Spring 框架课程

# Java基础增强：反射

## 解释

思考一个场景：

程序在运行时需要用到一个类的一个方法，但并不能在编码时写死，而是在运行时读取配置文件，根据配置文件中的配置来决定调用哪个类的方法

## 代码案例

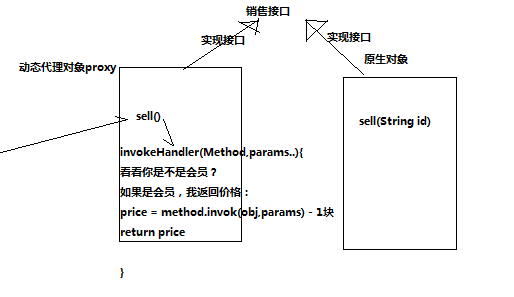
通过反射的方式可以获取class对象中的属性、方法、构造函数等，一下是实例：

|  |
| --- |
| package cn.java.reflect;  import java.lang.reflect.Constructor;  import java.lang.reflect.Field;  import java.lang.reflect.Method;  import java.util.ArrayList;  import java.util.List;  import org.junit.Before;  import org.junit.Test;  public class MyReflect {  public String className = null;  @SuppressWarnings("rawtypes")  public Class personClass = null;  /\*\*  \* 反射Person类  \* @throws Exception  \*/  @Before  public void init() throws Exception {  className = "cn.java.reflect.Person";  personClass = Class.forName(className);  }  /\*\*  \*获取某个class文件对象  \*/  @Test  public void getClassName() throws Exception {  System.out.println(personClass);  }  /\*\*  \*获取某个class文件对象的另一种方式  \*/  @Test  public void getClassName2() throws Exception {  System.out.println(Person.class);  }  /\*\*  \*创建一个class文件表示的真实对象，底层会调用空参数的构造方法  \*/  @Test  public void getNewInstance() throws Exception {  System.out.println(personClass.newInstance());  }  /\*\*  \*获取非私有的构造函数  \*/  @SuppressWarnings({ "rawtypes", "unchecked" })  @Test  public void getPublicConstructor() throws Exception {  Constructor constructor = personClass.getConstructor(Long.class,String.class);  Person person = (Person)constructor.newInstance(100L,"zhangsan");  System.out.println(person.getId());  System.out.println(person.getName());  }  /\*\*  \*获得私有的构造函数  \*/  @SuppressWarnings({ "rawtypes", "unchecked" })  @Test  public void getPrivateConstructor() throws Exception {  Constructor con = personClass.getDeclaredConstructor(String.class);  con.setAccessible(true);//强制取消Java的权限检测  Person person2 = (Person)con.newInstance("zhangsan");  System.out.println(person2.getName());  }  /\*\*  \*获取非私有的成员变量  \*/  @SuppressWarnings({ "rawtypes", "unchecked" })  @Test  public void getNotPrivateField() throws Exception {  Constructor constructor = personClass.getConstructor(Long.class,String.class);  Object obj = constructor.newInstance(100L,"zhangsan");    Field field = personClass.getField("name");  field.set(obj, "lisi");  System.out.println(field.get(obj));  }  /\*\*  \*获取私有的成员变量  \*/  @SuppressWarnings({ "rawtypes", "unchecked" })  @Test  public void getPrivateField() throws Exception {  Constructor constructor = personClass.getConstructor(Long.class);  Object obj = constructor.newInstance(100L);    Field field2 = personClass.getDeclaredField("id");  field2.setAccessible(true);//强制取消Java的权限检测  field2.set(obj,10000L);  System.out.println(field2.get(obj));  }  /\*\*  \*获取非私有的成员函数  \*/  @SuppressWarnings({ "unchecked" })  @Test  public void getNotPrivateMethod() throws Exception {  System.out.println(personClass.getMethod("toString"));    Object obj = personClass.newInstance();//获取空参的构造函数  Object object = personClass.getMethod("toString").invoke(obj);  System.out.println(object);  }  /\*\*  \*获取私有的成员函数  \*/  @SuppressWarnings("unchecked")  @Test  public void getPrivateMethod() throws Exception {  Object obj = personClass.newInstance();//获取空参的构造函数  Method method = personClass.getDeclaredMethod("getSomeThing");  method.setAccessible(true);  Object value = method.invoke(obj);  System.out.println(value);  }  /\*\*  \*  \*/  @Test  public void otherMethod() throws Exception {  //当前加载这个class文件的那个类加载器对象  System.out.println(personClass.getClassLoader());  //获取某个类实现的所有接口  Class[] interfaces = personClass.getInterfaces();  for (Class class1 : interfaces) {  System.out.println(class1);  }  //反射当前这个类的直接父类  System.out.println(personClass.getGenericSuperclass());  /\*\*  \* getResourceAsStream这个方法可以获取到一个输入流，这个输入流会关联到name所表示的那个文件上。  \*/  //path 不以’/'开头时默认是从此类所在的包下取资源，以’/'开头则是从ClassPath根下获取。其只是通过path构造一个绝对路径，最终还是由ClassLoader获取资源。  System.out.println(personClass.getResourceAsStream("/log4j.properties"));  //默认则是从ClassPath根下获取，path不能以’/'开头，最终是由ClassLoader获取资源。  System.out.println(personClass.getResourceAsStream("/log4j.properties"));    //判断当前的Class对象表示是否是数组  System.out.println(personClass.isArray());  System.out.println(new String[3].getClass().isArray());    //判断当前的Class对象表示是否是枚举类  System.out.println(personClass.isEnum());  System.out.println(Class.forName("cn.java.reflect.City").isEnum());    //判断当前的Class对象表示是否是接口  System.out.println(personClass.isInterface());  System.out.println(Class.forName("cn.java.reflect.TestInterface").isInterface());      }  } |

