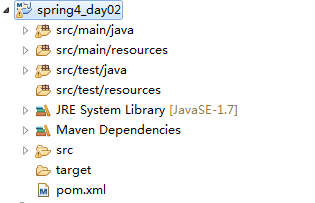
# Spring第二天

内容回顾：

1. spring:一站式框架
2. IOC:控制反转，将对象的创建权利反转给spring框架，解耦。
3. Spring中的工厂
   1. ApplicationContext
   2. BeanFactory
   3. 区别：创建对象的时机不一样
4. Spring中的核心配置文件
   1. id
   2. class
   3. scope:singleton prototype
   4. init-method
   5. destroy-method
5. Spring中生成bean的三种方式
   1. 无参构造
   2. 静态工厂方法实例化
   3. 实例工厂方法实例化
6. DI:依赖注入
   1. 构造方法注入：<constructor-arg>
   2. set方法注入：<property name=”” value=””>
7. 基于set方法注入新的写法
   1. p命名空间
   2. spel

# Spring IOC注解的快速入门

## 创建工程并引入依赖



pom.xml的内容如下：

<project xmlns=*"http://maven.apache.org/POM/4.0.0"* xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*

xsi:schemaLocation=*"http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd"*>

<modelVersion>4.0.0</modelVersion>

<parent>

<groupId>cn.itcast.parent</groupId>

<artifactId>itcast-parent</artifactId>

<version>0.0.1-SNAPSHOT</version>

</parent>

<groupId>cn.itcast</groupId>

<artifactId>spring4\_day02</artifactId>

<version>1.0.0-SNAPSHOT</version>

<dependencies>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-context</artifactId>

</dependency>

<dependency>

<groupId>org.slf4j</groupId>

<artifactId>slf4j-log4j12</artifactId>

</dependency>

<dependency>

<groupId>junit</groupId>

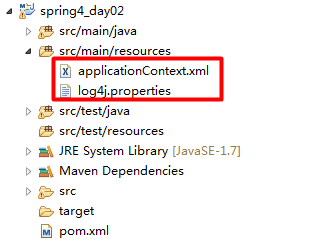
<artifactId>junit</artifactId>

</dependency>

</dependencies>

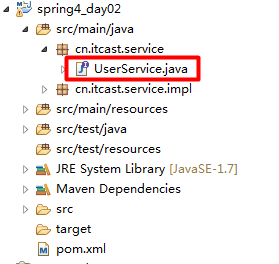
</project>

* 引入applicationContext.xml和log4j.properties



## 创建包结构并编写Java类

* 创建UserService接口



**package** cn.itcast.service;

**public** **interface** UserService {

/\*\*

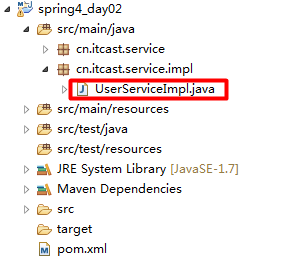
\* 业务层：用户保存

\*/

**public** **void** saveUser();

}

* 创建UserService的实现类UserServiceImpl



**package** cn.itcast.service.impl;

**import** cn.itcast.service.UserService;

**public** **class** UserServiceImpl **implements** UserService {

@Override

**public** **void** saveUser() {

System.***out***.println("业务层：用户保存...");

}

}

## 在Java类上添加注解

* 在UserServiceImpl实现类上添加注解@Component,相当于<bean id=”” class=””>,value属性给bean指定id，value属性的名字也可以不写

@Component("userService")

**public** **class** UserServiceImpl **implements** UserService{

@Override

**public** **void** saveUser() {

System.***out***.println("业务层：用户保存...");

