框架概述：

Struts2: 基于mvc模式的应用层框架！

主要是作为控制层组件，javabean， jsp！

Hibernate: 持久层组件，简化jdbc操作！

Spring: 主要有六大模块功能

事务管理！

与其他框架整合！

Mvc开发模式：

基于mvc模式的项目分层!

# 1. Spring框架

1. 框架理解

框架，都会预先实现一些功能，给开发人员开发提供便利！

提高开发效率、提升程序可扩展性、健壮！

1. 专业术语

(1)高内聚、低耦合：

类内部的关系越紧密越好，类与类的关系越少越好！

(2)非侵入式设计：

1)侵入式设计：

引入的组件对现有的类的结构会有影响，这种就是“侵入式设计”的组件！

Struts2: 侵入式设计！

2)非侵入式设计：

引入组件对现有的类的结构没有影响！

Hibernate: 非侵入式设计！

Spring, 非侵入式设计

(3)IOC 容器:Inversion of control 控制反转

1)控制反转容器! -🡪 对象创建的问题！

2)解释：

User user = new User(); 自己控制对象的创建

现在需要对象，自己不创建，交给外部的容器创建，叫控制反转！

IOC容器= bean.xml配置 + ApplicationContext容器类

3) 例子

public class Girl {

private Boy boy =new Boy();

public void kiss() {

System.out.println(boy.getName());

}

}

boy是在应用内部创建及维护的。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。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

(4)DI, dependency injection 依赖注入

1）创建对象后，处理对象的依赖关系！

User user = new User();

user.setAddress(..) ;// 需要DI（依赖注入）

2）当我们把依赖对象交给外部容器负责创建，那么Girl类可以改成如下：

public class Girl {

private Boy boy;

public void kiss() {

System.out.println(boy.getName());

}

}

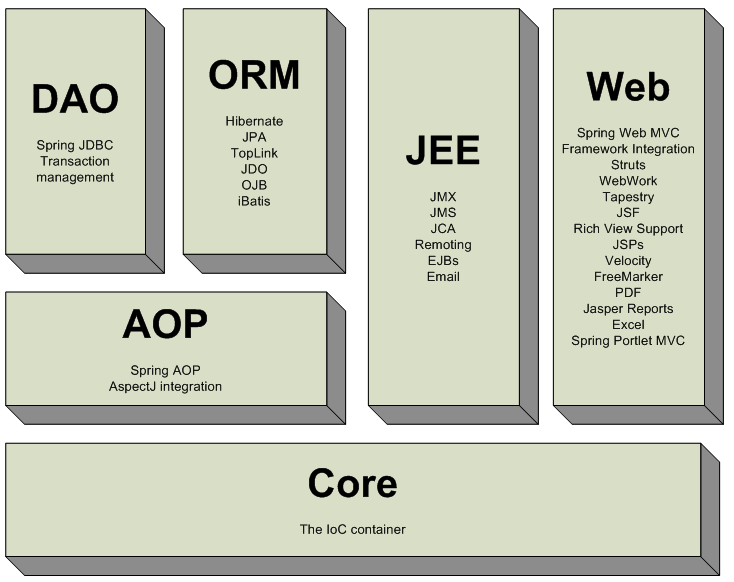
依赖注入：在运行期，由外部容器动态地将依赖对象注入到另一个对象中。

(5)Aop…

1. Spring框架

Spring 开源框架，提供的是一站式解决方案！

(1)六大模块:



(2) spring好处:

* 降低组件之间的耦合度,实现软件各层之间的解耦。

1)IOC(依赖注入)：包含并管理应用对象的配置和生命周期，可配置每个bean如何被创建，也可配置每个bean是只有一个实例，还是每次需要时都生成一个新的实例，以及它们是如何相互关联的

2)AOP(面向切面)：采用了面向切面编程来实现很多基础但是与业务逻辑无关的功能的解耦,比如：事务管理、日志、权限验证.....

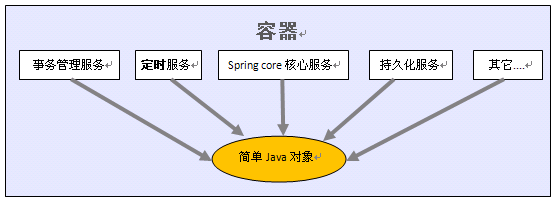
3)DAO:Spring 还提供了对数据库JDBC的封装,使用JdbcTemplate来简化数据操作