# Java基础增强：动态代理

## 解释

## 代码案例



在之前的代码调用阶段，我们用action调用service的方法实现业务即可。

由于之前在service中实现的业务可能不能够满足当先客户的要求，需要我们重新修改service中的方法，但是service的方法不只在我们这个模块使用，在其他模块也在调用，其他模块调用的时候，现有的service方法已经能够满足业务需求，所以我们不能只为了我们的业务而修改service，导致其他模块授影响。

那怎么办呢？

可以通过动态代理的方式，扩展我们的service中的方法实现，使得在原有的方法中增加更多的业务，而不是实际修改service中的方法，这种实现技术就叫做动态代理。

动态代理：在不修改原业务的基础上，基于原业务方法，进行重新的扩展，实现新的业务。

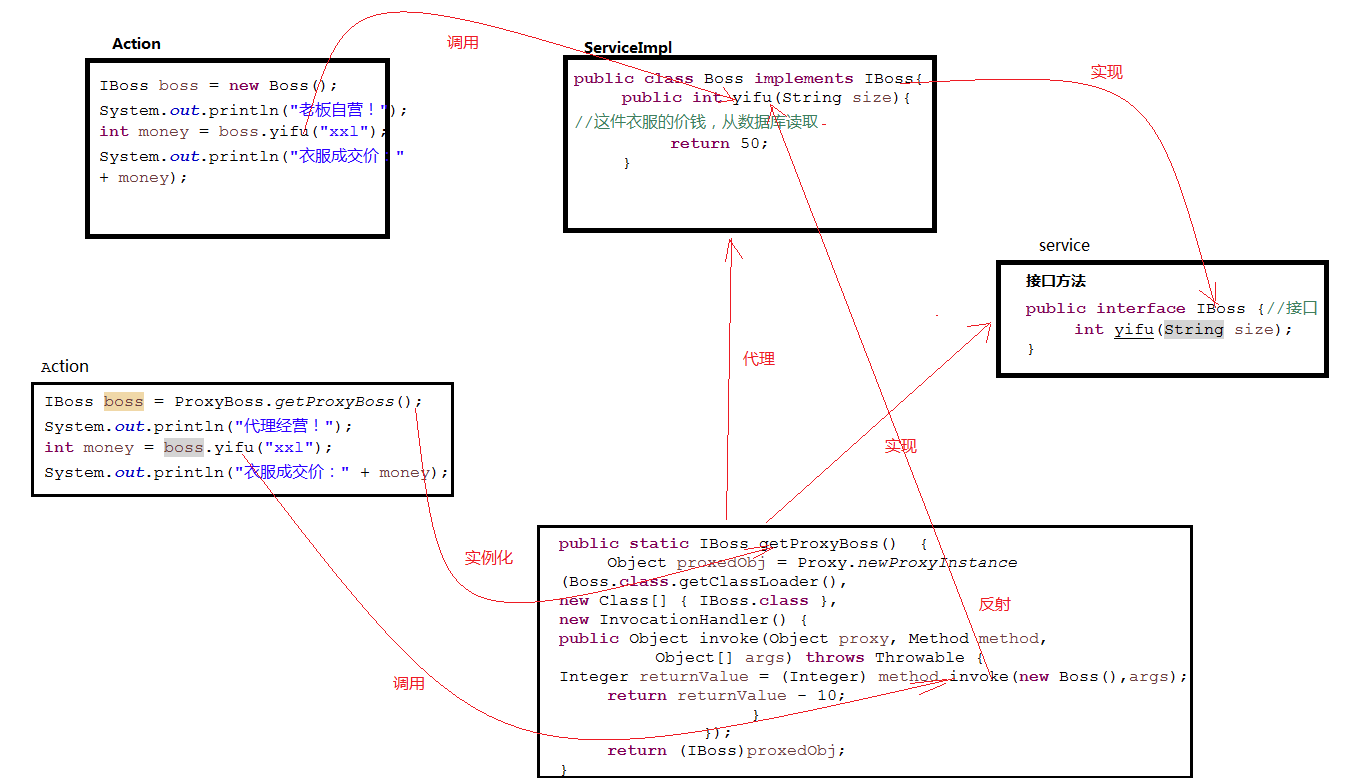
例如下面的例子：

1. 旧业务

买家调用action，购买衣服，衣服在数据库的标价为50元，购买流程就是简单的调用。

1. 新业务

在原先的价格上可以使用优惠券，但是这个功能在以前没有实现过，我们通过代理类，代理了原先的接口方法，在这个方法的基础上，修改了返回值。



代理实现流程：

1. 书写代理类和代理方法，在代理方法中实现代理Proxy.newProxyInstance
2. 代理中需要的参数分别为：被代理的类的类加载器soneObjectclass.getClassLoader()，被代理类的所有实现接口new Class[] { Interface.class }，句柄方法new InvocationHandler()
3. 在句柄方法中复写invoke方法，invoke方法的输入有3个参数Object proxy（代理类对象）, Method method（被代理类的方法）,Object[] args（被代理类方法的传入参数），在这个方法中，我们可以定制化的开发新的业务。
4. 获取代理类，强转成被代理的接口
5. 最后，我们可以像没被代理一样，调用接口的认可方法，方法被调用后，方法名和参数列表将被传入代理类的invoke方法中，进行新业务的逻辑流程。

原业务接口IBoss

|  |
| --- |
| public interface IBoss {//接口  int yifu(String size);  } |

原业务实现类

|  |
| --- |
| public class Boss implements IBoss{  public int yifu(String size){  System.err.println("天猫小强旗舰店，老板给客户发快递----衣服型号："+size);  //这件衣服的价钱，从数据库读取  return 50;  }  public void kuzi(){  System.err.println("天猫小强旗舰店，老板给客户发快递----裤子");  }  } |

原业务调用

|  |
| --- |
| public class SaleAction {  @Test  public void saleByBossSelf() throws Exception {  IBoss boss = new Boss();  System.out.println("老板自营！");  int money = boss.yifu("xxl");  System.out.println("衣服成交价：" + money);  }  } |

