université
PARIS-SACLAY

Bases de données
JDBC – Introduction, Ordres sans parametre

Silviu Maniu

2022/2023

Polytech App3

Le mode programme

Mode programme : accès à la BD par une application généraliste

- programme : gère interface, calcul, réseau
- ordres BD : traitement données

Problèmes mode programme :

- connexion : comment se connecter à une BD
- interface : ou placer les ordres BD, comment afficher/utiliser les résultats

Contenu

JDBC – mode programme Java

API Java – ensemble de classes et interfaces (« *package* ») Java

JDBC – Java DataBase Connectivity

- permet d'acceder a des SGBD par SQL
- **independant d'un SGBD** : « *drivers* » pour chaque SGBD
- **independant d'une architecture** : Java est portable

JDBC – exemple simple

Situation : incrementer l'age pour une anniversaire

- appeler l'ordre suivant en mode programme (Java) :

```
update personne
  set age = age+1
  where nom = 'Camille'
```

JDBC – exemple simple

```
import java.io.*;
import java.sql.*;

class ExJDBC {
  public static void main(String[] args)
   throws SQLException, ClassNotFoundException {
        String url = "jdbc:postgresql://tp-postgres/";
        Properties prop = new Properties();
        prop.setProperty("user","smaniu_a");
        prop.setProperty("password","smaniu_a");
    Connection c = DriverManager.getConnection(url, prop);
    String texte = "update personne set age=age+1 where nom='Camille'";
    Statement s = c.createStatement();
    int n;
    n = s.executeUpdate(texte);
    System.out.println("nombre de lignes traitees : "+n);
  }
}
```

JDBC – exemple simple

Execution (à partir d'un fichier `ExJDBC.java`) :

1. compilation : `javac ExJDBC.java` (suppose driver dans le CLASSPATH)
2. execution : `java ExJDBC`
3. sortie : « `nombre de lignes traitees : 1` »

Interface JDBC

Principe – Java appelle des fonctions d'une bibliothèque, qui encapsulent les accès et ordres au serveur BD

- des fonctions de la bibliothèque
- chaines de caractères (ordres SQL, identifiants de connexion)

Interface JDBC – Driver

Java recherche le driver JDBC correspondant au SGBD dans son `CLASSPATH`.

- possible d'avoir plusieurs driver dans même programme – interaction avec des différents serveurs (Oracle, MySQL, PostgreSQL, etc.)

Interface JDBC – Connexion

```
Connection c = DriverManager.getConnection(url, prop)
```

Fonction `getConnection` :

- récupère driver indiqué par `url` et les paramètres indiqués dans `prop` de type `Properties` (utilisateur, mot de passe, utilisation SSL, etc.)
- établit connexion avec serveur BD
- renvoie objet `Connection` qui est nécessaire pour toute interaction avec la BD

Interface JDBC – Connexion

Format `url` pour PostgreSQL (type `String`) :

`jdbc:postgresql://host:port/database`

- `host` : l'adresse du serveur, par defaut *localhost*
- `port` : port du serveur, par defaut *5432*
- `database` : le nom de la BD, par defaut meme que l'utilisateur

La classe Properties permet de remplir d'autres champs de connexion, par exemple :

- `user`, `password`, `ssl`

Interface JDBC – Connexion

Session : connexion a un serveur, séquence d'ordres, déconnexion

- plusieurs sessions possible dans une application JDBC

Objet `Connection` :

- définit une session
- utilise méthodes pour envoi ordres BD, gestion des transactions
- accès aux informations sur tables, procedures, etc.
- création seulement possible par `getConnection`

Interface JDBC – Déconnexion

Methode `void close() throws SQLException` de `Connection` :

- effectue déconnexion normale du serveur qui termine la session
- utile pour déconnexion immédiate
- note : déconnexion automatique quand l'objet `Connexion` est libéré par le « garbage collector »

