[Spring 注解总结](http://blog.csdn.net/wangshfa/article/details/9712379)

2013-08-02 08:59 11426人阅读 [评论](http://blog.csdn.net/wangshfa/article/details/9712379#comments)(0) [收藏](javascript:void(0);) [举报](http://blog.csdn.net/wangshfa/article/details/9712379#report)

http://static.blog.csdn.net/images/category_icon.jpg 分类：

ssh（42） http://static.blog.csdn.net/images/arrow_triangle%20_down.jpg

目录[(?)](http://blog.csdn.net/wangshfa/article/details/9712379" \o "系统根据文章中H1到H6标签自动生成文章目录)[[+]](http://blog.csdn.net/wangshfa/article/details/9712379)

**声明：这是转载的。  
内容根据网上资料整理。**  
相关链接：  
http://www.360doc.com/content/10/1118/16/2371584\_70449913.shtml  
http://www.iteye.com/topic/1121784

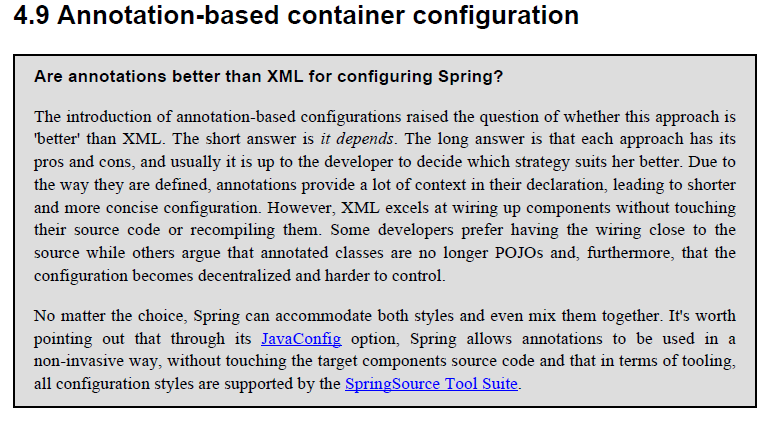
http://www.iteye.com/topic/295348

【正文】

**一 注解优点？注解解决了什么问题，为什么要使用注解？**

**二 注解的来龙去脉（历史）**

Spring 3.1 Reference 对注解与XML对比的一段叙述：



1 没有注解之前

**[java]** [view plaincopy](http://blog.csdn.net/bruce_sky/article/details/7645040)

1. **public** **class** UserManagerImpl **implements** UserManager {
2. **private** UserDao userDao;
3. **public** **void** setUserDao(UserDao userDao) {
4. **this**.userDao = userDao;
5. }
6. ...
7. }

配置文件

**[html]** [view plaincopy](http://blog.csdn.net/bruce_sky/article/details/7645040)

1. **<bean** id="userManagerImpl" class="com.kedacom.spring.annotation.service.UserManagerImpl"**>**
2. **<property** name="userDao" ref="userDao" **/>**
3. **</bean>**
4. **<bean** id="userDao" class="com.kedacom.spring.annotation.persistence.UserDaoImpl"**>**
5. **<property** name="sessionFactory" ref="mySessionFactory" **/>**
6. **</bean>**

2 注解诞生之后

**[java]** [view plaincopy](http://blog.csdn.net/bruce_sky/article/details/7645040)

1. **public** **class** UserManagerImpl **implements** UserManager {
2. @Autowired
3. **private** UserDao userDao;
4. ...
5. }

或者（对方法进行标注）

**[java]** [view plaincopy](http://blog.csdn.net/bruce_sky/article/details/7645040)

1. **public** **class** UserManagerImpl **implements** UserManager {
2. **private** UserDao userDao;
3. @Autowired
4. **public** **void** setUserDao(UserDao userDao) {
5. **this**.userDao = userDao;
6. }
7. ...
8. }