代理类

|  |
| --- |
| public static IBoss getProxyBoss(final int discountCoupon) throws Exception {  Object proxedObj = Proxy.newProxyInstance(Boss.class.getClassLoader(),  new Class[] { IBoss.class }, new InvocationHandler() {  public Object invoke(Object proxy, Method method,  Object[] args) throws Throwable {  Integer returnValue = (Integer) method.invoke(new Boss(),  args);// 调用原始对象以后返回的值  return returnValue - discountCoupon;  }  });  return (IBoss)proxedObj;  }  } |

新业务调用

|  |
| --- |
| public class ProxySaleAction {  @Test  public void saleByProxy() throws Exception {  IBoss boss = ProxyBoss.getProxyBoss(20);// 将代理的方法实例化成接口  System.out.println("代理经营！");  int money = boss.yifu("xxl");// 调用接口的方法，实际上调用方式没有变  System.out.println("衣服成交价：" + money);  }  } |

# SPRING简介

## Spring作用

**Spring封装了创建对象的功能**

**内部实现机制：框架写了自己的工厂类，读用户的配置文件，然后反射出配置文件中配置的各种对象**

Spring是分层的JavaSE/EE应用一站式的轻量级开源框架（官网： http://spring.io/ ），以Ioc（Inverse of control）控制反转和Aop（Aspect Oriented Programming）面向切面编程为核心。

轻量级：针对EJB来说，使用方便。

一站式：spring针对各各层（表现层、业务层、持久层）提出解决方案。

表现层：springmvc（spring自己的mvc框架），提供和其它web框架整合方案。

业务层：spring基于aop（面向切面编程）思想进行事务控制。

持久层：spring自己提供JdbcTemplate，提供和其它持久层框架整合的方案。

## Spring核心功能

IOC控制反转)

DI（依赖注入）

AOP（面向切面编程）。

重点是：IOC和DI，spring要管理各各层的bean。

一句话概括spring的作用： SPRING是一个大型系统的粘合剂！

# SPRING工作机制模拟

## 控制反转IOC和依赖注入DI

### 什么是控制反转

IoC (Inverse of Control)即控制反转。是指将原来程序中自己创建实现类对象的控制权反转到IOC容器中。只需要通过IOC获了对象的实例，将IOC当成一个黑盒子、工厂。

spring提供ioc容器，对 bean进行实例化。使用bean时候从容器中取。

IOC控制反转，将对象的创建权反转到了spring容器中。

### 什么是依赖注入

### 模拟实现IOC和DI

# SPRING入门示例程序

## 新建工程

## 导入JAR包

•spring-beans-4.2.4.RELEASE.jar

•spring-context-4.2.4.RELEASE.jar

•spring-core-4.2.4.RELEASE.jar

•spring-expression-4.2.4.RELEASE.jar

spring使用JCL日志体系（commons-logging-1.2.jar）

commons-logging：相当 于原来的slf4j，只有日志接口

还需要加入日志实现：log4j

## 业务类准备

### UserController

|  |
| --- |
| **public** **class** UserController {    UserService userService;  **public** UserController(UserService userService) {  **this**.userService = userService;  }      **public** String getUserById(**int** id){    **return** userService.getUserById(id);  }  } |

### UserSerivce接口

|  |
| --- |
| **public** **interface** UserService {  String getUserById(**int** id);  } |

### UserServiceImpl实现类

|  |
| --- |
| **public** **class** UserServiceImpl **implements** UserService {  UserDao userDao;    **public** UserServiceImpl(UserDao userDao) {  **this**.userDao = userDao;  }  @Override  **public** String getUserById(**int** id) {    **return** userDao.getUserById(id);  }  } |

### UserDao接口

|  |
| --- |
| **public** **interface** UserDao {  String getUserById(**int** id);  } |

### UserDaoImpl实现类

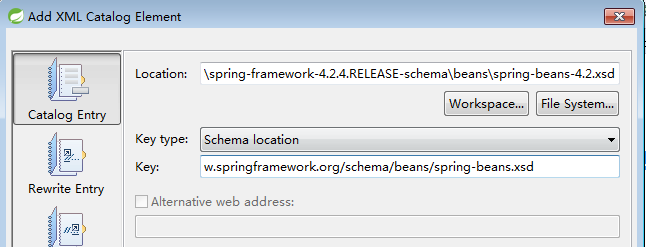
|  |
| --- |
| **public** **class** UserDaoImpl **implements** UserDao{  @Override  **public** String getUserById(**int** id) {  **return** "zhangsan";  }  } |

