**Spring第一天**

**上海千锋 文威**

**总览：**

Spring学习一共分为4天

第一天: Spring框架的概述以及Spring基于XMl的IOC配置

第二天：Spring中基于注解的IOC的案例

第三天：Spring中的AOP和基于XML以及注解的配置

第四天：Spring中的JdbcTemplate以及Spring对事务的支持

**课程目标：**

1、了解spring的基本概述

* 了解什么是Spring
* 掌握Spring的两大核心的思想**(重点)**
* 了解Spring的发展历程
* 了解Spring的体系结构

2、程序的耦合和解耦

* 了解曾经案例中的问题
* 理解工厂模式解耦

3、IOC概念和程序中的IOC

* 理解IOC的基本概念**(重点)**
* 掌握Spring中基于XML的IOC环境搭建**(重点)**

4、依赖注入(Dependency Injection)

* 掌握基于XML的依赖注入**(重点)**

## 1.1 Spring的基本概述

### 1.1.1 什么是Spring

Spring 是分层的 Java SE/EE 应用 full-stack轻量级开源框架，以 IoC(Inverse Of Control：控制反转)和 AOP(Aspect Oriented Programming：面向切面编程)为内核提供了展现层 SpringMVC 和持久层 Spring JDBC 以及业务层事务管理等众多的企业级应用技术，还能整合众多著名的第三方框架和类库，逐渐成为使用最多的 Java EE 企业应用开源框架。

### 1.1.2 Spring的发展历程

1997 年 IBM 提出了 EJB 的思想

1998 年，SUN 制定开发标准规范 EJB1.0

1999 年，EJB1.1 发布

2001 年，EJB2.0 发布

2003 年，EJB2.1 发布

2006 年，EJB3.0 发布

Rod Johnson（spring 之父）

Expert One-to-One J2EE Design and Development(2002)

阐述了 J2EE 使用 EJB 开发设计的优点及解决方案

Expert One-to-One J2EE Development without EJB(2004)

阐述了 J2EE 开发不使用 EJB 的解决方式（Spring 雏形）

2004年 Spring1.0版本发布,Spring框架迅速发展。

2006年10月Spring 2.0于发布

...

2017 年 9 月份发布了 spring 的最新版本 spring 5.0 通用版（GA）

### 1.1.3 Spring的优势

* 方便解耦，简化开发

通过 Spring 提供的 IoC 容器，可以将对象间的依赖关系交由 Spring 进行控制，避免硬编码所造成的过度程序耦合。用户也不必再为单例模式类、属性文件解析等这些很底层的需求编写代码，可以更专注于上层的应用。

* AOP 编程的支持

通过 Spring 的 AOP 功能，方便进行面向切面的编程，许多不容易用传统 OOP 实现的功能可以通过AOP 轻松应付。

* 声明式事务的支持

可以将我们从单调烦闷的事务管理代码中解脱出来，通过声明式方式灵活的进行事务的管理，提高开发效率和质量。

* 方便程序的测试

可以用非容器依赖的编程方式进行几乎所有的测试工作，测试不再是昂贵的操作，而是随手可做的事情。

* 方便集成各种优秀框架

Spring 可以降低各种框架的使用难度，提供了对各种优秀框架（ Struts、 Hibernate、 Hessian、 Quartz等）的直接支持。

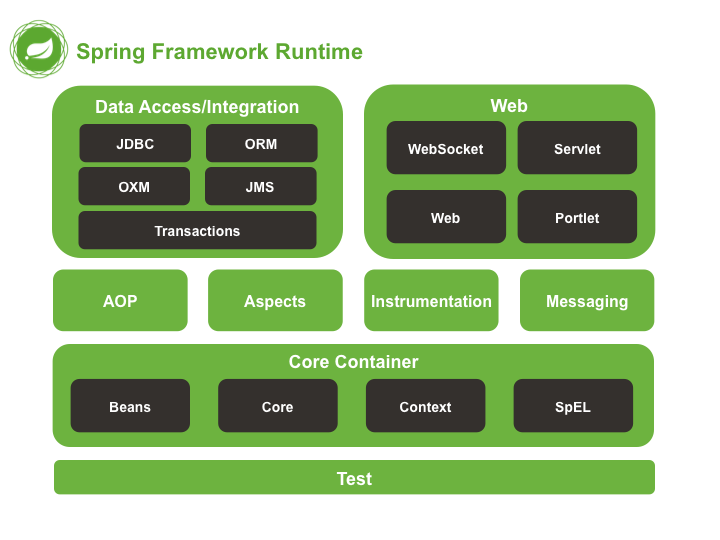
* 降低 JavaEE API 的使用难度

Spring 对 JavaEE API（如 JDBC、 JavaMail、远程调用等）进行了薄薄的封装层，使这些 API 的使用难度大为降低。

* Java 源码是经典学习范例

Spring 的源代码设计精妙、结构清晰、匠心独用，处处体现着大师对 Java 设计模式灵活运用以及对 Java 技术的高深造诣。它的源代码无意是 Java 技术的最佳实践的范例。

### 1.1.4 Spring体系



## 1.2 程序中的耦合

### 1.2.1 使用反射解耦

#### 1.2.1.1 代码示例

首先我们来看一段代码。这段代码描述的是使用jdbc查询数据库中的数据

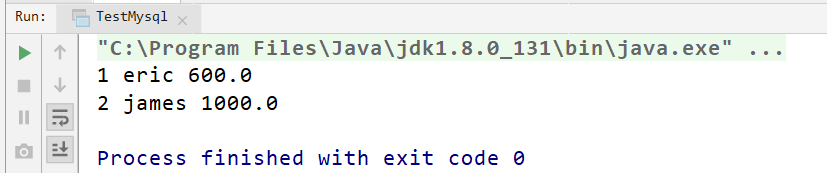
* 导入依赖

|  |
| --- |
| <**dependencies**>  <**dependency**>  <**groupId**>mysql</**groupId**>  <**artifactId**>mysql-connector-java</**artifactId**>  <**version**>5.1.6</**version**>  </**dependency**> </**dependencies**> |

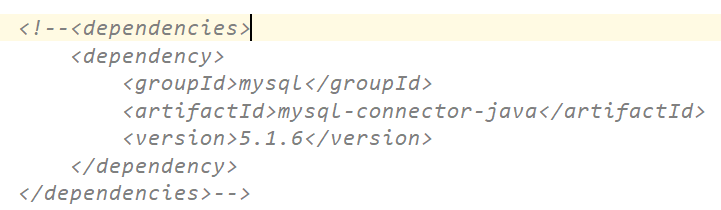
* 编写代码

|  |
| --- |
| **public static void** main(String[] args) **throws** Exception {  *//注册驱动* DriverManager.*registerDriver*(**new** com.mysql.jdbc.Driver());  String url = **"jdbc:mysql://192.168.10.137:3306/ssm"**;  *//获取连接对象* Connection connection = DriverManager.*getConnection*(url, **"root"**, **"Admin123!"**);  PreparedStatement statement = connection.prepareStatement(**"select** *\** **from account"**);  ResultSet resultSet = statement.executeQuery();  **while** (resultSet.next()){  Integer id = resultSet.getInt(**"id"**);  String name = resultSet.getString(**"name"**);  Double money = resultSet.getDouble(**"money"**);  System.***out***.println(id + **" "** + name + **" "** + money);  }  resultSet.close();  statement.close();  connection.close(); } |

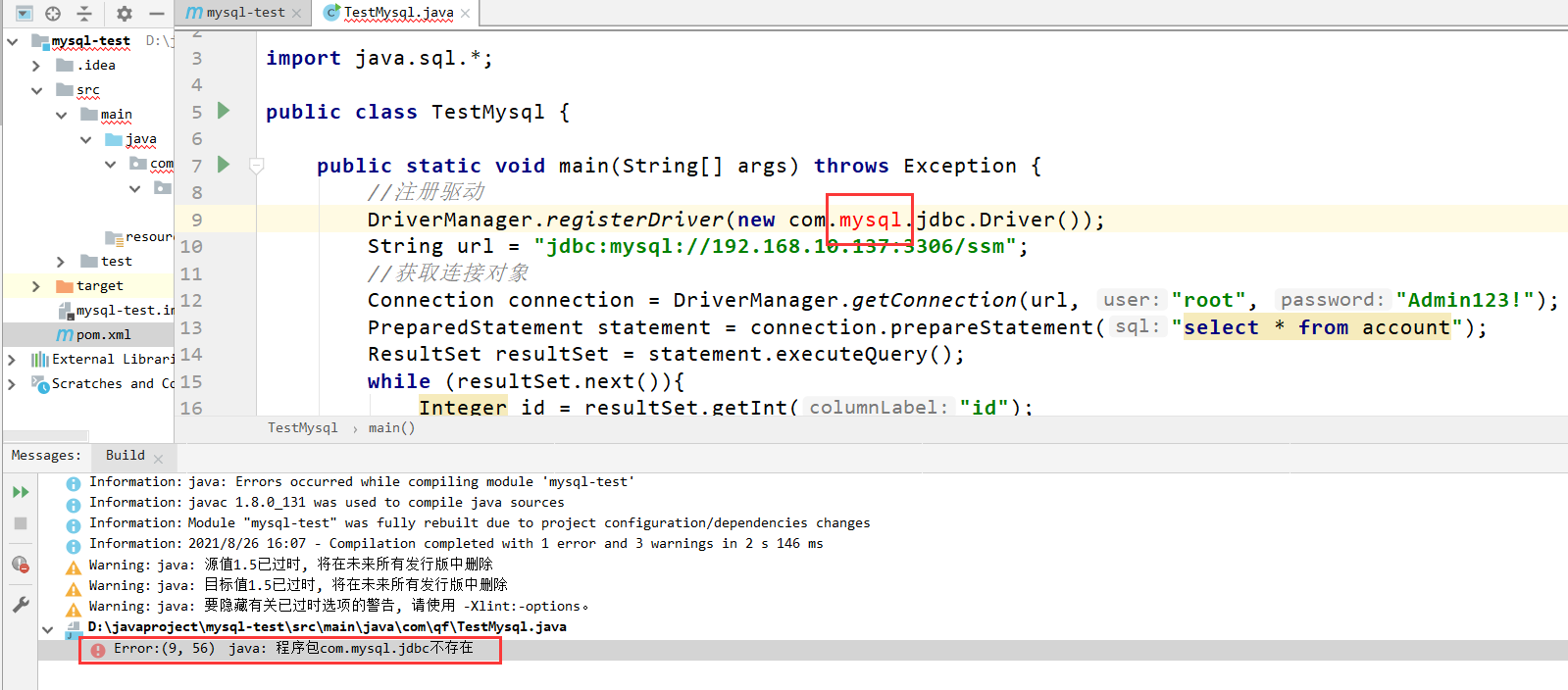
* 查看效果



此时代码是没有问题的。但是如果我们把POM文件中的依赖注释掉呢?



我们Ctrl+F9编译一下再看,此时程序报错



#### 1.2.1.2 耦合分析以及解决方案

什么是耦合?

