

题目：4 天贯通 JDBC 技术

主要内容

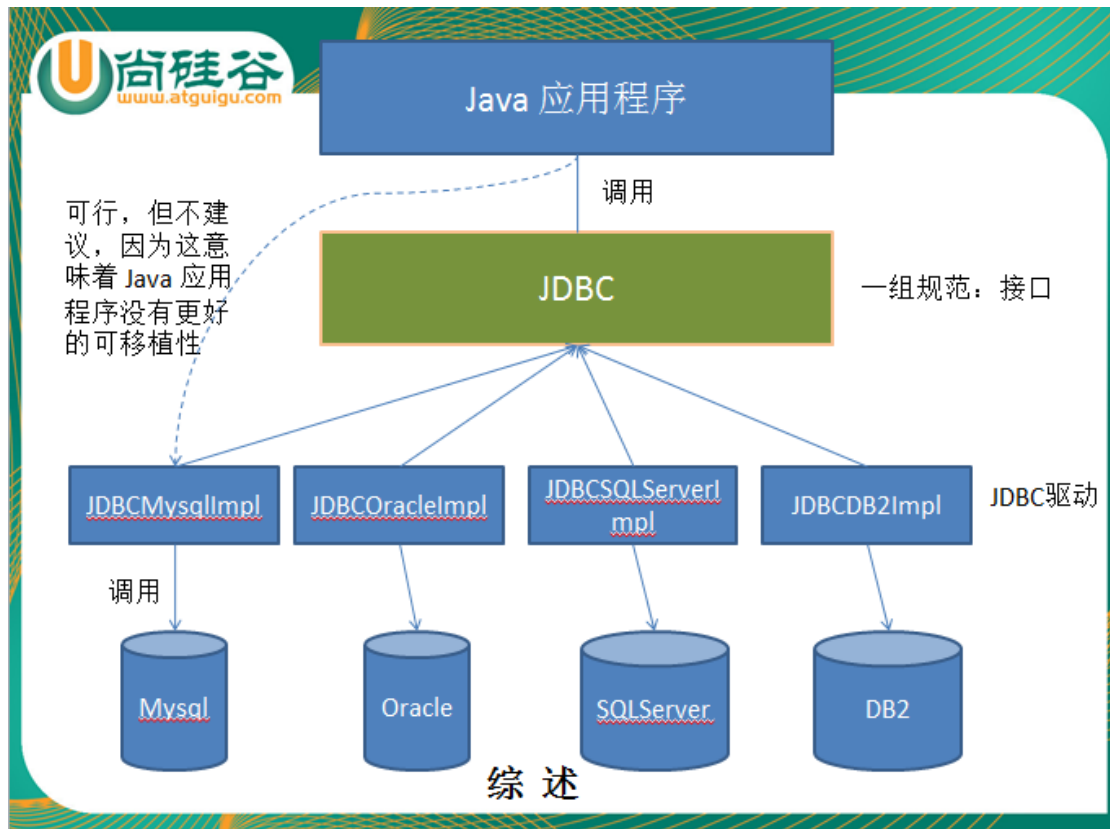
1. JDBC 概述
2. 获取数据库连接
3. 使用 Statement 操作数据表：UPDATE/DELETE/INSERT
4. 使用 ResultSet、ResultSetMetaData 操作数据表：SELECT
5. 使用 PreparedStatement
 - 实现数据表的 DML 操作
 - 向数据表中插入、读取大数据：BLOB 字段
6. 批量处理、数据库元数据
7. 数据库事务
8. 数据库连接池
 - C3P0 数据库连接池
 - DBCP 数据库连接池
9. DBUtils 工具类
 - 使用 QueryRunner，实现 UPDATE()和 QUERY()方法
 - 利用 DbUtils 编写 DAO 通用类

一、概述

JDBC:使用 Java 应用程序“操作”数据库的一门技术

jdbc，提供了使用 java 程序连接、操作数据库的一系列的 API。对于 java 程序来讲，只需要面向这套 API 进行编程即可。（面向接口的编程思想）。不同的数据库厂商只需要根据这套 JDBC API 提供各自的实现即可。这套实现类的集合即为不同数据库的驱动。

➔ java 应用程序 ---> JDBC API --> JDBC 的驱动 ---> 数据库



LAMP JavaEE .NET

二、获取数据库连接

//获取数据库的连接

```
public static Connection getConnection() throws Exception{
```

//1.获取数据库连接的基本信息

//1.1 创建 Properties 的对象，以流的形式，将配置文件中的基本信息读入程序

```
Properties info = new Properties();
```

```
info.load(new FileInputStream("jdbc.properties"));
```

//1.2 提供 4 个基本信息：url、driverClass、user、password

```
String url = info.getProperty("url");
```

```
String driverClass = info.getProperty("driverClass");
```

```
String user = info.getProperty("user");
```

```
String password = info.getProperty("password");
```

//2.加载驱动

```
Class.forName(driverClass);
```

//3.使用 DriverManager 的 getConnection(url,user,password)方法

```
Connection conn = DriverManager.getConnection(url, user, password);
```

```
return conn;
```

```
}

public static void close(ResultSet rs,Statement st,Connection conn){

    if(rs != null){
        try {
            rs.close();
        } catch (SQLException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }

    if(st != null){
        try {
            st.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();
        }
    }
}
```

【jdbc.properties】

```
url=jdbc:mysql://127.0.0.1:3306/test
user=root
password=123456
driverClass=com.mysql.jdbc.Driver
```

```
#user=scott
#password=tiger
#url=jdbc:oracle:thin:@127.0.0.1:1521:orcl
#driverClass=oracle.jdbc.driver.OracleDriver
```

三、Statement 与 ResultSet

- 通过调用 **Connection** 对象的 **createStatement** 方法创建该对象
 - **Statement st = conn.createStatement();**
- 该对象用于执行静态的 **SQL** 语句，并且返回执行结果
- **Statement** 接口中定义了下列方法用于执行 **SQL** 语句：
 - **ResultSet excuteQuery(String sql)**
 - **int excuteUpdate(String sql)**

//通用的 INSERT UPDATE DELETE 方法(version 1.0)

```
public void update(String sql){
//1.获取数据库的连接
Connection conn = null;
Statement st = null;
try{
conn = JDBCUtils.getConnection();
//2.提供一个 Statement 对象，将 sql 传递给数据库中执行
st = conn.createStatement();
st.execute(sql);

}catch(Exception e){
e.printStackTrace();
}finally{
//3.关闭 Statement 对象及连接
JDBCUtils.close(null, st, conn);
}
}
```

