typora-root-url: images

typora-copy-images-to: images

# **Spring JDBC 和 事务控制**

## **主要内容**

## **Spring 整合 JDBC 环境**

Spring 框架除了提供 IOC 与 AOP 核心功能外，同样提供了基于JDBC 的数据访问功能，使得访问持久层数据更加方便。使用 Spring JDBC 环境，首先需要一套 Spring 整合 JDBC 的环境。

### **添加依赖坐标**

<!-- 添加相关的依赖坐标 -->  
<!-- spring 框架坐标依赖添加 -->  
<dependency>  
    <groupId>org.springframework</groupId>  
    <artifactId>spring-context</artifactId>  
    <version>5.2.4.RELEASE</version>  
</dependency>  
<!-- spring 测试环境 -->  
<dependency>  
    <groupId>org.springframework</groupId>  
    <artifactId>spring-test</artifactId>  
    <version>5.2.4.RELEASE</version>  
    <scope>test</scope>  
</dependency>  
<!-- aop -->  
<dependency>  
    <groupId>org.aspectj</groupId>  
    <artifactId>aspectjweaver</artifactId>  
    <version>1.9.5</version>  
</dependency>  
<!-- spring jdbc -->  
<dependency>  
    <groupId>org.springframework</groupId>  
    <artifactId>spring-jdbc</artifactId>  
    <version>5.2.4.RELEASE</version>  
</dependency>  
<!-- spring事物 -->  
<dependency>  
    <groupId>org.springframework</groupId>  
    <artifactId>spring-tx</artifactId>  
    <version>5.2.4.RELEASE</version>  
</dependency>  
<!-- mysql 驱动包 -->  
<dependency>  
    <groupId>mysql</groupId>  
    <artifactId>mysql-connector-java</artifactId>  
    <version>8.0.19</version>  
</dependency>  
<!-- c3p0 连接池 -->  
<dependency>  
    <groupId>com.mchange</groupId>  
    <artifactId>c3p0</artifactId>  
    <version>0.9.5.5</version>  
</dependency>

### **添加 jdbc 配置文件**

在src/main/resources目录下新建jdbc.properties配置文件，并设置对应的配置信息

# 驱动名  
jdbc.driver=com.mysql.cj.jdbc.Driver  
# 数据库连接  
jdbc.url=jdbc:mysql://localhost:3306/(数据库名称)?useUnicode=true&characterEncoding=utf8&serverTimezone=GMT%2B8&useSSL=false  
# 数据库用户名称  
jdbc.user=(数据库账号)  
# 数据库用户密码  
jdbc.password=(数据库密码)

以下为可选配置

# 指定连接池的初始化连接数。取值应在minPoolSize 与 maxPoolSize 之间.Default:3  
initialPoolSize=20  
# 指定连接池中保留的最大连接数. Default:15  
maxPoolSize=100  
# 指定连接池中保留的最小连接数  
minPoolSize=10  
# 最大空闲时间,60秒内未使用则连接被丢弃。若为0则永不丢弃。 Default:0  
maxIdleTime=600  
# 当连接池中的连接耗尽的时候c3p0一次同时获取的连接数. Default:3  
acquireIncrement=5  
# JDBC的标准,用以控制数据源内加载的PreparedStatements数量。  
maxStatements=5  
# 每60秒检查所有连接池中的空闲连接.Default:0  
idleConnectionTestPeriod=60

### **修改 spring 配置文件**

<!-- 加载properties 配置文件，用来读取jdbc.properties文件中的数据 -->  
<context:property-placeholder location="jdbc.properties" />

spring.xml

<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  
       https://www.springframework.org/schema/beans/spring-beans.xsd  
       http://www.springframework.org/schema/context  
       http://www.springframework.org/schema/context/spring-context.xsd">  
​  
    <!-- Spring扫描注解的配置 -->  
    <context:component-scan base-package="com.xxxx" />  
​  
    <!-- 加载properties 配置文件 -->  
    <context:property-placeholder location="jdbc.properties" />  
      
</beans>

### **配置数据源**

由于建立数据库连接是一个非常耗时耗资源的行为，所以通过连接池预先同数据库建立一些连接，放在内存中，应用程序需要建立数据库连接时直接到连接池中申请一个就行，用完后再放回去。

C3P0 与 DBCP 二选一即可

DBCP(DataBase connection pool)，数据库连接池。是 apache 上的一个 java 连接池项目，也是 tomcat 使用的连接池组件。单独使用dbcp需要2个包：commons-dbcp.jar，commons-pool.jar dbcp，没有自动回收空闲连接的功能。

C3P0是一个开源的JDBC连接池，它实现了数据源，支持JDBC3规范和JDBC2的标准扩展。目前使用它的开源项目有Hibernate，Spring等。c3p0有自动回收空闲连接功能。

#### **C3P0 数据源配置**

<!-- 配置 c3p0 数据源 -->  
<bean id="dataSource" class="com.mchange.v2.c3p0.ComboPooledDataSource">  
    <!-- property标签的value属性对应的是jdbc.properties中的值 -->  
    <property name="driverClass" value="${jdbc.driver}"></property>  
    <property name="jdbcUrl" value="${jdbc.url}"></property>  
    <property name="user" value="${jdbc.user}"></property>  
    <property name="password" value="${jdbc.password}"></property>  
</bean>

C3P0 其他额外配置（对应的值在jdbc.properties文件中指定）

<!-- 指定连接池中保留的最大连接数。 Default:15-->  
<property name="maxPoolSize" value="${maxPoolSize}"/>  
<!-- 指定连接池中保留的最小连接数。-->  
<property name="minPoolSize" value="${minPoolSize}"/>  
<!-- 指定连接池的初始化连接数。取值应在minPoolSize 与 maxPoolSize 之间.Default:3-->  
<property name="initialPoolSize" value="${initialPoolSize}"/>  
<!-- 最大空闲时间,60秒内未使用则连接被丢弃。若为0则永不丢弃。 Default:0-->  
<property name="maxIdleTime" value="${maxIdleTime}"/>  
<!-- 当连接池中的连接耗尽的时候c3p0一次同时获取的连接数。 Default:3-->  
<property name="acquireIncrement" value="${acquireIncrement}"/>  
<!-- JDBC的标准,用以控制数据源内加载的PreparedStatements数量。    
但由于预缓存的statements属于单个connection，而不是整个连接池所以设置这个参数需要考虑到多方面的因数。如果maxStatements与maxStatementsPerConnection均为0，则缓存被关闭。Default:0-->  
<property name="maxStatements" value="${maxStatements}"/>  
<!-- 每60秒检查所有连接池中的空闲连接。Default:0 -->  
<property name="idleConnectionTestPeriod" value="${idleConnectionTestPeriod}"/>

