**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

### 答案：

1. update account set balance=500; 将两个人的账户都设置成500。

把set autocommit = 0，相当于关闭了自动提交。如果没有主动commit提交之前，后面的操作都可以回滚。

执行 UPDATE account SET BALANCE=BALANCE-200 WHERE NAME='Jack';

虽然这时候可以查询到好像已经发生了变化，但此时事务并没有结束，使用rollback，即可进行回滚。

2) 将set autocommit = 1，打开自动提交。

执行 UPDATE account SET BALANCE=BALANCE-200 WHERE NAME='Jack';

此时的事务已经提交，使用rollback，无法进行回滚。

3) update account set balance=500; 将两个人的账户都设置成500。

start transaction; -- 主动开启事务

执行 UPDATE account SET BALANCE=BALANCE-200 WHERE NAME='Jack';

虽然这时候可以查询到好像更新成功，但事务没有提交之前，可使用rollback进行回滚。

4) start transaction; -- 主动开启事务

执行 UPDATE account SET BALANCE=BALANCE-200 WHERE NAME='Jack';

使用commit以后，事务已经提交，不能进行回滚

## 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。

### 答案：

1. 工具类：

package com.igeek.util;

import java.sql.Connection;

import java.sql.ResultSet;

import java.sql.SQLException;

import java.sql.Statement;

import javax.sql.DataSource;

import com.mchange.v2.c3p0.ComboPooledDataSource;

public class C3P0Util {

// 创建一个私有的连接池

private static ComboPooledDataSource ds = new ComboPooledDataSource();

/\*\*

\* 得到连接池

\*/

public static DataSource getDataSource() {

return ds;

}

/\*\*

\* 得到连接对象

\*/

public static Connection getConnection() {

try {

return ds.getConnection();

} catch (SQLException e) {

e.printStackTrace();

throw new RuntimeException(e);

}

}

/\*\*

\* 关闭连接

\*/

public static void close(Connection conn, Statement ps, ResultSet rs) {

if (rs != null) {

try {

rs.close();

} catch (SQLException e) {

e.printStackTrace();

}

}

if (ps != null) {

try {

ps.close();

} catch (SQLException e) {

e.printStackTrace();

}

}

if (conn != null) {

try {

conn.close();

} catch (SQLException e) {

e.printStackTrace();

}

}

}

public static void close(Connection conn, Statement ps) {

close(conn, ps, null);

}

}

1. JDBC实现事务的代码：

/\*\*

\* 在Java中使用事务实现转账的功能

\* @author NewBoy

\*/

public class JdbcDemo{

public static void main(String[] args) {

Connection conn = null;

PreparedStatement ps = null;

try {

conn = C3P0Util.getConnection();

//关闭事务的自动提交

conn.setAutoCommit(false);

ps = conn.prepareStatement("update account set balance = balance - 200 where name='Jack'");

//更新

ps.executeUpdate();

//引发异常

//System.out.println(5 / 0);

ps = conn.prepareStatement("update account set balance = balance + 200 where name='Rose'");

ps.executeUpdate();

//手动提交事务

conn.commit();

System.out.println("转账成功");

} catch (Exception e) {

e.printStackTrace();

//事务回滚

try {

conn.rollback();

} catch (SQLException e1) {

e1.printStackTrace();

}

System.out.println("转账失败");

} finally {

C3P0Util.close(conn, ps);

}

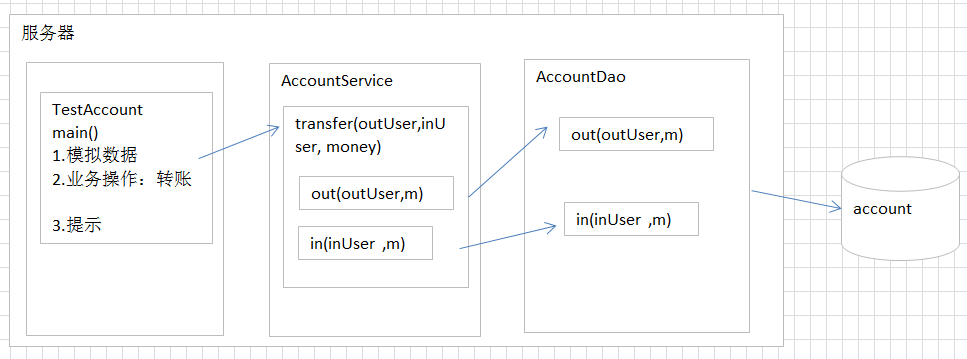
}

}

# 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

4. 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

### 答案：

1. 工具类C3P0Utils

public class C3P0Util {

// 创建一个连接池对象

private static ComboPooledDataSource ds = new ComboPooledDataSource();

// 给当前线程绑定 连接

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

/\*\*

\* 获得连接

\* @return

\*/

public static Connection getConnection(){

try {

//#1从当前线程中， 获得已经绑定的连接

Connection conn = local.get();

if(conn == null){

//#2 第一次获得，绑定内容 – 从连接池获得

conn = ds.getConnection();

//#3 将连接存 ThreadLocal

local.set(conn);

}

return conn; //获得连接

} catch (SQLException e) {

//将编译时异常 转换 运行时 ， 以后开发中 运行时异常使用比较多的。

throw new RuntimeException(e);

