**Spring第一天**

# 1：开启spring之旅

## （1）：spring介绍

Spring是一个非常活跃的开源框架；它是一个基于IOC和AOP来构架多层JavaEE系统的框架,它的主要目地是简化企业开发.

Spring以一种非侵入式的方式来管理你的代码，Spring提倡”最少侵入”，这也就意味着你可以适当的时候安装或卸载Spring

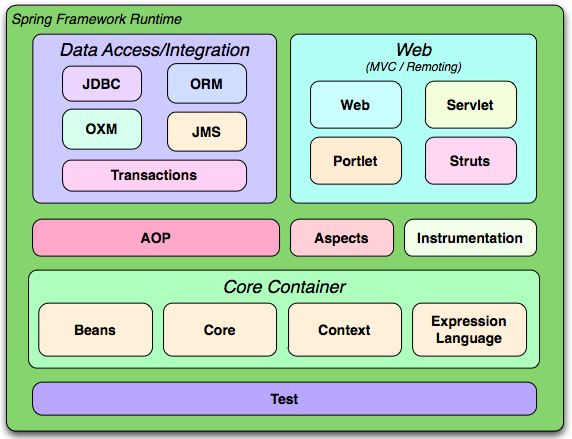
官网：http://www.springsource.org/

### 【1】项目中使用spring框架的优势：

* 方便解耦，简化开发
  + Spring就是一个大工厂，可以将所有对象创建和依赖关系维护，交给Spring管理
* 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Junit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

### 【2】Spring模块介绍

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,如下图所示



### （2）：导入spring相关的jar包

Spring开发需要哪些jar包

\* spring-beans-4.3.6.jar

所有应用都要用到的，它包含访问配置文件、创建和管理bean

以及进行Inversion of Control / Dependency Injection（IoC/DI）操作相关的所有类

\* spring-context-4.3.6.jar

Spring提供在基础IoC功能上的扩展服务，此外还提供许多企业级服务的支持,

如邮件服务、任务调度、JNDI定位、EJB集成、远程访问、缓存以及各种视图层框架的封装等。

\* spring-core-4.3.6.jar

包含Spring框架基本的核心工具类，Spring其它组件要都要使用到这个包里的类,是其它组件的基本核心。

\* spring-expression-4.3.6.jar

Spring表达式语言

\* commons-logging-1.2.jar 第三方的主要用于处理日志

### （3）：spring的核心容器

默认的写法：**applicationContext.xml**（默认文件放置到src下，但是也可以更改路径）

可以自定义命名：beans.xml

在beans.xml中引入约束的文件

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

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

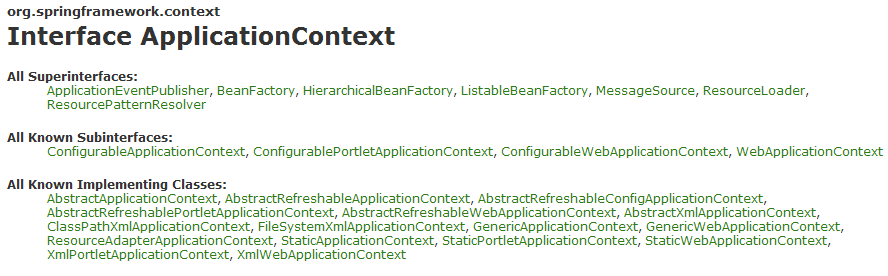
xsi:schemaLocation=*"http://www.springframework.org/schema/beans*

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

</beans>

spring加载容器的方式

核心对象：



/\*\*

\* new ClassPathXmlApplicationContext("beans.xml"):表示类路径下的beans.xml

\* new ClassPathXmlApplicationContext("com/offcn/ioc/beans.xml"):表示类路径下的com.offcn.ioc包下的beans.xml

\*/

ApplicationContext ac =

**new** ClassPathXmlApplicationContext("com/offcn/ioc/beans.xml");

### （4）：控制反转（IOC）

（1）概念：**所谓控制反转就是应用本身不负责依赖对象的创建及维护，依赖对象的创建及维护是由外部容器负责的。**这样控制权就由应用转移到了外部容器，控制权的转移就是所谓反转，目的是为了获得更好的扩展性和良好的可维护性。

（2）代码：

存在一个Boy对象：

**public** **class** Boy {

**public** **void** display(){

System.*out*.println("我是一个优秀的Boy！");

}

}

（3）在spring容器中定义：

<?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-4.3.xsd"*>

<!--

指定在容器中创建一个对象

\* id：表示在spring容器中的惟一标识

\* class：表示需要构造的类的全路径

-->

**<bean id=*"boy"* class=*"com.offcn.ioc.Boy"*></bean>**

</beans>

（4）使用App进行测试：

package com.offcn.ioc;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public class App {

public static void main(String[] args) {

// Boy boy = new Boy();

// boy.display();

/\*\*使用spring容器构造boy对象\*/

//使用applicationContext加载spring容器

/\*\*

\* new ClassPathXmlApplicationContext("beans.xml"):表示类路径下的beans.xml

\* new ClassPathXmlApplicationContext("com/offcn/ioc/beans.xml"):表示类路径下的com.offcn.ioc包下的beans.xml

\*/

**ApplicationContext ac = new ClassPathXmlApplicationContext("com/offcn/ioc/beans.xml");**

//获取容器下的id指定的对象

Boy boy = (Boy) ac.getBean("boy");

boy.display();

}

}

### （5）：依赖注入（DI）

概念：所谓依赖注入就是指：在运行期，由外部容器动态地将依赖对象注入到另一个对象的组件中。

（1）类中定义：创建2个对象，一个Boy对象和一个Girl对象

* Boy.java

**package** com.offcn.di;

**public** **class** Boy {

**public** **void** display(){

System.*out*.println("我是一个优秀的Boy！");

