day26_JDBC进阶

一 内容回顾（列举前一天重点难点内容）

1.1 教学重点:

```
1.熟练掌握jdbc的基本操作
2.熟练掌握jdbc的实现原理
3.基本掌握项目架构搭建初级思想
4.熟练掌握jdbc中模型封装
5.熟练掌握jdbc中工具类的封装
6.了解jdbc到批处理操作
7.了解jdbc到sql注入问题
```

1.2 教学难点:

```
jdbc的代码实现原理
```

二 教学目标

```
1.掌握事务的简单使用
2.掌握事务的特性
3.掌握事务的隔离级别
4.掌握常用的三方连接池
5.掌握DAO设计模式
6.熟练使用DButils实现增删改
7.熟练使用DButils实现模型封装
8.掌握xml和json的简单编写
```

三 教学导读

今天的内容是jdbc的进阶知识,主要涉及到事务,连接池,DAO层设计,DButils三方工具,以及xml和json的讲解

四 教学内容

4.1 JDBC的事务支持(会)

4.1.1. JDBC的事务支持

4.1.1.1. 事务的概念

```
当一个业务需求涉及到N个DML操作时，这个业务(或者时N个DML操作)当成一个整体来处理。在处理的过程中，如果有失败或异常，我们要回到业务开始时。如果成功处理，我们再将数据持久化到磁盘中。这样一个过程我们称之为一个事务。事务具有原子性。不可切割。

总结：
事务指逻辑上的一组操作，组成这组操作的各个单元，要么全成功，要么全不成功。

关键字：
commit
rollback
savepoint
```

4.1.1.2. 事务的特性(ACID)

- 原子性(Atomicity)

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)

事务必须使数据库从一个一致性状态变换到另外一个一致性状态。转账前和转账后的总金额不变。

- 隔离性(Isolation)

事务的隔离性是多个用户并发访问数据库时，数据库为每一个用户开启的事务，不能被其他事务的操作数据所干扰，多个并发事务之间要相互隔离。

- 持久性(Durability)

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。

4.1.1.3. MySQL事务

```
- 默认情况下，MySQL每执行一条SQL语句，都是一个单独的事务。
- 如果需要在一个事务中包含多条SQL语句，那么需要开启事务和结束事务。
   开启事务：start transaction;
   结束事务：commit或rollback;

   事务开始于
•连接到数据库上，并执行一条DML语句insert、update或delete
•前一个事务结束后，又输入了另一条DML语句

事务结束于
•执行commit或rollback语句。
•执行一条DDL语句，例如create table语句，在这种情况下，会自动执行commit语句。
•执行一条DDL语句，例如grant语句，在这种情况下，会自动执行commit。
•断开与数据库的连接
```

```
15  •执行了一条DML语句，该语句却失败了，在这种情况中，会为这个无效的
    DML语句执行rollback语句。
```

示例:sql语句实现事务支持

1.回滚情况

```sql
START TRANSACTION;
UPDATE account SET balance=balance-10000 WHERE id=1;
SELECT * FROM account;
UPDATE account SET balance=balance+10000 WHERE id=2;
ROLLBACK;
```

2.提交情况

```sql
START TRANSACTION;
UPDATE account SET balance=balance-10000 WHERE id=1;
SELECT * FROM account;
UPDATE account SET balance=balance+10000 WHERE id=2;
COMMIT;
```

4.1.1.4. JDBC的事务支持(手动控制事务)

```
Connection.setAutoCommit(boolean flag):此方法可以取消事务的
自动提交功能，值为false。
Connection.commit()： 进行事务提交。
Connection.rollback():进行事务回滚.
```

示例代码

```
Connection conn = null;
PreparedStatement ps = null;
try{
    conn = DBUtils.getConnection();
    conn.setAutoCommit(false); //相当于begin
    ps = conn.prepareStatement("update account set money=money-100 where id=1");
    ps.executeUpdate();
    //int i = 10/0;
    ps = conn.prepareStatement("update account set money=money+100 where id=2");
    ps.executeUpdate();

    conn.commit();  //提交事务 commit
}catch(Exception e){
    if(conn!=null){
        try {
            conn.rollback(); //回滚事务 rollback
        } catch (SQLException e1) {
            e1.printStackTrace();
        }
    }
    e.printStackTrace();
}finally{
    DBUtils.close(rs,ps,conn);
```

4.1.1.5. 多事务的情况

脏读：事务A读取了事务B刚刚更新的数据，但是事务B回滚了，这样就导致事务A读取的为脏数据，我们称之为脏读。

如公司某财务人员更新公司入账报表时，在DML语句中的数字后少添加了一个0，但是未提交，然后吃饭，吃饭回来，发现错误，然后更正后做了提交。而在吃饭期间，老板要求秘书查看一下报表，秘书看到的是少个0的数据。这就是脏读。

不可重复读：事务A读取同一条记录两次，但是在两次之间事务B对该条记录进行了修改并提交，导致事务A两次读取的数据不一致。

它和脏读的区别是，脏读是事务A读取了另一个事务B未提交的脏数据，而不可重复读则是事务A读取了事务B提交的数据。

多数情况下，不可重复读并不是问题，因为我们多次查询某个数据时，当然要以最后查询得到的结果为主。但在另一些情况下就有可能发生问题，比如，老板让B和C分别核对事务A操作的数据，结果可能不同，老板是怀疑B呢，还是C呢?

幻读：事务A在修改全表的数据，比如将字段age全部修改为0岁，在未提交时，事务B向表中插入或删除数据，如插入一条age为25岁的数据。这样导致事务A读取的数据与需要修改的数据不一致，就和幻觉一样。

幻读和不可重复读的相同点：都是针对于另外一个已经提交的事务而言。不同点：不可重复读是针对于同一条记录来说的（delete或update 同一条记录），而幻读是针对于一批数据来说的（insert）

总结:

数据库通过设置事务的隔离级别防止以上情况的发生

4.1.1.6. 隔离机制

- 隔离机制分类

1、未提交读（read uncommitted）：就是不做隔离控制，可以读到“脏数据”，可能发生不可重复读，也可能出现幻读。

2、提交读（read committed）：提交读就是不允许读取事务没有提交的数据。显然这种级别可以避免了脏读问题。但是可能发生不可重复读，幻读。这个隔离级别是大多数数据库（除了mysql）的默认隔离级别。

3、可重复读（repeatable read）：为了避免提交读级别不可重复读的问题，在事务中对符合条件的记录上"排他锁"，这样其他事务不能对该事务操作的数据进行修改，可避免不可重复读的问题产生。由于只对操作数据进行上锁的操作，所以当其他事务插入或删除数据时，会出现幻读的问题，此种隔离级别为Mysql默认的隔离级别。

4、序列化（Serializable），在事务中对表上锁，这样在事务结束前，其他事务都不能够对表数据进行操作（包括新增，删除和修改），这样避免了脏读，不可重复读和幻读，是最安全的隔离级别。但是由于该操作是堵塞的，因此会严重影响性能。

- 注意点

1.oracle的隔离级别是read committed

2.mysql的隔离级别是repeatable read

3.级别越高，性能越低，数据越安全

4.设置隔离级别必须在事务之前

- mysql中隔离级别相关操作

查看当前的事务隔离级别：SELECT @@TX_ISOLATION;

更改当前的事务隔离级别：SET TRANSACTION ISOLATION LEVEL 四个级别之一。

- JDBC控制事务的隔离级别

Connection接口：

static int	**TRANSACTION_READ_COMMITTED** 指示防止发生脏读的常量；不可重复读和虚读有可能发生。
static int	**TRANSACTION_READ_UNCOMMITTED** 指示可以发生脏读 (dirty read)、不可重复读和虚读 (phantom read) 的常量。
static int	**TRANSACTION_REPEATABLE_READ** 指示防止发生脏读和不可重复读的常量；虚读有可能发生。
static int	**TRANSACTION_SERIALIZABLE** 指示防止发生脏读、不可重复读和虚读的常量。

