# 1. JDBC进阶操作

## 1.1 用CallableStatement执行存储过程

要执行存储过程，就先用Connection对象获得CallableStatement，然后执行存储过程，结果依然在结果集ResultSet中，用案例讲解执行存储过程。

例1：带有输入参数（或者无参数）的存储过程的执行，该存储过程中有对表的查询。

比如之前的例子：

|  |
| --- |
| **delimiter** $; **create procedure** *proTest*(**in** empId **int**) *-- 参数格式：in/out/inout 参数名 参数类型* **begin  select** *\** **from** employee **where** eid = empId; **END** $;  *-- 调用* **call** *proTest*(1); |

程序代码：

|  |
| --- |
| **public static void** main(String[] args) {  Connection conn = **null**;  CallableStatement stmt = **null**;  ResultSet rs = **null**;  **try** {  conn = JdbcUtil.*getConnection*();  *// 使用prepareCall()获得CallableStatement对象，该对象也是预编译的SQL。PreparedStatement继承了Statement，CallableStatement继承了PreparedStatement。* stmt = conn.prepareCall(**"call proTest(?)"**);  stmt.setInt(1, 1); *// 设置参数  // 同样调用executeQuery()获得结果集* rs = stmt.executeQuery();  **while**(rs.next()) {  *// 输出查到的员工编号* System.***out***.println(rs.getInt(1));  *// 输出查到的员工姓名* System.***out***.println(rs.getString(2));  }  } **catch** (Exception e) {  **throw new** RuntimeException(e);  } **finally** {  JdbcUtil.*release*(conn, stmt, rs); *// 关闭连接* } } |

可以看出，对于有输入参数或者无参数的存储过程，使用方法也简单，和之前的步骤一样，只是用ResultSet获得结果集，如果有输入参数，直接用占位符先代替，之后设置即可。

当有输出参数的时候，就需要先用CallableStatement注册输出参数，用executeQuery()或executeUpdate()执行完存储过程后，再用CallableStatement的getXxx()方法获得输出结果。CallableStatement的getXxx()方法专门用于获得输出参数结果的。

例子如下：

使用的存储过程：

|  |
| --- |
| **delimiter** $; **create procedure** *testIf*(**in** num **int**, **out** str **varchar**(20)) **begin  if** num = 1 **then  set** str = **'星期一'**;  **elseif** num = 2 **then  set** str = **'星期二'**;  **elseif** num = 3 **then  set** str = **'星期三'**;  **else  set** str = **'输入错误'**;  **end if**; **end** $; -- 调用示例 **call** *testIf*(4, *@str*); |

程序代码：

|  |
| --- |
| **public static void** main(String[] args) {  Connection conn = **null**;  CallableStatement stmt = **null**;  **try** {  conn = JdbcUtil.*getConnection*();  stmt = conn.prepareCall(**"call testIf(?, ?)"**);  stmt.setInt(1, 2); *// 设置参数1  // 参数2是输出参数，调用registerOutParameter注册输出参数  // 方法参数指定了哪个参数是输出参数，并且是什么类型* stmt.registerOutParameter(2, Types.***VARCHAR***);  *// Types类是java.sql提供的类，其中的静态变量描述了数据库中的数据类型  // 然后，调用executeQuery()或者executeUpdate()执行* stmt.executeUpdate();  *// 最后还是通过stmt得到输出参数的结果，index要和上面注册的索引匹配* String result = stmt.getString(2);  System.***out***.println(result);  } **catch** (Exception e) {  **throw new** RuntimeException(e);  } **finally** {  JdbcUtil.*release*(conn, stmt, **null**); *// 关闭连接* } } |

## 1.2 批处理

有时，程序需要一次性执行多条SQL语句（例如批量增加、删除数据等），如果按照以前方法，每执行一次SQL就使用一系列的“获得连接、执行SQL、释放资源”操作的话，将导致程序与数据库交互次数频繁、开销大、效率降低。

