Spring第二天

# 作业：使用 spring的 IoC的实现账户的 CRUD[掌握]

需求：

1.独立编写实体类，业务层和持久层代码。（此处不涉及表现层，创建 maven 的 jar 工程即可）

2.持久层技术采用 dbutils 实现。（不要用 Mybatis）

3.连接池（数据源）dbcp 或者 c3p0 均可。

4.要求使用 junit 测试代码。

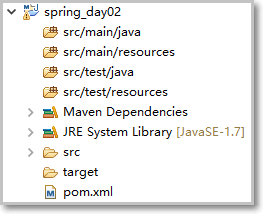
5.运用今天所学的 spring基于 xml的 ioc实现业务层和持久层解耦。

## 环境搭建

### 步骤一：导入父工程



### 步骤二：创建 maven 工程并导入坐标



pom.xml:

<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>

<groupId>cn.itcast.spring</groupId>

<artifactId>spring\_day02</artifactId>

<version>1.0.0-SNAPSHOT</version>

<parent>

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

<artifactId>itcast-parent</artifactId>

<version>0.0.1-SNAPSHOT</version>

</parent>

<dependencies>

<dependency>

<groupId>commons-dbutils</groupId>

<artifactId>commons-dbutils</artifactId>

</dependency>

<dependency>

<groupId>c3p0</groupId>

<artifactId>c3p0</artifactId>

</dependency>

<dependency>

<groupId>mysql</groupId>

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

</dependency>

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-context</artifactId>

</dependency>

</dependencies>

</project>

### 步骤三：创建数据库和实体类

create table account(

id int primary key auto\_increment,

name varchar(40),

money float

)character set utf8 collate utf8\_general\_ci;

insert into account(name,money) values('aaa',1000);

insert into account(name,money) values('bbb',1000);

insert into account(name,money) values('ccc',1000);

实体类

**public** **class** Account {

**private** Integer id;

**private** String name;

**private** Float money;

**public** Integer getId() {

**return** id;

}

**public** **void** setId(Integer id) {

**this**.id = id;

}

**public** String getName() {

**return** name;

}

**public** **void** setName(String name) {

**this**.name = name;

}

**public** Float getMoney() {

**return** money;

}

**public** **void** setMoney(Float money) {

**this**.money = money;

}

@Override

**public** String toString() {

**return** "Account [id=" + id + ", name=" + name + ", money=" + money

+ "]";

}

}

### 步骤四：创建service接口和实现类

接口：

**public** **interface** IAccountService {

/\*\*

\* 保存账户

\* **@param** account

\*/

**void** saveAccount(Account account);

/\*\*

\* 更新账户

\* **@param** account

\*/

**void** updateAccount(Account account);

/\*\*

\* 删除账户

\* **@param** accountId

\*/

**void** deleteAccountById(Integer accountId);

/\*\*

\* 根据id查询账户

\* **@param** accountId

\* **@return**

\*/

Account queryAccountById(Integer accountId);

/\*\*

\* 查询所有账户

\* **@return**

\*/

List<Account> queryAccountList();

}

实现类：

**public** **class** AccountServiceImpl **implements** IAccountService {

**private** IAccountDao accountDao;

**public** **void** setAccountDao(IAccountDao accountDao) {

**this**.accountDao = accountDao;

}

@Override

**public** **void** saveAccount(Account account) {

accountDao.saveAccount(account);

}

@Override

**public** **void** updateAccount(Account account) {

accountDao.updateAccount(account);

}

@Override

**public** **void** deleteAccountById(Integer accountId) {

accountDao.deleteAccountById(accountId);

}

@Override

**public** Account queryAccountById(Integer accountId) {

**return** accountDao.queryAccountById(accountId);

}

@Override

**public** List<Account> queryAccountList() {

**return** accountDao.queryAccountList();

}

}

### 步骤五：创建dao接口和实现类

接口：

**public** **interface** IAccountDao {

List<Account> queryAccountList();

Account queryAccountById(Integer accountId);

**void** deleteAccountById(Integer accountId);

**void** updateAccount(Account account);

**void** saveAccount(Account account);

}

实现类:

**public** **class** AccountDaoImpl **implements** IAccountDao {

**private** QueryRunner runner;

**public** **void** setRunner(QueryRunner runner) {

**this**.runner = runner;

}

@Override

**public** List<Account> queryAccountList() {

**try** {

**return** runner.query("select \* from account",

**new** BeanListHandler<Account>(Account.**class**));

} **catch** (SQLException e) {

**throw** **new** RuntimeException(e);

}

}

@Override

**public** Account queryAccountById(Integer accountId) {

**try** {

**return** runner.query("select \* from account where id = ?",

**new** BeanHandler<Account>(Account.**class**), accountId);

} **catch** (SQLException e) {

**throw** **new** RuntimeException(e);

}

}

@Override

**public** **void** deleteAccountById(Integer accountId) {

**try** {

runner.execute("delete from account where id = ?", accountId);

} **catch** (SQLException e) {

**throw** **new** RuntimeException(e);

}

}

@Override

**public** **void** updateAccount(Account account) {

**try** {

runner.update("update account set name=?,money=? where id=?",

account.getName(), account.getMoney(), account.getId());

