maven

maven

maven: maven

0.0.1

出版日期 2009-04-28 15:50:31

首页: http://www.mossle.com/

EMAIL: vivian.mossle@gmail.com

EMAIL: lingo.mossle@gmail.com

序言

作者说

jdbc基本概念

虽然有人说JDBC是Java Database Bridge Connection（java数据库桥接）的缩写，但sun公司一直没有承认这种解释。不过jdbc确实是一种桥接方式，所有服务器厂商都为jdbc提供对应自己数据库的驱动，我们只要学会使用jdbc中的类和方法，就可以通过它操作任何一款数据库了。

下面的内容里，我们会详细描述如何使用jdbc连接数据库进行各种操作，同时介绍其中运行的原理。

第#1#章#jdbc的基本操作

jdbc如何配置，如何连接数据，如何查询数据，如何更新数据，如何关闭连接释放资源。

我们例子中使用的是jdk内部默认提供的jdbc:odbc驱动，此驱动可以与windows平台的odbc进行桥接，连接odbc上已配置好的数据库。

1.1.#使用access数据库

> 注意
>
> 如果你以前没有使用过数据库，或者不熟悉如何使用access数据库，那么请继续阅读，否则请跳过此章继续下面的内容。

创建一个access数据库需要一下步骤。

选择：开始 -> 设置 -> 控制面板。

选择：管理工具。

选择：数据源（ODBC）。

进入管理数据库（ODBC）的界面，选择添加。

选择Microsoft Access Driver，点击完成。

在这里输入数据库的名称database，然后点击创建。

输入数据库文件的名称，选择生成到c:盘根目录下，点击确定。

提示数据库创建成功。

可以看到数据库已经创建成功，可以看到生成后数据库文件的完整路径，点击确定。

这是我们可以看到名叫database的数据数据库已经在列表中了，点击确定完成整个步骤。

生成到c:盘根目录下的数据库文件大概像这个这样。

有两种方法可以连接到database.mdb数据库。

1. 可以使用我们在odbc中定义的数据库名称database，通过odbc获得数据库连接。
2. 可以通过Driver={Microsoft Access Driver (*.mdb)};DBQ=c:\database.mdb的写法，通过文件路径获得数据库连接。

第一种方式只要定义了odbc中的数据库，不管实际数据库文件放到哪里都可以直接使用odbc中的名称直接连接。缺点是每次使用之前都要进入odbc对数据源进行定义。

第二种方式不需要进入odbc设置，只要指定数据库文件的路径就可以连接。缺点如果数据库位置变动，就要修改连接代码。

下面我们将使用jdbc操作新生成的database.mdb数据库。

1.2.#第一个jdbc程序

```
import java.sql.DriverManager;
import java.sql.Connection;

public class DbUtils {
    public static void main(String[] args) throws Exception {
        // 加载驱动
        Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
        // 连接数据库的地址
        // String url = "jdbc:odbc:database";
        String url = "jdbc:odbc:Driver={Microsoft Access Driver (*.mdb)};DBQ=database.mdb";

        Connection conn = null;
        try {
            // 创建与数据库的连接
            conn = DriverManager.getConnection(url);
            System.out.println("成功连接到数据库：" + conn);
        } catch(Exception ex) {
            System.out.println("连接失败：" + ex);
        } finally {
            // 关闭连接，释放资源
            if (conn != null) {
                conn.close();
            }
        }
    }
}
```

打开01-02目录可以看到如下内容。

DbUtils.java是包含jdbc连接的程序，我们将使用run.bat这个批处理脚本对它进行编译和运行，run.bat的主要内容如下：

```
javac DbUtils.java
java DbUtils
```

注意

这里要特别注意相对路径的概念，如果我们使用"jdbc:odbc:Driver={Microsoft Access Driver (*.mdb)};DBQ=database.mdb"的连接方式，这里的database.mdb就是指与run.bat在同一个目录下。

运行run.bat运行DbUtils.java得到如下结果。

现在让我们来看看DbUtils.java做了什么。

1. 加载jdbc-odbc驱动。

```
// 加载驱动
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
```

这个sun.jdbc.odbc.JdbcOdbcDriver是一个java类，它是jdk自带的驱动，我们不需要添加其他库就可以使用。

Class.forName()是固定用法，背诵即可。

2. 设置连接数据库的url地址。

```
// 连接数据库的地址
// String url = "jdbc:odbc:database";
String url = "jdbc:odbc:Driver={Microsoft Access Driver (*.mdb)};DBQ=database.mdb";
```

