**Spring第二天**

上海千锋 文威

**课程目标：**

* Spring中ioc的常用注解(重点掌握)
* 案例：使用xml方式和注解方式实现单表的crud操作(重点掌握)
* 改造基于注解的ioc案例，使用纯注解的方式实现(重点掌握)
* Spring和junit的整合(重点掌握)

**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（只是形式不一样），都是要降低程序间的耦合。关于实际的开发中到底使用xml还是注解，每家公司有着不同的使用习惯，所以这两种配置方式我们都需要掌握。

## 1.1 IOC注解

### 1.1.1 项目环境搭建

* dao部分

|  |
| --- |
| **public interface** AccountDao {  **public void** addAccount(); } |
| **public class** AccountDaoImpl **implements** AccountDao {  **public void** addAccount() {  System.***out***.println(**"新增账户的方法被实现了...."**);  } } |

* service部分

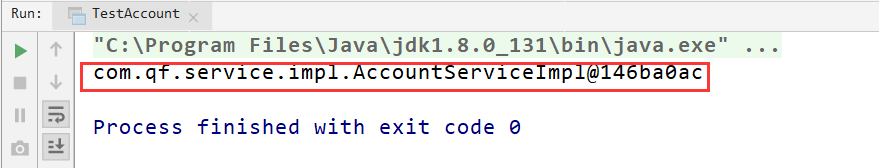
|  |
| --- |
| **public interface** AccountService {  **public void** addAccount(); } |
| **public class** AccountServiceImpl **implements** AccountService {  AccountDao **accountDao**;  **public void** addAccount() {  **accountDao**.addAccount();  } } |

* 配置文件部分

|  |
| --- |
| <**bean id="accountDao" class="com.qf.dao.impl.AccountDaoImpl"**></**bean**> <**bean id="accountService" class="com.qf.service.impl.AccountServiceImpl"**></**bean**> |

* 测试部分

|  |
| --- |
| **public class** TestAccount {  **public static void** main(String[] args) {  ApplicationContext context = **new** ClassPathXmlApplicationContext(**"bean.xml"**);  AccountService accountService = (AccountService) context.getBean(**"accountService"**);  System.***out***.println(accountService);  } } |



### 1.1.2 使用IOC注解来改造项目

曾经XML的配置：

|  |
| --- |
| <bean id="accountService" class="com.qf.service.impl.AccountServiceImpl"  scope="" init-method="" destroy-method="">  <property name="" value="" | ref=""></property>  </bean> |

通过这段配置，我们可以把注解分成以下几类：

* 用于创建对象的：

作用：他们的作用就和在XML配置文件中编写一个<bean>标签实现的功能是一样的。

* 用于注入数据的：

作用：他们的作用就和在xml配置文件中的bean标签中写一个<property>标签的作用是一样的。

* 用于改变作用范围的：

作用：他们的作用就和在bean标签中使用scope属性实现的功能是一样的。

* 和生命周期相关的：

作用：他们的作用就和在bean标签中使用init-method和destroy-methode的作用是一样的。

#### 1.1.2.1 用于创建对象的注解

使用创建对象的注解效果跟在XML配置文件中编写一个<bean>标签实现的功能是一样的。

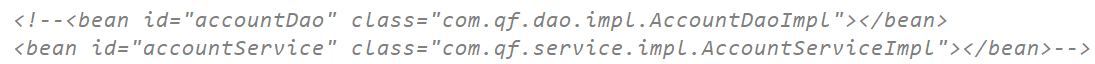
@Component:

用法：定义在类名上。

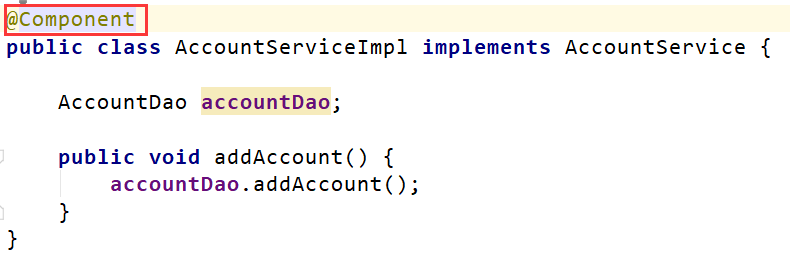
作用：用于把当前类对象存入spring容器中。

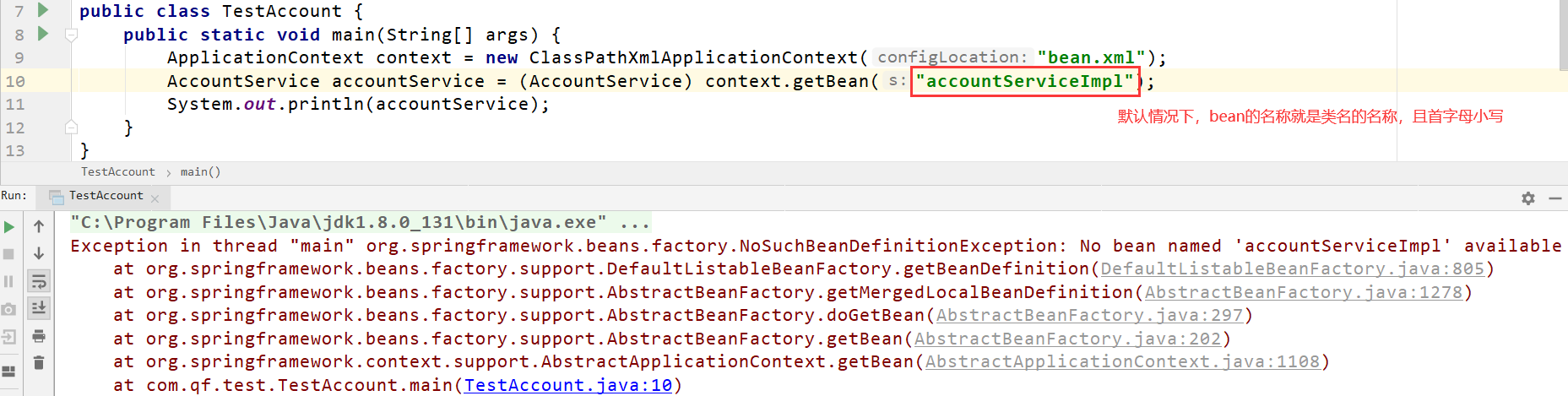
属性：value：用于指定bean的id。当我们不写时，它的默认值是当前类名，且首字母改小写。

此时我们把bean.xml里面的配置信息注释掉。



尝试使用@Component注解，不加任何参数，默认就是首字母改小写的类名。





此时发现报错。

原因：

ClassPathXmlApplicationContext初始化容器时会把bean.xml中声明的标签下的类都创建到容器中。但是因为bean.xml并不知道哪一个类加上注解了，所以无法创建对象.

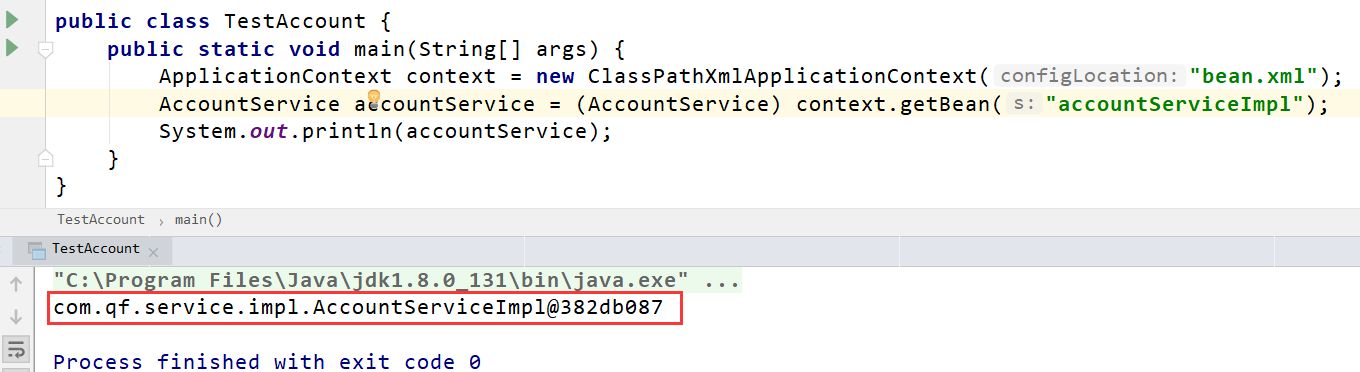
我们要在bean.xml告知spring在创建容器时要扫描的包。

|  |
| --- |
| <context:component-scan base-package="com.qf"></context:component-scan> |

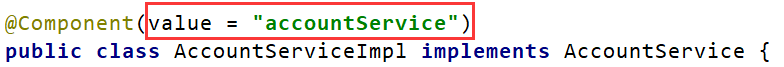
此时稍作修改：



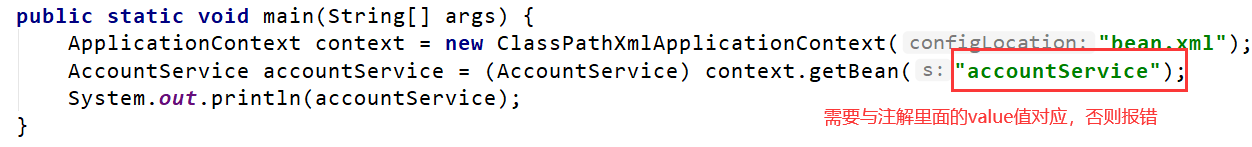
最后测试如下:



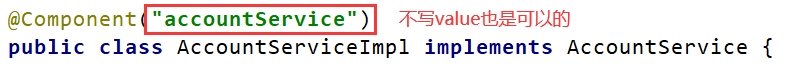
其实我们也可以自定义value属性的值



那么测试类需要修改一下:



当然注解只有一个参数，且参数为value时可以使用省略写法.



