第3章 chapter 3

DBUtils 工具包

本章学习目标

- 了解 DBUtils 工具包的概念和常用 API。
- 掌握 DBUtils 工具包的增删改查操作。
- 掌握 DBUtils 工具包的事务处理。

在使用 JDBC 访问数据库时，必然要经过注册驱动、获取连接、访问数据库、处理结果集、释放资源等操作，而这些操作中步骤烦琐、代码冗余，不利于开发效率的提升。为此，Apache 组织提供了 DBUtils 工具包来解决这些问题。

3.1 初识 DBUtils

3.1.1 DBUtils 简述

DBUtils 是一个对 JDBC 进行封装的开源工具类库，由 Apache 组织提供，它能够简化 JDBC 应用程序的开发，降低开发者的工作量。

简单概括，DBUtils 工具包主要有三个作用，具体如下。

- 写操作，对于数据表的增、删、改，只需写 SQL 语句即可。
- 读操作，把结果集转换成 Java 常用集合类，方便对结果集进行处理。
- 优化性能，可以使用数据源、JNDI、数据库连接池等技术来减少代码冗余。

3.1.2 DBUtils 核心成员

由于 DBUtils 是对 JDBC 的封装，所以它的类包是围绕实现 JDBC 的功能来设计的。JDBC 需要多行代码才能实现的功能，DBUtils 只需调用一个工具类即可实现。DBUtils 提供了一系列的 API，具体如图 3.1 所示。

从图 3.1 可以看出，DBUtils 工具包主要有三个核心 API。通过它们，DBUtils 工具包基本覆盖了 JDBC 的所有操作。

1．DBUtils 类

该类主要为装载 JDBC 驱动、关闭资源等常规操作提供方法，它的方法一般是静态

的，直接以类名调用。DBUtils 类的常用方法如表 3.1 所示。

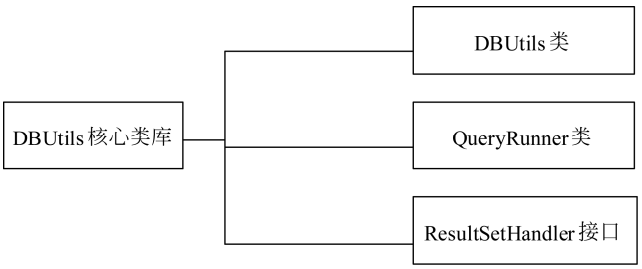

图 3.1　DBUtils 核心类库

表 3.1　DBUtils 类的常用方法

方 法 名 称	功 能 描 述
void close(Connection conn)	当连接不为 NULL 时，关闭连接
void close(Statement stat)	当声明不为 NULL 时，关闭声明
void close(ResultSet rs)	当结果集不为 NULL 时，关闭结果集
void closeQuietly(Connection conn)	当连接不为 NULL 时，关闭连接，并隐藏一些在程序中抛出的 SQL 异常
void closeQuietly(Statement stat)	当声明不为 NULL 时，关闭声明，并隐藏一些在程序中抛出的 SQL 异常
void closeQuietly(ResultSet rs)	当结果集不为 NULL 时，关闭结果集，并隐藏一些在程序中抛出的 SQL 异常
void commitAndCloseQuietly(Connection conn)	提交连接后关闭连接，并隐藏一些在程序中抛出的 SQL 异常
Boolean loadDriver(String driveClassName)	装载并注册 JDBC 驱动程序，如果成功就返回 TRUE

2．QueryRunner 类

该类用于执行 SQL 语句，和 JDBC 中 PreparedStatement 类的功能类似。它封装了执行 SQL 语句的代码，在获取结果集时和接口 ResultSetHandler 配合使用。QueryRunner 类的常用方法如表 3.2 所示。

表 3.2　QueryRunner 类的常用方法

方 法 名 称	功 能 描 述
Object query(Connection conn, String sql, ResultSetHandler rsh, Object[] params)	执行查询操作，需传入 Connection 对象
Object query(String sql, ResultSetHandler rsh，Object[] params)	执行查询操作
Object query(Connection conn, String sql, ResultSetHandler rsh)	用来执行一个不需要置换参数的更新操作
int update(Connection conn, String sql, Object[] params)	用来执行一个更新（插入、更新或删除）操作
int update(Connection conn, String sql)	用来执行一个不需要置换参数的更新操作
int[] batch(Connection conn, String sql, Object[][] params)	批量添加、更改、删除
int[] batch(String sql, Object[][] params)	批量添加、更改、删除

3．ResultSetHandler 接口

该接口主要用于处理查询之后获取的结果。为了应对各种各样的查询场景，DBUtils 提供了十余种该接口的实现类，每个实现类都有自己的独特之处，开发者可根据实际情况调用。ResultSetHandler 接口的实现类如表 3.3 所示。

表 3.3 ResultSetHandler 接口的实现类

类 名 称	功 能 描 述
ArrayHandler	将查询结果的第一行数据保存到 Object 数组中
ArrayListHandler	将查询结果的每一行先封装到 Object 数组中，然后将数据存入到 List 集合中
BeanHandler	将查询结果的第一行数据封装到类对象中
BeanListHandler	将查询结果的每一行封装到 JavaBean 对象中，然后再存入到 List 集合中
ColumnListHandler	将查询结果指定列的数据封装到 List 集合中
MapHandler	将查询结果的第一行数据封装到 map 集合中（key 是列名，value 是列值）
MapListHandler	将查询结果的每一行封装到 map 集合中（key 是列名，value 是列值），再将 map 集合存入 List 集合
BeanMapHandler	将查询结果的每一行数据封装到 User 对象中，再存入到 map 集合中（key 是列名，value 是列值）
KeyedHandler	将查询结果的每一行数据封装到 map1 集合中（key 是列名，value 是列值），然后将 map1 集合（有多个）存入到 map2 集合中（只有一个）
ScalarHandler	封装类似 count、avg、max、min、sum 等函数的执行结果

3.2 DBUtils 实现 DML 操作

3.2.1 创建 QueryRunner 对象

QueryRunner 类是 SQL 语句的执行者，它有两种构造方法。使用不同的构造方法，会对其成员方法的调用产生不同的影响。QueryRunner 类的构造方法如下。

- new QueryRunner(DataSource ds)。
- new QueryRunner()。

第一种是有参构造，需要传入数据源对象作为参数。它的事务是自动控制的，一个 SQL 命令即一个事务。使用此构造方法构建对象，当调用其方法（如 query、update）时，无须考虑 Connection 对象。

第二种是无参构造，可以进行事务的手动管理。使用此构造方法构建对象，当调用其方法（如 query、update）时，需要在参数中传入 Connection 对象。

在不考虑事务管理时，通常采用第一种方法。

3.2.2 DBUtils 实现 DML 操作

DML 操作主要包括添加、删除、修改等，由于不涉及结果集处理，步骤相对简单。

下面将通过案例分别讲解 DBUtils 对数据的添加、删除、修改等操作。

1．搭建开发环境

创建数据库 chapter03，在数据库中创建数据表 students，具体 SQL 语句如下。

```
DROP DATABASE IF EXISTS chapter03;
CREATE DATABASE chapter03;
USE chapter03;
CREATE TABLE students(
 s_id INT PRIMARY KEY AUTO_INCREMENT, #ID
 s_name VARCHAR(20), #姓名
 s_age INT#年龄
);
```

