JDBC

Драйверы, соединения и запросы

JDBC (Java DataBase Connectivity) – стандартный прикладной интерфейс (API) языка Java для организации взаимодействия между приложением и СУБД. Это взаимодействие осуществляется с помощью драйверов JDBC, обеспечивающих реализацию общих интерфейсов для конкретных СУБД и конкретных протоколов. В JDBC определяются четыре типа драйверов:

1. Драйвер, использующий другой прикладной интерфейс взаимодействия с СУБД, в частности ODBC (так называемый JDBC-ODBC – мост). Стандартный драйвер первого типа **`sun.jdbc.odbc.JdbcOdbcDriver`** входит в JDK.
2. Драйвер, работающий через внешние (native) библиотеки (т.е. клиента СУБД).
3. Драйвер, работающий по сетевому и независимому от СУБД протоколу с промежуточным Java-сервером, который, в свою очередь, подключается к нужной СУБД.
4. Сетевой драйвер, работающий напрямую с нужной СУБД и не требующий установки native-библиотек.

Предпочтение естественным образом отдается второму типу, однако если приложение выполняется на машине, на которой не предполагается установка клиента СУБД, то выбор производится между третьим и четвертым типами. Причем четвертый тип работает напрямую с СУБД по ее протоколу, поэтому можно предположить, что драйвер четвертого типа будет более эффективным по сравнению с третьим типом с точки зрения производительности. Первый же тип, как правило, используется редко, т.е. в тех случаях, когда у СУБД нет своего драйвера JDBC, зато есть драйвер ODBC.

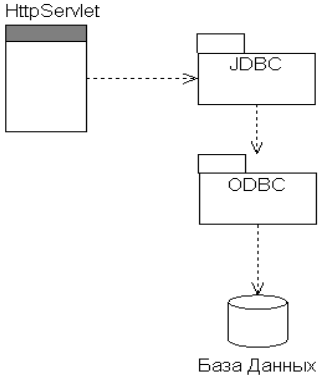

Рис. 20.1. Доступ к БД с помощью JDBC-ODBC-моста

JDBC предоставляет интерфейс для разработчиков, использующих различные СУБД. С помощью JDBC отсылаются SQL-запросы только к реляционным базам данных (БД), для которых существуют драйверы, знающие способ общения с реальным сервером базы данных.

Строго говоря, JDBC не имеет прямого отношения к J2EE, но так как взаимодействие с СУБД является неотъемлемой частью Web-приложений, то эта технология рассматривается в данном контексте.

Последовательность действий

1. *Загрузка класса драйвера базы данных при отсутствии экземпляра этого класса.*

Например:

```
String driverName = "org.gjt.mm.mysql.Driver";
```

для СУБД MySQL,

```
String driverName = "sun.jdbc.odbc.JdbcOdbcDriver";
```

для СУБД MSAccess или

```
String driverName = "org.postgreesql.Driver";
```

для СУБД PostgreeSQL.

После этого выполняется собственно загрузка драйвера в память:

```
Class.forName(driverName);
```

и становится возможным соединение с СУБД.

Эти же действия можно выполнить, импортируя библиотеку и создавая объект явно. Например, для СУБД DB2 от IBM объект-драйвер можно создать следующим образом:

```
new com.ibm.db2.jdbc.net.DB2Driver();
```

2. *Установка соединения с БД.*

Для установки соединения с БД вызывается статический метод **`getConnection()`** класса **`DriverManager`**. В качестве параметров методу передаются URL базы данных, логин пользователя БД и пароль доступа. Метод возвращает объект **`Connection`**. URL базы данных, состоящий

из типа и адреса физического расположения БД, может создаваться в виде отдельной строки или извлекаться из файла ресурсов. Например:

```
Connection cn = DriverManager.getConnection( "jdbc:mysql://localhost/my_db",
"root", "pass");
```

В результате будет возвращен объект **Connection** и будет одно установленное соединение с БД **my_db**. Класс **DriverManager** предоставляет средства для управления набором драйверов баз данных. С помощью метода
registerDriver() драйверы регистрируются, а методом **getDrivers()** можно получить список всех драйверов.

3. *Создание объекта для передачи запросов.*

После создания объекта **Connection** и установки соединения можно начинать работу с БД с помощью операторов SQL. Для выполнения запросов применяется объект **Statement**, создаваемый вызовом метода **createStatement()** класса **Connection**.

```
Statement st = cn.createStatement();
```

Объект класса **Statement** используется для выполнения SQL-запроса без его предварительной подготовки. Могут применяться также объекты классов **PreparedStatement** и **CallableStatement** для выполнения подготовленных запросов и хранимых процедур. Созданные объекты можно использовать для выполнения запроса SQL, передавая его в один из методов **executeQuery(String sql)** или **executeUpdate(String sql)**.

