# 一、JDBC其他API

## 可滚动结果集与分页技术

**static** **void** scroll() **throws** SQLException {

        Connection conn = **null**;

        Statement st = **null**;

        ResultSet rs = **null**;

**try** {

            conn = JdbcUtils.getConnection();

//在创建一个Statement的时候指定可滚动的结果集的类型

            st = conn.createStatement(ResultSet.TYPE\_SCROLL\_SENSITIVE,

                    ResultSet.CONCUR\_READ\_ONLY);

//在mysql中通过limit关键字实现分页，每种数据库产品的关键字不同例如：Oracle使用rownum，sqlServer用top

//现在有几十中关系型数据库如果数据库不支持这种关键字进行分页的时候可以用滚动的结果集来实现分页，但是性能比较低

    rs = st.executeQuery("select id, name, money, birthday  from user limit 150, 10");

**while** (rs.next()) {

                System.out.println(rs.getObject("id") + "\t"

                        + rs.getObject("name") + "\t"

                        + rs.getObject("birthday") + "\t"

                        + rs.getObject("money"));

            }

            //绝对定位，可以直接定位到rs所有返回结果中指定的一条记录上

            rs.absolute(150);

**int** i = 0;

            //可以通过i来控制循环次数，实现分页效果但是要数据库产品或者驱动支持此功能！

**while** (rs.next() && i < 10) {

                i++;

                System.out.println(rs.getObject("id") + "\t"

                        + rs.getObject("name") + "\t"

                        + rs.getObject("birthday") + "\t"

                        + rs.getObject("money"));

            }

            //如果上面设定的结果集是可以滚动的，那么你还可以用里面的很多的方法，在此不列举了

        } **finally** {

            JdbcUtils.free(rs, st, conn);

        }

}

## 可更新和对更新敏感的结果集

static void read() throws SQLException, InterruptedException {

Connection conn = null;

Statement st = null;

ResultSet rs = null;

try {

// 2.建立连接

conn = JdbcUtils.getConnection();

// conn = JdbcUtilsSing.getInstance().getConnection();

// 3.创建语句

//设置滚动结果集的类型为：ResultSet.TYPE\_SCROLL\_SENSITIVE，就是能感知到数据库的变化

st = conn.createStatement(ResultSet.TYPE\_SCROLL\_SENSITIVE,

ResultSet.CONCUR\_UPDATABLE);

// 4.执行语句

rs = st.executeQuery("select id, name, money, birthday from user where id < 5");

// 5.处理结果

//下面让rs每循环一次rs睡眠10秒钟，然后再这个过程中我们用mysql客户端修改数据库中的数据

//我们看看他读出来的是修改前的数据，还是修改后的。我们在上面设置的可滚动的结果集的类型

//是ResultSet.TYPE\_SCROLL\_SENSITIVE，也就是能感知数据库的变化，那如果在rs没有读出数据库里的

//那条数据之前我们在mysql的客户端将原先的数据修改掉，这里读出来的数据应该是修改后的数据，但是在

//测试的时候读出的数据却依然是修改之前的，这应该和数据库的驱动有关系

//但是如果是能感知数据库的变化，那么数据库的性能也是降低了，你执行executeQuery()方法后，他已经将数据查询完成

//打包后给你发送过来了，如果察觉到数据库的变化那么他要在输出之前在查询一遍数据库，这种需求用的比较少，作为

//了解即可

while (rs.next()) {

int id = rs.getInt("id");

System.out.println("show " + id + "...");

Thread.sleep(10000);

System.out.println(id + "\t" + rs.getObject("name") + "\t"

+ rs.getObject("birthday") + "\t"

+ rs.getObject("money"));

String name = rs.getString("name");

//可更新的结果集，我们并不建议这样做因为上面的sql语句是查询操作

//但是下面还隐藏着更新操作，对于程序的可读性不好，这种需求也比较少

//做了解即可

if("lisi".equals(name)) {

rs.updateFloat("money",300f);

//修改完成后要修改一行

rs.updateRow();

}

}

} finally {

JdbcUtils.free(rs, st, conn);

}

}

# 二、元数据

## 元数据

元数据其实就是数据库，表，列的定义信息，版本、隔离级别。

        java.sql.Connection conn = JdbcUtils.getConnection();

        //得到数据库的元信息

        DatabaseMetaData dbmd = conn.getMetaData();

        //取出当前使用数据库的名称

        System.out.println("db name: " + dbmd.getDatabaseProductName());

        //看看当前数据库支不支持事务

        System.out.println("tx: " + dbmd.supportsTransactions());

即使我们写了一个简单工具类，我们的代码还是非常冗余。**对于增删改而言，只有SQL和参数是不同的**，我们为何不把这些相同的代码抽取成一个方法？对于**查询而言**，**不同的实体查询出来的结果集是不一样的**。我们要**使用元数据获取结果集的信息，才能对结果集进行操作**。