#### 1.1.2.2 由Component衍生的注解

@Controller：一般用在表现层。

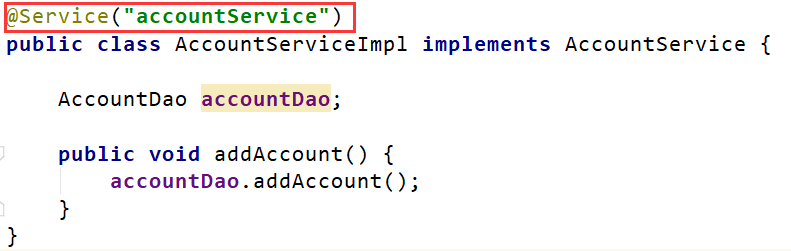
@Service：一般用在业务。

@Repository：一般用在持久层。

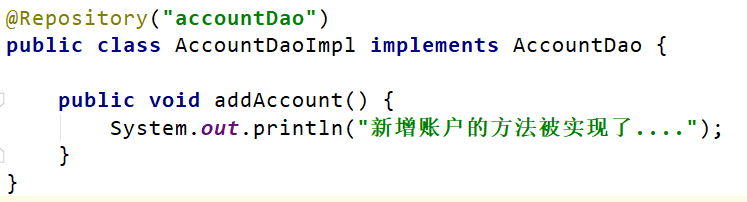
以上三个注解他们的作用和属性与@Component是一模一样，他们三个是spring框架为我们提供明确的三层使用的注解，使我们的三层对象更加清晰。

我们接下来改写业务层的注解，让它更规范，并且增加持久层的注解。

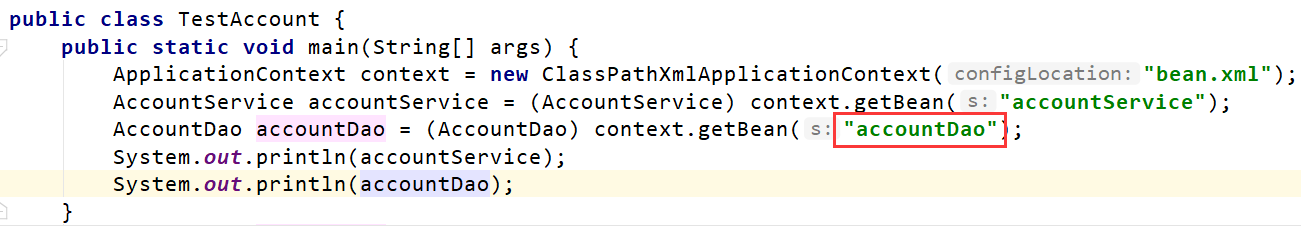
* 业务层

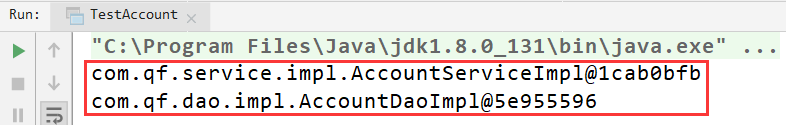


* 持久层



* 测试效果

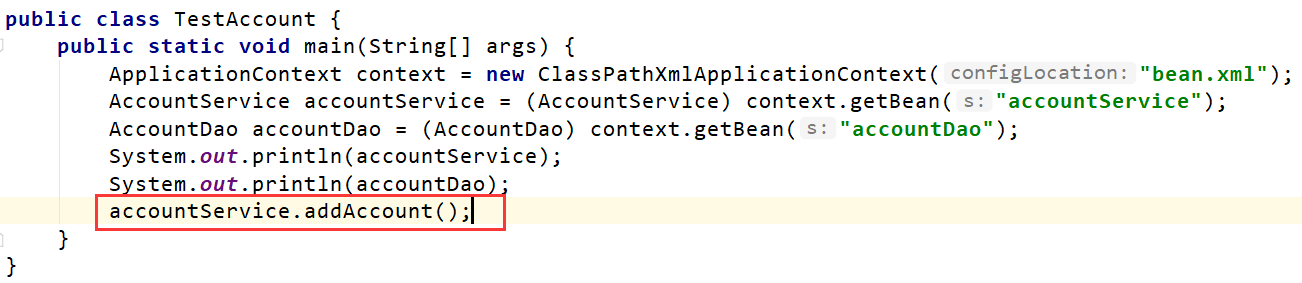




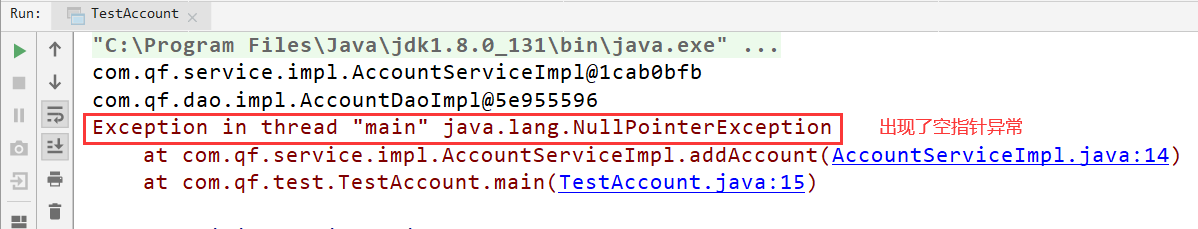
#### 1.1.2.3 自动按照类型注入

现在我们打开accountDao.addAccount()的方法。

如下:

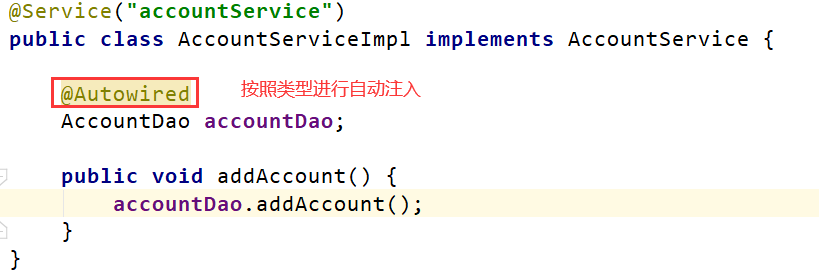


测试结果如下:

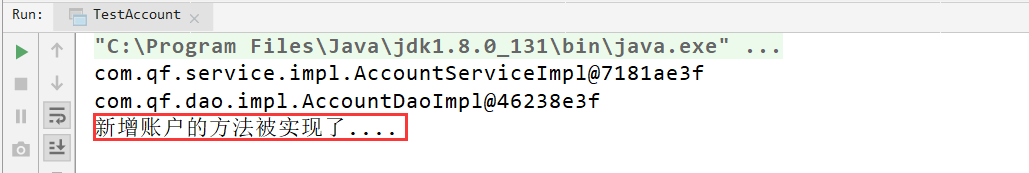


解决方案:

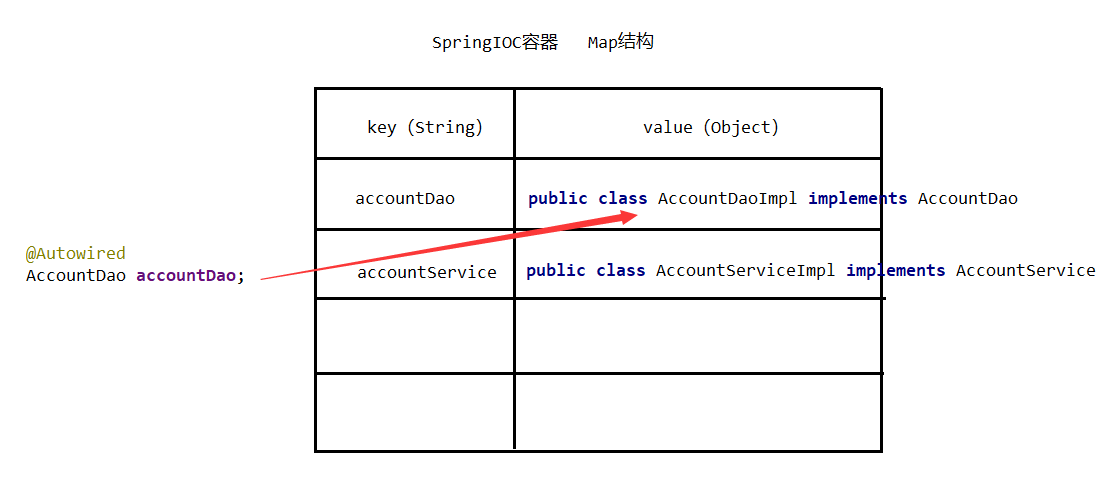
可以使用@Autowired注解实现自动按照类型注入，如下：



然后再观察测试结果:



那么这个原理是什么?



容器中有唯一的一个bean对象类型和要注入的变量类型匹配，就可以注入成功。

详细过程：首先会找到实现类，但是因为实现类继承了接口，所以可以把实现类看作接口（多态）。

那么如果ioc容器中有多个类型匹配呢？

我们来做一个实验:

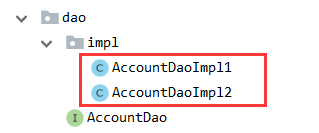
* accountDaoImpl1

|  |
| --- |
| @Repository(**"accountDao1"**) **public class** AccountDaoImpl1 **implements** AccountDao {  **public void** addAccount() {  System.***out***.println(**"新增账户的方法111被实现了...."**);  } } |

* accountDaoImpl2

|  |
| --- |
| @Repository(**"accountDao2"**) **public class** AccountDaoImpl2 **implements** AccountDao {  **public void** addAccount() {  System.***out***.println(**"新增账户的方法222被实现了...."**);  } } |

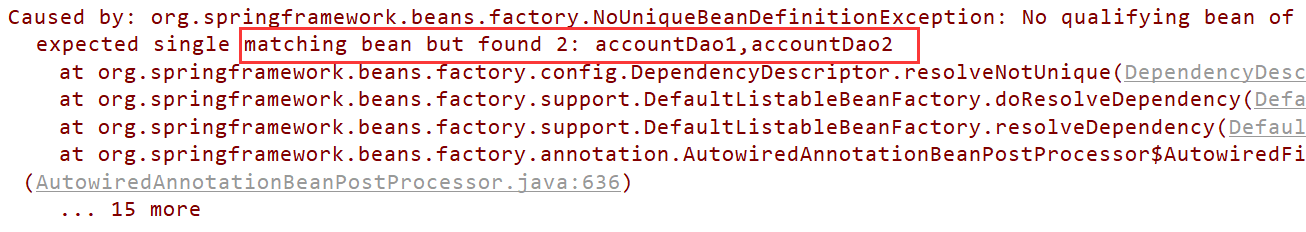
改造之后:



* 业务层代码不变

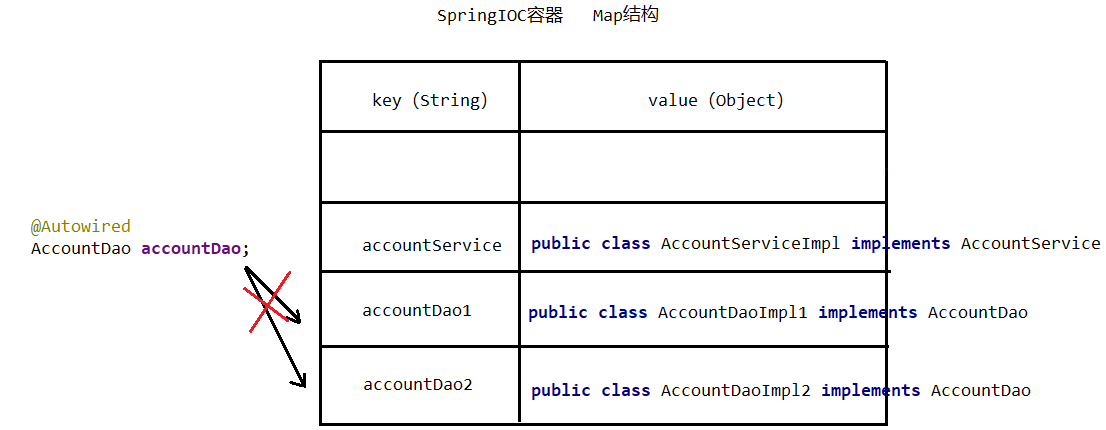
|  |
| --- |
| @Service(**"accountService"**) **public class** AccountServiceImpl **implements** AccountService {  @Autowired  AccountDao **accountDao**;  **public void** addAccount() {  **accountDao**.addAccount();  } } |

* 测试



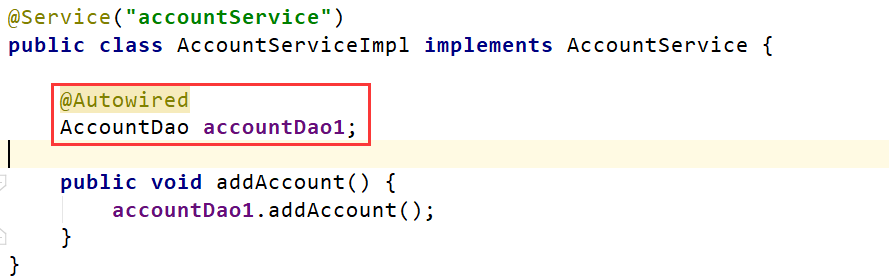
原因分析:

IOC容器中并没有发现名称为accountDao的bean。

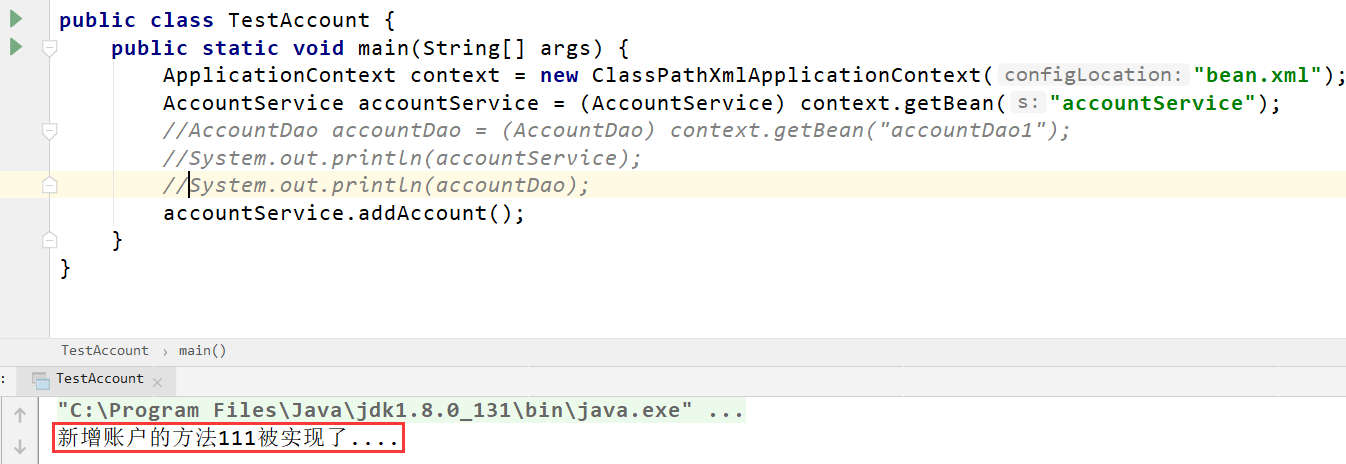


但是通过改造又可以解决问题。

* 改造service类



* 测试



* 总结

如果有唯一一个匹配时，就直接注入。

如果有多个匹配时，首先按照类型筛选出来匹配的对象。

然后使用变量名称作为bean的id，然后继续筛选。

如果两次筛选后有唯一匹配的，可以注入成功；否则失败。

#### 1.1.2.4 其他注入数据的注解

使用注入数据注解的效果跟在xml配置文件中的bean标签中写一个<property>标签的作用是一样的。

* @Qualifier:

作用：在按照类型注入的基础之上再按照名称注入。它在给类成员注入时不能单独使用；但是在给方法参数注入时可以(这个我们后面讨论)。

属性：value：用于指定注入bean的id。

* @Resource

作用：直接按照bean的id注入。它可以独立使用。

属性：name：用于指定bean的id。

以上三个注入都只能注入其他bean类型的数据，而基本类型和String类型无法使用上述注解实现。另外，集合类型的注入只能通过XML来实现。

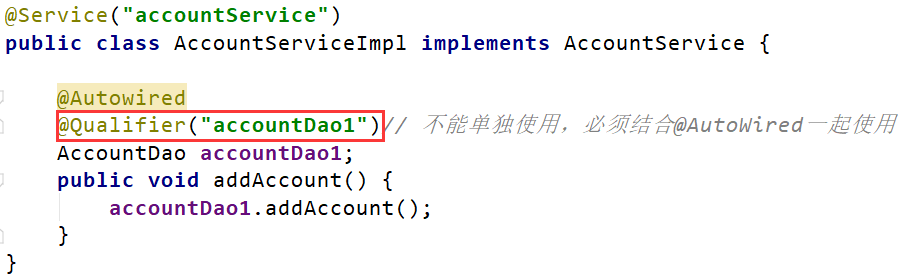
* @Value

作用：用于注入基本类型和String类型的数据。

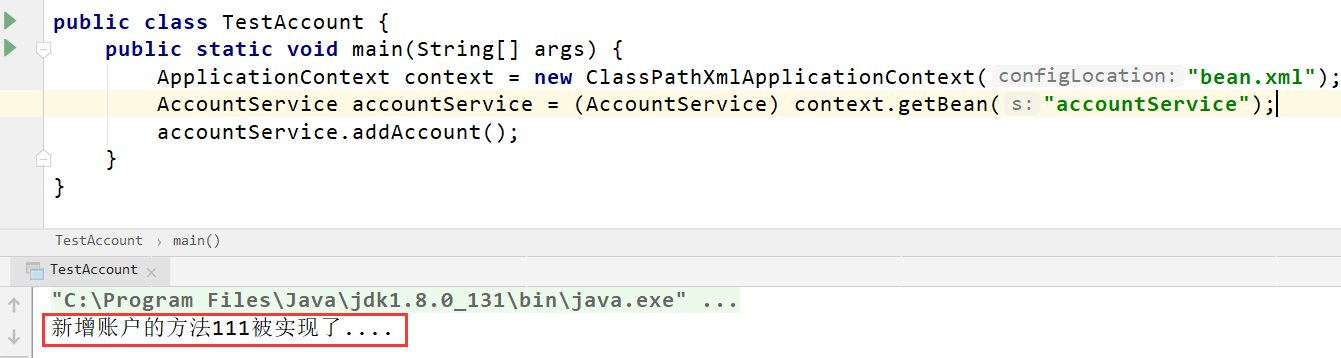
属性：value：用于指定数据的值。它也可以使用spring中SpEL(也就是spring的el表达式）。

SpEL的写法：${表达式}。

(1) 先看看@Qualifier，它可以在按照类中注入的基础之上再按照名称注入。



测试效果：



注意：在给类成员注入时，必须与Autowired注解配对使用，不能单独使用，否则会空指针异常。

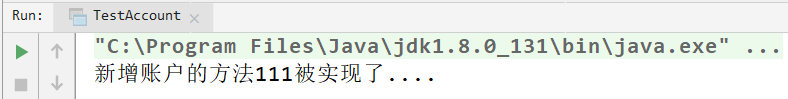
(2) @Resource注解的使用

我们单独使用@Autowired时需要改变变量名；使用@Qualifier时又不能单独使用，那有没有更方便的注解呢？下面介绍@Resource。

其中的name属性用于指定bean的id，此时不是value属性了，所以不能使用省略写法。



测试效果:



(3)@Value注解注入基本类型数据

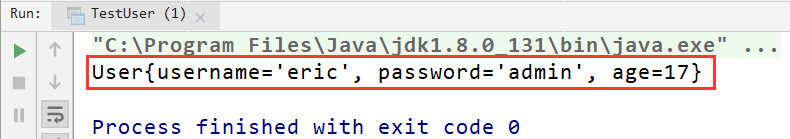
* 创建pojo对象

|  |
| --- |
| @Component(**"user"**) **public class** User {  @Value(**"eric"**)  **private** String **username**;  @Value(**"admin"**)  **private** String **password**;  @Value(**"17"**)  **private** Integer **age**;   @Override  **public** String toString() {  **return "User{"** +  **"username='"** + **username** + **'\''** +  **", password='"** + **password** + **'\''** +  **", age="** + **age** +  **'}'**;  } } |

* 测试

|  |
| --- |
| **public class** TestUser {  **public static void** main(String[] args) {  ApplicationContext context = **new** ClassPathXmlApplicationContext(**"bean.xml"**);  User user = (User) context.getBean(**"user"**);  System.***out***.println(user);  } } |

* 测试结果



#### 1.1.2.5 改变作用域的注解

@Scope:

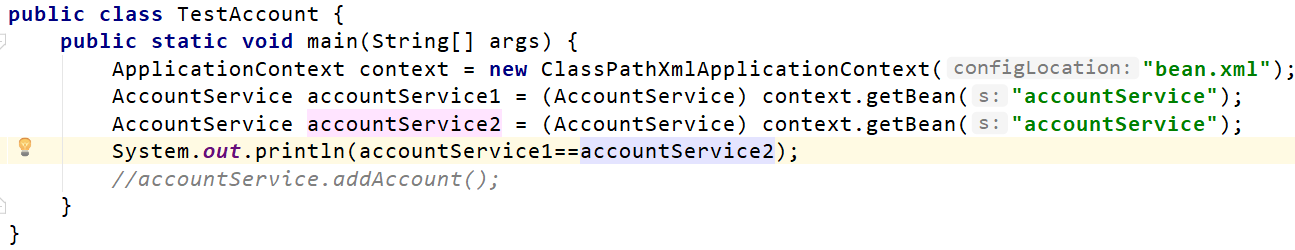
作用：用于指定bean的作用范围。

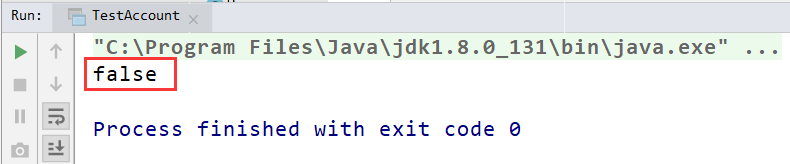
属性：value：指定范围的取值。常用取值：singleton(单例)、prototype(多例)。