#### **DBCP 数据源配置**

<!-- 配置dbcp数据源-->  
<bean id="myDataSource" class="org.apache.commons.dbcp2.BasicDataSource">  
     <property name="driverClassName" value="${jdbc.driver}" />  
     <property name="url" value="${jdbc.url}"/>  
     <property name="username" value="${jdbc.user}"/>  
     <property name="password" value="${jdbc.password}"/>  
     <!-- 连接池启动时的初始值 -->    
     <property name="initialSize" value="1"/>    
     <!-- 最大空闲值.当经过一个高峰时间后，连接池可以将已经用不到的连接慢慢释放一部分，一直减少到maxIdle为止 -->    
     <property name="maxIdle" value="2"/>    
     <!-- 最小空闲值.当空闲的连接数少于阀值时，连接池就会预申请一些连接，以避免洪峰来时再申请而造成的性能开销 -->    
     <property name="minIdle" value="1"/>    
</bean>

### **模板类配置**

Spring把 JDBC 中重复的操作建立成了一个模板类：org.springframework.jdbc.core.JdbcTemplate 。

<!-- 配置JdbcTemplate实例，并注入一个dataSource数据源-->  
<bean id="jdbcTemplate" class="org.springframework.jdbc.core.JdbcTemplate">  
 <property name="dataSource" ref="dataSource"></property>  
</bean>

### **JDBC 测试**

#### **创建指定数据库**

选择连接，右键选择"新建数据库"，设置数据库的名称和编码格式

#### **创建数据表**

#### **使用 JUnit 测试**

通过 junit 测试 jdbcTemplate bean 是否获取到

##### **JUnit 测试**

public class SpringJdbcTest01 {  
  
 @Test  
 public void testQueryCount() {  
 // 获取spring上下文环境  
 ApplicationContext ctx = new ClassPathXmlApplicationContext("spring.xml");  
 // 得到模板类 JdbcTemplate对象  
 JdbcTemplate jdbcTemplate = (JdbcTemplate) ctx.getBean("jdbcTemplate");  
  
 // 定义sql语句  
 String sql = "select count(1) from tb\_account";  
 // 执行查询操作（无参数）  
 Integer total= jdbcTemplate.queryForObject(sql, Integer.class);  
 System.out.println("总记录数:" + total);  
 }  
  
 @Test  
 public void testQueryCountByUserId() {  
 // 获取spring上下文环境  
 ApplicationContext ctx = new ClassPathXmlApplicationContext("spring.xml");  
 // 得到模板类 JdbcTemplate对象  
 JdbcTemplate jdbcTemplate = (JdbcTemplate) ctx.getBean("jdbcTemplate");  
  
 // 定义sql语句  
 String sql = " select count(1) from tb\_account where user\_id = ?";  
 // 执行查询操作（有参数）  
 Integer total = jdbcTemplate.queryForObject(sql, Integer.class, 1);  
 System.out.println("总记录数:" + total);  
 }  
}

##### **简单封装**

public class SpringJdbcTest02 {  
  
 private JdbcTemplate jdbcTemplate;  
  
 @Before  
 public void init() {  
 // 得到Spring上下文环境  
 ApplicationContext ac = new ClassPathXmlApplicationContext("spring.xml");  
 // 得到模板类 JdbcTemplate对象  
 jdbcTemplate = (JdbcTemplate) ac.getBean("jdbcTemplate");  
 }  
  
 @Test  
 public void testQueryCount() {  
 // 定义sql语句  
 String sql = "select count(1) from tb\_account";  
 // 执行查询操作（无参数）  
 Integer total= jdbcTemplate.queryForObject(sql, Integer.class);  
 System.out.println("总记录数:" + total);  
 }  
  
 @Test  
 public void testQueryCountByUserId() {  
 // 定义sql语句  
 String sql = " select count(1) from tb\_account where user\_id = ?";  
 // 执行查询操作（有参数）  
 Integer total = jdbcTemplate.queryForObject(sql, Integer.class, 1);  
 System.out.println("总记录数:" + total);  
 }  
}

##### **注解封装**

@RunWith  
 就是一个运行器  
 @RunWith(JUnit4.class) 就是指用JUnit4来运行  
 @RunWith(SpringJUnit4ClassRunner.class) 让测试运行于Spring测试环境  
@ContextConfiguration  
 Spring整合JUnit4测试时，使用注解引入多个配置文件  
 @ContextConfiguration(Locations="classpath：applicationContext.xml")    
 @ContextConfiguration(locations = {"classpath:spring.xml", "classpath:bean.xml"})

@RunWith(SpringJUnit4ClassRunner.class) // 将junit测试加到spring环境中  
@ContextConfiguration(locations = {"classpath:spring.xml"}) // 设置要加载的资源文件  
public class SpringJdbcTest03 {  
  
 @Resource  
 private JdbcTemplate jdbcTemplate;  
  
 @Test  
 public void testQueryCount() {  
 // 定义sql语句  
 String sql = "select count(1) from tb\_account";  
 // 执行查询操作（无参数）  
 Integer total= jdbcTemplate.queryForObject(sql, Integer.class);  
 System.out.println("总记录数:" + total);  
 }  
  
}

##### **通用封装**

定义一个父类，设置通用的配置信息

/\*\*  
 \* 通用的测试环境，需要使用环境的直接继承类即可  
 \*/  
@RunWith(SpringJUnit4ClassRunner.class) // 将junit测试加到spring环境中  
@ContextConfiguration(locations = {"classpath:spring.xml"}) // 设置要加载的资源文件  
public class BaseTest {  
   
}

继承通用的测试类

public class SpringJdbcTest04 extends BaseTest {  
  
 @Resource  
 private JdbcTemplate jdbcTemplate;  
   
 @Test  
 public void testQueryCount() {  
 // 定义sql语句  
 String sql = "select count(1) from tb\_account";  
 // 执行查询操作（无参数）  
 Integer total= jdbcTemplate.queryForObject(sql, Integer.class);  
 System.out.println("总记录数:" + total);  
 }   
}

## **持久层账户模块操作**

当完成 Spring Jdbc 环境集成后，这里使用spring jdbc 完成账户单表crud 操作。

### **账户接口方法定义**

#### **定义实体类**

Account.java

package com.xxxx.entity;  
  
import java.util.Date;  
  
/\*\*  
 \* 用户账户类  
 \*/  
public class Account {  
  
 private Integer accountId; // 账户ID，主键  
 private String accountName; // 账户名称  
 private String accountType; // 账户类型  
 private Double money; // 账户金额  
 private String remark; // 账户备注  
 private Integer userId; // 用户ID，账户所属用户  
 private Date createTime; // 创建时间  
 private Date updateTime; // 修改时间  
  
 public Account() {  
  
 }  
  
 public Account(String accountName, String accountType, Double money,  
 String remark, Integer userId) {  
 this.accountName = accountName;  
 this.accountType = accountType;  
 this.money = money;  
 this.remark = remark;  
 this.userId = userId;  
 }  
  
