# JDBC Day3

场景：通过控制台输入数据，模拟用户登录操作

# 1.三层架构

在软件开发工程，分解复杂的软件系统时，软件工程最常见的设计方式就是分层，将整个系统拆分为N个层次，每一层都有独立职责，多个层协同提供完整的功能，分层的好处：简化设计，各司其职，更容易扩展。在JavaEE项目中，三次架构分为视图层（UI,web层），业务层（service层）,数据访问层（DAO层）。

视图层：提供界面支持，向用户通过界面的方式展示数据，通过界面收集用户提交的数据

业务层：提供核心业务支持

数据库访问层：提供对数据库的操作

## 1.1 分层的设计思路

现有的代码没有进行分层，我们先看一下现有代码的问题：

```java
public class UserLoginTest {
    public static void main(String[] args) {
//        1.获取用户输入的数据
        Scanner sc = new Scanner(System.in);
        System.out.println("请输入用户名");
        String userName = sc.nextLine();
        System.out.println("请输入密码");
        String password = sc.nextLine();

//        2.使用JDBC将数据保存到数据库
        Connection conn = null;
        PreparedStatement pstm = null;
        ResultSet rs = null;
        User user = null;
            try{
                //            2.1 获取链接
                conn = JdbcUtils.getConnection();
                //             2.2 编写sql,并获取执行sql语句对象
                String sql = "select * from t_user where user_name = ? and password = ?";
                pstm = conn.prepareStatement(sql);
                pstm.setString( parameterIndex: 1,userName);
                pstm.setString( parameterIndex: 2,password);
//                2.3  发送sql
                rs = pstm.executeQuery();
//                2.4 处理结果
                while(rs.next()){
                    user = new User();
                    user.setUid(rs.getInt( columnLabel: "uid"));
                    user.setUserName(rs.getString( columnLabel: "user_name"));
                    user.setPassword(rs.getString( columnLabel: "password"));
                }
            }catch (Exception e){
                e.printStackTrace();
            }finally {
//                  2.5 关闭链接
                JdbcUtils.close(conn,pstm,rs);
            }

//        3.根据JDBC返回的结果，判断用户是否登录成功
        if(user != null){
            System.out.println("用户登录成功");
        }else{
            System.out.println("用户登录失败");
        }
    }
}
```

1 收集用户数据，视图层

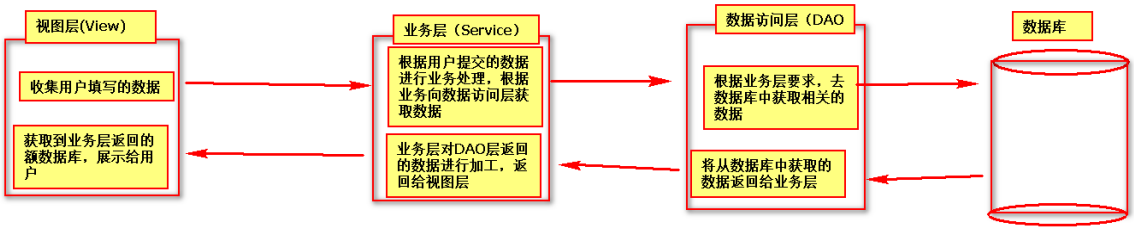
2 数据访问层

3 业务层，

程序的3个部分杂糅在一起，违反了程序设计单一职责原则，不利于程序的后期扩展

解决方案： 分层

分层的思路：



## 1.2 数据访问层

数据访问层(DAO)：用户和数据库直接打交道，提供对数据的增删改查操作。

dao层代码案例：

dao接口

```
/**
 * dao接口中封装了对对应的增删改查操作
 */
public interface PersonDao {
//    添加
    public int insertPerson(Person p);
//    修改
    public int updatePerson(Person p);
//    删除
    public int deletePersonById(int id);
//    根据id进行查询
    public Person selectPersonById(int id);
//    查询所有
    public List<Person> selectAllPerson();
}
```

dao层实现类

```
package com.baizhi.dao.impl;

import com.baizhi.dao.PersonDao;
import com.baizhi.entity.Person;
import com.baizhi.utils.JdbcUtils;

import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.util.ArrayList;
import java.util.List;

/**
 * Dao实现类需要重新dao接口中的所有方法
 * 方法实现基于jdbc操作6步
 */
public class PersonDaoImpl implements PersonDao {
    @Override
    public int insertPerson(Person p) {
        Connection conn = null;
        PreparedStatement pstm = null;
        int i = 0;
        try{
            //        1.获取链接
            conn = JdbcUtils.getConnection();
//          2.编写sql并获取执行sql语句对象
            String sql = "insert into t_person values(null,?,?,?,?,?)";
            pstm = conn.prepareStatement(sql);
            pstm.setString(1,p.getpName());
            pstm.setInt(2,p.getAge());
```

```
            pstm.setString(3,p.getGender());
            pstm.setString(4,p.getMobile());
            pstm.setString(5,p.getAddress());
//          3.发送sql
            i = pstm.executeUpdate();
//          4.处理结果集
        }catch (Exception e){
            e.printStackTrace();
        }finally {
            //          5.关闭链接
            JdbcUtils.close(conn,pstm);
        }

        return i;
    }

    @Override
    public int updatePerson(Person p) {
        Connection conn = null;
        PreparedStatement pstm = null;
        int i = 0;
        try{
            //          1.获取链接
            conn = JdbcUtils.getConnection();
//          2.编写sql并获取执行sql语句对象
            String sql = "update t_person set
p_name=?,age=?,gender=?,mobile=?,address=? where p_id=?";
            pstm = conn.prepareStatement(sql);
            pstm.setString(1,p.getpName());
            pstm.setInt(2,p.getAge());
            pstm.setString(3,p.getGender());
            pstm.setString(4,p.getMobile());
            pstm.setString(5,p.getAddress());
            pstm.setInt(6,p.getPid());
//          3.发送sql
            i = pstm.executeUpdate();
//          4.处理结果集
        }catch (Exception e){
            e.printStackTrace();
        }finally {
            //          5.关闭链接
            JdbcUtils.close(conn,pstm);
        }
        return i;
    }

    @Override
    public int deletePersonById(int id) {
        Connection conn = null;
        PreparedStatement pstm = null;
        int i = 0;
        try{
            //          1.获取链接
            conn = JdbcUtils.getConnection();
//          2.编写sql并获取执行sql语句对象
            String sql = "delete from t_person where p_id = ?";
            pstm = conn.prepareStatement(sql);
            pstm.setInt(1,id);
```

```
//          3.发送sql
            i = pstm.executeUpdate();
//          4.处理结果集
        }catch (Exception e){
            e.printStackTrace();
        }finally {
            //          5.关闭链接
            JdbcUtils.close(conn,pstm);
        }
        return i;
    }

    @Override
    public Person selectPersonById(int id) {
        Connection conn = null;
        PreparedStatement pstm = null;
        ResultSet rs = null;
        Person person = null;
        try{
            //          1.获取链接
            conn = JdbcUtils.getConnection();
//          2.编写sql并获取执行sql语句对象
            String sql = "select * from t_person where p_id = ?";
            pstm = conn.prepareStatement(sql);
            pstm.setInt(1,id);
//          3.发送sql
            rs = pstm.executeQuery();
//          4.处理结果集
            while (rs.next()){
                person = new Person();
                person.setPid(rs.getInt("p_id"));
                person.setpName(rs.getString("p_name"));
                person.setAge(rs.getInt("age"));
                person.setGender(rs.getString("gender"));
                person.setMobile(rs.getString("mobile"));
                person.setAddress(rs.getString("address"));
            }
        }catch (Exception e){
            e.printStackTrace();
        }finally {
            //          5.关闭链接
            JdbcUtils.close(conn,pstm);
        }
        return person;
    }

    @Override
    public List<Person> selectAllPerson() {
        Connection conn = null;
        PreparedStatement pstm = null;
        ResultSet rs = null;
        Person person = null;
        List<Person> list = new ArrayList<>();
        try{
            //          1.获取链接
            conn = JdbcUtils.getConnection();
//          2.编写sql并获取执行sql语句对象
            String sql = "select * from t_person";
```

```
            pstm = conn.prepareStatement(sql);
//          3.发送sql
            rs = pstm.executeQuery();
//          4.处理结果集
            while (rs.next()){
                person = new Person();
                person.setPid(rs.getInt("p_id"));
                person.setpName(rs.getString("p_name"));
                person.setAge(rs.getInt("age"));
                person.setGender(rs.getString("gender"));
                person.setMobile(rs.getString("mobile"));
                person.setAddress(rs.getString("address"));
                list.add(person);
            }
        }catch (Exception e){
            e.printStackTrace();
        }finally {
            //          5.关闭链接
            JdbcUtils.close(conn,pstm);
        }
        return list;
    }
}
```

