* JDBC原理

1.1. JDBC标准

**1.1.1. JDBC是什么**

Java Database Connectivity：Java访问数据库的解决方案。

JDBC是Java应用程序访问数据库的里程碑式解决方案。Java研发者希望用相同的方式访问不同的数据库，以实现与具体数据库无关的Java操作界面。

JDBC定义了一套标准接口，即访问数据库的通用API，不同的数据库厂商根据各自数据库的特点去实现这些接口

**1.1.2. JDBC接口及数据库厂商实现**

JDBC中定义了一些接口：

**1.1.3. JDBC工作原理**

JDBC只定义接口，具体实现由各个数据库厂商负责。

程序员使用时只需要调用接口，实际调用的是底层数据库厂商的实现部分。

* JDBC访问数据库的工作过程：

1.加载驱动，建立连接

2.创建语句对象

3.执行SQL语句

4.处理结果集

5.关闭连接

**1.1.4. Driver接口及驱动类加载**

要使用JDBC接口，需要先将对应数据库的实现部分（驱动）加载进来。

驱动类加载方式（Oracle）：

01.Class.forName("oracle.jdbc.driver.OracleDriver");

这条语句的含义是：装载驱动类，驱动类通过static块实现在DriverManager中的“自动注册”。

**1.1.5. Connection接口**

Connection接口负责应用程序对数据库的连接，在加载驱动之后，使用url、username、password三个参数，创建到具体数据库的连接。

01.Class.forName("oracle.jdbc.OracleDriver")

02.//根据url连接参数，找到与之匹配的Driver对象，调用其方法获取连接

03.Connection conn = DriverManager.getConnection(

04."jdbc:oracle:thin:@192.168.0.26:1521:tarena",

05."openlab","open123");

需要注意的是:Connection只是接口，真正的实现是由数据库厂商提供的驱动包完成的。

**1.1.6. Statement接口**

Statement接口用来处理发送到数据库的SQL语句对象，通过Connection对象创建。主要有三个常用方法：

01.Statement stmt=conn.createStatement();

02.//1.execute方法，如果执行的sql是查询语句且有结果集则返回true，如果是非查询语句或者没有结果集，返回false

03.boolean flag = stmt.execute(sql);

04.//2.执行查询语句，返回结果集

05.ResultSetrs = stmt.executeQuery(sql);

06.//3.执行DML语句，返回影响的记录数

07.int flag = stmt.executeUpdate(sql)

**1.1.7. ResultSet接口**

执行查询SQL语句后返回的结果集，由ResultSet接口接收。

常用处理方式：遍历 / 判断是否有结果（登录）。

01.String sql = "select \* from emp";

02.ResultSetrs = stmt.executeQuery(sql);

03.while (rs.next()) {

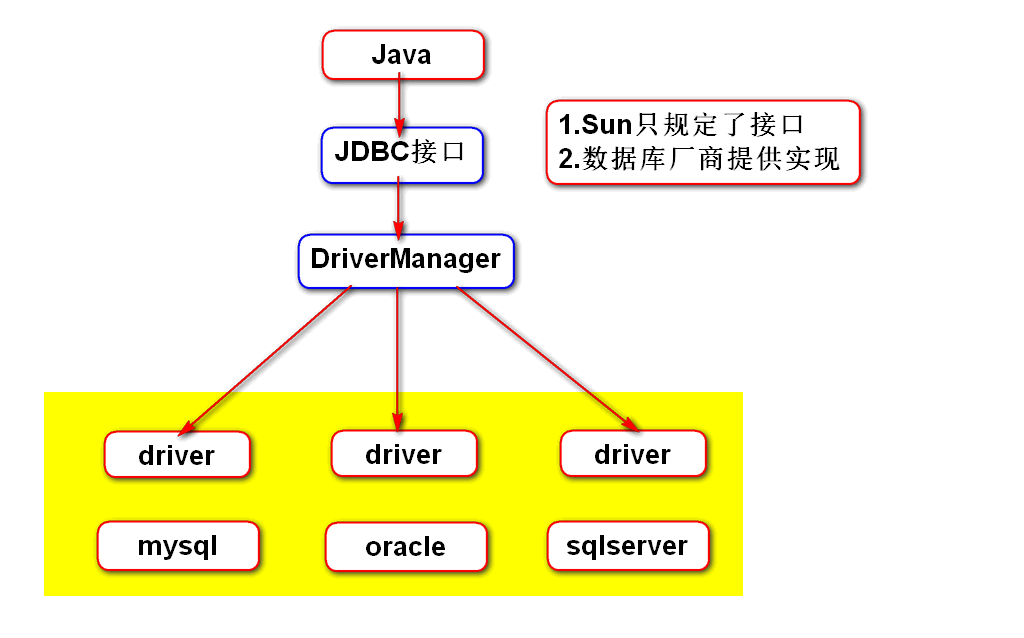
04. System.out.println(rs.getInt("empno")+",“

05. +rs.getString("ename") );

06.}

查询的结果存放在ResultSet对象的一系列行中，指针的最初位置在行首，使用next()方法用来在行间移动，getXXX()方法用来取得字段的内容。

* JDBC的要素：



1.2. 数据库厂商实现

**1.2.1. Oracle实现**

在Java程序中访问不同数据库，需要下载对应数据库的驱动。Oracle数据库提供的驱动为ojdbc6.jar或者ojdbc14.jar，在开发时需要将驱动类加载到项目中，通过设置MyEclipse的Build Path选项。

使用时就可以如下方式加载驱动类了：

01.Class.forName("oracle.jdbc.driver.OracleDriver");

**1.2.2. MySQL实现**

MySQL对应的数据库驱动名为mysql-connector-java-5.0.4-bin.jar（不同版本可能有不同名称），将驱动类加载到项目中同样通过设置MyEclipse的Build Path选项。

加载驱动类的方式：

01.Class.forName("com.mysql.jdbc.Driver");