 @Override  
 public String toString() {  
 return "Account{" +  
 "accountId=" + accountId +  
 ", accountName='" + accountName + '\'' +  
 ", accountType='" + accountType + '\'' +  
 ", money=" + money +  
 ", remark='" + remark + '\'' +  
 ", userId=" + userId +  
 ", createTime=" + createTime +  
 ", updateTime=" + updateTime +  
 '}';  
 }  
  
 public Integer getAccountId() {  
 return accountId;  
 }  
  
 public void setAccountId(Integer accountId) {  
 this.accountId = accountId;  
 }  
  
 public String getAccountName() {  
 return accountName;  
 }  
  
 public void setAccountName(String accountName) {  
 this.accountName = accountName;  
 }  
  
 public String getAccountType() {  
 return accountType;  
 }  
  
 public void setAccountType(String accountType) {  
 this.accountType = accountType;  
 }  
  
 public Double getMoney() {  
 return money;  
 }  
  
 public void setMoney(Double money) {  
 this.money = money;  
 }  
  
 public String getRemark() {  
 return remark;  
 }  
  
 public void setRemark(String remark) {  
 this.remark = remark;  
 }  
  
 public Integer getUserId() {  
 return userId;  
 }  
  
 public void setUserId(Integer userId) {  
 this.userId = userId;  
 }  
  
 public Date getCreateTime() {  
 return createTime;  
 }  
  
 public void setCreateTime(Date createTime) {  
 this.createTime = createTime;  
 }  
  
 public Date getUpdateTime() {  
 return updateTime;  
 }  
  
 public void setUpdateTime(Date updateTime) {  
 this.updateTime = updateTime;  
 }  
}

#### **定义接口类**

IAccountDao.java

package com.xxxx.dao;  
  
import com.xxxx.entity.Account;  
import java.util.List;  
  
/\*\*  
 \* 用户模块 接口定义  
 \* 1. 添加账户  
 \* 添加账户记录，返回受影响的行数  
 \* 添加账户记录，返回记录的主键  
 \* 批量添加账户记录，返回受影响的行数  
 \* 2. 查询账户  
 \* 查询指定用户的账户总记录数，返回记录数  
 \* 查询指定账户记录详情，返回账户对象  
 \* 多条件查询指定用户的账户列表，返回账户集合  
 \* 3. 更新账户  
 \* 更新账户记录，返回受影响的行数  
 \* 批量更新账户记录，返回受影响的行数  
 \* 4. 删除账户  
 \* 删除账户记录，返回受影响的行数  
 \* 批量删除账户记录，返回受影响的行数  
 \*/  
public interface IAccountDao {  
  
 /\*\*  
 \* 添加账户  
 \* 添加账户记录，返回受影响的行数  
 \* @param account  
 \* @return  
 \*/  
 public int addAccount(Account account) ;  
  
 /\*\*  
 \* 添加账户  
 \* 添加账户记录，返回记录的主键  
 \* @param account  
 \* @return  
 \*/  
 public int addAccountHasKey(Account account);  
  
 /\*\*  
 \* 添加账户  
 \* 批量添加账户记录，返回受影响的行数  
 \* @param accounts  
 \* @return  
 \*/  
 public int addAccountBatch(List<Account> accounts);  
  
 /\*\*  
 \* 查询账户  
 \* 查询指定用户的账户总记录数，返回记录数  
 \* @param userId  
 \* @return  
 \*/  
 public int queryAccountCount(Integer userId);  
  
  
 /\*\*  
 \* 查询账户  
 \* 查询指定账户记录详情，返回账户对象  
 \* @param accountId  
 \* @return  
 \*/  
 public Account queryAccountById(Integer accountId);  
  
  
 /\*\*  
 \* 查询账户  
 \* 多条件查询指定用户的账户列表，返回账户集合  
 \* @param userId  
 \* @param accountName  
 \* @param accountType  
 \* @param createTime  
 \* @return  
 \*/  
 public List<Account> queryAccountsByParams(Integer userId, String accountName, String accountType, String createTime);  
  
 /\*\*  
 \* 更新账户  
 \* 更新账户记录，返回受影响的行数  
 \* @param account  
 \* @return  
 \*/  
 public int updateAccountById(Account account);  
  
 /\*\*  
 \* 更新账户  
 \* 批量更新账户记录，返回受影响的行数  
 \* @param accounts  
 \* @return  
 \*/  
 public int updateAccountBatch(List<Account> accounts);  
  
 /\*\*  
 \* 删除账户  
 \* 删除账户记录，返回受影响的行数  
 \* @param accountId  
 \* @return  
 \*/  
 public Integer deleteAccoutById(Integer accountId);  
  
 /\*\*  
 \* 删除用户  
 \* 批量删除账户记录，返回受影响的行数  
 \* @param ids  
 \* @return  
 \*/  
 public int deleteAccountBatch(Integer[] ids);  
}

#### **定义接口实现类**

package com.xxxx.dao.impl;  
  
import com.xxxx.dao.IAccountDao;  
import com.xxxx.entity.Account;  
import org.springframework.jdbc.core.JdbcTemplate;  
import org.springframework.stereotype.Component;  
import javax.annotation.Resource;  
import java.util.List;  
  
/\*\*  
 \* 账户模块接口实现类  
 \*/  
@Repository  
public class AccountDaoImpl implements IAccountDao {  
  
 // JdbcTemplate 模板类注入  
 @Resource  
 private JdbcTemplate jdbcTemplate;  
  
 @Override  
 public int addAccount(Account account) {  
 return 0;  
 }  
  
 @Override  
 public int addAccountHasKey(Account account) {  
 return 0;  
 }  
  
 @Override  
 public int addAccountBatch(List<Account> accounts) {  
 return 0;  
 }  
  
 @Override  
 public int queryAccountCount(Integer userId) {  
 return 0;  
 }  
  
 @Override  
 public Account queryAccountById(Integer accountId) {  
 return null;  
 }  
  
 @Override  
 public List<Account> queryAccountsByParams(Integer userId, String accountName, String accountType, String createTime) {  
 return null;  
 }  
  
 @Override  
 public int updateAccountById(Account account) {  
 return 0;  
 }  
  
 @Override  
 public int updateAccountBatch(List<Account> accounts) {  
 return 0;  
 }  
  
 @Override  
 public Integer deleteAccoutById(Integer accountId) {  
 return null;  
 }  
  
 @Override  
 public int deleteAccountBatch(Integer[] ids) {  
 return 0;  
 }  
}

### **账户记录添加实现**

在企业项目开发时，对于记录的添加可能涉及到多种添加方式，比如添加单条记录，批量添加多条记录等情况。这里对于账户记录添加方式分为三种方式：添加单条记录返回受影响行数、添加单条记录返回主键、批量添加多条记录。

