Application BdD et JDBC

Anne-Cécile Caron

Licence MIAGE - Bases de Données

2016-2017

Application base de données

Pour le développeur :

- Quel est l'environnement ?
  - type de client (mode client serveur, intranet, ...)
  - langage utilisé
  - contraintes techniques (machines, OS, logiciels ...)
- Dans tout les cas, une seule problématique
  - intégrer du SQL dans un langage de haut niveau
  - gérer des problèmes de BD + IHM + réseau + ...
- Quelques alternatives :
  - Solution propriétaire, liée à un éditeur.
  - SQL intégré ou comment intégrer du SQL dans un langage *connu*. (norme SQL2)
  - utilisation d'API

JDBC

- API Java = paquetages `java.sql` ("core") et `javax.sql` ("option")
- Adopté par presque tous les constructeurs
- Plusieurs types de Pilotes en fonction des différents types de communication entre le SGBD et le programme java.
- Naissance en 1997, Version 4.0 en 2006 (suivie par Oracle 11g), Version 4.1 pour java 7 depuis 2011, Version 4.2 pour java 8.

Ce qui est vu ou non dans ce cours

- Nous verrons comment se connecter à une base, poser des requêtes, traiter le résultat.
- Nous ne verrons pas les classes liées à une utilisation dans un environnement J2EE
- Nous ne verrons pas les spécificités de la gestion des transactions : le mode par défaut implique une validation (`commit`) de chaque instruction envoyée au SGBD. On parle de mode `auto-commit`.

attendre le master pour approfondir le sujet ...

Utilisation de JDBC : les étapes

- Charger le pilote (Driver)
- Etablir la connexion avec la base de données
- Créer une zone de description de requête (Statement)
- exécuter la requête
- traiter les données retournées (ResultSet)
- fermer les différents espaces utilisés

Chargement du driver

- Disposer d'un driver propriétaire
- Charger le driver :

```
DriverManager.registerDriver(unObjetDriver);

DriverManager.registerDriver(
    new oracle.jdbc.driver.OracleDriver()
    );
```

- Avec JDBC 4.0, utilisation de Java SE Service Provider : à la connexion à la base, `DriverManager` charge tous les drivers JDBC4.0 qui sont présents dans le `CLASSPATH`.

Obtenir une connexion

- Utilisation d'une méthode static de `DriverManager`

```
Connection connect
       = DriverManager.getConnection(urlBase,
                     unNomLogin, unPassword);
```

- Il faut définir dans l'url :
  - le protocole et sous-protocole
  - l'adresse du SGBD et le nom de la base

```
String urlBase
= "jdbc:oracle:thin:@oracle.fil.univ-lille1.fr:1521:filora" ;
String urlBase
= "jdbc:postgresql://maMachine/maBase" ;
```

- Connexion réseau conforme internet
- Pour le modèle 3-tiers, utiliser plutôt `DataSource` comme alternative à `DriverManager`

L'interface `Statement`

- permet de définir les requêtes SQL à envoyer à la base connectée

```
Statement stmt = connect.createStatement();
```

- Permet d'exécuter deux types de requêtes :
  - Les requêtes de modification de la base
  - Les requêtes de consultation de la base
- Toutes les méthodes doivent prendre en compte l'exception `SQLException`

Modification de la base

- `int executeUpdate(String requeteSQL)`
- Valable aussi pour toutes les commandes SQL DDL.
- par exemple :

```
stmt.executeUpdate("create table tester" +
                   "(num  integer, ch text)") ;
stmt.executeUpdate("insert into tester " +
                "values (1,'dupont')") ;
stmt.executeUpdate("insert into tester "+
                "values (2,'durant')") ;
```

- retourne le nombre de lignes créées, modifiées,...

Consultation de la base

- `ResultSet executeQuery(String requeteSQL)`
- Exemple :

```
ResultSet rs1 = stmt.executeQuery(
                    "select * from tester") ;
ResultSet rs2 = stmt.executeQuery(
   "select * from tester where num =" + i ) ;
```

- L'objet résultat de type `ResultSet` peut être parcouru ligne par ligne.

Récupération des données

La classe `ResultSet` dispose des méthodes suivantes :

- `boolean next()` retourne true s'il reste un n-uplet à lire. Le premier appel à `next()` permet de lire la première ligne.
- *Type* `get`*Type*`(int i)` retourne l'objet de type *Type* de la colonne en position *i*
- *Type* `get`*Type*`(String s)` retourne l'objet de type *Type* de la colonne de nom *s*

```
int i = 0;
while (rs.next()) {
  int num = rs.getInt("num");
  String ch = rs.getString("ch");
  System.out.println("ligne " + i + ": "
                + num + ", " + ch);
  i++;
}
```


Préparation des requêtes

- Plus rapide lorsqu'une même requête est exécutée plusieurs fois, même avec des paramètres différents.
- Le SGBD a une version précompilée de la requête.
- JDBC permet de :
  - Préparer une requête (avec paramètres)
  - Passer les paramètres effectifs à une requête paramétrée.

