Android
–
Persistance et partage de données

C. Raïevsky
Avec la courtoisie de S. Jean

UNIVERSITÉ Grenoble Alpes — IUT Valence

Département Informatique

Persistance des données

- Certaines applications ont besoin de **sauvegarder des informations entre 2 exécutions**
- Plusieurs mécanismes existent dans Android
  - **Préférences**
  - **Stockage interne**
  - **Stockage externe**
  - **Base de données embarquée**
  - **Stockage distant**
- Certains types de stockage sont par nature **privés**
  - Il est possible de partager des données persistantes entre applications via des fichiers ou l'API ***Content Provider***

Préférences

- Stockage, par défaut privé, de **paires clé/valeurs**
  - Valeurs de **types simples** : `int`, `long`, `float`, ou `String`
- Obtenir (ou créer) un profil de préférences depuis une activité
  - `getSharedPreferences(name)` ou `getPreferences()`
  - Retourne une référence du type `SharedPreferences` permettant de retrouver les valeurs
  - L'ajout ou la suppression de valeur s'effectue à travers un objet spécifique (`SharedPreferences.Editor`)

Préférences

Summary

Nested Classes		
interface	SharedPreferences.Editor	Interface used for modifying values in a `SharedPreferences` object.
interface	SharedPreferences.OnSharedPreferenceChangeListener	Interface definition for a callback to be invoked when a shared preference is changed.

Public Methods	
abstract boolean	contains(String key) Checks whether the preferences contains a preference.
abstract SharedPreferences.Editor	edit() Create a new Editor for these preferences, through which you can make modifications to the data in the preferences and atomically commit those changes back to the SharedPreferences object.
abstract Map<String, ?>	getAll() Retrieve all values from the preferences.
abstract boolean	getBoolean(String key, boolean defValue) Retrieve a boolean value from the preferences.
abstract float	getFloat(String key, float defValue) Retrieve a float value from the preferences.
abstract int	getInt(String key, int defValue) Retrieve an int value from the preferences.
abstract long	getLong(String key, long defValue) Retrieve a long value from the preferences.
abstract String	getString(String key, String defValue) Retrieve a String value from the preferences.
abstract Set<String>	getStringSet(String key, Set<String> defValues) Retrieve a set of String values from the preferences.
abstract void	registerOnSharedPreferenceChangeListener(SharedPreferences.OnSharedPreferenceChangeListener listener) Registers a callback to be invoked when a change happens to a preference.
abstract void	unregisterOnSharedPreferenceChangeListener(SharedPreferences.OnSharedPreferenceChangeListener listener) Unregisters a previous callback.

Préférences

Summary

Public Methods	
abstract void	apply() Commit your preferences changes back from this Editor to the SharedPreferences object it is editing.
abstract SharedPreferences.Editor	clear() Mark in the editor to remove *all* values from the preferences.
abstract boolean	commit() Commit your preferences changes back from this Editor to the SharedPreferences object it is editing.
abstract SharedPreferences.Editor	putBoolean(String key, boolean value) Set a boolean value in the preferences editor, to be written back once commit() or apply() are called.
abstract SharedPreferences.Editor	putFloat(String key, float value) Set a float value in the preferences editor, to be written back once commit() or apply() are called.
abstract SharedPreferences.Editor	putInt(String key, int value) Set an int value in the preferences editor, to be written back once commit() or apply() are called.
abstract SharedPreferences.Editor	putLong(String key, long value) Set a long value in the preferences editor, to be written back once commit() or apply() are called.
abstract SharedPreferences.Editor	putString(String key, String value) Set a String value in the preferences editor, to be written back once commit() or apply() are called.
abstract SharedPreferences.Editor	putStringSet(String key, Set<String> values) Set a set of String values in the preferences editor, to be written back once commit() or apply() is called.
abstract SharedPreferences.Editor	remove(String key) Mark in the editor that a preference value should be removed, which will be done in the actual preferences once commit() is called.