#### **添加账户记录**

/\*\*  
 \* 添加单条记录，返回受影响的行数  
 \* @param account  
 \* @return  
 \*/  
@Override  
public int addAccount(Account account) {  
 String sql = "insert into tb\_account(account\_name,account\_type,money,remark," +  
 "user\_id,create\_time,update\_time) values (?,?,?,?,?,now(),now())";  
 Object[] objs = {account.getAccountName(),account.getAccountType(),  
 account.getMoney(),account.getRemark(),account.getUserId()};  
 return jdbcTemplate.update(sql,objs);  
}

测试方法

/\*\*  
 \* 添加账户记录，得到受影响的行数  
 \*/  
@Test  
public void testAddAccount() {  
 // 准备要添加的数据  
 Account account = new Account("张三","建设银行",100.0,"零花钱",1);  
  
 // 调用对象的添加方法，返回受影响的行数  
 int row = accountDao.addAccount(account);  
 System.out.println("添加账户受影响的行数：" + row);  
}

#### **添加记录返回主键**

/\*\*  
 \* 添加单条记录，返回主键  
 \* @param account  
 \* @return  
 \*/  
@Override  
public int addAccountHasKey(Account account) {  
 String sql = "insert into tb\_account(account\_name,account\_type,money,remark," +  
 "user\_id,create\_time,update\_time) values (?,?,?,?,?,now(),now())";  
 // 定义keyHolder 对象 获取记录主键值  
 KeyHolder keyHolder = new GeneratedKeyHolder();  
 jdbcTemplate.update(connection -> {  
 // 预编译sql语句，并设置返回主键  
 PreparedStatement ps =   
 connection.prepareStatement(sql, Statement.RETURN\_GENERATED\_KEYS);  
 // 设置参数  
 ps.setString(1,account.getAccountName());  
 ps.setString(2,account.getAccountType());  
 ps.setDouble(3,account.getMoney());  
 ps.setString(4,account.getRemark());  
 ps.setInt(5,account.getUserId());  
 return ps;  
 },keyHolder);  
 // 得到返回的主键  
 Integer key = keyHolder.getKey().intValue();  
  
 return key;  
}

测试方法

/\*\*  
 \* 添加账户记录，返回主键  
 \*/  
@Test  
public void testAddAccountHasKey() {  
 // 准备要添加的数据  
 Account account = new Account("李四","招商银行",200.0,"兼职费",2);  
  
 // 调用对象的添加方法，返回主键  
 int key = accountDao.addAccountHasKey(account);  
 System.out.println("添加账户返回的主键：" + key);  
}

#### **批量添加账户记录**

/\*\*  
 \* 添加多条记录，返回受影响的行数  
 \* @param accounts  
 \* @return  
 \*/  
@Override  
public int addAccountBatch(final List<Account> accounts) {  
 String sql = "insert into tb\_account(account\_name,account\_type,money,remark," +  
 "user\_id,create\_time,update\_time) values (?,?,?,?,?,now(),now())";  
 int rows = jdbcTemplate.batchUpdate(sql, new BatchPreparedStatementSetter() {  
 @Override  
 public void setValues(PreparedStatement preparedStatement, int i)   
 throws SQLException {  
 // 设置参数  
 preparedStatement.setString(1,accounts.get(i).getAccountName());  
 preparedStatement.setString(2,accounts.get(i).getAccountType());  
 preparedStatement.setDouble(3,accounts.get(i).getMoney());  
 preparedStatement.setString(4,accounts.get(i).getRemark());  
 preparedStatement.setInt(5,accounts.get(i).getUserId());  
 }  
 @Override  
 public int getBatchSize() {  
 return accounts.size();  
 }  
 }).length;  
  
 return rows;  
}

测试方法

/\*\*  
 \* 批量添加数据，返回受影响的行数  
 \*/  
@Test  
public void testAddAccountBatch() {  
 // 准备要添加的数据  
 Account account = new Account("王五","农业银行",2000.0,"工资",3);  
 Account account2 = new Account("赵六","中国银行",280.0,"奖金",3);  
 Account account3 = new Account("田七","工商银行",800.0,"零花钱",3);  
 List<Account> accountList = new ArrayList<>();  
 accountList.add(account);  
 accountList.add(account2);  
 accountList.add(account3);  
  
 // 调用对象的添加方法，返回主键  
 int rows = accountDao.addAccountBatch(accountList);  
 System.out.println("批量添加账户受影响的行数：" + rows);  
}

### **账户记录查询实现**

账户记录查询这里提供了三种查询方式，查询指定用户所有账户记录数，查询单条账户记录详情，多条件查询指定用户账户记录。

#### **查询用户的账户总记录数**

/\*\*  
 \* 查询指定用户的账户总记录数，返回记录数  
 \* @param userId  
 \* @return  
 \*/  
@Override  
public int queryAccountCount(Integer userId) {  
 String sql = "select count(1) from tb\_account where user\_id = ?";  
 int count = jdbcTemplate.queryForObject(sql,Integer.class,userId);  
 return count;  
}

测试方法

/\*\*  
 \* 查询用户的账户总记录数，返回总记录数  
 \*/  
@Test  
public void testQueryAccountCount(){  
 // 查询ID为1的用户的账户总记录数  
 int total = accountDao.queryAccountCount(1);  
 System.out.println("总记录数：" + total);  
}

#### **查询指定账户记录详情**

/\*\*  
 \* 查询某个账户记录详情，返回账户对象  
 \* @param accountId  
 \* @return  
 \*/  
@Override  
public Account queryAccountById(Integer accountId) {  
 String sql = "select \* from tb\_account where account\_id = ?";  
 Account account = jdbcTemplate.queryForObject(sql, new Object[]{accountId}, (resultSet, i) -> {  
 Account acc = new Account();  
 acc.setAccountId(resultSet.getInt("account\_id"));  
 acc.setMoney(resultSet.getDouble("money"));  
 acc.setAccountName(resultSet.getString("account\_name"));  
 acc.setAccountType(resultSet.getString("account\_type"));  
 acc.setRemark(resultSet.getString("remark"));  
 acc.setCreateTime(resultSet.getDate("create\_time"));  
 acc.setUpdateTime(resultSet.getDate("update\_time"));  
 acc.setUserId(resultSet.getInt("user\_id"));  
 return acc;  
 });  
 return account;  
 }

测试方法

/\*\*  
 \* 查询指定账户的记录详情，返回账户对象  
 \*/  
@Test  
public void testQueryAccountById(){  
 // 查询ID为1的账户记录的详情  
 Account account = accountDao.queryAccountById(1);  
 System.out.println("账户详情：" + account.toString());  
}