* 在类上面添加@Scope属性



* 测试





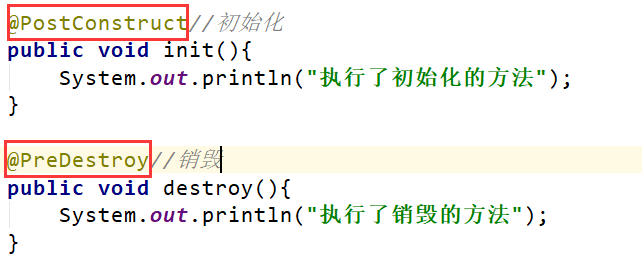
#### 1.1.2.6 生命周期相关的注解

使用与生命周期相关的注解的作用跟在bean标签中使用init-method和destroy-methode的作用是一样的。

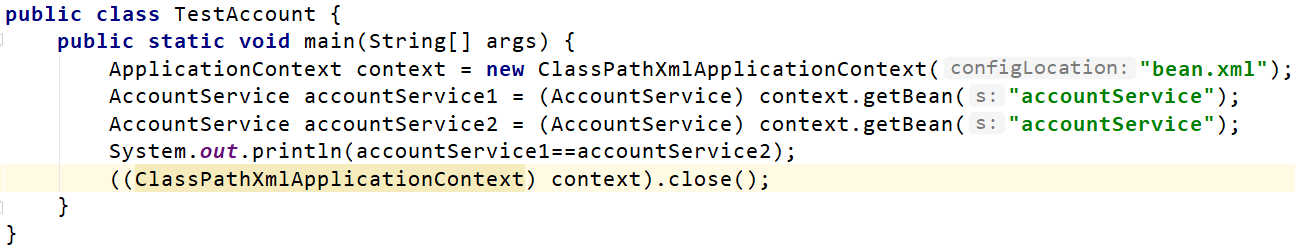
@PreDestroy 作用：用于指定销毁方法。

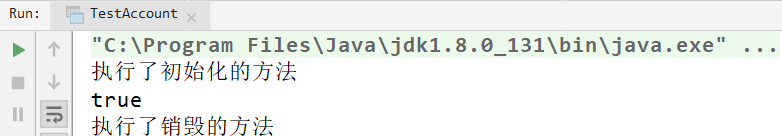
@PostConstruct 作用：用于指定初始化方法。

* 使用



* 测试



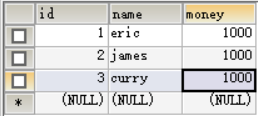


### 1.1.3 Spring之基于XML的IOC案例

#### 1.1.3.1 案例准备

* 创建数据表

|  |
| --- |
| create table account(  id int primary key auto\_increment,  name varchar(40),  money double  )character set utf8 collate utf8\_general\_ci;  insert into account(name,money) values(eric,1000);  insert into account(name,money) values('james',1000);  insert into account(name,money) values('curry',1000); |



* 然后新建一个普通的maven工程，导入以下坐标

|  |
| --- |
| <**dependencies**>  <**dependency**>  <**groupId**>org.springframework</**groupId**>  <**artifactId**>spring-context</**artifactId**>  <**version**>5.0.2.RELEASE</**version**>  </**dependency**>  <**dependency**>  <**groupId**>commons-dbutils</**groupId**>  <**artifactId**>commons-dbutils</**artifactId**>  <**version**>1.4</**version**>  </**dependency**>  <**dependency**>  <**groupId**>mysql</**groupId**>  <**artifactId**>mysql-connector-java</**artifactId**>  <**version**>5.1.6</**version**>  </**dependency**>  <**dependency**>  <**groupId**>c3p0</**groupId**>  <**artifactId**>c3p0</**artifactId**>  <**version**>0.9.1.2</**version**>  </**dependency**>  <**dependency**>  <**groupId**>junit</**groupId**>  <**artifactId**>junit</**artifactId**>  <**version**>4.12</**version**>  </**dependency**> </**dependencies**> |

* 编写实体类

|  |
| --- |
| **public class** Account {  **private** Integer **id**;  **private** String **name**;  **private** Double **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** Double getMoney() {  **return money**;  }   **public void** setMoney(Double money) {  **this**.**money** = money;  }   @Override  **public** String toString() {  **return "Account{"** +  **"id="** + **id** +  **", name='"** + **name** + **'\''** +  **", money="** + **money** +  **'}'**;  } } |

* 编写业务层接口 AccountService

|  |
| --- |
| **public interface** AccountService {  */\*\*  \* 查询所有  \*  \** ***@return*** *\*/* List<Account> findAllAccount();   */\*\*  \* 查询一个  \*  \** ***@return*** *\*/* Account findAccountById(Integer accountId);   */\*\*  \* 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** deleteAccount(Integer accountId); } |

* 编写持久层接口

|  |
| --- |
| **public interface** AccountDao {  */\*\*  \* 查询所有  \*  \** ***@return*** *\*/* List<Account> findAllAccount();   */\*\*  \* 查询一个  \*  \** ***@return*** *\*/* Account findAccountById(Integer accountId);   */\*\*  \* 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** deleteAccount(Integer accountId); } |

* 业务接口实现类

|  |
| --- |
| **public class** AccountServiceImpl **implements** AccountService {   **private** AccountDao **accountDao**;   */\*\*  \* 使用set方法注入AccountDao  \** ***@param accountDao*** *\*/* **public void** setAccountDao(AccountDao accountDao) {  **this**.**accountDao** = accountDao;  }   **public** List<Account> findAllAccount() {  **return accountDao**.findAllAccount();  }   **public** Account findAccountById(Integer accountId) {  **return accountDao**.findAccountById(accountId);  }   **public void** saveAccount(Account account) {  **accountDao**.saveAccount(account);  }   **public void** updateAccount(Account account) {  **accountDao**.updateAccount(account);  }   **public void** deleteAccount(Integer accountId) {  **accountDao**.deleteAccount(accountId);  } } |

* 编写持久层实现类

|  |
| --- |
| **public class** AccountDaoImpl **implements** AccountDao {   **private** QueryRunner **runner**;  */\*\*  \* 使用set方法注入QueryRunner  \** ***@param runner*** *\*/* **public void** setRunner(QueryRunner runner) {  **this**.**runner** = runner;  }   **public** List<Account> findAllAccount() {  **try**{  **return runner**.query(**"select \* from account"**,**new** BeanListHandler<Account>(Account.**class**));  }**catch** (Exception e) {  **throw new** RuntimeException(e);  }  }   **public** Account findAccountById(Integer accountId) {  **try**{  **return runner**.query(**"select \* from account where id = ? "**,**new** BeanHandler<Account>(Account.**class**),accountId);  }**catch** (Exception e) {  **throw new** RuntimeException(e);  }  }   **public void** saveAccount(Account account) {  **try**{  **runner**.update(**"insert into account(name,money)values(?,?)"**,account.getName(),account.getMoney());  }**catch** (Exception e) {  **throw new** RuntimeException(e);  }  }   **public void** updateAccount(Account account) {  **try**{  **runner**.update(**"update account set name=?,money=? where id=?"**,account.getName(),account.getMoney(),account.getId());  }**catch** (Exception e) {  **throw new** RuntimeException(e);  }  }   **public void** deleteAccount(Integer accountId) {  **try**{  **runner**.update(**"delete from account where id=?"**,accountId);  }**catch** (Exception e) {  **throw new** RuntimeException(e);  }  } } |

我们现在还无法运行，因为我们还没有配置ioc，还没有注入依赖。

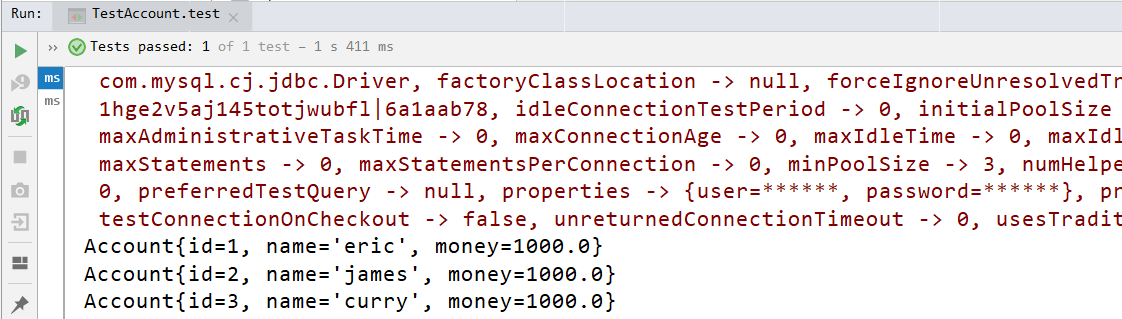
#### 1.1.3.2 编写SpringIOC配置文件

|  |
| --- |
| *<?***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"**>  *<!-- 配置Service对象-->* <**bean id="accountService" class="com.qf.service.impl.AccountServiceImpl"**>  *<!--注入dao对象-->* <**property name="accountDao" ref="accountDao"**></**property**>  </**bean**>   *<!-- 配置Dao对象-->* <**bean id="accountDao" class="com.qf.dao.impl.AccountDaoImpl"**>  *<!--注入QueryRunner对象-->* <**property name="runner" ref="runner"**></**property**>  </**bean**>   *<!--   配置QueryRunner对象  对于数据源对象创建多例的，因为可能是有多个用户都在用数据源，可能存在一个用户在用数据源的时候，另外一个用户正在用的问题。    -->* <**bean id="runner" class="org.apache.commons.dbutils.QueryRunner" scope="prototype"**>  *<!--注入数据源-->* <**constructor-arg name="ds" ref="dataSource"**></**constructor-arg**>  </**bean**>   *<!-- 配置数据源 -->* <**bean id="dataSource" class="com.mchange.v2.c3p0.ComboPooledDataSource"**>  *<!--连接数据库的必备信息-->* <**property name="driverClass" value="com.mysql.jdbc.Driver"**></**property**>  <**property name="jdbcUrl" value="jdbc:mysql://192.168.10.137:3306/ssm"**></**property**>  <**property name="user" value="root"**></**property**>  <**property name="password" value="Admin123!"**></**property**>  </**bean**> </**beans**> |