耦合指的是程序间的依赖关系

类的依赖关系

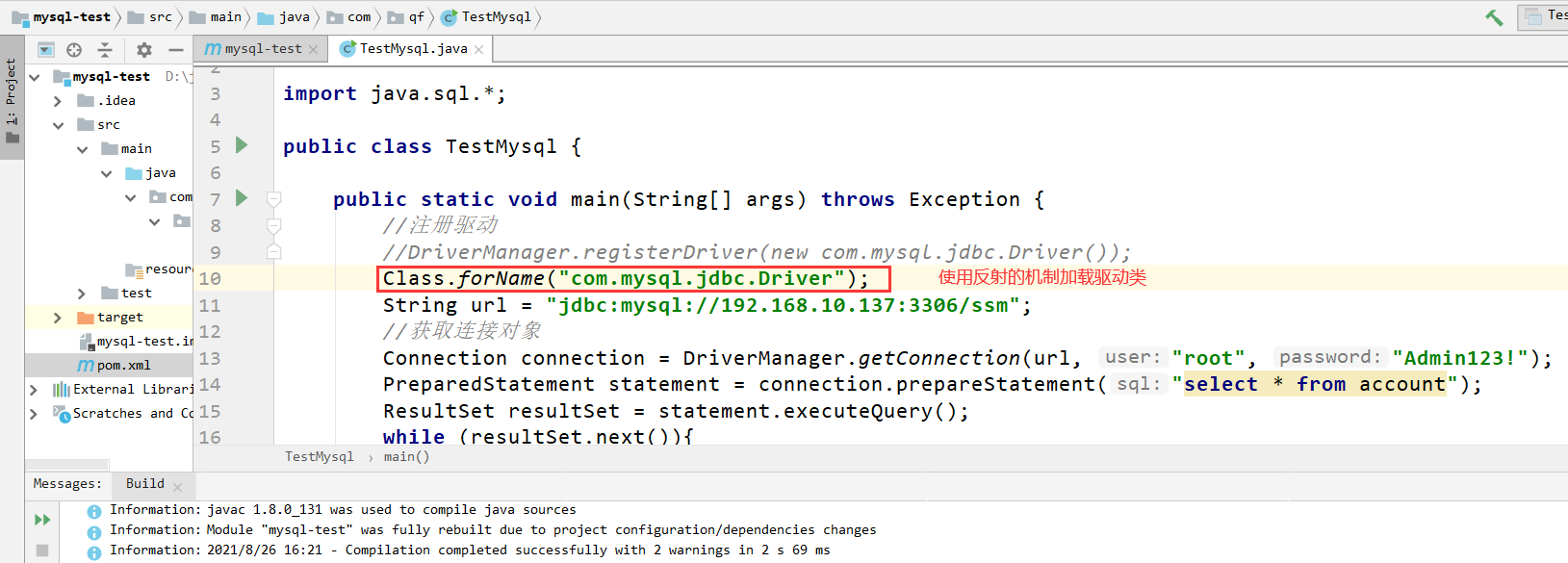
方法的依赖关系

注意：我们不能消除程序间的依赖关系，只能尽可能的降低程序的依赖关系。

这种降低程序间的依赖关系就叫做解耦。

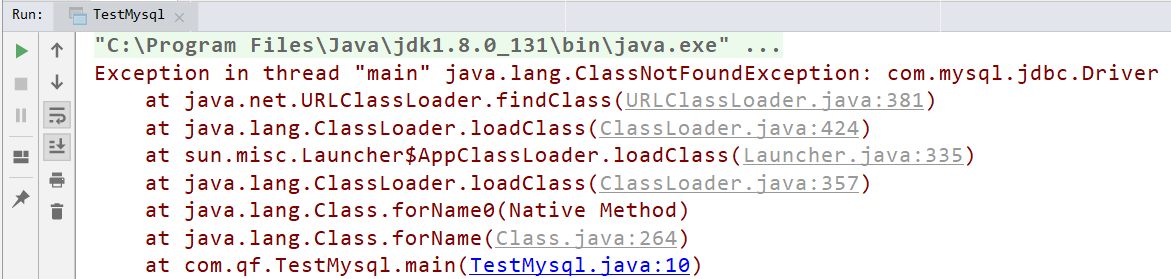
如何解耦：在实际开发中，我们应该做到在编译期不依赖，在运行期依赖。

所以我们对上面的代码进行改造



此时重新编译代码，这个时候就不会报错。

然后我们运行试试（此时pom.xml的坐标被注释了，在运行时会报异常）



但是！！！如果我现在不使用mysql驱动，换成oracle驱动呢?我们需要在源代码上去将mysql驱动改成oracle驱动，这样修改源代码，违反了项目开发的原则。

怎么解决?

将驱动信息定义在配置文件里面，通过读取配置文件的形式，来获取我们的配置信息。

总结：降低耦合是思路

1、使用反射的机制，避免使用new关键字。

2、通过读取配置文件的方式来获取资源的全限定名。

### 1.2.2使用工厂解耦

#### 1.2.2.1 示例代码

我们模拟一个新增用户的案例

* Dao部分

|  |
| --- |
| **public interface** AccountDao {  *//模拟新增account的方法* **public void** addAccount(); } |

|  |
| --- |
| **public class** AccountDaoImpl **implements** AccountDao{    *//模拟新增账户的dao* **public void** addAccount() {  System.***out***.println(**"新增账户的方法实现了....."**);  } } |

* Service部分

|  |
| --- |
| **public interface** AccountService {   *//模拟新增账户的业务* **public void** addAccount(); } |

|  |
| --- |
| **public class** AccountServiceImpl **implements** AccountService {   *//新增用户的方法* **public void** addAccount() {  **AccountDao dao = new AccountDaoImpl();**  **dao**.addAccount();  } } |

#### 1.2.2.2 耦合分析以及解决方案

此时我们也会发现问题：

在业务层里面(service层)会去调用dao层。这就是刚刚提到的耦合性问题。

如果此时删除掉接口的实现类，必然编译报错。

**在解决问题之前，我们先明确几个概念：**

1.Bean: 在计算机英语中，有可重用组件（如持久层和业务层的接口和重用）的含义。

2.JavaBean: 其实JavaBean并不完全等于实体类，因为实体类只是可重用组件的一部分，也就是JavaBean的范围大于实体类的范围。

JavaBean的含义：用Java语言编写的可重用组件。

3. 工厂就是创建我们的service和dao对象的(JavaBean)。

**解决步骤：**

1.需要一个配置文件来配置我们的service和dao。

配置的内容：唯一标志=全限定类名（key=value）。

2.通过读取配置文件中配置的内容，反射创建对象。

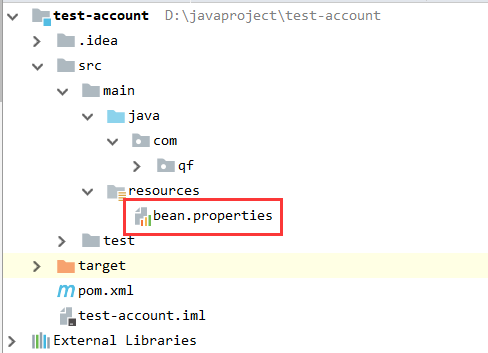
**配置文件的选取：**

1.xml（当然后期spring框架用的肯定是xml）

2.properties（读取更简单，我们这个例程先选用这种）

* 定义配置文件

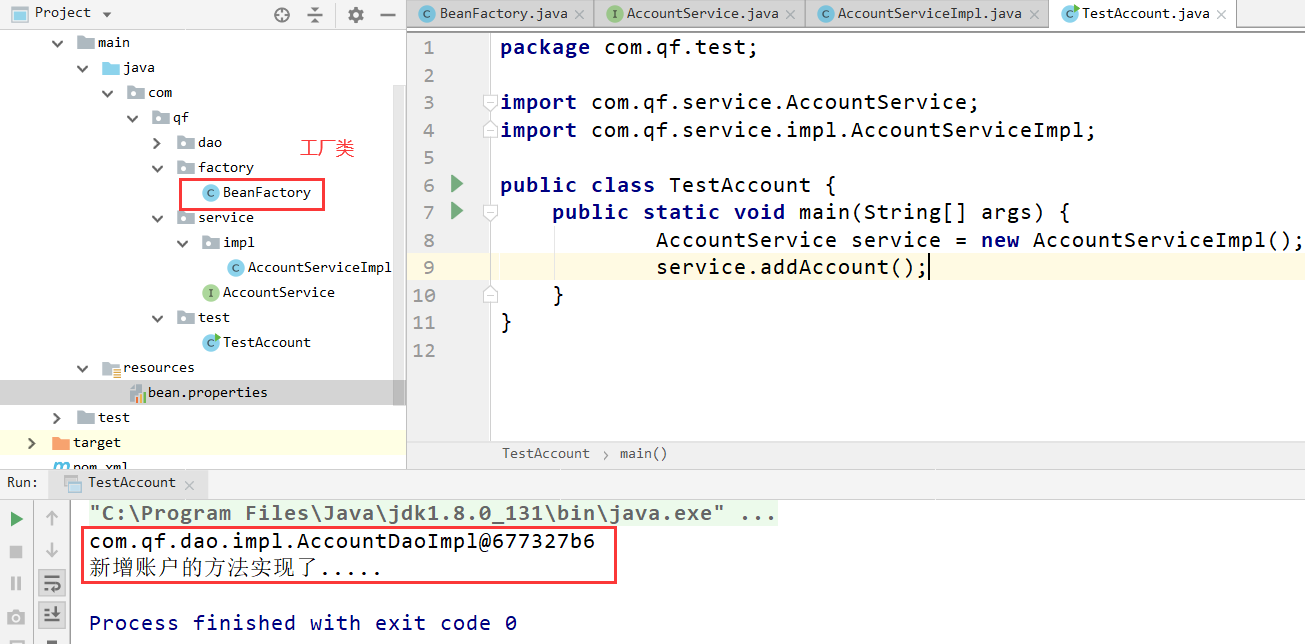
|  |
| --- |
| **accountService**=**com.qf.service.impl.AccountServiceImpl accountDao**=**com.qf.dao.impl.AccountDaoImpl** |



* 定义一个工厂类，通过工厂来获取bean

|  |
| --- |
| **public class** BeanFactory {   **static** Properties *properties*;   **static**{  InputStream inputStream = BeanFactory.**class**.getClassLoader().getResourceAsStream(**"bean.properties"**);  *properties* = **new** Properties();  **try** {  *properties*.load(inputStream);  } **catch** (IOException e) {  e.printStackTrace();  }  }   **public static** Object getBean(String name){  String value = *properties*.getProperty(name);  **try** {  Object o = Class.*forName*(value).newInstance();  **return** o;  } **catch** (Exception e) {  **throw new** RuntimeException();  }  } } |

* 测试

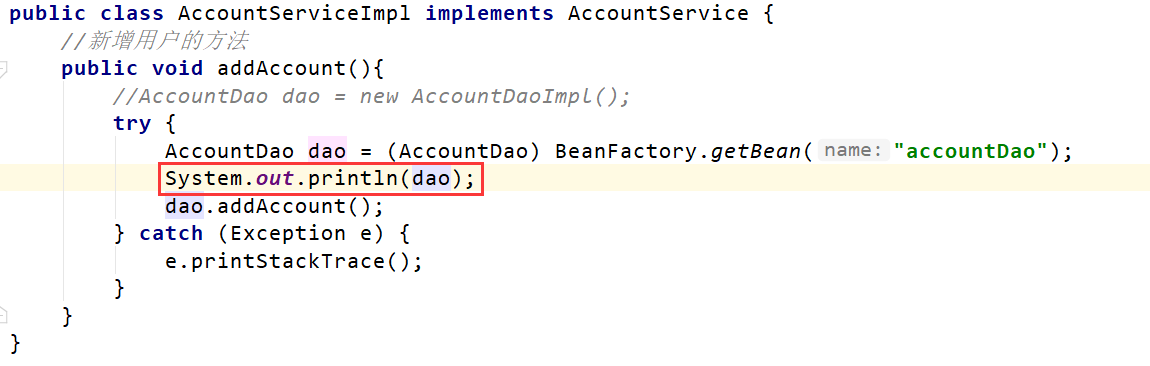


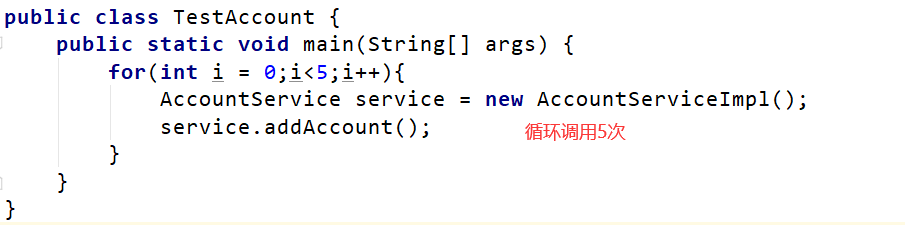
此时我们再尝试删除掉AccountDaoImpl这个实现类的时候.发现程序在编译的时候不会出错。