向表 students 中插入四条数据，具体语句如下。

```
INSERT INTO students(s_name, s_age) VALUES ('lilei',14);
INSERT INTO students(s_name, s_age) VALUES ('hanmeimei',13);
INSERT INTO students(s_name, s_age) VALUES ('tom',13);
INSERT INTO students(s_name, s_age) VALUES ('lucy',12);
```

向 MySQL 数据库发送查询语句，测试数据是否已经添加到数据库，运行结果如下。

```
mysql> SELECT * FROM STUDENTS;
+------+-----------+-------+
| s_id | s_name    | s_age |
+------+-----------+-------+
|    1 | lilei     |    14 |
|    2 | hanmeimei |    13 |
|    3 | tom       |    13 |
|    4 | lucy      |    12 |
+------+-----------+-------+
4 rows in set (0.00 sec)
```

打开 Eclipse，新建 Java 工程 chapter03，在工程 chapter03 下新建目录 lib，分别将 MySQL 数据库的驱动 jar 包，C3P0 数据库连接池的 jar 包，DBUtils 工具包的 jar 包复制到 lib 目录下，右击 lib 目录下的上述 jar 包，在弹出的菜单中选择 Build Path→Add to Build Path 命令，完成 jar 包的导入。将工程 chapter02 中的 c3p0-config.xml 文件复制到工程 chapter03 中的 src 目录下，将 c3p0-config.xml 文件中的数据库名改为 chapter03。

2．编写工具类

新建一个工具类 C3P0Utils，该类用于向 DBUtils 提供数据库连接池，具体代码如例 3.1 所示。

【例 3.1】 C3P0Utils.java

```
1   package com.qfedu.utils;
2   import java.sql.Connection;
3   import java.sql.SQLException;
4   import javax.sql.DataSource;
5   import com.mchange.v2.c3p0.ComboPooledDataSource;
6   public class C3P0Utils {
7       //通过读取 c3p0-config 文件获取连接池对象,使用 name 值为 qfedu 的配置
8       private static ComboPooledDataSource dataSource =
9           new ComboPooledDataSource("qfedu");
10      //提供一个 dataSource 数据源
11      public static DataSource getDataSource(){
12          return dataSource;
13      }
14  }
```

3．DBUtils 对数据的添加操作

新建一个测试类 TestDBUtils_Insert，该类用于测试 DBUtils 对数据的添加操作，具体代码如例 3.2 所示。

【例 3.2】 TestDBUtils_Insert.java

```
1   package com.qfedu.test;
2   import java.sql.SQLException;
3   import org.apache.commons.dbutils.QueryRunner;
4   import com.qfedu.utils.C3P0Utils;
5   public class TestDBUtils_Insert {
6       public static void main(String[] args) throws SQLException {
7           //通过有参构造方法生成一个 QueryRunner 对象
8           QueryRunner queryRunner = new
9               QueryRunner(C3P0Utils.getDataSource());
10          //创建一个 SQL 语句,向数据库插入数据
11          String sql = "insert into students(s_name,s_age)
12              values('david',15)";
13          //执行 SQL 语句
14          int count = queryRunner.update(sql);
15              if (count >0) {
16               System.out.println("数据添加成功");
17               } else {
18                   System.out.println("数据添加失败");
19               }
20      }
21  }
```

运行结果如图 3.2 所示，控制台显示数据添加成功。

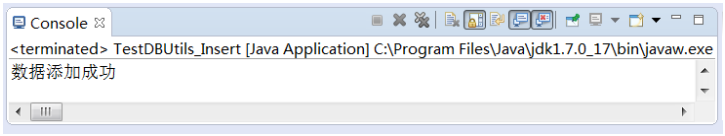

图 3.2　数据添加成功

向 MySQL 数据库发送查询语句，数据已成功添加，运行结果如下。

```
mysql> SELECT * FROM STUDENTS;
+------+-----------+-------+
| s_id | s_name    | s_age |
+------+-----------+-------+
|    1 | lilei     |    14 |
|    2 | hanmeimei |    13 |
|    3 | tom       |    13 |
|    4 | lucy      |    12 |
|    5 | david     |    15 |
+------+-----------+-------+
5 rows in set (0.00 sec)
```

4．DBUtils 对数据的修改操作

编写一个测试类 TestDBUtils_Update，该类用于测试 DBUtils 对数据的修改操作，具体代码如例 3.3 所示。

【例 3.3】 TestDBUtils_ Update.java

```
1   package com.qfedu.test;
2   import java.sql.SQLException;
3   import org.apache.commons.dbutils.QueryRunner;
4   import com.qfedu.utils.C3P0Utils;
5   public class TestDBUtils_Update {
6       public static void main(String[] args) throws SQLException {
7           //通过有参构造方法生成一个 QueryRunner 对象
8           QueryRunner queryRunner = new
9               QueryRunner(C3P0Utils.getDataSource());
10          //创建一个 SQL 语句,向数据库插入数据
11          String sql = "update students set s_age= 13 where s_name='david'";
12          //执行 SQL 语句
13          int count = queryRunner.update(sql);
14              if (count >0) {
15              System.out.println("数据修改成功");
16              } else {
17                  System.out.println("数据修改失败");
18              }
```

```
19        }
20    }
```

运行结果如图 3.3 所示，控制台显示数据修改成功。

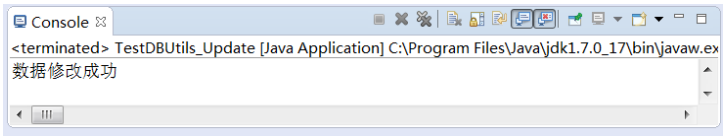

图 3.3　数据修改成功

向 MySQL 数据库发送查询语句，数据已成功修改，运行结果如下。

```
mysql> SELECT * FROM STUDENTS;
+------+-----------+-------+
| s_id | s_name    | s_age |
+------+-----------+-------+
|    1 | lilei     |    14 |
|    2 | hanmeimei |    13 |
|    3 | tom       |    13 |
|    4 | lucy      |    12 |
|    5 | david     |    13 |
+------+-----------+-------+
5 rows in set (0.00 sec)
```

5．DBUtils 对数据的删除操作

编写一个测试类 TestDBUtils_Delete，该类用于测试 DBUtils 对数据的删除操作，具体代码如例 3.4 所示。

【例 3.4】 TestDBUtils_ Delete.java

```
1   package com.qfedu.test;
2   import java.sql.SQLException;
3   import org.apache.commons.dbutils.QueryRunner;
4   import com.qfedu.utils.C3P0Utils;
5   public class TestDBUtils_Delete {
6       public static void main(String[] args) throws SQLException {
7           //通过有参构造方法生成一个 QueryRunner 对象
8           QueryRunner queryRunner = new
9               QueryRunner(C3P0Utils.getDataSource());
10          //创建一个 SQL 语句,向数据库插入数据
11          String sql = "delete from  students  where s_name='david'";
12          //执行 SQL 语句
13          int count = queryRunner.update(sql);
14              if (count >0) {
```

```
15                  System.out.println("数据删除成功");
16              } else {
17                  System.out.println("数据删除失败");
18              }
19      }
20  }
```

