JDBC

# 一、概述

在Java中，数据库存取技术可分为如下几类：

* JDBC直接访问数据库
* JDO技术（Java Data Object）
* 第三方O/R工具，如Hibernate, Mybatis 等

JDBC是java访问数据库的基石，JDO, Hibernate等只是更好的封装了JDBC。

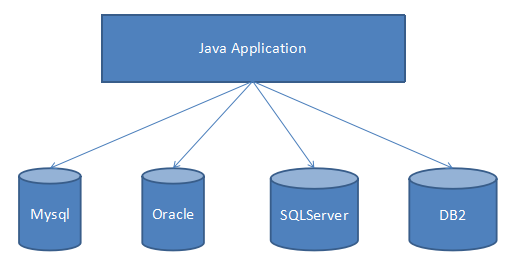
## 1、什么是JDBC

JDBC(Java Database Connectivity)是一个**独立于特定数据库管理系统（DBMS）、通用的SQL数据库存取和操作的公共接口**（一组API），定义了用来访问数据库的标准Java类库，使用这个类库可以以一种标准的方法、方便地访问数据库资源

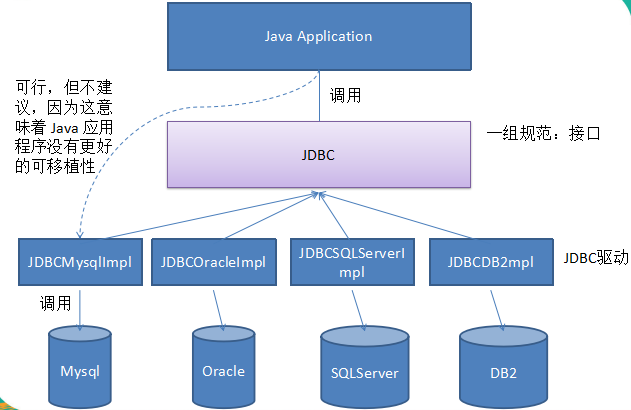
JDBC为访问不同的数据库提供了一种**统一的途径**，为开发者屏蔽了一些细节问题。

JDBC的目标是使Java程序员使用JDBC可以连接任何**提供了JDBC驱动程序**的数据库系统，这样就使得程序员无需对特定的数据库系统的特点有过多的了解，从而大大简化和加快了开发过程。

如果没有JDBC，那么Java程序访问数据库时是这样的：



现在：



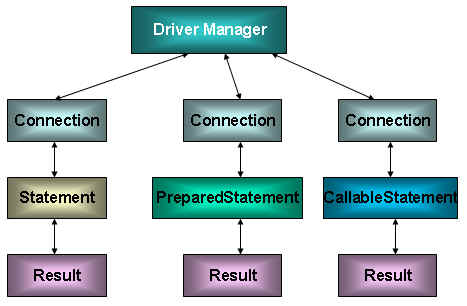
**结论：**

JDBC是SUN公司（Oracle公司甲骨文）提供一套用于数据库操作的接口API，Java程序员只需要面向这套接口编程即可。

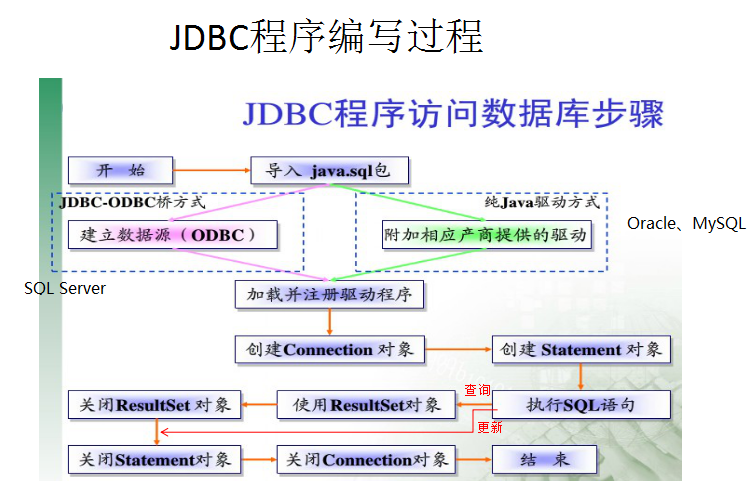
不同的数据库厂商，需要针对这套接口，提供不同实现。不同的实现的集合，即为不同数据库的驱动。

## 2、JDBC API

JDBC API是一系列的接口，它统一和规范了应用程序与数据库的连接、执行SQL语句，并到得到返回结果等各类操作。声明在java.sql与javax.sql包中。



## 3、JDBC程序编写步骤



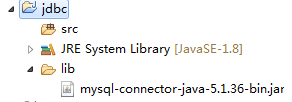
# 二、获取数据库连接

## （一）引入JDBC驱动程序

驱动程序由数据库提供商提供下载。 MySQL的驱动下载地址：http://dev.mysql.com/downloads/

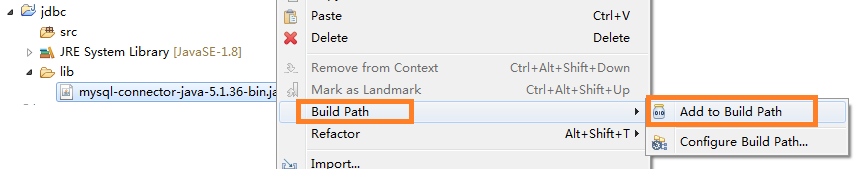
如何在Java Project项目应用中添加数据库驱动jar：

* （1）把拷贝到项目中一个目录中

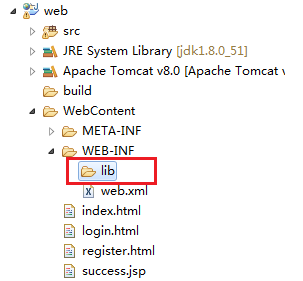


* （2）添加到项目的类路径下

在驱动jar上右键-->Build Path-->Add to Build Path



**注意：**如果是Dynamic Web Project（动态的web项目）话，则是把驱动jar放到WebContent（有的开发工具叫WebRoot）目录中的WEB-INF目录中的lib目录下即可



## （二）加载并注册驱动

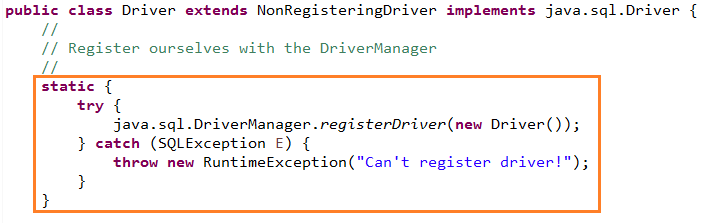
加载并注册驱动：

加载驱动，把驱动类加载到内存

注册驱动，把驱动类的对象交给DriverManager管理，用于后面创建连接等使用。

#### 1、Class.forName()

因为 Driver 接口的驱动程序类都包含了静态代码块，在这个静态代码块中，会调用 DriverManager.registerDriver() 方法来注册自身的一个实例，所以可以换一种方式来加载驱动。（即只要想办法让驱动类的这段静态代码块执行即可注册驱动类，而要让这段静态代码块执行，只要让该类被类加载器加载即可）

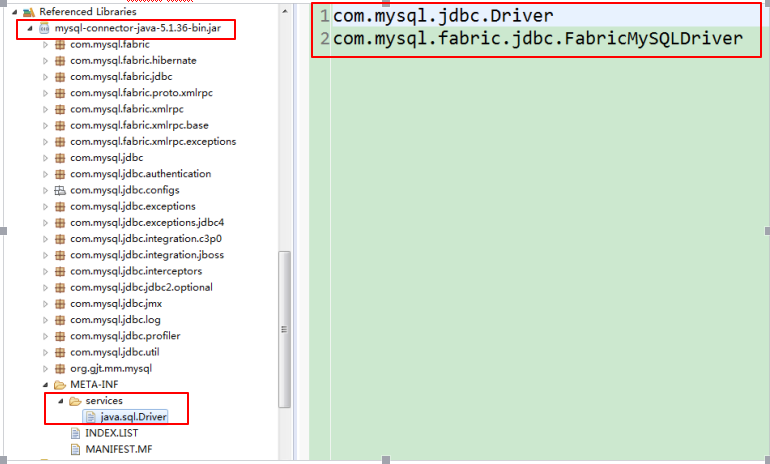


调用 Class 类的静态方法 forName()，向其传递要加载的 JDBC 驱动的类名

|  |
| --- |
| //通过反射，加载与注册驱动类，**解耦合（不直接依赖）**  Class.forName("com.mysql.jdbc.Driver"); |

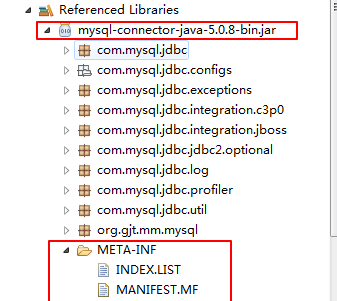
#### 2、服务提供者框架（例如：JDBC的驱动程序）自动注册（有版本要求）

符合JDBC 4.0规范的驱动程序包含了一个文件META-INF/services/java.sql.Driver，在这个文件中提供了JDBC驱动实现的类名。例如：mysql-connector-java-5.1.40-bin.jar文件中就可以找到java.sql.Driver文件，用文本编辑器打开文件就可以看到：com.mysql.jdbc.Driver类。



