JDBC - 李伟杰

> 使用JDBC需要加载对应数据库的jar文件(mysql)

简介

```
Java DataBase Connectivity  Java数据库连接

我们学习的技术是JDBC  不是 MYSQLJDBC  也不是 ORACLE JDBC
JDBC是一套标准,是Java与各大数据库厂商共同定制的一套接口. 这套接口由各大数据库厂商进行了实现.
```

发展历史

```
    自从Java语言于1995年5月正式公布以来，Java风靡全球。

    出现大量的用java语言编写的程序，其中也包括数据库应用程序。由于没有一个Java语言的数据库操作API，编程人员不得不在Java程序中加入C语言的ODBC函数调用。这就使很多Java的优秀特性无法充分发挥，比如平台无关性、面向对象特性等。

    随着越来越多的编程人员对Java语言的日益喜爱，越来越多的公司在Java程序开发上投入的精力日益增加，对java语言接口的访问数据库的API的要求越来越强烈。也由于ODBC的有其不足之处，比如它并不容易使用，没有面向对象的特性等等，SUN公司决定开发一Java语言为接口的数据库应用程序开发接口。

    在JDK1.x版本中，JDBC只是一个可选部件，到了JDK1.1公布时，SQL类包(也就是JDBCAPI)就成为Java语言的标准部件。
```

使用步骤: *

```
1.  引入jar文件.
2.  加载数据库驱动 (JavaSE项目中可以省略 , JavaWeb项目必须编写此步骤)
    Class.forName("com.mysql.jdbc.Driver");
3.  通过驱动管理器, 获取JDBC连接对象.
    Connection conn = DriverManager.getConnection("数据库连接地址","帐号","密码");
    //  数据库连接地址格式:  主协议:子协议://ip地址:端口号/数据库名称
    //  mysql的连接地址: jdbc:mysql://localhost:3306/java35
    //  oracle的连接地址: jdbc:oracle:thin:@localhost:1521:ORCL
4.  通过连接对象, 创建SQL执行对象 (SQL执行环境)
    Statement state = conn.createStatement();
5.  通过SQL执行对象 ,执行SQL语句.
    state.execute(String sql语句);
6.  释放资源
```

```
    state.close();
    conn.close();
```

JDBC中常用的类型与方法

```
1.  DriverManager   :   驱动管理器
        常用方法:
         -   获取数据库连接:
         static Connection getConnection(String 数据库地址,String 账号 ,String 密
码)

2.  Connection  :   数据库连接对象
        常用方法:
          -   创建SQL执行对象:  Statement createStatement();


3.  Statement   :   SQL执行对象
        常用方法:
            -   执行SQL语句(查询语句返回true, 其它语句返回false)
                boolean execute(String sql);

            -   执行DML语句(INSERT UPDATE DELETE) 和 DDL语句(create alter drop)
                (返回int值, 表示语句对数据库表格的影响行数 !)
                (通常我们认为 返回值>0 表示执行成功.)
                int executeUpdate(String sql);

            -   执行DQL语句 (select)
                ResultSet executeQuery(String sql);

4.  ResultSet   :   结果集对象 (指的是一个select语句的查询结果)
        常用方法:
            1.  控制游标移动的常用方法:
                -   boolean next()  ****
                        作用: 控制游标向下一行移动.
                        返回值:    移动成功返回true , 下一行不存在移动失败, 返回
false

                -   boolean privious() 了解
                        作用: 控制游标向上一行移动.
                        返回值:    移动成功返回true , 上一行不存在移动失败, 返回
false

                -   boolean absolute(int 行号) 了解
                        作用: 控制游标向指定行移动
                        返回值:    移动成功返回true , 行不存在移动失败, 返回false

                -   boolean beforeFirst()   了解
                        作用: 控制游标移动到第一行
                        返回值: 移动成功返回true, 没有第一行数据返回false

                -   boolean afterLast() 了解
                        作用: 控制游标移动到最后一行
                        返回值: 移动成功返回true, 没有最后一行数据返回false


            2.  获取游标指向行的字段值的常用方法:
                -   XXX getXXX(String 列名) ***
```

