La persistance des données avec SQLite

Jean-marc Farinone

Remarques sur SQLite

- La base de données `FILENAME` est stockée dans le smartphone sous `/data/data/NOM_PACKAGE_APPLI/databases/FILENAME` où `NOM_PACKAGE_APPLI` est le nom du package de l'application Android (celui indiqué dans `AndroidManifest.xml`)
- L'exécution des requêtes est effectuée dans le même processus que l'application Android et une BD est réservée à l'application créatrice : seule cette application créatrice peut y accéder. Si on veut des données partageables entre applications Android, voir les `ContentProvider`
- "Any databases you create will be accessible by name to any class in the application, but not outside the application."
  source :
  `http://developer.android.com/guide/topics/data/data-storage.html#db`

Une BD Android = une `SQLiteDatabase`

- Dans le code, une base de données est modélisée par un objet de la classe `android.database.sqlite.SQLiteDatabase`
- Cette classe permet donc d'insérer des données (`insert()`), de les modifier (`update()`), de les enlever (`delete()`), de lancer des requêtes SELECT (par `query()`) ou des requêtes qui ne retournent pas de données par `execSQL()`
- Les requêtes Create, Read, Update, Delete (~ INSERT, SELECT, UPDATE, DELETE de SQL) sont dites des requêtes CRUD

Une classe d'aide (helper)

- Pour créer et/ou mettre à jour une BD, on écrit une classe qui hérite de la classe abstraite `android.database.sqlite.SQLiteOpenHelper`
- Cela nécessite de créer un contructeur qui appelera un des constructeurs avec argument de `SQLiteOpenHelper` : il n'y a pas de constructeur sans argument dans `SQLiteOpenHelper`
- Le constructeur de `SQLiteOpenHelper` utilisé est
  `public SQLiteOpenHelper (Context context, String name, SQLiteDatabase.CursorFactory factory, int version)`
  - `context` est le contexte de l'application
  - `name` est le nom du fichier contenant la BD
  - `factory` est utilisé pour créer des `Cursor`. En général on met `null`
  - `version` est le numéro de version de la BD (commençant à 1)

Du helper à SQLiteDatabse

- = de la classe d'aide à la base de données

- Le constructeur précédent est un proxy qui est éxécuté rapidement
- La BD sera réellement créée au lancement de `getWritableDatabase()` (pour une base en lecture et écriture) ou de `getReadableDatabase()` (pour une base en lecture seule) sur cet objet de la classe d'aide
- Bref on a :

```
private class MaBaseOpenHelper extends SQLiteOpenHelper { ... }


MaBaseOpenHelper leHelper = new MaBaseOpenHelper(...);
SQLiteDatabase maBaseDonnees = leHelper.getXXXableDatabase();
```

et un helper est (évidemment) associé à une base de données. XXX est soit Writ soit Read

Création, mise à jour d'une BD : code du helper

- Sur un objet d'une classe héritant de `SQLiteOpenHelper` (classe d'aide), certaines méthodes sont appelées automatiquement :
  - `public void onCreate(SQLiteDatabase db)` est appelée automatiquement par l'environnement d'exécution quand la BD n'existe pas. On met le code de création des tables et leurs contenus dans cette méthode
  - `public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion)` quand le numéro de version de la BD a été incrémentée
- Ces deux méthodes sont abstraites dans la classe de base et doivent donc être implémentées dans la classe d'aide
- `maBaseDonnees` de classe `SQLiteDatabase` sera obtenu comme retour de `getWritableDatabase()` ou `getReadableDatabase()`

Code de la classe d'aide

MaBaseOpenHelper

```
private class MaBaseOpenHelper extends SQLiteOpenHelper {
    private static final String REQUETE_CREATION_TABLE = "create table "
            + TABLE_PLANETES + " (" + COLONNE_ID
            + " integer primary key autoincrement, " + COLONNE_NOM
            + " text not null, " + COLONNE_RAYON + " text not null);";

    public MaBaseOpenHelper(Context context, String nom,
            CursorFactory cursorfactory, int version) {
        super(context, nom, cursorfactory, version);
    }

    @Override
    public void onCreate(SQLiteDatabase db) {
        db.execSQL(REQUETE_CREATION_TABLE);
    }

    @Override
    public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) {
        // Dans notre cas, nous supprimons la base et les données pour en
        // créer une nouvelle ensuite. Vous pouvez créer une logique de mise
        // à jour propre à votre base permettant de garder les données à la
        // place.
        db.execSQL("drop table" + TABLE_PLANETES + ";");
        // Création de la nouvelle structure.
        onCreate(db);
    }
}
```

Insérer des données dans une `SQLiteDatabase` (1/2)