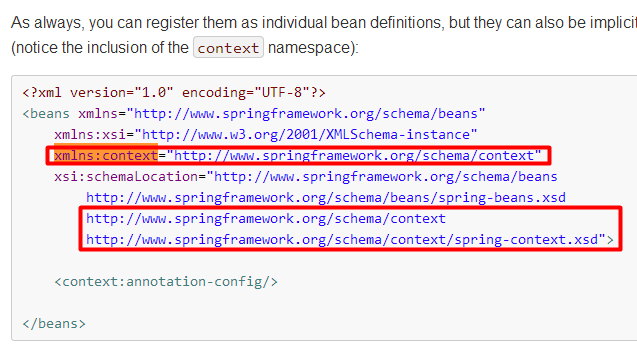
}

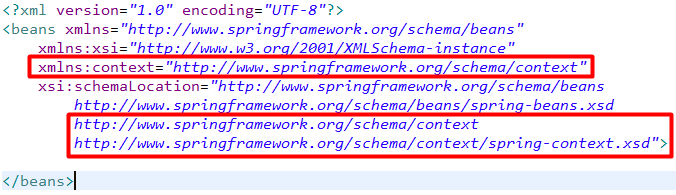
}

## 在applicationContext.xml中引入约束

注意：因为现在想使用注解，那么引入的约束发生了变化，需要context的约束。

【提示】：约束可以从spring开发文档中拷贝，也可以从笔记里拷贝。如果从spring开发文档中拷贝，可以参考spring开发文档的6.9节





添加完context约束之后的applicationContext.xml内容如下：

<?xml version=*"1.0"* encoding=*"UTF-8"*?>

<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"*>

</beans>

## 在applicationContext.xml中开启注解扫描

在applicationContext.xml通过context:component-scan标签开启spring注解扫描，扫描时是以包范围来扫描的：

<?xml version=*"1.0"* encoding=*"UTF-8"*?>

<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"*>

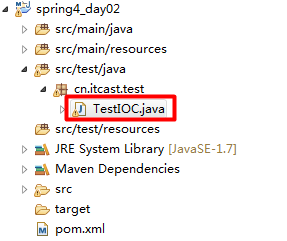
<!-- 开启注解扫描 -->

**<context:component-scan base-package=*"cn.itcast.service.impl"*></context:component-scan>**

</beans>

## 编写测试代码

* 创建单元测试类TestIOC，在其中创建单元测试方法test1



/\*\*

\* 测试注解

\*/

@Test

**public** **void** test1(){

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

UserService userService = (UserService) applicationContext.getBean("userService");

userService.saveUser();

}

注意：@Component注解的value可以不用指定，默认就是类名首字母小写。

# Spring框架中bean管理的常用注解

Spring中用于管理bean的注解分为四大类：

1. 用于创建对象
2. 用于给对象中的属性注入值
3. 用于改变作用范围
4. 用于定义生命周期

## 用于创建对象的（重点）

用于创建对象的有四个：@Component,@Controller,@Service,@Repository

### @Component注解

**作用：**

把资源让spring来管理。相当于在xml中配置一个bean。

**属性：**

value：指定bean的id。如果不指定value属性，默认bean的id是当前类的类名。首字母小写。



### @Controller @Service @Repository

他们三个注解都是针对一个的衍生注解，他们的作用及属性都是一模一样的。

他们只不过是提供了更加明确的语义化。

**@Controller：**一般用于表现层的注解。

**@Service：**一般用于业务层的注解。

**@Repository：**一般用于持久层的注解。

@Service的用法：修改UserServiceImpl类，把@Component改成@Service

//@Component("userService")

@Service("userService")

**public** **class** UserServiceImpl **implements** UserService{

@Override

**public** **void** saveUser() {

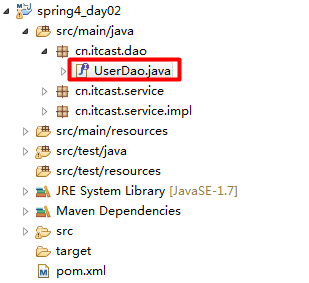
System.***out***.println("业务层：用户保存...");

}

}

@Repository的用法：

创建UserDao接口：

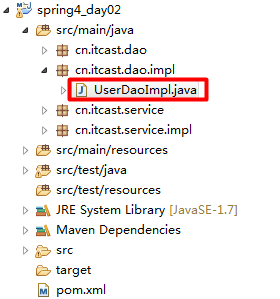


**public** **interface** UserDao {

**public** **void** save();

}

创建UserDao接口的实现类UserDaoImpl，在该类上加@Repository注解



@Repository("userDao")

**public** **class** UserDaoImpl **implements** UserDao {

@Override

**public** **void** save() {

System.***out***.println("持久层：用户保存...");

}

}

**注意：此处测试时，要把扫描的包定义为cn.itcast,不然的其它包的注解就不能识别了**

<?xml version=*"1.0"* encoding=*"UTF-8"*?>

<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*

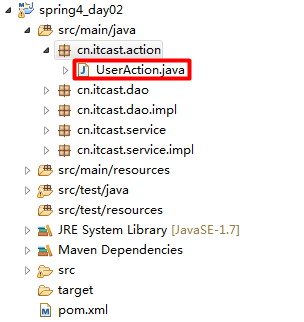
*http://www.springframework.org/schema/context/spring-context.xsd"*>

<!-- 开启注解扫描 -->

**<context:component-scan base-package=*"cn.itcast"*></context:component-scan>**

</beans>

@Controller的用法：创建UserAction类，在该类上加@Controller注解



@Controller("userAction")

**public** **class** UserAction {

}

**说明：这三个注解是为了让标注类本身的用途清晰**

## 用于注入数据的

用于注入数据的注解有：

1. @Value
2. @Autowired
3. @Qualifier
4. @Resource

**相当于：<property name="" ref="">**

**<property name="" value="">**

### @Value

**作用：**

注入基本数据类型和String类型数据的

**属性：**

value：用于指定值

修改UserServiceImpl类，增加一个字符串属性name，现在要通过@Value给name属性注入值

@Service("userService")

**public** **class** UserServiceImpl **implements** UserService {

**@Value("张三")**

**private** String name;

@Override

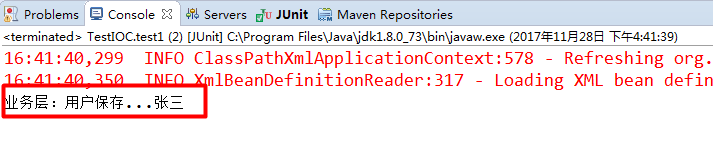
**public** **void** saveUser() {

System.***out***.println("业务层：用户保存..." + name);

}

}

运行TestIOC中的test1方法，测试结果如下：



### @Autowired（重点）

**作用：**

自动按照类型注入。当使用注解注入属性时，set方法可以省略。它只能注入其他bean类型。当有多个类型匹配时，使用要注入的对象变量名称作为bean的id，在spring容器查找，找到了也可以注入成功。找不到就报错。