运行结果如图 3.4 所示，控制台显示数据删除成功。

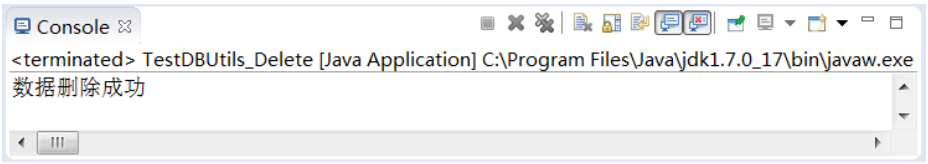

图 3.4　数据删除成功

向 MySQL 数据库发送查询语句，数据已成功删除，运行结果如下。

```
mysql> SELECT * FROM STUDENTS;
+------+-----------+-------+
| s_id | s_name    | s_age |
+------+-----------+-------+
|    1 | lilei     |    14 |
|    2 | hanmeimei |    13 |
|    3 | tom       |    13 |
|    4 | lucy      |    12 |
+------+-----------+-------+
4 rows in set (0.00 sec)
```

3.3　DBUtils 实现 DQL 操作

3.3.1　JavaBean

JavaBean 是 Java 语言中一个可重复利用的组件。简单而言，它本质上就是一个类，一个要遵循 JavaBean 编码规范的特殊类。编写一个 JavaBean 要遵循的规范，具体如图 3.5 所示。

使用 JavaBean 一定要遵循它的编码规范，避免程序出错。JavaBean 常用于封装数据，在 Java 程序与数据库的交互中，尤其是在处理从数据库获得的结果集时，通常会采用 JavaBean 封装数据。

为方便大家对 JavaBean 的理解，接下来自定义一个 JavaBean。

在工程 chapter03 的 src 目录下新建一个包 com.qfedu.bean，新建类 Students，具体代码如例 3.5 所示。

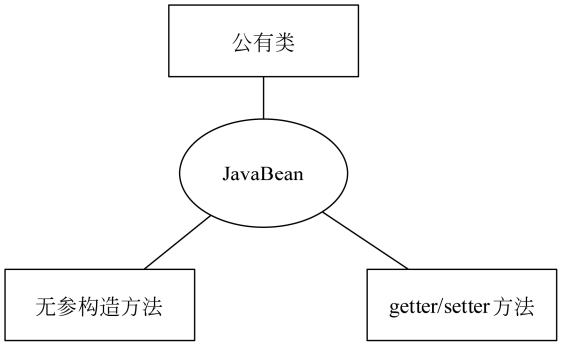

图 3.5 JavaBean 的编码规范

【例 3.5】 Students.java

```
1   package com.qfedu.bean;
2   //公有类
3   public class Students {
4       //提供私有属性
5       private String s_name;
6       private Integer s_age;
7       //提供无参构造方法
8       public Students() {
9           super();
10      }
11      //提供 getter/setter 方法
12      public String getS_name() {
13          return s_name;
14      }
15      public void setS_name(String s_name) {
16          this.s_name = s_name;
17      }
18      public Integer getS_age() {
19          return s_age;
20      }
21      public void setS_age(Integer s_age) {
22          this.s_age = s_age;
23      }
24  }
```

该 Students 类是一个简单的 JavaBean，它是公有类，提供了无参构造方法，及可供外界访问的 getter/setter 方法。

3.3.2 ArrayHandler 与 ArrayListHandler

调用 QueryRunner 类的 query()方法可以完成数据库的查询。DBUtils 提供了十余种处理方式，来应对不同场景下的结果集处理。

第一种处理方式是将结果集封装进数组，可以由 ArrayHandler 或 ArrayListHandler 实现。

ArrayHandler 与 ArrayListHandler 的对比情况如表 3.4 所示。前者是将结果集的第一行存储到数组中，而后者是将结果集的每一行封装到一个数组中，再把所有数组存储到 List 集合中。

表 3.4 ArrayHandler 与 ArrayListHandler 对比

类 名 称	相 同 点	不 同 点
ArrayHandler	都要首先将结果集封装进数组	封装单条数据，把结果集的第一条数据的字段值放入一个数组中
ArrayListHandler		封装多条数据，把每条数据的字段值各放入一个数组，再把所有数组都放入 List 集合中

下面通过实例来演示 ArrayHandler 的用法，具体代码如例 3.6 所示。

【例 3.6】 TestDBUtils_ArrayHandler.java

```
1  package com.qfedu.test;
2  import java.sql.SQLException;
3  import org.apache.commons.dbutils.QueryRunner;
4  import org.apache.commons.dbutils.handlers.ArrayHandler;
5  import com.qfedu.utils.C3P0Utils;
6  public class TestDBUtils_ArrayHandler {
7      public static void main(String[] args) throws SQLException {
8          //通过有参构造方法生成一个 QueryRunner 对象
9          QueryRunner queryRunner = new
10             QueryRunner(C3P0Utils.getDataSource());
11         //创建一个 SQL 语句,向数据库查询数据
12         String sql = "select * from students  where s_id= ? ";
13         //执行 SQL 语句,调用 QueryRunner 类的 query()方法
14         //第一个参数为 SQL 语句,第二个参数为结果集对象,第三个参数为 SQL 语句中的参数
15         Object[] arr = queryRunner.query(sql, new ArrayHandler(), new
16             Object[]{1});
17         //遍历结果集,并打印到控制台
18         for (int i = 0; i < arr.length; i++) {
19             System.out.print(arr[i]+",");
20         }
21     }
22 }
```

执行 TestDBUtils_ArrayHandler 类，运行结果如图 3.6 所示。

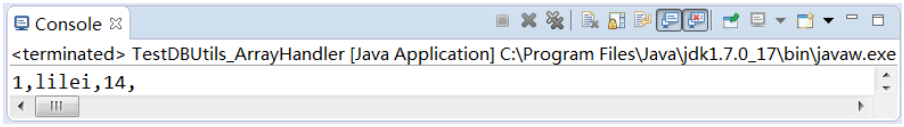

图 3.6 执行 TestDBUtils_ArrayHandler 类的运行结果

从以上示例代码可以看出，和 QueryRunner 类的 query()方法配合，ArrayHandler 类可将查询到的结果集封装到数组之中，并可将数组内的信息打印到控制台。

下面通过一个实例来测试 ArrayListHandler 的用法，具体代码如例 3.7 所示。

【例 3.7】 TestDBUtils_ArrayListHandler.java

```
1  package com.qfedu.test;
2  import java.sql.SQLException;
3  import java.util.List;
4  import org.apache.commons.dbutils.QueryRunner;
5  import org.apache.commons.dbutils.handlers.ArrayListHandler;
6  import com.qfedu.utils.C3P0Utils;
7  public class TestDBUtils_ArrayListHandler {
8      public static void main(String[] args) throws SQLException {
9          //通过有参构造方法生成一个 QueryRunner 对象
10         QueryRunner queryRunner = new
11             QueryRunner(C3P0Utils.getDataSource());
12         //创建一个 SQL 语句,向数据库查询数据
13         String sql = "select * from students ";
14         //执行 SQL 语句,调用 QueryRunner 类的 query()方法
15         //第一个参数为 SQL 语句,第二个参数为结果集对象
16          List<Object[]> list = queryRunner.query
17              (sql, new ArrayListHandler());
18         //遍历结果集,并打印到控制台
19          for (Object[] arr : list) {
20              for (int i = 0; i < arr.length; i++) {
21                     System.out.print(arr[i]+",");
22              }
23              System.out.println();
24         }
25     }
26 }
```