JVM的服务提供者框架在启动应用时就会注册服务，例如：MySQL的JDBC驱动就会被注册，而原代码中的Class.forName("com.mysql.jdbc.Driver")仍然可以存在，但是不会起作用。

**但是注意mysql-connector-java-5.0.8-bin.jar版本的jar中没有，如下**



## （三）获取数据库链接

可以通过 DriverManager 类建立到数据库的连接Connection：

DriverManager 试图从**已注册**的 JDBC 驱动程序集中选择一个**适当的**驱动程序。

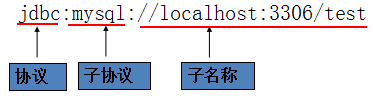
* public static Connection getConnection(String url)
* **public static Connection getConnection(String url,String user, String password)**
* public static Connection getConnection(String url,Properties info)其中Properties info通常至少应该包括 "user" 和 "password" 属性

JDBC URL 用于标识一个被注册的驱动程序，驱动程序管理器通过这个 URL 选择正确的驱动程序，从而建立到数据库的连接。JDBC URL的标准由三部分组成，各部分间用冒号分隔。

jdbc:<子协议>:<子名称>

* 协议：JDBC URL中的协议总是jdbc
* 子协议：子协议用于标识一个数据库驱动程序
* 子名称：一种标识数据库的方法。子名称可以依不同的子协议而变化，用子名称的目的是为了**定位数据库**提供足够的信息

例如：



* MySQL的连接URL编写方式：
  + jdbc:mysql://主机名称:mysql服务端口号/数据库名称?参数=值&参数=值
  + jdbc:mysql://localhost:3306/testdb
  + jdbc:mysql://localhost:3306/testdb**?useUnicode=true&characterEncoding=utf8（**如果JDBC程序与服务器端的字符集不一致，会导致乱码，那么可以通过参数指定服务器端的字符集**）**
  + jdbc:mysql://localhost:3306/testdb?user=root&password=123456
* Oracle9i:
  + jdbc:oracle:thin:@主机名称:oracle服务端口号:数据库名称
  + jdbc:oracle:thin:@localhost:1521:testdb
* SQLServer
  + jdbc:sqlserver://主机名称:sqlserver服务端口号:DatabaseName=数据库名称
  + jdbc:sqlserver://localhost:1433:DatabaseName=testdb

|  |
| --- |
| //1、加载与注册驱动  Class.forName("**com.mysql.jdbc.Driver**");    //2、获取数据库连接  String url = "**jdbc:mysql://localhost:3306/test**";  Connection conn = DriverManager.getConnection(url, "**root**", "**root**"); |

## （四）操作或访问数据库

数据库连接被用于向数据库服务器发送命令和 SQL 语句，并接受数据库服务器返回的结果。

其实一个数据库连接就是一个Socket连接。

在 java.sql 包中有 3 个接口分别定义了对数据库的调用的不同方式：

* Statement：用于执行静态 SQL 语句并返回它所生成结果的对象。
  + PrepatedStatement：SQL 语句被预编译并存储在此对象中，然后可以使用此对象多次高效地执行该语句。
    - CallableStatement：用于执行 SQL 存储过程

### Statement

通过调用 Connection 对象的 createStatement() 方法创建该对象

该对象用于执行静态的 SQL 语句，并且返回执行结果

Statement 接口中定义了下列方法用于执行 SQL 语句：

int excuteUpdate(String sql)：执行更新操作INSERT、UPDATE、DELETE

ResultSet excuteQuery(String sql)：执行查询操作SELECT

### ResultSet

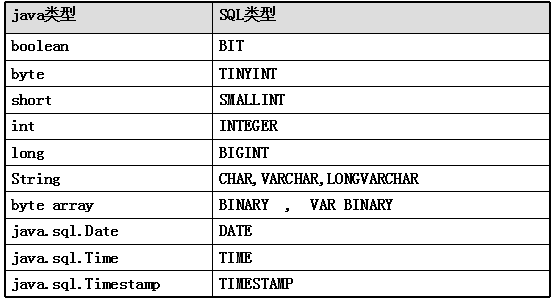
通过调用 Statement 对象的 excuteQuery() 方法创建该对象

ResultSet 对象以逻辑表格的形式封装了执行数据库操作的结果集，ResultSet 接口由数据库厂商实现

ResultSet 对象维护了一个指向当前数据行的**游标**，初始的时候，游标在第一行之前，可以通过 ResultSet 对象的 next() 方法移动到下一行

ResultSet 接口的常用方法：

* boolean next()
* getXxx(String columnLabel)：columnLabel使用 SQL AS 子句指定的列标签。如果未指定 SQL AS 子句，则标签是列名称
* getXxx(int index) :索引从1开始
* …



## （五）释放资源

Connection、Statement、ResultSet都是应用程序和数据库服务器的连接资源，使用后一定要关闭，可以在finally中关闭

演示未关闭后果：

|  |
| --- |
| **package** com.atguigu.conn;  **import** java.sql.Connection;  **import** java.sql.DriverManager;  **public** **class** TestConnectionClose {  **public** **static** **void** main(String[] args) **throws** Exception{  //1、加载与注册驱动  Class.*forName*("com.mysql.jdbc.Driver");    //2、获取数据库连接  String url = "jdbc:mysql://localhost:3306/test";    //my.ini中max\_connections=10  **for** (**int** i = 0; i < 15; i++) {  Connection conn = DriverManager.*getConnection*(url,"root", "123456");  System.***out***.println(conn);  //没有关闭，资源一直没有释放  }  }  } |

## （六）增、删、改、查示例代码

|  |
| --- |
| **package** com.atguigu.statement;  **import** java.sql.Connection;  **import** java.sql.DriverManager;  **import** java.sql.ResultSet;  **import** java.sql.Statement;  **import** org.junit.Test;  /\*  \* 网络编程：tcp  \*  \* 服务器端：  \* 1、ServerSocket server = new ServerSocket(3306);  \* 2、Socket socket = server.accept();  \* 3、InputStream input = socket.getInputStream();//接收sql，客户端传过来的  \* 4、在服务器执行sql  \* 5、把结果给客户端  \*  \* 客户端：  \* 1、Socket socket = new Socket(服务器的IP地址，3306);  \* 2、传sql  \* 3、OutputStream out = socket.getOutputStream();  \* 4、out.write(sql);  \* 5、接收结果  \* 6、断开连接 out.close();socket.close();  \*/  **public** **class** TestStatement {  @Test  **public** **void** testAdd()**throws** Exception{  Class.*forName*("com.mysql.jdbc.Driver");    String url = "jdbc:mysql://localhost:3306/test";  String user = "root";  String password = "123456";  Connection conn = DriverManager.*getConnection*(url, user, password);    Statement st = conn.createStatement();    String sql = "INSERT INTO t\_department(dname,description) VALUES('财务部','负责发钱工作')";  **int** len = st.executeUpdate(sql);//把insert,update,delete都用这个方法    **if**(len>0){  System.***out***.println("添加成功");  }**else**{  System.***out***.println("添加失败");  }    st.close();  conn.close();  }    @Test  **public** **void** testUpdate()**throws** Exception{  Class.*forName*("com.mysql.jdbc.Driver");    String url = "jdbc:mysql://localhost:3306/test";  String user = "root";  String password = "123456";  Connection conn = DriverManager.*getConnection*(url, user, password);    Statement st = conn.createStatement();    String sql = "UPDATE t\_department SET description = '负责发工资、社保、公积金工作' WHERE dname ='财务部'";  **int** len = st.executeUpdate(sql);//把insert,update,delete都用这个方法    **if**(len>0){  System.***out***.println("修改成功");  }**else**{  System.***out***.println("修改失败");  }    st.close();  conn.close();  }    @Test  **public** **void** testDelete()**throws** Exception{  Class.*forName*("com.mysql.jdbc.Driver");    String url = "jdbc:mysql://localhost:3306/test";  String user = "root";  String password = "123456";  Connection conn = DriverManager.*getConnection*(url, user, password);    Statement st = conn.createStatement();  String sql = "DELETE FROM t\_department WHERE did =6";  **int** len = st.executeUpdate(sql);//把insert,update,delete都用这个方法    **if**(len>0){  System.***out***.println("删除成功");  }**else**{  System.***out***.println("删除失败");  }    st.close();  conn.close();  }    @Test  **public** **void** testSelect()**throws** Exception{  Class.*forName*("com.mysql.jdbc.Driver");    String url = "jdbc:mysql://localhost:3306/test";  String user = "root";  String password = "123456";  Connection conn = DriverManager.*getConnection*(url, user, password);    Statement st = conn.createStatement();    String sql = "SELECT \* FROM t\_department";  ResultSet rs = st.executeQuery(sql);//select语句用query方法  **while**(rs.next()){//是否有下一行  //取这一行的单元格  **int** id = rs.getInt(1);  String name = rs.getString(2);  String desc = rs.getString(3);    System.***out***.println(id+"\t" + name + "\t" + desc);  }    rs.close();  st.close();  conn.close();    }    @Test  **public** **void** testSelect2()**throws** Exception{  Class.*forName*("com.mysql.jdbc.Driver");    String url = "jdbc:mysql://localhost:3306/test";  String user = "root";  String password = "123456";  Connection conn = DriverManager.*getConnection*(url, user, password);    Statement st = conn.createStatement();  String sql = "SELECT did,dname FROM t\_department";  ResultSet rs = st.executeQuery(sql);//select语句用query方法  **while**(rs.next()){//是否有下一行  //取这一行的单元格  **int** id = rs.getInt("did");  String name = rs.getString("dname");  System.***out***.println(id+"\t" + name);  }    rs.close();  st.close();  conn.close();  }  } |

