# 1. JDBC简介

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

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

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

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

## 1.1 JDBC开发前的准备

首先新建一个Java项目，并且引入MySQL的JDBC实现jar包。同样，MySQL官方JDBC资源可在搜狐镜像（<http://mirrors.sohu.com> ）上下载（也可去官网<https://dev.mysql.com/downloads/connector/j/> ），目录是“mysql/Connector-J/”，我们选择当前最新的5.1.40版本，即下载“mysql-connector-java-5.1.40.zip”文件。

解压下载的文件可以得到一个名为“mysql-connector-java-5.1.40-bin”的jar包，需要在项目中引入这个包才能进行JDBC的开发。

另外还要注意一点，最好此时将Eclipse整个工作空间的编码预先设置成UTF-8编码。因为我们在与MySQL进行交互时，使用的编码统一是UTF-8，所以使用JDBC进行连接数据库的开发，最好也将项目编码设置成UTF-8，防止中文乱码的发生。设置方法：点击“Window - Prefrences - General - Workspace”，将其中的“Text file encoding”设置成UTF-8编码。

将默认编码由系统的GBK设置成UTF-8后，程序代码在IDE中运行是没有任何问题的。但如果脱离IDE，自己在Windows命令行中编译运行项目，很可能在编译时就会提示编码错误。原因是javac命令默认也会使用Windows中文系统默认的GBK编码来加载Java源文件进行编译，这时编码错误导致源码乱码，编译时就会理所当然地出错了。此时，可在javac命令中使用“encoding UTF-8”参数告诉编译器源文件是采用UTF-8编码的，这样就不会错误。其实在IDE中自动帮助我们完成了这个工作。

当然Windows对于编码问题就是比较麻烦，而Linux中由于就是采用UTF-8编码，因此无此问题。这里顺便也提一下，如果项目是使用AWT（Abstract Window Toolkit）编码的图形界面项目，还是老老实实地使用GBK编码吧，因为使用别的编码的话，中文还是会乱码，因为Windows中文系统的API调用显示的字符只支持系统默认的GBK编码。大家可以自行测试。

## 1.2 用JDBC连接MySQL

下面写代码连接MySQL数据库。

首先，必须知道以下信息才能连接到MySQL数据库：

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

（2）数据库服务器的用户名和密码。

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

方式一：用JDBC接口来接收MySQL驱动的实现类对象，以此获得连接。

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

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

但是运行程序出现了下面的MySQL红色警告：

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

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

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

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

使用此url就不会出现警告信息。

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

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

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

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

方式二：使用驱动管理器类连接数据库。

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

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

|  |
| --- |
| **package** com.mysql.jdbc;  **import** java.sql.SQLException; **public class** Driver **extends** NonRegisteringDriver **implements** java.sql.Driver {  *//  // Register ourselves with the DriverManager  //* **static** {  **try** {  java.sql.DriverManager.*registerDriver*(**new** Driver());  } **catch** (SQLException E) {  **throw new** RuntimeException(**"Can't register driver!"**);  }  } **public** Driver() **throws** SQLException {} } |

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

改进如下：

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

以后都像这样写获得连接的代码，十分方便。

说明：如果url中IP地址和端口就是本机的localhost和3306端口，则可省略为“/”。例如“jdbc:mysql://localhost:3306/db”可省略为“jdbc:mysql:///db”，但一般不推荐这样使用。

# 2. JDBC API的使用

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

## 2.1 连接时使用的DriverManager和Connection对象

回顾第一章的简单示例，其实使用JDBC连接数据库是如下几个步骤：

（1）注册驱动；

（2）获取数据库的连接（JDBC所有与数据库的交互都是基于连接的）；

（3）基于连接对数据库进行操作（这是下面涉及到的内容）；

（4）释放数据库连接资源。

注册驱动时用到了DriverManager对象，其中有两种方式注册驱动，一种方式是“DriverManager.registerDriver(new com.mysql.jdbc.Driver());”，这种硬编码的方式严重依赖具体的数据库驱动，且会导致数据库驱动重复注册（因为通过查看源码知道，DriverManager在加载时就注册了驱动）。因此我们推荐使用第二种方式，直接“Class.forName("com.mysql.jdbc.Driver");”。

