# 第七讲Spring（中）

4、Spring API

（1）BeanFactory

BeanFactory是 Spring 的"心脏"。

BeanFactory是 IOC 容器的核心接口，它定义了IOC的基本功能。

Spring使用它来配置文档，管理bean的加载，实例化并维护bean之间的依赖关系，负责bean的声明周期。

（2）ApplicationContext

ApplicationContext由BeanFactory派生而来，可以比喻为Spring的躯体。

ApplicationContext在BeanFactory的基础上添加了很多功能：

支持了aop功能和web应用 MessageSource, 提供国际化的消息访问

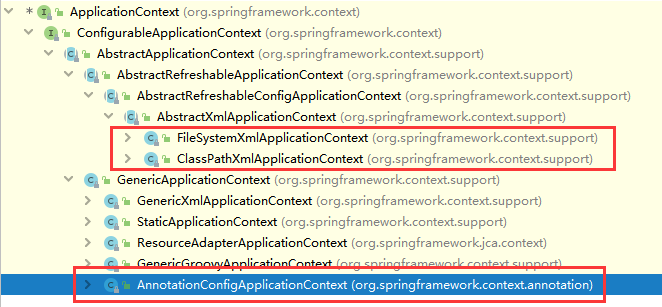
通过配置来实现BeanFactory中很多编码才能实现的功能

ApplicationContext的常用实现类

ClassPathXmlApplicationContext: 从classpath目录读取配置文件

FileSystemXmlApplicationContext:从文件系统或者url中读取配置文件 AnnotationConfigApplicationContext：当我们使用注解配置容器对象时，需要使用此类来创建 Spring 容器。它用来读取注解。

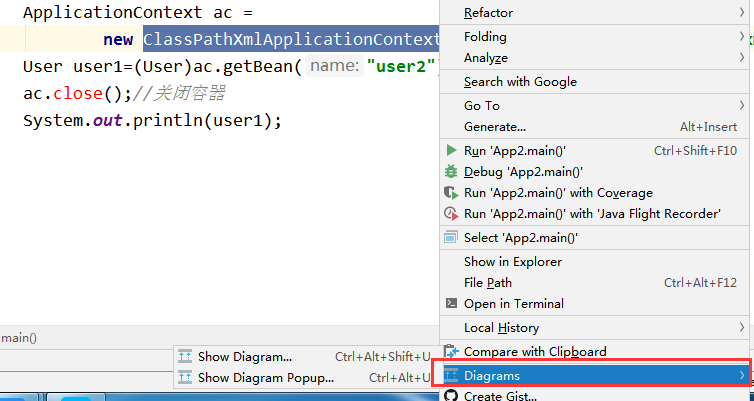
双击选中ApplicationContext ，ctrl+h

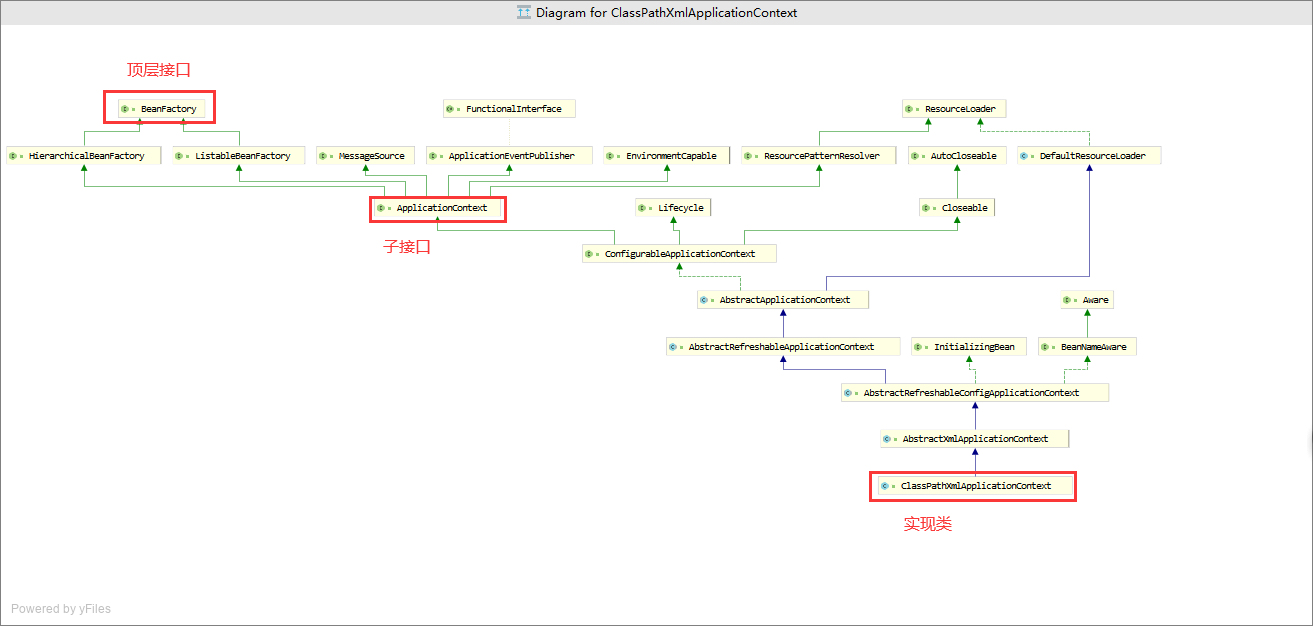


（3）两者区别:

beanFactory主要是面向Spring框架的基础设施，也就是供Spring自身内部调用， 而Applicationcontext 主要面向Spring的使用者。

BeanFactroy在第一次使用到某个Bean时(调用getBean())，才对该Bean进行加载实例化， 而ApplicationContext是在容器启动时，一次性创建并加载了所有的Bean。



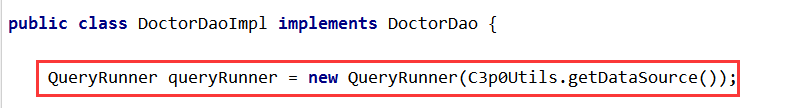


## Spring 整合 DbUtils

### （一）Spring 整合 DbUtils

DbUtils是Apache的一款用于简化Dao代码的工具类，它底层封装了JDBC技术。 核心类：QueryRunner 用于执行增删改查的SQL语句 ResultSetHandler 这是一个接口，主要作用是将数据库返回的记录封装进实体对象

