第08天 JDBC连接池&DBUtils工具类

**【学习目标】理解、了解、应用、记忆**

1. **【理解】c3p0连接池**
2. 【理解】阐述连接池的作用和特点
3. 【应用】独立完编写代码使用c3p0实现获取连接操作
4. 【应用】定义c3p0配置文件实现获取连接功能
5. 【应用】独立工具类实现获取Connection连接的功能
6. **【应用】dbcp连接池**
7. 【应用】独立编写代码使用dbcp实现获取连接操作
8. 【应用】编写代码使用配置文件的方式实现dbcp获取连接

功能

1. 【应用】独立编写dbcp工具类获取连接池对象和连接对象
2. **【应用】DBUtils工具类**
3. 【理解】独立编写代码使用QueryRunner核心类完成增删改查操作
4. 【应用】独立编写代码实现使用BeanHandler方式封装数据
5. 【理解】独立编写代码实现使用BeanListHandler方式封装数据
6. 【应用】独立编写代码实现使用ScalarHandler方式封装数据
7. 【应用】独立编写代码实现使用MapHandler方式封装数据
8. 【应用】独立编写代码实现使用MapListHandler方式封装数据
9. 【应用】独立编写代码实现使用ArrayHandler方式封装数据
10. 【应用】独立编写代码实现使用ArrayListHandler方式封装数据
11. 【应用】独立编写代码实现使用KeyedHandler方式封装数据
12. 【应用】独立编写代码实现使用ColumnListHandler方式封装数据

# c3p0连接池

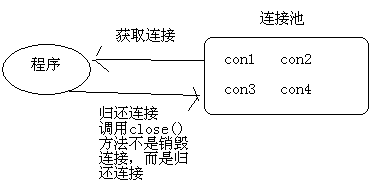
## 连接池概述

实际开发中“获得连接”或“释放资源”是非常消耗系统资源的两个过程，为了解决此类性能问题，通常情况我们采用连接池技术，来共享连接Connection。这样我们就不需要每次都创建连接、释放连接了，这些操作都交给了连接池

hikariCP 光 druid 德鲁伊(阿里爸爸 … )

* 概念

用池来管理Connection，这样可以重复使用Connection。有了池，所以我们就不用自己来创建Connection，而是通过池来获取Connection对象。当使用完Connection后，调用Connection的close()方法也不会真的关闭Connection，而是把Connection“归还”给池。池就可以再利用这个Connection对象了。



* 规范

Java为数据库连接池提供了公共的接口：**javax.sql.DataSource**，各个厂商需要让自己的连接池实现这个接口。这样应用程序可以方便的切换不同厂商的连接池！

常见的连接池：DBCP、C3P0。

接下来，我们就详细的学习连接池。

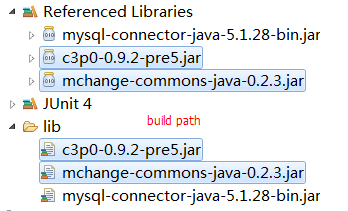
## c3p0连接池--核心类

### c3p0概述

C3P0开源免费的连接池！目前使用它的开源项目有：Spring、Hibernate等。使用第三方工具需要导入jar包，c3p0使用时还需要添加配置文件 c3p0-config.xml

### 准备工作

我们使用的0.9.2版本，需要导入2个jar包。如果使用的是0.9.2之前的版本，则只需要导入一个c3p0的包即可。



### 案例代码一

|  |
| --- |
| package com.igeek\_01\_c3p0;  import java.sql.Connection;  import org.junit.Test;  import com.mchange.v2.c3p0.ComboPooledDataSource;  /\*\*  \*  \* @ClassName: C3P0Demo\_01  \* @Description: TODO(C3P0的基本配置)  \* @date 2018年1月30日 下午1:44:42  \* Company www.igeekhome.com  \*  \*/  public class C3P0Demo\_01 {    @Test  public void demo01() throws Exception{    //1 获得连接池(数据源)  ComboPooledDataSource dataSource = new ComboPooledDataSource();  //1.1 设置基本项  dataSource.setDriverClass("com.mysql.jdbc.Driver");  dataSource.setJdbcUrl("jdbc:mysql://localhost:3306/igeek");  dataSource.setUser("root");  dataSource.setPassword("root");  //1.2其他项  // \* 初始化连接池中连接的个数  dataSource.setInitialPoolSize(5);  // \* 最小|最大 连接池中连接的个数  dataSource.setMinPoolSize(2);  dataSource.setMaxPoolSize(10);  // \* 最大空闲数  dataSource.setMaxIdleTime(60);  // \* 每次增长个数  dataSource.setAcquireIncrement(2);    //2获得连接  Connection conn = dataSource.getConnection();  System.out.println(conn);  }  } |