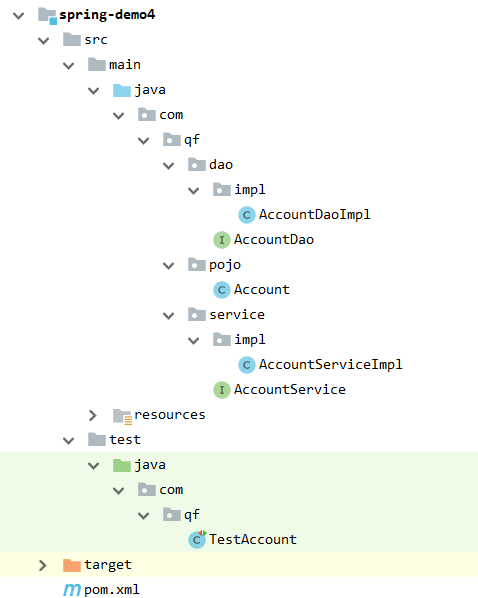
#### 1.1.3.3 测试

|  |
| --- |
| **public class** TestAccount {  @Test  **public void** test(){  *//1.获取容器* ApplicationContext ac = **new** ClassPathXmlApplicationContext(**"bean.xml"**);  *//2.得到业务层对象* AccountService as = ac.getBean(**"accountService"**, AccountService.**class**);  *//3.执行方法* List<Account> accounts = as.findAllAccount();  **for** (Account account : accounts) {  System.***out***.println(account);  }  } } |

结果:



最后工程目录结构如下:



### 1.1.4 使用注解配置改造IOC案例

我们创建一个普通的maven项目，并且把上个项目的文件拷贝过来。

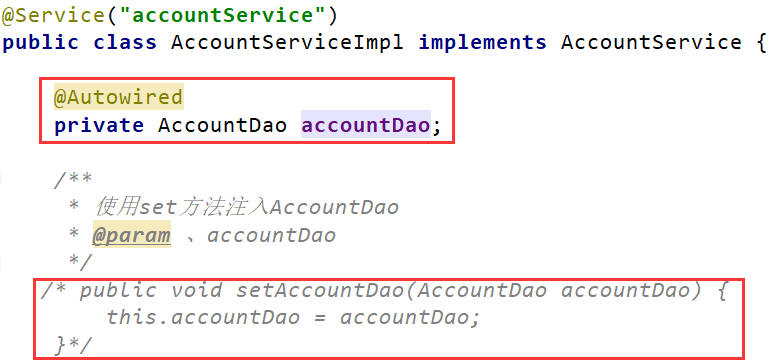
* 配置包扫描

|  |
| --- |
| *<!--配置包扫描-->* <**context:component-scan base-package="com.qf"**></**context:component-scan**> |

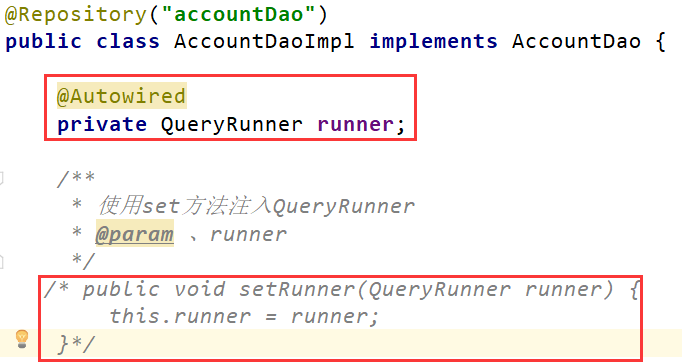
* 使用注解管理AccountDaoImpl AccountServiceImpl

|  |
| --- |
| @Repository(**"accountDao"**) **public class** AccountDaoImpl **implements** AccountDao |
| @Service(**"accountService"**) **public class** AccountServiceImpl **implements** AccountService |

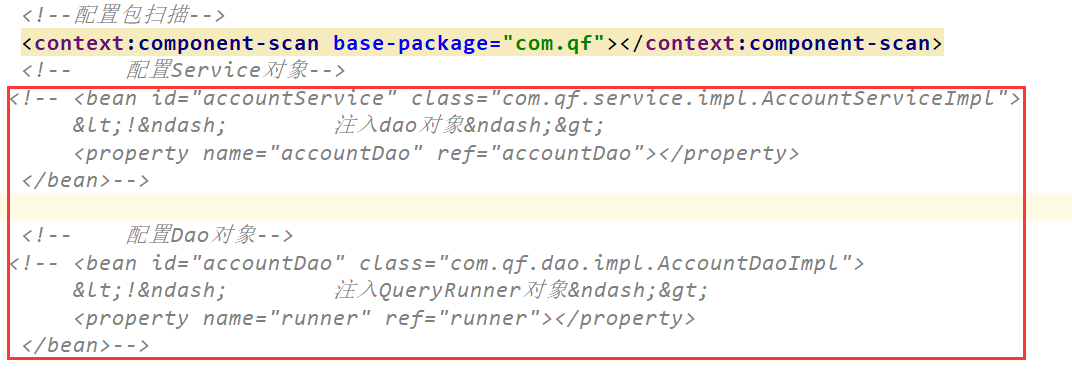
* 在业务实现类里面去掉set方法，改用注解注入。



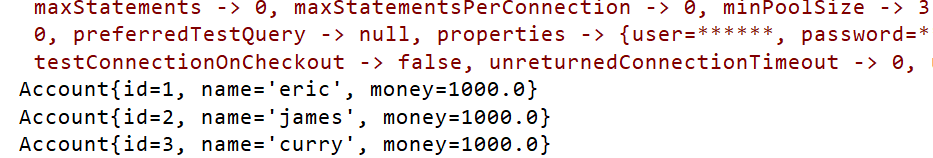
* 在持久层实现类上面也去掉set方法，改用注解注入。



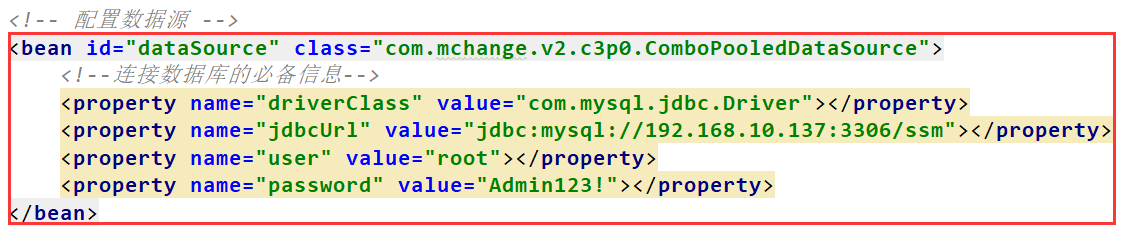
* 更改配置文件，去掉之前用bean标签管理的bean



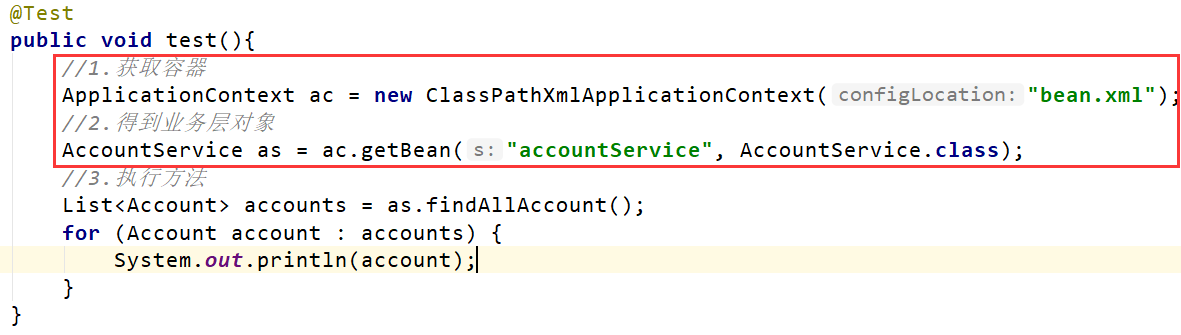
* 测试



以上代码还是存在问题：



我们的bean.xml配置文件依然存在xml配置。

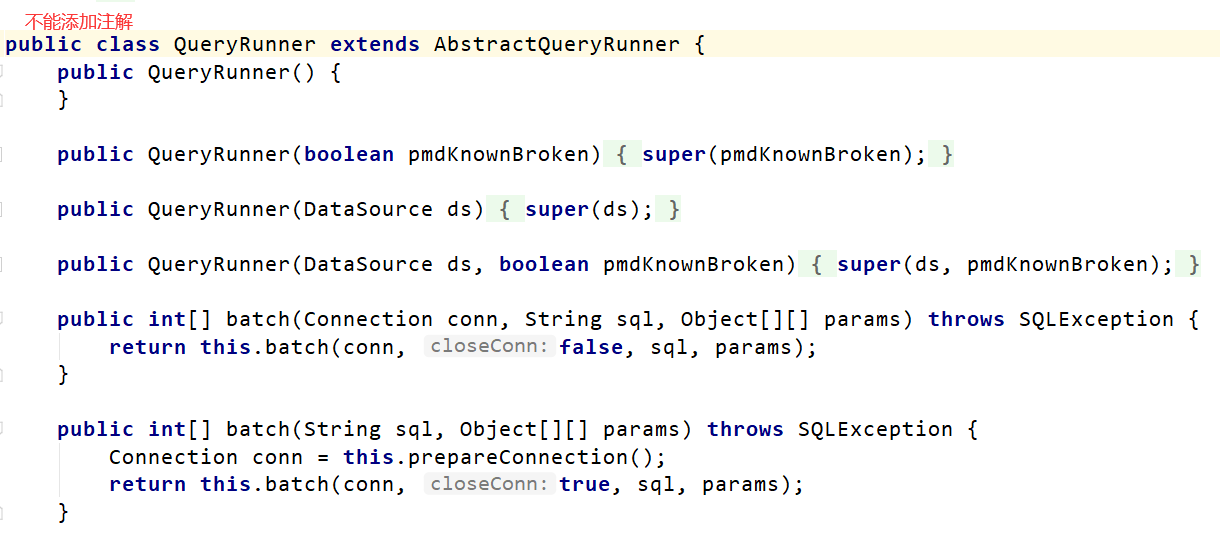


还有每次测试的时候，上面的两行代码都是重复的。

### 1.1.5 使用配置类改造案例

#### 1.1.5.1 Configuration和ComponentScan

我们目前似乎只能给我们自己写的类加注解，我们调用的QueryRunner类并不是我们自己的类，似乎不能加注解，只能用xml。



QueryRunner是如此，它所依赖的DataSource也是如此，无法加注解。只有加注解才能省掉那一大段xml，那有什么解决方案吗？

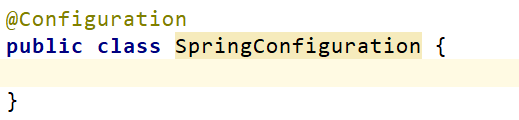
创建一个普通maven项目，并且把上一个项目的代码拷贝进去。

* 创建一个配置类

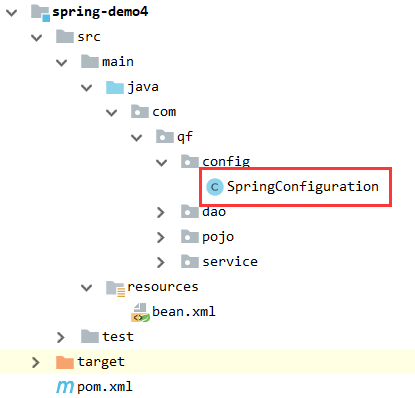
(1) 配置类的作用和bean.xml一样。首先介绍Configuration注解：

