1 什么是JDBC

JDBC全称为：Java Data Base Connectivity,它是可以执行SQL语句的Java API。

2 为什么要用JDBC

市面上有非常多的数据库，本来我们是需要根据不同的数据库学习不同的API，sun公司为了简化这个操作，定义了JDBC API【接口】，sun公司只是提供了JDBC API【接口】，数据库厂商负责实现。对于我们来说，操作数据库都是在JDBC API【接口】上，使用不同的数据库，只要用数据库厂商提供的数据库驱动程序即可，这大大简化了我们的学习成本。

3 简单操作JDBC

步骤:

1. 导入MySQL或者Oracle驱动包
2. 装载数据库驱动程序
3. 获取到与数据库连接
4. 获取可以执行SQL语句的对象
5. 执行SQL语句
6. 关闭连接

```
Connection connection = null;
Statement statement = null;
ResultSet resultSet = null;

try {
    /*
     * 加载驱动有两种方式
     * 1：会导致驱动会注册两次，过度依赖于mysql的api，脱离的mysql的开发包，程序则无法编译
     * 2：驱动只会加载一次，不需要依赖具体的驱动，灵活性高
     * 我们一般都是使用第二种方式
     * */
    //1.
    //DriverManager.registerDriver(new com.mysql.jdbc.Driver());
    //2.
    Class.forName("com.mysql.jdbc.Driver");
    //获取与数据库连接的对象-Connetcion
    connection =
DriverManager.getConnection("jdbc:mysql://localhost:3306/zhongfucheng", "root",
"root");
    //获取执行sql语句的statement对象
    statement = connection.createStatement();
    //执行sql语句,拿到结果集
    resultSet = statement.executeQuery("SELECT * FROM users");
    //遍历结果集，得到数据
    while (resultSet.next()) {
        System.out.println(resultSet.getString(1));
        System.out.println(resultSet.getString(2));
    }

} catch (SQLException e) {
```

```
            e.printStackTrace();
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        } finally {
            /*
             * 关闭资源，后调用的先关闭
             *
             * 关闭之前，要判断对象是否存在
             * */
            if (resultSet != null) {
                try {
                    resultSet.close();
                } catch (SQLException e) {
                    e.printStackTrace();
                }
            }
            if (statement != null) {
                try {
                    statement.close();
                } catch (SQLException e) {
                    e.printStackTrace();
                }
            }
            if (connection != null) {
                try {
                    connection.close();
                } catch (SQLException e) {
                    e.printStackTrace();
                }
            }
        }
```

4 Connection对象

客户端与数据库所有的交互都是通过Connection来完成的。

常用的方法：

```
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PrepareSatement对象。
prepareCall(sql)    //创建执行存储过程的callableStatement对象
setAutoCommit(boolean autoCommit)   //设置事务自动提交
commit() //提交事务
rollback() //回滚事务
```

5 Statement对象

Statement对象用于向数据库发送Sql语句，对数据库的增删改查都可以通过此对象发送sql语句完成。

Statement对象的常用方法：

```
executeQuery(String sql)    //查询
executeUpdate(String sql)   //增删改
execute(String sql)         //任意sql语句都可以，但是目标不明确，很少用
addBatch(String sql)        //把多条的sql语句放进同一个批处理中
executeBatch()              //向数据库发送一批sql语句执行
```

6 ResultSet对象

ResultSet对象代表Sql语句的执行结果，当Statement对象执行executeQuery()时，会返回一个ResultSet对象。

ResultSet对象维护了一个数据行的游标【简单理解成指针】，调用ResultSet.next()方法，可以让游标指向具体的数据行，进行获取该行的数据。

常用方法：

```
getObject(String columnName) //获取任意类型的数据
getString(String columnName) //获取指定类型的数据【各种类型，查看API】
//对结果集进行滚动查看的方法
next()
Previous()
absolute(int row)
beforeFirst()
afterLast()
```