4. *Выполнение запроса.*

Результаты выполнения запроса помещаются в объект **ResultSet**:

```
ResultSet rs = st.executeQuery(
"SELECT * FROM my_table");//выборка всех данных таблицы my_table
```

Для добавления, удаления или изменения информации в таблице вместо метода **executeQuery()** запрос помещается в метод **executeUpdate()**.

5. *Обработка результатов выполнения запроса* производится методами интерфейса **ResultSet**, где самыми распространенными являются **next()** и **getString(int pos)** а также аналогичные методы, начинающиеся с **get**Тип**(int pos)** (**getInt(int pos)**, **getFloat(int pos)** и др.) и **update**Тип**()**. Среди них следует выделить методы **getClob(int pos)** и **getBlob(int pos)**, позволяющие извлекать из полей таблицы специфические объекты (Character Large Object, Binary Large Object), которые могут быть, например, графическими или архивными файлами. Эффективным способом извлечения значения поля из таблицы ответа является обращение к этому полю по его позиции в строке.

При первом вызове метода **next()** указатель перемещается на таблицу результатов выборки в позицию первой строки таблицы ответа. Когда строки закончатся, метод возвратит значение **false**.

6. *Закрытие соединения*

```
cn.close();
```

После того как база больше не нужна, соединение закрывается.

Для того чтобы правильно пользоваться приведенными методами, программисту требуется знать типы полей БД. В распределенных системах это знание предполагается изначально.

СУБД MySQL

СУБД MySQL совместима с JDBC и будет применяться для создания экспериментальных БД. Последняя версия СУБД может быть загружена с сайта **www.mysql.com**. Для корректной установки необходимо следовать инструкциям мастера установки. Каталог лучше выбирать по умолчанию. В процессе установки следует создать администратора СУБД с именем `root` и паролем `pass`. Если планируется разворачивать реально работающее приложение, необходимо исключить тривиальных пользователей сервера БД (иначе злоумышленники могут получить полный доступ к БД). Для запуска следует использовать команду из папки **/mysql/bin**:

```
mysqld-nt -standalone
```

Если не появится сообщение об ошибке, то СУБД MySQL запущена. Для создания БД и таблиц используются команды языка SQL.

Дополнительно требуется подключить библиотеку, содержащую драйвер MySQL

mysql-connector-java-3.1.12.jar,

и разместить ее в каталоге **/WEB-INF/lib** проекта.

Простое соединение и простой запрос

Теперь следует воспользоваться всеми предыдущими инструкциями и создать пользовательскую БД с именем **db2** и одной таблицей **users**. Таблица должна содержать два поля: символьное – **name** и числовое – **phone** и несколько занесенных записей. Сервлет, осуществляющий простейший запрос на выбор всей информации из таблицы, выглядит следующим образом.

```
/* пример # 1 : соединение с базой : ServletToBase.java */
package chapt20;
import java.io.*;
import java.sql.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class ServletToBase extends HttpServlet {
      public void doGet(HttpServletRequest req,
                        HttpServletResponse resp)
                        throws ServletException {
            performTask(req, resp);
      }
      public void doPost(HttpServletRequest req,
                         HttpServletResponse resp)
                         throws ServletException {
            performTask(req, resp);
      }

      public void showInfo(PrintWriter out, ResultSet rs)
                                    throws SQLException {
            out.print("From DataBase:");
        while (rs.next()) {
            out.print("<br>Name:-> " + rs.getString(1)
                        + " Phone:-> " + rs.getInt(2));
            }
      }
      public void performTask(HttpServletRequest req,
                              HttpServletResponse resp) {
      resp.setContentType("text/html; charset=Cp1251");
      PrintWriter out = null;
            try {//1
                  out = resp.getWriter();
                  try {//2

      Class.forName("org.gjt.mm.mysql.Driver");
            // для MSAccess
            /* return "sun.jdbc.odbc.JdbcOdbcDriver" */
            // для PostgreeSQL
            /* return " org.postgreesql.Driver " */
      Connection cn = null;
                        try {//3
      cn =
DriverManager.getConnection("jdbc:mysql://localhost/db2",
                                              "root", "pass");
            // для MSAccess
            /* return "jdbc:odbc:db2"; */
            // для PostgreeSQL
            /* return "jdbc:postgreesql://localhost/db2"; */

            Statement st = null;
                              try {//4
            st = cn.createStatement();
                        ResultSet rs = null;
                                    try {//5
            rs = st.executeQuery("SELECT * FROM users");
      out.print("From DataBase:");
        while (rs.next()) {
```

```
            out.print("<br>Name:-> " + rs.getString(1)
                        + " Phone:-> " + rs.getInt(2));
                              }
                              } finally {// для 5-го блока try
        /*      закрыть ResultSet, если он был открыт и ошибка
                произошла во время чтения из него данных  */
                      // проверка успел ли создаться ResultSet
                      if (rs != null)   rs.close();
                      else
        out.print("ошибка во время чтения данных из БД");
                                    }
                              } finally {// для 4-го блока try
        /*      закрыть Statement, если он был открыт и ошибка
                произошла во время создания ResultSet     */
                      // проверка успел ли создаться Statement
                      if (st != null)   st.close();
                      else  out.print("Statement не создан");
                                    }
                        } finally {// для 3-го блока try
        /*      закрыть Connection, если он был открыт и ошибка произошла
                 во время создания ResultSet или создания и использования
                 Statement  */
                // проверка - успел ли создаться Connection
                if (cn != null) cn.close();
                else  out.print("Connection не создан");
                        }
        } catch (ClassNotFoundException e) {// для 2-го блока try
        out.print("ошибка во время загрузки драйвера БД");
                    }
              }
        /* вывод сообщения о всех SQLException и IOException в блоках finally, */
              /* поэтому следующие блоки catch оставлены пустыми */
              catch (SQLException e) {
              }// для 1-го блока try
              catch (IOException e) {
              }// для 1-го блока try
              finally {// для 1-го блока try
        /*      закрыть PrintWriter, если он был инициализирован и ошибка
                произошла во время работы с БД  */
                // проверка, успел ли инициализироваться PrintWriter
                if (out != null)  out.close();
                else
                out.print("PrintWriter не проинициализирован");
                }
        }
}
```