#### 1.2.2.3 工厂模式出现的问题

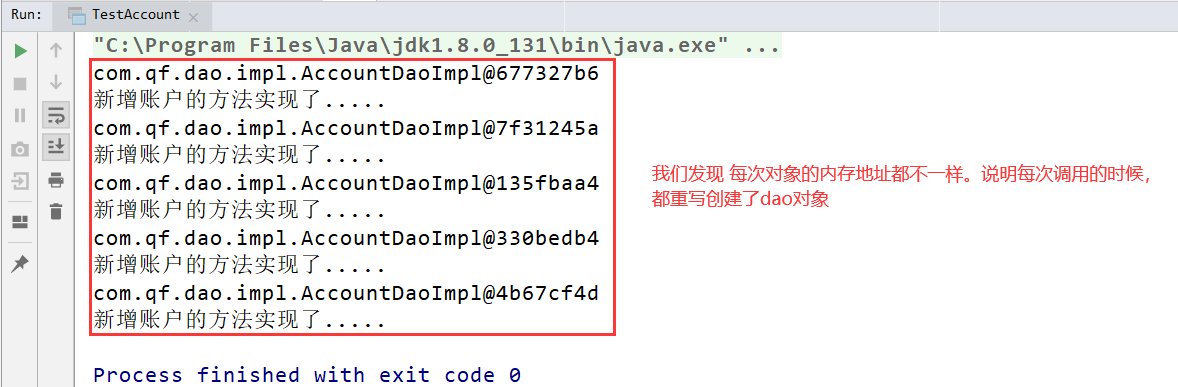
如果我们循环调用5次service会出现什么结果?

改造代码：



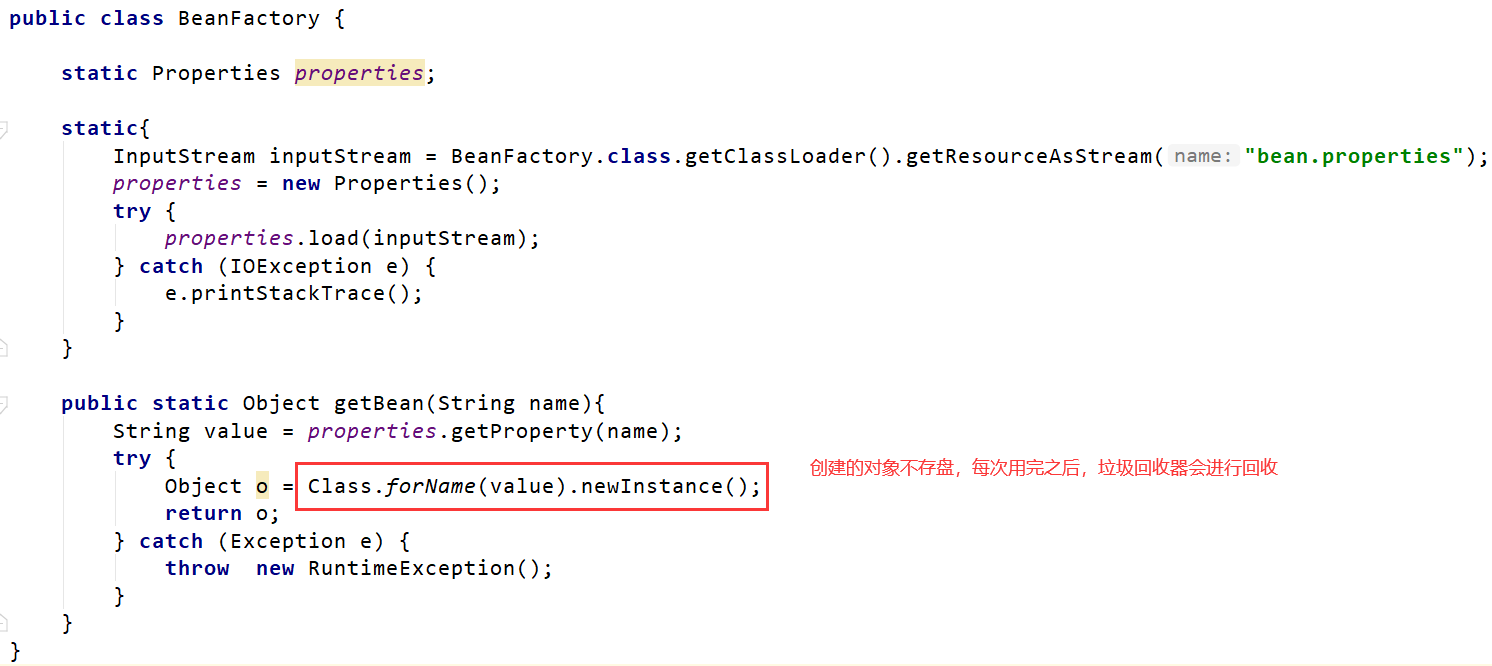


观察控制台打印输出情况:



结论:该业务实现类的对象是多例的。

我们的”工厂类”之所以是多例模式，是因为下面这句代码：



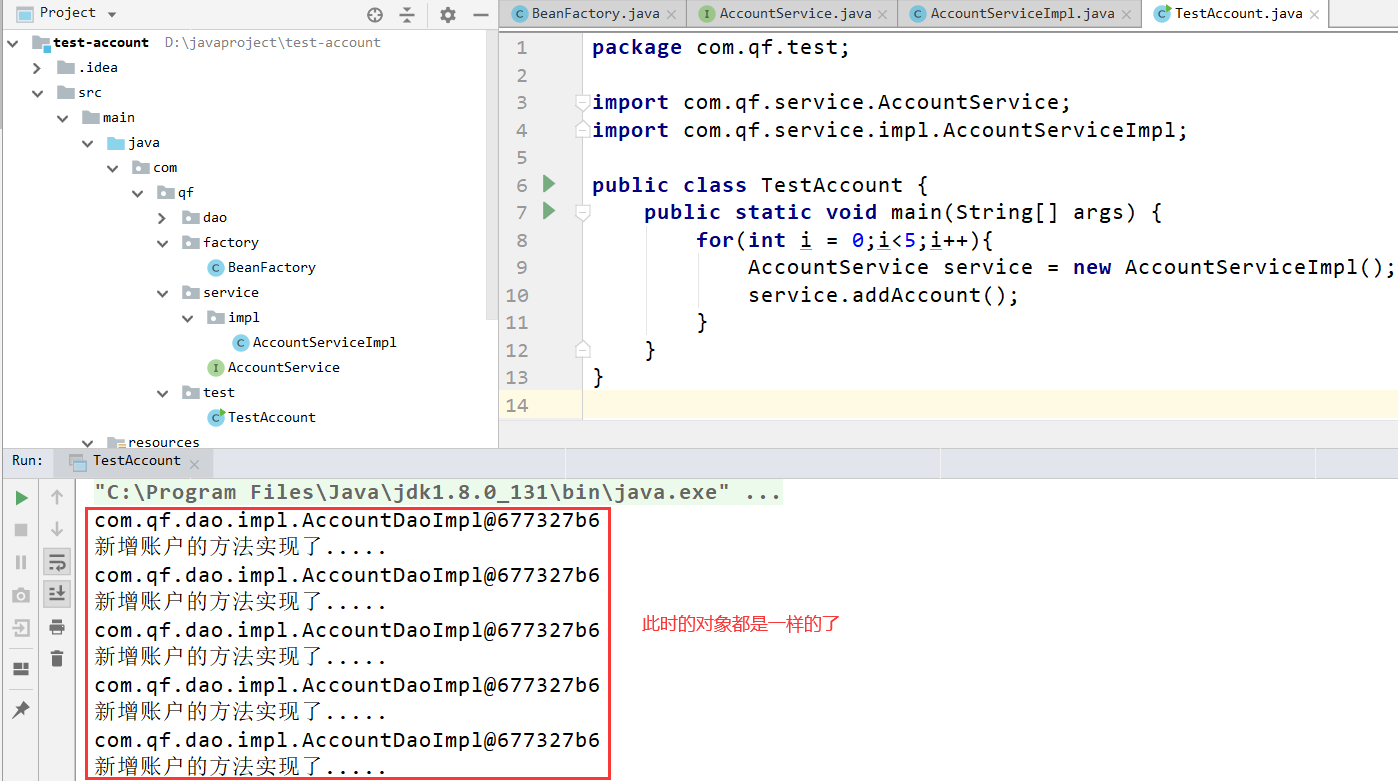
问题:每次调用都要重新创建新的对象，比较耗时。

#### 1.2.2.4 工程模式升级解耦

我们可以通过下面的步骤，把它改造成单例模式。

* 首先我们创建一个容器，用来存放对象





## 1.3 IOC的概念和作用

现在我们是时候揭晓IOC的谜底了。

通过以上分析，我们创建对象的方式有两种：

第一种：

AccountDao dao = new AccountDaoImpl();

第二种：

AccountDao dao = (AccountDao) BeanFactory.*getBean*(**"accountDao"**);

这两种有什么不同?

第一种创建对象的方式是我们主动创建的，控制权在我们手里。但是程序的耦合性高

第二种创建对象的方式是交给工厂帮我们创建的，控制权交给工厂了。这样降低了程序的耦合性。

但是我们每次都自己通过工厂+配置的方式创建对象的过程过于繁琐，那我们看看Spring是如何帮我们做的

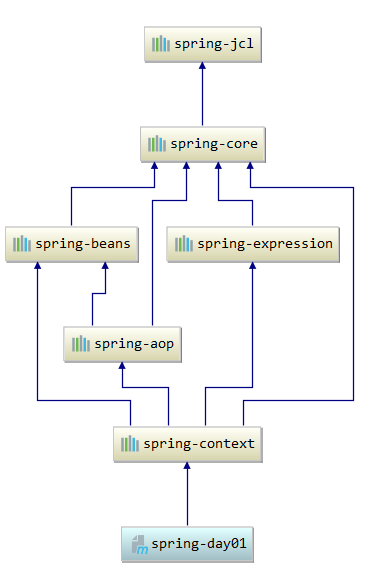
### 1.3.1 基于XML的IOC环境搭建和入门

#### 1.3.1.1 创建一个maven工程,导入spring的依赖

|  |
| --- |
| <**dependencies**>  <**dependency**>  <**groupId**>org.springframework</**groupId**>  <**artifactId**>spring-context</**artifactId**>  <**version**>5.2.0.RELEASE</**version**>  </**dependency**> </**dependencies**> |



我们也可以查询这些依赖的对应关系



#### 1.3.1.2 创建接口和实现类

* AccountDao

|  |
| --- |
| **public interface** AccountDao {  **public void** addAccount(); } |