在获得数据库连接对象Connection时，有三种方式。在第一章讲的两种方式中，其实就是DriverManager.getConnection()方法提供的不同重载，一种是将数据库连接url的用户名和密码这两个属性写成方法的参数（因为这两个是必须要使用到的），另一种是将数据库连接url的所有属性都封装到一个Properties对象中，再传递进去。当然，url中的属性不管是写在url中、写在方法参数中（用户名和密码属性）、还是封装到Properties对象中，效果都是一样的，甚至可以混用，例如第一章中就这样结合使用了：

|  |
| --- |
| DriverManager.*getConnection*(**"jdbc:mysql://localhost:3306/mytest?useSSL=false&useUnicode=true&characterEncoding=utf8"**, **"root"**, **"123456"**); |

只要有了这个属性，就会起作用，因为实际上他们最终还是会被解析成一个url字符串来连接MySQL，例如上面语句中最终的url是：

|  |
| --- |
| **"jdbc:mysql://localhost:3306/mytest?useSSL=false&useUnicode=true&characterEncoding=utf8&user=root&password=123456"** |

实际上DriverManager.getConnection()还提供了第三个重载，就直接是getConnection(String url)，直接传递一个url，这就是第三种获得连接的方式。例如：

|  |
| --- |
| DriverManager.*getConnection*(**"jdbc:mysql://localhost:3306/mytest?useSSL=false&useUnicode=true&characterEncoding=utf8&user=root&password=123456"**); |

但为了直观好看，还是建议使用Properties的方式。

那么MySQL的连接url都有哪些属性呢？同样，这时我们可以查看官方文档。官方文档的第27章介绍了连接器，其中27.3节“MySQL Connector/J”这节就是讲的JDBC连接器。但文档中让我们访问另一个链接查看文档，说明该部分文档被单独分离出来形成了“MySQL Connector/J”文档。因此最终在网址“https://dev.mysql.com/doc/connector-j/5.1/en/connector-j-reference-configuration-properties.html”中找到了介绍url属性的章节，读者可以看看，其中就有useSSL等，包括我们以后会使用的一些属性。这些属性理所当然地能应用在上面的代码中。当然该文档首页也提供了pdf等下载，和MySQL官方文档是类似的，如果嫌麻烦，也可以看之前提供的MySQL中文文档，其中这部分文档没有被分离。此外，也可看这部分文档了解MySQL的JDBC协议。

## 2.2 Statement对象

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

例1：使用Statement对象创建学生表并插入一条数据。

|  |
| --- |
| **package** com.zhang.test;  **import** java.sql.Connection; **import** java.sql.DriverManager; **import** java.sql.SQLException; **import** java.sql.Statement;  **public class** Demo {  **public static void** main(String[] args) **throws** ClassNotFoundException, SQLException {  String url = **"jdbc:mysql://localhost:3306/mytest?useSSL=false&useUnicode=true&characterEncoding=utf8"**;  Class.*forName*(**"com.mysql.jdbc.Driver"**);  Connection conn = DriverManager.*getConnection*(url, **"root"**, **"123456"**);  *// 得到Statement对象* Statement stmt = conn.createStatement();  String sql = **"create table student(sid int primary key, sname varchar(20)); "**;  **int** row = stmt.executeUpdate(sql); *// 执行建表语句。方法返回值是收影响行数* System.***out***.println(**"受影响行数："** + row);  sql = **"insert into student values(1, '张三');"**;  row = stmt.executeUpdate(sql); *// 执行插入语句* System.***out***.println(**"受影响行数："** + row);  *// 关闭Statement和Connection* stmt.close();  conn.close();  } } |

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

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

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

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

即next()用于“遍历”行，getXxx()用于获取列数据。

例2：使用Statement查询学生表中的数据。

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

此外，Statement还提供了boolean execute(String sql)方法，该方法可执行任意SQL语句，其返回值不代表执行成功与否，它代表的是执行的SQL是否有结果集返回，若有结果集返回，则返回值是true。那这时如果得到结果集呢？就需要再调用Statement的getResultSet()方法得到结果集。因此代码一般如下：

|  |
| --- |
| if (statement.execute(sql)) {  statement.getResultSet();  } |

这样的使用稍微麻烦一点，因此我们一般会直接使用executeUpdate()和executeQuery()两个方法。

## 2.3 ResultSet对象

ResultSet就是结果集对象，其中封装了结果集的数据，在前面的示例中我们已经使用过。在处理结果集时，要注意MySQL数据库中的数据类型和Java语言中数据类型的对应关系。现在列表如下，可供参考：

