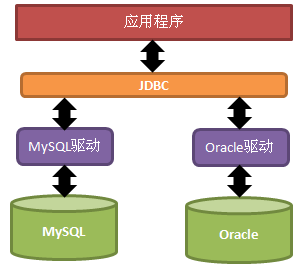
# JDBC入门

## 1　什么是JDBC

　　JDBC（Java DataBase Connectivity）就是Java数据库连接，说白了就是用Java语言来操作数据库。原来我们操作数据库是在控制台使用SQL语句来操作数据库，JDBC是用Java语言向数据库发送SQL语句。

## 2　JDBC原理

早期SUN公司的天才们想编写一套可以连接天下所有数据库的API，但是当他们刚刚开始时就发现这是不可完成的任务，因为各个厂商的数据库服务器差异太大了。后来SUN开始与数据库厂商们讨论，最终得出的结论是，由SUN提供一套访问数据库的规范（就是一组接口），并提供连接数据库的协议标准，然后各个数据库厂商会遵循SUN的规范提供一套访问自己公司的数据库服务器的API出现。SUN提供的规范命名为JDBC，而各个厂商提供的，遵循了JDBC规范的，可以访问自己数据库的API被称之为驱动！



JDBC是接口，而JDBC驱动才是接口的实现，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。

当然还有第三方公司专门为某一数据库提供驱动，这样的驱动往往不是开源免费的！

JDBC并不能直接访问数据库,必须依赖于数据库厂商提供的JDBC驱动程序;

## 3　JDBC核心类（接口）介绍

JDBC中的核心类有：DriverManager、Connection、Statement，和ResultSet！

### 3.1 DriverManager

DriverManger（驱动管理器）的作用有两个：

* 注册驱动：这可以让JDBC知道要使用的是哪个驱动；
* 获取Connection：如果可以获取到Connection，那么说明已经与数据库连接上了。

否则抛出SQLException异常,说明没有连接上;

### 3.2 Connection

Connection对象表示连接，与数据库的通讯都是通过这个对象展开的：

* Connection最为重要的一个方法就是用来获取Statement对象；

### 3.3 Statement

Statement是用来向数据库发送SQL语句的，这样数据库就会执行发送过来的SQL语句：

* void executeUpdate(String sql)：执行更新操作（insert、update、delete等）；
* ResultSet executeQuery(String sql)：执行查询操作，数据库在执行查询后会把查询结果临时存储在ResultSet，查询结果就是ResultSet；

ResultSet对象表示查询结果集，只有在执行查询操作后才会有结果集的产生。结果集是一个二维的表格，有行有列。操作结果集要学习移动ResultSet内部的“行光标”，以及获取当前行上的每一列上的数据：

* boolean next()：使“行光标”移动到下一行，并返回移动后的行是否存在；
* XXX getXXX(int col)：获取当前行指定列上的值，参数就是列数，列数从1开始，而不是0。

## 4　Hello JDBC

JDBC技术主要完成以下几个任务:

1. 与数据库建立一个连接
2. 向数据库发送SQL数据
3. 处理从数据库返回的结果

下面开始编写第一个JDBC程序:

### 4.1　各个数据库的驱动名称、连接参数及jar包：

4.1.1 SQL Server 2000 ：

（1）driver\_class：com.microsoft.jdbc.sqlserver.SQLServerDriver

（2）url：jdbc:microsoft:sqlserver://localhost:1433;DatabaseName=数据库名称

（3）jar包：msbase.jar、mssqlserver.jar、msutil.jar

4.1.2 SQL Server 2005：

（1）driver\_class：com.microsoft.sqlserver.jdbc.SQLServerDriver

（2）url：jdbc:microsoft:sqlserver://localhost:1433;DatabaseName=数据库名称

（3）jar包：sqljdbc.jar

4.1.3 SQL Server 2008：

（1）driver\_class：com.microsoft.sqlserver.jdbc.SQLServerDriver

（2）url：jdbc:sqlserver://localhost:1433;databaseName=数据库名称

（3）jar包：sqljdbc4.jar

4.1.4 MYSQL：

（1）driver\_class：com.mysql.jdbc.Driver

（2）url：jdbc:mysql://localhost:3306/数据库名称

（3）jar包：mysql-connector-java-5.1.13-bin.jar；

4.1.5 Oracle(thin模式)：

（1）driver\_class：oracle.jdbc.driver.OracleDriver

（2）url：jdbc:oracle:thin:@loaclhost:1521:数据库名称

（3）jar包：classes12.jar 或 ojdbc14.jar (存放在 oracle安装目录/jdbc/lib 下)，其中classes12.jar用于JDK1.2和JDK1.3，ojdbc14.jar用于JDK1.4及以上。

4.1.6 DB2：

（1）driver\_class：com.ibm.db2.jdbc.app.DB2.Driver

（2）url：jdbc:db2://localhost:5000/数据库名称

（3）jar包：db2jcc.jar

### 4.2　获取连接

获取连接需要两步，一是使用DriverManager来注册驱动，二是使用DriverManager来获取Connection对象。

注册驱动

**看清楚了，注册驱动就只有一句话：Class.forName(“com.mysql.jdbc.Driver”)，下面的内容都是对这句代码的解释。今后我们的代码中，与注册驱动相关的代码只有这一句。这是通过反射来注册包中的Driver类；**



DriverManager类的registerDriver()方法的参数是java.sql.Driver，但java.sql.Driver是一个接口，实现类由mysql驱动来提供，mysql驱动中的java.sql.Driver接口的实现类为com.mysql.jdbc.Driver！那么注册驱动的代码如下：

**DriverManager.registerDriver(new com.mysql.jdbc.Driver());**

上面代码虽然可以注册驱动，但是出现硬编码（代码依赖mysql驱动jar包），如果将来想连接Oracle数据库，那么必须要修改代码的。并且其实这种注册驱动的方式是注册了两次驱动！

JDBC中规定，驱动类在被加载时，需要自己“主动”把自己注册到DriverManger中，下面我们来看看com.mysql.jdbc.Driver类的源代码：

com.mysql.jdbc.Driver.java

|  |
| --- |
| **public** **class** Driver **extends** NonRegisteringDriver **implements** java.sql.Driver {  **static** {  **try** {  java.sql.DriverManager.*registerDriver*(**new** Driver());  } **catch** (SQLException E) {  **throw** **new** RuntimeException("Can't register driver!");  }  }  ……  } |

com.mysql.jdbc.Driver类中的static块会创建本类对象，并注册到DriverManager中。这说明只要去加载com.mysql.jdbc.Driver类，那么就会执行这个static块，从而也就会把com.mysql.jdbc.Driver注册到DriverManager中，所以可以把**注册驱动类**的代码修改为**加载驱动类**。

Class.forName(“com.mysql.jdbc.Driver”);

获取连接

获取连接的也只有一句代码：DriverManager.getConnection(url,username,password)，其中username和password是登录数据库的用户名和密码，如果我没说错的话，你的mysql数据库的用户名和密码分别是：root、123。

**url查对复杂一点，它是用来找到要连接数据库“网址”**，就好比你要浏览器中查找百度时，也需要提供一个url。下面是mysql的url：

jdbc:mysql://localhost:3306/mydb1

JDBC规定url的格式由三部分组成，每个部分中间使用逗号分隔。

* + 第一部分是jdbc，这是固定的；
  + 第二部分是数据库名称，那么连接mysql数据库，第二部分当然是mysql了；
  + 第三部分是由数据库厂商规定的，我们需要了解每个数据库厂商的要求，mysql的第三部分分别由数据库服务器的IP地址（localhost）、端口号（3306），以及DATABASE名称(mydb1)组成。

下面是获取连接的语句：

Connection con = DriverManager.getConnection(“jdbc:mysql://localhost:3306/mydb1”,”root”,”123”);

还可以在url中提供参数：

jdbc:mysql://localhost:3306/mydb1**?useUnicode=true&characterEncoding=UTF8**

useUnicode参数指定这个连接数据库的过程中，使用的字节集是Unicode字节集；

characherEncoding参数指定穿上连接数据库的过程中，使用的字节集编码为UTF-8编码。请注意，mysql中指定UTF-8编码是给出的是UTF8，而不是UTF-8。要小心了！

### 4.3　获取Statement

在得到Connectoin之后，说明已经与数据库连接上了，下面是通过Connection获取Statement对象的代码：

Statement stmt = con.createStatement();

Statement是用来向数据库发送要执行的SQL语句的！

### 4.4　发送SQL增、删、改语句

String sql = “insert into user value(’zhangSan’, ’123’)”;

int m = stmt.executeUpdate(sql);

其中int类型的返回值表示执行这条SQL语句所影响的行数，我们知道，对insert来说，最后只能影响一行，而update和delete可能会影响0~n行。

如果SQL语句执行失败，那么executeUpdate()会抛出一个SQLException。

### 4.5　发送SQL查询语句

String sql = “select \* from user”;

ResultSet rs = stmt.executeQuery(sql);

请注册，执行查询使用的不是executeUpdate()方法，而是executeQuery()方法。executeQuery()方法返回的是ResultSet，ResultSet封装了查询结果，我们称之为结果集。

### 4.6　读取结果集中的数据

ResultSet就是一张二维的表格，它内部有一个“行光标”，光标默认的位置在“第一行上方”，我们可以调用rs对象的next()方法把“行光标”向下移动一行，当第一次调用next()方法时，“行光标”就到了第一行记录的位置，这时就可以使用ResultSet提供的getXXX(int col)方法来获取指定列的数据了：

rs.next();//光标移动到第一行

rs.getInt(1);//获取第一行第一列的数据

当你使用rs.getInt(1)方法时，你必须可以肯定第1列的数据类型就是int类型，如果你不能肯定，那么最好使用rs.getObject(1)。在ResultSet类中提供了一系列的getXXX()方法，比较常用的方法有：

Object getObject(int col)

String getString(int col)

int getInt(int col)

double getDouble(int col)

### 4.7　关闭

与IO流一样，使用后的东西都需要关闭！关闭的顺序是先得到的后关闭，后得到的先关闭。

rs.close();

stmt.close();

con.close();

### 4.8　代码

|  |
| --- |
| **public** **static** Connection getConnection() **throws** Exception {  Class.*forName*("com.mysql.jdbc.Driver");  String url = "jdbc:mysql://localhost:3306/mydb1";  **return** DriverManager.*getConnection*(url, "root", "123");  } |
| @Test  **public** **void** insert() **throws** Exception {  Connection con = *getConnection*();  Statement stmt = con.createStatement();  String sql = "insert into user values('zhangSan', '123')";  stmt.executeUpdate(sql);  System.*out*.println("插入成功！");  } |
| @Test  **public** **void** update() **throws** Exception {  Connection con = *getConnection*();  Statement stmt = con.createStatement();  String sql = "update user set password='456' where username='zhangSan'";  stmt.executeUpdate(sql);  System.*out*.println("修改成功！");  } |
| @Test  **public** **void** delete() **throws** Exception {  Connection con = *getConnection*();  Statement stmt = con.createStatement();  String sql = "delete from user where username='zhangSan'";  stmt.executeUpdate(sql);  System.*out*.println("删除成功！");  } |
| @Test  **public** **void** query() **throws** Exception {  Connection con = *getConnection*();  Statement stmt = con.createStatement();  String sql = "select \* from user";  ResultSet rs = stmt.executeQuery(sql);  **while**(rs.next()) {  String username = rs.getString(1);  String password = rs.getString(2);  System.*out*.println(username + ", " + password);  }  } |

### 4.9　规范化代码

所谓规范化代码就是无论是否出现异常，都要关闭ResultSet、Statement，以及Connection，如果你还记得IO流的规范化代码，那么下面的代码你就明白什么意思了。

|  |
| --- |
| @Test  **public** **void** query() {  Connection con = **null**;  Statement stmt = **null**;  ResultSet rs = **null**;  **try** {  con = *getConnection*();  stmt = con.createStatement();  String sql = "select \* from user";  rs = stmt.executeQuery(sql);  **while**(rs.next()) {  String username = rs.getString(1);  String password = rs.getString(2);  System.*out*.println(username + ", " + password);  }  } **catch**(Exception e) {  **throw** **new** RuntimeException(e);  } **finally** {  **try** {  **if**(rs != **null**) rs.close();  **if**(stmt != **null**) stmt.close();  **if**(con != **null**) con.close();  } **catch**(SQLException e) {}  }  } |