执行 TestDBUtils_ArrayListHandler 类，运行结果如图 3.7 所示。

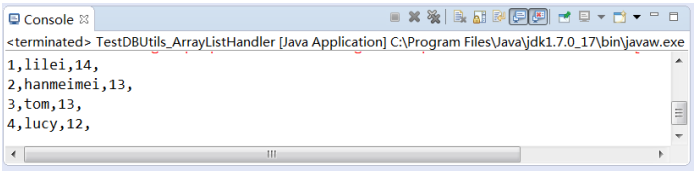

图 3.7　执行 TestDBUtils_ArrayListHandler 类的运行结果

从以上示例代码可以看出，和 QueryRunner 类的 query()方法配合，ArrayListHandler 类将查询到的结果封装到数组之中，并将所有数组封装到一个 List 对象中，控制台打印出了查询到的全部信息。

3.3.3　BeanHandler 与 BeanListHandler

第二种处理方式是将结果集封装进 JavaBean，可以由 BeanHandler 或 BeanListHandler 实现。在具体封装时，表中数据的字段和 JavaBean 的属性是相互对应的，一条数据表记录被封装进一个对应的 JavaBean 对象中。

表 3.5　BeanHandler 与 BeanListHandler 对比

类　名　称	相　同　点	不　同　点
BeanHandler	都要首先将结果集封装进 JavaBean	封装单条数据，把结果集的第一条数据的字段值放入一个 JavaBean 中
BeanListHandler		封装多条数据，把每条数据的字段值各放入一个 JavaBean 中，再把所有 JavaBean 都放入 List 集合中

BeanHandler 与 BeanListHandler 的对比情况如表 3.5 所示。前者是将结果集的第一行存储到 JavaBean 中，而后者是将结果集的每一行各封装到一个 JavaBean 中，再把所有 JavaBean 都存储到 List 集合中。

下面通过实例来演示 BeanHandler 的用法，具体代码如例 3.8 所示。

【例 3.8】 TestDBUtils_BeanHandler.java

```
1   package com.qfedu.test;
2   import java.sql.SQLException;
3   import org.apache.commons.dbutils.QueryRunner;
4   import org.apache.commons.dbutils.handlers.ArrayHandler;
5   import org.apache.commons.dbutils.handlers.BeanHandler;
6   import com.qfedu.bean.Students;
7   import com.qfedu.utils.C3P0Utils;
8   public class TestDBUtils_BeanHandler {
9       public static void main(String[] args) throws SQLException {
10          //通过有参构造方法生成一个 QueryRunner 对象
11          QueryRunner queryRunner = new
12              QueryRunner(C3P0Utils.getDataSource());
```

```
        //创建一个 SQL 语句,向数据库查询数据
        String sql = "select * from students  where s_id= ? ";
        //执行 SQL 语句,调用 QueryRunner 类的 query()方法
        //第一个参数为 SQL 语句,第二个参数为结果集对象,第三个参数为 SQL 语句中的参数
        Students students = queryRunner.query(sql, new
             BeanHandler(Students.class), new Object[]{1});
        //遍历结果集,并打印到控制台
        System.out.println("name 值为"+students.getS_name()+",age 值为
            "+students.getS_age());
    }
}
```

执行 TestDBUtils_BeanHandler 类，运行结果如图 3.8 所示。

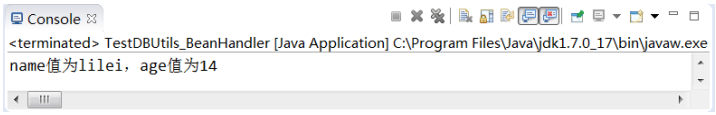

图 3.8　执行 TestDBUtils_BeanHandler 类的运行结果

从以上示例代码可以看出，和 QueryRunner 类的 query()方法配合，BeanHandler 类将查询到的结果集封装到 JavaBean 之中，并可将 JavaBean 内的信息打印到控制台。

下面通过一个实例来测试 BeanListHandler 的用法，具体代码如例 3.9 所示。

【例 3.9】 TestDBUtils_BeanListHandler.java

```
package com.qfedu.test;
import java.sql.SQLException;
import java.util.ArrayList;
import java.util.List;
import org.apache.commons.dbutils.QueryRunner;
import org.apache.commons.dbutils.handlers.BeanListHandler;
import com.qfedu.bean.Students;
import com.qfedu.utils.C3P0Utils;
public class TestDBUtils_BeanListHandler {
    public static void main(String[] args) throws SQLException {
        //通过有参构造方法生成一个 QueryRunner 对象
        QueryRunner queryRunner = new
            QueryRunner(C3P0Utils.getDataSource());
        //创建一个 SQL 语句,向数据库查询数据
        String sql = "select * from students ";
        //执行 SQL 语句,调用 QueryRunner 类的 query()方法
        //第一个参数为 SQL 语句,第二个参数为结果集对象
        ArrayList<Students>  list =  queryRunner.query(sql, new
            BeanListHandler(Students.class));
        //遍历结果集,并打印到控制台
        for (Students stu : list) {
            System.out.println("name 值为"+stu.getS_name()+",age 值为
                "+stu.getS_age());
```

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

执行 TestDBUtils_BeanListHandler 类，运行结果如图 3.9 所示。

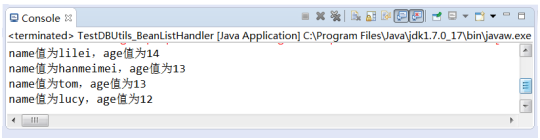

图 3.9　执行 TestDBUtils_BeanListHandler 类的运行结果

从以上示例代码可以看出，和 QueryRunner 类的 query()方法配合，BeanListHandler 类将查询到的结果封装到 JavaBean 之中，并将所有 JavaBean 封装到一个 List 对象中，控制台打印出了查询到的全部信息。

3.3.4　MapHandler、MapListHandler 与 KeyedHandler

第三种处理方式是将结果集封装进 Map 集合中，可以由 MapHandler、MapListHandler 或 KeyedHandler 实现。在具体封装时，数据表中数据的字段名和字段值以 Map 映射的方式存储。

MapHandler、MapListHandler、KeyedHandler 的对比情况如表 3.6 所示。开发者应根据具体场景选择使用。

表 3.6　MapHandler、MapListHandler、KeyedHandler 对比

类　名　称	相　同　点	不　同　点
MapHandler	都要首先将结果集存储为 Map 映射	封装单条数据，把结果集的第一条数据的字段名和字段值存储为 Map 映射
MapListHandler		封装多条数据，把每条数据的字段名和字段值各存储为一个 Map 映射，再把所有 Map 映射都放入 List 集合中
KeyedHandler		封装多条数据，把每条数据的字段名和字段值各存储为一个 Map 映射，再把所有 Map 映射根据指定 key 都放入一个新的 Map 映射中

下面通过实例来演示 MapHandler 的用法，具体代码如例 3.10 所示。

【例 3.10】 TestDBUtils_MapHandler.java

```
1  package com.qfedu.test;
2  import java.sql.SQLException;
3  import java.util.Map;
4  import org.apache.commons.dbutils.QueryRunner;
5  import org.apache.commons.dbutils.handlers.MapHandler;
6  import com.qfedu.utils.C3P0Utils;
```

```
7   public class TestDBUtils_MapHandler {
8       public static void main(String[] args) throws SQLException {
9           //通过有参构造方法生成一个 QueryRunner 对象
10          QueryRunner queryRunner = new
11              QueryRunner(C3P0Utils.getDataSource());
12          //创建一个 SQL 语句,向数据库查询数据
13          String sql = "select * from students  where s_id= ? ";
14          //执行 SQL 语句,调用 QueryRunner 类的 query()方法
15          //第一个参数为 SQL 语句,第二个参数为结果集对象,第三个参数为 SQL 语句中的参数
16          Map<String, Object> map = queryRunner.query(sql, new MapHandler(),
17              new Object[]{1});
18          //打印到控制台
19          System.out.println(map);
20      }
21
```