L'interface PreparedStatement

- Création d'une requête préparée :

```
PreparedStatement cmdSQL =
  connect.prepareStatement("select *
  from personne where nom=? and age > ?" );
```

- Paramétrer la requête :

```
cmdSQL.setString(1, "dupont") ;
cmdSQL.setInt(2,17) ;
```

- Exécution de la requête :

```
ResultSet rs = cmdSQL.executeQuery() ;
```


Procédures stockées

- JDBC propose une interface `CallableStatement` qui permet d'appeler des procédures ou fonctions stockées.

```
CallableStatement cs1
= connect.prepareCall( "{call ma_procedure (?,?)}" ) ;
CallableStatement cs2
= connect.prepareCall( "{? = call ma_function (?,?)}" ) ;
```

- Les paramètres en entrée sont gérés comme une requête préparée

```
cs1.setString(1,"aa");
cs2.setInt(2,refCompte);
```

- Invocation de la procédure ou fonction :

```
cs1.executeUpdate();
cs2.registerOutParameter(1,java.sql.Types.DOUBLE);
cs2.execute();
double d = cs2.getDouble(1);
```


Fermeture des ressources

On peut

- fermer un ResultSet : libère les ressources utilisées par ce ResultSet.
- fermer un Statement : libère les ressources utilisés par le Statement, et invalide le `ResultSet` issu de ce `Statement` (il faut attendre le passage du garbage collector pour récupérer les ressources utilisées par le `ResultSet`)
- fermer la connexion : la fermeture de la connexion entraîne la fermeture des `Statements` associés.

```
rs.close();
stmt.close() ;
connect.close() ;
```


Batch

- Depuis JDBC 3.0, on peut envoyer une liste d'instructions à exécuter.
- Cette fonctionnalité n'est pas requise, il peut donc y avoir des pilotes qui ne l'implémentent pas.
- Ce sont forcément des instructions qui ne renvoient pas un ResultSet : instructions DML update, delete, insert, ou instruction DDL.
- Un Statement dispose de méthodes pour gérer cette liste de commandes :
  - `void addBatch(String sql)` Ajoute dans la liste la requête passée en paramètre.
  - `void clearBatch()` Vide la liste des instructions.
  - `int[] executeBatch()` Exécute les instructions de la liste. Le tableau d'entiers renvoyé contient les résultats d'exécutions des commandes.

tableau résultat

- Le tableau renvoyé par `executeBatch` contient en i^{eme} position :
  1. Un nombre ≥ 0 : nombre de lignes affectées par l'exécution de la i^{eme} instruction, qui s'est correctement exécutée.
  2. `SUCCESS_NO_INFO` : l'instruction s'est bien exécutée mais on n'a pas d'information sur le nombre de lignes affectées.
  3. `EXECUTE_FAILED` : l'instruction ne s'est pas bien exécutée. Le tableau est récupéré via l'exception.
- Si échec d'une instruction : `BatchUpdateException`
  - le pilote peut continuer ou arrêter de traiter les commandes restantes.
  - on récupère le tableau des résultats par la méthode `BatchUpdateException.getUpdateCounts`.
  - si le pilote continue : ce tableau contient autant d'éléments qu'il y a de commandes dans la liste batch (et donc au moins un `EXECUTE_FAILED`).
  - Avec Oracle, le traitement s'arrête à la première erreur, et le tableau contient donc les compteurs pour toutes les instructions qui se sont correctement exécutées.

Batch : exemple

```
// s de type Statement, initialisé par un createStatement
int[] results ;
// on commence par ajouter des instructions dans le batch
try {
    s.addBatch("create table T(a number, b varchar(10))");
    s.addBatch("insert into T values (1,'toto')");
    s.addBatch("delete from T where a=3");
} catch (SQLException e) { ...  }
try {
    results = s.executeBatch();
    for (int i=0 ; i<results.length ; i++){
      System.out.println(results[i]);
    }// si tout se passe bien, affiche 0, 1, 0
} catch (BatchUpdateException e) {  // une instruction pose pb
    results = e.getUpdateCounts() ;
    for (int i=0 ; i<results.length ; i++){
        System.out.println(results[i]);
    }// pour les instructions qui précèdent celle qui pose pb
} catch (SQLException ee){ ... }
```


Un peu plus sur les ResultSet

- Par défaut, un ResultSet possède un sens unique de parcours (`TYPE_FORWARD_ONLY`)
  On peut changer ce sens de parcours
- Par défaut, un ResultSet est en lecture seule
  On peut aussi créer des ResultSet modifiables.
- On peut paramétrer son comportement par rapport aux autres