* AccountDaoImpl

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

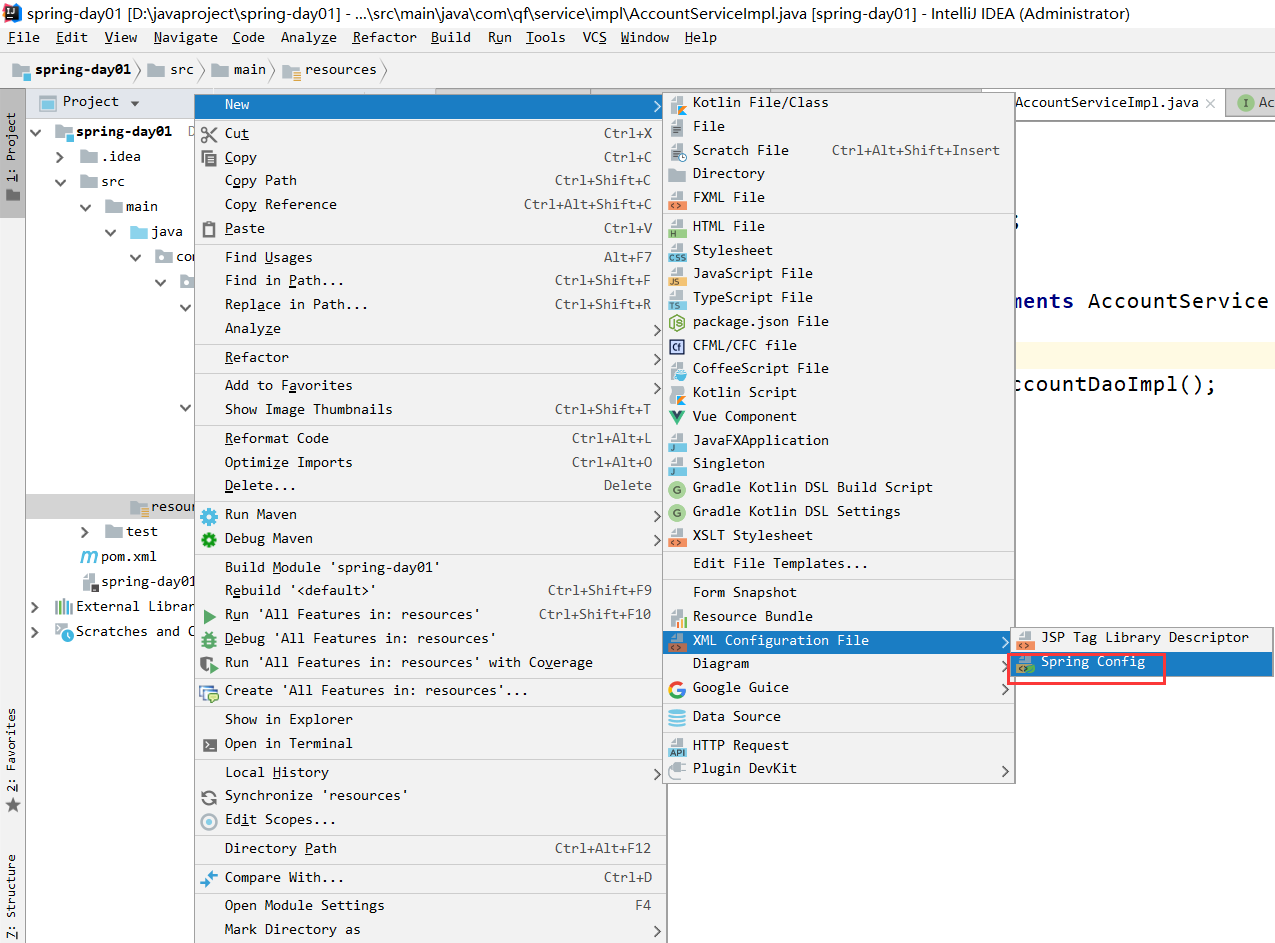
* AccountService

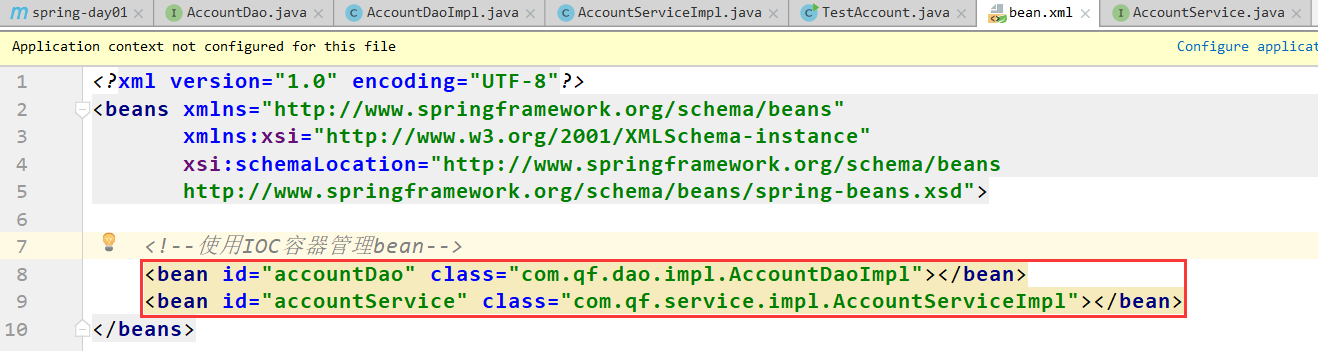
|  |
| --- |
| **public interface** AccountService {  **public void** addAccount(); } |

* AccountServiceImpl

|  |
| --- |
| **public class** AccountServiceImpl **implements** AccountService {  **public void** addAccount() {  AccountDao accountDao = **new** AccountDaoImpl();//高耦合了  accountDao.addAccount();  } } |

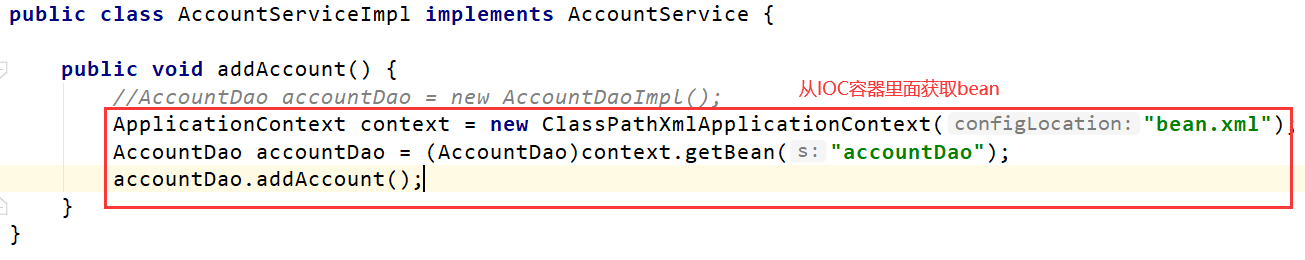
#### 1.3.1.3 创建Spring的配置文件



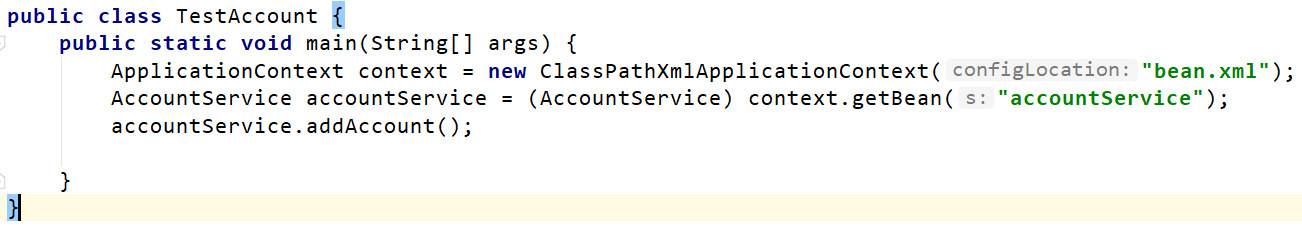


#### 1.3.1.4 测试

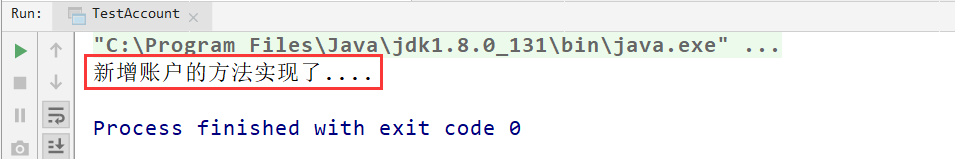
改造AccountServiceImpl



测试类

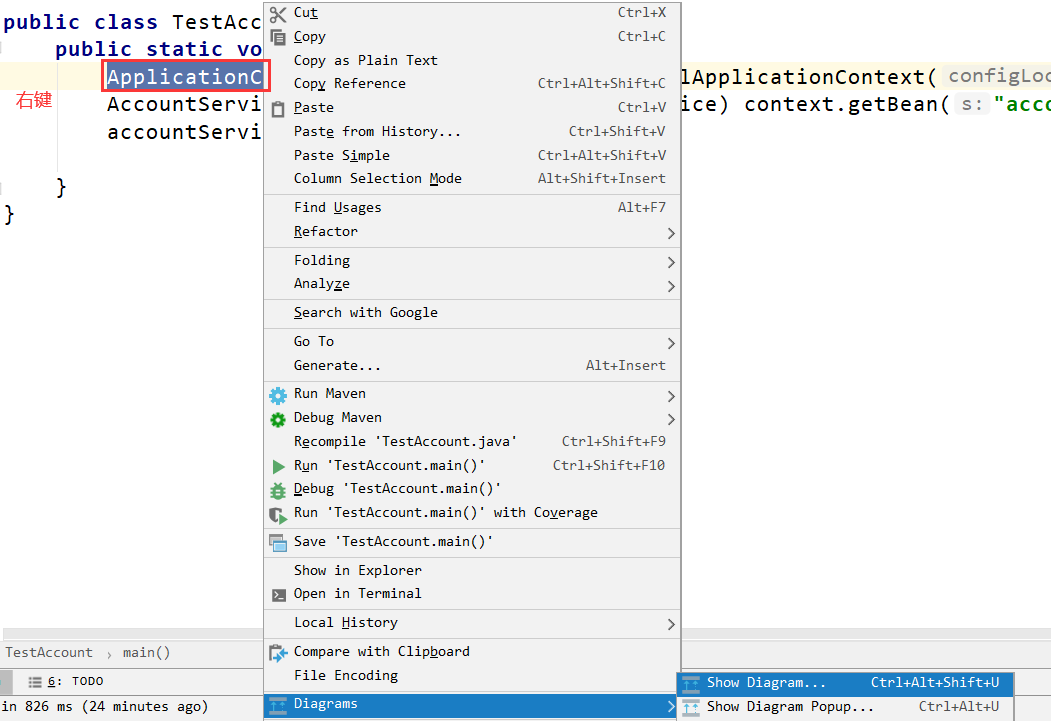


效果如下：

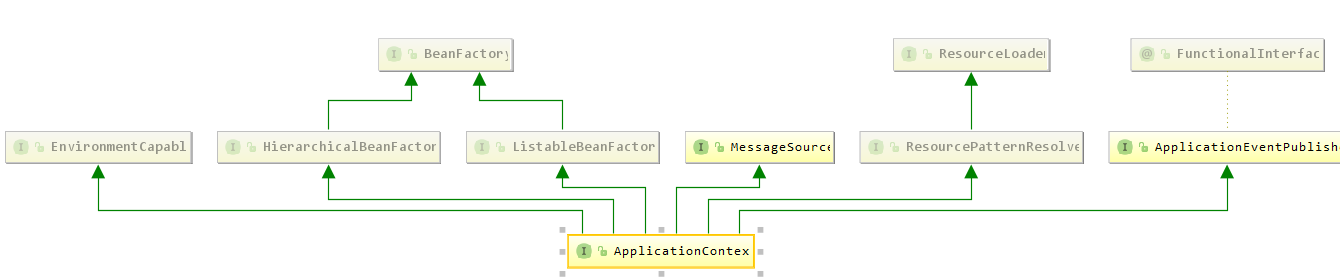


### 1.3.2 详解ApplicationContext

ApplicationContext是Spring给我们提供的核心容器。我们可以去查看它的依赖关系。



此时我们可以详细查看它的依赖关系



我们发现ApplicationContext继承了BeanFactory。BeanFactory才是顶级容器。

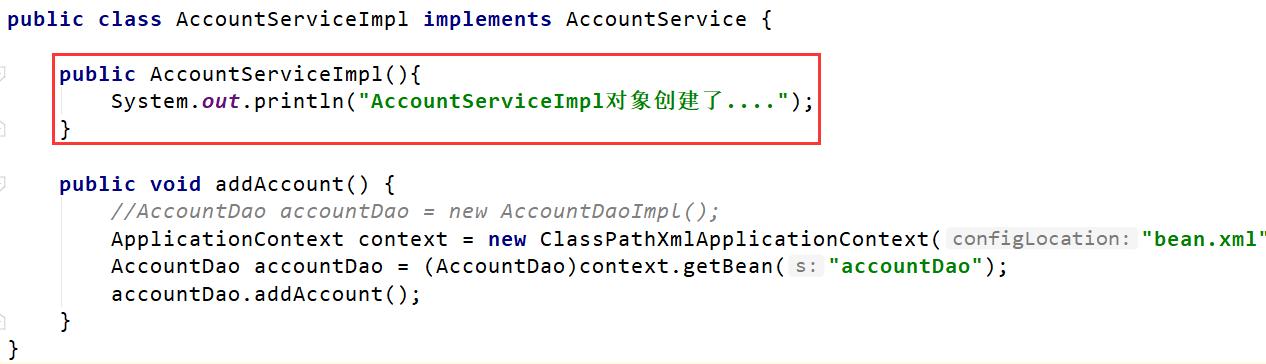
那么这两个核心的容器有什么区别呢?