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

说明：bit(1)可对应Boolean数据类型，是因为1位二进制只能表示0和1，而这时用getBoolean()获取值时，1就是true，0就是false。另外要注意，对应的时间类型是java.sql包下的类，而不是java.util下的Date类。对于java.sql.Date和java.sql.Timestamp来说，他们的父类就是java.util.Date，而java.sql.Time的父类是java.util.Time，因此取值和赋值时，直接赋值或者强转即可，最多再使用毫秒数转换一下即可。

ResultSet中提供了以下常用的方法：

（1）boolean previous()：移动到前一行，返回有没有记录；

（2）boolean next()：移动到下一行，返回有没有记录；

（3）boolean absolute(int row)：移动到指定行，返回有没有记录；

（4）void beforeFirst()：移动ResultSet的最前面，即一开始的默认位置；

（5）void afterLast()：移动到ResultSet的最后面。

一般来说，会将ResultSet结果集数据封装成Java系统中的实体bean对象，以便在系统中使用。在下面的项目中我们将会看到这样的做法。

## 2.3 PreparedStatement对象

### 2.3.1 Statement对象执行SQL的缺点

通过一个案例来了解Statement的缺点。

实现登录功能，需要查询数据库中的user表。事先建立user表和数据：

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

登录程序的代码：

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

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

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

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

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

### 2.3.2 使用PreparedStatement

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

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

上述案例可修改为：

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

这样就不会产生SQL注入问题了。且PreparedStatement是预编译的SQL对象，执行效率比Statement高。

使用PreparedStatement步骤：

（1）编写预编译SQL；

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

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

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

# 3. 抽取JdbcUtil模板

在实际项目中的数据访问层（DAO）中，主要的代码就是操作数据库。我们发现连接数据库的大部分代码是相同的（例如建立连接、释放连接等），如果每次操作数据库都编写这样的代码，导致代码封装性差、不利于提高效率。因此我们希望抽取一个JdbcUtil类，其中封装数据库连接的一些操作。

## 3.1 异常处理和资源释放

在封装之前，我们先了解JDBC中的异常处理和资源释放。在之前的代码中，我们直接调用close()方法释放资源，且直接抛出了异常。实际中不能这么干，因为数据库连接资源十分宝贵，一定要确保连接对象被释放。

因此在下面的代码中我们将会看到，代码被包含在try...catch...finally中，且我们在finally中编写释放资源的代码。在调用close()方法之前，还要判断该对象是否为null，这是因为连接对象可能还没有被实例化，代码就产生了异常，这时连接对象是null。同样在调用close()方法时，也要处理该方法的异常。为了确保对象被释放，最后还指定对象为null，明确让JVM可以回收资源。

因此，完整的代码如下：

|  |
| --- |
| **package** com.company;  **import** java.sql.\*; **import** java.util.Properties;  **public class** Main {   **public static void** main(String[] args) {  Connection connection = **null**;  PreparedStatement statement = **null**;  ResultSet rs = **null**;  **try** {  Class.*forName*(**"com.mysql.jdbc.Driver"**);  Properties properties = **new** Properties();  properties.setProperty(**"user"**, **"root"**);  properties.setProperty(**"password"**, **"123qwe!@#"**);  properties.setProperty(**"useSSL"**, **"false"**);  properties.setProperty(**"useUnicode"**, **"true"**);  properties.setProperty(**"characterEncoding"**, **"utf8"**);  connection = DriverManager.*getConnection*(**"jdbc:mysql://114.55.86.230:3306/mytest"**, properties);  statement = connection.prepareStatement(**"select id, name from student;"**);  rs = statement.executeQuery();  **while** (rs.next()) {  System.***out***.println(rs.getString(1));  System.***out***.println(rs.getString(2));  }  } **catch** (Exception e) {  **throw new** RuntimeException(e);  } **finally** {  *// 释放资源。注意是从rs开始释放，后使用的先释放。* **if** (rs != **null**) {  **try** {  rs.close();  } **catch** (SQLException e) {  e.printStackTrace();  }  rs = **null**;  }  **if** (statement != **null**) {  **try** {  statement.close();  } **catch** (SQLException e) {  e.printStackTrace();  }  statement = **null**;  }  **if** (connection != **null**) {  **try** {  connection.close();  } **catch** (SQLException e) {  e.printStackTrace();  }  connection = **null**;  }  }  } } |