## SPRING配置文件

spring的ioc容器的配置文件：applicationContext.xml（默认名称）

### 配置schema约束：

http://www.springframework.org/schema/beans/spring-beans.xsd



spring的ioc容器的配置文件：applicationContext.xml（默认名称）

配置schema约束：

http://www.springframework.org/schema/beans/spring-beans.xsd

## 配置BEAN--applicationContext.xml

1、编写好接口及接口实现类

2、需要在spring的容器的配置文件中配置spring要管理的bean。

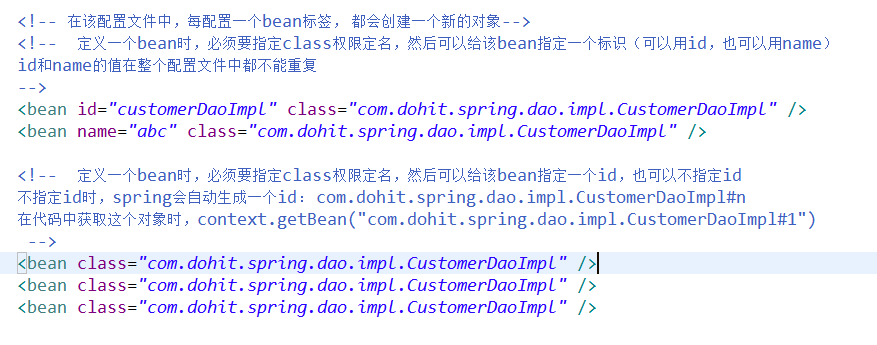
|  |
| --- |
| <?xml version=*"1.0"* encoding=*"UTF-8"*?>  <beans xmlns=*"http://www.springframework.org/schema/beans"*  xmlns:context=*"http://www.springframework.org/schema/context"* xmlns:p=*"http://www.springframework.org/schema/p"*  xmlns:aop=*"http://www.springframework.org/schema/aop"* xmlns:tx=*"http://www.springframework.org/schema/tx"*  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-4.0.xsd*  *http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context-4.0.xsd*  *http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop-4.0.xsd http://www.springframework.org/schema/tx http://www.springframework.org/schema/tx/spring-tx-4.0.xsd*  *http://www.springframework.org/schema/util http://www.springframework.org/schema/util/spring-util-4.0.xsd"*>    <!-- <context:component-scan base-package="com.dohit.springmvc.service"/> -->    <bean id=*"userController"* class=*"com.doit.spring.demo.UserController"*>  <constructor-arg name=*"userService"* ref=*"userService"*></constructor-arg>  </bean>      <bean id=*"userService"* class=*"com.doit.spring.demo.UserServiceImpl"*>  <constructor-arg name=*"userDao"* ref=*"userDao"*></constructor-arg>  </bean>    <bean id=*"userDao"* class=*"com.doit.spring.demo.UserDaoImpl"*></bean>    </beans> |

## 运行测试

|  |
| --- |
| **public** **class** MainTest {  **public** **static** **void** main(String[] args) {  // 启动spring的容器，加载配置文件，构造我们配置的所有bean  ClassPathXmlApplicationContext applicationContext = **new** ClassPathXmlApplicationContext("classpath:applicationContext.xml");    // 从spring的容器中获取我们想要的bean  UserDao userDao = (UserDao) applicationContext.getBean("userDao");    String userById = userDao.getUserById(1);    System.***out***.println(userById);  }  } |