# 四、PreparedStatement

## 1、Statement的不足

### （1）SQL拼接

### （2）SQL注入

SQL 注入是利用某些系统没有对用户输入的数据进行充分的检查，而在用户输入数据中注入非法的 SQL 语句段或命令，从而利用系统的 SQL 引擎完成恶意行为的做法。对于 Java 而言，要防范 SQL 注入，只要用 PreparedStatement 取代 Statement 就可以了。

### （3）处理Blob类型的数据

BLOB (binary large object)，二进制大对象，BLOB常常是数据库中用来存储二进制文件的字段类型。

插入BLOB类型的数据必须使用PreparedStatement，因为BLOB类型的数据无法使用字符串拼接写的。

MySQL的四种BLOB类型(除了在存储的最大信息量上不同外，他们是等同的)

|  |  |
| --- | --- |
|  | 如果还是报错：xxx too large，那么在mysql的安装目录下，找my.ini文件加上如下的配置参数：  max\_allowed\_packet=16M    注意：修改了my.ini文件，一定要重新启动服务 |

实际使用中根据需要存入的数据大小定义不同的BLOB类型。  
需要注意的是：如果存储的文件过大，数据库的性能会下降。

|  |
| --- |
| CREATE TABLE `user` (  `id` int(11) NOT NULL AUTO\_INCREMENT,  `username` varchar(20) COLLATE utf8\_unicode\_ci DEFAULT NULL,  `head\_picture` **mediumblob**,  PRIMARY KEY (`id`)  ) ENGINE=InnoDB AUTO\_INCREMENT=3 DEFAULT CHARSET=utf8 COLLATE=utf8\_unicode\_ci; |
| **package** com.atguigu.statement;  **import** java.sql.Connection;  **import** java.sql.DriverManager;  **import** java.sql.ResultSet;  **import** java.sql.Statement;  **import** java.util.Scanner;  **import** org.junit.Test;  /\*  \* Statement：  \* 1、SQL拼接  \* 2、SQL注入  \* 3、处理不了Blob类型的数据  \*/  **public** **class** TestStatementProblem {    @Test  **public** **void** add() **throws** Exception{  Scanner input = **new** Scanner(System.***in***);  System.***out***.println("请输入姓名：");  String name = input.nextLine();    System.***out***.println("请输入领导编号：");  **int** mid = input.nextInt();    System.***out***.println("请输入部门编号：");  **int** did = input.nextInt();    //1、连接数据库  Class.*forName*("com.mysql.jdbc.Driver");    String url = "jdbc:mysql://localhost:3306/1221db";  String user = "root";  String password = "123456";  Connection conn = DriverManager.*getConnection*(url, user, password);    //2、创建Statement对象  Statement st = conn.createStatement();    //3、编写sql  String sql = "INSERT INTO emp (ename,`mid`,did) VALUES('" + name+"'," + mid + "," + did + ")";    //4、执行sql  **int** update = st.executeUpdate(sql);  System.***out***.println(update>0?"添加成功":"添加失败");    //5、释放资源  st.close();  conn.close();  }    @Test  **public** **void** select()**throws** Exception{  Scanner input = **new** Scanner(System.***in***);  System.***out***.println("请输入姓名：");  String name = input.nextLine();    //1、连接数据库  Class.*forName*("com.mysql.jdbc.Driver");    String url = "jdbc:mysql://localhost:3306/1221db";  String user = "root";  String password = "123456";  Connection conn = DriverManager.*getConnection*(url, user, password);    //2、写sql  //孙红雷 ' or '1' = '1  String sql = "SELECT eid,ename,tel,gender,salary FROM t\_employee WHERE ename = '" + name + "'";  System.***out***.println(sql);  // SELECT eid,ename,tel,gender,salary FROM t\_employee WHERE ename = '孙红雷 ' or '1' = '1'    //3、用Statement执行  Statement st = conn.createStatement();    //4、执行查询sql  ResultSet rs = st.executeQuery(sql);  **while**(rs.next()){  **int** id = rs.getInt(1);  String ename = rs.getString(2);  String tel = rs.getString(3);  String gender =rs.getString(4);  **double** salary = rs.getDouble(5);    System.***out***.println(id+"\t" + ename + "\t" + tel + "\t" + gender + "\t" +salary);  }    //5、释放资源  st.close();  conn.close();  }    @Test  **public** **void** testAddBlob(){  String sql = "INSERT INTO `user` (username,`password`,photo)VALUES('chai','123',没法在String中处理Blob类型的数据);";  }  } |

## 2、PreparedStatement概述

可以通过调用 Connection 对象的 preparedStatement(**String sql**) 方法获取 PreparedStatement 对象

PreparedStatement 接口是 Statement 的子接口，它表示一条预编译过的 SQL 语句

* PreparedStatement 对象所代表的 SQL 语句中的参数用问号(?)来表示，调用 PreparedStatement 对象的 setXxx() 方法来设置这些参数. setXxx() 方法有两个参数，第一个参数是要设置的 SQL 语句中的参数的索引(从 1 开始)，第二个是设置的 SQL 语句中的参数的值
* ResultSet executeQuery()执行查询，并返回该查询生成的 ResultSet 对象。
* int executeUpdate()：执行更新，包括增、删、该

|  |
| --- |
| **package** com.atguigu.preparedstatement;  **import** java.io.FileInputStream;  **import** java.sql.Connection;  **import** java.sql.DriverManager;  **import** java.sql.PreparedStatement;  **import** java.sql.ResultSet;  **import** java.util.Scanner;  **import** org.junit.Test;  /\*  \* PreparedStatement：是Statement子接口  \* 1、SQL不需要拼接  \* 2、SQL不会出现注入  \* 3、可以处理Blob类型的数据  \* tinyblob：255字节以内  \* blob：65K以内  \* mediumblob:16M以内  \* longblob：4G以内  \*  \* 如果还是报错：xxx too large，那么在mysql的安装目录下，找my.ini文件加上如下的配置参数：  \* max\_allowed\_packet=16M  \* 注意：修改了my.ini文件，一定要重新启动服务  \*  \*/  **public** **class** TestPreparedStatement {  @Test  **public** **void** add() **throws** Exception {  Scanner input = **new** Scanner(System.***in***);  System.***out***.println("请输入姓名：");  String name = input.nextLine();    System.***out***.println("请输入手机号码：");  String tel = input.nextLine();    System.***out***.println("请输入性别：");  String gender = input.nextLine();    System.***out***.println("请输入薪资：");  **double** salary = input.nextDouble();    System.***out***.println("请输入部门编号：");  **int** did = input.nextInt();    //1、连接数据库  Class.*forName*("com.mysql.jdbc.Driver");    String url = "jdbc:mysql://localhost:3306/test";  String user = "root";  String password = "123456";  Connection conn = DriverManager.*getConnection*(url, user, password);  //2、编写带？的SQL  String sql = "INSERT INTO t\_employee (ename,tel,gender,salary,did) VALUES(?,?,?,?,?)";    // 3、准备一个PreparedStatement：预编译sql  PreparedStatement pst = conn.prepareStatement(sql);// 对带？的sql进行预编译  // 4、把?用具体的值进行代替  pst.setString(1, name);  pst.setString(2, tel);  pst.setString(3, gender);  pst.setDouble(4, salary);  pst.setInt(5, did);  // 5、执行sql  **int** len = pst.executeUpdate();  System.***out***.println(len>0?"添加成功":"添加失败");  // 6、释放资源  pst.close();  conn.close();  }  @Test  **public** **void** select() **throws** Exception {  Scanner input = **new** Scanner(System.***in***);  System.***out***.println("请输入姓名：");  String name = input.nextLine();    //1、连接数据库  Class.*forName*("com.mysql.jdbc.Driver");    String url = "jdbc:mysql://localhost:3306/test";  String user = "root";  String password = "123456";  Connection conn = DriverManager.*getConnection*(url, user, password);  //2、编写带?的sql  //孙红雷 ' or '1' = '1  String sql = "SELECT eid,ename,tel,gender,salary FROM t\_employee WHERE ename = ?";    // 3、把带？的sql语句进行预编译  PreparedStatement pst = conn.prepareStatement(sql);  // 4、把？用具体的变量的赋值  pst.setString(1, name);  // 5、执行sql  ResultSet rs = pst.executeQuery();  **while** (rs.next()) {  **int** id = rs.getInt("eid");  String ename = rs.getString("ename");  String tel = rs.getString("tel");  String gender = rs.getString("gender");  **double** salary = rs.getDouble("salary");  System.***out***.println(id + "\t" + ename + "\t" + tel + "\t" + gender + "\t" + salary);  }  // 6、释放资源  rs.close();  pst.close();  conn.close();  }  @Test  **public** **void** addBlob() **throws** Exception {  Scanner input = **new** Scanner(System.***in***);  System.***out***.println("请输入用户名：");  String username = input.nextLine();  System.***out***.println("请输入密码：");  String pwd = input.nextLine();  System.***out***.println("请指定照片的路径：");  String photoPath = input.nextLine();  //1、连接数据库  Class.*forName*("com.mysql.jdbc.Driver");    String url = "jdbc:mysql://localhost:3306/test";  String user = "root";  String password = "123456";  Connection conn = DriverManager.*getConnection*(url, user, password);    //2、 INSERT INTO `user` VALUES(NULL,用户名,密码,照片)  String sql = "INSERT INTO `user` (username,`password`,head\_picture)VALUES(?,?,?)";  // 3、准备一个PreparedStatement：预编译sql  PreparedStatement pst = conn.prepareStatement(sql);// 对带？的sql进行预编译  // 4、对？进行设置  pst.setString(1, username);  pst.setString(2, pwd);  pst.setBlob(3, **new** FileInputStream(photoPath));  // 5、执行sql  **int** len = pst.executeUpdate();  System.***out***.println(len > 0 ? "添加成功" : "添加失败");  // 6、释放资源  pst.close();  conn.close();  }  } |