以前



把QueryRunner queryRunner交给IOC容器

下午做转帐

1、导入依赖的坐标

<dependency>

<groupId>commons-dbutils</groupId>

<artifactId>commons-dbutils</artifactId>

<version>1.7</version>

</dependency>

<dependency>

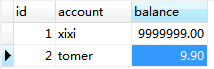
<groupId>com.alibaba</groupId>

<artifactId>druid</artifactId>

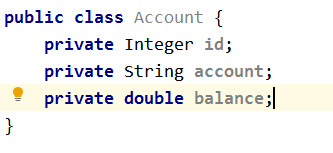
<version>1.1.15</version>

</dependency>

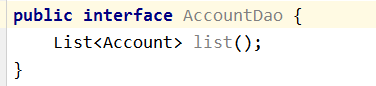
2、创建数据表

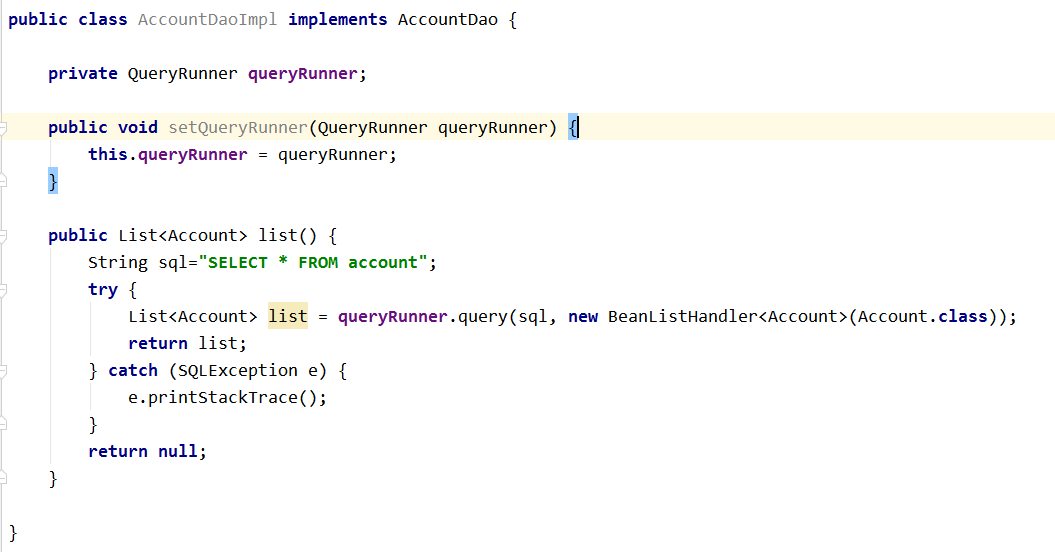


3、创建实体类

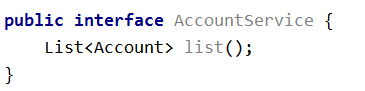


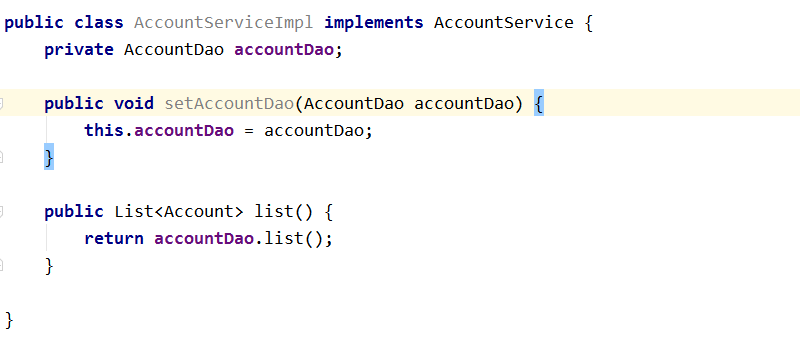
4、编写持久层代码



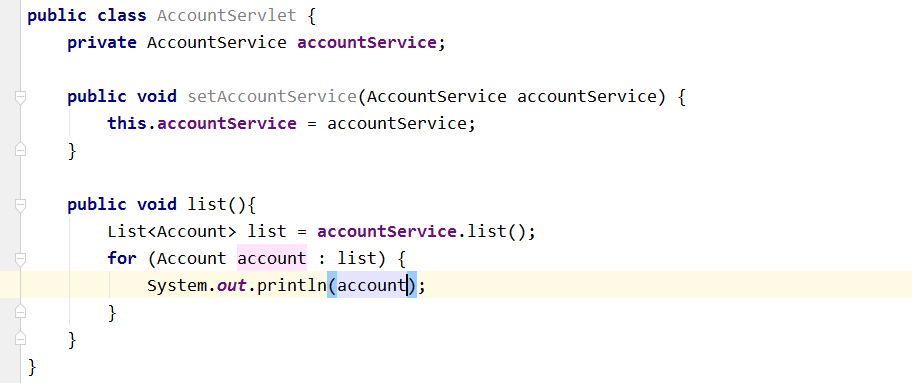


5、编写业务层代码





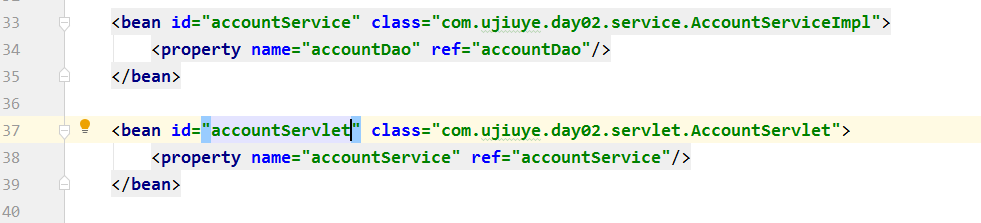
5、编写表现层代码



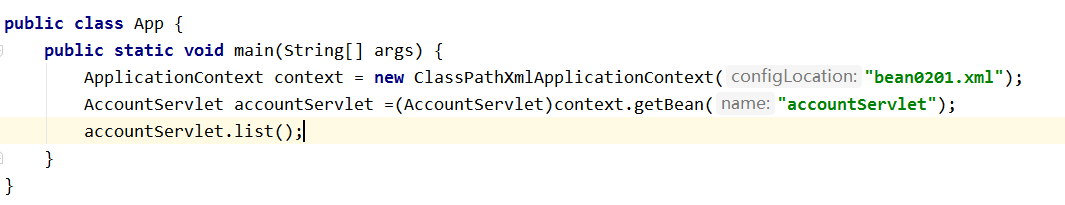
6、创建配置文件

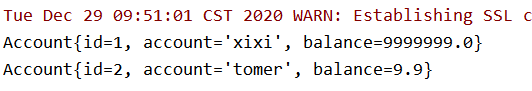
|  |
| --- |
| *<?***xml version="1.0" encoding="UTF-8"***?>* <**beans xmlns="http://www.springframework.org/schema/beans"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xmlns:context="http://www.springframework.org/schema/context"  xmlns:aop="http://www.springframework.org/schema/aop"  xmlns:tx="http://www.springframework.org/schema/tx"  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 http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop.xsd http://www.springframework.org/schema/tx http://www.springframework.org/schema/tx/spring-tx.xsd "**> |





7、测试代码





## 二、Spring整合各种数据源（dataSource）

Spring中配置数据源非常简单，只要两步：

1、引入数据源坐标

2、在配置文件中配置DataSource的bean

### （一）整合C3P0数据源

1、添加依赖坐标：

<dependency>

<groupId>c3p0</groupId>

<artifactId>c3p0</artifactId>

<version>0.9.1.2</version>

</dependency>

2、在Spring配置文件中添加配置内容

|  |
| --- |
| <**bean id="c3p0DataSource" class="com.mchange.v2.c3p0.ComboPooledDataSource"**>  <**property name="driverClass" value="${jdbc.driver}"**/>  <**property name="jdbcUrl" value="${jdbc.url}"**/>  <**property name="user" value="${jdbc.username}"**/>  <**property name="password" value="${jdbc.password}"**/> </**bean**> |

### （二）Spring-jdbc 自带数据源

Spring自带的数据源，没用到池技术，适合开发、测试环境使用

1、添加依赖坐标

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-jdbc</artifactId>

<version>5.1.16.RELEASE</version>

</dependency>

2、在配置文件中添加数据源的配置

|  |
| --- |
| <**bean id="springJdbcDataSource" class="org.springframework.jdbc.datasource.DriverManagerDataSource"**>  <**property name="driverClassName" value="com.mysql.jdbc.Driver"**/>  <**property name="url" value="jdbc:mysql://127.0.0.1:3306/mybatis"**/>  <**property name="username" value="root"**/>  <**property name="password" value="root"**/> </**bean**> |

### （三）阿里巴巴数据源 Druid

1、添加依赖坐标

<dependency>

<groupId>com.alibaba</groupId>

<artifactId>druid</artifactId>

<version>1.1.15</version>

</dependency>

2、在配置文件中添加数据源的配置

|  |
| --- |
| <**bean id="AlibabaDataSource" class="com.alibaba.druid.pool.DruidDataSource"**>  <**property name="driverClassName" value="${jdbc.driver}"**/>  <**property name="url" value="${jdbc.url}"**/>  <**property name="username" value="${jdbc.username}"**/>  <**property name="password" value="${jdbc.password}"**/> </**bean**> |