**1.ApplicationContext**

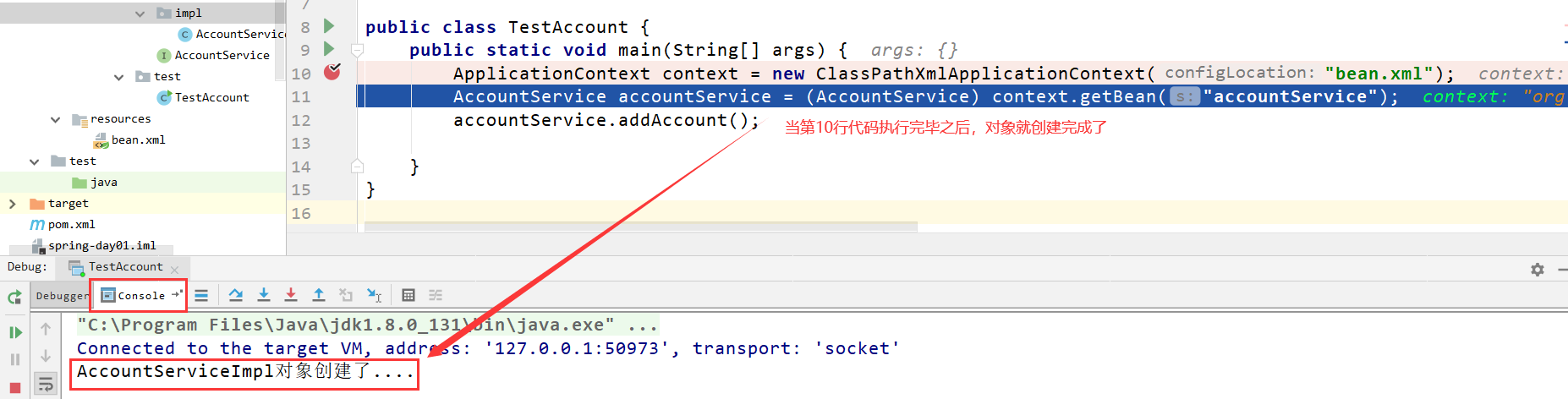
ApplicationContext在构建核心容器时，创建对象采取的策略是采用立即加载的方式。也就是说，只要一读取完配置文件马上就创建配置文件中配置的对象。

我们来证明一下:

* 在AccountServiceImpl上添加一个构造函数



* 然后在测试类上面打断点：

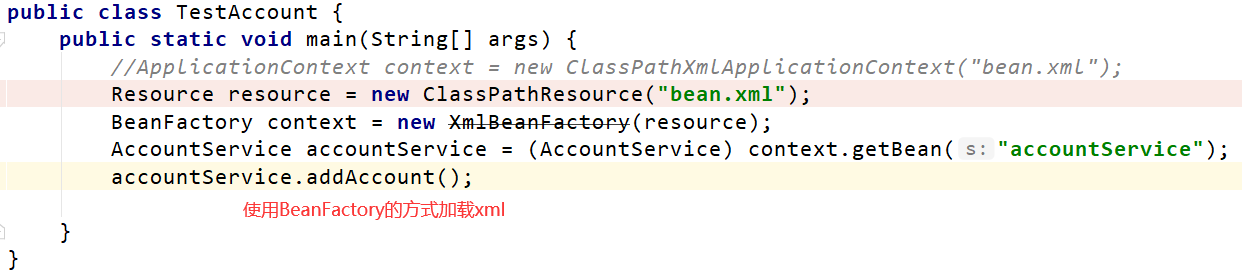


可以看到，当配置文件一旦加载完成，对象就已经创建。

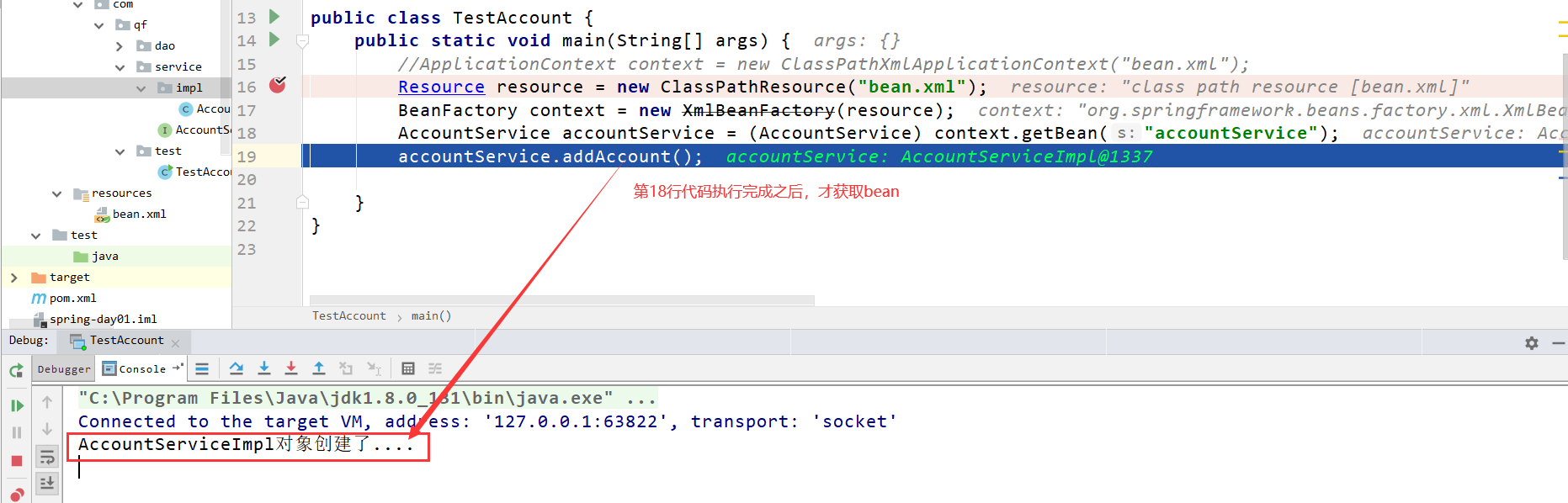
**2.BeanFactory**

BeanFactory它在构建核心容器时，创建对象采取的策略是采用延迟加载的方式,也就是说，什么时候根据id获取对象了，什么时候才真正的创建对象。

* 修改测试类



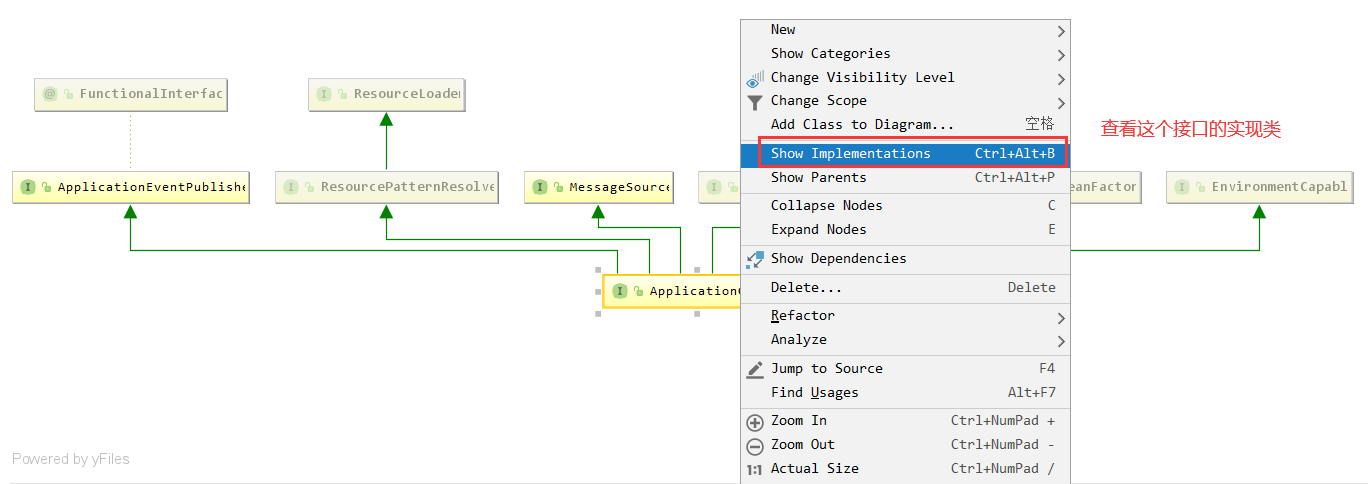
* 打断点测试



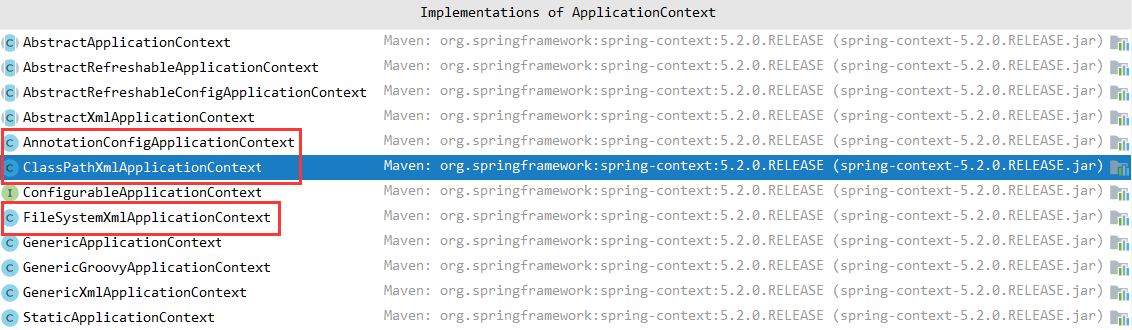
可以看到只有根据id创建对象时，才是真正的创建对象。

最后我们再来看一个问题：

ApplicationContext是一个接口，那么它的实现类呢?