* ParameterMetaData   --参数的元数据
* ResultSetMetaData      --结果集的元数据
* DataBaseMetaData      --数据库的元数据

### 参数的元数据信息

/\*\*链接数据库的url为：url = "jdbc:mysql://localhost:3306/jdbc?generateSimpleParameterMetadata=true

\* 要加入 generateSimpleParameterMetadata=true时才行，否则会抛

\* “java.sql.SQLException: Parameter metadata not available for the given statement”，

\* 因为mysql驱动默认generateSimpleParameterMetadata=false只有设置为true metadata类型会

\* 将每一个参数反射为Varchar类型

\*

\*/

public static void main(String[] args) throws SQLException {

String sql = "select \* from user where name = ? and sex = ?";

Object[] params = {"zhangsan", "男"};

query(sql, params);

}

public static void query(String sql, Object[] param) {

Connection conn = null;

PreparedStatement pstmt = null;

ResultSet rs = null;

try {

conn = JdbcUtils.getConnection();

pstmt = conn.prepareStatement(sql);

ParameterMetaData pmd = pstmt.getParameterMetaData();

int paramsCount = pmd.getParameterCount();//灵活性好

//得到参数信息的元数据

for(int i=0; i<=paramsCount/ param.length; i++) {

//用for循环的方法给sql语句中的占位符赋值，但是要约定：**sql语句中占位符的所表示的类型**

//和个数和参数数组中是一致的

pstmt.setObject(i+1, param[i ]);

}

rs = pstmt.executeQuery();

while(rs.next()) {

System.out.println(rs.getString(2));

}

} catch (SQLException e) {

throw new RuntimeException(e.getMessage());

}

}

### 将结果集元数据封装为Map

现在我们有一种需求将ResultSet结果集中的数据封装成Map map的key是数据库中字段的值value就是在字段中的值

public static void main(String[] args) {

Map<String, Object> datas = read("select id, name from user");

}

private static Map<String, Object> read(String sql) {

Connection conn = null;

PreparedStatement pstmt = null;

ResultSet rs = null;

try {

conn = JdbcUtils.getConnection();

pstmt = conn.prepareStatement(sql);

rs = pstmt.executeQuery();

//得到ResultSet的元数据

ResultSetMetaData rsmd = rs.getMetaData();

//得到一条ResultSet数据的列数

int colCount = rs.getColumnCount();

String[] colName = new String[colCount];

for(int i=1; i<=colCount; i++) {

//得到参数的类名，java.lang.String

/\*System.out.print(rmsd.getColumnClassName(i) + "\t");

//得到列名

System.out.print(rmsd.getColumnName(i) + "\t");

//得到别名：select name as n from user -->得到的是n

System.out.println(rmsd.getColumnLabel(i) + "\t");\*/

colName[i-1] = rmsd.getColumnLabel(i);

}

//新建一个Map对象

Map<String, Object> data = new HashMap<String, Object>();

while(rs.next()) {

for(int i=1; i<=colName.length; i++) {

data.put(colName[i-1], rs.getObject(i));

System.out.println(colName[i-1] + "======" +rs.getObject(i) );

}

}

//返回Map对象

return data;

} catch (SQLException e) {

throw new RuntimeException(e.getMessage());

}finally {

JdbcUtils.free(rs, pstmt, conn);

}

}

### 将查询结果封装为对象

约定数据库的**字段名要和属性的字段名一致，** String methodName = "set" + 数据库的字段名。

用这样的方法来构成 Javabean中属性的set方法来实现赋值操作，那么在传递sql语句的时候就有一定的局限性了，例如：你不能写 select id , name from user 。因为这样得到的set方法setid, setname这样就没法完成赋值操作，要这样写：select id as Id, name as Name fromt user,然后用ResultSetMetadata。getColumnLabel()得到他的别名，然后遍历Javabean中所有的方法找到和我们这个名字一样的，然后执行就可以了，但是通过sql语句来操纵属性名称然后构成方法名，这样的方式不大好。

### 改造JDBC工具类

增删改

//我们发现，增删改只有SQL语句和传入的参数是不知道的而已，所以让调用该方法的人传递进来  
//由于传递进来的参数是各种类型的，而且数目是不确定的，所以使用Object[]  
public static void update(String sql, Object[] objects) {  
    Connection connection = null;  
    PreparedStatement preparedStatement = null;  
    ResultSet resultSet = null;  
    try {  
        connection = getConnection();  
        preparedStatement = connection.prepareStatement(sql);  
        //根据传递进来的参数，设置SQL占位符的值  
        for (int i = 0; i < objects.length; i++) {  
            preparedStatement.setObject(i + 1, objects[i]);  
        }  
        //执行SQL语句  
        preparedStatement.executeUpdate();  
    } catch (Exception e) {  
        e.printStackTrace();  
}