### 常见配置项

|  |  |  |
| --- | --- | --- |
| 分类 | 属性 | 描述 |
| 必须项 | user | 用户名 |
| password | 密码 |
| driverClass | 驱动  mysql驱动，com.mysql.jdbc.Driver |
| jdbcUrl | 路径  mysql路径，jdbc:mysql://localhost:3306/数据库 |
| 基本配置 | acquireIncrement | 连接池无空闲连接可用时，一次性创建的新连接数  默认值：3 |
| initialPoolSize | 连接池初始化时创建的连接数  默认值：3 |
| maxPoolSize | 连接池中拥有的最大连接数  默认值：15 |
| minPoolSize | 连接池保持的最小连接数。 |
| maxIdleTime | 连接的最大空闲时间。如果超过这个时间，某个数据库连接还没有被使用，则会断开掉这个连接，如果为0，则永远不会断开连接。  默认值：0 |
| 管理连接池的大小和连接的生存时间（扩展） | maxConnectionAge | 配置连接的生存时间，超过这个时间的连接将由连接池自动断开丢弃掉。当然正在使用的连接不会马上断开，而是等待它close再断开。配置为0的时候则不会对连接的生存时间进行限制。默认值0 |
| maxIdleTimeExcessConnections | 这个配置主要是为了减轻连接池的负载，配置不为0，则会将连接池中的连接数量保持到minPoolSize，为0则不处理。 |
| 配置Prepared  Statement缓存（扩展） | maxStatements | 连接池为数据源缓存的PreparedStatement的总数。由于PreparedStatement属于单个Connection,所以这个数量应该根据应用中平均连接数乘以每个连接的平均PreparedStatement来计算。为0的时候不缓存，同时maxStatementsPerConnection的配置无效。 |
| maxStatementsPerConnection | 连接池为数据源单个Connection缓存的PreparedStatement数，这个配置比maxStatements更有意义，因为它缓存的服务对象是单个数据连接，如果设置的好，肯定是可以提高性能的。为0的时候不缓存。 |

## c3p0连接池--配置文件使用

### 配置项说明

* 配置文件名称：c3p0-config.xml （固定）
* 配置文件位置：src （类路径）
* 配置文件内容：默认配置

|  |
| --- |
| <c3p0-config>  <!-- 默认配置 -->  <default-config>  <property name=*"driverClass"*>com.mysql.jdbc.Driver</property>  <property name=*"jdbcUrl"*>jdbc:mysql://127.0.0.1:3306/igeek</property>  <property name=*"user"*>root</property>  <property name=*"password"*>root</property>  <property name=*"checkoutTimeout"*>30000</property>  <property name=*"idleConnectionTestPeriod"*>30</property>  <property name=*"initialPoolSize"*>10</property>  <property name=*"maxIdleTime"*>30</property>  <property name=*"maxPoolSize"*>100</property>  <property name=*"minPoolSize"*>10</property>  <property name=*"maxStatements"*>200</property>  <user-overrides user=*"test-user"*>  <property name=*"maxPoolSize"*>10</property>  <property name=*"minPoolSize"*>1</property>  <property name=*"maxStatements"*>0</property>  </user-overrides>  </default-config>  </c3p0-config> |

* 配置文件内容：命名配置

|  |
| --- |
| <c3p0-config>  <named-config name=*"igeekhome"*>  <!-- 连接数据库的4项基本参数 -->  <property name=*"driverClass"*>com.mysql.jdbc.Driver</property>  <property name=*"jdbcUrl"*>jdbc:mysql://127.0.0.1:3306/igeek</property>  <property name=*"user"*>root</property>  <property name=*"password"*>root</property>  <!-- 如果池中数据连接不够时一次增长多少个 -->  <property name=*"acquireIncrement"*>5</property>  <!-- 初始化连接数 -->  <property name=*"initialPoolSize"*>20</property>  <!-- 最小连接受 -->  <property name=*"minPoolSize"*>10</property>  <!-- 最大连接数 -->  <property name=*"maxPoolSize"*>40</property>  <!-- -JDBC的标准参数，用以控制数据源内加载的PreparedStatements数量 -->  <property name=*"maxStatements"*>0</property>  <!-- 连接池内单个连接所拥有的最大缓存statements数 -->  <property name=*"maxStatementsPerConnection"*>5</property>  </named-config>  </c3p0-config> |

