JDBC

VL Datenbanksysteme

Ingo Feinerer

Arbeitsbereich Datenbanken und Artificial Intelligence
Institut für Informationssysteme
Technische Universität Wien

Gliederung

DB-Anbindung an Programmiersprachen

- Erweiterung der Programmiersprache:
  - DB-Konstrukte zur Programmiersprache hinzufügen
  - eher historische Bedeutung (80er Jahre)
  - z.b. Pascal/R („Relationales Pascal“)
- Embedded SQL:
  - Speziell markierte SQL-Anweisungen
  - Pre-Compiler wandelt diese in Aufrufe von (meist proprietären) Bibliotheksfunktionen der „Wirtssprache“ um.
  - z.b.: Embedded SQL in C/C++, SQLJ
- CLI (Call Level Interface):
  - Verwendung von standardisierten Prozeduraufrufen
  - z.B.: SQL/CLI, ODBC, JDBC

Embedded SQL Beispiel: C/C++

Beispiel

```
EXEC SQL BEGIN DECLARE SECTION;
  long matrnr;
  char name[30];
  int semester;
EXEC SQL END DECLARE SECTION;
EXEC SQL DECLARE c_studenten CURSOR FOR
  SELECT * FROM Studenten;
EXEC SQL OPEN c_studenten;

while (SQLCODE == 0) {
  EXEC SQL FETCH c_studenten INTO :matrnr, :name, :semester;
  printf("%d, %s %d", matrnr, name, semester);
}

EXEC SQL CLOSE c_studenten;
```

Embedded SQL Beispiel: SQLJ

Beispiel

```
#sql iterator StudIterator (int MatrNr, String Name, int Semester);

StudIterator i_stud;
#sql i_stud = { SELECT * FROM Studenten };

while (i_stud.next()) {
  System.out.println(i_stud.MatrNr() + ", "
                     + i_stud.Name() + ", "
                     + i_stud.Semester());
}

i_stud.close();
```

Embedded SQL: Merkmale

- Üblicherweise „statisches SQL“, d.h.:
  - Struktur der Anfragen steht zur Compile-Zeit fest.
  - Vor-/Nachteile: weitergehende Überprüfungen möglich (z.B.: Syntaxprüfung der SQL-Anfragen), aber unflexibel
  - Häufig wird auch „dynamisches SQL“ unterstützt (d.h.: Anfrage in String-Variable), z.B. Pro*C von Oracle.
- Informationsaustausch zw. SQL und Wirtsprache:
  - Host-Variablen und Cursor
- Portabilität:
  - Ursprünglich nur auf Source Code Ebene, d.h.: für Verwendung eines anderen DBMS war Recompilierung nötig.
  - SQLJ: Pre-Compiler („Translator“) erzeugt JDBC: Gleiche Flexibilität wie CLI (aber bequemer).

Call Level Interfaces

- Erster CLI-Standard:
  - Microsoft, Lotus, etc. definieren eine Menge von Funktionen unter dem Namen „Database Connectivity“
  - SAG (SQL Access Group: Oracle, Informix, etc.): auf Database Connectivity aufbauend wird ein CLI definiert.
  - 1992: CLI wird von ANSI und ISO als Basis eines neuen Standards übernommen.
- ODBC (Open Database Connectivity):
  - 1992: von Microsoft speziell für Windows entwickelt, später auch andere Betriebssysteme (insbesondere UNIX)
  - Implementierung und Erweiterung des CLI-Standards
  - Hauptsächlich für C/C++; aber auch andere Sprachen

Call Level Interfaces

- SQL/CLI:
  - Standardisierungsarbeiten der ANSI/ISO SQL Gruppen auf der Basis von ODBC.
  - 1995: Verabschiedung des Standards 9075-3: SQL/CLI
  - 1999: Erweiterung des SQL/CLI im Zug der SQL-Erweiterungen durch SQL-99
- JDBC:
  - Basierend auf ODBC aber weniger komplex, nützt Java-Möglichkeiten (z.B. objektorientiert, try-catch für Error Handling, . . .)
  - Versionen:
    - 1997: JDBC 1.0
    - in J2SE, ab Version 1.4: JDBC 3.0
    - letzte Version: JDBC 4.0