Trois types de ResultSet

- `TYPE_FORWARD_ONLY` : Il n'y a que le sens de parcours en avant, de la première ligne à la dernière. Les lignes contenues dans le ResultSet peuvent être, selon l'implémentation,
  - Celles obtenues à l'éxécution de la commande SQL
  - ou Celles obtenues au moment de la lecture par la méthode next
- `TYPE_SCROLL_INSENSITIVE` On peut aller en avant, en arrière, et se rendre à une position absolue. Le ResultSet n'est pas sensible aux changements réalisés de façon concurrente sur les données sous-jacentes au ResultSet.
- `TYPE_SCROLL_SENSITIVE` Deux sens de parcours et sensible aux changements réalisées par ailleurs.

Opérations possibles

- `CONCUR_READ_ONLY` L'objet ResultSet ne peut pas être modifié.
- `CONCUR_UPDATABLE` L'objet ResultSet peut être utilisé pour faire des mises-à-jour.

```
Statement stmt =
connect.createStatement(
            ResultSet.TYPE_SCROLL_INSENSITIVE,
            ResultSet.CONCUR_UPDATABLE);
ResultSet rs
= stmt.executeQuery("SELECT a, b FROM TABLE2");
// rs will be scrollable,
// will not show changes made by others,
// and will be updatable
```


Déplacements (scrolling)

rs.next()	ligne suivante
rs.previous()	ligne précédente
rs.absolute(i)	aller à la $i^{ème}$ ligne
rs.absolute(-i)	aller à la $i^{ème}$ ligne en partant de la dernière
rs.relative(i)	descendre de i lignes
rs.relative(-i)	remonter de i lignes
rs.afterLast()	aller après la dernière ligne
rs.isAfterLast()	retourne vrai si après dernière ligne
rs.last()	aller à la dernière ligne (comme absolute(-1))
rs.beforeFirst()	aller avant la première ligne
rs.isBeforeFirst()	retourne vrai si avant première ligne
rs.first()	aller à la première ligne (comme absolute(1))

Déplacements (suite)

- La fonction `int getRow()` permet de récupérer le numéro de la ligne courante.
- Les fonctions `first`, `last`, `absolute`, `relative`, `next`, `previous` renvoient un booléen qui indique si la nouvelle ligne courante est une "vraie" ligne.

Modification de lignes

- méthodes de `ResultSet` :
  `void update`*`Type`*`(String nomColonne,` *`type`*
  `nouvelleValeur)`
  `void updateRow()`
  `void cancelRowUpdates()`
- Un exemple :

```
srs.updateString("nom","Dupont") ;
srs.updateRow() ; // changement effectué en base
srs.updateString("nom","Caron") ;
srs.cancelRowUpdates() ;
//seul le 2nd updateString est invalidé
```

Insertion de lignes

Position particulière dans le curseur pour une nouvelle ligne.

- méthodes de `ResultSet` :
  `void moveToInsertRow()`
  `void moveToCurrentRow()`
  `void insertRow()`
- Un exemple :

```
// phase d'insertion
srs.moveToInsertRow();
srs.updateInt("num",5) ;
srs.updateString("nom", "Zidane");
srs.insertRow(); // insertion dans la base
srs.moveToCurrentRow() ;
// retour à la position avant phase d'insertion
```


Suppression et "refresh"

- méthodes pour `ResultSet` :
  `void deleteRow()`
  Supprime la ligne courante.
  `void refreshRow()`
  en mode TYPE_SCROLL_SENSITIVE, permet de prendre en compte les modifications faites par d'autres *sur la ligne courante*.

Le niveau Meta

Une meta-donnée est une donnée qui décrit une donnée

- Des Exemples :
  - Modèle Relationnel : toutes les informations sur le schéma sont stockées dans le dictionnaire.
  - Java : getClass() , getMethods, getFields(), ...
- JDBC propose une API pour analyser une base (introspection)
  - `ResultSetMetaData` : Analyser dynamiquement la structure d'une relation résultat
  - `DatabaseMetaData` : API très riche qui permet de connaître les caractéristiques de la base de données.
- Alternative : les vues du dictionnaire, mais méthode non portable

ResultSetMetaData

- `ResultSetMetaData rsmd = rs.getMetaData() ;`

int getColumnCount()	le nombre de colonnes
int getColumnDisplaySize(int column)	taille d'affichage d'une col
String getColumnLabel(int column)	nom suggéré d'une colonne
String getColumnName(int column)	nom de colonne
int getColumnType(int column)	type (cste) de la colonne
String getColumnTypeName(int column)	nom du type de la colonne