## 1.什么是spring

spring 是分层的JavaSE/EE轻量级应用开源框架，已控制反转IOC和面向切面编程AOP为核心，提供了展现层SpringMVC,

和持久层Srping JDBC以及事务管理等。

spring是一个开源框架，为了解决企业应用开发的复杂性而创建的，但现在不止应用于企业应用。  
同时是一个轻量级的控制反转ioc和面向切面编程的容器框架  
轻量：从大小与开销对于spring都是轻量的  
通过控制反转ioc的技术达到松耦合  
提供面向切面编程的丰富支持，允许通过分离应用的业务逻辑与系统服务进行内聚性的开发  
包含并管理应用对象的配置和生命周期，这个意义是容器  
将简单的组件配置组合成复杂的应用，这个意义是框架

框架：框架就是定制一套规范或者规则（思想），大家在该规范或思想下进行工作，或者说  
使用别人打好的舞台，你来做表演

框架于类库的区别  
框架一般是封装了逻辑的，高内聚的，类库则是松散的工具集合  
框架一般似乎专注于某一领域，类库则是更通用的

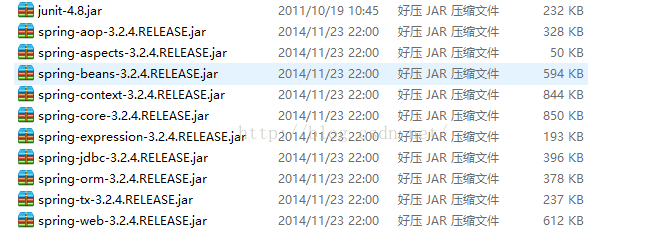
spring带来了复杂的JavaEE的春天

## 2.特点

方便解耦，简化开发  
spring提供ioc容器，可以将对象之间的依赖关系交给spring控制，避免硬编码所造成的过度程序耦合。  
  
  
AOP编程支持  
通过spring提供的aop功能，用户可以轻松的进行面向切面编程  
  
  
声明事务的支持  
用户可以通过spring来管理事务，提升开发效率  
  
  
方便程序的测试  
可以使用非容器的依赖的编程方式进行几乎所有的测试工作，在spring中，测试不再是  
昂贵的操作，而是随手可做的事  
方便集成各种优秀的框架  
spring不排斥各种优秀的框架，相反spring可以降低各种框架的使用难度。如可以集成（struts、Hibernate）  
  
降低JavaEE API的使用难度  
如JDBC、JavaMail、远程调用等

spring源码设计精妙、结构清晰，研究源码可以快速提升Java技术水平和开发应用水平

## 3.案例中使用的jar



## 4.直入主题IOC

#### 4.1小案例

    将对象的依赖交给配置文件来配置（配置文件的名字是可以任意的，不过一般写一个比较规范的名字）,这里使用IOC特性对类中的属性进行初始化

   使用junit来进行测试单元测试（注意：单元测试一些老的版本可能会存在bug,如calssNotFound...,建议下载新的junit版本）

   User.java 用户bean类

**[java]** [view plain](http://blog.csdn.net/u014292162/article/details/52277756) [copy](http://blog.csdn.net/u014292162/article/details/52277756)

1. **package** com.xxx.spring.ioc.bean;
3. **import** java.io.Serializable;
5. **public** **class** User **implements** Serializable {
6. **private** **static** **final** **long** serialVersionUID = 1L;
7. **private** **int** id; //用户编号
8. **private** String name; //用户名
9. **private** **int** age; //用户年龄
10. **private** String gender; //用户性别
12. **public** User() {}
14. **public** User(**int** id, String name, **int** age, String gender) {
15. **super**();
16. **this**.id = id;
17. **this**.name = name;
18. **this**.age = age;
19. **this**.gender = gender;
20. }
22. **public** **int** getId() {
23. **return** id;
24. }
26. **public** **void** setId(**int** id) {
27. **this**.id = id;
28. }
30. **public** String getName() {
31. **return** name;
32. }
34. **public** **void** setName(String name) {
35. **this**.name = name;
36. }
38. **public** **int** getAge() {
39. **return** age;
40. }
42. **public** **void** setAge(**int** age) {
43. **this**.age = age;
44. }
46. **public** String getGender() {
47. **return** gender;
48. }
50. **public** **void** setGender(String gender) {
51. **this**.gender = gender;
52. }
54. @Override
55. **public** String toString() {
56. **return** "User [id=" + id + ", name=" + name + ", age=" + age
57. + ", gender=" + gender + "]";
58. }
60. }

Spring中bean的配置：

这里说的Spring中的Bean概念，跟我们写JavaBean类不是一个概念，Spring中所有配置在xml中或使用spring来初始化的都叫Bean(dao,service,javaBean,Controller...)

IOC控制反转，控制权的转移，应用程序本身不负责依赖对象的创建和维护，而是有外部容器的创建和维护  
（就像我们需要房子，不是自己去画图纸，建房子而是去请开发商去做，或房屋中介住房）  
什么被反转了呢——————》获的对象的过程被反转了，依赖注入

set.xml配置文件初始化User.java中的相关属性，可以使用junit对其进行单元测试

**[html]** [view plain](http://blog.csdn.net/u014292162/article/details/52277756) [copy](http://blog.csdn.net/u014292162/article/details/52277756)

1. **<?xml** version="1.0" encoding="UTF-8"**?>**
2. **<beans** xmlns="http://www.springframework.org/schema/beans"
3. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4. xsi:schemaLocation="http://www.springframework.org/schema/beans
5. http://www.springframework.org/schema/beans/spring-beans-3.2.xsd"**>**
6. <!-- name中的user可以取别名    scope="prototype" 或singleton="false"可以设置为非单例模式 -->
7. **<bean** name="user,user2" class="com.xxx.spring.ioc.bean.User"**>**
8. **<property** name="id" value="1"**/>**
9. **<property** name="name" value="tom"**/>**
10. **<property** name="age" value="20"**/>**
11. **<property** name="gender" value="male"**/>**
12. **</bean>**
14. **</beans>**

测试：

**[java]** [view plain](http://blog.csdn.net/u014292162/article/details/52277756) [copy](http://blog.csdn.net/u014292162/article/details/52277756)

1. **import** org.junit.AfterClass;
2. **import** org.junit.BeforeClass;
3. **import** org.junit.Test;
4. **import** org.springframework.beans.factory.BeanFactory;
5. **import** org.springframework.beans.factory.xml.XmlBeanFactory;
6. **import** org.springframework.context.ApplicationContext;
7. **import** org.springframework.context.support.ClassPathXmlApplicationContext;
8. **import** org.springframework.context.support.FileSystemXmlApplicationContext;
9. **import** org.springframework.core.io.ClassPathResource;
11. **import** com.briup.spring.aop.bean.AwareTest;
12. **import** com.briup.spring.ioc.UserService;
13. **import** com.briup.spring.ioc.bean.Car;
14. **import** com.briup.spring.ioc.bean.Coll;
15. **import** com.briup.spring.ioc.bean.Life;
16. **import** com.briup.spring.ioc.bean.Student;
17. **import** com.briup.spring.ioc.bean.Teacher;
18. **import** com.briup.spring.ioc.bean.User;