* JDBC基础编程

2.1. 连接管理

**2.1.1. 通过连接工具类获取连接**

在工程中，通常编写一个访问数据库的工具类，此后所有访问数据库的操作，都从工具类中获取连接。

实现工具类的两种方式：

•直接把数据配置写在工具类。

•把数据库配置写在一个properties属性文件里，工具类读入属性文件，逐行获取数据库参数。

建议使用第二种。

**2.1.2. 通过属性文件维护连接属性**

* **db.properties的内容：**

01.#驱动类名

02.jdbc.driver=oracle.jdbc.driver.OracleDriver

03.#连接字符串

04.jdbc.url=jdbc:oracle:thin:@192.168.0.26:1521:tarena

05.#访问数据库的用户名

06.jdbc.user=openlab

07.#访问数据库的密码

08.jdbc.password=open123

注意：在properties文件中，#符号表示注释。

**2.1.3. 从类路径中加载属性文件**

定义好db.properties之后，需要在Java程序中找到它，可以使用从类路径加载的方式：

01.//属性文件所在的位置

02.String path = "com/tarena/dms/daodemo/v2/db.properties";

03.//获得当前类的路径，加载指定属性文件

04.properties.load(DBUtility.class.getClassLoader()

05..getResourceAsStream(path))

**2.1.4. 连接的关闭**

在工具类中定义公共的关闭连接的方法，所有访问数据库的应用，共享此方法。当完成功能，关闭连接。

01.protected static void closeConnection(Connection con) {

02. if (con != null) {

03. try {

04. con.close();

05. } catch (SQLException e) {

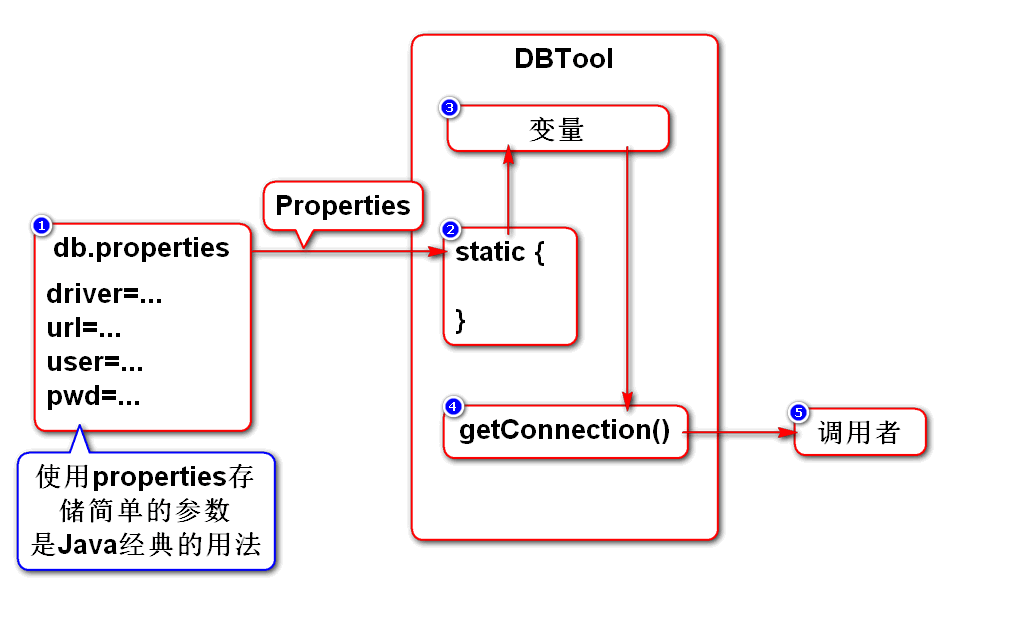
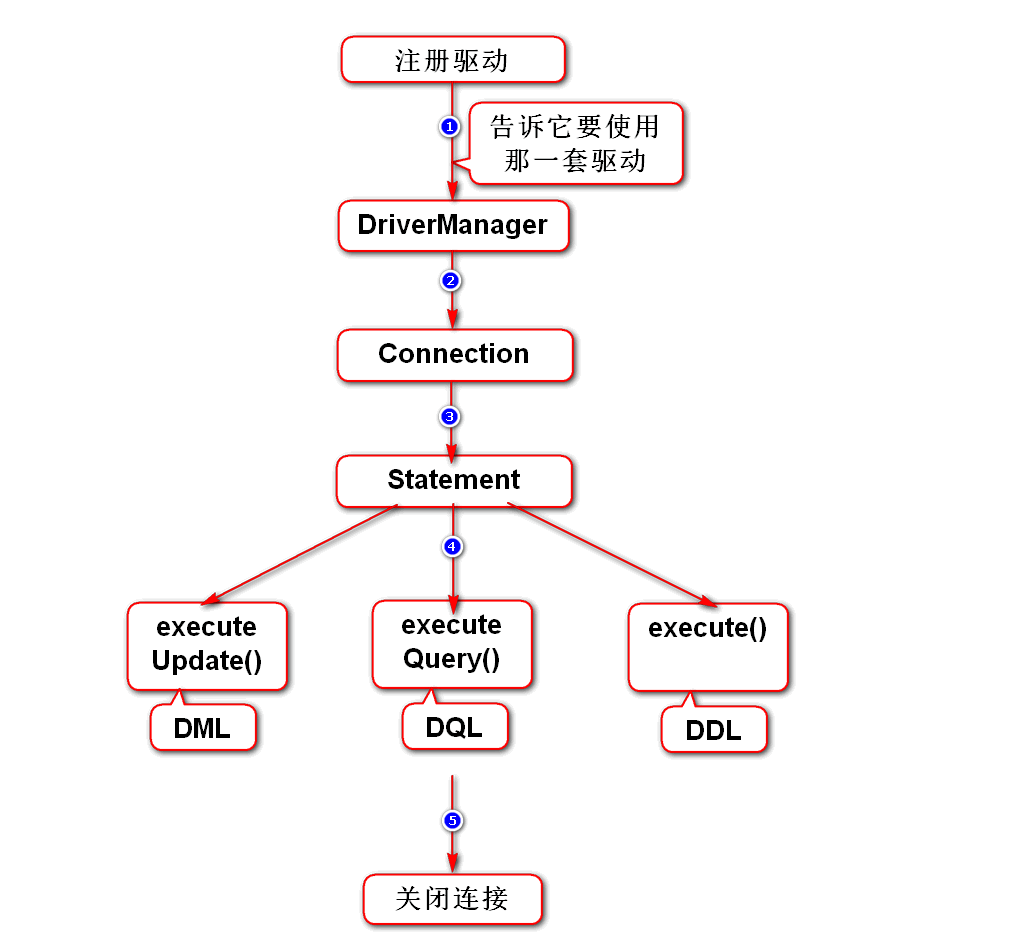
06.e.printStackTrace();