4)ORM：提供了对主流的对象关系映射框架的支持

5)JEE: 对Java企业级开发提供了一些解决方案,例如EJB、JMS等

6)WEB: 提供了自己的Spring MVC和对显示层框架的集合支持

* 可使用容器提供的众多服务



1. Spring 第一个案例， 解决项目中如何创建对象

(1) 引入jar文件 (3.2版本)

commons-logging-1.1.3.jar 【单独下载】

spring-beans-3.2.5.RELEASE.jar 【spring源码, bean节点管理】

spring-context-3.2.5.RELEASE.jar 【spring上下文类】

spring-core-3.2.5.RELEASE.jar 【IOC容器】

spring-expression-3.2.5.RELEASE.jar 【spring表达式】

注意：使用的版本Spring3.2

在这个版本中，只有spring自身的核心功能，spring依赖的其他组件，需要单独下载！ 例如：日志jar文件，就需要单独下载！

(2) 新建applicationContext.xml , 源码中拷贝约束

|  |
| --- |
| <?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.xsd  http://www.springframework.org/schema/context  http://www.springframework.org/schema/context/spring-context.xsd">  </beans> |

(3) 配置

|  |
| --- |
| <!-- 创建Dao实例 -->  <bean id="userDao" class="cn.itcast.dao.UserDao"></bean>  <!-- 创建Service实例 -->  <bean id="userService" class="cn.itcast.service.UserService">  <property name="userDao" ref="userDao"></property>  </bean>    <!-- 创建Action实例 -->  <bean id="userAction" class="cn.itcast.action.UserAction">  <property name="userService" ref="userService"></property>  </bean> |

(4) UserDao.java

|  |
| --- |
| package cn.itcast.dao;  public class UserDao {  public void save() {  System.out.println("UserDao.save()");  }  } |

(5) UserService.java

|  |
| --- |
| package cn.itcast.service;  import cn.itcast.dao.UserDao;  public class UserService {  // 创建dao对象 : 单例，启动创建  private UserDao userDao;  public void setUserDao(UserDao userDao) {  this.userDao = userDao;  }  public void save() {  userDao.save();  }  } |

(6) UserAction.java

|  |
| --- |
| package cn.itcast.action;  import cn.itcast.service.UserService;  /\*\*  \* 多例Action实例：在访问的时候创建对象！  \* @heyutong  \* 2017年6月2日  \*/  public class UserAction {  // 判断一个类是单例还是多例，主要看有没有维护成员变量、且对象成员变量进行修  改！ 如果有，这个类就应该是多例！  // 创建Service:单例，启动时候创建  private UserService userService;//= new UserService();  // 提供set方法，给外部容器注入  public void setUserService(UserService userService) {  this.userService = userService;  }    public String execute() {  userService.save();  return "success";  }  } |

(4) 测试

|  |
| --- |
| package junit.test;  import org.junit.Test;  import org.springframework.context.support.ClassPathXmlApplicationContext;  import cn.itcast.action.UserAction;  public class ActionTest {  //不从IOC获取对象  //报空指针异常  /\*@Test  public void test1() {  UserAction userAction = new UserAction();  userAction.execute();  }\*/    //从IOC容器获取对象  @Test  public void test2() {  // 容器对象(加载applicationContext.xml配置文件)  //调用了setUserDao和setUserService方法  ClassPathXmlApplicationContext ac = new  ClassPathXmlApplicationContext("applicationContext.xml");  // 获取对象  UserAction userAction = (UserAction)ac.getBean("userAction");  userAction.execute();  }  } |

# 2. Spring IOC 容器

## 2.1创建bean细节：

1. bean在容器中的写法

2. bean声明周期

|  |
| --- |
| <?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.xsd*  *http://www.springframework.org/schema/context*  *http://www.springframework.org/schema/context/spring-context.xsd"*>  <!-- 把对象加入IOC容器 -->  <!--  细节1：  id 与name：  id 不能以数字开头，不能含有特殊符号, 不能有空格、逗号等； id 不能重复！  name 可以数字开头，可以有特殊符号，如果name值重复，编译没问题但运行报错！  <bean id="user" name="user2,user3 user4" class="cn.itcast.a\_config.User"></bean>  <bean name="user5" class="cn.itcast.a\_config.User"></bean>  <bean id="user6" class="cn.itcast.a\_config.User"></bean>  <bean id="user6" class="cn.itcast.a\_config.User"></bean>  -->    <!-- 细节2： (单例/多例) -->  <!--  scope="singleton" 默认表示单例！  prototype 多例  init-method="" 在创建完对象之后执行初始化方法  destroy-method="" 在调用容器类的destroy()方法时候，对单例的对象有效！  lazy-init="true" 延迟初始化/设置只对单例有影响，对多例没有任何影响！  单例对象默认是在创建容器时创建所有单例对象，如希望在第一次访问时创建单例  对象，就设置延迟初始化  Bean生命周期：  bean在xml中配置，  1.singleton 单例  1) 创建对象  如果有配置延迟初始化，  lazy-init=true 如单例对象有配置延迟初始化， 在创建容器之后，在第一次从容器  获取对象时创建单例的对象！  如没有配置或延迟初始化为默认值, 单例对象会在创建容器时创建对象  2) 执行初始化方法 , init-method配置的方法会执行  3) 调用容器destroy() 方法时，容器在销毁单例对象实例时，会调用destroy-method  对应的方法 此时bean对象会被销毁！    2.prototype 多例  1) 每次从容器获取对象时，都会创建新的对象  2) 每次创建完对象后，就执行初始化方法  3) java回回收不用资源(jvm gc)  -->  <bean id=*"user"*  class=*"cn.itcast.a\_config.User"*  init-method=*"init"*  destroy-method=*"destroy\_"*  lazy-init=*"false"*  ~~scope=~~*~~"prototype"~~*>  </bean>  </beans> |