В несложном приложении достаточно контролировать закрытие соединения, так как незакрытое (“провисшее”) соединение снижает быстродействие системы.

Еще один способ соединения с базой данных возможен с использованием файла ресурсов **database.properties**, в котором хранятся, как правило, путь к БД, логин и пароль доступа. Например:

```
url=jdbc:mysql://localhost/my_db?useUnicode=true&
characterEncoding=Cp1251
driver=org.gjt.mm.mysql.Driver
user=root
password=pass
```

В этом случае соединение создается в классе бизнес-логики, отвечающем за взаимодействие с базой данных, с помощью следующего кода:

```
public Connection getConnection()
      throws SQLException {
```

```
      ResourceBundle resource =
            ResourceBundle.getBundle("database");
            String url = resource.getString("url");
            String driver = resource.getString("driver");
            String user = resource.getString("user");
            String pass = resource.getString("password");
            try {
                  Class.forName(driver).newInstance();
            } catch (ClassNotFoundException e) {
      throw new SQLException("Драйвер не загружен!");
            } catch (InstantiationException e) {
                  e.printStackTrace();
            } catch (IllegalAccessException e) {
                  e.printStackTrace();
            }
return DriverManager.getConnection(url, user, pass);
}
```

Объект класса **ResourceBundle**, содержащий ссылки на все внешние ресурсы проекта, создается с помощью вызова статического метода **getBundle(String filename)**, с параметром в виде имени необходимого файла ресурсов. Если требуемый файл отсутствует, то генерируется исключительная ситуация **MissingResourceException**. Для чтения из объекта ресурсов используется метод **getString(String name)**, извлекающий информацию по указанному в параметре ключу. В классе **ResourceBundle** определен ряд полезных методов, в том числе метод **getKeys(),** возвращающий объект **Enumeration**, который применяется для последовательного обращения к элементам. Методы **getObject(String key)** и **getStringArray(String key)** извлекают соответственно объект и массив строк по передаваемому ключу.

Метаданные

Существует целый ряд методов интерфейсов **ResultSetMetaData** и **DatabaseMetaData** для интроспекции объектов. С помощью этих методов можно получить список таблиц, определить типы, свойства и количество столбцов БД. Для строк подобных методов нет.

Получить объект **ResultSetMetaData** можно следующим образом:

```
ResultSetMetaData rsMetaData = rs.getMetaData();
```

Некоторые методы интерфейса **ResultSetMetaData**:

int getColumnCount() – возвращает число столбцов набора результатов объекта **ResultSet**;

String getColumnName(int column) – возвращает имя указанного столбца объекта **ResultSet**;

int getColumnType(int column) – возвращает тип данных указанного столбца объекта **ResultSet** и т.д.