07. }

08. }

09. }

* JDBC的使用步骤： 管理连接：



2.2. 连接池技术

**2.2.1. 为什么要使用连接池**

数据库连接的建立及关闭资源消耗巨大。传统数据库访问方式：一次数据库访问对应一个物理连接,每次操作数据库都要打开、关闭该物理连接, 系统性能严重受损。

解决方案：数据库连接池（Connection Pool）。

系统初始运行时，主动建立足够的连接，组成一个池.每次应用程序请求数据库连接时，无需重新打开连接，而是从池中取出已有的连接，使用完后，不再关闭，而是归还。

* 连接池中连接的释放与使用原则

•应用启动时，创建初始化数目的连接

•当申请时无连接可用或者达到指定的最小连接数，按增量参数值创建新的连接

•为确保连接池中最小的连接数的策略：

1.动态检查：定时检查连接池，一旦发现数量小于最小连接数，则补充相应的新连接，保证连接池正常运转

2.静态检查：空闲连接不足时，系统才检测是否达到最小连接数

•按需分配，用过归还，超时归还

连接池也只是JDBC中定义的接口，具体实现由厂商实完成。

**2.2.2. 使用Apache DBCP连接池**

DBCP(DataBase connection pool)：数据库连接池，是Apache的一个 Java 连接池开源项目，同时也是 Tomcat 使用的连接池组件。相当于是Apache开发的针对连接池接口的一个实现方案。

连接池是创建和管理连接的缓冲池技术，将连接准备好被任何需要它们的应用使用。

图-4连接池在数据访问中的应用

使用Apache DBCP需要两个jar包文件：

•commons-dbcp-1.4.jar 连接池的实现

•commons-pool-1.5.jar 连接池实现的依赖库

将上述两个文件在MyEclipse的Build Path选项中导入到项目。

**2.2.3. 通过DataSource获取连接**

先通过属性文件获取连接池参数，然后加载这些参数，获得连接：

01.//创建数据源对象

02.private static BasicDataSourcedataSource = new BasicDataSource();

03.//加载参数

04.dataSource.setDriverClassName(driveClassName);

05.dataSource.setUrl(url);

06.dataSource.setUsername(username);

07.dataSource.setPassword(password);

08.//获得连接

09.Connection conn = dataSource.getConnection();

**2.2.4. 连接池参数**

常用参数有：

•初始连接数

•最大连接数

•最小连接数

•每次增加的连接数

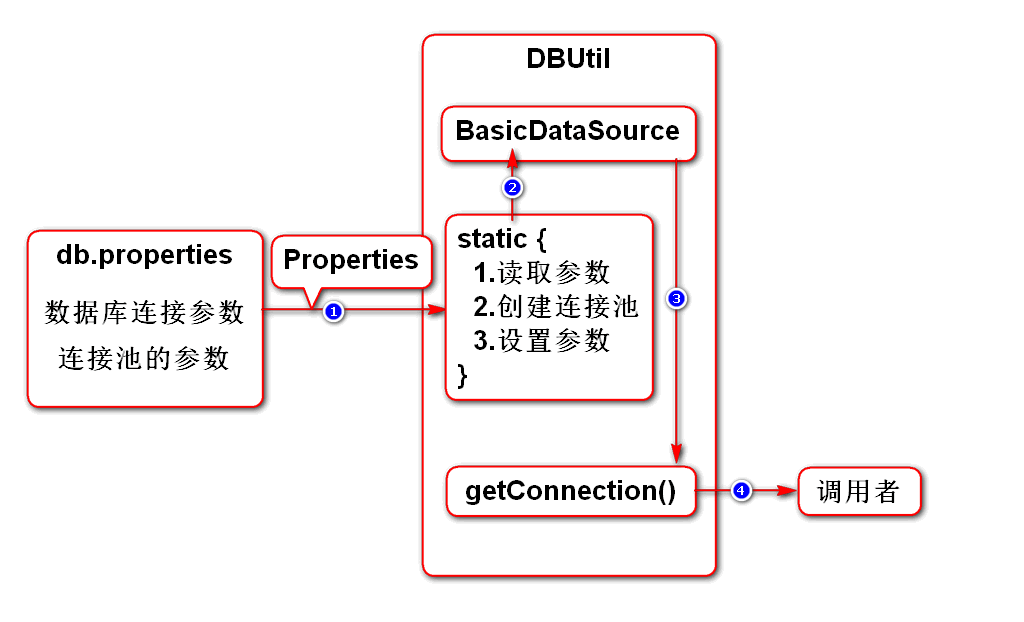
•超时时间

•最大空闲连接

•最小空闲连接

根据应用需要，设置合适的值。

* 如何使用连接池图示：



2.3. 异常处理

**2.3.1. SQLException简介**

Java.sql.SQLException是在处理JDBC时常见的exception对象，用来表示JDBC操作过程中发生的具体错误;

一般的SQLException都是因为操作数据库时出错 , 比如Sql语句写错，或者数据库中的表或数据出错。

常见异常：

•登录被拒绝

•可能原因：程序里取键值对信息时的大小写和属性文件中不匹配

•列名无效

•可能原因：查找的表和查找的列不匹配

•无效字符

•可能原因：SQL语句语法有错 , 比如语句结尾时不能有分号

•无法转换为内部表示

•可能原因：结果集取数据时注意数据类型。

•表或者视图不存在

•检查SQL中的表名是否正确

•不能将空值插入

•检查执行insert操作时，是否表有NOT NULL约束，而没有给出数据

•缺少表达式

•检查SQL语句的语法

•SQL 命令未正确结束

•检查SQL语句的语法

•无效数字：

•企图将字符串类型的值填入数字型而造成，检查SQL语句

其他可能出现的异常

•文件找不到

•可能原因：db.properties文件路径不正确

注意: 新增数据后务必要commit, 否则Java程序运行查询后找不到数据。

2.3.2. 处理SQLException

01.try {

02. //语句，省略

03.} catch (SQLException e) {

04. e.printStackTrace();//追踪处理

05. //throw new RuntimeException(e);//或者抛出

06.}

* JDBC核心API

1.1. Statement

**1.1.1. Statement执行查询**

通过Connection对象创建Statement的方式：

01.Connection.createStatement();