Préférences

- Exemple d'utilisation des préférences (source : android.developer.com)

```
public class Calc extends Activity {
    public static final String PREFS_NAME = "MyPrefsFile";

    @Override
    protected void onCreate(Bundle state){
       super.onCreate(state);
       . . .

       // Restore preferences
       SharedPreferences settings = getSharedPreferences(PREFS_NAME, 0);
       boolean silent = settings.getBoolean("silentMode", false);
       setSilent(silent);
    }

    @Override
    protected void onStop(){
       super.onStop();

      // We need an Editor object to make preference changes.
      // All objects are from android.context.Context
      SharedPreferences settings = getSharedPreferences(PREFS_NAME, 0);
      SharedPreferences.Editor editor = settings.edit();
      editor.putBoolean("silentMode", mSilentMode);

      // Commit the edits!
      editor.commit();
    }
}
```


Stockage interne

- Lecture/écriture de données dans un **fichier**, dans l'**espace de stockage interne** associé à l'application
  - API pour création/écriture/lecture
  - Données privées
  - Non visible par l'utilisateur
  - Destruction à la suppression de l'application
- Création possible d'un stockage interne en **lecture seule** par dépôt dans `/res/raw` et accessible via `R.raw`

Stockage interne

- **Création et écriture dans un fichier**
  - L'appel à `openFileOutput` sur l'activité ou le contexte retourne une référence sur le flux d'écriture dans le fichier (créé si inexistant)
    - cf. `FileOutputStream`
- **Lecture dans un fichier**
  - L'appel à `openFileInput` sur l'activité ou le contexte retourne une référence sur le flux de lecture dans le fichier
    - cf. `FileInputStream`
  - L'appel à `openRawInput(resId)` sur l'activité ou le contexte retourne une référence sur le flux de lecture dans le fichier en lecture seule
- `getCacheDir` retourne une référence de type `File` désignant le répertoire où peuvent être créés des **fichiers temporaires**

Stockage externe

- Utilisation de l'espace de stockage externe (typiquement SD)
  - Pas toujours disponible (par exemple si le stockage externe est déjà monté par un PC sur lequel l'équipement est connecté)
  - L'état de disponibilité et les restrictions d'accès en écriture sont disponibles via `Environment.getExternalStorageState()`
- Fichiers **publics** (catégories prédéfinies), ou **privés**
  - En cas de montage externe, les fichiers sont visibles (et lisibles)
- L'accès à des fichiers n'appartenant pas à l'application nécessite de requérir les **permissions** dans le manifeste (WRITE → read+write)

```
<manifest ...>
    <uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE" />
    ...
</manifest>
```


Stockage externe

- **Catégories prédéfinies** de fichiers (*c.f.* classe `Environment`)
  - `DIRECTORY_MUSIC`, `DIRECTORY_PODCASTS`, `DIRECTORY_RINGTONES`, `DIRECTORY_ALARMS`, `DIRECTORY_NOTIFICATIONS`, `DIRECTORY_DCIM`, `DIRECTORY_PICTURES`, `DIRECTORY_MOVIES`, `DIRECTORY_DOWNLOADS`
- Fichiers/répertoires publics
  - Accès via `getExternalStoragePublicDirectory(type)`
  - Catégories prédéfinies uniquement
- Fichiers/répertoires privés
  - Accès via `getExternalStorageDirectory(type)`
  - Catégories prédéfinies (espaces privés) ou `null`
- Accès en lecture/écriture via des flux, création de répertoires via `File.mkdirs`

Bases de données

- Base de données embarquée **SQLite**
  - Open source (`http://www.sqlite.org`)
  - **Empreinte mémoire faible** ($\simeq$ 250kB)
  - **support limité des types** (`TEXT`, `INTEGER`, `REAL`)
- API pour la création de bases et le traitement de requêtes SQL
  - Par extension de la classe `SQLiteOpenHelper`

Bases de données

SQLiteOpenHelper, exemple :