### 案例代码二

**C3P0Demo\_02 类**

|  |
| --- |
| **package com.igeek\_01\_c3p0;**  **import java.sql.Connection;**  **import org.junit.Test;**  **import com.mchange.v2.c3p0.ComboPooledDataSource;**  **/\*\***  **\***  **\* @ClassName: C3P0Demo\_02**  **\* @Description: TODO(使用c3p0-config.xml进行c3p0的配置)**  **\* @date 2018年1月30日 下午1:45:20**  **\* Company www.igeekhome.com**  **\***  **\*/**  **public class C3P0Demo\_02 {**  **@Test**  **public void demo01() throws Exception {**  **// 1 获得连接池(数据源)**  **// \* c3p0 jar包将自动加载“c3p0-config.xml”文件，并获得具体的配置信息**  **// \*\* 在配置文件中 <named-config name="名称"> ，可以通过核心类 new ComboPooledDataSource("名称")**  **ComboPooledDataSource dataSource = new ComboPooledDataSource("igeekhome");**  **// 2获得连接**  **Connection conn = dataSource.getConnection();**  **System.out.println(conn);**  **}**  **}** |

**c3p0-config.xml 文件**

|  |
| --- |
| <c3p0-config>  <!-- 默认配置 -->  <default-config>  <property name=*"driverClass"*>com.mysql.jdbc.Driver</property>  <property name=*"jdbcUrl"*>jdbc:mysql://127.0.0.1:3306/igeek</property>  <property name=*"user"*>root</property>  <property name=*"password"*>root</property>  <property name=*"checkoutTimeout"*>30000</property>  <property name=*"idleConnectionTestPeriod"*>30</property>  <property name=*"initialPoolSize"*>10</property>  <property name=*"maxIdleTime"*>30</property>  <property name=*"maxPoolSize"*>100</property>  <property name=*"minPoolSize"*>10</property>  <property name=*"maxStatements"*>200</property>  <user-overrides user=*"test-user"*>  <property name=*"maxPoolSize"*>10</property>  <property name=*"minPoolSize"*>1</property>  <property name=*"maxStatements"*>0</property>  </user-overrides>  </default-config>  <!-- 命名的配置 -->  <named-config name=*"igeekhome"*>  <!-- 连接数据库的4项基本参数 -->  <property name=*"driverClass"*>com.mysql.jdbc.Driver</property>  <property name=*"jdbcUrl"*>jdbc:mysql://127.0.0.1:3306/igeek</property>  <property name=*"user"*>root</property>  <property name=*"password"*>root</property>  <!-- 如果池中数据连接不够时一次增长多少个 -->  <property name=*"acquireIncrement"*>5</property>  <!-- 初始化连接数 -->  <property name=*"initialPoolSize"*>20</property>  <!-- 最小连接受 -->  <property name=*"minPoolSize"*>10</property>  <!-- 最大连接数 -->  <property name=*"maxPoolSize"*>40</property>  <!-- -JDBC的标准参数，用以控制数据源内加载的PreparedStatements数量 -->  <property name=*"maxStatements"*>0</property>  <!-- 连接池内单个连接所拥有的最大缓存statements数 -->  <property name=*"maxStatementsPerConnection"*>5</property>  </named-config>      </c3p0-config> |

## c3p0连接池--工具类

C3P0提供核心工具类：ComboPooledDataSource，如果要使用连接池，必须创建该类的实例对象。

* + new ComboPooledDataSource(“名称”); 使用配置文件“命名配置”

<named-config name="igeekhome">

* + new ComboPooledDataSource(); 使用配置文件“默认配置”

<default-config>

### 案例代码三

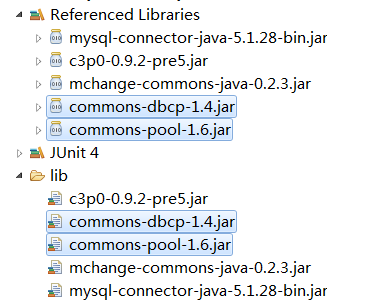
|  |
| --- |
| **package** com.igeek\_01\_c3p0;  **import** java.sql.Connection;  **import** java.sql.SQLException;  **import** javax.sql.DataSource;  **import** com.mchange.v2.c3p0.ComboPooledDataSource;  /\*\*  \* **@ClassName**: C3P0Utils  \* **@Description**: 等效 JdbcUtils，用于提供获得连接工具类  \* **@date** 2018年1月30日 下午1:56:50 Company www.igeekhome.com  \*  \*/  **public** **class** C3P0Utils {  // 连接池  **private** **static** ComboPooledDataSource *dataSource* = **new** ComboPooledDataSource("igeekhome");  /\*\*  \* 获得数据源(连接池)  \*  \* **@return**  \*/  **public** **static** DataSource getDataSource() {  **return** *dataSource*;  }  /\*\*  \* 获得连接  \*  \* **@return**  \* **@throws** SQLException  \*/  **public** **static** Connection getConnection() **throws** SQLException {  // 将从连接池中获得连接  **return** *dataSource*.getConnection();  }  } |