7 写一个简单工具类

通过上面的理解，我们已经能够使用JDBC对数据库的数据进行增删改查了，我们发现，无论增删改查都需要连接数据库，关闭资源，所以我们把连接数据库，释放资源的操作抽取到一个工具类。

```
/*
 * 连接数据库的driver，url，username，password通过配置文件来配置，可以增加灵活性
 * 当我们需要切换数据库的时候，只需要在配置文件中改以上的信息即可
 *
 * */
private static String  driver = null;
private static String  url = null;
private static String  username = null;
private static String password = null;

static {
    try {
        //获取配置文件的读入流
        InputStream inputStream =
UtilsDemo.class.getClassLoader().getResourceAsStream("db.properties");
        Properties properties = new Properties();
        properties.load(inputStream);
        //获取配置文件的信息
        driver = properties.getProperty("driver");
        url = properties.getProperty("url");
        username = properties.getProperty("username");
        password = properties.getProperty("password");
        //加载驱动类
        Class.forName(driver);
    } catch (IOException e) {
```

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

    public static Connection getConnection() throws SQLException {
        return DriverManager.getConnection(url,username,password);
    }
    public static void release(Connection connection, Statement statement, ResultSet
    resultSet) {
        if (resultSet != null) {
            try {
                resultSet.close();
            } catch (SQLException e) {
                e.printStackTrace();
            }
        }
        if (statement != null) {
            try {
                statement.close();
            } catch (SQLException e) {
                e.printStackTrace();
            }
        }
        if (connection != null) {
            try {
                connection.close();
            } catch (SQLException e) {
                e.printStackTrace();
            }
        }
    }
```

8 PreparedStatement对象

PreparedStatement对象继承Statement对象，它比Statement对象更强大，使用起来更简单。

1. Statement对象编译SQL语句时，如果SQL语句有变量，就需要使用分隔符来隔开，如果变量非常多，就会使SQL变得非常复杂。PreparedStatement可以使用占位符，简化sql的编写
2. Statement会频繁编译SQL。PreparedStatement可对SQL进行预编译，提高效率，预编译的SQL存储在PreparedStatement对象中
3. PreparedStatement防止SQL注入。【Statement通过分隔符'++',编写永等式，可以不需要密码就进入数据库】

```
String id = "2";//模拟查询id为2的信息
Connection connection = UtilsDemo.getConnection();
String sql = "SELECT * FROM users WHERE id = ?";
PreparedStatement preparedStatement = connection.preparedStatement(sql);
//第一个参数表示第几个占位符【也就是?号】，第二个参数表示值是多少
preparedStatement.setString(1,id);
ResultSet resultSet = preparedStatement.executeQuery();
if (resultSet.next()) {
    System.out.println(resultSet.getString("name"));
}
UtilsDemo.release(connection, preparedStatement, resultSet);//释放资源
```

9 批处理

当需要向数据库发送一批SQL语句执行时，应避免向数据库一条条发送执行，采用批处理以提升执行效率。

批处理有两种方式：

1. Statement
2. PreparedStatement

通过executeBath()方法批量处理执行SQL语句，返回一个int[]数组，该数组代表各句SQL的返回值。

以下代码是以Statement方式实现批处理

```
/*
* Statement执行批处理
* 优点：可以向数据库发送不同的SQL语句
* 缺点：SQL没有预编译；仅参数不同的SQL，需要重复写多条SQL
* */
Connection connection = UtilsDemo.getConnection();
Statement statement = connection.createStatement();
String sql1 = "UPDATE users SET name='zhongfucheng' WHERE id='3'";
String sql2 = "INSERT INTO users (id, name, password, email, birthday)" +
" VALUES('5','nihao','123','ss@qq.com','1995-12-1')";
//将sql添加到批处理
statement.addBatch(sql1);
statement.addBatch(sql2);
//执行批处理
statement.executeBatch();
//清空批处理的sql
statement.clearBatch();
UtilsDemo.release(connection, statement, null);
```

以下方式以PreparedStatement方式实现批处理