### 4.10 个人代码

|  |
| --- |
| **package** com.practice.jdbc.first;  **import** java.sql.Connection;  **import** java.sql.DriverManager;  **import** java.sql.ResultSet;  **import** java.sql.SQLException;  **import** java.sql.Statement;  **import** org.junit.Test;  /\*\*  \* JDBC:Java DataBase Connectivity Java数据库链接  \* 主要任务:  \* 与数据库建立一个连接  \* 向数据库发送SQL语句  \* 处理从数据库返回的结果  \* **@author** Yorick  \*/  **public** **class** JDBCInduction {  /\*\*  \* 连接数据库需要做的事:  \* 1.加载驱动,只需要第一次访问数据库加载一次  \* 2.获取连接,每次访问数据库创建一个连接  \* **@throws** ClassNotFoundException  \* **@throws** SQLException  \*/  @Test  **public** **void** jdbcConnection() **throws** ClassNotFoundException, SQLException{  /\*四大连接参数  \* 1.驱动名称--JDBC并不能直接访问数据库,必须依赖于数据库厂商提供的JDBC驱动程序,下面为各数据库厂商提供的驱动名称及相关参数  \* 2.数据库连接三个参数:url  \* 3.数据库连接三大参数:username  \* 4.数据库连接三大参数:password  \* \*/  String driverManager = "com.mysql.jdbc.Driver";  String url = "jdbc:mysql://localhost:3306/test1";  String username = "root";  String password = "123";  //加载驱动类  Class.*forName*(driverManager);  //获取数据库连接  Connection conn = DriverManager.*getConnection*(url, username, password);  //获取Statement对象  Statement sql = conn.createStatement();  //通过statement对象,发送增语句  String insert = "INSERT INTO stu(sname,age,gander,province,tuition) VALUES ('吕嬃',34,'女','湖南',3000)";  //对于 SQL 数据操作语言 (DML) 语句，返回行计数 (2) 对于什么都不返回的 SQL 语句，返回 0  **int** r = sql.executeUpdate(insert);  System.*out*.println(r);  //发送该语句  String update = "update stu set sid=1 where sid=152";  r = sql.executeUpdate(update);  System.*out*.println(r);  //发送删除语句  String delete = "delete from stu where sid>153";  r = sql.executeUpdate(delete);  System.*out*.println(r);  }  //查询操作  @Test  **public** **void** jdbcSelect() **throws** ClassNotFoundException, SQLException{  //获取连接  Class.*forName*("com.mysql.jdbc.Driver");  Connection conn = DriverManager.*getConnection*("jdbc:mysql://localhost:3306/test1", "root", "123");  //获取Statement对象  Statement sql = conn.createStatement();  String select = "select \* from dept order by deptno desc";  ResultSet set = sql.executeQuery(select);  //遍历结果集  **while** (set.next()) {//指针会停留在第一行的上边,把光标向下移动一行，并判断当前行是否存在，如果存在，打印当前行数据  //以 Java 编程语言中 int 的形式获取此 ResultSet 对象的当前行中指定列的值。  **int** deptno = set.getInt(1);  String dname = set.getString("dname");  System.*out*.println(deptno+" "+dname);  }  }  //规范化操作;连接、语句集、结果集都需要关闭  @Test  **public** **void** jdbcNormalize(){  /\*  \* try外给引用  \* try内实例化  \* finally来关闭  \*/  Connection conn = **null**;  Statement sql = **null**;  ResultSet set = **null**;  **try** {  String driverName = "com.mysql.jdbc.Driver";  String url = "jdbc:mysql://localhost:3306/test1";  String username = "root";  String password = "123";  Class.*forName*(driverName);  //try内实例化  conn = DriverManager.*getConnection*(url,username,password);  sql = conn.createStatement();  String select = "select \* from dept order by deptno";  set = sql.executeQuery(select);  **while** (set.next()) {  **int** deptno = set.getInt(1);  String dname = set.getString("dname");  System.*out*.println(deptno+" "+dname);  }  } **catch** (ClassNotFoundException e) {  // **TODO** Auto-generated catch block  e.printStackTrace();  } **catch** (SQLException e) {  // **TODO** Auto-generated catch block  e.printStackTrace();  }**finally**{  // 关闭要倒关,如果不关,那么会不停的耗费资源,直到资源耗尽,这种bug,我们很难发现,所以我们要养成良好习惯  **try** {  //因为我们可能出现一些输入错误,导致获取到的conn等是空的,那么这里我们要进行一个空指针判断  /\*如果conn为null,那么上边catch会处理异常,但是处理异常之前是要走到finally的,如果不加判断,那么finally中会报出空指针异常\*/  **if**(set != **null**) set.close();  **if**(sql != **null**) sql.close();  **if**(conn != **null**) conn.close();  } **catch** (SQLException e) {  // **TODO** Auto-generated catch block  e.printStackTrace();  }  }  }  } |

# JDBC对象介绍

## 1　JDBC中的主要类（接口）

在JDBC中常用的类有：

* DriverManager；
* Connection；
* Statement；
* ResultSet。

## 2　DriverManager

其实我们今后只需要会用DriverManager的getConnection()方法即可：

1. Class.forName(“com.mysql.jdbc.Driver”);//注册驱动
2. String url = “jdbc:mysql://localhost:3306/mydb1”;
3. String username = “root”;
4. String password = “123”;
5. Connection con = DriverManager.getConnection(url, username, password);

注意，上面代码可能出现的两种异常：

1. ClassNotFoundException：这个异常是在第1句上出现的，出现这个异常有两个可能：

* 你没有给出mysql的jar包；
* 你把类名称打错了，查看类名是不是com.mysql.jdbc.Driver。

1. SQLException：这个异常出现在第5句，出现这个异常就是三个参数的问题，往往username和password一般不是出错，所以需要认真查看url是否打错。

对于DriverManager.registerDriver()方法了解即可，因为我们今后注册驱动只会Class.forName()，而不会使用这个方法。

## 3　Connection

Connection最为重要的方法就是获取Statement：

* Statement stmt = con.createStatement();

后面在学习ResultSet方法时，还要学习一下下面的方法：

* Statement stmt = con.createStatement(int,int);

## 4　Statement

Statement最为重要的方法是：

* int executeUpdate(String sql)：执行更新操作，即执行insert、update、delete语句，其实这个方法也可以执行create table、alter table，以及drop table等语句，但我们很少会使用JDBC来执行这些语句；
* ResultSet executeQuery(String sql)：执行查询操作，执行查询操作会返回ResultSet，即结果集。
* boolean execute()

Statement还有一个boolean execute()方法，这个方法可以用来执行增、删、改、查所有SQL语句。该方法返回的是boolean类型，表示SQL语句是否执行成功。

如果使用execute()方法执行的是更新语句，那么还要调用int getUpdateCount()来获取insert、update、delete语句所影响的行数。

如果使用execute()方法执行的是查询语句，那么还要调用ResultSet getResultSet()来获取select语句的查询结果。

## 5　ResultSet之滚动结果集（了解）

ResultSet表示结果集，它是一个二维的表格！ResultSet内部维护一个行光标（游标），ResultSet提供了一系列的方法来移动游标：

* void beforeFirst()：把光标放到第一行的前面，这也是光标默认的位置；
* void afterLast()：把光标放到最后一行的后面；
* boolean first()：把光标放到第一行的位置上，返回值表示调控光标是否成功；
* boolean last()：把光标放到最后一行的位置上；
* boolean isBeforeFirst()：当前光标位置是否在第一行前面；
* boolean isAfterLast()：当前光标位置是否在最后一行的后面；
* boolean isFirst()：当前光标位置是否在第一行上；
* boolean isLast()：当前光标位置是否在最后一行上；
* boolean previous()：把光标向上挪一行；
* boolean next()：把光标向下挪一行；
* boolean relative(int row)：相对位移，当row为正数时，表示向下移动row行，为负数时表示向上移动row行；
* boolean absolute(int row)：绝对位移，把光标移动到指定的行上；
* int getRow()：返回当前光标所有行。

上面方法分为两类，一类用来判断游标位置的，另一类是用来移动游标的。如果结果集是不可滚动的，那么只能使用next()方法来移动游标，而beforeFirst()、afterLast()、first()、last()、previous()、relative()方法都不能使用！！！

结果集是否支持滚动，要从Connection类的createStatement()方法说起。也就是说创建的Statement决定了使用Statement创建的ResultSet是否支持滚动。

Statement createStatement(int resultSetType, int resultSetConcurrency)

resultSetType的可选值：

* ResultSet.TYPE\_FORWARD\_ONLY：不滚动结果集；
* ResultSet.TYPE\_SCROLL\_INSENSITIVE：滚动结果集，但结果集数据不会再跟随数据库而变化；

ResultSet.TYPE\_SCROLL\_SENSITIVE（没有数据库驱动会支持它！）：滚动结果集，但结果集数据不会再跟随数据库而变化；

可以看出，如果想使用滚动的结果集，我们应该选择TYPE\_SCROLL\_INSENSITIVE！其实很少有数据库驱动会支持TYPE\_SCROLL\_SENSITIVE的特性！通常我们也不需要查询到的结果集再受到数据库变化的影响。

resultSetConcurrency的可选值：

* [CONCUR\_READ\_ONLY](mk:@MSITStore:F:\\帮助\\J2SE6.0中文.chm::/java/sql/ResultSet.html" \l "CONCUR_READ_ONLY)：结果集是只读的，不能通过修改结果集而反向影响数据库；
* CONCUR\_UPDATABLE：结果集是可更新的，对结果集的更新可以反向影响数据库。

通常可更新结果集这一“高级特性”我们也是不需要的！

获取滚动结果集的代码如下：

Connection con = …

Statement stmt = con.createStatement(ResultSet.TYPE\_SCROLL\_INSENSITIVE, [CONCUR\_READ\_ONLY](mk:@MSITStore:F:\\帮助\\J2SE6.0中文.chm::/java/sql/ResultSet.html" \l "CONCUR_READ_ONLY));

String sql = …//查询语句

ResultSet rs = stmt.executeQuery(sql);//这个结果集是可滚动的

## 6　ResultSet之获取列数据

可以通过next()方法使ResultSet的游标向下移动，当游标移动到你需要的行时，就需要来获取该行的数据了，ResultSet提供了一系列的获取列数据的方法：

* String getString(int columnIndex)：获取指定列的String类型数据；
* int getInt(int columnIndex)：获取指定列的int类型数据；
* double getDouble(int columnIndex)：获取指定列的double类型数据；
* boolean getBoolean(int columnIndex)：获取指定列的boolean类型数据；
* Object getObject(int columnIndex)：获取指定列的Object类型的数据。

上面方法中，参数columnIndex表示列的索引，列索引从1开始，而不是0，这第一点与数组不同。如果你清楚当前列的数据类型，那么可以使用getInt()之类的方法来获取，如果你不清楚列的类型，那么你应该使用getObject()方法来获取。

ResultSet还提供了一套通过列名称来获取列数据的方法：

获取一列的数据，有两种方式，getXxx(int columIndex)，还有一种：getXxx(String columnName)

* String getString(String columnName)：获取名称为columnName的列的String数据；
* int getInt(String columnName)：获取名称为columnName的列的int数据；
* double getDouble(String columnName)：获取名称为columnName的列的double数据；
* boolean getBoolean(String columnName)：获取名称为columnName的列的boolean数据；
* Object getObject(String columnName)：获取名称为columnName的列的Object数据；

# PreparedStatement

## 1　什么是SQL攻击

在需要用户输入的地方，用户输入的是SQL语句的片段，最终用户输入的SQL片段与我们DAO中写的SQL语句合成一个完整的SQL语句！例如用户在登录时输入的用户名和密码都是为SQL语句的片段！

## 2　演示SQL攻击

首先我们需要创建一张用户表，用来存储用户的信息。

|  |
| --- |
| **CREATE TABLE user(**  **uid CHAR(32) PRIMARY KEY,**  **username VARCHAR(30) UNIQUE KEY NOT NULL,**  **PASSWORD VARCHAR(30)**  **);**  **INSERT INTO user VALUES('U\_1001', 'zs', 'zs');**  **SELECT \* FROM user;** |

现在用户表中只有一行记录，就是zs。

下面我们写一个login()方法！

|  |
| --- |
| **public** **void** login(String username, String password) {  Connection con = **null**;  Statement stmt = **null**;  ResultSet rs = **null**;  **try** {  con = JdbcUtils.*getConnection*();  stmt = con.createStatement();  String sql = "SELECT \* FROM user WHERE " +  "username='" + username +  "' and password='" + password + "'";  rs = stmt.executeQuery(sql);  **if**(rs.next()) {  System.*out*.println("欢迎" + rs.getString("username"));  } **else** {  System.*out*.println("用户名或密码错误！");  }  } **catch** (Exception e) {  **throw** **new** RuntimeException(e);  } **finally** {  JdbcUtils.*close*(con, stmt, rs);  }  } |

下面是调用这个方法的代码：

|  |
| --- |
| login("a' or 'a'='a", "a' or 'a'='a"); |

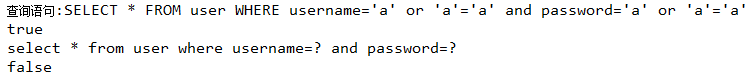
这行当前会使我们登录成功！因为是输入的用户名和密码是SQL语句片段，最终与我们的login()方法中的SQL语句组合在一起！我们来看看组合在一起的SQL语句：

|  |
| --- |
| SELECT \* FROM tab\_user WHERE username='**a' or 'a'='a**' and password='**a' or 'a'='a**' |

## 3　防止SQL攻击

* 过滤用户输入的数据中是否包含非法字符；
* 分步交验！先使用用户名来查询用户，如果查找到了，再比较密码；
* 使用PreparedStatement。

|  |
| --- |
| **package** com.practice.jdbc.preparedStatement;  **import** java.sql.Connection;  **import** java.sql.DriverManager;  **import** java.sql.PreparedStatement;  **import** java.sql.ResultSet;  **import** java.sql.SQLException;  **import** java.sql.Statement;  **import** org.junit.Test;  **public** **class** PreparedStatementDemo {  @Test  **public** **void** test() **throws** SQLException, ClassNotFoundException{  String username = "a' or 'a'='a";  String password = "a' or 'a'='a";  **boolean** flag = login(username, password);  //我们查看数据库发现并没有用户a,但是却返回了true,这就是SQL攻击的一个特例,我们要做的就是防止客户端进行这种攻击  System.*out*.println(flag);  flag = loginPreparedStatement(username, password);  System.*out*.println(flag);  }  //实验SQL攻击  **public** **boolean** login(String username,String password) **throws** SQLException, ClassNotFoundException{  Connection conn = **null**;  Statement sql = **null**;  ResultSet set = **null**;  Class.*forName*("com.mysql.jdbc.Driver");  //try内实例化,获取连接  conn = DriverManager.*getConnection*("jdbc:mysql://localhost:3306/test1", "root", "123");  sql = conn.createStatement();  String select = "SELECT \* FROM user WHERE username='" + username + "' and password='" + password + "'";  // String select = "select \* from user where username = '"+username+"' and password = '"+password+"'";  System.*out*.println("查询语句:"+select);  set = sql.executeQuery(select);  // while (set.next()) {  // String uid = set.getString(1);  // String username1 = set.getString("username");  // String password1 = set.getString("password");  // System.out.println(uid+":"+username1+":"+password1);  // }  **return** set.next();  }  //PreparedStatement防SQL攻击  **public** **boolean** loginPreparedStatement(String username,String password) **throws** ClassNotFoundException, SQLException{  //加载MySQL驱动  Class.*forName*("com.mysql.jdbc.Driver");  //获取数据库连接  Connection conn = DriverManager.*getConnection*("jdbc:mysql://localhost:3306/test1", "root", "123");  //SQL模板  String select = "select \* from user where username=? and password=?";  //使用SQL模板得到PreparedStatement实例  PreparedStatement pstml = conn.prepareStatement(select);  //给SQL模板中的问号赋值  pstml.setString(1, username);  pstml.setString(2, password);  //pstml:com.mysql.jdbc.JDBC4PreparedStatement@fbc9c9: select \* from user where username='a\' or \'a\'=\'a' and password='a\' or \'a\'=\'a'  System.*out*.println(select);  //获取结果集  ResultSet set = pstml.executeQuery();  **return** set.next();  }  } |



## 4　PreparedStatement是什么？

PreparedStatement叫预编译声明！

PreparedStatement是Statement的子接口，你可以使用PreparedStatement来替换Statement。

PreparedStatement的好处：

* 防止SQL攻击；（不只它可以防！）
* 提高代码的可读性，以可维护性；
* 提高效率（很重要！）。

常规的使用Statement进行SQL语句的操作时,需要先校验语法,然后编译,最后才是执行,而使用PreparedStatement的话,那么会在第一次就对SQL模板进行校验和编译,而后传递参数,执行,等到下一次再次使用这个模板的时候就不用再校验和编译了,只用传递参数执行就可以了,这也是他高效率的体现;