## 1.3 业务层

业务层（service）:根据功能的业务要求，实现具体的功能代码，通常业务实现由dao的调用以及一些逻辑判断代码组成。

service接口

```
一张表对应一个service接口
service层接口中定义表中所有的功能方法
service层接口名和表名进行关联， t_person --------> PersonService
所有的service层接口都需要放在service包里
```

service实现类

```
service实现类用户实现service接口
service实现类命名 = service接口名+Impl
所有的service实现类，定义在service包下的impl子包中
service实现类方法实现：调用dao+业务判断
```

代码案例：

service接口：

```
public interface PersonService {
//    添加
    public boolean addPerson(Person p);
//    修改
    public boolean modifyPerson(Person p);
//    删除
    public boolean removePersonById(int id);
//    根据id查询
    public Person queryPersonById(int id);
//    查询所有
    public List<Person> queryAllPerson();
}
```

service层实现类

```
package com.baizhi.service.impl;

import com.baizhi.dao.PersonDao;
import com.baizhi.dao.impl.PersonDaoImpl;
import com.baizhi.entity.Person;
import com.baizhi.service.PersonService;

import java.util.List;

/**
 * Service层需要依赖与DAO层，将DAO层对象定义在Service层的实现类中
 */
public class PersonServiceImpl implements PersonService {
//    依赖关系：在service层中定义全局DAO层的对象，
    PersonDao personDao = new PersonDaoImpl();
    @Override
    public boolean addPerson(Person p) {
//        service层的添加功能对应的是Dao层的insertXXX方法
        int i = personDao.insertPerson(p);
        if(i>0){
            return true;
        }else{
            return false;
        }
    }

    @Override
    public boolean modifyPerson(Person p) {
//        service层中的modifyXXX方法对应的DAO层updateXXX方法
        int i = personDao.updatePerson(p);
        if(i>0){
            return true;
        }else {
            return false;
        }

    }

    @Override
    public boolean removePersonById(int id) {
```

```
//        service层中的removeXXX方法对应DAO层的deleteXXX方法
        int i = personDao.deletePersonById(id);
        if(i>0){
            return true;
        }else{
            return false;
        }
    }

    @Override
    public Person queryPersonById(int id) {
//        service层中的queryXXXById对应的DAO层的selectXXXById方法
//        Person p = personDao.selectPersonById(id);
//        return p;
        return personDao.selectPersonById(id);
    }

    @Override
    public List<Person> queryAllPerson() {
//        servce成的queryAllXXX方法对应DAO层的selectAllXXX()方法
        return personDao.selectAllPerson();
    }
}
```

业务层-数据访问层-数据库操作之间的对应关系

| 操作 | 业务层 | 数据访问层 | 数据库 |
| --- | --- | --- | --- |
| 添加 | addXXX(XXX X) | insertXXX(XXX x) | insert into 表名... |
| 修改 | modifyXXX(XXX x) | updateXXX(XXX x) | update 表名 set.... |
| 删除 | removeXXXById(int id) | deleteXXXById(int id) | delete from 表名... |
| 查询单个 | queryXXXById(int id) | selectXXXById(int id) | select * from 表名 where... |
| 查询所有 | queryAllXXX() | selectAll() | select * from 表名; |
| | | | |

## 1.4 视图层（了解）

视图层（view）：负责功能的入口，（接收用户输入的数据）以及将功能执行的结果展示给用户

视图类：

```
一个功能对应一个视图类
视图类命名= 功能+view
视图类中的代码= 调用业务层代码+输出+scanner语句
所有的视图类都要定义在view包中
```

视图层代码案例：

添加视图

```
package com.baizhi.view;
```

```
import com.baizhi.entity.Person;
import com.baizhi.service.PersonService;
import com.baizhi.service.impl.PersonServiceImpl;

import java.util.Scanner;

public class AddPersonView {
    public static void main(String[] args) {
//        1.使用Scanner接收用户输入的数据
        Scanner sc = new Scanner(System.in);
        System.out.println("请输入姓名");
        String pName = sc.next();
        System.out.println("请输入年龄");
        int age = sc.nextInt();
        System.out.println("请输入性别");
        String gender = sc.next();
        System.out.println("请输入电话");
        String mobile = sc.next();
        System.out.println("请输入地址");
        String address = sc.next();
//        2.将输入的数据封装为对象
        Person p = new Person(null,pName,age,gender,mobile,address);
//        3.调用service层对象对应的添加方法
        PersonService personService = new PersonServiceImpl();
        boolean flag = personService.addPerson(p);
//        4.结果展示
        if(flag){
            System.out.println("添加成功");
        }else{
            System.out.println("添加失败");
        }
    }
}
```

查询视图

```
public class GetAllPersonView {
    public static void main(String[] args) {
//        1.创建PersonService层对象
        PersonService personService = new PersonServiceImpl();
//        2.通过personService对象调用查询所有方法
        List<Person> list =personService.queryAllPerson();
//        3.将结果展示
        for(Person p:list){
            System.out.println(p);
        }
    }
}
```