查询

/\*  
    1:对于查询语句来说，我们不知道对结果集进行什么操作【常用的就是把数据封装成一个Bean对象，封装成一个List集合】  
    2:我们可以定义一个接口，让调用者把接口的实现类传递进来  
    3:这样接口调用的方法就是调用者传递进来实现类的方法。【策略模式】  
\*/  
//这个方法的返回值是任意类型的，所以定义为Object。  
public static Object query(String sql, Object[] objects, ResultSetHandler rsh) {  
    Connection connection = null;  
    PreparedStatement preparedStatement = null;  
    ResultSet resultSet = null;  
    try {  
        connection = getConnection();  
        preparedStatement = connection.prepareStatement(sql);  
        //根据传递进来的参数，设置SQL占位符的值  
        if (objects != null) {  
            for (int i = 0; i < objects.length; i++) {  
                preparedStatement.setObject(i + 1, objects[i]);  
            }  
        }  
        resultSet = preparedStatement.executeQuery();  
        //调用调用者传递进来实现类的方法，对结果集进行操作  
        return rsh.hanlder(resultSet);  
}

**接口：**

/\*  
\* 定义对结果集操作的接口，调用者想要对结果集进行什么操作，只要实现这个接口即可  
\* \*/  
public interface ResultSetHandler {  
     Object hanlder(ResultSet resultSet);  
}

**实现类：**

//接口实现类，对结果集封装成一个Bean对象  
public class BeanHandler implements ResultSetHandler {  
    //要封装成一个Bean对象，首先要知道Bean是什么，这个也是调用者传递进来的。  
    private Class clazz;  
    public BeanHandler(Class clazz) {  
        this.clazz = clazz;  
    }  
    @Override  
    public Object hanlder(ResultSet resultSet) {  
        try {  
            //创建传进对象的实例化  
            Object bean = clazz.newInstance();  
            if (resultSet.next()) {  
                //拿到结果集元数据  
                ResultSetMetaData resultSetMetaData = resultSet.getMetaData();  
                for (int i = 0; i < resultSetMetaData.getColumnCount(); i++) {  
                    //获取到每列的列名  
                    String columnName = resultSetMetaData.getColumnName(i+1);  
                    //获取到每列的数据  
                    String columnData = resultSet.getString(i+1);  
                    //设置Bean属性  
                    Field field = clazz.getDeclaredField(columnName);  
                    field.setAccessible(true);  
                    field.set(bean,columnData);  
                }  
                //返回Bean对象  
                return bean;  
            }

# 三、[数据库连接池](https://www.cnblogs.com/xdp-gacl/p/4002804.html)

## 数据库连接池的基本概念

用户每次请求都需要向数据库获得链接，而数据库创建连接通常需要消耗相对较大的资源，创建时间也较长。假设网站一天10万访问量，数据库服务器就需要创10万次连接，极大的浪费数据库的资源，并且极易造成数据库服务器内存溢出、拓机。

**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**。

数据库连接池在初始化时将**创建一定数量的数据库连接放到连接池中,** 这些数据库连接的数量是由最小数据库连接数来设定的.无论这些数据库连接是否被使用,连接池都将一直保证至少拥有这么多的连接数量.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.**

 数据库连接池的最小连接数和最大连接数的设置要考虑到以下几个因素:

1. 最小连接数:是连接池一直保持的数据库连接,所以如果应用程序对数据库连接的使用量不大,将会有大量的数据库连接资源被浪费.
2. 最大连接数:是连接池能申请的最大连接数,如果数据库连接请求超过次数,后面的数据库连接请求将被加入到等待队列中,这会影响以后的数据库操作
3. 如果最小连接数与最大连接数相差很大:那么最先连接请求将会获利,之后超过最小连接数量的连接请求等价于建立一个新的数据库连接.不过,这些大于最小连接数的数据库连接在使用完不会马上被释放,他将被放到连接池中等待重复使用或是空间超时后被释放.

## 编写数据库连接池

1. 编写连接池需**实现java.sql.DataSource接口**
2. **创建批量的Connection用LinkedList保存**【既然是个池，当然用集合保存、、LinkedList底层是链表，对增删性能较好】
3. **实现getConnetion()**，让getConnection()每次调用，都是**在LinkedList中取一个Connection返回给用户**
4. **调用Connection.close()方法，Connction返回给LinkedList**

**解决思路：**

1. 写一个Connection子类，覆盖close()方法
2. 写一个Connection包装类，增强close()方法
3. 用动态代理，返回一个代理对象出去，拦截close()方法的调用，对close()增强

分析第一个思路：

* **Connection是通过数据库驱动加载的，保存了数据的信息**。写一个子类Connection，new出对象，**子类的Connction无法直接继承父类的数据信息，也就是说子类的Connection是无法连接数据库的**，更别谈覆盖close()方法了。

分析第二个思路：

* 写一个Connection包装类。
  1. 写一个类，实现与被增强对象的相同接口【Connection接口】
  2. 定义一个变量，指向被增强的对象
  3. 定义构造方法，接收被增强对象
  4. 覆盖想增强的方法
  5. 对于不想增强的方法，直接调用被增强对象的方法
* 这个思路本身是没什么毛病的，就是**实现接口时，方法太多了！**，所以我们也不使用此方法