Получить объект **DatabaseMetaData** можно следующим образом:

```
DatabaseMetaData dbMetaData = cn.getMetaData();
```

Некоторые методы весьма обширного интерфейса **DatabaseMetaData**:

String getDatabaseProductName() – возвращает название СУБД;

String getDatabaseProductVersion() – возвращает номер версии СУБД;

String getDriverName() – возвращает имя драйвера JDBC;

String getUserName() – возвращает имя пользователя БД;

String getURL() – возвращает местонахождение источника данных;

ResultSet getTables() – возвращает набор типов таблиц, доступных для данной БД, и т.д.

Подготовленные запросы и хранимые процедуры

Для представления запросов существуют еще два типа объектов **PreparedStatement** и **CallableStatement**. Объекты первого типа используются при выполнении часто повторяющихся запросов SQL. Такой оператор предварительно готовится и хранится в объекте, что ускоряет обмен информацией с базой данных. Второй интерфейс используется для выполнения хранимых процедур, созданных средствами самой СУБД.

Для подготовки SQL-запроса, в котором отсутствуют конкретные параметры, используется метод **prepareStatement(String sql)** интерфейса **Connection**, возвращающий объект **PreparedStatement**. Установка входных значений конкретных параметров этого объекта производится с помощью методов **setString()**, **setInt()** и подобных им, после чего и осуществляется

непосредственное выполнение запроса методами **executeUpdate()**, **executeQuery()**. Так как данный оператор предварительно подготовлен, то он выполняется быстрее обычных операторов, ему соответствующих. Оценить преимущества во времени можно, выполнив большое число повторяемых запросов с предварительной подготовкой запроса и без нее.

```
/* пример # 2 : создание и выполнение подготовленного запроса :
PreparedStatementServlet.java */
package chapt20;
import java.io.*;
import java.sql.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class PreparedStatementServlet extends HttpServlet {
      protected void doGet(HttpServletRequest req,
                           HttpServletResponse resp)
            throws ServletException, IOException {
            performTask(req, resp);
      }
      protected void doPost(HttpServletRequest req,
                           HttpServletResponse resp)
            throws ServletException, IOException {
            performTask(req, resp);
      }
      protected void performTask(HttpServletRequest req,      HttpServletResponse
resp)
            throws ServletException, IOException {
            resp.setContentType("text/html");
            PrintWriter out = resp.getWriter();
            try {
      Class.forName("org.gjt.mm.mysql.Driver");
            Connection cn = null;
                  try {
            cn = DriverManager.getConnection(
"jdbc:mysql://localhost/db3","root","");
                  PreparedStatement ps = null;
                  String sql =
"INSERT INTO emp(id,name,surname,salary) VALUES(?,?,?,?)";
            //компиляция (подготовка) запроса
            ps = cn.prepareStatement(sql);
      Rec.insert(ps, 2203, "Иван", "Петров", 230);
      Rec.insert(ps, 2308, "John", "Black", 450);
      Rec.insert(ps, 2505, "Mike", "Call", 620);
            out.println("COMPLETE");
                  } finally {
                  if (cn != null) cn.close();
                  }
            } catch (Exception e) {
                  e.printStackTrace();
            }
            out.close();
      }
}
class Rec {
      static void insert(PreparedStatement ps, int id, String name, String sur-
name, int salary)
             throws SQLException {
            //установка входных параметров
            ps.setInt(1, id);
            ps.setString(2, name);
            ps.setString(3, surname);
            ps.setInt(4, salary);
            //выполнение подготовленного запроса
            ps.executeUpdate();
      }
```

```
}
```

Результатом выполнения данной программы будет добавление в базу данных **db3** трех записей и вывод в окно браузера слова COMPLETE.

Интерфейс **CallableStatement** расширяет возможности интерфейса **PreparedStatement** и обеспечивает выполнение хранимых процедур.

Хранимая процедура – это в общем случае именованная последовательность команд SQL, рассматриваемых как единое целое, и выполняющаяся в адресном пространстве процессов СУБД, который можно вызвать извне (в зависимости от политики доступа используемой СУБД). В данном случае хранимая процедура будет рассматриваться в более узком смысле как последовательность команд SQL, хранимых в БД и доступных любому пользователю этой СУБД. Механизм создания и настройки хранимых процедур зависит от конкретной базы данных. Для создания объекта **CallableStatement** вызывается метод **prepareCall()** объекта **Connection**.