因为我们刚刚设置了名叫database的odbc数据源，使用第一种jdbc:odbc:database也可以连接成功。如果没有设置odbc数据源的情况下，就要用第二种指定具体文件路径的方式，这里连接的是当前目录下的database.mdb文件。

我们可以看到两个url都分成三段，中间用冒号（:）分隔。

开头的jdbc是惯用写法，所有jdbc连接数据库的url地址都是以jdbc:开头的。

中间部分用来决定使用哪个驱动程序，这里写成jdbc:odbc:就会自动调用刚刚加载的jdbc-odbc驱动程序，驱动程序会在加载的时候自动注册别名，使用时可以参考对应文档。

最后部分是驱动程序需要的连接协议，连接地址，数据库名等信息，使用时需要参考对应的文档。

3. 使用DriverManager获得指定url的连接。

```
try {
    // 创建与数据库的连接
    conn = DriverManager.getConnection(url);
    System.out.println("成功连接到数据库：" + conn);
} catch(Exception ex) {
    System.out.println("连接失败：" + ex);
}
```

如果连接成功DriverManager.getConnection()会返回一个Connection对象，如果连接失败会抛出异常，使用try{}catch(){}可以判断是否能够与数据库连接，连接成功的话我们直接打印出成功信息。

4. 最后一步非常关键，一定要在使用之后关闭连接释放资源。

```
} finally {
    // 关闭连接，释放资源
    if (conn != null) {
        conn.close();
    }
}
```

为了确保在正常连接和出现异常的情况下都执行关闭连接的代码，我们使用了finally。记得先要判断conn != null，避免出现NullPointerException。

注意

一定要记得在使用完Connection之后关闭连接，数据库连接需要承载大量的数据传输，它本事也是非常消耗资源的，数据库一般都有最大连接限制，当连接数过多超过限制的时候就会导致连接失败。如果我们没有调用conn.close()关闭连接，这个数据库连接就无法释放，即使你不再使用它，它也会一直占据着资源，最后就会超出最大连接数导致数据库无法响应。

1.3.#创建表，删除表

使用数据库之前，我们首先需要创建表结构才能向表中添加数据，查询结果。

为了便于使用，我们将数据库配置，获取连接，关闭连接的代码封装到一个类中，DbUtils.java的代码如下。

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;

public class DbUtils {

    static {
        try {
            // 加载驱动
            Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
        } catch(ClassNotFoundException ex) {
            throw new RuntimeException("load jdbc-odbc driver error.");
        }
    }

    public static Connection getConnection() throws SQLException {
        // 创建与数据库的连接
        return DriverManager.getConnection("jdbc:odbc:Driver={Microsoft Access Driver (*.mdb)};DBQ=database.mdb
    }

    public static void close(Connection conn) {
        try {
            if (conn != null) {
                // 关闭连接，释放资源
                conn.close();
            }
        } catch(SQLException ex) {
            //
```

```
        }
    }
}
```

static{}叫做静态初始化块，我们依靠它在加载DbUtils.java的同时加载jdbc-odbc驱动。

DbUtils.java提供两个static方法，getConnection()获得数据库连接，close()则用来关闭连接。

下一步，我们在CreateTable.java中编写创建表的代码，需要使用如下的sql语句。

```
create table test(
    id integer,
    name varchar(100)
);
```

sql语句的含义是创建一个名叫test的表，这个表包含两个字段，id字段的类型是integer（整数），name字段的类型是varchar(100)（最大长度为100的字符串）。

CreateTable.java中使用DbUtils来获得连接和关闭连接，代码如下。