执行INSERT, UPDATE和DELETE等DML操作：

01.Statement.executeUpdate();

执行SELECT：

01.Statement.executeQuery();

通过Statement对象返回SQL语句执行后的结果集:

01.String sql = "select empno, ename, sal, hiredate from emp";

02.Statement stmt = con.createStatement();

03.ResultSetrs = stmt.executeQuery(sql);

04.//对rs的处理

05.//代码略

06.stmt.close();

**1.1.2. Statement执行插入**

Statement.executeUpdate(sql)方法将返回SQL语句执行后影响的记录数:

01.String sql = "insert into emp(empno, ename, job, sal) values(1001, ‘张三丰’,‘Manager’9500)";

02.int flag = -1；

03.try {

04. con = ConnectionSource.getConnection();

05. stmt = con.createStatement();

06. flag = stmt.executeUpdate(sql);

07. //处理结果

08.}catch(SQLException e){

09. //处理异常

10.}

**1.1.3. Statement执行更改**

和INSERT操作完全相同，只是SQL语句不同：

01.String sql = "update emp set sal = 9900 where empno = 1001";

02.int flag = -1；

03.try {

04. con = ConnectionSource.getConnection();

05. stmt = con.createStatement();

06. flag = stmt.executeUpdate(sql);

07. //处理结果

08.}catch(SQLException e){

09. //处理异常

10.}

1.2. PreparedStatement

**1.2.1. PreparedStatement原理**

Statement主要用于执行静态SQL语句，即内容固定不变的SQL语句。Statement每执行一次都要对传入的SQL语句编译一次，效率较差。

某些情况下，SQL语句只是其中的参数有所不同，其余子句完全相同，适用于PreparedStatement。

PreparedStatement的另外一个好处就是预防sql注入攻击。

PreparedStatement是接口，继承自Statement接口。

使用PreparedStatement时，SQL语句已提前编译，三种常用方法 execute、executeQuery和executeUpdate已被更改，以使之不再需要参数。

PreparedStatement实例包含已事先编译的 SQL 语句，SQL 语句可有一个或多个 IN 参数，IN参数的值在 SQL 语句创建时未被指定。该语句为每个 IN 参数保留一个问号（“？”）作为占位符。

每个问号的值必须在该语句执行之前，通过适当的setInt或者setString等方法提供。

由于PreparedStatement对象已预编译过，所以其执行速度要快于 Statement 对象。因此，多次执行的 SQL 语句经常创建为PreparedStatement对象，以提高效率。

通常批量处理时使用PreparedStatement。

01.//SQL语句已发送给数据库，并编译好为执行作好准备

02.PreparedStatement pstmt = con.prepareStatement(

03. "UPDATE emp SET job= ? WHERE empno= ?");

04.//对占位符进行初始化

05.pstmt.setLong(1, "Manager");

06.pstmt.setInt(2,1001);

07.//执行SQL语句

08.pstmt.executeUpdate();

**1.2.2. 通过PreparedStatement提升性能**

在数据库执行SQL语句过程中，制定执行计划开销巨大。

数据库本身具备SQL缓存功能，可以对statement的执行计划进行缓存，以免重复分析。其缓存原理是使用statement本身作为key并将执行计划存入与statement对应的缓存中，对曾经执行过的statements，再运行时执行计划将重用。

举例：

01.SELECT a, b FROM t WHERE c = 1；

再次向数据库发送相同的statement时，数据库会对先前使用过的执行计划进行重用，降低开销。

但是，如下两条语句被视作不同的SQL语句,执行计划不可重用：

01.SELECT a, b FROM t WHERE c = 1；

02.SELECT a, b FROM t WHERE c = 2；

这就是为什么要使用PreparedStatement：

01.//执行计划可重用

02.String sql ="select a,b from t where c = ?";

03.PreparedStatementps = conn.prepareStatement(sql);

04.for (inti = 0; i< 1000; i++) {

05.ps.setInt(1, i);

06.ResultSetrs = ps.executeQuery();