Интерфейс **CallableStatement** позволяет исполнять хранимые процедуры, которые находятся непосредственно в БД. Одна из особенностей этого процесса в том, что **CallableStatement** способен обрабатывать не только входные (**IN**) параметры, но и выходящие (**OUT**) и смешанные (**INOUT**) параметры. Тип выходного параметра должен быть зарегистрирован методом **registerOutParameter()**. После установки входных и выходных параметров вызываются методы **execute()**, **executeQuery()** или **executeUpdate()**.

Пусть в БД существует хранимая процедура **getempname**, которая по уникальному для каждой записи в таблице **employee** числу SSN будет возвращать соответствующее ему имя:

```
CREATE PROCEDURE getempname
(emp_ssn IN INT, emp_name OUT VARCHAR) AS
BEGIN
SELECT name
INTO emp_name
FROM employee
WHERE SSN = EMP_SSN;
END
```

Тогда для получения имени служащего **employee** через вызов данной процедуры необходимо исполнить java-код вида:

```
String SQL = "{call getempname (?,?)}";
CallableStatement cs = conn.prepareCall(SQL);
cs.setInt(1,822301);
//регистрация выходящего параметра
cs.registerOutParameter(2,java.sql.Types.VARCHAR);
cs.execute();
String empName = cs.getString(2);
System.out.println("Employee with SSN:" + ssn
+ " is " + empName);
```

В результате будет выведено:

```
Employee with SSN:822301 is Spiridonov
```

В JDBC также существует механизм batch-команд, который позволяет запускать на исполнение в БД массив запросов SQL вместе, как одну единицу.

```
// turn off autocommit
con.setAutoCommit(false);
Statement stmt = con.createStatement();
stmt.addBatch("INSERT INTO employee VALUES
                                        (10, 'Joe ')");
stmt.addBatch("INSERT INTO location VALUES
                                        (260, 'Minsk')");
stmt.addBatch("INSERT INTO emp_dept VALUES
                                        (1000, 260)");
// submit a batch of update commands for execution
int[] updateCounts = stmt.executeBatch();
```

Если используется объект **PreparedStatement**, batch-команда состоит из параметризованного SQL-запроса и ассоциируемого с ним множества параметров.

Метод **PreparedStatement.executeBatch()** возвращает массив чисел, причем каждое характеризует число строк, которые были изменены конкретным запросом из batch-команды.

Пусть существует массив объектов типа **Employee** со стандартным набором методов **getТип()/setТип()** для каждого из его полей, и необходимо внести их значения в БД. Многократное выполнение методов **execute()** или **executeUpdate()** становится неэффективным, и в данном случае лучше использовать схему batch-команд:

```
try {
   Employee[] employees = new Employee[10];
   PreparedStatement statement =
   con.prepareStatement("INSERT INTO employee VALUES
                                        (?,?,?,?,?)");
   for (int i = 0; i < employees.length; i++) {
         Employee currEmployee = employees[i];
         statement.setInt(1, currEmployee.getSSN());
         statement.setString(2, currEmployee.getName());
   statement.setDouble(3, currEmployee.getSalary());
   statement.setString(4,currEmployee.getHireDate());
         statement.setInt(5, currEmployee.getLoc_Id());
         statement.addBatch();
   }
   updateCounts = statement.executeBatch();
   } catch (BatchUpdateException e) {
         e.printStackTrace();
   }
```

Транзакции

При проектировании распределенных систем часто возникают ситуации, когда сбой в системе или какой-либо ее периферийной части может привести к потере информации или к финансовым потерям. Простейшим примером может служить пример с перечислением денег с одного счета на другой. Если сбой произошел в тот момент, когда операция снятия денег с одного счета уже произведена, а операция зачисления на другой счет еще не произведена, то система, позволяющая такие ситуации, должна быть признана не отвечающей требованиям заказчика. Или должны выполняться обе операции, или не выполняться вовсе.
Такие две операции трактуют как одну и называют транзакцией.

Транзакцию (деловую операцию) определяют как единицу работы, обладающую свойствами ACID:

- Атомарность – две или более операций выполняются все или не выполняется ни одна. Успешно завершенные транзакции фиксируются, в случае неудачного завершения происходит откат всей транзакции.
- Согласованность – при возникновении сбоя система возвращается в состояние до начала неудавшейся транзакции. Если транзакция завершается успешно, то проверка согласованности удостоверяется в успешном завершении всех операций транзакции.
- Изолированность – во время выполнения транзакции все объекты-сущности, участвующие в ней, должны быть синхронизированы.
- Долговечность – все изменения, произведенные с данными во время транзакции, сохраняются, например, в базе данных. Это позволяет восстанавливать систему.