} **catch** (SQLException e) {

**throw** **new** RuntimeException(e);

}

}

@Override

**public** **void** saveAccount(Account account) {

**try** {

runner.update("insert into account(name,money)values(?,?)",

account.getName(), account.getMoney());

} **catch** (SQLException e) {

**throw** **new** RuntimeException(e);

}

}

}

### 步骤六：创建配置文件

<?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"*>

<!-- 将accountService装配到容器中 -->

<bean id=*"accountService"* class=*"cn.itcast.service.impl.AccountServiceImpl"*>

<!-- 注入accountDao -->

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

</bean>

<!-- 将accountDao装配到容器中 -->

<bean id=*"accountDao"* class=*"cn.itcast.dao.impl.AccountDaoImpl"*>

<!-- 将queryRunner注入到到中 -->

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

</bean>

<!-- 将queryRunner装配到容器中 -->

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

<!-- 通过构造函数将dataSource注入到queryRunner中 -->

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

</bean>

<!-- 将dataSource装配到容器中 -->

<bean id=*"dataSource"* class=*"com.mchange.v2.c3p0.ComboPooledDataSource"*>

<!-- 注入连接属性 -->

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

<property name=*"jdbcUrl"* value=*"jdbc:mysql://localhost:3306/heima25"*></property>

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

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

</bean>

</beans>

### ※步骤七：创建测试用例

**public** **class** IAccountServiceTest {

@Test

**public** **void** testSaveAccount() {

// 加载配置文件，获取spring工厂，从容器中获取dao和service的实现类

ApplicationContext ac = **new** ClassPathXmlApplicationContext(

"applicationContext.xml");

// 从容器中获取service

IAccountService accountService = (IAccountService) ac

.getBean("accountService");

Account account = **new** Account();

account.setName("admin1");

account.setMoney(100f);

accountService.saveAccount(account);

}

@Test

**public** **void** testUpdateAccount() {

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

IAccountService accountService = (IAccountService) ac.getBean("accountService");

Account account = **new** Account();

account.setId(4);

account.setName("admin100");

account.setMoney(500f);

accountService.updateAccount(account);

}

@Test

**public** **void** testDeleteAccountById() {

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

IAccountService accountService = (IAccountService) ac.getBean("accountService");

accountService.deleteAccountById(4);

}

@Test

**public** **void** testQueryAccountById() {

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

IAccountService accountService = (IAccountService) ac.getBean("accountService");

Account account = accountService.queryAccountById(1);

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

}

@Test

**public** **void** testQueryAccountList() {

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

IAccountService accountService = (IAccountService) ac.getBean("accountService");

List<Account> list = accountService.queryAccountList();

**for** (Account account : list) {

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

}

}

}

### 步骤八：加载外部资源文件

在类的根路径下添加jdbc.properties:



内如如下:

jdbc.driverClass=com.mysql.jdbc.Driver

jdbc.jdbcUrl=jdbc:mysql://localhost:3306/heima25

jdbc.user=root

jdbc.password=root

修改applicationContext.xml:

<?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"*>

<!-- 加载外部资源文件 -->

<context:property-placeholder location=*"classpath:jdbc.properties"*/>

<!-- 将accountService装配到容器中 -->

<bean id=*"accountService"* class=*"cn.itcast.service.impl.AccountServiceImpl"*>

<!-- 注入accountDao -->

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

</bean>

<!-- 将accountDao装配到容器中 -->

<bean id=*"accountDao"* class=*"cn.itcast.dao.impl.AccountDaoImpl"*>

<!-- 将queryRunner注入到到中 -->

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

</bean>

<!-- 将queryRunner装配到容器中 -->

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

<!-- 通过构造函数将dataSource注入到queryRunner中 -->

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

</bean>

<!-- 将dataSource装配到容器中 -->

<bean id=*"dataSource"* class=*"com.mchange.v2.c3p0.ComboPooledDataSource"*>

<!-- 注入连接属性 -->

<property name=*"driverClass"* value=*"${jdbc.driverClass}"*></property>

<property name=*"jdbcUrl"* value=*"${jdbc.jdbcUrl}"*></property>

<property name=*"user"* value=*"${jdbc.user}"*></property>

<property name=*"password"* value=*"${jdbc.password}"*></property>

</bean>

</beans>

# 基于注解的IOC配置[掌握]

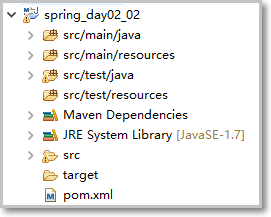
## 明确：写在最前

学习基于注解的IoC配置，大家脑海里首先得有一个认知，即注解配置和xml配置要实现的功能都是一样的，都是要降低程序间的耦合。只是配置的形式不一样。

关于实际的开发中到底使用xml还是注解，每家公司有不同的习惯。所以这两种配置方式我们都需要掌握。

## 环境搭建

### 步骤一：创建maven工程并导入坐标



pom.xml:

<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>

<groupId>cn.itcast.spring</groupId>

<artifactId>spring\_day02\_02</artifactId>

<version>1.0.0-SNAPSHOT</version>

<parent>

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

<artifactId>itcast-parent</artifactId>

<version>0.0.1-SNAPSHOT</version>

</parent>

<dependencies>

<dependency>

<groupId>commons-dbutils</groupId>

<artifactId>commons-dbutils</artifactId>

</dependency>

<dependency>

<groupId>c3p0</groupId>

<artifactId>c3p0</artifactId>

</dependency>

<dependency>

<groupId>mysql</groupId>

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

</dependency>

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-context</artifactId>

</dependency>

</dependencies>

</project>

### 步骤二：创建实体类

和之前的案例一致

### 步骤三：创建service接口和实现类

和之前的案例一致

### 步骤四：创建dao接口和实现类

和之前的案例一致

### 步骤五：使用@Component装配bean

装配service:

/\*\*

\* **@Component注解**：相当于配置了<bean>标签

\* value = "accountService":相当于配置了bean标签的id属性，单独配置value时，可以省略value。

\* **@author** zhang

\*

\*/

@Component("accountService")

**public** **class** AccountServiceImpl **implements** IAccountService {

**private** IAccountDao accountDao;

**public** **void** setAccountDao(IAccountDao accountDao) {

**this**.accountDao = accountDao;

}

@Override

**public** **void** saveAccount(Account account) {

accountDao.saveAccount(account);

}

@Override

**public** **void** updateAccount(Account account) {

accountDao.updateAccount(account);

}

@Override

**public** **void** deleteAccountById(Integer accountId) {

accountDao.deleteAccountById(accountId);

}

@Override

**public** Account queryAccountById(Integer accountId) {

**return** accountDao.queryAccountById(accountId);

}

@Override

**public** List<Account> queryAccountList() {

**return** accountDao.queryAccountList();

}

}

装配dao：

@Component("accountDao")

**public** **class** AccountDaoImpl **implements** IAccountDao {

**private** QueryRunner runner;

**public** **void** setRunner(QueryRunner runner) {

**this**.runner = runner;

}

@Override

**public** List<Account> queryAccountList() {

**try** {

**return** runner.query("select \* from account",

**new** BeanListHandler<Account>(Account.**class**));

} **catch** (SQLException e) {

**throw** **new** RuntimeException(e);

}

}

@Override

**public** Account queryAccountById(Integer accountId) {

**try** {

**return** runner.query("select \* from account where id = ?",

**new** BeanHandler<Account>(Account.**class**));

} **catch** (SQLException e) {

**throw** **new** RuntimeException(e);

}

}

@Override

**public** **void** deleteAccountById(Integer accountId) {

**try** {

runner.execute("delete from account where id = ?", accountId);

} **catch** (SQLException e) {

**throw** **new** RuntimeException(e);

}

}

@Override

**public** **void** updateAccount(Account account) {

**try** {

runner.update("update account set name=?,money=? where id=?",

account.getName(), account.getMoney(), account.getId());

} **catch** (SQLException e) {

**throw** **new** RuntimeException(e);

}

}

@Override

**public** **void** saveAccount(Account account) {

**try** {

runner.update("insert into account(name,money)values(?,?)",

account.getName(), account.getMoney());

} **catch** (SQLException e) {

**throw** **new** RuntimeException(e);

}

}

}

### 步骤六：创建配置文件



<?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"*>

<!-- 加载外部资源文件 -->

<context:property-placeholder location=*"classpath:jdbc.properties"*/>

<!-- 配置注解扫描的包：声明到指定的包下去进行扫描，如果发现类上有对应的注解，将其装配到容器中 -->

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

<!-- 装配dbutil -->

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

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

</bean>

<!-- 装配数据源 -->

<bean id=*"dataSource"* class=*"com.mchange.v2.c3p0.ComboPooledDataSource"*>

<property name=*"driverClass"* value=*"${jdbc.driverClass}"*></property>

<property name=*"jdbcUrl"* value=*"${jdbc.url}"*></property>

<property name=*"user"* value=*"${jdbc.username}"*></property>

<property name=*"password"* value=*"${jdbc.password}"*></property>

</bean>

</beans>

### 步骤七：进行测试

创建demo1和demo2的测试方法分别获取service的实现类和dao的实现类，发现@Component的注解已经生效。

但是如果使用其他的crud的测试方法，会报空指针异常。

原因:AccountDaoImpl并没有被注入到AccountServiceImpl中。

**public** **class** IAccountServiceTest {

@Test

**public** **void** demo1() {

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

// 从容器中获取service

IAccountService accountService = (IAccountService) ac.getBean("accountService");

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

}

@Test

**public** **void** demo2() {

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

// 从容器中获取service

IAccountDao accountDao = (IAccountDao) ac.getBean("accountDao");

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

}

@Test

**public** **void** testSaveAccount() {

// 加载配置文件，获取spring工厂，从容器中获取dao和service的实现类

ApplicationContext ac = **new** ClassPathXmlApplicationContext(

"applicationContext.xml");

// 从容器中获取service

IAccountService accountService = (IAccountService) ac

.getBean("accountService");

Account account = **new** Account();

account.setName("admin1");

account.setMoney(100f);

accountService.saveAccount(account);

}

@Test

**public** **void** testUpdateAccount() {

ApplicationContext ac = **new** ClassPathXmlApplicationContext(

"applicationContext.xml");

