Kapitola 1. Přednáška 3 - zajištění perzistence a správa dat v javových aplikacích, JBDC, JDO

Obsah

Java Database Connectivity (JDBC)

Co je JDBC

- JDBC je univerzální rozhraní pro přístup k (převážně relačním) databázím.
- Umožňuje dotazovat se pomocí SQL i modifikovat pomocí SQL data.
- JDBC odpovídá specifikaci X/Open SQL CLI.
- Je k dispozici na J2SE a J2EE v balících java.sql, javax.sql.
- Současná verze, JDBC 3.0, plně reflektuje finální verzi SQL99.

Postup práce s JDBC

Postup práce s JDBC je následující:

1. Výrobce příslušného systému řízení báze dat (DBMS) poskytne implementaci JDBC
2. Programátor používá JDBC k přístupu k datům spravovaným určitým DBMS - do značné míry nezávisle na konkrétním DBMS, provádí SQL příkazy...
3. Přesná podoba (odchylky) nicméně závisí na konkrétním DBMS.

Koncepce je podobná jako ODBC (Microsoft Windows).

Hlavní třídy (komponenty) JDBC

`DriverManager`	manažer databázových ovladačů - každý systém řízení báze dat (DBMS) musí mít vlastní databázový ovladač (`Driver`). V manažeru se ovladače zaregistrují a přístup k danému DBMS se realizuje prostřednictvím příslušného ovladače.
`DataSource`	abstraktní "obálka" konkrétního datového zdroje. Uživatele nezajímá, kterým DBMS je zdroj řízen, stačí, že nějakou službou získá objekt DataSource a z něj vytváří spojení (`Connection`).
`ConnectionPoolDataSource`	dtto, ale _oolable_ tj. datový zdroj schopný poolingu připojení - zvýšená efektivita, není třeba vždy fyzicky zřizovat nové spojení, ale "vypůjčit si" právě nepoužívané, "volné" spojení z poolu.

Hlavní třídy (komponenty) JDBC (2)

`Connection`	spojení k datovému zdroji. Nad spojením se spouštějí SQL příkazy (dotazy), spojení může být předmětem transakcí.
`Statement`	obálka jednoho SQL příkazu.
`ResultSet`	obálka výsledku SQL příkazu. U dotazů (čtecích) obsahuje typicky 1 až __řádků s nedefinovaným pořadím - odtud označení ResultSet.
`DatabaseMetaData`	zachycuje metadata (schéma) dané databáze

Příklad - zavedení ovladače DBMS

```
import java.sql.*;
public class TestJDBCMySQL {
   public static void main(String[] args) {
      String url = "jdbc:mysql://idealab.cs.uiowa.edu/WorksOn";
      String uid = "user";
      String pw = "testpw";
      try { // Load driver class
         Class.forName("com.mysql.jdbc.Driver");
      } catch (java.lang.ClassNotFoundException e) {
         System.err.println("ClassNotFoundException: " +e);
      }
      ...
```

Příklad - vytvoření spojení, příkazu a jeho spuštění

```
   ...
   Connection con = null;
   try {
      con = DriverManager.getConnection(url, uid, pw);
      Statement stmt = con.createStatement();
      ResultSet rst = stmt.executeQuery("SELECT ename,salary FROM Emp");
      System.out.println("Employee Name,Salary");
      while (rst.next()) {
         System.out.println(rst.getString("ename")+","+rst.getDouble("salary"));
      }
      con.close();
   } catch (SQLException ex) {
      System.err.println("SQLException: " + ex); }
   }
}
```

Charakteristika JDBC rozhraní

JDBC je velmi obecné rozhraní pro přístup k relačním databázím. Proto:

- je "slabě typované", nemá např. speciální metody pro jednotlivé typy SQL dotazů, vše se spouští executeQuery.
- totéž platí pro přístup k výsledkům dotazů: ResultSet je jednoduchý iterátor (přesněji kurzor), kde lze z aktuální pozice číst a případně na ní modifikovat.
- vše podstatné je ponecháno na uživateli - formulaci SQL dotazů uvnitř execute, executeQuery apod. - rozhraní nezkontroluje ani syntaktickou správnost, to až samotný DB stroj.
- taktéž výjimky jsou téměř vždy pouze SQLException s textovým popisem uvnitř

