起个名字. 在约束中采用ID的约束:唯一. 必须以字母开始，可以使用字母、数字、连字符、下划线、句话、冒号 id:不能出现特殊字符.

<bean id=”bookAction”>

# 装配Bean 基于XML

## 实例化方式

* 3种bean实例化方式：默认构造、静态工厂、实例工厂

### 默认构造

|  |
| --- |
| <bean id="" class=""> 必须提供默认构造 |

### 静态工厂

* 常用与spring整合其他框架（工具）
* 静态工厂：用于生成实例对象，所有的方法必须是static

|  |
| --- |
| <bean id="" class="工厂全限定类名" factory-method="静态方法"> |

工厂

|  |
| --- |
| **public** **class** MyBeanFactory {    /\*\*  \* 创建实例  \* **@return**  \*/  **public** **static** UserService createService(){  **return** **new** UserServiceImpl();  }  } |

### 实例工厂

|  |
| --- |
| <!-- 用spring创建工厂实例 单例的 -->  <bean id="facId" class="com.qf.c\_xml.factory.MyFactory" scope="singleton"></bean>      <!-- 配置bean来获得service实例 -->  <bean id="userServieId" factory-bean="facId" factory-method="getInstance"></bean> |

## Spring框架中<bean>标签的配置

* 作用域：用于确定spring创建bean实例个数



* 取值：

singleton 单例，默认值。

prototype 多例，每执行一次getBean将获得一个实例。例如：struts整合spring，配置action多例。

* 配置信息

|  |
| --- |
| <bean id="" class="" scope=""> |

|  |
| --- |
| 1. id属性的区别  \* id -- Bean起个名字，在约束中采用ID的约束，唯一  \* 取值要求：必须以字母开始，可以使用字母、数字、连字符、下划线、句话、冒号 id:不能出现特殊字符    2. class属性 -- Bean对象的全路径  3. 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项目中,每次HTTP请求都会创建一个新的Bean  \* session -- 应用在Web项目中,同一个HTTP Session 共享一个Bean  \* globalsession -- 应用在Web项目中,多服务器间的session    4. Bean对象的创建和销毁的两个属性配置（了解）  \* 说明：Spring初始化bean或销毁bean时，有时需要作一些处理工作，因此spring可以在创建和拆卸bean的时候调用bean的两个生命周期方法  \* init-method -- 当bean被载入到容器的时候调用init-method属性指定的方法  \* destroy-method -- 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） |

## 生命周期

### 初始化和销毁

* 目标方法执行前后执行后，将进行初始化或销毁。

|  |
| --- |
| <bean id="" class="" init-method="初始化方法名称" destroy-method="销毁的方法名称"> |

#### 目标类

|  |
| --- |
| **public** **class** UserServiceImpl **implements** UserService {  @Override  **public** **void** addUser() {  System.*out*.println("add user");  }    **public** **void** myInit(){  System.*out*.println("初始化");  }  **public** **void** myDestroy(){  System.*out*.println("销毁");  }  } |

#### spring配置

|  |
| --- |
| <!--  init-method 用于配置初始化方法,准备数据等  destroy-method 用于配置销毁方法,清理资源等  -->  <bean id=*"userServiceId"* class=*"com.qf.UserServiceImpl"*  init-method=*"myInit"* destroy-method=*"myDestroy"* ></bean> |

#### 测试

|  |
| --- |
| @Test  **public** **void** demo02() **throws** Exception{  //spring 工厂  String xmlPath = "com/qf/beans.xml";  ClassPathXmlApplicationContext applicationContext = **new** ClassPathXmlApplicationContext(xmlPath);  UserService userService = (UserService) applicationContext.getBean("userServiceId");  userService.addUser();      applicationContext.close();    } |

# 属性依赖注入

### 构造方法注入

#### 目标类

|  |
| --- |
| **public** **class** User {    **private** Integer uid;  **private** String username;  **private** Integer age;    **public** User(Integer uid, String username) {  **super**();  **this**.uid = uid;  **this**.username = username;  }    **public** User(String username, Integer age) {  **super**();  **this**.username = username;  **this**.age = age;  } |