IAccountService accountService = (IAccountService) ac

.getBean("accountService");

Account account = **new** Account();

account.setId(4);

account.setName("admin100");

account.setMoney(500f);

accountService.updateAccount(account);

}

@Test

**public** **void** testDeleteAccountById() {

ApplicationContext ac = **new** ClassPathXmlApplicationContext(

"applicationContext.xml");

IAccountService accountService = (IAccountService) ac

.getBean("accountService");

accountService.deleteAccountById(4);

}

@Test

**public** **void** testQueryAccountById() {

ApplicationContext ac = **new** ClassPathXmlApplicationContext(

"applicationContext.xml");

IAccountService accountService = (IAccountService) ac

.getBean("accountService");

Account account = accountService.queryAccountById(1);

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

}

@Test

**public** **void** testQueryAccountList() {

ApplicationContext ac = **new** ClassPathXmlApplicationContext(

"applicationContext.xml");

IAccountService accountService = (IAccountService) ac

.getBean("accountService");

List<Account> list = accountService.queryAccountList();

**for** (Account account : list) {

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

}

}

}

#### 问题

虽然从spring中获取service的实现类和dao的实现类都没有问题，但是在测试crud的方式时，发现会报空指针异常，原因是service的实现类中调用了dao的实现类，但是dao的实现类并没有被注入到service的实现类中，因此造成了空指针异常。

解决方案：使用注解注入。

## 常用注解

### 用于创建对象的

**相当于：<bean id="" class="">**

#### @Component

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。

细节：如果注解中有且只有一个属性要赋值时，且名称是**value**，**value**在赋值时可以不写。

### 用于注入数据的

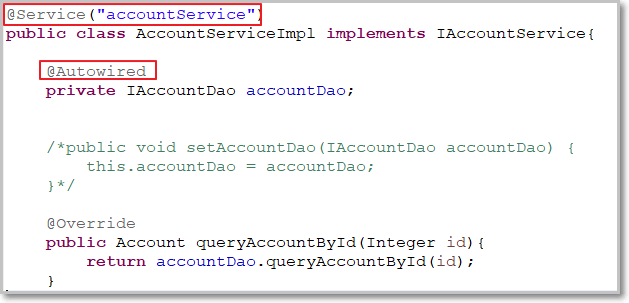
**相当于： <property name="" ref=""> <property name="" value="">**

#### @Autowired

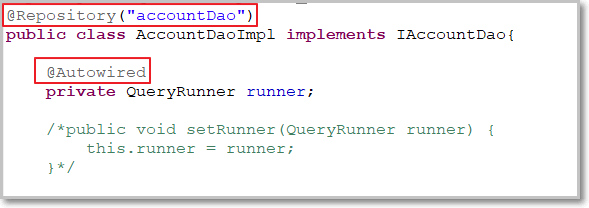
作用：

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，使用要注入的对象的变量名称作为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。

service实现类



dao实现类：



#### @Qualifier

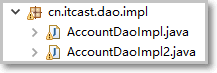
作用：

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。

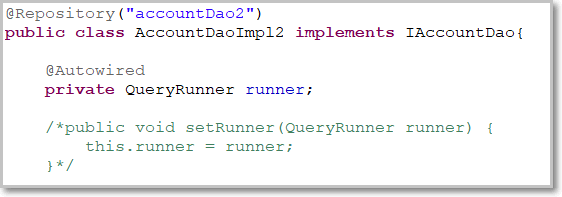
属性：

value：指定bean的id。

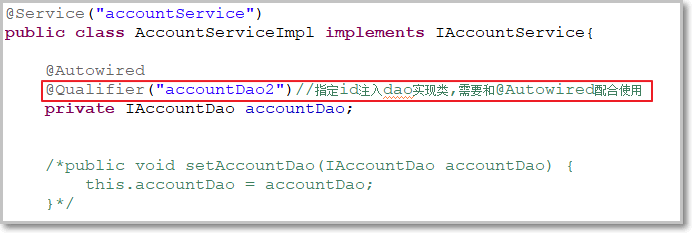
添加第二个dao实现类：



内容如下：复制AccountDaoImpl后重命名为AccountDaoImpl2即可



指定注入AccountDaoImpl2:



#### @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。



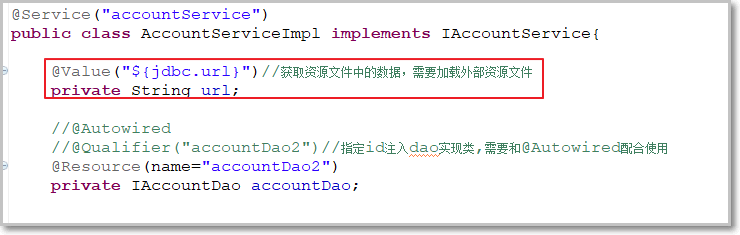
#### @Value

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



### 用于改变作用范围的

**相当于：<bean id="" class="" scope="">**

#### @Scope

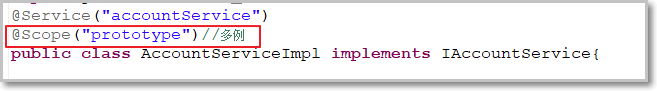
作用：

指定bean的作用范围。

属性：

value：指定范围的值。

取值：singleton prototype request session globalsession



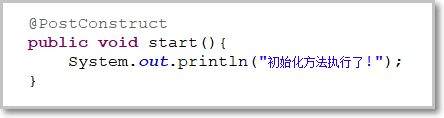
### 和生命周期相关的：(了解)

