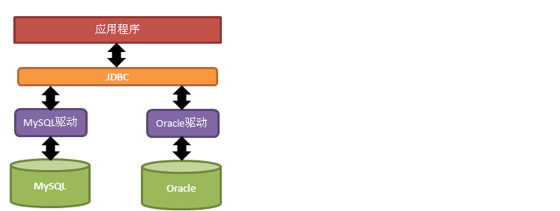
JDBC基础

JDBC(Java Database Connectivity)代表Java编程语言与数据库连接的标准API,然而JDBC只是接口,JDBC驱动才是真正的接口实现,没有驱动无法完成数据库连接. 每个数据库厂商都有自己的驱动,用来连接自己公司的数据库(如Oricle, MySQL, DB2, MS SQLServer).



下面我们以MySQL为例,JDBC编程大致步骤如下:

|  |
| --- |
| public class SQLClient {  public static void main(String[] args) throws ClassNotFoundException, SQLException {  /\* 第一步：加载数据库驱动 \*/  Class.forName("com.mysql.jdbc.Driver");  /\* 第二步：通过 DriverManager 获取数据库连接 \*/  Connection connection = DriverManager.getConnection("jdbc:mysql://host:port/database", "user", "password");    /\* 第三步：通过 Connection 创建 Statement \*/  Statement statement = connection.createStatement();    /\* 第四步：通过 Statement 执行SQL \*/  ResultSet users = statement.executeQuery("SELECT \* FROM user");    /\* 第五步：操作 ResultSet 结果集 \*/  int columnCount = users.getMetaData().getColumnCount();  while (users.next()) {  for (int i = 1; i <= columnCount; ++i) {  System.out.printf("%s\t", users.getObject(i));  }  System.out.println();  }  /\* 第六步：回收数据库资源(推荐使用Java1.7提供的 可以自动关闭资源的try) \*/  users.close();  statement.close();  connection.close();  }  }  注意: 需要在pom.xml中添加如下MySQL驱动:  <dependency>  <groupId>mysql</groupId>  <artifactId>mysql-connector-java</artifactId>  <version>5.1.36</version>  </dependency>  注: ResultSet参数columnIndex索引从1开始,而不是0! |

# 数据库连接

## DriverManger

JDBC规定: 驱动类在被加载时,需要主动把自己注册到DriverManger中:

|  |
| --- |
| com.mysql.jdbc.Driver:  public class Driver extends NonRegisteringDriver implements java.sql.Driver {   //   // Register ourselves with the DriverManager   //   static {   try {   java.sql.DriverManager.registerDriver(new Driver());   } catch (SQLException E) {   throw new RuntimeException("Can't register driver!");   }   }  /\*\*   \* Construct a new driver and register it with DriverManager   \*   \* @throws SQLException   \* if a database error occurs.   \*/   public Driver() throws SQLException {   // Required for Class.forName().newInstance()   }  }  代码显示:只要去加载com.mysql.jdbc.Driver类那么就会执行static块, 从而把com.mysql.jdbc.Driver注册到DriverManager中.  java.sql.DriverManager是用于管理JDBC驱动的服务类,其主要功能是获取Connection对象:   * static Connection getConnection(String url, Properties info) * static Connection getConnection(String url, String user, String password)   另: 还可以在获取Connection的URL中设置参数,如: jdbc:mysql://host:port/database？useUnicode=true&characterEncoding=UTF8  useUnicode=true&characterEncoding=UTF8指定连接数据库的过程中使用Unicode字符集/UTF-8编码; |

## Connection

java.sql.Connection代表数据库连接,每个Connection代表一个物理连接会话, 该接口提供如下创建Statement的方法, 只有获取Statement之后才可执行SQL语句：

| **方法** | | **描述** |
| --- | --- | --- |
| Statement createStatement() | Creates a Statement object for sending SQL statements to the database. | |
| PreparedStatement prepareStatement(String sql) | Creates a PreparedStatement object for sending parameterized SQL statements to the database. | |
| CallableStatement prepareCall(String sql) | Creates a CallableStatement object for calling database stored procedures. | |