```
/*
* PreparedStatement批处理
* 优点：SQL语句预编译了;对于同一种类型的SQL语句，不用编写很多条
* 缺点：不能发送不同类型的SQL语句
* */
Connection connection = UtilsDemo.getConnection();
String sql = "INSERT INTO test(id,name) VALUES (?,?)";
PreparedStatement preparedStatement = connection.prepareStatement(sql);
for (int i = 1; i <= 205; i++) {
    preparedStatement.setInt(1, i);
    preparedStatement.setString(2, (i + "zhongfucheng"));
    //添加到批处理中
    preparedStatement.addBatch();
    if (i %2 ==100) {
        //执行批处理
        preparedStatement.executeBatch();
        //清空批处理【如果数据量太大，所有数据存入批处理，内存肯定溢出】
        preparedStatement.clearBatch();
    }
}
//不是所有的%2==100，剩下的再执行一次批处理
preparedStatement.executeBatch();
```

```
//再清空
preparedStatement.clearBatch();
UtilsDemo.release(connection, preparedStatement, null);
```

10 处理大文本和二进制数据

clob和blob

- clob用于存储大文本
- blob用于存储二进制数据

MySQL存储大文本是用Test【代替clob】，Test又分为4类。

- TINYTEXT
- TEXT
- MEDIUMTEXT
- LONGTEXT

同理blob也有这4类。

11 获取数据库的自动主键列

为什么要获取数据库的自动主键列数据?

应用场景:

有一张老师表，一张学生表。现在来了一个新的老师，学生要跟着新老师上课。

我首先要知道老师的id编号是多少，学生才能知道跟着哪个老师学习【学生外键参照老师主键】。

12 调用数据库的存储过程

调用存储过程的语法:

```
{call <procedure-name>[(<arg1>,<arg2>, ...)]}
```

调用函数的语法:

```
{?= call <procedure-name>[(<arg1>,<arg2>, ...)]}
```

如果是Output类型的，那么在JDBC调用的时候是要注册的。

```
connection = JdbcUtils.getConnection();
callableStatement = connection.prepareCall("{call demoSp(?,?)}");
callableStatement.setString(1, "nihaoa");
//注册第2个参数,类型是VARCHAR
callableStatement.registerOutParameter(2, Types.VARCHAR);
callableStatement.execute();
//获取传出参数[获取存储过程里的值]
String result = callableStatement.getString(2);
System.out.println(result);
```

13 事务

一个SESSION所进行的所有更新操作要么一起成功，要么一起失败。

举个例子:A向B转账，转账这个流程中如果出现问题，事务可以让数据恢复成原来一样【A账户的钱没变，B账户的钱也没变】。