## 1.5 三层架构的开发步骤

```
1.建表
2.实体类
3.dao层（先接口，后实现）
4.service层（先接口，后实现）
5.view层
```

# 2.事务控制

## 2.1 事务回顾

事务：保证一个业务（多个sql）操作的完整性的数据库机制,保证同一个业务的多条sql要么同时执行成功，要么同时执行失败。JDBC的事务控制在service层中进行控制。

service层中的一个方法对应一个业务功能，一个功能对应的多条sql,(在一个service层的方法多次调用DAO层，为了保证业务层的一个方法执行多条sql时要么成功，要么全部失败。

场景：转账功能，需要2条跟新语句，在service层中对应的就是一个transfer方法，在transfer中需要保证两个跟新语句要么都成功，要么都失败。

MySQL事务的操作

```
-- 开启事务
begin/start transaction;
-- 执行sql语句
update t_account  set balance = balance -? where id = ?;
update t_account set balance = balance +? where id = ?;
-- 提交/回滚
commit;/rollback;
```

service层控制事务的模板

```
try{
// 开启事务     conn.setAutoCommit(false);
// 调用dao层对应的操作
// 提交事务     conn.commit();
}catch(Exception e){
    //回滚事务    conn.rollback();
}
```

service层控制事务核心

```
1.一个service层方法对应一个事务
2.一个service方法就是一个不可分割的整体
```

## 2.2 事务案例

场景：t_account表中，lishuo向huake转账500

准备：

```
-- 创建账号表
CREATE TABLE `t_account`  (
  `id` int(10) primary key AUTO_INCREMENT,
  `name` varchar(20)  NOT NULL,
  `balance` decimal(10, 0) NOT NULL
)
-- 添加测试数据
INSERT INTO `t_account` VALUES (1001, 'lishuo', 1500);
INSERT INTO `t_account` VALUES (1002, 'huake', 500);
INSERT INTO `t_account` VALUES (1003, 'css', 0);
```

代码案例：

实体层（entity）

```
package com.baizhi.entity;

import java.io.Serializable;

public class Account implements Serializable {
//    属性
    private Integer id;
    private String name;
    private Double balance;
//    get/set

    public Integer getId() {
        return id;
    }

    public void setId(Integer id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public Double getBalance() {
        return balance;
    }

    public void setBalance(Double balance) {
        this.balance = balance;
    }
    //有参无参构造

    public Account() {
    }

    public Account(Integer id, String name, Double balance) {
        this.id = id;
        this.name = name;
```

```
        this.balance = balance;
    }

    @Override
    public String toString() {
        return "Account{" +
                "id=" + id +
                ", name='" + name + '\'' +
                ", balance=" + balance +
                '}';
    }
}
```

数据访问层（DAO）

接口代码

```
public interface AccountDao {
//    加钱操作
    public int updateBalanceAdd(int id,double money);
//    减钱操作
    public int updateBalanceSub(int id,double money);
}
```

实现类代码

```
package com.baizhi.dao.impl;

import com.baizhi.dao.AccountDao;
import com.baizhi.utils.JdbcUtils;

import java.sql.Connection;
import java.sql.PreparedStatement;

public class AccountDaoImpl implements AccountDao {
    @Override
    public int updateBalanceAdd(int id, double money) {
        Connection conn = null;
        PreparedStatement pstm = null;
        int i = 0 ;
        try{
            //        1.获取链接
            conn = JdbcUtils.getConnection();
//          2.编写sql,并获取执行sql语句对象
            String sql = "update t_account set balance = balance-? where id =
?";
            pstm = conn.prepareStatement(sql);
            pstm.setDouble(1,money);
            pstm.setInt(2,id);
//          3.发送sql
            i = pstm.executeUpdate();
//          4.处理结果

        }catch (Exception e){
            e.printStackTrace();
```

```
        }finally {
            //        5.关闭链接
            JdbcUtils.close(conn,pstm);
        }

        return i;
    }

    @Override
    public int updateBalanceSub(int id, double money) {
        Connection conn = null;
        PreparedStatement pstm = null;
        int i = 0 ;
        try{
            //        1.获取链接
            conn = JdbcUtils.getConnection();
//          2.编写sql,并获取执行sql语句对象
            String sql = "update t_account set balance = balance+? where id =
?";
            pstm = conn.prepareStatement(sql);
            pstm.setDouble(1,money);
            pstm.setInt(2,id);
//          3.发送sql
            i = pstm.executeUpdate();
//          4.处理结果

        }catch (Exception e){
            e.printStackTrace();
        }finally {
            //        5.关闭链接
            JdbcUtils.close(conn,pstm);
        }

        return i;
    }
}
```

业务层（service）

接口代码

```
public interface AccountService {

    /**
     * 这是转账业务的方法,
     * @param addid  加钱账号id
     * @param subid  减钱账号id
     * @param money     转账钱数
     * @return  是否转账成功 true :成功   false:失败
     */
    public boolean transfer(int addid,int subid,double money);
}
```

实现类代码

```java
package com.baizhi.service.impl;

import com.baizhi.dao.AccountDao;
import com.baizhi.dao.impl.AccountDaoImpl;
import com.baizhi.service.AccountService;
import com.baizhi.utils.JdbcUtils;

import java.sql.Connection;
import java.sql.SQLException;

public class AccountServiceImpl implements AccountService {
//    1.添加dao层依赖
    AccountDao accountDao = new AccountDaoImpl();
    @Override
    public boolean transfer(int addid, int subid, double money) {
        Connection conn = JdbcUtils.getConnection();
        try{
//            1.开启事务

            conn.setAutoCommit(false);      //开启事务
//            2.调用dao层方法
//                  先进行减钱操作
            accountDao.updateBalanceSub(subid,money);
//                  在进行加钱操作
            accountDao.updateBalanceAdd(addid,money);
//            3.提交事务
            conn.commit();
            return true;
        }catch (Exception e){
//            4.回滚事务
            try {
                conn.rollback();
            } catch (SQLException throwables) {
                throwables.printStackTrace();
            }
            e.printStackTrace();
        }
        return false;
    }
}
```

测试

```java
public class TransferView {
    public static void main(String[] args) {
//        视图
        Scanner sc = new Scanner(System.in);
        System.out.println("请输入减钱id");
        int subid = sc.nextInt();
        System.out.println("请输入加钱id");
        int addid = sc.nextInt();
        System.out.println("请输入转账金额");
        double money = sc.nextDouble();
//        添加service层依赖
        AccountService accountService = new AccountServiceImpl();
        boolean flag =  accountService.transfer(addid,subid,money);
```

```
        if(flag){
            System.out.println("转账成功");
        }else{
            System.out.println("转账失败");
        }
    }
}

-- 程序执行结果 转账成功
```

思考：转账事务是不是就正常执行成功了呢?