其中Connection还提供了如下控制事务/保存点的方法:

| **方法** | **描述** |
| --- | --- |
| Savepoint setSavepoint(String name) | Creates a savepoint with the given name in the current transaction and returns the new Savepoint object that represents it. |
| void setTransactionIsolation(int level) | Attempts to change the transaction isolation level(事务隔离级别) for this Connection object to the one given. |
| void setAutoCommit(boolean autoCommit) | Sets this connection’s auto-commit mode to the given state. |
| void rollback() | Undoes all changes made in the current transaction and releases any database locks currently held by this Connection object. |
| void rollback(Savepoint savepoint) | Undoes all changes made after the given Savepoint object was set. |
| void commit() | Makes all changes made since the previous commit/rollback permanent and releases any database locks currently held by this Connection object. |

## ConnectionManger

由于获取Connection的步骤单一,每次可能只是加载的参数不同,因此我们可以将获取Connection的操作封装成一个方法,并使其从配置文件中加载配置:

|  |  |  |
| --- | --- | --- |
| 配置文件形式：  ## Data Source  mysql.driver.class=com.mysql.jdbc.Driver  mysql.url=jdbc:mysql://host:port/database  mysql.user=admin  mysql.password=admin   |  |  | | --- | --- | |  |  | |
| * /\* ConnectionManger \*/   public class ConnectionManger {  /\*获取原生Connection\*/  public static Connection getConnection(String file) {   Properties config = SQLUtil.loadConfig(file);   try {   Class.forName(config.getProperty("mysql.driver.class"));   String url = config.getProperty("mysql.url");   String username = config.getProperty("mysql.user");   String password = config.getProperty("mysql.password");   return DriverManager.getConnection(url, username, password);   } catch (SQLException | ClassNotFoundException e) {   throw new RuntimeException(e);   }   }  } |
| /\* SQLUtil \*/  public class SQLUtil {  /\*\*   \* 加载.properties配置文件   \*   \* @param file   \* @return   \*/   public static Properties loadConfig(String file) {  Properties properties = new Properties();   try {   properties.load(ClassLoader.getSystemResourceAsStream(file));   return properties;   } catch (IOException e) {   throw new RuntimeException(e);   }   }  } |

## 数据库连接池

前面通过DriverManger获得Connection, 一个Connection对应一个实际的物理连接,每次操作都需要打开物理连接, 使用完后立即关闭;这样频繁的打开/关闭连接会造成不必要的数据库系统性能消耗.

数据库连接池提供的解决方案是:当应用启动时,主动建立足够的数据库连接,并将这些连接组织成连接池,每次请求连接时,无须重新打开连接,而是从池中取出已有连接,使用完后并不实际关闭连接,而是归还给池.

JDBC数据库连接池使用javax.sql.DataSource表示, DataSource只是一个接口, 其实现通常由服务器提供商(如WebLogic, WebShere)或开源组织(如DBCP,C3P0和HikariCP)提供.数据库连接池的常用参数如下:

* 数据库初始连接数;
* 连接池最大连接数;
* 连接池最小连接数;
* 连接池每次增加的容量;

### C3P0

Tomcat默认使用的是DBCP连接池,但相比之下,C3P0则比DBCP更胜一筹(Hibernate推荐使用C3P0),C3P0不仅可以自动清理不再使用的Connection, 还可以自动清理Statement/ResultSet, 使用C3P0需要在pom.xml中添加如下依赖:

|  |
| --- |
| <dependency>   <groupId>com.mchange</groupId>   <artifactId>c3p0</artifactId>   <version>0.9.5.2</version>  </dependency>  <dependency>   <groupId>com.mchange</groupId>   <artifactId>mchange-commons-java</artifactId>   <version>0.2.11</version>  </dependency> |
| public class ConnectionManger {  /\*双重检测锁保证DataSource单例\*/   private static DataSource dataSource;  /\*获取DataSource\*/  public static DataSource getDataSourceC3P0(String file) {   if (dataSource == null) {   synchronized (ConnectionManger.class) {   if (dataSource == null) {   Properties config = SQLUtil.loadConfig(file);   try {   ComboPooledDataSource source = new ComboPooledDataSource();   source.setDriverClass(config.getProperty("mysql.driver.class"));   source.setJdbcUrl(config.getProperty("mysql.url"));   source.setUser(config.getProperty("mysql.user"));   source.setPassword(config.getProperty("mysql.password"));  // 设置连接池最大连接数   source.setMaxPoolSize(Integer.valueOf(config.getProperty("pool.max.size")));   // 设置连接池最小连接数   source.setMinPoolSize(Integer.valueOf(config.getProperty("pool.min.size")));   // 设置连接池初始连接数   source.setInitialPoolSize(Integer.valueOf(config.getProperty("pool.init.size")));   // 设置连接每次增量   source.setAcquireIncrement(Integer.valueOf(config.getProperty("pool.acquire.increment")));   // 设置连接池的缓存Statement的最大数   source.setMaxStatements(Integer.valueOf(config.getProperty("pool.max.statements")));   // 设置最大空闲时间   source.setMaxIdleTime(Integer.valueOf(config.getProperty("pool.max.idle\_time")));  dataSource = source;   } catch (PropertyVetoException e) {   throw new RuntimeException(e);   }   }   }   }   return dataSource;   }  /\*获取Connection\*/  public static Connection getConnectionC3P0(String file) {   return getConnection(getDataSourceC3P0(file));   }  public static Connection getConnection(DataSource dataSource) {   try {   return dataSource.getConnection();   } catch (SQLException e) {   throw new RuntimeException(e);   }   }  // ...  } |