# IOC更多配置

## BEAN的定义

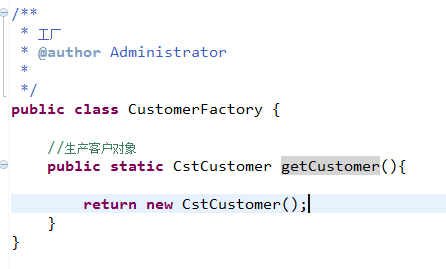


## BEAN的初始化配置

spring要对applicationContext.xml中配置的bean进行实例化（初始化）。

包括：通过无参构造器实例化、通过有参构造器实例化、通过静态工厂方法

|  |
| --- |
| <!-- 测试spring对bean的实例化方法 -->  <!-- 默认通过无参构造器 -->  <bean id=*"customer1"* class=*"com.dohit.pojo.CstCustomer"*></bean>  <!-- 通过有参构造器  构造器：public CstCustomer(Long custId,String custName)  -->  <bean id=*"customer2"* class=*"com.dohit.pojo.CstCustomer"*>  <!-- index：参数位置，第一个参数位置为0  value：参数值  type：参数类型  -->  <constructor-arg index=*"0"* value=*"101"* type=*"java.lang.Long"*/>  <constructor-arg index=*"1"* value=*"牛牛"* type=*"java.lang.String"*/>    </bean>  <!-- 了解，通过静态工厂方法获取bean的实例  class：配置工厂类的路径  factory-method：调用工厂方法，获取对象  -->  <bean id=*"customer3"* class=*"com.dohit.pojo.CustomerFactory"* factory-method=*"getCustomer"*></bean> |



## BEAN的获取

getBean(bean标识)

// 通过类型去获取bean时，spring配置文件中，该类型的bean只能有一个

getBean(bean类型.class)

## ApplicationContext加载配置文件

ApplicationContext理解为spring容器的上下文，通过上下文操作容器中bean。

ClassPathXmlApplicationContext：加载classpath下的配置文件创建一个容器实例

FileSystemXmlApplicationContext：加载文件系统中任意目录下的配置文件，创建一个容器实例

掌握：多文件的加载方法

|  |
| --- |
| // ClassPathXmlApplicationContext applicationContext = new ClassPathXmlApplicationContext("applicationContext-dao.xml","applicationContext-service.xml");  // ClassPathXmlApplicationContext applicationContext = new ClassPathXmlApplicationContext(new  String[]{"applicationContext-dao.xml","applicationContext-service.xml"});  //ClassPathXmlApplicationContext applicationContext = new ClassPathXmlApplicationContext(new String[] { "classpath:applicationContext-dao.xml", "applicationContext-service.xml" });  // ClassPathXmlApplicationContext applicationContext = new ClassPathXmlApplicationContext("classpath:applicationContext-\*.xml");  //ClassPathXmlApplicationContext applicationContext = new ClassPathXmlApplicationContext(new String[] { "classpath:applicationContext-\*.xml", "beans.xml" });  //ClassPathXmlApplicationContext applicationContext = new ClassPathXmlApplicationContext("classpath:\*.xml");  // ClassPathXmlApplicationContext applicationContext = new ClassPathXmlApplicationContext(new String[] { "classpath:\*.xml", "classpath:springmvc/beans.xml" });  //ClassPathXmlApplicationContext applicationContext = new ClassPathXmlApplicationContext("classpath\*:\*.xml"); |

# DI依赖注入

## DI的概念

**依赖注入（Dependency Injection）**”。**所谓依赖注入，就是由****IOC容器在运行期间，动态地将对象的依赖关系注入到对象的属性中。**

service：依赖dao，如何实现依赖注入？

1、spring要管理service（前提）

2、spring要管理dao（前提）

总结前提：依赖方（service）、被依赖方（dao）都需要被spring管理

3、根据依赖关系，service依赖dao，将dao实例注入至service的属性中。

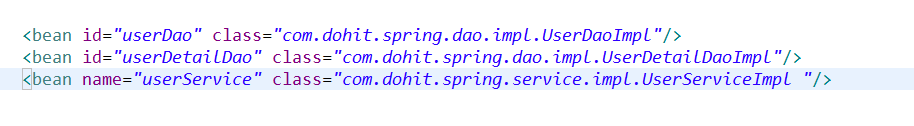
底层原理：spring根据配置文件中配置依赖关系，首先获取被依赖的对象dao实例，调用service对象的构造函数或者set方法将dao实例设置(*注入*)到service属性。