这时我们可以采用批处理的方法提高效率。批处理就可以实现在一次连接过程中执行多条SQL语句，以此提高数据库吞吐量。。

例1：在一次连接中执行三条SQL，分别是增加两条学生记录，再删除第一条学生记录。

|  |
| --- |
| **public static void** main(String[] args) {  Connection conn = **null**;  Statement stmt = **null**;  **try** {  conn = JdbcUtil.*getConnection*();  stmt = conn.createStatement();  String sql1 = **"INSERT INTO student(id, name) VALUES(1, '张三')"**;  String sql2 = **"INSERT INTO student(id, name) VALUES(2, '李四')"**;  String sql3 = **"DELETE FROM student where id=1"**;  *// 使用addBatch方法，将每条SQL加入到批处理中。实际上加入批处理就是把要执行的SQL添加进了Statement实现类维护的一个List对象中。该List对象起到了缓存SQL的作用。这点可通过查看MySQL JDBC源码验证。* stmt.addBatch(sql1);  stmt.addBatch(sql2);  stmt.addBatch(sql3);  *// 调用executeBatch()就会执行批处理，即一次性执行完添加进去的所有SQL，这些是MySQL的JDBC实现类按照规范已经实现好的，我们无需关注具体实现，会调用即可。* **int**[] rows = stmt.executeBatch(); *// 方法返回一个int数组，表示每条SQL执行后影响的行数* **for** (**int** i = 0; i < rows.**length**; i++) {  System.***out***.println(**"第"** + (i + 1) + **"条语句执行影响了"** + rows[i] + **"行"**);  }  } **catch** (Exception e) {  **throw new** RuntimeException(e);  } **finally** {  JdbcUtil.*release*(conn, stmt, **null**); *// 关闭连接* } } |

运行完上述程序，student表中只有一条“李四”的数据，因为其中第一条添加的记录随即也被删除了。

上面的例子中执行了不同的SQL语句，我们使用了Statement对象。如果我们是执行相同的SQL，可以使用PreparedStatement，例如要插入10条学生记录，预编译SQL是相同的，只是每条参数值不同。例如可这样使用：

|  |
| --- |
| **public static void** main(String[] args) {  Connection conn = **null**;  PreparedStatement stmt = **null**;  **try** {  conn = JdbcUtil.*getConnection*();  stmt = conn.prepareStatement(**"INSERT INTO student(id, name) VALUES(?, ?)"**);  *// 循环十次，由于只有参数不同，只要设置不同的参数，再加入批处理即可。* **for** (**int** i = 0; i < 10; i++) {  stmt.setInt(1, i);  stmt.setString(2, **"学生"** + i);  stmt.addBatch();  }  *// 最后执行* **int**[] rows = stmt.executeBatch();  **for** (**int** i = 0; i < rows.**length**; i++) {  System.***out***.println(**"第"**+ (i + 1) + **"条SQL影响了"** + rows[i] + **"行"**);  }  } **catch** (Exception e) {  **throw new** RuntimeException(e);  } **finally** {  JdbcUtil.*release*(conn, stmt, **null**); *// 关闭连接* } } |

如果要进行大批量数据的操作，例如添加10000个学生记录（甚至更多），这时就不能单纯地像上面一样直接把这么多数据缓存到批处理缓存中，而应该分批操作，例如每满足1000条学生才执行一次批处理。

因为大数据量不可能一次性处理完毕，否则导致内存溢出，因此采取这样的策略。其实，大数据量的处理基本都会采用这样的方法：每次批量地处理一定数量的数据。这实际和IO中使用的缓冲区是一样的道理，避免了一次只处理一丁点或者极大数量的数据。

程序代码如下：

|  |
| --- |
| **public static void** main(String[] args) {  Connection conn = **null**;  PreparedStatement stmt = **null**;  **try** {  conn = JdbcUtil.*getConnection*();  stmt = conn.prepareStatement(**"INSERT INTO student(id, name) VALUES(?, ?)"**);  *// 循环10000次。* **for** (**int** i = 1; i <= 10000; i++) {  stmt.setInt(1, i);  stmt.setString(2, **"学生"** + i);  stmt.addBatch();  *// 每达到1000条就执行一次批处理* **if** ( i % 1000 == 0) {  stmt.executeBatch();  *// 执行完后一定要清空缓存数据，否则数据一直累加* stmt.clearBatch();  }  }  stmt.executeBatch(); *// 最后再执行executeBatch()，这是因为最后可能还有剩余的不满1000条的记录。比如如果要插入10021条记录，最后取余后，还会有21条记录要存储，这是确保全部插入。* stmt.clearBatch();  System.***out***.println(**"执行结束"**);  } **catch** (Exception e) {  **throw new** RuntimeException(e);  } **finally** {  JdbcUtil.*release*(conn, stmt, **null**); *// 关闭连接* } } |