## 5　PreparedStatement的使用

* 使用Connection的prepareStatement(String sql)：即创建它时就让它与一条SQL模板绑定；
* 调用PreparedStatement的setXXX()系列方法为问号设置值
* 调用executeUpdate()或executeQuery()方法，但要注意，调用没有参数的方法；

|  |
| --- |
| String sql = “select \* from tab\_student where s\_number=?”;  PreparedStatement pstmt = con.prepareStatement(sql);  pstmt.setString(1, “S\_1001”);  ResultSet rs = pstmt.executeQuery();  rs.close();  //再次使用时需要把原来的设置清空。  pstmt.clearParameters();  pstmt.setString(1, “S\_1002”);  rs = pstmt.executeQuery(); |

在使用Connection创建PreparedStatement对象时需要给出一个SQL模板，所谓SQL模板就是有“?”的SQL语句，其中“?”就是参数。

在得到PreparedStatement对象后，调用它的setXXX()方法为“?”赋值，这样就可以得到把模板变成一条完整的SQL语句，然后再调用PreparedStatement对象的executeQuery()方法获取ResultSet对象。

注意PreparedStatement对象独有的executeQuery()方法是没有参数的，而Statement的executeQuery()是需要参数（SQL语句）的。因为在创建PreparedStatement对象时已经让它与一条SQL模板绑定在一起了，所以在调用它的executeQuery()和executeUpdate()方法时就不再需要参数了。

**PreparedStatement最大的好处就是在于重复使用同一模板，给予其不同的参数来重复的使用它。这才是真正提高效率的原因。**

**所以，建议大家在今后的开发中，无论什么情况，都去需要PreparedStatement，而不是使用Statement。**

|  |
| --- |
| **package** com.practice.jdbc.preparedStatement;  **import** java.sql.Connection;  **import** java.sql.DriverManager;  **import** java.sql.PreparedStatement;  **import** java.sql.SQLException;  **import** java.sql.Statement;  **import** org.junit.Test;  **public** **class** PreparedStatementPractice {    @Test  **public** **void** test() **throws** Exception{  User user = **new** User();  // user.setUid("U\_1002");  user.setUid("U\_1003");  user.setUsername("李四");  user.setPassword("lisi");  // addUser1(user);  addUser2(user);    }  //插入一个对象到数据库,使用Statement  **public** **void** addUser1(User user) **throws** ClassNotFoundException, SQLException{  Class.*forName*("com.mysql.jdbc.Driver");  Connection conn = DriverManager.*getConnection*("jdbc:mysql://localhost:3306/test1", "root", "123");  Statement sql = conn.createStatement();  String insert = "insert into user values('"+user.getUid()+"','"+user.getUsername()+"','"+user.getPassword()+"')";  **int** l = sql.executeUpdate(insert);  System.*out*.println(insert);  System.*out*.println(l);  //关闭  sql.close();  conn.close();  }  //插入一个对象到数据库,使用PreparedStatement  **public** **void** addUser2(User user) **throws** Exception{  Class.*forName*("com.mysql.jdbc.Driver");  Connection conn = DriverManager.*getConnection*("jdbc:mysql://localhost:3306/test1", "root","123");  //给出SQL模板  String insert = "insert into user values(?,?,?)";  //获取PreparedStatement实例,初始化SQL模板  PreparedStatement pstmt = conn.prepareStatement(insert);  //给问号赋值  pstmt.setString(1, user.getUid());  pstmt.setString(2, user.getUsername());  pstmt.setString(3, user.getPassword());  //执行SQL语句  **int** l = pstmt.executeUpdate();  System.*out*.println(l);  //倒序关闭  pstmt.close();  conn.close();  }  } |

## 6 PreparedStatement为什么能防SQL攻击

防SQL攻击也叫防SQL注入，不过是同一种现象的不同称呼。我们知道真正的项目中SQL语句是动态构成的其中的参数是客户端获取到的，那么就存在着客户端动态组织输入，以期达到和我们后台数据库语句构成一句正确的语句，这一句不会报错的语句对数据库的查询结果始终为true，或恶意更改数据库。为了防止这种情况，我们引入了PreparedStatement，那么它是如何防SQL攻击的呢？

使用PreparedStatement，SQL语句在程序运行前已经进行了预编译，在程序运行时第一次操作数据库之前，SQL语句已经被数据库分析，编译和优化，对应的执行计划也会缓存下来并允许数据库已参数化的形式进行查询，当运行时动态地把参数传给PreprareStatement时，即使参数里有敏感字符如 or '1=1'也数据库会作为一个参数一个字段的属性值来处理而不会作为一个SQL指令，如此，就起到了SQL注入的作用了！即Statement会把参数作为SQL语句的一部分来处理，而PreparedStatement则只会将参数作为SQL语句的一个属性值来处理不会作为SQL语句的一部分，所以不存在SQL注入的问题了！

## 7 PreparedStatement为什么能够提高效率

Statement是PreparedStatement的父接口,不进行预编译操作,减少了进行预编译的开销.单次运行PreparedStatement要比Statement要慢一些. PreparedStatement可以实现Statement的所有功能,但是之所以叫它预编译指令,是因为在创建它的一个对象时可以给定具有一定格式的SQL字符串,然后用它的setXXX方法给指定的SQL语句以填空的方式赋值,具有这样的特性后,它在多次执行一条固定格式的字符串时就很方便,也更效率.不像Statement那样每次执行都要先编译字符串在执行SQL了.PreparedStatement需要服务器端的支持来提高效率.比如在Oracle上就会有显著效果,而MySQL明确地说明了不支持PreparedStatement.Oracle中会将所有的SQL语句先编译,叫做"执行计划",放在Oracle内部的一个特定的缓存中,每次遇到相同的SQL,就会先调用缓存中的,如果不预编译,每次都用Statement,那么每次都要编译,在缓冲中会有很多重复的"执行计划"影响数据库的性能.还有一点就是在使用setObject()的时候,记得一定要使用带targetSqlType参数的方法,来提高效率.SQL注入攻击是利用设计上的漏洞,在目标服务器上运行SQL语句进行攻击,动态生成SQL语句时没有对用户输入的数据进行验证是SQL注入攻击得逞的主要原因.对于JDBC而言,SQL注入攻击只对Statement有效,对PreparedStatement是无效的,这是因为PreparedStatement不允许在插入时改变查询的逻辑结构.绕过验证,但这种手段只对Statement有效,对PreparedStatement无效.如果有一条SQL语句: "select \* from 表 where 用户名 = '用户名'"Statement的SQL语句是这样写的: "select \* from 表 where 用户名 = '"+ 变量值 +"'"PreparedStatement的SQL语句是这样写的: "select \* from 表 where 用户名 = ?" 然后对应?赋值这样我们就发现输入 "aa' or '1' = '1"Statement是将这个和SQL语句做字符串连接到一起执行PreparedStatement是将 "aa' or '1' = '1" 作为一个字符串赋值给?,做为"用户名"字段的对应值,显然这样SQL注入无从谈起了.实现机制不同,注入只对SQL语句的准备(编译)过程有破坏作用,而PreparedStatement已经准备好了,执行阶段只是把输入串作为数据处理,不再需要对SQL语句进行解析,准备,因此也就避免了SQL注入问题.

# JdbcUtils工具类

## 1　JdbcUtils的作用

你也看到了，连接数据库的四大参数是：驱动类、url、用户名，以及密码。这些参数都与特定数据库关联，如果将来想更改数据库，那么就要去修改这四大参数，那么为了不去修改代码，我们写一个JdbcUtils类，让它从配置文件中读取配置参数，然后创建连接对象。

## 2　JdbcUtils代码

JdbcUtils.java

|  |
| --- |
| **public** **class** JdbcUtils {  //配置文件路径  **private** **static** **final** String *dbconfig* = "dbconfig.properties";  //对应配置文件  **private** **static** Properties *prop* = **new** Properties();  //把配置文件内容加载到prop对象中。因为是放到static块中完成的加载操作，所以加载操作只会在JdbcUtils类被加载时完成对配置文件的加载。  **static** {  **try** {  InputStream in = Thread.*currentThread*().getContextClassLoader().getResourceAsStream(*dbconfig*);  *prop*.load(in);  Class.*forName*(*prop*.getProperty("driverClassName"));  } **catch**(IOException e) {  **throw** **new** RuntimeException(e);  }  }  //获取Connection方法，参数都是从prop中获取。  **public** **static** Connection getConnection() {  **try** {  **return** DriverManager.*getConnection*(*prop*.getProperty("url"),  *prop*.getProperty("username"), *prop*.getProperty("password"));  } **catch** (Exception e) {  **throw** **new** RuntimeException(e);  }  }  } |

dbconfig.properties

|  |
| --- |
| driverClassName=com.mysql.jdbc.Driver  url=jdbc:mysql://localhost:3306/mydb1?useUnicode=true&characterEncoding=UTF8  username=root  password=123 |

|  |
| --- |
| dirverClassName = com.mysql.jdbc.Driver  url = jdbc\:mysql\://localhost\:3306/test1  username = root  password = 123  package com.you.utils;  import java.io.IOException;  import java.io.InputStream;  import java.sql.Connection;  import java.sql.DriverManager;  import java.sql.SQLException;  import java.util.Properties;  public class JdbcUtils {  //创建一个配置文件对象  private static Properties prop = new Properties();  //静态代码块,加载配置文件,加载驱动,因为这些都只需要操作一次,所以我们放在静态代码块中,随着类的加载而加载  static{  try {  //把文件加载到流中,从流中获取参数  InputStream inputStream = JdbcUtils.class.getClassLoader().getResourceAsStream("dbconfig.properties");  prop.load(inputStream);  //加载驱动类  Class.forName(prop.getProperty("driverClassName"));  } catch (IOException e) {  // TODO Auto-generated catch block  e.printStackTrace();  } catch (ClassNotFoundException e) {  // TODO Auto-generated catch block  e.printStackTrace();  }  }  //获取连接  public static Connection getConnection(){  try {  return DriverManager.getConnection(prop.getProperty("url"), prop.getProperty("username"), prop.getProperty("password"));  } catch (SQLException e) {  // TODO Auto-generated catch block  throw new RuntimeException(e);  }    }  } |

# UserDao

## 1　DAO模式

DAO（Data Access Object）模式就是写一个类，把访问数据库的代码封装起来。DAO在数据库与业务逻辑（Service）之间。

* 实体域，即操作的对象，例如我们操作的表是user表，那么就需要先写一个User类；
* DAO模式需要先提供一个DAO接口；
* 然后再提供一个DAO接口的实现类；
* 再编写一个DAO工厂，Service通过工厂来获取DAO实现。

## 2　代码

User.java

|  |
| --- |
| **public** **class** User {  **private** String uid;  **private** String username;  **private** String password;  …  } |

UserDao.java

|  |
| --- |
| **public** **interface** UserDao {  **public** **void** add(User user);  **public** **void** mod(User user);  **public** **void** del(String uid);  **public** User load(String uid);  **public** List<User> findAll();  } |

UserDaoImpl.java

|  |
| --- |
| **public** **class** UserDaoImpl **implements** UserDao {  **public** **void** add(User user) {  Connection con = **null**;  PreparedStatement pstmt = **null**;  **try** {  con = JdbcUtils.*getConnection*();  String sql = "insert into user value(?,?,?)";  pstmt = con.prepareStatement(sql);  pstmt.setString(1, user.getUid());  pstmt.setString(2, user.getUsername());  pstmt.setString(3, user.getPassword());  pstmt.executeUpdate();  } **catch**(Exception e) {  **throw** **new** RuntimeException(e);  } **finally** {  **try** {  **if**(pstmt != **null**) pstmt.close();  **if**(con != **null**) con.close();  } **catch**(SQLException e) {}  }  }  **public** **void** mod(User user) {  Connection con = **null**;  PreparedStatement pstmt = **null**;  **try** {  con = JdbcUtils.*getConnection*();  String sql = "update user set username=?, password=? where uid=?";  pstmt = con.prepareStatement(sql);  pstmt.setString(1, user.getUsername());  pstmt.setString(2, user.getPassword());  pstmt.setString(3, user.getUid());  pstmt.executeUpdate();  } **catch**(Exception e) {  **throw** **new** RuntimeException(e);  } **finally** {  **try** {  **if**(pstmt != **null**) pstmt.close();  **if**(con != **null**) con.close();  } **catch**(SQLException e) {}  }  }  **public** **void** del(String uid) {  Connection con = **null**;  PreparedStatement pstmt = **null**;  **try** {  con = JdbcUtils.*getConnection*();  String sql = "delete from user where uid=?";  pstmt = con.prepareStatement(sql);  pstmt.setString(1, uid);  pstmt.executeUpdate();  } **catch**(Exception e) {  **throw** **new** RuntimeException(e);  } **finally** {  **try** {  **if**(pstmt != **null**) pstmt.close();  **if**(con != **null**) con.close();  } **catch**(SQLException e) {}  }  }  **public** User load(String uid) {  Connection con = **null**;  PreparedStatement pstmt = **null**;  ResultSet rs = **null**;  **try** {  con = JdbcUtils.*getConnection*();  String sql = "select \* from user where uid=?";  pstmt = con.prepareStatement(sql);  pstmt.setString(1, uid);  rs = pstmt.executeQuery();  **if**(rs.next()) {  **return** **new** User(rs.getString(1), rs.getString(2), rs.getString(3));  }  **return** **null**;  } **catch**(Exception e) {  **throw** **new** RuntimeException(e);  } **finally** {  **try** {  **if**(pstmt != **null**) pstmt.close();  **if**(con != **null**) con.close();  } **catch**(SQLException e) {}  }  }  **public** List<User> findAll() {  Connection con = **null**;  PreparedStatement pstmt = **null**;  ResultSet rs = **null**;  **try** {  con = JdbcUtils.*getConnection*();  String sql = "select \* from user";  pstmt = con.prepareStatement(sql);  rs = pstmt.executeQuery();  List<User> userList = **new** ArrayList<User>();  **while**(rs.next()) {  userList.add(**new** User(rs.getString(1), rs.getString(2), rs.getString(3)));  }  **return** userList;  } **catch**(Exception e) {  **throw** **new** RuntimeException(e);  } **finally** {  **try** {  **if**(pstmt != **null**) pstmt.close();  **if**(con != **null**) con.close();  } **catch**(SQLException e) {}  }  }  } |