21. **public** **class** SpringTest {
23. @BeforeClass
24. **public** **static** **void** setUpBeforeClass() **throws** Exception {
25. System.out.println("BeforeClass 标注的方法 会最先先被执行");
26. }
28. @AfterClass
29. **public** **static** **void** tearDownAfterClass() **throws** Exception {
30. System.out.println("AfterClass 标注的方法 会最后执行");
31. }
33. @Test
34. **public** **void** test() {
35. System.out.println("test");
36. //路经比较特殊
37. BeanFactory factory = **new** ClassPathXmlApplicationContext("com/xx/spring/chap1/ioc.xml");
38. UserService service = (UserService) factory.getBean("service");
39. service.getUserDao().save();
40. }
41. @Test
42. **public** **void** test2() {
43. BeanFactory factory = **new** ClassPathXmlApplicationContext("com/xxx/spring/chap1/set.xml");
44. //User user = (User) factory.getBean("user");
45. //User user = (User) factory.getBean("user",User.class);
46. //User user = (User) factory.getBean(User.class); //只有唯一的bean的时候才使用这种方式
47. //System.out.println(user);
48. System.out.println(factory.getType("user")); //获取user实例的类型
49. User user = (User) factory.getBean("user");
50. User user2 = (User) factory.getBean("user");
51. System.out.println(user == user2);//true -- 单例  --这是可以控制的在配置文件中 bean scope="prototype"-->会变成原型模式 这时结果会是false
52. System.out.println(factory.isPrototype("user"));//是否为原型     false
53. System.out.println(factory.isSingleton("user"));//是否为单例     true
55. System.out.println(factory.isTypeMatch("user", User.**class**));//判断 user实例是否为这种类型 true
57. String[] str = factory.getAliases("user"); //获取别名
58. **for**(**int** i=0;i<str.length;i++){
59. System.out.println(str[i]);//user2
60. }
61. }
62. }

使用junit测试时候，注解@BeforeClass的会先于@Test注解的方法运行，@AfterClass最后运行，junit相关的jar最好使用4.4以上的版本

上边的案例factory.getBean("user",User.class);第一参数是set.xml文件中对应bean的name值或id值.

**[java]** [view plain](http://blog.csdn.net/u014292162/article/details/52277756) [copy](http://blog.csdn.net/u014292162/article/details/52277756)

1. System.out.println(user == user2);//true

上边返回true,说明默认的是单利模式，可以通过scope改变其范围为scope="prototype"变为原型模式，这样每次初始化bean对象的时候，都会返回一个新的。

#### 4.2Bean容器的初始化

Bean容器的初始化  
两个基础包：  
org.springframework.beans  
org.springframework.context  
BeanFactory提供配置结构和基本功能，加载并初始化Bean  
ApplicationContext保存了Bean对象并在spring中被广泛使用

集中常用的使用场景：

常用的文件初始化方式：

FileSystemXmlApplicationContext context = new FileSystemXmlApplicationContext("F:/workspace/appcontext.xml");  
ApplicationContext ac = new ClassPathXmlApplicationContext("com/xxx/spring/chap1/coll.xml");  
BeanFactory factory = new ClassPathXmlApplicationContext("com/xxxspring/chap1/ioc.xml");

在webapp中的我们一般配置到web.xml文件中

1.

**[html]** [view plain](http://blog.csdn.net/u014292162/article/details/52277756) [copy](http://blog.csdn.net/u014292162/article/details/52277756)

1. <!-- 配置contextConfigLocation指定spring将要使用的配置文件 -->
2. **<context-param>**
3. **<param-name>**contextConfigLocation**</param-name>**
4. **<param-value>**classpath:action.xml,classpath:dao.xml,classpath:service.xml**</param-value>**
5. **</context-param>**
6. <!--  配置listner让spring读取配置文件-->
7. **<listener>**
8. **<listener-class>**org.springframework.web.context.ContextLoaderListener**</listener-class>**
9. **</listener>**

2.load-on-startup标签指定启动顺序，1为指在启动服务器的时候初始化容器

**[html]** [view plain](http://blog.csdn.net/u014292162/article/details/52277756) [copy](http://blog.csdn.net/u014292162/article/details/52277756)

1. **<listener>**
2. **<listener-class>**org.springframework.web.context.ContextLoaderListener**</listener-class>**
3. **</listener>**
5. lt;servlet**>**
6. **<servlet-name>**remoting**</servlet-name>**
7. **<servlet-class>**org.springframework.web.servlet.DispatcherServlet**</servlet-class>**
8. **<init-param>**
9. **<param-name>**contextConfigLocation**</param-name>**
10. **<param-value>**classpath:spring-remoting-servlet.xml**</param-value>**
11. **</init-param>**
12. **<load-on-startup>**1**</load-on-startup>**
13. **</servlet>**

#### 4.3Bean的两种注入方式

a.设置值注入

b.构造注入

设置值注入案例：

基本类型的注入： 通过<property name="属性名", value="属性值/">为对应类对象初始化的值，这种方式必须在类中为对应的属性提供getxxx,setxx方法

**[html]** [view plain](http://blog.csdn.net/u014292162/article/details/52277756) [copy](http://blog.csdn.net/u014292162/article/details/52277756)

1. **<?xml** version="1.0" encoding="UTF-8"**?>**
2. **<beans** xmlns="http://www.springframework.org/schema/beans"
3. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4. xsi:schemaLocation="http://www.springframework.org/schema/beans
5. http://www.springframework.org/schema/beans/spring-beans-3.2.xsd"**>**
6. **<bean** name="user,user2" class="com.xxx.spring.ioc.bean.User"**>**
7. **<property** name="id" value="1"**/>**
8. **<property** name="name" value="tom"**/>**
9. **<property** name="age" value="20"**/>**
10. **<property** name="gender" value="male"**/>**
11. **</bean>**
12. **</beans>**

引用类型的注入：<property name="属性名" ref="引用的bean"></property>,被引入的bean和引入处可以不在同一个xml文件中，因为所有bean都会被

容器初始化并保存到容器中

**[html]** [view plain](http://blog.csdn.net/u014292162/article/details/52277756) [copy](http://blog.csdn.net/u014292162/article/details/52277756)

