# 1. JDBC简介

前面学习了MySQL数据库的使用，现在我们需要在Java程序中连接数据库，在程序中执行SQL语句来操作数据库。

在Java程序中给数据库发送和执行SQL语句、获得数据库返回的数据等数据库操作，需要用到JDBC技术。JDBC（Java Data Base Connectivity）是Sun公司制定的一套规范，用于实现让Java语言用统一的、方便的方式连接不同的数据库。

Sun公司只负责设计JDBC的接口，但是JDBC的具体实现是由各个数据库软件厂商实现的。各个厂商提供了实现JDBC的jar包，用此jar包即可实现连接该数据库。开发者需要连接哪个数据库，就使用哪个厂商的jar包。而开发者只要学习使用JDBC接口即可，不需要知道每种数据库在底层是如何通过Java来连接数据库的（比如传递数据的格式、TCP连接协议的实现等）。

JDBC减轻了开发者的工作，实现了面向接口编程。而在JDBC出现之前，开发者需要自行写大量代码来实现连接不同的数据库。

## 1.1 用JDBC连接MySQL

首先新建Java项目，并且引入MySQL的JDBC实现jar包。同样，MySQL官方资源可在搜狐镜像（<http://mirrors.sohu.com> ）上下载（也可去官网<https://dev.mysql.com/downloads/connector/j/> ），目录是“mysql/Connector-J/”，我们选择当前最新的5.1.40版本，即“mysql-connector-java-5.1.40.zip”文件下载。

解压文件即可得到一个jar包：mysql-connector-java-5.1.40-bin.jar，我们需要在项目中引入这个jar包才行。

下面写代码连接MySQL数据库。

首先，必须知道以下信息才能连接到MySQL数据库：

（1）数据库服务器的IP地址和端口。对于本机，IP就是127.0.0.1，也可写localhost。默认端口是3306。

（2）数据库服务器的用户名和密码。

注意，导包时应该导入JDBC的接口，而不是导入MySQL的实现类，因为他们的名字都一样，只是包不同。JDBC接口所在的包名是java.sql或者javax.sql。这是额外需要注意的。

方式一：用JDBC接口来接收MySQL驱动的实现类对象，以此获得连接。

|  |
| --- |
| **package** com.zhang.test;  **import** java.sql.Connection; **import** java.sql.Driver; **import** java.sql.SQLException; **import** java.util.Properties;  **public class** Demo {  **public static void** main(String[] args) **throws** SQLException {  *// 数据库的URL，包含了协议、IP地址、端口、要连接的数据库名等信息* String url = **"jdbc:mysql://localhost:3306/mytest"**;  *// url的构成是：“协议名:子协议名://服务器IP:端口/数据库名”。  // 用户名和密码* String user = **"root"**;  String password = **"123456"**;   *// 创建驱动程序类对象(new 要用实现类，和接口是同名的，所以需要写类的全路径)* Driver driver = **new** com.mysql.jdbc.Driver();  *// 将用户名和密码存储在Properties对象中* Properties props = **new** Properties();  props.setProperty(**"user"**, user);  props.setProperty(**"password"**, password);  *// 使用url和props获得数据库连接  // 之所以将用户名和密码封装在props中，是因为这里可传递Properties对象，并且规定了格式。* Connection conn = driver.connect(url, props);  System.***out***.println(conn);  *// 注意关闭连接，释放资源* conn.close();  } } |

程序运行能正确获得数据库连接对象（前提是你先创建了一个mytest数据库，因为例子中用到了，也可以自定义）。

但是运行程序出现了下面的MySQL红色警告：

|  |
| --- |
| Tue Feb 14 14:14:19 CST 2017 WARN: Establishing SSL connection without server's identity verification is not recommended. According to MySQL 5.5.45+, 5.6.26+ and 5.7.6+ requirements SSL connection must be established by default if explicit option isn't set. For compliance with existing applications not using SSL the verifyServerCertificate property is set to 'false'. You need either to explicitly disable SSL by setting useSSL=false, or set useSSL=true and provide truststore for server certificate verification. |

