**每日作业卷答案**

**事务的操作**

# 关卡1

## 训练案例：使用MySQL命令实现转账的操作

### 训练描述

有如一下表，在MySQL中通过命令。实现银行转账的功能：Jack向Rose转200元钱的SQL语句

1) 创建账户表account(id,name,balance) -- 账号，名字，余额

create table account(

id int primary key auto\_increment,

name varchar(20),

balance double

);

2) 插入2条记录：Jack和Rose的余额都是500

insert into account values (null,'Jack',500);

insert into account values (null,'Rose',500);

3) 两条更新语句：一个转账的业务操作最少要执行下面的2条语句

update account set balance = balance - 200 where name='Jack';

update account set balance = balance + 200 where name='Rose';

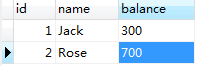
### 训练的知识点

1. mysql中与事务操作的命令
2. 事务结束的情况

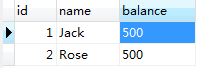
### 操作步骤描述

1. 将两个人的账户都设置成500
2. 分别使用两种不同的方式开启mysql的事务
3. 分别对Jack和Rose进行扣钱和加钱的操作
4. 分别使用提交和回滚操作查看运行的结果

提交



回滚



## 训练案例：使用JDBC实现转账的事务操作

### 训练描述

使用JDBC实现银行转账的功能：Jack向Rose转200元。如果运行过程中没有异常，则事务提交。如果运行过程中出现异常，则进行回滚。

### 训练的知识点

1. JDBC中事务操作有关的方法
2. JDBC中事务的提交与回滚

|  |  |
| --- | --- |
| Connection对象的方法 | 描述 |
| setAutoCommit(false) | 开启事务 |
| commit() | 提交事务 |
| rollback() | 回滚事务 |

### 操作步骤描述

使用事务的JDBC版本，分别查看转账成功和转账失败的情况

1) 得到连接对象以后，将连接对象的自动提交关闭。

2) executeUpdate()以后，没有异常，则提交

3) 出现异常，则回滚。

**package** com.igeek;

**import** java.sql.Connection;

**import** java.sql.PreparedStatement;

**import** org.junit.Test;

**public** **class** JdbcTransactionDemo1 {

@Test

**public** **void** demo01() **throws** Exception {

Connection conn = **null**;

PreparedStatement psmt = **null**;

**try** {

conn = C3P0Utils.*getConnection*();

conn.setAutoCommit(**false**);

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

psmt = conn.prepareStatement(sql);

psmt.setInt(1, -100);

psmt.setString(2, "Jack");

psmt.executeUpdate();

//int i = 1 / 0;

psmt.setInt(1, 100);

psmt.setString(2, "Rose");

psmt.executeUpdate();

conn.commit();

} **catch** (Exception e) {

conn.rollback();

**throw** **new** RuntimeException("程序回滚", e);

} **finally** {

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

conn.close();