- Ici, `SQL_CREATE_ENTRIES` et `SQL_DELETE_ENTRIES` sont des **requêtes de création et de destruction de tables**
- Il est préférable que chaque table possède une **clé primaire** de type `INTEGER` appelée `ID`

```
public class FeedReaderDbHelper extends SQLiteOpenHelper {
    // If you change the database schema, you must increment the database version.
    public static final int DATABASE_VERSION = 1;
    public static final String DATABASE_NAME = "FeedReader.db";

    public FeedReaderDbHelper(Context context) {
        super(context, DATABASE_NAME, null, DATABASE_VERSION);
    }
    public void onCreate(SQLiteDatabase db) {
        db.execSQL(SQL_CREATE_ENTRIES);
    }
    public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) {
        // This database is only a cache for online data, so its upgrade policy is
        // to simply to discard the data and start over
        db.execSQL(SQL_DELETE_ENTRIES);
        onCreate(db);
    }
    public void onDowngrade(SQLiteDatabase db, int oldVersion, int newVersion) {
        onUpgrade(db, oldVersion, newVersion);
    }
}
```

Bases de données

- L'objet de type `SQLiteOpenHelper` est créé en général dans la **méthode `onCreate` d'une activité**
- La base de données créée est stockée sous forme de **fichier** dans ***DATA*/data/APP_NAME/databases/DATABASE_NAME**
  - *DATA* est le répertoire retourné par un appel à `Environment.getDataDirectory()`
  - Remarque : l'exemple précédent est simpliste et écrase la base de données à chaque démarrage de l'application (un test d'existence du fichier permet de l'éviter)
- La base de données est **détruite à la désinstallation** de l'application
- Remarque : il est aussi possible d'embarquer dans l'archive `.apk` le fichier d'une base de données créée et alimentée hors de l'équipement (cf répertoire `asset`)

Bases de données

- **Ajout de données dans une table**
  - `getWritableDatabase` permet d'accéder en écriture à la base

```
// Gets the data repository in write mode
SQLiteDatabase db = mDbHelper.getWritableDatabase();

// Create a new map of values, where column names are the keys
ContentValues values = new ContentValues();
values.put(FeedEntry.COLUMN_NAME_ENTRY_ID, id);
values.put(FeedEntry.COLUMN_NAME_TITLE, title);
values.put(FeedEntry.COLUMN_NAME_CONTENT, content);

// Insert the new row, returning the primary key value of the new row
long newRowId;
newRowId = db.insert(
         FeedEntry.TABLE_NAME,
         FeedEntry.COLUMN_NAME_NULLABLE,
         values);
```


Bases de données

- **Recherche de données dans une table**
  - `Cursor` permet d'analyser les résultats (≃ `ResultSet` dans JDBC)

```
SQLiteDatabase db = mDbHelper.getReadableDatabase();

// Define a projection that specifies which columns from the database
// you will actually use after this query.
String[] projection = {
    FeedEntry._ID,
    FeedEntry.COLUMN_NAME_TITLE,
    FeedEntry.COLUMN_NAME_UPDATED,
    ...
    };

// How you want the results sorted in the resulting Cursor
String sortOrder =
    FeedEntry.COLUMN_NAME_UPDATED + " DESC";

Cursor c = db.query(
    FeedEntry.TABLE_NAME,  // The table to query
    projection,                               // The columns to return
    selection,                                // The columns for the WHERE clause
    selectionArgs,                            // The values for the WHERE clause
    null,                                     // don't group the rows
    null,                                     // don't filter by row groups
    sortOrder                                 // The sort order
    );
```


Bases de données

- **Suppression de données dans une table**

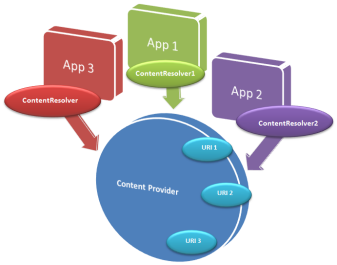
```
// Define 'where' part of query.
String selection = FeedEntry.COLUMN_NAME_ENTRY_ID + " LIKE ?";
// Specify arguments in placeholder order.
String[] selectionArgs = { String.valueOf(rowId) };
// Issue SQL statement.
db.delete(table_name, selection, selectionArgs);
```