配置文件

**[html]** [view plaincopy](http://blog.csdn.net/bruce_sky/article/details/7645040)

1. **<bean** id="userManagerImpl" class="com.kedacom.spring.annotation.service.UserManagerImpl" **/>**
2. **<bean** id="userDao" class="com.kedacom.spring.annotation.persistence.UserDaoImpl"**>**
3. **<property** name="sessionFactory" ref="mySessionFactory" **/>**
4. **</bean>**

@Autowired可以对成员变量、方法和构造函数进行标注，来完成自动装配的工作。以上两种不同实现方式中，@Autowired的标注位置不同，它们都会在Spring在初始化userManagerImpl这个bean时，自动装配userDao这个属性，区别是：第一种实现中，Spring会直接将UserDao类型的唯一一个bean赋值给userDao这个成员变量；第二种实现中，Spring会调用setUserDao方法来将UserDao类型的唯一一个bean装配到userDao这个属性。

要使@Autowired能够工作，还需要在配置文件中加入以下代码

**[html]** [view plaincopy](http://blog.csdn.net/bruce_sky/article/details/7645040)

1. **<bean** class="org.springframework.beans.factory.annotation.AutowiredAnnotationBeanPostProcessor" **/>**

可以看到确实是减少了代码和配置文件中的配置。

**三 注解总体介绍**

注解实现Bean配置主要用来进行如依赖注入、生命周期回调方法定义等，不能消除XML文件中的Bean元数据定义，且基于XML配置中的依赖注入的数据将覆盖基于注解配置中的依赖注入的数据。  
Spring3的基于注解实现Bean依赖注入支持如下三种注解：  
Spring自带依赖注入注解： Spring自带的一套依赖注入注解；  
JSR-250注解：Java平台的公共注解，是Java EE 5规范之一，在JDK6中默认包含这些注解，从Spring2.5开始支持。  
JSR-330注解：Java 依赖注入标准，Java EE 6规范之一，可能在加入到未来JDK版本，从Spring3开始支持；  
JPA注解：用于注入持久化上下文和尸体管理器。  
这三种类型的注解在Spring3中都支持，类似于注解事务支持，想要使用这些注解需要在Spring容器中开启注解驱动支持，即使用如下配置方式开启：

**[html]** [view plaincopy](http://blog.csdn.net/bruce_sky/article/details/7645040)

1. **<beans** xmlns="http://www.springframework.org/schema/beans"
2. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3. xmlns:context="http://www.springframework.org/schema/context"
4. xsi:schemaLocation=" http://www.springframework.org/schema/beans
5. http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
6. http://www.springframework.org/schema/context
7. http://www.springframework.org/schema/context/spring-context-3.0.xsd"**>**
9. **<context:annotation-config/>**
11. **</beans>**

这样就能使用注解驱动依赖注入了  
**Spring自带依赖注入注解**  
  
1 @Required：依赖检查；  
2 @Autowired：自动装配2   
自动装配，用于替代基于XML配置的自动装配  
基于@Autowired的自动装配，默认是根据类型注入，可以用于构造器、字段、方法注入  
3 @Value：注入SpEL表达式  
用于注入SpEL表达式，可以放置在字段方法或参数上  
@Value(value = "SpEL表达式")    
@Value(value = "#{message}")    
4 @Qualifier：限定描述符，用于细粒度选择候选者  
@Qualifier限定描述符除了能根据名字进行注入，但能进行更细粒度的控制如何选择候选者  
@Qualifier(value = "限定标识符")

(

@Autowired是根据类型进行自动装配的。如果当Spring上下文中存在不止一个UserDao类型的bean时，就会抛出BeanCreationException异常;如果Spring上下文中不存在UserDao类型的bean，也会抛出BeanCreationException异常。我们可以使用@Qualifier配合@Autowired来解决这些问题。如下：

①可能存在多个UserDao实例

**[java]** [view plain](http://blog.csdn.net/clerk0324/article/details/25198457) [copy](http://blog.csdn.net/clerk0324/article/details/25198457)