3. User.java

|  |
| --- |
| package cn.itcast.a\_config;  public class User {  private int id;  private String name;    public User() {  System.out.println("User.User(), 创建User对象...");  }  public int getId() {  return id;  }  public void setId(int id) {  this.id = id;  }  public String getName() {  return name;  }  public void setName(String name) {  this.name = name;  }    //初始化时候调用  public void init() {  System.out.println("初始化...");  }    //销毁对象实例时候调用  public void destroy\_() {  System.out.println("销毁...");  }  } |

4. App.java

|  |
| --- |
| package cn.itcast.a\_config;  import org.junit.Test;  import org.springframework.context.support.ClassPathXmlApplicationContext;  public class App {  @Test  public void test() {  ClassPathXmlApplicationContext ac = new  ClassPathXmlApplicationContext("cn/itcast/a\_config/bean.xml");  System.out.println("----------");    //获取/创建对象  User user = (User) ac.getBean("user");  System.out.println(user);  User user2 = (User) ac.getBean("user");  System.out.println(user2);  // 销毁容器实例  ac.destroy();  }  } |

1. 运行测试结果:
2. 单例：未配置scope=*"prototype"*
3. new ClassPathXmlApplicationContext("cn/itcast/a\_config/bean.xml");时，打印:

User.User(), 创建User对象... //new User对象

初始化... //执行init-method="init"

2)user 和user2 对象一样，例如都是cn.itcast.a\_config.User@1147d1a

3) 销毁容器实例,即执行ac.destroy();时执行destroy-method="destroy\_",打印:

销毁...

1. 多例: 配置scope=*"prototype"*
2. User user = (User) ac.getBean("user");时,打印:

|  |
| --- |
| User.User(), 创建User对象... //创建一个User对象  初始化... //执行init-method="init"  cn.itcast.a\_config.User@111cccd |

1. User user2 = (User) ac.getBean("user");时,打印:

|  |
| --- |
| User.User(), 创建User对象... //创建另一个User对象  初始化... //执行init-method="init"  cn.itcast.a\_config.User@18441c1 |

3) 销毁容器实例,即执行ac.destroy();时不执行destroy-method="destroy\_",

## 2.2创建对象的几种方式

**2.2.1.调用无参数构造器**

**2.2.2.调用有参数构造器**

1.bean.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.xsd  http://www.springframework.org/schema/context  http://www.springframework.org/schema/context/spring-context.xsd">  <!--IOC容器， 创建对象 -->  <!—1) 调用无参数构造器 -->  <bean id="user1" class="cn.itcast.b\_create\_obj.User"></bean>    <!—2) 调用有参数构造器, 创建对象 -->  <!-- String str = new String("Jack"); -->  <bean id="str" class="java.lang.String">  <constructor-arg value="Jack"></constructor-arg>  </bean>    <!-- 创建user对象，且赋值 -->  <!--  value 当直接给属性值的时候使用value赋值  ref 当引用的是IOC容器中的对象的时候，使用ref  -->  <bean id="user" class="cn.itcast.b\_create\_obj.User">  <constructor-arg index="0" type="int" value="10000"></constructor-arg>  <constructor-arg index="1" type="String" ref="str"></constructor-arg>  </bean>  </beans> |

**2.** User.java

|  |
| --- |
| package cn.itcast.b\_create\_obj;  public class User {  private int id;  private String name;    public User() {  System.out.println("User.User(), 创建User对象...");  }  public User(int id, String name) {  super();  this.id = id;  this.name = name;  }  public int getId() {  return id;  }  public void setId(int id) {  this.id = id;  }  public String getName() {  return name;  }  public void setName(String name) {  this.name = name;  }  @Override  public String toString() {  return "User [id=" + id + ", name=" + name + "]";  }    // 初始化时候调用  public void init() {  System.out.println("初始化...");  }    // 销毁对象实例时候调用  public void destroy\_() {  System.out.println("销毁...");  }  } |