### （四）DBCP数据源

1、添加依赖坐标

<dependency>

<groupId>commons-dbcp</groupId>

<artifactId>commons-dbcp</artifactId>

<version>1.4</version>

</dependency>

2、在配置文件中添加数据源的配置信息

|  |
| --- |
| <**bean id="dbcpDataSource" class="org.apache.commons.dbcp.BasicDataSource"**>  <**property name="driverClassName" value="com.mysql.jdbc.Driver"**/>  <**property name="url" value="jdbc:mysql://127.0.0.1:3306/mybatis"**/>  <**property name="username" value="root"**/>  <**property name="password" value="root"**/> </**bean**> |

### （五）HikariCP数据源

1、添加依赖坐标

<dependency>

<groupId>com.zaxxer</groupId>

<artifactId>HikariCP</artifactId>

<version>1.3.5</version>

</dependency>

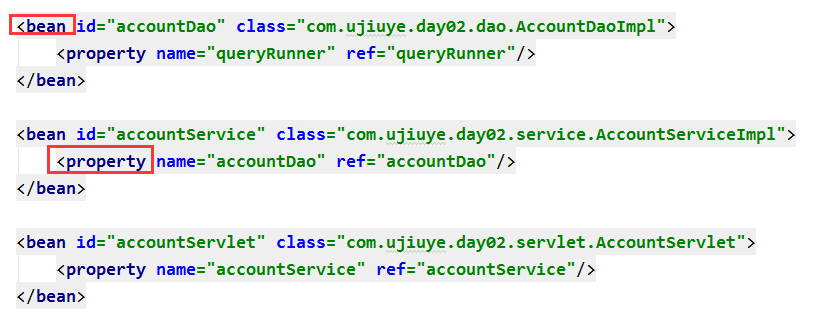
2、在配置文件中添加数据源的配置信息

|  |
| --- |
| <**bean id="hikariCPDataSource" class="com.zaxxer.hikari.HikariDataSource"**>  <**property name="driverClassName" value="com.mysql.jdbc.Driver"**/>  <**property name="jdbcUrl" value="jdbc:mysql://127.0.0.1:3306/mybatis"**/>  <**property name="username" value="root"**/>  <**property name="password" value="root"**/> </**bean**> |

## 三、基于注解的IOC配置(重点重点)

### (一)常用注解介绍

IOC/di



内容多了后不好维护，不用写，用注解（工作中主要的方式）

注解分为两类：

DI依赖注入（属性赋值）

IOC对象创建

1、依赖注入的注解

@Autowired 按照类型注入。

相当于配置文件中的< property name="" ref="">

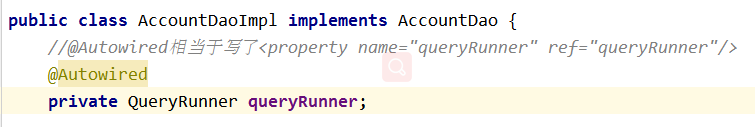
当使用注解注入属性时，set 方法可以省略。

当要注入的属性存在多个实现类时会报错。

##### 1.1@Autowired

修改AccountDaoImpl

**1、注解在属性上（推荐）**

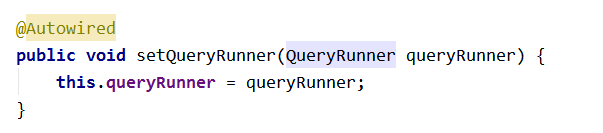


1以QueryRunner queryRunner属的类型为条件，去到容器中找这个类的对象

2、找到了，直接通过反射技术把找到的这个对象赋给这个属性

3、找不到，异常

2、**注解在set方法上（了解）**



1、获取set方法的参数类型，以这个类型为条件，去到容器中找这个类的对象

2、找到了，调用set方法，把找到的这个对象赋给形参，完成赋值

3、找不到，异常

修改配置文件

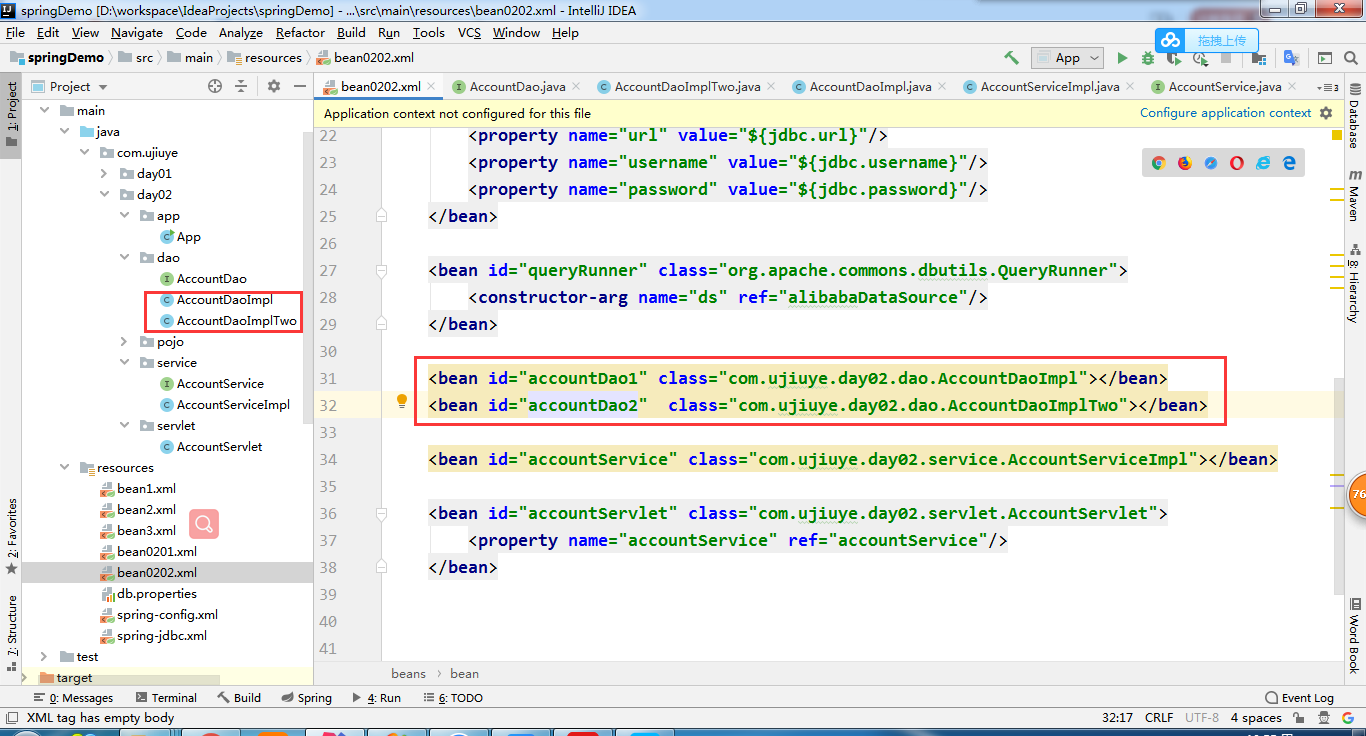
|  |
| --- |
| <**context:annotation-config** /> |