UserDaoFactory.java

|  |
| --- |
| **public** **class** UserDaoFactory {  **private** **static** UserDao *userDao*;  **static** {  **try** {  InputStream in = Thread.*currentThread*().getContextClassLoader()  .getResourceAsStream("dao.properties");  Properties prop = **new** Properties();  prop.load(in);  String className = prop.getProperty("cn.itcast.jdbc.UserDao");  Class clazz = Class.*forName*(className);  *userDao* = (UserDao) clazz.newInstance();  } **catch** (Exception e) {  **throw** **new** RuntimeException(e);  }  }  **public** **static** UserDao getUserDao() {  **return** *userDao*;  }  } |

dao.properties

|  |
| --- |
| cn.itcast.jdbc.UserDao=cn.itcast.jdbc.UserDaoImpl |

# 时间类型

SQL包下相关时间的方法。

* java.sql.ResultSet#java.sql.Date getDate()
* java.sql.PreparedStatement#setDate(int col, java.sql.Date date)

例如：user.setBirthday(rs.getDate("birthday"));，其中rs.getDate()方法返回的是java.sql.Date，而User的birthday必须是java.util.Date类型。把sql包的Date赋给util包下的Date，这是子类给父类，不用强转！！！

例如：pstmt.setDate(3, user.getBirthday())，其中user.getBirthday()返回的是util包下的Date，而setDate()方法的参数类型是sql包下的Date，这说明需要把utilDate转换成sqlDate。这需要处理类型转换问题！！！

pstmt.setDate(3, new java.sql.Date(user.getBirthday().getTime()))

## 1　Java中的时间类型

java.sql包下给出三个与数据库相关的日期时间类型，分别是：

* Date：表示日期，只有年月日，没有时分秒。会丢失时间；
* Time：表示时间，只有时分秒，没有年月日。会丢失日期；
* Timestamp：表示时间戳，有年月日时分秒，以及毫秒。

这三个类都是java.util.Date的子类。

## 2　时间类型相互转换

把数据库的三种时间类型赋给java.util.Date，基本不用转换，因为这是把子类对象给父类的引用，不需要转换。

java.sql.Date date = …

java.util.Date d = date;

java.sql.Time time = …

java.util.Date d = time;

java.sql.Timestamp timestamp = …

java.util.Date d = timestamp;

当需要把java.util.Date转换成数据库的三种时间类型时，这就不能直接赋值了，这需要使用数据库三种时间类型的构造器。java.sql包下的Date、Time、TimeStamp三个类的构造器都需要一个long类型的参数，表示毫秒值。创建这三个类型的对象，只需要有毫秒值即可。我们知道java.util.Date有getTime()方法可以获取毫秒值，那么这个转换也就不是什么问题了。

java.utl.Date d = new java.util.Date();

java.sql.Date date = new java.sql.Date(d.getTime());//会丢失时分秒

Time time = new Time(d.getTime());//会丢失年月日

Timestamp timestamp = new Timestamp(d.getTime());

## 3　代码

我们来创建一个dt表：

|  |
| --- |
| CREATE TABLE dt(  d DATE,  t TIME,  ts TIMESTAMP  ) |

下面是向dt表中插入数据的代码：

|  |
| --- |
| @Test  **public** **void** fun1() **throws** SQLException {  Connection con = JdbcUtils.*getConnection*();  String sql = "insert into dt value(?,?,?)";  PreparedStatement pstmt = con.prepareStatement(sql);    java.util.Date d = **new** java.util.Date();  pstmt.setDate(1, **new** java.sql.Date(d.getTime()));  pstmt.setTime(2, **new** Time(d.getTime()));  pstmt.setTimestamp(3, **new** Timestamp(d.getTime()));  pstmt.executeUpdate();  } |

下面是从dt表中查询数据的代码：

|  |
| --- |
| @Test  **public** **void** fun2() **throws** SQLException {  Connection con = JdbcUtils.*getConnection*();  String sql = "select \* from dt";  PreparedStatement pstmt = con.prepareStatement(sql);  ResultSet rs = pstmt.executeQuery();    rs.next();  java.util.Date d1 = rs.getDate(1);  java.util.Date d2 = rs.getTime(2);  java.util.Date d3 = rs.getTimestamp(3);    System.*out*.println(d1);  System.*out*.println(d2);  System.*out*.println(d3);  } |

# 大数据

## 1　什么是大数据

所谓大数据，就是大的字节数据，或大的字符数据。标准SQL中提供了如下类型来保存大数据类型：

|  |  |
| --- | --- |
| 类型 | 长度 |
| tinyblob | 28--1B（256B） |
| blob | 216-1B（64K） |
| mediumblob | 224-1B（16M） |
| longblob | 232-1B（4G） |
| tinyclob | 28--1B（256B） |
| clob | 216-1B（64K） |
| mediumclob | 224-1B（16M） |
| longclob | 232-1B（4G） |

但是，在mysql中没有提供tinyclob、clob、mediumclob、longclob四种类型，而是使用如下四种类型来处理文本大数据：

|  |  |
| --- | --- |
| 类型 | 长度 |
| tinytext | 28--1B（256B） |
| text | 216-1B（64K） |
| mediumtext | 224-1B（16M） |
| longtext | 232-1B（4G） |

首先我们需要创建一张表，表中要有一个mediumblob（16M）类型的字段。

|  |
| --- |
| **CREATE TABLE tab\_bin(**  **id INT PRIMARY KEY AUTO\_INCREMENT,**  **filename VARCHAR(100),**  **data MEDIUMBLOB**  **);** |

向数据库插入二进制数据需要使用PreparedStatement为原setBinaryStream(int, InputSteam)方法来完成。

|  |
| --- |
| con = JdbcUtils.*getConnection*();  String sql = "insert into tab\_bin(filename,data) values(?, ?)";  pstmt = con.prepareStatement(sql);  pstmt.setString(1, "a.jpg");  //得到一个输入流对象  InputStream in = **new** FileInputStream("f:\\a.jpg");  //为第二个参数赋值为流对象  pstmt.setBinaryStream(2, in);  pstmt.executeUpdate(); |

读取二进制数据，需要在查询后使用ResultSet类的getBinaryStream()方法来获取输入流对象。也就是说，PreparedStatement有setXXX()，那么ResultSet就有getXXX()。

|  |
| --- |
| con = JdbcUtils.*getConnection*();  String sql = "select filename,data from tab\_bin where id=?";  pstmt = con.prepareStatement(sql);  pstmt.setInt(1, 1);  rs = pstmt.executeQuery();  rs.next();  String filename = rs.getString("filename");  //使用文件名来创建输出流对象。  OutputStream out = **new** FileOutputStream("F:\\" + filename);  //读取输入流对象  InputStream in = rs.getBinaryStream("data");  //把in中的数据写入到out中。  IOUtils.*copy*(in, out);  out.close(); |

还有一种方法，就是把要存储的数据包装成Blob类型，然后调用PreparedStatement的setBlob()方法来设置数据

|  |
| --- |
| con = JdbcUtils.*getConnection*();  String sql = "insert into tab\_bin(filename,data) values(?, ?)";  pstmt = con.prepareStatement(sql);  pstmt.setString(1, "a.jpg");  File file = **new** File("f:\\a.jpg");  **byte**[] datas = FileUtils.*getBytes*(file);//获取文件中的数据  Blob blob = **new** SerialBlob(datas);//创建Blob对象  pstmt.setBlob(2, blob);//设置Blob类型的参数  pstmt.executeUpdate(); |
| con = JdbcUtils.*getConnection*();  String sql = "select filename,data from tab\_bin where id=?";  pstmt = con.prepareStatement(sql);  pstmt.setInt(1, 1);  rs = pstmt.executeQuery();  rs.next();  String filename = rs.getString("filename");  File file = **new** File("F:\\" + filename) ;  Blob blob = rs.getBlob("data");  **byte**[] datas = blob.getBytes(0, (**int**)file.length());  FileUtils.*writeByteArrayToFile*(file, datas); |

# 批处理

## 1　Statement批处理

批处理就是一批一批的处理，而不是一个一个的处理！

当你有10条SQL语句要执行时，一次向服务器发送一条SQL语句，这么做效率上很差！处理的方案是使用批处理，即一次向服务器发送多条SQL语句，然后由服务器一次性处理。

批处理只针对更新（增、删、改）语句，批处理没有查询什么事儿！

可以多次调用Statement类的addBatch(String sql)方法，把需要执行的所有SQL语句添加到一个“批”中，然后调用Statement类的executeBatch()方法来执行当前“批”中的语句。

* void addBatch(String sql)：添加一条语句到“批”中；
* int[] executeBatch()：执行“批”中所有语句。返回值表示每条语句所影响的行数据；
* void clearBatch()：清空“批”中的所有语句。

|  |
| --- |
| **for**(**int** i = 0; i < 10; i++) {  String number = "S\_10" + i;  String name = "stu" + i;  **int** age = 20 + i;  String gender = i % 2 == 0 ? "male" : "female";  String sql = "insert into stu values('" + number + "', '" + name + "', " + age + ", '" + gender + "')";  //内部有个集合，用来装载sql语句  stmt.addBatch(sql);  }  //执行批，即一次把批中的所有sql语句发送给服务器  stmt.executeBatch(); |

当执行了“批”之后，“批”中的SQL语句就会被清空！也就是说，连续两次调用executeBatch()相当于调用一次！因为第二次调用时，“批”中已经没有SQL语句了。

还可以在执行“批”之前，调用Statement的clearBatch()方法来清空“批”！

## 2　PreparedStatement批处理

PreparedStatement的批处理有所不同，因为每个PreparedStatement对象都绑定一条SQL模板。所以向PreparedStatement中添加的不是SQL语句，而是给“?”赋值。

|  |
| --- |
| con = JdbcUtils.*getConnection*();  String sql = "insert into stu values(?,?,?,?)";  pstmt = con.prepareStatement(sql);  **for**(**int** i = 0; i < 10; i++) {  pstmt.setString(1, "S\_10" + i);  pstmt.setString(2, "stu" + i);  pstmt.setInt(3, 20 + i);  pstmt.setString(4, i % 2 == 0 ? "male" : "female");  //PreparedStatement的addBatch()方法没有参数！  pstmt.addBatch();  }  //执行批  pstmt.executeBatch(); |

# 事务

## 事务概述

为了方便演示事务，我们需要创建一个account表：

|  |
| --- |
| **CREATE TABLE account(**  **id INT PRIMARY KEY AUTO\_INCREMENT,**  **NAME VARCHAR(30),**  **balance NUMERIC(10.2)**  **);**  **INSERT INTO account(NAME,balance) VALUES('zs', 100000);**  **INSERT INTO account(NAME,balance) VALUES('ls', 100000);**  **INSERT INTO account(NAME,balance) VALUES('ww', 100000);**  **SELECT \* FROM account;** |

### 1　什么是事务

银行转账！张三转10000块到李四的账户，这其实需要两条SQL语句：

* 给张三的账户减去10000元；
* 给李四的账户加上10000元。

如果在第一条SQL语句执行成功后，在执行第二条SQL语句之前，程序被中断了（可能是抛出了某个异常，也可能是其他什么原因），那么李四的账户没有加上10000元，而张三却减去了10000元。这肯定是不行的！

你现在可能已经知道什么是事务了吧！事务中的多个操作，要么完全成功，要么完全失败！不可能存在成功一半的情况！也就是说给张三的账户减去10000元如果成功了，那么给李四的账户加上10000元的操作也必须是成功的；否则给张三减去10000元，以及给李四加上10000元都是失败的！

### 2　事务的四大特性（ACID）

事务的四大特性是：

* + 原子性（Atomicity）：事务中所有操作是不可再分割的原子单位。事务中所有操作要么全部执行成功，要么全部执行失败。
  + 一致性(其他特性都是为了这一特性服务的。)（Consistency）：事务执行后，数据库状态与其它业务规则保持一致。如转账业务，无论事务执行成功与否，参与转账的两个账号余额之和应该是不变的。
  + 隔离性（Isolation）：隔离性是指在并发操作中，不同事务之间应该隔离开来，使每个并发中的事务不会相互干扰。
  + 持久性（Durability）：一旦事务提交成功，事务中所有的数据操作都必须被持久化到数据库中，即使提交事务后，数据库马上崩溃，在数据库重启时，也必须能保证通过某种机制恢复数据。

### 3　MySQL中的事务

在默认情况下，MySQL每执行一条SQL语句，都是一个单独的事务。如果需要在一个事务中包含多条SQL语句，那么需要开启事务和结束事务。

* + 开启事务：**start transaction**；
  + 结束事务：**commit**或**rollback**。

在执行SQL语句之前，先执行strat transaction，这就开启了一个事务（事务的起点），然后可以去执行多条SQL语句，最后要结束事务，commit表示提交，即事务中的多条SQL语句所做出的影响会持久化到数据库中。或者rollback，表示回滚，即回滚到事务的起点，之前做的所有操作都被撤消了！

下面演示zs给li转账10000元的示例：

|  |
| --- |
| START TRANSACTION;  UPDATE account SET balance=balance-10000 WHERE id=1;  UPDATE account SET balance=balance+10000 WHERE id=2;  回滚结束，事务执行失败  ROLLBACK; |
| START TRANSACTION;  UPDATE account SET balance=balance-10000 WHERE id=1;  UPDATE account SET balance=balance+10000 WHERE id=2;  //提交结束，事务执行成功  COMMIT; |
| START TRANSACTION;  UPDATE account SET balance=balance-10000 WHERE id=1;  UPDATE account SET balance=balance+10000 WHERE id=2;  //退出，MySQL会自动回滚事务。  quit; |

## JDBC事务

### 1　JDBC中的事务

Connection的三个方法与事务相关：

* + setAutoCommit(boolean)：设置是否为自动提交事务，如果true（默认值就是true）表示自动提交，也就是每条执行的SQL语句都是一个单独的事务，如果设置false，那么就相当于开启了事务了；**con.setAutoCommit(false)表示开启事务！！！**
  + commit()：提交结束事务；**con.commit();表示提交事务**
  + rollback()：回滚结束事务。**con.rollback();表示回滚事务**

jdbc处理事务的代码格式：

try {

con.setAutoCommit(false);//开启事务…

….