修改UserServiceImpl类，增加一个对象属性userDao，现在通过@Autowired给userDao注入值

@Service("userService")

**public** **class** UserServiceImpl **implements** UserService {

@Value("张三")

**private** String name;

@Autowired

**private** UserDao userDao;

@Override

**public** **void** saveUser() {

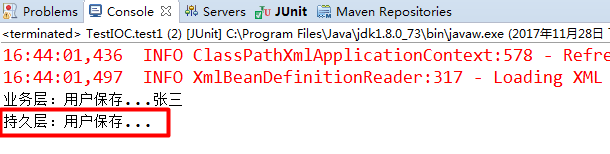
System.***out***.println("业务层：用户保存..." + name);

userDao.save();

}

}

运行TestIOC中的test1方法，测试结果如下：



### @Qualifier

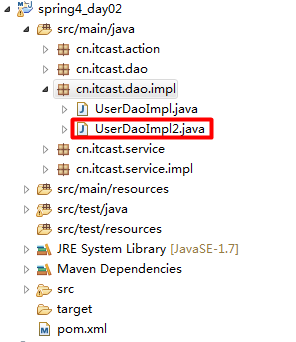
**作用：**

在自动按照类型注入的基础之上，再按照Bean的id注入。它在给字段注入时不能独立使用，必须和@Autowire一起使用；但是给方法参数注入时，可以独立使用。

**属性：**

value：指定bean的id。

* 创建UserDao接口的第二个实现类UserDaoImpl2



@Repository("userDao2")

**public** **class** UserDaoImpl2 **implements** UserDao {

@Override

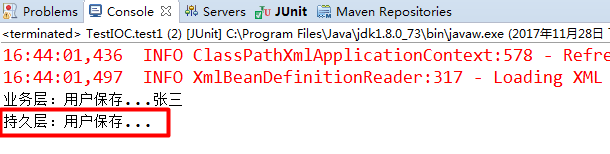
**public** **void** save() {

System.***out***.println("持久层：用户保存2222...");

}

}

运行TestIOC中的test1方法，测试结果如下：



* 测试发现，UserServiceImpl中注入的还是第一个UserDaoImpl，因为当有多个bean都满足的情况下，优先注入bean的id与属性的名字一样的bean；想指定注入UserDaoImpl2，需要使用@Qualifier注解根据名字来注入

@Service("userService")

**public** **class** UserServiceImpl **implements** UserService {

@Value("张三")

**private** String name;

@Autowired

**@Qualifier("userDao2")**

**private** UserDao userDao;

@Override

**public** **void** saveUser() {

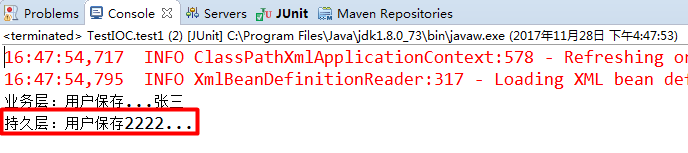
System.***out***.println("业务层：用户保存..." + name);

userDao.save();

}

}

再次运行TestIOC中的test1方法，发现UserServiceImpl中注入的是UserDaoImpl2;测试结果如下：



### @Resource

**作用：**

直接按照Bean的id注入。它也只能注入其他bean类型。

**属性：**

name：指定bean的id。

修改UserServiceImpl类，使用@Resource给userDao注入值。@Resource是按照bean的id来注入，只能注入对象类型

@Service("userService")

**public** **class** UserServiceImpl **implements** UserService {

@Value("张三")

**private** String name;

// @Autowired

// @Qualifier("userDao2")

@Resource(name="userDao2")

**private** UserDao userDao;

@Override

**public** **void** saveUser() {

System.***out***.println("业务层：用户保存..." + name);

userDao.save();

}

}

## 用于改变作用域范围的

### @Scope（重点）

**作用：**

指定bean的作用范围。

**属性：**

value：指定范围的值。

取值：singleton prototype request session globalsession

* @Scope指定bean的作用域，默认值是singleton，单例的。
* 修改UserServiceImpl，在该类上加@Scope注解，指定该类是多例的的，默认是单例的。给该类显示指定一个无参构造方法，方便测试

@Service("userService")

**@Scope("prototype")**

**public** **class** UserServiceImpl **implements** UserService{

@Value("张三")

**private** String name;

// @Autowired

// @Qualifier("userDao2")

@Resource(name="userDao2")

**private** UserDao userDao;

**public** UserServiceImpl() {

System.***out***.println("调用了无参构造方法...");

}

@Override

**public** **void** saveUser() {

System.***out***.println("业务层：用户保存..." + name);

userDao.save();

}

}

* 在TestIOC中创建test2单元测试方法

@Test

**public** **void** test2(){

//创建ioc容器

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

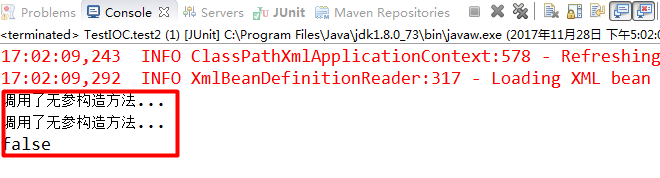
UserService userService1 = (UserService) ac.getBean("userService");