}

}

* Girl.java

**public** **class** Girl {

**Boy boy;**

**//完成依赖注入，必须有set方法**

**public void setBoy(Boy boy) {**

**this.boy = boy;**

**}**

**public** **void** kiss(){

boy.display();

System.*out*.println("执行Kiss的方法！");

}

}

（2）在spring容器中定义：

<?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-4.3.xsd"*>

<!--

指定在容器中创建一个对象

\* id：表示在spring容器中的惟一标识

\* class：表示需要构造的类的全路径

-->

<bean id=*"boy"* class=*"com.offcn.di.Boy"*></bean>

<bean id=*"girl"* class=*"com.offcn.di.Girl"*>

<!--

将spring容器中创建的boy对象传递给Girl对象中的set方法

\* name:在Girl对象中的set方法的属性名称

\* ref：将容器创建的对象注入到set方法中

-->

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

</bean>

</beans>

（3）最后使用app进行测试：

package com.offcn.di;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public class App {

public static void main(String[] args) {

// Boy boy = new Boy();

// boy.display();

//使用applicationContext加载spring容器

**ApplicationContext ac = new ClassPathXmlApplicationContext("com/offcn/di/beans.xml");**

//获取容器下的id指定的对象

Girl girl = (Girl) ac.getBean("girl");

girl.kiss();

}

}

### （6）：加载spring容器2种方式

/\*\*

**\* 方法一:**

**在类路径下寻找配置文件来实例化容器**

**\*/**

//使用applicationContext加载spring容器

//如果定义成数组一次可以加载多个spring容器

**ApplicationContext ac = new ClassPathXmlApplicationContext(new String[]{"com/offcn/di/beans.xml"});**

/\*\*

\* 方法二:

在文件系统路径下寻找配置文件来实例化容器

\*/

ApplicationContext ac = new FileSystemXmlApplicationContext(new String[]{"D:\\workspaceSpring\\SpringBean\\src\\com\\offcn\\di\\beans.xml"});

## 2：装配bean

### （1）：三种实例化bean的方式

定义一个接口和实现类

* 接口：

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

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

}

* 实现类

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

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

System.*out*.println("访问UserServiceImpl类的saveUser()的方法");

}

}

#### （1）使用接口和类，直接在spring容器中创建一个类，使用接口进行操作

* 1：在容器中定义：

**<bean id="userService" class="com.offcn.c\_class.UserServiceImpl"></bean>**

* 2：App.java进行测试

**public** **class** App {

**public** **static** **void** main(String[] args) {

ApplicationContext ac = **new** ClassPathXmlApplicationContext("com/offcn/c\_class/beans.xml");

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

userService.saveUser();

}

}

注意：**class一定要是类，不能使用接口，因为接口不能被实例化**。

#### （2）使用静态工厂方法实例化(简单工厂模式)

* 1：Spring容器中定义：

<bean id="userObject" class="com.offcn.d\_staticMethod.UserObject" **factory-method="createUser"**>

</bean>

* 2： UserObject对象中提供一个静态的的方法，用来创建对象

**public** **class** UserObject {

**//静态的方法**

**public static IUserService createUser(){**

**return new UserServiceImpl();**

**}**

}

* 3：App类进行测试

**public** **class** App {

**public** **static** **void** main(String[] args) {

ApplicationContext ac = **new** ClassPathXmlApplicationContext("com/offcn/d\_staticMethod/beans.xml");

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

userService.saveUser();

}

}

#### （3）使用实例工厂方法实例化(工厂方法模式):

* 1：Spring容器中定义：

<bean id="userObject" class="com.offcn.e\_method.UserObject"></bean>

<bean id="user" **factory-bean="userObject" factory-method="createUser"**></bean>

* 2：UserObject中定义

**public** **class** UserObject {

//非静态的方法

**public IUserService createUser(){**

**return new UserServiceImpl();**

**}**

}

* 3：使用App调用的形式

**public** **class** App {

**public** **static** **void** main(String[] args) {

ApplicationContext ac = **new** ClassPathXmlApplicationContext("com/offcn/e\_method/beans.xml");

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

userService.saveUser();

}

}

### （2）：bean 的作用域

在spring容器中存在1个scope的属性，例如：

<bean id="userService" class="com.offcn.f\_scope.UserServiceImpl" **scope="singleton"**></bean>

* **singleton(单例：默认值)**

表示创建UserServiceImpl对象，容器只创建一次

\* lazy-init属性（只针对单例有效）：

如果在spring容器中设置：

<bean id="userService" class="com.offcn.f\_scope.UserServiceImpl" scope="singleton" **lazy-init="true"**></bean>

表示此时在使用

ApplicationContext ac = new ClassPathXmlApplicationContext("com/offcn/f\_scope/beans.xml");

加载spring容器，将不会执行UserServiceImpl类的构造方法。

当执行到ac.getBean("userService");才会执行UserServiceImpl类的方法。大家可以理解成懒加载对象

如果在spring容器中设置：

<bean id="userService" class="com.offcn.f\_scope.UserServiceImpl" scope="singleton" **lazy-init="false"**></bean>

此时表示只要调用加载spring容器的代码，就会执行UserServiceImpl类的构造方法。

\* 如果在beans下配置：default-lazy-init="true"，此时对beans下的所有bean节点都有效，如果都进行设置，局部将覆盖全局。

* **prototype （多例）**

表示每次加载ac.getBean("userService");都会执行UserServiceImpl的构造方法，容器此时可以创建多个对象。

**注意：用途：如果通过spring创建struts的Action实例的时候，就必须指定prototype类型，因为struts2是多实例多线程，否则struts2的值栈会出现问题。**

测试代码

* 接口：

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

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

}

* 实现类

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

//构造方法（单例，只执行一次，多例，执行多次）

**public** UserServiceImpl(){

System.*out*.println("这是UserServiceImpl的构造方法！");