这段话意思是，若要连接新版本MySQL数据库，必须指明是否使用SSL协议。SSL是一种对通信加密的协议，可参考：<http://www.ruanyifeng.com/blog/2014/02/ssl_tls.html> 详细了解。总之，使用SSL会使通信加密，有利于安全，但会产生SSL延迟，若不使用SSL，则通信速度快一些，通信无加密。

上述警告告诉我们必须通过连接参数指明是否使用SSL，若使用SSL，则设置参数useSSL=true，若不使用SSL，则设置参数useSSL=false。连接参数是写在url的最后，参数和前面的连接信息用问号?分开，各个参数之间用&符号分开。例如：

|  |
| --- |
| jdbc:mysql://localhost:3306/mytest?useSSL=false |

使用此url就不会出现警告信息。

再介绍两个重要的连接参数：useUnicode和characterEncoding。

useUnicode表示传递的数据是否使用unicode字符集，可设置为true或false，characterEncoding用于指定传递数据的编码，可设置为utf8等。

由于MySQL数据库默认就是UTF8编码，因此推荐设置useUnicode为true并且使用utf8编码，这样可解决中文乱码问题。即url如下：

|  |
| --- |
| jdbc:mysql://localhost:3306/mytest?useSSL=false&useUnicode=true&characterEncoding=utf8 |

方式二：使用驱动管理器类连接数据库。

|  |
| --- |
| **package** com.zhang.test;  **import** java.sql.Connection; **import** java.sql.DriverManager; **import** java.sql.SQLException;  **public class** Demo {  **public static void** main(String[] args) **throws** SQLException {  String url = **"jdbc:mysql://localhost:3306/mytest?useSSL=false&useUnicode=true&characterEncoding=utf8"**;  *// 首先用DriverManager来注册一个MySQL驱动程序  // （DriverManager也可注册多个不同的驱动程序）* DriverManager.*registerDriver*(**new** com.mysql.jdbc.Driver());  *// 直接用驱动管理器DriverManager的getConnection获得数据库连接  // getConnection中传递url、user和password。* Connection conn = DriverManager.*getConnection*(url, **"root"**, **"123456"**);  *// DriverManager会根据url信息找到已注册的、相匹配的数据库驱动程序来获得数据库连接* System.***out***.println(conn);  conn.close();  } } |

还能否更简单呢？我们查看MySQL提供的Driver实现类的源码，发现Driver实现类主要就是以下代码：

|  |
| --- |
| **package** com.mysql.jdbc;  **import** java.sql.SQLException; **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!"**);  }  } **public** Driver() **throws** SQLException {} } |

即Driver的静态代码块中，自行将自己注册到DriverManager中了。所以我们使用时，不再需要自己注册驱动，直接加载一下Driver类，即可从DriverManager中拿到连接。

改进如下：

|  |
| --- |
| **package** com.zhang.test;  **import** java.sql.Connection; **import** java.sql.DriverManager; **import** java.sql.SQLException;  **public class** Demo {  **public static void** main(String[] args) **throws** SQLException, ClassNotFoundException {  String url = **"jdbc:mysql://localhost:3306/mytest?useSSL=false&useUnicode=true&characterEncoding=utf8"**;  *// 直接加载一下Driver类* Class.*forName*(**"com.mysql.jdbc.Driver"**);  *// 即可拿到连接* Connection conn = DriverManager.*getConnection*(url, **"root"**, **"123456"**);  System.***out***.println(conn);  conn.close();  } } |

以后都像这样写获得连接的代码，十分方便。

# 2. JDBC API的使用

获得了数据库连接Connection对象后，就能拿到其他对象对数据库进行操作了。下面讲JDBC主要的API以及对MySQL数据库的操作。

## 2.1 Statement对象

Statement对象用于执行SQL语句。可调用Connection对象的createStatement()方法来获得Statement对象。Statement对象可执行各种SQL语句，常来执行增删改查等操作。

