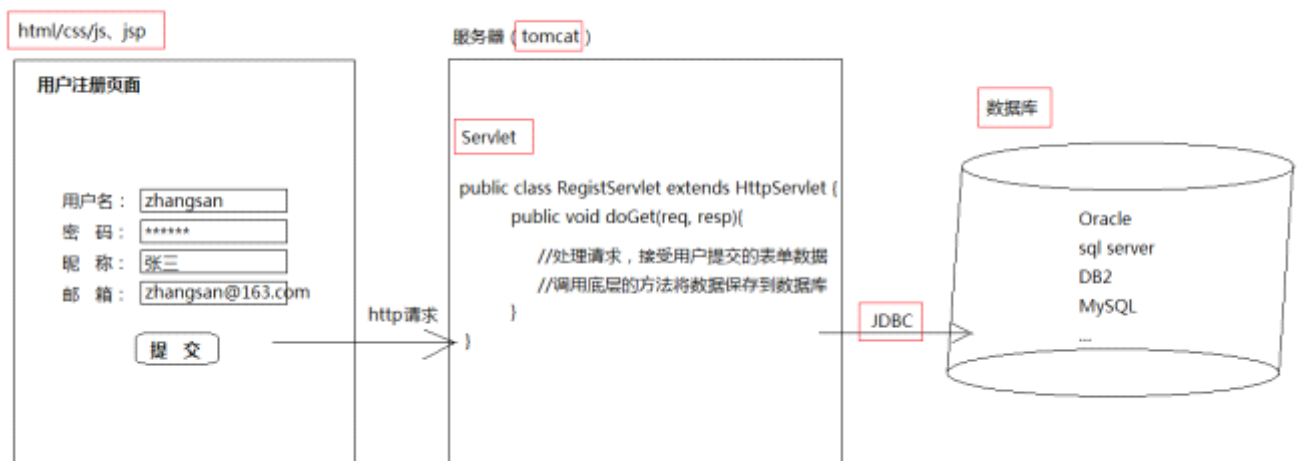
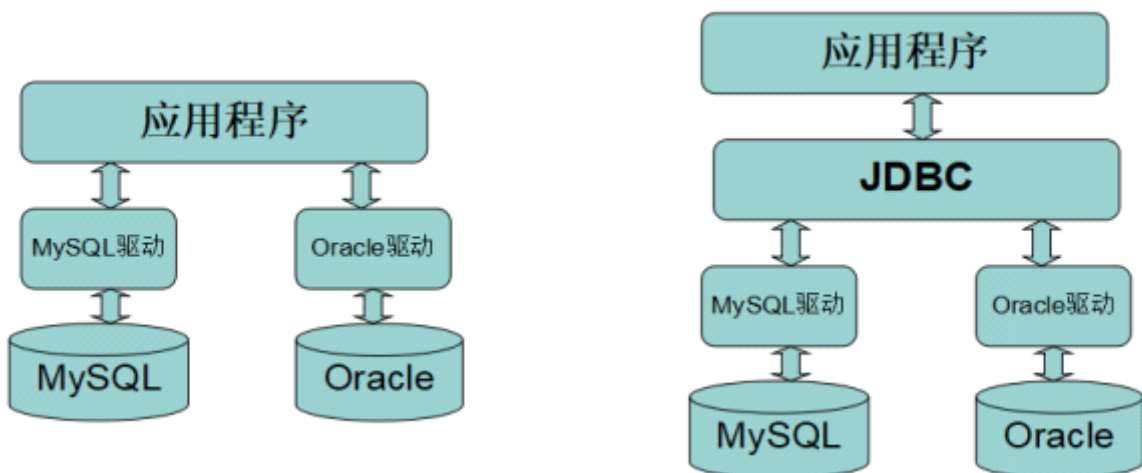


