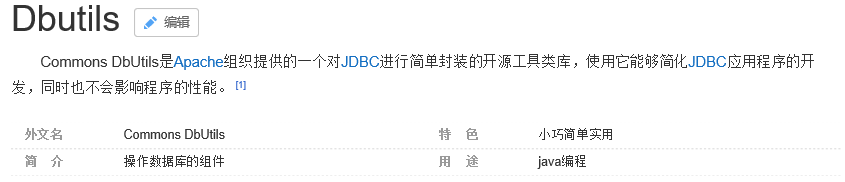
# JDBC高级开发事务

# DBUtils 框架

**DAO 层代码书写： 封装的过程很繁琐, 可以使用DBUtils工具类完成 DAO 的封装 :**



DBUtils学习

1、**QueryRunner** 框架核心类 ，所有数据库操作都是必须通过 QueryRunner 进行的

2、**ResultSetHandler** 结果集封装接口，完成将ResultSet 结果集 封装为一个Java对象

3、DbUtils 工具类 提供驱动管理、事务管理、释放资源等一系列公共方法

QueryRunner 与 ResultSetHandler 一起使用，完成数据表增删改查

构造器：

QueryRunner() -------- 没有传递连接池给DBUtils 框架，框架不能获得数据库连接，接下来每个操作，必须将数据库连接传给框架 （手动管理事务）

QueryRunner(DataSource ds) ---- 将连接池给DBUtils框架，以后每个操作，都会从连接池中获取一个新的连接 （每条SQL 一个单独的事务）

更新操作 insert update delete

public int update(Connection conn, String sql, Object... params) ---- 手动管理事务，没有将连接池提供框架，传入连接

public int update(String sql, Object... params) ----- 将连接池交给框架，由框架管理事务，不需要传入连接

查询操作 select

public Object query(Connection conn, String sql, ResultSetHandler<T> rsh, Object... params)

public Object query(String sql, ResultSetHandler<T> rsh, Object... params)

**1、需要手动管理事务**

QueryRunner()

public int update(Connection conn, String sql, Object... params)

query(Connection conn, String sql, ResultSetHandler<T> rsh, Object... params)

**2、由框架管理事务 （每条SQL都是一个单独事务）**

QueryRunner(DataSource ds)

public int update(String sql, Object... params)

public Object query(String sql, ResultSetHandler<T> rsh, Object... params)

## DBUtils 工具的 `增删改` 操作 :

@Test

**public** **void** test\_insert() **throws** SQLException {

// 1. 创建 QueryRunner 对象, 同时将 `数据库` 对象传入

QueryRunner queryRunner = **new** QueryRunner(JDBCUtils.*getDataSource*());

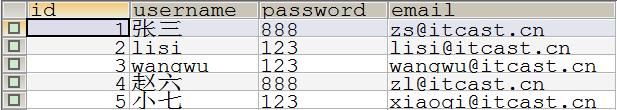
// 2. 执行 update 方法

String sql = "insert into user values(?,?,?,?);";

Object[] params = {5, "小七", "123", "xiaoqi@itcast.cn"};

queryRunner.update(sql, params);

}



@Test

**public** **void** test\_delete() **throws** SQLException {

// 1. 创建 QueryRunner 对象, 同时将 `数据库` 对象传入

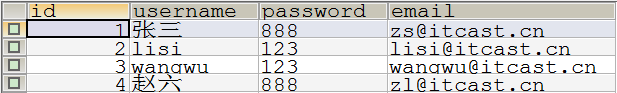
QueryRunner queryRunner = **new** QueryRunner(JDBCUtils.*getDataSource*());

// 2. 执行 update 方法

String sql = "delete from user where id=?;";

queryRunner.update(sql, 5);

}



@Test

**public** **void** test\_update() **throws** SQLException {

// 1. 创建 QueryRunner 对象, 同时将 `数据库` 对象传入

QueryRunner queryRunner = **new** QueryRunner(JDBCUtils.*getDataSource*());

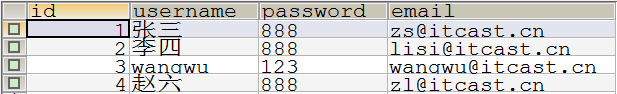
// 2. 执行 update 方法

String sql = "update user set username=?, password=? where id=?;";

Object[] params = {"李四", "888", 2};

queryRunner.update(sql, params);

}



## DBUtils 工具的 `查询` 操作 :

@Test

**public** **void** test\_query() **throws** SQLException {

// 1. 创建 QueryRunner 对象, 同时将 `数据库` 对象传入

QueryRunner queryRunner = **new** QueryRunner(JDBCUtils.*getDataSource*());

// 2. 执行 query 方法

String sql = "select \* from user where id = ?;";

User user = queryRunner.query(sql, **new** ResultSetHandler<User>(){

@Override

**public** User handle(ResultSet rs) **throws** SQLException {

**if** (rs.next()) {

**int** id = rs.getInt("id");

String username = rs.getString("username");

String password = rs.getString("password");

String email = rs.getString("email");

User user = **new** User();

user.setId(id);

user.setUsername(username);

user.setPassword(password);

user.setEmail(email);

**return** user;

}

**return** **null**;

}

}, 3);

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

}



@Test

**public** **void** test\_query() **throws** SQLException {

// 1. 创建 QueryRunner 对象, 同时将 `数据库` 对象传入

QueryRunner queryRunner = **new** QueryRunner(JDBCUtils.*getDataSource*());

// 2. 执行 query 方法

String sql = "select \* from user;";

List<User> list = queryRunner.query(sql, **new** ResultSetHandler<List<User>>(){

@Override

**public** List<User> handle(ResultSet rs) **throws** SQLException {

List<User> list = **new** ArrayList<User>();

User user = **null**;

**while** (rs.next()) {

user = **new** User();

**int** id = rs.getInt("id");

String username = rs.getString("username");

String password = rs.getString("password");

String email = rs.getString("email");

user.setId(id);

user.setUsername(username);

user.setPassword(password);

user.setEmail(email);

list.add(user);

}

**return** list;

}

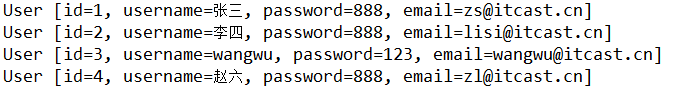
});

**for** (User user : list) {

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

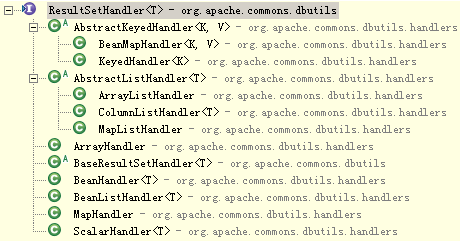
}

}



## ResultSetHandler 的实现类使用

**说明 : 结果集处理类, 将从数据库中获取的数据进行封装处理, 从Dao层返回数据给 Service 层.**



|  |  |
| --- | --- |
| ArrayHandler | 将结果集中的第一条记录封装到一个Object[]数组中，数组中的每一个元素就是这条记录中的每一个字段的值 |
| ArrayListHandler | 将结果集中的每一条记录都封装到一个Object[]数组中，将这些数组在封装到List集合中。 |
| BeanHandler | 将结果集中第一条记录封装到一个指定的javaBean中。 |
| BeanListHandler | 将结果集中每一条记录封装到指定的javaBean中，将这些javaBean在封装到List集合中 |
| ColumnListHandler | 将结果集中指定的列的字段值，封装到一个List集合中 |
| KeyedHandler | 将结果集中每一条记录封装到Map<String,Object>,在将这个map集合做为另一个Map的value,另一个Map集合的key是指定的字段的值。 |
| MapHandler | 将结果集中第一条记录封装到了Map<String,Object>集合中，key就是字段名称，value就是字段值 |
| MapListHandler | 将结果集中每一条记录封装到了Map<String,Object>集合中，key就是字段名称，value就是字段值，在将这些Map封装到List集合中。 |
| ScalarHandler | 它是用于单数据。例如select count(\*) from 表操作。 |