1. **<?xml** version="1.0" encoding="UTF-8"**?>**
2. **<beans** xmlns="http://www.springframework.org/schema/beans"
3. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4. xsi:schemaLocation="http://www.springframework.org/schema/beans
5. http://www.springframework.org/schema/beans/spring-beans-3.2.xsd"**>**
6. **<bean** name="memberService" class="com.xxx.run.service.impl.IMemberServiceImpl"**>**
7. **<property** name="memberDao" ref="memberDao"**></property>**
8. **</bean>**
9. **<bean** name="memberDao" class="com.xxx.run.dao.impl.IMemberDaoImpl"**>**
10. **</bean>**
11. **</beans>**

构造注入

顾名思义，使用构造器对对象的初始化注入对应的值，实现方式有如下3种

**[html]** [view plain](http://blog.csdn.net/u014292162/article/details/52277756) [copy](http://blog.csdn.net/u014292162/article/details/52277756)

1. **<?xml** version="1.0" encoding="UTF-8"**?>**
2. **<beans** xmlns="http://www.springframework.org/schema/beans"
3. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4. xsi:schemaLocation="http://www.springframework.org/schema/beans
5. http://www.springframework.org/schema/beans/spring-beans-3.2.xsd"**>**
6. **<bean** name="teacher" class="com.xxx.spring.ioc.bean.Teacher"**>**
7. <!--  1.按照属性名赋值 ，调用有参数的构造器，顺序是参数顺序-->
8. **<constructor-arg** name="id" value="1"**/>** <!-- person(int id,String name, String gender) -->
9. **<constructor-arg** name="name" value="tom"**/>**
10. **<constructor-arg** name="gender" value="male"**/>**
11. <!-- 2.index从0开始，按照属性在构造器中出现的顺序赋值 索引值是构造器中的属性顺序 -->
12. <!-- **<constructor-arg** index="0" value="2"**/>**
13. **<constructor-arg** index="1" value="jack"**/>**
14. **<constructor-arg** index="2" value="male"**/>** --**>**
15. <!-- 3.按照类型进行赋值，如果出现相同的类型，按照属性在构造器中出现的顺序进行复制 -->
16. <!--     **<constructor-arg** type="int" value="3"**/>**
17. **<constructor-arg** type="String" value="rose"**/>**
18. **<constructor-arg** type="String" value="female"**/>** --**>**
19. **</bean>**
20. **</beans>**

Teacher.java

**[java]** [view plain](http://blog.csdn.net/u014292162/article/details/52277756) [copy](http://blog.csdn.net/u014292162/article/details/52277756)

1. **public** **class** Teacher **implements** Serializable{
2. **private** **static** **final** **long** serialVersionUID = 1L;
3. **private** **int** id;
4. **private** String name;
5. **private** String gender;
7. **public** Teacher(**int** id, String name, String gender) {
8. **super**();
9. **this**.id = id;
10. **this**.name = name;
11. **this**.gender = gender;
12. }
14. @Override
15. **public** String toString() {
16. **return** "Teacher [id=" + id + ", name=" + name + ", gender=" + gender
17. + "]";
18. }
19. }

测试

**[java]** [view plain](http://blog.csdn.net/u014292162/article/details/52277756) [copy](http://blog.csdn.net/u014292162/article/details/52277756)

1. @Test
2. **public** **void** test3() **throws** Exception {
3. ApplicationContext ac = **new** ClassPathXmlApplicationContext("com/xxx/spring/chap1/constructor.xml");
4. Teacher teacher = (Teacher) ac.getBean("teacher");
5. System.out.println(teacher);//Teacher [id=1, name=tom, gender=male]
6. }

## 5.Bean

下边我们来了解一下Bean的：

Bean的作用域  
Bean的生命周期  
Bean的自动装配  
Resources和ResourceLoader

#### 5.1Bean的生命周期

Spring中Bean的声明周期的创建和初始化都是由Spring管理创建的，如下Life实现BeanNameAware,BeanFacotryAware,可以获取到一些对应的资源

**[java]** [view plain](http://blog.csdn.net/u014292162/article/details/52277756) [copy](http://blog.csdn.net/u014292162/article/details/52277756)

1. **import** org.springframework.beans.BeansException;
2. **import** org.springframework.beans.factory.BeanFactory;
3. **import** org.springframework.beans.factory.BeanFactoryAware;
4. **import** org.springframework.beans.factory.BeanNameAware;
6. **public** **class** Life **implements** BeanNameAware,BeanFactoryAware{
7. **private** String name;
9. **public** Life(){//一加载就会调到用
10. System.out.println("调用无参构造器");
11. }
13. **public** String getName() {
14. **return** name;
15. }
17. **public** **void** setName(String name) {
18. System.out.println("调用setName方法");
19. **this**.name = name;
20. }
22. **public** **void** myInit() {
23. System.out.println("调用myInit方法");
24. }
26. **public** **void** myDestory(){
27. System.out.println("调用myDestory方法");
28. }
30. @Override
31. **public** **void** setBeanFactory(BeanFactory arg0) **throws** BeansException {
32. System.out.println("调用setBeanFactory方法");
34. }
36. @Override
37. **public** **void** setBeanName(String arg0) {
38. System.out.println("调用setBeanName方法");
39. }
40. }

life.xml文件配置

init-method指明bean初始化需要执行的方法，

destory-method指明bean销毁需要执行的方法

**[html]** [view plain](http://blog.csdn.net/u014292162/article/details/52277756) [copy](http://blog.csdn.net/u014292162/article/details/52277756)

1. **<?xml** version="1.0" encoding="UTF-8"**?>**
2. **<beans** xmlns="http://www.springframework.org/schema/beans"
3. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4. xmlns:u="http://www.springframework.org/schema/util"
5. xsi:schemaLocation="http://www.springframework.org/schema/beans
6. http://www.springframework.org/schema/beans/spring-beans-3.2.xsd
7. http://www.springframework.org/schema/util
8. http://www.springframework.org/schema/util/spring-util-3.2.xsd"**>**
9. <!-- 调用set方法赋值后会调用myInit方法    myDestory方法最后调用-->
10. **<bean** name="life" class="com.xxx.spring.ioc.bean.Life" init-method="myInit" destroy-method="myDestory"**>**
11. **<property** name="name" value="tom"**></property>**
12. **</bean>**
13. **</beans>**

测试：

注意：在xml文件中指明destory-method需要执行的方法后，bean生命周期并不会自动去掉用myDestory方法，需要ac.detstory(),才会调用

**[java]** [view plain](http://blog.csdn.net/u014292162/article/details/52277756) [copy](http://blog.csdn.net/u014292162/article/details/52277756)

1. @Test
2. **public** **void** life(){//springBean的生命周期
3. ClassPathXmlApplicationContext ac = **new** ClassPathXmlApplicationContext("com/xxx/spring/chap2/life.xml");
4. Life life = ac.getBean("life",Life.**class**);
5. System.out.println(life);
6. ac.destroy();
7. }