- **Mise à jour de données dans une table**

```
SQLiteDatabase db = mDbHelper.getReadableDatabase();

// New value for one column
ContentValues values = new ContentValues();
values.put(FeedEntry.COLUMN_NAME_TITLE, title);

// Which row to update, based on the ID
String selection = FeedEntry.COLUMN_NAME_ENTRY_ID + " LIKE ?";
String[] selectionArgs = { String.valueOf(rowId) };

int count = db.update(
    FeedReaderDbHelper.FeedEntry.TABLE_NAME,
    values,
    selection,
    selectionArgs);
```

L'API Content Provider

- Permet l'**échange de données entre applications**
  - Echange contrôlé par un ensemble de **permissions**
- Les données sont vues avec un **formalisme proche de celui d'une base de données relationnelle**
- L'application "serveur de données" implémente l'interface `ContentProvider`, les applications "clientes de données" utilisent un `ContentResolver` et désignent les données cibles à travers des URIs



L'API Content Provider

- Utilisation du *ContentResolver* par une application cliente pour obtenir des données

```
// Queries the user dictionary and returns results
mCursor = getContentResolver().query(
    UserDictionary.Words.CONTENT_URI,   // The content URI of the words table
    mProjection,                        // The columns to return for each row
    mSelectionClause                    // Selection criteria
    mSelectionArgs,                     // Selection criteria
    mSortOrder);                        // The sort order for the returned rows
```

- Remarque : données obtenues à travers un objet de type `Cursor`

L'API Content Provider

- Similitudes `query` / `SQL`

query() argument	SELECT keyword/parameter	Notes
`Uri`	`FROM table_name`	`Uri` maps to the table in the provider named *table_name*.
`projection`	`col,col,col,...`	`projection` is an array of columns that should be included for each row retrieved.
`selection`	`WHERE col = value`	`selection` specifies the criteria for selecting rows.
`selectionArgs`	(No exact equivalent. Selection arguments replace `?` placeholders in the selection clause.)	
`sortOrder`	`ORDER BY col,col,...`	`sortOrder` specifies the order in which rows appear in the returned `Cursor`.

L'API Content Provider

- Implémentation d'un content provider
  - Choix du stockage des données
    - Fichier, ou base de données
  - Définition des URIs
  - Implémentation des méthodes de l'API `ContentProvider`

L'API Content Provider

- Méthodes de l'interface `ContentProvider`

`query()`
Retrieve data from your provider. Use the arguments to select the table to query, the rows and columns to return, and the sort order of the result. Return the data as a `Cursor` object.

`insert()`
Insert a new row into your provider. Use the arguments to select the destination table and to get the column values to use. Return a content URI for the newly-inserted row.

`update()`
Update existing rows in your provider. Use the arguments to select the table and rows to update and to get the updated column values. Return the number of rows updated.

`delete()`
Delete rows from your provider. Use the arguments to select the table and the rows to delete. Return the number of rows deleted.

`getType()`
Return the MIME type corresponding to a content URI. This method is described in more detail in the section Implementing Content Provider MIME Types.

`onCreate()`
Initialize your provider. The Android system calls this method immediately after it creates your provider. Notice that your provider is not created until a `ContentResolver` object tries to access it.

Notice that these methods have the same signature as the identically-named `ContentResolver` methods.

L'API Content Provider

- Déclaration du *content provider* dans le manifeste

```
<provider
       android:authorities="de.vogella.android.todos.contentprovider"
       android:name=".contentprovider.MyTodoContentProvider" >
</provider>
```

- Les URIs permettant l'accès aux données sont construites sous la forme `content://authority/path`

L'API Content Provider

- Déclaration des permissions

```
    <provider
        android:name=".data.DataProvider"
        android:multiprocess="true"
        android:authorities="myapp.data.DataProvider"
        android:readPermission="myapp.permission.READ"
        android:writePermission="myapp.permission.WRITE" />
```

```
<permission android:name="myapp.permission.READ"
    android:permissionGroup="myapp.permission-group.MYAPP_DATA"
    android:label="@string/perm_read"
    android:description="@string/perm_read_summary"
    android:protectionLevel="signature" />

<permission android:name="myapp.permission.WRITE"
    android:permissionGroup="myapp.permission-group.MYAPP_DATA"
    android:label="@string/perm_write"
    android:description="@string/perm_write_summary"
    android:protectionLevel="signature" />
```


Fin !