实际上上面代码的处理方式和MySQL Connector/J官方文档的JDBC示例程序的处理方式是一样的，是“学来的”。可以参见文档：“https://dev.mysql.com/doc/connector-j/5.1/en/connector-j-usagenotes-statements.html”。

## 3.2 封装JdbcUtil类

在我们即将要编写的JdbcUtil类中，只提供两个静态方法：getConnection()和release()，分别用于获得连接和释放资源。其中，release方法提供了两个重载：

public static void release(Connection conn, Statement stmt)；和：

public static void release(Connection conn, Statement stmt, ResultSet rs)。

因为有的数据库操作只有增删改，没有查询，就不会有ResultSet对象释放。使用Statement作为参数类型，可以兼容PreparedStatement，因为他们是父子关系。

为了使用地更灵活，我们希望把有关数据库的连接配置写在一份“db.properties”文件中，形式如下：

|  |
| --- |
| **driverClass**=**com.mysql.jdbc.Driver url**=**jdbc:mysql://114.55.86.230:3306/mytest user**=**root password**=**123qwe!@# useSSL**=**false useUnicode**=**true characterEncoding**=**utf8** |

然后JdbcUtil类会加载这个文件，以其中的数据作为连接数据库的信息。最终我们编写的JdbcUtil代码如下：

|  |
| --- |
| **package** com.company;  **import** java.io.IOException; **import** java.io.InputStream; **import** java.sql.\*; **import** java.util.Properties;  **public class** JdbcUtil {  *// 工具类禁止创建实例* **private** JdbcUtil() {}  *// 用于保存连接信息的Properties静态对象* **private static** Properties *properties*;   **static** {  *// 加载文件* InputStream inputStream = JdbcUtil.**class**.getClassLoader().getResourceAsStream(**"db.properties"**);  **try** {  *properties* = **new** Properties();  *properties*.load(inputStream);  } **catch** (IOException ex) {  *// 直接抛出初始化异常错误，文件加载错误* **throw new** ExceptionInInitializerError(ex);  }  }   */\*\*  \* 获得数据库连接对象  \** ***@return*** *数据库连接对象  \** ***@throws*** *Exception 连接异常  \*/* **public static** Connection getConnection() **throws** Exception {  Class.*forName*(*properties*.getProperty(**"driverClass"**));  *// 获得connection时用到properties对象，但其中的driverClass和url键不是连接url的属性，  // 因此这里克隆一个urlProp，并去掉其中的driverClass和url键* Properties urlProp = (Properties) *properties*.clone();  urlProp.remove(**"driverClass"**);  urlProp.remove(**"url"**);  *// 获得连接* **return** DriverManager.*getConnection*(*properties*.getProperty(**"url"**), urlProp);  }   */\*\*  \* 释放资源  \** ***@param conn*** *连接资源  \** ***@param stmt*** *statement资源  \** ***@param rs*** *结果集资源  \*/* **public static void** release(Connection conn, Statement stmt, ResultSet rs) {  **if** (rs != **null**) {  **try** {  rs.close();  } **catch** (SQLException e) {  e.printStackTrace();  }  rs = **null**;  }  **if** (stmt != **null**) {  **try** {  stmt.close();  } **catch** (SQLException e) {  e.printStackTrace();  }  stmt = **null**;  }  **if** (conn != **null**) {  **try** {  conn.close();  } **catch** (SQLException e) {  e.printStackTrace();  }  conn = **null**;  }  }   */\*\*  \* 释放资源  \** ***@param conn*** *连接资源  \** ***@param stmt*** *statement资源  \*/* **public static void** release(Connection conn, Statement stmt) {  *// 这里需要调用上面的方法，最后一个参数给null即可。  release*(conn, stmt, **null**);  } } |