重新运行

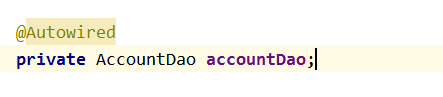
##### 1.2@Autowired

当要注入的属性存在多个实现类时会报错。

复制AccountDaoImpl，为AccountDaoImplTwo并且在配置文件装配一下



修改AccountService

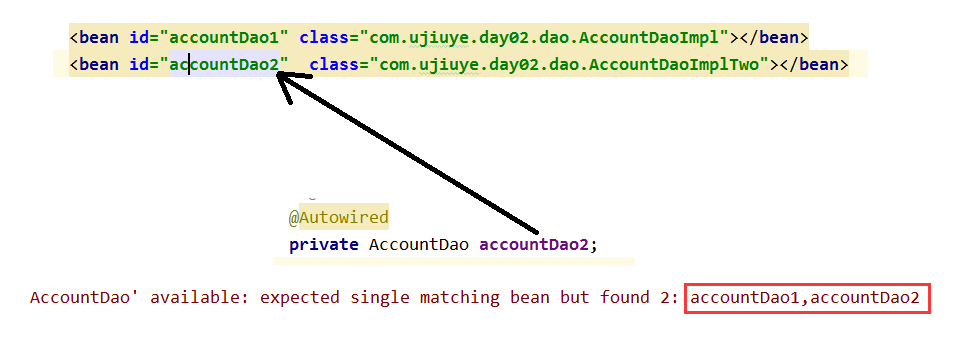


重新运行



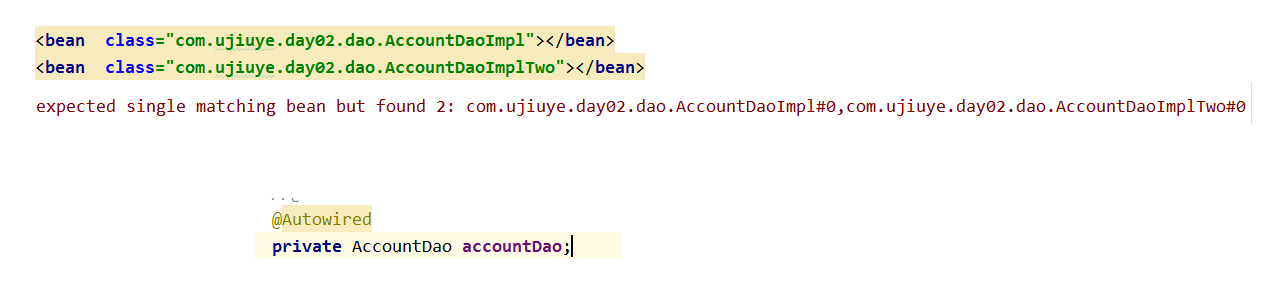
怎么办？同一个类

如果【配置文件】中写了id,那么Autowired按类型找到两类型，再以属性名作为条件找，找到就赋值：



如果不写id，默认名称会带上包名

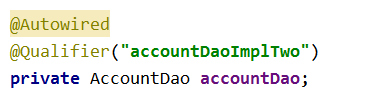




##### 1.3@Qualifier

在@Autowired注入的基础之上，再按照 Bean 的 id 注入。

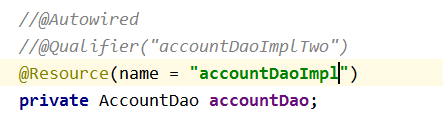
一般在@Autowired注入存在多个实现类的时候，使用@Qualifier按照bean的id选取



##### 1.4@Resource

直接按照 Bean 的 id 注入

1、指定name（有多个实现类时，一般不要指定名称，增加容错率）

  
@Resource(name = "accountDaoImpl")相当于同时写了@Autowired和@Qualifier("accountDaoImplTwo")

1. 以name的值为条件去容器找，找到就直反射赋值
2. 找不到不异常

2、不指定name（常用）



1、以属性名accountDao为条件，去容器找，找到就直反射赋值

2、找不到退回按类型找，获取属的类型为条件，去到容器中找这个类的对象

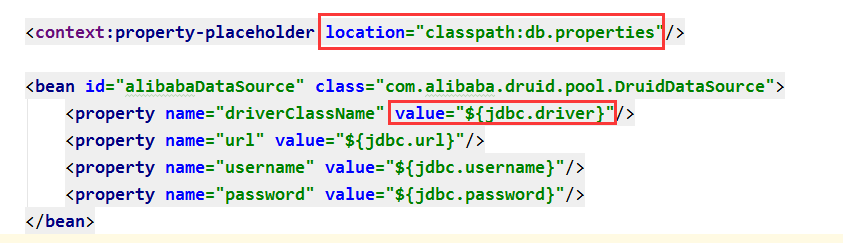
2、找到了，直接通过反射技术把找到的这个对象赋给这个属性

3、找不到，异常

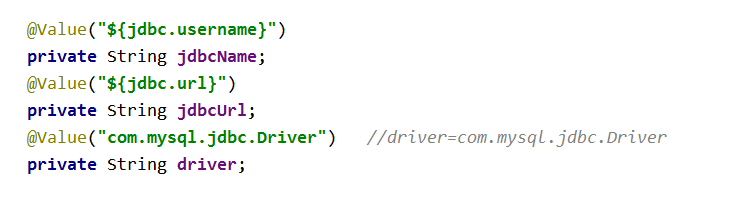
##### 1.5@Value

用于简单数据类型的注入，相当于< property name="" value="" >

在配置文件中读取xxxxx.properties的内容



如果我们想程序中（java代码）中读取



注意：一定要引文件

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

##### 1.6总结：

@Autowired和@Resource 是去容器中找已经的bean（对象-自定义的引用类型）

@Value 基本的数据类（读取xxxx.properties）

2、对象实例化注解

@Component

用于实例化对象，相当于配置文件中的< bean id="" class=""/> 它支持一个属性value,相当于xml中bean的id。如果不写，默认值为类名的首字母小写