结果：

调用无参构造器  
调用setName方法  
调用setBeanName方法  
调用setBeanFactory方法  
调用myInit方法  
com.briup.spring.ioc.bean.Life@4f0b5b  
调用myDestory方法  
AfterClass 标注的方法 会最后执行

#### 5.2Bean作用域

介绍两个常用的

scope="prototype"原型模式，该模式下每次都会创建一个新的对象

<bean id="user" class="com.briup.spring.ioc.bean.User" scope="prototype"></bean>

scope = "singleton" 单例模式，该模式下容器中只会存在一个这样的对象

<bean id="user" class="com.briup.spring.ioc.bean.User" scope="singleton"></bean>

其他的参考：

http://blog.csdn.net/fengruifang/article/details/6522262

5.3Bean的自动装配

上边的Bean不管是通过设置值注入，或通过构造器注入，我们都显示的声明了需要注入的值

还有一种方式也比较常用，对应引用类型的注入我们可以通过autowire自动注入

autowire有三种方式

1.constructor

如下边我们不显示声明address属性的值，autowire=“constructor”,即初Student中提供了

public Student(Address address) {  
this.address = address;  
}

构造器，初始化Bean的时候，会去容器中查找Address.class对象是否存在，如果存在，则注入到该bean对象中，如不存在则address为null

**[html]** [view plain](http://blog.csdn.net/u014292162/article/details/52277756) [copy](http://blog.csdn.net/u014292162/article/details/52277756)

1. **<?xml** version="1.0" encoding="UTF-8"**?>**
2. **<beans** xmlns="http://www.springframework.org/schema/beans"
3. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4. xsi:schemaLocation="http://www.springframework.org/schema/beans
5. http://www.springframework.org/schema/beans/spring-beans-3.2.xsd"**>**
6. **<bean** name="student" class="com.xxx.spring.ioc.bean.Student" autowire="constructor"**>**<!-- byName byType  constructor(一定要提供一个带参数的构造器)-->
7. **<property** name="name" value="tom"**/>**
8. **<property** name="age" value="20"**/>**
9. <!--     <property name="address" ref="address"/> -->
10. **</bean>**
11. **<bean** name="address" class="com.briup.spring.ioc.bean.Address"**>**
12. **<property** name="country" value="中国"**></property>**
13. **<property** name="province" value="江苏"**></property>**
14. **<property** name="city" value="苏州"**></property>**
15. **</bean>**
16. **</beans>**

2.byName

同理，autowire=“byName”,初始化bean的时候，会去容器中查找名字为address的bean注入到student对象中，Student需要提供对应的getxx,setxx

3.byType

autowire=“byType”,初始化bean的时候，会取容器中通过类型查找是否有Class为Address.class类型的对象，查找到则注入到student对象中需要提供getxx,setxx

#### 5.3 Aware

spring中提供了许多已Aware结尾的类，这些类可以获取容器中的一些资源

比如ApplicationContextAware，可以获取applicationCcontext中的内容

BeanNameAware可以获取到Bean的beanName

aware.xml

**[html]** [view plain](http://blog.csdn.net/u014292162/article/details/52277756) [copy](http://blog.csdn.net/u014292162/article/details/52277756)

1. **<?xml** version="1.0" encoding="UTF-8"**?>**
2. **<beans** xmlns="http://www.springframework.org/schema/beans"
3. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4. xsi:schemaLocation="http://www.springframework.org/schema/beans
5. http://www.springframework.org/schema/beans/spring-beans-3.2.xsd"**>**
6. **<bean** name="applicationAawareTest" class="com.xxx.spring.aop.bean.AwareTest"**></bean>**
7. **</beans>**

AwareTest.java

**[java]** [view plain](http://blog.csdn.net/u014292162/article/details/52277756) [copy](http://blog.csdn.net/u014292162/article/details/52277756)

1. **import** org.springframework.beans.BeansException;
2. **import** org.springframework.beans.factory.BeanNameAware;
3. **import** org.springframework.context.ApplicationContext;
4. **import** org.springframework.context.ApplicationContextAware;
6. **public** **class** AwareTest **implements** ApplicationContextAware,BeanNameAware{
8. @Override
9. **public** **void** setApplicationContext(ApplicationContext applicationContext) **throws** BeansException {
10. System.out.println(applicationContext.getBean(AwareTest.**class**));
11. }
13. @Override
14. **public** **void** setBeanName(String beanName) {
15. System.out.println(beanName);
16. }
18. }

测试：

**[java]** [view plain](http://blog.csdn.net/u014292162/article/details/52277756) [copy](http://blog.csdn.net/u014292162/article/details/52277756)

1. @Test
2. **public** **void** AwareTest(){
3. ClassPathXmlApplicationContext ac = **new** ClassPathXmlApplicationContext("com/xxx/spring/chap1/aware.xml");
4. AwareTest awareTest = ac.getBean("applicationAawareTest",AwareTest.**class**);
5. System.out.println(awareTest);
6. }

结果：

applicationAawareTest  
com.xxx.spring.aop.bean.AwareTest@1d8fe20  
com.xxx.spring.aop.bean.AwareTest@1d8fe20

#### 5.4Resource统一文件资源接口

Resources针对文件的统一接口，用于操作本地资源或网络资源，或其他  
-UrlResource:URL对应的资源，根据一个URL地址既可以构建  
-ClassPathResource:获取类路径下的资源文件  
-FileSystemResource：获取文件系统中的资源文件  
-ServletContextResource:ServletContext封装资源，用于访问ServletContext环境下的资源  
-InputStreamResource:针对输入流封装的资源  
-ByteArrayResource:针对字节数组封装的资源

 ResourceLoader  
-所用的application context 实现了ResourceLoader接口

spring中ResourceLoader定义如下：

**[java]** [view plain](http://blog.csdn.net/u014292162/article/details/52277756) [copy](http://blog.csdn.net/u014292162/article/details/52277756)

1. **public** **interface** ResourceLoader{
2. Resource getResource(String location);
3. }

 getResource中location的写法有如下几种  
prefix前缀  案例     说明  
classpath:  classpath:com/briup/spring/chap2/life.xml 从classpath中加载  
file： file:/data/life.xml用URL从文件系统中加载  
http: http://myserver/logoo.png通过URL从网络加载  
（none）  /spring/chap2/life.xml 这种相对路径的写法依赖于ApplicationContext

spring中的使用

Resource template = ctx.getResource("some/resource/path/myTemplate.txt");  
Resource template = ctx.getResource("classpath:some/resource/path/myTemplate.txt");  
Resource template = ctx.getResource("file:some/resource/path/myTemplate.txt");

案例：

resources.xml

**[html]** [view plain](http://blog.csdn.net/u014292162/article/details/52277756) [copy](http://blog.csdn.net/u014292162/article/details/52277756)