注意的是每次批处理不要忘了清空缓存。

## 1.3 获取自增长值

比如部门表和员工表，部门表的主键是自增长的。有一种情况：在程序中添加了一个部门信息后，需要马上得到该部门自增长的主键，然后给该部门添加员工。

那么此时获得自增长键就十分有必要。现在就用程序获得部门表自增长的键值。

（1）部门表SQL：

|  |
| --- |
| **create table** dept(  **did int primary key auto\_increment**,  **dname varchar**(20) ); |

（2）程序代码：

|  |
| --- |
| **public static void** main(String[] args) {  Connection conn = **null**;  PreparedStatement stmt = **null**;  ResultSet rs = **null**;  **try** {  conn = JdbcUtil.*getConnection*();  String sql = **"INSERT INTO dept(dname) VALUES(?)"**;  *// 这里调用prepareStatement方法时，用参数2表示需要获得执行此SQL完毕的自增长键的值* stmt = conn.prepareStatement(sql, Statement.***RETURN\_GENERATED\_KEYS***);  *// 设置ps参数* stmt.setString(1, **"系统研发部"**);  *// 先执行完ps* stmt.executeUpdate();  *// 然后用ps的getGeneratedKeys()方法获得自增长的结果集* rs = stmt.getGeneratedKeys();  **int** keys = 0;  **if** (rs.next()) {  keys = rs.getInt(1);  }  System.***out***.println(**"获得的自增长键是："** + keys);  } **catch** (Exception e) {  **throw new** RuntimeException(e);  } **finally** {  JdbcUtil.*release*(conn, stmt, rs); *// 关闭连接* } } |

## 1.4 LOB的存取

LOB是Large Object的意思，即是用于存储大数据的对象。LOB分为CLOB和BLOB，CLOB用于存储大的文本数据（Character Large Object）；BLOB用于存储大的二进制数据（Binary Large Object），例如文件、图片、视频等。

正如之前所讲，在MySQL中，本身就有BLOB数据类型，而MySQL中的TEXT数据类型就对应着CLOB数据类型。

MySQL中，分别有四种TEXT数据类型和四种BLOB数据类型，它们之间的区别就是最大存储的数据长度不同，这个可参阅前面章节的知识。MySQL支持的LOB类型都是可变长度类型，即以数据的实际长度进行存储。

下面我们了解如何通过JDBC进行MySQL的LOB类型存取操作。

（1）插入大文本数据

首先创建一张用于测试的表：

|  |
| --- |
| *-- 新建测试表* **CREATE TABLE** t1 (  **id INT PRIMARY KEY**,  **content LONGTEXT** ); |

并在src目录下新建一个a.txt文件（要求是UTF8编码的），里面书写内容，下面的程序将文本内容保存到数据库中。代码如下：

|  |
| --- |
| **public static void** main(String[] args) {  Connection conn = **null**;  PreparedStatement stmt = **null**;  **try** {  conn = JdbcUtil.*getConnection*();  stmt = conn.prepareStatement(**"INSERT INTO t1(id, content) VALUES(?,?)"**);  *// 正常设置参数* stmt.setInt(1, 1);  *// 设置LONGTEXT的内容时，要调用setCharacterStream方法。其中需要reader对象，因此这里读取文件为Reader流。* Reader reader = **new** FileReader(**"src/a.txt"**);  stmt.setCharacterStream(2, reader); // 设置内容  stmt.executeUpdate();  } **catch** (Exception e) {  **throw new** RuntimeException(e);  } **finally** {  JdbcUtil.*release*(conn, stmt, **null**);  } } |

（2）读取大文本数据。

这里读取刚刚保存的数据。代码如下：

|  |
| --- |
| **public static void** main(String[] args) {  Connection conn = **null**;  PreparedStatement stmt = **null**;  ResultSet rs = **null**;  **try** {  conn = JdbcUtil.*getConnection*();  stmt = conn.prepareStatement(**"SELECT** *\** **FROM t1 WHERE id=?"**);  stmt.setInt(1, 1);  rs = stmt.executeQuery();  **if** (rs.next()) {  *// 下面就是流的读取和写入处理* Writer writer = **new** FileWriter(**"d:/b.txt"**); *// 写到d盘的b.txt中* Reader reader = rs.getCharacterStream(2); *// 得到对应的Reader流* **char**[] buff = **new char**[1024];  **int** length = 0;  **while** ( (length = reader.read(buff)) != -1 ) {  writer.write(buff, 0, length);  }  reader.close();  writer.close();  }  } **catch** (Exception e) {  **throw new** RuntimeException(e);  } **finally** {  JdbcUtil.*release*(conn, stmt, rs);  } } |