例1：使用Statement对象创建学生表并插入一条数据。

|  |
| --- |
| **package** com.zhang.test;  **import** java.sql.Connection; **import** java.sql.DriverManager; **import** java.sql.SQLException; **import** java.sql.Statement;  **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Statement对象* Statement stmt = conn.createStatement();  String sql = **"create table student(sid int primary key, sname varchar(20)); "**;  **int** row = stmt.executeUpdate(sql); *// 执行建表语句。方法返回值是收影响行数* System.***out***.println(**"受影响行数："** + row);  sql = **"insert into student values(1, '张三');"**;  row = stmt.executeUpdate(sql); *// 执行插入语句* System.***out***.println(**"受影响行数："** + row);  *// 关闭Statement和Connection* stmt.close();  conn.close();  } } |

上面可看到使用executeUpdate(sql)方法可执行一般的SQL语句。方法返回值是受影响的行数。一次executeUpdate不能执行多句SQL。

但是executeUpdate(sql)不适合执行查询语句，因为SQL查询返回的是“一张表”数据。此时，查询SQL使用executeQuery(sql)方法，此方法返回ResultSet结果集对象。结果集中就存储了二维表中的数据。

ResultSet对象中维护了一个指向当前行数的光标，最初光标在第一行之前。ResultSet提供了next()和getXxx(int columnIndex)方法来获得某行的某列数据。

当每次调用next()方法，结果集中光标就下移一行，若无下一行，则返回false。而getXxx(int columnIndex)方法用于获得该行第columnIndex列的数据，Xxx就是该列的数据类型。索引columnIndex是从1开始的。getXxx方法还有重载：getXxx(String columnLabel)，就是指定列名获取数据，列名是不区分大小写的。

即next()用于“遍历”行，getXxx()用于获取列数据。

例2：使用Statement查询学生表中的数据。

|  |
| --- |
| **package** com.zhang.test;  **import** java.sql.\*;  **public class** Demo {  **public static void** main(String[] args) **throws** ClassNotFoundException, SQLException {  Class.*forName*(**"com.mysql.jdbc.Driver"**);  Connection conn = DriverManager.*getConnection*(**"jdbc:mysql://localhost:3306/mytest?useSSL=false&useUnicode=true&characterEncoding=utf8"**, **"root"**, **"123456"**);  Statement stmt = conn.createStatement();  ResultSet rs = stmt.executeQuery(**"select** *\** **from student"**);  *// 循环读取rs* **while**(rs.next()) {  *// 输出该行学生编号，用索引* System.***out***.println(rs.getInt(1));  *// 输出该行学生姓名，用列名* System.***out***.println(rs.getString(**"sname"**));  }  rs.close();  stmt.close();  conn.close();  } } |

ResultSet中还提供了以下常用的方法：

（1）previous()：移动到前一行；

（2）absolute(int row)：移动到指定行；

（3）beforeFirst()：移动ResultSet的最前面；

（4）afterLast()：移动到ResultSet的最后面。

## 2.2 PreparedStatement对象

### 2.2.1 Statement对象执行SQL的缺点

通过一个案例来了解Statement的缺点。

实现登录功能，需要查询数据库中的user表。事先建立user表和数据：

|  |
| --- |
| **create table user**(  **account varchar**(10),  **password varchar**(10) ); **insert into user values**(**'201701'**, **'mima'**); |

登录程序的代码：

|  |
| --- |
| **package** com.zhang.test;  **import** java.io.BufferedReader; **import** java.io.InputStreamReader; **import** java.sql.\*;  **public class** Demo {  **public static void** main(String[] args) **throws** Exception {  BufferedReader readConsole = **new** BufferedReader(**new** InputStreamReader(System.***in***));  System.***out***.println(**"请输入用户名："**);  String account = readConsole.readLine();  System.***out***.println(**"请输入密码"**);  String pwd = readConsole.readLine();  **boolean** status = *checkLogin*(account, pwd);  **if**(status) {  System.***out***.println(**"登录成功！"**);  } **else** {  System.***out***.println(**"登录失败！"**);  }  }   **public static boolean** checkLogin(String userName, String pwd) **throws** Exception {  Class.*forName*(**"com.mysql.jdbc.Driver"**);  Connection conn = DriverManager.*getConnection*(**"jdbc:mysql://localhost:3306/mytest?useSSL=false&useUnicode=true&characterEncoding=utf8"**, **"root"**, **"123456"**);  Statement stmt = conn.createStatement();  *// 这里需要字符串的拼接。userName和pwd都是字符串，所以在它们前后要加上单引号表示是字符串，而单引号是需要转义的。* String sql = **"select** *\** **from user where account=\'"** + userName + **"\'and password=\'"** + pwd + **"\';"**;  *// 这里输出观察最终的SQL语句。* System.***out***.println(sql);  ResultSet rs = stmt.executeQuery(sql);  **boolean** flag = **false**;  **if**(rs.next()) {  *// 若能查询到数据，则设置flag为true* flag = **true**;  }  rs.close();  stmt.close();  conn.close();  *// 返回flag* **return** flag;  } } |