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

#### @PostConstruct

作用：

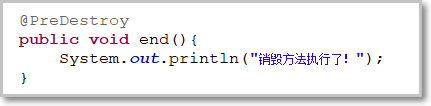
用于指定初始化方法。



#### ※@PreDestroy

作用：

用于指定销毁方法。



### 关于Spring注解和XML的选择问题

注解的优势：

配置简单，维护方便（我们找到类，就相当于找到了对应的配置）。

XML的优势：

修改时，不用改源码。

Spring管理Bean方式的比较：



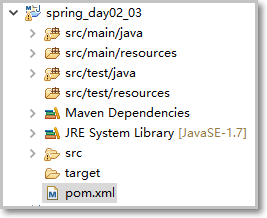
## spring管理对象细节

基于注解的spring IoC配置中，bean对象的特点和基于XML配置是一模一样的。

# 案例：使用注解IOC改造作业[掌握]

## Xml混合注解配置

### 步骤一：创建maven工程并导入坐标



pom.xml:

<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>

<groupId>cn.itcast.spring</groupId>

<artifactId>spring\_day02\_03</artifactId>

<version>1.0.0-SNAPSHOT</version>

<parent>

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

<artifactId>itcast-parent</artifactId>

<version>0.0.1-SNAPSHOT</version>

</parent>

<dependencies>

<dependency>

<groupId>commons-dbutils</groupId>

<artifactId>commons-dbutils</artifactId>

</dependency>

<dependency>

<groupId>c3p0</groupId>

<artifactId>c3p0</artifactId>

</dependency>

<dependency>

<groupId>mysql</groupId>

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

</dependency>

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-context</artifactId>

</dependency>

</dependencies>

</project>

### 步骤二：创建实体类

使用之前的数据库，因此实体类直接从之前的案例中复制即可

### 步骤三：创建web层

创建一个AccountController，假设属于web层，只需测试新增方法即可。

@Controller("accountController")

**public** **class** AccountController {

@Autowired

**private** IAccountService accountService;

**public** **void** saveAccuont(Account account){

accountService.saveAccount(account);

}

}

### 步骤四：创建service接口和实现类

复制之前的接口和实现类，然后在实现类修改注入方式即可。

接口：

**public** **interface** IAccountService {

/\*\*

\* 保存账户

\* **@param** account

\*/

**void** saveAccount(Account account);

/\*\*

\* 更新账户

\* **@param** account

\*/

**void** updateAccount(Account account);

/\*\*

\* 删除账户

\* **@param** accountId

\*/

**void** deleteAccountById(Integer accountId);

/\*\*

\* 根据id查询账户

\* **@param** accountId

\* **@return**

\*/

Account queryAccountById(Integer accountId);

/\*\*

\* 查询所有账户

\* **@return**

\*/

List<Account> queryAccountList();

}

实现类：

@Component("accountService")

**public** **class** AccountServiceImpl **implements** IAccountService {

@Autowired

**private** IAccountDao accountDao;

@Override

**public** **void** saveAccount(Account account) {

accountDao.saveAccount(account);

}

@Override

**public** **void** updateAccount(Account account) {

accountDao.updateAccount(account);

}

@Override

**public** **void** deleteAccountById(Integer accountId) {

accountDao.deleteAccountById(accountId);

}

@Override

**public** Account queryAccountById(Integer accountId) {

**return** accountDao.queryAccountById(accountId);

}

@Override

**public** List<Account> queryAccountList() {

**return** accountDao.queryAccountList();

}

}

### 步骤五：创建dao接口和实现类

接口：

**public** **interface** IAccountDao {

List<Account> queryAccountList();

Account queryAccountById(Integer accountId);

**void** deleteAccountById(Integer accountId);

**void** updateAccount(Account account);

**void** saveAccount(Account account);

}

实现类：

@Component("accountDao")

**public** **class** AccountDaoImpl **implements** IAccountDao {

@Autowired

**private** QueryRunner runner;

@Override

**public** List<Account> queryAccountList() {

**try** {

**return** runner.query("select \* from account",

**new** BeanListHandler<Account>(Account.**class**));

} **catch** (SQLException e) {

**throw** **new** RuntimeException(e);

}

}

@Override

**public** Account queryAccountById(Integer accountId) {

**try** {

**return** runner.query("select \* from account where id = ?",

**new** BeanHandler<Account>(Account.**class**));

} **catch** (SQLException e) {

**throw** **new** RuntimeException(e);

}

}

@Override

**public** **void** deleteAccountById(Integer accountId) {

**try** {

runner.execute("delete from account where id = ?", accountId);

} **catch** (SQLException e) {

**throw** **new** RuntimeException(e);

}

}

@Override

**public** **void** updateAccount(Account account) {

**try** {

runner.update("update account set name=?,money=? where id=?",

account.getName(), account.getMoney(), account.getId());

