# 2.spring注解

## 2.1.构造方法注入

构造方法的作用：

1.创建对象

2.为属性赋值

步骤：首先，创建User类，加入无参和有参构造，和toString

|  |
| --- |
| public class User {  private Integer id;  private String name; |

编写配置文件

|  |
| --- |
| <!-- 通过构造方法为属性赋值  index="" :参数的下标（从0开始）  name="" :参数的名字  value="" :参数的值  ref="" : 参数的引用  type="": 参数的类型（可以省略）  -->  <bean id="user" class="pojo.User">  <constructor-arg index="0" name="id" value="11"></constructor-arg>  <constructor-arg index="1" name="name" value="姚明"></constructor-arg>  </bean> |

测试

也可以为引用类型赋值，创建Dog类，User中加入dog属性，配置文件

|  |
| --- |
| <constructor-arg name="dog" ref="dog"></constructor-arg>  </bean>  <bean id="dog" class="pojo.Dog"></bean> |

说明：1.配置文件中的参数必须与构造方法中的参数一一对应。

2.index和name属性都可以唯一定位构造方法中的参数（建议用index）

注：在set注入和构造注入中，set注入更灵活，我们一般开发的时候用set注入，但是spring的源码中使用的构造注入多一些。

## 2.2.自动装配

在spring中，可以将对象的应用类型自动的赋值，必须要依赖set方法

步骤：首先，在User中加入引用类型，和get、set方法

|  |
| --- |
| public class User {  private Dog dog;  private Cat cat; |

配置文件

|  |
| --- |
| <!-- 自动装配  autowire="byName":根据名称与id匹配  autowire="byType":根据引用类型匹配  -->  <bean id="user" class="pojo.User" autowire="byType"></bean>    <bean id="dog" class="pojo.Dog"></bean>  <bean id="cat" class="pojo.Cat"></bean> |

最后测试，发现，如果byname，若改变bean的id，则无法注入，而bytype中，如果改变bean中的id，则可以注入。

自动注入原理：

autowire="byName"：首先会解析类中的所有set方法，经过编写，将所有set去掉，首字母小写，通过该名称，匹配所有bean中的id，如果匹配上，则注入，如果没有匹配上，则为null。

autowire="byType"：首先会解析类中的所有set方法，寻找参数类型，根据参数类型匹配配置文件中的class，如果匹配成功，则注入，否则为null。

我们依然可以进行全局配置（建议使用byname，否则如果同一个类配置两个不同id的bean，依然会报错。）

|  |
| --- |
| http://www.springframework.org/schema/beans/spring-beans-3.2.xsd "  default-autowire="byName"> |

## 2.3.spring中的属性注解

首先，要让spring容器知道我们要用注解了

在头标签中告诉spring

3.1步骤：

3.1.1.导入头文件（复制第一行，并添加context，修改beans为context，复制最后两行，beans改为context，总共修改五处。）

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <beans xmlns="http://www.springframework.org/schema/beans"  xmlns:context="http://www.springframework.org/schema/context"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xsi:schemaLocation="http://www.springframework.org/schema/beans  http://www.springframework.org/schema/beans/spring-beans-3.2.xsd  http://www.springframework.org/schema/context  http://www.springframework.org/schema/context/spring-context-3.2.xsd "> |

3.1.2.导入context约束，并将头文件添加到模板（昨天已经讲的很清楚了，大家自己导，忘了的可以自己看看昨天的笔记）

3.1.3.创建User、Dog、Cat类，在User类中写上Dog和Cat引用

3.1.4.编写配置文件

|  |
| --- |
| <!-- 开启属性注入 -->  <context:annotation-config/>    <bean id="user" class="pojo.User"></bean>  <bean id="dog" class="pojo.Dog"></bean>  <bean id="cat" class="pojo.Cat"></bean> |

3.1.5.在User属性上写@Autowired注解，加入toString

|  |
| --- |
| public class User {  /\*  \* 根据自动装配注入，有两种匹配规则：  \* 1.byName  \* 2.byType  \*/  @Autowired  private Dog dog;  @Autowired  private Cat cat; |

3.1.6.测试

3.2.原理：spring启动的时候，首先扫描到属性注入的标签，然后扫描到bean，当创建bean的时候发现属性上有@Autowired注解，则根据name找bean中的id，如果匹配上，则注入，如果没有匹配，则再根据type匹配，如果匹配上则注入，如果没有匹配上，直接报错。