@Controller

@Service

@Repository

这4个注解的功能跟完全一样，只不过下面三个比较有语义化。

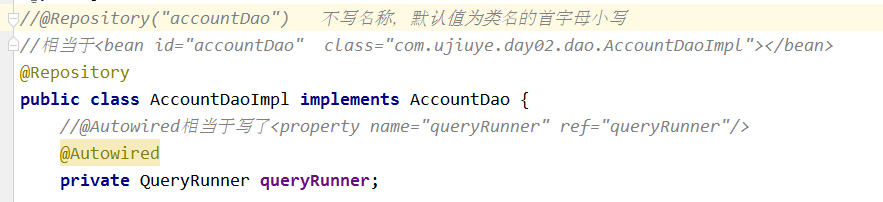
@Controller 一般标注在表现层的类上

@Service 一般标注在业务层的类上

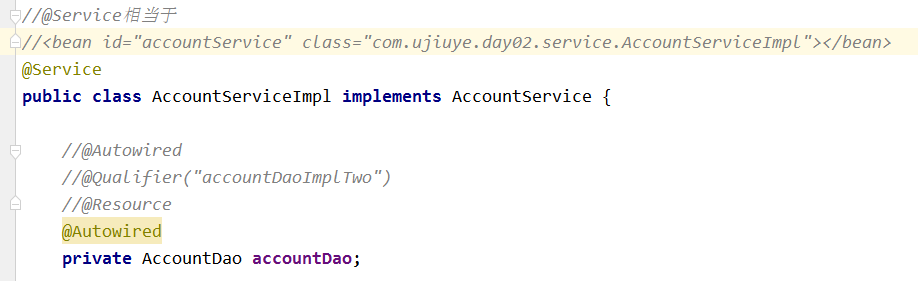
@Repository 一般标注在持久层的类上

推荐使用这三个，当一个类实在不好归属在这三个层上时，再使用@Component！！！

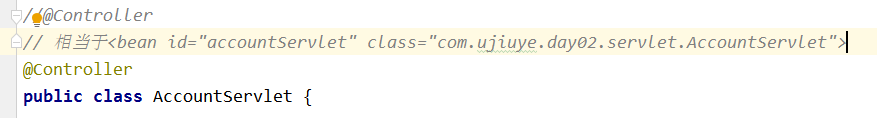
修改daoImpl



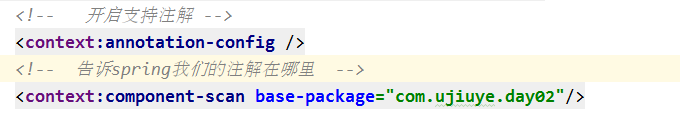
修改service



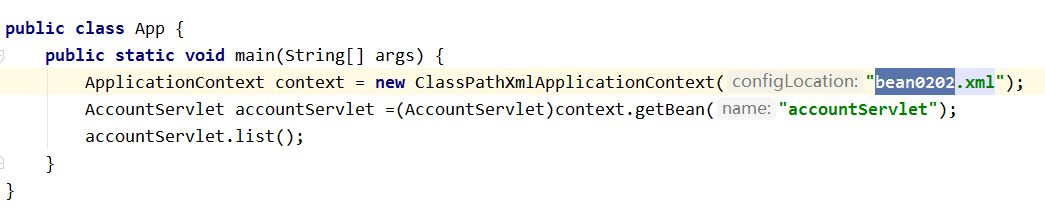
修改controller



最后：告诉spring我们的注解在哪里，好让spring在启动容器去找到这个注解，并把带有注解的类创建对应对象。



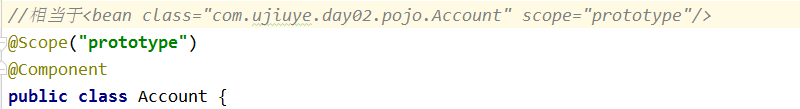
运行



3、用于改变注入范围的注解

@Scope

用于指定bean的作用范围，相当于配置文件中的< bean scope="">

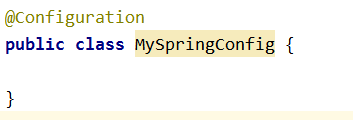


2、纯注解配置（了解）

我们的配置文件已经去掉好多内容了，但是我是想不写xml,能不能用一个java类来代替。

但是这种作了解就好了，一般工作中都用xml

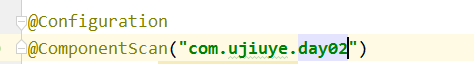
@Configuration用于定义配置类，可替换xml配置文件(告诉程序这是一个spring的配置文件)



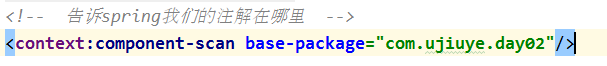
@Bean注解用于告诉方法，产生一个Bean对象，然后这个Bean对象交给Spring管理。只能用于方法上。



扫包



相当于



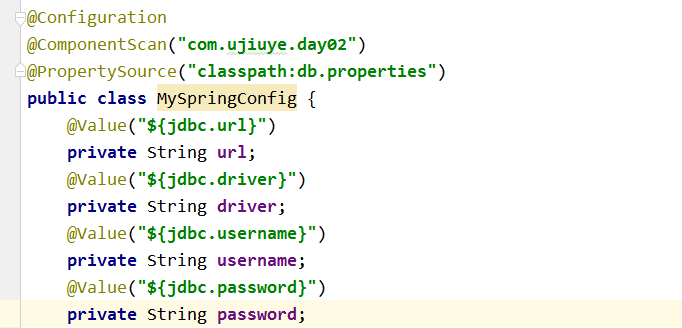
加载db.properties



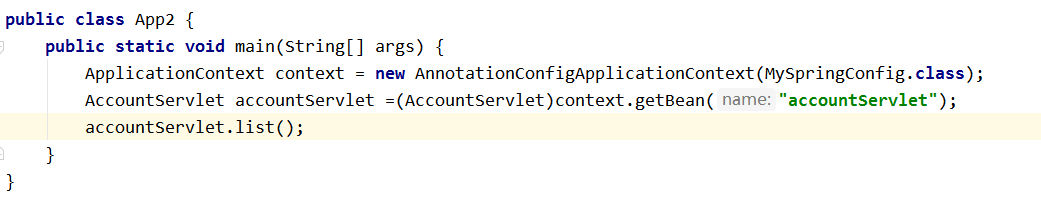
相当于



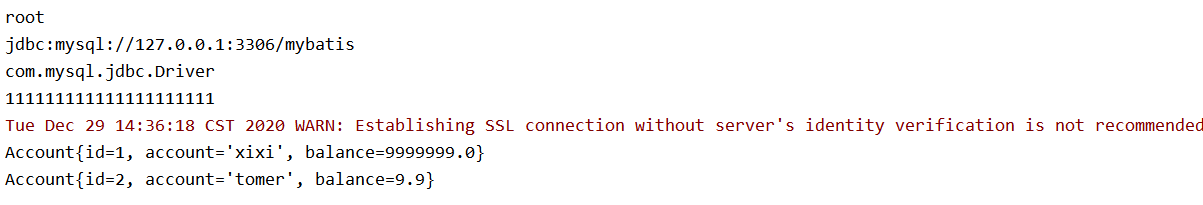
读取



测试类



运行



## 四、Spring和junit的整合

这是Spring提供的对Junit单元测试的一种支持。使用步骤如下:

1：引入坐标

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-test</artifactId>

<version>5.1.16.RELEASE</version>

</dependency>

<dependency>

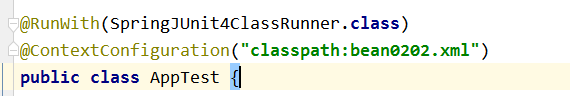
<groupId>junit</groupId>

<artifactId>junit</artifactId>

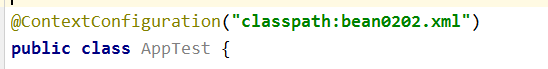
<version>4.13</version>

</dependency>

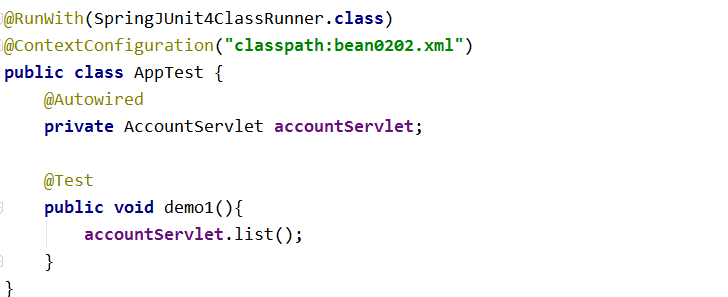
2：在测试类上使用@RunWith指定Spring的单元测试运行器



3:使用@ContextConfiguration指定配置文件，它支持文件和类的形式



4:下面可以直接使用@Autowired的形式，让Spring容器为我们注入需要的bean了



## 五、动态代理模式详解

### （一）代理模式简介

代理模式是常用的java设计模式，他的特征是代理类与委托类有同样的接口，代理类主要负责为委托类预处理消息、过滤消息、把消息转发给委托类，以及事后处理消息等。代理类与委托类之间通常会存在关联关系，一个代理类的对象与一个委托类的对象关联，代理类的对象本身并不真正实现服务，而是通过调用委托类的对象的相关方法，来提供特定的服务。简单的说就是，我们在访问实际对象时，是通过代理对象来访问的，代理模式就是在访问实际对象时引入一定程度的间接性。

代理类在程序运行时创建的代理方式被成为动态代理。动态代理是在运行时根据我们在Java代码中的“指示”动态生成的。相比于静态代理，动态代理的优势在于可以很方便的对代理类的函数进行统一的处理，而不用修改每个代理类中的方法。