UserService userService2 = (UserService) ac.getBean("userService");

System.***out***.println(userService1 == userService2);

}

* 测试发现：当scope指定为prototype时，两次获取UserService的对象是不一致的。测试结果如下：



## 和生命周期相关的

**相当于：<bean id="" class="" init-method="" destroy-method="" />**

### @PostConstruct注解

@PostConstruct加在方法上，指定bean对象创建好之后，调用该方法初始化对象，类似于xml的init-method方法。修改UserServiceImpl类，在其中增加一个init方法，在该方法上指定@PostConstruct注解

@Service("userService")

**public** **class** UserServiceImpl **implements** UserService {

@Value("张三")

**private** String name;

// @Autowired

// @Qualifier("userDao2")

@Resource(name="userDao2")

**private** UserDao userDao;

**public** UserServiceImpl() {

System.***out***.println("调用了无参构造方法...");

}

@PostConstruct

**public** **void** init(){

System.***out***.println("调用了init方法...");

}

@Override

**public** **void** saveUser() {

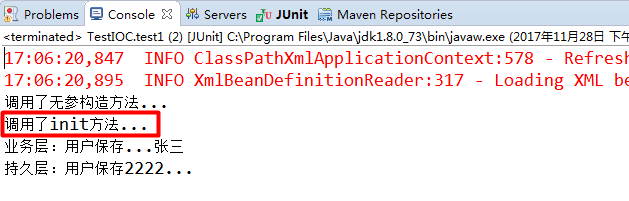
System.***out***.println("业务层：用户保存..." + name);

userDao.save();

}

}

运行TestIOC中的test1方法，测试结果如下：



### @PreDestory注解

@PreDestory加在方法上，指定bean销毁之前，调用该方法，类似于xml的destory-method方法。修改UserServiceImpl类，在该类中增加一个destroy方法，在该方法上加@PreDestroy注解

@Service("userService")

**public** **class** UserServiceImpl **implements** UserService {

@Value("张三")

**private** String name;

// @Autowired

// @Qualifier("userDao2")

@Resource(name="userDao2")

**private** UserDao userDao;

**public** UserServiceImpl() {

System.***out***.println("调用了无参构造方法...");

}

@PostConstruct

**public** **void** init(){

System.***out***.println("调用了init方法...");

}

@Override

**public** **void** saveUser() {

System.***out***.println("业务层：用户保存..." + name);

userDao.save();

}

@PreDestroy

**public** **void** destroy(){

System.***out***.println("调用了destroy方法...");

}

}

注意：[要看到@PreDestory的效果，需要调用ClassPathXmlApplicationContext.close方法，同时scope的值要是singleton](mailto:要看到@PreDestory的效果，需要调用ClassPathXmlApplicationContext.close方法，同时scope的值要是singleton)。所以，还得修改test1方法，显示关闭ioc容器

@Test

**public** **void** test1(){

//创建ioc容器

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

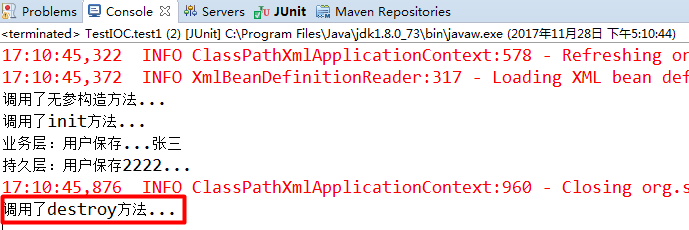
UserService userService = (UserService) ac.getBean("userService");

userService.saveUser();

((ClassPathXmlApplicationContext)ac).close();

}

测试结果如下：



注解小结：

与创建对象相关

@Component

@Controller

@Service

@Repository

与注入值相关的

@Value

@Autowired

@Qualifier

@Resource

与范围相关

@Scope

与生命周期相关

@PostContruct

@PreDestroy

# XML和注解的比较

**注解的优势：**

配置简单，维护方便。（我们找到了类，就相当于找到了配置）

**XML的优势：**

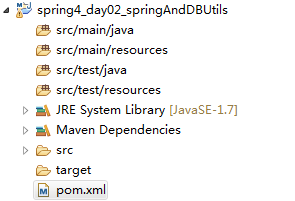
修改时，不用改源码。不涉及重新编译和部署。

Xml和注解的比较



# 案例：Spring整合DBUtils实现增删改查

## 2第一步：创建工程并引入依赖



需要引入的依赖有：spring ioc+druid+mysql+dbutils。完整的pom.xml内容如下：

<project xmlns=*"http://maven.apache.org/POM/4.0.0"* xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*

xsi:schemaLocation=*"http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd"*>

<modelVersion>4.0.0</modelVersion>

<parent>

<groupId>cn.itcast.parent</groupId>

<artifactId>itcast-parent</artifactId>

<version>0.0.1-SNAPSHOT</version>

</parent>

<groupId>cn.itcast</groupId>

<artifactId>spring4\_day02\_springAndDBUtils</artifactId>

<version>1.0.0-SNAPSHOT</version>

<dependencies>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-context</artifactId>