- Ayant obtenu une `SQLiteDatabase`, on utilise les méthodes de cette classe pour faire des opérations sur la base de données
- `public long insert (String table, String nullColumnHack, ContentValues values)` insert dans la table `table` les valeurs indiquées par `values`
- `values`, de classe `ContentValues`, est une suite de couples (clé, valeur) où la clé, de classe `String`, est le nom de la colonne et `valeur`, sa valeur
- Bref on prépare tout, la ligne à insérer en remplissant `values` par des `put()` successifs puis on lance `insert()`
- Exemple :

```
/**
 * Insère une planète dans la table des planètes.
*/
public long insertPlanete(Planete planete) {
    ContentValues valeurs = new ContentValues();
    valeurs.put(COLONNE_NOM, planete.getNom());
    valeurs.put(COLONNE_RAYON, planete.getRayon());
    return maBaseDonnees.insert(TABLE_PLANETES, null, valeurs);
}
```

Insérer des données dans une `SQLiteDatabase` (2/2)

- Le second argument `nullColumnHack` est le nom de colonne qui aura la valeur NULL si `values` est vide. Cela est du au fait que SQLite ne permet pas de lignes vides. Ainsi avec cette colonne, au moins un champ dans la ligne aura une valeur (= NULL). Bref cela sert seulement lorsque `values` est vide !
- Cette méthode `insert()` retourne le numéro de la ligne insérée ou -1 en cas d'erreur
- En fait cette insertion est le Create (C) de CRUD

Récupérer des données dans une `SQLiteDatabase`

- La méthode la plus simple (!) pour récupérer des données (~ SELECT) est :
  `public Cursor query (String table, String[] columns, String whereClause, String[] selectionArgs, String groupBy, String having, String orderBy)`
  - `columns` est la liste des colonnes à retourner. Mettre `null` si on veut toutes les colonnes
  - `whereClause` est la clause WHERE d'un SELECT (sans le mot WHERE). Mettre `null` si on veut toutes les lignes
  - `selectionArgs` est utile si dans `whereClause` (~ WHERE), il y a des paramêtres notés `?`. Les valeurs de ces paramêtres sont indiqués par `selectionArgs`. Bref en général on met `null`
  - `groupBy` est la clause GROUP BY d'un SELECT (sans les mots GROUP BY). Utile pour des SELECT COUT(*). Bref on général on met `null`
  - `having` indique les groupes de lignes à retourner (comme HAVING de SQL = un WHERE sur résultat d'un calcul, pas sur les données)
  - `orderBy` est la clause ORDER BY d'un SELECT (sans les mots ORDER BY). Mettre `null` si on ne tient pas compte de l'ordre

Exemple de requête SELECT pour Android

- Euh, la classe `android.database.sqlite.SQLiteQueryBuilder` est (semble) faite pour cela
- Voir à `http://sqlite.org/lang.html`
- Sinon, quelques exemples
- Rappel : `SELECT` peut être obtenu avec : `public Cursor query (String table, String[] columns, String whereClause, String[] selectionArgs, String groupBy, String having, String orderBy)`
- `db.query(TABLE_CONTACTS, new String[] { KEY_ID, KEY_NAME, KEY_PH_NO }, KEY_ID + "=?", new String[] { String.valueOf(id) }, null, null, null, null);` est l'équivalent de `"SELECT KEY_ID, KEY_NAME, KEY_PH_NO FROM TABLE_CONTACTS WHERE KEY_ID='" + id + "'"`

rawQuery() : requête SELECT pour Android

- La méthode rawQuery() de SQLiteDatabase permet de lancer des simples requêtes SELECT comme SELECT * FROM " + TABLE_CONTACTS WHERE *condition paramétrée*
- Sa signature est public Cursor rawQuery (String sql, String[] selectionArgs) où sql est une requête SELECT (qui ne doit pas être terminée par ;) et selectionArgs est le tableau qui fixe les valeurs des paramètres (noté ?) dans la clause WHERE
- Par exemple :

```
String countQuery = "SELECT  * FROM " + TABLE_CONTACTS;
SQLiteDatabase db = this.getReadableDatabase();
Cursor cursor = db.rawQuery(countQuery, null);
```

L'objet `Cursor` (1/2)

- La méthode `query()` retourne un `android.database.Cursor` (~ `java.sql.ResultSet`). `android.database.Cursor` est une interface. Ce qui est retourné est un objet d'une classe qui implémente cette interface
- C'est similaire à JDBC. Le `Cursor` représente un ensemble de "lignes" contenant le résultat de la requête SELECT
- `public int getCount()` retourne le nombre de lignes contenues dans le `Cursor`
- On se positionne au début du `Cursor` (= avant la première ligne) par la méthode `public boolean moveToFirst()` (qui retourne `false` si le `Cursor` est vide
- On teste si on a une nouvelle ligne à lire par la méthode `public boolean moveToNext()` (qui retourne `false` si on était positionné après la dernière ligne)