// 通用的查询方法,返回一个对象(version 1.0)

```
public <T> T get(String sql, Class<T> clazz) {
Connection conn = null;
Statement st = null;
ResultSet rs = null;
T t = null;
try {
t = clazz.newInstance();
conn = JDBCUtils.getConnection();
st = conn.createStatement();
rs = st.executeQuery(sql);
/*
* 通过 ResultSet 调用 getMetaData()返回一个结果集的元数据:ResultSetMetaData
*
* 1.getColumnCount():返回结果集的列数
*/
}
```

```
* 2.getColumnLabel():返回列的别名
*/
ResultSetMetaData rsmd = rs.getMetaData();
int columnCount = rsmd.getColumnCount();
if (rs.next()) {
    for (int i = 0; i < columnCount; i++) {
        Object columnVal = rs.getObject(i + 1);// 相应列的值
        //String columnName = rsmd.getColumnName(i + 1);
        String columnName = rsmd.getColumnLabel(i + 1);
        //使用 PropertyUtils 将指定对象 t 的指定属性 columnName 设置为指定的值
        columnVal
        PropertyUtils.setProperty(t, columnName, columnVal);
    }
}
} catch (Exception e) {
    e.printStackTrace();
} finally {
    JDBCUtils.close(rs, st, conn);
}
return t;
}

//通用的返回多个对象的查询操作(version 1.0)
public <T> List<T> getInstances(String sql,Class<T> clazz){
    Connection conn = null;
    Statement st = null;
    ResultSet rs = null;
    List<T> list = new ArrayList<T>();
    try {
        conn = JDBCUtils.getConnection();
        st = conn.createStatement();
        rs = st.executeQuery(sql);
        /*
        * 通过 ResultSet 调用 getMetaData()返回一个结果集的元数据: ResultSetMetaData
        *
        * 1.getColumnCount():返回结果集的列数
        * 2.getColumnLabel():返回列的别名
        */
        ResultSetMetaData rsmd = rs.getMetaData();
        int columnCount = rsmd.getColumnCount();
        while (rs.next()) {
            T t = clazz.newInstance();
            5
        }
    }
}
```

```
for (int i = 0; i < columnCount; i++) {  
    Object columnVal = rs.getObject(i + 1); // 相应列的值  
    //String columnName = rsmd.getColumnName(i + 1);  
    String columnName = rsmd.getColumnLabel(i + 1);  
    //使用 PropertyUtils 将指定对象 t 的指定属性 columnName 设置为指定的值  
    columnVal  
    PropertyUtils.setProperty(t, columnName, columnVal);  
}  
list.add(t);  
}  
} catch (Exception e) {  
    e.printStackTrace();  
} finally {  
    JDBCUtils.close(rs, st, conn);  
}  
return list;  
}
```

流程:

order_id	order_name	order_date
1	AA	2010-03-04
2	BB	2000-02-01
4	GG	1994-06-28
5	CC	1998-09-08
6	DD	1998-09-08

----> String sql = "select order_id id,order_name name,order_date date from `order` where order_id = 1";

id	name	date
1	AA	2010-03-04

得到一个ResultSet rs = Connection.getConnection().createStatement().executeQuery(sql);

——> 得到结果集的元数据: ResultSetMetaData rmsd = rs.getMetaData();

id	name	date
1	AA	2010-03-04

调用rmsd.getColumnCount():获取结果集有多少列; rmsd.getColumnLabel():获取相应列的别名。

注: 当表的列名与类的属性名不一致时(如: order表的order_id与Order类的id属性对应, 但名不一样), 通常在select语句中, 使用列的别名指明对应的类的属性名。

//使用PropertyUtils将指定对象t的指定属性columnName设置为指定的值columnVal
PropertyUtils.setProperty(t, columnName, columnVal);

//总结:

两种思想:

➤ 面向接口编程的思想;

➤ **ORM 思想:**

* ORM:Object Relational Mapping

* 数据库中的表与 java 中的一个类对应(如: customers 表与 Customer 类对应)

* 数据库中表的一个列与 java 类的一个属性对应(如: 表中的 id 列与 Customer

类的 id 属性对应)

* 数据库中表的一行（一条数据）与 java 类的一个对象对应

两个技术:

- 结果集的元数据: **ResultSetMetaData**;
- **PropertyUtils**

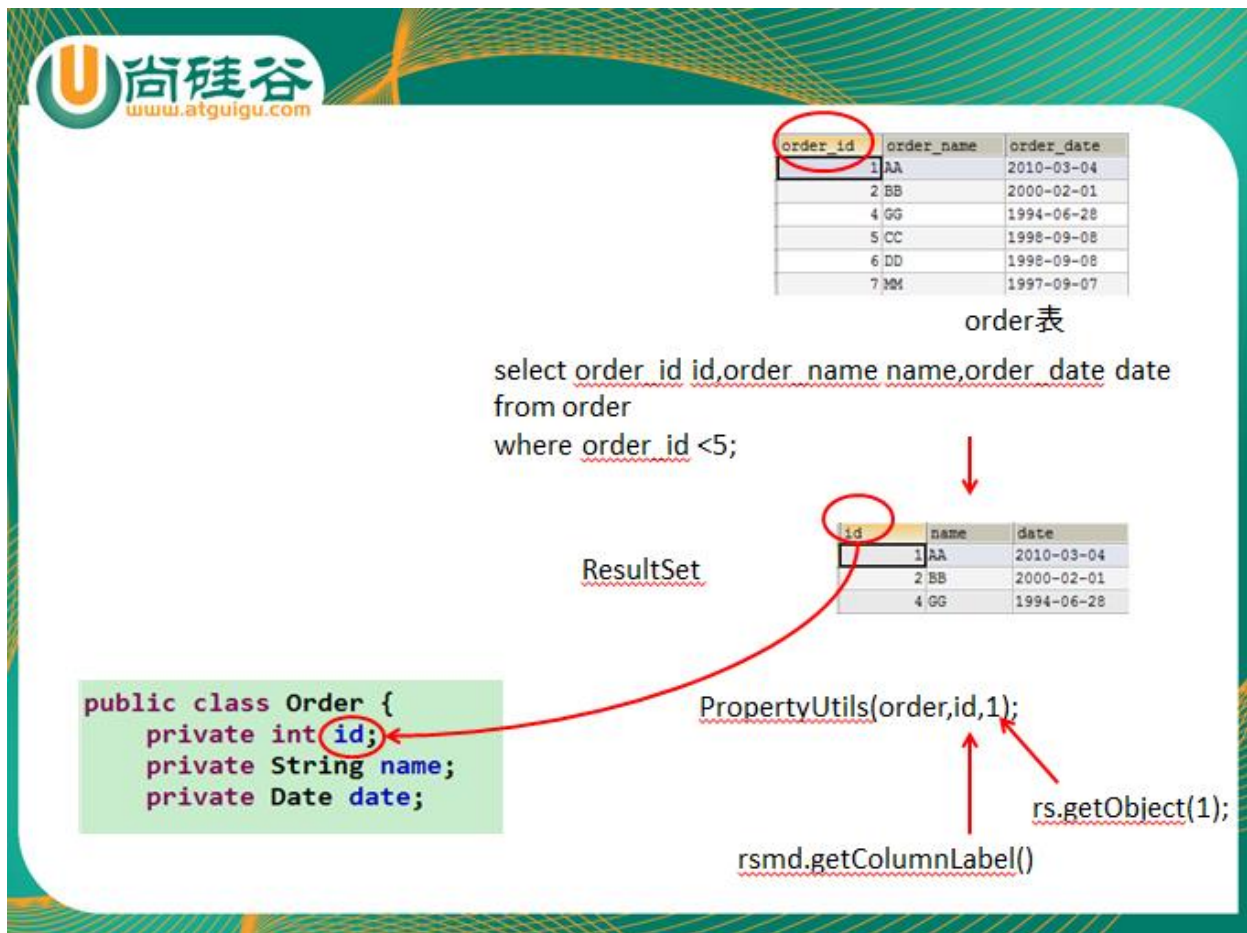
1.结果集的元数据: **ResultSetMetaData**

//获取: **ResultSet.getMetaData()**;

//两个方法: 1)**getColumnCount()**:获取结果集中有多少列

2)**getColumnLabel()**:获取结果集的相应列的列名, 相当于是对应的表的列的别名。

--**getColumnName()**:不用。



```
public void testResultSetMetaData(){  
    Connection conn = null;  
    Statement st = null;  
    ResultSet rs = null;  
    String sql = "select order_id id,order_name name,order_date date from `order`";  
    try{
```



```
conn = JDBCUtils.getConnection();
st = conn.createStatement();
rs = st.executeQuery(sql);

ResultSetMetaData rsmd = rs.getMetaData();
int columnCount = rsmd.getColumnCount();
System.out.println(columnCount);
while(rs.next()){
    for(int i = 0;i < columnCount;i++){
        System.out.print(rsmd.getColumnName(i + 1) + " ");
        System.out.print(rsmd.getColumnLabel(i + 1) + " ");
        System.out.println(rs.getObject(i + 1));
    }
    System.out.println();
}

}

}catch(Exception e){
    e.printStackTrace();
}finally{
    JDBCUtils.close(rs, st, conn);
}

}
```

2. PropertyUtils 工具类，使用它的 setProperty(Object obj,String fieldName,Object fieldValue)

```
public void testPropertyUtils() throws Exception{
    Order order = new Order();
    System.out.println(order);

    PropertyUtils.setProperty(order, "id", 1001);
    PropertyUtils.setProperty(order, "name", "AA");
    PropertyUtils.setProperty(order, "date", new Date(new java.util.Date().getTime()));

    System.out.println(order);
}
```

四、PreparedStatement

PreparedStatement 是 Statement 的子接口

- ①需要预编译 SQL 语句: `PreparedStatement ps = conn.prepareStatement(sql);`
- ②填充占位符: `setObject(int index);`//index 从 1 开始
- ③`execute()` / `executeUpdate()` ; `executeQuery()`; 返回一个 `ResultSet`

1. 替换原来的 `Statement`, 实现增删改和查的操作

-->`Statement` 的问题: ①拼串 不方便, 容易出错 ②存在 sql 注入的问题, 可以对数据库进行恶意攻击。

// 实现一个通用的 UPDATE INSERT DELETE 的操作的方法(version 2.0)

```
public void update(String sql, Object... args) {
    Connection conn = null;
    PreparedStatement ps = null;
    try {
        // 1. 获取连接
        conn = JDBCUtils.getConnection();
        // 2. 返回 PreparedStatement 对象, 预编译 sql 语句
        ps = conn.prepareStatement(sql);
        // 3. 填充占位符
        for (int i = 0; i < args.length; i++) {
            ps.setObject(i + 1, args[i]);
        }

        ps.execute();
    } catch (Exception e) {
        e.printStackTrace();
    } finally {
        JDBCUtils.close(null, ps, conn);
    }
}
```

// 实现一个通用的查询操作, 返回一个对象(version 2.0)

```
public <T> T getInstance(String sql, Class<T> clazz, Object... args) {
    Connection conn = null;
    PreparedStatement ps = null;
    ResultSet rs = null;
    try {
        // 1. 获取连接
        conn = JDBCUtils.getConnection();
        // 2. 预编译 sql 语句, 返回 PreparedStatement 对象
        ps = conn.prepareStatement(sql);
        // 3. 填充占位符
        for (int i = 0; i < args.length; i++) {
```

```
ps.setObject(i + 1, args[i]);
}
// 4.执行并返回 ResultSet 的对象
rs = ps.executeQuery();

if (rs.next()) {
// 5.创建 T 的对象
T t = clazz.newInstance();
// 6.将结果集中的列值作为 T 的对象的属性，给予赋值
ResultSetMetaData rsmd = rs.getMetaData();
int columnCount = rsmd.getColumnCount();
for (int i = 0; i < columnCount; i++) {
Object columnVal = rs.getObject(i + 1);
String columnLabel = rsmd.getColumnLabel(i + 1);
PropertyUtils.setProperty(t, columnLabel, columnVal);
}
return t;
}

} catch (Exception e) {
e.printStackTrace();
} finally {
// 7.关闭相应的操作
JDBCUtils.close(rs, ps, conn);
}
return null;
}

// 实现一个通用的查询操作,返回一个对象的集合(version 2.0)
public <T> List<T> getForList(String sql,Class<T> clazz,Object ... args){
Connection conn = null;
PreparedStatement ps = null;
ResultSet rs = null;
List<T> list = new ArrayList<T>();

try{
conn = JDBCUtils.getConnection();
ps = conn.prepareStatement(sql);

for(int i = 0;i < args.length;i++){
ps.setObject(i + 1, args[i]);
}
```

```
rs = ps.executeQuery();
ResultSetMetaData rsmd = rs.getMetaData();
int columnCount = rsmd.getColumnCount();
while(rs.next()){
    T t = clazz.newInstance();

    for(int i = 0;i < columnCount;i++){
        Object columnVal = rs.getObject(i + 1);
        String columnLabel = rsmd.getColumnLabel(i + 1);

        PropertyUtils.setProperty(t, columnLabel, columnVal);
    }
    list.add(t);
}

}catch(Exception e){
    e.printStackTrace();
}finally{
    JDBCUtils.close(rs, ps, conn);
}
return list;
}
```