C3P0还可以使用配置文件来初始化连接池(配置文件可以是properties/XML, 在此仅介绍XML),C3P0配置文件名必须为c3p0-config.xml,其放在类路径下:

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <c3p0-config>   <default-config>   <property name="jdbcUrl">jdbc:mysql://host:port/database</property>   <property name="driverClass">com.mysql.jdbc.Driver</property>   <property name="user">user</property>   <property name="password">password</property>   <property name="acquireIncrement">5</property>   <property name="initialPoolSize">10</property>   <property name="minPoolSize">3</property>   <property name="maxPoolSize">20</property>   </default-config>   <named-config name="mysql-config">   <property name="jdbcUrl">jdbc:mysql://host:port/common</property>   <property name="driverClass">com.mysql.jdbc.Driver</property>   <property name="user">user</property>   <property name="password">password</property>   <property name="acquireIncrement">5</property>   <property name="initialPoolSize">10</property>   <property name="minPoolSize">3</property>   <property name="maxPoolSize">20</property>   </named-config>  </c3p0-config> |
| 这样, 我们在创建ComboPooledDataSource时就默认加载配置文件中的配置, 无须手动配置:  public static DataSource getDataSourceC3P0(String file) {   if (dataSource == null) {   synchronized (ConnectionManger.class) {   if (dataSource == null) {   dataSource = new ComboPooledDataSource();   }   }   }   return dataSource;  } |
| C3P0配置文件可以配置多个连接信息, 并为每个配置命名, 这样可以方便的通过配置名称来切换配置信息:  public static DataSource getDataSourceC3P0(String file) {   if (dataSource == null) {   synchronized (ConnectionManger.class) {   if (dataSource == null) {   dataSource = new ComboPooledDataSource("mysql-config");   }   }   }   return dataSource;  } |

# SQL执行

## Statement

java.sql.Statement可用于执行DDL/DML/DCL语句:

| **方法** | **描述** |
| --- | --- |
| boolean execute(String sql) | Executes the given SQL statement, which may return multiple results. |
| ResultSet executeQuery(String sql) | Executes the given SQL statement, which returns a single ResultSet object. |
| int executeUpdate(String sql) | Executes the given SQL statement, which may be an INSERT, UPDATE, or DELETE statement or an SQL statement that returns nothing, such as an SQL DDL statement. |
| int[] executeBatch() | Submits a batch of commands to the database for execution and if all commands execute successfully, returns an array of update counts. |

Java 1.7还新增了closeOnCompletion()方法,当所有依赖于当前Statement的ResultSet关闭时,该Statement自动关闭.

### executeUpdate

Statement使用executeUpdate方法执行DDL/DML(不包含select)语句:执行DDL该方法返回0; 执行DML返回受影响的记录数.

|  |
| --- |
| #DDL  public void ddlClient() throws SQLException {   try (   Connection connection = ConnectionManger.getConnectionHikari("common.properties");   Statement statement = connection.createStatement()   ) {   int res = statement.executeUpdate("CREATE TABLE t\_ddl(" +   "id INT auto\_increment PRIMARY KEY, " +   "username VARCHAR(64) NOT NULL, " +   "password VARCHAR (36) NOT NULL " +   ")");   System.out.println(res);   }  } |
| #DML  public void dmlClient() throws SQLException {   try (   Connection connection = ConnectionManger.getConnectionHikari("common.properties");   Statement statement = connection.createStatement()   ) {   int res = statement.executeUpdate("INSERT INTO " +   "t\_ddl(username, password) " +   "SELECT name, password FROM user");   System.out.println(res);   }  } |