#### **多条件查询用户账户记录**

/\*\*  
 \* 多条件查询指定用户的账户列表，返回账户集合  
 \* @param userId 用户Id  
 \* @param accountName 账户名称 （模糊查询）  
 \* @param accountType 账户类型  
 \* @param createTime 账户创建时间  
 \* @return  
 \*/  
@Override  
public List<Account> queryAccountsByParams(Integer userId, String accountName, String accountType,  
 String createTime) {  
 String sql = "select \* from tb\_account where user\_id = ? ";  
 List<Object> params = new ArrayList<>();  
 params.add(userId);  
  
 // 判断是否有条件查询  
 // 如果账户名称不为空，通过账户名称模糊匹配  
 if (StringUtils.isNotBlank(accountName)) {  
 sql += " and account\_name like concat('%',?,'%') ";  
 params.add(accountName);  
 }  
 // 如果账户类型不为空，通过指定类型名称查询  
 if (StringUtils.isNotBlank(accountType)) {  
 sql += " and account\_type = ? ";  
 params.add(accountType);  
 }  
 // 如果创建时间不为空，查询创建时间大于指定时间的账户记录  
 if (StringUtils.isNotBlank(createTime)) {  
 sql += " and create\_time > ? ";  
 params.add(createTime);  
 }  
  
 // 将集合转换成数组  
 Object[] objs = params.toArray();  
  
 List<Account> accountList = jdbcTemplate.query(sql, objs, (resultSet, rowNum) -> {  
 Account acc = new Account();  
 acc.setAccountId(resultSet.getInt("account\_id"));  
 acc.setMoney(resultSet.getDouble("money"));  
 acc.setAccountName(resultSet.getString("account\_name"));  
 acc.setAccountType(resultSet.getString("account\_type"));  
 acc.setRemark(resultSet.getString("remark"));  
 acc.setCreateTime(resultSet.getDate("create\_time"));  
 acc.setUpdateTime(resultSet.getDate("update\_time"));  
 acc.setUserId(resultSet.getInt("user\_id"));  
 return acc;  
 });  
   
 return accountList;  
}

测试方法

/\*\*  
 \* 多条件查询用户的账户记录，返回账户集合  
 \*/  
@Test  
public void testQueryAccountByParams(){  
 // 查询用户的账户列表  
 List<Account> accountList = accountDao.queryAccountsByParams(3,null,null,null);  
 // 通过指定条件查询用户的账户列表  
 List<Account> accountList02 = accountDao.queryAccountsByParams(3,"张",null,null);  
  
 System.out.println(accountList.toString());  
 System.out.println(accountList02.toString());  
}

### **账户记录更新实现**

#### **更新账户记录**

/\*\*  
 \* 更新指定账户记录，返回受影响的行数  
 \* @param account  
 \* @return  
 \*/  
@Override  
public int updateAccountById(Account account) {  
 String sql = "update tb\_account set account\_name = ?, account\_type = ?, " +  
 " money = ? ,remark = ?,user\_id = ? ,update\_time = now() " +  
 " where account\_id = ? ";  
 Object[] objs = {account.getAccountName(),account.getAccountType(),  
 account.getMoney(), account.getRemark(),account.getUserId(),  
 account.getAccountId()};  
 return jdbcTemplate.update(sql,objs);  
}

测试方法

/\*\*  
 \* 更新指定账户记录，返回受影响的行数  
 \*/  
@Test  
public void testUpdateAccount(){  
 // 准备要修改的数据  
 Account account = new Account("张三1","建设银行1",500.0,"零花钱加倍",1);  
 account.setAccountId(1);  
 int row = accountDao.updateAccountById(account);  
 System.out.println("修改账户返回受影响的行数：" + row);  
}

#### **批量更新账户记录**

/\*\*  
 \* 批量新账户记录，返回受影响的行数  
 \* @param accounts  
 \* @return  
 \*/  
@Override  
public int updateAccountBatch(List<Account> accounts) {  
 String sql = "update tb\_account set account\_name = ?, account\_type = ?, " +  
 " money = ? ,remark = ?,user\_id = ? ,update\_time = now() " +  
 " where account\_id = ? ";  
 int rows = jdbcTemplate.batchUpdate(sql, new BatchPreparedStatementSetter() {  
 @Override  
 public void setValues(PreparedStatement ps, int i) throws SQLException {  
 // 设置参数  
 ps.setString(1,accounts.get(i).getAccountName());  
 ps.setString(2,accounts.get(i).getAccountType());  
 ps.setDouble(3,accounts.get(i).getMoney());  
 ps.setString(4,accounts.get(i).getRemark());  
 ps.setInt(5,accounts.get(i).getUserId());  
 ps.setInt(6,accounts.get(i).getAccountId());  
 }  
 @Override  
 public int getBatchSize() {  
 return accounts.size();  
 }  
 }).length;  
 return rows;  
}

测试方法

/\*\*  
 \* 批量更新账户记录，返回受影响的行数  
 \*/  
@Test  
public void testUpdateAccountBatch(){  
 // 准备要修改的数据  
 Account account = new Account("a3","建设银行3",300.0,"零花钱加倍3",3);  
 account.setAccountId(3);  
 Account account2 = new Account("a4","建设银行4",400.0,"零花钱加倍4",3);  
 account2.setAccountId(4);  
 List<Account> accountList = new ArrayList<>();  
 accountList.add(account);  
 accountList.add(account2);  
  
 int rows = accountDao.updateAccountBatch(accountList);  
 System.out.println("批量修改账户记录返回受影响的行数：" + rows);  
}

### **账户记录删除实现**

#### **删除账户记录**

/\*\*  
 \* 删除账户记录，返回受影响的行数  
 \* @param accountId  
 \* @return  
 \*/  
@Override  
public Integer deleteAccoutById(Integer accountId) {  
 String sql = "delete from tb\_account where account\_id= ? ";  
 Object[] objs = {accountId};  
 return jdbcTemplate.update(sql,objs);  
}

测试方法

/\*\*  
 \* 删除账户记录，返回受影响的行数  
 \*/  
@Test  
public void testDeleteAccount(){  
 // 删除ID为1的账户记录  
 int row = accountDao.deleteAccoutById(1);  
 System.out.println("删除账户记录返回受影响的行数：" + row);  
}

#### **批量删除账户记录**

/\*\*  
 \* 批量删除账户记录，返回受影响的行数  
 \* @param ids  
 \* @return  
 \*/  
@Override  
public int deleteAccountBatch(Integer[] ids) {  
 String sql = "delete from tb\_account where account\_id = ?";  
 int row = jdbcTemplate.batchUpdate(sql, new BatchPreparedStatementSetter() {  
 @Override  
 public void setValues(PreparedStatement ps, int i) throws SQLException {  
 ps.setInt(1,ids[i]);  
 }  
  