Для фиксации результатов работы SQL-операторов, логически выполняемых в рамках некоторой транзакции, используется SQL-оператор `COMMIT`. В API JDBC эта операция выполняется по умолчанию после каждого вызова методов
`executeQuery()` и **`executeUpdate()`**. Если же необходимо сгруппировать запросы и только после этого выполнить операцию `COMMIT`, сначала вызывается метод **`setAutoCommit(boolean param)`** интерфейса **`Connection`** с параметром **`false`**, в результате выполнения которого текущее соединение с БД переходит в режим неавтоматического подтверждения операций. После этого выполнение любого запроса на изменение информации в таблицах базы данных не приведет к необратимым последствиям, пока операция `COMMIT` не будет выполнена непосредственно. Подтверждает выполнение SQL-запросов метод **`commit()`** интерфейса **`Connection`**, в результате действия которого все изменения таблицы производятся как одно логическое действие. Если же транзакция не выполнена, то методом **`rollback()`** отменяются действия всех запросов SQL, начиная от последнего вызова **`commit()`**. В следующем примере информация добавляется в таблицу в режиме действия транзакции, подтвердить или отменить действия которой можно, снимая или добавляя комментарий в строках вызова методов **`commit()`** и **`rollback()`**.

```
<!-- пример # 3 : вызов сервлета : index.jsp -->
<%@ page language="java" contentType="text/html; charset=windows-1251" pageEncod-
ing="windows-1251"%>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional //EN">
<html><head>
<meta http-equiv="Content-Type"
      content="text/html; charset=windows-1251">
```

```
<title>Simple Transaction Demo</title>
</head>
<body>
<form name="students" method="POST"
      action="SQLTransactionServlet">

      id:<br/>
      <input type="text" name="id" value=""><br/>
      Name:<br/>
      <input type="text" name="name" value=""><br/>

      Course:<br/>
      <select name="course">
        <option>Java SE 6
        <option>XML
        <option>Struts
      </select><br/>

      <input type="submit" value="Submit">
</form>
</body></html>
```

/ пример # 4 : выполнение транзакции : метод perform() сервлета SQLTransactionServlet.java */*

```
public void taskPerform(HttpServletRequest request,
                  HttpServletResponse response)
                  throws ServletException, IOException {
 response.setContentType("text/html; charset=Cp1251");
 PrintWriter out = null;
 Connection cn = null;
 try {
      out = response.getWriter();
      String id = request.getParameter("id");
      String name = request.getParameter("name");
      String course = request.getParameter("course");
      out.print("ID студента: " + id + ", " + name +<br>");
            cn = getConnection();
            cn.setAutoCommit(false);
            Statement st = cn.createStatement();
      try {
            String upd;
            upd =
      "INSERT INTO student (id, name) VALUES ('"
                              + id  + "', '" + name + "')";
            st.executeUpdate(upd);
            out.print("Внесены данные в students: "
                              + id + ", " + name + "<br>");

            upd =
 "INSERT INTO course(id_student, name_course) VALUES('"
                              + id + "','" + course + "')";
            st.executeUpdate(upd);
            out.print("Внесены данные в course: " + id
                              + ", " + course + "<br>");

            cn.commit();// подтверждение
 out.print("<b>Данные внесены - транзакция завершена"
                                    + "</b><br>");
      } catch (SQLException e) {
            cn.rollback();// откат
            out.println("<b>Произведен откат транзакции:"
                        + e.getMessage()  + "</b>");
      } finally {
            if (cn != null)
            cn.close();
```

```
        }
    } catch (SQLException e) {
        out.println("<b>ошибка при закрытии соединения:"
                                        + e.getMessage());
    }
}
```

Если таблицы `student` и `course` базы данных **`db1`** до изменения выглядели, например, следующим образом,

id	name
71	Goncharenko

id_student	name_course
71	Java SE 6

Рис. 20.2. Таблицы до выполнения запроса

то после внесения изменений и их подтверждения они примут вид:

id	name
71	Goncharenko
83	Petrov

id_student	name_course
71	Java SE 6
83	XML

Рис. 20.3. Таблицы после подтверждения выполнения запросов

ID студента: 83, Petrov
Внесены данные в students: 83, Petrov
Внесены данные в course: 83, XML
Данные внесены - транзакция завершена

Приведенный пример в полной мере не отражает принципы транзакции, но демонстрирует способы ее поддержки методами языка Java.