执行 TestDBUtils_MapHandler 类，运行结果如图 3.10 所示。

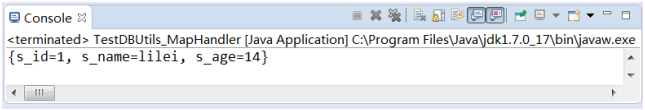

图 3.10　执行 TestDBUtils_MapHandler 类的运行结果

从以上示例代码可以看出，和 QueryRunner 类的 query()方法配合，MapHandler 类将查询到的结果集封装到 Map 之中，并可将 Map 内的信息打印到控制台。

下面通过一个实例来测试 MapListHandler 的用法，具体代码如例 3.11 所示。

【例 3.11】 TestDBUtils_MapListHandler.java

```
1   package com.qfedu.test;
2   import java.sql.SQLException;
3   import java.util.List;
4   import java.util.Map;
5   import org.apache.commons.dbutils.QueryRunner;
6   import org.apache.commons.dbutils.handlers.MapListHandler;
7   import com.qfedu.utils.C3P0Utils;
8   public class TestDBUtils_MapListHandler {
9       public static void main(String[] args) throws SQLException {
10          //通过有参构造方法生成一个 QueryRunner 对象
11          QueryRunner queryRunner = new
12              QueryRunner(C3P0Utils.getDataSource());
13          //创建一个 SQL 语句,向数据库查询数据
14          String sql = "select * from students ";
15          //执行 SQL 语句,调用 QueryRunner 类的 query()方法
16          //第一个参数为 SQL 语句,第二个参数为结果集对象
```

```
        List<Map<String,Object>> list = queryRunner.query(sql, new
            MapListHandler());
        //遍历结果集,并打印到控制台
        for (Map<String,Object> map : list) {
            System.out.print(map);
            System.out.println();
        }
    }
}
```

执行 TestDBUtils_MapListHandler 类，运行结果如图 3.11 所示。

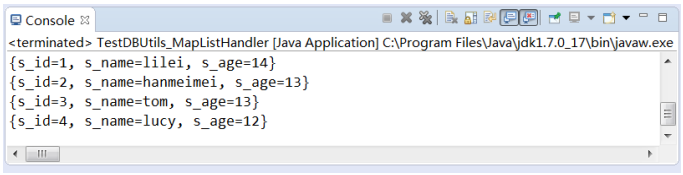

图 3.11　执行 TestDBUtils_MapListHandler 类的运行结果

从以上示例代码可以看出，与 QueryRunner 类的 query()方法配合，MapListHandler 类将查询到的结果封装到 Map 之中，并将所有 Map 都封装到一个 List 对象中，控制台打印出了查询到的全部信息。

下面通过一个实例来测试 KeyedHandler 的用法，具体代码如例 3.12 所示。

【例 3.12】 TestDBUtils_KeyedHandler.java

```
package com.qfedu.test;
import java.sql.SQLException;
import java.util.Map;
import org.apache.commons.dbutils.QueryRunner;
import org.apache.commons.dbutils.handlers.KeyedHandler;
import com.qfedu.utils.C3P0Utils;
public class TestDBUtils_KeyedHandler {
    public static void main(String[] args) throws SQLException {
        //通过有参构造方法生成一个 QueryRunner 对象
        QueryRunner queryRunner = new
            QueryRunner(C3P0Utils.getDataSource());
        //创建一个 SQL 语句,向数据库查询数据
        String sql = "select * from students ";
        //执行 SQL 语句,调用 QueryRunner 类的 query()方法
        //第一个参数为 SQL 语句,第二个参数为结果集对象,需传入封装大 Map 时所需的键值
        Map<Object,Map<String,Object>>map=queryRunner.query(sql,new
            KeyedHandler<Object>("s_id"));
        //获取第一条数据
        Map<String, Object> m = map.get(new Integer(1));
```

```
20          //根据字段名获取字段值
21          String sname = (String) m.get("s_name");
22          Integer sage = (Integer) m.get("s_age");
23          System.out.println("name 值为"+sname+",age 值为"+sage);
24      }
25  }
```

执行 TestDBUtils_ KeyedHandler 类，运行结果如图 3.12 所示。

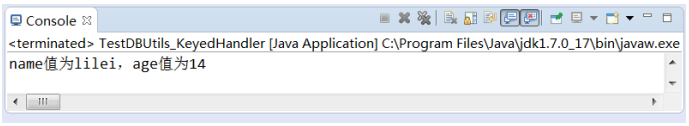

图 3.12　执行 TestDBUtils_ KeyedHandler 类的运行结果

从以上示例代码可以看出，与 QueryRunner 类的 query()方法配合，KeyedHandler 类将查询到的结果封装到 Map 之中，并将所有 Map 都封装到一个大 Map 中，控制台打印出了查询到的信息。

3.3.5　ColumnListHandler 与 ScalarHandler

第四种处理方式是对指定的列数据进行封装，可以由 ColumnListHandler 或 ScalarHandler 来实现。在具体封装时，查询指定列获得的数据被封装到容器中。

ColumnListHandler 与 ScalarHandler 的对比情况如表 3.7 所示。前者可以对指定列的所有数据进行封装，后者主要针对单行单列的数据封装。

表 3.7　ColumnListHandler 与 ScalarHandler 对比

类　名　称	相　同　点	不　同　点
ColumnListHandler	都是对指定列的查询结果集进行封装	封装指定列的所有数据，将它们放入一个 List 集合中
ScalarHandler		封装单条单列数据，也可以封装类似 count、avg、max、min、sum 等聚合函数的执行结果

下面通过实例来演示 ColumnListHandler 的用法，具体代码如例 3.13 所示。

【例 3.13】 TestDBUtils_ColumnListHandler.java

```
1   package com.qfedu.test;
2   import java.sql.SQLException;
3   import java.util.List;
4   import org.apache.commons.dbutils.QueryRunner;
5   import org.apache.commons.dbutils.handlers.ColumnListHandler;
6   import com.qfedu.utils.C3P0Utils;
7   public class TestDBUtils_ColumnListHandler {
8       public static void main(String[] args) throws SQLException {
```

```
9           //通过有参构造方法生成一个 QueryRunner 对象
10          QueryRunner queryRunner = new
11              QueryRunner(C3P0Utils.getDataSource());
12          //创建一个 SQL 语句,向数据库查询数据
13          String sql = "select * from students ";
14          //执行 SQL 语句,调用 QueryRunner 类的 query()方法
15          //第一个参数为 SQL 语句,第二个参数为结果集对象,需要传入要查的字段名
16          List<Object> list = queryRunner.query(sql, new
17               ColumnListHandler("s_name"));
18          //将 List 集合中的信息打印到控制台
19          System.out.println(list);
20      }
21  }
```

执行 TestDBUtils_ ColumnListHandler 类，运行结果如图 3.13 所示。

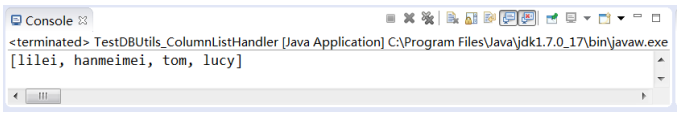

图 3.13　执行 TestDBUtils_ ColumnListHandler 类的运行结果

从以上示例代码可以看出，与 QueryRunner 类的 query()方法配合，ColumnListHandler 类可以将查询指定列的结果集封装到 List 集合之中。