//2.使用 PreparedStatement 的其他优点

1.实现大数据类型的数据的插入、修改、查询的操作.

setBlob() getBlob();

// 从数据表中将大数据类型的数据取出

@Test

```
public void testBlob3(){
    Connection conn = null;
    PreparedStatement ps = null;
    String sql = "select id,name,email,birth,photo from customers where id = ?";
    ResultSet rs = null;
    InputStream is = null;
    FileOutputStream fos = null;
    try{
        conn = JDBCUtils.getConnection();
        ps = conn.prepareStatement(sql);
        fos = new FileOutputStream("ym1.jpg");
        ps.setInt(1, 21);
```

```
rs = ps.executeQuery();

if(rs.next()){
    int id = rs.getInt("id");
    String name = rs.getString("name");
    Date birth = rs.getDate("birth");
    String email = rs.getString("email");
    Customer cust = new Customer(id,name,email,birth);
    System.out.println(cust);
}

Blob photo = rs.getBlob(5);

is = photo.getBinaryStream();
byte[] b = new byte[1024];
int len;
while((len = is.read(b)) != -1){
    fos.write(b, 0, len);
}

}catch (Exception e) {
    e.printStackTrace();
} finally {
    JDBCUtils.close(rs, ps, conn);

    if(fos != null){
        try {
            fos.close();
        } catch (IOException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }
    if(is != null){
        try {
            is.close();
        } catch (IOException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }
}
```

```
}  
}
```

```
// 向数据表中修改现有的大数据类型的数据
```

```
@Test
```

```
public void testBlob2() {
```

```
    Connection conn = null;
```

```
    PreparedStatement ps = null;
```

```
    String sql = "update customers set photo = ? where id = ?";
```

```
    try {
```

```
        conn = JDBCUtils.getConnection();
```

```
        ps = conn.prepareStatement(sql);
```

```
        ps.setBlob(1, new FileInputStream("ym.jpg"));
```

```
        ps.setInt(2, 21);
```

```
        ps.execute();
```

```
    } catch (Exception e) {
```

```
        e.printStackTrace();
```

```
    } finally {
```

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

```
    }
```

```
}
```

```
// 向数据库的表中写入大数据类型的数据
```

```
@Test
```

```
public void testBlob1() {
```

```
    Connection conn = null;
```

```
    PreparedStatement ps = null;
```

```
    String sql = "insert into customers(name,email,birth,photo)values(?,?,?,?)";
```

```
    try {
```

```
        conn = JDBCUtils.getConnection();
```

```
        ps = conn.prepareStatement(sql);
```

```
        ps.setString(1, "杨幂 1");
```

```
        ps.setString(2, "yang@126.com");
```

```
        ps.setDate(3, new Date(new java.util.Date().getTime()));
```

```
        ps.setBlob(4, new FileInputStream("1.jpg"));
```

```
        ps.execute();
```

```
} catch (Exception e) {  
    e.printStackTrace();  
} finally {  
    JDBCUtils.close(null, ps, conn);  
}
```

```
}
```

2.使用 PreparedStatement 进行批量操作时，效率优于 Statement.

//批量操作，主要指的是批量插入。

//oracle 是支持批量插入的。

//如何实现最优？ ①使用 PreparedStatement ②addBatch() executeBatch()

clearBatch()

```
public void test4() {  
    Connection conn = null;  
    PreparedStatement ps = null;  
    long start = System.currentTimeMillis();  
    String sql = "insert into dept values(?,?)";  
    try {  
        conn = JDBCUtils.getConnection();  
        ps = conn.prepareStatement(sql);  
        for (int i = 0; i < 100000; i++) {  
            ps.setInt(1, i + 1);  
            ps.setString(2, "dept_" + (i + 1) + "_name");  
            //1. “攒” SQL  
            ps.addBatch();  
            if (i + 1) % 250 == 0 {  
                //2.执行 sql  
                ps.executeBatch();  
                //3.清空 sql  
                ps.clearBatch();  
            }  
        }  
    }  
}
```

```
} catch (Exception e) {  
    e.printStackTrace();  
} finally {  
    JDBCUtils.close(null, ps, conn);  
}  
long end = System.currentTimeMillis();  
System.out.println("花费时间: " + (end - start)); //2427  
}
```

五、数据库的元数据：DataBaseMetaData（了解）

“元”数据： String name = "AA";

ResultSet :结果集

ResultSetMetaData: 结果集的元数据

DatabaseMetaData:数据库的元数据

```
public class TestDataBaseMetaData {  
    public static void main(String[] args) {  
        Connection conn = null;  
        DatabaseMetaData dbmd = null;  
        ResultSet rs = null;  
        try{  
            conn = JDBCUtils.getConnection();  
            //获取数据库的元数据  
            dbmd = conn.getMetaData();  
            //以字符串的形式返回数据库的名字  
            System.out.println(dbmd.getDatabaseProductName());  
            //返回数据库的版本号  
            System.out.println(dbmd.getDatabaseProductVersion());  
  
            rs = dbmd.getCatalogs();  
            //返回含有的各个数据库的名字  
            while(rs.next()){  
                String databaseName = rs.getString(1);  
                System.out.println(databaseName);  
            }  
  
        } catch (Exception e) {  
            e.printStackTrace();  
        } finally {  
            JDBCUtils.close(rs, null, conn);  
        }  
    }  
}
```

六、数据库事务（重点）

1.事务：一组逻辑操作单元,使数据从一种状态变换到另一种状态。

2.事务处理的原则：保证所有事务都作为一个工作单元来执行，即使出现了故障，都不能改变这种执行方式。

当在一个事务中执行多个操作时，要么所有的事务都被提交(commit)，那么这些修改就永久地保存下来；

要么数据库管理系统将放弃所作的所有修改，整个事务回滚(rollback)到最初状态。

// 以下的两个操作共同构成一个数据库事务。但是在两个操作之间可能出现异常问题。

// 原则上，一旦出现问题，就需要将之前的操作“回滚”！需要对如下的操作进行完善。

@Test

```
public void testUpdate() {
    String sql1 = "update user_table set balance = balance - 100 where user = ?";
    update(sql1, "AA");

    System.out.println(10 / 0);

    String sql2 = "update user_table set balance = balance + 100 where user = ?";
    update(sql2, "BB");
}
```

3.考虑到数据库事务的话，我们又将原来使用 PreparedStatement 重构的 Statement 的增删改和查的操作，再升级。

// 实现一个通用的 UPDATE INSERT DELETE 的操作的方法(version 3.0)

```
public void update(Connection conn,String sql, Object... args) {
    PreparedStatement ps = null;
    try {

        ps = conn.prepareStatement(sql);

        for (int i = 0; i < args.length; i++) {
            ps.setObject(i + 1, args[i]);
        }

        ps.execute();
    } catch (Exception e) {
        e.printStackTrace();
    } finally {
        JDBCUtils.close(null, ps, null);
    }
}
```