```
import java.sql.*;

public class CreateTable {
    public static void main(String[] args) throws Exception {
        Connection conn = DbUtils.getConnection();
        Statement state = null;
        try {
            String sql = "create table test(id integer, name varchar(100))";
            state = conn.createStatement();
            state.executeUpdate(sql);
        } catch(Exception ex) {
            ex.printStackTrace();
        } finally {
            if (state != null) {
                state.close();
            }
            DbUtils.close(conn);
        }
    }
}
```

Connection的部分与上例相同，咱们不再赘述，集中注意在Statement的部分。

执行sql语句创建表结构需要两步，第一步执行conn.createStatement()创建一个Statement，第二步执行state.executeUpdate(sql)执行sql语句，经过这两步数据库里就会多了一张test表，如果安装了visual studio中的access管理器，就可以打开database.mdb看到如下结果。

一个Connection可以创建多个Statement，当我们用用这些Statement执行sql语句时，sql语句会在处理后通过Connection发送给数据库执行，设计良好的jdbc驱动会能够保证conn关闭

的时候，同时释放依附于这个连接的所有Statement，但是为了以防万一，我们还是建议调用state.close()手工关闭Statement，因为这些Statement也会消耗连接资源。

finally里还要注意关闭的顺序，按照Statement -> Connection的顺序关闭，如果先关闭Connection再操作Statement可能会引发异常。

删除表的操作与创建表操作极为相似，实际上只有sql语句不同而已，删除表的sql语句如下：

```
drop table test
```

删除表的代码见01-03/DropTable.java，可以执行dropTable.bat进行编译和执行。

> 注意
>
> 重复建表或者删除不存在的表，都会抛出异常导致操作失败，大家可以多执行几次createTable.bat和dropTable.bat看一下效果。

1.4.#插入，修改，删除数据

实际工作时表结构都是建立好的，我们只需要获取数据库连接，进行CRUD操作（Create创建、Read读取、Update更新和Delete删除）。

我们先把Create创建、Update更新和Delete删除挑出来说说，这三个操作都是修改数据库中的数据，同属于更新（update）操作。Read读取则是从数据库中搜索已有数据，属于查询（query）操作，我们后面单独讲解。

先看看Insert.java中的代码。

```
import java.sql.*;

public class Insert {
    public static void main(String[] args) throws Exception {
        Connection conn = DbUtils.getConnection();
        Statement state = null;
        try {
            String sql = "insert into test(id,name) values(1,'lingirl')";
            state = conn.createStatement();
            state.executeUpdate(sql);
        } catch(Exception ex) {
            ex.printStackTrace();
        } finally {
            if (state != null) {
                state.close();
            }
            DbUtils.close(conn);
        }
    }
}
```

这里也是创建Statement然后执行executeUpdate()，与CreateTable.java，DropTable.java不同的地方就只有sql语句。

```
insert into test(id,name) values(1,'lingirl')
```

这是向数据库中插入一条id=1,name='lingirl'的记录，操作成功后可以在access中看到如下结果。

多执行几次Insert.java数据库中的数据就会一直增加。Update.java与Delete.java中的操作方式也都与Insert.java相同，这里只贴出各自使用的sql语句。

- Update.java

```
update test set name='叮咚' where id=1
```

将id=1的记录对应的name修改为'叮咚'。

- Delete.java

```
delete from test where id=1
```

删除id=1的记录。

1.5.#查询数据库

通过上面的实例，我们已经可以使用jdbc对数据库进行插入，修改，删除操作了，现在我们需要了解数据库中保存了哪些数据，这样我们就需要使用查询（query）操作。

就用刚才的Insert.java向database.mdb多插入几条语句，然后调用Select.java进行查询，查询的结果直接在console控制台中打印出来。

Select.java中的代码如下。

```
import java.sql.*;

public class Select {
    public static void main(String[] args) throws Exception {
        Connection conn = DbUtils.getConnection();
        Statement state = null;
        ResultSet rs = null;
        try {
            String sql = "select * from test";
            state = conn.createStatement();
            rs = state.executeQuery(sql);
            // 显示查询结果
            while (rs.next()) {
                System.out.println(rs.getInt(1) + " " + rs.getString(2));
            }
        } catch(Exception ex) {
            ex.printStackTrace();
```