```
try{
    //开启事务,对数据的操作就不会立即生效。
    connection.setAutoCommit(false);
    //A账户减去500块
    String sql = "UPDATE a SET money=money-500 ";
    preparedStatement = connection.prepareStatement(sql);
    preparedStatement.executeUpdate();
    //在转账过程中出现问题
    int a = 3 / 0;
    //B账户多500块
    String sql2 = "UPDATE b SET money=money+500";
    preparedStatement = connection.prepareStatement(sql2);
    preparedStatement.executeUpdate();
    //如果程序能执行到这里，没有抛出异常，我们就提交数据
    connection.commit();
    //关闭事务【自动提交】
    connection.setAutoCommit(true);
} catch (SQLException e) {
    try {
        //如果出现了异常，就会进到这里来，我们就把事务回滚【将数据变成原来那样】
        connection.rollback();
        //关闭事务【自动提交】
        connection.setAutoCommit(true);
    } catch (SQLException e1) {
        e1.printStackTrace();
    }
}
```

注意：当Connection遇到一个未处理的SQLException时，系统会非正常退出，事务也会自动回滚，但如果程序捕获到了异常，是需要在catch中显式回滚事务的。

1 savapoint

我们还可以使用savepoint设置中间点。如果在某地方出错了，我们设置中间点，回滚到出错之前即可。

应用场景：现在我们要算一道数学题，算到后面发现算错数了。前面的运算都是正确的，我们不可能重头再算【直接rollback】，最好的做法就是在保证前面算对的情况下，设置一个保存点。从保存点开始重新算。

注意：savepoint不会结束当前事务，普通提交和回滚都会结束当前事务的。

2 事务的隔离级别

数据库定义了4个隔离级别：

1. Serializable【可避免脏读，不可重复读，虚读】
2. Repeatable read【可避免脏读，不可重复读】
3. Read committed【可避免脏读】
4. Read uncommitted【级别最低，什么都避免不了】

分别对应Connection类中的4个常量

1. TRANSACTION_READ_UNCOMMITTED

2. TRANSACTION_READ_COMMITTED
3. TRANSACTION_REPEATABLE_READ
4. TRANSACTION_SERIALIZABLE

脏读：一个事务读取到另外一个事务未提交的数据

例子：A向B转账，A执行了转账语句，但A还没有提交事务，B读取数据，发现自己账户钱变多了！B跟A说，我已经收到钱了。A回滚事务【rollback】，等B再查看账户的钱时，发现钱并没有多。

不可重复读：一个事务读取到另外一个事务已经提交的数据，也就是说一个事务可以看到其他事务所做的修改

注：A查询数据库得到数据，B去修改数据库的数据，导致A多次查询数据库的结果都不一样【危害：A每次查询的结果都是受B的影响的，那么A查询出来的信息就没有意思了】

虚读(幻读)：是指在一个事务内读取到了别的事务插入的数据，导致前后读取不一致。

注：和不可重复读类似，但虚读(幻读)会读到其他事务的插入的数据，导致前后读取不一致

简单总结：脏读是不可容忍的，不可重复读和虚读在一定的情况下是可以的【做统计的肯定就不行】

14 元数据

元数据其实就是数据库，表，列的定义信息

即使我们写了一个简单工具类，我们的代码还是非常冗余。对于增删改而言，只有SQL和参数是不同的，我们为何不把这些相同的代码抽取成一个方法？对于查询而言，不同的实体查询出来的结果集是不一样的。我们要使用元数据获取结果集的信息，才能对结果集进行操作。

- ParameterMetaData --参数的元数据
- ResultSetMetaData --结果集的元数据
- DataBaseMetaData --数据库的元数据

15 改造JDBC工具类

问题：我们对数据库的增删改查都要连接数据库，关闭资源，获取PreparedSteatment对象，获取Connection对象此类的操作，这样的代码重复率是极高的，所以我们要对工具类进行增强

1 增删改

```
//我们发现，增删改只有SQL语句和传入的参数是不知道的而已，所以让调用该方法的人传递进来
//由于传递进来的参数是各种类型的，而且数目是不确定的，所以使用Object[]
public static void update(String sql, Object[] objects) {
    Connection connection = null;
    PreparedStatement preparedStatement = null;
    ResultSet resultSet = null;
    try {
        connection = getConnection();
        preparedStatement = connection.prepareStatement(sql);
        //根据传递进来的参数，设置SQL占位符的值
        for (int i = 0; i < objects.length; i++) {
            preparedStatement.setObject(i + 1, objects[i]);
        }
        //执行SQL语句
        preparedStatement.executeUpdate();
    } catch (Exception e) {
        e.printStackTrace();
    }
```

```
}
```

2 查询

```
/*
1:对于查询语句来说，我们不知道对结果集进行什么操作【常用的就是把数据封装成一个Bean对象，封装成一个List集合】
2:我们可以定义一个接口，让调用者把接口的实现类传递进来
3:这样接口调用的方法就是调用者传递进来实现类的方法。【策略模式】
 */
//这个方法的返回值是任意类型的，所以定义为Object。
public static Object query(String sql, Object[] objects, ResultSetHandler rsh) {
    Connection connection = null;
    PreparedStatement preparedStatement = null;
    ResultSet resultSet = null;
    try {
        connection = getConnection();
        preparedStatement = connection.prepareStatement(sql);
        //根据传递进来的参数，设置SQL占位符的值
        if (objects != null) {
            for (int i = 0; i < objects.length; i++) {
                preparedStatement.setObject(i + 1, objects[i]);
            }
        }
        resultSet = preparedStatement.executeQuery();
        //调用调用者传递进来实现类的方法，对结果集进行操作
        return rsh.hanlder(resultSet);
    }catch (Exception e) {
        e.printStackTrace();
    }
}
```

接口:

```
/*
* 定义对结果集操作的接口，调用者想要对结果集进行什么操作，只要实现这个接口即可
* */
public interface ResultSetHandler {
     Object hanlder(ResultSet resultSet);
}
```

实现类:

```
//接口实现类，对结果集封装成一个Bean对象
public class BeanHandler implements ResultSetHandler {
    //要封装成一个Bean对象，首先要知道Bean是什么，这个也是调用者传递进来的。
    private Class clazz;
    public BeanHandler(Class clazz) {
        this.clazz = clazz;
    }
    @Override
    public Object hanlder(ResultSet resultSet) {
        try {
            //创建传进对象的实例化
            Object bean = clazz.newInstance();
```

```
            if (resultSet.next()) {
                //拿到结果集元数据
                ResultSetMetaData resultSetMetaData = resultSet.getMetaData();
                for (int i = 0; i < resultSetMetaData.getColumnCount(); i++) {
                    //获取到每列的列名
                    String columnName = resultSetMetaData.getColumnName(i+1);
                    //获取到每列的数据
                    String columnData = resultSet.getString(i+1);
                    //设置Bean属性
                    Field field = clazz.getDeclaredField(columnName);
                    field.setAccessible(true);
                    field.set(bean,columnData);
                }
                //返回Bean对象
                return bean;
            }
        }catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

16 数据库连接池

简单来说：数据库连接池就是提供连接的

- 数据库的连接的建立和关闭是非常消耗资源的
- 频繁地打开、关闭连接造成系统性能低下

1 编写连接池

1. 编写连接池需实现java.sql.DataSource接口
2. 创建批量的Connection用LinkedList保存【既然是个池，当然用集合保存、、LinkedList底层是链表，对增删性能较好】
3. 实现getConnetion()，让getConnection()每次调用，都是在LinkedList中取一个Connection返回给用户
4. 调用Connection.close()方法，Connction返回给LinkedList

问题：我们调用Conncetion.close()方法，是把数据库的物理连接关掉，而不是返回给LinkedList的

解决思路：

1. 写一个Connection子类，覆盖close()方法
2. 写一个Connection包装类，增强close()方法
3. 用动态代理，返回一个代理对象出去，拦截close()方法的调用，对close()增强

第一个思路：写一个Connection子类。

Connection是通过数据库驱动加载的，保存了数据的信息。写一个子类Connection，new出对象，子类的Connction无法直接继承父类的数据信息，也就是说子类的Connection是无法连接数据库的，更别谈覆盖close()方法了。

第二个思路：写一个Connection包装类。

1. 写一个类，实现与被增强对象的相同接口【Connection接口】
2. 定义一个变量，指向被增强的对象
3. 定义构造方法，接收被增强对象
4. 覆盖想增强的方法

5. 对于不想增强的方法，直接调用被增强对象的方法

这个思路本身是没什么毛病的，就是实现接口时，方法太多了！，所以我们也不使用此方法。

第三个思路：动态代理

```
@Override
public Connection getConnection() throws SQLException {
    if (list.size() > 0) {
        final Connection connection = list.removeFirst();
        //看看池的大小
        System.out.println(list.size());
        //返回一个动态代理对象
        return (Connection) Proxy.newProxyInstance(Demo1.class.getClassLoader(),
connection.getClass().getInterfaces(), new InvocationHandler() {
            @Override
            public Object invoke(Object proxy, Method method, Object[] args)
throws Throwable {
                //如果不是调用close方法，就按照正常的来调用
                if (!method.getName().equals("close")) {
                    return method.invoke(connection, args);
                } else {
                    //进到这里来，说明调用的是close方法
                    list.add(connection);
                    //再看看池的大小
                    System.out.println(list.size());
                }
                return null;
            }
        });
    }
    return null;
}
```