实现类如下:



* ClassPathXmlApplicationContext

它可以加载类路径下的配置文件，要求配置文件必须在类路径下，否则加载不了(这种比较常用)。之前演示的就是ClassPathXmlApplicationContext，这里不再重复演示了。

* FileSystemXmlApplicationContext

它可以加载磁盘任意路径下的配置文件(必须有访问权限）。

* AnnotationConfigApplicationContext

它是用于读取注解创建容器的，后面再介绍。

## 1.4 Spring对bean的管理方式

### 1.4.1 Spring实例化bean的方式

关于spring对bean管理方式的研究，我们重新创建一个工程。

#### 1.4.1.1 无参数构造函数的实例化方式

无参数构造函数实例化方式是Spring默认的bean的实例化方式。

* 创建接口及实现类

|  |
| --- |
| **public interface** UserDao {  **public void** addUser(); } |

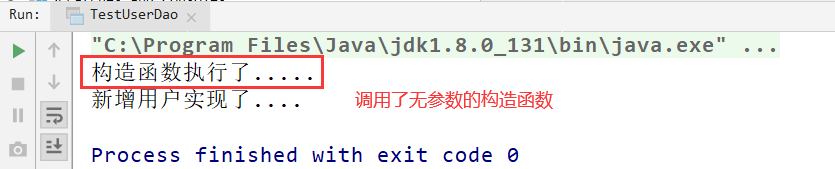
|  |
| --- |
| **public class** UserDaoImpl **implements** UserDao {   **public** UserDaoImpl(){  System.***out***.println(**"构造函数执行了....."**);  }   **public void** addUser() {  System.***out***.println(**"新增用户实现了...."**);  } } |

* 定义配置文件

|  |
| --- |
| *<?***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"**>  <**bean id="userDao" class="com.qf.dao.impl.UserDaoImpl"**></**bean**> </**beans**> |

* 测试

|  |
| --- |
| **public class** TestUserDao {  **public static void** main(String[] args) {  ApplicationContext context = **new** ClassPathXmlApplicationContext(**"bean.xml"**);  UserDao userDao = (UserDao) context.getBean(**"userDao"**);  userDao.addUser();  } } |



#### 1.4.1.2 使用工厂中的普通方法实例化对象

我们设想一个场景：我们要使用别人写好的代码，别人写好的代码通常是封装在一个jar包中的，我们要怎么创建jar包中字节码文件的对象呢？（我们无法通过修改源码的方式来提供默认构造方法）

我们接下来模拟一个工厂类（它可能是存在于jar包中的），通常jar包都会暴露一个工厂类，给调用者创建对象。

* 创建接口以及其实现类

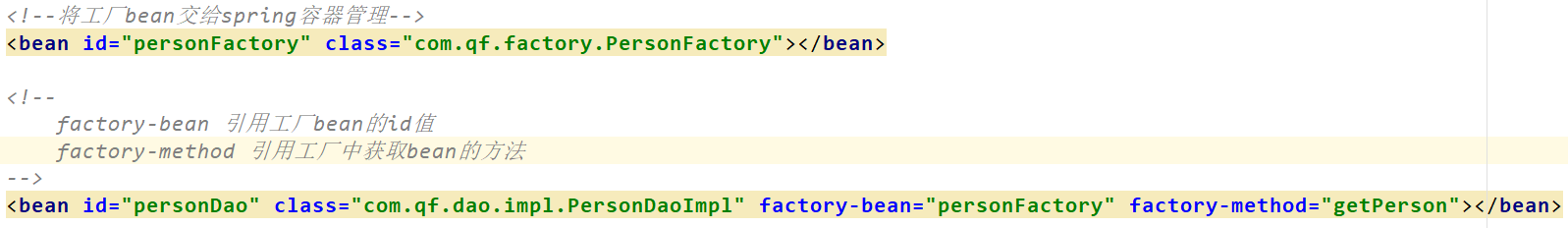
|  |
| --- |
| **public interface** PersonDao {  **public void** addPerson(); } |

|  |
| --- |
| **public class** PersonDaoImpl **implements** PersonDao {  **public void** addPerson() {  System.***out***.println(**"新增person执行了...."**);  } } |

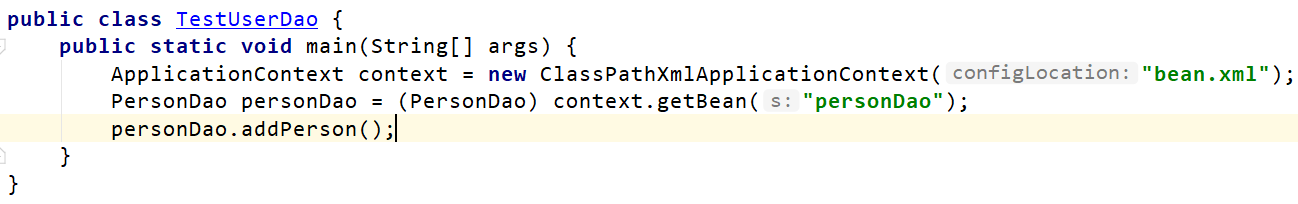
* 创建工厂类

|  |
| --- |
| **public class** PersonFactory {  *//方法的返回值就是需要管理的bean的类型* **public** PersonDao getPerson(){  **return new** PersonDaoImpl();  } } |

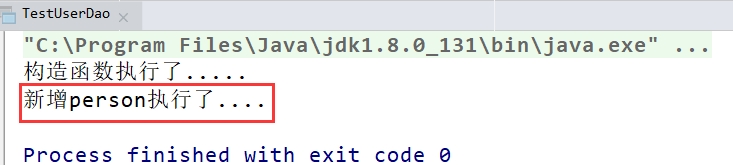
* 配置文件



* 测试类



* 最后测试效果



#### 1.4.1.3 使用工厂中的静态方法实例化对象

* 创建接口以及其实现类

|  |
| --- |
| **public interface** OrderDao {  **public void** addOrder(); } |

|  |
| --- |
| **public class** OrderDaoImpl **implements** OrderDao {  **public void** addOrder() {  System.***out***.println(**"addOrder方法执行了....."**);  } } |

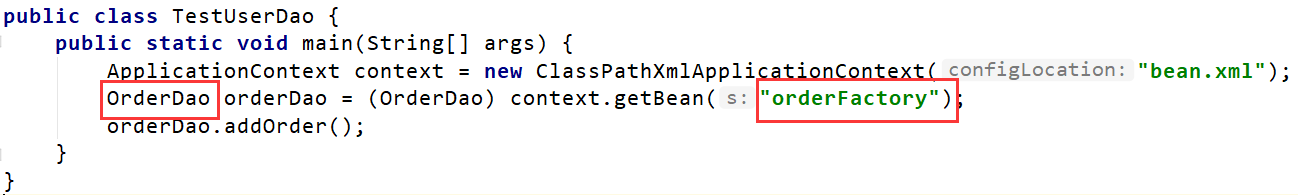
* 创建工厂类

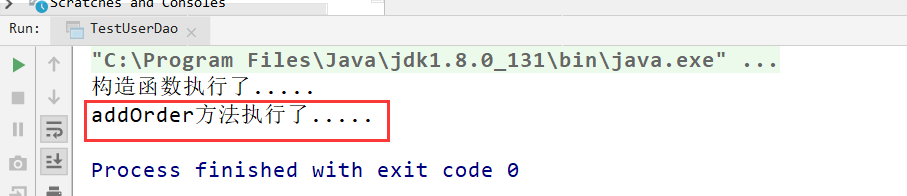
|  |
| --- |
| **public class** OrderFactory {  **public static** OrderDao getOrder(){  **return new** OrderDaoImpl();  } } |

* 配置文件

|  |
| --- |
| <**bean id="orderFactory" class="com.qf.factory.OrderFactory" factory-method="getOrder"**></**bean**> |