实际上上述代码是有问题的，黑客能轻松登录系统，比如黑客不知道用户名和密码，但是精通SQL，他可能第一次胡乱输入，第二次输入“特殊语句”，比如输入（绿色部分为输入）：

|  |
| --- |
| 请输入用户名：  **我不知用户名**  请输入密码  **我也不知密码' or '1' = '1**  select \* from user where account='我不知用户名'and password='我也不知密码' or '1' = '1';  登录成功！ |

这样竟然成功登录进系统了！Why？我故意在程序中输出了拼接完成的字符串，发现这正是利用了SQL语言的逻辑判断来“骗取”登录系统。

这其实就是一种SQL注入，使用Statement不方便防止SQL注入，而且有时进行字符串拼接十分麻烦。

### 2.2.2 使用PreparedStatement

PreparedStatement就可以解决Statement的上述缺点，可以防止SQL注入，并避免拼接字符串，避免出错。推荐以后都使用PreparedStatement。

PreparedStatement对象用于执行预编译的SQL语句，就是在需要动态变化的地方先用问号代替，之后再插入。可见下面例子。PreparedStatement对象可用Connection对象的prepareStatement(sql)获取。

上述案例可修改为：

|  |
| --- |
| **package** com.zhang.test;  **import** java.io.BufferedReader; **import** java.io.InputStreamReader; **import** java.sql.\*;  **public class** Demo {  **public static void** main(String[] args) **throws** Exception {  BufferedReader readConsole = **new** BufferedReader(**new** InputStreamReader(System.***in***));  System.***out***.println(**"请输入用户名："**);  String account = readConsole.readLine();  System.***out***.println(**"请输入密码"**);  String pwd = readConsole.readLine();  **boolean** status = *checkLogin*(account, pwd);  **if**(status) {  System.***out***.println(**"登录成功！"**);  } **else** {  System.***out***.println(**"登录失败！"**);  }  }   **public static boolean** checkLogin(String userName, String pwd) **throws** Exception {  Class.*forName*(**"com.mysql.jdbc.Driver"**);  Connection conn = DriverManager.*getConnection*(**"jdbc:mysql://localhost:3306/mytest?useSSL=false&useUnicode=true&characterEncoding=utf8"**, **"root"**, **"123456"**);  *// 编写预编译的SQL。每个问号就是一个占位符。* String sql = **"select** *\** **from user where account=? and password=?"**;  *// 执行预编译，此时会进行语法检查。返回预编译PreparedStatement对象* PreparedStatement ps = conn.prepareStatement(sql);  *// 然后通过ps给每个占位符设置数据  // 通过setXxx(int parameterIndex, Object value)方法设置。  // 参数1是占位符索引，从1开始；参数2是该占位符的值* ps.setString(1, userName);  ps.setString(2, pwd);  *// 最后调用executeUpdate()或者executeQuery()方法* ResultSet rs = ps.executeQuery();  **boolean** flag = **false**;  **if**(rs.next()) {  *// 若能查询到数据，则设置flag为true* flag = **true**;  }  rs.close();  ps.close();  conn.close();  *// 返回flag* **return** flag;  } } |