设置方法:

Connection.setTransactionIsolation (int level);

4.1.2. 银行转账案例演示

4.1.2.1. 案例分析

```
1.需求：一个账号fromAccount向另一个账号toAccount转入money元钱
2.分析：
  - 检查两个账号是否存在，不存在的话，结束转账行为
  - 检查转出账号的里金额是否充足，不充足，结束转账行为,充足的话，进行扣款money元
  - 转入账号进行增加money元
```

4.1.2.2. 代码实现

```
/**
 * @Author 千锋大数据教学团队
 * @Company 千锋好程序员大数据
 * @Description 银行转账案例
 */
    /*
     * 事务:是逻辑上的一组操作,默认一个sql语句对应一个事务
     *
     * 一个事务中操作的特点:要么全成功,要么全不成功--保持一致
     */

public class AccountTest {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.println("请输入要出账的账号和进账的账号");
        String fromAccount = sc.next();
```

```
17          String toAccount = sc.next();
18          System.out.println("请输入转账金额：");
19          double money = sc.nextDouble();
20          boolean flag =
oneToOne(fromAccount,toAccount,money);
21          if(flag){
22              System.out.println("success");
23          }else{
24              System.out.println("fail");
25          }
26      }
27
28      /**
29       * 封装了两个人之间的转账逻辑
30       *  true:表示转账成功
31       *  false：表示转账失败
32       */
33      public static boolean oneToOne(String
fromAccount,String toAccount,double money){
34          /*第一步：先校验账户信息是否有效*/
35          if(fromAccount==null||fromAccount.length()==0)
{
36              return false;
37          }
38          if(toAccount==null||toAccount.length()==0){
39              return false;
40          }
41          if(money<=0){
42              return false;
43          }
44          Connection conn = null;
45          //预编译sql语句执行对象
46          PreparedStatement ps = null;
```

```
47          ResultSet rs = null;
48          try{
49              //验证转出账号是否存在
50              conn = DBUtil.getConnection();
51              String sql1 = "select * from bank_account
    where account_id=?";
52              ps = conn.prepareStatement(sql1);
53              ps.setString(1,fromAccount);
54              rs = ps.executeQuery();
55              boolean flag = rs.next();
56              if(!flag){
57                  System.out.println("出账账号不存在");
58                  return false;
59              }
60
61              //，如果存在，先提取出余额，保存到一个变量中
62              double balance =
    rs.getDouble("account_balance");
63
64              //验证进账账号是否存在
65              sql1 = "select * from bank_account where
    account_id=?";
66              ps = conn.prepareStatement(sql1);
67              ps.setString(1,toAccount);
68              rs = ps.executeQuery();
69              flag = rs.next();
70              if(!flag){
71                  System.out.println("进账账号不存在");
72                  return false;
73              }
74
75
```

```
76          /*第二步：检查formAccount账户内的余额，足够的前提下，修改余额*/
77                  //判断出账账号的余额是否充足
78                  if(money>balance){
79                      System.out.println("余额不足");
80                      return false;
81                  }
82
83                  //此时，来到此处后就可以进行转账业务了
84                  //先扣除出账账号的money
85                  String sql2 = "update bank_account set account_balance=account_balance-? where account_id = ?";
86                  ps = conn.prepareStatement(sql2);
87                  ps.setDouble(1,money);
88                  ps.setString(2,fromAccount);
89                  ps.executeUpdate();
90
91                  /*第三步：修改toAccount账户内的余额*/
92                  //进账账号再+money
93                  String sql3 = "update bank_account set account_balance=account_balance+? where account_id = ?";
94                  ps = conn.prepareStatement(sql3);
95                  ps.setDouble(1,money);
96                  ps.setString(2,toAccount);
97                  ps.executeUpdate();
98                  return true;
99              }catch (Exception e){
100                 e.printStackTrace();
101             }finally{
102                 DBUtil.closeConnection(conn,ps,rs);
103             }
```

```
104             return false;
105         }
106     }
```

4.1.3. 转账异常演示

```
在转出账户转出金额之后和转入账户收入金额之前模拟空指针异常

String str = null;
System.out.println(str.length());
```

4.1.4. 修改转账代码

改为手动提交

```
    /*
     * 事务:是逻辑上的一组操作,默认一个sql语句对应一个事务
     *
     * 一个事务中操作的特点:要么全成功,要么全不成功--保持一致
     */

public static boolean oneToOne(String fromAccount,
String toAccount, double money) {
    /*第一步：先校验账户信息是否有效*/
    if (fromAccount == null || fromAccount.length() ==
0) {
        return false;
    }
    if (toAccount == null || toAccount.length() == 0) {
        return false;
    }
    if (money <= 0) {
        return false;
```

```
17      }
18      Connection conn = null;
19      PreparedStatement ps = null;
20      ResultSet rs = null;
21      try {
22          conn = DBUtil.getConnection();
23
24        //在开启事务之前可以手动设置隔离级别--最高级别
25
connection.setTransactionIsolation(Connection.TRANSACTI
ON_SERIALIZABLE);
26
27          //取消事务自动提交功能，改为手动提交---将默认的自动提交
关闭,相当于开启了手动提交
28          conn.setAutoCommit(false);
29
30          String sql1 = "select * from bank_account where
account_id=?";
31          ps = conn.prepareStatement(sql1);
32          ps.setString(1, fromAccount);
33          rs = ps.executeQuery();
34          boolean flag = rs.next();
35          if (!flag) {
36              System.out.println("出账账号不存在");
37              return false;
38          }
39          /*第二步：检查formAccount账户内的余额，足够的前提下，
修改余额*/
40          //取出余额
41          double balance =
rs.getDouble("account_balance");
42
```

```
43          //sql1 = "select * from bank_account where
    account_id=?";
44          //ps = conn.prepareStatement(sql1);
45          ps.setString(1, toAccount);
46          rs = ps.executeQuery();
47          flag = rs.next();
48          if (!flag) {
49              System.out.println("进账账号不存在");
50              return false;
51          }
52
53          //判断出账账号的余额是否充足
54          if (money > balance) {
55              System.out.println("余额不足");
56              return false;
57          }
58
59          //此时，来到此处后就可以进行转账业务了
60          //先扣除出账账号的money
61          String sql2 = "update bank_account set
    account_balance=account_balance-? where account_id =
    ?";
62          ps = conn.prepareStatement(sql2);
63          ps.setDouble(1, money);
64          ps.setString(2, fromAccount);
65          ps.executeUpdate();
66
67          //模拟一个空指针异常
68          String str = null;
69          System.out.println(str.length());
70      /*第三步：修改toAccount账户内的余额*/
71          //进账账号再+money
```

```
72          String sql3 = "update bank_account set
    account_balance=account_balance+? where account_id =
    ?";
73          ps = conn.prepareStatement(sql3);
74          ps.setDouble(1, money);
75          ps.setString(2, toAccount);
76          ps.executeUpdate();
77
78          //手动提交事务
79          //提交事务-将借钱与收钱手动放在一个事务中
80          conn.commit();
81          return true;
82      } catch (Exception e) {
83          e.printStackTrace();
84          //当出现异常，我们才需要回滚事务
85          //当发生这个异常的时候,让事务回滚:让当前事务退回到开启事
    务之前,当前对事务的所有操作失效
86          try{
87              conn.rollback();
88          }catch (Exception e1){
89              e1.printStackTrace();
90          }
91      } finally {
92          DBUtil.closeConnection(conn, ps, rs);
93      }
94      return false;
95  }
```