```
        } finally {
            if (rs != null) {
                rs.close();
            }
            if (state != null) {
                state.close();
            }
            DbUtils.close(conn);
        }
    }
}
```

这次使用到的sql语句是select * from test，作用是从test表里搜索出所有数据来。因为这次是查询（query），所以在Statement上调用executeQuery()方法执行sql语句，并返回一个ResultSet（结果集）。

在得到rs之后，可以通过rs.next()判断其中是否有可供使用的数据，如果为真就执行rs.getInt()或rs.getString()将数据库中的数据取出来。

因为数据库中数据量可能非常庞大，所以ResultSet并不是一次性把所有数据都读取出来，而是每执行一次rs.next()才去读取下一行记录，如果没有记录就返回false结束循环。因此最少要调用一次rs.next()来判断是否有结果返回。

如果rs.next()为真，所有已经取到了数据，这就可以用rs.getInt(1)取得这一行id的值，用rs.getString(2)取得这一行name的值，rs.getInt(1)表示要返回第一列integer类型的数据，rs.getString(2)说明要返回第二列varchar(100)类型的数据。ResultSet中会自动为咱们进行数据类型的转换，我们得到的直接就是java中的数据类型。如果不确定具体是哪一列，也可以通过列名取值，比如rs.getInt("id")就会返回id列的值，rs.getString("name")就会返回name列的值，实际情况中根据需要选用。

一个Statement可以返回多个ResultSet，这些ResultSet都是依附与Statement存在的。如果将Statement关闭，对应的那些ResultSet也无法从数据库获得数据，执行rs.next()或rs.getInt()的时候会抛出异常。操作结束后也要注意在Statement关闭前关闭对应的ResultSet，关闭顺序错误也会导致异常的出现。

1.6.#注入攻击与PreparedStatement

何为注入攻击？让我们先看一个小例子，我们编写一段程序，根据用户指定的name值返回记录。

```
String sql = "select * from test where name='" + name + "'";
state = conn.createStatement();
rs = state.executeQuery(sql);
// 显示查询结果
while (rs.next()) {
    System.out.println(rs.getInt(1) + " " + rs.getString(2));
}
```

这里我们直接使用用户传递过来的name变量拼接了一条sql语句进行查询。在String name = "lingirl2"的时候程序会返回正确的结果。

上面的例子里，我们乐观的认为用户输入的都是正常的字符串，没有考虑到恶意攻击的情况，如果用户输入了这样一段内容：

```
String name = "xxx' or '1'='";
```

经过拼接得到的sql就变成了这样。

```
select * from test where name='xxx' or '1'='1'
```

好啦，这会搜索出所有满足name='xxx'或者满足'1'='1'条件的记录，结果变成搜索test库中所有的记录了。

当String name = "xxx' or '1'='";的时候，查询结果如下：

所谓sql注入攻击，是因为程序没有对用户输入进行校验，造成用户可以在输入中包含恶意代码篡改程序功能。上面的例子仅仅是造成数据泄密，更严重的用户还可能窃取最高管理权限，删除数据库中所有的数据。

为了解决这个问题，我们可以使用PreparedStatement，修改后的代码如下。

```
import java.sql.*;

public class Select3 {
    public static void main(String[] args) throws Exception {
        String name = "xxx' or '1'='1";

        Connection conn = DbUtils.getConnection();
        PreparedStatement state = null;
        ResultSet rs = null;
        try {
            String sql = "select * from test where name=?";
            state = conn.prepareStatement(sql);
            state.setString(1, name);
            rs = state.executeQuery();
            // 显示查询结果
            while (rs.next()) {
                System.out.println(rs.getInt(1) + " " + rs.getString(2));
            }
        } catch(Exception ex) {
            ex.printStackTrace();
        } finally {
            if (rs != null) {
                rs.close();
            }
            if (state != null) {
                state.close();
            }
            DbUtils.close(conn);
        }
    }
}
```

通过如下几步将Statement改为PreparedStatement。

1. sql语句修改为

```
select * from test where name=?
```

注意这个问号（?）就是我们需要传递参数的地方。

2. 使用prepareStatement获得PreparedStatement变量。

```
state = conn.prepareStatement(sql);
```

与createStatement不同，创建的PreparedStatement的同时要传入sql语句，让PreparedStatement对sql语句进行预处理，以备后用。

3. 传入name参数。

```
state.setString(1, name);
```

这里的参数“1”代表使用name替代sql中的第一个问号“?”，name中有什么特殊字符，PreparedStatement会帮助咱们自动转换。

4. 最后执行查询，rs = state.executeQuery();因为创建PreparedStatement的时候就已经对sql进行了处理，这里直接执行查询就能得到结果。

看一下使用PreparedStatement查询String name = "xxx' or '1'='";的情况。

实际上，自己手工拼接sql非常容易出错，即便不担心注入攻击也应该尽量使用PreparedStatement，至少可以减小写错sql的机会。

最后放上一个使用PreparedStatement进行插入的例子，可以自己写一个不使用PreparedStatement的例子对比一下。