2 DBCP

用DBCP数据源的步骤：

1. 导入两个jar包【Commons-dbcp.jar和Commons-pool.jar】
2. 读取配置文件
3. 获取BasicDataSourceFactory对象
4. 创建DataSource对象

```
private static DataSource dataSource = null;
static {
    try {
        //读取配置文件
        InputStream inputStream =
Demo3.class.getClassLoader().getResourceAsStream("dbcpconfig.properties");
        Properties properties = new Properties();
        properties.load(inputStream);
        //获取工厂对象
        BasicDataSourceFactory basicDataSourceFactory = new
BasicDataSourceFactory();
        dataSource = basicDataSourceFactory.createDataSource(properties);
    } catch (IOException e) {
        e.printStackTrace();
```

```
    } catch (Exception e) {
        e.printStackTrace();
    }
}
public static Connection getConnection() throws SQLException {
    return dataSource.getConnection();

}
//这里释放资源不是把数据库的物理连接释放了，是把连接归还给连接池【连接池的Connection内部自己做好了】
public static void release(Connection conn, Statement st, ResultSet rs) {
    if (rs != null) {
        try {
            rs.close();
        } catch (Exception e) {
            e.printStackTrace();
        }
        rs = null;
    }
    if (st != null) {
        try {
            st.close();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
    if (conn != null) {
        try {
            conn.close();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

3 C3P0

C3P0数据源的性能更胜一筹，并且它可以使用XML配置文件配置信息！

步骤：

1. 导入开发包【c3p0-0.9.2-pre1.jar】和【mchange-commons-0.2.jar】
2. 导入XML配置文件【可以在程序中自己一个一个配，C3P0的doc中的Configuration有XML文件的事例】
3. new出ComboPooledDataSource对象

```
private static ComboPooledDataSource comboPooledDataSource = null;
static {
    //如果我什么都不指定，就是使用XML默认的配置，这里我指定的是oracle的
    comboPooledDataSource = new ComboPooledDataSource("oracle");
}
public static Connection getConnection() throws SQLException {
    return comboPooledDataSource.getConnection();
}
```

4 Tomcat数据源

Tomcat服务器也给我们提供了连接池，内部其实就是DBCP

步骤：

1. 在META-INF目录下配置context.xml文件【文件内容可以在tomcat默认页面的 JNDI Resources下 Configure Tomcat's Resource Factory找到】
2. 导入Mysql或oracle开发包到tomcat的lib目录下
3. 初始化JNDI->获取JNDI容器->检索以XXX为名字在JNDI容器存放的连接池

context.xml文件的配置：

```
<Context>
  <Resource name="jdbc/EmployeeDB"
            auth="Container"
            type="javax.sql.DataSource"
            username="root"
            password="root"
            driverClassName="com.mysql.jdbc.Driver"
            url="jdbc:mysql://localhost:3306/zhongfucheng"
            maxActive="8"
            maxIdle="4"/>
</Context>
```

```
try {
    //初始化JNDI容器
    Context initCtx = new InitialContext();
    //获取到JNDI容器
    Context envCtx = (Context) initCtx.lookup("java:comp/env");
    //扫描以jdbc/EmployeeDB名字绑定在JNDI容器下的连接池
    DataSource ds = (DataSource)envCtx.lookup("jdbc/EmployeeDB");
    Connection conn = ds.getConnection();
    System.out.println(conn);
}
```

17 dbutils框架

dbutils它是对JDBC的简单封装，极大简化jdbc编码的工作量。

1 DbUtils类

提供了关闭连接，装载JDBC驱动，回滚提交事务等方法的工具类【比较少使用，因为我们学了连接池，就应该使用连接池连接数据库】

2 QueryRunner类

该类简化了SQL查询，配合ResultSetHandler使用，可以完成大部分的数据库操作，重载了许多的查询，更新，批处理方法。大大减少了代码量。

3 ResultSetHandler接口

该接口规范了对ResultSet的操作，要对结果集进行什么操作，传入ResultSetHandler接口的实现类即可。

- ArrayHandler：把结果集中的第一行数据转成对象数组。
- ArrayListHandler：把结果集中的每一行数据都转成一个数组，再存放到List中。
- BeanHandler：将结果集中的第一行数据封装到一个对应的JavaBean实例中。

- BeanListHandler：将结果集中的每一行数据都封装到一个对应的JavaBean实例中，存放到List里。
- ColumnListHandler：将结果集中某一列的数据存放到List中。
- KeyedHandler(name)：将结果集中的每一行数据都封装到一个Map里，再把这些map再存到一个map里，其key为指定的key。
- MapHandler：将结果集中的第一行数据封装到一个Map里，key是列名，value就是对应的值。
- MapListHandler：将结果集中的每一行数据都封装到一个Map里，然后再存放到List
- ScalarHandler 将ResultSet的一个列到一个对象中。

使用DbUtils框架对数据库的CRUD