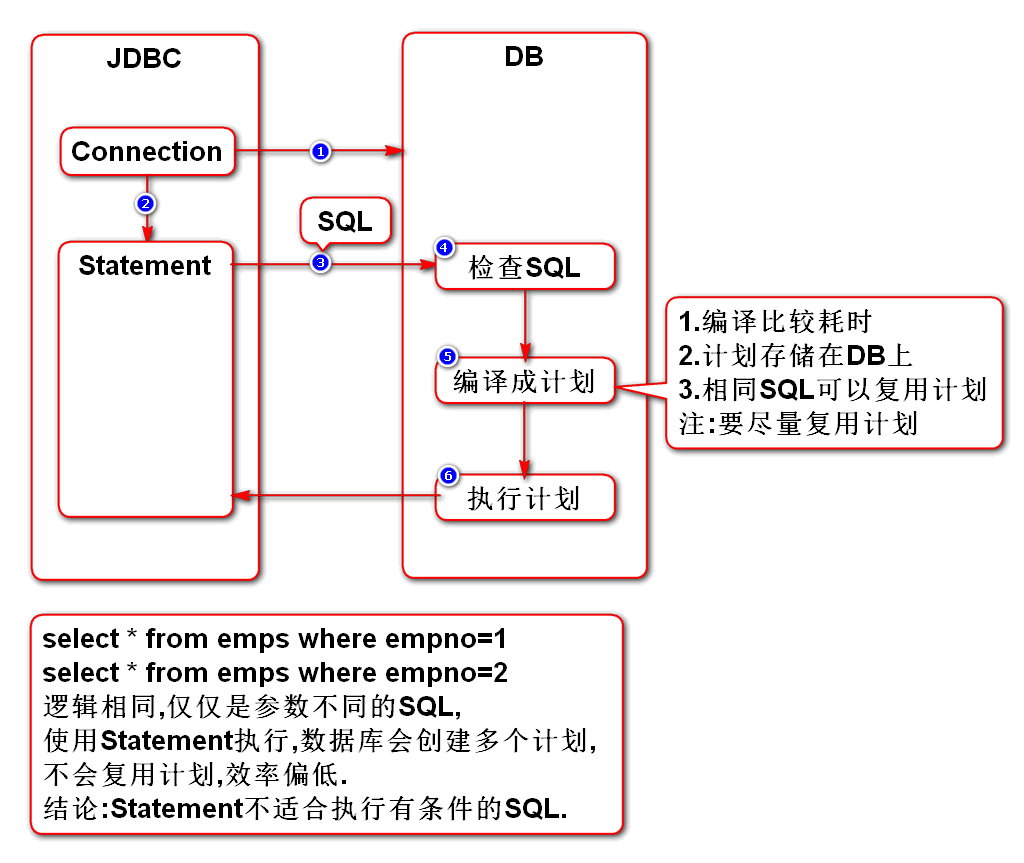
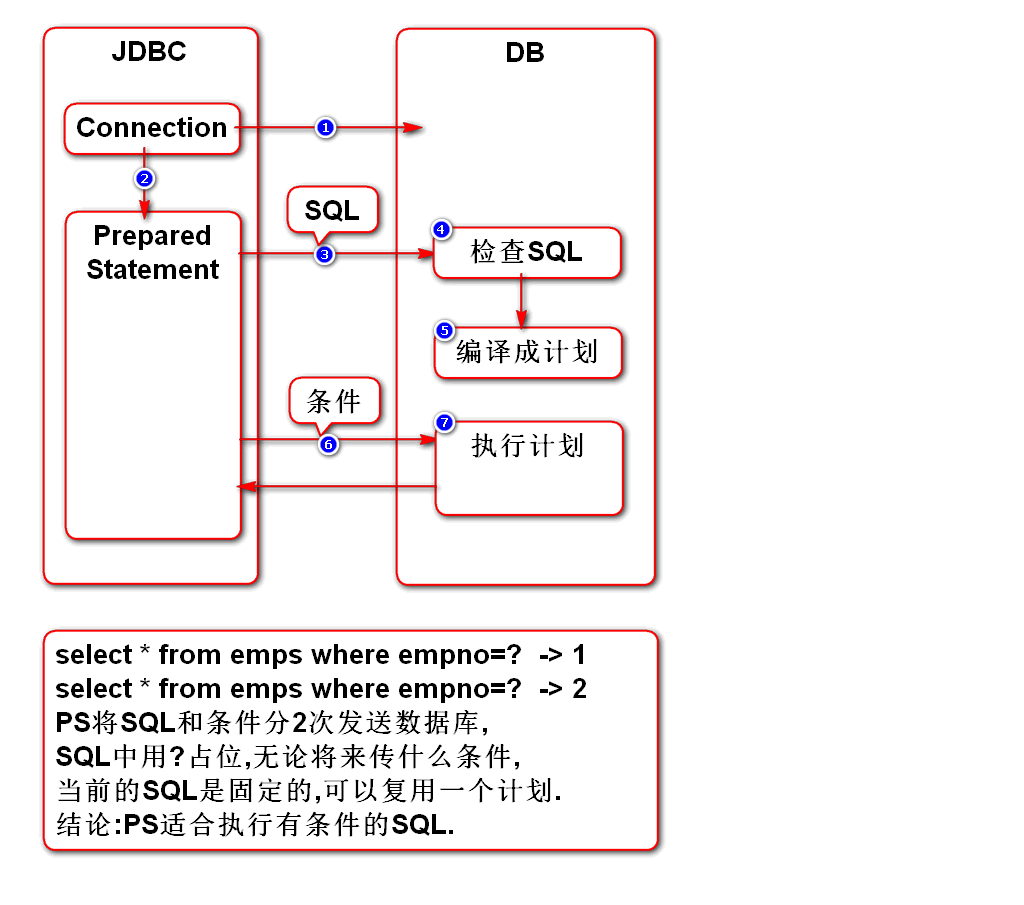
07. //处理rs，省略

08.rs.close();

09.}

10.ps.close();

* Statement和PreparedStatement原理：

**1.2.3. SQL Injection简介**

场景：如下SQL语句被发送到数据库中：

01.String sql = "select \* from t where username = '" + name + "' and password = '" + passwd + "'";

输入用户名和密码参数后，数据库接受到的完整sql语句将是这种形式：

01.select \* from t where username = 'scott' and password = 'tiger';

如果用户输入的passwd参数是：' or '1'='1, 则数据库收到的SQL语句将是：

01.select \* from t where username = 'scott' and password = '' or '1'='1’;

此SQL语句的where条件将永远为true。即用户不需要输入正确的帐号密码，也能登录。这种现象称作SQL注入（SQL Injection）。

**1.2.4. 通过PreparedStatement防止SQL Injection**

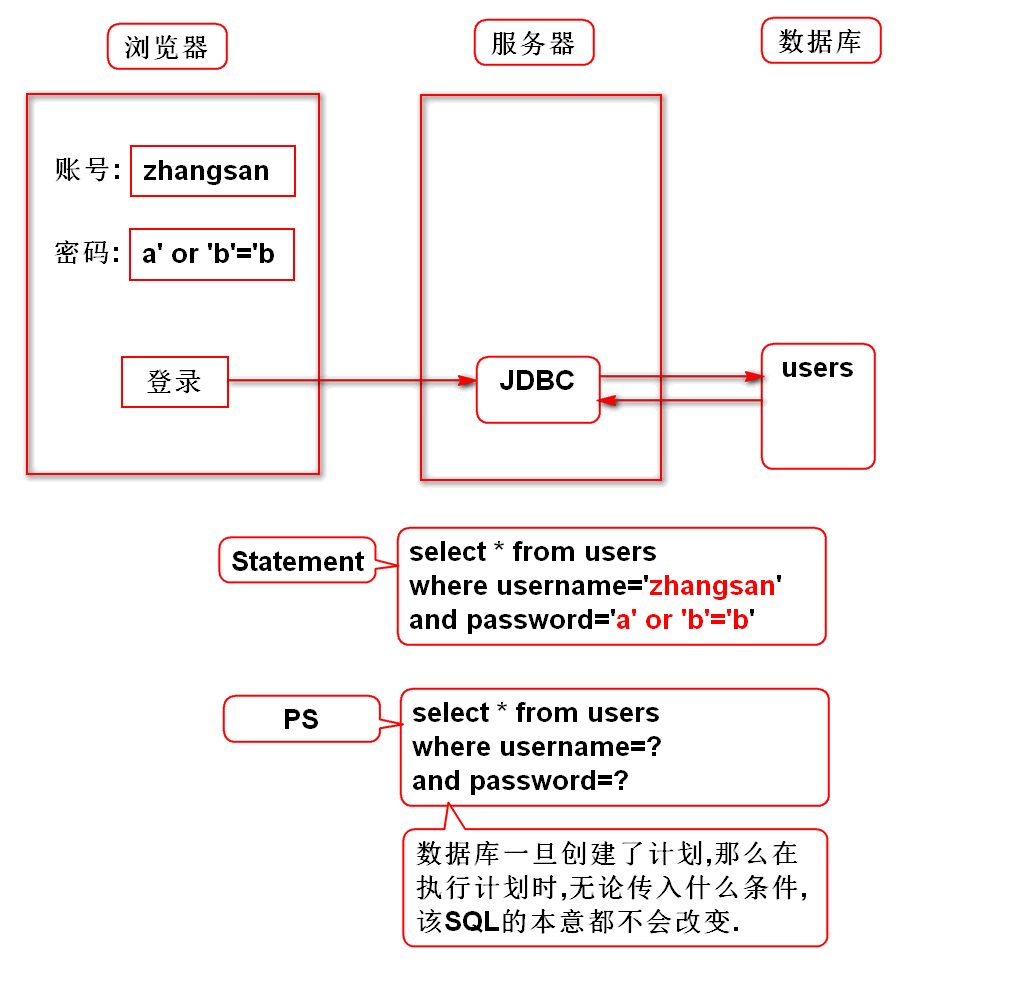
对JDBC而言，SQL注入攻击只对Statement有效，对PreparedStatement无效，因为PreparedStatement不允许在插入参数时改变SQL语句的逻辑结构。