JDBC rozhraní - balíky

JDBC je rozloženo do balíků:

java.sql	základní rozhraní a třídy
javax.sql	rozšiřující rozhraní a třídy
třídy a rozhraní konkrétního DBMS	závisejí na použitém DBMS, např. balík org.hsqldb

Práce s databází

Klasickým způsobem práce s JDBC je tento postup:

- získáme spojení typicky oslovením `DriverManager`u s uvedením url zdroje, jména a hesla uživatele:

  con = DriverManager.getConnection(url, uid, pw);

- na získaném spojení vytváříme příkazy (`Statement`):

  Statement stmt = con.createStatement();

- na příkazu spustíme SQL dotaz:

  ResultSet rst = stmt.executeQuery("SELECT ename,salary FROM Emp");

- výsledky dotazu projdeme iterací výsledného `ResultSet`u:

```
while (rst.next()) {
         System.out.println(rst.getString("ename")+","+rst.getDouble("salary"));
```

```
}
```

- spojení "po použití" uzavřeme.

Varování

Nesmíme uzavřít, dokud nedočteme ResultSet - jeho zbytek by se ztratil.

Poznámka

Také se obecně nelze v ResultSetu vracet.

Poznámka

Je nutné počítat s tím, že ResultSet neexistuje v jednu chvíli v paměti celý; je to virtuální přístupový bod k výsledkům, které se obecně postupně načítají/generují.

Transakce

Implicitně se pro zřízené spojení nastaví režim "auto-commit", každý dotaz je uzavřen do samostatné transakce.

Chceme-li toto změnit (což je vzhledem k výkonu i vhodné), nastavíme **con.setAutoCommit(false)** a řídíme si transakce sami.

Řízení transakcí provádíme na daném spojení pomocí:

commit()	potvrdí a trvale uloží změny provedené od posledního commit/rollback
rollback()	"zahodí" změny provedené od posledního commit/rollback, tj. vrátí databázi do stavu po posledním commit/rollback
rollback(Savepoint p)	vrátí databázi do stavu p

Transakce - příklad

Příklad kódu s řízením transakce: první změna je potvrzena (commit), druhá zamítnuta (rollback):

```
con.setAutoCommit(false);

// inserts first two messages
for (int i = 0; i < messages.length-1; i++) {
```

```
   stmt.executeUpdate(
          "INSERT INTO MESSAGES VALUES (" ... ")");

con.commit();

// inserts last message ?
stmt.executeUpdate(
          "INSERT INTO MESSAGES VALUES (" ... ")");
// no. last message will not be inserted!
con.rollback();
```

Demo - HSQLDB

Na rychlé a nenáročné vyzkoušení databázových věcí v Javě můžeme použít volně dostupný javový systém řízení báze dat HSQLDB (dříve HypersonicSQL), http://hsqldb.sf.net.

Všechny demoprogramy dostupné ve zdrojové podobě jsou bez úprav učené pro HSQLDB.

HSQLDB - režimy práce

HSQLDB může pracovat ve třech režimech:

in-memory/in-proces	běží v rámci procesu klientského programu, není nutné spouštět zvlášť, pouze v programu přistoupíme k databázi pomocí zvláštního URL - viz manuál HSQLDB nebo příklady
standalone	DB server běží zvlášť, tudíž je přístupný více klienty, tabulky se ukládají na disk
webserver	dtto, přístupné přes HTTP přes mini HTTP-server

Předpřipravené příkazy

Předpřipravené příkazy (Prepared Statements) jsou prostředkem, jak efektivně provádět často opakované SQL dotazy, např.:

- více vložení do stejné tabulky
- dotazy, kde mění jen jeden parametr, např. hledaný klíč

Technicky předpřipravený dotaz vypadá tak, že na místě dosazovaného parametru je znak ?.

Pomocí set-metod se daný formální parametr ("otazník") nahradí skutečnou hodnotou a pak se příkaz spustí.

Předpřipravené příkazy - příklad

Příklad:

```
// příprava příkazu
PreparedStatement pstmt = con.prepareStatement(
        "SELECT E.FIRSTNAME, E.SURNAME, M.CREATED, M.TEXT " +
        "FROM EMPLOYEES AS E, MESSAGES AS M " +
        "WHERE E.SURNAME = ?"
        +" AND E.ID = M.TO " +
        "ORDER BY M.CREATED DESC");

// naplnění parametrů
pstmt.setString(1, "Novak");

// vlastní provedení
ResultSet result = pstmt.executeQuery();
```

Uložené procedury

Uložené procedury - Stored Procedures - jsou vzdáleně podobné předpřipraveným příkazům, jsou však zcela v server-side režii.