### 编写一个基本的连接池来实现连接的复用

public class MyDataSource {

private static String url = "jdbc:mysql://localhost:3306/jdbc?generateSimpleParameterMetadata=true";

private static String user = "root";

private static String password = "admin";

LinkedList<Connection> connectionsPool = new LinkedList<Connection>();

//向我们的LinkedList集合中加入10个链接作为我们的连接池

public MyDataSource() {

for(int i=0; i<10; i++) {

try {

//将Connection放到链表的最后面

connectionsPool.addLast(this.createConnection());

} catch (SQLException e) {

throw new ExceptionInInitializerError(e.getMessage());

}

}

}

//创建10个链接

public Connection createConnection() throws SQLException{

return DriverManager.getConnection(url, user, password);

}

//得到一个链接(先进先出算法)

public Connection getConnection() {

return connectionsPool.removeFirst();

}

//关闭一个链接，这个关闭不是真正意义上的关闭，而是又把他放回到连接池中，实现了Connection的复用

public void free(Connection conn) {

connectionsPool.addLast(conn);

}

}

### 对基本连接池进行一些工程细节上的优化

在上面实现的连接池中我们只是默认创建了5个连接，但是如果这个时候有5个线程同时都来拿连接，那连接池里就没有连接 在有线程过来拿的时候就会报错了，现在我们进行一些优化(重复的代码就不写了，只写改动的)

//规定默认创建的连接数

private static int initCount = 5;

//规定最大可以创建的连接数

private static int maxCount = 10;

//统计当前共创建了多少个连接

private int currentCount = 0;

LinkedList<Connection> connectionsPool = new LinkedList<Connection>();

public MyDataSource() {

for(int i=0; i<initCount; i++) {

try {

connectionsPool.addLast(this.createConnection());

//每创建一个链接 currentCount ++

this.currentCount ++;

} catch (SQLException e) {

throw new ExceptionInInitializerError(e.getMessage());

}

}

}

public Connection createConnection() throws SQLException{

return DriverManager.getConnection(url, user, password);

}

public Connection getConnection() throws SQLException {

//因为Connection不是线程安全的，所以我必须保证每个线程拿到的链接不是一个，所以要进行同步：当两个线程同时来拿的时候

//另外一个线程必须等待

synchronized (connectionsPool) {

if(connectionsPool.size() > 0)

return connectionsPool.removeFirst();

//如果当前创建的链接数没有到最大值，那就继续创建链接

if(this.currentCount < maxCount) {

this.currentCount ++;

return this.createConnection();

}

//抛出异常

throw new SQLException("当前已经没有可用连接了");

}

}

### 代理模式来保持用户关闭连接的习惯

在上面的示例中我们在关闭链接的时候，调用的是free方法来把这个连接又放回到了池中，但是按照开发人员的使用习惯应该是调用colse()方法来关闭一个链接，但是如果调用close方法关闭，那这个连接就真的关闭了，也就是说我们这个方法设计的不符合开发人员的使用习惯下面我用代理的 方法来解决这个问题：   
 **定义一个类实现Connection接口**,Connectio接口中有很多的方法，这些方法我们都无法自己完成，**我们交给通过构造方法传递进来的真正的Connection 的对象来完成，我们只是修改它的close方法，**在用户得到链接的时候我们返回给用户这个类的对象，那么当用户调用close方法关闭链接的时候，我们就可以在这个close方法中将用户要关闭的那个链接再次的放到连接池中，这样链接就不会真正的关闭了。

proxyConn = (Connection) Proxy.newProxyInstance(this.getClass()

.getClassLoader(), conn.getClass().getInterfaces(),

new InvocationHandler() {

//此处为内部类，当close方法被调用时将conn还回池中,其它方法直接执行

public Object invoke(Object proxy, Method method,

Object[] args) throws Throwable {

if (method.getName().equals("close")) {

pool.addLast(conn);

return null;

}

return method.invoke(conn, args);

}

});

### 数据库连接池编写范例：

public class JdbcPool implements DataSource{

private static LinkedList<Connection> listConnections = new LinkedList<Connection>();

static{

//在静态代码块中加载db.properties数据库配置文件

InputStream in = JdbcPool.class.getClassLoader().getResourceAsStream("db.properties");

Properties prop = new Properties();

try {

prop.load(in);

String driver = prop.getProperty("driver");

String url = prop.getProperty("url");

String username = prop.getProperty("username");

String password = prop.getProperty("password");

//数据库连接池的初始化连接数大小

int jdbcPoolInitSize =Integer.parseInt(prop.getProperty("jdbcPoolInitSize"));

//加载数据库驱动

Class.forName(driver);

for (int i = 0; i < jdbcPoolInitSize; i++) {

Connection conn = DriverManager.getConnection(url, username, password);