</dependency>

<dependency>

<groupId>org.slf4j</groupId>

<artifactId>slf4j-log4j12</artifactId>

</dependency>

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

</dependency>

<dependency>

<groupId>mysql</groupId>

<artifactId>mysql-connector-java</artifactId>

</dependency>

<dependency>

<groupId>commons-dbutils</groupId>

<artifactId>commons-dbutils</artifactId>

<version>1.7</version>

</dependency>

<dependency>

<groupId>com.alibaba</groupId>

<artifactId>druid</artifactId>

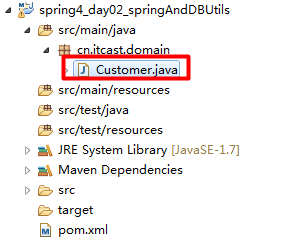
</dependency>

</dependencies>

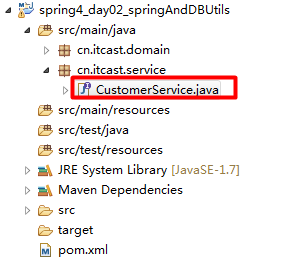
</project>

## 第二步：创建业务层接口及实现类

创建Customer实体类



创建CustomerService接口



**public** **interface** CustomerService {

/\*\*

\* 业务层：查询所有客户

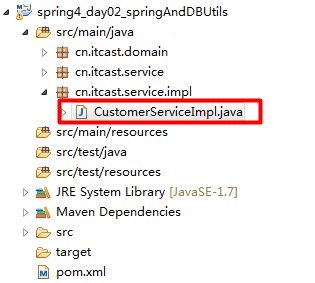
\* **@return**

\*/

**public** List<Customer> findAllCustomer();

}

创建CustomerService接口的实现类CustomerServiceImpl



**public** **class** CustomerServiceImpl **implements** CustomerService {

**private** CustomerDao customerDao;

**public** **void** setCustomerDao(CustomerDao customerDao) {

**this**.customerDao = customerDao;

}

@Override

**public** List<Customer> findAllCustomer() {

List<Customer> list = customerDao.findAll();

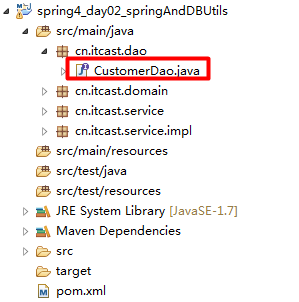
**return** list;

}

}

## 第三步：创建dao层接口及实现类

创建UserDao接口

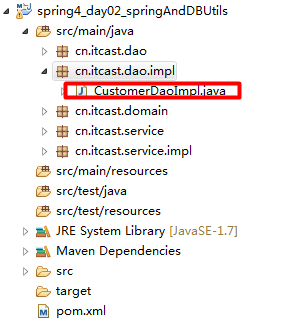


**public** **interface** CustomerDao {

**public** List<Customer> findAll();

}

创建UserDao接口的实现类



**public** **class** CustomerDaoImpl **implements** CustomerDao {

**private** QueryRunner queryRunner;//不需要实例化，通过spring依赖注入进来

**public** **void** setQueryRunner(QueryRunner queryRunner) {

**this**.queryRunner = queryRunner;

}

@Override

**public** List<Customer> findAll() {

List<Customer> list = **null**;

**try** {

list = queryRunner.query("select \* from cst\_customer", **new** BeanListHandler<Customer>(Customer.**class**));

} **catch** (SQLException e) {

e.printStackTrace();

}

**return** list;

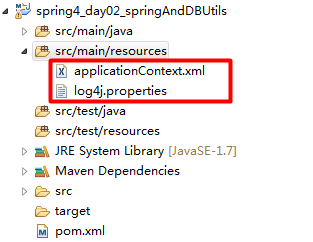
}

}

## 第四步：编写spring配置文件

在src下创建spring的配置文件applicationContext.xml,把Service、Dao、QueryRunner、DataSource配置到Spring中

1. Service中需要注入DAO
2. DAO中需要注入QueryRunner
3. QueryRunner中需要注入DataSource
4. DataSource中需要注入驱动、连接地址、用户名、密码



<?xml version=*"1.0"* encoding=*"UTF-8"*?>

<beans xmlns=*"http://www.springframework.org/schema/beans"*

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.xsd"*>

<bean id=*"customerService"* class=*"cn.itcast.service.impl.CustomerServiceImpl"*>

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

</bean>

<bean id=*"customerDao"* class=*"cn.itcast.dao.impl.CustomerDaoImpl"*>

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

</bean>

<bean id=*"queryRunner"* class=*"org.apache.commons.dbutils.QueryRunner"*>

<constructor-arg name=*"ds"* ref=*"dataSource"*></constructor-arg>

</bean>

<bean id=*"dataSource"* class=*"com.alibaba.druid.pool.DruidDataSource"*>

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

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

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

<property name=*"password"* value=*"123456"*></property>

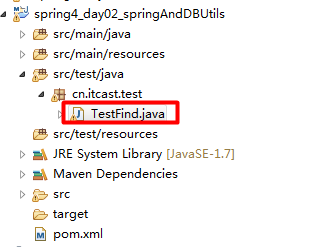
</bean>

</beans>

注意：还得引入log4j.properties文件

## 第五步：编写测试类

创建单元测试类TestFind,在其中创建test1方法



@Test

**public** **void** test1(){

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

CustomerService customerService = (CustomerService) ac.getBean("customerService");