根据字段名，得到此字段的值

- XXX getXXX(int 字段的索引) *
根据字段的索引，得到字段的值 ，索引从1开始

工厂方法设计模式（静态工厂方法模式）

工厂方法模式一种创建对象的模式.
工厂方法模式基于"输入"，应用在超类和多个子类之间的情况，这种模式将创建对象的责任转移到工厂类；

工厂设计模式的优点:
1. 面向接口编程，体现了面向对象的思想
2. 降低了耦合，将创建对象的工作转移到了工厂类

代码案例:
1. 水果接口
```
public interface Fruit {
    void eat();
}
```
2. 苹果 (水果的一种)
```
public class Apple implements Fruit{
    @Override
    public void eat() {
        System.out.println("苹果吃起来甜甜的脆脆的.");
    }
}
```
3. 香蕉 (水果的一种)
```
public class Banana implements Fruit{
    @Override
    public void eat() {
        System.out.println("香蕉吃起来软软的甜甜的");
    }
}
```
4. 静态工厂类
```
public class FruitFactory {
   public static Fruit get(){
       //return new Apple();
       return new Banana();
   }
}
```

DAO

DAO(Data Access Object)是一个数据访问接口，数据访问：顾名思义就是与数据库打交道。夹在业务逻辑与数据库资源中间。

为了建立一个健壮的Java应用，应该将所有对数据源的访问操作抽象封装在一个公共API中。用程序设计的语言来说，就是建立一个接口，接口中定义了此应用程序中将会用到的所有事务方法。在这个应用程序中，当需要和数据源进行交互的时候则使用这个接口，并且编写一个单独的类来实现这个接口在逻辑上对应这个特定的数据存储。

DAO模式是标准的JavaEE设计模式之一.开发人员使用这个模式把底层的数据访问操作和上层的商务逻辑分开.一个典型的DAO实现有下列几个组件:

1. 一个DAO工厂类;
2. 一个DAO接口;
3. 至少一个实现DAO接口的具体类;
4. 数据传递对象(有些时候叫做Bean对象).

SQL注入问题 *

进行用户登录时, 输入不存在的帐号 和 如下的密码:

```
1' or '1'='1
```

结果显示登录成功.

因为用户输入的密码, 与我们的查询语句拼接后, 使得我们的查询语句产生了歧义:

原查询语句:

```
select * from xzk_user where username='' and password='密码'
```

拼接后:

```
select * from xzk_user where username='hahahaheiheihei' and password='1' or '1'='1'
```

解决sql注入问题 *

我们可以将SQL语句与参数分离,将参数作为SQL的特殊部分进行预处理.

PreparedStatement 预编译的SQL执行环境

内部实现原理:

1. 将未拼接参数的SQL语句, 作为SQL指令, 先传递给数据库 进行编译.
2. 再将参数传递给数据库, 此时传递的参数不会再作为指令执行, 只会被当作文本存在.

操作流程与Statement基本一致:

1. 如何得到一个PreparedStatement 对象

```
PreparedStatement state = conn.prepareStatement("预编译的SQL语句");
```

2. 预编译的SQL语句如何编写

需要填充参数的位置, 使用?代替即可! 例如:

```
select id from xzk_user where username=? and password=?
```

3. 参数如何填充

```
state.setXXX(int index,XXX value);
```

setXXX中XXX指的是数据类型,

参数1: index : SQL语句中?的索引值 , 从1开始

参数2: value : 填充的参数值.

4. 如何执行填充完毕参数的SQL

- boolean execute();
- int executeUpdate();
- ResultSet executeQuery();

PreparedStatement与Statement谁的性能高?

```
看是什么数据库

在mysql中, preparedStatement原理是拼接SQL,  所以Statement性能高.
在Oracle中, preparedStatement原理是对SQL指令进行预处理, 再传递的参数不具备特殊含义.有更好的SQL缓存策略,PreparedStatement高.
```

事务

```
概述: 将多条SQL语句, 看作一个整体. 要么一起成功, 要么一起失败.

事务在mysql中, 是默认自动提交的 .
```

操作方式1: 命令行

```
-   开启事务:   start transaction;
-   回滚      :    rollback;   --此次事务中所有的sql操作, 放弃.
-   提交      :    commit; --此次事务中所有的sql操作, 作为一个整体, 提交.
```