ResultSetHandler 在DBUtils 框架中提供九个默认 实现类，直接使用九个默认实现类，可以完成常规操作，而不需要自定义结果集封装

1) ArrayHandler 和 ArrayListHandler 将数据表的每行记录保存Object[] 中

2) BeanHandler 和 BeanListHandler 将数据表每行记录 保存JavaBean对象中

\* 封装javabean属性时，必须保证数据表列名与 javabean属性名一致，否则无法封装

3) MapHandler和 MapListHandler 将结果每行记录保存到一个Map集合，key是列名，value是值

4) ColumnListHandler 查询结果集中指定一列数据

5) KeyedHandler(name) 结果集每行数据封装map，再将map存入另一个map 作为value，指定一列作为key

6) ScalarHandler 进行单值查询 select count(\*) from account;

**重点：BeanHandler 和 BeanListHandler、ScalarHandler**

### BeanHandler

@Test

**public** **void** test\_BeanHandler() **throws** SQLException {

// 1. 创建QueryRunner对象, 同时将 `数据库` 对象传入

QueryRunner queryRunner = **new** QueryRunner(JDBCUtils.*getDataSource*());

// 2. 执行 query 方法

String sql = "select \* from user where id=?;";

User user = queryRunner.query(sql, **new** BeanHandler<User>(User.**class**), 3);

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

}



### BeanListHandler

@Test

**public** **void** test\_BeanListHandler() **throws** SQLException {

// 1. 创建QueryRunner对象, 同时将 `数据库` 对象传入

QueryRunner queryRunner = **new** QueryRunner(JDBCUtils.*getDataSource*());

// 2. 执行 query 方法

String sql = "select \* from user;";

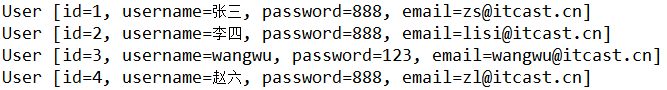
List<User> list = queryRunner.query(sql, **new** BeanListHandler<User>(User.**class**));

**for** (User user : list) {

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

}

}



### ScalarHandler

@Test

**public** **void** test\_scalarHandler() **throws** SQLException {

// 1. 创建QueryRunner对象, 同时将 `数据库` 对象传入

QueryRunner queryRunner = **new** QueryRunner(JDBCUtils.*getDataSource*());

// 2. 执行 query 方法

String sql = "select count(\*) from user;";

Long result = queryRunner.query(sql, **new** ScalarHandler<Long>());

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

}



### MapHandler

@Test

**public** **void** test\_MapHandler() **throws** SQLException {

// 1. 创建QueryRunner对象, 同时将 `数据库` 对象传入

QueryRunner queryRunner = **new** QueryRunner(JDBCUtils.*getDataSource*());

String sql = "select \* from user;";

Map<String, Object> map = queryRunner.query(sql, **new** MapHandler());

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

}



### MapListHandler

@Test

**public** **void** test\_MapListHandler() **throws** SQLException {

// 1. 创建QueryRunner对象, 同时将 `数据库` 对象传入

QueryRunner queryRunner = **new** QueryRunner(JDBCUtils.*getDataSource*());

String sql = "select \* from user;";

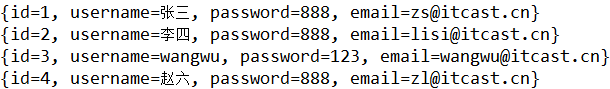
List<Map<String, Object>> list = queryRunner.query(sql, **new** MapListHandler());

**for** (Map<String, Object> map : list) {

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

}

}



### ArrayHandler

@Test

**public** **void** test\_ArrayHandler() **throws** SQLException {

// 1. 创建QueryRunner对象, 同时将 `数据库` 对象传入

QueryRunner queryRunner = **new** QueryRunner(JDBCUtils.*getDataSource*());

// 2. 执行 query 操作

String sql = "select \* from user;";

Object[] arr = queryRunner.query(sql, **new** ArrayHandler());

System.***out***.println(Arrays.*toString*(arr));

}



### ArrayListHandler

@Test

**public** **void** test\_ArrayListHandler() **throws** SQLException {

// 1. 创建QueryRunner对象, 同时将 `数据库` 对象传入

QueryRunner queryRunner = **new** QueryRunner(JDBCUtils.*getDataSource*());

// 2. 执行 query 操作

String sql = "select \* from user;";

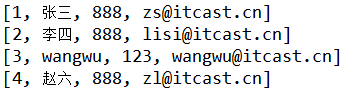
List<Object[]> list = queryRunner.query(sql, **new** ArrayListHandler());

**for** (Object[] arr : list) {

System.***out***.println(Arrays.*toString*(arr));

}

}



### ColumnListHandler

@Test

**public** **void** test\_ColumnListHandler() **throws** SQLException {

// 1. 创建QueryRunner对象, 同时将 `数据库` 对象传入

QueryRunner queryRunner = **new** QueryRunner(JDBCUtils.*getDataSource*());

// 2. 执行 query 操作

String sql = "select \* from user;";

List<String> list = queryRunner.query(sql, **new** ColumnListHandler<String>("username"));

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

}



### KeyedHandler

@Test

**public** **void** test\_KeyedHandler() **throws** SQLException {

// 1. 创建QueryRunner对象, 同时将 `数据库` 对象传入

QueryRunner queryRunner = **new** QueryRunner(JDBCUtils.*getDataSource*());

// 2. 执行 query 操作

String sql = "select \* from user;";

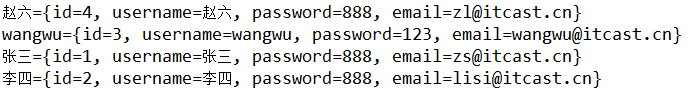
Map<User, Map<String, Object>> map = queryRunner.query(sql, **new** KeyedHandler<User>("username"));