```
import java.sql.*;

public class Insert {
    public static void main(String[] args) throws Exception {
        int id = 1;
        String name = "lingirl";

        Connection conn = DbUtils.getConnection();
        PreparedStatement state = null;
        try {
            String sql = "insert into test(id,name) values(?,?)";
            state = conn.prepareStatement(sql);
            state.setInt(1, id);
            state.setString(2, name);

            state.executeUpdate();
```

```
        } catch(Exception ex) {
            ex.printStackTrace();
        } finally {
            if (state != null) {
                state.close();
            }
            DbUtils.close(conn);
        }
    }
}
```

这节的例子都放在01-06目录下，提供对应的脚本，可以直接运行。

1.7.#事务隔离与事务锁

三种隔离级别

- 脏读，读取到未提交的数据。

  路人甲向路人乙汇钱，钱汇入帐号后，选择取消进行事务回滚，路人乙在回滚前查询可以看到这笔钱已经出现在自己的帐号上了，但事务回滚后这笔钱没没有存到路人乙的帐户。

- 不可重复读，两次读取的同一数据被其他事务修改，造成两次读取的数据不一致。

  路人乙统计自己全年和季度的结账情况，第一次按照全年查询，第二次按照嫉妒查询，如果这期间路人甲向路人乙的帐号汇了一笔钱，路人乙查询两次的结果就不相符了。

- 虚读，某一事务修改数据还未提交，另一个事务进行对上个事务有影响的插入或删除操作，造成上一个事务提交出错。

  路人乙正在更新路人甲账单的信息，在提交前路人甲突然将此条信息删除，路人乙提交修改的时候就会出错。

jdbc提供四种隔离级：

- TRANSACTION_READ_UNCOMMITTED。

  在提交前一个事务可以看到另一个事务的变化，脏读，不可重复读和虚度都可能发生。

- TRANSACTION_READ_COMMITTED。

  读取未提交的数据是不允许的，不可重复的读和虚读可能发生。

- TRANSACTION_REPEATABLE_READ。

  可以保证一个事务内读取的数据不会被改变，虚读仍然会出现。

- TRANSACTION_SERIALIZABLE。

  最高的事务级别，防止脏读、不可重复的读和虚读。

表 1.1. 隔离级别关系

隔离级别	脏读	不可重复读	虚读
TRANSACTION_READ_UNCOMMITTED			

隔离级别	脏读	不可重复读	虚读
TRANSACTION_READ_COMMITTED	not		
TRANSACTION_REPEATABLE_READ	not	not	
TRANSACTION_SERIALIZABLE	not	not	not

为了在程序中使用事务，首先要将Connection的autoCommit（自动提交）功能关闭。

```
import java.sql.*;

public class Main {
    public static void main(String[] args) throws Exception {
        int id = 1;
        String name = "lingirl";

        Connection conn = DbUtils.getConnection();
        // 关闭自动提交
        conn.setAutoCommit(false);
        // System.out.println(conn.getTransactionIsolation());
        if (conn.getTransactionIsolation() != Connection.TRANSACTION_NONE) {
            conn.setTransactionIsolation(Connection.TRANSACTION_READ_COMMITTED);
        }
        PreparedStatement state = null;
        try {
            String sql = "insert into test(id,name) values(?,?)";
            state = conn.prepareStatement(sql);
            state.setInt(1, id);
            state.setString(2, name);

            state.executeUpdate();
            // 事务提交
            conn.commit();
        } catch(Exception ex) {
            ex.printStackTrace();
            // 事务回滚
            conn.rollback();
        } finally {
            if (state != null) {
                state.close();
            }
            DbUtils.close(conn);
        }
    }
}
```

获得Connection之后调用setAutoCommit(false)关闭自动提交，然后判断数据库是否支持事务，如果getTransationIsolation()不等于TRANSACTION_NONE就说明我们使用的数据库支持事务，下面才可以使用setTransactionIsolation()设置隔离级别。

注意

这个TRANSACTION_NONE是用来判断数据库是否支持事务的，不能使用conn.setTransaction(Connection.TRANSACTION_NONE);把一个数据库设置成不支持事务，这里能用的参数只有上边说的四种。

我们这里使用的access只支持TRANSACTION_READ_COMMITTED，在使用其他数据库时需要先确认是否支持某一隔离级别再进行设置，否则会抛出异常。

例子在01-07目录下，运行main.bat查看结果。

第#2#章#hsqldb

hsqldb是一个完全用java编写的数据库，既可以独立运行也可以嵌入java开发的项目中。整个jar包只有600多K，是测试和演示时的极品选择。

2.1.#第一个程序

首先我们要下载hsqldb，他们的官方网站在http://www.hsqldb.org，下载了zip发布包找到里边的hsqldb.jar，这就是我们需要的数据库程序。

写一个测试连接的程序。

