day06-jdbc事务&ThreadLocal

- [ ] 理解事务的概念
- [ ] 理解脏读,不可重复读,幻读的概念及解决办法
- [ ] 能够在MySQL中使用事务
- [ ] 能够在JDBC中使用事务
- [ ] 能够理解ThreadLocal的作用

第一章 事务操作

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SQL操作要么全都成功,要么全都失败.

1.1 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
```

1.2 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

> 代码演示

```
public class Transaction {

    public static void main(String[] args) {
        Connection conn = null;
        Statement state = null;
        try{
            Class.forName("com.mysql.jdbc.Driver");
            conn =
DriverManager.getConnection("jdbc:mysql://localhost:3306/webdb","root","root");
            //1:开启事务
            conn.setAutoCommit(false);
            //2: 获取语句执行平台
            state = conn.createStatement();
            // jack-1000
            int row1 = state.executeUpdate("update account set money=money-1000 where
name='jack'");
            // rose+1000

            int row2 = state.executeUpdate("update account set money=money+1000 where
```

```
name='rose'");
            //事务提交
            conn.commit();
            System.out.println("事务提交成功，数据库已经修改");
        }catch(Exception ex){

            try {
                //失败回滚
                conn.rollback();
                System.out.println("事务进行回滚，数据库没有更改");
            } catch (SQLException e) {
                e.printStackTrace();
            }
        }finally{
            if(state!=null){
                try {
                    state.close();
                } catch (SQLException e) {
                    e.printStackTrace();
                }
            }
            if(conn!=null){
                try {
                    conn.close();
                } catch (SQLException e) {
                    e.printStackTrace();
                }
            }
        }

    }
}
```

1.3 JDBC事务案例分层（dao、service）

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

- 工具类JDBCUtils

```
public class JDBCUtils {
    // 1. 声明静态数据源成员变量
    private static DataSource ds;

    // 2. 创建连接池对象
    static {
        // 加载配置文件中的数据
        InputStream is = JDBCUtils.class.getResourceAsStream("/druid.properties");
        Properties pp = new Properties();
        try {
            pp.load(is);
            // 创建连接池，使用配置文件中的参数
            ds = DruidDataSourceFactory.createDataSource(pp);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    // 3. 定义公有的得到数据源的方法
    public static DataSource getDataSource() {
        return ds;
    }

    // 4. 定义得到连接对象的方法
    public static Connection getConnection() throws SQLException {
        return ds.getConnection();
    }
}
```

- 步骤1：编写入口程序

```
public class AccountWeb {


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

```
        String outUser = "jack";
        String inUser = "rose";
        Integer money = 100;
        //2 转账
        AccountService accountService = new AccountService();
        accountService.doAccount(outUser, inUser, money);
        //3 提示
        System.out.println("转账成功");
    }
}
```

- service层

```
public class AccountService {

    public void doAccount(String fromName,String toName,int money){
        Connection conn = null;
        AccountDao dao = new AccountDao();

        try{
          //获取连接对象
          conn = JDBCUtils.getConnection();
            // 开启事务
         conn.setAutoCommit(true);
         // 转账操作
            dao.payMoney(fromName,money,conn);
            dao.incomeMoney(toName,money,conn);
            conn.commit();
        }catch(Exception ex){
            try {
                // 失败回滚
                conn.rollback();
                System.out.println("失败");
            } catch (SQLException e) {
                e.printStackTrace();
            }
        }finally{
            try {
                conn.close();
            } catch (SQLException e) {
                e.printStackTrace();
            }
        }
    }
}
```

- dao层

```
public class AccountDao {
```

```
    /**
     * 汇款
     * @param fromName 汇款人
     * @param money  金额
     * @param conn   连接对象
     * @throws SQLException
     */
    public void payMoney(String fromName, int money, Connection conn)
            throws SQLException {
        //预编译sql语句
        String sql = "update account set money = money - ? where name = ? ";
        //预处理对象
        PreparedStatement ps = conn.prepareStatement(sql);
        //设置参数值
        ps.setInt(1,money);
        ps.setString(2,fromName);
        int row = ps.executeUpdate();

        JDBCUtils.close(null,ps,null);
    }

    /**
     * 收款
     * @param toName 收款人
     * @param money  金额
     * @param conn   连接对象
     * @throws SQLException
     */
    public void incomeMoney(String toName, int money, Connection conn)
            throws SQLException {
        //预编译sql语句
        String sql = "update account set money = money + ? where name = ? ";
        //预处理对象
        PreparedStatement ps = conn.prepareStatement(sql);
        //设置参数值
        ps.setInt(1,money);
        ps.setString(2,toName);
        int row = ps.executeUpdate();

        JDBCUtils.close(null,ps,null);
    }
}
```