验证事务是否正常执行，需要验证在发生异常情况下，事务是否能够回滚。

验证代码：在service层的转账方法中，在两次dao层调用之间手动制造异常。

案例如下：

```
public class AccountServiceImpl implements AccountService {
//    1.添加dao层依赖
    AccountDao accountDao = new AccountDaoImpl();
    @Override
    public boolean transfer(int addid, int subid, double money) {
        Connection conn = JdbcUtils.getConnection();
        try{
//            1.开启事务
            conn.setAutoCommit(false);       //开启事务
//            2.调用dao层方法
//                先进行减钱操作
            accountDao.updateBala
//            通过手动制造异常，模拟特殊情况，看事务是否能够回滚
            int i = 1/0;
//                在进行加钱操作
            accountDao.updateBalanceAdd(addid,money);
//            3.提交事务
            conn.commit();
            return true;
        }catch (Exception e){
//            4.回滚事务
            try {
                conn.rollback();
            } catch (SQLException throwables) {
                throwables.printStackTrace();
            }
            e.printStackTrace();
        }
        return false;
    }
}
```

制造异常发生，模拟特殊情况，验证事务是否能够回滚

验证结果：出现异常，事务没有进行回滚，

结论：事务失败

事务失败的原因：同一个事务的多个sql必须位于同一给链接中，而上面的案例中在service层，dao层中的每个操作都是一个独立的链接,各个链接相互独立，导致事务失败。

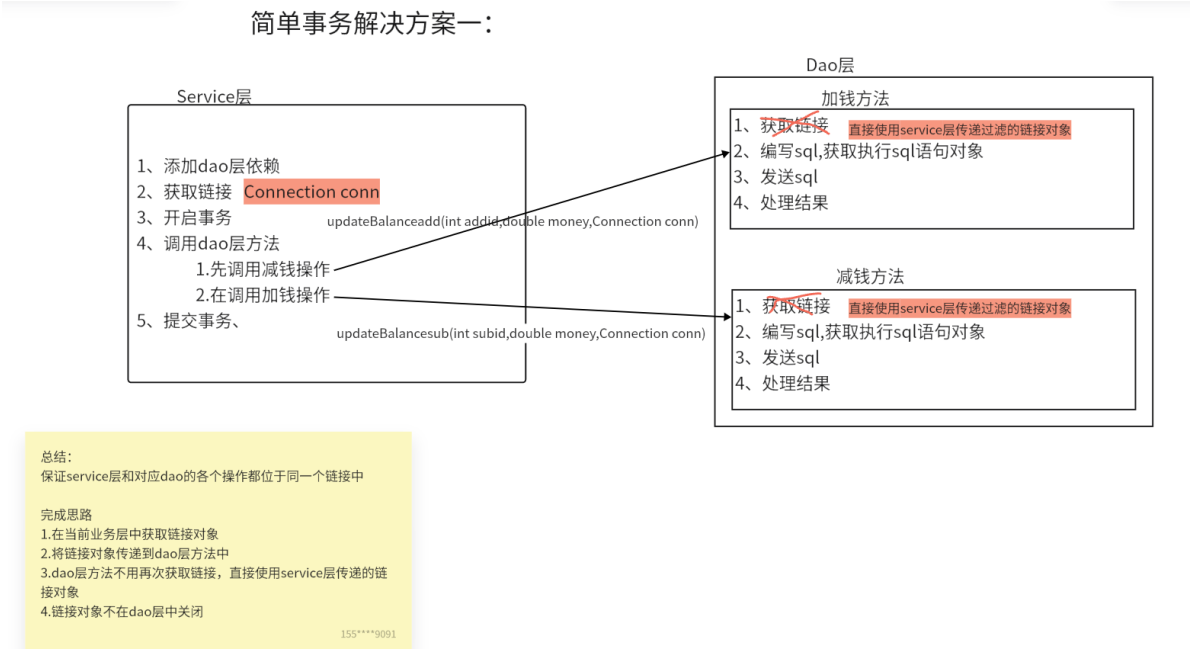
如何解决?

通过代码保证service,dao层的每个操作都位于同一个链接中即可。

## 2.3 事务控制解决方案一

总结：

保证service层和对应dao的各个操作都位于同一个链接中




完成思路

1.在当前业务层中获取链接对象

2.将链接对象传递到dao层方法中

3.dao层方法不用再次获取链接，直接使用service层传递的链接对象

4.链接对象不在dao层中关闭

代码案例：

1.改造dao层接口和实现类

```
// dao层接口
public interface AccountDao {
//     加钱操作
    public int updateBalanceAdd(int id, double money, Connection conn);
//     减钱操作
    public int updateBalanceSub(int id,double money,Connection conn);
}

//dao层实现类

package com.baizhi.dao.impl;

import com.baizhi.dao.AccountDao;
import com.baizhi.utils.JdbcUtils;

import java.sql.Connection;
import java.sql.PreparedStatement;

public class AccountDaoImpl implements AccountDao {
    @Override
    public int updateBalanceAdd(int id, double money,Connection conn) {
```

```
        PreparedStatement pstm = null;
        int i = 0 ;
        try{
            //        1.获取链接
//            conn = JdbcUtils.getConnection();
//        2.编写sql,并获取执行sql语句对象
            String sql = "update t_account set balance = balance+? where id =
?";
            pstm = conn.prepareStatement(sql);
            pstm.setDouble(1,money);
            pstm.setInt(2,id);
//        3.发送sql
            i = pstm.executeUpdate();
//        4.处理结果

        }catch (Exception e){
            e.printStackTrace();
        }finally {
//            因为要保证同一个事务的多个操作位于同一个链接中，所以在dao层操作中不能关闭链接
            JdbcUtils.close(null,pstm);
        }
        return i;
    }

    @Override
    public int updateBalanceSub(int id, double money,Connection conn) {
        PreparedStatement pstm = null;
        int i = 0 ;
        try{
            //        1.获取链接
//            conn = JdbcUtils.getConnection();
//        2.编写sql,并获取执行sql语句对象
            String sql = "update t_account set balance = balance-? where id =
?";
            pstm = conn.prepareStatement(sql);
            pstm.setDouble(1,money);
            pstm.setInt(2,id);
//        3.发送sql
            i = pstm.executeUpdate();
//        4.处理结果

        }catch (Exception e){
            e.printStackTrace();
        }finally {
            //        5.关闭链接
            //            因为要保证同一个事务的多个操作位于同一个链接中，所以在dao层操作中
不能关闭链接
            JdbcUtils.close(null,pstm);
        }

        return i;
    }
}
```