3.3.指定ID进行注入（比如我们有两个不同ID相同类型的bean，则可以通过指定ID进行注入）

|  |
| --- |
| //指定id进行注入  @Autowired  @Qualifier("dogA")  private Dog dog; |

通过指定id进行注入，则程序只会按照ID 进行注入。（可以注掉dogB，将dogA变dog进行试验）

3.4.[增强型@Resource](mailto:3.4.增强型@Resource)注解

注意：以后开发中用@Autowired，如果是单个项目，则用谁都可以，如果是聚合项目，一个大项目拆分成几个项目共同完成，这时候@resource不起作用。

作用：相当于@Autowired+@Qualifier

## 2.4.spring中类的注解

步骤：开启包扫描（此时，可以将之前的开启注解注掉，一行搞定配置）

|  |
| --- |
| <!-- 开启包扫描 -->  <context:component-scan base-package="pojo"/> |

在需要管理的类上加上@Component注解

|  |
| --- |
| @Component  public class User { |

注解的使用原理：spring容器启动的时候，加载配置文件，解析包扫描中指定的包pojo，之后会找到所有含有注解@Component的类，为其创建对象，然后存入Map中，map中的key就是类名首字母小写，value为创建的对象。

创建对象的时候，如果发现有属性注解，则按照属性注解的注入方式进行注入。至此，整个对象创建完成。

为了验证对象的ID就是类名首字母小写，我们可以来做个例子：

实现接口

|  |
| --- |
| @Component  public class User implements BeanNameAware{ |

重写方法

|  |
| --- |
| @Override  public void setBeanName(String name) {  System.out.println("User对象的ID为："+name);  } |

测试

这个接口也在12个生命周期中。

## 2.5.注解的ID生成策略

新建一个NBA类，实现BeanNameAware接口，重写方法。（这个方法是在生成ID的时候就会执行的）

|  |
| --- |
| @Component  public class NBA implements BeanNameAware{  @Override  public void setBeanName(String arg0) {  System.out.println("User对象的ID为："+name);  }  } |

然后正常测试

类名：NBA ID：NBA

类名：NBa ID：NBa

类名：Nba ID：nba

类名：NbA ID：nbA

总结：spring中的ID生成策略：如果第二个字母为大写，则首字母不变，如果第二个字母为小写，则首字母变小写。

## 2.6.指定类的ID

|  |
| --- |
| @Component("god")  public class User{ |

## 2.7.使用注解为属性赋值

7.1.为简单类型赋值（无需get和set，只需在需要赋值的属性上加上@Value）

|  |
| --- |
| public class User{  @Value("2")  private Integer id;  @Value("宋江")  private String name; |

但是这样就把值写死了，很不灵活。

7.2. 引入外部配置文件动态赋值

创建配置文件hero.properties，并添加数据

|  |
| --- |
| id=3  name=\u5173\u80DC |

在这里，配置文件不支持中文，所以当我们写中文的时候，它会自动转码中文为unicode编码(\u开头)

在spring中引入配置文件

|  |
| --- |
| <!-- 引入外部配置文件  location:配置文件的路径，classpath:表示根目录，配置文件的根目录在src下-->  <context:property-placeholder location="hero.properties"/> |

在属性上根据key引入值

|  |
| --- |
| public class User{  //@Value("2")  @Value("${id}")  private Integer id;  //@Value("宋江")  @Value("${name}")  private String name; |

这样配置后，为属性赋值就变得灵活了，如果想要改属性，只需要改配置文件就可以了。

那么配置文件中一般写哪些东西呢？

一般连接数据库的时候，一般用户名密码等信息都是些在配置文件中的。

7.3. 为复杂类型赋值（实际开发中用的很少，你们就当了解即可。）

我们需要借助一套工具，导入头标签util，我修改一下头，这里我们主要练习一下如何导入头标签，给大家5分钟时间，自己修改一下。

导入util头

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <beans xmlns="http://www.springframework.org/schema/beans"  xmlns:context="http://www.springframework.org/schema/context"  xmlns:util="http://www.springframework.org/schema/util"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xsi:schemaLocation="http://www.springframework.org/schema/beans  http://www.springframework.org/schema/beans/spring-beans-3.2.xsd  http://www.springframework.org/schema/context  http://www.springframework.org/schema/context/spring-context-3.2.xsd  http://www.springframework.org/schema/util  http://www.springframework.org/schema/util/spring-util-3.2.xsd "> |

在配置文件中加入list集合

|  |
| --- |
| <!-- 通过工具标签，为list赋值 -->  <util:list id="list">  <value>唐僧</value>  <value>孙悟空</value>  <value>猪八戒</value>  <value>沙和尚</value>  </util:list> |