# dbcp连接池

## dbcp连接池--核心类

DBCP也是一个开源的连接池，是Apache Common成员之一，在企业开发中也比较常见，tomcat内置的连接池。

### 准备工作



### 常见配置项

|  |  |  |
| --- | --- | --- |
| 分类 | 属性 | 描述 |
| 必须项 | driverClassName |  |
| url |  |
| Username |  |
| Password |  |
| 基本项 | maxActive | 最大连接数量 |
| minIdle | 最小空闲连接 |
| maxIdle | 最大空闲连接 |
| initialSize | 初始化连接 |
|  |  |  |
| 优化配置（扩展） | logAbandoned | 连接被泄露时是否打印 |
| removeAbandoned | 是否自动回收超时连接 |
| removeAbandonedTimeout | 超时时间(以秒数为单位) |
| maxWait | 超时等待时间以毫秒为单位 1000等于60秒 |
| timeBetweenEvictionRunsMillis | 在空闲连接回收器线程运行期间休眠的时间值,以毫秒为单位 |
| numTestsPerEvictionRun | 在每次空闲连接回收器线程(如果有)运行时检查的连接数量 |
| minEvictableIdleTimeMillis | 连接在池中保持空闲而不被空闲连接回收器线程 |

参考文档：http://commons.apache.org/proper/commons-dbcp/configuration.html

### 案例代码四

|  |
| --- |
| **package** com.igeek\_02\_dbcp;  **import** java.sql.Connection;  **import** org.apache.commons.dbcp.BasicDataSource;  **import** org.junit.Test;  /\*\*  \* **@ClassName**: DBCPDemo\_01  \* **@Description**: DBCP的基本配置  \* **@date** 2018年1月30日 下午2:01:42  \* Company www.igeekhome.com  \*  \*/  **public** **class** DBCPDemo\_01 {  @Test  **public** **void** demo01() **throws** Exception{  //1 获得连接池  BasicDataSource dataSource = **new** BasicDataSource();  //1.1 基本项  dataSource.setDriverClassName("com.mysql.jdbc.Driver");  dataSource.setUrl("jdbc:mysql://localhost:3306/igeek");  dataSource.setUsername("root");  dataSource.setPassword("root");    //1.2 其他项  // \* 初始化连接池中连个的个数  dataSource.setInitialSize(5);  // \* 最大活动数  dataSource.setMaxActive(10);    //2获得连接  Connection conn = dataSource.getConnection();  System.***out***.println(conn);    }  } |

## dbcp连接池--配置文件使用

* 配置文件名称：\*.properties
* 配置文件位置：任意，建议src（classpath/类路径）
* 配置文件内容：properties不能编写中文，不支持在STS中修改，必须使用记事本修改内容，否则中文注释就乱码了

|  |
| --- |
| #连接设置  driverClassName=com.mysql.jdbc.Driver  url=jdbc:mysql://localhost:3306/webdb  username=root  password=root  #<!-- 初始化连接 -->  initialSize=10  #最大连接数量  maxActive=50  #<!-- 最大空闲连接 -->  maxIdle=20  #<!-- 最小空闲连接 -->  minIdle=5  #<!-- 超时等待时间以毫秒为单位 6000毫秒/1000等于60秒 -->  maxWait=60000  #JDBC驱动建立连接时附带的连接属性属性的格式必须为这样：[属性名=property;]  #注意："user" 与 "password" 两个属性会被明确地传递，因此这里不需要包含他们。  connectionProperties=useUnicode=true;characterEncoding=gbk  #指定由连接池所创建的连接的自动提交（auto-commit）状态。  defaultAutoCommit=true  #driver default 指定由连接池所创建的连接的只读（read-only）状态。  #如果没有设置该值，则“setReadOnly”方法将不被调用。（某些驱动并不支持只读模式，如：Informix）  defaultReadOnly=  #driver default 指定由连接池所创建的连接的事务级别（TransactionIsolation）。  #可用值为下列之一：（详情可见javadoc。）NONE,READ\_UNCOMMITTED, READ\_COMMITTED, REPEATABLE\_READ, SERIALIZABLE  defaultTransactionIsolation=READ\_UNCOMMITTED |