 @Override  
 public int getBatchSize() {  
 return ids.length;  
 }  
 }).length;  
 return row;  
}

测试方法

/\*\*  
 \* 批量删除账户记录，返回受影响的行数  
 \*/  
@Test  
public void testDeleteAccountBatch(){  
 // 删除多个id的账户记录  
 Integer[] ids = new Integer[]{2,3};  
 int rows = accountDao.deleteAccountBatch(ids);  
 System.out.println("批量删除账户记录返回受影响的行数：" + rows);  
}

## **Spring 事务控制**

### **转账场景模拟实现**

#### **接口方法定义**

/\*\*  
 \* 收入  
 \* @param tarAid 收入金额的账户ID  
 \* @param money 收入金额  
 \* @return  
 \*/  
public int inAccount(Integer tarAid, Double money);  
  
/\*\*  
 \* 支出  
 \* @param outAid 支出金额的账户ID  
 \* @param money 支出金额  
 \* @return  
 \*/  
public int outAccount(Integer outAid, Double money);

#### **实现对应接口**

对于转账涉及到双方账户以及对应转账金额，所以有入账和出账两个方法。

/\*\*  
 \* 账户收入  
 \* @param tarAid 账户ID  
 \* @param money 收入金额  
 \* @return  
 \*/  
@Override  
public int inAccount(Integer tarAid, Double money) {  
 // 修改指定ID的金额 (加上金额)  
 String sql = "update tb\_account set money = money + ? where account\_id = ? ";  
 Object[] objs = {money, tarAid};  
 return jdbcTemplate.update(sql,objs);  
}  
  
/\*\*  
 \* 账户支出  
 \* @param outAid 账户ID  
 \* @param money 支出金额  
 \* @return  
 \*/  
@Override  
public int outAccount(Integer outAid, Double money) {  
 // 修改指定ID的金额 （减去金额）  
 String sql = "update tb\_account set money = money - ? where account\_id = ? ";  
 Object[] objs = {money, outAid};  
 return jdbcTemplate.update(sql,objs);  
}

#### **转账方法实现**

package com.xxxx.service;  
  
import com.xxxx.dao.IAccountDao;  
import org.springframework.stereotype.Service;  
import javax.annotation.Resource;  
  
@Service  
public class AccountService {  
  
 @Resource  
 private IAccountDao accountDao;  
  
 /\*\*  
 \* 转账业务操作  
 \* @param outAid 支出账户  
 \* @param inAid 收入账户  
 \* @param money 支出金额/收入金额  
 \* @return  
 \*/  
 public int updateAccountByTransfer(Integer outAid, Integer inAid, Double money){  
 int row = 0;  
 /\*\*  
 \* 张三账户向李四账户转账100元  
 \* 张三账户的金额 - 100  
 \* 李四账户的金额 + 100  
 \*/  
 // 支出，修改金额返回受影响的行数  
 int outRow = accountDao.outAccount(1,100.0);  
 // 收入，修改金额返回受影响的行数  
 int inRow = accountDao.inAccount(2,100.0);  
  
 // 当两个操作都执行成功时，转账成功  
 if (outRow == 1 && inRow == 1) {  
 row = 1; // 成功  
 }  
 return row;  
 }  
  
}

仔细思考代码会发现，在程序运行中无法保证 service 层业务代码不发生异常，如果通过 jdbc 的方式处理事务，此时需要手动方式控制事务，这样的话凡是涉及到事务控制的业务方法均需要开发人员手动来进行事务处理，无法满足生产的需要。

### **Spring 事务概念**

#### **事务的四大特性（ACID）**

原子性（Atomicity）

共生死，要么全部成功，要么全部失败！

一致性（Consistency）

事务在执行前后，数据库中数据要保持一致性状态。（如转账的过程 账户操作后数据必须保持一致）

隔离性（Isolation）

事务与事务之间的执行应当是相互隔离互不影响的。（多个角色对统一记录进行操作必须保证没有任何干扰），当然没有影响是不可能的，为了让影响级别降到最低，通过隔离级别加以限制：

1. READ\_UNCOMMITTED （读未提交）

隔离级别最低的一种事务级别。在这种隔离级别下，会引发脏读、不可重复读和幻读。

2. READ\_COMMITTED （读已提交）

读到的都是别人提交后的值。这种隔离级别下，会引发不可重复读和幻读，但避免了脏读。

3. REPEATABLE\_READ （可重复读）

这种隔离级别下，会引发幻读，但避免了脏读、不可重复读。

4. SERIALIZABLE （串行化）

最严格的隔离级别。在Serializable隔离级别下，所有事务按照次序依次执行。

脏读、不可重复读、幻读都不会出现。

持久性（Durability）

事务提交完毕后，数据库中的数据的改变是永久的。

#### **Spring 事务核心接口**

Spring 事务管理的实现有许多细节，如果对整个接口框架有个大体了解会非常有利于我们理解事务，下面通过讲解 Spring 的事务接口来了解 Spring 实现事务的具体策略。

Spring 并不直接管理事务，而是提供了多种事务管理器，他们将事务管理的职责委托给 Hibernate 或者 JTA 等持久化机制所提供的相关平台框架的事务来实现。

Spring 事务管理器的接口是org.springframework.transaction.PlatformTransactionManager，通过这个接口，Spring 为各个平台如 JDBC、Hibernate 等都提供了对应的事务管理器，但是具体的实现就是各个平台自己的事情了。此接口的内容如下：

public interface PlatformTransactionManager(){   
 // 由 TransactionDefinition 得到 TransactionStatus 对象   
 TransactionStatus getTransaction(TransactionDefinition definition) throws TransactionException;   
 // 提交   
 void commit(TransactionStatus status) throws TransactionException;   
 // 回滚   
 void rollback(TransactionStatus status) throws TransactionException;   
}

从这里可知具体的具体的事务管理机制对 Spring 来说是透明的，它并不关心那些，那些是对应各个平台需要关心的，所以 Spring 事务管理的一个优点就是为不同的事务 API 提供一致的编程模型，如 JTA、JDBC、Hibernate、JPA。下面分别介绍各个平台框架实现事务管理的机制。

##### **JDBC 事务**

如果应用程序中直接使用 JDBC 来进行持久化，此时使用 DataSourceTransactionManager 来处理事务边界。为了使用DataSourceTransactionManager，需要使用如下的 XML 将其装配到应用程序的上下文定义中：

<bean id="transactionManager"  
class="org.springframework.jdbc.datasource.DataSourceTransactionManager">  
 <property name="dataSource" ref="dataSource" />  
</bean>

实际上，DataSourceTransactionManager 是通过调用 java.sql.Connection 来管理事务，而后者是通过 DataSource 获取到的。通过调用连接的 commit() 方法来提交事务，同样，事务失败则通过调用 rollback() 方法进行回滚。