```
import java.sql.DriverManager;
import java.sql.Connection;

public class DbUtils {
    public static void main(String[] args) throws Exception {
        // 加载驱动
        Class.forName("org.hsqldb.jdbcDriver");
        // 连接数据库的地址
        String url = "jdbc:hsqldb:mem:.";
        String username = "sa";
        String password = "";

        Connection conn = null;
        try {
            // 创建与数据库的连接
            conn = DriverManager.getConnection(url, username, password);
            System.out.println("成功连接到数据库：" + conn);
        } catch(Exception ex) {
            System.out.println("连接失败：" + ex);
        } finally {
            // 关闭连接，释放资源
            if (conn != null) {
                conn.close();
            }
        }
    }
}
```

加载驱动程序类名是org.hsqldb.jdbcDriver，它就在hsqldb.jar中。

连接数据库使用的用户名为sa，密码为空，这是默认的数据库管理员。

数据库连接的url地址是jdbc:hsqldb:mem:.。所有jdbc连接url都是以jdbc开头的，第一个冒号后便的部分是使用的驱动别名，这里的hsqldb就代表我们将使用org.hsqldb.jdbcDriver，第二个冒号后是每个数据库自己特定的访问地址，这里的mem:.表示使用内存存储模式，连接的数据库名称是“.”。

2.2.#单独运行与嵌入调用

之前提到hsqldb支持单独运行和嵌入调用两种，在此我们讨论一下这两种调用方式的利弊。

1. 嵌入调用

hsqldb是完全使用java编写的，我们可以在自己写的程序里调用hsqldb，就像启动一个普通的线程一样，让hsqldb与我们的程序运行在同一个jvm中。

像我们上面的例子中使用的就是嵌入调用的形式。DbUtils执行的过程中会启动hsqldb服务器并与之建立连接，在关闭连接释放资源后，jvm关闭的同时也将hsqldb服务器关闭。DbUtils和hsqldb运行在同一个jvm上，共享这个jvm分配的内存等资源。

2. 单独运行

单独启动一个jvm运行hsqldb，客户程序与服务器程序运行在不同的jvm中，双方通过socket交换数据，客户程序的启动和关闭不会直接影响hsqldb。

我们可以使用它自带的org.hsqldb.Server，启动脚本如下。

```
set classpath=hsqldb-1.8.0.7.jar
java org.hsqldb.Server
```

org.hsqldb.Server会自动去读取server.properties中的配置。

```
server.port=9100
server.database.0=mem:test
server.dbname.0=test
```

这里启动一个监听9100端口的hsqldb服务器，服务器中包含一个数据库，使用mem:test的存储方式，对外的数据库名称是test。

为了连接这个数据库，需要修改连接数据库使用的url。

```
String url = "jdbc:hsqldb:hsql://localhost:9100/test"
```

hsql表示我们将使用hsqldb自身的socket方式进行连接。连接的数据库在localhost的9100端口，test是我们需要进行连接的数据库名称。

如果希望关闭hsqldb数据库，可以直接关闭java弹出的console窗口。

2.3.#四种存储方式

hsqldb拥有四种存储数据的方式，之前见过的有mem和hsql，下面来介绍一下它们的用法和区别。

1. mem，内存（memory）存储方式。

hsqldb启动的时候会在内存中建立对应的表结构，运行期间对数据做出的所有修改都保存到内存中，最后关闭hsqldb的同时，内存中的所有数据也会全部丢失。

示例代码02-03/Mem.java中，首先创建表结构，并向数据库中写入两条语句：

```
String[] sqls = {
    "create table test(id bigint,name varchar(200))",
    "insert into test(id,name) values(1,'lingirl')",
    "insert into test(id,name) values(2,'叮咚')"
};
```

然后执行查询，可以看到两条数据已经成功保存进数据库了。