* 测试

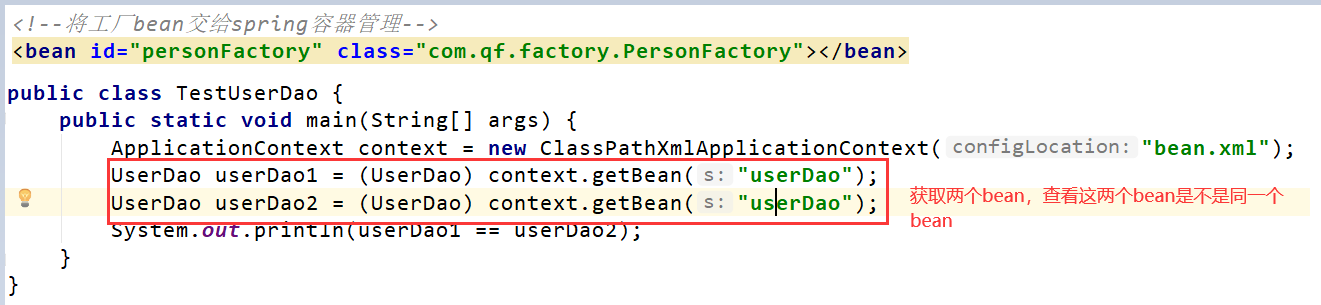




## 1.5 Bean的细节

### 1.5.1 Bean的作用域

在默认情况下，bean是单例的。我们来测试一下。

查看控制台输出效果



那我们要怎么修改bean对象的作用范围呢？

bean标签的scope属性：

作用：用于指定bean的作用范围

取值： 常用的就是单例的和多例的

singleton：单例的（默认值）

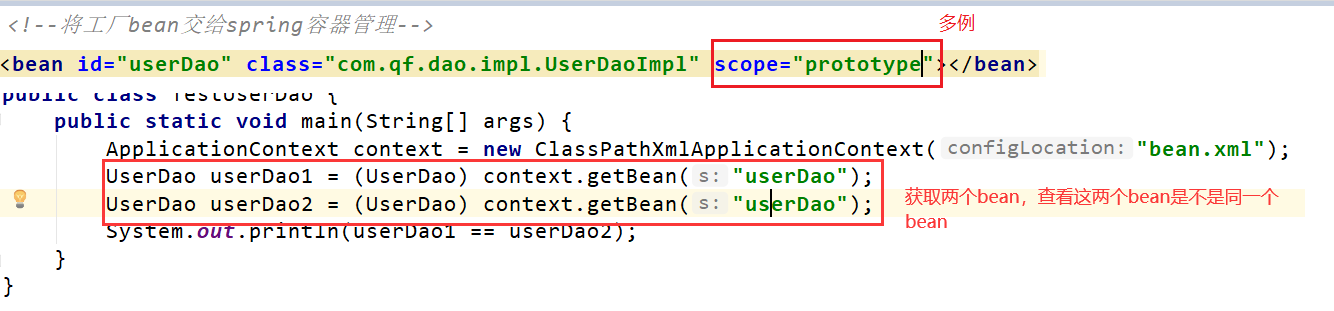
prototype：多例的

request：作用于web应用的请求范围

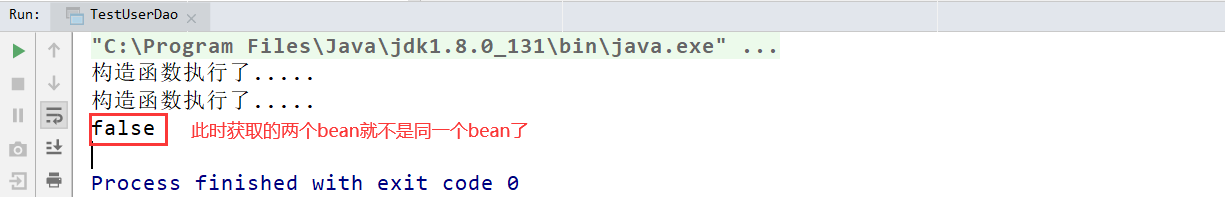
session：作用于web应用的会话范围

global-session：作用于集群环境的会话范围（全局会话范围），当不是集群环境时，它就是session。

此时我们把bean的作用域修改为prototype



再次查看控制台输出效果



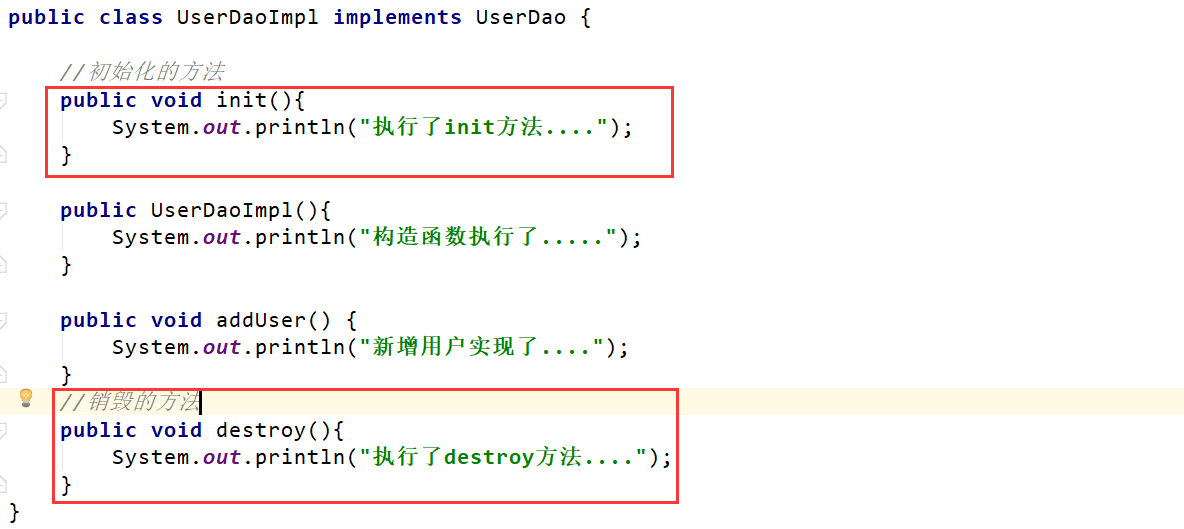
### 1.5.2 Bean的生命周期

**（1） 单例对象**

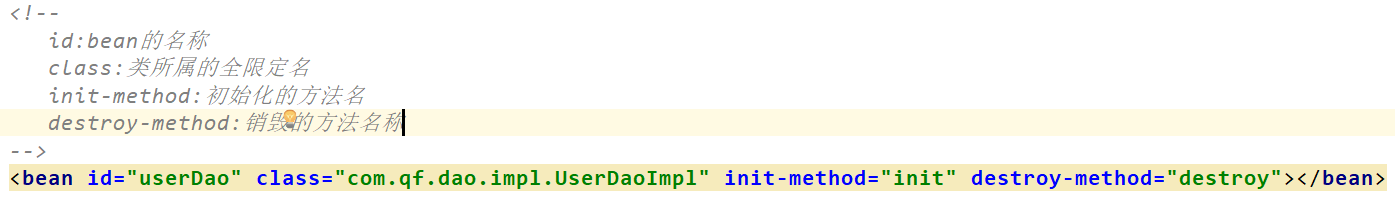
单例对象的生命周期和容器的生命周期是一致的。当容器创建时，对象就实例化好了。当容器还在的时候，对象也就一直存在。当容器销毁，对象也就消亡。

下面来实例单例对象的生命周期：

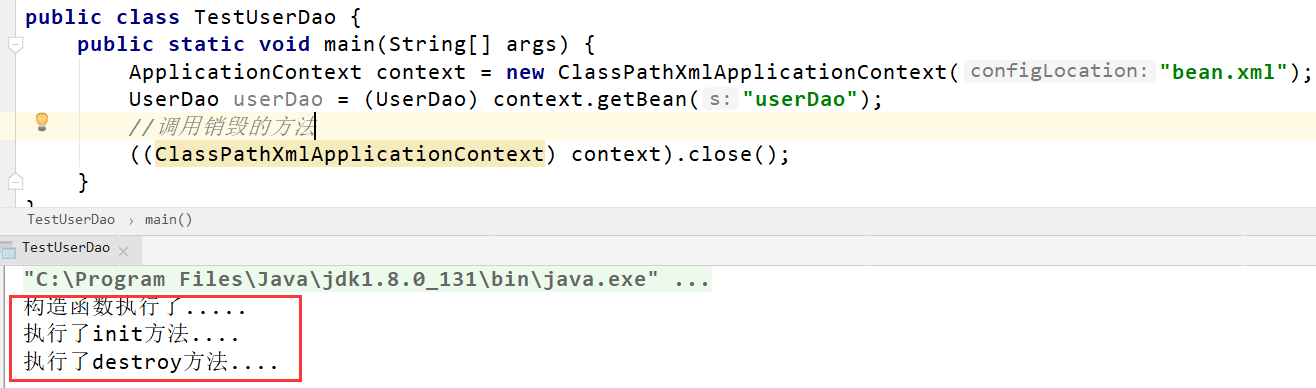
* 改造UserDaoImpl



* 配置文件



* 测试并观察效果



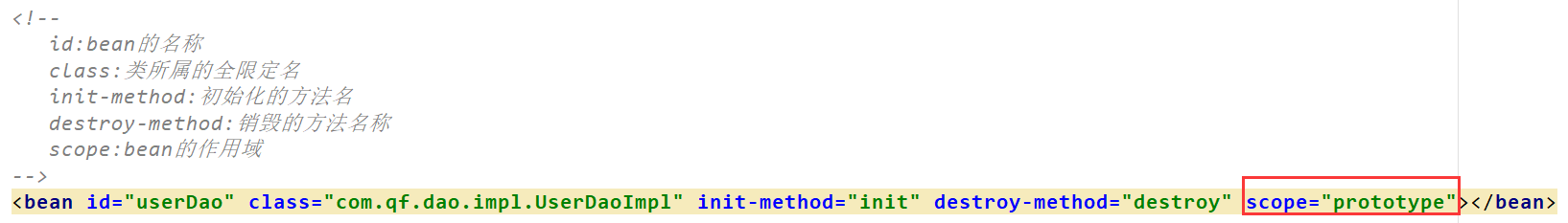
**（2）多例对象**

出生：当我们使用对象时spring框架为我们创建

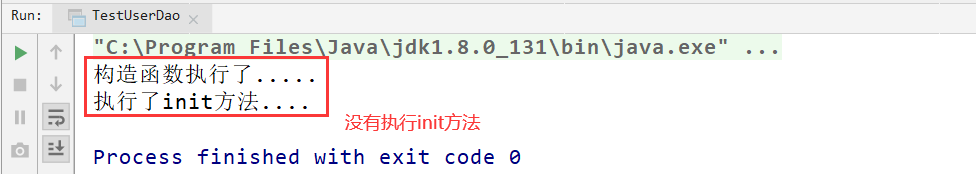
活着：对象只要是在使用过程中就一直活着

死亡：当对象长时间不用，且没有别的对象引用时，由Java的垃圾回收器回收

我们只需要把配置改成多例，其他不变。



观察控制台输出效果



我们发现销毁方法没有被执行，这是为什么呢？

因为spring交给了Java的垃圾回收器进行回收，回收与否不能干涉

具体参考: https://www.cnblogs.com/javazhiyin/p/10905294.html

## 1.6 DI的概念和作用

依赖注入：

Dependency Injection

IOC的作用：

降低程序间的耦合（依赖关系）

DI的作用：依赖关系的管理，依赖都交给spring来维护

在当前类需要用到其他类的对象，由spring为我们提供，我们只需要在配置文件中说明

依赖关系的维护：

就称之为依赖注入

依赖注入：

能注入的数据：有三类

基本类型和String

其他bean类型（在配置文件中或者注解配置过的bean）

复杂类型/集合类型

注入的方式：有三种

第一种：使用构造函数提供

第二种：使用set方法提供

第三种：使用注解提供（后面再介绍）

### 1.6.1 构造函数注入

使用的标签：constructor-arg

标签出现的位置：bean标签的内部

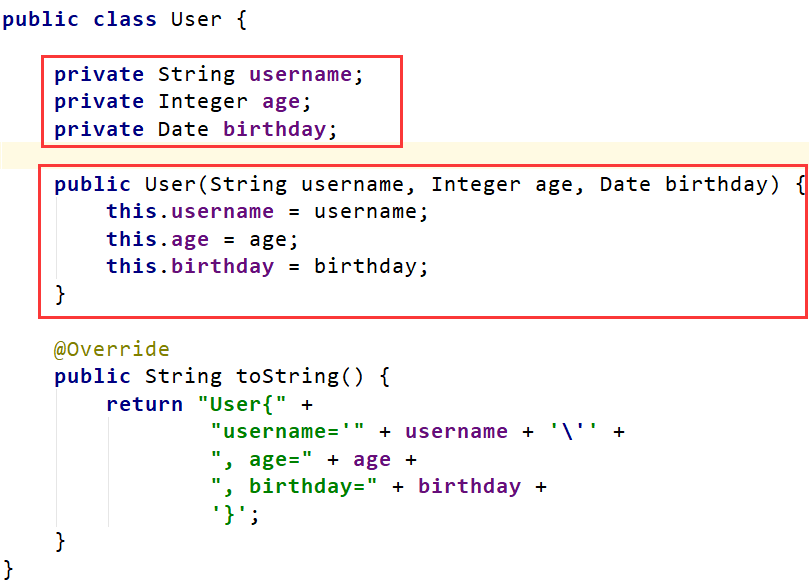
标签中的属性：

name（最常用）：用于指定给构造函数中指定名称的参数赋值

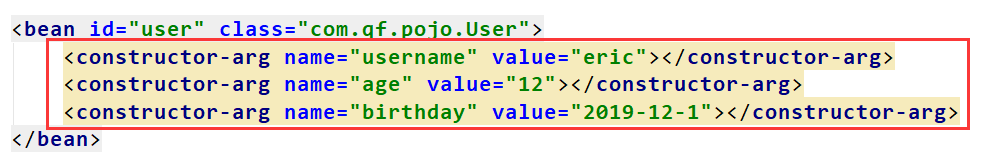
value：用于提供基本类型和String类型的数据

ref：用于指定其他的bean类型数据（它指的就是在spring的ioc核心容器中出现过的bean对象）

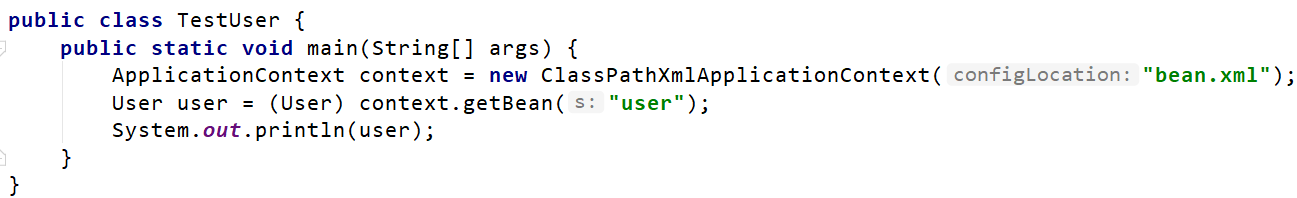
* 创建pojo类



* 配置文件

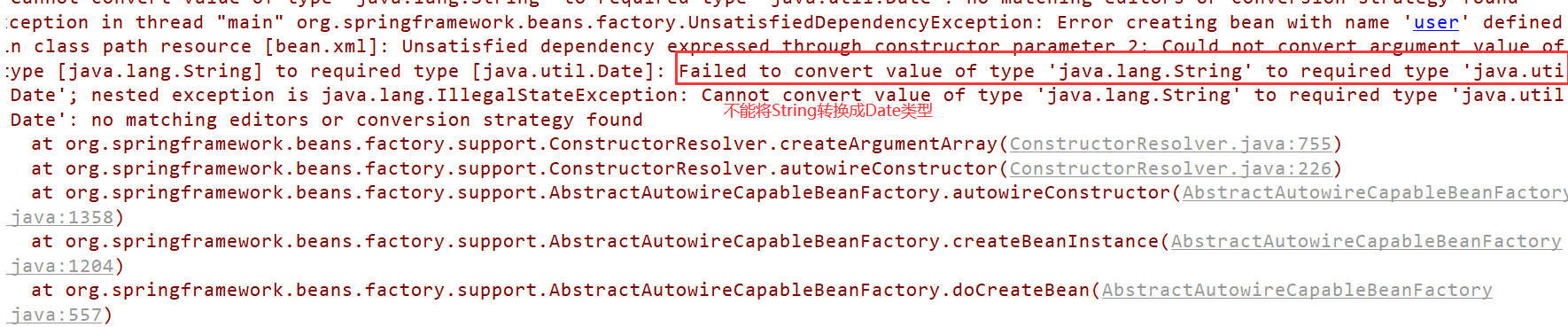


* 测试类



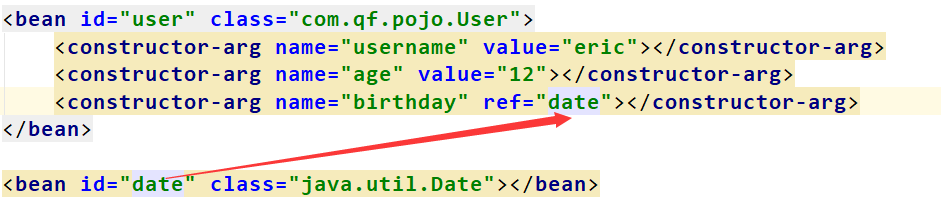
* 效果

发现报错

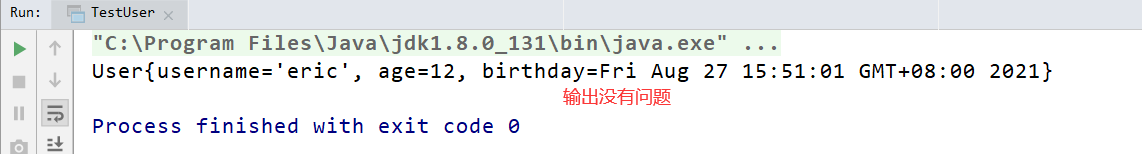


由于xml只能输入字符串，所以spring会自动把String类型的"18"转成Integer类型的18。是对于Date类型，spring就无能为力了，直接运行会报错。