以上的代码中用到了一个JDK提供的有意义的异常对象，这些都是自己看框架源码学来的，可以借鉴。这样有利于规范，并且知道大概是什么原因导致错误。

编写好JdbcUtil后，后续我们使用该类操作数据库就能减少冗余代码、提高开发效率了，例如3.1节的代码就可更改为：

|  |
| --- |
| **package** com.company;  **import** java.sql.\*;  **public class** Main {   **public static void** main(String[] args) {  Connection connection = **null**;  PreparedStatement statement = **null**;  ResultSet rs = **null**;  **try** {  connection = JdbcUtil.*getConnection*();  statement = connection.prepareStatement(**"select id, name from student;"**);  rs = statement.executeQuery();  **while** (rs.next()) {  System.***out***.println(rs.getString(1));  System.***out***.println(rs.getString(2));  }  } **catch** (Exception e) {  **throw new** RuntimeException(e);  } **finally** {  JdbcUtil.*release*(connection, statement, rs);  }  } } |

# 4. 改写用户注册和登录案例

之前学完Servlet时做了一个用户注册和登录的小案例，采用了MVC和三层架构，但其中的数据是保存在XML文件中的。现在我们要把该小案例改为数据库版本的项目，即数据保存在MySQL数据库中。之前的项目上传在了Github上，地址：https://github.com/zhang13690/SimpleJavaWebDemo。

再回顾一下三层架构：

在真实的企业项目中，为了便于项目管理和维护，一般采用三层架构，使得项目的结构清晰可观，易于扩展维护。三层架构就是“数据访问层”、“业务逻辑层”和“表示层”。各层各司其职，分工明确，实现了“高内聚、低耦合”。

基于以前的工作上，我们要做以下工作：

（1）在数据库中创建user表。按照需要，建表的SQL如下：

|  |
| --- |
| *-- 用户表* **CREATE TABLE** `user` (  **`name` varchar**(100), *# 姓名* **`email` varchar**(100), *# 邮箱* **`birthday` datetime**, *# 生日* **`pwd` varchar**(100) *# 密码* ); |

（2）在项目中引入MySQL的JDBC驱动包（放入WEB-INF目录下），并且在src目录下添加db.properties文件，自行按照之前的方法进行配置。

（3）将JdbcUtil类引入到项目中以便开发（放入util包中）。

那么我们需要开发什么呢？这时我们就能看到三层架构的好处，由于我们只需要将数据换成用数据库存储，其他部分是不需要改动的，因此我们只要再编写一套数据库的DAO实现即可。

此时，我们只要在com.webdemo.dao.impl中新建一个“UserDaoDBImpl”类，实现UserDao接口即可，逻辑和之前的XML实现方式类似。同时这里如果遇到异常，抛出时会使用一个自定义的DaoException（该异常自己定义），这样出现异常时很明确地知道哪里出错了。

因此代码如下：

|  |
| --- |
| **package** com.webdemo.dao.impl;  **import** com.webdemo.dao.UserDao; **import** com.webdemo.domain.User; **import** com.webdemo.exception.DaoException; **import** com.webdemo.util.JdbcUtil;  **import** java.sql.Connection; **import** java.sql.PreparedStatement; **import** java.sql.ResultSet;  **public class** UserDaoDBImpl **implements** UserDao {  @Override  **public** User findUser(String name) {  Connection conn = **null**;  PreparedStatement stmt = **null**;  ResultSet rs = **null**;  **try** {  conn = JdbcUtil.*getConnection*();  stmt = conn.prepareStatement(**"SELECT** *\** **FROM `user` WHERE name = ?;"**);  stmt.setString(1, name);  rs = stmt.executeQuery();  User user = **null**;  **if** (rs.next()) { *// 只有一条记录，使用if即可，不用while* user = **new** User();  user.setName(rs.getString(**"name"**));  user.setEmail(rs.getString(**"email"**));  user.setBirthday(rs.getDate(**"birthday"**));  user.setPwd(rs.getString(**"pwd"**));  }  **return** user;  } **catch** (Exception e) {  **throw new** DaoException(e);  } **finally** {  JdbcUtil.*release*(conn, stmt, rs);  }  }   @Override  **public** User findUser(String name, String pwd) {  Connection conn = **null**;  PreparedStatement stmt = **null**;  ResultSet rs = **null**;  **try** {  conn = JdbcUtil.*getConnection*();  stmt = conn.prepareStatement(**"SELECT** *\** **FROM `user` WHERE name = ? and pwd = ?;"**);  stmt.setString(1, name);  stmt.setString(2, pwd);  rs = stmt.executeQuery();  User user = **null**;  **if** (rs.next()) {  user = **new** User();  user.setName(rs.getString(**"name"**));  user.setEmail(rs.getString(**"email"**));  user.setBirthday(rs.getDate(**"birthday"**));  user.setPwd(rs.getString(**"pwd"**));  }  **return** user;  } **catch** (Exception e) {  **throw new** DaoException(e);  } **finally** {  JdbcUtil.*release*(conn, stmt, rs);  }  }   @Override  **public void** saveUser(User user) {  Connection conn = **null**;  PreparedStatement stmt = **null**;  **try** {  conn = JdbcUtil.*getConnection*();  stmt = conn.prepareStatement(**"INSERT INTO `user` VALUES (?,?,?,?);"**);  stmt.setString(1, user.getName());  stmt.setString(2, user.getEmail());  stmt.setDate(3, **new** java.sql.Date(user.getBirthday().getTime()));  stmt.setString(4, user.getPwd());  stmt.executeUpdate();  } **catch** (Exception e) {  **throw new** DaoException(e);  } **finally** {  JdbcUtil.*release*(conn, stmt);  }  } } |