1. **<?xml** version="1.0" encoding="UTF-8"**?>**
2. **<beans** xmlns="http://www.springframework.org/schema/beans"
3. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4. xsi:schemaLocation="http://www.springframework.org/schema/beans
5. http://www.springframework.org/schema/beans/spring-beans-3.2.xsd"**>**
6. **<bean** name="resourcetest" class="com.briup.spring.aop.bean.ResourceTest"**/>**
7. **</beans>**

ResourceTest.java

由于spring中所有的applicationcontext实现了ContextLoader接口， 所以我们实现applicationContext即有了ResourceLoader的能力

下边：classpath：在eclipse中会加载src下的config.txt文件

**[java]** [view plain](http://blog.csdn.net/u014292162/article/details/52277756) [copy](http://blog.csdn.net/u014292162/article/details/52277756)

1. **import** java.io.IOException;
3. **import** org.springframework.beans.BeansException;
4. **import** org.springframework.context.ApplicationContext;
5. **import** org.springframework.context.ApplicationContextAware;
6. **import** org.springframework.core.io.Resource;

9. //所有的ApplicationContext实现了ResourceLoader接口
10. **public** **class** ResourceTest **implements** ApplicationContextAware{
12. **private** ApplicationContext ApplicationContext;
14. @Override
15. **public** **void** setApplicationContext(ApplicationContext applicationContext) **throws** BeansException {
16. **this**.ApplicationContext = applicationContext;
17. }
19. **public** **void** resource() **throws** IOException{
20. //Resource resource = ApplicationContext.getResource("config.txt");//默认为classpath
21. //Resource resource = ApplicationContext.getResource("classpath:config.txt");
22. //Resource resource = ApplicationContext.getResource("file:D:\\workspace\\xnxy\_spring\\src\\config.txt");
23. Resource resource = ApplicationContext.getResource("url:http://repo.springsource.org/libs-release-local/org/springframework/spring/3.2.4.RELEASE/spring-framework-3.2.4.RELEASE-dist.zip");
24. System.out.println(resource.getFilename());//获取文件名
25. System.out.println(resource.contentLength()); //获取文件长度
26. System.out.println(resource.getInputStream());//获取输入流
27. }
28. }

测试：

**[html]** [view plain](http://blog.csdn.net/u014292162/article/details/52277756) [copy](http://blog.csdn.net/u014292162/article/details/52277756)

1. @Test
2. public void ResourceTest(){
3. ClassPathXmlApplicationContext ac = new ClassPathXmlApplicationContext("com/briup/spring/chap1/resources.xml");
4. ResourceTest resourceTest = ac.getBean("resourcetest",ResourceTest.class);
5. try {
6. resourceTest.resource();
7. } catch (IOException e) {
8. e.printStackTrace();
9. }
10. }

## 6.Bean容器的注解实现

下边我们主要了解如下使用：

Classpath扫描与组件管理  
类的自动检测与注册Bean  
<context:annotation-config/>  
@Component, @Repository, @Service, @Constroller  
@Required  
@Autowired  
@Qualifier  
@Resource

#### 6.1classpath扫描与组件管理

从Spring3.0开始，Spring JavaConfig项目提供了很多特性，包括使用java而不是XML定义Bean,比如  
@Configuration, @Bean, @Import, @DependsOn  
@Component是Spring中的一个通用注解，可以用于任何Bean,相当于注解的超类,如果不知道位于那个层，一般使用该注解  
@Repository, @Service, @Controller是更具有针对性的注解  
- @Repository,通常用于注解DAO,即持久层的注解  
- @Service,通常用于追注解Service类，即服务层  
- @Controller通常用于注解Controller，即控制层（MVC）

#### 6.2类的自动检测与注册Bean

我们再xml下配置如下标签，可以指定某个包路劲，扫描该包以及子包下，使用了spring注解的bean注册到容器中,basp-package指向扫描那个包下的注解类

**[html]** [view plain](http://blog.csdn.net/u014292162/article/details/52277756) [copy](http://blog.csdn.net/u014292162/article/details/52277756)

1. **<context:component-scan** base-package="spring.aop.bean.annotation"**></context:component-scan>**

我们还可以使用如下标签，context:annotation-config，不过context:component-scan包含context:annotation-config的全部功能，通常使用前者后，不再使用后者context:component-scan一般用于基于类的注解（包括成员变量或成员方法的注解）,但是context:annotation-config只能在完成bean注册后，去处理bean类中的成员变量或成员方法的注解.

过虑注解：

 <!--默认情况下，spring中自动发现并被注册bean的条件是：  
      使用@Component, @Repository, @Service, @Constroller其中之一的注解  
      或者使用基于@Component的自定义注解  
        
      可以通过过滤器修改上边的行为，如下边的例子XML配置忽略所有@Repository注解并用“stub”代替  
        -->

**[html]** [view plain](http://blog.csdn.net/u014292162/article/details/52277756) [copy](http://blog.csdn.net/u014292162/article/details/52277756)

1. **<context:component-scan** base-package="spring.aop.bean.annotation"**>**
2. <!--  -->
3. **<context:include-filter** type="regex" expression=".\*Stub.\*Repository"**/>**
4. <!-- 排除@Repository注解 -->
5. **<context:exclude-filter** type="annotation" expression="org.springframework.stereotype.Repository"**/>**
6. **</context:component-scan>**

#### 6.3使用注解管理bean

这里我们先使用 @Bean @Service @Repository @Componet

扫描过程中组件自动检测，那么Bean名称是有BeanNameGenerator生成的( @Component, @Repository, @Service, @Controller  
都会有个name属性用于显示设置BeanName)

Service

//显示设置beanName,相当于在xml配置bean的是id的值  
@Service("myMoveLister")  
public class simpleLlister{  
//..