service层代码

```java
// service层接口
public interface AccountService {

    /**
     * 这是转账业务的方法，
     * @param addid  加钱账号id
     * @param subid  减钱账号id
     * @param money      转账钱数
     * @return  是否转账成功 true :成功   false:失败
     */
    public boolean transfer(int addid,int subid,double money);
}

// service层实现类
package com.baizhi.service.impl;

import com.baizhi.dao.AccountDao;
import com.baizhi.dao.impl.AccountDaoImpl;
import com.baizhi.service.AccountService;
import com.baizhi.utils.JdbcUtils;

import java.sql.Connection;
import java.sql.SQLException;

public class AccountServiceImpl implements AccountService {
//    1.添加dao层依赖
    AccountDao accountDao = new AccountDaoImpl();
    @Override
    public boolean transfer(int addid, int subid, double money) {
//        在servcie层中获取链接对象
        Connection conn = JdbcUtils.getConnection();
        try{
//            1.开启事务
            conn.setAutoCommit(false);      //开启事务
//            2.调用dao层方法
//                  先进行减钱操作
            accountDao.updateBalanceSub(subid,money,conn);
//            通过手动制造异常，模拟特殊情况，看事务是否能够回滚
            int i = 1/0;
//                  在进行加钱操作
            accountDao.updateBalanceAdd(addid,money,conn);
//            3.提交事务
            conn.commit();
            return true;
        }catch (Exception e){
//            4.回滚事务
            try {
                conn.rollback();
            } catch (SQLException throwables) {
                throwables.printStackTrace();
            }
            e.printStackTrace();
        }finally {
            JdbcUtils.close(conn,null);
        }
        return false;
    }
```

```
}
```

验证：

代码案例：

```
package com.baizhi.view;

import com.baizhi.service.AccountService;
import com.baizhi.service.impl.AccountServiceImpl;

import java.util.Scanner;

public class TransferView {
    public static void main(String[] args) {
//        视图
        Scanner sc = new Scanner(System.in);
        System.out.println("请输入减钱id");
        int subid = sc.nextInt();
        System.out.println("请输入加钱id");
        int addid = sc.nextInt();
        System.out.println("请输入转账金额");
        double money = sc.nextDouble();
//        添加service层依赖
        AccountService accountService = new AccountServiceImpl();
        boolean flag =  accountService.transfer(addid,subid,money);
        if(flag){
            System.out.println("转账成功");
        }else{
            System.out.println("转账失败");
        }
    }
}
程序运行结果
请输入减钱id
1001
请输入加钱id
1002
请输入转账金额
500
java.lang.ArithmeticException: / by zero
    at
com.baizhi.service.impl.AccountServiceImpl.transfer(AccountServiceImpl.java:25)
    at com.baizhi.view.TransferView.main(TransferView.java:20)
转账失败

    查看数据库发现数据回滚成功!
```

简单事务总结

1.同一个业务多个dao层操作要在同一个conn中进行

2.service层中conn和dao层中的conn是同一个对象。

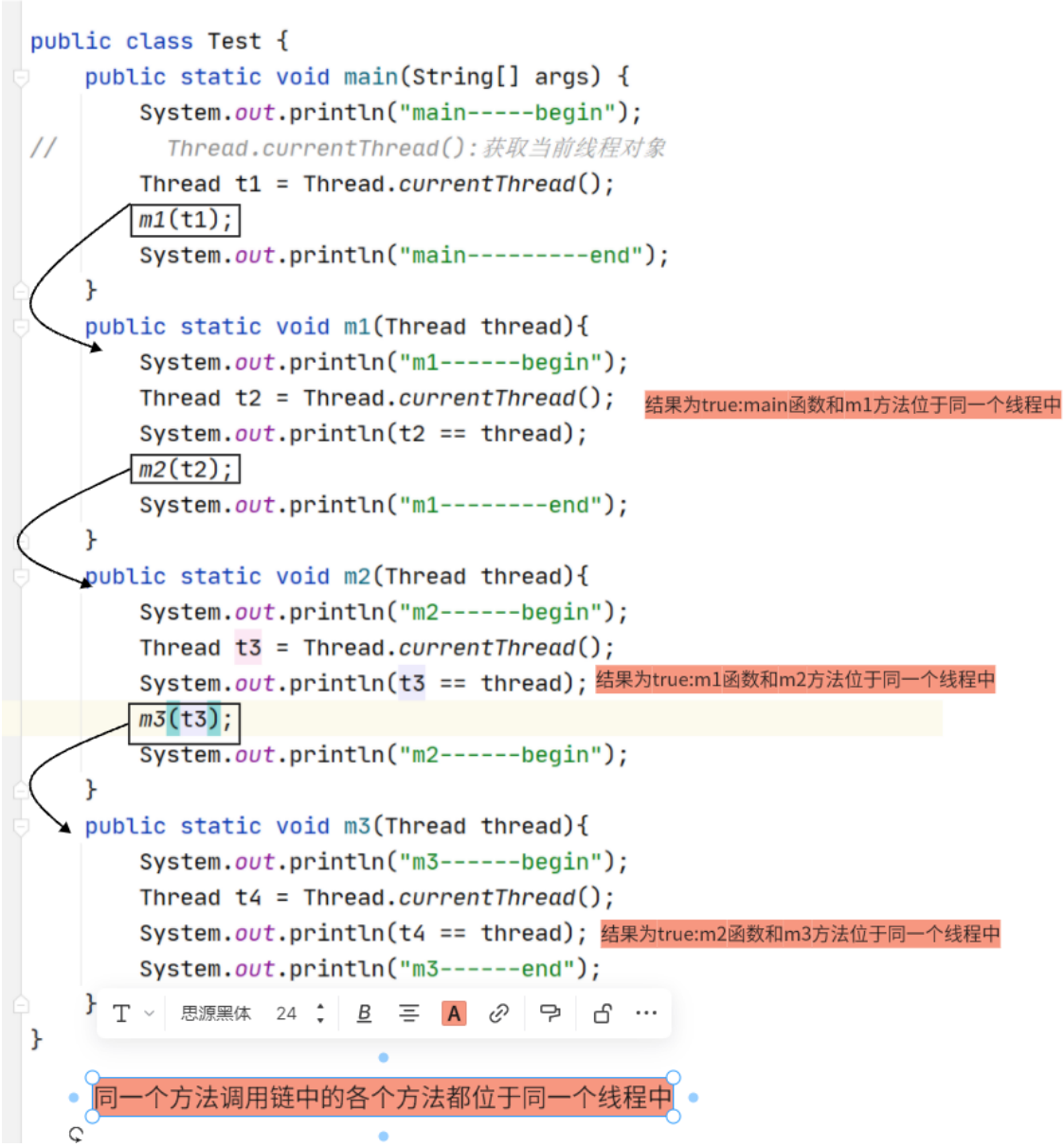
结论，如果要保证事务正常的执行，必须保证事务中每个dao层操作使用同一个conn对象。