// 实现一个通用的查询操作,返回一个对象(version 3.0)

```
public <T> T getInstance(Connection conn,String sql, Class<T> clazz, Object... args) {
```

```
PreparedStatement ps = null;
ResultSet rs = null;
try {

    ps = conn.prepareStatement(sql);
    // 填充占位符
    for (int i = 0; i < args.length; i++) {
        ps.setObject(i + 1, args[i]);
    }
    // 4.执行并返回 ResultSet 的对象
    rs = ps.executeQuery();

    if (rs.next()) {
        // 5.创建 T 的对象
        T t = clazz.newInstance();
        // 6.将结果集中的列值作为 T 的对象的属性，给予赋值
        ResultSetMetaData rsmd = rs.getMetaData();
        int columnCount = rsmd.getColumnCount();
        for (int i = 0; i < columnCount; i++) {
            Object columnVal = rs.getObject(i + 1);
            String columnLabel = rsmd.getColumnLabel(i + 1);
            PropertyUtils.setProperty(t, columnLabel, columnVal);
        }
        return t;
    }

} catch (Exception e) {
    e.printStackTrace();
} finally {
    // 7.关闭相应的操作
    JDBCUtils.close(rs, ps, null);
}
return null;
}

// 实现一个通用的查询操作,返回一个对象的集合(version 3.0)
public <T> List<T> getForList(Connection conn,String sql,Class<T> clazz,Object ...
    args){
    PreparedStatement ps = null;
    ResultSet rs = null;
    List<T> list = new ArrayList<T>();

    try{
```

```
ps = conn.prepareStatement(sql);

for(int i = 0;i < args.length;i++){
    ps.setObject(i + 1, args[i]);
}

rs = ps.executeQuery();
ResultSetMetaData rsmd = rs.getMetaData();
int columnCount = rsmd.getColumnCount();
while(rs.next()){
    T t = clazz.newInstance();

    for(int i = 0;i < columnCount;i++){
        Object columnVal = rs.getObject(i + 1);
        String columnLabel = rsmd.getColumnLabel(i + 1);

        PropertyUtils.setProperty(t, columnLabel, columnVal);
    }
    list.add(t);
}

}catch(Exception e){
    e.printStackTrace();
}finally{
    JDBCUtils.close(rs, ps, null);
}
return list;
}

//考虑到数据库事务，通过 java 程序对数据库中表的操作的模板（掌握）
public void method(){
    Connection conn = null;
    try{
        //1.获取数据库的连接（①conn = JDBCUtils.getConnection(); ②数据库连接池（开发者选择此）
        //2.开启事务
        conn.setAutoCommit(false);
        //3.对数据库中表进行相应的操作(增、删、改、查)（①version 3.0 ②
```

DBUtils 工具类：update() 和 query()方法)

```
//4.提交事务
conn.commit();

}catch(Exception e){
    e.printStackTrace();
    try{
        //5.回滚事务
        conn.rollback();
    }catch(Exception e1){
        e1.printStackTrace();
    }
}finally{
    //6.关闭数据库的连接（①自己实现数据库相应资源的关闭
    JDBCUtils.close(null,null,conn); ②使用 DBUtils 工具类的 close()方法）
}
}
```

了解：

● 事务的ACID(acid)属性 BAT ATM

➤ 1. 原子性 (Atomicity)

原子性是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。

➤ 2. 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。

➤ 3. 隔离性 (Isolation)

事务的隔离性是指一个事务的执行不能被其他事务干扰，即一个事务内部的操作及使用的数据对并发的其他事务是隔离的，并发执行的各个事务之间不能互相干扰。

➤ 4. 持久性 (Durability)

持久性是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，接下来的其他操作和数据库故障不应该对其有任何影响

针对隔离性，我们提供了不同数据库处理的隔离级别，针对处理数据发生的问题（脏读、不可重复读、幻读）提供了多张隔离级别

对于同时运行的多个事务,当这些事务访问数据库中相同的数据时,如果没有采取必要的隔离机制,就会导致各种并发问题:

- **脏读**: 对于两个事务 T1, T2, T1 读取了已经被 T2 更新但还没有被提交的字段。之后,若 T2 回滚, T1 读取的内容就是临时且无效的。
- **不可重复读**: 对于两个事务 T1, T2, T1 读取了一个字段,然后 T2 更新了该字段。之后, T1 再次读取同一个字段,值就不同了。
- **幻读**: 对于两个事务 T1, T2, T1 从一个表中读取了一个字段,然后 T2 在该表中插入了一些新的行。之后,如果 T1 再次读取同一个表,就会多出几行。

隔离级别: (脏读是我们一定哟啊避免的, 而不可重复读、幻读是允许存在的)



数据库的隔离级别

● 数据库提供的 4 种事务隔离级别:

隔离级别	描述
READ UNCOMMITTED (读未提交数据)	允许事务读取未被其他事物提交的变更。脏读,不可重复读和幻读的问题都会出现
READ COMMITTED (读已提交数据)	只允许事务读取已经被其它事务提交的变更。可以避免脏读,但不可重复读和幻读问题仍然可能出现
REPEATABLE READ (可重复读)	确保事务可以多次从一个字段中读取相同的值。在这个事务持续期间,禁止其他事物对这个字段进行更新。可以避免脏读和不可重复读,但幻读的问题仍然存在。
SERIALIZABLE(串行化)	确保事务可以从一个表中读取相同的行。在这个事务持续期间,禁止其他事务对该表执行插入,更新和删除操作。所有并发问题都可以避免,但性能十分低下。

- Oracle 支持的 2 种事务隔离级别: **READ COMMITTED**, **SERIALIZABLE**。Oracle 默认的事务隔离级别为: **READ COMMITTED**
- Mysql 支持 4 种事务隔离级别. Mysql 默认的事务隔离级别为: **REPEATABLE READ**