（3）插入二进制数据。首先创建测试表如下：

|  |
| --- |
| **CREATE TABLE** t2 (  **id INT PRIMARY KEY**,  **content LONGBLOB** ); |

在src下准备一张test.jpg文件，下述程序将图片数据保存到t2表中：

|  |
| --- |
| **public static void** main(String[] args) {  Connection conn = **null**;  PreparedStatement stmt = **null**;  **try** {  conn = JdbcUtil.*getConnection*();  stmt = conn.prepareStatement(**"INSERT INTO t2(id, content) VALUES(?,?)"**);  stmt.setInt(1, 1);  *// 这时调用的是setBinaryStream方法，需要InputStream对象* InputStream in = **new** FileInputStream(**"src/test.jpg"**);  stmt.setBinaryStream(2, in);  stmt.executeUpdate();  } **catch** (Exception e) {  **throw new** RuntimeException(e);  } **finally** {  JdbcUtil.*release*(conn, stmt, **null**);  } } |

这样就把图片数据保存在了BLOB中了。

（4）读取二进制数据。把刚才保存的数据读取并生成一个jpg文件。代码如下：

|  |
| --- |
| **public static void** main(String[] args) {  Connection conn = **null**;  PreparedStatement stmt = **null**;  ResultSet rs = **null**;  **try** {  conn = JdbcUtil.*getConnection*();  stmt = conn.prepareStatement(**"SELECT content FROM t2 WHERE id = ?"**);  stmt.setInt(1, 1);  rs = stmt.executeQuery();  **if** (rs.next()) {  InputStream in = rs.getBinaryStream(**"content"**);  *// 下面就是操作流了。* OutputStream out = **new** FileOutputStream(**"d:/test.jpg"**);  **byte**[] buff = **new byte**[1024];  **int** length = 0;  **while** ( (length = in.read(buff)) != -1 ) {  out.write(buff, 0, length);  }  in.close();  out.close();  }  } **catch** (Exception e) {  **throw new** RuntimeException(e);  } **finally** {  JdbcUtil.*release*(conn, stmt, rs);  } } |

但是在实际应用中，一般也不会把文件保存到数据库中，而是直接把文件保存在服务器的硬盘上，数据库中的字段保存文件名，使得服务器能找到该文件即可。

## 1.5 元数据Metadata

有时想获得所连接数据库的一些信息，比如数据库版本、名称等信息，或者获得结果集的列信息。想要获得这些信息，就需要用到“元数据”。

JDBC提供了三种元数据对象，分别是数据库元数据、参数元数据和结果集元数据。见案例。

案例1：数据库元数据。通过Connection对象的getMetaData方法即可得到DatabaseMetaData，可从中获取数据库有关信息。

代码：

|  |
| --- |
| **package** com.zhang.test;  **import** java.sql.Connection; **import** java.sql.DatabaseMetaData; **import** java.sql.DriverManager; **import** java.sql.SQLException;  **public class** Demo {  **public static void** main(String[] args) **throws** ClassNotFoundException, SQLException {  String url = **"jdbc:mysql://localhost:3306/mytest?useSSL=false&useUnicode=true&characterEncoding=utf8"**;  Class.*forName*(**"com.mysql.jdbc.Driver"**);  Connection conn = DriverManager.*getConnection*(url, **"root"**, **"123456"**);  *// 得到数据库元对象* DatabaseMetaData metaData = conn.getMetaData();  System.***out***.println(metaData.getUserName()); *//获得用户名* System.***out***.println(metaData.getURL()); *//获得连接字符串url* System.***out***.println(metaData.getDatabaseProductName()); *// 数据库名称* System.***out***.println(metaData.getDatabaseProductVersion()); *// 数据库版本号* System.***out***.println(metaData.getDriverName()); *// 驱动名称* System.***out***.println(metaData.getDriverVersion()); *// 驱动版本* System.***out***.println(metaData.isReadOnly()); *// 是否只读的* conn.close();  } } |