**for** (Entry<User, Map<String, Object>> entry : map.entrySet()) {

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

}

}



# 转账案例 - 案例分析

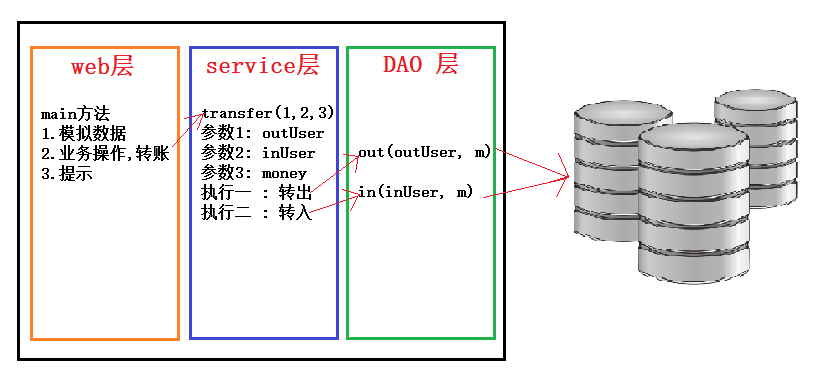
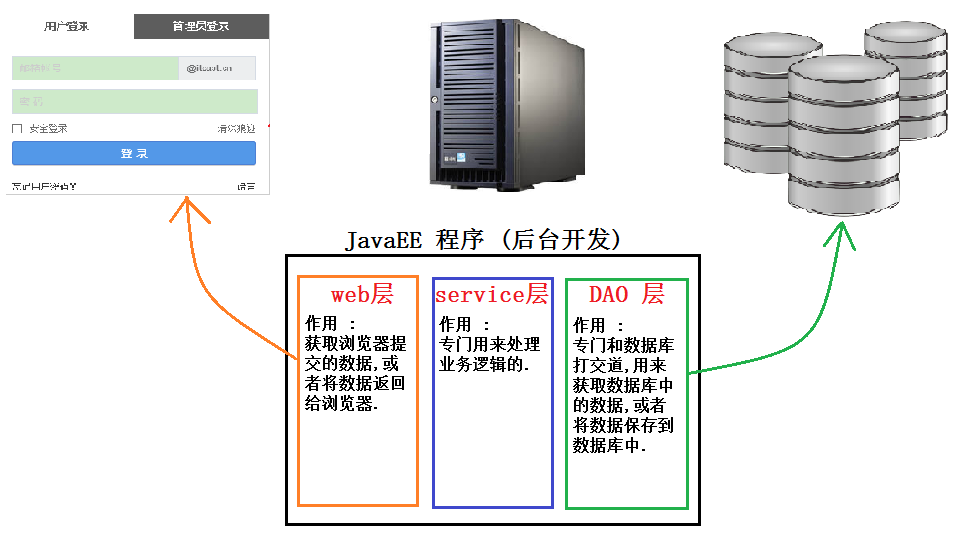
案例分析：之前给大家介绍过三层开发，相信大家对三层有了基本的了解，接下来我们就使用三层的开发模式来完成转账的案例

1 首先在这里，我们web层就直接通过main方法，直接填充数据，调用service层的业务逻辑。

2 而service业务逻辑层主要完成转账的核心业务逻辑操作，包括转出和转入的2个核心操作。

3 很明显，转入和转出是和数据库打交道的，所以需要将转入和转出这2个操作放在dao层。

大体分工如下图所示：



* 开发中，常使用分层思想
  + 不同的层次结构分配不同的解决过程，各个层次间组成严密的封闭系统
  + 不同层级结构彼此平等
  + 分层的目的是：
    - 解耦
    - 可维护性
    - 可扩展性
    - 可重用性
* 不同层次，使用不同的包表示
  + com.itheima 公司域名倒写
  + com.itheima.dao dao层
  + com.itheima.service service层
  + com.itheima.domain javabean
  + com.itheima.utils 工具

## 代码实现

* 步骤1：准备数据

-- 创建一个数据库

create database day06;

-- 使用数据库

use day06;

-- 创建一张账户表

create table account(

id int primary key auto\_increment,

name varchar(20),

money double

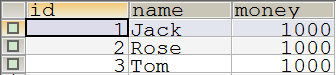
);

-- 初始化数据

insert into account values(null, 'Jack', 1000);

insert into account values(null, 'Rose', 1000);

insert into account values(null, 'Tom', 1000);



* 步骤2：工具类的准备

C3p0工具类的配置：

<?xml version=*"1.0"* encoding=*"UTF-8"*?>

<c3p0-config>

<!-- 默认配置, c3p0框架默认架子啊这段默认配置 -->

<default-config>

<!-- 配置 JDBC 四个基本属性 -->

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

<property name=*"jdbcUrl"*>jdbc:mysql:///day04</property>

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

<property name=*"password"*>123</property>

</default-config>

<!-- 可以自定义配置, 为这段配置起一个名字, c3p0指定该名称加载配置 -->

<named-config name=*"day06"*>

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

<property name=*"jdbcUrl"*>jdbc:mysql:///day06</property>

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

<property name=*"password"*>123</property>

</named-config>

</c3p0-config>

**工具类的编写：**

**public** **class** JDBCUtils {

// c3p0 数据库连接池对象属性

**private** **static** **final** ComboPooledDataSource ***dataSource*** = **new** ComboPooledDataSource("day06");

// 获取连接

**public** **static** Connection getConnection() **throws** SQLException {

**return** ***dataSource***.getConnection();

}

// 获取数据库连接池对象

**public** **static** DataSource getDataSource() {

**return** ***dataSource***;

}

// 释放资源

**public** **static** **void** release(Connection conn, Statement stmt, ResultSet rs) {

**if** (rs != **null**) {

**try** {

rs.close();

} **catch** (SQLException e) {

e.printStackTrace();

}

rs = **null**;

}

*release*(conn, stmt);

}

**public** **static** **void** release(Connection conn, Statement stmt) {

**if** (stmt != **null**) {

**try** {

stmt.close();

} **catch** (SQLException e) {

e.printStackTrace();

}

stmt = **null**;

}

**if** (conn != **null**) {

**try** {

conn.close();

} **catch** (SQLException e) {

e.printStackTrace();

}

conn = **null**;

}

}

}

* 步骤3：编写dao层

**public** **class** AccountDao {

// 转出

**public** **void** out(String outUser, **int** money) **throws** SQLException {

QueryRunner queryRunner = **new** QueryRunner(JDBCUtils.*getDataSource*());

String sql = "update account set money = money - ? where name = ?;";

Object[] params = {money, outUser};

queryRunner.update(sql, params);

}

// 转入

**public** **void** in(String inUser, **int** money) **throws** SQLException {

QueryRunner queryRunner = **new** QueryRunner(JDBCUtils.*getDataSource*());

String sql = "update account set money = money + ? where name = ?;";

Object[] params = {money, inUser};

queryRunner.update(sql, params);

}

}

* 步骤3：编写AccountService

**public** **class** AccountService {

// 转账操作

**public** **boolean** transfer(String outUser, String inUser, **int** money) {

AccountDao accountDao = **new** AccountDao();

**try** {

// 转出

accountDao.out(outUser, money);

// 转入

accountDao.in(inUser, money);

} **catch** (Exception e) {

e.printStackTrace();

// 一旦发生异常, 表示转账失败了.

**return** **false**;

}

**return** **true**;

}

}

* 步骤4：web层，也就是这里转账的程序入口.

**public** **class** AccountWeb {

**public** **static** **void** main(String[] args) {

// 模拟数据

// Jack -> Rose 100

String outUser = "Jack";

String inUser = "Rose";

**int** money = 100;

AccountService accountService = **new** AccountService();

**boolean** result = accountService.transfer(outUser, inUser, money);

**if** (result == **false**) {

System.***out***.println("转账失败!");

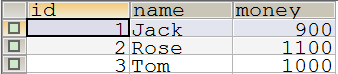
} **else** {

System.***out***.println("转账成功!");

}

}

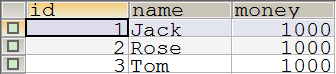
}



思考：在上面的案例中，在业务层里面，假设在一个人付款成功，另一个人还没有收款的时候，中间有其他业务操作的话，出现了异常，

也就是在付款和收款之间加上如下图所示代码，模拟转账的异常。

**注意 : 将数据改回到 10000.**



**public** **class** AccountService {

// 转账操作

**public** **boolean** transfer(String outUser, String inUser, **int** money) {

AccountDao accountDao = **new** AccountDao();

**try** {

// 转出

accountDao.out(outUser, money);

**int** i = 10 / 0;

// 转入

accountDao.in(inUser, money);

} **catch** (Exception e) {

e.printStackTrace();

// 一旦发生异常, 表示转账失败了.