List<Customer> customers = customerService.findAllCustomer();

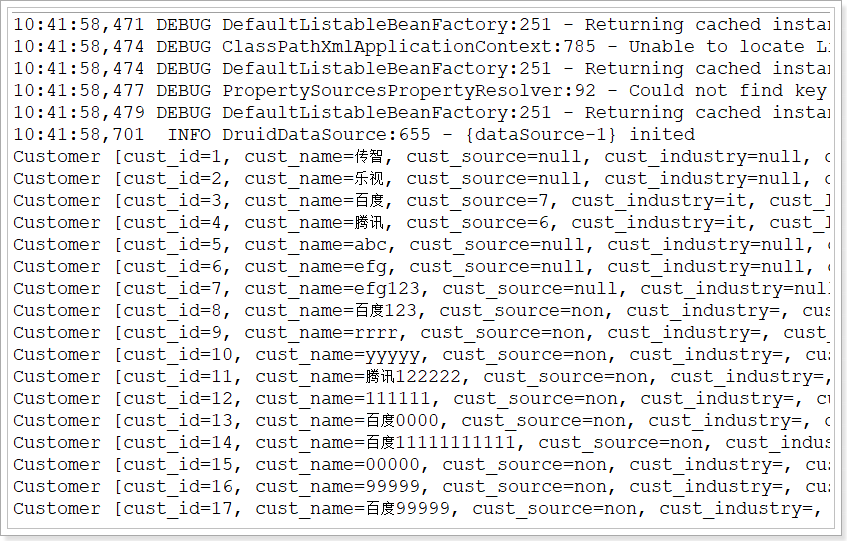
**for** (Customer customer : customers) {

System.***out***.println(customer);

}

}

测试结果如下：



# 案例：采用注解把Spring和DBUtils进行整合

可以把第四章的工程里直接改成注解的形式

## 第一步：在类上加对应的注解

修改CustomerServiceImpl类，在该类上加@Service注解。在customerDao上加@Autowired注解，表示给该属性注入值

@Service("customerService")

**public** **class** CustomerServiceImpl **implements** CustomerService {

@Autowired

**private** CustomerDao customerDao;

@Override

**public** List<Customer> findAllCustomer() {

List<Customer> list = customerDao.findAll();

**return** list;

}

}

修改CustomerDaoImpl类，在该类上加@Repository注解。在queryRunner上加@Autowired注解，表示给该属性注入值。

@Repository("customerDao")

**public** **class** CustomerDaoImpl **implements** CustomerDao {

@Autowired

**private** QueryRunner queryRunner;

@Override

**public** List<Customer> findAll() {

List<Customer> list = **null**;

**try** {

list = queryRunner.query("select \* from cst\_customer", **new** BeanListHandler<Customer>(Customer.**class**));

} **catch** (SQLException e) {

e.printStackTrace();

}

**return** list;

}

}

## 第三步：修改spring配置文件

在applicationContext.xml中开启spring注解扫描。

注意：QueryRunner和DataSource这两个bean暂时没有办法用注解来配置，因为属于jar包里的bean.

<?xml version=*"1.0"* encoding=*"UTF-8"*?>

<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"*>

<context:component-scan base-package=*"cn.itcast"*></context:component-scan>

<bean id=*"queryRunner"* class=*"org.apache.commons.dbutils.QueryRunner"*>

<constructor-arg name=*"ds"* ref=*"dataSource"*></constructor-arg>

</bean>

<bean id=*"dataSource"* class=*"com.alibaba.druid.pool.DruidDataSource"*>

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

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

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

<property name=*"password"* value=*"123456"*></property>

</bean>

</beans>

再次运行TestFind中的test1方法，结果和xml的配置是一样的。

# Spring框架整合JUnit单元测试

## 目的

为了简化了JUnit的测试，使用Spring框架也可以整合测试。

## 具体步骤

要求：必须先有JUnit的环境（默认会使用Eclipse导入单元测试的环境）！！

### 步骤一：在程序中引入spring-test.jar的依赖

可以直接在spring4\_day02中引入spring-test.jar包的依赖，把spring4\_day02中的单元测试类全部改成spring的写法

<project xmlns=*"http://maven.apache.org/POM/4.0.0"* xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*

xsi:schemaLocation=*"http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd"*>

<modelVersion>4.0.0</modelVersion>

<parent>

<groupId>cn.itcast.parent</groupId>

<artifactId>itcast-parent</artifactId>

<version>0.0.1-SNAPSHOT</version>

</parent>

<groupId>cn.itcast</groupId>

<artifactId>spring4\_day01</artifactId>

<version>1.0.0-SNAPSHOT</version>

<dependencies>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-context</artifactId>

</dependency>

<dependency>

<groupId>org.slf4j</groupId>

<artifactId>slf4j-log4j12</artifactId>

</dependency>

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

</dependency>

**<dependency>**

**<groupId>org.springframework</groupId>**

**<artifactId>spring-test</artifactId>**

**</dependency>**

</dependencies>

</project>

### 步骤二：在具体测试类上添加注解

修改单元测试类TestIOC，在该类上添加@RunWith和@ContextConfiguration两个注解。我们需要测试的是UserService，在TestIOC中声明一个UserService的属性，并在该属性上添加@Autowired注解，为该属性注入值。修改后的TestIOC如下：