使用预编译的语句对象时，用户传入的任何数据不会和原SQL语句发生匹配关系，无需对输入的数据做过滤。

如果用户将' or '1'='1传入赋值给占位符，下述SQL语句将无法执行：

1. select \* from t where username = ? and password = ?;

* PS可以避免注入攻击图示：



1.3. ResultSet

**1.3.1. 结果集遍历**

结果集常用的遍历方式（使用rs.getXXX方法）：

01.String sql = "select empno, ename, sal, hiredate from emp";

02. rs = stmt.executeQuery(sql);

03. while (rs.next()) {

04. intempno = rs.getInt("empno");

05. String ename = rs.getString("ename");

06. doublesal = rs.getDouble("sal");

07.Date hiredate = rs.getDate("hiredate");

08. }

09.rs.close();

**1.3.2. ResultSetMetaData**

ResultSetMetaData: 数据结果集的元数据，和查询出来的结果集相关,从结果集(ResultSet)中获取。

下列代码获得ResultSetMetaData对象后，从中获取数据表的所有列名：

01.ResultSetMetaDatarsm = rs.getMetaData();

02.intcolumnCount = rsm.getColumnCount();

03.String columnName = null;

04.for (int i = 1; i<=columnCount; i++) {

05. columnName = rsm.getColumnName(i);

06.}

**1.3.3. 可滚动结果集**

常用的ResultSet，返回后，其初始指针在第一行之前（Before First），并且只能使用next()方法将指针向后移动，不能反向，一次移动一行，不能跳行。

可滚动的结果集：指针可以在结果集中任意移动。使用在需要指针移动的场合，比如分页。

获得可滚动的ResultSet，Statement或者PreparedStatement的创建有所不同：

01.Statement stmt = conn.createStatement(type, concurrency);

02.PreparedStatementstmt = conn.prepareStatement(sql, type, concurrency);