4.2. 数据库连接池技术

4.2.1. 连接池技术出现的原因

在与数据库连接过程中，会非常消耗内存，性能大打折扣。如果每次请求都去重新连接数据库。那么，宕机的几率很高。

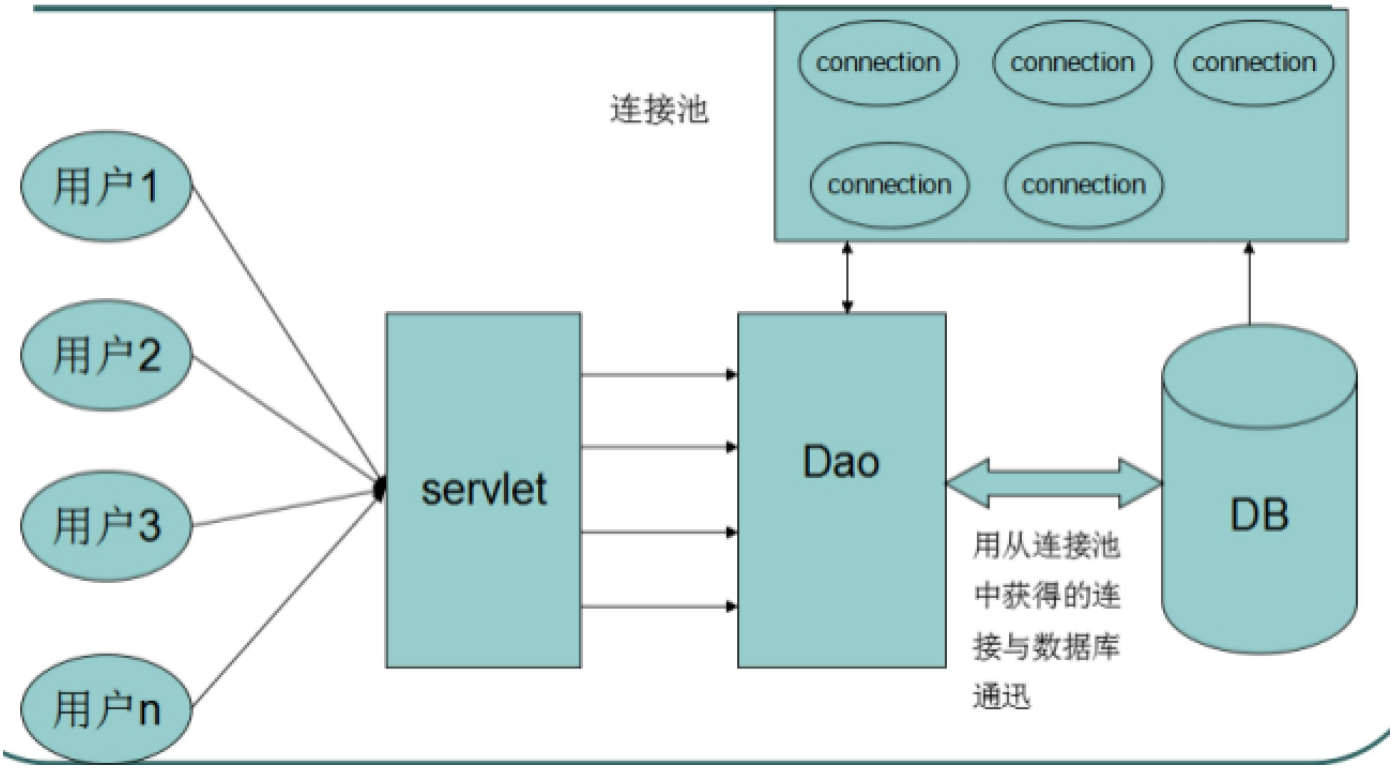
4.2.2. 连接池技术原理和优势(会)

- 原理：

  连接池对象在初始化阶段 一次性创建N个连接对象，这些连接对象存储在连接池对象中。当有请求过来时，先从连接池中寻找空闲连接对象并使用，当使用完后，将连接对象归还给连接池，而不是真正意义上断开连接。

- 优势：

  这样也可以满足成千上万个请求,解决建立数据库连接耗费资源和时间很多的问题，提高性能。



4.2.3. 编写标准的数据源(了解)

自定义数据库连接池要实现javax.sql.DataSource接口，一般都叫数据源。

4.2.3.1. 代码实现

```
public class MyDataSource implements DataSource{
    //存放连接的池子
    private static LinkedList<Connection> pool = new LinkedList<Connection>();
    //创建10个连接放在池中
        static{
            for (int i = 0; i < 10; i++) {
                Connection conn = null;
                try {
                    conn = JdbcUtil.getConnection();
                    pool.addLast(conn);
                } catch (SQLException e) {
                    e.printStackTrace();
                }
            }
        }


    public Connection getConnection() throws SQLException {
        if(pool.size()>0){
            Connection conn = pool.removeFirst();
            return conn;
        }else{
            //等待多长时间
```

4.2.3.2. 编写数据源时遇到的问题

描述:连接对象要放回池子,不能关闭.

```
public class TestDataSource {
    public static void main(String[] args) {
        Connection conn = null;
        PreparedStatement ps = null;
        DataSource ds = new MyDataSource();

        try{
            conn = ds.getConnection();
            ps = conn.prepareStatement("");
//          ...
        }catch(Exception e){
            e.printStackTrace();
        }finally{
            if(conn!=null){
                try {
                    conn.close();//关闭连接。不能关的。
                } catch (SQLException e) {
                    e.printStackTrace();
                }
            }
        }
```

4.2.3.3. 编写数据源时的解决办法

a.装饰设计模式：使用频率很高

目的：改写已存在的类的某个方法或某些方法，装饰设计模式（包装模式）

口诀：

1、编写一个类，实现与被包装类相同的接口。（具备相同的行为）

2、定义一个被包装类类型的变量。

3、定义构造方法，把被包装类的对象注入，给被包装类变量赋值。

4、对于不需要改写的方法，调用原有的方法。

5、对于需要改写的方法，写自己的代码。

```
//4、对于不需要改写的方法，调用原有的方法。
//5、对于需要改写的方法，写自己的代码。
public class MyConnection implements Connection {

    private Connection oldConn;
    LinkedList<Connection> pool;
    public MyConnection(Connection oldConn,LinkedList<Connection> pool){
        this.oldConn = oldConn;
        this.pool = pool;
    }

    public void close() throws SQLException {
        pool.add(oldConn);
    }

    public Statement createStatement() throws SQLException {
        return oldConn.createStatement();
    }

    @Override
    public <T> T unwrap(Class<T> iface) throws SQLException {
        return oldConn.unwrap(iface);
    }
```

b.默认适配器：装饰设计模式一个变体

```java
//本身就是一个装饰类，对原类没有任何改变。
//1、编写一个类，实现与被包装类相同的接口。（具备相同的行为）
//2、定义一个被包装类类型的变量。
//3、定义构造方法，把被包装类的对象注入，给被包装类变量赋值。
//4、对于不需要改写的方法，调用原有的方法。
public class ConnectionWarper implements Connection {

    private Connection oldConn;
    public ConnectionWarper(Connection oldConn){
        this.oldConn = oldConn;
    }
    @Override
    public <T> T unwrap(Class<T> iface) throws SQLException {
        return oldConn.unwrap(iface);
    }

    @Override
    public boolean isWrapperFor(Class<?> iface) throws SQLException {
        return oldConn.isWrapperFor(iface);
    }
```

```java
//1、编写一个类，继承包装类适配器。（具备相同的行为）
//2、定义一个被包装类类型的变量。
//3、定义构造方法，把被包装类的对象注入，给被包装类变量赋值。
//4、对于不需要改写的方法，调用原有的方法。
public class MyConnection1 extends ConnectionWarper{

/*  public MyConnection1(){
        super();
    }*/
    private Connection conn;
    public MyConnection1(Connection conn){
        super(conn);
        this.conn = conn;
    }
    public void close() throws SQLException {
//      ... 写自己的代码
    }
```