下面通过一个实例来测试 ScalarHandler 的用法，具体代码如例 3.14 所示。

【例 3.14】 TestDBUtils_ScalarHandler.java

```
1   package com.qfedu.test;
2   import java.sql.SQLException;
3   import org.apache.commons.dbutils.QueryRunner;
4   import org.apache.commons.dbutils.handlers.ScalarHandler;
5   import com.qfedu.utils.C3P0Utils;
6   public class TestDBUtils_ScalarHandler {
7       public static void main(String[] args) throws SQLException {
8           //通过有参构造方法生成一个 QueryRunner 对象
9           QueryRunner queryRunner = new
10              QueryRunner(C3P0Utils.getDataSource());
11          //创建一个 SQL 语句,向数据库查询数据
12          String sql = "select s_name  from students  where s_id= ?";
13          //执行 SQL 语句,调用 QueryRunner 类的 query()方法,第一个参数为 SQL 语句
14          //第二个参数为结果集对象,需要传入要查的字段名,第三个参数为 SQL 语句中的参数
15           String s_name =(String) queryRunner.query(sql, new
16               ScalarHandler("s_name"),new Object[]{1});
17          //将查询信息打印到控制台
18          System.out.println(s_name);
```

```
19      }
20  }
```

执行 TestDBUtils_ScalarHandler 类，运行结果如图 3.14 所示。

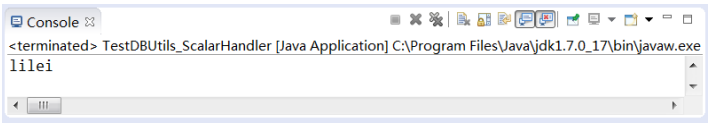

图 3.14　执行 TestDBUtils_ScalarHandler 类的运行结果

从以上示例代码可以看出，和 QueryRunner 类的 query()方法配合，ScalarHandler 类将查询到的结果封装，控制台打印出了查询到的信息。

3.4　DBUtils 的高级操作

3.4.1　DBUtils 批处理

QueryRunner 类提供的 batch()方法用于对 SQL 语句进行批量操作，但是只能执行相同的 SQL 语句，其中参数可以不同。

下面通过具体实例来讲解 QueryRunner 类的批处理功能。如果要向数据表 Students 中插入三条数据，具体代码如例 3.15 所示。

【例 3.15】 TestDBUtils_Batch.java

```
1   package com.qfedu.test;
2   import java.sql.SQLException;
3   import org.apache.commons.dbutils.QueryRunner;
4   import com.qfedu.utils.C3P0Utils;
5   public class TestDBUtils_Batch {
6       public static void main(String[] args) throws SQLException {
7           //通过有参构造方法生成一个 QueryRunner 对象
8           QueryRunner queryRunner = new
9               QueryRunner(C3P0Utils.getDataSource());
10          //创建一个 SQL 语句,向数据库插入数据
11          String sql = "insert into students(s_name,s_age) values(?,?) ";
12          //设置类型为二维数组的参数,里层的一维数组为每条 SQL 语句的参数
13          Object[][] param= new Object[3][];
14              for(int i=0; i<3; i++){
15                  param[i]= new Object[]{"name"+i,10+i};
16              }
17          //执行 SQL 语句,调用 QueryRunner 类的 query()方法
18          queryRunner.batch(sql, param);
19      }
20  }
```

运行 TestDBUtils_Batch 类，通过命令行窗口向数据库发送 select 语句，结果显示三条数据已被批量添加至数据库，运行结果如下。

```
mysql> SELECT * FROM STUDENTS;
+------+-----------+-------+
| s_id | s_name    | s_age |
+------+-----------+-------+
|    1 | lilei     |    14 |
|    2 | hanmeimei |    13 |
|    3 | tom       |    13 |
|    4 | lucy      |    12 |
|    6 | name0     |    10 |
|    7 | name1     |    11 |
|    8 | name2     |    12 |
+------+-----------+-------+
8 rows in set (0.00 sec)
```

3.4.2 DBUtils 事务管理

在前面的讲解中，创建 QueryRunner 对象时，调用了有参构造方法，此时 QueryRunner 对象能自动建立和释放数据库连接，没有考虑事务管理的问题。

在实际的开发过程中，如果涉及事务管理，开发者要调用无参构造方法创建 QueryRunner 对象，将 Connection 对象分离出来，进行手动管理。

在讲解事务的概念时，本书介绍过李磊到商店购物时扫码支付的场景，接下来通过 DBUtils 重新实现这个案例，步骤如下。

1．创建数据库和表

在数据库 chapter03 中创建名为 account 的表，向表中插入若干条数据，具体的 SQL 语句如下。

```
CREATE TABLE account(
id INT PRIMARY KEY AUTO_INCREMENT, #ID
aname VARCHAR(20), #姓名
money DOUBLE #余额
);
INSERT INTO account(aname,money) VALUES('lilei',3000);
INSERT INTO account(aname,money) VALUES('shop',20000);
```

上述 SQL 语句运行完毕后，在命令行窗口检查数据库环境是否搭建成功，发送 select 语句，运行结果如下。

```
mysql> SELECT * FROM account;
+----+-------+-------+
```

```
| id | aname | money |
+----+-------+-------+
|  1 | lilei |  3000 |
|  2 | shop  | 20000 |
+----+-------+-------+
2 rows in set (0.00 sec)
```

2．编写代码

改写本章 3.2 节中提到的 C3P0Utils 类，加入处理事务的方法。为了保证从连接池中拿到的连接是同一个，代码中使用了 ThreadLocal 类，将 Connection 对象与线程绑定，具体代码如例 3.16 所示。

【例 3.16】 C3P0Utils.java

```
1   package com.qfedu.utils;
2   import java.sql.Connection;
3   import java.sql.SQLException;
4   import javax.sql.DataSource;
5   import com.mchange.v2.c3p0.ComboPooledDataSource;
6   public class C3P0Utils {
7       //通过读取 c3p0-config 文件获取连接池对象,使用 name 值为 qfedu 的配置
8       private static ComboPooledDataSource dataSource = new
9           ComboPooledDataSource("qfedu");
10      //提供一个 dataSource 数据源
11      public static DataSource getDataSource(){
12          return dataSource;
13      }
14      //创建一个 ThreadLocal 对象
15      private static ThreadLocal<Connection> threadLocal = new
16          ThreadLocal<Connection>();
17      //提供当前线程中的 Connection
18      public static Connection getConnection() throws SQLException{
19          Connection conn = threadLocal.get();
20          if (null==conn) {
21              conn  = dataSource.getConnection();
22              threadLocal.set(conn);
23          }
24          return conn;
25      }
26      //开启事务
27      public static void startTransaction(){
28          //首先获取当前线程的连接
29          try {
30              Connection  conn = getConnection();
```

```
31              //关闭事务自动提交
32              conn.setAutoCommit(false);
33          } catch (SQLException e) {
34              e.printStackTrace();
35          }
36      }
37      //提交事务
38      public static void commit(){
39              //首先获取当前线程的连接
40              Connection conn = threadLocal.get();;
41              if(null!= conn){
42                  //提交事务
43                  try {
44                      conn.commit();
45                  } catch (SQLException e) {
46                      e.printStackTrace();
47                  }
48              }
49          }
50      //回滚事务
51      public static void rollback(){
52              //首先获取当前线程的连接
53              Connection conn = threadLocal.get();;
54              if(null!= conn){
55                  //回滚事务
56                  try {
57                      conn.rollback();
58                  } catch (SQLException e) {
59                      e.printStackTrace();
60                  }
61              }
62          }
63      //关闭连接
64      public static void close(){
65              Connection conn = threadLocal.get();
66              if(null!= conn){
67                  try {
68                      conn.close();
69                  } catch (SQLException e) {
70                      e.printStackTrace();
71                  }finally{
72                      //从当前线程移除连接,避免造成内存泄露
                    threadLocal.remove();
73                  }
```

```
            }
        }
}
```