}

}

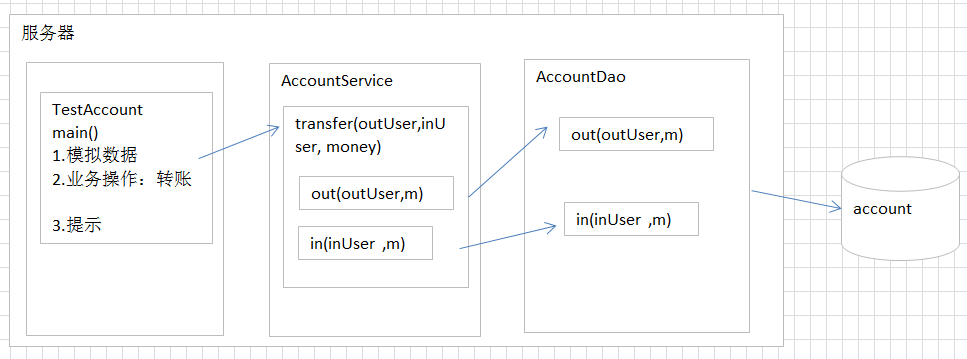
}

}

# 关卡2

## 训练案例：使用三层结构和DbUtils实现转账的功能

### 训练描述

使用三层结构和DbUtils实现用户转账的功能，要求结构如下图所示。Connection对象要求放在ThreadLocal中，以降低代码的耦合度。

### 训练的知识点

1. 三层结构的实现
2. DbUtils中有关事务的操作的方法
3. ThreadLocal类的使用

### 操作步骤描述

1. 表示层：TransferSystem用户的输入输出，调用业务层。

2. 业务层：AccountService方法 transfer(String outUser, String inUser, double money)

3. DAO层：

* + - outMoney(String user, double money) 转出
    - inMoney(String user, double money) 转入
    - update(String user, double money) 更新数据库

1. 工具类：不属于任何一层，案例中的工具类C3P0Util使用从C3P0的连接池中得到连接对象

**public** **class** TransferSystem {

@Test

**public** **void** Test01(){

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

**try** {

accountService.transfer("Jack", "Rose", 200);

System.***out***.println("转账成功");

} **catch** (Exception e) {

// **TODO**: handle exception

System.***out***.println("转账失败");

}

}

}

**public** **class** JdbcUtils {

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

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

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

**try** {

Connection conn = *local*.get();

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

conn = *dataSource*.getConnection();

*local*.set(conn);

}

**return** conn;

} **catch** (Exception e) {

**throw** **new** RuntimeException(e);

}

}

}

**package** com.igeek.dao;

**import** org.apache.commons.dbutils.QueryRunner;

**import** com.igeek.C3P0Utils;

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

**public** **void** outMoney(String outUser,**double** money) **throws** Exception{

QueryRunner qr = **new** QueryRunner();

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

Object[] params = {money,outUser};

qr.update(C3P0Utils.*getConnection*(),sql, params);

}

**public** **void** inMoney(String inUser,**double** money) **throws** Exception{

QueryRunner qr = **new** QueryRunner();

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

Object[] params = {money,inUser};

qr.update(C3P0Utils.*getConnection*(),sql, params);

}

}

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

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

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

Connection conn = **null**;

**try** {

conn = C3P0Utils.*getConnection*();

conn.setAutoCommit(**false**);

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

accountDao.outMoney(outUser, money);

//int n = 1 / 0;

accountDao.inMoney(inUser, money);

DbUtils.*commitAndCloseQuietly*(conn);

} **catch** (Exception e) {

DbUtils.*rollbackAndCloseQuietly*(conn);

**throw** **new** RuntimeException(e);

}

}

}

# 关卡3

## 训练案例：事务的隔离级别

### 训练描述

通过案例的练习，熟悉mysql中事务的隔离级别，以及每种隔离级别解决的问题

### 训练的知识点

1. 了解事务的隔离级别有哪些
2. 不同的事务隔离级别可以解决哪些问题

### 操作步骤描述

1. 一个事务读到另一个事务未提交的数据。

read uncommitted 读未提交，一个事务读到另一个事务没有提交的数据。

存放：3个问题（脏读、不可重复读、虚读）。

解决：0个问题

在该隔离级别，所有事务都可以看到其他未提交事务的执行结果。本隔离级别很少用于实际应用，因为它的性能也不比其他级别好多少。读取未提交的数据，也被称之为脏读（Dirty Read）。

1. 在同一个事务中的多次查询应该出现相同的结果，两次读取不能出现不同的结果。
2. 这是MySQL的默认事务隔离级别，它确保同一事务的多个实例在并发读取数据时，会看到同样的数据行。不过理论上，这会导致另一个棘手的问题：幻读（Phantom Read）。简单的说，幻读指当用户读取某一范围的数据行时，另一个事务又在该范围内插入了新行，当用户再读取该范围的数据行时，会发现有新的“幻影” 行。InnoDB和Falcon存储引擎通过多版本并发控制（MVCC，Multiversion Concurrency Control）机制解决了该问题。