#### spring配置

|  |
| --- |
| <!-- 构造方法注入  \* <constructor-arg> 用于配置构造方法一个参数argument  name ：参数的名称  value：设置普通数据  ref：引用数据，一般是另一个bean id值    index ：参数的索引号，从0开始 。如果只有索引，匹配到了多个构造方法时，默认使用第一个。  type ：确定参数类型  例如：使用名称name  <constructor-arg name="username" value="jack"></constructor-arg>  <constructor-arg name="age" value="18"></constructor-arg>  例如2：【类型type 和 索引 index】  <constructor-arg index="0" type="java.lang.String" value="1"></constructor-arg>  <constructor-arg index="1" type="java.lang.Integer" value="2"></constructor-arg>  -->  <bean id=*"userId"* class=*"com.qf.constructor.User"* >  <constructor-arg index=*"0"* type=*"java.lang.String"* value=*"1"*></constructor-arg>  <constructor-arg index=*"1"* type=*"java.lang.Integer"* value=*"2"*></constructor-arg>  </bean> |

### setter方法注入

|  |
| --- |
| <!-- setter方法注入  \* 普通数据  <property name="" value="值">  等效  <property name="">  <value>值  \* 引用数据  <property name="" ref="另一个bean">  等效  <property name="">  <ref bean="另一个bean"/>    -->  <bean id=*"personId"* class=*"com.qf.setter.Person"*>  <property name=*"pname"* value=*"小苍"*></property>  <property name=*"age"*>  <value>1234</value>  </property>    <property name=*"homeAddr"* ref=*"homeAddrId"*></property>  <property name=*"companyAddr"*>  <ref bean=*"companyAddrId"*/>  </property>  </bean>    <bean id=*"homeAddrId"* class=*"com.qf.setter.Address"*>  <property name=*"addr"* value=*"北京"*></property>  <property name=*"tel"* value=*"****110*** *"*></property>  </bean>  <bean id=*"companyAddrId"* class=*"com.qf.setter.Address"*>  <property name=*"addr"* value=*"北京海淀"*></property>  <property name=*"tel"* value=*"120"*></property>  </bean> |

### P命令空间[了解]

**名称空间p的属性注入的方式:Spring2.x版本后提供的方式.**

* 对“setter方法注入”进行简化，替换<property name="属性名">，而是在

<bean p:属性名="普通值" p:属性名-ref="引用值">

* p命名空间使用前提，必须添加命名空间

<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>">

|  |
| --- |
| <bean id=*"personId"* class=*"com.qf.p.Person"*  p:pname=*"小泽"* p:age=*"22"*  p:homeAddr-ref=*"homeAddrId"* p:companyAddr-ref=*"companyAddrId"*>  </bean>    <bean id=*"homeAddrId"* class=*"com.qf.p.Address"*  p:addr=*"东莞"* p:tel=*"666666"*>  </bean>  <bean id=*"companyAddrId"* class=*"com.qf.p.Address"*  p:addr=*"日本"* p:tel=*"333333"*>  </bean> |

### SpEL的方式的属性注入[了解]

**Spring3.x版本后提供的方式.**

* 对<property>进行统一编程，所有的内容都使用value

<property name="" value="#{表达式}">

#{123}、#{'jack'} ： 数字、字符串

#{beanId} ：另一个bean引用

#{beanId.propName} ：操作数据

#{beanId.toString()} ：执行方法

#{T(类).字段|方法} ：静态方法或字段

|  |
| --- |
| <!--  <property name="cname" value="#{'jack'}"></property>  <property name="cname" value="#{customerId.cname.toUpperCase()}"></property>  通过另一个bean，获得属性，调用的方法  <property name="cname" value="#{customerId.cname?.toUpperCase()}"></property>  ?. 如果对象不为null，将调用方法  -->  <bean id=*"customerId"* class=*"com.qf.spel.Customer"* >  <property name=*"cname"* value=*"#{customerId.cname?.toUpperCase()}"*></property>  <property name=*"pi"* value=*"#{T(java.lang.Math).PI}"*></property>  </bean> |