七、实现 DAO 及其实现类 CustomerDAO 的代码

【DAO.java】

//DAO: database access object

```
class ReflectionUtils{  
    //获取 clazz 对象对应的运行时类的父类的泛型
```

```
public static Class getSuperGeneric(Class clazz){
    Type type = clazz.getGenericSuperclass();
    ParameterizedType p = (ParameterizedType)type;
    Type[] ts = p.getActualTypeArguments();

    return (Class)ts[0];
}

public class DAO<T> {
    private Class<T> clazz = null;
    //this.getClass()在这个问题中，就是 CustomerDAO
    public DAO(){
        clazz = ReflectionUtils.getSuperGeneric(this.getClass());
    }
    //获取数据库的标准的特定含义的值
    public <E> E getValue(Connection conn,String sql,Object...args){
        PreparedStatement ps = null;
        ResultSet rs = null;
        try{
            ps = conn.prepareStatement(sql);

            for(int i = 0;i < args.length;i++){
                ps.setObject(i + 1, args[i]);
            }

            rs = ps.executeQuery();

            if(rs.next()){
                return (E)rs.getObject(1);
            }

        }catch(Exception e){
            e.printStackTrace();
        }finally{
            JDBCUtils.close(rs, ps, null);
        }

        return null;
    }
}
```

//返回多个对象，以集合的形式返回

```
public List<T> getForList(Connection conn,String sql,Object ...args){
    PreparedStatement ps = null;
    ResultSet rs = null;
    List<T> list = new ArrayList<>();
    try{
        //1.预编译 sql 语句，获取 PreparedStatement 对象
        ps = conn.prepareStatement(sql);
        //2.填充占位符
        for(int i = 0;i < args.length;i++){
            ps.setObject(i + 1, args[i]);
        }
        //3.返回一个结果集
        rs = ps.executeQuery();
        ResultSetMetaData rsmd = rs.getMetaData();
        int columnCount = rsmd.getColumnCount();
        while(rs.next()){
            T t = clazz.newInstance();

            //给 t 对象的相应属性赋值
            for(int i = 0;i < columnCount;i++){
                Object columnVal = rs.getObject(i + 1);
                String columnLabel = rsmd.getColumnLabel(i + 1);
                PropertyUtils.setProperty(t, columnLabel, columnVal);
            }
            list.add(t);
        }

    }catch(Exception e){
        e.printStackTrace();
    }finally{
        JDBCUtils.close(rs, ps, null);
    }
    //System.out.println(clazz);
    return list;
}
```

//返回一个对象

```
public T get(Connection conn,String sql,Object ...args){
    PreparedStatement ps = null;
    ResultSet rs = null;
    try{
        //1.预编译 sql 语句，获取 PreparedStatement 对象
```



```
        ps = conn.prepareStatement(sql);
        //2.填充占位符
        for(int i = 0;i < args.length;i++){
            ps.setObject(i + 1, args[i]);
        }
        //3.返回一个结果集
        rs = ps.executeQuery();
        ResultSetMetaData rsmd = rs.getMetaData();
        int columnCount = rsmd.getColumnCount();
        if(rs.next()){
            T t = clazz.newInstance();

            //给 t 对象的相应属性赋值
            for(int i = 0;i < columnCount;i++){
                Object columnVal = rs.getObject(i + 1);
                String columnLabel = rsmd.getColumnLabel(i + 1);
                PropertyUtils.setProperty(t, columnLabel, columnVal);
            }
            return t;
        }

    }catch(Exception e){
        e.printStackTrace();
    }finally{
        JDBCUtils.close(rs, ps, null);
    }
    //System.out.println(clazz);
    return null;
}

//通用的增删改的操作
public void update(Connection conn,String sql,Object ... args){
    PreparedStatement ps = null;
    try{
        ps = conn.prepareStatement(sql);

        for(int i = 0;i < args.length;i++){
            ps.setObject(i + 1, args[i]);
        }

        ps.executeUpdate();
    }catch(Exception e){
```

```
        e.printStackTrace();
    }finally{
        JDBCUtils.close(null, ps, null);
    }
}
}
```

【CustomerDAO.java】

//CustomerDAO 类是用来操作 Customer 类的

```
public class CustomerDAO extends DAO<Customer>{
```

```
// @Test
// public void testGeneric(){
//     Class clazz = CustomerDAO.class;
//     Type type = clazz.getGenericSuperclass();
//     ParameterizedType p = (ParameterizedType)type;
//     Type[] ts = p.getActualTypeArguments();
//     System.out.println(ts[0]);
// }
/**
 * 获取对应的表中的记录的个数
 */
public long getCount(Connection conn){
    String sql = "select count(*) from customers";
    return (long)getValue(conn, sql);
}

/**
 * 返回 customers 表中的所有数据
 * @param conn
 * @return
 */
public List<Customer> getAll(Connection conn){
    String sql = "select id,name,email,birth from customers";
    return getForList(conn, sql);
}

/**
 * 根据指定的 id 返回相应的对象
 * @param conn
 * @param customerId
 */
public Customer getInstance(Connection conn,int customerId){
```

```
String sql = "select id,name,email,birth from customers where id = ?";
return get(conn, sql, customerId);
}

/**
 * 删除指定 customerId 的数据表中的记录
 * @param conn
 * @param customerId
 */
public void delete(Connection conn,int customerId){
    String sql = "delete from customers where id = ?";
    update(conn, sql, customerId);
}

/**
 * 向数据表中修改指定 id 的信息为 Customer 对象的信息
 * @param conn
 * @param cust
 */
public void update(Connection conn, Customer cust){
    String sql = "update customers set name = ?,email = ?,birth = ? where id
= ?";
    update(conn, sql,
cust.getName(),cust.getEmail(),cust.getBirth(),cust.getId());
}

/**
 * 向数据表中插入一条数据
 * @param conn 数据库的连接
 * @param cust 要插入的 Customer 对象
 */
public void insert(Connection conn, Customer cust){
    String sql = "insert into customers(name,email,birth)values(?,?,?)";
    update(conn, sql, cust.getName(),cust.getEmail(),cust.getBirth());
}
}

【TestCustomerDAO.java】
public class TestCustomerDAO {
    CustomerDAO customerDAO = new CustomerDAO();
}
```

```
@Test
public void testGetCount(){
    Connection conn = null;
    try{
        conn = JDBCUtils.getConnection();
        long count = customerDAO.getCount(conn);
        System.out.println(count);

    }catch(Exception e){
        e.printStackTrace();
    }finally{
        JDBCUtils.close(null, null, conn);
    }
}

@Test
public void testGetAll(){
    Connection conn = null;
    try{
        conn = JDBCUtils.getConnection();
        List<Customer> list = customerDAO.getAll(conn);
        //System.out.println(list);

        Iterator<Customer> iterator = list.iterator();
        while(iterator.hasNext()){
            System.out.println(iterator.next());
        }

    }catch(Exception e){
        e.printStackTrace();
    }finally{
        JDBCUtils.close(null, null, conn);
    }
}

@Test
public void testQuery(){
    Connection conn = null;
    try{
        conn = JDBCUtils.getConnection();
        Customer cust = customerDAO.getInstance(conn, 13);
```

```
        System.out.println(cust);
    }catch(Exception e){
        e.printStackTrace();
    }finally{
        JDBCUtils.close(null, null, conn);
    }
}

@Test
public void testDelete(){
    Connection conn = null;
    try{
        conn = JDBCUtils.getConnection();
        customerDAO.delete(conn, 10);
    }catch(Exception e){
        e.printStackTrace();
    }finally{
        JDBCUtils.close(null, null, conn);
    }
}

@Test
public void testUpdate(){
    Connection conn = null;
    try{
        conn = JDBCUtils.getConnection();
        Customer cust = new Customer(10, "张卫健", "zwj@gmail.com", new
Date(new java.util.Date().getTime()));
        customerDAO.update(conn, cust);
    }catch(Exception e){
        e.printStackTrace();
    }finally{
        JDBCUtils.close(null, null, conn);
    }
}

@Test
public void testInsert(){
    Connection conn = null;
    try{
        conn = JDBCUtils.getConnection();
        Customer cust = new Customer(10, "张卫健", "zwj@gmail.com", new
```

```
Date(new java.util.Date().getTime()));
        customerDAO.insert(conn, cust);
    }catch(Exception e){
        e.printStackTrace();
    }finally{
        JDBCUtils.close(null, null, conn);
    }
}
}
```