[在CODE上查看代码片](https://code.csdn.net/snippets/331452)

1. @Autowired
2. @Qualifier("userServiceImpl")
3. **public** IUserService userService;

或者

**[java]** [view plain](http://blog.csdn.net/clerk0324/article/details/25198457) [copy](http://blog.csdn.net/clerk0324/article/details/25198457)

[在CODE上查看代码片](https://code.csdn.net/snippets/331452)

1. @Autowired
2. **public** **void** setUserDao(@Qualifier("userDao") UserDao userDao) {
3. **this**.userDao = userDao;
4. }

这样Spring会找到id为userServiceImpl和userDao的bean进行装配。

②可能不存在UserDao实例

**[java]** [view plain](http://blog.csdn.net/clerk0324/article/details/25198457) [copy](http://blog.csdn.net/clerk0324/article/details/25198457)

[在CODE上查看代码片](https://code.csdn.net/snippets/331452)

1. @Autowired(required = **false**)
2. **public** IUserService userService

)  
  
  
**JSR-250注解**  
  
1 @Resource：自动装配，默认根据类型装配，如果指定name属性将根据名字装配，可以使用如下方式来指定  
@Resource(name = "标识符")    
字段或setter方法

2 @PostConstruct和PreDestroy：通过注解指定初始化和销毁方法定义

**JSR-330注解**  
1 @Inject：等价于默认的@Autowired，只是没有required属性  
2 @Named：指定Bean名字，对应于Spring自带@Qualifier中的缺省的根据Bean名字注入情况  
3 @Qualifier：只对应于Spring自带@Qualifier中的扩展@Qualifier限定描述符注解，即只能扩展使用，没有value属性  
  
  
**JPA注解**  
@PersistenceContext  
@PersistenceUnit

用于注入EntityManagerFactory和EntityManager

**四 注解需要的jar包及配置**



**五 拓展：结合局部代码来说明注解**

**1 @Autowired**

同二 注解历史中的内容，此处不重复。

**2 @Qualifier**

@Autowired是根据类型进行自动装配的。在上面的例子中，如果当Spring上下文中存在不止一个UserDao类型的bean时，就会抛出BeanCreationException异常；如果Spring上下文中不存在UserDao类型的bean，也会抛出BeanCreationException异常。我们可以使用@Qualifier配合@Autowired来解决这些问题。

1>可能存在多个UserDao实例

**[java]** [view plaincopy](http://blog.csdn.net/bruce_sky/article/details/7645040)

1. @Autowired
2. **public** **void** setUserDao(@Qualifier("userDao") UserDao userDao) {
3. **this**.userDao = userDao;
4. }

这样，Spring会找到id为userDao的bean进行装配。   
2>可能不存在UserDao实例

**[java]** [view plaincopy](http://blog.csdn.net/bruce_sky/article/details/7645040)

1. @Autowired(required = **false**)
2. **public** **void** setUserDao(UserDao userDao) {
3. **this**.userDao = userDao;
4. }

**3 @Resource（JSR-250标准注解，推荐使用它来代替Spring专有的@Autowired注解）**  
Spring 不但支持自己定义的@Autowired注解，还支持几个由JSR-250规范定义的注解，它们分别是@Resource、@PostConstruct以及@PreDestroy。   
@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，分别是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   
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  
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  
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  
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（见2）；如果没有匹配，则回退为一个原始类型（UserDao）进行匹配，如果匹配则自动装配；

**4 @PostConstruct（JSR-250）**   
在方法上加上注解@PostConstruct，这个方法就会在Bean初始化之后被Spring容器执行（注：Bean初始化包括，实例化Bean，并装配Bean的属性（依赖注入））。   
它的一个典型的应用场景是，当你需要往Bean里注入一个其父类中定义的属性，而你又无法复写父类的属性或属性的setter方法时，如：