}

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

System.*out*.println("访问UserServiceImpl类的saveUser()的方法！");

}

}

* Spring容器（beans.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-4.3.xsd"* default-lazy-init=*"true"*>

<bean id=*"userService"* class=*"com.offcn.f\_scope.UserServiceImpl"* scope=*"singleton"* lazy-init=*"false"*></bean>

</beans>

* App.java（测试类）

**public** **class** App {

**public** **static** **void** main(String[] args) {

ApplicationContext ac = **new** ClassPathXmlApplicationContext("com/offcn/f\_scope/beans.xml");

IUserService userService1 = (IUserService) ac.getBean("userService");

userService1.saveUser();

IUserService userService2 = (IUserService) ac.getBean("userService");

userService2.saveUser();

IUserService userService3 = (IUserService) ac.getBean("userService");

userService3.saveUser();

}

}

### （3）：创建bean的生命周期

Spring管理bean的生命周期的行为，主要为如下两个时机：

注入依赖关系之后：

--使用init-method属性

--实现InitializingBean接口（入侵性的代码设计）

销毁bean之前：

--使用destory-method属性

--实现DisposeableBean接口（入侵性的代码设计）

## 3：依赖注入

\* **手工装配**

\* 使用xml配置文件

\* 使用属性setter方法注入

\* 使用构造器注入

\* 使用注解

\* autowired注解

\* resource注解

**\* 自动装配**

**\* byType：**按类型装配

**\* byName**：按名称装配

**\* constructor装配**，

**\* autodetect** 不确定装配。

注入依赖对象可以采用手工装配或自动装配，在实际应用中建议使用手工装配，因为自动装配会产生未知情况,开发人员无法预见最终的装配结果。

### （1）手工装配：通过setter方法注入依赖

**通过setter方法注入依赖**

<bean>元素的< property >子元素指明了使用它们的set方法来注入。可以注入任何东西，从基本类型到集合类，甚至是应用系统的bean。

#### 【1】简单bean配置

配置bean的简单属性，基本数据类型和String。

<bean id="personService" class="com.offcn.bean.impl.PersonServiceImpl">

<!-- 基本类型,string类型 -->

<property name="age" value="20"></property>

<property name="name" value="张无忌"></property>

</bean>

#### 【2】引用其它bean（外部bean）

<bean id="person" class="com.offcn.bean.Person" />

<bean id="personService" class="com.offcn.bean.impl.PersonServiceImpl">

<!-- 引用类型 -->

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

</bean>

#### 【3】引用其它bean（内部bean）

<bean id="personService" class="com.offcn.bean.impl.PersonServiceImpl">

<!-- 内部bean注入 -->

<property name="personClass">

**<bean class="com.offcn.bean.PersonClass" />**

</propert>

</bean>

这种方式的缺点是你无法在其它地方重用这个personClass实例，原因是它是专门为personService而用。

#### 【4】装配集合List和数组

若bean的属性是集合类型，按如下处理：

**<!-- 装配list -->**

**<property name="lists">**

**<list>**

**<value>list1</value>**

**<value>list2</value>**

**<ref bean="person"/>**

**</list>**

**</property>**

**<!-- 装配数组 -->**

**<property name="obj">**

**<array>**

**<value>obj1</value>**

**<value>obj2</value>**

**<ref bean="person"/>**

**</array>**

</property>

#### 【5】装配set

**<!-- 装配set -->**

**<property name="sets">**

**<set>**

**<value>set1</value>**

**<value>set2</value>**

**<ref bean="person"/>**

**</set>**

**</property>**

set使用方法和list一样，不同的是对象被装配到set中，而list是装配到List或数组中装配。

#### 【6】装配map：

**<!-- 装配map-->**

**<property name="maps">**

**<map>**

**<entry key="01">**

**<value>map01</value>**

**</entry>**

**<entry key="02">**

**<value>map02</value>**

**</entry>**

**</map>**

**</property>**

简化为：

<property name="maps">

<map>

<entry key="01" value="map01"></entry>

<entry key="02" value="map02"></entry>

</map>

</property>

map中的<entry>的数值和<list>以及<set>的一样，可以使任何有效的属性元素，**需要注意的是key值必须是String的。**

#### 【7】装配集合Property

**<!--装配Properties -->**

**<property name="props">**

**<props>**

**<prop key="01">prop1</prop>**

**<prop key="02">prop2</prop>**

**</props>**

**</property>**

#### 【8】装配集合List（null）

**<!-- 装配null -->**

**<property name="listnull">**

**<null/>**

**</property>**

#### 【9】测试代码

* 接口

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

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

}

* 实现类

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

/\*\*基本数据类型和String\*/

**private** Integer uid;

**private** String uname;

/\*\*注入对象\*/

**private** Person person;

/\*\*list集合\*/

**private** List<Object> list;

/\*\*数组\*/

**private** Object [] arrays;

/\*\*set集合\*/

**private** Set<Object> set;

/\*\*map集合\*/

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

/\*\*Properties集合\*/

**private** Properties properties;

/\*\*集合属性\*/

**private** List<Object> nullList = **new** ArrayList<Object>();

**public** **void** setNullList(List<Object> nullList) {

**this**.nullList = nullList;

}

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

**this**.properties = properties;

}

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

**this**.map = map;

}

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

**this**.set = set;

}

**public** **void** setArrays(Object[] arrays) {

**this**.arrays = arrays;

}

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

**this**.list = list;

}

**public** **void** setPerson(Person person) {

**this**.person = person;

}

**public** **void** setUid(Integer uid) {

**this**.uid = uid;

}

**public** **void** setUname(String uname) {

**this**.uname = uname;

}

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

System.*out*.println("访问UserServiceImpl类的saveUser()的方法！");