静态代理：在程序运行时就写好的

租房：

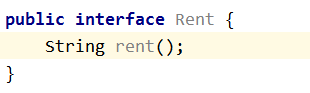
真房东 委托类

二手房东（中介） 代理类

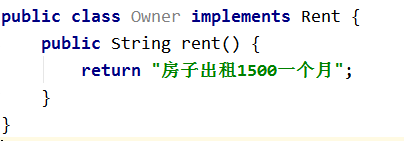
共同行为出租房子

客户

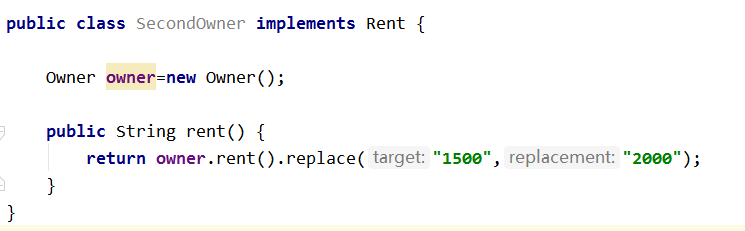
1、新建接口



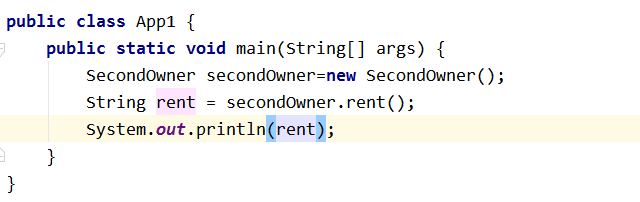
2、新建房东



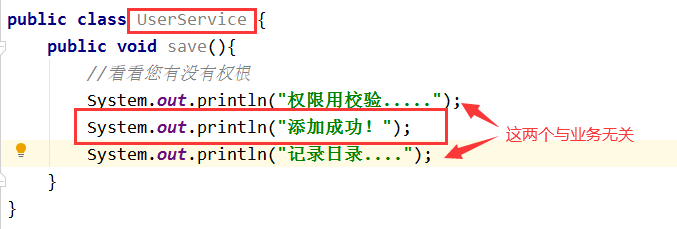
3、二手房东



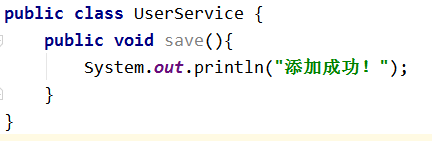
4、客户端



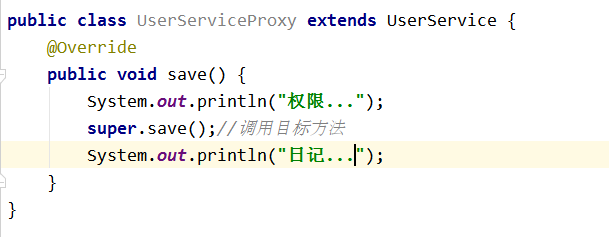
写一个没接口的



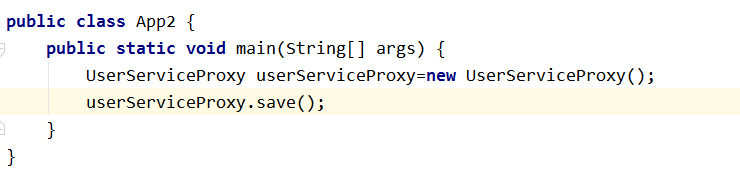
userService应该专心做业，与业务无关我不做，交给别人帮我做（代理）



新建代理类



测试



总结：代理是一个思想

代理分动态代理和静态代理

实现方式：

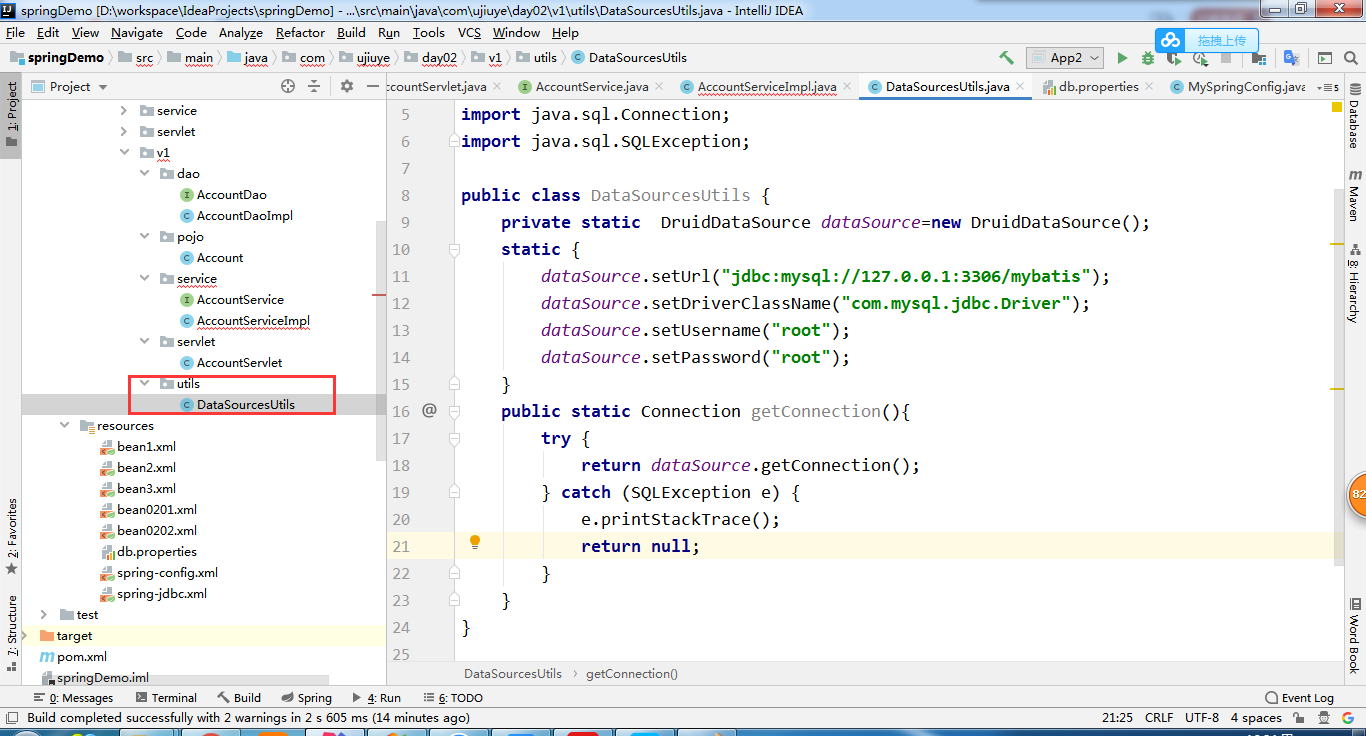
1. 实现接口 动态代理中使用JDK代理
2. 没有实现接口，继承 动态代理中使用Cglib代理