存在问题

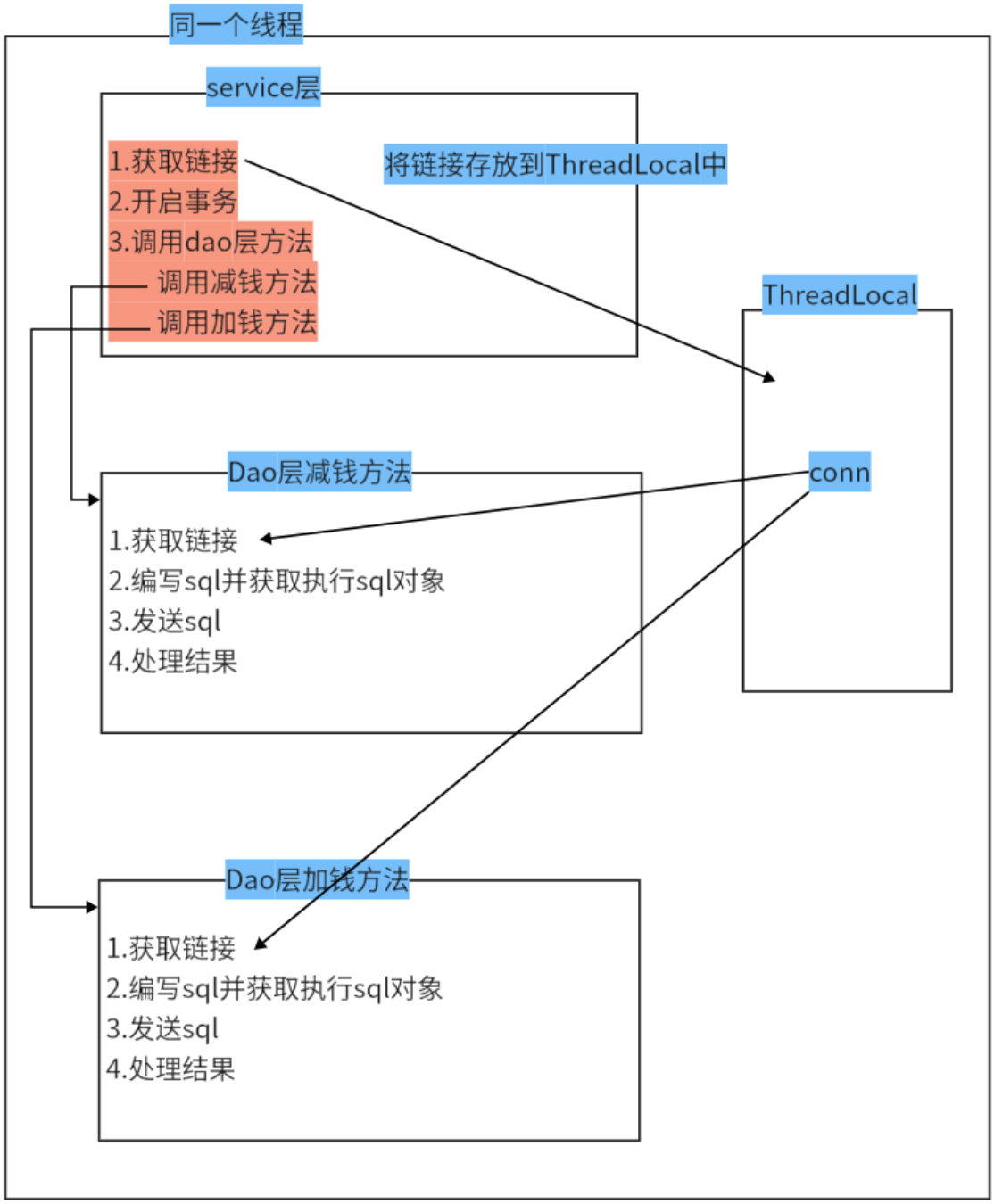
```
1.定义繁琐，开发效率低下
2.API污染，无形中和原生的JDBC技术耦合，不易进行后期的项目扩展
```

## 2.4 事务控制解决方案二--ThreadLocal

ThreadLocal:通过线程对象将service创建的链接传递的dao层




ThreadLocal:线程对象中有一块内存存储空间，可以用来保存数据（链接对象），在线程的执行流程中所设计到的代码都可以对该内存进行操作（保存和读取），这块内存对于该线程中所有的被调用的方法而言都是共享的。




方案：可以使用Thread对象在service层和dao层中传递链接，解决service层和dao层使用同一个链接的问题。

使用步骤：1.在业务层中第一次获取链接，存放到ThreadLocal中

2.在DAO层中的每个方法中直接通过ThreadLocal来获取链接对象。

**如何操作当前线程的存储空间?**

ThreadLocal:操作线程存储空间的工具，该类型的对象可以使用当前线程对象的存储空间，进行保存，获取，删除数据等操作

ThreadLocal操作方法：

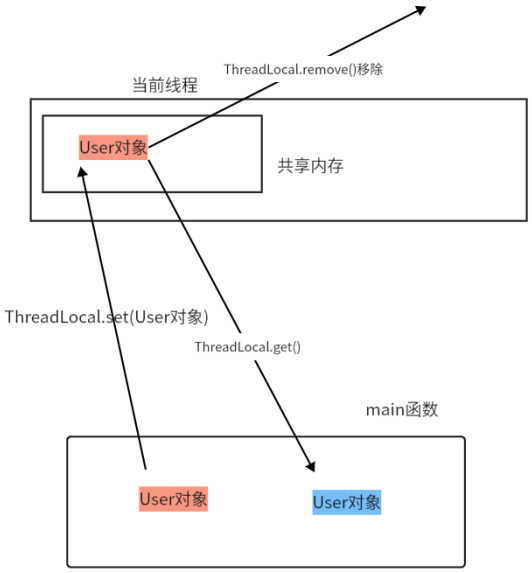
1. 向线程存储空间保存数据： ThreadLocal对象.set(数据)
2. 从线程存储空间获取数据：ThreadLocal对象.get()
3. 从当前线程存储空间移除数据： ThreadLocal对象.remove()

工具介绍：ThreadLocal
作用：操作当前线程内部一块存储空间（属性）
创建：ThreadLocal<泛型> tdl = new ThreadLocal<泛型>（）
常用方法
1.threadLocal.set(值)； 将数据存入当前线程
2.threadLocal.get()；从当前线程中获取值
3.threadLocal.remove();从当前线中移除值

代码案例:

```
public static void main(String[] args) {
        User user = new User(1001,"zhangsan","123456",12,"110");
    //    创建ThreadLocal对象
        ThreadLocal<User> tdl = new ThreadLocal<>();
    //    通过ThreadLocal对象将user对象存放到当前线程中
        tdl.set(user);
    //    从当前线程中获取存放的对象
        User user1 = tdl.get();
    }
```