System.*out*.println("uid："+uid);

System.*out*.println("uname："+uname);

System.*out*.println("person.getName:"+person.getName());

System.*out*.println("person.getAge:"+person.getAge());

**if**(list!=**null** && list.size()>0){

**for**(Object o:list){

System.*out*.println("list:"+o);

}

}

**if**(arrays!=**null** && arrays.length>0){

**for**(Object o:arrays){

System.*out*.println("array:"+o);

}

}

Iterator<Object> ite = set.iterator();

**while**(ite.hasNext()){

Object o = ite.next();

System.*out*.println("set:"+o);

}

System.*out*.println("map001:"+map.get("mapKey001"));

System.*out*.println("map002:"+map.get("mapKey002"));

System.*out*.println("map003:"+map.get("mapKey003"));

System.*out*.println("prop001:"+properties.get("prop001"));

System.*out*.println("prop002:"+properties.get("prop002"));

System.*out*.println("nullList:"+nullList);

}

}

* Spring容器

<?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-4.3.xsd"*>

<!-- 使用p:name="三丰" p:age="60"完成赋值 -->

<bean id=*"person"* class=*"com.offcn.h\_set.Person"* p:name=*"三丰"* p:age=*"60"*></bean>

<bean id=*"userService"* class=*"com.offcn.h\_set.UserServiceImpl"*>

<!-- 基本数据类型和String -->

<property name=*"uid"* value=*"10"*></property>

<property name=*"uname"* value=*"翠花"*></property>

<!-- 外部注入bean ，开发的时候，都会使用外部注入bean的方法，因为可以在每个bean中都可以使用ref的属性注入，对象-->

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

<!-- 内部注入bean

<property name="person">

<bean class="com.offcn.h\_set.Person"></bean>

</property>-->

<!-- A、装配List和数组: -->

<property name=*"list"*>

<list>

<value>list001</value>

<value>list002</value>

<ref bean=*"person"*/>

</list>

</property>

<property name=*"arrays"*>

<array>

<value>array001</value>

<value>array002</value>

<ref bean=*"person"*/>

</array>

</property>

<!-- B、 装配set： -->

<property name=*"set"*>

<set>

<value>set001</value>

<value>set002</value>

<ref bean=*"person"*/>

</set>

</property>

<!-- 装配集合 C、 装配map： -->

<property name=*"map"*>

<map>

<entry key=*"mapKey001"* value=*"mapValue001"*></entry>

<entry key=*"mapKey002"*>

<value>mapValue002</value>

</entry>

<entry key=*"mapKey003"*>

<ref bean=*"person"*/>

</entry>

</map>

</property>

<!-- D:装配Properties： -->

<property name=*"properties"*>

<props>

<prop key=*"prop001"*>propValue001</prop>

<prop key=*"prop002"*>propValue002</prop>

</props>

</property>

<!-- E:注入空值 -->

<property name=*"nullList"*>

<null/>

</property>

</bean>

</beans>

* 测试类App.java

**public** **class** App {

**public** **static** **void** main(String[] args) {

ApplicationContext ac = **new** ClassPathXmlApplicationContext("com/offcn/h\_set/beans.xml");

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

userService.saveUser();

}

}

* 创建javabean对象，进行注入

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

**private** String name = "翠山";

**private** Integer age = 28;

**public** String getName() {

**return** name;

}

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

**this**.name = name;

}

**public** Integer getAge() {

**return** age;

}

**public** **void** setAge(Integer age) {

**this**.age = age;

}

}

### （2）手工装配：通过构造器函数

#### 【1】通过参数的顺序：

**<constructor-arg index="0">**

**<value>张三</value>**

**</constructor-arg>**

**<constructor-arg index="1">**

**<value>56</value>**

**</constructor-arg>**

#### 【2】通过参数的类型：

**<!-- 通过参数的类型 -->**

**<constructor-arg type="java.lang.Integer">**

**<value>56</value>**

**</constructor-arg>**

**<constructor-arg type="java.lang.String">**

**<value>张三</value>**

**</constructor-arg>**

#### 【3】测试代码：

* 接口：

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

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

}

* 实现类

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

**private** Integer id;

**private** String name;

**private** String education;

//使用构造函数注入

**public** UserServiceImpl(String name,Integer id){

System.*out*.println("先String，再Integer");

**this**.id = id;

**this**.name = name;

}

//使用构造函数注入

**public** UserServiceImpl(Integer id,String name){

System.*out*.println("先Integer，再String");

**this**.id = id;

**this**.name = name;

}

**public** UserServiceImpl(String name,String education){

System.*out*.println("先String，再String");

**this**.name = name;

**this**.education = education;

}

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

System.*out*.println("访问UserServiceImpl类的saveUser()的方法！");

System.*out*.println("id:"+id);

System.*out*.println("name:"+name);

System.*out*.println("education:"+education);

}

}

* Spring容器

<?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-4.3.xsd"*>

<bean id=*"userService"* class=*"com.offcn.i\_constructor.UserServiceImpl"*>

<!-- 方式1，使用参数的位置绑定，index=0，表示第一个参数

问题：如果一个类存在多个构造器，此时注入的时候，可能注入的构造器不是我们想需要的构造器（先查找String和String类型）

<constructor-arg index="0">

<value>10</value>

</constructor-arg>

<constructor-arg index="1">

<value>如花</value>

</constructor-arg>

-->

<!--

方式2：使用参数的类型绑定

问题：如果一个类存在多个构造器而且参数的数量是相同的，使用构造器完成注入的时候，注入的构造器也可能不是我们想注入的构造器

<constructor-arg type="java.lang.Integer">

<value>10</value>

</constructor-arg>

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

<value>如花</value>

</constructor-arg> -->

<!-- 要求，index和type一起使用，解决上面的问题 -->

<constructor-arg type=*"java.lang.Integer"* index=*"0"*>

<value>10</value>

</constructor-arg>

<constructor-arg type=*"java.lang.String"* index=*"1"*>

<value>如花</value>

</constructor-arg>

</bean>

</beans>

* 测试代码App.java

**public** **class** App {

**public** **static** **void** main(String[] args) {

ApplicationContext ac = **new** ClassPathXmlApplicationContext("com/offcn/i\_constructor/beans.xml");