}

Dao

//设置beanName默认使用类名，首字母小写作为beanName  
@Repository  
public class MovieFinderImpl implements MovieFinder{  
  
}

##### 6.3.1 作用域scope

作用域的注解Scope  
通常情况下自动查找的Spring组件，其Scope是singleton,其Spring2.5提供了Scope的注解 @Scope  
  
@Scope("prototype")  //括号中指定Scope的范围,默认  
@Repository  
public class MovieFinderImpl implements MovieFinder{  
  
  
}  
  
  
也可以自定义scope策略，实现ScopeMetadataResolver接口并提供一无参数的构造器  
  
 <context:component-scan base-package="spring.aop.bean.MyScopeResolver"></context:component-scan>

##### 6.3.2注解的具体案例使用

//由于不知道其作用于DAO或Service所以使用通用注解，如果知道具体作用在那层，我们一班使用更具体注解方式如@Service,@Repository等

**[java]** [view plain](http://blog.csdn.net/u014292162/article/details/52277756) [copy](http://blog.csdn.net/u014292162/article/details/52277756)

1. //@Component -->默认使用类名小写作为bean的name
2. @Scope("prototype") //括号中为Scope的范围,这里设置为原型模式
3. @Component("beanAnnotation")
4. **public** **class** BeanAnnotation {
6. **public** **void** say(String arg){
7. System.out.println("BeanAnnotation: "+arg);
8. }
9. }

测试：

**[java]** [view plain](http://blog.csdn.net/u014292162/article/details/52277756) [copy](http://blog.csdn.net/u014292162/article/details/52277756)

1. @Test
2. **public** **void** testAnnotation(){
3. ClassPathXmlApplicationContext ac = **new** ClassPathXmlApplicationContext("com/xxx/spring/chap4/annotation.xml");
4. //@Component没有value值的话，默认使用类名首字母小写作为bean的id，指定value以value值为准作为id
5. BeanAnnotation beanAnnotation1  = ac.getBean("beanAnnotation",BeanAnnotation.**class**);
6. BeanAnnotation beanAnnotation2  = ac.getBean("beanAnnotation",BeanAnnotation.**class**);
7. System.out.println(beanAnnotation1);
8. System.out.println(beanAnnotation2);
9. //结果
10. //com.xxx.spring.aop.bean.annotation.BeanAnnotation@1598d5f
11. //com.xxx.spring.aop.bean.annotation.BeanAnnotation@505fd8
12. }

##### 6.3.3一个不常用的注解@Required

@Required注解使用于注解bean属性的setter方法  
这个注解仅仅标识，受影响的bean属性必须在配置时被填充，通过bean定义或通过自动装配一个明确的属性值

这个注解并不是很常用，更常用的是@Autowired

pulic class simpleMovieLister{  
  
private MoiveFinder movieFinder;  
  
@Required  
public void setMovieFinder(MovieFinder movieFinder){  
this.movieFinder  = movieFinder;  
}  
//..  
}

##### 6.3.4@Autowired

这个注解相当于我们之前在xml文件中配置的autowire="constructor/byName/byType",只不过我们这里使用@Autowired方式注解方式，且默认是通过类型判断，意思就是不使用byName,和construtor。通过@Autowired注解，spring会自动去容器中查找对应的类型，注入到该属性中，且bean类中，使用@Autowired注解其属性，我们可以不用提供getter,setter方法

使用@Autowired

@Autowried对属性进行注解的时候，我们可以省略getter,setter方法，通过对应的bean的类型，对属性值注入  
@Autowried对seter方法进行注解的时候，可以注入对应的值  
@Autowried对构造器进行注解的时候，可以通过类型找到对应的bean注入  
@Autowried可以将 @Autowried为”传统“的setter方法代替 @Required  
@Autowried自动注入，会去容器中按照类型查找对应的bean注入

案例：

setter中使用

**[java]** [view plain](http://blog.csdn.net/u014292162/article/details/52277756) [copy](http://blog.csdn.net/u014292162/article/details/52277756)

1. pulic **class** simpleMovieLister{
3. **private** MoiveFinder movieFinder;
5. @Autowried
6. **public** **void** setMovieFinder(MovieFinder movieFinder){
7. **this**.movieFinder  = movieFinder;
8. }
9. //..
10. }

属性和构造器中使用

**[java]** [view plain](http://blog.csdn.net/u014292162/article/details/52277756) [copy](http://blog.csdn.net/u014292162/article/details/52277756)

1. pulic **class** MovieRreCommender{
3. 成员变量中
4. @Autowried
5. **private** MovieCatalog movieCatalog;
7. **private** CustomerPreferenceDao customerPreferenceDao;
9. //构造器中
10. @Autowried
11. **public** MovieRreCommender(CustomerPreferenceDao customerPreferenceDao){
12. **this**.CustomerPreferenceDao  = CustomerPreferenceDao;
13. }
14. }

上边的seter方式，构造器方式，属性方式，效果都是一样的，使用其中任何一种，都可以实现注入。不过由于，@Autowired是通过类型判断是否注入到使用该注解地方，假如容器中出现两个以上的相同类型的bean实例，就会报错，这时我们就必须指定注入那个id名的bean实例，主要有两种方法解决该问题：

@Autowired(requried=false), @Qualifie("beanName)指定@Autowired注入那个bean实例

##### 6.3.5@Autowried(requried=false)

默认情况下，如果因找不到合适的bean将会导致autowiring失败抛出异常，可以通过下边  
这种方式避免  
  
pulic class simpleMovieLister{  
  
private MoiveFinder movieFinder;  
  
@Autowried(requried=false)//指明该属性不是必须的，找不到的情况下不会抛出异常  
public void setMovieFinder(MovieFinder movieFinder){  
this.movieFinder  = movieFinder;  
}  
//..  
}  
提示：每一类中只能有一个构造器被标记为requried=ture建议将 @Autowired的必要属性时，使用 @Requried注解

##### 6.3.6@Qualifier--配合 @Autowired

如果使用 @Autowired自动装配可能存在多个相同类型的bean的时候，可以使用spring的 @Qualifier  
注解缩小注解范围（或指定唯一），也可以用于指定单独的构造参数的方法参数  
可以适用于注解集合类型的变量  
  
案例：

**[java]** [view plain](http://blog.csdn.net/u014292162/article/details/52277756) [copy](http://blog.csdn.net/u014292162/article/details/52277756)

1. **public** **class** MovieRecommander{
2. @Autowired
3. @Qualifier("beanName")
4. **private** MovieCatalog movieCatalog;
6. **private** CustomerPreferenceDao customerPreferenceDao;
7. <span style="white-space:pre">    </span>//@Qualifier也可以实现参数的注入
8. **public** **void** prepare(@Qualifier("beanName")CustomerPreferenceDao customerPreferenceDao){
9. **this**.customerPreferenceDao = customerPreferenceDao;
10. }
11. }

上边的案例：假设MovieCatalog在容器中存在多个相同的类型的情况下，可以结合使用 @Qualifier("beanName")  
指定一个bean的id注入到该属性中，可以在方法的参数中使用

##### 6.3.7@Autowired注解可以方便的注解那些众所周知的解析依赖性接口

比如说：BeanFacotry,ApplicationContext,Environment,ResourceLoader,ApplicaiontEventPublisher, MessageSource等

**[java]** [view plain](http://blog.csdn.net/u014292162/article/details/52277756) [copy](http://blog.csdn.net/u014292162/article/details/52277756)

1. pulic **class** simpleMovieLister{
3. @Autowired
4. **private** AplicationContext context;
6. **public** simpleMovieLister(){}
8. }