JDBC - Principe général

1. création objet ordre générique (`Statement`)
2. envoi de l'ordre au serveur par les méthodes fournies par l'objet
3. si ordre SQL `select` : récupération des résultats (curseur)
4. permet gestion erreurs

Types :

- ordres SQL avec ou sans paramètres
- appels procédure stockée PL/pgSQL (ou autre)

Déroulement d'un programme

1. **import** classes JDBC
2. **lancement** et début de l'exécution – inconnu du serveur BD
3. **chargement** driver
4. **connexion** par JDBC au serveur SGBD – programme devient **client**
5. **envoi** des ordres BD par JDBC
6. **deconnexion** – SGBD ne connait pas la suite d'exécution
7. suite exécution programme

Contenu

JDBC - Ordres sans paramètre

Ordre sans paramètre : tout ordre qui n'utilise pas de variables pour l'exécution

- `select` (curseurs)
- les autres : `update`,`delete`,...

Objet `Statement`, crée par méthode de `Connection`

- `Statement createStatement() throws SQLException`
- en général, un seul object `Statement`

JDBC - Ordres non-SELECT sans paramètre

Envoi ordre au serveur par la méthode de `Statement` :

- `int executeUpdate(String sql) throws SQLException`

Retour :

- nombre de lignes traitées
- 0 si aucune ligne traitée

Contenu

Gestion erreurs BD

Un ordre BD peut générer une **erreur**

Gestion JDBC :

- la fonction JDBC qui demande l'ordre **lève une exception** type `SQLException`
- si pas gérée dans la méthode Java correspondante, il faut ajouter `throws SQLException` à la déclaration

Gestion erreurs BD – Utilisation

Gestion « locale » :

```
try {
    // ... appel JDBC
} catch(SQLException e) {
   // utiliser e pour afficher, traiter, etc.
}
```

Sinon :

- **envoi** exception à la méthode « supérieure » (si `throws`)
- si aucun `throws`, **arrêt brutal programme**

Dans tous le cas : **annulation transaction BD**

Contenu

Curseurs – rappel

Ordres `SELECT` – curseurs :

1. déclaration curseur
2. remplissage une seule fois a l'exécution
3. récupération lignes une par une

Même principe en JDBC

Exemple – premier résultat

```
Statement s = c.createStatement();
ResultSet rset =
  s.executeQuery("select dest, jour from train where "+
   "client = 'Julie' order by dest");
rset.next();
System.out.println("Julie va à "+rset.getString(1)+
   " le jour "+rset.getInt("JOUR"));
r.close();
s.close();
```

Curseur JDBC

Création : `Statement` (sans paramètre) ou `PreparedStatement` (avec paramètre)

Remplissage : `executeQuery`

- renvoie objet `ResultSet` – la **zone curseur**

Avancement : méthode de la classe `ResultSet`

- `Boolean next() throws SQLException`
- `true` si une nouvelle ligne trouvée

Curseur JDBC

Recuperation ligne – colonne par colonne, méthode de `ResultSet`

- `TYPE getTYPE(int indiceColonne) throws SQLException`
- `TYPE getTYPE(String nomColonne) throws SQLException`
- `TYPE` : tout type primitif Java (`Int`,`String`,...)
- **exemple** : l'entier en première colonne `.getInt(1)`

Liberation – méthode de `ResultSet`

- `void close() throws SQLException`

Exemple – premier résultat et deuxième si existant

```
Statement s = c.createStatement();
ResultSet rset =
  s.executeQuery("select dest, jour from train where "+
    "client = 'Julie' order by dest");
rset.next();
System.out.println("Julie va à "+rset.getString(1)+
  " le jour "+rset.getInt("JOUR"));
if(rset.next())
  System.out.println("Julie va à "+rset.getString(1)+
    " le jour "+rset.getInt("JOUR"));
r.close();
s.close();
```

Exemple – toutes les lignes

```
Statement s = c.createStatement();
ResultSet rset =
 s.executeQuery("select dest, jour from train where "+
  "client = 'Julie' order by dest");
while(rset.next())
  System.out.println("Julie va à "+rset.getString(1)+
   " le jour "+rset.getInt("JOUR"));
r.close();
s.close();
```