案例2：参数元数据，可调用PreparedStatement对象的getParameterMetaData()方法得到元数据对象。可获得参数个数。

|  |
| --- |
| **package** com.zhang.test;  **import** java.sql.\*;  **public class** Demo {  **public static void** main(String[] args) **throws** ClassNotFoundException, SQLException {  String url = **"jdbc:mysql://localhost:3306/mytest?useSSL=false&useUnicode=true&characterEncoding=utf8"**;  Class.*forName*(**"com.mysql.jdbc.Driver"**);  Connection conn = DriverManager.*getConnection*(url, **"root"**, **"123456"**);  PreparedStatement ps = conn.prepareStatement(**"insert into account values(?,?)"**);  ps.setString(1, **"zhang"**);  ps.setInt(2, 1);  *// 得到参数元数据对象* ParameterMetaData pmd = ps.getParameterMetaData();  System.***out***.println(pmd.getParameterCount()); *// 获得参数个数* ps.close();  conn.close();  } } |

案例3：结果集元数据。通过ResultSet的getMetaData()方法得到，可获得当前列的个数和名称等信息。

|  |
| --- |
| **package** com.zhang.test;  **import** java.sql.\*;  **public class** Demo {  **public static void** main(String[] args) **throws** ClassNotFoundException, SQLException {  String url = **"jdbc:mysql://localhost:3306/mytest?useSSL=false&useUnicode=true&characterEncoding=utf8"**;  Class.*forName*(**"com.mysql.jdbc.Driver"**);  Connection conn = DriverManager.*getConnection*(url, **"root"**, **"123456"**);  PreparedStatement ps = conn.prepareStatement(**"select** *\** **from account"**);  ResultSet rs = ps.executeQuery();  ResultSetMetaData rsmd = rs.getMetaData();  *// getColumnCount()获得有多少列。由于索引从1开始，因此遍历就从1开始  // 这里遍历列，在循环中获得每列信息* **for**(**int** i = 1; i <= rsmd.getColumnCount(); i++) {  System.***out***.println(rsmd.getColumnLabel(i)); *// 列名* System.***out***.println(rsmd.getColumnClassName(i)); *// 列在Java中的类型* System.***out***.println(rsmd.getColumnTypeName(i)); *// 列在数据库中的类型* }  rs.close();  ps.close();  conn.close();  } } |

# 2. 事务（Transaction）

## 2.1 数据库中的事务

事务是指逻辑上的一组操作，组成这组操作的各个单元，要么全部成功，要么全部失败。事务的最简单的例子就是银行转账。例如A给B转账，那么A的存款减少和B的存款增加这两件事必须是原子性的，即要么都成功，转账就成功，要么都失败，转账就失败，失败那么A和B的账户还是保持原来的状态。不允许出现只有一方成功或者失败。

也就是说，组成事务的各个单元必须全部成功，否则整个操作就应该回滚（rollback）到之前的状态。因此，事务保证了数据的准确性，事务在数据库中非常重要。

MySQL中，事务是自动开启和提交的，即每条语句处于独立的事务中且自动提交事务。例如我们之前执行的所有SQL都是自动提交的，即我们让数据库执行一条SQL时，数据库自动在执行该语句前开启事务，执行完毕后提交事务，这样执行完后我们就能马上看见该SQL对数据库所作的更改。

我们也可通过手动提交事务，让一组语句处于同一个事务中。例如我们让上面的转账操作处于同一事务中执行，那么就需要进行这样的操作：

（1）开启事务。在MySQL中执行语句：

|  |
| --- |
| **START TRANSACTION**; |

该语句就表示开启一个事务，这表示以后执行的语句都处于同一个事务中（除非遇到commit或rollback语句）。

（2）执行本事务中的一系列SQL。例如转账操作（自己预先建立账户表）：

|  |
| --- |
| *-- A 向 B转账100元* **UPDATE account SET** money = money - 100 **WHERE name**=**'A'**; **UPDATE account SET** money = money + 100 **WHERE name**=**'B'**; |

（3）执行commit或者rollback语句。

如果执行commit，表示提交该事务，即执行此事务，若事务执行成功，表示其中SQL均执行成功，转账将成功；若事务执行中有误，则事务中的语句均不会执行成功，数据库还是开启事务之前的状态（即失败自动回滚）。