Běží tedy vzdáleně přímo na serveru.

Jsou výhodné nejen z hlediska výkonu, ale i údržby - jsou centralizovaně uložené, lze je lépe měnit při změnách datového modelu, konfigurace DB atd.

Modifikovatelné výsledky dotazu

Podporuje-li to daný DBMS, můžeme řádky vrácených tabulek modifikovat:

```
ResultSet result = stmt.executeQuery(
   "SELECT TEXT " +
   "FROM MESSAGES " +
   "WHERE ID > 10");

// posuň se na pátý řádek výsledku
result.absolute(5);

// změň hodnotu atributu/pole
result.updateString("TEXT", "Zmeneny text zpravy");

// proveď změny
result.updateRow();
```

Metadata

DBMS poskytne základní informace o

- svých schopnostech (dostupných funkcích) a
- údaje o příslušně databázi (metadata)

voláními, jako jsou:

```
DatabaseMetaData dbmd = con.getMetaData();

// základní údaje o DBMS
System.out.println(
   "DBMS: " +
   dbmd.getDatabaseProductName()  + ", " +
   dbmd.getDatabaseProductVersion() );

// údaje o driveru
System.out.println(
   "Driver: " +
   dbmd.getDriverName()  + ", " +
   dbmd.getDriverVersion() );

// dostupné funkce DBMS
   System.out.println("String functions: "+dbmd.getStringFunctions());
   System.out.println("TimeDate functions: "+dbmd.getTimeDateFunctions());
   System.out.println("Numeric functions: "+dbmd.getNumericFunctions());
   System.out.println("System functions: "+dbmd.getSystemFunctions());
```

Zpřístupnění datového zdroje přes JNDI

Moderní přístup k datovým zdrojům je založen na přidání další úrovně adresovací abstrakce:

- místo URL databáze, jména a hesla se
- prostřednictvím jmenné a adresářové služby (JNDI) najde datový zdroj - aplikační programátor ani předem neví, kde je zdroj fyzicky uložen, jaký DBMS se o něj stará
- datový zdroj se zpřístupní nikoli jako Connection, ale ponovu jako `javax.sql.DataSource`
- teprve z něj se získá spojení.

Poznámka

JNDI služba je obvykle ve správě aplikačního (webového) serveru (Tomcat, jboss...)

Java Data Objects (JDO)

JDO - charakteristika

JDO je Java-centrická alternativa k tradičním způsobům zajištění perzistence dat:

- *JDBC* (přístup z Javy k relačním datům)
- *serializace* (základní technika převodu objektů - potenciálně libovolných - do binární, uložitelné, podoby)

JDO - forma

JDO je specifikace, která prochází tzv. *Java Community Process* (JCP) *Program*, blíže viz http://jcp.org.

Mají na ni tedy vliv i nezávislí vývojáři.

Její aktuální platná verze je 1.0.1, připravuje se 2.0.

Specifikace JDO 1.0.1 [http://javashoplm.sun.com/ECom/docs/Welcome.jsp?StoreId=22&PartDetailId=7734-jdo-1.0.1-mr-spec-oth-JSpec&SiteId=JCP&TransactionId=noreg] má 200 stran.

JDO - motivace

Motivací pro JDO bylo dát pohodlnější techniku (než JDBC a serializace) pro zachycení stavu běžných javových objektů v programu a možnost jeho obnovy.

JDO - struktura

JDO - implementace

Existuje několik open-source i komerčních implementací JDO:

Solametric Kodo JDO — komerční implementace JDO, dostupná na http://www.solarmetric.com/

Popis: Kodo JDO is SolarMetric's robust, high-performing, feature-rich implementation of the Java™ Data Objects specification for transparent persistence. Unlike many proprietary object/relational mapping tools, Kodo JDO provides access to relational databases through the JDO standard, enabling Java developers to use existing relational database technology from Java without needing to know SQL or be an expert in relational database design. It can be used with existing database schemas, or can automatically generate its own schema.