八、数据库连接池

C3P0 数据库连接池

// 保证在所有的通过 C3P0 获取的连接中，只有一个 DataSource 的对象。(推荐)

```
private static DataSource source = null;
```

```
static {
    source = new ComboPooledDataSource("helloc3p0");
}
```

// 获取数据库的连接方式 3:使用 c3p0 数据库连接池获取数据库的连接,使用配置文件

```
public static Connection getConnection3() throws Exception {
    return source.getConnection();
}
```

对应的配置文件: c3p0-config.xml

<c3p0-config>

<named-config name="helloc3p0">

<!-- 提供数据库连接的 4 个基本信息 -->

<property name="jdbcUrl">jdbc:mysql:///test</property>

<property name="driverClass">com.mysql.jdbc.Driver</property>

<property name="user">root</property>

<property name="password">123456</property>

<!-- 当连接池中的数量不足时，c3p0 连接一次性向数据库服务器申请的连接数 -->

<property name="acquireIncrement">5</property>

<!-- 初始化数据库连接池时，池中存在的连接数 -->

<property name="initialPoolSize">10</property>

<!-- 数据库连接池中最少容纳的连接数 -->

<property name="minPoolSize">5</property>

<!-- 数据库连接池中最大容纳的连接数 -->

<property name="maxPoolSize">100</property>

```
<!-- 连接池中，最多允许存在的 Statement 的数量 -->
<property name="maxStatements">10</property>
<!-- 一次连接中，最多容纳的 Statement 的个数 -->
<property name="maxStatementsPerConnection">5</property>

</named-config>
</c3p0-config>
```

DBCP 数据库连接池

//随着类的加载,使用 BasicDataSourceFactory 的静态方法 createDataSource()返回一个

//DataSource 的对象

```
private static DataSource source1 = null;
static {
    Properties info = new Properties();
    // info.load(new FileInputStream("dbcp.properties"));
    InputStream is = JDBCUtils.class.getClassLoader().getResourceAsStream(
        "com/atguigu/java/dbcp.properties");
    try {
        info.load(is);
        source1 = BasicDataSourceFactory.createDataSource(info);
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

// 获取数据库的连接方式 4:使用 DBCP 数据库连接池获取数据库的连接（推荐）

```
public static Connection getConnection5() throws Exception {

    return source1.getConnection();
}
```

配置文件 dbcp.properties:

```
username=root
password=123456
url=jdbc:mysql://127.0.0.1:3306/test
driverClassName=com.mysql.jdbc.Driver
```

```
initialSize=10
maxActive=100
```


九、DBUtils

提供了 QueryRunner 类，类中有诸多重载 update() 和 query() 方法，供使用，用于堆数据库实现操作：增删改查

```
public class TestDBUtils {
    QueryRunner runner = new QueryRunner();

    // ScalarHandler: 用于查询其他的一些信息，比如表中的记录数，薪资最高的员工姓名。
    // 因为返回的类型不确定，故使用 Object 接收
    @Test
    public void testScalarHandler() {
        Connection conn = null;
        try {
            conn = JDBCUtils.getConnection5();
            //String sql = "select count(*) from customers";
            String sql = "select min(birth) from customers";
            ResultSetHandler<Object> handler = new ScalarHandler();

            //long count = (long) runner.query(conn, sql, handler);
            Date d = (Date)runner.query(conn, sql, handler);
            System.out.println(d);
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            // “关闭连接”，实际上是对 conn 的释放
            JDBCUtils.close(null, null, conn);
        }
    }

    // MapListHandler
    @Test
    public void testMapListHandler() {
        Connection conn = null;
        try {
            conn = JDBCUtils.getConnection5();
            String sql = "select id,name,email,birth from customers where id < ?";

            ResultSetHandler<List<Map<String, Object>>> handler = new
MapListHandler();
            // MapListHandler handler = new MapListHandler();
        }
    }
}
```

```
List<Map<String, Object>> maps = runner.query(conn, sql, handler,
12);

System.out.println(maps);

} catch (Exception e) {
    e.printStackTrace();
} finally {
    // “关闭连接”，实际上是对 conn 的释放
    JDBCUtils.close(null, null, conn);
}
}

// MapHandler:返回数据库中一条记录，key 为记录的列的列名，value 为记录
// 的列的值。一条记录
// 的多个列构成了多个 key-value，组成了一个 map
@Test
public void testMapHandler() {
    Connection conn = null;
    try {
        conn = JDBCUtils.getConnection5();
        String sql = "select id,name,email,birth from customers where id = ?";

        // ResultSetHandler<Map<String,Object>> handler = new
MapHandler();
        MapHandler handler = new MapHandler();
        Map<String, Object> map = runner.query(conn, sql, handler, 22);

        System.out.println(map);

    } catch (Exception e) {
        e.printStackTrace();
    } finally {
        // “关闭连接”，实际上是对 conn 的释放
        JDBCUtils.close1(null, null, conn);
    }
}