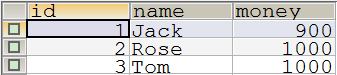
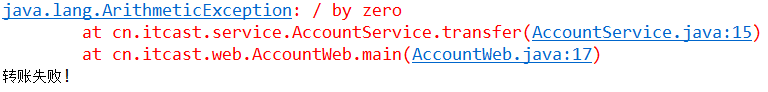
**return** **false**;

}

**return** **true**;

}

}



那么这时候很明显，程序终止，而后面收款的操作并不会去做，那么另一个人就不会收到钱，而前一个人确确实实转账了。所以最终得出。上面的转账业务逻辑是存在bug的。那么针对这样的一个bug我们该如何去处理呢？

针对上面出现的问题的分析：

上面的bug，因为异常的原因，导致了我们的一个完整的业务逻辑被中断了。所以导致只执行了一条sql语句，后一条sql语句没有去执行。因此，我们要保证，我们一个完整的业务逻辑要么全部跑完，如果中间出现异常的话，为了保证最终的结果符合我们实际生活的一个情况，所以我们需要撤销与当前业务逻辑有关的操作。所以这就需要学习接来下的事务来保证这样一个情况。

## 事务概述

* 事务指的是逻辑上的一组操作,组成这组操作的各个单元要么全都成功,要么全都失败.
* 事务作用：保证在一个事务中多次操作要么全都成功,要么全都失败.

事务的出现 解决上面的问题。

事务是如何处理正常情况的呢？

a=1000 b=1000

开启事务

update account set money = money -100 where name='a';

update account set money = money +100 where name='b';

提交事务 （sql语句对数据库产生的操作才会被永久的保存）

事务是如何处理异常情况的呢？

a=1000 b=1000

开启事务

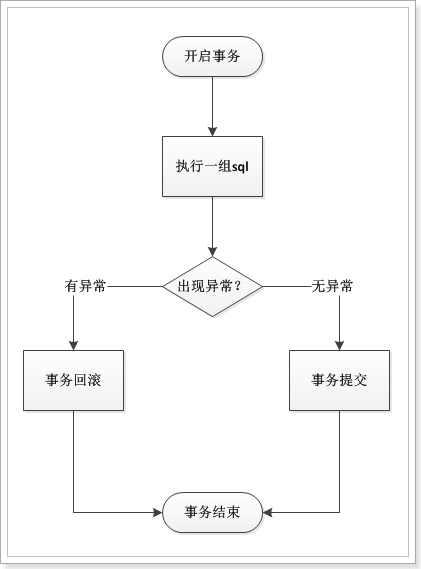
update account set money = money -100 where name='a'; a=900

出现异常

update account set money = money +100 where name='b';

事务的回滚（撤销已经成功执行的sql语句）

A=1000 b=1000;

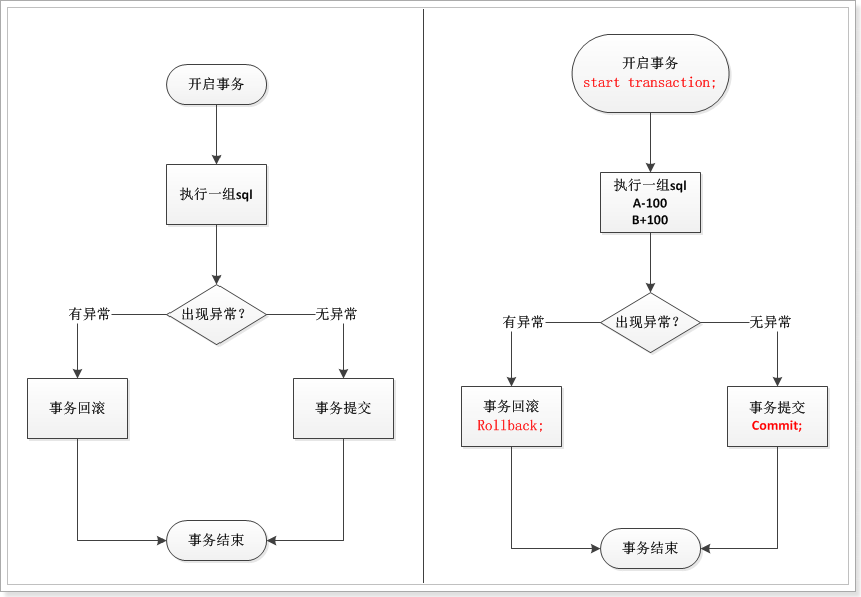


简单总结：

|  |  |
| --- | --- |
| 开启事务 | |
| 执行sql语句群 | |
| 出现异常 回滚事务（撤销） | 无异常 事务提交（生效） |

### mysql事务操作

|  |  |
| --- | --- |
| sql语句 | 描述 |
| start transaction; | 开启事务(mysql 默认事务自提交,我们应该设置手动提交) |
| commit; | 提交事务 |
| rollback; | 回滚事务 |



* 操作：
  + MYSQL中可以有两种方式进行事务的管理：
    - 自动提交：MySql默认自动提交。及执行一条sql语句提交一次事务。
    - 手动提交：先开启，再提交
  + 方式1：手动提交

start transaction;

update account set money=money-1000 where name='Jack';

update account set money=money+1000 where name='Rose';

commit;

#或者

rollback;

* + 方式2：自动提交，通过修改mysql全局变量“autocommit”进行控制

show variables like '%commit%';



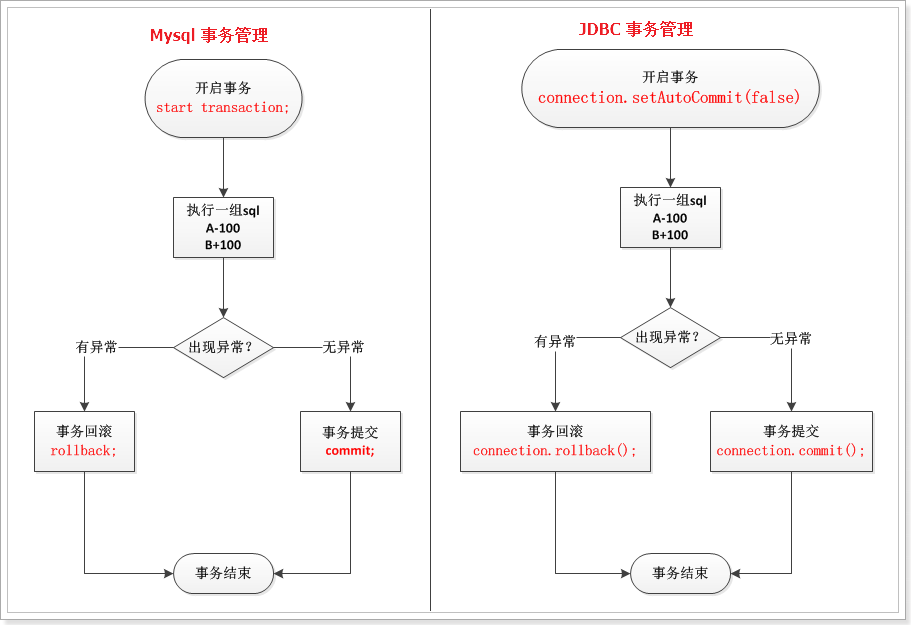
\* 设置自动提交的参数为OFF:

set autocommit = 0; -- 0:OFF 1:ON

* 扩展：Oracle数据库事务不自动提交

### JDBC事务操作

|  |  |
| --- | --- |
| Connection对象的方法名 | 描述 |
| conn.setAutoCommit(**false**) | 开启事务 start transaction; |
| conn.commit() | 提交事务 |
| conn.rollback() | 回滚事务 |