### execute

execute方法几乎可以执行任何SQL语句,但较为繁琐(除非不清楚SQL语句类型,否则不要使用execute方法).该方法返回值为boolean,代表执行该SQL语句是否返回ResultSet,然后Statement提供了如下方法来获取SQL执行的结果:

| **方法** | **描述** |
| --- | --- |
| ResultSet getResultSet() | Retrieves the current result as a ResultSet object. |
| int getUpdateCount() | Retrieves the current result as an update count; if the result is a ResultSet object or there are no more results, -1 is returned. |

|  |
| --- |
| public class SQLUtil {  // ...  public static void executeSQL(Statement statement, String sql) {   try {   // 如果含有ResultSet   if (statement.execute(sql)) {   ResultSet rs = statement.getResultSet();   ResultSetMetaData meta = rs.getMetaData();   int columnCount = meta.getColumnCount();   for (int i = 1; i <= columnCount; ++i) {   System.out.printf("%s\t", meta.getColumnName(i));   }   System.out.println();  while (rs.next()) {   for (int i = 1; i <= columnCount; ++i) {   System.out.printf("%s\t", rs.getObject(i));   }   System.out.println();   }   } else {   System.out.printf("该SQL语句共影响%d条记录%n", statement.getUpdateCount());   }   } catch (SQLException e) {   throw new RuntimeException(e);   }   }  } |
| @Test  public void executeClient() throws SQLException {   try(   Connection connection = SQLUtil.getConnection("common.properties");   Statement statement = connection.createStatement()   ){   SQLUtil.executeSQL(statement, "UPDATE t\_ddl SET username = 'feiqing'");   SQLUtil.executeSQL(statement, "SELECT \* FROM t\_ddl");   }  } |

## **PreparedStatement**

PreparedStatement是Statement的子接口, 它可以预编译SQL语句,编译后的SQL模板被存储在PreparedStatement对象中,每次使用时首先为SQL模板设值,然后执行该语句(因此使用PreparedStatement效率更高).创建PreparedStatement需要使用Connection的prepareStatement(String sql)方法,该方法需要传入SQL模板,可以包含占位符参数:

PreparedStatement statement = connection.prepareStatement("INSERT INTO t\_ddl(username, password) VALUES (?, ?)")

PreparedStatement也提供了excute等方法来执行SQL语句, 只是这些方法无须传入参数, 因为SQL语句已经存储在PreparedStatement对象中.

由于执行SQL前需要为SQL模板传入参数值,PreparedStatement提供了一系列的setXxx(int parameterIndex, X x)方法;另外,如果不清楚SQL模板各参数的类型,可以使用setObject(int parameterIndex, Object x)方法传入参数, 由PreparedStatement来负责类型转换.

|  |
| --- |
| @Test  public void comparisonPrepared() throws SQLException {   Connection connection = null;   try {   connection = SQLUtil.getConnection("common.properties");   long start = System.currentTimeMillis();   try (Statement statement = connection.createStatement()) {   for (int i = 0; i < 1000; ++i) {   statement.executeUpdate("INSERT INTO t\_ddl(username, password) VALUES ('name" + i + "','password" + i + "')");   }   }   long mid = System.currentTimeMillis();  try (PreparedStatement statement = connection.prepareStatement("INSERT INTO t\_ddl(username, password) VALUES (?, ?)")) {   for (int i = 0; i < 1000; ++i) {   statement.setString(1, "name" + i);   statement.setObject(2, "password" + i);   statement.execute();   }   }   long end = System.currentTimeMillis();  System.out.printf("Statement: %d%n", mid - start);   System.out.printf("Prepared: %d%n", end - mid);   } finally {   try {   assert connection != null;   connection.close();   } catch (SQLException e) {   }   }  }  注意: SQL语句的占位符参数只能代替普通值, 不能代替表名/列名等数据库对象, 更不能代替INSERT/SELECT等关键字.使用PreparedStatement还有另外一个优点:使用PreparedStatement无须拼接SQL字符串,因此可以防止SQL注入(关于SQL注入的问题可参考SQL Injection, 现代的ORM框架都解决了该问题).  注:  1. 默认使用PreparedStatement是没有开启预编译功能的,需要在URL中给出useServerPrepStmts=true参数来开启此功能;  2. 当使用不同的PreparedStatement对象来执行相同SQL语句时,还是会出现编译两次的现象,这是因为驱动没有缓存编译后的函数key,导致二次编译.如果希望缓存编译后的函数key,那么就要设置cachePrepStmts=true参数.  3. 另外, 还可以设置预编译缓存的大小:cachePrepStmts=true&prepStmtCacheSize=50&prepStmtCacheSqlLimit=300`  jdbc:mysql://host:port/database?useServerPrepStmts=true&cachePrepStmts=true&prepStmtCacheSize=50&prepStmtCacheSqlLimit=300 |