JDBC-Treiber Typen

- Type 1: JDBC-ODBC-Bridge + ODBC-driver
  - JDBC-ODBC-Bridge ruft ODBC-Methoden auf
  - ODBC-Treiber nötig
- Type 2: Native-API partly-Java driver
  - JDBC-Treiber ruft herstellerspezifische DB-Bibliotheken auf. Diese Bibliotheken sind am Client erforderlich.
  - z.B. Oracle OCI Driver
- Type 3: JDBC-Net pure Java driver
  - JDBC-Treiber ruft Middleware-spezifische Methoden auf.
  - Middleware stellt Connectivity zu unterschiedlichen DB-Systemen bereit.
- Type 4: Native-protocol pure Java driver
  - JDBC-Treiber kommuniziert direkt mit dem DB-Server.
  - Außer JDBC-Treiber keine weitere SW am Client nötig.
  - z.B. PostgreSQL JDBC Driver

JDBC-Klassen/Interfaces

package java.sql, z.B.:

- DriverManager
- Connection
- DatabaseMetaData
- Statement, PreparedStatement, CallableStatement
- ResultSet
- ResultSetMetaData
- SQLException, SQLWarning
- Types

package javax.sql, z.B.:

- DataSource

JDBC-Quellen

- `http://download-llnw.oracle.com/javase/6/docs/api/java/sql/package-tree.html`
- `http://download-llnw.oracle.com/javase/tutorial/jdbc/`

Beispiel

Beispiel

```
Class.forName("org.postgresql.Driver");
Connection c = DriverManager.getConnection(
  "jdbc:postgresql://localhost", "user", "passwd");
Statement stmt = c.createStatement();
ResultSet rs = stmt.executeQuery(
                          "SELECT * FROM students");
while(rs.next()) {
  System.out.print(rs.getString(1) + ", "
                   + rs.getString(3));
}
rs.close(); stmt.close(); c.close();
```

Gliederung

Verbindungsaufbau

- DB-spezifischer JDBC-Treiber im Classpath:
  Für die Laborübung: `postgresql-8.4-701.jdbc4.jar`
- Möglichkeiten einen Treiber zu registrieren:
  - `Class.forName("org.postgresql.Driver");`
  - ```
    DriverManager.registerDriver(
                  new org.postgresql.Driver());
    ```
- Verbindungsparameter in der Laborübung:

Treiber	jdbc:postgresql
Adresse	bordo.dbai.tuwien.ac.at:5432
Datenbank	Laut E-Mail der LVA-Leitung
Benutzer	Laut E-Mail der LVA-Leitung
Passwort	Laut E-Mail der LVA-Leitung
    ```

Verbindungsaufbau

- „Klassische“ Methode:
  `DriverManager.getConnection("...")`
- „Modernere“ Methode: mittels DataSource Interface
  - Die DB-Hersteller stellen spezifische Implementierungen von DataSource bereit.
  - Basis-Implementierung funktioniert im wesentlichen wie DriverManager.
  - Daneben stellen die DB-Hersteller aber auch Connection Pooling Implementierungen bereit.
  - Connection Objekt als Teil eines Connection Pools
  - Vorteil: Connection Pooling ist wesentlich Ressourcen-schonender.

Beispiel

Beispiel

```
DriverManager.registerDriver(new Driver());
PGSimpleDataSource ds = new PGSimpleDataSource();
ds.setServerName("bordo.dbai.tuwien.ac.at");
ds.setPortNumber(5432);
ds.setUser("u1234567");
ds.setPassword("password");
ds.setDatabaseName("u1234567");
Connection conn = ds.getConnection();
Statement stmnt = conn.createStatement();
```

Eigenschaften/Methoden

- DB-Metadaten: mittels `conn.getMetaData()`
  Informationen über die DB bzw. die Verbindung, z.B.:

```
DatabaseMetaData dbmeta = conn.getMetaData();
S.o.println(dbmeta.getDatabaseProductName());
S.o.println(dbmeta.getUserName());
```

- Transaktionssteuerung:
  - AUTOCOMMIT on/off: `conn.setAutoCommit(false);`
  - EOT: `conn.commit();` bzw. `conn.rollback();`
  - Isolation Level:
    `setTransactionIsolation(TRANSACTION_READ_COMMITTED);`

Verbindungsabbau

Ressourcenbelegung in der Datenbank:

- JDBC-Objekte belegen auch Ressourcen im Datenbank Server, z.B.: Oracle-Fehler: „max cursors exceeded“, wenn zu viele Statements gleichzeitig offen sind.
- Connections und Statements (und am besten auch ResultSets) sollten deshalb immer explizit geschlossen werden, sobald sie nicht mehr benötigt werden, z.B.:
  `rs.close(); stmt.close(); conn.close();`
- Hierarchie ResultSet $<$ Statement $<$ Connection, d.h.: Beim Schließen einer Connection werden implizit alle Statements dieser Connection geschlossen.

Fehlerbehandlung

- SQLException / SQLWarning
  - Die Methoden der JDBC-API werfen im Fehlerfall eine SQLException.
  - Connections, Statements und ResultSets können auch SQLWarnings produzieren. Diese werden aber nicht geworfen sondern müssen aktiv angefordert werden mittels `stmt.getWarnings()` oder `conn.getWarnings()`
  - Es kann auch mehrere SQLExceptions und SQLWarnings geben (als verkettete Liste).
- Fehler von SQLException und SQLWarning:

code	DB-spezifischer Fehlercode
state	SQL-State laut SQL/CLI Standard
msg	DB-spezifische Fehlermeldung

Fehlerbehandlung

Beispiel

```
try { ... // JDBC-Aufrufe
} catch (SQLException e) {
    while (e != null) {
      S.o.p("Code = " + e.getErrorCode());
      S.o.p("Message = " + e.getMessage());
      S.o.p("State = " + e.getSQLState());
      e = e.getNextException();
    }
} finally {
// sauberer: finally mit geschachteltem try-catch
    try { if (rs != null) { rs.close(); } } catch...
    try { if (stmt != null) { stmt.close(); } } catch...
    try { if (conn != null) { conn.close(); } } catch...
}
```

Gliederung

execute-Methoden: Überblick

- executeQuery():
  - für SELECT-Anweisungen
  - executeQuery() liefert ein ResultSet
- executeUpdate():
  - für Schreibzugriff: DDL- und DML-Befehle
  - executeUpdate() liefert einen Integer-Wert (= Anzahl der betroffenen Tupel, bei DDL ist dieser Wert immer 0)
- Weitere Methoden:
  - executeBatch(): zuerst mehrere SQL-Befehle mittels addBatch() speichern und dann gemeinsam abschicken
  - execute()

Beispiel: DDL + DML

Beispiel

```
count = stmt.executeUpdate
  ("DROP TABLE Abteilungen CASCADE");
count = stmt.executeUpdate
  ("CREATE TABLE Abteilungen (" +
   "AbtNr INTEGER PRIMARY KEY," +
   "Chef INTEGER," +
   "Bezeichnung VARCHAR(30)," +
   "Standort VARCHAR(30))");
count = stmt.executeUpdate
  ("ALTER TABLE Abteilungen " +
   "ADD CONSTRAINT chef_fk FOREIGN KEY (Chef) " +
   "REFERENCES Mitarbeiter (PersNr)");
count = stmt.executeUpdate
  ("INSERT INTO Abteilungen "+
   "VALUES (1001, 102, 'Abteilung A', 'Ort A')");
```

Gliederung

ResultSet-Verarbeitung

- SELECT-Anweisung verwendet die `executeQuery()` Methode. Diese liefert das Ergebnis als ResultSet.
- Das ResultSet ist ein Iterator, der mittels `next()` Methode die einzelnen Datensätze durchläuft.
- Der Zugriff auf die Spalten des aktuellen Datensatzes erfolgt durch typspezifische get-Methoden des ResultSet, z.B.: `getInt()`, `getString()`, `getDouble()`, `getDate()`, `...`
- Dieser Zugriff auf die Spalten geschieht entweder über den Spaltennamen oder über die Spaltenposition.
- Standardmäßig dient ein ResultSet nur zum sequentiellen Durchlaufen in Vorwärtsrichtung und nur zum Lesen. Man kann aber auch explizit ein „scrollable“ und/oder „updatable“ ResultSet definieren.

ResultSet-Verarbeitung

Beispiel

```
ResultSet rs = stmt.executeQuery("SELECT ...");
while(rs.next()){
  // Zugriff mittels Spaltenposition
  String s = rs.getString(1); // Name
  int i = rs.getInt(2); // PersNr