155****9091




## 2.4.1JDBC工具类最终版

```
package utils;

import java.io.IOException;
import java.io.InputStream;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.util.Properties;

public class JdbcUtilsFinal {
//    添加全局的ThreadLocal对象
    private static ThreadLocal<Connection> tl = new ThreadLocal<Connection>();
    //    声明属性对象
    static Properties properties = new Properties();
    public static String driverClass = null;
    public static String url = null;
    public static String user = null;
    public static String password = null;
    //    使用静态代码块读取配置文件的内容
    static{
        try {
//            InputStream input = new FileInputStream("jdbc.properties");
            //            通过IO流读取配置文件
            InputStream input =
JdbcUtilsFinal.class.getResourceAsStream("jdbc.properties");
            //        通过属性对象加载配置文件
            properties.load(input);
//            通过属性对象获取配置文件中的配置参数
            driverClass = properties.getProperty("driverClass");
            url = properties.getProperty("url");
            user = properties.getProperty("user");
            password = properties.getProperty("password");
        } catch (IOException e) {
            e.printStackTrace();
```

```
        }
    }
    public static Connection getConnection(){
//          直接使用ThreadLocal对象获取当前线程存储空间的链接对象
        Connection conn = tl.get();
//          如果直接从线程存储空间获取的链接为null,则重新创建链接，通过ThreadLocal对象存放到当
前线程存储空间中
        if(conn == null){
            try{
//              加载驱动
                Class.forName(driverClass);
//              获取链接
                conn = DriverManager.getConnection(url,user,password);
//                  向当前线程存储空间中添加链接对象
                tl.set(conn);

            }catch (Exception e){
                e.printStackTrace();
            }
            return conn;
        }else{
//              可以从当前线程存储空间获取链接对象，直接返回
            return conn;
        }
    }
    public static void close(Connection conn, PreparedStatement pstm, ResultSet
rs){
        try{
//              在关闭之前判断，该对象是否为空
            if(rs!=null){
                rs.close();
            }
//              不要将三个关闭动作写作同一个trycatch中，否则等前面的出现异常，后面的资源就无法
关闭

        }catch (Exception e){
            e.printStackTrace();
        }
        try{
            if(pstm!=null){
                pstm.close();
            }
        }catch (Exception e){
            e.printStackTrace();
        }
        try{
            if(conn!=null){
                conn.close();
//                  当使用完毕后，将链接对象从线程存储空间中移除
                tl.remove();
            }
        }catch (Exception e){
            e.printStackTrace();
        }
    }

    /**
     * close方法的重载，为了方便处理不同的关闭情况，两个参数的close方法主要用来关闭增删改操作
```

```
     * @param conn
     * @param pstm
     */
    public static void close(Connection conn, PreparedStatement pstm){

        try{
            if(pstm!=null){
                pstm.close();
            }
        }catch (Exception e){
            e.printStackTrace();
        }
        try{
            if(conn!=null){
                conn.close();
                tl.remove();
            }
        }catch (Exception e){
            e.printStackTrace();
        }

    }
}
```

**基于ThreadLocal完整事务案例**

第二版基于ThreadLocal事务管理步骤：

1.在Service层的业务方法中获取链接对象，

2.在JdbcUtils中创建全局静态ThreadLocal对象，

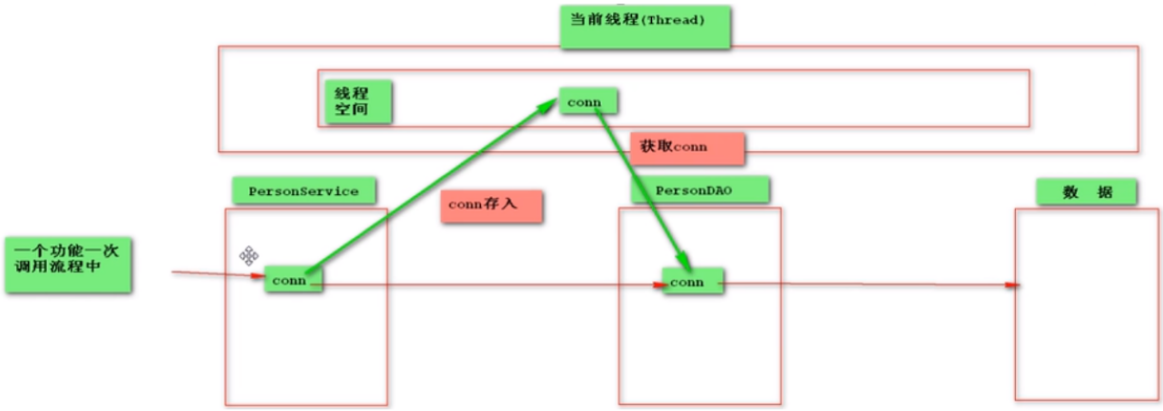
3.在JdbcUtils工具类中的getConnection()中，先通过全局的ThreadLocald对象从当前线程中获取conn对象，

4.判断获取的conn对象是否为空，如果为空则新建一个链接，通过ThreadLocal存放进当前线程。

5.dao层中的操作方法可以正常从Jdbcutils工具类获取链接对象

6.链接对象不能在dao层中关闭

7.链接对象只能在service层中关闭，在JdbcUtils工具类的close方法中，要将链接对象从当前线程移除。




entity代码

```
package com.baizhi.entity;

import java.io.Serializable;

public class Account implements Serializable {
//    属性
    private Integer id;
    private String name;
    private Double balance;
//    get/set

    public Integer getId() {
        return id;
    }

    public void setId(Integer id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public Double getBalance() {
        return balance;
    }

    public void setBalance(Double balance) {
        this.balance = balance;
    }
    //有参无参构造

    public Account() {
    }

    public Account(Integer id, String name, Double balance) {
        this.id = id;
```

```
        this.name = name;
        this.balance = balance;
    }

    @Override
    public String toString() {
        return "Account{" +
                "id=" + id +
                ", name='" + name + '\'' +
                ", balance=" + balance +
                '}';
    }
}
```

dao层接口代码

```
package dao;

import java.sql.Connection;

public interface AccountDao {
//    加钱操作
    public int updateBalanceAdd(int id, double money);
//    减钱操作
    public int updateBalanceSub(int id,double money);
}
```