…

con.commit();//try的最后提交事务

} catch() {

con.rollback();//回滚事务

}

|  |
| --- |
| **public** **void** transfer(**boolean** b) {  Connection con = **null**;  PreparedStatement pstmt = **null**;    **try** {  con = JdbcUtils.*getConnection*();  //手动提交  //设置为手动提交事务，即开启了事务。  con.setAutoCommit(**false**);  String sql = "update account set balance=balance+? where id=?";  pstmt = con.prepareStatement(sql);  //操作  pstmt.setDouble(1, -10000);  pstmt.setInt(2, 1);  pstmt.executeUpdate();  // 在两个操作中抛出异常  //如果出现了异常就回滚结束事务  **if**(b) {  **throw** **new** Exception();  }    pstmt.setDouble(1, 10000);  pstmt.setInt(2, 2);  pstmt.executeUpdate();    //提交事务  //当两个操作都执行完了，提交结束事务。  con.commit();  } **catch**(Exception e) {  //回滚事务  **if**(con != **null**) {  **try** {  //当出现异常时，回滚事务。  con.rollback();  } **catch**(SQLException ex) {}  }  **throw** **new** RuntimeException(e);  } **finally** {  //关闭  JdbcUtils.*close*(con, pstmt);  }  } |

### 2　保存点（了解）

保存点:Spring中的七种事务的传播机制

保存点是JDBC3.0的东西！当要求数据库服务器支持保存点方式的回滚。

校验数据库服务器是否支持保存点！

|  |
| --- |
| boolean b = con.getMetaData().supportsSavepoints(); |

保存点的作用是允许事务回滚到指定的保存点位置。在事务中设置好保存点，然后回滚时可以选择回滚到指定的保存点，而不是回滚整个事务！**注意，回滚到指定保存点并没有结束事务！！！只有回滚了整个事务才算是结束事务了！**

Connection类的设置保存点，以及回滚到指定保存点方法：

* 设置保存点：Savepoint setSavepoint()；
* 回滚到指定保存点：void rollback(Savepoint)。

|  |
| --- |
| /\*  \* 李四对张三说，如果你给我转1W，我就给你转100W。  \* ==========================================  \*  \* 张三给李四转1W（张三减去1W，李四加上1W）  \* 设置保存点！  \* 李四给张三转100W（李四减去100W，张三加上100W）  \* 查看李四余额为负数，那么回滚到保存点。  \* 提交事务  \*/  @Test  **public** **void** fun() {  Connection con = **null**;  PreparedStatement pstmt = **null**;    **try** {  con = JdbcUtils.*getConnection*();  //手动提交  **con.setAutoCommit(false);**    String sql = "update account set balance=balance+? where name=?";  pstmt = con.prepareStatement(sql);    //操作1（张三减去1W）  pstmt.setDouble(1, -10000);  pstmt.setString(2, "zs");  pstmt.executeUpdate();    //操作2（李四加上1W）  pstmt.setDouble(1, 10000);  pstmt.setString(2, "ls");  pstmt.executeUpdate();    // 设置保存点  Savepoint sp = con.setSavepoint();    //操作3（李四减去100W）  pstmt.setDouble(1, -1000000);  pstmt.setString(2, "ls");  pstmt.executeUpdate();    //操作4（张三加上100W）  pstmt.setDouble(1, 1000000);  pstmt.setString(2, "zs");  pstmt.executeUpdate();    //操作5（查看李四余额）  sql = "select balance from account where name=?";  pstmt = con.prepareStatement(sql);  pstmt.setString(1, "ls");  ResultSet rs = pstmt.executeQuery();  rs.next();  **double** balance = rs.getDouble(1);  　　　　　　//如果李四余额为负数，那么回滚到指定保存点  **if**(balance < 0) {  //发现李四余额小于0，回滚到指定还原点！即撤销了李四给张三转账100万的操作  con.rollback(sp);  System.*out*.println("张三，你上当了！");  }    //提交事务  //注意，一定要提交事务，因为回滚到指定保存点不会结束事务！保存点之前的操作没有被回滚，只能提交了才能真正把没有回滚的操作执行了。  con.commit();  } **catch**(Exception e) {  //回滚事务  **if**(con != **null**) {  **try** {  con.rollback();  } **catch**(SQLException ex) {}  }  **throw** **new** RuntimeException(e);  } **finally** {  //关闭  JdbcUtils.*close*(con, pstmt);  }  } |

## 事务隔离级别

### 1 事务的并发读问题

脏读：读取到另一个事务未提交数据；//不能允许出来的事情！

* + 不可重复读：两次读取不一致；
  + 幻读（虚读）：读到另一事务已提交数据。

### 2　并发事务问题

因为并发事务导致的问题大致有5类，其中两类是更新问题，三类是读问题。

* 脏读（dirty read）：**读到未提交更新数据，即读取到了脏数据；**
* 不可重复读（unrepeatable read）：**对同一记录的两次读取不一致，因为另一事务对该记录做了修改；**
* 幻读（phantom read）：**对同一张表的两次查询不一致，因为另一事务插入了一条记录；**

脏读

事务1：张三给李四转账100元

事务2：李四查看自己的账户

* t1：事务1：开始事务
* t2：事务1：张三给李四转账100元
* t3：事务2：开始事务
* t4：事务2：李四查看自己的账户，看到账户多出100元（脏读）
* t5：事务2：提交事务
* t6：事务1：回滚事务，回到转账之前的状态

不可重复读

事务1：酒店查看两次1048号房间状态

事务2：预订1048号房间

* t1：事务1：开始事务
* t2：事务1：查看1048号房间状态为空闲
* t3：事务2：开始事务
* t4：事务2：预定1048号房间
* t5：事务2：提交事务
* t6：事务1：再次查看1048号房间状态为使用
* t7：事务1：提交事务

对同一记录的两次查询结果不一致！

幻读

事务1：对酒店房间预订记录两次统计

事务2：添加一条预订房间记录

* t1：事务1：开始事务
* t2：事务1：统计预订记录100条
* t3：事务2：开始事务
* t4：事务2：添加一条预订房间记录
* t5：事务2：提交事务
* t6：事务1：再次统计预订记录为101记录
* t7：事务1：提交

　　对同一表的两次查询不一致！

不可重复读和幻读的区别：

* 不可重复读是读取到了另一事务的更新；
* 幻读是读取到了另一事务的插入（MySQL中无法测试到幻读）；

### 3　四大隔离级别

4个等级的事务隔离级别，在相同数据环境下，使用相同的输入，执行相同的工作，根据不同的隔离级别，可以导致不同的结果。不同事务隔离级别能够解决的数据并发问题的能力是不同的。

1　SERIALIZABLE（串行化）

不会出现任何并发问题，因为它是对同一数据的访问是串行的，非并发访问的；三种读问题都能处理

* + 性能最差；

2　REPEATABLE READ（可重复读）（MySQL）

防止脏读和不可重复读；脏读、不可重复读，不能处理幻读

* + 性能比SERIALIZABLE好

3　READ COMMITTED（读已提交数据）（Oracle）

防止脏读；只能处理脏读，不能处理不可重复读和幻读。

* + 性能比REPEATABLE READ好

4　READ UNCOMMITTED（读未提交数据）

可能出现任何事务并发问题,啥也不处理！

* + 性能最好

**MySQL的默认隔离级别为REPEATABLE READ，这是一个很不错的选择吧！**

### 5　MySQL隔离级别

MySQL的默认隔离级别为Repeatable read，可以通过下面语句查看：

|  |
| --- |
| select @@tx\_isolation |

也可以通过下面语句来设置当前连接的隔离级别：

|  |
| --- |
| set transaction isolationlevel [4先1] |

### 6　JDBC设置隔离级别

con. setTransactionIsolation(int level)

参数可选值如下：

* Connection.TRANSACTION\_READ\_UNCOMMITTED；
* Connection.TRANSACTION\_READ\_COMMITTED；
* Connection.TRANSACTION\_REPEATABLE\_READ；
* Connection.TRANSACTION\_SERIALIZABLE。

## 事务总结：

* 事务的特性：ACID；
* 事务开始边界与结束边界：开始边界（con.setAutoCommit(false)），结束边界（con.commit()或con.rollback()）；
* 事务的隔离级别： READ\_UNCOMMITTED、READ\_COMMITTED、REPEATABLE\_READ、SERIALIZABLE。多个事务并发执行时才需要考虑并发事务。

# 数据库连接池

## 数据库连接池

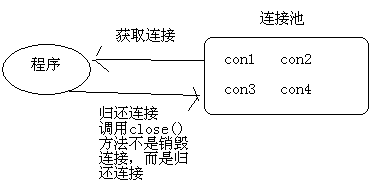
用java代码操作数据库，需要数据库连接对象，一个用户至少要用到一个连接。现在假设有成千上百万个用户，就要创建十分巨大数量的连接对象，这会使数据库承受极大的压力，为了解决这种现象，一种技术出现了，这就是数据库连接池。

## **什么是数据库连接池（原理）**

所谓数据库连接池，可以看作 ：在用户和数据库之间创建一个”池”，这个池中有若干个连接对象，当用户想要连接数据库，就要先从连接池中获取连接对象，然后操作数据库。一旦连接池中的连接对象被拿光了，下一个想要操作数据库的用户必须等待，等待其他用户释放连接对象，把它放回连接池中，这时候等待的用户才能获取连接对象，从而操作数据库。

### 1　数据库连接池的概念

用池来管理Connection，这可以重复使用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。

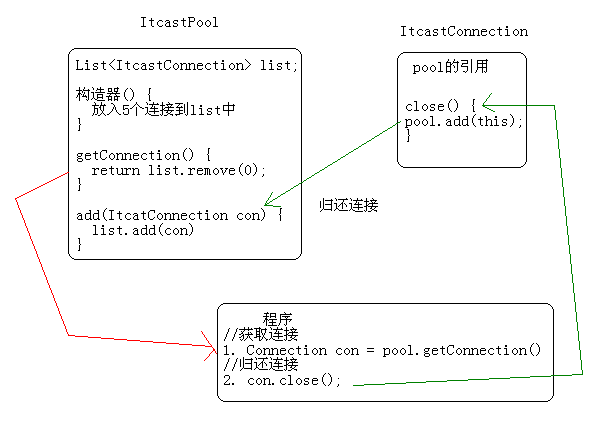


### 2　JDBC数据库连接池接口（DataSource）

　　Java为数据库连接池提供了公共的接口：javax.sql.DataSource，各个厂商可以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！

### ?3　自定义连接池（ItcastPool）

　　分析：ItcastPool需要有一个List，用来保存连接对象。在ItcastPool的构造器中创建5个连接对象放到List中！当用人调用了ItcastPool的getConnection()时，那么就从List拿出一个返回。当List中没有连接可用时，抛出异常。

　　我们需要对Connection的close()方法进行增强，所以我们需要自定义ItcastConnection类，对Connection进行装饰！即对close()方法进行增强。因为需要在调用close()方法时把连接“归还”给池，所以ItcastConnection类需要拥有池对象的引用，并且池类还要提供“归还”的方法。

ItcastPool.java

|  |
| --- |
| **public** **class** ItcastPool **implements** DataSource {  **private** **static** Properties *props* = **new** Properties();  **private** List<Connection> list = **new** ArrayList<Connection>();  **static** {  InputStream in = ItcastPool.**class**.getClassLoader()  .getResourceAsStream("dbconfig.properties");  **try** {  *props*.load(in);  Class.*forName*(*props*.getProperty("driverClassName"));  } **catch** (Exception e) {  **throw** **new** RuntimeException(e);  }  }  **public** ItcastPool() **throws** SQLException {  **for** (**int** i = 0; i < 5; i++) {  Connection con = DriverManager.*getConnection*(  *props*.getProperty("url"), *props*.getProperty("username"),  *props*.getProperty("password"));  ItcastConnection conWapper = **new** ItcastConnection(con, **this**);  list.add(conWapper);  }  }    **public** **void** add(Connection con) {  list.add(con);  }  **public** Connection getConnection() **throws** SQLException {  **if**(list.size() > 0) {  **return** list.remove(0);  }  **throw** **new** SQLException("没连接了");  }  ......  } |

ItcastConnection.java

|  |
| --- |
| **public** **class** ItcastConnection **extends** ConnectionWrapper {  **private** ItcastPool pool;    **public** ItcastConnection(Connection con, ItcastPool pool) {  **super**(con);  **this**.pool = pool;  }  @Override  **public** **void** close() **throws** SQLException {  pool.add(**this**);  }  } |

## DBCP

### 1　什么是DBCP？

DBCP是Apache提供的一款开源免费的数据库连接池！

Hibernate3.0之后不再对DBCP提供支持！因为Hibernate声明DBCP有致命的缺欠！DBCP因为Hibernate的这一毁谤很是生气，并且说自己没有缺欠。

### 2　DBCP的使用

|  |
| --- |
| **public** **void** fun1() **throws** SQLException {  BasicDataSource ds = **new** BasicDataSource();  //基本配置  ds.setUsername("root");  ds.setPassword("123");  ds.setUrl("jdbc:mysql://localhost:3306/mydb1");  ds.setDriverClassName("com.mysql.jdbc.Driver");  //最大连接数  ds.setMaxActive(20);  //最大空闲连接数  ds.setMaxIdle(10);  //初始化连接数  ds.setInitialSize(10);  //最小空闲连接数  ds.setMinIdle(2);  //最大等待毫秒数  ds.setMaxWait(1000);    Connection con = ds.getConnection();  System.*out*.println(con.getClass().getName());  //关闭连接只是把连接归还给池！  con.close();  } |

### 3　DBCP的配置信息

下面是对DBCP的配置介绍：

|  |
| --- |
| #基本配置  driverClassName=com.mysql.jdbc.Driver  url=jdbc:mysql://localhost:3306/mydb1  username=root  password=123  #初始化池大小，即一开始池中就会有10个连接对象  默认值为0  initialSize=0  #最大连接数，如果设置maxActive=50时，池中最多可以有50个连接，当然这50个连接中包含被使用的和没被使用的（空闲）  #你是一个包工头，你一共有50个工人，但这50个工人有的当前正在工作，有的正在空闲  #默认值为8，如果设置为非正数，表示没有限制！即无限大  maxActive=8  #最大空闲连接  #当设置maxIdle=30时，你是包工头，你允许最多有20个工人空闲，如果现在有30个空闲工人，那么要开除10个  #默认值为8，如果设置为负数，表示没有限制！即无限大  maxIdle=8  #最小空闲连接  #如果设置minIdel=5时，如果你的工人只有3个空闲，那么你需要再去招2个回来，保证有5个空闲工人  #默认值为0  minIdle=0  #最大等待时间  #当设置maxWait=5000时，现在你的工作都出去工作了，又来了一个工作，需要一个工人。  #这时就要等待有工人回来，如果等待5000毫秒还没回来，那就抛出异常  #没有工人的原因：最多工人数为50，已经有50个工人了，不能再招了，但50人都出去工作了。  #默认值为-1，表示无限期等待，不会抛出异常。  maxWait=-1  #连接属性  #就是原来放在url后面的参数，可以使用connectionProperties来指定  #如果已经在url后面指定了，那么就不用在这里指定了。  #useServerPrepStmts=true，MySQL开启预编译功能  #cachePrepStmts=true，MySQL开启缓存PreparedStatement功能，  #prepStmtCacheSize=50，缓存PreparedStatement的上限  #prepStmtCacheSqlLimit=300，当SQL模板长度大于300时，就不再缓存它  connectionProperties=useUnicode=true;characterEncoding=UTF8;useServerPrepStmts=true;cachePrepStmts=true;prepStmtCacheSize=50;prepStmtCacheSqlLimit=300  #连接的默认提交方式  #默认值为true  defaultAutoCommit=true  #连接是否为只读连接  #Connection有一对方法：setReadOnly(boolean)和isReadOnly()  #如果是只读连接，那么你只能用这个连接来做查询  #指定连接为只读是为了优化！这个优化与并发事务相关！  #如果两个并发事务，对同一行记录做增、删、改操作，是不是一定要隔离它们啊？  #如果两个并发事务，对同一行记录只做查询操作，那么是不是就不用隔离它们了？  #如果没有指定这个属性值，那么是否为只读连接，这就由驱动自己来决定了。即Connection的实现类自己来决定！  defaultReadOnly=false  #指定事务的事务隔离级别  #可选值：NONE,READ\_UNCOMMITTED, READ\_COMMITTED, REPEATABLE\_READ, SERIALIZABLE  #如果没有指定，那么由驱动中的Connection实现类自己来决定  defaultTransactionIsolation=REPEATABLE\_READ |