} **catch** (SQLException e) {

**throw** **new** RuntimeException(e);

}

}

@Override

**public** **void** saveAccount(Account account) {

**try** {

runner.update("insert into account(name,money)values(?,?)",

account.getName(), account.getMoney());

} **catch** (SQLException e) {

**throw** **new** RuntimeException(e);

}

}

}

### 步骤六：创建配置文件

<?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"*>

<!-- 配置注解扫描的包 -->

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

<!-- 将queryRunner装配到容器中 -->

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

<!-- 通过构造函数将dataSource注入到queryRunner中 -->

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

</bean>

<!-- 将dataSource装配到容器中 -->

<bean id=*"dataSource"* class=*"com.mchange.v2.c3p0.ComboPooledDataSource"*>

<!-- 注入连接属性 -->

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

<property name=*"jdbcUrl"* value=*"jdbc:mysql://localhost:3306/heima25"*></property>

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

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

</bean>

</beans>

### 步骤七：创建测试用例

创建web层的测试用例：

**public** **class** AccountControllerTest {

@Test

**public** **void** testSaveAccuont() {

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

AccountController accountController = (AccountController) ac.getBean("accountController");

Account account = **new** Account();

account.setName("admin1");

account.setMoney(5000f);

accountController.saveAccuont(account);

}

}

## Spring的纯注解配置

写到此处，基于注解的IoC配置已经完成，但是大家都发现了一个问题：我们依然离不开spring的xml配置文件，那么能不能不写这个bean.xml，所有配置都用注解来实现呢？

当然，同学们也需要注意一下，我们选择哪种配置的原则是简化开发和配置方便，而非追求某种技术。

### 待改造的问题

我们发现，之所以我们现在离不开xml配置文件，是因为我们有一句很关键的配置：

<!-- 告知spring框架在，读取配置文件，创建容器时，扫描注解，依据注解创建对象，并存入容器中 -->

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

如果他要也能用注解配置，那么我们就离脱离xml文件又进了一步。

另外，数据源和QueryRunner的配置也需要靠注解来实现。

<!-- 将queryRunner装配到容器中 -->

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

<!-- 通过构造函数将dataSource注入到queryRunner中 -->

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

</bean>

<!-- 将dataSource装配到容器中 -->

<bean id=*"dataSource"* class=*"com.mchange.v2.c3p0.ComboPooledDataSource"*>

<!-- 注入连接属性 -->

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

<property name=*"jdbcUrl"* value=*"jdbc:mysql://localhost:3306/heima25"*></property>

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

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

</bean>

### 新注解说明

#### @Configuration

作用：

用于指定当前类是一个spring配置类，当创建容器时会从该类上加载注解。

获取容器时需要使用AnnotationApplicationContext(有@Configuration注解的类.class)。

示例代码：

/\*\*

\* spring的配置类，相当于applicationContext.xml文件

\* **@author** zhang

\*

\*/

@Configuration

**public** **class** SpringConfiguration {

}

注意：

我们已经把配置文件用类来代替了，但是如何配置创建容器时要扫描的包呢？

请看下一个注解。

#### @ComponentScan

作用：

用于指定spring在初始化容器时要扫描的包。作用和在spring的xml配置文件中的：

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

属性：

Value（单独使用可省略）：用于指定要扫描的包。和标签中的basePackages属性作用一样。

示例代码：

/\*\*

\* spring的配置类，相当于bean.xml文件

\* **@author** zhang

\*

\*/

@Configuration

@ComponentScan("cn.itcast")

**public** **class** SpringConfiguration {

}

注意：

我们已经配置好了要扫描的包，但是数据源和QueryRuner对象如何从配置文件中移除呢？

请看下一个注解。

#### @Bean

作用：

该注解只能写在方法上，将方法的返回值作为一个bean,并且放入spring容器。

属性：

name：给当前@Bean注解方法创建的对象指定一个名称(即bean的id）。

示例代码：

@Bean(name="dataSource")

**public** DataSource createDataSource() **throws** Exception{

ComboPooledDataSource ds = **new** ComboPooledDataSource();

ds.setDriverClass("com.mysql.jdbc.Driver");

ds.setJdbcUrl("jdbc:mysql://localhost:3306/heima-26");

ds.setUser("root");

ds.setPassword("root");

**return** ds;

}

注意**:**

我们已经把数据源和QueryRunner从配置文件中移除了，此时可以删除bean.xml了。

但是由于没有了配置文件，创建数据源的配置又都写死在类中了。如何把它们配置出来呢？

请看下一个注解。

#### @PropertySource

作用：

用于加载.properties文件中的配置。例如我们配置数据源时，可以把连接数据库的信息写到properties配置文件中，就可以使用此注解指定properties配置文件的位置。

属性：

value[]：用于指定properties文件位置。如果是在类路径下，需要写上classpath:

示例代码：

@PropertySource(value = { "classpath:jdbc.properties" })

**public** **class** JdbcConfig {

@Value("${jdbc.driver}")

**private** String driverClass;

@Value("${jdbc.url}")

**private** String url;

@Value("${jdbc.username}")

**private** String username;

@Value("${jdbc.password}")

**private** String password;

@Bean(name = "dataSource")