调用工具类，测试 DBUtils 的事务管理，在创建 QueryRunner 对象时，调用无参构造方法，对 Connection 对象进行单独管理，具体代码如例 3.17 所示。

【例 3.17】 TestPayment.java

```
package com.qfedu.test;
import java.sql.Connection;
import java.sql.SQLException;
import org.apache.commons.dbutils.QueryRunner;
import com.qfedu.utils.C3P0Utils;
public class TestPayment {
    public static void main(String[] args) {
     Connection conn = null;
     try {
         //开启事务
         C3P0Utils.startTransaction();
         conn = C3P0Utils.getConnection();
         QueryRunner queryRunner = new QueryRunner();
         //lilei 的账户减去 100 元
         String sql_1 = "update account set money = money-? where aname=?";
         queryRunner.update(conn, sql_1, new Object[]{100,"lilei"});
         //shop 的账户增加 100 元
         String sql_2 = "update account set money = money+? where aname=?";
         queryRunner.update(conn, sql_2, new Object[]{100,"shop"});
         //提交事务
         conn.commit();
         System.out.println("支付完毕");
             } catch (Exception e) {
                 //如果有异常,回滚事务
                try {
                 conn.rollback();
                     System.out.println("支付失败");
                 } catch (SQLException e1) {
                     e1.printStackTrace();
                 }
             } finally{
             //释放资源
             C3P0Utils.close();
             }
    }
```

```
36  }
```

代码执行完毕，再次发送 select 语句，DBUtils 已将本次事务提交到数据库，运行结果如下。

```
mysql> SELECT * FROM account;
+----+-------+-------+
| id | aname | money |
+----+-------+-------+
|  1 | lilei |  2900 |
|  2 | shop  | 20100 |
+----+-------+-------+
2 rows in set (0.18 sec)
```

3.5 DBUtils 实现 Dao 封装

在实际的项目开发中，经常会把操作数据库的代码封装为 Dao 层，以降低各模块之间的耦合。在大家掌握了 DBUtils 的基本操作之后，本书将继续讲解使用 DBUtils 完成 Dao 封装的方法。

首先，在工程 chapter03 的 src 目录下新建 com.qfedu.dao 类包，然后在该包下新建一个 Dao 类 StudentsDao，具体代码如例 3.18 所示。

【例 3.18】 StudentsDao.java

```
1   package com.qfedu.dao;
2   import java.sql.SQLException;
3   import java.util.List;
4   import org.apache.commons.dbutils.QueryRunner;
5   import org.apache.commons.dbutils.handlers.BeanHandler;
6   import org.apache.commons.dbutils.handlers.BeanListHandler;
7   import com.qfedu.bean.Students;
8   import com.qfedu.utils.C3P0Utils;
9   public class StudentsDao {
10      //添加
11      public int insert(Students stu) throws SQLException{
12          QueryRunner queryRunner = new
13              QueryRunner(C3P0Utils.getDataSource());
14          String sql ="insert into students(s_name,s_age) values(?,?)";
15          Object[] params = new Object[]{stu.getS_name(),stu.getS_age()};
16          int count = queryRunner.update(sql,params);
17         return count;
18      }
19      //修改
```

```
20      public int update(Students stu) throws SQLException{
21          QueryRunner queryRunner = new
22              QueryRunner(C3P0Utils.getDataSource());
23          String sql ="update students set  s_age= ? where s_name=?";
24          Object[] params = new Object[]{stu.getS_age(),stu.getS_name()};
25          int count = queryRunner.update(sql,params);
26         return count;
27      }
28      //删除
29      public int delete(Students stu) throws SQLException{
30          QueryRunner queryRunner = new
31              QueryRunner(C3P0Utils.getDataSource());
32          String sql ="delete from students where s_name= ?";
33          Object[] params = new Object[]{stu.getS_name()};
34          int count = queryRunner.update(sql,params);
35         return count;
36      }
37      //根据 id 查询单个数据
38      public Students selectOne(Integer id) throws SQLException{
39          QueryRunner queryRunner = new
40              QueryRunner(C3P0Utils.getDataSource());
41          String sql ="select * from students where s_id= ?";
42          Object[] params = new Object[]{id};
43          Students newStu = queryRunner.query(sql, new
44              BeanHandler(Students.class),params);
45          return newStu;
46      }
47      //查询所有数据
48      public List<Students> selectAll() throws SQLException{
49          QueryRunner queryRunner = new
50              QueryRunner(C3P0Utils.getDataSource());
51          String sql ="select * from students";
52          List<Students> list = queryRunner.query(sql, new
53              BeanListHandler(Students.class));
54         return list;
55      }
56  }
```

利用 StudentsDao 类实现插入功能，具体代码如例 3.19 所示。

【例 3.19】 TestStuDao_insert.java

```
1   package com.qfedu.test;
2   import java.sql.SQLException;
3   import com.qfedu.bean.Students;
4   import com.qfedu.dao.StudentsDao;
```

```
5   public class TestStuDao_insert {
6       public static void main(String[] args) throws SQLException {
7           StudentsDao dao =new StudentsDao();
8           Students students = new Students();
9           students.setS_name("david");
10          students.setS_age(15);
11          int count = dao.insert(students);
12          if (count >0) {
13              System.out.println("数据添加成功");
14          } else {
15              System.out.println("数据添加失败");
16          }
17      }
18  }
```

执行 TestStuDao_insert 类，运行结果如图 3.15 所示。

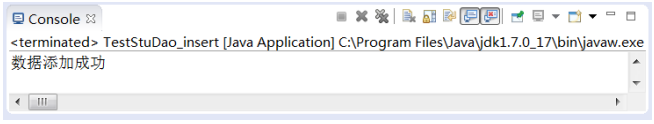

图 3.15　执行 TestStuDao_insert 类的运行结果

向数据库发送查询语句，一条记录已被成功添加，查询结果如下。

```
mysql> SELECT * FROM STUDENTS;
+------+-----------+-------+
| s_id | s_name    | s_age |
+------+-----------+-------+
|    1 | lilei     |    14 |
|    2 | hanmeimei |    13 |
|    3 | tom       |    13 |
|    4 | lucy      |    12 |
|    6 | name0     |    10 |
|    7 | name1     |    11 |
|    8 | name2     |    12 |
|    9 | david     |    15 |
+------+-----------+-------+
9 rows in set (0.00 sec)
```

利用 StudentsDao 类实现修改功能，具体代码如例 3.20 所示。

【例 3.20】 TestStuDao_update.java

```
1   package com.qfedu.test;
2   import java.sql.SQLException;
3   import com.qfedu.bean.Students;
```

```
4   import com.qfedu.dao.StudentsDao;
5   public class TestStuDao_update {
6       public static void main(String[] args) throws SQLException {
7           StudentsDao dao =new StudentsDao();
8           Students students = new Students();
9           students.setS_name("david");
10          students.setS_age(13);
11          int count = dao.update(students);
12          if (count >0) {
13          System.out.println("数据修改成功");
14          } else {
15              System.out.println("数据修改失败");
16          }
17      }
18  }
```