L'objet `Cursor` (2/2)

- On récupère la `columnIndex` cellule de la ligne par la méthode : `public XXX getXXX(int columnIndex)`. `columnIndex` est (évidemment) le numéro de la cellule dans la requête. `XXX` est le type retourné (String, short, int, long, float, double)
  - Il n'y a pas de getXXX(String nomDeColonne) contrairement à JDBC
- On referme le `Cursor` (et libère ainsi les ressources) par `public void close ()`
- On peut avoir des renseignements sur le résultat de la requête SELECT (* FROM ...) (Méta données) à l'aide du `Cursor` comme :
  - `public int getColumnCount()` qui retourne le nombre de colonnes contenues dans le `Cursor`
  - `public String getColumnName(int columnIndex)` qui retourne le nom de la `columnIndex` ième colonne

L'accès aux données : un DAO

- Manipuler le `Cursor`, c'est bien. C'est "un peu" de la programmation "bas niveau"
- Bref un DAO (= Data Access Object), voire une façade s'impose !
- Pour accéder aux données, on masque les bidouilles sous jacentes (requête SQL, etc.) par un objet d'une classe DAO : un bon design pattern !
- Les bidouilles SQL masquées par le DAO sont :
  - insérer des données dans la BD par la méthode `insert()` de la classe `SQLiteDatabase`
  - retourner toutes les données d'une table par la méthode `query()` de la classe `SQLiteDatabase`

Les constantes de l'application

- Ce sont :

```
    private static final int BASE_VERSION = 1;
    private static final String BASE_NOM = "planetes.db";

    private static final String TABLE_PLANETES = "table_planetes";

    public static final String COLONNE_ID = "id";
    public static final int COLONNE_ID_ID = 0;
    public static final String COLONNE_NOM = "nom";
    public static final int COLONNE_NOM_ID = 1;
    public static final String COLONNE_RAYON = "rayon";
    public static final int COLONNE_RAYON_ID = 2;
```

Le code du DAO (1/4)

```
public class PlanetesDB_DAO {
    private SQLiteDatabase maBaseDonnees;
    private MaBaseOpenHelper baseHelper;

    public PlanetesDB_DAO(Context ctx) {
        baseHelper = new MaBaseOpenHelper(ctx, BASE_NOM, null, BASE_VERSION);
    }

    public SQLiteDatabase open() {
        maBaseDonnees = baseHelper.getWritableDatabase();
        return maBaseDonnees;
    }

    public void close() {
        maBaseDonnees.close();
    }

    /**
     * Récupère une planète en fonction de son nom.
     * @param nom
     *            Le nom de la planète à retourner.
     * @return La planète dont le nom est égale au paramètre 'nom'.
     */
    public Planete getPlanete(String nom) {
        Cursor c = maBaseDonnees.query(TABLE_PLANETES, new String[] {
                COLONNE_ID, COLONNE_NOM, COLONNE_RAYON }, null, null, null,
                COLONNE_NOM + " LIKE " + nom, null);
        return cursorToPlanete(c);
    }
```

Le code du DAO (2/4)

```
private Planete cursorToPlanete(Cursor c) {
     // Si la requête ne renvoie pas de résultat
     if (c.getCount() == 0)
          return null;

     Planete retPlanete = new Planete();
     // Extraction des valeurs depuis le curseur
     retPlanete.setId(c.getInt(COLONNE_ID_ID));
     retPlanete.setNom(c.getString(COLONNE_NOM_ID));
     retPlanete.setRayon(c.getFloat(COLONNE_RAYON_ID));
     // Ferme le curseur pour libérer les ressources
     c.close();
     return retPlanete;
}
```

Le code du DAO (3/4)

```
    /**
     * Retourne toutes les planètes de la base de données.
     *
     * @return Un ArrayList<Planete> contenant toutes les planètes de la BD
     */

    public ArrayList<Planete> getAllPlanetes() {
        Cursor c = maBaseDonnees.query(TABLE_PLANETES, new String[] {
                 COLONNE_ID, COLONNE_NOM, COLONNE_RAYON }, null, null, null,
                 null, null);
        return cursorToPlanetes(c);
    }

    private ArrayList<Planete> cursorToPlanetes(Cursor c) {
        // Si la requête ne renvoie pas de résultat
        if (c.getCount() == 0)
            return new ArrayList<Planete>(0);

        ArrayList<Planete> retPlanetes = new ArrayList<Planete>(c.getCount());
        c.moveToFirst();
        do {
            Planete planete = new Planete();
            planete.setId(c.getInt(COLONNE_ID_ID));
            planete.setNom(c.getString(COLONNE_NOM_ID));
            planete.setRayon(c.getFloat(COLONNE_RAYON_ID));
            retPlanetes.add(planete);
        } while (c.moveToNext());
        // Ferme le curseur pour libérer les ressources
        c.close();
        return retPlanetes;
    }
```