## 1. 知识回顾



## 2. JDBC概述



### a. 驱动

- 为了能让程序员利用java程序操作数据库，数据库厂商提供了一套jar包，通过导入这个jar包就可以直接调用其中的方法操作数据库，这个jar包称之为驱动。
  - 两个数据库驱动互不兼容。
- 行业中有很多种的数据库，要使用这么多数据库需要学习很多数据库驱动，对于程序员来说学习成本非常高。
- 想要让java程序兼容数据库，所有的数据库驱动都实现了jdbc这套接口。

### b. JDBC简介：

- JDBC全称为：Java Data Base Connectivity (java数据

库连接)，它主要由接口组成。

- 组成JDBC的 2 个包：

java.sql包

javax.sql包

- 开发JDBC应用需要以上2个包的支持外，还需要导入相应JDBC的数据库实现(即数据库驱动)。



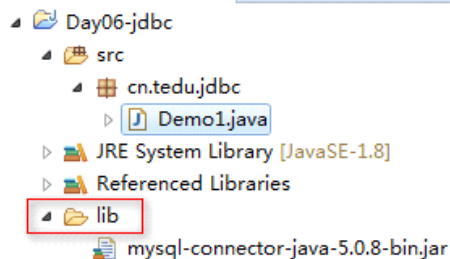
mysql-con  
nector...

- 不仅需要jdbc接口，还需要驱动这个实现，驱动中就是对jdbc接口的一些实现。

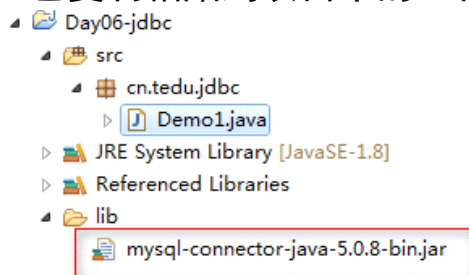
### 3. java程序中引入mysql数据库驱动包

- 课前资料找到mysql-connector-java-5.0.8-bin.jar文件

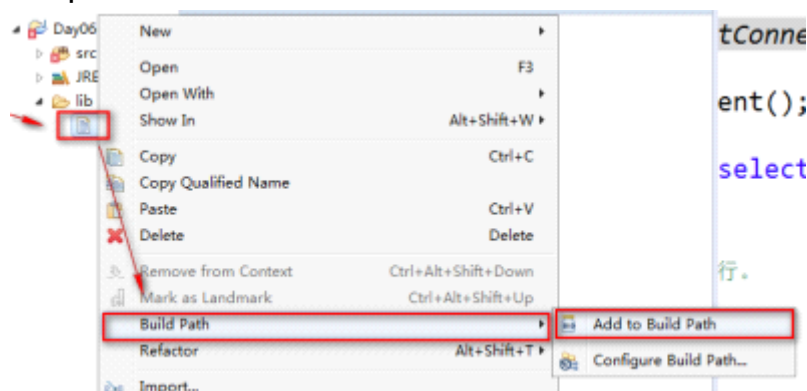
- 在项目下创建一个新的lib目录



- 将jar包复制粘贴到项目中的lib目录下



- 右键点击lib下的包，选择Build Path->add to build path



### 4. 6步实现jdbc

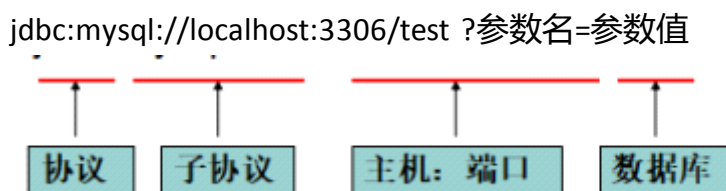
- 注册数据库驱动
- 获取数据库连接
- 创建传输器
- 传输sql并返回结果
- 遍历结果
- 关闭资源