## **3、PreparedStatement vs Statement**

* 代码的可读性和可维护性. Statement的sql拼接是个难题。
* PreparedStatement 可以防止 SQL 注入
* PreparedStatement 可以处理Blob类型的数据
* PreparedStatement 能最大可能提高性能：**（Oracle和PostgreSQL8是这样，但是对于MySQL不一定比Statement高）**
  + DBServer会对预编译语句提供性能优化。因为预编译语句有可能被重复调用，所以语句在被DBServer的编译器编译后的执行代码被缓存下来，那么下次调用时只要是相同的预编译语句就不需要编译，只要将参数直接传入编译过的语句执行代码中就会得到执行。

## 4、JDBC 取得数据库自动生成的主键

获取自增长的键值：

（1）在创建PreparedStatement对象时

原来：

PreparedStatement pst = conn.preparedStatement(sql);

现在：

PreparedStatement pst = conn.prepareStatement(orderInsert,Statement.RETURN\_GENERATED\_KEYS);

（2）原来执行更新

原来：

int len = pst.executeUpdate();

现在：

int len = pst.executeUpdate();

ResultSet rs = pst.getGeneratedKeys();

if(rs.next()){

Object key = rs.getObject(第几列);//获取自增长的键值

}

|  |
| --- |
| **package** com.atguigu.preparedstatement;  **import** java.sql.Connection;  **import** java.sql.DriverManager;  **import** java.sql.PreparedStatement;  **import** java.sql.ResultSet;  **import** java.sql.Statement;  **import** java.util.Scanner;  **public** **class** TestGenerateKey {  **public** **static** **void** main(String[] args) **throws** Exception{  Scanner input = **new** Scanner(System.***in***);  System.***out***.println("请输入姓名：");  String name = input.nextLine();    System.***out***.println("请输入手机号码：");  String tel = input.nextLine();    System.***out***.println("请输入性别：");  String gender = input.nextLine();    System.***out***.println("请输入薪资：");  **double** salary = input.nextDouble();    System.***out***.println("请输入部门编号：");  **int** did = input.nextInt();    //1、连接数据库  Class.*forName*("com.mysql.jdbc.Driver");    String url = "jdbc:mysql://localhost:3306/test";  String user = "root";  String password = "123456";  Connection conn = DriverManager.*getConnection*(url, user, password);  //2、编写带？的SQL  String sql = "INSERT INTO t\_employee (ename,tel,gender,salary,did) VALUES(?,?,?,?,?)";    // 3、准备一个PreparedStatement：预编译sql  // 执行添加语句，如果需要获取自增长的键值，那么在此处要告知mysql服务器，在创建PreparedStatement对象时，增加一个参数  //autoGeneratedKeys - 指示是否应该返回自动生成的键的标志，它是 Statement.RETURN\_GENERATED\_KEYS 或 Statement.NO\_GENERATED\_KEYS 之一  PreparedStatement pst = conn.prepareStatement(sql,Statement.***RETURN\_GENERATED\_KEYS***);    // 4、把?用具体的值进行代替  pst.setString(1, name);  pst.setString(2, tel);  pst.setString(3, gender);  pst.setDouble(4, salary);  pst.setInt(5, did);  // 5、执行sql  **int** len = pst.executeUpdate();  System.***out***.println(len>0?"添加成功":"添加失败");    ResultSet rs = pst.getGeneratedKeys();  **if**(rs.next()){  System.***out***.println("新员工编号是：" + rs.getObject(1));  }  // 6、释放资源  pst.close();  conn.close();  }  } |

## 5、批处理

当需要成批插入或者更新记录时。可以采用Java的批量更新机制，这一机制允许多条语句一次性提交给数据库批量处理。通常情况下比单独提交处理更有效率。

JDBC的批量处理语句包括下面两个方法：

* addBatch()：添加需要批量处理的SQL语句或参数
* executeBatch()：执行批量处理语句；

通常我们会遇到两种批量执行SQL语句的情况：

* 多条SQL语句的批量处理；
* 一个SQL语句的批量传参；

注意：

**JDBC连接MySQL时，如果要使用批处理功能，请再url中加参数****?rewriteBatchedStatements=true**

**PreparedStatement作批处理“插入”时使用values（使用value没有效果）**

|  |
| --- |
| **package** com.atguigu.preparedstatement;  **import** java.sql.Connection;  **import** java.sql.DriverManager;  **import** java.sql.PreparedStatement;  **import** org.junit.Test;  **public** **class** TestBatch {  @Test  **public** **void** noBatch()**throws** Exception{  Class.*forName*("com.mysql.jdbc.Driver");    String url = "jdbc:mysql://localhost:3306/test";  String user = "root";  String password = "123456";  Connection conn = DriverManager.*getConnection*(url, user, password);      String sql = "INSERT INTO t\_department(dname,description) VALUES(?,?)";  PreparedStatement st = conn.prepareStatement(sql);    **for**(**int** i=0; i<1000; i++){  st.setString(1, "测试部门" + i);  st.setString(2, "测试部门描述" + i);    st.executeUpdate();  }    st.close();  conn.close();  }    @Test  **public** **void** useBatch()**throws** Exception{  Class.*forName*("com.mysql.jdbc.Driver");    String url = "jdbc:mysql://localhost:3306/test?rewriteBatchedStatements=true";  String user = "root";  String password = "123456";  Connection conn = DriverManager.*getConnection*(url, user, password);      String sql = "INSERT INTO t\_department(dname,description) VALUES(?,?)";  PreparedStatement st = conn.prepareStatement(sql);    **for**(**int** i=0; i<1000; i++){  st.setString(1, "测试部门" + i);  st.setString(2, "测试部门描述" + i);    st.addBatch();  }    st.executeBatch();    st.close();  conn.close();  }  } |

## 6、事务

JDBC程序中当**一个连接对象**被创建时，默认情况下是自动提交事务：每次执行一个 SQL 语句时，如果执行成功，就会向数据库自动提交，而不能回滚。

JDBC程序中为了让多个 SQL 语句作为一个事务执行：（重点）

* 调用 Connection 对象的 setAutoCommit(false); 以取消自动提交事务
* 在所有的 SQL 语句都成功执行后，调用 commit(); 方法提交事务
* 在其中某个操作失败或出现异常时，调用 rollback(); 方法回滚事务
* 若此时 Connection 没有被彻底关闭，还可能被重复使用, 则需要恢复其自动提交状态 setAutoCommit(true);