如果执行rollback，则直接回滚这个事务，即放弃本事务中所有语句所作的更改，回到开启事务之前的状态。

其实，start transaction就是定义了一个标记，用于提交或回退。执行commit或者rollback都将使本事务结束，本事务产生的最终结果将会作用到数据库上。

下面介绍一下事务的特性，常作为面试用：

（1）原子性：指处于同一个事务中的多条语句是不可分割的。

（2）一致性：事务必须使数据库的一个一致性状态变换到另外一个一致性状态。比如，转账前两个账户余额之和为2k，转账之后也应该是2K。

（3）隔离性：指多线程环境下，一个线程中的事务不能被其他线程中的事务打扰。（对于隔离性，下面的2.3节专门对隔离性进行讲解）

（4）持久性：事务一旦提交，就应该被永久保存起来。

## 2.2 JDBC中控制事务

以前使用的JDBC也是自动提交事务的。如果想手动控制事务，则要先调用Connection对象的setAutoCommit()方法，设置自动提交为false，实现手动控制事物提交。其中使用commit()方法提交事务，使用rollback()方法回滚事务。

下面是手动提交的例子：

|  |
| --- |
| **public static void** main(String[] args) {  Connection conn = **null**;  PreparedStatement stmt = **null**;  **try** {  conn = JdbcUtil.*getConnection*();  conn.setAutoCommit(**false**); *// 一定要在执行之前设置手动提交* stmt = conn.prepareStatement(**"UPDATE account SET money = money - 100 WHERE name = 'A'"**);  stmt.executeUpdate(); *// 执行* stmt = conn.prepareStatement(**"UPDATE account SET money = money + 100 WHERE name = 'B'"**);  stmt.executeUpdate(); *// 执行  // 最后要提交。否则无效果。* conn.commit();  } **catch** (Exception e) {  **if** (conn != **null**) {  **try** {  conn.rollback(); *// 遇到异常，回滚* } **catch** (SQLException e1) {  e1.printStackTrace();  }  }  e.printStackTrace();  } **finally** {  JdbcUtil.*release*(conn, stmt);  } } |

上述执行成功后，数据库中发生了一次“成功的转账”。其实transaction也是“交易”的意思，简称是“TX”，T是首字母，X是transaction的“近似”发音。

为了验证能成功回滚，我们故意在执行完第一个SQL后执行一句“int x = 1/0”的错误代码，使下一句SQL无法执行，以此验证能够回滚到开启事务之前的状态。其实，rollback可以不用调用，因为出现异常会自动回滚，则代码如下：

|  |
| --- |
| **public static void** main(String[] args) {  Connection conn = **null**;  PreparedStatement stmt = **null**;  **try** {  conn = JdbcUtil.*getConnection*();  conn.setAutoCommit(**false**); *// 一定要在执行之前设置手动提交* stmt = conn.prepareStatement(**"UPDATE account SET money = money - 100 WHERE name = 'A'"**);  stmt.executeUpdate(); *// 执行* **int** x = 1 / 0; *// 故意出错* stmt = conn.prepareStatement(**"UPDATE account SET money = money + 100 WHERE name = 'B'"**);  stmt.executeUpdate(); *// 执行  // 最后要提交。否则无效果。* conn.commit();  } **catch** (Exception e) {  *// 省略了rollback* e.printStackTrace();  } **finally** {  JdbcUtil.*release*(conn, stmt);  } } |

有时，只想回滚事务的一部分，那么可以使用保存点（回滚点）。例如两次转账，如果已经保证第一次转账是正确的，如果出现了错误，就回滚到第一次转账之后。使用setSavePoint()设置一个保存点，然后使用rollback(sp)回滚到该保存点。例如下述代码：