Contenu

Ordres SQL avec paramètre

Tout ordre : `insert`, `update`, `delete`, `select`

Deroulement :

- création objet `PreparedStatement` (ordre fixe avec des parts variables)
- pre-compilation
- affectation valeurs aux paramètres
- envoi au serveur

Exemple

« Anniversaire » – `nom` paramétré

```
PreparedStatement stmt = conn.prepareStatement(
  "update personne set age = age+1 where nom = ?");
stmt.setString(1,"Jeanne");
stmt.executeUpdate();
```

PreparedStatement

Création – méthode de `Connection` :

- `PreparedStatement prepareStatement(String texteSql) throws SQLException`
- dans `texteSql`, un symbole « `?` » pour chaque paramètre
- paramètres indexés par position de **gauge à droite**

PreparedStatement

Affectation paramètres :

- methode `setTYPE` de `PreparedStatement`
- `TYPE` type primitif Java (comme pour les curseurs)

```
void setString(int indiceParametre, String x) throws
SQLException
```

- `x` converti en `VARCHAR` par le SGBD

PreparedStatement

Execution :

- `executeUpdate` si ordre `insert`, `update`, `delete`
- `executeQuery` si ordre `select`

Note : `PreparedStatement` utilisable même si aucun paramètre, mais inutile

PreparedStatement vs. Statement

Situation – plusieurs « anniversaires »

Avec `Statement`, compilation et exécution à chaque fois :

```
String[] p = {"Jeanne", "Jules"};
Statement stmt = conn.createStatement();
for (int i=0; i<p.length; i++)
  stmt.executeUpdate(
   "update personne set age = age+1 where nom = '"
   + p[i] + "'");
```

PreparedStatement vs. Statement

Situation – plusieurs « anniversaires »

Avec `PreparedStatement`, une seule (pré-)compilation, plusieurs exécutions :

```
String[] p = {"Jeanne", "Jules"};
PreparedStatement stmt = conn.prepareStatement(
 "update personne set age = age+1 where nom = ?");
for (int i=0; i<p.length; i++) {
  stmt.setString(1, p[i]);
  stmt.executeUpdate();
}
```

Contenu

Appel procédure stockée

Situation – nous avons une procédure stockée `anniversaire(n varchar)`

- fonctionnalité : incrémente de 1 l'age de la ligne correspondante au nom `n`

Appel direct de la procédure en JDBC :

```
String n = "Jeanne";
CallableStatement stmt = conn.prepareCall(
                                "{call anniversaire(?)}");
stmt.setString(1, "Jeanne"); // comme PreparedStatement
stmt.executeUpdate(); // comme PreparedStatement
```

Note : pour pouvoir appeler des procedures en JDBC PostgreSQL, il faut mettre la propriété de connexion `escapeSyntaxCallMode` à `callIfNoReturn` !

Appel fonction stockée

`.registerOutParameters` – declarer le type d'un paramètre (`java.sql.Types`)

- lecture de paramètre `OUT` d'une procédure stockée
- lecture valeur renvoyée par une fonction stockée

Supposons une fonction `age` :

```
CallableStatement stmt = conn.prepareCall(
                                "{? = call age(?)}");
stmt.setString(2, "Jeanne");
stmt.registerOutParameter(1, Types.INTEGER);
stmt.execute();
System.out.println(stmt.getInt(1));
```

Récapitulatif

Ordres SQL sans parametres, y compris `select` : `Statement`

Ordres SQL avec parametres, y compris `select` : `PreparedStatement`

Appel procedure stockée, avec ou sans paramètres :
`CallableStatement`

Remerciements

Site JDBC pour PostgreSQL – téléchargement driver, documentation, syntaxe :

- `https://jdbc.postgresql.org/`

Le contenu de ce cours est grandement inspiré des cours d'Emmanuel Waller

- `https://www.lri.fr/~waller/`