## C3P0

### 1　C3P0简介

　　C3P0也是开源免费的连接池！C3P0被很多人看好！

### 2　C3P0的使用

　　C3P0中池类是：ComboPooledDataSource。

|  |
| --- |
| **public** **void** fun1() **throws** PropertyVetoException, SQLException {  ComboPooledDataSource ds = **new** ComboPooledDataSource();  //基本配置  ds.setJdbcUrl("jdbc:mysql://localhost:3306/mydb1");  ds.setUser("root");  ds.setPassword("123");  ds.setDriverClass("com.mysql.jdbc.Driver");  //每次的增量为5  ds.setAcquireIncrement(5);  //初始化连接数  ds.setInitialPoolSize(20);  //最少连接数  ds.setMinPoolSize(2);  //最多连接数  ds.setMaxPoolSize(50);    Connection con = ds.getConnection();  System.*out*.println(con);  con.close();  } |

c3p0也可以指定配置文件，而且配置文件可以是properties，也可骒xml的。当然xml的高级一些了。但是c3p0的配置文件名必须为c3p0-config.xml，并且必须放在类路径下。

|  |
| --- |
| <?xml version=*"1.0"* encoding=*"UTF-8"*?>  <c3p0-config>  //默认配置  <default-config>  <property name=*"jdbcUrl"*>jdbc:mysql://localhost:3306/mydb1</property>  <property name=*"driverClass"*>com.mysql.jdbc.Driver</property>  <property name=*"user"*>root</property>  <property name=*"password"*>123</property>  <property name=*"acquireIncrement"*>3</property>  <property name=*"initialPoolSize"*>10</property>  <property name=*"minPoolSize"*>2</property>  <property name=*"maxPoolSize"*>10</property>  </default-config>  //命名配置  <named-config name=*"oracle-config"*>  <property name=*"jdbcUrl"*>jdbc:mysql://localhost:3306/mydb1</property>  <property name=*"driverClass"*>com.mysql.jdbc.Driver</property>  <property name=*"user"*>root</property>  <property name=*"password"*>123</property>  <property name=*"acquireIncrement"*>3</property>  <property name=*"initialPoolSize"*>10</property>  <property name=*"minPoolSize"*>2</property>  <property name=*"maxPoolSize"*>10</property>  </named-config>  </c3p0-config> |

c3p0的配置文件中可以配置多个连接信息，可以给每个配置起个名字，这样可以方便的通过配置名称来切换配置信息。上面文件中默认配置为mysql的配置，名为oracle-config的配置也是mysql的配置，呵呵。

|  |
| --- |
| **public** **void** fun2() **throws** PropertyVetoException, SQLException {  //不用定配置文件名称，因为配置文件名必须是c3p0-config.xml，这里使用的是默认配置。  ComboPooledDataSource ds = **new** ComboPooledDataSource();  Connection con = ds.getConnection();  System.*out*.println(con);  con.close();  } |
| **public** **void** fun2() **throws** PropertyVetoException, SQLException {  //使用名为oracle-config的配置。  ComboPooledDataSource ds = **new** ComboPooledDataSource("orcale-config");  Connection con = ds.getConnection();  System.*out*.println(con);  con.close();  } |

## Tomcat配置连接池

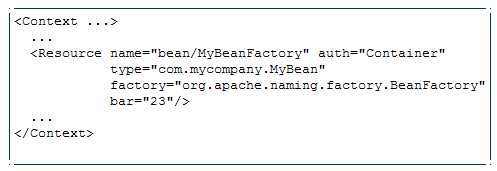
### 1　Tomcat配置JNDI资源

JNDI（Java Naming and Directory Interface），Java命名和目录接口。JNDI的作用就是：在服务器上配置资源，然后通过统一的方式来获取配置的资源。

我们这里要配置的资源当然是连接池了，这样项目中就可以通过统一的方式来获取连接池对象了。

【JNDI全名为Java Naming and Directory Interface.JNDI主要提供应用程序所需要资源上[命名](http://baike.baidu.com/item/%E5%91%BD%E5%90%8D" \t "http://baike.baidu.com/_blank)与[目录服务](http://baike.baidu.com/item/%E7%9B%AE%E5%BD%95%E6%9C%8D%E5%8A%A1" \t "http://baike.baidu.com/_blank)。在Java EE环境中，JNDI扮演了一个很重要的角色，它提供了一个接口让用户在不知道资源所在位置的情形下，取得该资源服务。就好比网络[磁盘驱动器](http://baike.baidu.com/item/%E7%A3%81%E7%9B%98%E9%A9%B1%E5%8A%A8%E5%99%A8" \t "http://baike.baidu.com/_blank)的功能一样。如果有人事先将另一台机器上的磁盘驱动器接到用户的机器上，用户在使用的时候根本就分辨不出现在的驱动器是存在本端，还是在另一端的机器上，用户只需取得资源来用，根本就不知道资源在什么地方。】

下图是Tomcat文档提供的：



配置JNDI资源需要到<Context>元素中配置<Resource>子元素：

* name：指定资源的名称，这个名称可以随便给，在获取资源时需要这个名称；
* factory：用来创建资源的工厂，这个值基本上是固定的，不用修改；
* type：资源的类型，我们要给出的类型当然是我们连接池的类型了；
* bar：表示资源的属性，如果资源存在名为bar的属性，那么就配置bar的值。对于DBCP连接池而言，你需要配置的不是bar，因为它没有bar这个属性，而是应该去配置url、username等属性。

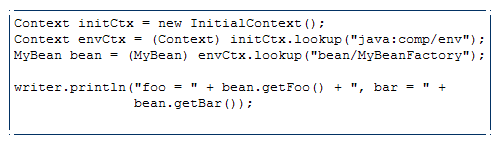
|  |
| --- |
| <Context>  <Resource name="mydbcp"  type="org.apache.tomcat.dbcp.dbcp.BasicDataSource"  factory="org.apache.naming.factory.BeanFactory"  username="root"  password="123"  driverClassName="com.mysql.jdbc.Driver"  url="jdbc:mysql://127.0.0.1/mydb1"  maxIdle="3"  maxWait="5000"  maxActive="5"  initialSize="3"/>  </Context> |
| <Context>  <Resource name="myc3p0"  type="com.mchange.v2.c3p0.ComboPooledDataSource"  factory="org.apache.naming.factory.BeanFactory"  user="root"  password="123"  classDriver="com.mysql.jdbc.Driver"  jdbcUrl="jdbc:mysql://127.0.0.1/mydb1"  maxPoolSize="20"  minPoolSize ="5"  initialPoolSize="10"  acquireIncrement="2"/>  </Context> |

### 2　获取资源

<!-- type="javax.sql.DataSource" url="jdbc:mysql://localhost:3306/test1?characterEncoding=UTF-8"-->

　　配置资源的目的当然是为了获取资源了。只要你启动了Tomcat，那么就可以在项目中任何类中通过JNDI获取资源的方式来获取资源了。

下图是Tomcat文档提供的，与上面Tomcat文档提供的配置资源是对应的。



获取资源：

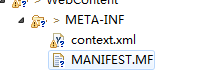
* Context：javax.naming.Context；
* InitialContext：javax.naming.InitialContext；
* lookup(String)：获取资源的方法，其中”java:comp/env”是资源的入口（这是固定的名称），获取过来的还是一个Context，这说明需要在获取到的Context上进一步进行获取。”bean/MyBeanFactory”对应<Resource>中配置的name值，这回获取的就是资源对象了。

|  |
| --- |
| Context cxt = **new** InitialContext();  DataSource ds = (DataSource)cxt.lookup("java:/comp/env/mydbcp");  Connection con = ds.getConnection();  System.*out*.println(con);  con.close(); |
| Context cxt = **new** InitialContext();  Context envCxt = (Context)cxt.lookup("java:/comp/env");  DataSource ds = (DataSource)env.lookup("mydbcp");  Connection con = ds.getConnection();  System.*out*.println(con);  con.close(); |

上面两种方式是相同的效果。

### 实际操作:

1. 在META-INF文件夹下创建context.XML文件



1. 配置Context.xml文件

<?xml version=*"1.0"* encoding=*"UTF-8"*?>

<Context>

<Resource name=*"jdbc/test1"*

type=*"com.mchange.v2.c3p0.ComboPooledDataSource"*

factory=*"org.apache.naming.factory.BeanFactory"*

user=*"root"*

password=*"123"*

driverClass=*"com.mysql.jdbc.Driver"*

jdbcUrl=*"jdbc:mysql://127.0.0.1/test1"*

maxPoolSize=*"20"*

minPoolSize =*"5"*

initialPoolSize=*"10"*

acquireIncrement=*"2"*/>

</Context>

3.配置web.xml文件

<?xml version=*"1.0"* encoding=*"UTF-8"*?>

<web-app xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"* xmlns=*"http://java.sun.com/xml/ns/javaee"* xsi:schemaLocation=*"http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app\_3\_0.xsd"* id=*"WebApp\_ID"* version=*"3.0"*>

<display-name>JDBCPractice02</display-name>

<welcome-file-list>

<welcome-file>index.html</welcome-file>

<welcome-file>index.htm</welcome-file>

<welcome-file>index.jsp</welcome-file>

<welcome-file>default.html</welcome-file>

<welcome-file>default.htm</welcome-file>

<welcome-file>default.jsp</welcome-file>

</welcome-file-list>

<resource-ref>

<description>DB Connection</description>

<res-ref-name>jdbc/test1</res-ref-name>

<res-type>com.mchange.v2.c3p0.ComboPooledDataSource</res-type>

<res-auth>Container</res-auth>

</resource-ref>

</web-app>

1. 在类中配置,加载,获取连接

package com.jdbc.practice.pool.jndi;

import java.sql.Connection;

import java.sql.SQLException;

import javax.naming.Context;

import javax.naming.InitialContext;

import javax.naming.NamingException;

import javax.sql.DataSource;

public class JNDIPracticeDAO {

public void xxx(){

//获取JNDI资源

try {

//创建上下文

Context initCtx = new InitialContext();

/\*

\* 使用上下文查找指定名称的资源

\* 名称：java:comp/env，表示入口点！得到的还是上下文

\*/

// Context envCtx = (Context) initCtx.lookup("java:comp/env");

/\*

\* 继承查找资源，名称为：jdbc/mydatsource，它对应的是<Resource name="jdbc/mydatsource">

\*/

// DataSource dataSource = (DataSource) envCtx.lookup("jdbc/test1");

DataSource dataSource = (DataSource) initCtx.lookup("java:comp/env/jdbc/test1");

// DataSource dataSource = (DataSource) envCtx.lookup("jdbc/mydatasource");

//获取连接

Connection conn = dataSource.getConnection();

System.out.println(conn.getClass().getName());

//归还连接

conn.close();

} catch (NamingException e) {

// TODO Auto-generated catch block

e.printStackTrace();

} catch (SQLException e) {

// TODO Auto-generated catch block

e.printStackTrace();

}

}

}

package com.jdbc.practice.pool.jndi;

import java.io.IOException;

import javax.servlet.ServletException;

import javax.servlet.annotation.WebServlet;

import javax.servlet.http.HttpServlet;

import javax.servlet.http.HttpServletRequest;

import javax.servlet.http.HttpServletResponse;

/\*\*

\* Servlet implementation class JNDIServlet

\*/

@WebServlet("/JNDIServlet")

public class JNDIServlet extends HttpServlet {

protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {

JNDIPracticeDAO dao = new JNDIPracticeDAO();

dao.xxx();

}

}

## 修改JdbcUtils

因为已经学习了连接池，那么JdbcUtils的获取连接对象的方法也要修改一下了。

JdbcUtils.java

|  |
| --- |
| **public** **class** JdbcUtils {  **private** **static** DataSource *dataSource* = **new** ComboPooledDataSource();  **public** **static** DataSource getDataSource() {  **return** *dataSource*;  }  **public** **static** Connection getConnection() {  **try** {  **return** *dataSource*.getConnection();  } **catch** (Exception e) {  **throw** **new** RuntimeException(e);  }  }  } |

# ThreadLocal

|  |  |
| --- | --- |
| key | value |
| thread1 | aaa |
| thread2 | bbb |
| thread3 | ccc |

## 什么是ThreadLocal

ThreadLocal翻译成中文比较准确的叫法应该是：线程局部变量。

　　这个玩意有什么用处，或者说为什么要有这么一个东东？先解释一下，在并发编程的时候，成员变量如果不做任何处理其实是线程不安全的，各个线程都在操作同一个变量，显然是不行的，并且我们也知道volatile这个关键字也是不能保证线程安全的。那么在有一种情况之下，我们需要满足这样一个条件：变量是同一个，但是每个线程都使用同一个初始值，也就是使用同一个变量的一个新的副本。这种情况之下ThreadLocal就非常使用，比如说DAO的数据库连接，我们知道DAO是单例的，那么他的属性Connection就不是一个线程安全的变量。而我们每个线程都需要使用他，并且各自使用各自的。这种情况，ThreadLocal就比较好的解决了这个问题。

　　我们从源码的角度来分析这个问题。

　　首先定义一个ThreadLocal：

public class ConnectionUtil {

private static ThreadLocal<Connection> tl = new ThreadLocal<Connection>();

private static Connection initConn = null;

static {

try {

initConn = DriverManager.getConnection("url, name and password");