|  |
| --- |
| **public static void** main(String[] args) {  Connection conn = **null**;  PreparedStatement stmt = **null**;  Savepoint sp = **null**;  **try** {  conn = JdbcUtil.*getConnection*();  conn.setAutoCommit(**false**); *// 一定要在执行之前设置手动提交  // 第一次转账* stmt = conn.prepareStatement(**"UPDATE account SET money = money - 100 WHERE name = 'A'"**);  stmt.executeUpdate(); *// 执行* stmt = conn.prepareStatement(**"UPDATE account SET money = money + 100 WHERE name = 'B'"**);  stmt.executeUpdate(); *// 执行  // 第一次转账结束设置保存点* sp = conn.setSavepoint();  *// 第二次转账* stmt = conn.prepareStatement(**"UPDATE account SET money = money - 100 WHERE name = 'A'"**);  stmt.executeUpdate(); *// 执行* **int** x = 1 / 0; *// 模拟出错* stmt = conn.prepareStatement(**"UPDATE account SET money = money + 100 WHERE name = 'B'"**);  stmt.executeUpdate(); *// 执行* } **catch** (Exception e) {  *// 这时要回滚到保存点* **if** (conn != **null**) {  **try** {  conn.rollback(sp);  } **catch** (SQLException e1) {  e1.printStackTrace();  }  }  e.printStackTrace();  } **finally** {  *// 这时将提交写在finally中，因为部分回滚也是要提交的，否则保存点之前的也不会提交进数据库。* **if** (conn != **null**) {  **try** {  conn.commit();  } **catch** (SQLException e) {  e.printStackTrace();  }  }  JdbcUtil.*release*(conn, stmt);  } } |

## 2.3 事务的隔离级别、JDBC中控制事务的隔离级别

之前我之所以说，在事务中没提交前执行的SQL不是真正地将操作结果保存到数据库中，那是因为没有提交（或回滚）前，是不知道最终的操作结果的。但实际上，事务中执行的SQL就是在数据库上操作的，并且在事务中能看到执行过的SQL产生的效果（即使没有提交）。比如之前转账的例子，事务中执行让A减少100元后，即使未提交事务，这时在本事务中用SELECT查询数据，会发现查出的数据确实是A减少100元。

这说明事务执行过程中就按照SQL操作了数据库，这样是没问题的，以后最终不管是要commit还是rollback，总之都有开始事务时的标记，能够最终正确地提交或回滚，保证事务的正确执行。

在多线程环境中，不同线程可能同时操作同一块数据，这时不同事务之间就可能产生干扰，可能导致数据的不准确和不一致，这就是事务的隔离性（Isolation）问题。如果不考虑事务的隔离性，就可能出现以下问题：

（1）脏读：指一个线程中的事务读取到了另外一个线程中未提交的数据。

（2）不可重复读：指一个线程中的事务读取到了另外一个线程中提交的update的数据。

（3）虚读（也叫幻读）：指一个线程中的事务读取到了另外一个线程中提交的insert的数据。

下面举例说明何为脏读、不可重复读和虚读。首先打开两个连接到MySQL的控制台客户端（Navicat支持这样做），模拟同时执行的两个线程，可分别叫它们T1和T2线程以便区分。下面进行隔离性问题的模拟：

Step1：准备工作。

我们演示的account表。account表的原始数据是：

|  |  |
| --- | --- |
| name | money |
| A | 1000 |
| B | 1000 |

首先在两个“线程”中都执行以下命令：

|  |
| --- |
| **SET SESSION TRANSACTION ISOLATION LEVEL READ UNCOMMITTED**; |

该语句是设置MySQL的当前会话连接的隔离级别是最低的“READ UNCOMMITTED”级别，以便演示脏读、不可重复读和虚读。学完本节就知道这个的意义了。

然后，两个“线程”均使用“START TRANSACTION;”开启事务。

Step2：查询数据。

T1和T2线程都先查询account表中的数据：

|  |
| --- |
| **SELECT** *\** **FROM account**; |

发现A和B账户都是1000元，没有问题。

Step3：演示脏读。

假设CPU先执行T1，T1中更新数据：

|  |
| --- |
| **UPDATE account SET** money = money + 100 **WHERE name** = **'A'**; *-- A减少100元* |

此时T1尚未提交，而T2抢占了CPU，T2通过SELECT语句查询数据，会发现A的账户变为1100元。这就是脏读，即线程T2读到了线程T1中未提交的数据。

Step4：演示不可重复读。

现在，在T1和T2中都执行rollback重回初始状态，重新执行Step1和Step2，跳过Step3。这时在T1中先更新数据并提交：

|  |
| --- |
| **UPDATE account SET** money = money + 100 **WHERE name** = **'A'**; **COMMIT**; |