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

userService.saveUser();

}

}

### （3）手工装配：bean之间的关系（了解知识）

#### 继承

* 继承，如果多个Bean具有相同的方法和属性，则可以引入父类Bean，配置父子bean关系

<!-- ：继承

<bean id="cat1" class="com.offcn.j\_relation.Cat">

<property name="name" value="黑猫警长"></property>

</bean>

<bean id="cat2" class="com.offcn.j\_relation.Cat">

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

</bean>

可以修改成：

<bean id="cat" class="com.offcn.j\_relation.Cat" abstract="true"></bean>

<bean id="cat1" parent="cat">

<property name="name" value="黑猫警长"></property>

</bean>

<bean id="cat2" parent="cat">

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

</bean>-->

* Spring容器

<?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-4.3.xsd"*>

<!-- 1：继承

<bean id="cat1" class="com.offcn.j\_relation.Cat">

<property name="name" value="黑猫警长"></property>

</bean>

<bean id="cat2" class="com.offcn.j\_relation.Cat">

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

</bean>

可以修改成：

<bean id="cat" class="com.offcn.j\_relation.Cat" abstract="true"></bean>

<bean id="cat1" parent="cat">

<property name="name" value="黑猫警长"></property>

</bean>

<bean id="cat2" parent="cat">

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

</bean>-->

</beans>

* 测试类App.java

**public** **class** App {

**public** **static** **void** main(String[] args) {

ApplicationContext ac = **new** ClassPathXmlApplicationContext("com/offcn/j\_relation/beans.xml");

/\*\*测试继承\*/

// Cat cat1 = (Cat) ac.getBean("cat1");

// System.out.println(cat1.getName());

// Cat cat2 = (Cat) ac.getBean("cat2");

// System.out.println(cat2.getName());

/\*\*测试依赖\*/

// TaskManager taskManager = (TaskManager)ac.getBean("taskManager");

/\*\*测试引用\*/

// Student student = (Student)ac.getBean("student");

// student.readBook();

/\*\*测试一个beans.xml可以通过<import resource="classpath:com/offcn/i\_constructor/beans.xml"/>注入多个beans.xml\*/

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

userService.saveUser();

}

}

### （4）注解配置：使用@autowired

**在java代码中使用@Autowired或@Resource注解方式进行装配的前提条件是。**

#### 【1】引入context命名空间 需要在xml配置文件中配置以下信息：

<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-4.3.xsd

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

**http://www.springframework.org/schema/context/spring-context-4.3.xsd**">

**<context:annotation-config/>**

</beans>

#### 【2】在配置文件中添加context:annotation-config标签

**<context:annotation-config/>**

这个配置隐式注册了多个对注解进行解析处理的处理器

AutowiredAnnotationBeanPostProcessor，CommonAnnotationBeanPostProcessor，

PersistenceAnnotationBeanPostProcessor，RequiredAnnotationBeanPostProcessor

**但是这里：**

**如果使用jdk1.5，需要引入com.springsource.javax.annotation-1.0.0.jar包**

**如果使用jdk1.6及以上，不需要引入com.springsource.javax.annotation-1.0.0.jar包**

#### 【3】使用注解@autowired标注在属性上

**Autowired标注在字段上**

\* @Autowired注解默认按类型进行装配

\* 获取该注解标注的字段的类型---PersonDao类型

\* 以该类型为条件到spring容器(beans.xml)中去查找bean的id节点的类型是PersonDao类型.

\* 找到以后,获取该节点对应的对象,利用反射直接为personDao变量赋值

#### 【4】使用注解@autowired标注在set方法上

**Autowired标注在setter方法上**

@Autowired注解默认按类型进行装配

\* 获取 setPersonDao()方法的参数的类型---PersonDao类型

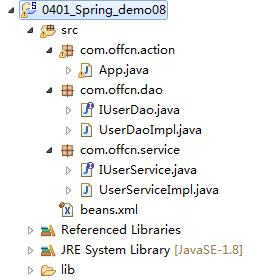
\* 以该类型为条件到spring容器(beans.xml)中去查找bean的id节点的类型是PersonDao类型.

\* 找到以后,获取该节点对应的对象, 把该对象作为实参传递给该setPersonDao(

PersonDao personDao)的形参.

#### 【5】测试代码

结构：



* IUserDao接口

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

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

}

* UserDaoImpl类

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

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

System.*out*.println("执行UserDaoImpl类的save()方法！");

}

}

* IUserService接口

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

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

}

* UserServiceImpl类

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

/\*\*

\* Autowired标注在字段上

\* **@Autowired注解默认按类型进行装配**

\* 获取该注解标注的字段的类型---IUserDao类型

\* 以该类型为条件到spring容器(beans.xml)中去查找bean的id节点的类型是IUserDao类型.

\* 找到以后,获取该节点对应的对象,利用反射直接为userDao变量赋值

\*/

//@Autowired @Qualifier("userDao")

**private** IUserDao userDao;

/\*\*

\* Autowired标注在setter方法上

**@Autowired注解默认按类型进行装配**

\* 获取 setUserDao()方法的参数的类型---UserDao类型

\* 以该类型为条件到spring容器(beans.xml)中去查找bean的id节点的类型是UserDao类型.