### （二） 动态代理模式入门案例

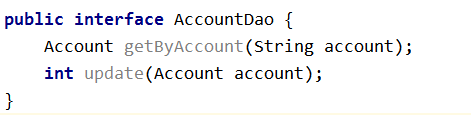
需求： A账户需要转账给B账户

1、 Version 1.0 使用传统的分层调用方式，同时在service中添加事务管理

数据源工具类

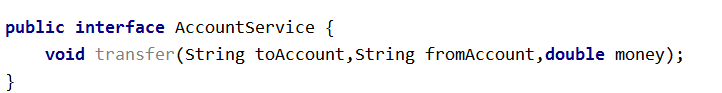


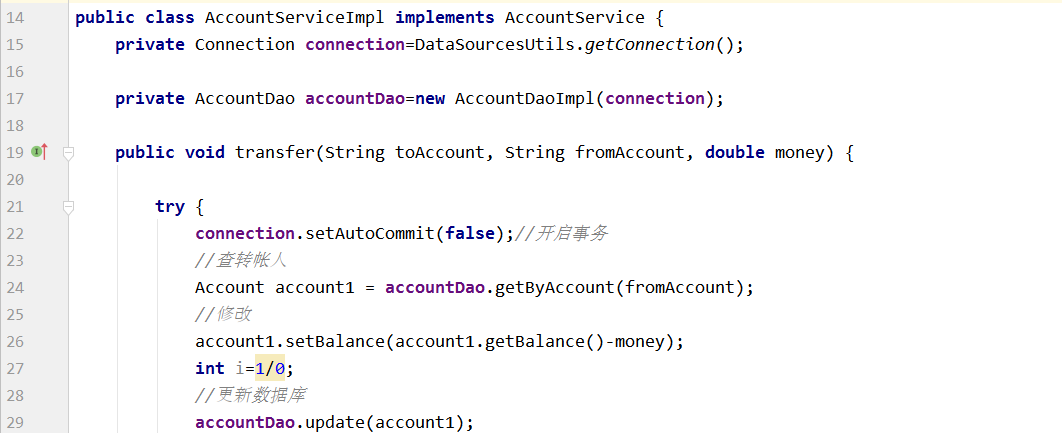
Dao

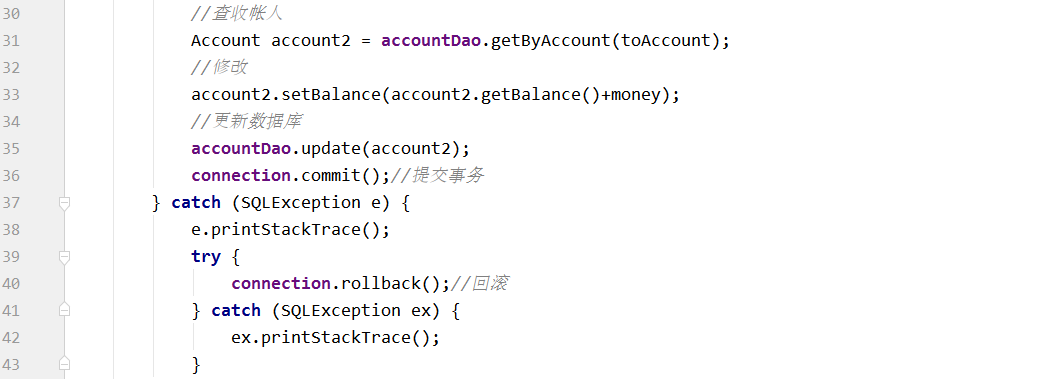


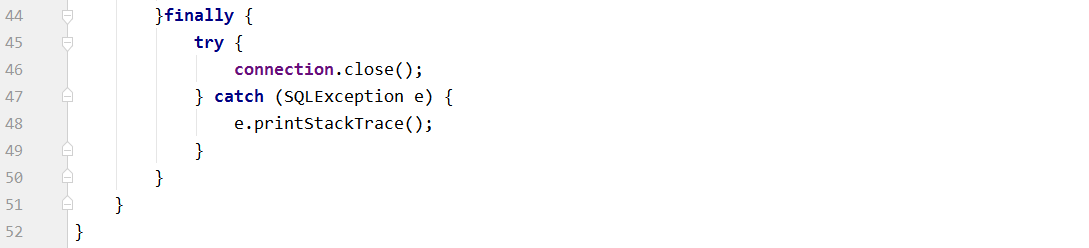


service

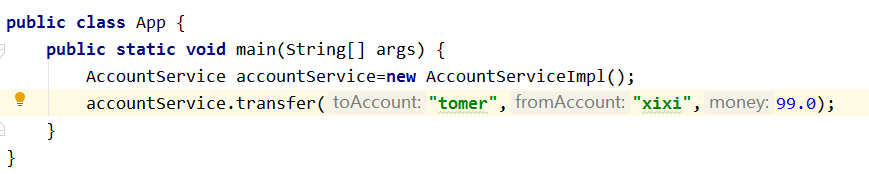








测试



发现事务已经得到控制了。

但是Connecttion是属于dao层代码，但是现又跑到service

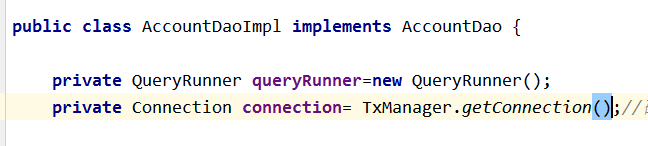
2、 V2.0 从业务层剥离跟持久层相关的代码

新建v2包，复制v1包的内容过去

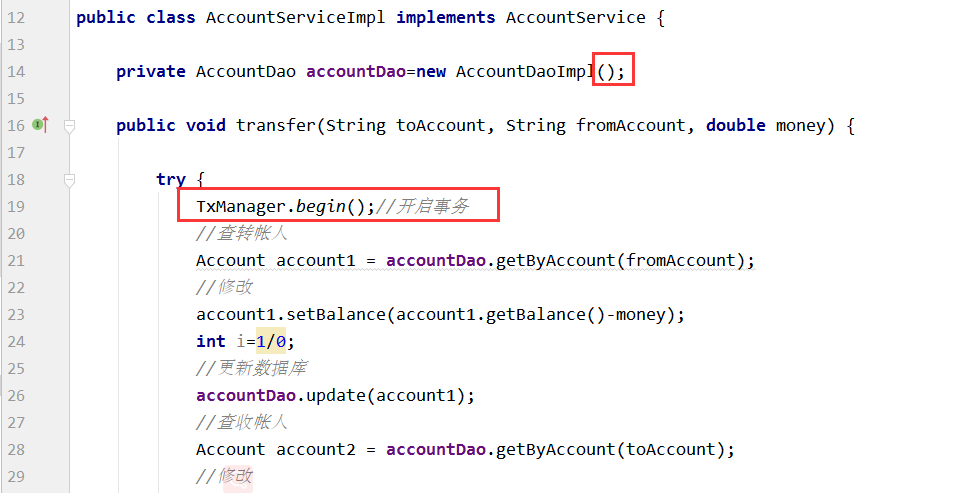
新建工具类

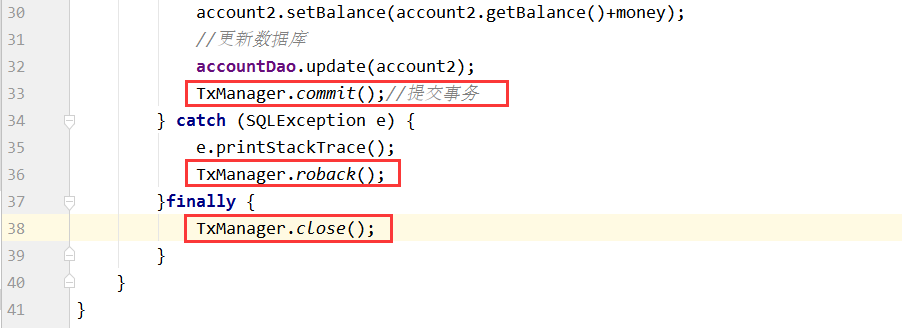
|  |
| --- |
| **package** com.ujiuye.day02.v2.utils;  **import** java.sql.Connection; **import** java.sql.SQLException;  **public class** TxManager {  *//确保同一个业使用同一个connection  //ThreadLocal每个线程都有独立ThreadLocal，确保同一个线程拿的内容是相同的* **private static** ThreadLocal<Connection> *t1*=**new** ThreadLocal<Connection>();   **public static** Connection getConnection(){  Connection connection = *t1*.get();  **if**(connection==**null**){  connection=DataSourcesUtils.*getConnection*();  *t1*.set(connection);  }  **return** connection;  }  **public static void** begin(){  **try** {  *getConnection*().setAutoCommit(**false**);  } **catch** (SQLException e) {  e.printStackTrace();  }  }   **public static void** commit(){  **try** {  *getConnection*().commit();  } **catch** (SQLException e) {  e.printStackTrace();  }  }   **public static void** roback(){  **try** {  *getConnection*().rollback();  } **catch** (SQLException e) {  e.printStackTrace();  }  }   **public static void** close(){  **try** {  *getConnection*().close();  } **catch** (SQLException e) {  e.printStackTrace();  }  } } |