### 案例代码五

**DBCPDemo\_02类**

|  |
| --- |
| **package** com.igeek\_02\_dbcp;  **import** java.io.InputStream;  **import** java.sql.Connection;  **import** java.util.Properties;  **import** javax.sql.DataSource;  **import** org.apache.commons.dbcp.BasicDataSourceFactory;  **import** org.junit.Test;  /\*\*  \* **@ClassName**: DBCPDemo\_02  \* **@Description**: 使用配置文件配置dbcp  \* **@date** 2018年1月30日 下午2:11:33  \* Company www.igeekhome.com  \*  \*/  **public** **class** DBCPDemo\_02 {  @Test  **public** **void** demo01() **throws** Exception{  //1 获得连接池  //1.1 加载properties文件，获得Properties对象  InputStream is = DBCPDemo\_02.**class**.getClassLoader().getResourceAsStream("dbcp-config.properties");  Properties props = **new** Properties();  props.load(is);    //1.2获得连接池  DataSource dataSource = BasicDataSourceFactory.*createDataSource*(props);      //2获得连接  Connection conn = dataSource.getConnection();  System.***out***.println(conn);    }  } |

**dbcp-config.properties 配置文件**

#基本配置内容

driverClassName=com.mysql.jdbc.Driver

url=jdbc:mysql://localhost:3306/igeek

username=root

password=root

initialSize=10

maxActive=50

maxIdle=20

minIdle=5

maxWait=60000

connectionProperties=useUnicode=true;characterEncoding=UTF-8

defaultAutoCommit=true

defaultReadOnly=NONE

defaultTransactionIsolation=READ\_UNCOMMITTED

## dbcp连接池--工具类

### 工具类概述

工具类提供两个功能，分别是提供连接池对象，提供连接对象

### 案例练习五

**工具类DBCPUtils**

|  |
| --- |
| **package** com.igeek\_02\_dbcp;  **import** java.io.InputStream;  **import** java.sql.Connection;  **import** java.sql.SQLException;  **import** java.util.Properties;  **import** javax.sql.DataSource;  **import** org.apache.commons.dbcp.BasicDataSourceFactory;  /\*\*  \* **@ClassName**: DBCPUtils  \* **@Description**: DBCP工具类  \* **@date** 2018年1月30日 下午2:13:55  \* Company www.igeekhome.com  \*  \*/  **public** **class** DBCPUtils {  //1 创建连接池  **private** **static** DataSource *dataSource*;  // \* 编写固定代码  **static**{  **try** {  //1) 加载properties文件，获得Properties对象  InputStream is = DBCPUtils.**class**.getClassLoader().getResourceAsStream("dbcp-config.properties");  Properties prop = **new** Properties();  prop.load(is);  //2) 通过工厂，创建连接池  *dataSource* = BasicDataSourceFactory.*createDataSource*(prop);  } **catch** (Exception e) {  **throw** **new** RuntimeException(e);  }  }      /\*\*  \* 获得数据库(连接池)  \* **@return**  \*/  **public** **static** DataSource getDataSource(){  **return** *dataSource*;  }      /\*\*  \* 获得连接  \* **@return**  \* **@throws** SQLException  \*/  **public** **static** Connection getConnection() **throws** SQLException{  **return** *dataSource*.getConnection();  }    } |

**测试类TestDBCP类**

|  |
| --- |
| **package** com.igeek\_02\_dbcp;  **import** java.sql.Connection;  **import** java.sql.SQLException;  **import** org.junit.Test;  /\*\*  \* **@ClassName**: TestDBCP  \* **@Description**: 测试类  \* **@date** 2018年1月30日 下午2:15:58  \* Company www.igeekhome.com  \*  \*/  **public** **class** TestDBCP {  @Test  **public** **void** demo01() **throws** SQLException{    Connection conn = DBCPUtils.*getConnection*();    System.***out***.println(conn);  }  } |

# DBUtils工具类

## 概述

DBUtils是java编程中的数据库操作实用工具，小巧简单实用。

DBUtils封装了对JDBC的操作，简化了JDBC操作，可以少写代码。

Dbutils三个核心功能介绍

* QueryRunner中提供对sql语句操作的API.
* ResultSetHandler接口，用于定义select操作后，怎样封装结果集.
* DbUtils类，它就是一个工具类,定义了关闭资源与事务处理的方法