4.2.4. 常用的连接池技术

```
- dbcp     :是apache组织旗下的一个数据库连接池技术产品
- c3p0     :是一个开源的连接池技术
- druid    :是阿里的数据库连接池技术
```

4.2.5. dbcp(会)

4.2.5.1. 资源jar包

```
commons-dbcp2-2.6.0.jar
commons-pool2-2.4.3.jar
commons-logging.jar
```

4.2.5.2. 配置文件dbcp.properties

此配置文件请放在src目录下

```
driver=com.mysql.jdbc.Driver
url=jdbc:mysql://localhost:3306/qianfeng
username=root
pwd=123456
maxTotal=50
maxIdle=10
minIdle=3
initialSize=5
maxWaitMillis=60000
```

4.2.5.3. DBUtildbcp类型的编写

```
import org.apache.commons.dbcp2.BasicDataSource;

import java.io.InputStream;
import java.sql.Connection;
```

```java
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import java.util.Properties;

/**
 * @Author 千锋大数据教学团队
 * @Company 千锋好程序员大数据
 * @Description 使用DBCP连接池
 */
public class DBUtildbcp {
    private static String driver;
    private static String url;
    private static String username;
    private static String password;
    private static int maxTotal;
    private static int maxIdle;
    private static int minIdle;
    private static int initialSize;
    private static long maxWaitMillis;

    //声明一个dbcp连接池变量
    private static BasicDataSource pool;
    static{
        try{
            pool = new BasicDataSource();//连接池对象

            //使用类加载器中提供的方法来获取字节流对象，同时指定配置文件
            InputStream is = DBUtildbcp.class.getClassLoader()

.getResourceAsStream("dbcp.properties");
```

```
35              Properties prop = new Properties();
36              prop.load(is);//将配置文件里的内容封装到prop对象
    内
37
38              driver = prop.getProperty("driver");
39              url = prop.getProperty("url");
40              username = prop.getProperty("username");
41              password = prop.getProperty("pwd");
42              maxTotal=
    Integer.parseInt(prop.getProperty("maxTotal"));
43              maxIdle=
    Integer.parseInt(prop.getProperty("maxIdle"));
44              minIdle=
    Integer.parseInt(prop.getProperty("minIdle"));
45              initialSize=
    Integer.parseInt(prop.getProperty("initialSize"));
46              maxWaitMillis=
    Long.parseLong(prop.getProperty("maxWaitMillis"));
47
48              pool.setDriverClassName(driver);
49              pool.setUrl(url);
50              pool.setUsername(username);
51              pool.setPassword(password);
52              //连接池支持的最大连接数
53              pool.setMaxTotal(maxTotal);
54              //连接池支持的最大空闲数
55              pool.setMaxIdle(maxIdle);
56              //支持的最小空闲数
57              pool.setMinIdle(minIdle);
58              //连接池对象创建时初始化的连接数
59              pool.setInitialSize(initialSize);
60              //空闲等待时间
61              pool.setMaxWaitMillis(maxWaitMillis);
```

```
62              //注册驱动
63              Class.forName(driver);
64
65          }catch (Exception e){
66              e.printStackTrace();
67          }
68      }
69      public static Connection getConnection() throws
SQLException {
70          //从连接池中获取空闲对象
71          return pool.getConnection();
72      }
73      public static void closeConnection(Connection conn,
Statement stat, ResultSet rs){
74          try {
75              if(rs!=null){
76                  rs.close();
77              }
78              if(stat!=null){
79                  stat.close();
80              }
81              if(conn !=null){
82                  conn.close();    //会将连接对象归还给连接池
内
83              }
84          } catch (SQLException e) {
85              e.printStackTrace();
86          }
87      }
88
89      public static void main(String[] args) throws
SQLException {
90          Connection conn = getConnection();
```

```
91          System.out.println(conn);
92          conn.close();
93      }
94  }
```

4.2.6. c3p0(会)

4.2.6.1. 资源jar包

```
1  c3p0-0.9.5-pre8.jar
2  mchange-commons-java-0.2.7.jar
```

4.2.6.2. 配置文件c3p0-config.xml

配置文件请放在src目录下

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <c3p0-config>
3      <default-config>
4          <property name="user">root</property>
5          <property name="password">123456</property>
6          <property
   name="jdbcUrl">jdbc:mysql://localhost:3306/bd1901</prop
   erty>
7          <property
   name="driverClass">com.mysql.jdbc.Driver</property>
8          <property name="acquireIncrement">10</property>
9          <property name="maxPoolSize">50</property>
10         <property name="minPoolSize">2</property>
11         <property name="initialPoolSize">5</property>
12         <property name="maxIdleTime">600</property>
13     </default-config>
14 </c3p0-config>
15
```

4.2.6.3. DBUtilc3p0类型的编写

```
import com.mchange.v2.c3p0.ComboPooledDataSource;
import java.sql.Connection;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;

/**
 * @Author 千锋大数据教学团队
 * @Company 千锋好程序员大数据
 * @Description 使用c3p0连接池
 */
public class DBUtilC3p0 {
    //构造器会自动检索src下有没有指定文件名称的配置文件，然后会自动赋值给其相应的属性
    private static ComboPooledDataSource pool = new ComboPooledDataSource("c3p0-config");

    public static Connection getConnection() throws SQLException {
        //从连接池中获取空闲对象
        return pool.getConnection();
    }
    public static void closeConnection(Connection conn, Statement stat, ResultSet rs){
        try {
            if(rs!=null){
                rs.close();
            }
            if(stat!=null){
                stat.close();
            }
```

```
            if(conn !=null){
                conn.close();    //会将连接对象归还给连接池内
            }
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }

    public static void main(String[] args) throws SQLException {
        Connection conn = getConnection();
        System.out.println(conn);
        conn.close();

    }
}
```

4.2.7. druid(会)

4.2.7.1. 资源jar包

```
druid-1.1.18.jar
```

4.2.7.2. 配置文件druid.properties

放在src目录下。注意，前面的key值是固定写法

```
driverClassName=com.mysql.jdbc.Driver
url=jdbc:mysql://localhost:3306/qianfeng
username=root
password=123456
maxActive=20
minIdle=3
initialSize=5
maxWait=60000
```

4.2.7.3. DBUtildruid类型的编写

```
import java.io.InputStream;
import java.sql.Connection;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import java.util.Properties;
import javax.sql.DataSource;
import com.alibaba.druid.pool.DruidDataSourceFactory;

/**
 * @Author 千锋大数据教学团队
 * @Company 千锋好程序员大数据
 * @Description 使用Druid连接池
 */
public class DBUtil_druid {
    //创建连接池对象
    private static DataSource pool = null;
    static {
        try {
            //使用类加载器提供的方法读取db.properties,返回一个字节流对象
```

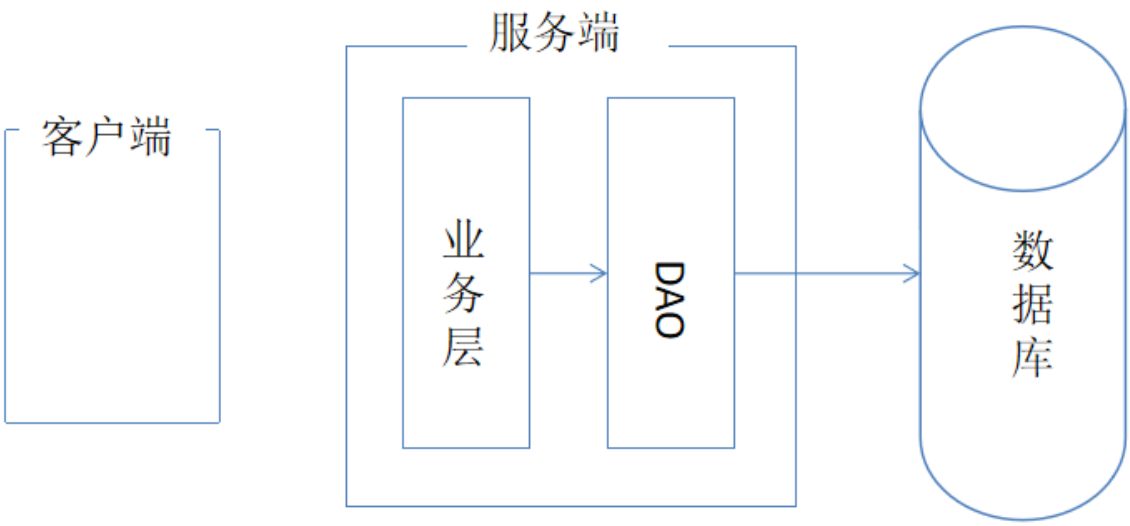
```
            InputStream is =
DBUtil_druid.class.getClassLoader().