```
/*
* 使用DbUtils框架对数据库的CRUD
* 批处理
* */
public class Test {
    @org.junit.Test
    public void add() throws SQLException {
        //创建出QueryRunner对象
        QueryRunner queryRunner = new QueryRunner(JdbcUtils.getDataSource());
        String sql = "INSERT INTO student (id,name) VALUES(?,?)";
        //我们发现query()方法有的需要传入Connection对象，有的不需要传入
        //区别：你传入Connection对象是需要你来销毁该Connection，你不传入，由程序帮你把
Connection放回到连接池中
        queryRunner.update(sql, new Object[]{"100", "zhongfucheng"});
    }
    @org.junit.Test
    public void query()throws SQLException {

        //创建出QueryRunner对象
        QueryRunner queryRunner = new QueryRunner(JdbcUtils.getDataSource());
        String sql = "SELECT * FROM student";

        List list = (List) queryRunner.query(sql, new
BeanListHandler(Student.class));
        System.out.println(list.size());

    }
    @org.junit.Test
    public void delete() throws SQLException {
        //创建出QueryRunner对象
        QueryRunner queryRunner = new QueryRunner(JdbcUtils.getDataSource());
        String sql = "DELETE FROM student WHERE id='100'";

        queryRunner.update(sql);
    }
    @org.junit.Test
    public void update() throws SQLException {
        //创建出QueryRunner对象
        QueryRunner queryRunner = new QueryRunner(JdbcUtils.getDataSource());
        String sql = "UPDATE student SET name=? WHERE id=?";

        queryRunner.update(sql, new Object[]{"zhongfuchengaaa", 1});
    }
    @org.junit.Test
    public void batch() throws SQLException {
        //创建出QueryRunner对象
```

```
        QueryRunner queryRunner = new QueryRunner(JdbcUtils.getDataSource());
        String sql = "INSERT INTO student (name,id) VALUES(?,?)";

        Object[][] objects = new Object[10][];
        for (int i = 0; i < 10; i++) {
            objects[i] = new Object[]{"aaa", i + 300};
        }
        queryRunner.batch(sql, objects);
    }
}
```

18 分页

分页技术是非常常见的，在搜索引擎下搜索页面，不可能把全部数据都显示在一个页面里边。所以我们用到了分页技术。

1 Mysql实现分页

```
/*
Mysql分页语法：
@start---偏移量，不设置就是从0开始【也就是(currentPage-1)*lineSize】
@length---长度，取多少行数据
*/
SELECT *
    FROM 表名
    LIMIT [START], length;
/*
例子：我现在规定每页显示5行数据，我要查询第2页的数据
分析：1：第2页的数据其实就是从第6条数据开始，取5条
实现：1：start为5【偏移量从0开始】    2：length为5
*/
```

总结：Mysql从(currentPage-1)*lineSize开始取数据，取lineSize条数据。

2 使用JDBC连接数据库实现分页

下面是常见的分页图片

配合图片，看下我们的需求是什么：

1. 算出有多少页的数据，显示在页面上。
2. 根据页码，从数据库显示相对应的数据。

分析：

1. 算出有多少页数据这是非常简单的【在数据库中查询有多少条记录，你每页显示多少条记录，就可以算出有多少页数据了】
2. 使用Mysql或Oracle的分页语法即可

通过上面分析，我们会发现需要用到4个变量

- currentPage--当前页【由用户决定的】
- totalRecord--总数据数【查询表可知】
- lineSize--每页显示数据的数量【由我们开发人员决定】

- pageCount--页数【totalRecord和lineSize决定】