Castor JDO	open-source, free (BSD-like licence), dostupný na http://www.castor.org/jdo.html
TriActive JDO (TJDO)	open-source, free implementace, dostupná na http://tjdo.sourceforge.net/. Charakteristika: Supports JDO 1.0.1. Implements the entire JDOQL query language, including several useful method enhancements. Auto-creates all necessary schema elements (tables, foreign keys, indexes) according to your app's classes and JDO metadata. Auto-validates expected schema structure at runtime, reducing failures from schema evolution. Can map Java classes to SQL views, and allows for direct SQL queries, to leverage SQL capabilities not normally accessible through JDO. Designed to be lightweight and fast.

JDO - informační zdroje

- Java Data Objects [http://access1.sun.com/jdo/] - informace na stránkách Sun
- JDO Central.com [http://www.jdocentral.com/] - hlavní portál o JDO

JDO - alternativy

Alternativou pro pohodlnou serializaci mnoha typů objektů do XML je např. open-source balík XStream [http://xstream.codehaus.org/], viz dále.

Serializace dat do XML

Serializace obecně - připomenutí

Javová třída může implementovat rozhraní `java.io.Serializable`, které

- nepředepisuje žádné metody, ale zajistí, že
- objekty dané třídy lze serializovat, tj. převést jejich obsah (proměnné) na binární (vnější) podobu zapsatelnou např. do souboru
- inverzně lze zase objekty z této binární podoby deserializovat

Serializace

Při serializaci se standardně

- převádějí do binární podoby všechny instanční proměnné

- vyjma těch označených klíčovým slovem transient

Tak lze zajistit serializaci skutečně jen potřebných složek stavu objektu, zbytek se po deserializaci může "dopočítat".

Poznámka

Typický příklad: symetrickou matici (např. podle hlavní diagonály) neukládáme celou, ale jen "jednu polovinu".

Serializace - metody

Požadujeme-li speciální chování (např. kvůli serializaci návazných/odkazovaných objektů), můžeme v objektu poskytnout metody:

private void writeObject(java.io.ObjectOutputStream out) throws IOException	realizuje zápis objektu do výstupního proudu - metodu si napíšeme sami
private void readObject(java.io.ObjectInputStream in) throws IOException, ClassNotFoundException;	realizuje čtení objektu ze vstupního proudu - metodu si napíšeme sami Serializační mechanizmus zajistí vytvoření "prázdného" objektu, ten pak naplníme metodou `readObject`.

Serializace do XML - vazba XML na Javu

Jednoduché nástroje serializace - XStream

Potřebujeme-li serializaci javových objektů jen na prosté účely typu

- uložení a opětovné načtení konfigurace do/ze XML souboru
- přenesení objektů jinému procesu, programu, na jiný stroj

pak je XStream pravděpodobně nejlepší volbou, pokud nám nevadí nepodpora non-ASCII znaků...

Jak funguje XStream

XStream je extrémně jednoduše a přímočaře použitelné API pro serializaci jakýchkoli objektů do XML.

Není třeba psát žádné popisovače, stačí vybrat objekt a poslat ho metodě `toXML` objektu serializátoru `org.codehaus.xstream.XStream`.

XStream - příklad, krok 1

Vytvoříme pole se zadaným počtem prvků, postupně do něj vřadíme nově vytvořené objekty Person a toto pole serializujeme.

```
public static int LEN = 10000;
public static void main(String[] args) {
   Person[] people = new Person[LEN];
   XStream xs = new XStream();
   xs.alias("person", Person.class);
   for(int i = 0; i < LEN; i++) {
      people[i] = new Person("Clovek "+i, i, i * 10);
   }
   // serialize
   String xml = xs.toXML(people);
   System.out.println(xml);

// uncomment the following code to test deserialization
/*
   Person[] secondPeople = (Person[])xs.fromXML(xml);
   for(int i = 0; i < secondPeople.length; i++) {
      increaseSalary(secondPeople[i], 20);
   }
*/
}
```

XStream - příklad, krok 2

Vytvořený soubor vypadá takto:

```
<person-array>
  <person>
    <name>Clovek 0</name>
    <age>0</age>
    <salary>0.0</salary>
  </person>
  <person>
    <name>Clovek 1</name>
    <age>1</age>
    <salary>10.0</salary>
  </person>
  <person>
    <name>Clovek 2</name>
    <age>2</age>
    <salary>20.0</salary>
  </person>
  ...
```

Hibernate

Hibernate

Hibernate