}

}

}

2. 业务层的转账方法

|  |
| --- |
| /\*\*  \* 业务层  \* @author NewBoy  \*/  public class AccountService {  /\*\*  \* 业务层转账的方法：  \* @param from :付款人  \* @param to :收款人  \* @param money :转账金额  \*/  public void transfer(String from, String to, double money) {  // 调用DAO：  AccountDao accountDao = new AccountDao();  //得到连接对象  Connection conn = C3P0Util.getConnection();  try {  //开启事务  conn.setAutoCommit(false);  accountDao.outMoney(from, money);  //假设这里出现异常，则应该回滚事务，并且转账失败  System.out.println(100/0);  accountDao.inMoney(to, money);  //没有异常提交事务  conn.commit();  } catch (SQLException e) {  e.printStackTrace();  //出现异常回滚事务  try {  conn.rollback();  } catch (SQLException se) {  se.printStackTrace();  }  } finally {  //在业务层关闭连接  C3P0Util.close(conn, null);  }  }  } |

3. DAO层

|  |
| --- |
| /\*\*  \* 持久层  \* @author NewBoy  \*/  public class AccountDao {  /\*\*  \* 付款的方法  \*/  public void outMoney(String name, double money) throws SQLException {  String sql = "update account set balance = balance-? where name=?";  // 调用方法收款  updateAccount(sql, name, money);  }  /\*\*  \* 收款的方法  \*/  public void inMoney(String name, double money) throws SQLException {  String sql = "update account set balance = balance+? where name=?";  // 调用方法收款  updateAccount(sql, name, money);  }  /\*\*  \* 更新账户信息  \*/  public void updateAccount(String sql, String name, double money)  throws SQLException {  Connection conn = C3P0Util.getConnection();  // 不再获得连接对象，而是使用参数传递进来的连接对象  PreparedStatement pstmt = null;  try {  // 预编译SQL:  pstmt = conn.prepareStatement(sql);  // 设置参数:  pstmt.setDouble(1, money);  pstmt.setString(2, name);  // 执行SQL：  pstmt.executeUpdate();  } finally {  // 异常抛出，而且不能关闭连接  C3P0Util.close(null, pstmt);  }  }  } |

4. 表示层：

/\*\*

\* 转账系统的表示层

\* @author NewBoy

\*/

public class TransferSystem {

public static void main(String[] args) {

try (Scanner scanner = new Scanner(System.in);) {

System.out.println("-- 欢迎进入银行的转账系统 -- ");

System.out.println("请输入付款人的名字：");

String outUser = scanner.next();

System.out.println("请输入付款的金额：");

double money = scanner.nextDouble();

System.out.println("请输入收款人的名字：");

String inUser = scanner.next();

// 2 转账

AccountService accountService = new AccountService();

accountService.transfer(outUser, inUser, money);

// 3 提示

System.out.println("转账" + money + "成功");

} catch (Exception e) {

System.out.println("转账失败");

}

}

}

# 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。
2. 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。

### 答案：

1. 脏读的操作：

1) 打开命令行Jack，设置全局的隔离级别为最低：

set global transaction isolation level read uncommitted;

2) 开启事务

use db;

start transaction;

3) 更新2个人的账户，未提交

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

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

4) 打开另一个命令行Rose，查询账户

use db;

select \* from account;

发现钱已经到账，发货

5) 命令行Jack回滚

rollback;

6) 命令行Rose，查询账户，钱没了。

2. 解决脏读的问题：将全局的隔离级别进行提升

1) 打开命令行Jack，设置全局的隔离级别为read committed：

set global transaction isolation level read committed;

2) 重复上面的操作：需要重新登录。

会发现命令行Jack在没有提交和回滚之前，命令行Rose看不到账户发生任何变化。

3) 命令行Jack，使用commit提交以后

命令行Rose，可以看到账户发生了变化。

4) 结论：read committed的方式可以避免脏读的发生。

3. 出现不可重复读的问题

1). 将数据进行恢复，并关闭窗口重新登录。

update account set balance=500;

2) 开启1个银行账户窗口

select @@global.tx\_isolation; -- 确保当前的事务隔离是read committed

3) 在银行窗口中开启一个事务

use db;

start transaction;

查询用户Jack的账户，输出到屏幕。

select \* from account where name='Jack'; -- 查到是500块

4) 开启1个Jack账户窗口

-- 开启一个事务：

use db;

start transaction

更新账户，加100元

update account set balance=balance+100 where name='Jack';

提交事务

commit;

5) 银行窗口在同一个事务中再次查询Jack的账户，输出到文件中：

select \* from account where name='Jack'; -- 查到是600块

commit; -- 事务结束

两次查询输出的结果不同，到底哪次是对的？

4. 解决：不可重复读的问题

1). 将数据进行恢复

update account set balance=500;

设置隔离级别为repeatable read

set global transaction isolation level **repeatable read;**

select @@global.tx\_isolation; -- 查询隔离级别

2) 重新登录

银行窗口登录

use db;

start transaction;

select \* from account where name='Jack';-- 查询到500

3) Jack窗口登录

use db;

start transaction;

select \* from account where name='Jack';

update account set balance=balance+100 where name='Jack';

commit; -- 提交事务

select \* from account where name='Jack'; -- 这时Jack看到的是600

4) 银行窗口再查

select \* from account where name='Jack'; -- 查询还是500，很好的体现了事务的隔离性。

commit;-- 2次查询结果相同

5) 银行提交当前事务

commit;

再开启一个新的事务

start transaction;

select \* from account where name='Jack'; -- 查询变成了600

commit;

6) 结论：为了保存多次查询数据一致，必须使用repeatable read隔离级别