System.out.println("获取到了链接" + conn);

//将获取到的数据库连接加入到listConnections集合中，listConnections集合此时就是一个**存放了数据库连接的连接池**

listConnections.add(conn);

}

} catch (Exception e) {

throw new ExceptionInInitializerError(e);

}

}

/\* 获取数据库连接

\* @see javax.sql.DataSource#getConnection()

\*/

@Override

public Connection getConnection() throws SQLException {

//如果数据库连接池中的连接对象的个数大于0

if (listConnections.size()>0) {

//从listConnections集合中获取一个数据库连接

final Connection conn = listConnections.removeFirst();

//这说明引用变量被final修饰之后，虽然不能再指向其他对象，但是它指向的对象的内容是可变的。

System.out.println("listConnections数据库连接池大小是" + listConnections.size());

//返回Connection对象的代理对象

return (Connection) Proxy.newProxyInstance(JdbcPool.class.getClassLoader(), conn.getClass().getInterfaces(), new InvocationHandler(){

@Override

public Object invoke(Object proxy, Method method, Object[] args)

throws Throwable {

if(!method.getName().equals("close")){

return method.invoke(conn, args);

}else{

//如果调用的是Connection对象的close方法，就把conn还给数据库连接池

listConnections.add(conn);

System.out.println(conn + "被还给listConnections数据库连接池了！！");

System.out.println("listConnections数据库连接池大小为" + listConnections.size());

return null;

}

}

});

}else {

throw new RuntimeException("对不起，数据库忙");

}

}

}

}

## 开源数据库连接池

### DBCP数据源

　　DBCP 是 Apache 软件基金组织下的开源连接池实现，要使用DBCP数据源，需要应用程序应在系统中增加如下两个 jar 文件：

* Commons-dbcp.jar：连接池的实现
* Commons-pool.jar：连接池实现的依赖库

　　Tomcat 的连接池正是采用该连接池来实现的。该数据库连接池既可以与应用服务器整合使用，也可由应用程序独立使用。

**dbcpconfig.properties**

#连接设置

driverClassName=com.mysql.jdbc.Driver

url=jdbc:mysql://localhost:3306/jdbcstudy

username=root

password=111111

#<!-- 初始化连接 -->

initialSize=10

#最大连接数量

maxActive=50

#<!-- 最大空闲连接 -->

maxIdle=20

#<!-- 最小空闲连接 -->

minIdle=5

#<!-- 超时等待时间以毫秒为单位 6000毫秒/1000等于60秒 -->

maxWait=60000

#JDBC驱动建立连接时附带的连接属性属性的格式必须为这样：[属性名=property;]

#注意："user" 与 "password" 两个属性会被明确地传递，因此这里不需要包含他们。

connectionProperties=useUnicode=true;characterEncoding=UTF8

#指定由连接池所创建的连接的自动提交（auto-commit）状态。

defaultAutoCommit=true

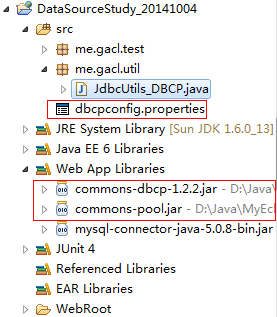
#driver default 指定由连接池所创建的连接的只读（read-only）状态。

#如果没有设置该值，则“setReadOnly”方法将不被调用。（某些驱动并不支持只读模式，如：Informix）

defaultReadOnly=

#driver default 指定由连接池所创建的连接的事务级别（TransactionIsolation）。

#可用值为下列之一：（详情可见javadoc。）NONE,READ\_UNCOMMITTED, READ\_COMMITTED, REPEATABLE\_READ, SERIALIZABLE

defaultTransactionIsolation=READ\_UNCOMMITTED

**在获取数据库连接的工具类(如jdbcUtils)的静态代码块中创建池**

package me.gacl.util;

import java.io.InputStream;

import java.sql.Connection;

import java.sql.ResultSet;

import java.sql.SQLException;

import java.sql.Statement;

import java.util.Properties;

import javax.sql.DataSource;

import org.apache.commons.dbcp.BasicDataSourceFactory;

public class JdbcUtils\_DBCP {

/\*\*

\* 在java中，编写数据库连接池需实现java.sql.DataSource接口，每一种数据库连接池都是DataSource接口的实现

\* DBCP连接池就是java.sql.DataSource接口的一个具体实现

\*/

private static DataSource ds = null;

//在静态代码块中创建数据库连接池

static{

try{

//加载dbcpconfig.properties配置文件

InputStream in = JdbcUtils\_DBCP.class.getClassLoader().getResourceAsStream("dbcpconfig.properties");

Properties prop = new Properties();

prop.load(in);

//创建数据源

ds = BasicDataSourceFactory.createDataSource(prop);

}catch (Exception e) {

throw new ExceptionInInitializerError(e);

}

}

public static Connection getConnection() throws SQLException{

//从数据源中获取数据库连接

return ds.getConnection();