**注意**：在jdbc事务操作中，事务的控制都是通过Connection对象完成的，当一个完整的业务操作前，我们首先使用connection.setAutoCommit(false)来开启事务，当业务操作完成之后，我们需要使用connection.commit()来提交事务。当然了，如果出现了异常，我们需要撤销所有的操作，所以出现异常，需要进行事务的回滚。

如下是jdbc操作事务的模板代码，具体操作我们会在我们的转账案例中完成。

//事务模板代码

public void demo01() throws SQLException{

// 获得连接

Connection conn = null;

try {

//#1 开始事务

conn.setAutoCommit(false);

//.... 加钱 ,减钱

//#2 提交事务

conn.commit();

} catch (Exception e) {

//#3 回滚事务

conn.rollback();

} finally{

// 释放资源

conn.close();

}

}

#### Web层

**public** **class** AccountWeb {

**public** **static** **void** main(String[] args) {

// 1. 模拟数据

String outUser = "Jack";

String inUser = "Rose";

**int** money = 100;

// 2. 业务转账的操作

AccountService accountService = **new** AccountService();

**boolean** result = accountService.tranfer(outUser, inUser, money);

// 3. 判断

**if** (result == **false**) {

System.***out***.println("转账失败!");

} **else** {

System.***out***.println("转账成功!");

}

}

}

#### service层

**注意 :** service层要获取 Connection 连接对象, 开始事务, 提交事务, 回滚事务, 最后别忘了将 `连接对象` 传入到 dao层的 out, in 方法参数中.

**public** **class** AccountService {

// 转账操作

**public** **boolean** tranfer(String outUser, String inUser, **int** money) {

// 开启事务 : (由 `连接` 对象来控制)

Connection conn = **null**;

AccountDao accountDao = **new** AccountDao();

**try** {

// 获取连接对象

conn = JDBCUtils.*getConnection*();

// 开启事务

conn.setAutoCommit(**false**);

// 转出

accountDao.out(conn, outUser, money);

**int** i = 10 / 0;

// 转入

accountDao.in(conn, inUser, money);

// 提交事务 (让修改永久保存)

conn.commit();

// 返回

**return** **true**;

} **catch** (Exception e) {

e.printStackTrace();

// 回滚事务

**if** (conn != **null**) {

**try** {

conn.rollback();

} **catch** (SQLException e1) {

e1.printStackTrace();

}

conn = **null**;

}

}

**return** **false**;

}

}

#### dao层

注意 : 在执行 update 方法是一定要传入参数中的 conn 连接对象, 并且不要在创建 QueryRunner 对象是传入 `数据库连接池`.

**public** **class** AccountDao {

// Dao 层如果出异常, 如何处理 ??? 直接抛, 交给 Service 层处理.

// 转出

**public** **void** out(Connection conn, String outUser, **int** money) **throws** SQLException {

// 1. 创建一个 QueryRunner 对象

QueryRunner queryRunner = **new** QueryRunner();

// 2. 执行相对应的方法

// 准备参数

String sql = "update account set money = money - ? where name = ?;";

Object[] params = {money, outUser};

queryRunner.update(conn, sql, params);

}

// 转入

**public** **void** in(Connection conn, String inUser, **int** money) **throws** SQLException {

// 1. 创建一个 QueryRunner 对象

QueryRunner queryRunner = **new** QueryRunner();

// 2. 执行相对应的方法

// 准备参数

String sql = "update account set money = money + ? where name = ?;";

Object[] params = {money, inUser};

queryRunner.update(conn, sql, params);

}

}

### DBUtils事务操作

|  |  |
| --- | --- |
| Connection对象的方法名 | 描述 |
| conn.setAutoCommit(false) | 开启事务 |
| new QueryRunner() | 创建核心类，不设置数据源(手动管理连接) |
| query(conn , sql , handler, params ) 或  update(conn, sql , params) | 手动传递连接 |
| DbUtils.commitAndCloseQuietly(conn) 或  DbUtils.rollbackAndCloseQuietly(conn) | 提交并关闭连接  回滚并关闭连接 |

过程与JDBC操作一样，只不过需要注意的是，connection必须手动控制，不能交给DBUtils去控制。

**说明 :** Service 层代码中关于 `事务提交与回滚` 使用 DBUtils 方式实现, 代码如下 :

**public** **class** AccountService {

// 转账操作

**public** **boolean** tranfer(String outUser, String inUser, **int** money) {

// 开启事务 : (由 `连接` 对象来控制)

Connection conn = **null**;

AccountDao accountDao = **new** AccountDao();

**try** {

// 获取连接对象

conn = JDBCUtils.*getConnection*();

// 开启事务

conn.setAutoCommit(**false**);

// 转出

accountDao.out(conn, outUser, money);

**int** i = 10 / 0;

// 转入

accountDao.in(conn, inUser, money);

// 提交事务 (让修改永久保存)

DbUtils.*commitAndCloseQuietly*(conn);

// 返回

**return** **true**;

} **catch** (Exception e) {

e.printStackTrace();

// 回滚事务

DbUtils.*rollbackAndCloseQuietly*(conn);

}

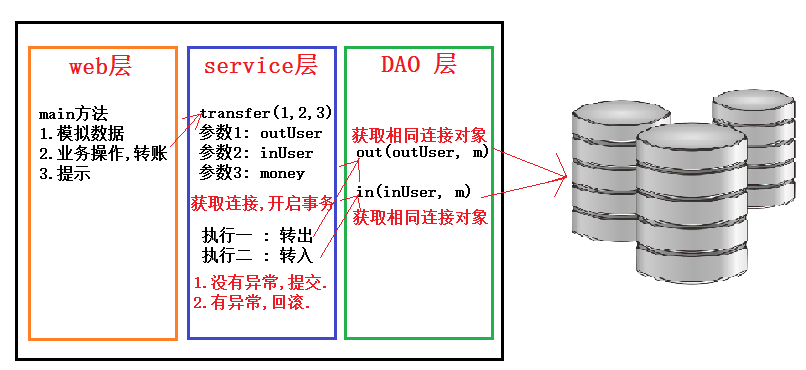
**return** **false**;

}

}

## 事务管理：传递Connection

* 修改service和dao，service将connection传递给dao，dao不需要自己获得连接



### Dao层

很明显，我们的事务是针对整个业务逻辑的。所以事务的管理应该在业务层进行管理。并且，需要注意的是，我们在整个事务的控制过程中应该是一个connection对象。所以，在业务层我们需要拿到connection对象，用connection对象对业务层进行事务处理。而既然在业务层获取了connection对象，那么就必须要保证我们在dao层拿到的connection对象和业务层拿到的connection对象是同一个，那么这里我们可以使用传递参数的方式将connection对象传递给dao层。

