1. 使用Spring注解来注入属性   
1.1. 使用注解以前我们是怎样注入属性的   
类的实现：

**Java代码  [收藏代码](javascript:void())**

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：

**Java代码  [收藏代码](javascript:void())**

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>

1.2. 引入@Autowired注解（不推荐使用，建议使用@Resource）   
类的实现（对成员变量进行标注）

**Java代码  [收藏代码](javascript:void())**

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

或者（对方法进行标注）

**Java代码  [收藏代码](javascript:void())**

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

**Java代码  [收藏代码](javascript:void())**

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。   
  
1.3. 让@Autowired工作起来   
要使@Autowired能够工作，还需要在配置文件中加入以下代码

**Java代码  [收藏代码](javascript:void())**

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

1.4. @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代码  [收藏代码](javascript:void())**

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代码  [收藏代码](javascript:void())**

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

1.5. @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 

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（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；

1.6. @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代码  [收藏代码](javascript:void())**

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。   
  
1.7. @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。   
  
1.8. 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 />：

**Java代码  [收藏代码](javascript:void())**

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。   
  
2. 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。   
  
2.1. @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代码  [收藏代码](javascript:void())**

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。   
  
2.2. 使用<context:component-scan />让Bean定义注解工作起来

**Java代码  [收藏代码](javascript:void())**

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：

**Java代码  [收藏代码](javascript:void())**

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。   
  
2.3. 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代码  [收藏代码](javascript:void())**

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

<bean id=*"mySessionFactory"* class=*"org.springframework.orm.hibernate3.annotation.AnnotationSessionFactoryBean"*>

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

<property name=*"packagesToScan"* value=*"com.ssh.po"*> </property>

<property name=*"hibernateProperties"*>

<props>

<prop key=*"hibernate.dialect"*>org.hibernate.dialect.MySQL5Dialect</prop>

<prop key=*"hibernate.show\_sql"*>true</prop>

<prop key=*"hibernate.format\_sql"*>true</prop>

<prop key=*"hibernate.hbm2ddl.auto"*>update</prop>

<prop

key=*"hibernate.current\_session\_context\_class"*>thread</prop>

</props>

</property>

</bean>

<bean id=*"txManager"* class=*"org.springframework.orm.hibernate3.HibernateTransactionManager"*>

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

</bean>

1. 配置数据连接池

<bean id=*"dataSource"* class=*"org.apache.commons.dbcp.BasicDataSource"* destroy-method=*"close"*>

<property name=*"driverClassName"* value=*"com.mysql.jdbc.Driver"* />

<property name=*"url"* value=*"jdbc:mysql://localhost:3306/test"* />

<property name=*"username"* value=*"root"* />

<property name=*"password"* value=*"root"* /> </bean>

1. 测试：

**import** java.sql.Connection;

**import** java.sql.SQLException;

**import** org.apache.commons.dbcp.BasicDataSource;

**import** org.springframework.context.ApplicationContext;

**import** org.springframework.context.support.ClassPathXmlApplicationContext;

**public** **class** Test {

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

// **TODO** Auto-generated method stub

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

BasicDataSource bds = ac.getBean("dataSource",BasicDataSource.**class**);

Connection conn = **null**;

**try** {

conn = bds.getConnection();

conn.createStatement().executeUpdate("insert into t\_user1 values(null,'admin','12345')");

} **catch** (SQLException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

}

}

2.

**public** **void** addUser(User user) {

// **TODO** Auto-generated method stub

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

SessionFactory sessionFactory = ac.getBean("mySessionFactory",SessionFactory.**class**);

Session session = sessionFactory.openSession();

Transaction ts = session.beginTransaction();

**try**{

Query query = session.createQuery("from User where username=?");

query.setParameter(0, "admin");

User user1 = (User) query.uniqueResult();

System.*out*.println(user1.getPassword());

}**catch** (Exception e) {

// **TODO**: handle exception

ts.rollback();

}

}