// 实现对数据库中表的查询的操作：返回表中的多条记录，用 List 接口接收。
使用的是：
```

```
// ResultSetHandler 的实现类： BeanListHandler
@Test
```

```
public void testQueryList() {
    Connection conn = null;
    try {
        conn = JDBCUtils.getConnection5();
        String sql = "select id,name,email,birth from customers where id < ?";
        // 使用 BeanListHandler 接收多条记录, 以多个对象组成的 list 返回。
        ResultSetHandler<List<Customer>> handler = new BeanListHandler<>(
            Customer.class);
        List<Customer> custs = runner.query(conn, sql, handler, 15);
        // System.out.println(custs);
        Iterator<Customer> iterator = custs.iterator();
        while (iterator.hasNext()) {
            System.out.println(iterator.next());
        }
    } catch (Exception e) {
        e.printStackTrace();
    } finally {
        // “关闭连接”, 实际上是对 conn 的释放
        JDBCUtils.close(null, null, conn);
    }
}
```

// 自定义一个 ResultSetHandler 的实现类, 实现与 BeanHandler 一样的功能,
用于返回

// 数据库中表的一条记录

@Test

```
public void testSelfHandler() {
    Connection conn = null;
    try {
        conn = JDBCUtils.getConnection5();
        String sql = "select id,name,email,birth from customers where id = ?";
        ResultSetHandler<Customer> handler = new ResultSetHandler() {

            @Override
            public Object handle(ResultSet rs) throws SQLException {
                Customer cust = null;
                if (rs.next()) {
                    int id = rs.getInt(1);
                    String name = rs.getString(2);
                    String email = rs.getString(3);
                    Date birth = rs.getDate(4);
                    cust = new Customer(id, name, email, birth);
                }
                return cust;
            }
        };
    } catch (Exception e) {
        e.printStackTrace();
    } finally {
        JDBCUtils.close(null, null, conn);
    }
}
```

```
        }  
        return cust;  
    }  
  
    };  
    Customer cust = runner.query(conn, sql, handler, 22);  
    System.out.println(cust);  
} catch (Exception e) {  
    e.printStackTrace();  
} finally {  
    // “关闭连接”，实际上是对 conn 的释放  
    JDBCUtils.close(null, null, conn);  
}  
}
```

// 实现对数据库中表的查询的操作：返回表中的一条记录，对应着一个对象使用的是：

// ResultSetHandler 的实现类：BeanHandler

@Test

```
public void testQuery1() {  
    Connection conn = null;  
    try {  
        conn = JDBCUtils.getConnection5();  
        String sql = "select id,name,email,birth from customers where id = ?";  
        ResultSetHandler<Customer> handler = new BeanHandler<>(  
            Customer.class);  
        Customer cust = runner.query(conn, sql, handler, 22);  
        System.out.println(cust);  
    } catch (Exception e) {  
        e.printStackTrace();  
    } finally {  
        // “关闭连接”，实际上是对 conn 的释放  
        JDBCUtils.close(null, null, conn);  
    }  
}
```

@Test

```
public void testDelete() {  
    Connection conn = null;  
    try {  
        // 使用 DBCP 数据库连接池，获取数据库的连接  
        conn = JDBCUtils.getConnection5();  
    }  
}
```

```
String sql = "delete from customers where id = ?";  
// 对数据库中表的 INSERT UPDATE DELETE 操作，都是调用 runner 的  
update()方法
```

```
runner.update(conn, sql, 23);  
  
    } catch (Exception e) {  
        e.printStackTrace();  
    } finally {  
        // “关闭连接”，实际上是对 conn 的释放  
        JDBCUtils.close(null, null, conn);  
    }  
}  
  
@Test  
public void testInsert() {  
    Connection conn = null;  
    try {  
        // 使用 DBCP 数据库连接池，获取数据库的连接  
        conn = JDBCUtils.getConnection5();  
        String sql = "insert into customers(name,email,birth) values (?,?,:)";  
        runner.update(conn, sql, "张嘉译", "zjy@gmail.com", new Date(  
            new java.util.Date().getTime()));  
  
    } catch (Exception e) {  
        e.printStackTrace();  
    } finally {  
        // “关闭连接”，实际上是对 conn 的释放  
        JDBCUtils.close(null, null, conn);  
    }  
}  
}
```

DBUtils 类

DBUtils 类中有 close(ResultSet rs) 、 close(Statement st) 、 close(Connection conn)
用于释放数据库的连接

```
public static void close1(ResultSet rs, Statement st, Connection conn){  
    try {  
        DbUtils.close(rs);  
    } catch (SQLException e) {  
        // TODO Auto-generated catch block  
        e.printStackTrace();  
    }  
}
```

```
}
try {
    DbUtils.close(st);
} catch (SQLException e) {
    // TODO Auto-generated catch block
    e.printStackTrace();
}
try {
    DbUtils.close(conn);
} catch (SQLException e) {
    // TODO Auto-generated catch block
    e.printStackTrace();
}
}
```

重构 DAO

//返回一个对象

```
QueryRunner runner = new QueryRunner();
public T get(Connection conn,String sql,Object ...args){
    //    PreparedStatement ps = null;
    //    ResultSet rs = null;
    //    try{
    //        //1.预编译 sql 语句，获取 PreparedStatement 对象
    //        ps = conn.prepareStatement(sql);
    //        //2.填充占位符
    //        for(int i = 0;i < args.length;i++){
    //            ps.setObject(i + 1, args[i]);
    //        }
    //        //3.返回一个结果集
    //        rs = ps.executeQuery();
    //        ResultSetMetaData rsmd = rs.getMetaData();
    //        int columnCount = rsmd.getColumnCount();
    //        if(rs.next()){
    //            T t = clazz.newInstance();
    //
    //            //给 t 对象的相应属性赋值
    //            for(int i = 0;i < columnCount;i++){
    //                Object columnVal = rs.getObject(i + 1);
    //                String columnLabel = rsmd.getColumnLabel(i + 1);
    //                PropertyUtils.setProperty(t, columnLabel, columnVal);
    //            }
    //        }
    //    }
    //}
```

35

```
//          return t;
//      }
//
//      }catch(Exception e){
//          e.printStackTrace();
//      }finally{
//          JDBCUtils.close(rs, ps, null);
//      }
//      //System.out.println(clazz);
//      return null;
    ResultSetHandler<T> handler = new BeanHandler<>(clazz);
    try{
        return runner.query(conn, sql, handler, args);
    }catch(Exception e){
        e.printStackTrace();
    }
    return null;
}
```

//通用的增删改的操作

```
public void update(Connection conn,String sql,Object ... args){
//    PreparedStatement ps = null;
//    try{
//        ps = conn.prepareStatement(sql);
//
//        for(int i = 0;i < args.length;i++){
//            ps.setObject(i + 1, args[i]);
//        }
//
//        ps.executeUpdate();
//    }catch(Exception e){
//        e.printStackTrace();
//    }finally{
//        JDBCUtils.close(null, ps, null);
//    }
    try{
        runner.update(conn, sql, args);
    }catch(Exception e){
        e.printStackTrace();
    }
}
```

关于 JDBC 详细内容及课件、视频，请登录 www.atguigu.java 下载区下载