**注意：**

**如果多个操作，每个操作使用的是自己单独的连接，则无法保证事务。即同一个事务的多个操作必须在同一个连接下**

|  |
| --- |
| **package** com.atguigu.transaction;  **import** java.sql.Connection;  **import** java.sql.DriverManager;  **import** java.sql.PreparedStatement;  **import** java.sql.SQLException;  **public** **class** TestTransaction {  **public** **static** **void** main(String[] args){  Connection conn = **null**;  **try** {  //1、连接数据库  Class.*forName*("com.mysql.jdbc.Driver");  String url = "jdbc:mysql://localhost:3306/test";  String user = "root";  String password = "123456";  conn = DriverManager.*getConnection*(url, user, password);  //设置手动提交  conn.setAutoCommit(**false**);  String sql1 = "update t\_department set description = ? where did = ?";  PreparedStatement pst1 = conn.prepareStatement(sql1);  pst1.setObject(1, "挣大钱的");  pst1.setObject(2, 4);  **int** len1 = pst1.executeUpdate();  System.***out***.println(len1>0?"更新部门信息成功":"更新部门信息失败");  pst1.close();    String sql2 = "update t\_employee set salary = salary + ? where did = ?";  PreparedStatement pst2 = conn.prepareStatement(sql2);  pst2.setObject(1, 20000);  pst2.setObject(2, 4);  **int** len2 = pst2.executeUpdate();  System.***out***.println(len2>0?"更新部门信息成功":"更新部门信息失败");  pst2.close();    conn.commit();  }**catch** (Exception e) {  **try** {  **if**(conn!=**null**){  conn.rollback();  }  } **catch** (SQLException e1) {  e1.printStackTrace();  }  } **finally**{  **try** {  **if**(conn!=**null**){  //恢复自动提交  conn.setAutoCommit(**true**);  //释放连接  conn.close();  }  } **catch** (SQLException e) {  e.printStackTrace();  }  }  }  } |

# 五、数据库连接池

## 1、数据库连接池

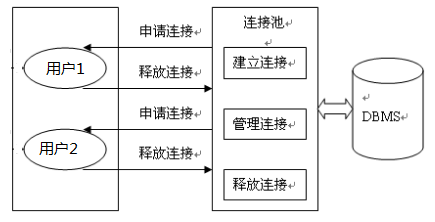
### （1）数据库连接池的必要性

不使用数据库连接池存在的问题:

* 普通的JDBC数据库连接使用 DriverManager 来获取，每次向数据库建立连接的时候都要将 Connection 加载到内存中，再验证IP地址，用户名和密码(得花费0.05s～1s的时间)。需要数据库连接的时候，就向数据库要求一个，执行完成后再断开连接。这样的方式将会消耗大量的资源和时间。**数据库的连接资源并没有得到很好的重复利用**.若同时有几百人甚至几千人在线，频繁的进行数据库连接操作将占用很多的系统资源，严重的甚至会造成服务器的崩溃。
* 对于每一次数据库连接，使用完后都得断开。否则，如果程序出现异常而**未能关闭**，将会导致**数据库系统中的内存泄漏**，最终将导致重启数据库。
* 这种开发不能控制被创建的**连接对象数**，系统资源会被毫无顾及的分配出去，如连接**过多**，也可能导致**内存泄漏，服务器崩溃**。

为解决传统开发中的数据库连接问题，可以采用**数据库连接池技术（connection pool）**。

**数据库连接池的基本思想**就是为数据库连接建立一个“缓冲池”。预先在缓冲池中放入一定数量的连接，当需要建立数据库连接时，只需从“缓冲池”中取出一个，使用完毕之后再放回去。数据库连接池负责分配、管理和释放数据库连接，它**允许应用程序重复使用一个现有的数据库连接，而不是重新建立一个**。连接池的最大数据库连接数量限定了这个连接池能占有的最大连接数，当应用程序向连接池请求的连接数超过最大连接数量时，这些请求将被加入到等待队列中。



数据库连接池技术的优点：

* **资源重用：**
  + 由于数据库连接得以重用，避免了频繁创建，释放连接引起的大量性能开销。在减少系统消耗的基础上，另一方面也增加了系统运行环境的平稳性。
* **更快的系统反应速度**
  + 数据库连接池在初始化过程中，往往已经创建了若干数据库连接置于连接池中备用。此时连接的初始化工作均已完成。对于业务请求处理而言，直接利用现有可用连接，避免了数据库连接初始化和释放过程的时间开销，从而减少了系统的响应时间
* **新的资源分配手段**
  + 对于多应用共享同一数据库的系统而言，可在应用层通过数据库连接池的配置，实现某一应用最大可用数据库连接数的限制，避免某一应用独占所有的数据库资源
* **统一的连接管理，避免数据库连接泄露**
  + 在较为完善的数据库连接池实现中，可根据预先的占用超时设定，强制回收被占用连接，从而避免了常规数据库连接操作中可能出现的资源泄露

### （2）多种开源的数据库连接池

JDBC 的数据库连接池使用 javax.sql.DataSource 来表示，DataSource 只是一个接口，该接口通常由服务器(Weblogic, WebSphere, Tomcat)提供实现，也有一些开源组织提供实现：

* DBCP 是Apache提供的数据库连接池，速度相对c3p0较快，但因自身存在BUG，Hibernate3已不再提供支持
* C3P0 是一个开源组织提供的一个数据库连接池，速度相对较慢，稳定性还可以
* Proxool 是sourceforge下的一个开源项目数据库连接池，有监控连接池状态的功能，稳定性较c3p0差一点
* BoneCP 是一个开源组织提供的数据库连接池，速度快
* Druid 是阿里提供的数据库连接池，据说是集DBCP 、C3P0 、Proxool 优点于一身的数据库连接池，但是速度不知道是否有BoneCP快

DataSource 通常被称为数据源，它包含连接池和连接池管理两个部分，习惯上也经常把 DataSource 称为连接池

注意：

* 数据源和数据库连接不同，数据源无需创建多个，它是产生数据库连接的工厂，因此整个应用只需要一个数据源即可。
* 当数据库访问结束后，程序还是像以前一样关闭数据库连接：conn.close(); 但conn.close()并没有关闭数据库的物理连接，它仅仅把数据库连接释放，归还给了数据库连接池。

### （3）Druid（德鲁伊）数据源

Druid是阿里巴巴开源平台上一个数据库连接池实现，它结合了C3P0、DBCP、Proxool等DB池的优点，同时加入了日志监控，可以很好的监控DB池连接和SQL的执行情况，可以说是针对监控而生的DB连接池，据说是目前最好的连接池。



|  |
| --- |
| **package** com.atguigu.druid;  **import** java.sql.Connection;  **import** java.util.Properties;  **import** javax.sql.DataSource;  **import** com.alibaba.druid.pool.DruidDataSourceFactory;  **public** **class** TestDruid {  **public** **static** **void** main(String[] args) **throws** Exception {  Properties pro = **new** Properties();  pro.load(TestDruid.**class**.getClassLoader().getResourceAsStream("druid.properties"));  DataSource ds = DruidDataSourceFactory.*createDataSource*(pro);  Connection conn = ds.getConnection();  System.***out***.println(conn);  //测试获取超过最大连接数的连接数量（连接关闭、未关闭的情况）  }  } |
| url=jdbc:mysql://localhost:3306/test?rewriteBatchedStatements=true  username=root  password=123456  driverClassName=com.mysql.jdbc.Driver  initialSize=10  maxActive=20  maxWait=1000 |

详细配置参数：

| **配置** | **缺省** | **说明** |
| --- | --- | --- |
| name |  | 配置这个属性的意义在于，如果存在多个数据源，监控的时候可以通过名字来区分开来。 如果没有配置，将会生成一个名字，格式是：”DataSource-” + System.identityHashCode(this) |
| jdbcUrl |  | 连接数据库的url，不同数据库不一样。例如：mysql : jdbc:mysql://10.20.153.104:3306/druid2 oracle : jdbc:oracle:thin:@10.20.149.85:1521:ocnauto |
| username |  | 连接数据库的用户名 |
| password |  | 连接数据库的密码。如果你不希望密码直接写在配置文件中，可以使用ConfigFilter。详细看这里：[https://github.com/alibaba/druid/wiki/%E4%BD%BF%E7%94%A8ConfigFilter](https://github.com/alibaba/druid/wiki/%E4%BD%BF%E7%94%A8ConfigFilter" \t "https://blog.csdn.net/deng11408205/article/details/_blank) |
| driverClassName |  | 根据url自动识别 这一项可配可不配，如果不配置druid会根据url自动识别dbType，然后选择相应的driverClassName(建议配置下) |
| initialSize | 0 | 初始化时建立物理连接的个数。初始化发生在显示调用init方法，或者第一次getConnection时 |
| maxActive | 8 | 最大连接池数量 |
| maxIdle | 8 | 已经不再使用，配置了也没效果 |
| minIdle |  | 最小连接池数量 |
| maxWait |  | 获取连接时最大等待时间，单位毫秒。配置了maxWait之后，缺省启用公平锁，并发效率会有所下降，如果需要可以通过配置useUnfairLock属性为true使用非公平锁。 |
| poolPreparedStatements | false | 是否缓存preparedStatement，也就是PSCache。PSCache对支持游标的数据库性能提升巨大，比如说oracle。在mysql下建议关闭。 |
| maxOpenPreparedStatements | -1 | 要启用PSCache，必须配置大于0，当大于0时，poolPreparedStatements自动触发修改为true。在Druid中，不会存在Oracle下PSCache占用内存过多的问题，可以把这个数值配置大一些，比如说100 |
| validationQuery |  | 用来检测连接是否有效的sql，要求是一个查询语句。如果validationQuery为null，testOnBorrow、testOnReturn、testWhileIdle都不会其作用。 |
| testOnBorrow | true | 申请连接时执行validationQuery检测连接是否有效，做了这个配置会降低性能。 |
| testOnReturn | false | 归还连接时执行validationQuery检测连接是否有效，做了这个配置会降低性能 |
| testWhileIdle | false | 建议配置为true，不影响性能，并且保证安全性。申请连接的时候检测，如果空闲时间大于timeBetweenEvictionRunsMillis，执行validationQuery检测连接是否有效。 |
| timeBetweenEvictionRunsMillis |  | 有两个含义： 1)Destroy线程会检测连接的间隔时间2)testWhileIdle的判断依据，详细看testWhileIdle属性的说明 |
| numTestsPerEvictionRun |  | 不再使用，一个DruidDataSource只支持一个EvictionRun |
| minEvictableIdleTimeMillis |  |  |
| connectionInitSqls |  | 物理连接初始化的时候执行的sql |
| exceptionSorter |  | 根据dbType自动识别 当数据库抛出一些不可恢复的异常时，抛弃连接 |
| filters |  | 属性类型是字符串，通过别名的方式配置扩展插件，常用的插件有： 监控统计用的filter:stat日志用的filter:log4j防御sql注入的filter:wall |
| proxyFilters |  | 类型是List，如果同时配置了filters和proxyFilters，是组合关系，并非替换关系 |