## 准备数据

* 创建表：

|  |
| --- |
| **create** **table** product(  pid **int** **primary** **key**,  pname **varchar**(20),  price **double**,  category\_id **varchar**(32)  ); |

* 插入表记录：

|  |
| --- |
| **INSERT** **INTO** product(pid,pname,price,category\_id) **VALUES**(1,'IBM',5500,'c001');  **INSERT** **INTO** product(pid,pname,price,category\_id) **VALUES**(2,'长城',3200,'c001');  **INSERT** **INTO** product(pid,pname,price,category\_id) **VALUES**(3,'惠普',5000,'c001');  **INSERT** **INTO** product(pid,pname,price,category\_id) **VALUES**(4,'华为',3800,'c002');  **INSERT** **INTO** product(pid,pname,price,category\_id) **VALUES**(5,'小米',2000,'c002');  **INSERT** **INTO** product(pid,pname,price,category\_id) **VALUES**(6,'OPPO',3400,'c002');  **INSERT** **INTO** product(pid,pname,price,category\_id) **VALUES**(7,'NOKIA',2000,'c002');  **INSERT** **INTO** product(pid,pname,price,category\_id) **VALUES**(8,'佰草集',800,'c003');  **INSERT** **INTO** product(pid,pname,price,category\_id) **VALUES**(9,'玉兰油',200,'c003');  **INSERT** **INTO** product(pid,pname,price,category\_id) **VALUES**(10,'大宝',5,'c003'); |

## QueryRunner核心类介绍

### 提供数据源

* 构造方法
  + QueryRunner(DataSource) 创建核心类，并提供数据源，内部自己维护Connection
* 普通方法
  + update(String sql , Object ... params) 执行DML语句
  + query(String sql , ResultSetHandler , Object ... params) 执行DQL语句，并将查询结果封装到对象中。

### 提供连接

* 构造方法
  + QueryRunner() 创建核心类，**没有**提供数据源，在进行具体操作时，需要手动提供Connection
* 普通方法
  + update(Connection conn , String sql , Object ... params) 使用提供的Connection，完成DML语句
  + query(Connection conn , String sql , ResultSetHandler , Object ... params) 使用提供的Connection，执行DQL语句，并将查询结果封装到对象中。

## QueryRunner实现添加、更新、删除操作

* update(String sql, Object... params) ，用来完成表数据的增加、删除、更新操作

### 添加

|  |
| --- |
| **public** **void** insert() **throws** SQLException{  //将sql和实际参数 进行抽取  //1 核心类  QueryRunner queryRunner = **new** QueryRunner(C3P0Utils.*getDataSource*());    //2 执行update方法  String sql = "insert into product(pid,pname,price,category\_id) values(?,?,?,?)";  Object[] params = { 990,"测试",100,"c009" };  **int** r = queryRunner.update(sql,params);    System.***out***.println(r);    } |

### 更新

|  |
| --- |
| **public** **void** update() **throws** SQLException{  //1 核心类  QueryRunner queryRunner = **new** QueryRunner(C3P0Utils.*getDataSource*());  //2 准备sql语句  String sql = "update product set pname=?,price=?,category\_id=? where pid=?";    //3 准备实际参数  Object[] params = {"芒果99","998","c009",13};    //4 执行  **int** r = queryRunner.update(sql, params);  System.***out***.println(r);    } |

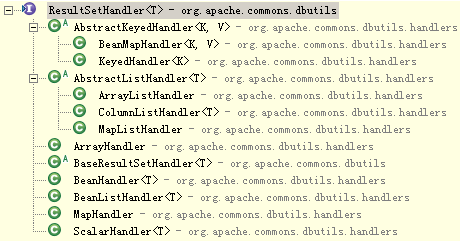
### 删除

|  |
| --- |
| **public** **void** delete() **throws** SQLException{    QueryRunner queryRunner = **new** QueryRunner(C3P0Utils.*getDataSource*());  String sql = "delete from product where pid = ?";  Object[] params = {99};  **int** r = queryRunner.update(sql, params);  System.***out***.println(r);    } |

