# JDBC高级开发事务

## 事务管理

### 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.

### mysql事务操作

|  |  |
| --- | --- |
| sql语句 | 描述 |
| start transaction; | 开启事务 |
| commit; | 提交事务 |
| rollback; | 回滚事务 |

* 准备数据

# 创建一个表：账户表.

create database webdb;

# 使用数据库

use webdb;

# 创建账号表

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',10000);

insert into account values (null,'rose',10000);

insert into account values (null,'tom',10000);

* 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 |
| conn.commit() | 提交事务 |
| conn.rollback() | 回滚事务 |

//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();

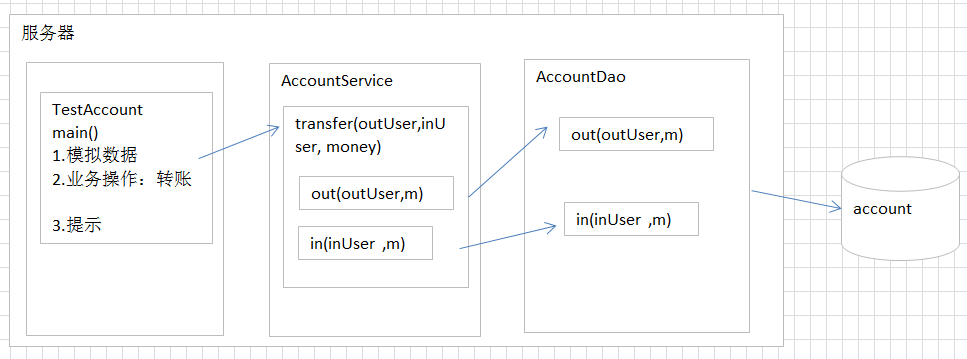
}

}

### 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.commitAndClose(conn) 或  DbUtils.rollbackAndClose(conn) | 提交并关闭连接  回滚并关闭连接 |

## 案例分析



* 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：编写入口程序

public static void main(String[] args) {

try {

String outUser = "jack";

String inUser = "rose";

Integer money = 100;

//2 转账

AccountService accountService = new AccountService();

accountService.transfer(outUser, inUser, money);

//3 提示

System.out.println("转账成功");

} catch (Exception e) {

System.out.println("转账失败");

}

}

* 步骤3：编写AccountService

public class AccountService {

/\*\*

\* 业务层转账的方法：

\* @param outUser :付款人

\* @param inUser :收款人

\* @param money :转账金额

\*/

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

// 调用DAO：

AccountDao accountDao = new AccountDao();

try {

accountDao.outMoney(outUser, money);

// int d = 1/0;

accountDao.inMoney(inUser, money);

} catch (SQLException e) {

e.printStackTrace();

}

}

}

步骤4：编写AccountDao

public class AccountDao {

/\*\*

\* 付款的方法

\* @param name

\* @param money

\* @throws SQLException

\*/

public void outMoney(String name,int money) throws SQLException{

Connection conn = null;

PreparedStatement pstmt = null;

try{

// 获得连接:

conn = JDBCUtils.getConnection();

// 编写一个SQL:

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

// 预编译SQL:

pstmt = conn.prepareStatement(sql);

// 设置参数:

pstmt.setInt(1, money);

pstmt.setString(2, name);

// 执行SQL：

pstmt.executeUpdate();

}catch(Exception e){

e.printStackTrace();

}finally{

pstmt.close();

conn.close();

}

}

/\*\*

\* 收款的方法

\* @param name

\* @param money

\* @throws SQLException

\*/

public void inMoney(String name,int money) throws SQLException{

Connection conn = null;

PreparedStatement pstmt = null;

try{

// 获得连接:

conn = JDBCUtils.getConnection();

// 编写一个SQL:

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

// 预编译SQL:

pstmt = conn.prepareStatement(sql);

// 设置参数:

pstmt.setInt(1, money);

pstmt.setString(2, name);

// 执行SQL：

pstmt.executeUpdate();

}catch(Exception e){

e.printStackTrace();

}finally{

pstmt.close();

conn.close();

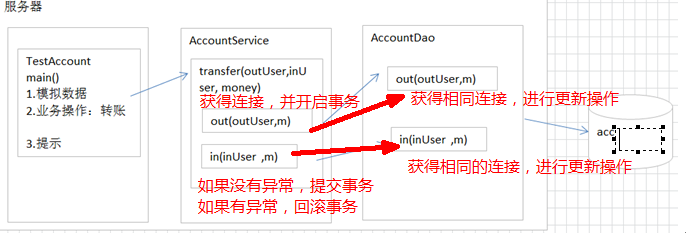
}

}

}

## 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



### service层

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

Connection conn =null;

try{

//1 获得连接

conn = JdbcUtils.getConnection();

//2 开启事务

conn.setAutoCommit(false);

accountDao.outMoney(conn,outUser, money);