\* 找到以后,获取该节点对应的对象, 把该对象作为实参传递给该setUserDao(IUserDao userDao)的形参.

\*

\*/

@Autowired

**public** **void** setUserDao(IUserDao userDao) {

**this**.userDao = userDao;

}

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

System.*out*.println("执行UserServiceImpl类的saveUser()的方法！");

userDao.save();

}

}

* Spring容器的配置（beans.xml）

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

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

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

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

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

*http://www.springframework.org/schema/beans/spring-beans-4.3.xsd*

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

*http://www.springframework.org/schema/context/spring-context-4.3.xsd"*>

<!--

. 在java代码中使用@Autowired或@Resource注解方式进行装配的前提条件是。

1、引入context命名空间 需要在xml配置文件中配置以下信息：

2、在配置文件中添加context:annotation-config标签

\* 这个配置隐式注册了多个对注解进行解析处理的处理器

-->

<context:annotation-config/>

<!-- 创建DAO对象 -->

<bean id=*"userDao"* class=*"com.offcn.dao.UserDaoImpl"*></bean>

<!-- 创建Service对象 -->

<bean id=*"userService"* class=*"com.offcn.service.UserServiceImpl"*>

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

</bean>

</beans>

* 测试类App.java

**public** **class** App {

**public** **static** **void** main(String[] args) {

//调用Service类

//加载spring容器：

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

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

userService.saveUser();

}

}

### （5）注解配置：使用@Resource

#### 【1】使用注解@Resource标注在属性上

**Resource注解标注在字段上**

\* @Resource注解默认按名称装配。

**\* 如果没有指定name属性**

\* 获取该注解标注的字段值---personDao

\* 以该字段值为条件到spring容器(beans.xml)中去查找bean的id节点的值是 personDao的节点

\* 找到以后,获取该节点对应的对象, 利用反射直接为personDao变量赋值

\* 如果没有找到.并且按照默认的名称找不到依赖对象时, @Resource注解会回退到按类型装配

\* 获取该注解标注的字段类型--PersonDao

\* 以该类型为条件到spring容器(beans.xml)中去查找bean的节点的类型是PersonDao类型的对象

\* 找到以后,获取该节点对应的对象,利用反射直接为personDao变量赋值

**\* 如果指定name属性，只能按名称装配**

\* 获取name属性的值 personDao

\* 以该值为条件到spring容器(beans.xml)中去查找bean的id节点的值是PersonDao的对象

\* 找到以后,获取该节点对应的对象, 利用反射直接为personDao变量赋值

\* 如果不存在该名称,抛出异常

例如：



#### 【2】使用注解@Resource标注在set方法上

**resource注解标注在setter方法上**

\* @Resource注解默认按名称装配。

**\* 如果没有指定name属性**

\* 获取setPersonDao()方法的属性名---personDao

\* 以该属性名为条件到spring容器(beans.xml)中去查找bean的id节点的值是personDao的节点

\* 找到以后,获取该节点对应的对象, 把该对象作为实参传递给该setPersonDao(

PersonDao personDao)的形参.

\* 如果没有找到.并且按照默认的名称找不到依赖对象时，@Resource注解会回退到按类型装配

\* 获取setPersonDao()方法的参数类型---PersonDao

\* 以该类型为条件到spring容器(beans.xml)中去查找bean的节点的类型是PersonDao类型对象

\* 找到以后,获取该节点对应的对象, 把该对象作为实参传递给该setPersonDao(

PersonDao personDao)方法的形参

**\* 如果指定name属性 只能按名称装配**

\* 获取name属性的值

\* 以该值为条件到spring容器(beans.xml)中去查找bean的id节点的值是PersonDao的对象

\* 找到以后,获取该节点对应的对象, 把该对象作为实参传递给该setPersonDao(

PersonDao personDao)的形参.

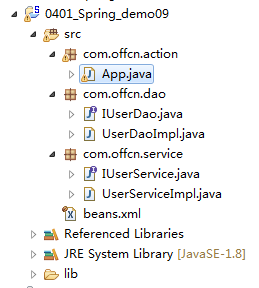
\* 如果不存在该名称,抛出异常

例如：



#### 【4】测试代码

结构：



* IUserDao接口

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

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

}

* UserDaoImpl类

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

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

System.*out*.println("执行UserDaoImpl类的save()方法！");

}

}

* IUserService接口

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

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

}

* UserServiceImpl类

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

@Value("100")

**private** Integer id;

@Value("张三丰")

**private** String name;

/\*\*

\* Resource注解标注在字段上

\* **@Resource注解默认按名称装配**。

\* 如果没有指定name属性

\* 获取该注解标注的字段值---userDao

\* 以该字段值为条件到spring容器(beans.xml)中去查找bean的id节点的值是 userDao的节点

\* 找到以后,获取该节点对应的对象, 利用反射直接为userDao变量赋值

\* 如果没有找到.并且按照默认的名称找不到依赖对象时, **@Resource注解会回退到按类型装配**

\* 获取该注解标注的字段类型--UserDao

\* 以该类型为条件到spring容器(beans.xml)中去查找bean的节点的类型是UserDao类型

的对象

\* 找到以后,获取该节点对应的对象, 利用反射直接为userDao变量赋值

\* 如果指定name属性 只能按名称装配

\* 获取name属性的值 userDao

\* 以该值为条件到spring容器(beans.xml)中去查找bean的id节点的值是userDao的对象

\* 找到以后,获取该节点对应的对象, 利用反射直接为userDao变量赋值

\* 如果不存在该名称,抛出异常

\*/

//@Resource(name="userDao")

**private** IUserDao userDao;

/\*\*

\* resource注解标注在setter方法上

\* **@Resource注解默认按名称装配**。

\* 如果没有指定name属性

\* 获取setUserDao()方法的属性名---userDao

\* 以该属性名为条件到spring容器(beans.xml)中去查找bean的id节点的值是userDao的节点

\* 找到以后,获取该节点对应的对象, 把该对象作为实参传递给该setUserDao(IUserDao userDao)的形参.