}

/\*\*

\* @Method: release

\* @Description: 释放资源，

\* 释放的资源包括Connection数据库连接对象，负责执行SQL命令的Statement对象，存储查询结果的ResultSet对象

**//这里释放资源不是把数据库的物理连接释放了，是把连接归还给连接池【连接池的Connection内部自己做好了】**

\*/

public static void release(Connection conn,Statement st,ResultSet rs){

if(rs!=null){

try{

//关闭存储查询结果的ResultSet对象

rs.close();

}catch (Exception e) {

e.printStackTrace();

}

rs = null;

}

if(st!=null){

try{

//关闭负责执行SQL命令的Statement对象

st.close();

}catch (Exception e) {

e.printStackTrace();

}

}

if(conn!=null){

try{

//将Connection连接对象还给数据库连接池

conn.close();

}catch (Exception e) {

e.printStackTrace();

}

}

}

}

**测试DBCP数据源**

package me.gacl.test;

import java.sql.Connection;

import java.sql.PreparedStatement;

import java.sql.ResultSet;

import org.junit.Test;

import me.gacl.util.JdbcUtils\_DBCP;

public class DataSourceTest {

@Test

public void dbcpDataSourceTest() {

Connection conn = null;

PreparedStatement st = null;

ResultSet rs = null;

try{

//获取数据库连接

conn = JdbcUtils\_DBCP.getConnection();

String sql = "insert into test1(name) values(?)";

st = conn.prepareStatement(sql);

st.setString(1, "gacl");

st.executeUpdate();

//获取数据库自动生成的主键

rs = st.getGeneratedKeys();

if(rs.next()){

System.out.println(rs.getInt(1));

}

}catch (Exception e) {

e.printStackTrace();

}finally{

//释放资源

JdbcUtils\_DBCP.release(conn, st, rs);

}

}

}

### C3P0数据源

　C3P0是一个开源的JDBC连接池，它实现了数据源和JNDI绑定，支持JDBC3规范和JDBC2的标准扩展。目前使用它的开源项目有***Hibernate，Spring***等。C3P0数据源在项目开发中使用得比较多。

　　c3p0与dbcp区别

1. dbcp没有自动回收空闲连接的功能
2. c3p0有自动回收空闲连接功能
3. C3P0数据源的性能更胜一筹，并且它可以**使用XML配置文件配置信息！**

**在应用程序中加入C3P0连接池**

**1.导入相关jar包**  
    　　　c3p0-0.9.2-pre1.jar、mchange-commons-0.2.jar，如果操作的是Oracle数据库，那么还需要导入c3p0-oracle-thin-extras-0.9.2-pre1.jar  
**2、在类目录下加入C3P0的配置文件：c3p0-config.xml**

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

<!--

c3p0-config.xml必须位于类路径下面

private static ComboPooledDataSource ds;

static{

try {

ds = new ComboPooledDataSource("MySQL");

} catch (Exception e) {

throw new ExceptionInInitializerError(e);

}

}

-->

<c3p0-config>

<!--

C3P0的缺省(默认)配置，

如果在代码中“ComboPooledDataSource ds = new ComboPooledDataSource();”这样写就表示使用的是C3P0的缺省(默认)配置信息来创建数据源

-->

<default-config>

<property name="driverClass">com.mysql.jdbc.Driver</property>

<property name="jdbcUrl">jdbc:mysql://localhost:3306/jdbcstudy</property>

<property name="user">root</property>

<property name="password">XDP</property>

<property name="acquireIncrement">5</property>

<property name="initialPoolSize">10</property>

<property name="minPoolSize">5</property>

<property name="maxPoolSize">20</property>

</default-config>

<!--

C3P0的命名配置，

如果在代码中“ComboPooledDataSource ds = new ComboPooledDataSource("MySQL");”这样写就表示使用的是name是MySQL的配置信息来创建数据源

-->

<named-config name="MySQL">

<property name="driverClass">com.mysql.jdbc.Driver</property>

<property name="jdbcUrl">jdbc:mysql://localhost:3306/jdbcstudy</property>

<property name="user">root</property>

<property name="password">XDP</property>

<property name="acquireIncrement">5</property>

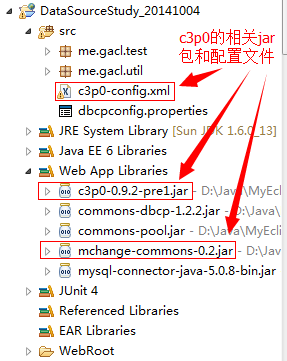
<property name="initialPoolSize">10</property>

<property name="minPoolSize">5</property>

<property name="maxPoolSize">20</property>

</named-config>

</c3p0-config>



**在获取数据库连接的工具类(如jdbcUtils)的静态代码块中创建池**

package me.gacl.util;

import java.sql.Connection;

import java.sql.ResultSet;

import java.sql.SQLException;

import java.sql.Statement;

import com.mchange.v2.c3p0.ComboPooledDataSource;