Le code du DAO (4/4)

```
    /**
     * Insère une planète dans la table des planètes.
     *
     * @param planete
     *            La planète à insérer.
     */
    public long insertPlanete(Planete planete) {
        ContentValues valeurs = new ContentValues();
        valeurs.put(COLONNE_NOM, planete.getNom());
        valeurs.put(COLONNE_RAYON, planete.getRayon());
        return maBaseDonnees.insert(TABLE_PLANETES, null, valeurs);
    }

    public void videLaBase() {
        // Dans notre cas, nous supprimons la base et les données pour en
        // créer une nouvelle ensuite. Vous pouvez créer une logique de mise
        // à jour propre à votre base permettant de garder les données à la
        // place.
        maBaseDonnees.execSQL("drop table " + TABLE_PLANETES + ";");
        // Création de la nouvelle structure.
        maBaseDonnees.execSQL(REQUETE_CREATION_TABLE);
    }
}
```

Démonstration

- Projet ProgAndroidBDJMFProjet dans ...\Travail

Retour sur l'IHM : le TableLayout

- Pour afficher des enregistrements, une table est appropriée
- Android propose le `TableLayout` calqué sur l'attribut `table` de HTML :

```
<table>
     <tr><td>et 1</td><td>et 2</td></tr>
</table>
```

a pour équivalent dans les fichiers d'IHM d'Android :

```
<TableLayout>
     <TableRow><TextView ...>et 1</TextView><TextView ...>et 2</TextView></TableRow>
</TableLayout>
```

- Néanmoins, il faut souvent construire cette `TableLayout` dynamiquement (on ne connait pas le nombre d'articles à afficher) :

```
TableLayout table = (TableLayout) findViewById(R.id.tableLayoutLesContacts);
Iterator<Contact> it = arContacts.iterator();
while (it.hasNext()) {
     Contact ct = (Contact)it.next();
     // création d'une nouvelle TableRow
     TableRow row = new TableRow(this);
     TextView tNom = new TextView(this);
     tNom.setText(ct.getName());
     row.addView(tNom);
     TextView tNumTel = new TextView(this);
     tNumTel.setText(ct.getPhoneNumber());
     row.addView(tNumTel);
     table.addView(row,new TableLayout.LayoutParams(LayoutParams.WRAP_CONTENT,
LayoutParams.WRAP_CONTENT));
}
```

Modifier une table : UPDATE de SQL pour Android

- Pour mettre à jour des lignes dans une table, la méthode utilisée (de la classe `SQLiteDatabase`) est `public int update (String table, ContentValues values, String whereClause, String[] whereArgs)` où :
  - `table` est la table qui doit être mise à jour
  - `values` est une suite de couples (clé, valeur) où la clé, de classe `String`, est le nom de la colonne et valeur, sa valeur
  - `whereClause` est la clause WHERE filtrant les lignes à mettre à jour. Si la valeur est `null`, toutes les lignes sont mises à jour
  - `whereArgs` indiquent les valeurs à passer aux différents arguments de la clause WHERE qui sont notés `?` dans `whereClause`
- Cette méthode retourne le nombre de ligne qui ont été affectées
- Exemple :

```
SQLiteDatabase db = this.getWritableDatabase();
ContentValues values = new ContentValues();
values.put(KEY_NAME, contact.getName());
values.put(KEY_PH_NO, contact.getPhoneNumber());
db.update(TABLE_CONTACTS, values, KEY_ID + " = ?",
         new String[] { String.valueOf(contact.getID()) });
```

Supprimer des lignes dans une table : DELETE

- Pour supprimer des lignes dans une table, la méthode utilisée (de la classe `SQLiteDatabase`) est `public int delete (String table, String whereClause, String[] whereArgs)`
  - `table` est la table à manipuler
  - `whereClause` est la clause WHERE filtrant les lignes à supprimer. Si la valeur est `null`, toutes les lignes sont détruites
  - `whereArgs` indiquent les valeurs à passer aux différents arguments de la clause WHERE qui sont notés `?` dans `whereClause`
- Cette méthode retourne le nombre de ligne qui ont été supprimées
- Exemple :

```
public void deleteContact(Contact contact) {
     SQLiteDatabase db = this.getWritableDatabase();
     db.delete(TABLE_CONTACTS, KEY_ID + " = ?",
               new String[] { String.valueOf(contact.getID()) });
     db.close();
}
```

Bibliographie pour ce chapitre

- Programmation Android, De la conception au déploiement avec lee SDK Google Android 2, *Damien Guignard, Julien Chable, Emmanuel Robles* ; editions Eyrolles, chapitre 6
- Un tutorial sur SQLite : `http://www.androidhive.info/2011/11/android-sqlite-database-tutorial/`

Fin