操作方式2: Java

```
JDBC事务通过连接对象开启, 回滚 ,提交. 只针对当前连接对象生效.

-   开启事务:   conn.setAutoCommit(false);
-   回滚事务:   conn.rollback();
-   提交事务:   conn.commit();
```

事务面试题: *

```
1.  请描述事务的四大特征 :
        <1>.    原子性:    事务是一个整体 , 不可分割 , 要么同时成功, 要么同时失败.
        <2>.    持久性:    当事务提交或回滚后, 数据库会持久化的保存数据.
        <3>.    隔离性:    多个事务之间, 隔离开, 相互独立.
        <4>.    一致性:    事务操作的前后 , 数据总量不变 (例如: 转账时: 孟亮给帅兵转账是一个事务, 转账完毕后. 两人余额的和不变.)

2.  请描述什么是脏读, 幻读, 不可重复读 ?
        -   脏读: 读取到了一个事务 未提交的数据.
        -   不可重复读:  一个事务中, 两次连续的读取 ,结果不一致(中间被其它事务更改了).
        -   幻读: 一个事务A在执行DML语句时, 另一个事务B也在执行DML语句 , B修改了A修改过的数据, 导致A在查询时就像发生了幻觉一样(A更改的内容A看不到了.)

3.  请描述事务的隔离级别
            //三种级别锁:    页级,表级,行级(共享锁,排它锁).
        1.  读未提交    :   read uncommitted; (可能产生:脏读, 不可重复读, 幻读)
        2.  读已提交    :   read committed; (可能产生: 不可重复度, 幻读)
        3.  可重复读    :   repeatable read;(mysql默认值) (可能产生: 幻读)
        4.  串行化 :   serializable;

        -   查看数据库当前的隔离级别:   select @@tx_isolation; (了解)
        -   数据库设置隔离级别:  set global transaction isolation level 级别字符串; (了解)
```

批处理

```
将多条语句, 放到一起批量处理 .
批处理的原理: 将多条SQL语句, 转换为一个SQL指令.  显著的提高大量SQL语句执行时的数据库性能.


Statement对象使用流程:
    1.  得到Statement对象
        Statement state = conn.createStatement();

    2.  将一条SQL语句, 加入到批处理中.
        state.addBatch(String sql);

    3.  执行批处理
        state.executeBatch();

    4.  清空批处理
        state.clearBatch();

PreparedStatement对象使用流程:
    1.  得到PreparedStatement对象
        PreparedStatement state = conn.prepareStatement("预编译的SQL");

    2.  填充预编译的参数
        state.setXXX(1,填充参数);

    3.  将一条填充完毕参数的SQL, 加入到批处理中.
        state.addBatch();

    4.  执行批处理
        state.executeBatch();

    5.  清空批处理
        state.clearBatch();
```

Properties 作为配置文件

```
 Properties类 是Java中的Map集合的实现类.
.properties文件 用于通过文件描述一组键值对!
.properties文件 ,可以快速的转换为Properties类的对象.

文件中内容的格式:
    文件内容都是字符串 , 键与值之间通过等号连接 , 多个键值对之间换行分割.
    例如:
    url=xxx
    user=xxx
    password=xxx

如何将文件 转换为 集合:
    步骤:
        1.  创建Properties对象
                Properties ppt = new Properties();
        2.  创建一个字节输入流 , 指向.properties文件
                InputStream is = new FileInputStream("文件地址");
        3.  将字节输入流, 传递给properties对象, 进行加载.
                ppt.load(is);
```

连接池(DataSource)的使用 *

连接池用于缓存连接!

当我们需要使用连接时，可以不用再创建连接 ！ 可以直接从连接池中获取连接.
当连接池中存在空闲连接时，会将空闲连接给到程序使用.
当连接池中不存在空闲连接时，且连接池未满时 ，则创建连接提供给程序使用 ,并在程序使用完毕后，缓存连接.
当连接池中不存在空闲连接时，且连接池已满时 ，则排队等候空闲连接的出现.