其中type取值：

•TYPE\_FORWARD\_ONLY：只能向前移动，默认参数

•TYPE\_SCROLL\_INSENSITIVE：可滚动，不感知数据变化

•TYPE\_SCROLL\_SENSITIVE：可滚动，感知数据变化

concurrency取值：

•CONCUR\_READ\_ONLY：只读

•CONCUR\_UPDATABLE：可更新

获得可滚动结果集后，常用方法如下：

•first：指针移动到第一条

•last：指针移动到最后一条

•beforeFirst：指针移动到第一条之前

•afterLast：指针移动到最后一条之后

•isFirst：判断指针是否指向第一条

•isLast：判断指针是否指向最后一条

•isBeforeFirst：判断指针是否在第一条之前

•isAfterLast：判断指针是否在最后一条之后

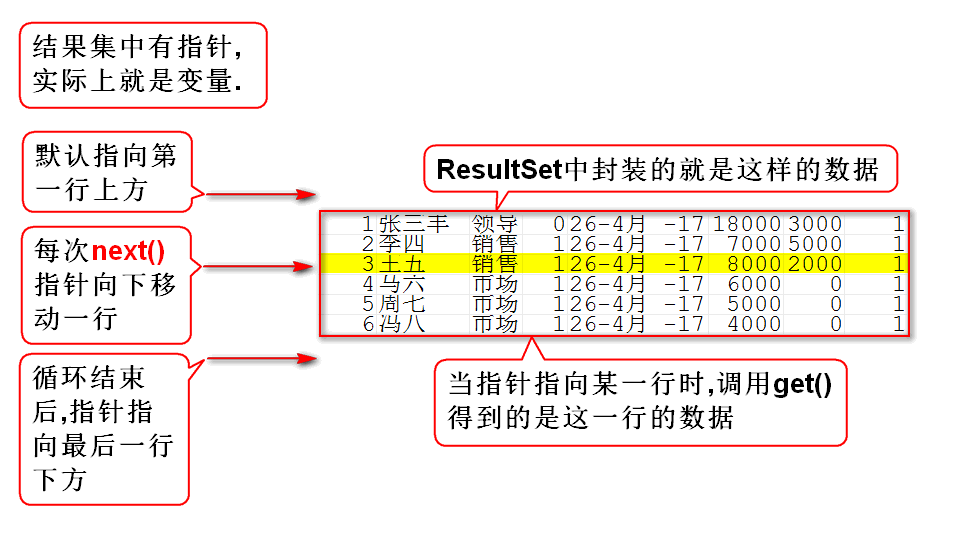
•relative：移动到当前指针的相对位置

•next：移动到下一条

•previous：移动到前一条

•absolute：移动到绝对位置

* 关于结果集的指针图示：



* JDBC高级编程

1.1.JDBC高级编程

**1.1.1. 事务简介**

事务（Transaction）：数据库中保证交易可靠的机制。JDBC支持数据库中的事务概念，并且在JDBC中，事务默认是自动提交的。

可能存在的问题：某些业务逻辑必须在一个事务中完成。如图-1所示。

图-1事务示意图

事务特性介绍：ACID

•原子性（Atomicity）：事务必须是原子工作单元；对于其数据修改，要么全都执行，要么全都不执行

•一致性（Consistency）：事务在完成时，必须使所有的数据都保持一致状态

•隔离性（Isolation）：由并发事务所作的修改必须与任何其它并发事务所作的修改隔离

•持久性（Durability）：事务完成之后，它对于系统的影响是永久性的

事务是数据库的概念，JDBC支持事务，本质还是在数据库中实现的。

**1.1.2. JDBC事务API**

JDBC中和事务相关API：

•Connection.getAutoCommit() :获得当前事务的提交方式，默认为true

•Connection.setAutoCommit():设置事务的提交属性，参数是true：自动提交；false：不自动提交

•Connection.commit():提交事务

•Connection.rollback():回滚事务

**1.1.3. JDBC标准事务编程模式**

JDBC处理事务的通常模式：

•先将事务的自动提交关闭；

•执行事务中的若干SQL语句；

•事务提交；SQL失败则回滚；

•恢复JDBC的事务提交状态；释放资源。

01.try{

02.// 1.定义用于在事务中执行的SQL语句

03.String sql1 = "update account set amount = amount - " + amount + " where id = '" + from + "'";

04.String sql2 = "update account set amount = amount + " + amount + " where id = '" + to + "'";

05.autoCommit = con.getAutoCommit(); // 2.获得自动提交状态

06.con.setAutoCommit(false); // 3.关闭自动提交

07.stmt.executeUpdate(sql1); // 4.执行SQL语句

08.stmt.executeUpdate(sql2);

09.con.commit(); // 5.提交

10.con.setAutoCommit(autoCommit); // 6.将自动提交功能恢复到原来的状态

11.//其他语句;

12.}catch(SQLException e){

13. conn.rollback();//异常时回滚

14.}

1.2.批量更新

**1.2.1. 批量更新的优势**

批处理：发送到数据库作为一个单元执行的一组更新语句。批处理降低了应用程序和数据库之间的网络调用，相比单个SQL语句的处理，批处理更为有效。

**1.2.2. 批量更新API**

•addBatch(String sql)

•Statement类的方法, 可以将多条sql语句添加Statement对象的SQL语句列表中

•addBatch()

•PreparedStatement类的方法, 可以将多条预编译的sql语句添加到PreparedStatement对象的SQL语句列表中

•executeBatch()

•把Statement对象或PreparedStatement对象语句列表中的所有SQL语句发送给数据库进行处理

•clearBatch()

•清空当前SQL语句列表

**1.2.3. 防止OutOfMemory**

如果PreparedStatement对象中的SQL列表包含过多的待处理SQL语句, 可能会产生OutOfMemory错误。所以需要及时处理SQL语句列表。

01.for (inti = 0; i< 1000; i++) {

02.sql = “insert into emp(empno, ename) values(emp\_seq.nextval, 'name" + i + "'")";

03.//将SQL语句加入到Batch中

04.stmt.addBatch(sql);

05.//根据需要设置及时处理的条件

06.if (i % 500 == 0) {

07.stmt.executeBatch(); //及时处理

08.stmt.clearBatch(); //清空列表

09.}

10.}

11.// 最后一次列表不足500条，处理

12.stmt.executeBatch();

1.3. 返回自动主键

**1.3.1. 关联数据插入**

在主表/从表关联关系下，插入数据时需要保证数据完整性。关联数据插入时的流程：

1.3.2. 通过序列产生主键（Oracle）

不同数据库的主键自增方式不同，在oracle中，建议主键通过序列获得。在SQL语句中，指定由哪个序列为表产生主键：

01.String sql =

02.“insert into dept (deptno, dname, loc) values(dept\_seq.nextval,?,?)”；

如果仅仅是单表操作，不需要返回刚刚插入的主键值，但如果有关联关系的表操作，需要获得刚刚插入的主键值。

**1.3.3. JDBC返回自动主键API**

方法一：先通过序列的nextval获取序列的下一个值，再作为参数插入到主表和从表。

01.//1.获得主键的SQL语句

02.String sql = "select dept\_seq.nextval as id from dual";

03.//执行上条SQL语句，通过ResultSet获得主键值，省略

05.//2.利用刚刚得到的主键值插入主表

06.String sql1=“insert into dept(deptno, dname, loc) values(?,?,?)";

07.//执行insert语句，省略

09.//3.利用刚刚得到的主键值，作为外键插入从表

10.String sql2 = “insert into emp(empno, ename, deptno) values(?,?,?)”;

11.//执行insert语句，省略

这种方式操作简单，但需要额外多一次访问数据库，影响性能。

方法二（建议）：利用PreparedStatement的getGeneratedKeys方法获取自增类型的数据，性能良好，只要一次SQL交互。

01.//1.插入主表SQL，使用序列作为主键

02.sql = "insert into dept (deptno, dname, loc) values(dept\_seq.nextval,?,?)";

03.//2.定义stmt时，第二个参数是GeneratedKeys的主键的字段名列表，类型是字符串数组

04.stmt = con.prepareStatement(sql, new String[] { "deptno" });

05.//3.将占位符赋值

06.stmt.setString(1, “Research”);

07.stmt.setString(2, “beijing”);

08.//4.执行插入主表的insert语句

09.stmt.executeUpdate();

10.//5.获得主键值

11.rs = stmt.getGeneratedKeys();

12.rs.next();

13.intdeptno = rs.getInt(1);

14.//6.将刚刚得到的主表主键值，作为外键插入到从表中。

15.String sql2 = “insert into emp(empno, ename, deptno) values(?,?,?)”;

1.4.分页查询

**1.4.1. JDBC实现Oracle分页查询**

利用数据库的分页SQL语句，实现在Java程序中数据表的分页。在Oracle中使用rownum，以获得分页结果集：

01.String sql = "select \* from (select rownumrn, empno, ename, job,mgr, hiredate, sal, comm, deptno from (select \* from emp order by empno) ) where rn between ? and ?";