**public** DataSource createDataSource() **throws** Exception {

ComboPooledDataSource ds = **new** ComboPooledDataSource();

ds.setDriverClass(driverClass);

ds.setJdbcUrl(url);

ds.setUser(username);

ds.setPassword(password);

**return** ds;

}

}

#### @Import

作用：

用于导入其他配置类，在引入其他配置类时，其他类上可以不用再写@Configuration注解。当然，写上也没问题。

属性：

value[]：用于指定其他配置类的字节码。

示例代码：

@Configuration

@ComponentScan("cn.itcast")

@Import(value = { JdbcConfig.**class** })

**public** **class** SpringConfiguration {

}

@Configuration//写不写都行

@PropertySource(value = { "classpath:jdbc.properties" })

**public** **class** JdbcConfig {

}

注意：

我们已经把要配置的都配置好了，但是新的问题产生了，由于没有配置文件了，如何获取容器呢？

请看下一小节。

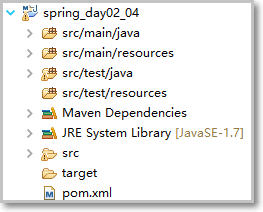
#### 通过注解获取容器

ApplicationContext ac = **new** AnnotationConfigApplicationContext(SpringConfiguration.**class**);

## 纯注解案例

### 步骤一：环境搭建

复制之前的工程重命名后修改即可



将pom.xml中的artifactId修改成spring\_day02\_04

### 步骤二：创建配置类

@Configuration//声明这是个配置类，用来代替xml

@ComponentScan("cn.itcast")//用来代替xml中的注解扫描包

**public** **class** SpringConfiguration {

@Bean(name="dataSource")

**public** DataSource createDataSource() **throws** Exception{

ComboPooledDataSource ds = **new** ComboPooledDataSource();

ds.setDriverClass("com.mysql.jdbc.Driver");

ds.setJdbcUrl("jdbc:mysql://localhost:3306/heima25");

ds.setUser("root");

ds.setPassword("root");

**return** ds;

}

@Bean(name="queryRunner")

**public** QueryRunner createQueryRunner(@Qualifier("dataSource")DataSource dataSource){

**return** **new** QueryRunner(dataSource);

}

}

### ※步骤三：测试

@Test

**public** **void** demo() {

ApplicationContext ac = **new** AnnotationConfigApplicationContext(SpringConfiguration.**class**);

AccountController accountController = (AccountController) ac.getBean("accountController");

Account account = **new** Account();

account.setName("admin3");

account.setMoney(5000f);

accountController.saveAccuont(account);

}

### 优化一

如果在一个配置类中需要装配很多的bean到容器中，那么显然将所有的bean放在一个配置类是非常不利于管理的，因此我们可以选择将bean分拆到其他类中单独管理，最后再通过配置类来引入其他类即可。

@Configuration//声明这是个配置类，用来代替xml

@ComponentScan("cn.itcast")//用来代替xml中的注解扫描包

@Import(value = { JdbcConfig.**class** })

**public** **class** SpringConfiguration {

}

**public** **class** JdbcConfig {

@Bean(name="dataSource")

**public** DataSource createDataSource() **throws** Exception{

ComboPooledDataSource ds = **new** ComboPooledDataSource();

ds.setDriverClass("com.mysql.jdbc.Driver");

ds.setJdbcUrl("jdbc:mysql://localhost:3306/heima25");

ds.setUser("root");

ds.setPassword("root");

**return** ds;

}

@Bean(name="queryRunner")

**public** QueryRunner createQueryRunner(@Qualifier("dataSource")DataSource dataSource){

**return** **new** QueryRunner(dataSource);

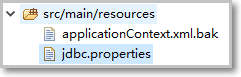
}

}

### 优化二

dataSource的连接数据硬编码了，可以通过读取properties外部资源文件来获取连接数据数据。

在resources类的根目录下创建properties资源文件：



jdbc.driverClass=com.mysql.jdbc.Driver

jdbc.url=jdbc:mysql://localhost:3306/heima25

jdbc.username=root

jdbc.password=root

@Configuration//声明这是个配置类，用来代替xml

@ComponentScan("cn.itcast")//用来代替xml中的注解扫描包

@Import(value = { JdbcConfig.**class** })

**public** **class** SpringConfiguration {

}

@PropertySource(value={"classpath:jdbc.properties"})

**public** **class** JdbcConfig {

@Value("${jdbc.driverClass}")

**private** String driverClass;

@Value("${jdbc.url}")

**private** String url;

@Value("${jdbc.username}")

**private** String username;

@Value("${jdbc.password}")

**private** String password;

@Bean(name="dataSource")

**public** DataSource createDataSource() **throws** Exception{

ComboPooledDataSource ds = **new** ComboPooledDataSource();

ds.setDriverClass("com.mysql.jdbc.Driver");

ds.setJdbcUrl("jdbc:mysql://localhost:3306/heima25");

ds.setUser("root");

ds.setPassword("root");

**return** ds;

}

@Bean(name="queryRunner")

**public** QueryRunner createQueryRunner(@Qualifier("dataSource")DataSource dataSource){

**return** **new** QueryRunner(dataSource);

}

}

# Spring整合junit

## 测试类中的问题和解决思路

### 问题

在测试类中，每个测试方法都有以下两行代码：

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

AccountController accountController = (AccountController) ac.getBean("accountController");