Для транзакций существует несколько типов чтения:

- Грязное чтение (dirty reads) происходит, когда транзакциям разрешено видеть несохраненные изменения данных. Иными словами, изменения, сделанные в одной транзакции, видны вне ее до того, как она была сохранена. Если изменения не будут сохранены, то, вероятно, другие транзакции выполняли работу на основе некорректных данных;
- Непроверяющееся чтение (nonrepeatable reads) происходит, когда транзакция А читает строку, транзакция Б изменяет эту строку, транзакция А читает ту же строку и получает обновленные данные;
- Фантомное чтение (phantom reads) происходит, когда транзакция А считывает все строки, удовлетворяющие **`WHERE`**-условию, транзакция Б вставляет новую или удаляет одну из строк, которая удовлетворяет этому условию, транзакция А еще раз считывает все строки, удовлетворяющие **`WHERE`**-условию, уже вместе с новой строкой или недосчитавшись старой.

JDBC удовлетворяет четырем уровням изоляции транзакций, определенным в стандарте SQL:2003.

Уровни изоляции транзакций определены в виде констант интерфейса **`Connection`** (по возрастанию уровня ограничения):

- **`TRANSACTION_NONE`** – информирует о том, что драйвер не поддерживает транзакции;
- **`TRANSACTION_READ_UNCOMMITTED`** – позволяет транзакциям видеть несохраненные изменения данных, что разрешает грязное, непроверяющееся и фантомное чтения;
- **`TRANSACTION_READ_COMMITTED`** – означает, что любое изменение, сделанное в транзакции, не видно вне неё, пока она не сохранена. Это предотвращает грязное чтение, но разрешает непроверяющееся и фантомное;
- **`TRANSACTION_REPEATABLE_READ`** – запрещает грязное и непроверяющееся, но фантомное чтение разрешено;
- **`TRANSACTION_SERIALIZABLE`** – определяет, что грязное, непроверяющееся и фантомное чтения запрещены.

Метод **`boolean supportsTransactionIsolationLevel(int level)`** интерфейса **`DatabaseMetaData`** определяет, поддерживается ли заданный уровень изоляции транзакций.

В свою очередь, методы интерфейса **`Connection`** определяют доступ к уровню изоляции:

`int getTransactionIsolation()` – возвращает текущий уровень изоляции;

`void setTransactionIsolation(int level)` – устанавливает нужный уровень.

Точки сохранения

Точки сохранения дают дополнительный контроль над транзакциями. Установкой точки сохранения обозначается логическая точка внутри транзакции, которая может быть использована для отката данных. Таким образом, если произойдет ошибка, можно вызвать метод **`rollback()`** для отмены всех изменений, которые были сделаны после точки сохранения.

Метод **boolean supportsSavepoints()** интерфейса **DatabaseMetaData** используется для того, чтобы определить, поддерживает ли точки сохранения драйвер JDBC и сама СУБД.

Методы **setSavepoint(String name)** и **setSavepoint()** (оба возвращают объект **Savepoint**) интерфейса **Connection** используются для установки именованной или неименованной точки сохранения во время текущей транзакции. При этом новая транзакция будет начата, если в момент вызова **setSavepoint()** не будет активной транзакции.

```
/* пример # 5 : применение точек сохранения : SavepointServlet.java */
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.IOException;
import java.io.PrintWriter;
import java.sql.*;

public class SavepointServlet extends HttpServlet {
      protected void doGet(HttpServletRequest req,
            HttpServletResponse resp)
            throws ServletException, IOException {
                  performTask(req, resp);
            }
      protected void doPost(HttpServletRequest req,
                  HttpServletResponse resp)
            throws ServletException, IOException {
                  performTask(req, resp);
            }
      protected void performTask(HttpServletRequest req,                HttpS-
ervletResponse resp)
            throws ServletException, IOException {
      resp.setContentType("text/html; charset=Cp1251");
                  PrintWriter out = resp.getWriter();
                  try {
                  Class.forName("org.gjt.mm.mysql.Driver");
                        Connection cn = null;
                        Savepoint savepoint = null;
                        try {
                        cn = DriverManager.getConnection(
      "jdbc:mysql://localhost/db3","root","pass");
                  cn.setAutoCommit(false);
                  out.print("<b>Соединение с БД...</b>");
                  out.print("<br>");
                  Statement stmt = cn.createStatement();
                  String trueSQL =
            "INSERT INTO emp (id,name,surname,salary) "
                  + "VALUES(2607,'Петя','Иванов',540)";
                  stmt.executeUpdate(trueSQL);
                              //установка точки сохранения
                  savepoint =
                        cn.setSavepoint("savepoint1");
                              //выполнение некорректного запроса
                  String wrongSQL =
                  "INSERT INTO (id,name,surname,salary) "
                  + "VALUES(2607,'Петя','Иванов',540)";
                  stmt.executeUpdate(wrongSQL);
                        } catch (SQLException ex) {
                              out.print(ex + "<br>");
                              cn.rollback(savepoint);
                  out.print("<b>Откат к точке сохранения: "
                                    + savepoint + "</b>");
                        } finally {
                              if (cn != null) cn.close();
                        }
                  } catch (Exception e) {
                        e.printStackTrace();
                  }
```

```
                out.close();
            }
}
```

В результате в браузер будет выведено:

```
Соединение с БД...
java.sql.SQLException: You have an error in your SQL syntax ...
Откат к точке сохранения: savepoint1
```

При этом результаты выполнения запроса **trueSQL** будут сохранены перед попыткой повторения транзакции.