\* 如果没有找到.并且按照默认的名称找不到依赖对象时，@Resource注解会回退到按类型装配

\* 获取setUserDao()方法的参数类型---UserDao

\* 以该类型为条件到spring容器(beans.xml)中去查找bean的节点的类型是UserDao类型对象

\* 找到以后,获取该节点对应的对象, 把该对象作为实参传递给该setUserDao(IUserDao userDao)方法的形参

\* 如果指定name属性 只能按名称装配

\* 获取name属性的值

\* 以该值为条件到spring容器(beans.xml)中去查找bean的id节点的值是userDao的对象

\* 找到以后,获取该节点对应的对象, 把该对象作为实参传递给该setUserDao(IUserDao userDao)的形参.

\* 如果不存在该名称,抛出异常

\*

\*/

@Resource(name="userDao")

**public** **void** setUserDao(IUserDao userDao) {

**this**.userDao = userDao;

}

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

System.*out*.println("执行UserServiceImpl类的saveUser()的方法！");

System.*out*.println("id："+id);

System.*out*.println("name:"+name);

userDao.save();

}

@PostConstruct

**public** **void** initMethod(){

System.*out*.println("这是初始化的方法！");

}

@PreDestroy

**public** **void** destroyMethod(){

System.*out*.println("这是销毁的方法！");

}

}

* Spring容器的配置（beans.xml）

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

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

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

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

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

*http://www.springframework.org/schema/beans/spring-beans-4.3.xsd*

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

*http://www.springframework.org/schema/context/spring-context-4.3.xsd"*>

<!--

. 在java代码中使用@Autowired或@Resource注解方式进行装配的前提条件是。

1、引入context命名空间 需要在xml配置文件中配置以下信息：

2、在配置文件中添加context:annotation-config标签

\* 这个配置隐式注册了多个对注解进行解析处理的处理器

-->

<context:annotation-config/>

<!-- 创建DAO对象 -->

<bean id=*"userDao"* class=*"com.offcn.dao.UserDaoImpl"*></bean>

<!-- 创建Service对象 -->

<bean id=*"userService"* class=*"com.offcn.service.UserServiceImpl"*>

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

</bean>

</beans>

* 测试类App.java

**public** **class** App {

**public** **static** **void** main(String[] args) {

//调用Service类

//加载spring容器：

ConfigurableApplicationContext ac = new ClassPathXmlApplicationContext(new String[] {"beans.xml"}); IUserService userService = (IUserService)ac.getBean("userService");

userService.saveUser();

ac.close();

}

}

### （6）注解配置：使用classpath自动扫描把组件纳入spring容器中管理

作用：在开发时对项目的每一层进行划分

前面的例子我们都是使用XML的bean定义来配置组件。在一个稍大的项目中，通常会有上百个组件，如果这些组件采用xml的bean定义来配置，显然会增加配置文件的体积，查找及维护起来也不太方便。

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

它的作用和在xml文件中使用bean节点配置组件是一样的。

**要使用自动扫描机制，我们需要打开以下配置信息:**

#### 【1】引入context命名空间 需要在xml配置文件中配置以下信息：

<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-4.3.xsd

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

**http://www.springframework.org/schema/context/spring-context-4.3.xsd">**

**<context:component-scan base-package="com.offcn"/>**

</beans>

#### 【2】在配置文件中添加context:component-scan标签

**<context:component-scan base-package="com.offcn"/>**

其中base-package为需要扫描的包(含子包)。

**注**:

**1、在使用组件扫描元素时，**

AutowiredAnnotationBeanPostProcessor 和CommonAnnotationBeanPostProcessor会隐式地被包括进来。也就是说，连个组件都会被自动检测*并*织入，所有这一切都不需要在XML中提供任何bean配置元数据。

**2、功能介绍**

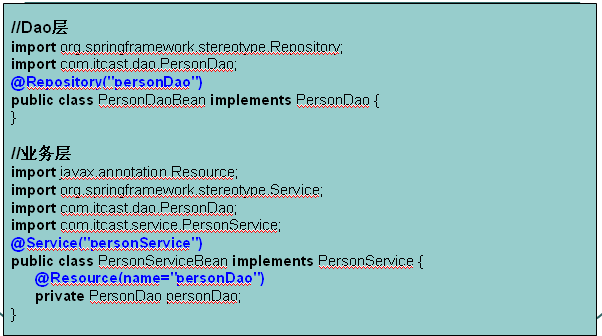
@Service用于标注业务层组件、

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

@Repository用于标注数据访问组件，即DAO组件。

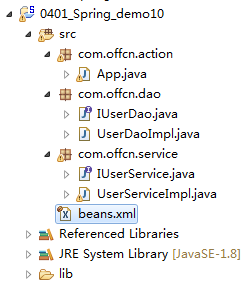
而@Component泛指组件，当组件不好归类的时候，我们可以使用这个注解进行标注。

例如：



#### 【3】测试代码

结构：



* IUserDao接口

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

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

}

* UserDaoImpl类

/\*\*

\* **@Repository标注在类的上面**

\* 相当于spring容器中定义：

\* <bean id="userDaoImpl" class="com.offcn.dao.UserDaoImpl"></bean>

\* 此时如果没有为@Repository指定value属性的名称，此时就是按照类的名称作为spring容器中bean节点的id，并且首字母要小些

\*

\* **@Repository**(value="userDao")