所以dao层需要修改，connection对象需要从业务层传递过来。并且需要注意的是，connection对象是不能在dao层关闭的，因为我们业务层还需要继续使用connection对象。所以dao层代码的修改如下：

**public** **class** AccountDao {

// 转出

**public** **void** out(Connection conn, String outUser, **int** money) **throws** SQLException {

QueryRunner queryRunner = **new** QueryRunner();

String sql = "update account set money = money - ? where name = ?;";

Object[] params = {money, outUser};

queryRunner.update(conn, sql, params);

}

// 转入

**public** **void** in(Connection conn, String inUser, **int** money) **throws** SQLException {

QueryRunner queryRunner = **new** QueryRunner();

String sql = "update account set money = money + ? where name = ?;";

Object[] params = {money, inUser};

queryRunner.update(conn, sql, params);

}

}

### service层

同样service层需要获取连接，而且这里的连接需要被传递到dao层去，并且最后需要释放资源。

**public** **class** AccountService {

// 转账操作

**public** **boolean** transfer(String outUser, String inUser, **int** money) {

Connection conn = **null**;

AccountDao accountDao = **new** AccountDao();

**try** {

// 1. 获取连接

conn = JDBCUtils.*getConnection*();

// 2. 开启事务

conn.setAutoCommit(**false**);

// 转出

accountDao.out(conn, outUser, money);

**int** i = 10 / 0;

// 转入

accountDao.in(conn, inUser, money);

// 如果转出与转入都执行成功了, 需要提交事务

conn.commit();

} **catch** (Exception e) {

e.printStackTrace();

// 一旦发生异常, 表示转账失败了.

// 需要回滚事务, 并返回 false

**if** (conn != **null**) {

**try** {

conn.rollback();

} **catch** (SQLException e1) {

e1.printStackTrace();

}

conn = **null**;

}

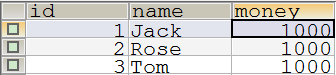
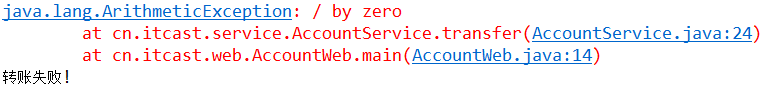
**return** **false**;

}

**return** **true**;

}

}



## 使用JDBCUtils工具类集成提交与回滚

**public** **class** JDBCUtils {

// 定义了一个 c3p0 数据库连接池对象

**private** **static** ComboPooledDataSource *dataSource* = **new** ComboPooledDataSource("day06");

// 获取连接对象

**public** **static** DataSource getDataSource() {

**return** *dataSource*;

}

// 建立连接

**public** **static** Connection getConnection() **throws** SQLException {

// 该连接对象已经从 c3p0 的数据库连接池中获取

Connection conn = *dataSource*.getConnection();

**return** conn;

}

// 释放资源

**public** **static** **void** release(ResultSet rs, Statement stmt, Connection conn) {

**if** (rs != **null**) {

**try** {

rs.close();

} **catch** (SQLException e) {

e.printStackTrace();

}

rs = **null**;

}

// 调用下面的方法释放 `执行对象和连接对象`

*release*(stmt, conn);

}

**public** **static** **void** release(Statement stmt, Connection conn) {

**if** (stmt != **null**) {

**try** {

stmt.close();

} **catch** (SQLException e) {

e.printStackTrace();

}

stmt = **null**;

}

**if** (conn != **null**) {

**try** {

conn.close(); // conn 调用任何方法都是执行 `调用处理器` 的 invoke 方法.

} **catch** (SQLException e) {

e.printStackTrace();

}

conn = **null**;

}

}

// 提交并关闭连接

**public** **static** **void** commitAndCloseQuietly(Connection conn) {

DbUtils.*commitAndCloseQuietly*(conn);

}

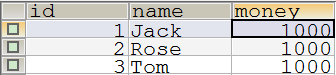
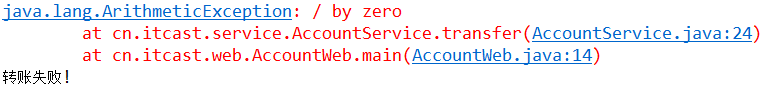
// 回滚并关闭连接

**public** **static** **void** rollbackAndCloseQuietly(Connection conn) {

DbUtils.*rollbackAndCloseQuietly*(conn);

}

}



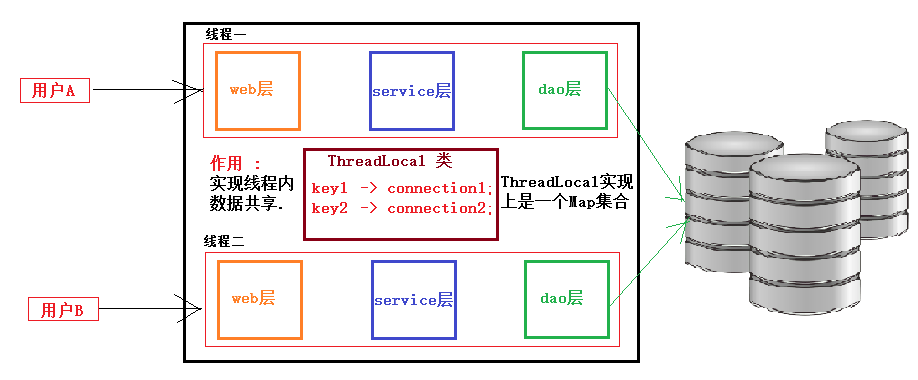
## 提高：ThreadLocal

### 相关知识与案例介绍

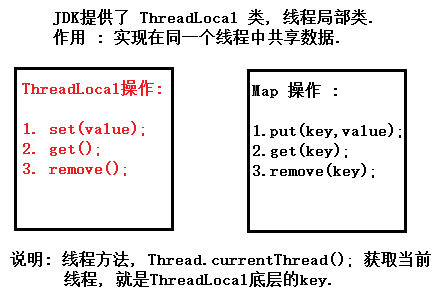
在“事务传递参数版”中，我们必须修改方法的参数个数，传递连接对象，才可以完成整个事务操作。如果不传递参数，是否可以完成？

思考：在上述事务传递参数的管理方式中，我们这样传递参数的目的无非就是为了完成在整个业务逻辑过程中任何地方所用到的都是同一个connection对象就行了。所以，如果不传递参数，我们只需要保证我们在一条业务线中，保证我们任何地方所使用的的connection对象是同一个就行了。基于这个需要，我们考虑是否可以将当前业务线中所用到的connection存到一个地方，然后让当前业务线的所有connection都从这个地方去拿。这就需要使用我们接下来学习的ThreadLocal。

ThreadLocal底层分析.



java.lang.ThreadLocal 该类提供了线程局部 (thread-local) 变量，用于在当前线程中共享数据。ThreadLocal工具类底层就是一个Map，key存放的当前线程，value存放需要共享的数据。如下图，在整个程序运行中，我们有个Map变量，每条线程又有属于自己的id，也就是thread.currentThread。所以我们可以向map中存储一个值，key就是当前线程的id，值就是我们需要存储的值。这样，我们在整个线程中的任何一个地方，我们可以通过当前线程的id从map中获取的值都是同一个。因此，我们在一条线程中，我们在service层和dao层从map中就能获取同一个connection对象了。



### 分析并实现JDBCUtils中getConnection方法的修改

程序在获取连接时, 如果 ThreadLocal类中没有当前线程对象的存储连接, 那么就从数据库连接池中获取连接, 然后将该连接存储到 ThreadLocal 中.此时,当该线程再次获取连接时,先从ThreadLocal中查找是否存在该线程的连接对象,如果有,直接返回并获取,而不再从连接池中获取,这样就达到了在同一个线程中无论获取多少次连接, 其获取的都是同一个连接对象.