## 1. 程序详解—DriverManager

- Jdbc程序中的DriverManager用于加载驱动，并创建与数据库的链接，这个API的常用方法：  
`DriverManager.registerDriver(new Driver())`  
`DriverManager.getConnection(url, user, password)`，
- 注意：在实际开发中并不推荐采用registerDriver方法注册驱动。原因有二：
  - 查看Driver的源代码可以看到，如果采用此种方式，会导致驱动程序注册两次，也就是在内存中会有两个Driver对象。
  - 程序依赖mysql的api，脱离mysql的jar包，程序将无法编译，将来程序切换底层数据库将会非常麻烦。
- 推荐方式：`Class.forName("com.mysql.jdbc.Driver");`
  - 采用此种方式不会导致驱动对象在内存中重复出现，并且采用此种方式，程序仅仅只需要一个字符串，不需要依赖具体的驱动，使程序的灵活性更高。
  - 同样，在开发中也不建议采用具体的驱动类型指向getConnection方法返回的connection对象。

## 2. 数据库URL

URL用于标识数据库的位置，程序员通过URL地址告诉JDBC程序连接哪个数据库，URL的写法为：



## 3. 常用数据库URL地址的写法：

Oracle写法：`jdbc:oracle:thin:@localhost:1521:sid`

SqlServer—`jdbc:microsoft:sqlserver://localhost:1433; DatabaseName=sid`

MySql—`jdbc:mysql://localhost:3306/sid`

Mysql的url地址的简写形式：`jdbc:mysql:///sid`

常用属性：`useUnicode=true&characterEncoding=UTF-8`

## 4. 程序详解—Connection

- Jdbc程序中的Connection，它用于代表数据库的链接，Connection是数据库编程中最重要的一个对象，客户端与数据库所有交互都是通过connection对象完成的，这个对象的常用方法：

`createStatement()`：创建向数据库发送sql的statement对象。

`prepareStatement(sql)`：创建向数据库发送预编译sql的PrepareStatement对象。

`prepareCall(sql)`：创建执行存储过程的callableStatement对象。

setAutoCommit(boolean autoCommit)：设置事务是否自动提交。

commit()：在链接上提交事务。

rollback()：在此链接上回滚事务。

## 5. 程序详解—Statement

- Jdbc程序中的Statement对象用于向数据库发送SQL语句，Statement对象常用方法：

executeQuery(String sql)：用于向数据库发送查询语句。

executeUpdate(String sql)：用于向数据库发送insert、update或delete语句

execute(String sql)：用于向数据库发送任意sql语句

addBatch(String sql)：把多条sql语句放到一个批处理中。

executeBatch()：向数据库发送一批sql语句执行。

## 6. 程序详解—ResultSet

- Jdbc程序中的ResultSet用于代表Sql语句的执行结果。ResultSet封装执行结果时，采用的类似于表格的方式。ResultSet 对象维护了一个指向表格数据行的游标，初始的时候，游标在第一行之前，调用ResultSet.next() 方法，可以使游标指向具体的数据行，进行调用方法获取该行的数据。
- ResultSet既然用于封装执行结果的，所以该对象提供的都是用于获取数据的get方法：
- 获取任意类型的数据
  - getObject(int index)
  - getObject(String columnName)
- 获取指定类型的数据，例如：
  - getString(int index)
  - getString(String columnName)
- 提问：数据库中列的类型是varchar，获取该列的数据调用什么方法？Int类型呢？bigInt类型呢？Boolean类型？

## 7. 常用数据类型转换表

SQL类型	Jdbc对应方法	返回类型
BIT(1) bit(n)	getBoolean getBytes()	Boolean byte[]
TINYINT	getByte()	Byte
SMALLINT	getShort()	Short
Int	getInt()	Int
BIGINT	getLong()	Long
CHAR, VARCHAR, LONGVARCHAR	getString()	String
Text(clob) Blob	getClob getBlob()	Clob Blob
DATE	getDate()	java.sql.Date
TIME	getTime()	java.sql.Time
TIMESTAMP	getTimestamp()	java.sql.Timestamp