上边的案例使用autowired注解ApplicationContext，这样我们就可以活ApplicatioinContext容器总的bean对象

##### 6.3.8@Autowired将容器中相关类型的bean注入到一个集合或数组中

使用@Autowired注解，可以将容器中的bean实例注入到集合或数组中，如果是注入到数组中通过配合@Order("排序值")，可以实现对数组或list的排序，也只能对数组或list排序，其他的如Map是不需要排序的。

案例：接口BeanInfterface

**[java]** [view plain](http://blog.csdn.net/u014292162/article/details/52277756) [copy](http://blog.csdn.net/u014292162/article/details/52277756)

1. **public** **interface** BeanInterface {

4. }

实现类1：

**[java]** [view plain](http://blog.csdn.net/u014292162/article/details/52277756) [copy](http://blog.csdn.net/u014292162/article/details/52277756)

1. @Order(1)
2. @Component
3. **public** **class** BeanImplOne **implements** BeanInterface {
5. }

实现类2：

**[java]** [view plain](http://blog.csdn.net/u014292162/article/details/52277756) [copy](http://blog.csdn.net/u014292162/article/details/52277756)

1. @Order(2)  //Order排序注解只对list,或数组集合有效括号里边是排序顺序
2. @Component
3. **public** **class** BeanImplTwo **implements** BeanInterface {
5. }

调用类：

**[java]** [view plain](http://blog.csdn.net/u014292162/article/details/52277756) [copy](http://blog.csdn.net/u014292162/article/details/52277756)

1. **import** java.util.List;
2. **import** java.util.Map;
3. **import** java.util.Map.Entry;
4. **import** java.util.Set;
5. **import** org.springframework.beans.factory.annotation.Autowired;
6. **import** org.springframework.stereotype.Component;
8. @Component
9. **public** **class** BeanInvoker {
11. @Autowired //该注解会将所有的BeanInterface类型的bean注入到该list中
12. //如果bean有 @Order注解可以实现排序
13. **private** List<BeanInterface> list;
15. //该注解会将所有的BeanInterface类型的bean注入到该map中，key值为bean的名字
16. //是String类型，map类型无排序可言
17. @Autowired
18. **private** Map<String, BeanInterface> map;
20. **public** **void** print(){
21. **if**(list != **null** && 0 != list.size()){
22. System.out.println("list...");
23. **for**(BeanInterface beanInterface:list){
24. System.out.println(beanInterface.getClass().getName());
25. }
26. }
27. **if**(map != **null** && 0 != map.size()){
28. System.out.println("map...");
29. Set<Entry<String, BeanInterface>> entrySet = map.entrySet();
30. **for**(Entry<String, BeanInterface> entry: entrySet){
31. System.out.println(entry.getKey()+"--"+entry.getValue().getClass().getName());
32. }
33. }
34. }
35. }

测试类：

**[java]** [view plain](http://blog.csdn.net/u014292162/article/details/52277756) [copy](http://blog.csdn.net/u014292162/article/details/52277756)

1. @Test
2. **public** **void** testAutowired2(){
3. ClassPathXmlApplicationContext ac = **new** ClassPathXmlApplicationContext("com/xxx/spring/chap4/annotation.xml");
4. BeanInvoker beanInvoker = (BeanInvoker) ac.getBean("beanInvoker");
5. beanInvoker.print();
6. }

结果：

list...  
com.xxx.spring.aop.bean.annotation.BeanImplOne  
com.xxx.spring.aop.bean.annotation.BeanImplTwo  
map...  
beanImplOne--com.xxx.spring.aop.bean.annotation.BeanImplOne  
beanImplTwo--com.xxx.spring.aop.bean.annotation.BeanImplTwo

#### 6.4@Bean注解的使用

@Bean是基于容器的注解，我们可以在使用@Compent注解的地方配合@Bean使用，不过@Bean注解一般不于@Compent注解使用，我们一般使用

@Bean注解配合@Configuration注解使用，相当于我们再xml配置文件中定义了<bean></bean>

使用：

**[java]** [view plain](http://blog.csdn.net/u014292162/article/details/52277756) [copy](http://blog.csdn.net/u014292162/article/details/52277756)

1. @Configuration   //相当于配置文件
2. **public** **class** Appconfig{
4. @Bean("myservice")//假如bean的name属性没有指定名字的话，注入的是id为方法名的bean，一般我们指定name属性不容易出错
5. **public** Myservice myservice(){
6. **return** **new** MyServiceImpl();
7. }
8. /\*
9. 对比基于XML文件中的配置效果类似
10. <bean id="myservice" class="com.xxx.service.MyserviceImpl"></bean>
11. \*/
12. }

@Bean中的其他他几个属性

之前我们再配置文件中使用过如下的配置，指定bean的初始化调时会执行的方法，和销毁会执行的方法

**[html]** [view plain](http://blog.csdn.net/u014292162/article/details/52277756) [copy](http://blog.csdn.net/u014292162/article/details/52277756)

1. **<bean** name="life" class="com.briup.spring.ioc.bean.Life" init-method="myInit" destroy-method="myDestory"**>**
2. **<property** name="name" value="tom"**></property>**
3. **</bean>**

我们使@Bean配置也可以实现上边这种效果

**[java]** [view plain](http://blog.csdn.net/u014292162/article/details/52277756) [copy](http://blog.csdn.net/u014292162/article/details/52277756)

1. **public** **class** Foo{
2. **public** **void** init(){
4. }
5. }
7. **public** **class** Bar{
8. **public** **void** cleanup(){
10. }
11. }

**[java]** [view plain](http://blog.csdn.net/u014292162/article/details/52277756) [copy](http://blog.csdn.net/u014292162/article/details/52277756)

1. @Configuration
2. **public** **class** Appconfig{
4. @Bean(name="life")  //定义bean的name
5. **public** Life life(){
6. **return** **new** Life();
7. }
9. @Bean(initMethod="init") //在初始化Foo的时候，会调用Foo.java中的init方法
10. **public** Foo foo(){
11. **return** **new** Foo();
12. }
14. @Bean(destoryMethod=“cleanup”) //在销毁Bar的时候会调用Bar.java中的cleanup中的方法
15. **public** Bar bar(){
16. **return** **new** Bar();
17. }
18. }

#### 6.5使用注解模拟连接数据库

db.properties内容如下：

jdbc.driver=oracle.jdbc.driver.OracleDriver  
jdbc.url=jdbc:oracle:thin:@localhost:1521:XE  
jdbc.username=caojx  
jdbc.password=caojx

config.xml配置如下

**[java]** [view plain](http://blog.csdn.net/u014292162/article/details/52277756) [copy](http://blog.csdn.net/u014292162/article/details/52277756)