作用：指定当前类是一个配置类

细节：当配置类作为AnnotationConfigApplicationContext对象创建的参数时，该注解可以不写（后面再介绍这个细节）。



创建之后的目录结构如下:



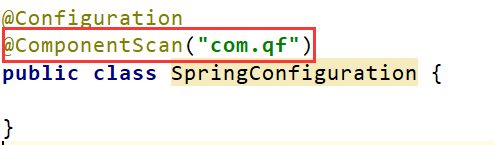
(2) 然后介绍ComponentScan注解。

作用：用于通过注解指定spring在创建容器时要扫描的包

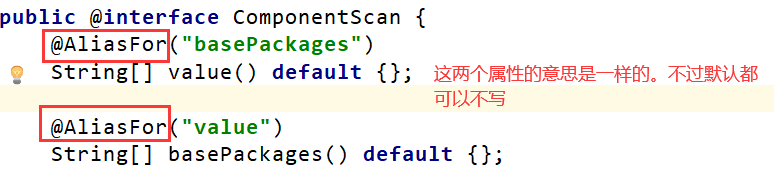
属性：value：它和basePackages的作用是一样的，都是用于指定创建容器时要扫描的包。

我们使用此注解就等同于在xml中配置了:

<context:component-scan base-package="com.itheima"></context:component-scan>

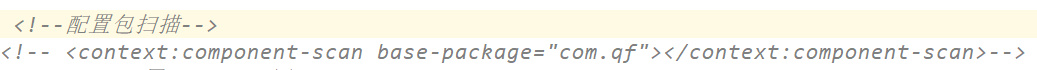


此注解的属性有两个：**value**和**basePackages**，并且互为别名。



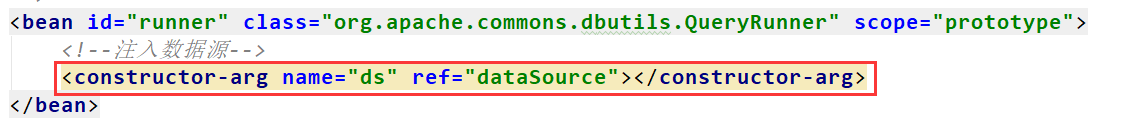
我们使用其中一个即可，并且用value的话还可以使用省略写法，并且由于数组中只有一个元素，所以可以省略花括号。

此时我们可以删除bean.xml的扫描包配置。



#### 1.1.5.2 Bean注解

我们先看看QueryRunner对象，我们发现它是用构造函数注入的，并且是带参数的。



那我们又将如何把对象初始化后放入容器呢？

接下来介绍Bean注解。

作用：用于把当前方法的返回值作为bean对象存入spring的ioc容器中

属性: name：用于指定bean的id。当不写时，默认值是当前方法的名称

细节：当我们使用注解配置方法时，**如果方法有参数，spring框架会去容器中查找有没有可用的bean对象。**

查找的方式和Autowired注解的作用是一样的。



此时bean.xml里面的所有配置都可以去掉了(甚至bean.xml都可以删除了)。

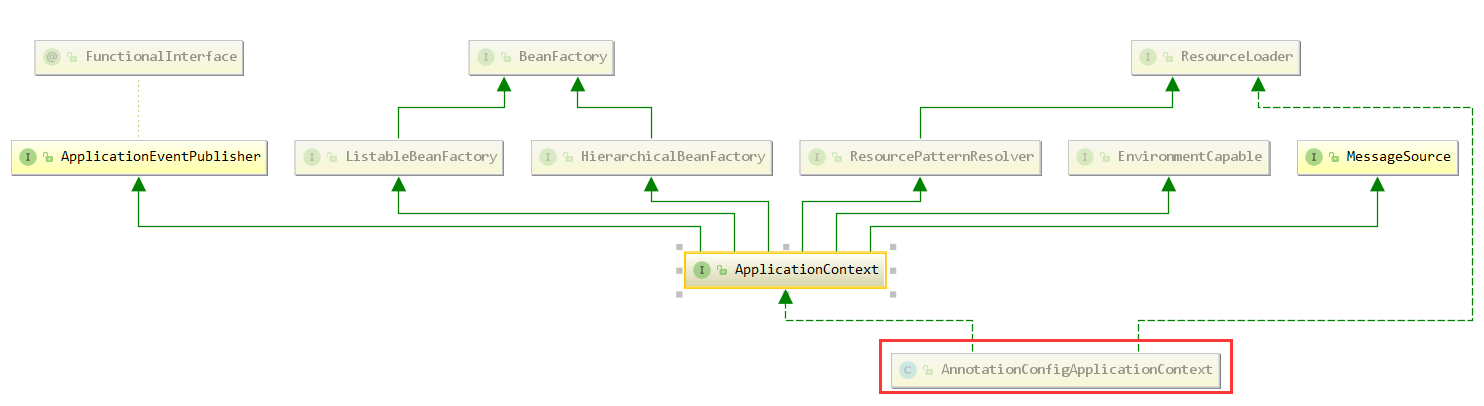


#### 1.1.5.3 AnnotationConfigApplicationContext类

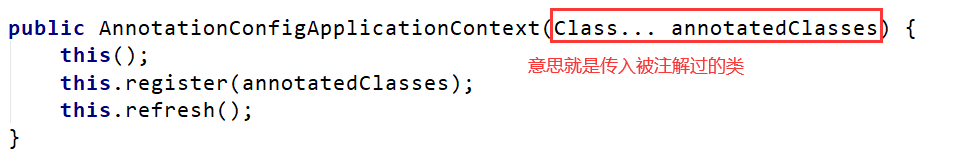
此时删除bean.xml，那么如何测试?

|  |
| --- |
| ApplicationContext ac = new ClassPathXmlApplicationContext("**bean.xml**"); |

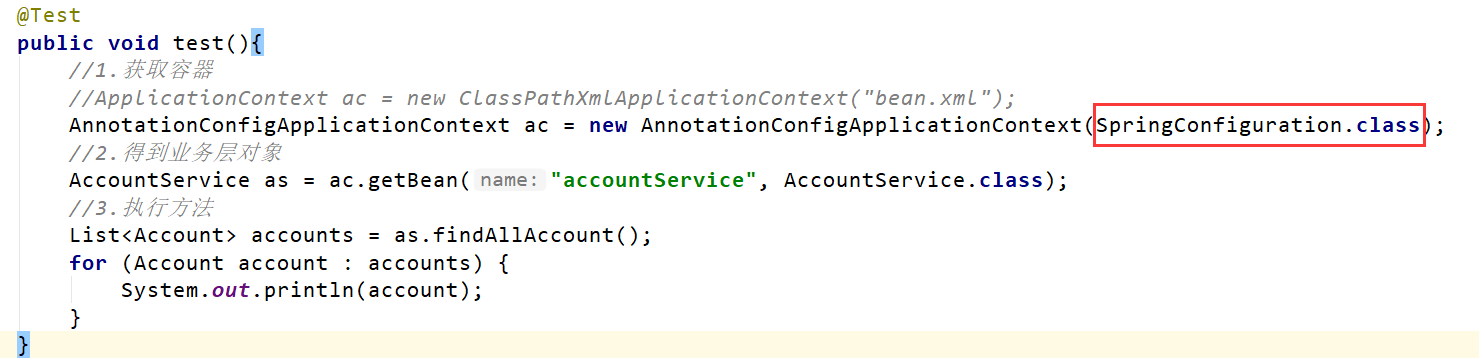
我们的测试类还是依赖于bean.xml的，我们回想之前讲ApplicationContext实现类的时候，还有最后一种没有讲，现在可以讲了。



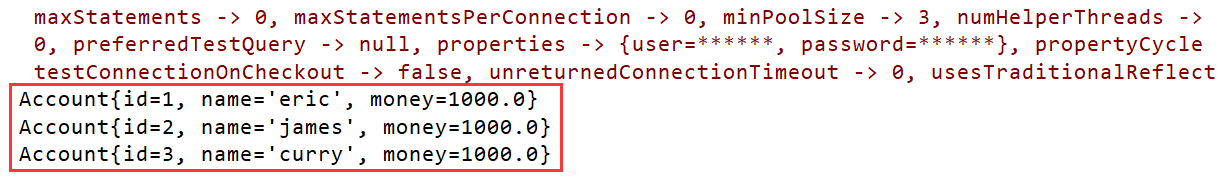
进去看看它的构造方法：



替换成AnnotationConfigApplicationContext即可。



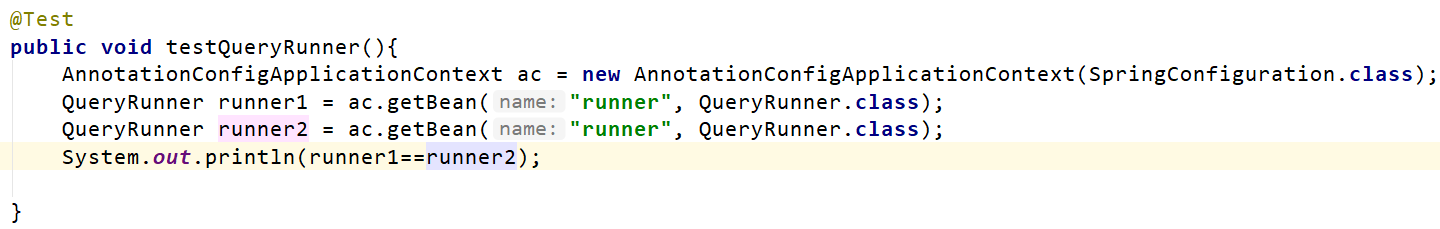
最后测试效果:



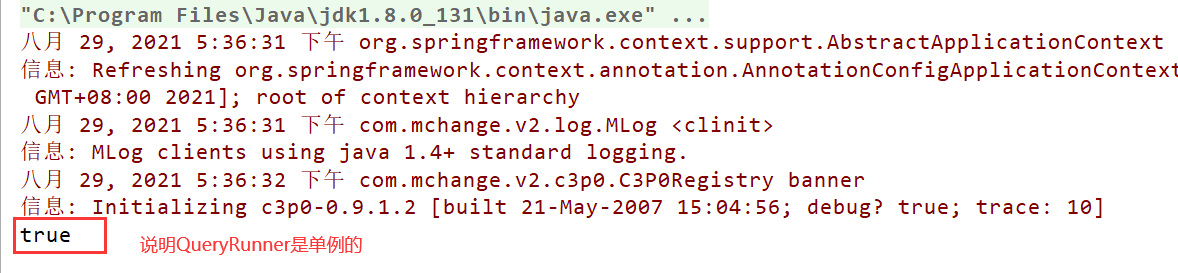
#### 1.1.5.4 改变作用域范围

虽然成功运行了，但是有一些问题还是要注意的：曾经的**QueryRunner**是多例的，但是现在的是**单例的。**

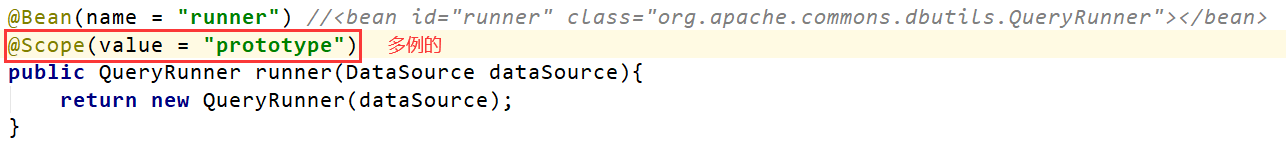
我们可以测试一下：



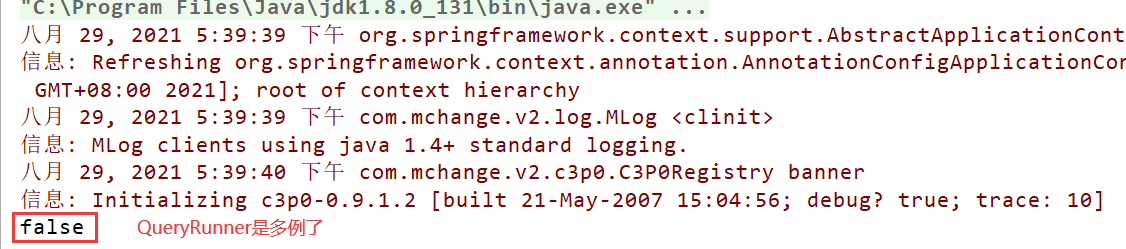
测试结果：



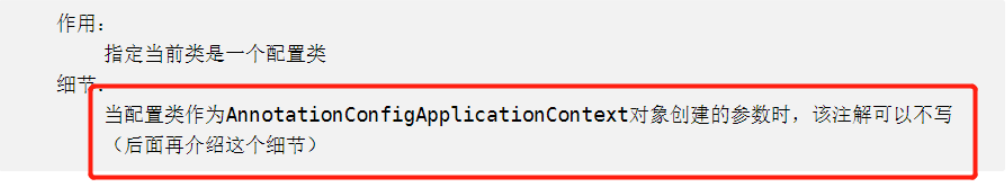
那我们要怎么改成多例呢？



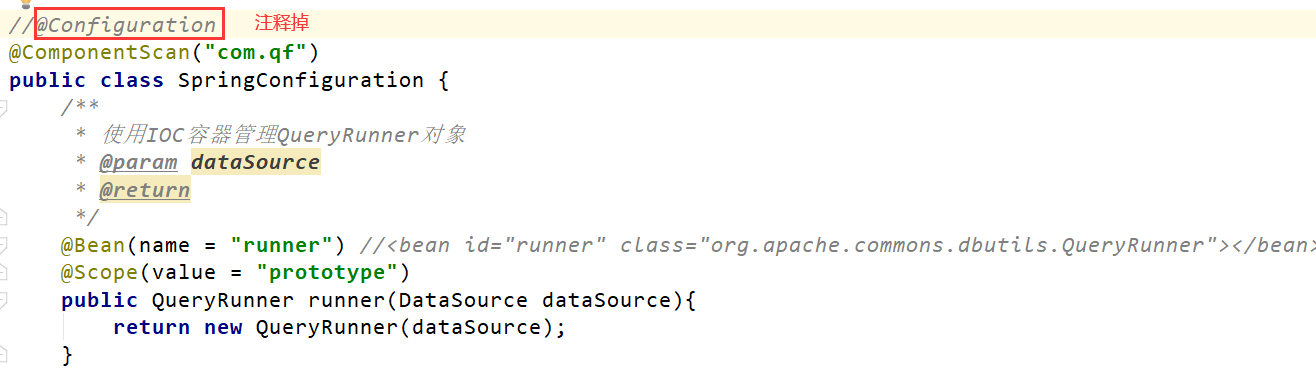
然后再测试:



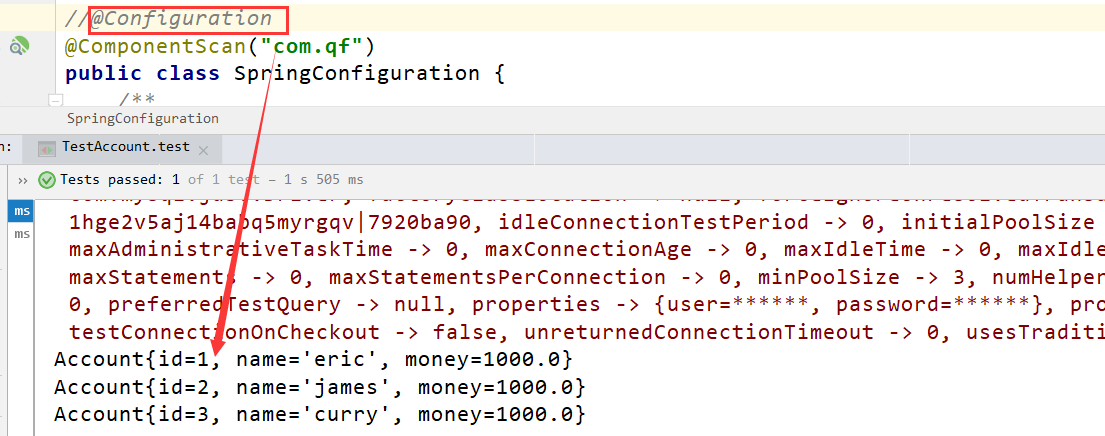
#### 1.1.5.5 Configuration注解之小细节



我们验证一下：

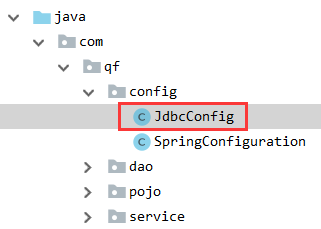


测试效果:



但是并不是所有情况下都能省略Configuration注解的，下面这种情况就不能省略:

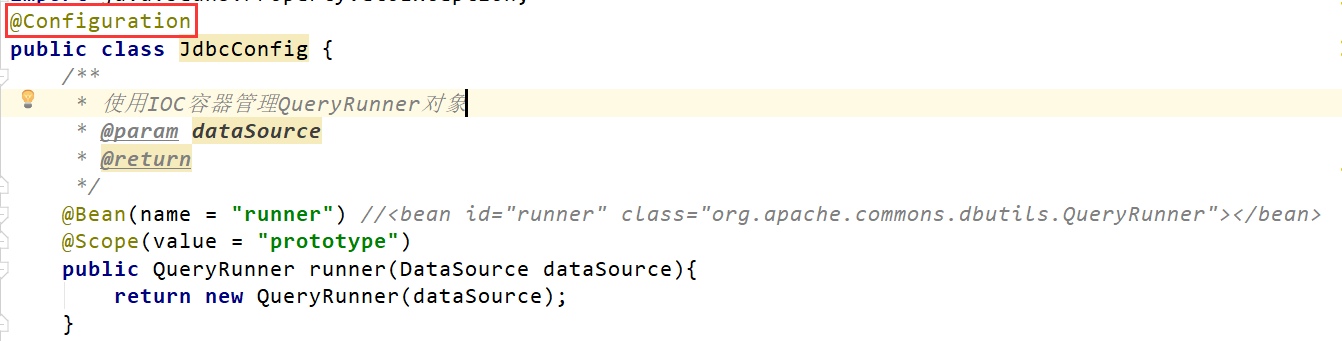
如果觉得初始化代码全部都写在SpringConfiguration一个类中太过于臃肿，我们可以分开写。比如说创建一个JdbcConfig专门用来初始化连接池对象。