//断电

//int i = 1 / 0;

accountDao.inMoney(conn,inUser, money);

//3 提交事务

conn.commit();

} catch (Exception e) {

try {

//回顾

if (conn != null) {

conn.rollback();

}

} catch (Exception e2) {

}

throw new RuntimeException(e);

} finally{

JdbcUtils.closeResource(conn, null, null);

}

}

### dao层

/\*\*

\* 汇款

\* @param outUser 汇款人

\* @param money -

\*/

public void outMoney(Connection conn, String outUser , int money){

//Connection conn = null;

PreparedStatement psmt = null;

ResultSet rs = null;

try {

//1 获得连接

//conn = JdbcUtils.getConnection();

//2 准备sql语句

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

//3预处理

psmt = conn.prepareStatement(sql);

//4设置实际参数

psmt.setInt(1, money);

psmt.setString(2, outUser);

//5执行

int r = psmt.executeUpdate();

System.out.println(r);

} catch (Exception e) {

throw new RuntimeException(e);

} finally{

//6释放资源

JdbcUtils.closeResource(null, psmt, rs);

}

}

/\*\*

\* 收款

\* @param inUser 收款人

\* @param money +

\*/

public void inMoney(Connection conn,String inUser , int money){

//Connection conn = null;

PreparedStatement psmt = null;

ResultSet rs = null;

try {

//1 获得连接

//conn = JdbcUtils.getConnection();

//2 准备sql语句

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

//3预处理

psmt = conn.prepareStatement(sql);

//4设置实际参数

psmt.setInt(1, money);

psmt.setString(2, inUser);

//5执行

int r = psmt.executeUpdate();

System.out.println(r);

} catch (Exception e) {

throw new RuntimeException(e);

} finally{

//6释放资源

JdbcUtils.closeResource(null, psmt, rs);

}

}

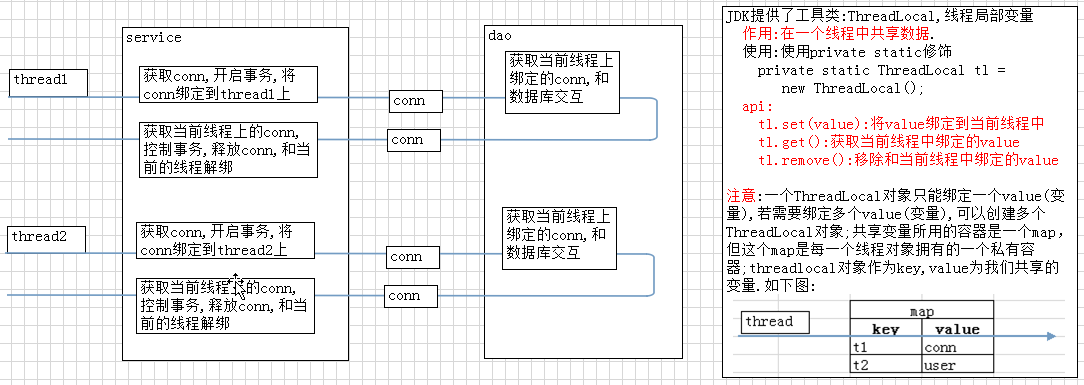
## 提高：ThreadLocal

### 案例介绍

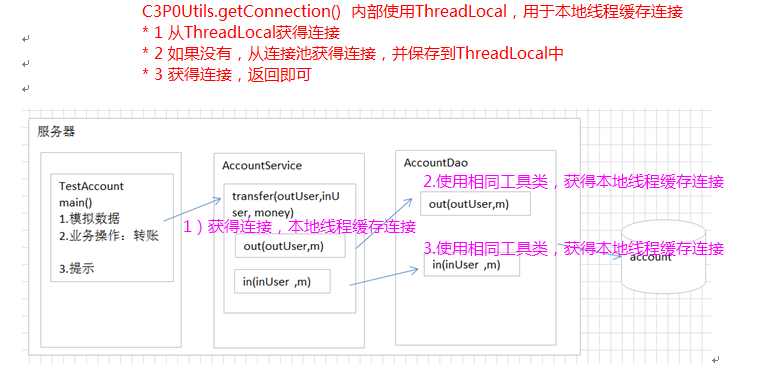
在“事务传递参数版”中，我们必须修改方法的参数个数，传递链接，才可以完成整个事务操作。如果不传递参数，是否可以完成？在JDK中给我们提供了一个工具类：ThreadLocal，此类可以在一个线程中共享数据。

### 相关知识：ThreadLocal

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。



### 分析



### 实现

#### 工具类JDBCUtils

//连接池

private static ComboPooledDataSource dataSource = new ComboPooledDataSource("itcast");

//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 = dataSource.getConnection();

//#3 将连接存 ThreadLocal

local.set(conn);

}

return conn; //获得连接

} catch (Exception 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。

// \* 此处可以编写自定义异常。

throw new RuntimeException(e);