```
1 lingirl
2 叮咚
```

mem方式的缺点是每次启动hsqldb都要先创建表结构，插入初始数据，关闭hsqldb的同时数据也会全部丢失。

mem方式的优点是所有操作都在内存中进行，不会生成额外的数据文件，可以保证每次创建的数据库都是绝对干净的，这点对测试来讲很有用。

2. file，文件存储方式。

   hsqldb使用文件保存数据库配置，表结构和更新的数据。

   hsqldb运行过程中会生成四个文件，它们的文件名都是与连接url地址对应的，示例02-03/File.java中配置的url为"jdbc:hsqldb:file:db/file"，对应的将是db目录下所有以file开头的文件。

   a. file.properties存放数据库配置，包括数据库版本，缓存，表结构设置等等。

   b. file.lck用来标记当前数据库是否已经被某一个hsqldb访问了，同一时间只有一个hsqldb能操作数据库文件，这样才能保证不会出现数据冲突。

      这个文件在hsqldb启动时自动生成，正常关闭时会自动删除，但非法关闭会无法删除这个文件，如果因为这个文件出现数据库服务启动的情况，就需要手工删除。

   c. file.script和file.log

      hsqldb使用这两个文件保存数据表结构和数据，因为用到了缓存，更新的数据不会直接写入文件，而是在内存中积累一定量后，才会批量写入file.log这个日志文件。

      file.script用来保存最终数据，hsqldb正常关闭的时候会把内存和日志文件（file.log）中的数据写入file.script，并删除日志（file.log）文件。

      如果出现非正常关闭的情况，内存中缓存的数据会丢失，file.log也无法删除，不过不用担心，下次启动的时候hsqldb会先检测是否有未删除的file.log，将其中的数据写入file.script，再读取file.script中的数据，进行初始化操作。

      File.java中的sql语句有了变化，因为使用文件保存了表结构，每次操作之前还是先删除它们比较保险。

```
String[] sqls = {
    "drop table test if exists",
    "create table test(id bigint,name varchar(200))",
```

```
    "insert into test(id,name) values(1,'lingirl')",
    "insert into test(id,name) values(2,'叮咚')"
};
```

多了这条清空语句，我们可以得到与mem相同的查询结果，如果不删除表，把插入语句修改成不冲突的形式，我们就可以看到每次查询结果不断增多。因为即使hsqldb关闭了，数据也会保存在文件里。

拜缓存所赐，像我们这样放任hsqldb随jvm关闭，一定会丢失数据，为了这一点，我们在url上加入了特殊标记String url = "jdbc:hsqldb:file:db/file;shutdown=true"，保证每次conn.close()都会关闭数据，将数据写入file.script，相对的也使缓存完全丧失了效果。

当然，也可以向数据库发送shutdown语句，正常关闭数据库。

3. res，资源存储方式。

它是mem与file两者的结合，我们把res.properties和res.script放到classpath下的db/目录下，对应url = "jdbc:hsqldb:res:/db/res"。数据库启动的时候会去这两个文件里读取数据进行初始化，但不会生成res.lck和res.log，以后进行的所有操作就都在内存里了，关闭数据库也不会写入res.script。

res就像是使用file提供初始数据的mem内存访问方式，它在系统演示的时候十分方便，设置一次初始数据就可以演示多次，而且不会影响初始数据的结果。用于测试就不太适合了，res.script明显没有sql语句便于维护，还是老老实实用sql导入更合适。

res唯一的限制是不能进行建表和删表的操作，你可以插入，修改，删除数据，但是不能修改表的结构。

4. hsql，socket访问方式。

使用socket链接，从一个单独运行的hsqldb服务器中获得数据，这个服务器中的hsqldb又可能是这四种存储方式。只要你想，就完全可以做成一条链状结构。

2.4.#正常关闭jdbc

第一种方式，url = "jdbc:hsqldb:file:db/file;shutdown=true"，这样每次conn.close()都会执行shutdown命令关闭数据，实际上扼杀了缓存的效果。

另一个方式，在jvm关闭前，向数据库发送shutdown命令，让它正常关闭。

附录#A.#修改日志

修订历史

修订 0.0.1 2008-03-24

1. 初稿完成。序言