第二章 ThreadLocal

2.1 分析

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。

`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。

2.2 相关知识：ThreadLocal

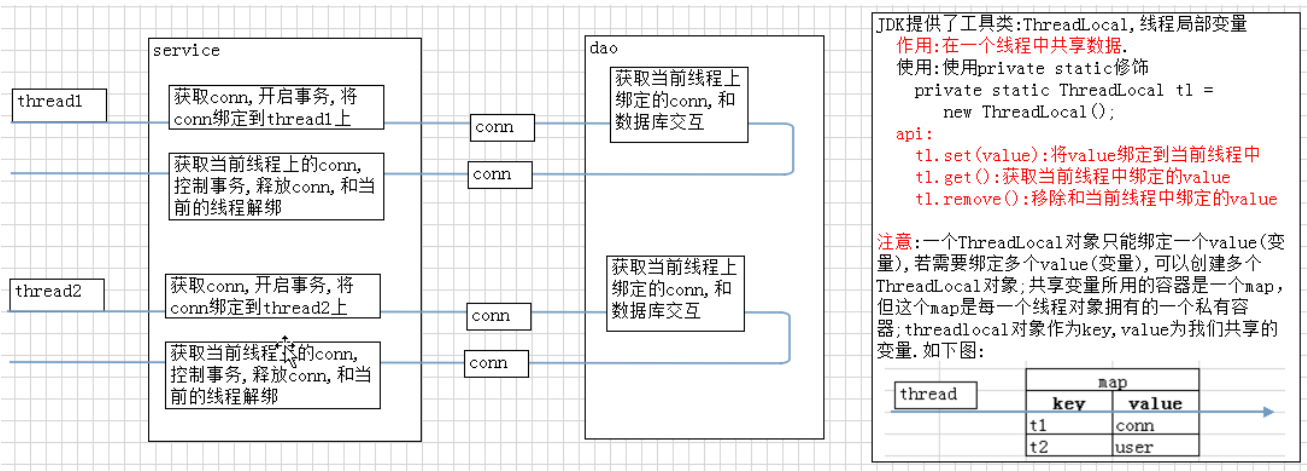
java.lang.ThreadLocal 该类提供了线程局部(thread-local) 变量，用于在当前线程中共享数据。ThreadLocal工具类底层就是相当于一个Map，key存放的当前线程，value存放需要共享的数据。

举例

```
public class ThreadLocalDemo {

    public static void main(String[] args) {
        ThreadLocal<String>  mainThread = new ThreadLocal<>();

        mainThread.set("传智播客");

        System.out.println(mainThread.get());//传智播客

        new Thread(()->{
            System.out.println(mainThread.get());//null
        }).start();
    }
}
```

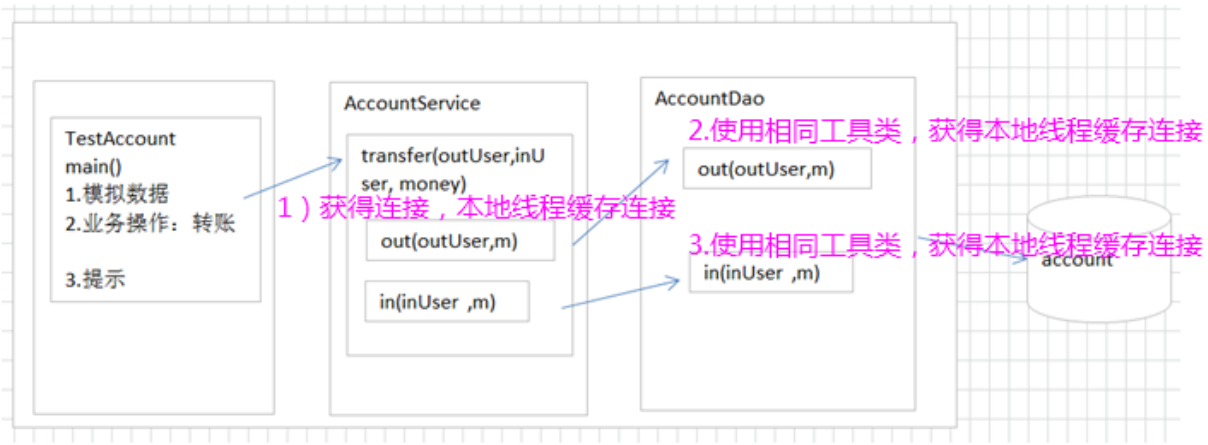
结论：向ThreadLocal对象中添加的数据只能在当前线程下使用。

2.3 结合案例使用



分析

JDBCUtils.getConnection() 内部使用ThreadLocal，用于本地线程缓存连接
* 1 从ThreadLocal获得连接
* 2 如果没有，从连接池获得连接，并保存到ThreadLocal中
* 3 获得连接，返回即可