**[java]** [view plaincopy](http://blog.csdn.net/bruce_sky/article/details/7645040)

1. **public** **class** UserDaoImpl **extends** HibernateDaoSupport **implements** UserDao {
2. **private** SessionFactory mySessionFacotry;
3. @Resource
4. **public** **void** setMySessionFacotry(SessionFactory sessionFacotry) {
5. **this**.mySessionFacotry = sessionFacotry;
6. }
7. @PostConstruct
8. **public** **void** injectSessionFactory() {
9. **super**.setSessionFactory(mySessionFacotry);
10. }
11. ...
12. }

这里通过@PostConstruct，为UserDaoImpl的父类里定义的一个sessionFactory私有属性，注入了我们自己定义的sessionFactory（父类的setSessionFactory方法为final，不可复写），之后我们就可以通过调用super.getSessionFactory()来访问该属性了。

**5 @PreDestroy（JSR-250）**  
在方法上加上注解@PreDestroy，这个方法就会在Bean初始化之后被Spring容器执行。由于我们当前还没有需要用到它的场景，这里不不去演示。其用法同@PostConstruct。

**6 使用<context:annotation-config />简化配置**  
Spring2.1添加了一个新的context的Schema命名空间，该命名空间对注释驱动、属性文件引入、加载期织入等功能提供了便捷的配置。我们知道注释本身是不会做任何事情的，它仅提供元数据信息。要使元数据信息真正起作用，必须让负责处理这些元数据的处理器工作起来。

AutowiredAnnotationBeanPostProcessor和CommonAnnotationBeanPostProcessor就是处理这些注释元数据的处理器。但是直接在Spring配置文件中定义这些Bean显得比较笨拙。Spring为我们提供了一种方便的注册这些BeanPostProcessor的方式，这就是<context:annotation-config />：

**[html]** [view plaincopy](http://blog.csdn.net/bruce_sky/article/details/7645040)

1. **<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"
2. xsi:schemaLocation="http://www.springframework.org/schema/beans
3. http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
4. http://www.springframework.org/schema/context
5. http://www.springframework.org/schema/context/spring-context-2.5.xsd"**>**
6. **<context:annotation-config** **/>**
7. **</beans>**

<context:annotationconfig />将隐式地向Spring容器注册AutowiredAnnotationBeanPostProcessor、CommonAnnotationBeanPostProcessor、 PersistenceAnnotationBeanPostProcessor以及RequiredAnnotationBeanPostProcessor这4个BeanPostProcessor。

使用Spring注解完成Bean的定义   
以上我们介绍了通过@Autowired或@Resource来实现在Bean中自动注入的功能，下面我们将介绍如何注解Bean，从而从XML配置文件中完全移除Bean定义的配置。

**7  @Component（不推荐使用）、@Repository、@Service、@Controller**  
只需要在对应的类上加上一个@Component注解，就将该类定义为一个Bean了：

**[java]** [view plaincopy](http://blog.csdn.net/bruce_sky/article/details/7645040)

1. @Component
2. **public** **class** UserDaoImpl **extends** HibernateDaoSupport **implements** UserDao {
3. ...
4. }

使用@Component注解定义的Bean，默认的名称（id）是小写开头的非限定类名。如这里定义的Bean名称就是userDaoImpl。你也可以指定Bean的名称：   
@Component("userDao")   
@Component是所有受Spring管理组件的通用形式，Spring还提供了更加细化的注解形式：@Repository、@Service、@Controller，它们分别对应存储层Bean，业务层Bean，和展示层Bean。目前版本（2.5）中，这些注解与@Component的语义是一样的，完全通用，在Spring以后的版本中可能会给它们追加更多的语义。所以，我们推荐使用@Repository、@Service、@Controller来替代@Component。