工具类的获取连接方法的改写，首先从local中获取连接，获取不到, 再从数据库连接池去获取

**public** **class** JDBCUtils {

// c3p0 数据库连接池对象属性

**private** **static** **final** ComboPooledDataSource dataSource = **new** ComboPooledDataSource("day06");

// 创建一个 ThreadLocal 对象

**private** **static** **final** ThreadLocal<Connection> local = **new** ThreadLocal<Connection>();

// 获取连接

**public** **static** Connection getConnection() **throws** SQLException {

Connection conn = **null**;

// Connection 对象优先从 ThreadLocal 中获取

conn = local.get();

// 如果 conn 为 null

**if** (conn == **null**) {

// 说明当前线程是第一次获取连接对象, 所以需要从数据库连接池中获取连接对象

conn = dataSource.getConnection();

// 然后将该连接对象存储到 local 中

local.set(conn);

}

// 返回连接

**return** conn;

}

// 获取数据库连接池对象

**public** **static** DataSource getDataSource() {

**return** dataSource;

}

// 提交

**public** **static** **void** commitAndCloseQuietly(Connection conn) {

// 调用 DBUtils 框架的方法

DbUtils.commitAndCloseQuietly(conn);

}

// 回滚

**public** **static** **void** rollbackAndCloseQuietly(Connection conn) {

// 调用 DBUtils 框架的方法

DbUtils.rollbackAndCloseQuietly(conn);

}

// 释放资源

**public** **static** **void** release(Connection conn, Statement stmt, ResultSet rs) {

**if** (rs != **null**) {

**try** {

rs.close();

} **catch** (SQLException e) {

e.printStackTrace();

}

rs = **null**;

}

release(conn, stmt);

}

**public** **static** **void** release(Connection conn, Statement stmt) {

**if** (stmt != **null**) {

**try** {

stmt.close();

} **catch** (SQLException e) {

e.printStackTrace();

}

stmt = **null**;

}

**if** (conn != **null**) {

**try** {

conn.close();

// 如果当前线程关闭连接时, 将 ThreadLocal 中存储的连接清除

local.remove();

} **catch** (SQLException e) {

e.printStackTrace();

}

conn = **null**;

}

}

}

#### service层

这时候service层就不需要向dao层传递connection对象了。

**public** **class** AccountService {

// 转账操作

**public** **boolean** transfer(String outUser, String inUser, **int** money) {

Connection conn = **null**;

AccountDao accountDao = **new** AccountDao();

**try** {

// 1. 获取连接

conn = JDBCUtils.*getConnection*();

// 2. 开启事务

conn.setAutoCommit(**false**);

// 转出

accountDao.out(outUser, money);

**int** i = 10 / 0;

// 转入

accountDao.in(inUser, money);

// 如果转出与转入都执行成功了, 需要提交事务

JDBCUtils.*commitAndCloseQuietly*(conn);

} **catch** (Exception e) {

e.printStackTrace();

// 一旦发生异常, 表示转账失败了.

// 需要回滚事务, 并返回 false

JDBCUtils.*rollbackAndCloseQuietly*(conn);

**return** **false**;

}

**return** **true**;

}

}

#### dao层

而dao层又回到了直接通过jdbcUtils工具类去获取连接对象。这时候是能够保证获取的connection对象和业务层是同一个。

**public** **class** AccountDao {

// 转出

**public** **void** out(String outUser, **int** money) **throws** SQLException {

Connection conn = JDBCUtils.*getConnection*();

QueryRunner queryRunner = **new** QueryRunner();

String sql = "update account set money = money - ? where name = ?;";

Object[] params = {money, outUser};

queryRunner.update(conn, sql, params);

}

// 转入

**public** **void** in(String inUser, **int** money) **throws** SQLException {

Connection conn = JDBCUtils.*getConnection*();

QueryRunner queryRunner = **new** QueryRunner();

String sql = "update account set money = money + ? where name = ?;";

Object[] params = {money, inUser};

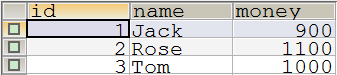
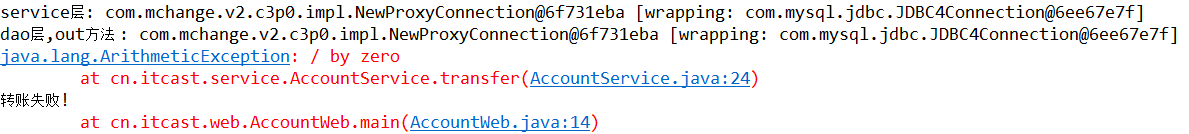
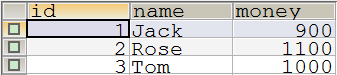
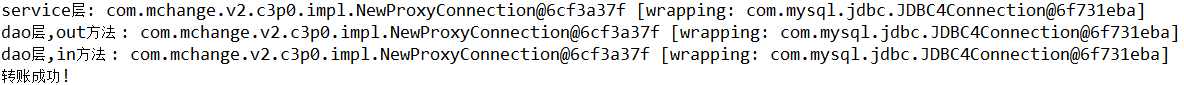
queryRunner.update(conn, sql, params);

}

}

**测试连接对象：**

测试发现，当前线程中3次获取的connection对象都是同一个。



## 案例总结：事务的特性 ACID (理解)

数据库的事务必须具备ACID特性，ACID是指 Atomic（原子性）、Consistensy（一致性）、Isolation（隔离型）和Durability（持久性）的英文缩写。

1. **原子性（Atomicity）**

**事务包装的一组sql，要么都执行成功，要么都失败。这些操作是不可分割的。**

1. **一致性（Consistency）**

**数据库的数据状态是一致的。**

**事务的成功与失败，最终数据库的数据都是符合实际生活的业务逻辑。一致性绝大多数依赖业务逻辑和原子性。**

1. **持久性：（**Durability**）**

**事务成功提交之后，对于数据库的改变是永久的。哪怕数据库发生异常，重启之后数据亦然存在。**

1. **隔离性（Isolation）**

**一个事务的成功或者失败对于其他的事务是没有影响。2个事务应该相互独立。**

### 事务的隔离级别

如果不考虑事务的隔离性，由于事务的并发，将会出现以下问题：

1、脏读 -- 最严重，杜绝发生

2、不可重复读

3、幻读（虚读）

* 查询数据库的隔离级别

select @@tx\_isolation;



* 设置数据库的隔离级别
  + set session transaction isolation level 级别字符串
    - 级别字符串：read uncommitted、read committed、repeatable read、serializable
  + 例如：set session transaction isolation level read uncommitted;