这样就不会产生SQL注入问题了。

使用PreparedStatement步骤：

（1）编写预编译SQL；

（2）调用prepareStatement(sql)获得PreparedStatement对象；

（3）调用PreparedStatement对象的setXxx()方法设置预编译SQL中的参数。

（4）使用PreparedStatement执行executeQuery()获得查询结果或者executeUpdate()执行操作。

## 2.3 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：

|  |
| --- |
| **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"**);  *// 用prepareCall()获得CallableStatement对象  // 也是预编译的SQL* CallableStatement stmt = conn.prepareCall(**"call proTest(?)"**);  *// 设置参数* stmt.setInt(1, 1);  *// 同样调用executeQuery()获得结果集* ResultSet 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));  }  *// 关闭连接* rs.close();  stmt.close();  conn.close();  } } |

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：

|  |
| --- |
| **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"**);  CallableStatement stmt = conn.prepareCall(**"call testIf(?, ?)"**);  *// 设置参数1* stmt.setInt(1, 2);  *// 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   // 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);   stmt.close();  conn.close();  } } |

附：数据库中数据类型与Java数据类型的对应关系。

|  |  |  |
| --- | --- | --- |
| SQL类型 | JDBC对应方法 | 返回类型 |
| bit(1) bit(n) | getBoolean() getBytes() | boolean byte[] |
| tinyint | getByte() | byte |
| smallint | getShort() | short |
| int | getInt() | int |
| bigint | getLong() | long |
| char, varchar, longvarchar | getString() | String |
| text(clob) blob | getClob() getBlob() | Clob Blob |
| date | getDate() | java.sql.Date |
| time | getTime() | java.sql.Time |
| timestamp | getTimestamp() | java.sql.Timestamp |

## 2.4 批处理

有时需要执行多个相同的SQL语句，那么就可以使用批处理提高效率。比如程序中要连续插入100条员工记录，这时SQL语句是相同的，只是数据不一样。

如果直接在循环中每次都执行executeUpdate()的话，效率低，因此学习使用批处理，来一次提交多条SQL语句执行，这样来提高数据库吞吐量。

例子：用批处理在数据库中插入数条员工数据。在程序中，这些数据保存在List< Employee >对象中。

（1）员工类

|  |
| --- |
| **package** com.zhang.bean;  **public class** Employee {  **private int id**; *// 员工编号* **private** String **name**;  **public** Employee(**int** id, String name) {  **this**.**id** = id;  **this**.**name** = name;  }   **public int** getId() {  **return id**;  }   **public void** setId(**int** id) {  **this**.**id** = id;  }   **public** String getName() {  **return name**;  }   **public void** setName(String name) {  **this**.**name** = name;  } } |

（2）主类

|  |
| --- |
| **package** com.zhang.test;  **import** com.zhang.bean.Employee;  **import** java.sql.\*; **import** java.util.ArrayList; **import** java.util.List;  **public class** Demo {  **public static void** main(String[] args) **throws** Exception{  List<Employee> empList = **new** ArrayList<>();  *// 创建21个员工对象* **for**(**int** i = 1; i <= 21; i++) {  empList.add(**new** Employee(i, **"员工"** + i));  }  *// 将员工数据插入到数据库中  addEmployees*(empList);  System.***out***.println(**"成功插入数据"**);  }   **public static void** addEmployees(List<Employee> empList) **throws** Exception {  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"**);  *// 预编译SQL* String sql = **"insert into employee values(?,?)"**;  *// 准备PreparedStatement对象* PreparedStatement ps = conn.prepareStatement(sql);  *// 不一样的只是在循环中，不是每次都执行executeUpdate。  // 而是设置好ps参数后，调用addBatch()方法把当前ps加到批处理中  // 当满足一定数量，再执行批处理，使用executeBatch()方法。* **for**(**int** i = 0; i < empList.size(); i++) {  *// 设置参数* ps.setInt(1, empList.get(i).getId());  ps.setString(2, empList.get(i).getName());  *// 添加到批处理* ps.addBatch();  *// 若有5条，则执行一次批处理* **if**(i % 5 == 0) {  *// 执行批处理* ps.executeBatch();  *// 注意执行完了要清空* ps.clearBatch();  }  }  *// 循环外，还要执行一次批处理，防止有剩余的没有满足5条的批处理* ps.executeBatch();  ps.clearBatch();  *// 关闭资源* ps.close();  conn.close();  } } |