  // oder: Zugriff mittels Spaltenname
  float f = rs.getFloat("Gehalt");
  double do = rs.getDouble("Gehalt");
  Date da = rs.getDate("Geboren");
  Timestamp t = rs.getTimestamp("Geboren");
}
```

ResultSet-Verarbeitung

- Zu jedem ResultSet kann man Metadaten abfragen:
  `ResultSetMetaData rsmd = rs.getMetaData();`
- Diese Metadaten enthalten Informationen über die Anzahl der Spalten einer Zeile sowie über die einzelnen Spalten (wie Name, JDBC-Typangabe, DB-spezifische Typbezeichnung, ...)

Beispiel

```
int anzahl = rsmd.getColumnCount();
for (int i = 1; i <= anzahl; i++) {
  columnName = rsmd.getColumnName(i);
  dbSpecificType = rsmd.getColumnTypeName(i);
  columnTypeInt = rsmd.getColumnType(i);
  ...
}
```

SQL- vs. Java-Typen

Generische Typ-Bezeichnungen für SQL-Typen:

- Problem: Die verschiedenen DB-Systeme bieten zwar SQL-Typen mit ähnlicher Funktionalität an; diese Typen haben aber eventuell unterschiedliche Namen, z.B.: LONG RAW, IMAGE, BYTE, LONGBINARY sind im wesentlichen gleich.
- Für Portabilität wäre es wichtig, generische Bezeichnungen für die SQL-Typen zu verwenden, z.B. als return-Wert von `getcolumnType()` (in `ResultMetaData`)
- Lösung in JDBC: Die Klasse `Types` (in java.sql) bietet eine Reihe von static int Definitionen für die verschiedenen SQL-Typen, z.B. LONGVARBINARY für die SQL-Typen LONG RAW, IMAGE, etc.

SQL- vs. Java-Typen

Konvertierung von Werten zw. Java-Typen und SQL-Typen

- Problem: Selbst wenn man für die SQL-Typen einheitliche Bezeichner definiert, muss man erst festlegen, wie man Werte solcher SQL-Typen im Java-Programm speichert.
- Enfachste Lösung: Solange man z.B. bloß die Werte am Bildschirm ausgeben will, kann man alle SQL-Werte als Strings auffassen. Für die Weiterverarbeitung im Java-Programm ist das aber üblicherweise ungeeignet.
- Bessere Lösung: Type Mapping zwischen den generischen JDBC-Types und tatsächlichen Java-Typen, z.b.: siehe nächste Folie.

SQL- vs. Java-Typen

JDBC Type	**Java Type**
CHAR, VARCHAR, LONGVARCHAR	String
NUMERIC, DECIMAL	java.math.BigDecimal
BIT, BOOLEAN	boolean
TINYINT, SMALLINT, INTEGER, BIGINT	byte, short, int, long
REAL	float
FLOAT, DOUBLE	double
BINARY, VARBINARY, LONGVARBINARY	byte[]
DATE	java.sql.Date
TIME	java.sql.Time
TIMESTAMP	java.sql.Timestamp

Scrollable/Updatable ResultSet

- Standardverhalten von ResultSets:
  - Read-only
  - nur Vorwärtsbewegung möglich (mittels `rs.next()`)
- Erweiterte Möglichkeiten bei ResultSets:
  - „Type“: forward_only vs. scrollable
  - „Concurrency Mode“: read_only vs. updatable
- Das ResultSet Verhalten wird beim `createStatement()` Aufruf festgelegt:
  - Standardverhalten: `conn.createStatement();`
  - erweitert, z.B.:
    ```
    conn.createStatement(ResultSet.CONCUR_UPDATABLE,
    ResultSet.TYPE_SCROLL_INSENSITIVE);
    ```

Navigation im Scrollable ResultSet

Folgende Methoden stehen zur Verfügung:

- `first()`
- `last()`
- `next()`
- `previous()`
- `beforeFirst()`
- `afterLast()`
- `absolute(x)` Sprung zur Zeile mit Zeilennummer x
- `relative(x)` Bewegung nach vorwärts um x Zeilen (bei $x < 0$ erfolgt entsprechende Rückwärtsbewegung)

Schreibzugriff auf ResultSet

- UPDATE:
  - Ändern von einzelnen Spaltenwerten der aktuellen Zeile mittels Typ-spezifischer update-Methode
  - Festschreiben der geänderten, aktuellen Zeile mittels `updateRow()`-Methode
- DELETE: mittels `deleteRow()`-Methode

Beispiel