Пул соединений

При большом количестве клиентов, работающих с приложением, к его базе данных выполняется большое количество запросов. Установление соединения с БД является дорогостоящей (по требуемым ресурсам) операцией. Эффективным способом решения данной проблемы является организация пула (pool) используемых соединений, которые не закрываются физически, а хранятся в очереди и предоставляются повторно для других запросов.

Пул соединений – это одна из стратегий предоставления соединений приложению (не единственная, да и самих стратегий организации пула существует несколько).

Пул соединений можно организовать с помощью **server.xml** дескрипторного файла Apache Tomcat в виде:

```
<Context docBase="FirstProject" path="/FirstProject"
reloadable="true" source="com.ibm.etools.webtools.server: FirstProject">
<!—создание пул соединений для СУБД MySQL -->
<Resource auth="Container" name="jdbc/db1"
                           type="javax.sql.DataSource"/>
<ResourceParams name="jdbc/db1">
      <parameter>
            <name>factory</name>
      <value>org.apache.commons.dbcp.BasicDataSourceFactory</value>
      </parameter>
      <parameter>
            <name>driverClassName</name>
            <value>org.gjt.mm.mysql.Driver</value>
      </parameter>
<!—url-адрес соединения JDBC для конкретной базы данных db1
Аргумент autoReconnect=true, заданный для url-адреса драйвера JDBC, должен автоматически разорвать соединение, если mysqld его закроет. По умолчанию mysqld закроет соединение через 8 часов.-->
      <parameter>
            <name>url</name>
<value>jdbc:mysql://localhost:3306/db1?autoReconnect=true
</value>
      </parameter>
      <parameter>
            <name>username</name>
            <value>root</value>
      </parameter>
      <parameter>
            <name>password</name>
            <value>pass</value>
      </parameter>
      <parameter>
            <name>maxActive</name>
            <value>500</value>
      </parameter>
      <parameter>
            <name>maxIdle</name>
            <value>10</value>
      </parameter>
      <parameter>
            <name>maxWait</name>
            <value>10000</value>
      </parameter>
      <parameter>
            <name>removeAbandoned</name>
```

```
            <value>true</value>
        </parameter>
        <parameter>
            <name>removeAbandonedTimeout</name>
            <value>60</value>
        </parameter>
        <parameter>
            <name>logAbandoned</name>
            <value>true</value>
        </parameter>
</ResourceParams>
</Context>
```

Найти и запустить данный пул соединений можно с помощью JNDI.

Разделяемый доступ к источнику данных можно организовать, например, путем объявления статической переменной типа **DataSource** из пакета **javax.sql**, однако в J2EE принято использовать для этих целей каталог. Источник данных типа **DataSource** – это компонент, предоставляющий соединение с приложением СУБД.

Класс **InitialContext**, как часть JNDI API, обеспечивает работу с каталогом именованных объектов. В этом каталоге можно связать объект источника данных **DataSource** с некоторым именем (не только с именем БД, но и вообще с любым), предварительно создав объект **DataSource**.

Затем созданный объект можно получить с помощью метода **lookup()** по его имени. Методу **lookup()** передается имя, всегда начинающееся с имени корневого контекста.

```
javax.naming.Context ct =
   new javax.naming.InitialContext();
DataSource ds = (DataSource)ct.lookup("java:jdbc/db1");
Connection cn = ds.getConnection("root", "pass");
```

После выполнения запроса соединение завершается, и его объект возвращается обратно в пул вызовом:

```
cn.close();
```

Некоторые производители СУБД для облегчения создания пула соединений определяют собственный класс на основе интерфейса **DataSource**. В этом случае пул соединений может быть создан, например, следующим образом:

```
import COM.ibm.db2.jdbc.DB2DataSource;
      ...
      DB2DataSource ds = new DB2DataSource();
      ds.setServerName("//localhost:50000/db1");
      Connection cn = ds.getConnection("db2inst1", "pass");
```

Драйвер определяется автоматически в объекте **DB2DataSource**.