} catch (SQLException e) {

e.printStackTrace();

}

}

public Connection getConn() {

Connection c = tl.get();

if(null == c) tl.set(initConn);

return tl.get();

}

}

　　这样子，都是用同一个连接，但是每个连接都是新的，是同一个连接的副本。

　　那么实现机制是如何的呢？

　　1、每个Thread对象内部都维护了一个ThreadLocalMap这样一个ThreadLocal的Map，可以存放若干个ThreadLocal。

|  |  |
| --- | --- |
| 1  2  3 | /\* ThreadLocal values pertaining to this thread. This map is maintained   \* by the ThreadLocal class. \*/  ThreadLocal.ThreadLocalMap threadLocals = null; |

　　2、当我们在调用get()方法的时候，先获取当前线程，然后获取到当前线程的ThreadLocalMap对象，如果非空，那么取出ThreadLocal的value，否则进行初始化，初始化就是将initialValue的值set到ThreadLocal中。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | public T get() {      Thread t = Thread.currentThread();      ThreadLocalMap map = getMap(t);      if (map != null) {          ThreadLocalMap.Entry e = map.getEntry(this);          if (e != null)              return (T)e.value;      }      return setInitialValue();  } |

　　3、当我们调用set()方法的时候，很常规，就是将值设置进ThreadLocal中。

　　4、总结：当我们调用get方法的时候，其实每个当前线程中都有一个ThreadLocal。每次获取或者设置都是对该ThreadLocal进行的操作，是与其他线程分开的。

　　5、应用场景：当很多线程需要多次使用同一个对象，并且需要该对象具有相同初始化值的时候最适合使用ThreadLocal。

　　6、其实说再多也不如看一下源码来得清晰。如果要看源码，其中涉及到一个WeakReference和一个Map，这两个地方需要了解下，这两个东西分别是a.Java的弱引用，也就是GC的时候会销毁该引用所包裹(引用)的对象，这个threadLocal作为key可能被销毁，但是只要我们定义成他的类不卸载，tl这个强引用就始终引用着这个ThreadLocal的，永远不会被gc掉。b.和HashMap差不多。

　　事实上，从本质来讲，就是每个线程都维护了一个map，而这个map的key就是threadLocal，而值就是我们set的那个值，每次线程在get的时候，都从自己的变量中取值，既然从自己的变量中取值，那肯定就不存在线程安全问题，总体来讲，ThreadLocal这个变量的状态根本没有发生变化，他仅仅是充当一个key的角色，另外提供给每一个线程一个初始值。如果允许的话，我们自己就能实现一个这样的功能，只不过恰好JDK就已经帮我们做了这个事情。

## 1　ThreadLocal API

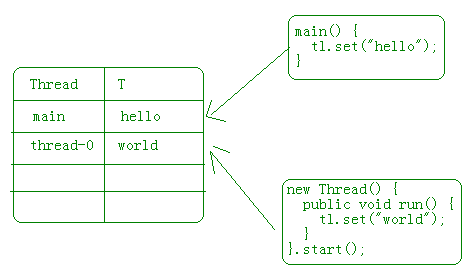
ThreadLocal类只有三个方法：

* void set(T value)：保存值；
* T get()：获取值；
* void remove()：移除值。

## 2　ThreadLocal的内部是Map

ThreadLocal内部其实是个Map来保存数据。虽然在使用ThreadLocal时只给出了值，没有给出键，其实它内部使用了当前线程做为键。

|  |
| --- |
| **class** MyThreadLocal<T> {  **private** Map<Thread,T> map = **new** HashMap<Thread,T>();  **public** **void** set(T value) {  map.put(Thread.*currentThread*(), value);  }    **public** **void** remove() {  map.remove(Thread.*currentThread*());  }    **public** T get() {  **return** map.get(Thread.*currentThread*());  }  } |



# BaseServlet

## 1　BaseServlet的作用

在开始客户管理系统之前，我们先写一个工具类：BaseServlet。

我们知道，写一个项目可能会出现N多个Servlet，而且一般一个Servlet只有一个方法（doGet或doPost），如果项目大一些，那么Servlet的数量就会很惊人。

为了避免Servlet的“膨胀”，我们写一个BaseServlet。它的作用是让一个Servlet可以处理多种不同的请求。不同的请求调用Servlet的不同方法。我们写好了BaseServlet后，让其他Servlet继承BaseServlet，例如CustomerServlet继承BaseServlet，然后在CustomerServlet中提供add()、update()、delete()等方法，每个方法对应不同的请求。

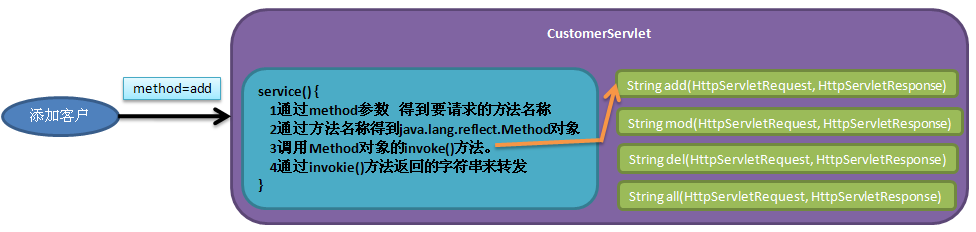


## 2　BaseServlet分析

我们知道，Servlet中处理请求的方法是service()方法，这说明我们需要让service()方法去调用其他方法。例如调用add()、mod()、del()、all()等方法！具体调用哪个方法需要在请求中给出方法名称！然后service()方法通过方法名称来调用指定的方法。

无论是点击超链接，还是提交表单，请求中必须要有method参数，这个参数的值就是要请求的方法名称，这样BaseServlet的service()才能通过方法名称来调用目标方法。例如某个链接如下：

<a href=”/xxx/CustomerServlet?method=add”>添加客户</a>



## 3　BaseServlet代码

|  |
| --- |
| **public** **class** BaseServlet **extends** HttpServlet {  /\*  \* 它会根据请求中的m，来决定调用本类的哪个方法  \*/  **protected** **void** service(HttpServletRequest req, HttpServletResponse res)  **throws** ServletException, IOException {  req.setCharacterEncoding("UTF-8");  res.setContentType("text/html;charset=utf-8");  // 例如：http://localhost:8080/demo1/xxx?m=add  String methodName = req.getParameter("method");// 它是一个方法名称    // 当没用指定要调用的方法时，那么默认请求的是execute()方法。  **if**(methodName == **null** || methodName.isEmpty()) {  methodName = "execute";  }  Class c = **this**.getClass();  **try** {  // 通过方法名称获取方法的反射对象  Method m = c.getMethod(methodName, HttpServletRequest.**class**,  HttpServletResponse.**class**);  // 反射方法目标方法，也就是说，如果methodName为add，那么就调用add方法。  String result = (String) m.invoke(**this**, req, res);  // 通过返回值完成请求转发  **if**(result != **null** && !result.isEmpty()) {  req.getRequestDispatcher(result).forward(req, res);  }  } **catch** (Exception e) {  **throw** **new** ServletException(e);  }  }  } |

|  |
| --- |
| package com.BaseServlet;  import java.io.IOException;  import java.lang.reflect.Method;  import javax.servlet.ServletException;  import javax.servlet.annotation.WebServlet;  import javax.servlet.http.HttpServlet;  import javax.servlet.http.HttpServletRequest;  import javax.servlet.http.HttpServletResponse;  /\*\*  \* 为了避免Servlet的“膨胀”，我们写一个BaseServlet。它的作用是让一个Servlet可以处  \* 理多种不同的请求。不同的请求调用Servlet的不同方法。  \* @author Yorick  \*/  @WebServlet("/BaseServlet")  public class BaseServlet extends HttpServlet {    protected void service(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {  //设置接受和响应字符集  request.setCharacterEncoding("utf-8");  response.setContentType("text/html;charset=utf-8");  //获取方法名称  String methodName = request.getParameter("method");  System.out.println(methodName);  // 当没用指定要调用的方法时，那么默认请求的是execute()方法。  if(methodName==null||methodName.isEmpty()){  throw new RuntimeException("您必须传递要调用的方法名称，通过method参数来完成！");  }  //得到Method  Class c= this.getClass();  System.out.println(c);  Method method = null;  try {  method = c.getMethod(methodName, HttpServletRequest.class,HttpServletResponse.class);  System.out.println(method);    } catch (NoSuchMethodException | SecurityException e) {  throw new RuntimeException("您要调用的方法：" + methodName + "不存在！", e);  }  /\* 调用method的invoke()  \*/  //对象.方法(参数)  //method.invoke(对象, 参数)  try {  String path = (String) method.invoke(this,request,response);  if(path==null||path.trim().isEmpty()){//如果请求处理方法返回的是null或空，那么表示不重定向也不转发  return;  }  /\*  \* path就是login或regist方法返回的是字符串  \*/  /\*  \* path中有前缀，我们使用冒号分隔字符串，得到前缀和后缀  \* 其中前缀说明了转发还是重定向，后缀说明了要转发或重定向的路径  \* 如果没有提前缀，表示转发  \*/  int index = path.indexOf(":");  if(index==-1){//如果没有冒号，说明没有前缀，那么就是转发  request.getRequestDispatcher(path).forward(request, response);  }else {  String s1 = path.substring(0, index);  String s2 = path.substring(index+1);  if(s1.equalsIgnoreCase("f")){//前缀是f，那么说明要转发  request.getRequestDispatcher(s2).forward(request, response);  }else if (s1.equalsIgnoreCase("r")) {//前缀是r，那么说明要重定向  if(s2.toLowerCase().startsWith("http://")){//是其他项目  response.sendRedirect(s2);  }else {//本项目内容  response.sendRedirect(request.getContextPath()+s2);  }  }else {//抛出异常，说明前缀是错误的！  throw new RuntimeException("你指定的操作：" + s1 + ", 当前还不支持！");  }  }  } catch (Exception e) {  throw new RuntimeException("您调用的方法：" + methodName + "抛出了异常！", e);  }  }  }  package com.BaseServlet;  import javax.servlet.annotation.WebServlet;  import javax.servlet.http.HttpServlet;  import javax.servlet.http.HttpServletRequest;  import javax.servlet.http.HttpServletResponse;  /\*\*  \* Servlet implementation class UserServlet  \*/  @WebServlet("/UserServlet")  public class UserServlet extends BaseServlet {  //登录  public String login(HttpServletRequest request,HttpServletResponse response){  System.out.println("login().....");  throw new RuntimeException("login方法内部异常！");  }  //注册  public String regist(HttpServletRequest request,HttpServletResponse response){  System.out.println("regist()......");  return "R:http://www.baidu.com";  }  } |

# DBUtils

## 1　DBUtils简介

DBUtils是Apache Commons组件中的一员，开源免费！

DBUtils是对JDBC的简单封装，但是它还是被很多公司使用！

DBUtils的Jar包：dbutils.jar

## 2　DBUtils主要类

* DbUtils：都是静态方法，一系列的close()方法；
* QueryRunner：
* update()：执行insert、update、delete；
* query()：执行select语句；
* batch()：执行批处理。

## 3　QueryRunner之更新

QueryRunner的update()方法可以用来执行insert、update、delete语句。

1. 创建QueryRunner

构造器：QueryRunner();

1. update()方法

int update(Connection con, String sql, Object… params)

|  |
| --- |
| @Test  **public** **void** fun1() **throws** SQLException {  QueryRunner qr = **new** QueryRunner();  String sql = "insert into user values(?,?,?)";  qr.update(JdbcUtils.*getConnection*(), sql, "u1", "zhangSan", "123");  } |

还有另一种方式来使用QueryRunner

1. 创建QueryRunner

构造器：QueryRunner(DataSource)

1. update()方法

int update(String sql, Object… params)

这种方式在创建QueryRunner时传递了连接池对象，那么在调用update()方法时就不用再传递Connection了。

|  |
| --- |
| @Test  **public** **void** fun2() **throws** SQLException {  QueryRunner qr = **new** QueryRunner(JdbcUtils.*getDataSource*());  String sql = "insert into user values(?,?,?)";  qr.update(sql, "u1", "zhangSan", "123");  } |

## 4　ResultSetHandler

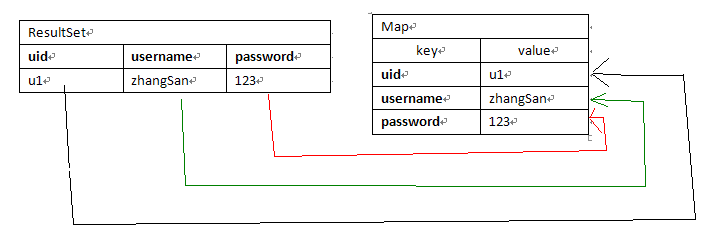
我们知道在执行select语句之后得到的是ResultSet，然后我们还需要对ResultSet进行转换，得到最终我们想要的数据。你可以希望把ResultSet的数据放到一个List中，也可能想把数据放到一个Map中，或是一个Bean中。

DBUtils提供了一个接口ResultSetHandler，它就是用来ResultSet转换成目标类型的工具。你可以自己去实现这个接口，把ResultSet转换成你想要的类型。

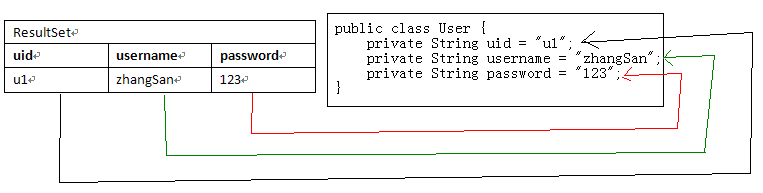
DBUtils提供了很多个ResultSetHandler接口的实现，这些实现已经基本够用了，我们通常不用自己去实现ResultSet接口了。

* MapHandler：单行处理器！把结果集转换成Map<String,Object>，其中列名为键！
* MapListHandler：多行处理器！把结果集转换成List<Map<String,Object>>；
* BeanHandler：单行处理器！把结果集转换成Bean，该处理器需要Class参数，即Bean的类型；
* BeanListHandler：多行处理器！把结果集转换成List<Bean>；
* ColumnListHandler：多行单列处理器！把结果集转换成List<Object>，使用ColumnListHandler时需要指定某一列的名称或编号，例如：new ColumListHandler(“name”)表示把name列的数据放到List中。
* ScalarHandler：单行单列处理器！把结果集转换成Object。一般用于聚集查询，例如select count(\*) from tab\_student。