**3.** App.java

|  |
| --- |
| package cn.itcast.b\_create\_obj;  import org.junit.Test;  import org.springframework.context.support.ClassPathXmlApplicationContext;  public class App {  @Test  public void test() {  // 方式1：创建容器对象  /\*ClassPathXmlApplicationContext ac = new ClassPathXmlApplicationContext("cn/itcast/b\_create\_obj/bean.xml");//打印:User.User(), 创建User对象...  User user = (User) ac.getBean("user1");  System.out.println(user);\*/    // 方式2： 会从当前类所在的包下找bean.xml 【测试方便】  /\*ClassPathXmlApplicationContext ac = new  ClassPathXmlApplicationContext("bean.xml", this.getClass());  //打印:User.User(), 创建User对象...  String str = (String)ac.getBean("str");  System.out.println(str);//打印:Jack\*/    // 方式2： 会从当前类所在的包下找bean.xml 【测试方便】  ClassPathXmlApplicationContext ac = new  ClassPathXmlApplicationContext("bean.xml", this.getClass());  //打印:User.User(), 创建User对象...  User user = (User) ac.getBean("user");  System.out.println(user);//打印:User [id=10000, name=Jack]  }  } |

**2.2.3. 工厂**

1. 静态方法:<bean id=”’ class=”” factory-method=””/>

(1)nean2.xml

|  |
| --- |
| <!-- \* 工厂静态方法,创建对象 -->  <!--  class 指定工厂的类型；  factory-method: 工厂类的静态方法  -->  <bean id="user1" class="cn.itcast.b\_create\_obj.UserFactory"  factory-method="getStaticInstace"></bean> |

(2) App.java

|  |
| --- |
| package cn.itcast.b\_create\_obj;  import org.junit.Test;  import org.springframework.context.support.ClassPathXmlApplicationContext;  public class App {  @Test  public void test() {  //方式2： 会从当前类所在的包下找bean2.xml 【测试方便】  //打印:(1)调用getStaticInstace方法 (2)调用User(int id, String name)构造方法  ClassPathXmlApplicationContext ac = new ClassPathXmlApplicationContext("bean2.xml", this.getClass());  User user = (User) ac.getBean("user1");  System.out.println(user);//User [id=101, name=工厂静态方法，创建对象]  }  } |

(3) UserFactory.java

|  |
| --- |
| package cn.itcast.b\_create\_obj;  // 创建user对象的工厂  public class UserFactory {  // 非静态  public User getInstace() {  return new User(100,"工厂实例方法，创建对象");  }    // 静态  public static User getStaticInstace() {  System.out.println("调用getStaticInstace方法");  return new User(101,"工厂静态方法，创建对象");  }  } |

2. 非静态方法:

<bean id=”factory” class=”..”>

<bean id=”” factory-bean=” factory” factory-method=”实例方法” />

(1)nean2.xml

|  |
| --- |
| <!-- \* 非静态方法创建对象 -->  <!-- 先创建工厂实例 -->  <bean id="factory" class="cn.itcast.b\_create\_obj.UserFactory"></bean>  <bean id="user" factory-bean="factory" factory-method="getInstace"></bean> |

(2)

|  |
| --- |
| package cn.itcast.b\_create\_obj;  import org.junit.Test;  import org.springframework.context.support.ClassPathXmlApplicationContext;  public class App {  @Test  public void test() {  //方式2： 会从当前类所在的包下找bean2.xml 【测试方便】  //打印:(1)调用getInstace方法 (2)调用User(int id, String name)构造方法  ClassPathXmlApplicationContext ac = new  ClassPathXmlApplicationContext("bean2.xml", this.getClass());  User user = (User) ac.getBean("user");  System.out.println(user);//打印:User [id=100, name=工厂实例方法，创建对象]  }  } |

**2.24. 反射**

## 2.3 处理对象的依赖关系

给对象属性赋值(DI, 依赖注入)方式：

**2.3.1构造函数赋值**

**2.3.2 set方法注入值**

\* 普通字段赋值

\* 集合属性 (list/map/property)

1. 例1

(1)bean.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.xsd  http://www.springframework.org/schema/context  http://www.springframework.org/schema/context/spring-context.xsd">  <!--IOC容器， 给对象属性赋值 -->  <bean id="user" class="cn.itcast.c\_di.User">  <property name="id" value="1000"></property>  <property name="name" value="Jacky"></property>  <!-- list集合赋值 -->  <property name="list">  <list>  <value>cn</value>  <value>usa</value>  </list>  </property>  <!-- map 集合赋值 -->  <property name="map">  <map>  <entry key="cn" value="China"></entry>  <entry key="usa" value="1234"></entry>  </map>  </property>  <!-- Properties对象赋值 -->  <property name="prop">  <props>  <prop key="cn">China</prop>  <prop key="usa">America</prop>  </props>  </property>  </bean>  </beans> |