执行 TestStuDao_ update 类，运行结果如图 3.16 所示。

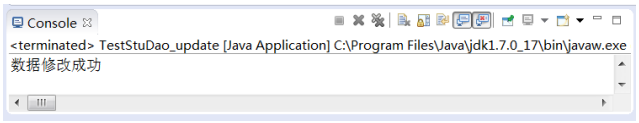

图 3.16　执行 TestStuDao_ update 类的运行结果

向数据库发送查询语句，指定记录已被成功修改，查询结果如下。

```
mysql> SELECT * FROM STUDENTS;
+------+-----------+-------+
| s_id | s_name    | s_age |
+------+-----------+-------+
|    1 | lilei     |    14 |
|    2 | hanmeimei |    13 |
|    3 | tom       |    13 |
|    4 | lucy      |    12 |
|    6 | name0     |    10 |
|    7 | name1     |    11 |
|    8 | name2     |    12 |
|    9 | david     |    13 |
+------+-----------+-------+
9 rows in set (0.00 sec)
```

利用 StudentsDao 类实现删除功能，具体代码如例 3.21 所示。

【例 3.21】 TestStuDao_delete.java

```
1   package com.qfedu.test;
2   import java.sql.SQLException;
```

```
3  import com.qfedu.bean.Students;
4  import com.qfedu.dao.StudentsDao;
5  public class TestStuDao_delete {
6      public static void main(String[] args) throws SQLException {
7          StudentsDao dao =new StudentsDao();
8          Students students = new Students();
9          students.setS_name("david");
10         int count = dao.delete(students);
11         if (count >0) {
12         System.out.println("数据删除成功");
13         } else {
14             System.out.println("数据删除失败");
15         }
16     }
17 }
```

执行 TestStuDao_ delete 类，运行结果如图 3.17 所示。

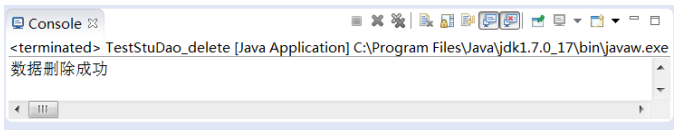

图 3.17　执行 TestStuDao_ delete 类的运行结果

向数据库发送查询语句，指定记录已被成功删除，查询结果如下。

```
mysql> SELECT * FROM STUDENTS;
+------+-----------+-------+
| s_id | s_name    | s_age |
+------+-----------+-------+
|    1 | lilei     |    14 |
|    2 | hanmeimei |    13 |
|    3 | tom       |    13 |
|    4 | lucy      |    12 |
|    6 | name0     |    10 |
|    7 | name1     |    11 |
|    8 | name2     |    12 |
+------+-----------+-------+
8 rows in set (0.00 sec)
```

利用 StudentsDao 类实现单个对象的查询，具体代码如例 3.22 所示。

【例 3.22】 TestStuDao_selectOne.java

```
1   package com.qfedu.test;
2   import java.sql.SQLException;
3   import com.qfedu.bean.Students;
```

```
4   import com.qfedu.dao.StudentsDao;
5   public class TestStuDao_selectOne {
6       public static void main(String[] args) throws SQLException {
7           StudentsDao dao =new StudentsDao();
8           Students stu = dao.selectOne(1);
9           System.out.println("name 值为"+stu.getS_name()+",age 值为
10              "+stu.getS_age());
11      }
12  }
```

执行 TestStuDao_ selectOne 类，运行结果如图 3.18 所示。

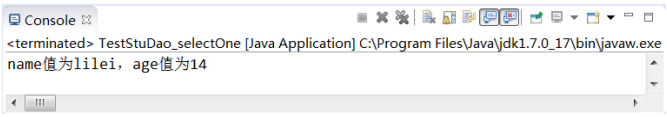

图 3.18　执行 TestStuDao_ selectOne 类的运行结果

利用 StudentsDao 类实现所有数据的查询，具体代码如例 3.23 所示。

【例 3.23】 TestStuDao_selectAll.java

```
1   package com.qfedu.test;
2   import java.sql.SQLException;
3   import java.util.List;
4   import com.qfedu.bean.Students;
5   import com.qfedu.dao.StudentsDao;
6   public class TestStuDao_selectAll {
7       public static void main(String[] args) throws SQLException {
8           StudentsDao dao =new StudentsDao();
9          List<Students> list = dao.selectAll();
10          for (Students stu : list) {
11              System.out.println("name 值为"+stu.getS_name()+",age 值为
12                  "+stu.getS_age());
13          }
14      }
15  }
```

执行 TestStuDao_ selectAll 类，运行结果如图 3.19 所示。

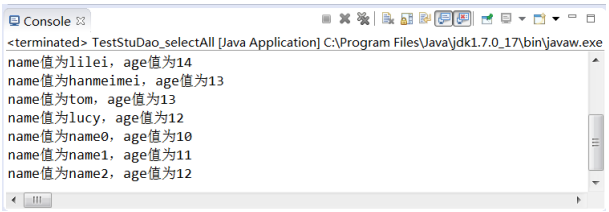

图 3.19　执行 TestStuDao_ selectAll 类的运行结果

3.6 本章小结

本章主要介绍 DBUtils 工具包的相关操作，包括增、删、查、改以及事务处理等。学习完本章的内容后，大家应该掌握利用 DBUtils 工具包操作数据库的方法。

3.7 习　　题

1．填空题

（1）在 DBUtils 工具包提供的 API 中，用于关闭资源的类是_____。
（2）当执行查询操作时，需要调用 QueryRunner 类的_____方法。
（3）当执行插入操作时，需要调用 QueryRunner 类的_____方法。
（4）当执行批量操作时，需要调用 QueryRunner 类的_____方法。
（5）将查询结果的第一行数据封装到数组的 API 是_____。

2．选择题

（1）在 DBUtils 工具包提供的 API 中，用于执行 SQL 语句的是（　　）。
A．PreparedStatement　　B．DBUtils
C．QueryRunner　　D．ResultSetHandler
（2）在 DBUtils 工具包提供的 API 中，用于封装结果集的是（　　）。
A．PreparedStatement　　B．DBUtils
C．QueryRunner　　D．ResultSetHandler
（3）在 DBUtils 工具包提供的 API 中，用于封装结果集中的一列数据的是（　　）。
A．ColumnListHandler　　B．MapHandler
C．ArrayHandler　　D．BeanHandler
（4）使用 DBUtils 工具包完成如下操作：select count(*) from 表，可以封装该查询结果的是（　　）。
A．BeanHandler　　B．ScalarHandler
C．ArrayHandler　　D．MapHandler
（5）关于 JavaBean 的设计规范，下列说法错误的是（　　）。
A．作为 JavaBean 的类，必须要提供无参的构造方法
B．作为 JavaBean 的类，要提供 get 和 set 方法访问其属性
C．作为 JavaBean 的类，其所有的属性最好定义为私有的
D．作为 JavaBean 的类，必须要提供有参的构造方法

3．思考题

简述 BeanHanlder 和 BeanListHanlder 的异同。

4．编程题

已有数据库 chapter03 中的 stundents 表，请完成以下操作。

（1）查询 stundents 表中 s_id 为 1 的数据，并将结果打印至控制台。

（2）查询 stundents 表中的所有数据，并将结果打印至控制台。

（3）查询 stundents 表中学生的个数。