这两行代码的作用是获取容器，如果不写的话，直接会提示空指针异常。所以又不能轻易删掉。

### 解决思路分析

针对上述问题，我们需要的是程序能自动帮我们创建容器。一旦程序能自动为我们创建spring容器，我们就无须手动创建了，问题也就解决了。

我们都使用过junit，但是junit都无法知晓我们是否使用了spring框架，更不用说帮我们创建spring容器了。不过好在，junit给我们暴露了一个注解，可以让我们替换掉它的运行器。

这时，我们需要依靠spring框架，因为它提供了一个运行器，可以读取配置文件（或注解）来创建容器。我们只需要告诉它配置文件在哪就行了。

### 涉及的注解

#### @RunWith

作用：

替换掉junit的运行器,换成一个可以初始化spring容器的运行器。

属性：

value：单独配置时，value属性名称可以省略，配置SpringJUnit4ClassRunner.class来代替原来junit的运行器

#### @ContextConfiguration

作用：

加载配置类或者xml配置文件

属性：

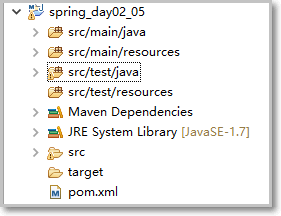
value[]：用来指定xml配置文件的路径

class[]: 用来指定配置类

## Xml的配置步骤

### 步骤一：环境搭建

复制之前的spring\_day02\_03工程重命名后修改即可



将pom.xml中的artifactId修改成spring\_day02\_05

### 步骤二：导入spring-test的坐标

此处需要注意的是，**spring5**及以上版本要求**junit**的版本必须是**4.12**及以上，否则用不了。

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-test</artifactId>

</dependency>

### 步骤三：测试

@RunWith(SpringJUnit4ClassRunner.**class**)

@ContextConfiguration(locations={"classpath:applicationContext.xml"})

**public** **class** AccountControllerTest {

@Autowired

**private** AccountController accountController;

@Test

**public** **void** testSaveAccount(){

Account account = **new** Account();

account.setName("admin12");

account.setMoney(2000f);

accountController.saveAccount(account);

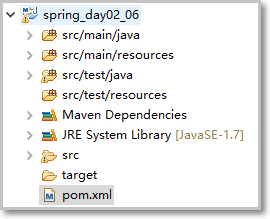
}

}

## 纯注解的配置步骤(配置类)

### 步骤一：环境搭建

复制spring\_day02\_04重命名即可。



将pom.xml中的artifactId修改成spring\_day02\_06

### 步骤二：导入spring-test的坐标

此处需要注意的是，**spring5**及以上版本要求**junit**的版本必须是**4.12**及以上，否则用不了。

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-test</artifactId>

</dependency>

### 步骤三：测试

@RunWith(SpringJUnit4ClassRunner.**class**)

@ContextConfiguration(classes={SpringConfiguration.**class**})

**public** **class** AccountControllerTest {

@Autowired

**private** AccountController accountController;

@Test

**public** **void** testSaveAccount(){

Account account = **new** Account();

account.setName("admin12");

account.setMoney(2000f);

accountController.saveAccount(account);

}

}

# 总结

## 装配bean的注解

@Component：

将该注解下的类装配到spring容器中，value属性是用来指定bean的id的，如果不配置默认使用类名的首字母小写。

使用前提：开启包的注解扫描

@Component的三个衍生注解：

1、@Controller：针对web层的

2、@Service:针对service层的

3、@Repository:针对dao层的

## 注入数据的注解

@Autowired:自动按照bean的类型注入，只能注入bean类型，如果有多个类型匹配，默认使用属性名称作为bean的id.

@Qualifire:一般和@Autowired配合使用，当有多个类型匹配时，可以指定使用id注入，还可以单独在方法的参数中指定id注入bean

@Resource:指定id注入bean

@Value：只能注入基本类型或者String类型，一般用来获取资源文件中的数据，前提是需要先加载外部资源文件

## 作用域相关的注解

@Scope:可以指定bean是单例还是多例

## 生命周期相关的

@PostConstruct:指定某个方法为初始化方法

@PreDestory:指定某个方法为销毁方法

## 配置类相关的

@Configuration:声明当前的类是一个配置类，用来代替xml配置的

@ComponentScan:开启注解扫描的

@Import:引入其他配置类

@PropertySource:加载外部资源文件的

@Bean:将方法返回的bean类型的对象装配到容器中。Name属性就相当于bean的id,如果不配置，默认使用方法名。

## Spring整合Junti相关的

@RunWith:替换底层的运行器，可以换成可以加载和初始化spring容器的运行器

@ContextConfiguration:加载配置类或者xml配置文件的。

使用上面的两个注解，需要导入spring-test的依赖

# 作业

1、使用注解改造昨天的作业

要求：两个版本

第一个版本：xml和注解组合使用

第二个版本：纯注解配置

2、在业务层添加一个转账方法

实现账户名称aaa给账户名称bbb转账功能。

在转账方法中模拟转账异常，并分析问题产生的原因。

解决转账异常带来的事务问题（不满足事务的一致性）