## 8. ResultSet中的api

- ResultSet还提供了对结果集进行滚动的方法：
  - next()：移动到下一行
  - Previous()：移动到前一行
  - absolute(int row)：移动到指定行

beforeFirst()：移动resultSet的最前面。

afterLast()：移动到resultSet的最后面。

## 9. 程序详解—释放资源

### ○ 为什么要关闭资源?

在安装数据库的时候，设置过最大连接数量，如果用了不还连接，别人就无法使用了。

rs对象中可能包含很大的一个数据，对象保存在内存中，这样就十分占用内存。需要将他关闭。

最晚创建的对象，最先关闭。

- Jdbc程序运行完后，切记要释放程序在运行过程中，创建的那些与数据库进行交互的对象，这些对象通常是ResultSet, Statement和Connection对象。
- 特别是Connection对象，它是非常稀有的资源，用完后必须马上释放，如果Connection不能及时、正确的关闭，极易导致系统宕机。Connection的使用原则是尽量晚创建，尽量早的释放。
- 为确保资源释放代码能运行，资源释放代码也一定要放在finally语句中。

## 10. 释放资源

- 在关闭过程中可能会出现异常，为了能够关闭资源，需要将资源在finally中关闭。
- 如果在finally中关闭资源则需要将conn,stat,rs三个对象定义成全局的变量。
- 在conn,stat,rs三个变量出现异常的时候可能会关闭不成功，我们需要将他们在finally中置为null。conn,stat,rs这三个对象是引用，将引用置为null，它引用的对象就会被JVM回收，也能保证资源的释放。

11.

1. 由于在多个方法中都调用相同的创建连接和关闭资源的代码，所以可以将这些代码整理成一个方法。
2. 编写代码如下：

```
package cn.tedu.utils;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;

/**
 * 创建连接和关闭资源的工具类
 * 工具类中一般是通过类名.方法的形式来调用其中的方法的。
 * @author Administrator
 *
 */
public class JDBCUtils {
    private JDBCUtils(){
    }
    /**
     * 创建连接
     * @throws Exception
     */
    public static Connection getConnection() throws Exception{
        Class.forName("com.mysql.jdbc.Driver");
        return DriverManager.getConnection("jdbc:mysql:"
            + "///localhost:3306/mydb1?user=root&password=root");
    }
    /**
     * 关闭资源
     */
    public static void close(ResultSet rs,Statement stat,Connection conn){
        if(rs != null){
            try {
                rs.close();
            } catch (SQLException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            }finally{
                rs = null;
            }
        }
        if(stat != null){
            try {
                stat.close();
            } catch (SQLException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            }finally{
                stat =null;
            }
        }
        if(conn != null){
            try {
                conn.close();
            } catch (SQLException e) {
                // TODO Auto-generated catch block
            }
        }
    }
}
```

```
e.printStackTrace();  
    }finally{  
        conn = null;  
    }  
}  
  
}
```

### 3. 工具类优化：读取配置文件

在src目录下新建一个conf.properties文件，在其中书写jdbc所需的四项配置信息。

a. 配置内容如下:

conf. properties:

```
driver=com.mysql.jdbc.Driver
url=jdbc:mysql://localhost:3306/mydb1
user=root
password=root
```

b. 在JDBCUtils中修改如下:

i. `JDBCUtils.class.getClassLoader().getResource("conf.properties").getPath()`是利用类加载器启动的时候读取src目录下的配置文件名称的方式找到conf.properties配置文件的。

JDBCUtils.class.getClassLoader()	--获取类加载器
.getResource("文件名称")	-- 获取src目录中对应名称的文件。
.getPath()	-- 将路径转换成String类型的数据。