@RunWith(SpringJUnit4ClassRunner.**class**)

@ContextConfiguration("classpath:applicationContext.xml")

**public** **class** TestIOC {

@Autowired

**private** UserService userService;

@Test

**public** **void** test1(){

userService.saveUser();

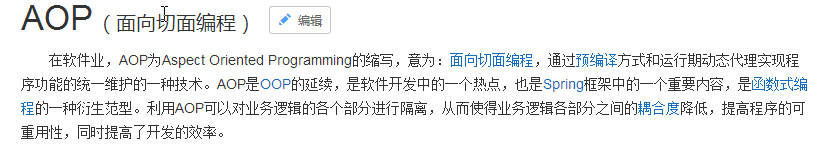
}

}

* @RunWith注解：这个是指定使用的单元测试执行类，这里就指定的是SpringJUnit4ClassRunner.class；
* @ContextConfiguration注解：这个指定spring配置文件所在的路径，可以同时指定多个文件；

# Spring的AOP编程

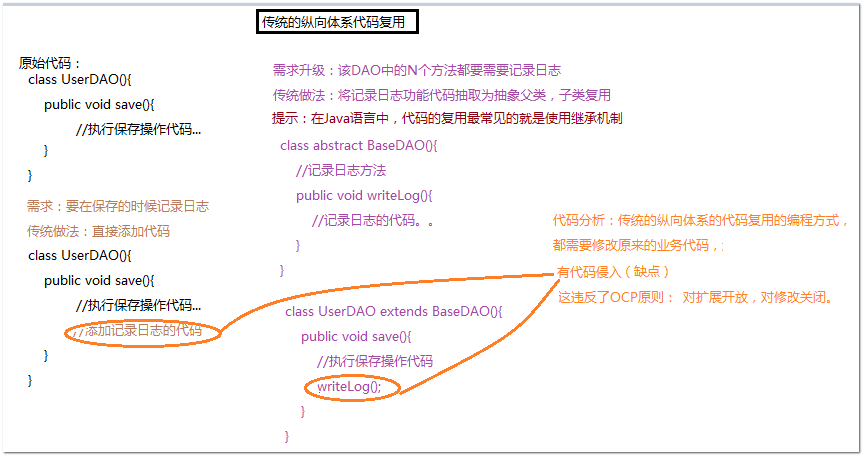
## AOP概述



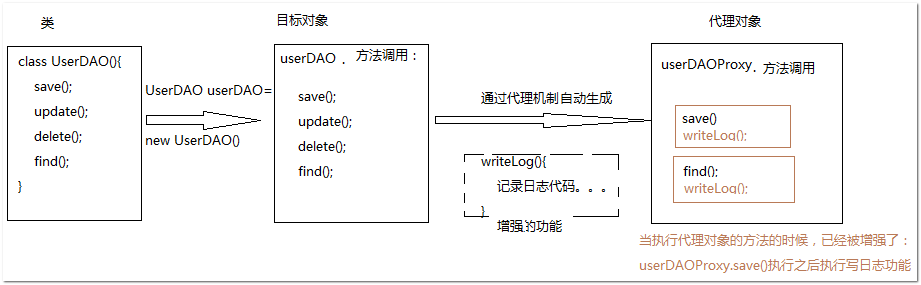
## AOP的作用

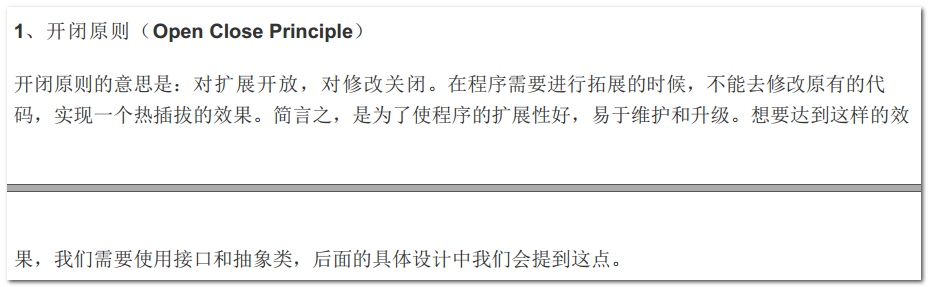
**AOP的作用：在不修改源代码的情况下，可以实现功能的增强。**

传统的纵向体系代码复用：



横向抽取机制（AOP思想）：





AOP 思想： 基于代理思想，对原来目标对象，创建代理对象，在不修改原对象代码情况下，通过代理对象，调用增强功能的代码，从而对原有业务方法进行增强 ！

## AOP的应用场景

场景一： 记录日志

场景二： 监控方法运行时间 （监控性能）

场景三： 权限控制

场景四： 缓存优化 （第一次调用查询数据库，将查询结果放入内存对象， 第二次调用， 直接从内存对象返回，不需要查询数据库 ）

场景五： 事务管理 （调用方法前开启事务， 调用方法后提交关闭事务 ）

## AOP的实现原理（了解）

那Spring中AOP是怎么实现的呢？Spring中AOP的有两种实现方式：

1. JDK动态代理
2. Cglib动态代理

### JDK动态代理

* 创建maven工程并引入依赖，pom.xml内容如下：

<project xmlns=*"http://maven.apache.org/POM/4.0.0"* xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"* xsi:schemaLocation=*"http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd"*>