/\*\*

\* @ClassName: JdbcUtils\_C3P0

\* @Description: 数据库连接工具类

\* @author: 孤傲苍狼

\* @date: 2014-10-4 下午6:04:36

\*

\*/

public class JdbcUtils\_C3P0 {

private static ComboPooledDataSource ds = null;

//在静态代码块中创建数据库连接池

static{

try{

//通过代码创建C3P0数据库连接池

/\*ds = new ComboPooledDataSource();

ds.setDriverClass("com.mysql.jdbc.Driver");

ds.setJdbcUrl("jdbc:mysql://localhost:3306/jdbcstudy");

ds.setUser("root");

ds.setPassword("XDP");

ds.setInitialPoolSize(10);

ds.setMinPoolSize(5);

ds.setMaxPoolSize(20);\*/

//通过读取C3P0的xml配置文件创建数据源，C3P0的xml配置文件c3p0-config.xml必须放在src目录下

//ds = new ComboPooledDataSource();//使用C3P0的默认配置来创建数据源

ds = new ComboPooledDataSource("MySQL");//使用C3P0的命名配置来创建数据源

}catch (Exception e) {

throw new ExceptionInInitializerError(e);

}

}

public static Connection getConnection() throws SQLException{

//从数据源中获取数据库连接

return ds.getConnection();

}

public static void release(Connection conn,Statement st,ResultSet rs){

if(rs!=null){

try{

//关闭存储查询结果的ResultSet对象

rs.close();

}catch (Exception e) {

e.printStackTrace();

}

rs = null;

}

if(st!=null){

try{

//关闭负责执行SQL命令的Statement对象

st.close();

}catch (Exception e) {

e.printStackTrace();

}

}

if(conn!=null){

try{

//将Connection连接对象还给数据库连接池

conn.close();

}catch (Exception e) {

e.printStackTrace();

}

}

}

}

**测试C3P0数据源**

package me.gacl.test;

import java.sql.Connection;

import java.sql.PreparedStatement;

import java.sql.ResultSet;

import org.junit.Test;

import me.gacl.util.JdbcUtils\_C3P0;

import me.gacl.util.JdbcUtils\_DBCP;

public class DataSourceTest {

@Test

public void c3p0DataSourceTest() {

Connection conn = null;

PreparedStatement st = null;

ResultSet rs = null;

try{

//获取数据库连接

conn = JdbcUtils\_C3P0.getConnection();

String sql = "insert into test1(name) values(?)";

st = conn.prepareStatement(sql);

st.setString(1, "gacl");

st.executeUpdate();

//获取数据库自动生成的主键

rs = st.getGeneratedKeys();

if(rs.next()){

System.out.println(rs.getInt(1));

}

}catch (Exception e) {

e.printStackTrace();

}finally{

//释放资源

JdbcUtils\_C3P0.release(conn, st, rs);

}

}

}

### Tomcat数据源

Tomcat服务器也给我们提供了连接池，内部其实就是DBCP

**JNDI技术简介**

**JNDI(Java Naming and Directory Interface)，Java命名和目录接口**，它对应于J2SE中的javax.naming包，  
　　**这套API的主要作用在于：它可以把Java对象放在一个容器中（JNDI容器），并为容器中的java对象取一个名称，以后程序想获得Java对象，只需 通过名称检索即可。其核心API为Context，它代表JNDI容器，其lookup方法为检索容器中对应名称的对象。**

Tomcat服务器创建的数据源是以JNDI资源的形式发布的，所以说在Tomat服务器中配置一个数据源实际上就是在配置一个JNDI资源，通过查看Tomcat文档，我们知道使用如下的方式配置tomcat服务器的数据源：

<Context>

<Resource name="jdbc/datasource" auth="Container"

type="javax.sql.DataSource" username="root" password="111111"

driverClassName="com.mysql.jdbc.Driver"

url="jdbc:mysql://localhost:3306/jdbcstudy"

maxActive="8" maxIdle="4"/>

</Context>

服务器创建好数据源之后，我们的应用程序又该怎么样得到这个数据源呢，Tomcat服务器创建好数据源之后是**以JNDI的形式绑定到一个JNDI容器中的**，我们可以把JNDI想象成一个大大的容器，我们可以往这个容器中存放一些对象，一些资源，JNDI容器中存放的对象和资源都会有一个独一无二的名称，应用程序想从JNDI容器中获取资源时，只需要**告诉JNDI容器要获取的资源的名称**，JNDI根据名称去找到对应的资源后返回给应用程序。我们平时做javaEE开发时，服务器会为我们的应用程序创建很多资源，比如request对象，response对象，服务器创建的这些资源有两种方式提供给我们的应用程序使用：第一种是通过方法参数的形式传递进来，比如我们在Servlet中写的doPost和doGet方法中使用到的request对象和response对象就是服务器以参数的形式传递给我们的。第二种就是JNDI的方式，**服务器把创建好的资源绑定到JNDI容器中去**，应用程序想要使用资源时，就直接从JNDI容器中获取相应的资源即可。