(2) User.java

|  |
| --- |
| package cn.itcast.c\_di;  import java.util.List;  import java.util.Map;  import java.util.Properties;  public class User {  private int id;  private String name;  // list集合  private List<String> list;  // Map集合  private Map<String,Object> map;  // Properties 对象  private Properties prop;  public int getId() {  return id;  }  public void setId(int id) {  this.id = id;  }  public String getName() {  return name;  }  public void setName(String name) {  this.name = name;  }  public List<String> getList() {  return list;  }  public void setList(List<String> list) {  this.list = list;  }  public Map<String, Object> getMap() {  return map;  }  public void setMap(Map<String, Object> map) {  this.map = map;  }  public Properties getProp() {  return prop;  }  public void setProp(Properties prop) {  this.prop = prop;  }  @Override  public String toString() {  return "User [id=" + id + ", list=" + list + ", map=" + map + ", name="  + name + ", prop=" + prop + "]";  }  } |

(3) App.java

|  |
| --- |
| package cn.itcast.c\_di;  import org.junit.Test;  import org.springframework.context.ApplicationContext;  import org.springframework.context.support.ClassPathXmlApplicationContext;  public class App {  @Test  public void testApp() throws Exception {  // 方式2： 会从当前类所在的包下找bean.xml 【测试方便】  //调用了setId、setName、setList、setMap、setProp方法  ApplicationContext ac = new  ClassPathXmlApplicationContext("bean.xml",this.getClass());  User user = (User) ac.getBean("user");  //打印:User [id=1000, list=[cn, usa],  //map={cn=China, usa=1234}, name=Jacky, prop={usa=America, cn=China}]  System.out.println(user);  }  } |

2.例2

(1)bean.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:p="http://www.springframework.org/schema/p"  xmlns:context="http://www.springframework.org/schema/context"  xsi:schemaLocation="  http://www.springframework.org/schema/beans  http://www.springframework.org/schema/beans/spring-beans.xsd  http://www.springframework.org/schema/context  http://www.springframework.org/schema/context/spring-context.xsd">  <!-- 配置： 需要创建的对象、对象依赖关系处理 -->  <!-- 方式1：通过set方法注入，这种方式最常用！-->  <bean id="userdao" class="cn.itcast.d\_di2.UserDao"></bean>  <bean id="userService" class="cn.itcast.d\_di2.UserService">  <property name="userDao" ref="userdao"></property>  </bean>  <!-- 多例 -->  <bean id="userAction" class="cn.itcast.d\_di2.UserAction" scope="prototype">  <property name="userService" ref="userService"></property>  </bean>  </beans> |

(2) UserDao.java

|  |
| --- |
| package cn.itcast.d\_di2;  public class UserDao implements IUserDao {  @Override  public void save() {  System.out.println("UserDao.save()");  }  } |

IUserDao.java

|  |
| --- |
| package cn.itcast.d\_di2;  public interface IUserDao {  // 模拟：保存对象  //接口中的方法都是public abstract,平时习惯abstract不写,public也不写  public abstract void save();  } |

(3) IUserService.java

|  |
| --- |
| package cn.itcast.d\_di2;  public interface IUserService {  public abstract void save();  } |

UserService.java

|  |
| --- |
| package cn.itcast.d\_di2;  public class UserService implements IUserService{  private UserDao userDao;  // 接收IOC容器注入值  public void setUserDao(UserDao userDao) {  this.userDao = userDao;  }  @Override  public void save() {  userDao.save();  }  } |

(4) UserAction.java

|  |
| --- |
| package cn.itcast.d\_di2;  public class UserAction {  private IUserService userService;  //接收IOC容器注入值  public void setUserService(IUserService userService) {  this.userService = userService;  }  public String execute(){  userService.save();  return "success";  }  } |

(5)

|  |
| --- |
| package cn.itcast.d\_di2;  import org.junit.Test;  import org.springframework.context.ApplicationContext;  import org.springframework.context.support.ClassPathXmlApplicationContext;  public class App {  private ApplicationContext ac = new  ClassPathXmlApplicationContext("bean.xml",App.class);  @Test  public void testApp() throws Exception {  //从容器中获取Action实例  UserAction userAction = (UserAction) ac.getBean("userAction");  System.out.println(userAction);//cn.itcast.d\_di2.UserAction@898802  // 执行方法  userAction.execute();//打印:UserDao.save()  System.out.println("----------------");  userAction =  (UserAction)ac.getBean("userAction");//cn.itcast.d\_di2.UserAction@1405b61  System.out.println(userAction);  userAction.execute();//打印:UserDao.save()  }  } |