即用Statement维护批处理。执行批处理后不要忘了清空批处理。

## 2.5 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：

|  |
| --- |
| **package** com.zhang.test;  **import** com.zhang.bean.Employee;  **import** java.sql.\*; **import** java.util.List;  **public class** Demo {  **public static void** main(String[] args) **throws** Exception{  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"**);  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* PreparedStatement ps = conn.prepareStatement(sql, Statement.***RETURN\_GENERATED\_KEYS***);  *// 设置ps参数* ps.setString(1, **"系统研发部"**);  *// 先执行完ps* ps.executeUpdate();  *// 然后用ps的getGeneratedKeys()方法获得自增长的结果集* ResultSet rs = ps.getGeneratedKeys();  **int** keys = 0;  **while**(rs.next()) {  keys = rs.getInt(1);  }  System.***out***.println(**"获得的自增长键是："** + keys);  ps.close();  conn.close();  } } |

注意的是，获得自增长键调用的是Statement的方法。（PreparedStatement继承了Statement，CallableStatement继承了PreparedStatement。）

## 2.6 事物

事物的一个案例就是银行转账。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。不允许出现只有一方成功或者失败。

组成事务的一部分必须全部成功，否则整个操作就回滚。事物在数据库中非常重要。在上述的操作中，我们执行的操作都是自动提交的。我们可调用Connection的setAutoCommit()方法，设置自动提交为false，实现自己手动控制事物提交。

手动提交事物常用：

（1）void commit()：手动提交事物，一切正常，就可提交，提交后不可回滚；

（2）void rollback()：回滚，出现异常时，已执行但未提交的代码可回退到事务开始前的状态；

（3）可调用setSavepoint()方法设置一个保存点，可指定回滚到该点。方法返回一个Savepoint对象。

例子：

（1）账户表

|  |
| --- |
| **create table account**(  **name varchar**(20),  **money int** ); *-- 添加两个人* **insert into account values**(**'张三'**, 5000); **insert into account values**(**'李四'**, 5000); |

（2）程序代码。

|  |
| --- |
| **package** com.zhang.test;  **import** java.sql.\*;  **public class** Demo {  **public static void** main(String[] args) **throws** Exception{  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"**);  *// 首先设置手动提交* conn.setAutoCommit(**false**);  *// 张三给李四转账1000* String zs = **"update account set money = money - 1000 where name = '张三'"**;  String ls = **"update account set money = money + 1000 where name = '李四'"**;  PreparedStatement ps = **null**;  **try** {  *// 执行第一条SQL* ps = conn.prepareStatement(zs);  ps.executeUpdate();  **int** a = 1 / 0; *// 如果不加这句，转账即成功。如果不使用事物的话，那么就会张三少了1000，而李四没有增加1000  // 执行第二条SQL* ps = conn.prepareStatement(ls);  ps.executeUpdate();  } **catch**(Exception e) {  *// 出现异常，则回滚* conn.rollback();  }  *// 无异常，则提交* conn.commit();  ps.close();  conn.close();  } } |

案例2：使用Savepoint，并且rollback(sp)。

|  |
| --- |
| **package** com.zhang.test;  **import** java.sql.\*;  **public class** Demo {  **public static void** main(String[] args) **throws** Exception {  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"**);  conn.setAutoCommit(**false**);  *// 张三给李四转账1000* String zs = **"update account set money = money - 1000 where name = '张三'"**;  String ls = **"update account set money = money + 1000 where name = '李四'"**;  *// 第一次转账* PreparedStatement ps = conn.prepareStatement(zs);  ps.executeUpdate();  ps = conn.prepareStatement(ls);  ps.executeUpdate();  *// 第一次转账后可设置个保存点。可回滚到此位置* Savepoint sp = conn.setSavepoint();  **try** {  *// 第二次转账* ps = conn.prepareStatement(zs);  ps.executeUpdate();  **int** a = 1 / 0;  ps = conn.prepareStatement(ls);  ps.executeUpdate();  } **catch** (Exception e) {  conn.rollback(sp);  }  *// 无异常，则提交* conn.commit();  ps.close();  conn.close();  } } |