在属性上引入list

|  |
| --- |
| @Value("#{@list}")  private List list; |

测试

如何为数组赋值？

其实list的底层就是ArrayList，所以数组和list一样，可以用list的方式赋值，我们直接拿过来用

|  |
| --- |
| @Value("#{@list}")  private String[] str; |

发现也能注入。

Set和List类似，可以简单演示一下，会去重

map注入和list类似，配置文件：

|  |
| --- |
| <!-- 通过工具标签，为map赋值 -->  <util:map id="map">  <entry key="一" value="师傅"></entry>  <entry key="二" value="大师兄"></entry>  <entry key="三" value="二师兄"></entry>  <entry key="四" value="沙师弟"></entry>  </util:map> |

属性引入

|  |
| --- |
| @Value("#{@map}")  private Map map; |

Properties注入

|  |
| --- |
| <util:properties id="pro">  <prop key="a">白骨精</prop>  <prop key="b">黑熊精</prop>  <prop key="c">女儿国</prop>  </util:properties> |

属性引入

|  |
| --- |
| @Value("#{@pro}")  private Properties pro; |

## 2.8.spring中的其他注解。

[8.1.@Scope(“prototype”)](mailto:8.1.@Scope(“prototype”))单例和多例

之前在bean标签中加入Scope属性，但是现在没bean了，如何控制单例多例呢？

只需要在类名上加入@Scope属性即可。

|  |
| --- |
| @Component  @Scope("prototype")  public class Person { |

测试

|  |
| --- |
| //测试多例  @Test  public void test02(){  ApplicationContext context =  new ClassPathXmlApplicationContext("applicationContext.xml");  Person p = (Person) context.getBean("person");  Person p1 = (Person) context.getBean("person");  System.out.println(p);  System.out.println(p1);  } |

[8.2.@Lazy](mailto:8.2.@Lazy)懒加载

Spring默认不执行懒加载，（可以测试）只需在类上加上@Lazy即表示懒加载生效。

|  |
| --- |
| @Component  //@Scope("prototype")  @Lazy  public class Person {  public Person() {  System.out.println("我是person的默认构造");  } |

测试

|  |
| --- |
| //测试懒加载  @Test  public void test03(){  ApplicationContext context =  new ClassPathXmlApplicationContext("applicationContext.xml");    } |

[8.3.@PostConstruct](mailto:8.3.@PostConstruct)初始化方法

@PreDestroy销毁方法

在Person中加入初始化和销毁方法

|  |
| --- |
| @PostConstruct  public void init(){  System.out.println("我是一个初始化方法");  }  @PreDestroy  public void destroy(){  System.out.println("我是一个销毁的方法");  } |

测试

|  |
| --- |
| //测试初始化和销毁方法  @Test  public void test04(){  ClassPathXmlApplicationContext context =  new ClassPathXmlApplicationContext("applicationContext.xml");  Person p = (Person) context.getBean("person");  context.close();  } |

8.4.为了让程序调用更加有条理，分别在不同的层使用不同的注解。

我们之前所有类只要让spring管理，都加@Component，但是以后，包括我们工作中，其实每一层都有一个自己的注解，为了方便区分每一层

web: @Controller

service: @Service

dao: @Repository

这些注解使用上与@Component一模一样。

## 2.9.使用注解实现代码分层

分别创建web、service、dao包，编写接口和实现类。

思考：为什么注入的是接口，依然可以实现注入？

因为@Autowired注入分为三种方式，但是没有@Qualifier，相当于两种注入方式，一种是byname，根据id，显然不能注入，另一种是bytype，这里会根据类型找到接口，但是接口显然不是我们要的，这里spring就做的很好，它会根据接口寻找接口的实现类，完成注入。

在正常开发中，一个接口一般只有一个实现类，几乎不会出现一个接口多个实现类。那如果一个接口有多个实现类，spring还能注入吗？

会直接报错，可以再来一个UserDao的实现类启动测试。

如果我们在特殊情况下，需要一个接口多个实现类，该怎么办？

可以指定id注入。

|  |
| --- |
| @Repository("userDao")  public class UserDaoImpl implements UserDao { |

## 2.10.注解+面向接口编程

面向接口编程，我们调用的都是接口，一般一个程序在设计的时候，需要哪些业务，在接口中就已经定好了，如果我们需要修改业务，只需要修改具体的实现类就可以了，比如我们重新写一个实现类，直接将原有实现类替换即可。这样极大的降低了程序修改所带来的影响。

再比如，我们程序要和银行对接，银行就会给你一大堆接口，只需要你实现了这些接口就可以了。那么，因为有了这些接口，所以数据才会得到规范。比如你传过去一个数据，银行应该怎么解析你这个数据呢？就用到了接口，接口中规范了每个数据的类型。