## QueryRunner实现查询操作

* query(String sql, ResultSetHandler<T> rsh, Object... params) ，用来完成表数据的查询操作

### ResultSetHandler结果集处理类



|  |  |
| --- | --- |
| ArrayHandler | 将结果集中的第一条记录封装到一个Object[]数组中，数组中的每一个元素就是这条记录中的每一个字段的值 |
| ArrayListHandler | 将结果集中的每一条记录都封装到一个Object[]数组中，将这些数组在封装到List集合中。 |
| BeanHandler | 将结果集中第一条记录封装到一个指定的javaBean中。 |
| BeanListHandler | 将结果集中每一条记录封装到指定的javaBean中，将这些javaBean在封装到List集合中 |
| ColumnListHandler | 将结果集中指定的列的字段值，封装到一个List集合中 |
| KeyedHandler | 将结果集中每一条记录封装到Map<String,Object>,在将这个map集合做为另一个Map的value,另一个Map集合的key是指定的字段的值。 |
| MapHandler | 将结果集中第一条记录封装到了Map<String,Object>集合中，key就是字段名称，value就是字段值 |
| MapListHandler | 将结果集中每一条记录封装到了Map<String,Object>集合中，key就是字段名称，value就是字段值，在将这些Map封装到List集合中。 |
| ScalarHandler | 它是用于单数据。例如select count(\*) from 表操作。 |

### JavaBean

JavaBean就是一个类，在开发中常用语封装数据。具有如下特性

1. 需要实现接口：java.io.Serializable ，通常实现接口这步骤省略了，不会影响程序。
2. 提供私有字段：private 类型 字段名;
3. 提供getter/setter方法：
4. 提供无参构造

|  |
| --- |
| **public** **class** Product {    **private** String pid;  **private** String pname;  **private** Double price;  **private** String category\_id;    **// 省略getter/setter方法**  } |

### BeanHandler

|  |
| --- |
| /\*\*  \* 查询数据表结果集处理其中一种方式:  \* BeanHandler处理方式  \* 将数据表的结果集第一行数据,封装成JavaBean类的对象  \* 构造方法:  \* BeanHandler(Class<T> type)  \* 传递一个Class类型对象,将结果封装到哪个类的对象呢  \* ZhangWu类的Class对象  \*/  @Test  **public** **void** demo01() **throws** SQLException{  // 通过id查询详情，将查询结果封装到JavaBean product    //1核心类  QueryRunner queryRunner = **new** QueryRunner(C3P0Utils.*getDataSource*());  //2 sql语句  String sql = "select \* from product where pid = ?";  //3 实际参数  Object[] params = {6};  //4 查询并封装  Product product = queryRunner.query(sql, **new** BeanHandler<Product>(Product.**class**), params);    System.***out***.println(product);    } |

### BeanListHandler

|  |
| --- |
| /\*\*  \* 查询数据表结果集处理其中一种方式:  \* BeanListHandler处理方式  \* 将数据表的每一行数据,封装成JavaBean类对象  \* 多行数据了,多个JavaBean对象,存储List集合  \*/  @Test  **public** **void** demo02() **throws** SQLException{  //查询所有，将每一条记录封装到一个JavaBean，然后将JavaBean添加到List中，最后返回List，BeanListHandler  QueryRunner queryRunner = **new** QueryRunner(C3P0Utils.*getDataSource*());  String sql = "select \* from product";  Object[] params = {};  List<Product> list = queryRunner.query(sql, **new** BeanListHandler<Product>(Product.**class**), params);    **for**(Product product : list){  System.***out***.println(product);  }  } |

### ScalarHander

|  |
| --- |
| /\*\*  \* 查询数据表结果集处理其中一种方式:  \* ScalarHandler处理方式  \* 处理单值查询结果,执行的select语句后,结果集只有1个  \*/  @Test  **public** **void** demo03() **throws** SQLException{  // ScalarHandler : 用于处理聚合函数执行结果(一行一列)  // \* 查询总记录数  QueryRunner queryRunner = **new** QueryRunner(C3P0Utils.*getDataSource*());  String sql = "select count(\*) from product";    Long obj = queryRunner.query(sql, **new** ScalarHandler<Long>());    //System.out.println(obj.getClass());  System.***out***.println(obj);  } |

### MapHandler

|  |
| --- |
| /\*\*  \* 查询数据表结果集处理其中一种方式:  \* MapHandler处理方式  \* 将数据表结果集的第一行数据,封装成Map集合  \* 键: 数据表中的列  \* 值: 这个列中的数据  \*  \* 处理方式的Map集合,是LinkedHashMap的子类  \*/  @Test  **public** **void** demo04() **throws** SQLException{  // MapHandler ： 将查询到的一条记录，封装到Map中，map.key=字段名，map.value=值  // \* 主要用途：多表操作、将数据转换json 等  QueryRunner queryRunner = **new** QueryRunner(C3P0Utils.*getDataSource*());  String sql = "select \* from product where pid = ?";  Object[] params = {6};  Map<String,Object> map = queryRunner.query(sql, **new** MapHandler(), params);    System.***out***.println(map);  // 将Map数据封装到指定JavaBean  } |