dao层实现类代码

```
package dao.impl;

import dao.AccountDao;
import utils.JdbcUtilsFinal;

import java.sql.Connection;
import java.sql.PreparedStatement;

public class AccountDaoImpl implements AccountDao {
    @Override
    public int updateBalanceAdd(int id, double money) {
        Connection conn = null;
        PreparedStatement pstm = null;
        int i = 0 ;
        try{
            //        1.获取链接
            conn = JdbcUtilsFinal.getConnection();
//        2.编写sql,并获取执行sql语句对象
            String sql = "update t_account set balance = balance+? where id =
?";
            pstm = conn.prepareStatement(sql);
            pstm.setDouble(1,money);
            pstm.setInt(2,id);
//        3.发送sql
            i = pstm.executeUpdate();
```

```
//        4.处理结果

        }catch (Exception e){
            e.printStackTrace();
        }finally {
//            因为要保证同一个事务的多个操作位于同一个链接中，所以在dao层操作中不能关闭链接
            JdbcUtilsFinal.close(null,pstm);
        }
        return i;
    }

    @Override
    public int updateBalanceSub(int id, double money) {
        Connection conn=null;
        PreparedStatement pstm = null;
        int i = 0 ;
        try{
            //        1.获取链接
            conn = JdbcUtilsFinal.getConnection();
//        2.编写sql,并获取执行sql语句对象
            String sql = "update t_account set balance = balance-? where id =
?";
            pstm = conn.prepareStatement(sql);
            pstm.setDouble(1,money);
            pstm.setInt(2,id);
//        3.发送sql
            i = pstm.executeUpdate();
//        4.处理结果

        }catch (Exception e){
            e.printStackTrace();
        }finally {
            //        5.关闭链接
            //            因为要保证同一个事务的多个操作位于同一个链接中，所以在dao层操作中
不能关闭链接
            JdbcUtilsFinal.close(null,pstm);
        }

        return i;
    }
}
```

service接口代码

```
package service;

public interface AccountService {

    /**
     * 这是转账业务的方法，
     * @param addid  加钱账号id
     * @param subid  减钱账号id
     * @param money     转账钱数
     * @return  是否转账成功 true :成功   false:失败
     */
    public boolean transfer(int addid,int subid,double money);
```

```
}
```

service实现类代码

```
package service.impl;

import dao.AccountDao;
import dao.impl.AccountDaoImpl;
import service.AccountService;
import utils.JdbcUtilsFinal;

import java.sql.Connection;
import java.sql.SQLException;

public class AccountServiceImpl implements AccountService {
//    1.添加dao层依赖
    AccountDao accountDao = new AccountDaoImpl();
    @Override
    public boolean transfer(int addid, int subid, double money) {
//        在servcie层中获取链接对象
        Connection conn = JdbcUtilsFinal.getConnection();
        try{
//            1.开启事务
            conn.setAutoCommit(false);      //开启事务
//            2.调用dao层方法
//                  先进行减钱操作
            accountDao.updateBalanceSub(subid,money);
//            通过手动制造异常，模拟特殊情况，看事务是否能够回滚
            int i = 1/0;
//                  在进行加钱操作
            accountDao.updateBalanceAdd(addid,money);
//            3.提交事务
            conn.commit();
            return true;
        }catch (Exception e){
//            4.回滚事务
            try {
                conn.rollback();
            } catch (SQLException throwables) {
                throwables.printStackTrace();
            }
            e.printStackTrace();
        }finally {
            JdbcUtilsFinal.close(conn,null);
        }
        return false;
    }
}
```

view层测试代码

```
package view;

import service.AccountService;
```

```
import service.impl.AccountServiceImpl;

import java.util.Scanner;

public class TransferView {
    public static void main(String[] args) {
//        视图
        Scanner sc = new Scanner(System.in);
        System.out.println("请输入减钱id");
        int subid = sc.nextInt();
        System.out.println("请输入加钱id");
        int addid = sc.nextInt();
        System.out.println("请输入转账金额");
        double money = sc.nextDouble();
//        添加service层依赖
        AccountService accountService = new AccountServiceImpl();
        boolean flag =  accountService.transfer(addid,subid,money);
        if(flag){
            System.out.println("转账成功");
        }else{
            System.out.println("转账失败");
        }
    }
}
```

## 3.Junit测试框架

为了尽可能减少项目后期的Bug,在开发的时候一定会对代码进行测试，JDBC中主要是dao，service进行测试，当前测试环境的问题。

```
public class TransferView {
    public static void main(String[] args) {
//        视图
        Scanner sc = new Scanner(System.in);
        System.out.println("请输入减钱id");
        int subid = sc.nextInt();
        System.out.println("请输入加钱id");
        int addid = sc.nextInt();
        System.out.println("请输入转账金额");
        double money = sc.nextDouble();
//        添加service层依赖
        AccountService accountService = new AccountServiceImpl();
        boolean flag =  accountService.transfer(addid,subid,money);
        if(flag){
            System.out.println("转账成功");
        }else{
            System.out.println("转账失败");
        }
    }
}
```

一个要测试的方法就要由一个测试类，后期项目的功能方法会特别多，那么就需要更多的测试类，导致测试代码越来越臃肿，难以管理。

解决办法：使用Junit测试框架

作用：在junit中每个方法都可以单独运行，每个方法都可以单独测试对应的功能，减少了测试类的数量

使用思路：在方法上添加@Test注解，这样改方法就可以像main函数一样单独执行。

使用步骤：

1. 在项目中引入junit4框架对应的jar包


2. 使用框架 ,在一个测试类中，定义多个方法，在方法上面添加@Test注解

```
import org.junit.Test;

public class AccountTest {
    @Test
    public void testAccountDaoAddMoney(){
        System.out.println("hello world");
    }
}
```

```
tDaoAddMoney
Tests passed: 1 of 1 test – 6 ms
"C:\Program Files\Java\jdk1.8.0_161\bin\java.exe" ...
hello world
```

使用要求：

1. 测试类必须是公开的
2. 测试方法必须是公开的非静态的无参无返回值方法
3. 注意：同包下尽量不要定义名为Test的类，避免冲突
4. 注意：在junit的方法中尽量不要使用scanner对象，如果测试需要数据，则自行编写死数据。

dao+service中的每个方法都应该测试一下，

# 4.JDBC项目开发步骤总结

1.新建一个项目

2.环境准备

- 在项目根目录下新建lib目录，导入mysql驱动包，junit测试包，添加为类库
- 添加jdbc.properties配置文件
- 添加JDBCUtils工具类

3.建表

4.创建实体类

- 一张表对应一个实体类
- 实体的属性对应表中的字段
- 私有的属性+公开的get/set方法+有参无参构造方法+toString()
- 实现Serializable接口
- 所有的实体类都放在entity(bean,pojo)包中

5.dao层

dao接口

- 一张表对应一个dao接口
- dao接口中定义对该表的操作方法
- dao接口命名和表名进行关联t_person ------> PersonDao
- 所有的dao接口都放在dao包中

dao实现类

- 实现类必须实现对应的dao接口，重写接口中的方法
- 实现类的方法准许jdbc6步操作
- 实现类的命名 = 接口名+ Impl
- 所有的实现类必须放在dao包下的impl子包中

6.service层

service接口

- 一张表对应一个service接口
- service接口中定义对于该表的功能方法
- service命名和表名进行关联 t_person---------> PersonService
- service接口都要放在service包中

service实现类

- 实现类必须实现对应的service接口，重写接口中的方法
- 实现类的方法中可以添加事务控制
- 实现类方法组成 = dao层调用+业务处理+事务控制
- 实现类命名= service接口名+Impl
- 所有的service实现类必须放在service包下的impl子包中

7.view层(了解)

一个功能对应一个view视图类，通常使用junit进行替换