## 2、ThreadLocal

JDK 1.2的版本中就提供java.lang.ThreadLocal，为解决多线程程序的并发问题提供了一种新的思路。使用这个工具类可以很简洁地编写出优美的多线程程序。

ThreadLocal用于保存某个线程共享变量，原因是在Java中，每一个线程对象中都有一个ThreadLocalMap<ThreadLocal, Object>，其key就是一个ThreadLocal，而Object即为该线程的共享变量。而这个map是通过ThreadLocal的set和get方法操作的。对于同一个static ThreadLocal，不同线程只能从中get，set，remove自己的变量，而不会影响其他线程的变量。

1、ThreadLocal.get: 获取ThreadLocal中当前线程共享变量的值。

2、ThreadLocal.set: 设置ThreadLocal中当前线程共享变量的值。

3、ThreadLocal.remove: 移除ThreadLocal中当前线程共享变量的值。

|  |
| --- |
| **package** com.atguigu.threadlocal;  **import** java.util.Random;  **public** **class** Tools {  **private** **static** Random *rand* = **new** Random();  **private** **static** ThreadLocal<Integer> *threadLocal* = **new** ThreadLocal<>();    **public** **static** Integer getNumber() {  // 从当前线程中获取数字  Integer num = *threadLocal*.get();  **if** (num == **null**) {  //随机产生一个  num = *rand*.nextInt(100);  // 将数字与当前线程绑定  *threadLocal*.set(num);  }  **return** num;  }  **public** **static** **void** removeNumber() {  // 获取当前线程中的数字  Integer num = *threadLocal*.get();  **if** (num != **null**) {  // 将数字从当前线程中移除  *threadLocal*.remove();  }  }  } |
| **package** com.atguigu.threadlocal;  **public** **class** TestThreadLocal {  **public** **static** **void** main(String[] args) {  MyRunnable my = **new** MyRunnable();  **new** Thread(my,"a").start();  **new** Thread(my,"B--").start();  }  }  **class** MyRunnable **implements** Runnable{  **public** **void** run(){  **for** (**int** i = 1; i <=5; i++) {  System.***out***.println(Thread.*currentThread*().getName() + ":" + Tools.*getNumber*());  **try** {  Thread.*sleep*(100);  } **catch** (InterruptedException e) {  e.printStackTrace();  }  **if**(i==3){  Tools.*removeNumber*();  }  }  }  } |
| a:3  B--:53  a:3  B--:53  a:3  B--:53  remove后  a:9  remove后  B--:40  a:9  B--:40 |

发现ThreadLocal用于保存每一个线程共享变量，只要不删除remove，那么该值在该线程的生命周期中一直存在。

最常见的ThreadLocal使用场景为 用来解决 数据库连接、Session管理等。

## 3、封装JDBCTools

|  |
| --- |
| **package** com.atguigu.util;  **import** java.sql.Connection;  **import** java.sql.SQLException;  **import** java.util.Properties;  **import** javax.sql.DataSource;  **import** com.alibaba.druid.pool.DruidDataSourceFactory;  /\*  \* 获取连接或释放连接的工具类  \*/  **public** **class** JDBCTools {  // 1、数据源,即连接池  **private** **static** DataSource *dataSource*;    // 2、ThreadLocal对象  **private** **static** ThreadLocal<Connection> *threadLocal*;  **static** {  **try** {  //1、读取druip.properties文件  Properties pro = **new** Properties();  pro.load(JDBCTools.**class**.getClassLoader().getResourceAsStream("druid.properties"));    //2、连接连接池  *dataSource* = DruidDataSourceFactory.*createDataSource*(pro);  //3、创建线程池  *threadLocal* = **new** ThreadLocal<>();  } **catch** (Exception e) {  e.printStackTrace();  }  }  /\*\*  \* 获取连接的方法  \*  \* **@return**  \* **@throws** SQLException  \*/  **public** **static** Connection getConnection() {  // 从当前线程中获取连接  Connection connection = *threadLocal*.get();  **if** (connection == **null**) {  // 从连接池中获取一个连接  **try** {  connection = *dataSource*.getConnection();  // 将连接与当前线程绑定  *threadLocal*.set(connection);  } **catch** (SQLException e) {  e.printStackTrace();  }  }  **return** connection;  }  /\*\*  \* 释放连接的方法  \*  \* **@param** connection  \*/  **public** **static** **void** releaseConnection() {  // 获取当前线程中的连接  Connection connection = *threadLocal*.get();  **if** (connection != **null**) {  **try** {  connection.close();  // 将已经关闭的连接从当前线程中移除  *threadLocal*.remove();  } **catch** (SQLException e) {  e.printStackTrace();  }  }  }  } |

## 4、封装BasicDAOImpl

DAO：Data Access Object访问数据信息的类和接口，包括了对数据的CRUD（Create、Retrival、Update、Delete），而不包含任何业务相关的信息

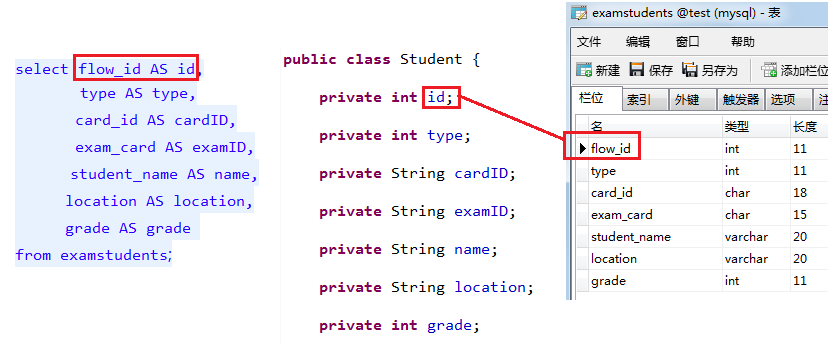
作用：为了实现功能的模块化，更有利于代码的维护和升级。

|  |
| --- |
| **package** com.atguigu.dao;  **import** java.lang.reflect.Field;  **import** java.sql.Connection;  **import** java.sql.PreparedStatement;  **import** java.sql.ResultSet;  **import** java.sql.ResultSetMetaData;  **import** java.sql.SQLException;  **import** java.util.ArrayList;  **import** com.atguigu.utils.JdbcUtils;  **public** **class** BasicDAOImpl {  //适用于insert,update,delete语句  **public** **int** update(String sql,Object... args) {  //1、获取连接  Connection conn = JdbcUtils.*getConnection*();  //2、创建PreparedStatement  PreparedStatement pst = **null**;  **try** {  pst = conn.prepareStatement(sql);    //3、设置？的值  **if**(args!=**null** && args.length>0){  **for** (**int** i = 0; i < args.length; i++) {//数组的下标从0开始  pst.setObject(i+1, args[i]);//？的序号从1开始  }  }    //4、执行更新sql  **return** pst.executeUpdate();  } **catch** (SQLException e) {  **throw** **new** RuntimeException(e);  }**finally**{  **try** {  //5、关闭  pst.close();  JdbcUtils.*free*();  } **catch** (SQLException e) {  e.printStackTrace();  }  }  }    //T可能代表Department，Employee等各种Javabean的对象  //clazz是决定你查询的结果是Department，Employee类型中的那个对象  **public** <T> ArrayList<T> getBeanList(Class<T> clazz, String sql,Object... args){  //1、获取连接  Connection conn = JdbcUtils.*getConnection*();    //2、创建PreparedStatement  PreparedStatement pst = **null**;    **try** {  pst = conn.prepareStatement(sql);    //3、设置？的值  **if**(args!=**null** && args.length>0){  **for** (**int** i = 0; i < args.length; i++) {//数组的下标从0开始  pst.setObject(i+1, args[i]);//？的序号从1开始  }  }    //4、执行查询  ResultSet rs = pst.executeQuery();    //获取结果集的元数据对象，该对象有对结果集的数据进行描述的相关信息  ResultSetMetaData rsm = rs.getMetaData();  //(1)获取结果集的列数  **int** count = rsm.getColumnCount();    ArrayList<T> list = **new** ArrayList<T>();  //5、把ResultSet结果集中的数据封装到一个一个JavaBean对象中，并且存到list中  **while**(rs.next()){//循环一次，代表一行，一行就是一个JavaBean对象  //(2)创建一个JavaBean的对象  T obj = clazz.newInstance();    //有几列，就代表有几个属性  //为obj的每一个属性赋值  **for** (**int** i = 0; i < count; i++) {  //通过反射设置属性的值  //(3)从结果集的元数据对象中获取第几列的字段名  String columnName = rsm.getColumnLabel(i+1);//mysql的序号从1开始    //(4)获取属性对象  Field field = clazz.getDeclaredField(columnName);//根据字段名称，获取属性对象    //(5)设置属性可以被访问  field.setAccessible(**true**);    //(6)设置属性的值  field.set(obj, rs.getObject(i+1));//从结果集中获取第i+1的值，赋值给该属性  }    list.add(obj);  }  //7、返回结果  **return** list;  } **catch** (Exception e) {  **throw** **new** RuntimeException(e);  }**finally**{  **try** {  //6、关闭  pst.close();  JdbcUtils.*free*();  } **catch** (SQLException e) {  e.printStackTrace();  }  }  }    **public** <T> T getBean(Class<T> type,String sql, Object... params) {  **return** getBeanList(type,sql,params).get(0);  }  } |