 getResourceAsStream("druid.properties");
            //创建Properties对象，用于加载流内部的数据
            Properties prop = new Properties();
            prop.load(is); //加载流内部的信息，以key-value
的形式进行加载
            //调用静态方法，会自动给自己的属性赋值
            pool =
DruidDataSourceFactory.createDataSource(prop);
        } catch (Exception e) {
            System.out.println("注册驱动失败");
            e.printStackTrace();
        }
    }
    /**
     * 获取连接对象
     *
     * @return 连接对象
     * @throws SQLException
     * @throws ClassNotFoundException
     */
    public static Connection getConnection() throws
SQLException,
    ClassNotFoundException {

        //return DriverManager.getConnection(url,
username, password);
        //从连接池中获取连接对象
        return pool.getConnection();
    }

```

```
48      /**
49       * 关闭数据库连接
50       *
51       * @param rs   结果集对象
52       * @param stat 处理sql的执行对象Statement
53       * @param conn 连接对象
54       */
55      public static void closeConnection(ResultSet rs,
    Statement stat, Connection conn) {
56          try {
57              if (rs != null) {
58                  rs.close();
59              }
60              if (stat != null) {
61                  stat.close();
62              }
63              if (conn != null) {
64                  conn.close();//释放连接，归还给连接池
65              }
66          } catch (Exception e) {
67              System.out.println("数据库连接关闭失败");
68              e.printStackTrace();
69          }
70      }
71
72      public static void main(String[] args) throws
    ClassNotFoundException, SQLException {
73          Connection conn = getConnection();
74          System.out.println(conn);
75          closeConnection(null, null, conn);
76      }
77  }
```

4.3. DAO设计模式(会)

4.3.1. DAO简介

```
- DAO是数据访问对象(Data Access Object)的简写。
- 建立在数据库与业务层之间，封装所有对数据库的访问操作，我们也可称之为持久层。
- 目的：将数据访问逻辑和业务逻辑分开。
```



一个DAO设计模式包含以下内容

1. 定义实体类: 通过对象关系映射(ORM)将数据库的表结构映射成java类型;表中的每一条记录映射成类的实例。用于数据的传递。
2. 定义一个接口： 在此接口中，定义应用程序对此表的所有访问操作，如增，删，改、查，等方法。
3. 定义接口的实现类 实现接口中的所有抽象方法。
4. 定义一个DAO工厂类型 用于返回接口实例 这样，开发人员只需要使用DAO接口即可，具体逻辑就变得透明了，无需了解内部细节。

4.3.2. DAO的案例示范

4.3.2.1. 创建项目，导入相关资源

4.3.2.2. 编写工具类DBUtil

4.3.2.3. 编写实体类

```
import java.sql.Date;
import java.util.Objects;

/**
 * @Author 千锋大数据教学团队
 * @Company 千锋好程序员大数据
 * @Description 以orm关系将数据库中的emp表映射成java中的Emp类型表的字段映射成类的属性
 */
public class Emp {
    private int empno;
    private String ename;
    private String job;
    private int mgr;
    private Date hiredate;
    private double salary;
    private double comm;
    private int deptno;
    public Emp(){}

    public Emp(int empno, String ename, String job, int mgr, Date hiredate, double salary, double comm, int deptno) {
        this.empno = empno;
        this.ename = ename;
        this.job = job;
```

```
24          this.mgr = mgr;
25          this.hiredate = hiredate;
26          this.salary = salary;
27          this.comm = comm;
28          this.deptno = deptno;
29      }
30
31      public int getEmpno() {
32          return empno;
33      }
34
35      public void setEmpno(int empno) {
36          this.empno = empno;
37      }
38
39      public String getEname() {
40          return ename;
41      }
42
43      public void setEname(String ename) {
44          this.ename = ename;
45      }
46
47      public String getJob() {
48          return job;
49      }
50
51      public void setJob(String job) {
52          this.job = job;
53      }
54
55      public int getMgr() {
56          return mgr;
```

```
57      }
58
59      public void setMgr(int mgr) {
60          this.mgr = mgr;
61      }
62
63      public Date getHiredate() {
64          return hiredate;
65      }
66
67      public void setHiredate(Date hiredate) {
68          this.hiredate = hiredate;
69      }
70
71      public double getSalary() {
72          return salary;
73      }
74
75      public void setSalary(double salary) {
76          this.salary = salary;
77      }
78
79      public double getComm() {
80          return comm;
81      }
82
83      public void setComm(double comm) {
84          this.comm = comm;
85      }
86
87      public int getDeptno() {
88          return deptno;
89      }
```

```
90
91      public void setDeptno(int deptno) {
92          this.deptno = deptno;
93      }
94
95      @Override
96      public String toString() {
97          return "Emp{" +
98                  "empno=" + empno +
99                  ", ename='" + ename + '\'' +
100                 ", job='" + job + '\'' +
101                 ", mgr=" + mgr +
102                 ", hiredate=" + hiredate +
103                 ", salary=" + salary +
104                 ", comm=" + comm +
105                 ", deptno=" + deptno +
106                 '}';
107     }
108 }
```

4.3.3.4. 定义接口

```
1  import com.qianfeng.jdbc03.entity.Emp;
2  import java.util.List;
3
4  /**
5   * @Author 千锋大数据教学团队
6   * @Company 千锋好程序员大数据
7   * @Description 设计针对于实体类Emp和数据库里的emp表设计对数据
   库操作的接口提供相应操作的抽象方法
8   */
9  public interface EmpDao {
10     /**
```

```
11    * 提供向数据库中插入数据的方法,
12    * @param e    面向对象思想可以使用实体类的实例
13    */
14      void addEmp(Emp e);
15      /**
16    * 提供删除数据库内的一条记录方法，通过id进行删除
17    * @param empno    数据库表中的主键
18    */
19      void deleteById(int empno);
20
21      /**
22    * 修改方法。
23    * @param e 传入前先设置成要修改的数据，然后传入方法中进行
      update语句赋值
24    */
25      void modifyEmp(Emp e);
26
27      /**
28    * 通过唯一键查询一条记录
29    * @param empno
30    * @return  封装成实体类实例
31     */
32      Emp findById(int empno);
33
34      /**
35    * 查询所有的记录。
36    * @return  封装成类的实例，并存入集合
37    */
38      List<Emp> findAll();
39
40      /**
41    * 分页查询
42    * @param page    要查询的页数
```

```
43      * @param pageSize  每页显示的条数
44      * @return  一页的所有记录，封装到集合中
45      */
46        List<Emp> findByPage(int page,int pageSize);
47  }
48
```

4.3.3.5. 编写实现类