1. <?xml version="1.0" encoding="UTF-8"?>
2. <beans xmlns="http://www.springframework.org/schema/beans"
3. xmlns:context="http://www.springframework.org/schema/context"
4. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5. xsi:schemaLocation="http://www.springframework.org/schema/beans
6. http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
7. http://www.springframework.org/schema/context
8. http://www.springframework.org/schema/context/spring-context.xsd" >
9. <!-- 加载db.properties文件 -->
10. <context:property-placeholder location="classpath:db.properties"/>
11. <!--context:component-scan包含context:annotation-config的全部功能，通常使用前者后，不再使用后者
12. <context:component-scan base-**package**="com.briup.spring.aop.bean.annotation"></context:component-scan>
14. </beans>

使用如下类，打印出配置文件中db.properties中的信息

**[java]** [view plain](http://blog.csdn.net/u014292162/article/details/52277756) [copy](http://blog.csdn.net/u014292162/article/details/52277756)

1. **public** **class** MyDriverManager {
3. **public** MyDriverManager(String url, String userName, String password){
4. System.out.println("url :"+url);
5. System.out.println("userName :"+userName);
6. System.out.println("password :"+password);
7. }
9. }

读取配置文件中的信息

**[java]** [view plain](http://blog.csdn.net/u014292162/article/details/52277756) [copy](http://blog.csdn.net/u014292162/article/details/52277756)

1. @Configuration
2. @ImportResource("classpath:com/xxx/spring/chap4/config.xml")  //指定配置文件的路径
3. **public** **class** MyConnection {

6. @Value("${jdbc.url}")  //基本类型的变量使用@Value注解(括号里边是注入的值)  ，这是使用${是读取配db.properties中的值}
7. **private** String url;
9. @Value("${jdbc.username}")  //如果db.properties中写法为username默认取的是当前操作系统用户的名称，可以在db.properties定义username的时候使用jdbc.username
10. **private** String userName;
12. @Value("${jdbc.password}")
13. **private** String password;
15. @Bean(name="myDriverManager")
17. **public** MyDriverManager MyDriverManager(){
18. **return** **new** MyDriverManager(url,userName,password);
19. }
21. }

测试：  
ClassPathXmlApplicationContext ac = new ClassPathXmlApplicationContext("com/xxx/spring/chap4/annotation.xml");  
System.out.println(ac.getBean("myDriverManager"));  
结果：  
url :jdbc:oracle:thin:@localhost:1521:XE  
userName :caojx  
password :caojx  
com.briup.spring.aop.bean.annotation.MyDriverManager@152b54b

同时：@Bean注解也可以配置@Scope使用

**[java]** [view plain](http://blog.csdn.net/u014292162/article/details/52277756) [copy](http://blog.csdn.net/u014292162/article/details/52277756)

1. @Bean(name="myDriverManager")
2. @Scope("prototype")
3. **public** MyDriverManager MyDriverManager(){
4. **return** **new** MyDriverManager(url,userName,password);
5. }
7. @Bean(name="myDriverManager")
8. @Scope("singleton")
9. **public** MyDriverManager MyDriverManager(){
10. **return** **new** MyDriverManager(url,userName,password);
11. }

提示：spring配置数据库连接，或事务管理这一块，将会专门使用一篇来说明。

#### 6.6Spring对JSR的注解支持

JSR常见的注解有如下

@Resource等效于@Autowired与@Inject  
@PostConstrct  初始化回掉  
@PreDetory   销毁回调用  
@Inject 等效于 @Autowired  
@Named 与 @Compenet等效

##### 6.6.1@Resource

@Resource的作用相当于@Autowired，只不过@Autowired按byType自动注入，  
而@Resource默认按 byName自动注入罢了。  
@Resource有两个属性是比较重要的，分是name和type，  
Spring将@Resource注解的name属性解析为bean的名字，而type属性则解析为bean的类型。  
所以如果使用name属性，则使用byName的自动注入策略，而使用type属性时则使用byType自动注入策略。如果既不指定name也不指定type属性，这时将通过反射机制使用byName自动注入策略。  
　　@Resource装配顺序  
　　1. 如果同时指定了name和type，则从Spring上下文中找到唯一匹配的bean进行装配，找不到则抛出异常  
　　2. 如果指定了name，则从上下文中查找名称（id）匹配的bean进行装配，找不到则抛出异常  
　　3. 如果指定了type，则从上下文中找到类型匹配的唯一bean进行装配，找不到或者找到多个，都会抛出异常  
　　4. 如果既没有指定name，又没有指定type，则自动按照byName方式进行装配；如果没有匹配，则回退为一个原始类型进行匹配，如果匹配则自动装配；   
       5. 如果 @Resource用于方法中，默认使用方法名作为beanName,指定名字则使用名字

案例:

DAO

**[java]** [view plain](http://blog.csdn.net/u014292162/article/details/52277756) [copy](http://blog.csdn.net/u014292162/article/details/52277756)

1. **import** org.springframework.stereotype.Repository;
3. @Repository
4. **public** **class** JsrDAO {
6. **public** **void** save(){
7. System.out.println("JsrDao invoker");
8. }
10. }

Service

**[java]** [view plain](http://blog.csdn.net/u014292162/article/details/52277756) [copy](http://blog.csdn.net/u014292162/article/details/52277756)

1. **import** javax.annotation.PostConstruct;
2. **import** javax.annotation.PreDestroy;
3. **import** javax.annotation.Resource;
5. **import** org.springframework.stereotype.Service;
7. **import** com.briup.spring.aop.bean.annotation.dao.JsrDAO;
9. @Service
10. **public** **class** JsrService {
12. @Resource
13. **private** JsrDAO jsrDAO;
15. @Resource  //作用与上边一样,二选一都可以
16. **public** **void** setJsrDAO(JsrDAO jsrDAO){
17. **this**.jsrDAO = jsrDAO;
18. }
20. **public** **void** save(){
21. jsrDAO.save();
22. }
24. @PostConstruct
25. **public** **void** init(){
26. System.out.println("jsr Service init");
27. }
29. @PreDestroy
30. **public** **void** destory(){
31. System.out.println("jsr Service destory");
32. }
34. }

提示：

@Resource的处理是由ApplicationContext中的CommonAnnotationBeanPostProecssor发现并处理的  
CommonAnnotationBeanPostProecssor不仅支持 @Resource注解，还支持 @PostConstruct初始回调  
和 @PreDestory销毁回调，前提是CommonAnnotationBeanPostProecssor是在ApplicationContext中注册的

测试结果：

@Test  
public void testJsr(){  
ClassPathXmlApplicationContext ac = new ClassPathXmlApplicationContext("com/briup/spring/chap4/annotation.xml");  
System.out.println(ac.getBean("jsrService"));  
ac.destroy();  
}

结果：

jsr Service init  
com.briup.spring.aop.bean.annotation.service.JsrService@7dc4cb  
jsr Service destory

@Resource是一个比比较常用的JSR注解，对于JSR中的其他注解，这里不进行详细的介绍。