注意:
    使用连接池中的连接对象操作数据库时，操作完毕依然需要释放连接(调用close()).
    连接池中的连接在设计时，使用了动态代理设计模式+装饰者设计模式 . 我们调用它的close方法，代理没有关闭这个连接，而是将连接重新放入了池中.

DBCP连接池的使用步骤 *

1. 引入相关的jar文件
    - dbcp.jar
    - poll.jar

2. 将配置文件引入
3. 将配置文件，转换为Properties对象
    Properties ppt = new Properties();
    ppt.load(配置文件的输入流);

4. 通过连接池的工厂类(BasicDataSourceFactory)的创建连接池的方法(createDataSource())
    DataSource ds = BasicDataSourceFactory.createDataSource(ppt);

5. 从连接池中 获取连接对象
    Connection conn = ds.getConnection();

德鲁伊连接池的使用步骤 *

1. 引入相关的jar文件
    - druid-1.0.9.jar
2. 将配置文件引入
3. 将配置文件，转换为Properties对象
    Properties ppt = new Properties();
    ppt.load(配置文件的输入流);

4. 通过连接池的工厂类(DruidDataSourceFactory)的创建连接池的方法(createDataSource())
    DataSource ds = DruidDataSourceFactory.createDataSource(ppt);

5. 从连接池中 获取连接对象
    Connection conn = ds.getConnection();

连接池工具类

Druid

```
public class DruidUtil{
    private static DataSource data = null;
    static {
```

```
        InputStream is =
DruidUtil.class.getClassLoader().getResourceAsStream("druid.properties");
        Properties ppt = new Properties();
        try {
            ppt.load(is);
            data = DruidDataSourceFactory.createDataSource(ppt);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    /**
     * 用于从DBCP连接池中 获取一个连接
     * @return DBCP连接池中的一个连接对象.
     */
    public static Connection getConnection() {
        try {
            return data.getConnection();
        } catch (SQLException e) {
            e.printStackTrace();
            return null;
        }
    }

    /**
     * 用于释放连接  , 执行环境 , 结果集 等资源
     * @param conn 要释放的连接资源
     * @param state 要释放的执行环境资源
     * @param result 要释放的结果集资源
     */
    public static void close(Connection conn,Statement state,ResultSet result) {
            if(result != null) {
                try {
                    result.close();
                } catch (SQLException e) {
                    // TODO Auto-generated catch block
                    e.printStackTrace();
                }
            }
            if(state != null) {
                try {
                    state.close();
                } catch (SQLException e) {
                    // TODO Auto-generated catch block
                    e.printStackTrace();
                }
            }
            if(conn != null) {
                try {
                    conn.close();
                } catch (SQLException e) {
                    // TODO Auto-generated catch block
                    e.printStackTrace();
                }
            }
    }
}
```

DBCP

```
public class DBCPUtil{
    private static DataSource data = null;
    static {
        InputStream is =
DBCPUtil.class.getClassLoader().getResourceAsStream("dbcp.properties");
        Properties ppt = new Properties();
        try {
            ppt.load(is);
            data = BasicDataSourceFactory.createDataSource(ppt);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
    /**
     * 用于从DBCP连接池中 获取一个连接
     * @return DBCP连接池中的一个连接对象.
     */
    public static Connection getConnection() {
        try {
            return data.getConnection();
        } catch (SQLException e) {
            e.printStackTrace();
            return null;
        }
    }

    /**
     * 用于释放连接  , 执行环境 , 结果集 等资源
     * @param conn 要释放的连接资源
     * @param state 要释放的执行环境资源
     * @param result 要释放的结果集资源
     */
    public static void close(Connection conn,Statement state,ResultSet result) {
            if(result != null) {
                try {
                    result.close();
                } catch (SQLException e) {
                    // TODO Auto-generated catch block
                    e.printStackTrace();
                }
            }
            if(state != null) {
                try {
                    state.close();
                } catch (SQLException e) {
                    // TODO Auto-generated catch block
                    e.printStackTrace();
                }
            }
            if(conn != null) {
                try {
                    conn.close();
                } catch (SQLException e) {
                    // TODO Auto-generated catch block
                    e.printStackTrace();
```

```
                }
            }
        }
    }
```