```
1   /**
2    * @Author 千锋大数据教学团队
3    * @Company 千锋好程序员大数据
4    * @Description EmpDao接口的实现类
5    */
6   public class EmpDaoImpl implements EmpDao {
7       @Override
8       public void addEmp(Emp e) {
9           Connection conn = null;
10          PreparedStatement ps = null;
11          try{
12              conn = DBUtil.getConnection();
13              String sql = "insert into emp values
    (?,?,?,?,?,?,?,?)";
14              ps = conn.prepareStatement(sql);
15              ps.setInt(1,e.getEmpno());
16              ps.setString(2,e.getEname());
17              ps.setString(3,e.getJob());
18              ps.setInt(4,e.getMgr());
19              ps.setDate(5,e.getHiredate());
20              ps.setDouble(6,e.getSalary());
21              ps.setDouble(7,e.getComm());
22              ps.setInt(8,e.getDeptno());
23              ps.executeUpdate();
```

```
24
25          }catch (Exception e1){
26              e1.printStackTrace();
27          }finally{
28              DBUtil.closeConnection(conn,ps,null);
29          }
30      }
31
32      @Override
33      public void deleteById(int empno) {
34
35      }
36
37      @Override
38      public void modifyEmp(Emp e) {
39
40      }
41
42      @Override
43      public Emp findById(int empno) {
44          return null;
45      }
46
47      @Override
48      public List<Emp> findAll() {
49          return null;
50      }
51
52      @Override
53      //limit 0,5
54      public List<Emp> findByPage(int page, int pageSize)
    {
55          Connection conn = null;
```

```
56          PreparedStatement ps = null;
57          ResultSet rs = null;
58          List<Emp> emps = new ArrayList<Emp>();
59          try{
60              conn = DBUtil.getConnection();
61              String sql = "select * from emp order by
empno limit ?,?";
62              ps = conn.prepareStatement(sql);
63              ps.setInt(1,(page-1)*pageSize);
64              ps.setInt(2,pageSize);
65
66              rs = ps.executeQuery();
67              Emp e = null;
68              while(rs.next()){
69                  int empno = rs.getInt(1);
70                  String ename = rs.getString(2);
71                  String job = rs.getString("job");
72                  int mgr = rs.getInt("mgr");
73                  Date hiredate = rs.getDate("hiredate");
74                  double salary = rs.getDouble("sal");
75                  double comm = rs.getDouble("comm");
76                  int deptno = rs.getInt("deptno");
77                  e = new
Emp(empno,ename,job,mgr,hiredate,salary,comm,deptno);
78                  emps.add(e);
79              }
80          }catch (Exception e1){
81              e1.printStackTrace();
82          }finally{
83              DBUtil.closeConnection(conn,ps,rs);
84          }
85          return emps;
86      }
```

```
}
```

4.3.3.6. 编写DAO单例类

```
/**
 * @Author 千锋大数据教学团队
 * @Company 千锋好程序员大数据
 * @Description Dao单例类
 */
public class DaoFactory{
  //定义属性EmpDao属性
    private static EmpDao empdao = new EmpDaoImpl();
    //让构造函数为 private，这样该类就不会被实例化
    private DaoFactory(){}
    public  static EmpDao getInstance(){
        return empdao;
    }
}
```

4.3.3.7. 编写测试类

```
/**
 * @Author 千锋大数据教学团队
 * @Company 千锋好程序员大数据
 * @Description Dao测试类
 */
public class TestDao {
    @Test
    public void testAddEmp(){
        EmpDao dao = DaoFactory.getInstance();
        Emp e = new Emp(9007,"huanghua","manager",7369,
                        Date.valueOf("2019-1-
1"),3000.0,200.0,20);
```

```
        dao.addEmp(e);
    }
    @Test
    public void testFindByPage(){
        EmpDao dao = DaoFactory.getInstance();
        List<Emp> emps = dao.findByPage(3,5);
        for(Emp e:emps){
            System.out.println(e);
        }
    }
}

```

4.4. dbutils第三方工具类的使用

4.4.1. 简介

```
- 作用:
DBUtils是java编程中的数据库操作实用工具，小巧简单实用。
DBUtils封装了DAO层(持久层)的逻辑。减少了开发周期。
1.对于数据表的读操作，他可以把结果转换成List，Array，Set等java集合，便于程序员操作;
2.对于数据表的写操作，也变得很简单（只需写sql语句）
3.可以使用数据源，使用JNDI，数据库连接池等技术来优化性能--重用已经构建好的数据库连接对象

- jar包: commons-dbutils-1.7.jar

- 常用API:
1. QueryRunner类型:可以直接使用连接池技术来操作数据库，进行增删改查
   构造器: QueryRunner(DataSource ds)
        返回一个指定数据库连接池得QueryRunner对象
```

```
   非静态方法：query(String sql, ResultSetHandler<T> rsh)
      通过sql，及其ReusltSetHandler的子类型来获取数据并封装成相应对象
  它主要有三个方法
query() 用于执行select
update() 用于执行insert update delete
batch() 批处理
2. ResultSetHandler：关于结果集的一个接口。
  用于定义select操作后，怎样封装结果集.

  其实现类举例如下：
  BeanHandler:将查询到的数据的第一条封装成实体类对象
  BeanListHandler:将查询到的数据的第一条封装成实体类对象的集合

```

4.4.2. 增删改代码测试(会)

```
package com.qianfeng.test;

import java.sql.Connection;
import java.sql.Date;
import java.sql.PreparedStatement;
import java.sql.SQLException;

import org.apache.commons.dbutils.QueryRunner;
import org.junit.Test;

import com.qianfeng.util.C3P0Util;

/*
 * @Test:他下面的方法是测试的方法,这个方法可以直接被调用,但是必须是在测试的环境下
 * 注意点:1.方法不能有参数   2.不能有返回值
```

```
 */
public class TestDemo1 {
  //@Test
  //没有使用DBUtils时的代码实现
  public void test1(){
    Connection connection = null;
    PreparedStatement pStatement = null;
    int num = 0;
    try {
      connection = C3P0Util.getConnection();
      String sql = "insert into user(id,name,password) values(?,?,?)";
      pStatement = connection.prepareStatement(sql);

      pStatement.setInt(1, 4);
      pStatement.setString(2, "王五");
      pStatement.setString(3, "123");

      num = pStatement.executeUpdate();
    } catch (SQLException e) {
      // TODO Auto-generated catch block
      e.printStackTrace();
    }finally {
      C3P0Util.release(connection, pStatement, null);
    }
  }

  //使用DBUtis后实现插入
  //@Test
  public void  test2() throws SQLException{
    //写sql语句
    String sql = "insert into user(id,name,password) values(?,?,?)";
```

```
47
48      //第一种:使用的是无参的QueryRunner()方法----为了更加方便的使用事务,因为可以直接获取到Connection对象
49      //1.创建干活的的对象--QueryRunner
50      QueryRunner qRunner = new QueryRunner();
51      //2.获取连接对象
52      Connection connection = C3P0Util.getConnection();
53      //3.调用update()方法实现对数据库的访问
54      int num = qRunner.update(connection, sql,6,"马六","345");
55      if (num >0) {
56        System.out.println("增加成功");
57      }else {
58        System.out.println("增加失败");
59      }
60
61      //第二种:使用的是有参的QueryRunner()方法--参数是数据源
62      //1.创建干活儿的对象并绑定数据源
63  //    QueryRunner qRunner2 = new QueryRunner(C3P0Util.getDataSource());
64  //    //2.调用update()方法
65  //    int num2 = qRunner2.update(sql,7,"马六1","3451");
66  //    if (num2 >0) {
67  //      System.out.println("增加成功");
68  //    }else {
69  //      System.out.println("增加失败");
70  //    }
71    }
72
73    //批量增加--只能同时执行一种操作
74    @Test
75    public void test3() throws SQLException{
```