##### **Hibernate 事务**

如果应用程序的持久化是通过 Hibernate 实现的，那么你需要使用 HibernateTransactionManager。对于 Hibernate3，需要在 Spring 上下文定义中添加如下的声明：

<bean id="transactionManager"  
class="org.springframework.orm.hibernate3.HibernateTransactionManager">  
 <property name="sessionFactory" ref="sessionFactory" />  
</bean>

sessionFactory 属性需要装配一个 Hibernate 的 session 工厂，HibernateTransactionManager 的实现细节是它将事务管理的职责委托给 org.hibernate.Transaction 对象，而后者是从 Hibernate Session 中获取到的。当事务成功完成时，HibernateTransactionManager 将会调用 Transaction 对象的 commit() 方法，反之，将会调用 rollback() 方法。

##### **Java 持久化 API 事务（JPA）**

Hibernate 多年来一直是 Java 持久化标准，但是现在 Java 持久化 API 作为真正的 Java 持久化标准进入大家的视野。如果你计划使用 JPA 的话，那你需要使用 Spring 的 JpaTransactionManager 来处理事务。你需要在 Spring 中这样配置 JpaTransactionManager：

<bean id="transactionManager"  
class="org.springframework.orm.jpa.JpaTransactionManager">  
 <property name="sessionFactory" ref="sessionFactory" />  
</bean>

JpaTransactionManager 只需要装配一个 JPA 实体管理工厂（javax.persistence.EntityManagerFactory 接口的任意实现）。 JpaTransactionManager 将与由工厂所产生的 JPA EntityManager 合作来构建事务。

##### **Java 原生 API 事务**

如果应用程序没有使用以上所述的事务管理，或者是跨越了多个事务管理源（比如两个或者是多个不同的数据源），此时需要使用 JtaTransactionManager：

<bean id="transactionManager"  
class="org.springframework.transaction.jta.JtaTransactionManager">  
 <property name="transactionManagerName" value="java:/TransactionManager" />  
</bean>

JtaTransactionManager 将事务管理的责任委托给 javax.transaction.UserTransaction 和javax.transaction.TransactionManager 对象，其中事务成功完成通过 UserTransaction.commit() 方法提交，事务失败通过 UserTransaction.rollback() 方法回滚。

### **Spring 事务控制配置**

通过 jdbc 持久化事务，对于事务配置实现由两种方式即：Xml 配置，注解配置。

#### **XML 配置**

##### **添加命名空间**

在spring.xml配置文件的添加事务和aop的命名空间

事务

xmlns:tx="http://www.springframework.org/schema/tx"  
  
http://www.springframework.org/schema/tx  
http://www.springframework.org/schema/tx/spring-tx.xsd

AOP

xmlns:aop="http://www.springframework.org/schema/aop"  
  
http://www.springframework.org/schema/aop  
http://www.springframework.org/schema/aop/spring-aop.xsd

配置如下

<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:tx="http://www.springframework.org/schema/tx"  
 xmlns:aop="http://www.springframework.org/schema/aop"  
 xsi:schemaLocation="http://www.springframework.org/schema/beans  
 https://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/tx  
 http://www.springframework.org/schema/tx/spring-tx.xsd  
 http://www.springframework.org/schema/aop  
 http://www.springframework.org/schema/aop/spring-aop.xsd">

##### **设置aop代理**

<!-- 开启AOP代理 -->  
<aop:aspectj-autoproxy />

##### **配置事务管理器**

<!-- 事务管理器定义 -->  
<bean id="txManager" class="org.springframework.jdbc.datasource.DataSourceTransactionManager">  
 <!--数据源 -->  
 <property name="dataSource" ref="dataSource"></property>  
</bean>

##### **配置事务相关通知**

一般来说增删改方法 propagation=Required，对于查询方法使用 read-only="true"

<!-- 配置事务通知 transaction-manager属性表示这个事务通知是哪个事务管理器管理的-->  
<!--  
 tx:method的属性:  
 name   
 是必须的,表示与事务属性关联的方法名(业务方法名),对切入点进行细化。  
 通配符（\*）可以用来指定一批关联到相同的事务属性的方法。  
 如：'get\*'、'handle\*'、'on\*Event'等等.  
 propagation   
 不是必须的，默认值是REQUIRED  
 表示事务传播行为, 包括：   
 REQUIRED,SUPPORTS,MANDATORY,NEVER  
 REQUIRES\_NEW,NOT\_SUPPORTED,NESTED  
 isolation   
 不是必须的，默认值DEFAULT  
 表示事务隔离级别(数据库的隔离级别)  
 timeout   
 不是必须的，默认值-1(永不超时)  
 表示事务超时的时间（以秒为单位）  
 read-only   
 不是必须的，默认值false不是只读的  
 表示事务是否只读  
 rollback-for   
 不是必须的  
 表示将被触发进行回滚的 Exception(s)；以逗号分开。  
 如：'com.foo.MyBusinessException,ServletException'  
 no-rollback-for   
 不是必须的  
 表示不被触发进行回滚的 Exception(s)；以逗号分开。  
 如：'com.foo.MyBusinessException,ServletException'  
 任何 RuntimeException 将触发事务回滚  
 -->  
<tx:advice id="txAdvice" transaction-manager="txManager">  
 <!--对以add update delete query开头的所有方法进行事务处理-->  
 <tx:attributes>  
 <!--定义什么方法需要使用事务 name代表的是方法名（或方法匹配）-->  
 <!-- 匹配以 add 开头的所有方法均加入事务 -->  
 <tx:method name="add\*" propagation="REQUIRED" />  
 <!-- 匹配以 update 开头的所有方法均加入事务 -->  
 <tx:method name="update\*" propagation="REQUIRED" />  
 <!-- 匹配以 delete 开头的所有方法均加入事务 -->  
 <tx:method name="delete\*" propagation="REQUIRED" />  
 <!-- 匹配以 query 开头的所有方法均加入事务 -->  
 <tx:method name="query\*" read-only="true" />  
 </tx:attributes>  
</tx:advice>

事务传播行为介绍:  
 @Transactional(propagation=Propagation.REQUIRED)  
 如果有事务, 那么加入事务, 没有的话新建一个(默认情况下)  
 @Transactional(propagation=Propagation.NOT\_SUPPORTED)  
 容器不为这个方法开启事务  
 @Transactional(propagation=Propagation.REQUIRES\_NEW)  
 不管是否存在事务,都创建一个新的事务,原来的挂起,新的执行完毕,继续执行老的事务  
 @Transactional(propagation=Propagation.MANDATORY)  
 必须在一个已有的事务中执行,否则抛出异常  
 @Transactional(propagation=Propagation.NEVER)  
 必须在一个没有的事务中执行,否则抛出异常(与 Propagation.MANDATORY 相反)  
 @Transactional(propagation=Propagation.SUPPORTS)  
 如果其他 bean 调用这个方法,在其他 bean 中声明事务,那就用事务.  
 如果其他 bean 没有声明事务,那就不用事务.  
 @Transactional(propagation=Propagation.NESTED)   
 支持当前事务，如果当前事务存在，则执行一个嵌套事务，如果当前没有事务，就新建一个事务。