```
rs = stmt.executeQuery("SELECT MatrNr, VorlNr,
                                    ... FROM pruefen");
while(rs.next()) {
  l_Note = rs.getFloat("Note");
  if (l_Note == 5)
    rs.deleteRow();
  else if (l_Note >= 2) {
    rs.updateFloat("Note", l_Note - 1);
    rs.updateRow();
} }
```

Schreibzugriff auf ResultSet

- Warnung: Erzeugung eines updatable ResultSet funktioniert nur wenn die Query nur eine Tabelle und von dieser Tabelle alle Primary Keys selektiert.
- INSERT:
  - Es gibt eine eigene „insert row“.
  - Das Einfügen einer neuen Zeile geschieht durch Wertzuweisungen an die Spalten der „insert row“.
  - Festschreiben der neuen Zeile: mittels `insertRow()`-Methode

Beispiel

```
rs.moveToInsertRow();
rs.updateFloat("Note", l_Note - 1);
rs.updateInt("MatrNr", l_MatrNr);
rs.updateInt("PersNr", neuerPruefer);
rs.updateInt("VorlNr", l_VorlNr);
rs.insertRow();
rs.moveToCurrentRow();
```

Gliederung

Prepared Statements

- Ziel: die gleiche Art von SQL-Statement jeweils mit teilweise unterschiedlichen Werten mehrmals ausführen, z.B.:

  ```
  SELECT * FROM Studenten WHERE SEMESTER = ?
  ```

- Lösung in JDBC:
  - Mittels „prepared statement“ ein SQL-Statement mit *bind variables* definieren.
  - Vor jedem execute-Aufruf werden diese bind variables mittels Typ-spezifischen Set-Methoden neu zugewiesen.
  - Vorteil: Das Datenbanksystem muss manche Arbeiten nur ein Mal durchführen (z.B. Parsen, Anfrageoptimierung).

Prepared Statements

Beispiel

```
PreparedStatement pstmt = conn.prepareStatement
  ("SELECT * FROM Studenten WHERE Semester = ? ");
for (int j = 1; j <= 20; j++ ) {
  pstmt.setInt (1, j);
  rs = pstmt.executeQuery();
  System.out.println("Studenten im " + j +
    "-ten Semester: ");
  while(rs.next()){
    System.out.print(rs.getString(1) + ", " );
    System.out.print(rs.getString(2) + ", " );
    System.out.println(rs.getString(3));
  }
}
```

Callable Statements

- Ziel: Bei Aufruf einer stored procedure mit Output-Parametern werden Werte zurückgeliefert. Diese müssen in Java-Variablen zugänglich sein.
- Lösung in JDBC:
  - Mittels „callable statement“ ein SQL-Statement mit bind variables definieren.
  - Der Typ jeder Output-Variable (bzw. des Return-Wertes einer stored procedure) muss „registriert“ werden.
  - Vor jedem execute-Aufruf werden die Input-Variablen mittels Typ-spezifischen Set-Methoden zugewiesen.
  - Nach jedem execute-Aufruf werden die Output-Variablen mittels Typ-spezifischen Get-Methoden ausgelesen.

Callable Statements

Beispiel

```
CallableStatement cs =
  con.prepareCall("{call accountlogin(?,?,?)}");
cs.setString(1,theuser);
cs.setString(2,password);
cs.registerOutParameter(3,Types.DATE);

cs.executeQuery();
Date lastLogin = cs.getDate(3);
```

Escape Syntax

- Problem: Die konkrete Syntax für manche SQL-Features sind in besonderer Weise DB-Hersteller spezifisch, z.B.: Aufruf von stored procedures, built-in Funktionen, Date/Time-Format, etc.
- Ziel: trotzdem portablen Java-Code erstellen
- Lösung in JDBC:
  - Mittels { } wird dem JDBC-Treiber mitgeteilt, dass er die (standardisierte) escape syntax in die DB-spezifische Syntax umwandeln soll.
  - Schreibweise: `{<keyword ...}`, wobei `<keyword>` z.B. folgende Werte haben kann:
    - call (für Aufruf einer stored procedure/function)
    - fn (für Aufruf einer built-in Funktion)
    - d bzw. t (für date bzw. Time-Angabe)

Beispiel

```
conn.prepareCall("{ call p_copy_student (?,?) }");
stmt.executeUpdate("Update employees" +
  "SET born_at = {d '1954-12-26'}, " +
  " income = {fn ceiling(12345.67)} WHERE id = 1005");
```