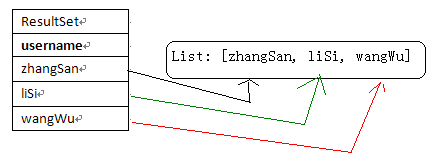
Map处理器



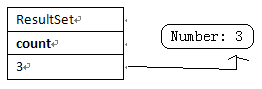
Bean处理器



Column处理器



Scalar处理器



## 5　QueryRunner之查询

QueryRunner的查询方法是：

public <T> T query(String sql, ResultSetHandler<T> rh, Object… params)

public <T> T query(Connection con, String sql, ResultSetHandler<T> rh, Object… params)

query()方法会通过sql语句和params查询出ResultSet，然后通过rh把ResultSet转换成对应的类型再返回。

|  |
| --- |
| @Test  **public** **void** fun1() **throws** SQLException {  DataSource ds = JdbcUtils.*getDataSource*();  QueryRunner qr = **new** QueryRunner(ds);  String sql = "select \* from tab\_student where number=?";  //把一行记录转换成一个Map，其中键为列名称，值为列值  Map<String,Object> map = qr.query(sql, **new** MapHandler(), "S\_2000");  System.*out*.println(map);  }    @Test  **public** **void** fun2() **throws** SQLException {  DataSource ds = JdbcUtils.*getDataSource*();  QueryRunner qr = **new** QueryRunner(ds);  String sql = "select \* from tab\_student";  //把转换集转换成List<Map>，其中每个Map对应一行记录  List<Map<String,Object>> list = qr.query(sql, **new** MapListHandler());  **for**(Map<String,Object> map : list) {  System.*out*.println(map);  }  }    @Test  **public** **void** fun3() **throws** SQLException {  DataSource ds = JdbcUtils.*getDataSource*();  QueryRunner qr = **new** QueryRunner(ds);  String sql = "select \* from tab\_student where number=?";  //把结果集转换成一个Bean对象，在使用BeanHandler时需要指定Class，即Bean的类型  Student stu = qr.query(sql, **new** BeanHandler<Student>(Student.**class**), "S\_2000");  System.*out*.println(stu);  }    @Test  **public** **void** fun4() **throws** SQLException {  DataSource ds = JdbcUtils.*getDataSource*();  QueryRunner qr = **new** QueryRunner(ds);  String sql = "select \* from tab\_student";  //把结果集转换成List<Bean>，其中每个Bean对应一行记录  List<Student> list = qr.query(sql, **new** BeanListHandler<Student>(Student.**class**));  **for**(Student stu : list) {  System.*out*.println(stu);  }  }    @Test  **public** **void** fun5() **throws** SQLException {  DataSource ds = JdbcUtils.*getDataSource*();  QueryRunner qr = **new** QueryRunner(ds);  String sql = "select \* from tab\_student";  //多行单例处理器，即获取name列数据  List<Object> list = qr.query(sql, **new** ColumnListHandler("name"));  **for**(Object s : list) {  System.*out*.println(s);  }  }    @Test  **public** **void** fun6() **throws** SQLException {  DataSource ds = JdbcUtils.*getDataSource*();  QueryRunner qr = **new** QueryRunner(ds);  String sql = "select count(\*) from tab\_student";  //单行单列处理器，一般用于聚合查询，在使用ScalarHandler时可以指定列名，如果不指定，默认为第1列。  Number number = (Number)qr.query(sql, **new** ScalarHandler());  //对聚合函数的查询结果，有的驱动返回的是Long，有的返回的是BigInteger，所以这里我们把它转换成Number，Number是Long和BigInteger的父类！然后我们再调用Number的intValue()或longValue()方法就OK了。  **int** cnt = number.intValue();  System.*out*.println(cnt);  } |

## 5　QueryRunner之批处理

QueryRunner还提供了批处理方法：batch()。

我们更新一行记录时需要指定一个Object[]为参数，如果是批处理，那么就要指定Object[][]为参数了。即多个Object[]就是Object[][]了，其中每个Object[]对应一行记录：

|  |
| --- |
| @Test  **public** **void** fun10() **throws** SQLException {  DataSource ds = JdbcUtils.*getDataSource*();  QueryRunner qr = **new** QueryRunner(ds);  String sql = "insert into tab\_student values(?,?,?,?)";  //注意，这里是二维数组，这个二维数组有10个一维数组。  Object[][] params = **new** Object[10][];//表示 要插入10行记录  **for**(**int** i = 0; i < params.length; i++) {  params[i] = **new** Object[]{"S\_300" + i, "name" + i, 30 + i, i%2==0?"男":"女"};  }  //执行批处理  qr.batch(sql, params);  } |

# Service事务

　　在Service中使用ThreadLocal来完成事务，为将来学习Spring事务打基础！

## 1　DAO中的事务

在DAO中处理事务真是“小菜一碟”。

|  |
| --- |
| public void xxx() {  Connection con = null;  try {  con = JdbcUtils.getConnection();  con.setAutoCommitted(false);  QueryRunner qr = new QueryRunner();  String sql = …;  Object[] params = …;  qr.update(con, sql, params);  sql = …;  Object[] params = …;  qr.update(con, sql, params);  con.commit();  } catch(Exception e) {  try {  if(con != null) {con.rollback();}  } catch(Exception e) {}  } finally {  try {  con.close();  } catch(Exception e) {}  }  } |

## 2　Service才是处理事务的地方

　　我们要清楚一件事，DAO中不是处理事务的地方，因为DAO中的每个方法都是对数据库的一次操作，而Service中的方法才是对应一个业务逻辑。也就是说我们需要在Service中的一方法中调用DAO的多个方法，而这些方法应该在一起事务中。

怎么才能让DAO的多个方法使用相同的Connection呢？方法不能再自己来获得Connection，而是由外界传递进去。

|  |
| --- |
| public void daoMethod1(Connection con, …) {  }  public void daoMethod2(Connection con, …) {  } |

在Service中调用DAO的多个方法时，传递相同的Connection就可以了。

|  |
| --- |
| public class XXXService() {  private XXXDao dao = new XXXDao();  public void serviceMethod() {  Connection con = null;  try {  con = JdbcUtils.getConnection();  con.setAutoCommitted(false);  dao.daoMethod1(con, …);  dao.doaMethod2(con, …);  com.commint();  } catch(Exception e) {  try {  con.rollback();  } catch(Exception e) {}  } finally {  try {  con.close();  } catch(Exception e) {}  }  }  } |

但是，在Service中不应该出现Connection，它应该只在DAO中出现，因为它是JDBC的东西，JDBC的东西是用来连接数据库的，连接数据库是DAO的事儿！！！但是，事务是Service的事儿，不能放到DAO中！！！

## 3　修改JdbcUtils

我们把对事务的开启和关闭放到JdbcUtils中，在Service中调用JdbcUtils的方法来完成事务的处理，但在Service中就不会再出现Connection这一“禁忌”了。

DAO中的方法不用再让Service来传递Connection了。DAO会主动从JdbcUtils中获取Connection对象，这样，JdbcUtils成为了DAO和Service的中介！

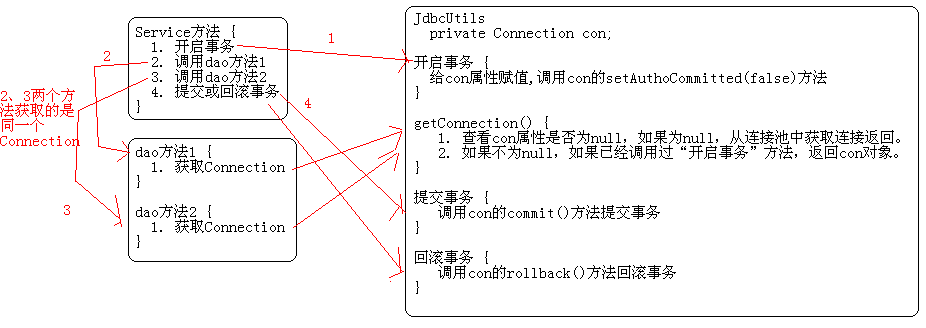
我们在JdbcUtils中添加beginTransaction()和rollbackTransaction()，以及commitTransaction()方法。这样在Service中的代码如下：

|  |
| --- |
| public class XXXService() {  private XXXDao dao = new XXXDao();  public void serviceMethod() {  try {  JdbcUtils.beginTransaction();  dao.daoMethod1(…);  dao.daoMethod2(…);  JdbcUtils.commitTransaction();  } catch(Exception e) {  JdbcUtils.rollbackTransaction();  }  }  } |

DAO

|  |
| --- |
| public void daoMethod1(…) {  Connection con = JdbcUtils.getConnection();  }  public void daoMethod2(…) {  Connection con = JdbcUtils.getConnection();  } |

在Service中调用了JdbcUtils.beginTransaction()方法时，JdbcUtils要做准备好一个已经调用了setAuthCommitted(false)方法的Connection对象，因为在Service中调用JdbcUtils.beginTransaction()之后，马上就会调用DAO的方法，而在DAO方法中会调用JdbcUtils.getConnection()方法。这说明JdbcUtils要在getConnection()方法中返回刚刚准备好的，已经设置了手动提交的Connection对象。



在JdbcUtils中创建一个Connection con属性，当它为null时，说明没有事务！当它不为null时，表示开启了事务。

* 在没有开启事务时，可以调用“开启事务”方法；
* 在开启事务后，可以调用“提交事务”和“回滚事务”方法；
* getConnection()方法会在con不为null时返回con，再con为null时，从连接池中返回连接。

beginTransaction()

判断con是否为null，如果不为null，就抛出异常！

如果con为null，那么从连接池中获取一个Connection对象，赋值给con！然后设置它为“手动提交”。

getConnection()

判断con是否为null，如果为null说明没有事务，那么从连接池获取一个连接返回；

如果不为null，说明已经开始了事务，那么返回con属性返回。这说明在con不为null时，无论调用多少次getConnection()方法，返回的都是同个Connection对象。

commitTransaction()

判断con是否为null，如果为null，说明没有开启事务就提交事务，那么抛出异常；

如果con不为null，那么调用con的commit()方法来提交事务；

调用con.close()方法关闭连接；

con = null，这表示事务已经结束！

rollbackTransaction()

判断con是否为null，如果为null，说明没有开启事务就回滚事务，那么抛出异常；

如果con不为null，那么调用con的rollback()方法来回滚事务；

调用con.close()方法关闭连接；

con = null，这表示事务已经结束！

JdbcUtils.java

|  |
| --- |
| **public** **class** JdbcUtils {  **private** **static** DataSource *dataSource* = **new** ComboPooledDataSource();  **private** **static** Connection *con* = **null**;  **public** **static** DataSource getDataSource() {  **return** *dataSource*;  }  **public** **static** Connection getConnection() **throws** SQLException {  **if**(*con* == **null**) {  **return** *dataSource*.getConnection();  }  **return** *con*;  }    **public** **static** **void** beginTranscation() **throws** SQLException {  **if**(*con* != **null**) {  **throw** **new** SQLException("事务已经开启，在没有结束当前事务时，不能再开启事务！");  }  *con* = *dataSource*.getConnection();  *con*.setAutoCommit(**false**);  }    **public** **static** **void** commitTransaction() **throws** SQLException {  **if**(*con* == **null**) {  **throw** **new** SQLException("当前没有事务，所以不能提交事务！");  }  *con*.commit();  *con*.close();  *con* = **null**;  }    **public** **static** **void** rollbackTransaction() **throws** SQLException {  **if**(*con* == **null**) {  **throw** **new** SQLException("当前没有事务，所以不能回滚事务！");  }  *con*.rollback();  *con*.close();  *con* = **null**;  }  } |

## 4　再次修改JdbcUtils

现在JdbcUtils有个问题，如果有两个线程！第一个线程调用了beginTransaction()方法，另一个线程再调用beginTransaction()方法时，因为con已经不再为null，所以就会抛出异常了。

我们希望JdbcUtils可以多线程环境下被使用！这说明最好的方法是为每个线程提供一个Connection，这样每个线程都可以开启自己的事务了。

还记得ThreadLocal类么？

|  |
| --- |
| **public** **class** JdbcUtils {  **private** **static** DataSource *dataSource* = **new** ComboPooledDataSource();  //tl其实是一个Map，每个线程都可以来保存自己的Connection  **private** **static** ThreadLocal<Connection> *tl* = **new** ThreadLocal<Connection>();  **public** **static** DataSource getDataSource() {  **return** *dataSource*;  }  **public** **static** Connection getConnection() **throws** SQLException {  //获取当前线程的连接  Connection con = *tl*.get();  **if**(con == **null**) {  **return** *dataSource*.getConnection();  }  **return** con;  }    **public** **static** **void** beginTranscation() **throws** SQLException {  //获取当前线程的连接  Connection con = *tl*.get();  //如果当前线程已经有了连接，说明已经开启了事务  **if**(con != **null**) {  **throw** **new** SQLException("事务已经开启，在没有结束当前事务时，不能再开启事务！");  }  //如果当前线程没有连接，说明还没有开启事务，那么就获取一个连接  con = *dataSource*.getConnection();  //设置连接为手动提交  con.setAutoCommit(**false**);  //把连接放到tl中，当前线程就有了连接了。  *tl*.set(con);  }    **public** **static** **void** commitTransaction() **throws** SQLException {  Connection con = *tl*.get();  **if**(con == **null**) {  **throw** **new** SQLException("当前没有事务，所以不能提交事务！");  }  con.commit();  con.close();  *tl*.remove();  }    **public** **static** **void** rollbackTransaction() **throws** SQLException {  Connection con = *tl*.get();  **if**(con == **null**) {  **throw** **new** SQLException("当前没有事务，所以不能回滚事务！");  }  con.rollback();  con.close();  *tl*.remove();  }  } |

## 5　转账示例

|  |
| --- |
| **public** **class** AccountDao {  **public** **void** updateBalance(String name, **double** balance) **throws** SQLException {  String sql = "update account set balance=balance+? where name=?";  Connection con = JdbcUtils.*getConnection*();  QueryRunner qr = **new** QueryRunner();  qr.update(con, sql, balance, name);  }  } |
| **public** **class** AccountService {  **private** AccountDao dao = **new** AccountDao();    **public** **void** transfer(String from, String to, **double** balance) {  **try** {  JdbcUtils.*beginTranscation*();  dao.updateBalance(from, -balance);  dao.updateBalance(to, balance);  JdbcUtils.*commitTransaction*();  } **catch**(Exception e) {  **try** {  JdbcUtils.*rollbackTransaction*();  } **catch** (SQLException e1) {  **throw** **new** RuntimeException(e);  }  }  }  } |
| AccountService as = **new** AccountService();  as.transfer("zs", "ls", 100); |

# JDBC案例--客户关系管理系统

详见项目文档