### 隔离级别：解决问题

* 数据库规范规定了4种隔离级别，分别用于描述两个事务并发的所有情况。

1. read uncommitted 读未提交，一个事务读到另一个事务没有提交的数据。
   1. 存在：3个问题（脏读、不可重复读、虚读）。
   2. 解决：0个问题
2. read committed 读已提交，一个事务读到另一个事务已经提交的数据。
   1. 存在：2个问题（不可重复读、虚读）。
   2. 解决：1个问题（脏读）
3. repeatable read（）: 可重复读，在一个事务中读到的数据始终保持一致，无论另一个事务是否提交。
   1. 存在：1个问题（虚读）。
   2. 解决：2个问题（脏读、不可重复读）
4. serializable 串行化，同时只能执行一个事务，相当于事务中的单线程。
   1. 存在：0个问题。
   2. 解决：3个问题（脏读、不可重复读、虚读）

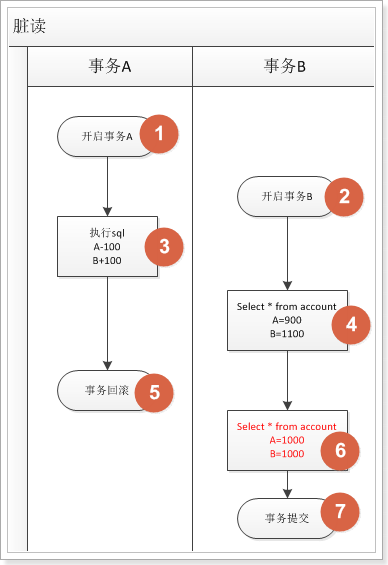
* 安全和性能对比
  + 安全性：serializable > repeatable read > read committed > read uncommitted
  + 性能 ： serializable < repeatable read < read committed < read uncommitted
* 常见数据库的默认隔离级别：
  + MySql：repeatable read
  + Oracle：read committed

在mysql数据库中，底层对于幻读做了优化，演示不了。

#### 脏读：指一个事务读取了另外一个事务 未提交的数据。

**脏读：指一个事务读取了另外一个事务 未提交的数据。**

set session transaction isolation level read uncommitted;



假设A向B转账100元，对应的sql语句如下：

开启事务  
 update account set money=money-100 where name='a';  
 update account set money=money+100 where nam='b';

两条sql语句执行完，b查询自己的账户多了100元。

b走后,a将事务进行回滚，这样B就损失了100元。

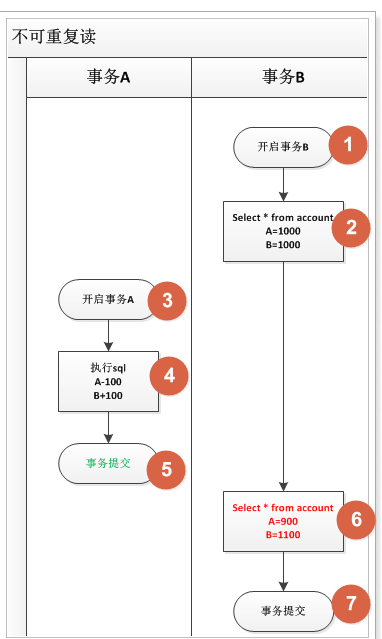
一个事务读取了另一个事务没有提交的数据，非常严重。应当尽量避免脏读。

#### 不可重复读：在一个事务内多次读取表中的数据，多次读取的结果不同。

**不可重复读：在一个事务内多次读取表中的数据，多次读取的结果不同。**

和脏读的区别： 不可重复读是读取的已提交数据

set session transaction isolation level read committed;



* 例如: 银行想查询A账户的余额，第一次查询的结果是200元，A向账户中又存了100元。此时，银行再次查询的结果变成了300元。两次查询的结果不一致，银行就会很困惑，以哪次为准。
* 和脏读不同的是：脏读读取的是前一事务未提交的数据，不可重复度 读取的是前一事务已提交的事务。
* 很多人认为这有啥好困惑的，肯定是以后面的结果为准了。我们需要考虑这样一种情况，查询A账户的余额，一个打印到控制台，一个输出到硬盘上，同一个事务中只是顺序不同，两次查询结果不一致，到底以哪个为准，你会不会困惑呢？

当前事务查询A账户的余额为100元，另外一个事务更新余额为300元并提交，导致当前事务使用同一查询结果却变成了300元。

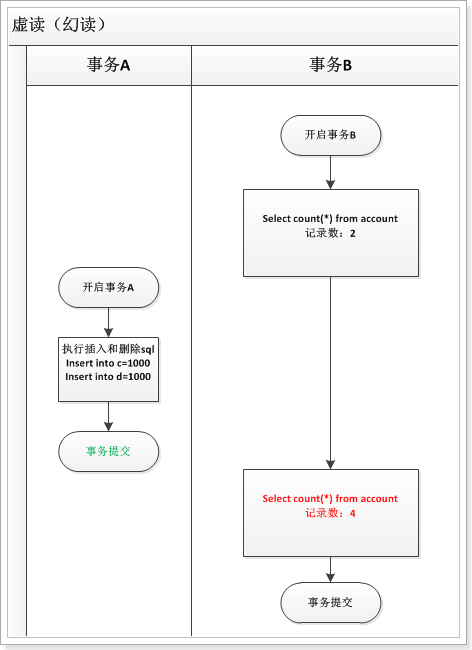
#### 幻读（虚读）

虚读和不可重复读的区别：

虚读 强调的是数据表 记录数 的变化，主要是 insert 和 delete 语句。

不可重复读 强调的是数据表 内容 的变化，主要是 update 语句。

set session transaction isolation level repeatable read;



* 指在一个事务中 读取 另一个事务 插入或删除 数据记录，导致当前事务 读取数据前后不一致。
* 丙 存款100元但未提交，这时银行做报表 统计总额为500元，丙将事务提交，银行再统计就变成了 600元，两次统计结果不一致，银行便会不知所措。

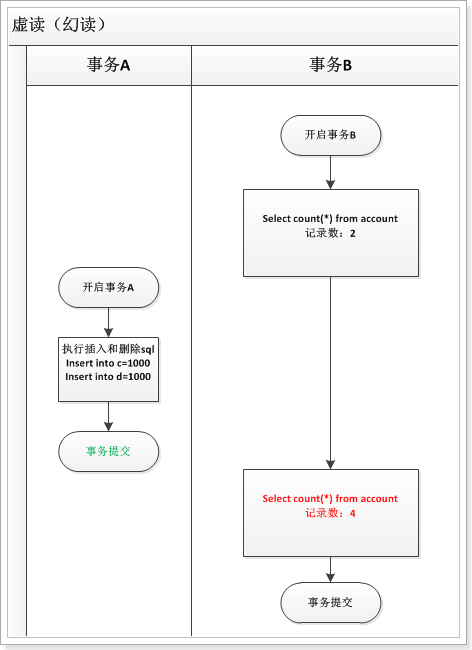
一个事务 读取 另一个事务 已经提交的数据，强调的是 记录数 的变化，常有sql类型为 insert和 delete。

说明 : 在mysql数据库中，底层对于幻读做了优化，演示不了。

#### serializable 串行化

**可以避免所有的问题。数据库执行这个事务，其他事务必须等待当前事务执行完毕，才能被执行。**

set session transaction isolation level serializable;



思考题：为什么串行可以解决所有的问题？

上述所有的问题都是 事务 并行执行引起的，所以改成所有事务依次执行（串行），当然所有问题就都解决了。