## DI测试

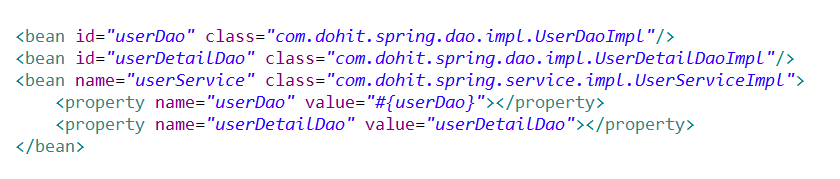
目标：

让spring将service依赖的dao注入。

1、在spring的容器中配置dao和service



2、配置依赖关系，service依赖dao



## 小结Ioc和Di的区别

ioc：控制反转，将对象的创建权反转到ioc容器。

DI：依赖注入，将对象所依赖的对象注入到对象的属性中。 就是IOC的具体实现方法。

1、IOC就是一个容器

2、IOC容器中包括spring管理的所有bean。

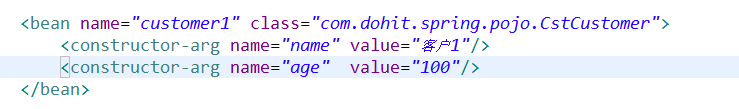
3、IOC容器负责对bean进行实例化

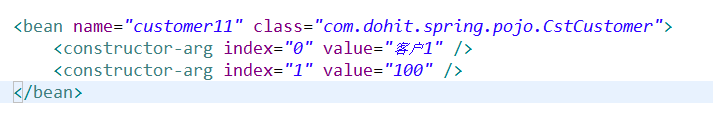
4、IOC容器对bean进行实例化时候，检查有哪些依赖的属性，将依赖的属性注入到实例化的bean的属性中。

要实现依赖注入，需要spring管理依赖方和被依赖方（spring要对依赖方和被依赖方实例化）。

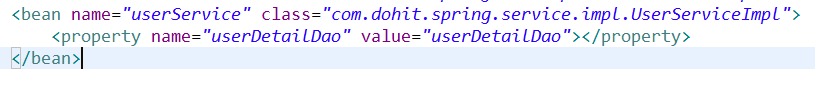
## 依赖注入方法

### 通过有参构造器注入属性值（常用！）





### 通过 set方法注入（常用）



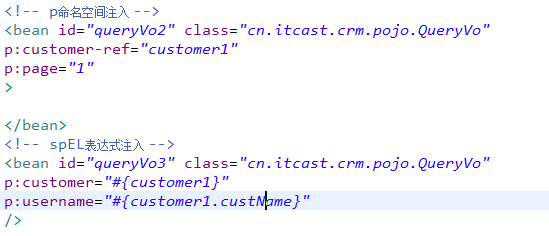
### 测试set方法注入支持属性类型：

|  |
| --- |
| **public** **class** QueryVo {  **private** AImpl aImpl;  **private** **int** page;  **private** String username;  **private** List<String> listString;  **private** List<AImpl> aImplList;  **private** Map<String,Object> map;  **private** Properties properties; |

bean配置：

|  |
| --- |
| <!-- 测试set方法依赖注入的属性类型 -->  <bean id=*"queryVo"* class=*"com.dohit.spring.pojo.QueryVo"*>  <!-- 基本类型 -->  <property name=*"page"* value=*"1"*></property>  <property name=*"username"* value=*"大牛"*></property>  <!-- pojo属性 -->  <property name=*"customer"* ref=*"customer1"*></property>  <!-- list List<String> -->  <property name=*"listString"*>  <list>  <value>小牛</value>  <value>小牛</value>  </list>  </property>  <!-- list List<CstCustomer> -->  <property name=*"customerList"*>  <list>  <ref bean=*"customer3"*/>  <ref bean=*"customer3"*/>  </list>  </property>  <!-- map -->  <property name=*"map"*>  <map>  <entry key=*"101"* value=*"小牛"*></entry>  <entry key=*"102"* value=*"大牛"*></entry>  </map>  </property>  <!-- properties -->  <property name=*"properties"*>  <props>  <prop key=*"101"* >小牛</prop>  <prop key=*"102"* >大牛</prop>  </props>  </property>  </bean> |