## CallableStatement

调用存储过程需要使用CallableStatement,可以通过Connection的prepareCall()方法来创建,创建时需要传入调用存储过程的SQL语句,形式为:

{CALL procedure\_name(?, ?, ?)}

存储过程的参数既有入参,也有回参; 入参可通过setXxx(int parameterIndex/String parameterName, X x)方法传入;回参可以通过调用registerOutParameter(int parameterIndex, int sqlType)来注册, 经过上面步骤, 就可以调用execute()方法来调用该存储过程, 执行结束, 则通过getXxx(int parameterIndex/String parameterName)方法来获取指定回参的值:

|  |
| --- |
| @Test  public void callableClient() throws SQLException {   try (   Connection connection = SQLUtil.getConnection("common.properties");   CallableStatement statement = connection.prepareCall("{CALL add\_pro(?, ?, ?)}")   ) {   // statement.setInt("a", 1);   statement.setInt(1, 11);   // statement.setInt("b", 2);   statement.setInt(2, 22);  // 注册CallableStatement回参   statement.registerOutParameter(3, Types.INTEGER);   // 执行存储过程   statement.execute();   // statement.getInt(3);   System.out.printf("存储过程执行结果为: %d%n", statement.getInt("sum"));   }  } |

# 操作结果集

JDBC使用ResultSet封装查询结果,然后通过ResultSet的记录指针来读取/更新记录.并提供了ResultSetMetaDate来获得ResultSet对象的元数据信息.

## ResultSet

java.sql.ResultSet是结果集对象,可以通过列索引/列名来读/写数据, 它提供了如下常用方法来移动记录指针:

| **方法** | **描述** |
| --- | --- |
| boolean next() | Moves the cursor froward one row from its current position. |
| boolean previous() | Moves the cursor to the previous row in this ResultSet object. |
| boolean first() | Moves the cursor to the first row in this ResultSet object. |
| boolean last() | Moves the cursor to the last row in this ResultSet object. |
| void beforeFirst() | Moves the cursor to the front of this ResultSet object, just before the first row. |
| void afterLast() | Moves the cursor to the end of this ResultSet object, just after the last row. |
| boolean absolute(int row) | Moves the cursor to the given row number in this ResultSet object. |
| boolean relative(int rows) | Moves the cursor a relative number of rows, either positive or negative. |

## ResultSetMetaData

ResultSet提供了getMetaData()方法来获取ResultSetMetaData以分析关于ResultSet的描述信息(前面我们已经使用ResultSetMetaData来获取结果集的列数以及列名):

| **ResultSetMetaData方法** | **描述** |
| --- | --- |
| int getColumnCount() | Returns the number of columns in this ResultSet object. |
| String getColumnLabel(int column) | Gets the designated column’s suggested title for use in printouts and displays. |
| String getColumnName(int column) | Get the designated column’s name. |
| int getColumnType(int column) | Retrieves the designated column’s SQL type. |
| String getColumnTypeName(int column) | Retrieves the designated column’s database-specific type name. |
| boolean isAutoIncrement(int column) | Indicates whether the designated column is automatically numbered. |

# 事务

事务是由一步/几步数据库操作序列组成的逻辑执行单元, 这些操作要么全部执行, 要么全部不执行. MySQL事务功能需要有InnoDB存储引擎的支持, 详见MySQL存储引擎InnoDB与Myisam的主要区别.事务的

ACID特性：

* 原子性(A: Atomicity): 事务是不可再分的最小逻辑执行体;
* 一致性(C: Consistency): 事务执行的结果, 必须使数据库从一个一致性状态, 变为另一个一致性状态.
* 隔离性(I: Isolation): 各个事务的执行互不干扰, 任意一个事务的内部操作对其他并发事务都是隔离的(并发执行的事务之间不能看到对方的中间状态,不能互相影响)
* 持续性(D: Durability): 持续性也称持久性(Persistence), 指事务一旦提交, 对数据所做的任何改变都要记录到永久存储器(通常指物理数据库).