##### **配置aop**

<!-- aop 切面定义 （切入点和通知） -->  
<aop:config>  
 <!-- 设置切入点 设置需要被拦截的方法 -->  
 <aop:pointcut expression="execution(\* com.xxxx.service..\*.\*(..) )" id="cut" />  
 <!-- 设置通知 事务通知 -->  
 <aop:advisor advice-ref="txAdvice" pointcut-ref="cut"/>  
<aop:adviso

#### **注解配置**

##### **配置事务管理器**

<!-- spring 注解式事务声明 -->  
<!-- 事务管理器定义 -->  
<bean id="txManager"  
class="org.springframework.jdbc.datasource.DataSourceTransactionManager">  
 <property name="dataSource" ref="dataSource"></property>  
</bean>

##### **配置注解支持**

<tx:annotation-driven transaction-manager="txManager"/>

##### **方法上加入事务注解**

Service 方法上在需要添加事务的方法上加入事务注解

@Override  
@Transactional(propagation=Propagation.REQUIRED)  
public void saveUser(String userName,String userPwd){  
 User user1=new User();  
 user1.setUserName(userName);  
 user1.setUserPwd(userPwd);  
 userDao.saveUser(user1);  
 userDao.delUserById(2);  
}

备注：默认 spring 事务只在发生未被捕获的 runtimeexcetpion 时才回滚。

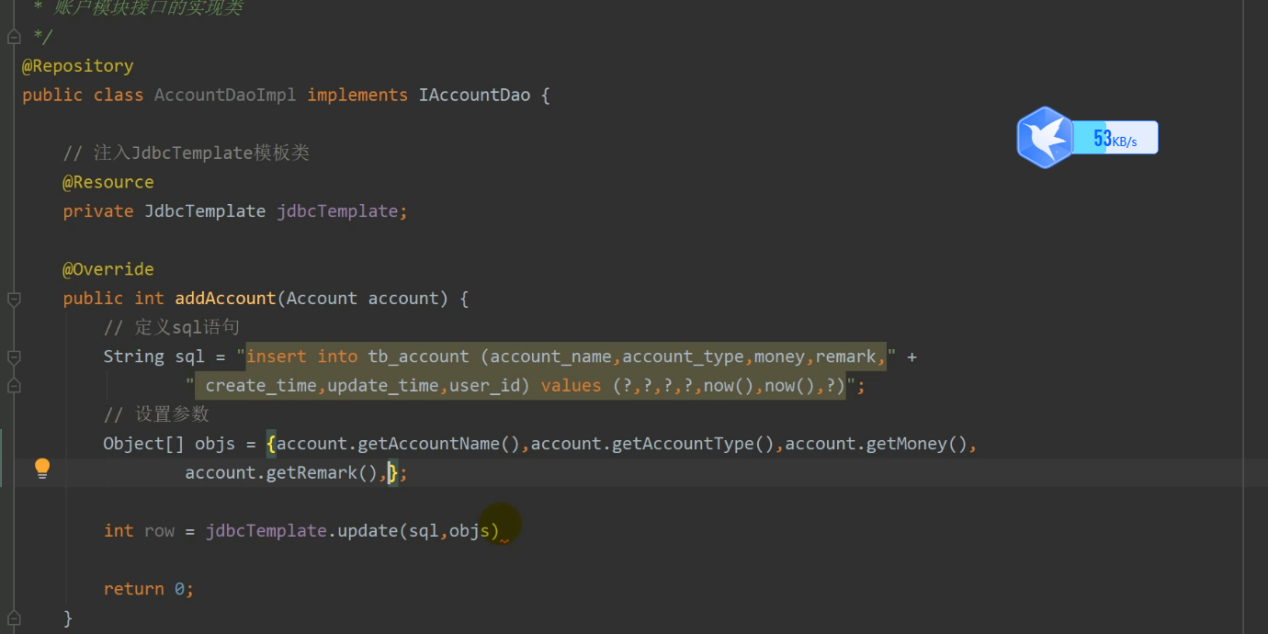
spring aop 异常捕获原理：

被拦截的方法需显式抛出异常，并不能经任何处理，这样aop 代理才能捕获到方法的异常，才能进行回滚，默认情况下 aop 只捕获 runtimeexception 的异常，但可以通过配置来捕获特定的异常并回滚换句话说在 service 的方法中不使用 try catch 或者在 catch 中最后加上 throw new RunTimeexcetpion()，这样程序异常时才能被 aop 捕获进而回滚.

# Myself part

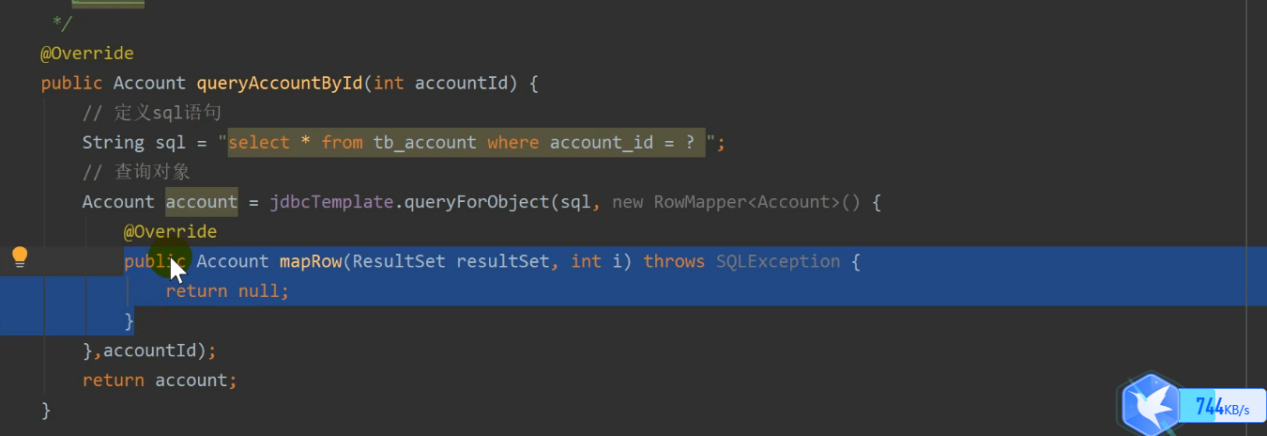
Set the argument by obj array

Jdbc template



增加并返回主键

根据id 返回单个对象





事务管理器中 注入数据源

Tx 中 advice 标签 使用通知