解决方案:



最后查看控制台



**总结：**

优势：

在获取bean对象时，注入数据是必须的操作，否则对象无法创建成功（因为没有默认空参构造）

弊端：

改变了bean对象的实例化方式，使我们在创建对象时，如果用不到这些数据，也必须提供

### 1.6.2 set方法注入(默认的)

涉及的标签：property

出现的位置：bean标签的内部

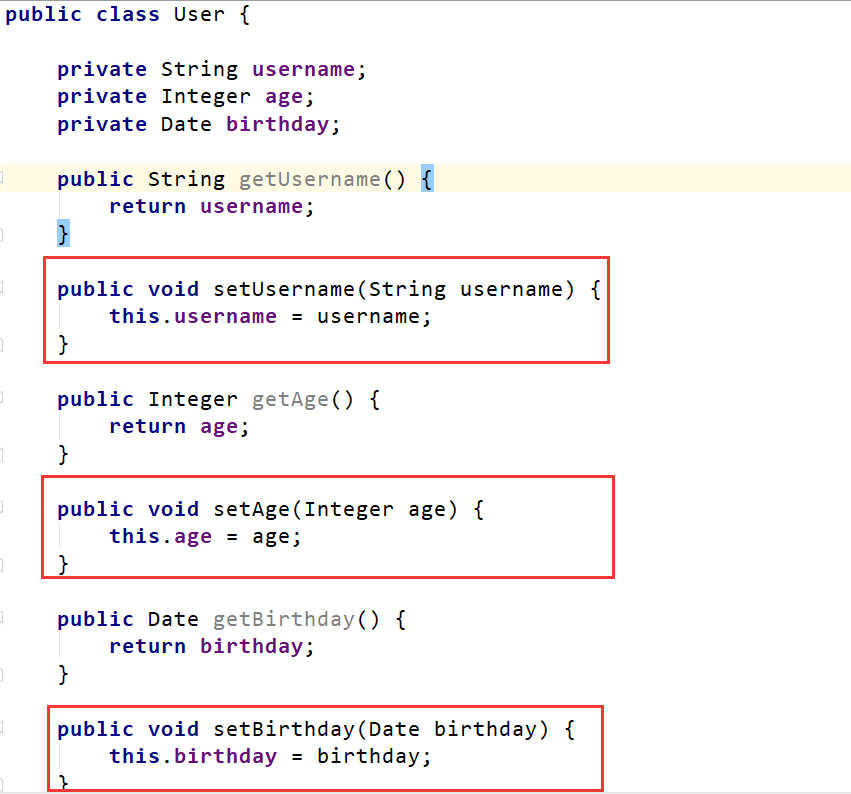
标签的属性：

name：用于指定注入时所调用的set方法名称（如：setName --> name）

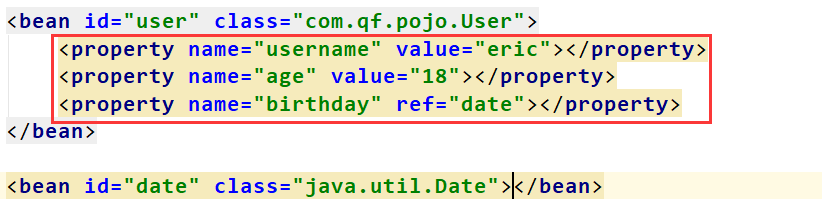
value：用于提供基本类型和String类型的数据

ref：用于指定其他的bean类型数据。它指的就是在spring的ioc核心容器中出现过的bean对象

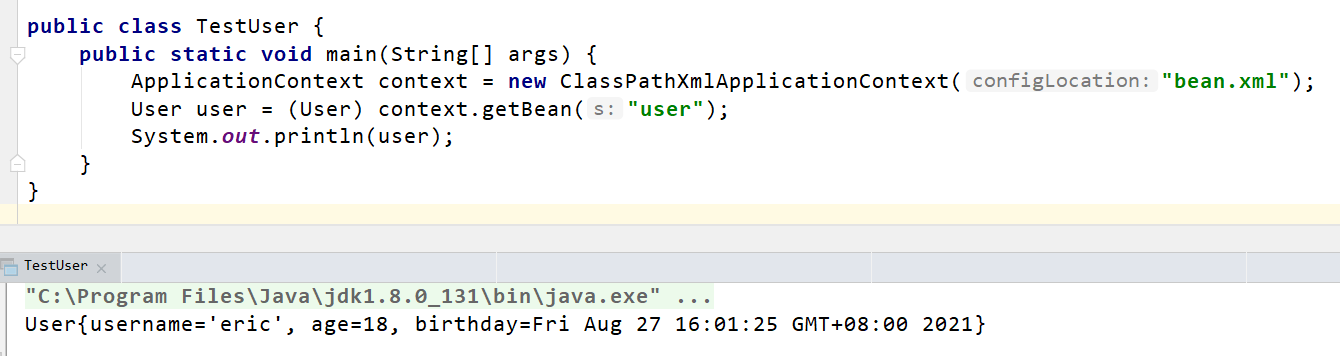
* 创建实体类，并提供set方法



* 配置文件

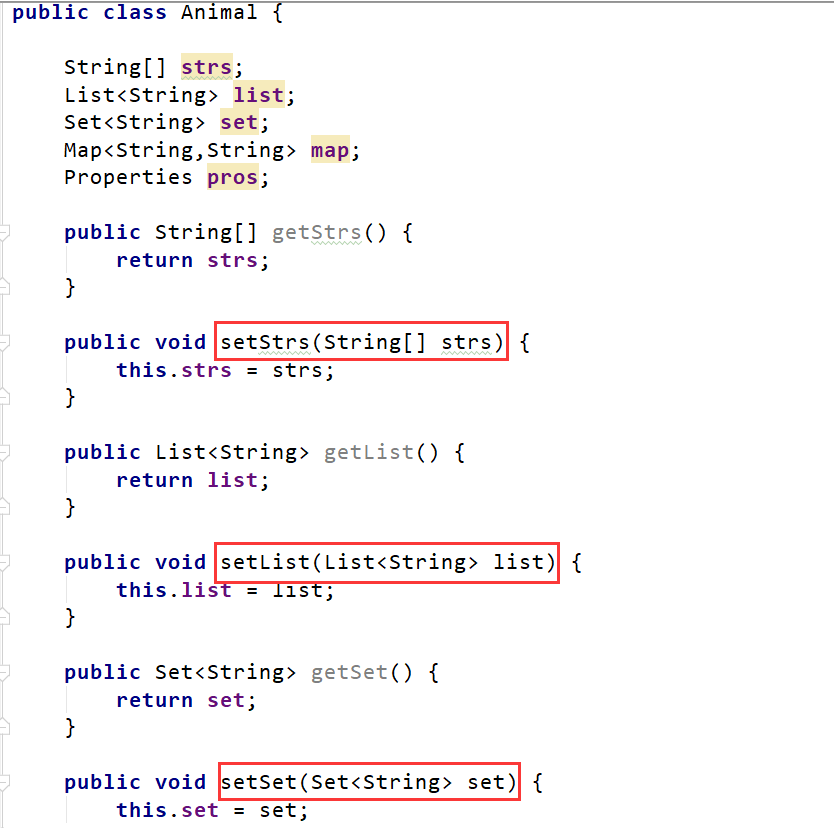


* 运行结果：

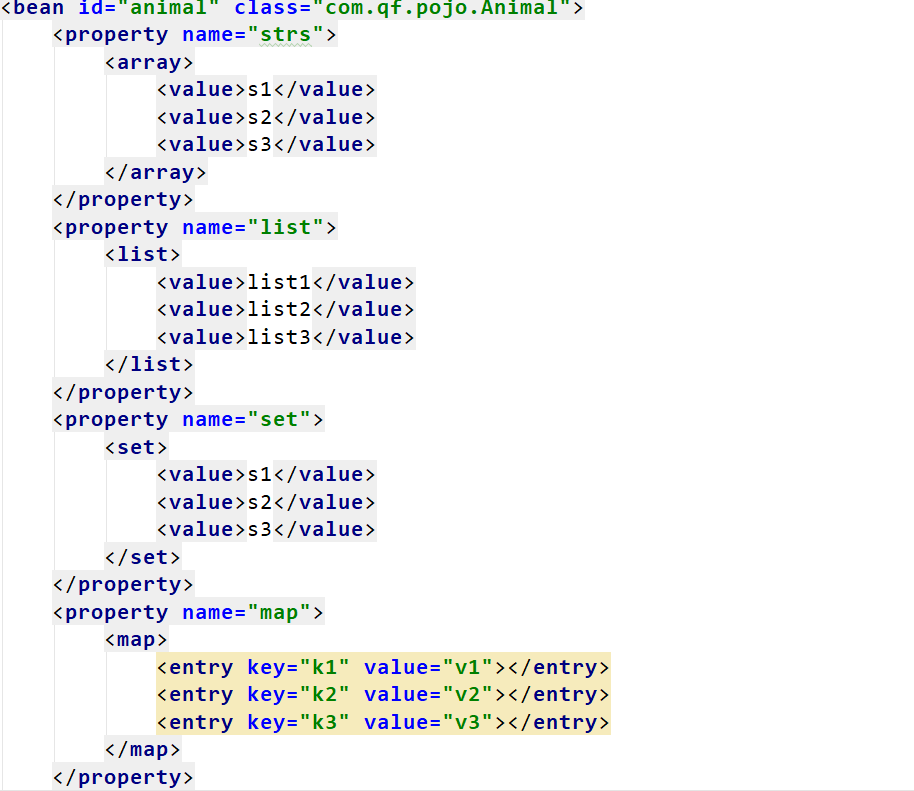


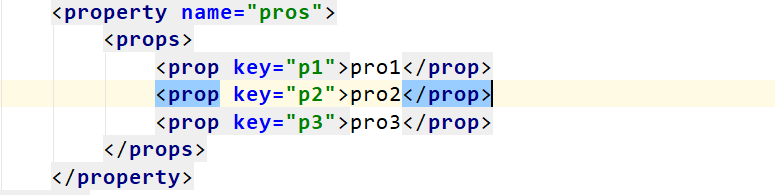
### 1.6.3 复杂类型数据注入

* 创建实体类，并提供set方法



* 创建配置文件





* 运行结果