两个占位符分别是结果集的起点和终点，计算后，替代SQL中的占位符：

01. int begin = (page - 1) \* pageSize + 1;

02. int end = begin + pageSize - 1;

其中：

•page:返回第几页

•pageSize：每页多少条

01.stmt = con.prepareStatement(sql);

02.stmt.setInt(1, begin);

03.stmt.setInt(2, end);

04.rs = stmt.executeQuery();

这种分页方式，每次只向数据库请求一页的数据量，内存压力小适合大数据量数据表。

另一种分页策略介绍：基于缓存的分页技术（也被称作假分页），一次性把数据全部取出来放在缓存中,根据用户要看的页数(page)和每页记录数(pageSize),计算把哪些数据输出显示，将可滚动结果集的指针移动到指定位置。

这种方式只访问数据库一次 , 第一次取数比较慢 , 以后每页都从缓存中取 , 比较快，比较适合小数据量 , 如果数据量大 , 对内存压力较大。

**1.4.2. JDBC实现MySQL分页查询**

不同数据库获取部分结果集的SQL是有区别的，在MySQL中的实现方式：

01. select \* from t limit begin，pageSize;

其中：

•begin: 从第几条开始显示

•pageSize:每页多少条

在Java程序中，MySQL和Oracle分页的方式，仅限于SQL语句不同。

* DAO

2.1. 什么是DAO

**2.1.1. DAO封装对数据的访问**

DAO (Data Access Object)：数据访问对象，是建立在数据库和业务层之间，封装所有对数据库的访问。

目的：数据访问逻辑和业务逻辑分开。

为了建立一个健壮的Java应用，需将所有对数据源的访问操作抽象封装在一个公共API中，需要：

•建立一个接口，接口中定义了应用程序中将会用到的所有事务方法

•建立接口的实现类，实现接口对应的所有方法，和数据库直接交互

在应用程序中，当需要和数据源交互时则使用DAO接口，不涉及任何数据库的具体操作。DAO通常包括：

1. 一个DAO工厂类；

2. 一个DAO接口；

3. 一个实现DAO接口的具体类；

4. 数据传递对象（实体对象（Entity）或值对象（Value Object，简称VO））.

**2.1.2. 实体对象**

DAO层需要定义对数据库中表的访问。

对象关系映射(ORM：Object/Relation Mapping)描述对象和数据表之间的映射，将Java程序中的对象对应到关系数据库的表中：

•表和类对应

•表中的字段和类的属性对应

•记录和对象对应

2.2. 编写DAO

**2.2.1. 查询方法**

01. public Account findById(Integer id) throws SQLException {

02. Connection conn = getConnection();

03. String sql = SELECT\_BY\_ID; //预先定义好的SQL查询语句

04. PreparedStatementps = conn.prepareStatement(sql);

05. ps.setInt(1, id);//传入参数

06. ResultSetrs = ps.executeQuery();

07. Account account = null;

08. while(rs.next()){//处理结果集

09. account = new Account();

10. account.setId(rs.getInt("ACCOUNT\_ID"));

11. //设置account对象所有的属性，略 }

12. return account;

13. }

**2.2.2. 更新方法**

01. publicboolean update(Account account) throws SQLException {

02. Connection conn = getConnection();

03. String sql = UPDATE\_STATUS; //预先定义好的SQL语句

04. PreparedStatementps = conn.prepareStatement(sql);

05. ps.setInt(1,account.getId());//传入参数

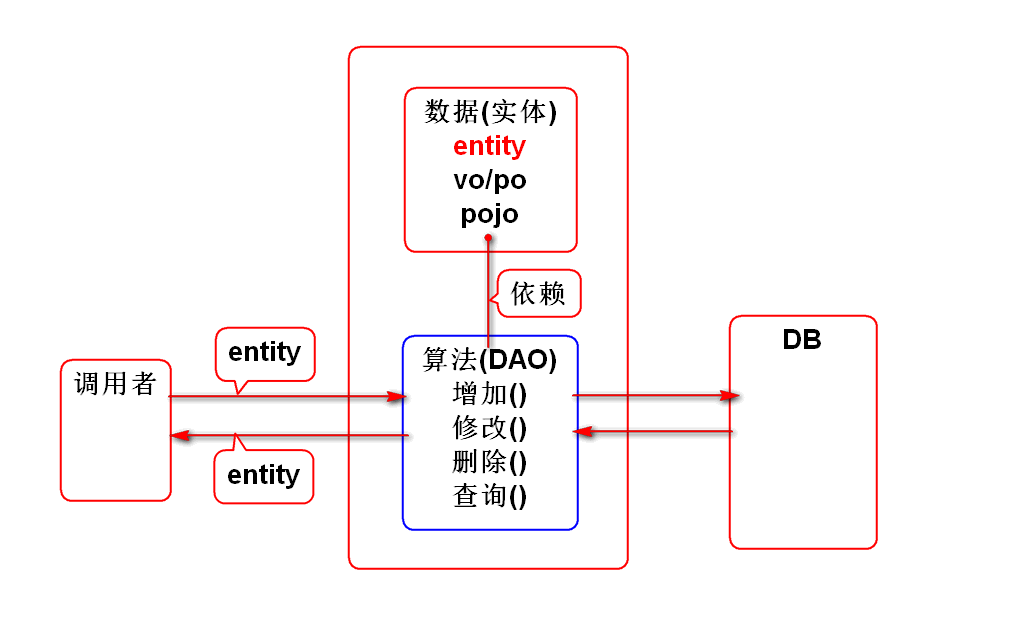
06.ps.setString(2, account.getStatus());

07. int flag= ps.executeUpdate();

08. return (flag>0 ? true : false);

09. }

* 对JDBC封装的思想：



**2.2.3. 异常处理机制**

多层系统的异常处理原则：

•谁抛出的异常，谁捕捉处理，因为只有异常抛出者，知道怎样捕捉处理异常；

•尽量在当前层中捕捉处理抛出的异常，尽量不要抛出到上层接口；

•尽量在每层中封装每层的异常类，这样可准确定位异常抛出的层；

•如异常无法捕捉处理，则向上层接口抛出，直至抛给JVM，尽量避免；

* **Java体系结构图：**