**2.3.3内部bean**

(1)bean.xml

|  |
| --- |
| <!-- 方式2： 内部bean -->  <!-- 总结： 当创建的对象，不被容器其他地方引用的时候，可以这样写！ 这样写不通用，内部对象只能用一次 -->  <!--action如是单例，内部bean默认是单例; action如是多例，内部bean就是多例 -->  <bean id="userAction" class="cn.itcast.d\_di2.UserAction" scope="prototype">  <property name="userService">  <bean class="cn.itcast.d\_di2.UserService">  <property name="userDao">  <bean class="cn.itcast.d\_di2.UserDao"></bean>  </property>  </bean>  </property>  </bean> |

(2)其它同2.3.2节-例2

**2.3.4 p名称空间**

|  |
| --- |
| <!-- 方式3： p名称空间，给属性注入值 -->  <bean id="userDao" class="cn.itcast.d\_di2.UserDao"></bean>  <bean id="userService" class="cn.itcast.d\_di2.UserService" p:userDao-ref="userDao"></bean>  <bean id="userAction" class="cn.itcast.d\_di2.UserAction"  p:userService-ref="userService"></bean> |

**2.3.5自动装配**

1.(1)bean.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:p="http://www.springframework.org/schema/p"  xmlns:context="http://www.springframework.org/schema/context"  xsi:schemaLocation="  http://www.springframework.org/schema/beans  http://www.springframework.org/schema/beans/spring-beans.xsd  http://www.springframework.org/schema/context  http://www.springframework.org/schema/context/spring-context.xsd"  default-autowire="byType">  <!-- 配置： 需要创建的对象、对象依赖关系处理 -->  <!-- 方式4：自动装配 (了解) -->  <!--  a. default-autowire="byType" 配置到全局  当前所有的bean都采用”根据类型自动装配“  b. 配置到bean节点  autowire="byName" 根据名称自动装配， 会去容器找指定名称的对象，  注入到set方法的参数中！  autowire="byType" 根据类型自动装配, 要确保改类型对应的对象在IOC  容器中唯一，否则报错！  总结：简化配置，但是维护麻烦！默认配置是no，推荐用这种  -->  <bean id="userDao" class="cn.itcast.e\_autowire.UserDao"></bean>  <bean id="userService" class="cn.itcast.e\_autowire.UserService"></bean>  <bean id="userAction" class="cn.itcast.e\_autowire.UserAction"></bean>  </beans> |

(2)其它同2.3.2节-例2

2. default-autowire自动装配类型

自动装配可使配置与java代码同步更新。例如，如果你需要给一个java类增加一个依赖，那么该依赖将被自动实现而不需要修改配置。因此强烈推荐在开发过程中采用自动装配，而在系统趋于稳定的时候改为显式装配的方式。

(1)no 默认.必须显示使用"<ref />"标签明确地指定bean合作者，对于部署给予更大的

控制和明了。

(2)byName 根据属性名自动装配。此选项将检查容器并根据名字查找与属性完全一致的bean，并将其与属性自动装配。若变量与id名称不一样，就报错。

例如，在bean定义中将 autowire设置为by name，而该bean包含master属性（同时提供setMaster(..)方法），Spring就会查找名为master的bean定义，并用它来装配给master属性。

|  |  |  |
| --- | --- | --- |
| 例:   |  | | --- | | Service.java  public class Service  {  Source source;  public void setSource(Source source)  {  this.source = source;  }  } |  |  | | --- | | applicationContext.xml  <beans  ...  default-autowire="byName">  <bean id="source" class="cn.hh.spring.DBCPSource" scope="prototype"/>  <bean id="service" class="cn.hh.spring.Service" scope="prototype">  </bean>  </beans> |   applicationContext.xml中并没给bean id="service"配Source属性，但在beans中设置了autowire="byName".配置文件会自动根据 cn.hh.spring.Service 中的setSource找bean id="Source"的bean ，然后自动配上去，如果没找到就不装配。  注意：byName的name是java中setXxxx 的Xxxx, 和上面设置的Source source中source拼写毫无关系，完全可以是  public class Service  {  Source source1;  public void setSource(Source source1)  {  this.source1 = source1;  }  }  结果相同。 |