// \* 类与类之间 进行数据交换时，可以使用return返回值。也可以自定义异常返回值，调用者try{} catch(e){ e.getMessage() 获得需要的数据}

//throw new MyConnectionException(e);

}

}

#### service层

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

Connection conn =null;

try{

//1 获得连接

conn = JdbcUtils.getConnection();

//2 开启事务

conn.setAutoCommit(false);

accountDao.out(outUser, money);

//断电

//int i = 1 / 0;

accountDao.in(inUser, money);

//3 提交事务

conn.commit();

} catch (Exception e) {

try {

//回顾

if (conn != null) {

conn.rollback();

}

} catch (Exception e2) {

}

throw new RuntimeException(e);

} finally{

JdbcUtils.closeResource(conn, null, null);

}

}

#### dao层

/\*\*

\* 汇款

\* @param outUser 汇款人

\* @param money -

\*/

public void out(String outUser , int money){

Connection conn = null;

PreparedStatement psmt = null;

ResultSet rs = null;

try {

//1 获得连接

conn = JdbcUtils.getConnection();

//2 准备sql语句

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

//3预处理

psmt = conn.prepareStatement(sql);

//4设置实际参数

psmt.setInt(1, money);

psmt.setString(2, outUser);

//5执行

int r = psmt.executeUpdate();

System.out.println(r);

} catch (Exception e) {

throw new RuntimeException(e);

} finally{

//6释放资源--不能关闭连接

JdbcUtils.closeResource(null, psmt, rs);

}

}

/\*\*

\* 收款

\* @param inUser 收款人

\* @param money +

\*/

public void in(String inUser , int money){

Connection conn = null;

PreparedStatement psmt = null;

ResultSet rs = null;

try {

//1 获得连接

conn = JdbcUtils.getConnection();

//2 准备sql语句

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

//3预处理

psmt = conn.prepareStatement(sql);

//4设置实际参数

psmt.setInt(1, money);

psmt.setString(2, inUser);

//5执行

int r = psmt.executeUpdate();

System.out.println(r);

} catch (Exception e) {

throw new RuntimeException(e);

} finally{

//6释放资源--注意：不能关闭链接

JdbcUtils.closeResource(null, psmt, rs);

}

}

## 案例总结：事务总结

### 事务特性：ACID

* 原子性（Atomicity）原子性是指事务是一个不可分割的工作单位，事务中的操作要么都发生，要么都不发生。
* 一致性（Consistency）事务前后数据的完整性必须保持一致。
* 隔离性（Isolation）事务的隔离性是指多个用户并发访问数据库时，一个用户的事务不能被其它用户的事务所干扰，多个并发事务之间数据要相互隔离。
* 持久性（Durability）持久性是指一个事务一旦被提交，它对数据库中数据的改变就是永久性的，接下来即使数据库发生故障也不应该对其有任何影响。

### 并发访问问题

如果不考虑隔离性，事务存在3中并发访问问题。

1. 脏读：一个事务读到了另一个事务未提交的数据.
2. 不可重复读：一个事务读到了另一个事务已经提交(update)的数据。引发另一个事务，在事务中的多次查询结果不一致。
3. 虚读 /幻读：一个事务读到了另一个事务已经提交(insert)的数据。导致另一个事务，在事务中多次查询的结果不一致。

### 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

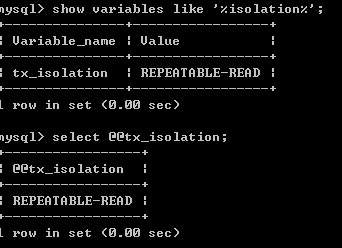
### 演示

* 隔离级别演示参考：资料/隔离级别操作过程.doc【增强内容,了解】
* 查询数据库的隔离级别

show variables like '%isolation%';

或

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;
* 读未提交：read uncommitted
  + A窗口设置隔离级别
  + AB同时开始事务
  + A 查询
  + B 更新，但不提交
  + A 再查询？-- 查询到了未提交的数据
  + B 回滚
  + A 再查询？-- 查询到事务开始前数据
* 读已提交：read committed
  + A窗口设置隔离级别
  + AB同时开启事务
  + A查询
  + B更新、但不提交
  + A再查询？--数据不变，解决问题【脏读】
  + B提交
  + A再查询？--数据改变，存在问题【不可重复读】
* 可重复读：repeatable read
  + A窗口设置隔离级别
  + AB 同时开启事务
  + A查询
  + B更新， 但不提交
  + A再查询？--数据不变，解决问题【脏读】
  + B提交
  + A再查询？--数据不变，解决问题【不可重复读】
  + A提交或回滚
  + A再查询？--数据改变，另一个事务
* 串行化：serializable
  + A窗口设置隔离级别
  + AB同时开启事务
  + A查询
  + B更新？--等待(如果A没有进一步操作，B将等待超时)
  + A回滚
  + B 窗口？--等待结束，可以进行操作