### MapListHandler

|  |
| --- |
| /\*\*  \* 查询数据表结果集其中一种处理方式:  \* MapListHandler处理方式  \* 将数据表的结果集的每一行封装成Map集合  \* 数据表多行数据,出现多个Map集合,存储List集合  \*/  @Test  **public** **void** demo05() **throws** SQLException{  // MapListHandler ： 查询所有数据，将每一条记录封装到Map中，然后将Map添加到List中，最后返回List  // \* 主要用途：多表操作 等  QueryRunner queryRunner = **new** QueryRunner(C3P0Utils.*getDataSource*());  String sql = "select \* from product";  List<Map<String,Object>> list = queryRunner.query(sql, **new** MapListHandler());    **for**(Map<String,Object> map : list){  System.***out***.println(map);  }  } |

### ArrayHandler

|  |
| --- |
| /\*\*  \* 查询数据表结果集处理其中一种方式:  \* ArrayHandler处理方式  \* 将数据表中的第一行数据,存储到对象数组Object[]中  \*  \* 注意: 获取查询后的第一行数据,如果查询不到结果集,返回的对象数组的length=0  \*/  @Test  **public** **void** demo06() **throws** SQLException{  // ArrayHandler ：查询一条记录，将数据封装到数组中  QueryRunner queryRunner = **new** QueryRunner(C3P0Utils.*getDataSource*());  String sql = "select \* from product where pid = ?";  Object[] params = {6};  Object[] arr = queryRunner.query(sql, **new** ArrayHandler(), params);    System.***out***.println(arr);  System.***out***.println(Arrays.*toString*(arr));  } |

### ArrayListHandler

|  |
| --- |
| /\*\*  \* 查询数据表结果集处理其中一种方式:  \* ArrayListHandler处理方式  \* 将数据表中的每一行数据,存储到一个对象数组Object[]中  \* 而数据表中会有多行数据,产生多个对象数组, 存储到List集合中  \*/  @Test  **public** **void** demo07() **throws** SQLException{  // ArrayListHandler ：查询所有，将每一条记录封装到数组中，然后添加到List，最后返回list  QueryRunner queryRunner = **new** QueryRunner(C3P0Utils.*getDataSource*());  String sql = "select \* from product";  List<Object[]> list = queryRunner.query(sql, **new** ArrayListHandler());  **for**(Object[] arr : list){  System.***out***.println(Arrays.*toString*(arr));  }  } |

### KeyedHandler

|  |
| --- |
| @Test  **public** **void** demo08() **throws** SQLException{  // KeyedHandler ： new KeyedHandler("字段名称")，查询所有，将查询结果封装到Map中  // \* map.key=为指定“字段名称”对应的值  // \* map.value=为当前整条记录所有的值，数据为Map<字段名,值>  // 类型 Map<String , Map<String,Object> >  QueryRunner queryRunner = **new** QueryRunner(C3P0Utils.*getDataSource*());  String sql = "select \* from product";  Map<String,Map<String,Object>> map = queryRunner.query(sql, **new** KeyedHandler<String>("pname"));  **for**(Map.Entry<String, Map<String,Object>> entry : map.entrySet()){  System.***out***.println(entry.getKey());  System.***out***.println(entry.getValue());  }    } |

### ColumnListHandler

|  |
| --- |
| /\*\*  \* 查询数据表结果集处理其中一种方式:  \* ColumnListHandler处理方式  \* 将查询数据表结果集中的某一列数据,存储到List集合  \* 哪个列不清楚,数据类型也不清楚, List<Object>  \* ColumnListHandler构造方法  \* 空参数: 获取就是数据表的第一列  \* int参数: 传递列的顺序编号  \* String参数: 传递列名  \*  \* 创建对象,可以加入泛型,但是加入的数据类型,要和查询的列类型一致  \*/  @Test  **public** **void** demo09() **throws** SQLException{  // ColumnListHandler ： 查询指定一列数据  QueryRunner queryRunner = **new** QueryRunner(C3P0Utils.*getDataSource*());  String sql = "select \* from product";  List<String> list = queryRunner.query(sql, **new** ColumnListHandler<String>("pname"));  System.***out***.println(list);  } |