<modelVersion>4.0.0</modelVersion>

<parent>

<groupId>cn.itcast.parent</groupId>

<artifactId>itcast-parent</artifactId>

<version>0.0.1-SNAPSHOT</version>

</parent>

<groupId>cn.itcast</groupId>

<artifactId>spring4\_day03\_myaop</artifactId>

<version>1.0.0-SNAPSHOT</version>

<dependencies>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-context</artifactId>

</dependency>

<dependency>

<groupId>org.slf4j</groupId>

<artifactId>slf4j-log4j12</artifactId>

</dependency>

<dependency>

<groupId>junit</groupId>

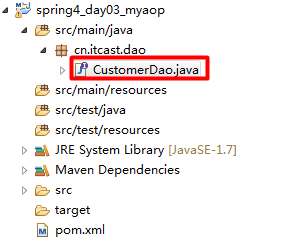
<artifactId>junit</artifactId>

</dependency>

</dependencies>

</project>

* 创建CustomerDao接口

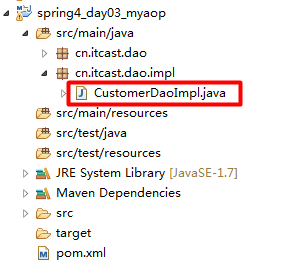


**public** **interface** CustomerDao {

**public** **void** save();

}

* 创建CustomerDao接口的实现类CustomerDaoImpl



**public** **class** CustomerDaoImpl **implements** CustomerDao {

@Override

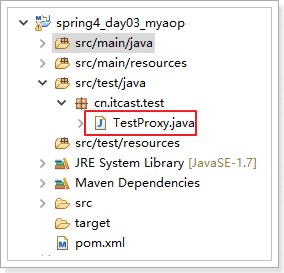
**public** **void** save() {

System.***out***.println("持久层：客户保存...");

}

}

* 创建TestProxy单元测试类，在其中创建test1单元测试方法：



@Test

**public** **void** test1() {

**final** CustomerDao customerDao = **new** CustomerDaoImpl();

/\*

\* newProxyInstance的三个参数解释：

\* 参数1：代理类的类加载器，同目标类的类加载器

\* 参数2：代理类要实现的接口列表，同目标类实现的接口列表

\* 参数3：回调，是一个InvocationHandler接口的实现类对象；当调用代理对象的方法时，其实执行的是回调中的invoke方法

\*/

CustomerDao proxy = (CustomerDao) Proxy.*newProxyInstance*(customerDao.getClass().getClassLoader(),

customerDao.getClass().getInterfaces(), **new** InvocationHandler() {

/\*

\* invoke方法的三个参数解释：

\* 参数1：代理对象

\* 参数2：目标方法，也就是被增强的方法，代理对象调用哪个方法，method就表示该方法

\* 参数3：目标方法的形参列表

\*/

@Override

**public** Object invoke(Object proxy, Method method, Object[] args) **throws** Throwable {

System.***out***.println("记录日志...");

//调用目标方法

Object result = method.invoke(customerDao, args);

**return** result;

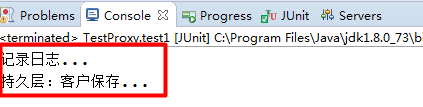
}

});

proxy.save();

}

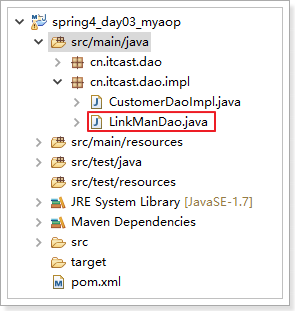
测试结果如下：



### Cglib动态代理（了解）

在实际开发中，可能需要对没有实现接口的类增强，用JDK动态代理的方式就没法实现。采用Cglib动态代理可以对没有实现接口的类产生代理，实际上是生成了目标类的子类来增强。

* 首先，需要导入Cglib所需的jar包。提示：spring已经集成了cglib，我们已经导入了spring包，故不需要再导入其它包了。
* 创建LinkManDao类,没有实现任何接口



**public** **class** LinkManDao {

**public** **void** save(){

System.***out***.println("持久层：联系人保存...");

}

}

* 在TestProxy中创建单元测试方法test2

@Test

**public** **void** test2() {

LinkManDao linkManDao = **new** LinkManDao();

// 创建cglib的核心对象

Enhancer enhancer = **new** Enhancer();

// 设置父类

enhancer.setSuperclass(linkManDao.getClass());

// 设置回调

enhancer.setCallback(**new** MethodInterceptor() {

/\*

\* 当你调用目标方法时，实质上是调用该方法

\* intercept四个参数：

\* proxy:代理对象

\* method:目标方法

\* args：目标方法的形参

\* methodProxy:代理方法

\*/

@Override

**public** Object intercept(Object proxy, Method method, Object[] args, MethodProxy methodProxy) **throws** Throwable {

//添加记录日志的功能

System.***out***.println("记录日志...");

//调用目标(父类)方法

// Object result = method.invoke(linkManDao, args);

Object result = methodProxy.invokeSuper(proxy, args);

**return** result ;

}

});

// 创建代理对象

LinkManDao proxy = (LinkManDao) enhancer.create();

proxy.save();

}

测试结果如下：