1. byType 如果容器中存在一个与指定属性类型相同的bean，那么将与该属性自动装配。

如存在多个该类型的bean，将会抛出异常，并指出不能使用byType方式进行自动装配。

若没有找到相匹配的bean，则什么事都不发生，属性也不会被设置。

如不希望这样，可通过设置dependency-check="objects"让Spring抛出异常。

若变量与id不匹配，也没关系

(4)constructor 与byType的方式类似，不同之处在于它应用于构造器参数。如果在容器中没有找到与构造器参数类

型一致的bean，那么将会抛出异常。

(5)autodetect 通过bean类的自省机制（introspection）来决定是使用constructor还是byType方式进行自动装配。

如果发现默认的构造器，那么将使用byType方式。

**2.3.6注解**

2.3.6.1 @Repository、@Service、@Controller 和 @Component 将类标识为Bean

自 spring 2.0 引入注解。

1. @Repository

将数据访问层(DAO 层)的类标识为 Spring Bean。同时，为了让 Spring 能够扫描类路径中的类并识别出 @Repository 注解，需在 XML 配置文件中启用Bean 的自动扫描功能，可通过<context:component-scan/>实现：

|  |
| --- |
| //首先使用 @Repository 将 DAO 类声明为 Bean  package bookstore.dao;  @Repository  public class UserDaoImpl implements UserDao{ …… }  // 其次，在 XML 配置文件中启动 Spring 的自动扫描功能  <beans … >  ……  <context:component-scan base-package=”bookstore.dao” />  ……  </beans> |

就不再需要在 XML 中显式使用 <bean/> 进行Bean 的配置。Spring 在容器初始化时将自动扫描 base-package 指定的包及其子包下的所有 class文件，所有标注了 @Repository 的类都将被注册为 Spring Bean。

@Repository只能标注在DAO 类上:因为该注解的作用不只是将类识别为Bean,还能将所标注的类中抛出的数据访问异常封装为 Spring 的数据访问异常类型。Spring本身提供了一个丰富的并且是与具体的数据访问技术无关的数据访问异常结构，用于封装不同的持久层框架抛出的异常，使得异常独立于底层的框架。

1. @Component :泛化的概念，仅表示一个组件 (Bean) ，可作用在任何层次。
2. @Service 常作用在业务层，但目前该功能与 @Component 相同。
3. @Constroller 常作用在控制层，但目前该功能与 @Component 相同。

通过在类上使用 @Repository、@Component、@Service 和 @Constroller 注解，Spring会自动创建相应的 BeanDefinition 对象，并注册到 ApplicationContext 中。这些类就成了 Spring受管组件。这三个注解除了作用于不同软件层次的类，其使用方式与 @Repository 是完全相同的。

5.另外,用户可以创建自定义注解，然后在注解上标注@Component，那么,该自定义注解便具有了与所@Component 相同的功能。不过这个功能并不常用。

当一个 Bean 被自动检测到时，会根据那个扫描器的 BeanNameGenerator 策略生成它的 bean名称。默认情况下，对于包含 name 属性的 @Component、@Repository、 @Service 和@Controller，会把 name 取值作为 Bean 的名字。如果这个注解不包含 name值或是其他被自定义过滤器发现的组件，默认 Bean 名称会是小写开头的非限定类名。如果你不想使用默认 bean命名策略，可以提供一个自定义的命名策略。首先实现 BeanNameGenerator接口，确认包含了一个默认的无参数构造方法。然后在配置扫描器时提供一个全限定类名，如下所示：

|  |
| --- |
| <beans ...>  <context:component-scan  base-package="a.b" name-generator="a.SimpleNameGenerator"/>  </beans> |

与通过 XML 配置的 Spring Bean 一样，通过上述注解标识的Bean，其默认作用域是"singleton"，为了配合这四个注解，在标注 Bean 的同时能够指定 Bean 的作用域，Spring2.5 引入了 @Scope 注解。使用该注解时只需提供作用域的名称就行了，如下所示：

|  |
| --- |
| @Scope("prototype")  @Repository  public class Demo { … } |

如果你想提供一个自定义的作用域解析策略而不使用基于注解的方法，只需实现 ScopeMetadataResolver接口，确认包含一个默认的没有参数的构造方法。然后在配置扫描器时提供全限定类名：

|  |
| --- |
| <context:component-scan base-package="a.b"  scope-resolver="footmark.SimpleScopeResolver" /> |

6.@Autowired与@Resource的区别

(1)都可用来装配bean. 都可写在字段上,或写在setter方法上。

(2)@Autowired默认按类型装配（属于spring的），默认要求依赖对象必须存在，如要允许null 值，可设置@Autowired(required=false) ，如想使用名称装配可以结合@Qualifier注解进行使用，如下：

|  |
| --- |
| @Autowired() @Qualifier("baseDao")  private BaseDao baseDao; |

(3)@Resource(属于J2EE)，默认按名称进行装配，名称可通过name属性进行指定，

如没有指定name属性，当注解写在字段上时，默认取字段名进行按照名称查找，如果注解写在setter方法上默认取属性名进行装配。 当找不到与名称匹配的bean时才按照类型进行装配。但是需要注意的是，如果name属性一旦指定，就只会按照名称进行装配。