```
76      QueryRunner qRunner2 = new
    QueryRunner(C3P0Util.getDataSource());
77
78      //创建一个二维数组装 数据
79      Object[][] params = new Object[3][];
80      for (int i = 0; i < params.length; i++) {
81        params[i] = new Object[]
    {i+8,"bingbing"+i,"333"+i};
82      }
83
84      //写sql语句
85        String sql = "insert into user(id,name,password)
    values(?,?,?)";
86      int[] nums = qRunner2.batch(sql, params);
87    }
88  }
```

4.4.3. 查找功能实现

4.4.3.1. 直接使用ResultSetHandler接口

```
1   /*
2    * ResultSetHandler接口
3    */
4   @SuppressWarnings("unchecked")
5   //@Test
6   //获取全部的数据
7   public void test1() throws SQLException{
8     QueryRunner qRunner2 = new
    QueryRunner(C3P0Util.getDataSource());
9
10    ResultSetHandler handler =
11    new ResultSetHandler<List<User>>() {
12
```

```
13          /*
14           * 这是真正处理数据的方法
15           * 参数就是得到的结果集
16           * 这个方法会被自动调用
17           */
18          @Override
19          public List<User> handle(ResultSet set) throws
    SQLException {
20            List<User> list = new ArrayList<User>();
21
22            while (set.next()) {
23              User user = new User();
24              user.setName(set.getString("name"));
25              user.setPassword(set.getString("password"));
26
27              list.add(user);
28            }
29
30            return list;
31          }
32        };
33
34        List<User> list =
    (List<User>)qRunner2.query("select * from user",
    handler);
35
36        System.out.println(list);
37      }
```

我们发现通过实现ResultSetHandler接口,可以以各种形式获取数据库的数据,所以系统就封装了一批ResultSetHandler的子类实现各种功能.

4.4.3.2. ResultSetHandler下的所有结果处理器(子类)

注意:以下的子类中,重点掌握BeanHandler和BeanListHandler的功能实现

ArrayHandler:适合取1条记录。把该条记录的每列值封装到一个数组中Object[]

ArrayListHandler:适合取多条记录。把每条记录的每列值封装到一个数组中Object[]，把数组封装到一个List中

ColumnListHandler:取某一列的数据。封装到List中。

KeyedHandler:取多条记录，每一条记录封装到一个Map中，再把这个Map封装到另外一个Map中，key为指定的字段值。

MapHandler:适合取1条记录。把当前记录的列名和列值放到一个Map中

MapListHandler:适合取多条记录。把每条记录封装到一个Map中，再把Map封装到List中

ScalarHandler:适合取单行单列数据

BeanHandler:返回一条记录-得到的是模型----重点掌握

BeanListHandler:返回所有的数据--将记录装入模型,将模型装入集合(list)---重点掌握

4.4.3.3. 子类功能代码实现

```
1 import java.sql.ResultSet;
2 import java.sql.SQLException;
3 import java.util.ArrayList;
4 import java.util.List;
5 import java.util.Map;
6
```

```
7
8  import org.apache.commons.dbutils.QueryRunner;
9  import org.apache.commons.dbutils.ResultSetHandler;
10 import
   org.apache.commons.dbutils.handlers.ArrayHandler;
11 import
   org.apache.commons.dbutils.handlers.ArrayListHandler;
12 import
   org.apache.commons.dbutils.handlers.BeanHandler;
13 import
   org.apache.commons.dbutils.handlers.BeanListHandler;
14 import
   org.apache.commons.dbutils.handlers.ColumnListHandler;
15 import
   org.apache.commons.dbutils.handlers.KeyedHandler;
16 import org.apache.commons.dbutils.handlers.MapHandler;
17 import
   org.apache.commons.dbutils.handlers.MapListHandler;
18 import
   org.apache.commons.dbutils.handlers.ScalarHandler;
19 import org.junit.Test;
20
21 import com.qianfeng.domain.User;
22 import com.qianfeng.util.C3P0Util;
23
24 public class TestDemo2 {
25   //@Test
26   //返回所有的数据--将记录装入模型,将模型装入list
27
28   public void test2() throws SQLException{
29     QueryRunner qRunner2 = new
   QueryRunner(C3P0Util.getDataSource());
```

```
                                                    //<对应模型的类>(模型的字节码文件)
        List<User> list = qRunner2.query("select * from user", new BeanListHandler<User>(User.class));
        System.out.println(list);
    }

    @Test
    //返回一条记录-得到的是模型
    public void test3() throws SQLException{
        QueryRunner qRunner2 = new QueryRunner(C3P0Util.getDataSource());
        User user = qRunner2.query("select * from user where id=?", new BeanHandler<User>(User.class),1);
        System.out.println(user);
    }

    //取一条记录
    //ArrayHandler:只能返回一个记录,并且没有存储到javabean中,默认返回的是第一个记录
    //@Test
    public void test13() throws SQLException{
        QueryRunner qRunner = new QueryRunner(C3P0Util.getDataSource());
        //
        Object[] objects = qRunner.query("select * from user", new ArrayHandler());

        for (Object object : objects) {
            System.out.println(object);
        }
    }

```

```
56
57    //取多条记录
58    //ArrayListHandler:可以返回查到的多行记录,将一条记录的每个字段存储到一个数组中,再将这些数组放入一个集合中,并返回
59    //@Test
60    public void test4() throws SQLException{
61
62      QueryRunner qRunner = new QueryRunner(C3P0Util.getDataSource());
63
64      List<Object[]> list = qRunner.query("select * from user", new ArrayListHandler());
65
66      for (Object[] objects : list) {
67        for (Object object : objects) {
68          System.out.print(object+"   ");
69        }
70        System.out.println();
71      }
72
73    }
74
75    //ColumnListHandler:根据指定的列数,取某一列的数据,封装到list中返回
76    //@Test
77    public void test5() throws SQLException{
78      QueryRunner qRunner = new QueryRunner(C3P0Util.getDataSource());
79      //参数的意思是:指定查询的列号---从1开始计数
80      List<Object> list = qRunner.query("select * from user", new ColumnListHandler(2));
81
82      for (Object object : list) {
```

```
83        System.out.print(object+"   ");
84      }
85    }
86
87    //KeyedHandler:可以取多条记录,每一条记录被封装在了map中,然后再将所有的map对象放在一个大的map中,让每条记录中的某个字段充当外层map的key
88    //注意:如果充当key的字段有重复,和面的记录会将前面所有的记录覆盖
89    //@Test
90    public void test6() throws SQLException{
91      QueryRunner qRunner = new QueryRunner(C3P0Util.getDataSource());
92
93      Map<Object, Map<String, Object>> map=qRunner.query("select * from user", new KeyedHandler(5));
94
95      for(Map.Entry<Object, Map<String, Object>> en:map.entrySet()){
96        Object key = en.getKey();
97        Map<String, Object> subMap = en.getValue();
98
99        for(Map.Entry<String, Object> suben:subMap.entrySet()){
100         String subkey = suben.getKey();
101         Object value = suben.getValue();
102
103         System.out.print(subkey+"="+value+"    ");
104       }
105       System.out.println("  key="+key);
106     }
107   }
```

```
108
109   //只能取一条记录
110   //MapHandler:将一条记录的字段和对应的值封装到一个map中,返回,默认取的是第一行
111   //@Test
112   public void test7() throws SQLException{
113     QueryRunner qRunner = new QueryRunner(C3P0Util.getDataSource());
114
115     Map<String, Object> map = qRunner.query("select * from user", new MapHandler());
116
117     for(String key:map.keySet()){
118       System.out.print(key+"="+map.get(key)+"    ");
119     }
120   }
121
122
123   //MapHandler:将一条记录的字段和对应的值封装到一个map中,再将所有的map放到一个list中并返回
124   //@Test
125   public void test8() throws SQLException{
126     QueryRunner qRunner = new QueryRunner(C3P0Util.getDataSource());
127     List<Map<String, Object>> list = qRunner.query("select * from user", new MapListHandler());
128
129     for (Map<String, Object> map : list) {
130       for(String key:map.keySet()){
131         System.out.print(key+"="+map.get(key)+"    ");
132       }
133
```