### p命名空间和spEL表达式注入方式（了解）



# 基于注解方式进行IOC开发

使用注解，spring通过注解方式完成bean的管理。

bean的管理包括两部分：

1、对 bean实例化

2、对bean依赖注入

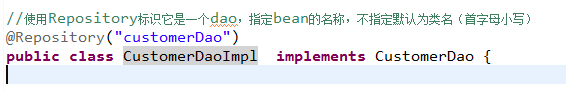
## 准备环境

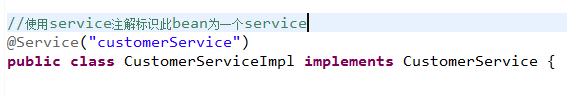
将applicationContext.xml文件置空

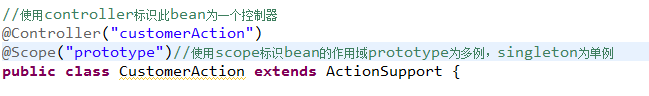
加入spring-aop-4.2.4.RELEASE.jar

## 实例化的注解

@Repository：标识此bean为一个dao







在spring中提供了@component标识就是一个bean。

@Component可以用在任何的bean上。

## 配置context:component-scan组件扫描

引入context命名空间

<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"*>

<!-- 配置组件扫描

component-scan可以扫描注解：@controller、@Service、@Repository、@Component

指定扫描的包com.dohit.spring，可以扫描子包下的类

-->

<context:component-scan base-package=*"com.dohit.spring"*></context:component-scan>

</beans>

原理：

在加载spring容器时，根据上边配置的包路径，扫描包下的及子包下的类，如果标识了：@controller、@Service、@Repository、@Component进行实例化。

~~了解，进行过虑配置：~~

<context:component-scan base-package=*"com.dohit.spring"*>

<!-- 扫描 org.springframework.stereotype.Service-->

<context:include-filter type=*"annotation"* expression=*"org.springframework.stereotype.Service"*/>

<!-- 扫描org.springframework.stereotype.Repository-->

<context:include-filter type=*"annotation"* expression=*"org.springframework.stereotype.Repository"*/>

<!-- 不扫描org.springframework.stereotype.Controller -->

<context:exclude-filter type=*"annotation"* expression=*"org.springframework.stereotype.Controller"*/>

</context:component-scan>

## 依赖注入的注解

@Autowired：

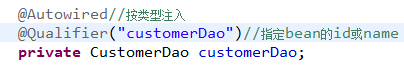
按类型注入

1、可以标识在属性上，根据属性类型去spring容器中找同类型的bean，找到则将bean注入到属性中。

@Autowired//按类型注入

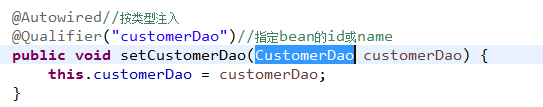
**private** CustomerDao customerDao;

注意：如果容器中同一个类型的bean如果有多个，使用Autowried报错，找到多个同类型的bean，使用@Qualifier和Autowired组合配置，Qualifier指定将哪个bean注入进来。



2、可以用在set方法上。

找set方法中参数类型，根据参数类型从容器中找bean，注入。



3、使用jdk的注解@Resource：

按名称注入



和autoWired区别：Resource注解是jdk下的，而autoWired是spring提供。

# @Component 和@autowired的id问题

@Component 构造的bean，id默认为类名的首写字母小写

但是也可以手动设置一个id，方式如下：

@Component(“beanid”) 或者 @Component(value=“beanid”)

@autowired 是按类型来注入bean

但，如果同类型在容器中有多个bean时，@autowired 就会报错

解决办法：

可以为@autowired 加一个注解来注入指定id的bean

@autowired

@Qualifier(value=“beanid”)

UserService userService;