## 5、Employee的增删改查

|  |  |
| --- | --- |
| int,double等在Java中都用包装类，因为mysql中的所有类型都可能是NULL，而Java只有引用数据类型才有NULL值 | public class Employee {  private Integer eid;  private String ename;  private String tel;  private String gender;//mysql中用char,在Java中也要用String  private Double salary;  private Double commissionPct;  private Date birthday;//此处用String或Date  private Date hiredate;  private Integer jobId;  private String email;  private Integer mid;  private String address;  private String nativePlace;  private Integer did;  ...  } |

通过给列取别名的方式，来告知数据库的列名与其对应实体的属性名



|  |
| --- |
| **package** com.atguigu.dao;  **import** java.util.List;  **import** com.atguigu.bean.Employee;  **public** **interface** EmployeeDAO {  **void** addEmployee(Employee emp);  List<Employee> getAllEmployee();  **void** update(Employee emp);  **void** deleteByEid(**int** eid);  Employee getById(**int** eid);  } |
| **package** com.atguigu.dao;  **import** java.util.List;  **import** com.atguigu.bean.Employee;  **public** **class** EmployeeDAOImpl **extends** BasicDAOImpl **implements** EmployeeDAO{  @Override  **public** **void** addEmployee(Employee emp) {  **try** {  //eid,ename,tel,gender,salary,commission\_pct,birthday,hiredate,job\_id,email,MID,address,native\_place,did  String sql = "INSERT INTO t\_employee VALUES(NULL,?,?,?,?,?,?,?,?,?,?,?,?,?)";  update(sql, emp.getEname(),  emp.getTel(),  emp.getGender(),  emp.getSalary(),  emp.getCommissionPct(),  emp.getBirthday(),  emp.getHiredate(),  emp.getJobId(),  emp.getEmail(),  emp.getMid(),  emp.getAddress(),  emp.getNativePlace(),  emp.getDid());  } **catch** (Exception e) {  **throw** **new** RuntimeException(e.getMessage());  }  }  @Override  **public** List<Employee> getAllEmployee() {  **try** {  //如果字段名和属性名对不上，那么可以取别名，通过别名来代表属性名  String sql = "select eid,ename,tel,gender,salary,commission\_pct as commissionPct,birthday,hiredate,job\_id as jobId,email,mid,address,native\_place as nativePlace,did from t\_employee";  **return** getBeanList(Employee.**class**, sql);  } **catch** (Exception e) {  **throw** **new** RuntimeException(e.getMessage());  }  }  @Override  **public** **void** update(Employee emp) {  **try**{  String sql = "update t\_employee set ename=?,tel=?,gender=?,salary=?,commission\_pct =?,birthday=?,hiredate=?,job\_id=?,email=?,mid=?,address=?,native\_place=? ,did=? where eid=?";  update(sql, emp.getEname(),  emp.getTel(),  emp.getGender(),  emp.getSalary(),  emp.getCommissionPct(),  emp.getBirthday(),  emp.getHiredate(),  emp.getJobId(),  emp.getEmail(),  emp.getMid(),  emp.getAddress(),  emp.getNativePlace(),  emp.getDid(),  emp.getEid());  } **catch** (Exception e) {  **throw** **new** RuntimeException(e.getMessage());  }  }  @Override  **public** **void** deleteByEid(**int** eid) {  **try** {  String sql = "delete from t\_employee where eid = ?";  update(sql, eid);  } **catch** (Exception e) {  **throw** **new** RuntimeException(e.getMessage());  }  }    @Override  **public** Employee getById(**int** eid) {  **try** {  String sql = "select eid,ename,tel,gender,salary,commission\_pct as commissionPct,birthday,hiredate,job\_id as jobId,email,mid,address,native\_place as nativePlace,did from t\_employee where eid = ?";  **return** getBean(Employee.**class**, sql, eid);  } **catch** (Exception e) {  **throw** **new** RuntimeException(e.getMessage());  }  }  } |
| **package** com.atguigu.dao;  **import** java.util.List;  **import** org.junit.Test;  **import** com.atguigu.bean.Employee;  **public** **class** TestEmployeeDAOImpl {  EmployeeDAOImpl ed = **new** EmployeeDAOImpl();    @Test  **public** **void** test1(){  List<Employee> all = ed.getAllEmployee();  **for** (Employee employee : all) {  System.***out***.println(employee);  }  }  } |

# 六、Apache—DBUtils简介

commons-dbutils 是 Apache 组织提供的一个开源 JDBC工具类库，它是对JDBC的简单封装，学习成本极低，并且使用dbutils能极大简化jdbc编码的工作量，同时也不会影响程序的性能。



## 1、DbUtils类

DbUtils ：提供如关闭连接、装载JDBC驱动程序等常规工作的工具类，里面的所有方法都是静态的。主要方法如下：

* public static void close(…) throws java.sql.SQLException：　DbUtils类提供了三个重载的关闭方法。这些方法检查所提供的参数是不是NULL，如果不是的话，它们就关闭Connection、Statement和ResultSet。
* public static void closeQuietly(…): 这一类方法不仅能在Connection、Statement和ResultSet为NULL情况下避免关闭，还能隐藏一些在程序中抛出的SQLEeception。
* public static void commitAndClose(Connection conn)throws SQLException 用来提交连接的事务，然后关闭连接
* public static void commitAndCloseQuietly(Connection conn)： 用来提交连接的事务，然后关闭连接，并且在关闭连接时不抛出SQL异常。
* public static void rollback(Connection conn)throws SQLException允许conn为null，因为方法内部做了判断
* public static void rollbackAndClose(Connection conn)throws SQLException
* rollbackAndCloseQuietly(Connection)
* public static boolean loadDriver(java.lang.String driverClassName)：这一方装载并注册JDBC驱动程序，如果成功就返回true。使用该方法，你不需要捕捉这个异常ClassNotFoundException。

## 2、QueryRunner类

该类封装了SQL的执行，是线程安全的。

1. 可以实现增、删、改、查、批处理、
2. 考虑了事务处理需要共用Connection。

（3）该类最主要的就是简单化了SQL查询，它与ResultSetHandler组合在一起使用可以完成大部分的数据库操作，能够大大减少编码量。

QueryRunner类提供了两个构造方法：

* QueryRunner()：默认的构造方法
* QueryRunner(DataSource ds)：需要一个 javax.sql.DataSource 来作参数的构造方法。

### （1）更新

* public int update(Connection conn, String sql, Object... params) throws SQLException:用来执行一个更新（插入、更新或删除）操作。
* ......

### （2）插入

* public <T> T insert(Connection conn,String sql,ResultSetHandler<T> rsh, Object... params) throws SQLException：只支持INSERT语句，其中 rsh - The handler used to create the result object from the ResultSet of auto-generated keys. 返回值: An object generated by the handler.即自动生成的键值
* ....

### （3）批处理

* public int[] batch(Connection conn,String sql,Object[][] params)throws SQLException： INSERT, UPDATE, or DELETE语句
* public <T> T insertBatch(Connection conn,String sql,ResultSetHandler<T> rsh,Object[][] params)throws SQLException：只支持INSERT语句
* .....

### （4）使用QueryRunner类实现查询

* public Object query(Connection conn, String sql, ResultSetHandler rsh,Object... params) throws SQLException：执行一个查询操作，在这个查询中，对象数组中的每个元素值被用来作为查询语句的置换参数。该方法会自行处理 PreparedStatement 和 ResultSet 的创建和关闭。
* ......