修改dao



修改service



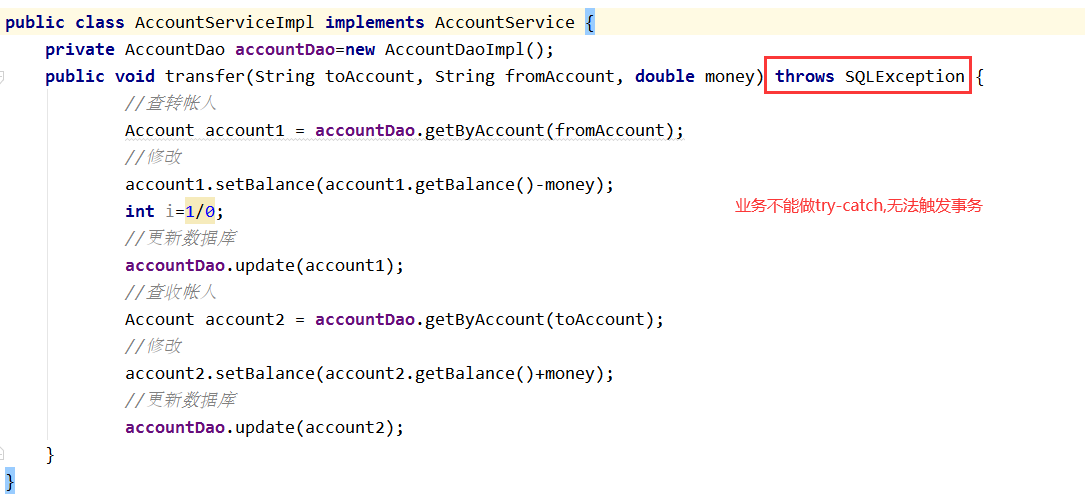


运行

但是业务层还存公共代码，如：事务开启和事务关闭。我应该专心做转帐，这些公共代码交给代理。

3、 V3.0 使用动态代理优化代码

修改service



事务控制的代码已经没有了。

1. JDK实现代码（动态代理----代理类是在程运行时才生成的）

JDK代理就JDK提供的，不用添加依赖

写代理规则而，代理类是在程运行时才生成的

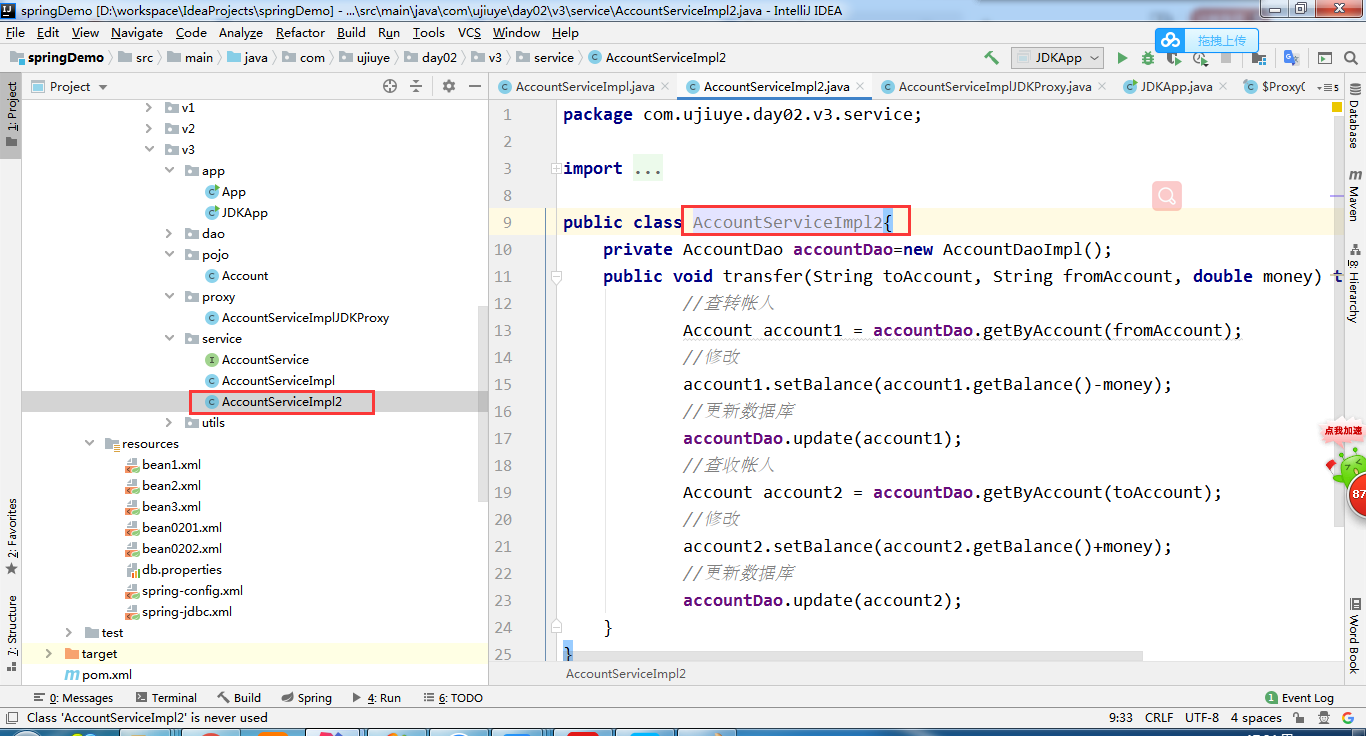


测试类



Cglib代理

复制实现，不要实现接口



添加依赖

<dependency>

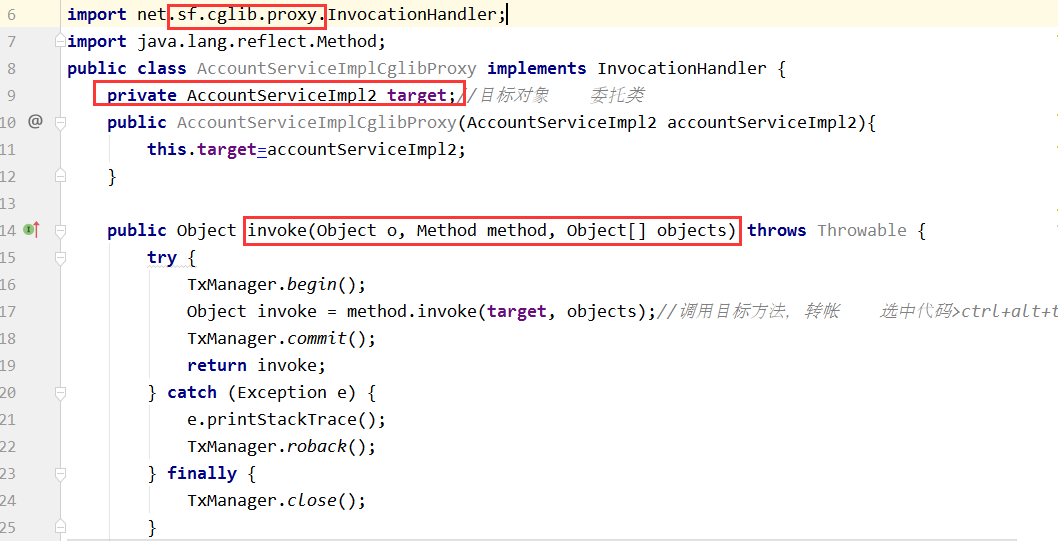
<groupId>cglib</groupId>

<artifactId>cglib</artifactId>

<version>3.3.0</version>

</dependency>

新建规则



测试类



通过上面的案例，我们体会到一种新的编程思想：

当核心业务（转账）和增强业务（事务）同时出现时，我们可以在开发时对他们分别开发，运行时再组装在一起（使用动态代理的方式）。

这样做的好处是：

1. 逻辑清晰，开发核心业务的时候，不必关注增强业务的代码

2. 代码复用性高：增强代码不用重复书写

这就是一种AOP的思想。