## iii. 代码:

```
package cn.tedu.utils;

import java.io.File;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import java.util.Properties;

/**
 * 创建连接和关闭资源的工具类
 * 工具类中一般是通过类名.方法的形式来调用其中的方法的。
 * @author Administrator
 *
 */
public class JDBCUtils {
    private JDBCUtils(){

    }
    private static Properties prop = null;
    //JDBCUtils.class.getClassLoader().getResource("conf.properties").getPath()
    //在类加载器启动的时候，可以读取当前类所在的工程的src目录，通过getClassLoader()方法获取类加载器，
    //再使用getResource()获取src目录，在参数位置书写要获取的文件名称。
    //最后通过getPath()获取String类型的路径返回值。
    static{
        //由于配置文件只需要读取一次即可，所以讲读取配置文件的过程书写在一个静态代码块中
        //这样在类加载的时候就会读取一个配置信息，并且保留在内存中，可以避免重复读取。
        prop = new Properties();
    }
}
```



```

        try {
            prop.load(new FileInputStream(new
                File(JDBCUtils.class.getClassLoader().getResource("conf.properties").getPath())));
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}

/**
 * 创建连接
 * @throws Exception
 */
public static Connection getConnection() throws Exception{

    Class.forName(prop.getProperty("driver"));
    return
        DriverManager.getConnection(prop.getProperty("url"),prop.getProperty("user"),prop.getProperty("password
            "));
}
/**
 * 关闭资源
 */
public static void close(ResultSet rs,Statement stat,Connection conn){
    if(rs != null){
        try {
            rs.close();
        } catch (SQLException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }finally{
            rs = null;
        }
    }
    if(stat != null){
        try {
            stat.close();
        } catch (SQLException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }finally{
            stat =null;
        }
    }
    if(conn != null){
        try {
            conn.close();
        } catch (SQLException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }finally{
            conn = null;
        }
    }
}
}
}

```

## 1. 在控制台输入用户名和密码之后实现登录操作：

代码：

```
package cn.tedu.jdbc;

import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.Statement;
import java.util.Scanner;

import cn.tedu.utils.JDBCUtils;

/**
 * 登录功能 ---JDBC实现
 * @author Administrator
 *
 */
public class Login {
    public static void main(String[] args) {
        //要求用户输入用户名和密码
        Scanner sc = new Scanner(System.in);
        System.out.println("请求输入用户名：");
        String name = sc.nextLine();
        System.out.println("请输入密码：");
        String password = sc.nextLine();
        //        loginTest(name,password);
        loginTestPrepared(name,password);
    }
    //prepared实现JDBC
    private static void loginTestPrepared(String name, String password) {
        Connection conn = null;
        PreparedStatement ps = null;
        ResultSet rs = null;
        try {
            conn = JDBCUtils.getConnection();
            ps = conn.prepareStatement("select * from user "
                    + "where name = ? and password = ?");
            ps.setString(1, name);
            ps.setString(2, password);
            rs = ps.executeQuery();
            if(rs.next()){
                System.out.println("登录成功");
            }else{
                System.out.println("登录失败");
            }
        } catch (Exception e) {
            e.printStackTrace();
        }finally{
            JDBCUtils.close(rs, ps, conn);
        }
    }

    private static void loginTest(String name, String password) {
        //将获取到的值和数据库中的数据作比对
        Connection conn = null;
```

```
Statement stat = null;
ResultSet rs = null;
try {
    conn = JDBCUtils.getConnection();
    stat = conn.createStatement();
    rs = stat.executeQuery("select * from user where name='"+name+"' and password='"+password+"'");
    if(rs.next()){
        System.out.println("登录成功");
    }else{
        System.out.println("登录失败");
    }
} catch (Exception e) {
    e.printStackTrace();
}finally{
    JDBCUtils.close(rs, stat, conn);
}
}
```

## 1. 什么是SQL注入攻击

由于sql语句中的参数是由前台传输而来，后台是对sql语句的拼接，在参数传递的过程中，用户可能输入一些数据库的关键字，这些关键字可能会导致sql的语义发生改变从而执行一些特殊的操作，像这样的操作就称之为sql注入攻击。

## 2. PreparedStatement的使用

有预编译效果。

### a. 预编译如何保证sql语句语义不改变：

- i. 在preparedStatement中，会先将sql语句发送到数据库，发送的语句到达数据库服务器之后，会在服务器中保留一份，并将这个语句转换成一个机器码，这个机器码是不允许也不能够被修改的，这样就保证了sql语句的语义不会被改变。
- ii. 在sql语句中如果有参数需要传入，则在sql语句的主干部分用?来代替参数。这时用户传入参数，传入的参数还会被当做纯文本来使用。第二次保证sql语句语义不会发生改变。
- iii. 总结：利用preparedStatement只需操作两步：
  - 1) 发送sql语句的主干部分，如果有参数用?预留。
  - 2) 发送当前sql语句的参数。