代码实现

- 工具类ConnectionManager

```
/*
   连接对象管理类
      保证连接对象在一个线程中唯一
      管理 连接对象的获取
      管理 连接对象的事务开启
      管理 连接对象的事务提交
      管理 连接对象的事务回滚
      管理 连接对象的释放
 */
public class ConnectionManager {

    // 创建当前线程 存储 连接对象的ThreadLocal对象
    private static ThreadLocal<Connection> threadLocal = new ThreadLocal<>();
    // 唯一的连接对象
    private static Connection conn = null;


    // 获取唯一的 连接对象
    public static Connection getConnection(){
       //1从当前线程中， 获得已经绑定的连接
       conn = threadLocal.get();

        if(conn==null){//如果为空
            // 第一次获得，绑定内容 - 从连接池获得
            conn = JDBCUtils.getConnection();

            // 存到当前线程中
            threadLocal.set(conn);
        }
```

```
        return conn;
    }
    /*
        开启事务
     */
    public static void begin(){
        try {
            conn.setAutoCommit(false);
        } catch (SQLException e) {
            System.out.println("开启失败");
        }
    }
    /*
       提交事务
     */
    public static void commit(){
        try {
            conn.commit();
        } catch (SQLException e) {
            System.out.println("提交异常");
        }
    }
    /*
      回滚
     */
    public static void rollback(){
        try {
            conn.rollback();
        } catch (SQLException e) {
            System.out.println("回滚异常");
        }
    }
    /*
      释放
     */
    public static void close(){
        try {
            conn.close();
        } catch (SQLException e) {
            System.out.println("释放异常");
        }
    }
}
```

- service层

```
public class AccountService {

    public void doAccount(String fromName,String toName,int money){
        Connection conn = null;
        AccountDao dao = new AccountDao();
```

```
        try{
          //获取连接对象
          conn = ConnectionManager.getConnection();
            // 开启事务
          ConnectionManager.begin();
         // 转账操作
            dao.payMoney(fromName,money);
            dao.incomeMoney(toName,money);
            ConnectionManager.commit();
        }catch(Exception ex){
            try {
                // 失败回滚
                conn.rollback();
                System.out.println("失败");
            } catch (SQLException e) {
                e.printStackTrace();
            }
        }finally{
           ConnectionManager.close();
        }
    }
}
```

- dao层

```
    public class AccountDao {

    /**
     * 汇款
     * @param fromName 汇款人
     * @param money  金额
     * @throws SQLException
     */
    public void payMoney(String fromName, int money)
            throws SQLException {
        //预编译sql语句
        String sql = "update account set money = money - ? where name = ? ";
        System.out.println(1/0);
        //预处理对象
        PreparedStatement ps = ConnectionManager.getConnection().prepareStatement(sql);
        //设置参数值
        ps.setInt(1,money);
        ps.setString(2,fromName);
        int row = ps.executeUpdate();

        JDBCUtils.close(null,ps,null);
    }

    /**
     * 收款
     * @param toName 收款人
     * @param money  金额
```

```
     * @throws SQLException
     */
    public void incomeMoney(String toName, int money)
            throws SQLException {
        //预编译sql语句
        String sql = "update account set money = money + ? where name = ? ";
        //预处理对象
        PreparedStatement ps = ConnectionManager.getConnection().prepareStatement(sql);
        //设置参数值
        ps.setInt(1,money);
        ps.setString(2,toName);
        int row = ps.executeUpdate();

        JDBCUtils.close(null,ps,null);
    }
}
```

第二章 事务总结

2.1 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。

2.2 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。

2.3 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。

   a)存在：3个问题（脏读、不可重复读、虚读）。

   b)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。

   a)存在：2个问题（不可重复读、虚读）。

   b)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。

   a)存在：1个问题（虚读）。

   b)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。

   a)存在：0个问题。

   b)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`

2.4 演示演示

- 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;
```

```
ysql> show variables like '%isolation%';
+---------------+-----------------+
| Variable_name | Value           |
+---------------+-----------------+
| tx_isolation  | REPEATABLE-READ |
+---------------+-----------------+
1 row in set (0.00 sec)

ysql> select @@tx_isolation;
+-----------------+
| @@tx_isolation  |
+-----------------+
| REPEATABLE-READ |
+-----------------+
1 row in set (0.00 sec)
```

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

- 读未提交：readuncommitted
  - 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