总结:name指定，只按名称;

~未指定，先按名称，再按类型

|  |
| --- |
| @Resource(name="baseDao")  private BaseDao baseDao; |

减少了与spring的耦合

2.3.6.2 例子

1.bean.xml

|  |
| --- |
| <!-- 注解的方式，实现springIOC容器配置 -->  <!-- 1. 开启注解扫描 -->  <context:component-scan base-package="cn.itcast.f\_anno"></context:component-scan>  <bean class="cn.itcast.f\_anno.UserDao"></bean> |

2. IUserDao.java

|  |
| --- |
| package cn.itcast.f\_anno;  public interface IUserDao {  // 模拟：保存对象  public abstract void save();  } |

UserDao.java

|  |
| --- |
| package cn.itcast.f\_anno;  import org.springframework.stereotype.Component;  import org.springframework.stereotype.Repository;  // 把userDao对象加入IOC容器  //@Component("userdao") // <bean id=userDao class="...">  //@Repository("userdao") // <bean id=userDao class="...">  @Repository //默认名称是类名，且第一个字母小写  public class UserDao implements IUserDao {  // 模拟：保存对象  public void save() {  System.out.println("UserDao.save()");  }  } |

3. IUserService.java

|  |
| --- |
| package cn.itcast.f\_anno;  public interface IUserService {  public abstract void save();  } |

UserService.java

|  |
| --- |
| package cn.itcast.f\_anno;  import javax.annotation.Resource;  import org.springframework.stereotype.Component;  import org.springframework.stereotype.Service;  //@Component("userService") // 把当前类加入IOC容器  //@Service("userService") // 把当前类加入IOC容器  @Service  public class UserService implements IUserService {  //@Resource(name = "userDao") //根据”userDao“去ioc容器找对象，  //找到后注入到当前方法参数  @Resource // 默认会根据private IUserDao userDao 这里的名称去容器找； 如果没有  //找到，再根据类型找，再没有找到就报错！  private IUserDao userDao;  public void save() {  userDao.save();  }  } |

4. UserAction.java

|  |
| --- |
| package cn.itcast.f\_anno;  import javax.annotation.Resource;  import org.springframework.context.annotation.Scope;  import org.springframework.stereotype.Controller;  //@Component("userAction") // 加入IOC容器  //@Controller("userAction") // 加入IOC容器  @Scope("prototype") // 指定加入ioc容器的对象为多例  @Controller  public class UserAction {  // @Resource(name = "userService") // ioc容器找名称是userService的对象并注入  @Resource  private IUserService userService;    public String execute(){  userService.save();  return "success";  }  } |

3.案例

Dao/service/action实例，处理依赖关系

\* 常用的通过set方法注入

\* 内部bean

\* p名称空间

\* 自动装配

\* 注解

注解总结：

@Component 表示一个组件(类)，把当前组件加入ioc容器

加入容器的组件的名称默认是类名第一个字母小写

@Component(“”) 指定加入ioc容器的组件类的类名

@Repository 标识是一个持久层的组件

@Service 标识是一个业务逻辑层的组件

@Controller 标识是一个控制层的组件

@Scope("prototype") 指定对象单例/多例

@Resource 1. 默认根据修饰的字段名称会取ioc容器找对象自动注入

找到后注入

2. 如果名称没有找到，再根据类型查找 找到后就立刻注入

如果改类型在ioc容器中有多个对象，报错！

3. 根据类型也没有找到对象，报错！

@Resource(name =””) 会根据指定的名称去容器找对象自动注入

配置方式与注解方式:

1. 配置， 便于维护

(配置过多，比较繁琐)

2. 注解， 开发方便

(简化配置，不利于后期维护，如果修改对象创建、关系处理，需要改代码！)

# 3. Spring 与Struts整合

Spring与struts整合，关键点:

把action的创建，交给spring的ioc容器！

所以，需要引入jar文件：

**struts2-spring-plugin-2.3.4.1.jar 【struts源码】**

**spring-web-3.2.5.RELEASE.jar 【spring源码】**

整合完整步骤：

1. 引入jar

Struts核心jar

Spring

Core 核心 （5个）

Web 对web支持 (2个)

2. 配置

bean.xml

struts.xml

web.xml

struts2核心过滤器

启动时候，配置加载springIOC容器

3. Action 类

任务：

1. 写简单案例，练习spring 、

\* 搭建spring环境

\* 测试OK

\* dao/service/action

然后处理对象依赖关系，

\* set方法注入值, 几种方式

\* 注解

2. spring与struts整合，小案例

3. 优化前面项目，

融入spring功能！