### 集合注入

|  |
| --- |
| <!--  集合的注入都是给<property>添加子标签  数组：<array>  List：<list>  Set：<set>  Map：<map> ，map存放k/v 键值对，使用<entry>描述  Properties：<props> <prop key=""></prop> 【】    普通数据：<value>  引用数据：<ref>  -->  <bean id=*"collDataId"* class=*"com.qf.coll.CollData"* >  <property name=*"arrayData"*>  <array>  <value>hello wolrd</value>  <value>jack</value>  <value>rose</value>  </array>  </property>    <property name=*"listData"*>  <list>  <value>芙蓉</value>  <value>菊花</value>  <value>如花</value>  </list>  </property>    <property name=*"setData"*>  <set>  <value>芙蓉</value>  <value>菊花</value>  <value>如花</value>  </set>  </property>    <property name=*"mapData"*>  <map>  <entry key=*"jack"* value=*"杰克"*></entry>  <entry key=*"rose"* value=*"肉丝"*></entry>  </map>  </property>    <property name=*"propsData"*>  <props>  <prop key=*"高富帅"*>冠希哥</prop>  <prop key=*"白富美"*>阿凤姐</prop>  <prop key=*"男屌丝"*>菊花信</prop>  </props>  </property>  </bean> |

# Spring的分配置文件的开发

一种:创建工厂的时候加载多个配置文件:

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

二种:在一个配置文件中包含另一个配置文件：

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

# 技术分析之Spring框架的IOC功能之注解的方式

## Spring框架的IOC之注解方式的快速入门

### 1. 步骤一：导入注解开发所有需要的jar包

引入IOC容器必须的6个jar包

多引入一个：Spring框架的AOP的jar包，spring-aop的jar包

### 2. 步骤二：创建对应的包结构，编写Java的类

UserService -- 接口

UserServiceImpl -- 具体的实现类

### 3. 步骤三：引入约束

在src的目录下，创建applicationContext.xml的配置文件，然后引入约束。注意：因为现在想使用注解的方式，那么引入的约束发生了变化

\* 需要引入context的约束，具体的约束如下

<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" 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"> <!-- bean definitions here -->

</beans>

### 4. 步骤四：在applicationContext.xml配置文件中开启组件扫描

Spring的注解开发:组件扫描

<context:component-scan base-package="com.qf.demo1"/>

注意：可以采用如下配置

<context:component-scan base-package="com.qf"/> 这样是扫描com.qf包下所有的内容

5. 步骤五：在UserServiceImpl的实现类上添加注解

@Component(value="userService") -- 相当于在XML的配置方式中 <bean id="userService" class="...">

6. 步骤六：编写测试代码

public class SpringDemo1 {

@Test

public void run1(){

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

UserService us = (UserService) ac.getBean("userService");

us.save();

}

}

## Spring框架中Bean管理的常用注解

**1. @Component:组件.(作用在类上)**

**2. Spring中提供@Component的三个衍生注解:(功能目前来讲是一致的)**

@Controller -- 作用在WEB层

@Service -- 作用在业务层

@Repository -- 作用在持久层

说明：这三个注解是为了让标注类本身的用途清晰，Spring在后续版本会对其增强

3. 属性注入的注解(说明：使用注解注入的方式,可以不用提供set方法)

如果是注入的普通类型，可以使用value注解

@Value -- 用于注入普通类型

如果注入的是对象类型，使用如下注解

@Autowired -- 默认按类型进行自动装配

如果想按名称注入

@Qualifier -- 强制使用名称注入

@Resource -- 相当于@Autowired和@Qualifier一起使用

强调：Java提供的注解

属性使用name属性

**Bean的作用范围和生命周期的注解**

1. Bean的作用范围注解

注解为@Scope(value="prototype")，作用在类上。值如下：

singleton -- 单例，默认值

prototype -- 多例

2. Bean的生命周期的配置（了解）

注解如下：

@PostConstruct -- 相当于init-method

@PreDestroy -- 相当于destroy-method