```
134         System.out.println();
135       }
136     }
137
138     //取第一行的某一列的值
139     //ScalarHandler:适合取单行单列的数据
140     @Test
141     public void test9() throws SQLException{
142       QueryRunner qRunner = new
    QueryRunner(C3P0Util.getDataSource());
143       //参数:代表第一行对应的行号,取这个列下的数据,若果不写,默认
    取第一列
144       //Object object = qRunner.query("select * from
    user", new ScalarHandler(3));
145       //与聚合函数联合使用,参数可以取的列是select后面可以获取的
    列数
146       Object object1 = qRunner.query("select *,count(*)
    from user", new ScalarHandler(6));
147       System.out.println(object1);
148     }
149 }
150
```

4.5. xml与json(了解)

4.5.1. XML

4.5.1.1. 简介

eXtensible Markup Language 可扩展标记语言，里面的标记都是用户自定义的，标记都是成对的。作用是用来存储数据和传输数据。特别适合网络传输。

4.5.1.2. 语法

```
* 文档声明:
  * 必须写在xml文档的第一行。
  * 写法: <?xml version="1.0" ?>
  * 属性:
    * version: 版本号 固定值 1.0
    * encoding:指定文档的码表。默认值为 iso-8859-1
    * standalone: 指定文档是否独立  yes 或 no

    * 元素: xml文档中的标签
   ** 文档中必须有且只能有一个根元素
   * 元素需要正确闭合。<body></body> <br/>
   * 元素需要正确嵌套
   * 元素名称要遵守:
     * 元素名称区分大小写
     * 数字不能开头

* 文本:
    - 特殊符号:
        < : &lt;
        > : &gt;
        & : &amp;
        " : &quot;
        ' : &apos;
   * CDATA: 里边的数据会原样显示
   *  <![CDATA[ 数据内容 ]]>

* 属性:
```

```
    * 属性值必须用引号引起来。单双引号都行
    * 注释：
  <!--    注释内容    -->
* 处理指令：现在基本不用
  <?xml-stylesheet type="text/css" href="1.css"?>
```

- 简单案例

```
<?xml version="1.0" encoding="UTF-8"?>

<书架>
  <书 出版社=呵呵>
    <书名>金瓶梅</书名>
    <作者>陈冠希</作者>
    <单价>10</单价>
    <批发价>20</批发价>
  </书>
  <书>
    <书名>葵花宝典</书名>
    <作者>东方不败</作者>
    <单价>10</单价>
  </书>

</书架>
```

4.5.1.3. 功能

- 数据存储

  与文件对比

- 配置文件

作为配置文件存在，xml中主要配置的一些具有复杂的层级关系的数据

Properties文件中主要配置的一些key和value这样的数据。

- 数据传输

大部分使用json

json格式:{user:[{},{}],address:千锋}

{}代表map []代表数组

- 数据显示

可以使用html和xml,html主要用于网页显示.

4.5.1.4. 与html的区别

- xml是可扩展的标记语言，html是超文本标记语言
- xml里的标记都是用户自定义的，html都是预定义的
- xml用于存储和传输，html用于显示数据
- html语法松散，xml语法严格

4.5.1.5. 约束

1.约束就是xml的书写规则

2.约束的分类

dtd(Document Type Definition)约束

schema约束

3.dtd分类(简单约束)

- 内部dtd：在xml内部定义dtd
- 外部dtd：在外部文件中定义dtd
    - 本地dtd文件：

本地dtd文件的编写规则介绍:以student.dtd为例 1. 构成:,students是根节点,括号中是父节点的直接子节点 2.举例:正则表达式中(student*)表示根节点的直接子节点必须都是student,并且可以出现0次或多次(name,age,sex) 表示student的元素是他们 (#PCDATA)表示当前节点是叶子节点 3. !ATTLIST表示属性设置,number ID:表示每个student节点的ID号,必须是唯一的 #REQUIRED:表示ID号必须有 4.对应的xml中的设置:,将这个设置放在文档声明的下面,表示当前文档遵守dtd的约束 ; 每个student节点的属性中都要标明 ID号, 如:number="s0001",且每个ID号都不一样.

    - 网络dtd文件:

- 简单案例

原始文件 (student.xml)

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE students SYSTEM  "student.dtd">

<students>
  <student number="s0001">
    <name>zs</name>
    <age>abc</age>
    <sex>yao</sex>
  </student>
</students>
```

dtd约束文件(student.dtd)

```
<!ELEMENT students (student*) >
<!ELEMENT student (name,age,sex)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT age (#PCDATA)>
<!ELEMENT sex (#PCDATA)>
<!ATTLIST student number ID #REQUIRED>
```

4.schema约束(详细约束)

```
    导入xsd约束文档:
      1、编写根标签
      2、引入实例名称空间
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      3、引入名称空间
xsi:schemaLocation="http://www.itcast.cn/xml
student.xsd"
      4、引入默认的名称空间

    例子:
    xmlns:xmlnamespace:xml的命名空间,命名空间标识是命名空间最重要的属性，重要到当输出一个命名空间时就直接转换为它的标识。标识有个规范的称呼:URI(统一资源定位符)。URI的最大特点是唯一性。如果不唯一就失去了辨识的意义。

    xmlns="http://www.qianfeng.cn/xml"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance"
    xsi:schemaLocation="http://www.qianfeng.cn/xml
student.xsd"
```

- 简单案例

原始文件(stuent.xml)

```xml
<?xml version="1.0" encoding="UTF-8" ?>

<students   xmlns="http://www.qianfeng.cn/xml"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.qianfeng.cn/xml
student.xsd">
  <student number="feng_0001">
    <name>zs</name>
    <age>abc</age>
    <sex>yao</sex>
  </student>

</students>
```

schema约束文件(student.xsd)

```xml
<?xml version="1.0"?>
<xsd:schema xmlns="http://www.qianfeng.cn/xml"
        xmlns:xsd="http://www.w3.org/2001/XMLSchema"
        targetNamespace="http://www.qianfeng.cn/xml"
elementFormDefault="qualified">
    <xsd:element name="students" type="studentsType"/>
    <xsd:complexType name="studentsType">
        <xsd:sequence>
            <xsd:element name="student"
type="studentType" minOccurs="0"
maxOccurs="unbounded"/>
        </xsd:sequence>
    </xsd:complexType>
    <xsd:complexType name="studentType">
        <xsd:sequence>
            <xsd:element name="name"
type="xsd:string"/>
```

```
            <xsd:element name="age" type="ageType" />
            <xsd:element name="sex" type="sexType" />
        </xsd:sequence>
        <xsd:attribute name="number" type="numberType"
use="required"/>
    </xsd:complexType>
    <xsd:simpleType name="sexType">
        <xsd:restriction base="xsd:string">
            <xsd:enumeration value="male"/>
            <xsd:enumeration value="female"/>
        </xsd:restriction>
    </xsd:simpleType>
    <xsd:simpleType name="ageType">
        <xsd:restriction base="xsd:integer">
            <xsd:minInclusive value="0"/>
            <xsd:maxInclusive value="256"/>
        </xsd:restriction>
    </xsd:simpleType>
    <xsd:simpleType name="numberType">
        <xsd:restriction base="xsd:string">
            <xsd:pattern value="feng_\d{4}"/>
        </xsd:restriction>
    </xsd:simpleType>
</xsd:schema>

```

4.5.2. json字符串

```
就是用特殊的字符串
需要使用双引号。

传输数据时多以json串形式传递数据,可读性差,但是效率比xml高

表示对象：必须使用{}
var json = "
{'city':'hangzhou','BESTJINGDIAN':'XIHU','SECONDJINGDIA
N':'LEIFENGTA'}"

表示数组：必须使用[]
var json = '[{"city":"hangzhou"},{"city":"beijing"},
{"city":"shanghai"}]'

```