**8 使用<context:component-scan />让Bean定义注解工作起来**

**[html]** [view plaincopy](http://blog.csdn.net/bruce_sky/article/details/7645040)

1. **<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"
2. xsi:schemaLocation="http://www.springframework.org/schema/beans
3. http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
4. http://www.springframework.org/schema/context
5. http://www.springframework.org/schema/context/spring-context-2.5.xsd"**>**
6. **<context:component-scan** base-package="com.kedacom.ksoa" **/>**
7. **</beans>**

这里，所有通过<bean>元素定义Bean的配置内容已经被移除，仅需要添加一行<context:component-scan />配置就解决所有问题了——Spring XML配置文件得到了极致的简化（当然配置元数据还是需要的，只不过以注释形式存在罢了）。<context:component-scan />的base-package属性指定了需要扫描的类包，类包及其递归子包中所有的类都会被处理。   
<context:component-scan />还允许定义过滤器将基包下的某些类纳入或排除。Spring支持以下4种类型的过滤方式：   
过滤器类型 表达式范例   说明  
注解 org.example.SomeAnnotation将所有使用SomeAnnotation注解的类过滤出来  
类名指定 org.example.SomeClass过滤指定的类  
正则表达式 com\.kedacom\.spring\.annotation\.web\..\*通过正则表达式过滤一些类  
AspectJ表达式 org.example..\*Service+通过AspectJ表达式过滤一些类  
  
  
以正则表达式为例，我列举一个应用实例：

**[html]** [view plaincopy](http://blog.csdn.net/bruce_sky/article/details/7645040)

1. **<context:component-scan** base-package="com.casheen.spring.annotation"**>**
2. **<context:exclude-filter** type="regex" expression="com\.casheen\.spring\.annotation\.web\..\*" **/>**
3. **</context:component-scan>**

值得注意的是<context:component-scan />配置项不但启用了对类包进行扫描以实施注释驱动Bean定义的功能，同时还启用了注释驱动自动注入的功能（即还隐式地在内部注册了AutowiredAnnotationBeanPostProcessor和CommonAnnotationBeanPostProcessor），因此当使用<context:component-scan />后，就可以将<context:annotation-config />移除了.

**9 使用@Scope来定义Bean的作用范围**

在使用XML定义Bean时，我们可能还需要通过bean的scope属性来定义一个Bean的作用范围，我们同样可以通过@Scope注解来完成这项工作

**[java]** [view plaincopy](http://blog.csdn.net/bruce_sky/article/details/7645040)

1. @Scope("session")
2. @Component()
3. **public** **class** UserSessionBean **implements** Serializable {
4. ...
5. }

**六 一个完整的例子**

为了加深印象，又增加了一个完整的例子。

例子来源

<http://blog.csdn.net/pk490525/article/details/8096326>

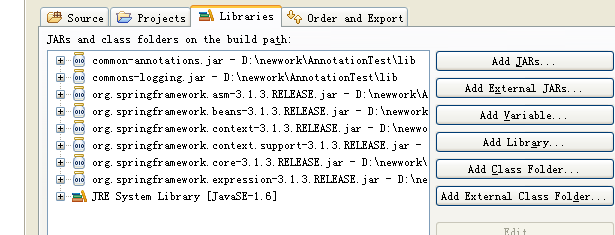
spring零配置（Annotation）学习笔记

本地例子：

AnnotationTest

本地有细小的改变

project用到的jar包



先上bean-config.xml

**[html]** [view plaincopy](http://blog.csdn.net/bruce_sky/article/details/7645040)

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:context="http://www.springframework.org/schema/context"
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-3.0.xsd"**>**
10. **<context:annotation-config/>**
11. <!-- action暂未用注解 -->
12. **<bean** id="myAction" class="com.demo.annotation.action.MyAction" scope="prototype" **/>**
14. <!-- 注解测试   -->
15. **<context:component-scan** base-package="com.demo.annotation" **/>**