对于上面的name="jdbc/datasource"数据源资源，在应用程序中可以用如下的代码去获取

1 Context initCtx = new InitialContext();//初始化JNDI容器

2 Context envCtx = (Context) initCtx.lookup("java:comp/env");//获取到JNDI容器

3 dataSource = (DataSource)envCtx.lookup("jdbc/datasource");//获取到JNDI容器

　　**此种配置下，数据库的驱动jar文件需放置在tomcat的lib下**

步骤：

1. 在META-INF目录下配置context.xml文件【文件内容可以在tomcat默认页面的 JNDI Resources下Configure Tomcat's Resource Factory找到】
2. 导入Mysql或oracle开发包到tomcat的lib目录下
3. 初始化JNDI->获取JNDI容器->检索以XXX为名字在JNDI容器存放的连接池

# 四、[Apache的DBUtils框架学习](https://www.cnblogs.com/xdp-gacl/p/4007225.html)

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。因此dbutils成为很多不喜欢hibernate的公司的首选。

　　commons-dbutilsAPI介绍：

* org.apache.commons.dbutils.QueryRunner
* org.apache.commons.dbutils.ResultSetHandler

　　工具类

* org.apache.commons.dbutils.DbUtils

## DbUtils类

提供了**关闭连接，装载JDBC驱动，回滚提交事务等方法**的工具类【比较少使用，因为我们学了连接池，就应该使用连接池连接数据库】

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 commitAndCloseQuietly(Connection conn)： 用来提交连接，然后关闭连接，并且在关闭连接时不抛出SQL异常。   
　　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。

## QueryRunner类

该类简单化了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：

* 默认的构造方法
* 需要一个 javax.sql.DataSource 来作参数的构造方法。

### QueryRunner类的主要方法

public Object query(Connection conn, String sql, Object[] params, ResultSetHandler rsh) 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。  
　　public Object query(String sql, Object[] params, ResultSetHandler rsh) throws SQLException:　几乎与第一种方法一样；唯一的不同在于它不将数据库连接提供给方法，并且它是从提供给构造方法的数据源(DataSource) 或使用的setDataSource 方法中重新获得 Connection。  
　　public Object query(Connection conn, String sql, ResultSetHandler rsh) throws SQLException : 执行一个不需要置换参数的查询操作。  
　　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。  
　　public int update(Connection conn, String sql) throws SQLException：用来执行一个不需要置换参数的更新操作。

### 使用QueryRunner类实现CRUD

package me.gacl.test;

public class QueryRunnerCRUDTest {

/\*

\*测试表

create table users(

id int primary key auto\_increment,

name varchar(40),

password varchar(40),

email varchar(60),

birthday date

);

\*/

@Test

public void add() throws SQLException {

//将数据源传递给QueryRunner，QueryRunner内部通过数据源获取数据库连接

QueryRunner qr = new QueryRunner(JdbcUtils.getDataSource());

String sql = "insert into users(name,password,email,birthday) values(?,?,?,?)";

Object params[] = {"孤傲苍狼","123", "gacl@sina.com", new Date()};

//Object params[] = {"白虎神皇","123", "gacl@sina.com", "1988-05-07"};

qr.update(sql, params);

}

@Test

public void delete() throws SQLException {

QueryRunner qr = new QueryRunner(JdbcUtils.getDataSource());

String sql = "delete from users where id=?";

qr.update(sql, 1);

}

@Test

public void update() throws SQLException {

QueryRunner qr = new QueryRunner(JdbcUtils.getDataSource());

String sql = "update users set name=? where id=?";

Object params[] = { "ddd", 5};

qr.update(sql, params);

}

@Test

public void find() throws SQLException {

QueryRunner qr = new QueryRunner(JdbcUtils.getDataSource());

String sql = "select \* from users where id=?";

Object params[] = {2};

User user = (User) qr.query(sql, params, new BeanHandler(User.class));

System.out.println(user.getBirthday());

}

@Test

public void getAll() throws SQLException {

QueryRunner qr = new QueryRunner(JdbcUtils.getDataSource());

String sql = "select \* from users";

List list = (List) qr.query(sql, new BeanListHandler(User.class));

System.out.println(list.size());

}

/\*\*

\* @Method: testBatch

\* @Description:批处理

\*/

@Test

public void testBatch() throws SQLException {

QueryRunner qr = new QueryRunner(JdbcUtils.getDataSource());

String sql = "insert into users(name,password,email,birthday) values(?,?,?,?)";

Object params[][] = new Object[10][];

for (int i = 0; i < 10; i++) {

params[i] = new Object[] { "aa" + i, "123", "aa@sina.com",

new Date() };

}

qr.batch(sql, params);

}

## ResultSetHandler接口

### ResultSetHandler接口的实现类

* 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。
* 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。
* 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。
* 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