## 3、ResultSetHandler接口

该接口用于处理 java.sql.ResultSet，将数据按要求转换为另一种形式。ResultSetHandler 接口提供了一个单独的方法：Object handle (java.sql.ResultSet rs)该方法的返回值将作为QueryRunner类的query()方法的返回值。

该接口有如下实现类可以使用：

* BeanHandler：将结果集中的第一行数据封装到一个对应的JavaBean实例中。
* BeanListHandler：将结果集中的每一行数据都封装到一个对应的JavaBean实例中，存放到List里。
* ScalarHandler：查询单个值对象
* MapHandler：将结果集中的第一行数据封装到一个Map里，key是列名，value就是对应的值。
* MapListHandler：将结果集中的每一行数据都封装到一个Map里，然后再存放到List
* ColumnListHandler：将结果集中某一列的数据存放到List中。
* KeyedHandler(name)：将结果集中的每一行数据都封装到一个Map里，再把这些map再存到一个map里，其key为指定的key。
* ArrayHandler：把结果集中的第一行数据转成对象数组。
* ArrayListHandler：把结果集中的每一行数据都转成一个数组，再存放到List中。

## 4、示例代码

DAO：Data Access Object访问数据信息的类和接口，包括了对数据的CRUD（Create、Retrival、Update、Delete），而不包含任何业务相关的信息

作用：为了实现功能的模块化，更有利于代码的维护和升级。

|  |  |
| --- | --- |
|  |  |
|  |  |

### 1、DAO接口

|  |
| --- |
| **package** com.atguigu.dao;  **import** com.atguigu.bean.User;  **public** **interface** UserDAO {  /\*\*  \* 根据用户名和密码获取数据库中的记录  \*  \* **@param** user  \* **@return** User：用户名和密码正确 null：用户名或密码不正确  \*/  **public** User getUser(User user);  /\*\*  \* 根据用户名获取数据库中的记录  \*  \* **@param** user  \* **@return** true：用户名已存在， false：用户名可用  \*/  **public** **boolean** checkUserName(User user);  /\*\*  \* 将用户保存到数据库  \*  \* **@param** user  \*/  **public** **void** saveUser(User user);  } |
| **package** com.atguigu.dao;  **import** java.util.List;  **import** com.atguigu.bean.Book;  **public** **interface** BookDAO {  /\*\*  \* 获取所有图书的方法  \*  \* **@return**  \*/  **public** List<Book> getBooks();  /\*\*  \* 添加图书的方法  \*  \* **@param** book  \*/  **public** **void** addBook(Book book);  /\*\*  \* 根据图书的id删除图书的方法  \*  \* **@param** bookId  \*/  **public** **void** deleteBookById(String bookId);  /\*\*  \* 根据图书的id获取图书信息  \*  \* **@param** bookId  \* **@return**  \*/  **public** Book getBookById(String bookId);  /\*\*  \* 更新图书信息的方法  \*  \* **@param** book  \*/  **public** **void** updateBook(Book book);  /\*\*  \* 批量更新图书的库存和销量  \*  \* **@param** params  \*/  **public** **void** batchUpdateSalesAndStock(Object[][] params);  } |
| **package** com.atguigu.dao;  **import** java.util.List;  **import** com.atguigu.bean.Order;  **public** **interface** OrderDAO {  /\*\*  \* 保存订单的方法  \*  \* **@param** order  \*/  **public** **void** saveOrder(Order order);  /\*\*  \* 获取所用订单的方法  \*  \* **@return**  \*/  **public** List<Order> getOrders();  /\*\*  \* 获取我的订单的方法  \*  \* **@param** userId  \* **@return**  \*/  **public** List<Order> getMyOrders(**int** userId);  /\*\*  \* 更新订单的状态的方法，例如已发货、确认收货等  \*  \* **@param** orderId  \* **@param** state  \*/  **public** **void** updateOrderState(String orderId, **int** state);  } |
| **package** com.atguigu.dao;  **import** java.util.List;  **import** com.atguigu.bean.OrderItem;  **public** **interface** OrderItemDAO {  /\*\*  \* 根据订单号获取对应的订单项  \*  \* **@param** orderId  \* **@return**  \*/  **public** List<OrderItem> getOrderItemsByOrderId(String orderId);  /\*\*  \* 批量插入订单项的方法  \*  \* **@param** params  \*/  **public** **void** batchInsertOrderItems(Object[][] params);  } |

### 2、BasicDAOImpl

|  |
| --- |
| **package** com.atguigu.dao.impl;  **import** java.sql.Connection;  **import** java.sql.SQLException;  **import** java.util.List;  **import** org.apache.commons.dbutils.QueryRunner;  **import** org.apache.commons.dbutils.handlers.BeanHandler;  **import** org.apache.commons.dbutils.handlers.BeanListHandler;  **import** org.apache.commons.dbutils.handlers.ScalarHandler;  **import** com.atguigu.util.JDBCTools;  /\*\*  \* 定义一个用来被继承的对数据库进行基本操作的Dao  \*/  **public** **class** BasicDAOImpl {  **private** QueryRunner queryRunner = **new** QueryRunner();  /\*\*  \* 通用的增删改操作  \*  \* **@param** sql  \* **@param** params  \* **@return**  \*/  **public** **int** update(String sql, Object... params) {  // 获取连接  Connection connection = JDBCTools.*getConnection*();  **int** count = 0;  **try** {  count = queryRunner.update(connection, sql, params);  } **catch** (SQLException e) {  //将编译时异常转换为运行时异常向上抛  **throw** **new** RuntimeException(e);  }  **return** count;  }  /\*\*  \* 获取一个对象  \*  \* **@param** sql  \* **@param** params  \* **@return**  \*/  **public** <T> T getBean(Class<T> type,String sql, Object... params) {  // 获取连接  Connection connection = JDBCTools.*getConnection*();  T t = **null**;  **try** {  t = queryRunner.query(connection, sql, **new** BeanHandler<T>(type), params);  } **catch** (SQLException e) {  //将编译时异常转换为运行时异常向上抛  **throw** **new** RuntimeException(e);  }  **return** t;  }  /\*\*  \* 获取所有对象  \*  \* **@param** sql  \* **@param** params  \* **@return**  \*/  **public** <T> List<T> getBeanList(Class<T> type,String sql, Object... params) {  // 获取连接  Connection connection = JDBCTools.*getConnection*();  List<T> list = **null**;  **try** {  list = queryRunner.query(connection, sql, **new** BeanListHandler<T>(type), params);  } **catch** (SQLException e) {  //将编译时异常转换为运行时异常向上抛  **throw** **new** RuntimeException(e);  }  **return** list;  }  /\*\*  \* 获取一个单一值的方法，专门用来执行像select count(\*)... 这样的sql语句  \*  \* **@param** sql  \* **@param** params  \* **@return**  \*/  **public** Object getSingleValue(String sql, Object... params) {  // 获取连接  Connection connection = JDBCTools.*getConnection*();  Object value = **null**;  **try** {  value = queryRunner.query(connection, sql, **new** ScalarHandler(), params);  } **catch** (SQLException e) {  //将编译时异常转换为运行时异常向上抛  **throw** **new** RuntimeException(e);  }  **return** value;  }  /\*\*  \* 进行批处理的方法  \* 关于二维数组Object[][] params  \* 二维数组的第一维是sql语句要执行的次数  \* 二维数组的第二维就是每条sql语句中要填充的占位符  \*  \* **@param** sql  \* **@param** params  \*/  **public** **void** batchUpdate(String sql , Object[][] params){  //获取连接  Connection connection = JDBCTools.*getConnection*();  **try** {  queryRunner.batch(connection ,sql, params);  } **catch** (SQLException e) {  //将编译时异常转换为运行时异常向上抛  **throw** **new** RuntimeException(e);  }  }  } |

### 3、DAO实现类

|  |
| --- |
| **package** com.atguigu.dao.impl;  **import** com.atguigu.bean.User;  **import** com.atguigu.dao.UserDAO;  **public** **class** UserDAOImpl **extends** BasicDAOImpl **implements** UserDAO{  @Override  **public** User getUser(User user) {  // 写查询数据库的sql语句  String sql = "select id , username , password , email from users where username = ? and password = ?";  // 调用BaseDao中的getBean方法  User bean = getBean(User.**class**, sql, user.getUsername(), user.getPassword());  **return** bean;  }  @Override  **public** **boolean** checkUserName(User user) {  // 写查询数据库的sql语句  String sql = "select id , username , password , email from users where username = ?";  // 调用BaseDao中的getBean方法  User bean = getBean(User.**class**, sql, user.getUsername());  **return** bean!=**null**;//不为空，说明已存在，返回true，如果bEan是空的，没找到，bean!=null返回false，说明不存在  }  @Override  **public** **void** saveUser(User user) {  //写添加数据到数据库的sql语句  String sql = "insert into users(username,password,email) values(?,?,?)";  //调用BaseDao中通用的增删改的方法  update(sql, user.getUsername(),user.getPassword(),user.getEmail());  }  } |