\* 相当于spring容器中定义：

\* <bean id="userDao" class="com.offcn.dao.UserDaoImpl"></bean>

\*/

@Repository(value="userDao")

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

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

System.*out*.println("执行UserDaoImpl类的save()方法！");

}

}

* IUserService接口

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

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

}

* UserServiceImpl类

/\*\*

\* **@Service标注在类的上面**

\* 相当于spring容器中定义：

\* <bean id="userServiceImpl" class="com.offcn.service.UserServiceImpl"></bean>

\* 此时如果没有为@@Service指定value属性的名称，此时就是按照类的名称作为spring容器中bean节点的id，并且首字母要小些

\*

\* **@Service**(userService)

\* 相当于spring容器中定义：

\* <bean id="userService" class="com.offcn.service.UserServiceImpl"></bean>

\*

\* 再添加@Resource(name="userDao")

\* 此时表示

\* <bean id="userService" class="com.offcn.service.UserServiceImpl">

\* <property name="userDao" ref="userDao">

\* </bean>

\*/

@Service("userService")

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

@Resource(name="userDao")

**private** IUserDao userDao;

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

System.*out*.println("执行UserServiceImpl类的saveUser()的方法！");

userDao.save();

}

}

* Spring容器的配置（beans.xml）

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

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

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

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

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

*http://www.springframework.org/schema/beans/spring-beans-4.3.xsd*

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

*http://www.springframework.org/schema/context/spring-context-4.3.xsd"*>

<!—必须加入扫描指定包下面的注解，自动注解才能生效 -->

*<context:component-scan base-package="com.offcn"/>*

<!--

. 在java代码中使用@Autowired或@Resource注解方式进行装配的前提条件是。

1、引入context命名空间 需要在xml配置文件中配置以下信息：

2、在配置文件中添加context:annotation-config标签

\* 这个配置隐式注册了多个对注解进行解析处理的处理器

-->

<context:annotation-config/>

<!-- 创建DAO

<bean id="userDao" class="com.offcn.dao.UserDaoImpl"></bean> -->

<!-- 创建Service

<bean id="userService" class="com.offcn.service.UserServiceImpl">

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

</bean>

-->

</beans>

* 测试类App.java

**public** **class** App {

**public** **static** **void** main(String[] args) {

//调用Service类

//加载spring容器：

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

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

userService.saveUser();

}

}

\* 功能介绍

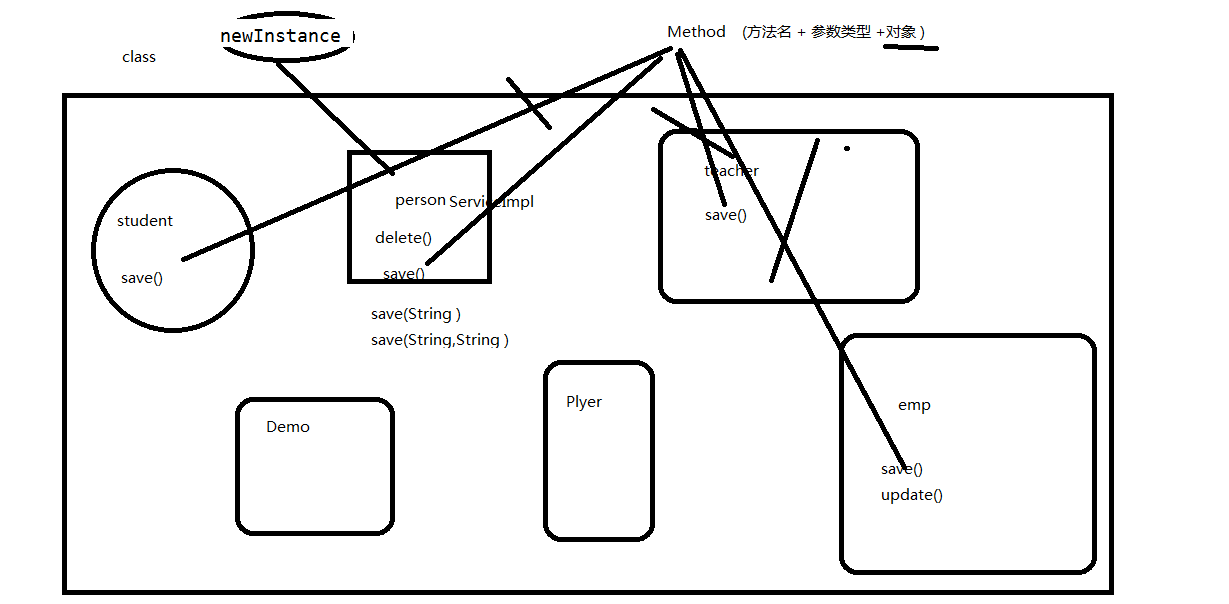
**@Service用于标注业务层组件、**

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

**@Repository用于标注数据访问组件，即DAO组件。**

而@Component泛指组件，当组件不好归类的时候，我们可以使用这个注解进行标注。

附加：反射



@Test

**public** **void** test002() **throws** Exception{

// class="com.offcn.service.impl.PersonServiceimpl"

// 属性名字 pid

// 数据 1001

Class<PersonServiceimpl> c = PersonServiceimpl.**class**;

PersonServiceimpl newInstance = c.newInstance();

String methodNmae= "pid";

// setPid()

String targetMethod = "set"+methodNmae.substring(0,1).toUpperCase()+methodNmae.substring(1);

String targetMethodGet = "get"+methodNmae.substring(0,1).toUpperCase()+methodNmae.substring(1);

Method method = c.getMethod(targetMethod, **int**.**class**);

method.invoke(newInstance, 1001);

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

Method method2 = c.getMethod(targetMethodGet);

Object invoke = method2.invoke(newInstance);

System.*out*.println(invoke);

}