此时在T2中查询数据，发现A的账户变为1100元。这就是不可重复读，线程T2读到了线程T1中提交的UPDATE数据。不可重复读的问题就是在一个事务范围内，两个相同的查询却返回了不同数据（例如Step2中的查询和Step4中的T2查询的数据不同，但是在一个事务内）。举例说明问题：例如，一个读者两次读取同一文档，但在两次读取之间，作者重写了该文档。当读者第二次读取文档时，文档已更改，导致原始数据读取不可重复。

Step5：演示虚读。

现在，重新执行Step1和Step2，然后T1线程插入一条数据并提交：

|  |
| --- |
| **INSERT INTO account VALUES**(**'C'**, 1000);  **COMMIT**; |

T2再查询时发现多了一条数据。这就是虚读，即线程T2读到了线程T1中提交的INSERT数据。

以上出现的隔离性问题都是发生在并发情况下执行的事务中，且两个线程都分别是同一个事务范围内的。如果T2线程中执行的都不是一个事务范围内的语句，那么查询到不同数据当然是理所当然的（除了脏读），注意区分。

数据库一共提供了四种事务隔离级别，用于解决隔离性问题，列表如下：

|  |  |  |
| --- | --- | --- |
| 名称 | 说明 | 备注 |
| READ UNCOMMITTED | 脏读、不可重复读、虚读都有可能发生 | 隔离级别最低，并发性能高，无锁 |
| READ COMMITTED | 防止脏读发生，不可重复读、虚读都有可能发生 | 锁定正在读取的行 |
| REPEATABLE READ | 防止脏读、不可重复读的发生，虚读有可能发生 | 锁定所读取的所有行 |
| SERIALIZABLE | 防止脏读、不可重复读、虚读的发生 | 锁表 |

上述的级别从上到下是越来越高的，级别越高，数据越安全，但性能越低。因为要实现高级别的隔离，就需要更严格的加锁处理，就会导致并发降低。特别是SERIALIZABLE，如果应用了这个隔离级别，并发插入时，必须等待其他线程提交完毕才可插入，性能低。

MySQL的默认事务隔离级别是Repeatable Read，每个会话连接都可设置适合自己的隔离级别。设置当前会话连接隔离级别的方法是：

|  |
| --- |
| **SET SESSION TRANSACTION ISOLATION LEVEL 隔离级别**; |

这里的隔离级别只要写上上表中对应的英文单词即可。可使用：

|  |
| --- |
| **SELECT** *@@tx\_isolation*; |

查看当前会话连接的隔离级别。读者可分别采用不同的隔离级别再按照上述的Step进行隔离性的验证。一般来说，隔离级别使用“READ COMMITTED”或者“REPEATABLE READ”即可。当然，设置事务的隔离级别一定要在开启事务前进行设置。

那么，怎么在JDBC中设置连接的隔离级别呢？Connection对象提供了setTransactionIsolation(int level)方法，该方法用于设置连接的隔离级别，可选用Connection中的常量：

|  |
| --- |
| */\*\*  \* 指示事务不受支持的常量  \*/* **int *TRANSACTION\_NONE*** = 0; */\*\*  \* 指示防止发生脏读的常量，不可重复读和虚读有可能发生  \*/* **int *TRANSACTION\_READ\_UNCOMMITTED*** = 1; */\*\*  \* 指示可以发生脏读、不可重复读和虚读的常量  \*/* **int *TRANSACTION\_READ\_COMMITTED*** = 2; */\*\*  \* 指示防止发生脏读、不可重复读的常量，虚读有可能发生  \*/* **int *TRANSACTION\_REPEATABLE\_READ*** = 4; */\*\*  \* 指示防止发生脏读、不可重复读和虚读的常量  \*/* **int *TRANSACTION\_SERIALIZABLE*** = 8; |

只要在开启事务之前设置隔离级别即可，例如：

|  |
| --- |
| **public static void** main(String[] args) {  Connection conn = **null**;  PreparedStatement stmt = **null**;  Savepoint sp = **null**;  **try** {  conn = JdbcUtil.*getConnection*();  *// 设置隔离级别* conn.setTransactionIsolation(Connection.***TRANSACTION\_READ\_COMMITTED***);  conn.setAutoCommit(**false**);  stmt = conn.prepareStatement(**"UPDATE account SET money = money - 100 WHERE name = 'A'"**);  stmt.executeUpdate();  conn.commit();  } **catch** (Exception e) {  e.printStackTrace();  } **finally** {  JdbcUtil.*release*(conn, stmt);  } } |