这样只需要再将UserServiceImpl类中维护的UserDao成员变量的实现类，改为采用UserDaoDBImpl类实现即可。这样运行项目，就会使用数据库保存数据了。除此之外，无需任何代码变动。

这里我们看到使用分层的好处。下面我们将看到使用接口的好处。

在上面，我们改变系统使用的DAO层实现时，还要改变Service中的代码，我们希望无需改变代码，使用配置文件就能决定系统用哪个DAO实现，而不用重新编译源文件。例如src下有dao.properties配置文件，内容为：

|  |
| --- |
| **userDaoClass**=**com.webdemo.dao.impl.UserDaoDBImpl** |

该配置表名系统DAO层使用UserDaoDBImpl实现。如果改变这个配置，则DAO实现也变化。要想实现这个功能，可以利用工厂来实现。对于UserServiceImpl中来讲，它所需要的只是一个实现了UserDao接口的类的对象，如果一个有个工厂返回的对象类型实现了UserDao接口，它就能被UserServiceImpl使用。这体现了接口的好处，当然面向接口编程也有利于各层的同步开发，方便协作，因为接口就像一套预先定义好的规范，只要大家按照这个规范来做事就能最终很好地完成。然后工厂中根据配置文件创建Dao层对象返回即可。

工厂类BeanFactory代码如下：

|  |
| --- |
| **package** com.webdemo.util;  **import** com.webdemo.dao.UserDao;  **import** java.io.InputStream; **import** java.util.Properties;  **public class** BeanFactory {   **private** BeanFactory() {}   **private static** String *userDaoClass*;  **static** {  **try** {  *// 读取配置文件* InputStream inputStream = BeanFactory.**class**.getClassLoader().getResourceAsStream(**"dao.properties"**);  Properties properties = **new** Properties();  properties.load(inputStream);  *userDaoClass* = properties.getProperty(**"userDaoClass"**);  } **catch** (Exception e) {  **throw new** ExceptionInInitializerError(e);  }  }   */\*\*  \* 根据配置文件返回创建的UserDao对象  \** ***@return*** *UserDao对象  \*/* **public static** UserDao getUserDao() {  **try** {  **return** (UserDao) Class.*forName*(*userDaoClass*).newInstance();  } **catch** (Exception e) {  **throw new** RuntimeException(e);  }  } } |

UserServiceImpl中只要改成这样：

|  |
| --- |
| *// 维护UserDao成员变量* UserDao **userDao** = BeanFactory.*getUserDao*(); |

就能“一劳永逸”地维护UserDao。

当然，如果Controller也想这样维护Service对象，也可利用这样的方式。利用这样的工厂模式实现了Service层和DAO层的解耦，也能继续实现Controller和Service的解耦，使得上层不依赖具体的下层实现。

实际上，上面的做法其实可以说是Spring的“核心”。Spring是以后我们要重点学习的Java应用框架。Spring的核心是IOC，即控制反转，就是控制权的反转。例如上面的例子，本来Service层对DAO对象的依赖是通过我们自行创建对象解决了，而使用工厂后，对于Service来说，是工厂给它提供了一个对象，而不是自己创建的，这就“控制反转”了。学习完Spring后，会理解地更深刻。只是在Spring中，上面的功能不是通过简单的工厂类，而是使用一些XML配置和注解来实现。

本案例的代码仍然上传到Github上，该数据库版本的网址是：“https://github.com/zhang13690/SimpleJavaWebDemo/tree/database”。