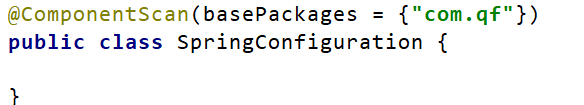




我们尝试运行测试类，发现报错。

此时Spring认为JdbcConfig不是一个配置类，所以@Configuration注解不能省略了。





测试没有问题。



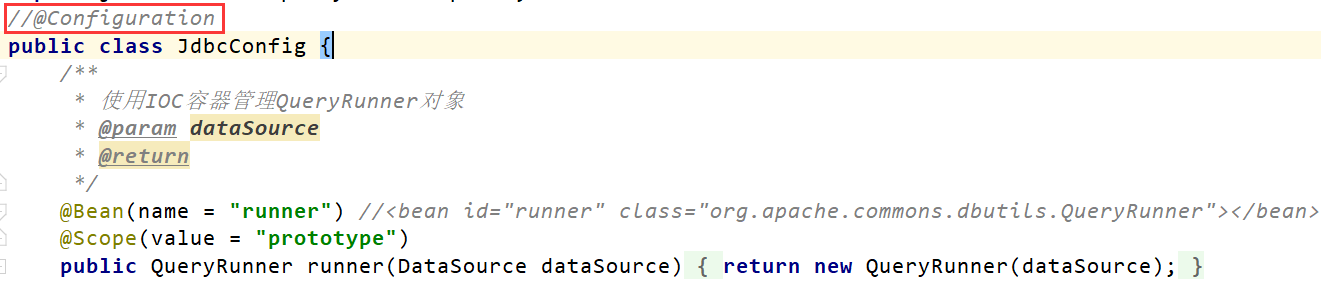
#### 1.1.5.6 Import注解

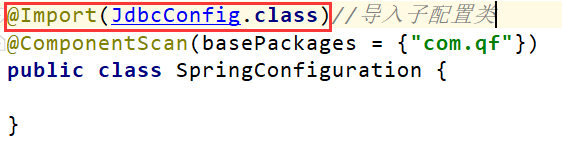
作用：用于导入其他的配置类

属性：value：用于指定其他配置类的字节码

细节：当我们使用Import的注解之后，有Import注解的类就父配置类，而导入的都是子配置类。

我们此时不需要加Configuration注解，只需要在父配置类上添加Import注解







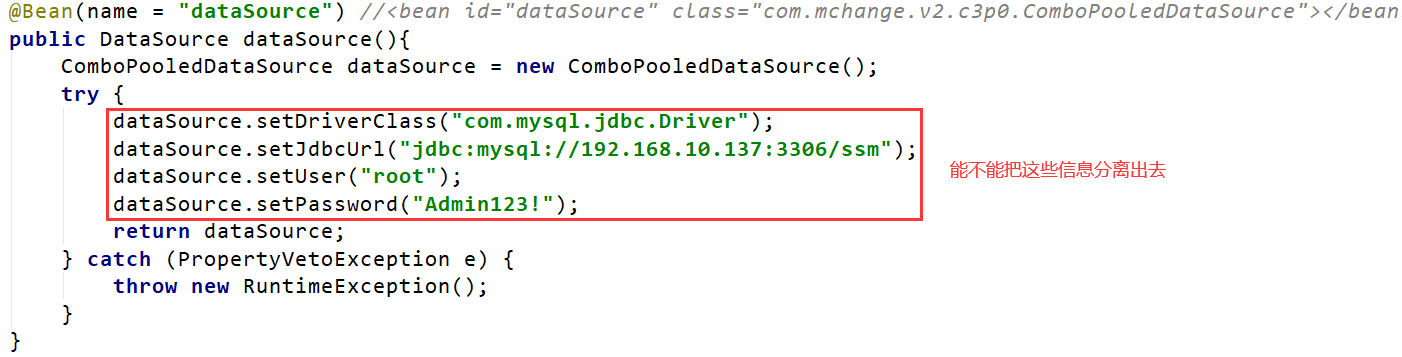
#### 1.1.5.7 PropertySource注解

作用：用于指定properties文件的位置

属性：value：指定文件的名称和路径

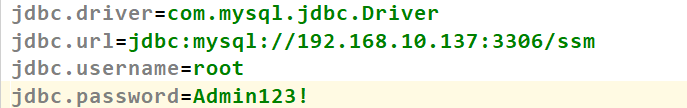
关键字：classpath：表示类路径下

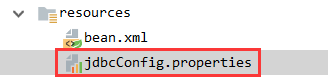
我们的目标是把连接池对象的配置信息分离出来，如下：



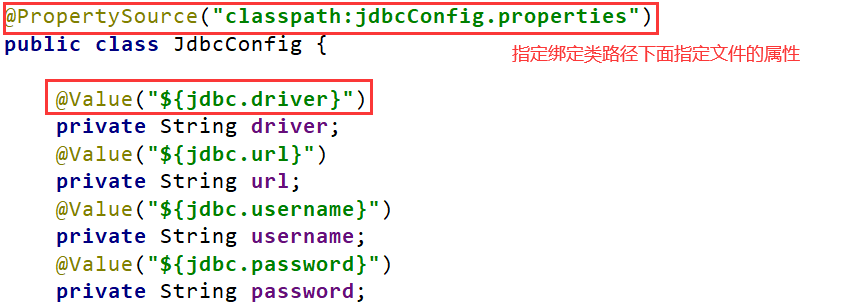
我们创建一个jdbcConfig.properties配置文件，具体步骤如下：

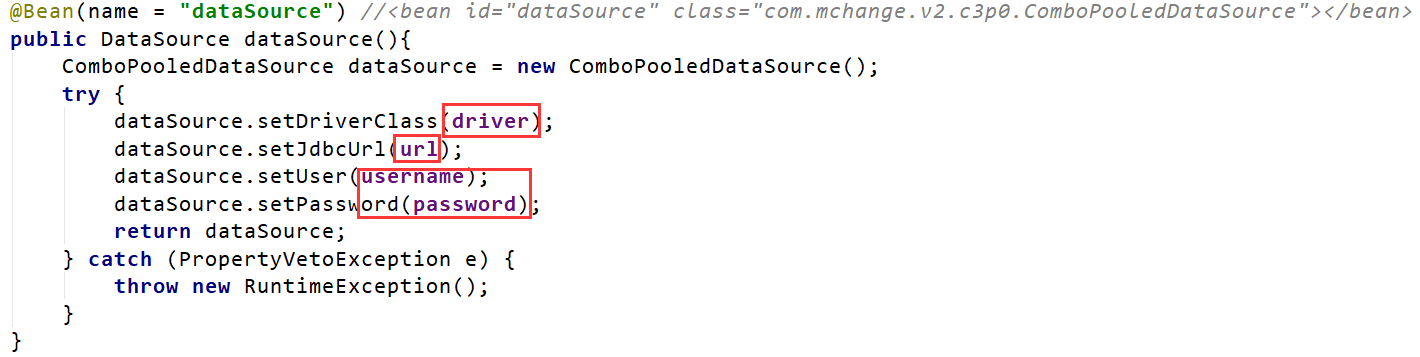
* 创建jdbcConfig.properties配置文件





* 改造JdbcConfig





* 测试效果略。

#### 1.1.5.8 问题：@Configuration注解一定需要吗?

答案是：@Configuration注解加不加都对程序的结果没有影响，不会因为@Configuration注解不加，造成bean注入失败。

演示：

第一步:首先我们准备两个类。(Brunner依赖于ADataSource)

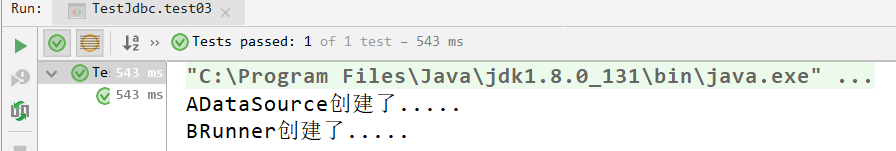
|  |
| --- |
| **public class** ADataSource {  **public** ADataSource(){  System.***out***.println(**"ADataSource创建了....."**);  } } |

|  |
| --- |
| **public class** BRunner {  **public** BRunner(**ADataSource dataSource**){  System.***out***.println(**"BRunner创建了....."**);  } } |

第二步：将这两个类通过java配置类，交给spring容器管理。

|  |
| --- |
| **@Configuration** *//标识当前类是一个配置类 它的作用类似于bean.xml* @ComponentScan(basePackages = {**"com.qf"**}) *//包扫描的注解* **public class** DataSourceConfig {  @Bean  **public** BRunner bRunner(){  **return new** BRunner(aDataSource());  }   @Bean  **public** ADataSource aDataSource(){  **return new** ADataSource();  } } |

测试结果：



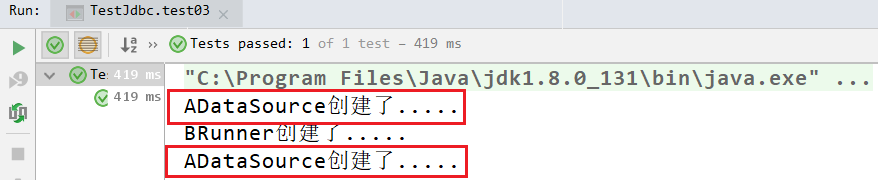
疑问1:为什么ADataSource只创建了1次?

当我们使用@Bean注解修饰aDataSource方法的时候，会将ADataSource这个bean交给容器管理(就会调用ADataSource的无参数构造函数)。

我们使用@Bean注解修饰bRunner方法的时候,**由于需要使用到ADataSource,这个时候，不再重新的创建ADataSource的对象，而是直接从spring容器(spring的单例池中获取)。所以ADataSource只会创建1次。**

以为2：如果去掉@Configuration注解只会会如何?

经过测试，程序不会报错，ADataSource和Brunner两个bean都会交给spring容器管理，但是区别在于ADataSource创建了2次：



为什么创建两次?

第一次创建是使用@Bean注解修饰aDataSource方法的时候，创建的。第二次创建是@Bean注解修饰bRunner方法的时候，由于BRunner对象的创建需要依赖于ADataSource。此时ADataSource这个bean不是从单例池中获取，**而是重新又创建了1次。**

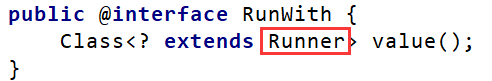
### 1.1.6 Spring整合Junit

这些测试代码，软件工程师当然可以编写没问题，但是测试工程师不一定可以写的出来,所以spring专门为测试工程师整合了junit，简化了操作。

* 导入Spring整合Junit的maven坐标

|  |
| --- |
| <**dependency**>  <**groupId**>org.springframework</**groupId**>  <**artifactId**>spring-test</**artifactId**>  <**version**>5.0.2.RELEASE</**version**> </**dependency**> |

* 使用junit提供的一个注解（@Runwith）把原有的main方法替换成spring提供的main方法。



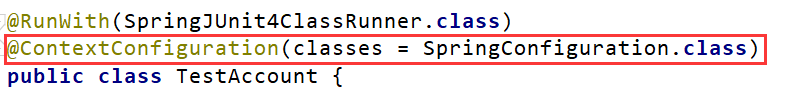
该注解的功能是替换Runner（运行器）。



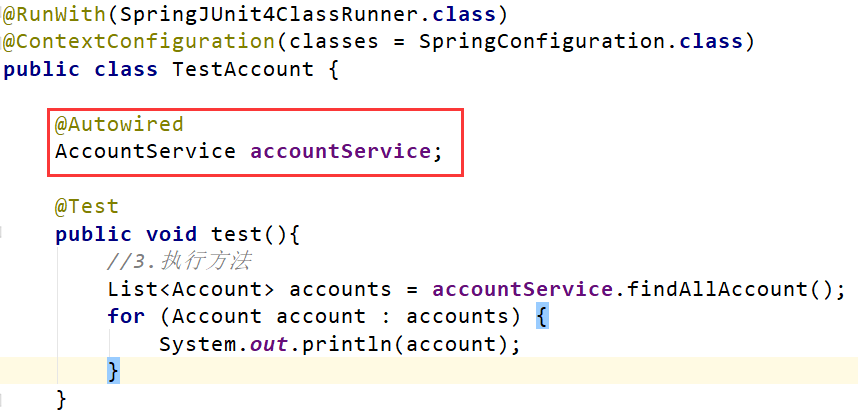
* 告知spring的运行器，spring和ioc创建是基于xml还是注解的，并且说明位置

使用ContextConfiguration注解说明。

* locations：指定xml文件的位置，加上classpath关键字，表示在类路径下。
* classes：指定注解类所在地位置。



* 测试



* 测试效果