18. **</beans>**

service 接口

**[java]** [view plaincopy](http://blog.csdn.net/bruce_sky/article/details/7645040)

1. /\*\*
2. \*
3. \* Annotation
4. \*
5. \*/
7. **public** **interface** TestService {
8. /\*\*
9. \* 注解测试
10. \* @return
11. \*/
12. **public** String getTestAnnotation();
13. }

service实现类

**[java]** [view plaincopy](http://blog.csdn.net/bruce_sky/article/details/7645040)

1. **import** org.springframework.stereotype.Service;
3. **import** com.demo.annotation.dao.TestDao;
4. **import** org.springframework.beans.factory.annotation.Autowired;
5. **import** org.springframework.beans.factory.annotation.Qualifier;


9. /\*\*
10. \*
11. \* 注解测试
12. \*
13. \*/
15. @Service("testService")
16. **public** **class** TestServiceImp **implements** TestService {
17. /\*\*
18. \* 自动装配
19. \*/
20. @Autowired
21. @Qualifier("testDao")
22. //@Resource(name="testDao"), 等价于<property ………… ref="testDao" />
23. **private** TestDao testDao;
25. **public** TestDao getTestDao() {
26. **return** testDao;
27. }
29. **public** **void** setTestDao(TestDao testDao) {
30. **this**.testDao = testDao;
31. }

34. @Override
35. **public** String getTestAnnotation() {
36. // TODO Auto-generated method stub
37. **return** testDao.getTestDaoAnnotation();
38. }
40. }

dao接口

**[java]** [view plaincopy](http://blog.csdn.net/bruce_sky/article/details/7645040)

1. /\*\*
2. \* 测试注解
3. \*
4. \*/
5. **public** **interface** TestDao {
6. /\*\*
7. \* 得到dao层注解
8. \* @return
9. \*/
10. **public** String getTestDaoAnnotation();
11. }

dao层实现类

**[java]** [view plaincopy](http://blog.csdn.net/bruce_sky/article/details/7645040)

1. **import** org.springframework.stereotype.Repository;
3. /\*\*
4. \* 测试Annotation
5. \*
6. \*/
8. @Repository("testDao")
9. **public** **class** TestDaoImpl **implements** TestDao {
11. @Override
12. **public** String getTestDaoAnnotation() {
13. // TODO Auto-generated method stub
14. **return** "This is testDao Annotation...";
15. }
17. }

Action类

**[java]** [view plaincopy](http://blog.csdn.net/bruce_sky/article/details/7645040)

1. **import** javax.annotation.Resource;
3. **import** com.demo.annotation.service.TestService;
5. **public** **class** MyAction{
6. @Resource(name="testService")
7. **private** TestService testService;
8. **public** String testAnnotation(){
9. **if**(**null** == testService){
10. System.out.println("Service is null!!");
11. }
12. String result = testService.getTestAnnotation();
13. System.out.println(result);
14. **return** "success";
15. }
17. **public** TestService getTestService() {
18. **return** testService;
19. }
21. **public** **void** setTestService(TestService testService) {
22. **this**.testService = testService;
23. }
25. }

测试类

**[java]** [view plaincopy](http://blog.csdn.net/bruce_sky/article/details/7645040)

1. **import** org.springframework.context.ApplicationContext;
2. **import** org.springframework.context.support.ClassPathXmlApplicationContext;
4. **import** com.demo.annotation.action.MyAction;
6. **public** **class** TestAnnotation {
7. **public** **static** **void** main(String[] args) {
8. ApplicationContext context = **new** ClassPathXmlApplicationContext("bean-config.xml");
9. MyAction action = (MyAction)context.getBean("myAction");
10. action.testAnnotation();
12. }
13. }

**七 总结**

Annotation的好处：

简化了xml文件

坏处：

1 spring一直宣称是无侵入的，annotation是否是侵入了？

2 很多类都是pojo的对象，引入annotation后还是pojo么？

3 增加了复杂度

**spring官方的态度是两种让你混合着用。**

最后：

**推荐**

http://snowolf.iteye.com/blog/577989   
snowolf的Spring 注解学习手札 有代码有文字说明可运行可测试