## 1. 使用情景：

如果有多条sql语句需要执行，一条书写一个JDBC的完整过程十分浪费资源，和编码的时间，所以希望使用一种方案一次性执行多条sql语句，那么像这样的方法数据库当中称之为批处理操作。

## 2. 批处理实现：

原理：一次性发送多条sql语句到数据库执行。减少代码量以及多次发送sql的过程。

## 3. 批处理中的api：

Statement.addBatch()	添加一条批处理语句。
Statement.executeBatch()	执行当前批处理。
Statement.clearBatch()	清除批处理中的语句。

**注意：**PreparedStatement在添加批处理的时候操作的都是统一语义的sql，所以之传输sql 的参数即可。

## 4. Statement批处理：

```
package cn.tedu.batch;

import java.sql.Connection;
import java.sql.Statement;

import cn.tedu.utils.JDBCUtils;

/**
 * Statement批处理
 * 优点：可以处理不同语义的sql语句。
 * 缺点：1.执行效率低下。
 * 2.sql语句无法预留在数据库服务器中，
 * 每次都是新发送一条sql语句到达数据库。数据库需要重新解读sql语句。
 * 3.没有预编译功能。
 * @author Administrator
 */
public class StatementBatchDemo1 {
    /**
     * create table t1(id int,name varchar(20));
     * insert into t1 values(1,'阿一');
     * insert into t1 values(2,'阿二');
     * insert into t1 values(3,'阿三');
     * insert into t1 values(4,'阿四');
     */
    public static void main(String[] args) {
        Connection conn = null;
        Statement stat = null;
        try {
            conn = JDBCUtils.getConnection();
            stat = conn.createStatement();
            //批处理过程
            stat.addBatch("create table t1(id int,name varchar(20))");
            stat.addBatch("insert into t1 values(1,'阿一')");
            stat.addBatch("insert into t1 values(2,'阿二')");
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            JDBCUtils.closeConnection(conn);
            JDBCUtils.closeStatement(stat);
        }
    }
}
```

```

        stat.addBatch("insert into t1 values(3,'阿三')");
        stat.addBatch("insert into t1 values(4,'阿四')");
        //通知数据库服务器sql语句已经准备完成
        //请执行当前批处理。
        stat.executeBatch();
        System.out.println("执行成功！");
    } catch (Exception e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    } finally{
        JDBCUtils.close(null, stat, conn);
    }
}
}

```

## 5. PreparedStatement批处理：

```

package cn.tedu.batch;

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

import cn.tedu.utils.JDBCUtils;

/**
 * PreparedStatement批处理：
 * 优点：1.执行效率较高
 *       2.sql语句已经预留在内存中可以重复使用。
 *       3.有预编译功能。
 * 缺点：不能处理不同语义的sql语句。
 * @author Administrator
 *
 */
public class PreparedBatchDemo1 {
    public static void main(String[] args) {
        Connection conn = null;
        PreparedStatement ps = null;
        try {
            conn = JDBCUtils.getConnection();
            ps = conn.prepareStatement("insert into t1 values(?,?)");
            for(int i=0;i<100199;i++){
                ps.setInt(1, i);
                ps.setString(2, "name"+i);
                ps.addBatch();
                if(i%1000 == 0){
                    //满1000条执行批处理
                    ps.executeBatch();
                    //将执行完成的数据释放,避免内存溢出。
                    ps.clearBatch();
                    System.out.println("第"+i/1000+"次成功");
                }
            }
            //如果数据中有不满1000的，为了保证数据都可以执行需要最后执行一遍executeBatch()
            ps.executeBatch();
            System.out.println("全部处理成功");
        } catch (Exception e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        } finally{
    
```

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