Commit/Rollback，当事务所包含的全部操作都成功执行后提交事务,使操作永久生效,事务提交有两种方式:

* 显式提交: 使用commit;
* 自动提交: 执行DDL/DCL语句或程序正常退出;

当事务所包含的任意一个操作执行失败后应该回滚事务, 使该事务中所做的修改全部失效, 事务回滚也有两种方式:

* 显式回滚: 使用rollback;
* 自动回滚: 系统错误或强行退出.

注意: 同一事务中所有的操作,都必须使用同一个Connection.

|  |
| --- |
| public void byTransaction() throws SQLException {  boolean autoCommitFlag = connection.getAutoCommit();   // 关闭自动提交, 开启事务   connection.setAutoCommit(false);  try (   PreparedStatement minusSM = connection.prepareStatement("UPDATE `account` SET `money`=(`money` - ?) WHERE `name`=?");   PreparedStatement addSM = connection.prepareStatement("UPDATE `account` SET `money`=(`money` + ?) WHERE `name`=?")   ) {   // 从feiqing账户转出   minusSM.setBigDecimal(1, new BigDecimal(100));   minusSM.setString(2, "feiqing");   minusSM.execute();  // 中途抛出异常: rollback   if (true) {   throw new RuntimeException("no-transaction");   }  // 转入xiaofang账户   addSM.setBigDecimal(1, new BigDecimal(100));   addSM.setString(2, "xiaofang");   addSM.execute();   connection.commit();   } catch (Throwable e) {   connection.rollback();   throw new RuntimeException(e);   } finally {   connection.setAutoCommit(autoCommitFlag);   }  } |

# 批处理

多条SQL语句被当做同一批操作同时执行.调用Statement对象的addBatch(String sql)方法将多条SQL语句收集起来, 然后调executeBatch()同时执行.为了让批量操作可以正确进行, 必须把批处理视为单个事务, 如果在执行过程中失败, 则让事务回滚到批处理开始前的状态.

|  |
| --- |
| public class SQLClient {  private Connection connection = null;   private Random random = new Random();  @Before   public void setUp() {   connection = ConnectionManger.getConnectionHikari("common.properties");   }  @Test   public void updateBatch() throws SQLException {   List<String> sqlList = Lists.newArrayListWithCapacity(10);   for (int i = 0; i < 10; ++i) {   sqlList.add("INSERT INTO user(name, password) VALUES('student" + i + "','" + encodeByMd5(random.nextInt() + "") + "')");   }   int[] results = update(connection, sqlList);   for (int result : results) {   System.out.printf("%d ", result);   }   }  private int[] update(Connection connection, List<String> sqlList) {  boolean autoCommitFlag = false;   try {   autoCommitFlag = connection.getAutoCommit();  // 关闭自动提交, 打开事务   connection.setAutoCommit(false);  // 收集SQL语句   Statement statement = connection.createStatement();   for (String sql : sqlList) {   statement.addBatch(sql);   }  // 批量执行 & 提交事务   int[] result = statement.executeBatch();   connection.commit();  return result;   } catch (SQLException e) {   try {   connection.rollback();   } catch (SQLException ignored) {   }   throw new RuntimeException(e);   } finally {   try {   connection.setAutoCommit(autoCommitFlag);   } catch (SQLException ignored) {   }   }   }  private String encodeByMd5(String input) {   try {   MessageDigest md5 = MessageDigest.getInstance("MD5");   BASE64Encoder base64Encoder = new BASE64Encoder();   return base64Encoder.encode(md5.digest(input.getBytes("utf-8")));   } catch (NoSuchAlgorithmException | UnsupportedEncodingException e) {   throw new RuntimeException(e);   }   }  @After   public void tearDown() {   try {   connection.close();   } catch (SQLException ignored) {   }   }  }  注:对于批处理,也可以使用PreparedStatement,建议使用Statement,因为PreparedStatement的预编译空间有限,当数据量过大时,可能会引起内存溢出.  MySQL默认也没有打开批处理功能,需要在URL中设置rewriteBatchedStatements=true参数打开. |

目标：添加（批量），更新，删除，查询（单表，多表，条件不定），事务的代码实现。