Persistenza ed SQL Lite

Programmazione Mobile

A.A. 2021/22

M.O. Spata

Persistenza ed SQLite

- La *persistenza* dei dati strutturati in Android può essere implementata attraverso diversi strumenti/meccanismi:
  - **database SQLite**: la libreria di database relazionali SQLite rappresenta il miglior sistema di gestione di dati strutturati, messo a disposizione da Android. Ogni applicazione può creare le proprie basi di dati ed averne il controllo assoluto.
  - I database di Android sono memorizzati nella directory del dispositivo *data / <nome_pacchetto> / database* e per default sono accessibili solo dall'applicazione che li ha creati.

SQLite

- **SQLite**: si tratta di un RDBMS open source, standard-compliant, computazionalmente leggero, single-tier.
- La versione di Android è una implementazione Compact C integrata nello stack.
- Questa scelta architetturale prevede che il database sia integrato nell'applicazione che lo ha creato.
- Ciò permette di ridurre le dipendenze esterne, minimizza i tempi di latenza e semplifica i meccanismi di lock e di sincronizzazione.

Content providers

- **content Providers**: sono oggetti che forniscono una interfaccia ben definita per l'utilizzo e la condivisione (tra applicazioni) dei dati.
  - Tale interfaccia è basata sul modello di indirizzamento attraverso URI e lo schema *content://* .
  - L'utilizzo di content providers permette di disaccoppiare il livello applicazione del livello dati.
  - E' possibile interrogare Content Providers condivisi per conoscere il numero e lo stato dei record (cancellati aggiornati o nuovi) ed inoltre ogni applicazione con i giusti privilegi (permission) può aggiungere, aggiornare o rimuovere dati di altre applicazioni (comprese le native).

Cursor

- **Oggetti Cursor**: Tutte le query in Android restituiscono dati sotto forma di oggetti Cursor. Questi rappresentano un puntatore al'insieme dei risultati e non una semplice copia dei valori richiesti. Come nella maggior parte di casi come questo i **Cursor** forniscono una serie di metodi che permettono di fare il browsing dei dati del database che esso contiene come ad esempio:

```
Cursor result = getAllContacts();
```

- `moveToFirst` Moves the cursor to the first row in the query result
- `moveToNext` Moves the cursor to the next row
- `moveToPrevious` Moves the cursor to the previous row
- `getCount` Returns the number of rows in the result set
- `getColumnIndexOrThrow` Returns the index for the column with the specified name (throwing an exception if no column exists with that name)
- `getColumnName` Returns the name of the specified column index
- `getColumnNames` Returns a string array of all the column names in the current Cursor
- `moveToPosition` Moves the Cursor to the specified row
- `getPosition` Returns the current Cursor position

Estrarre i dati da un Cursor

- Per estrarre dati dal **Cursor** restituito da una query, bisognerà dapprima posizionarsi nella riga corretta del Cursor, quindi si possono utilizzare i metodi **get<type>** che ricevono un parametro che rappresenta l'indice dalla colonna, e restituiscono il valore memorizzato nella riga corrente per la colonna specificata:

```
int GOLD_HOARDED_COLUMN = 2;
Cursor myGold = myDatabase.query("GoldHoards", null, null, null, null, null, null);
float totalHoard = 0f;

// Make sure there is at least one row.
if (myGold.moveToFirst()) {
  // Iterate over each cursor.
  do {
    float hoard = myGold.getFloat(GOLD_HOARDED_COLUMN);
    totalHoard += hoard;
  } while(myGold.moveToNext());
}

float averageHoard = totalHoard / myGold.getCount();
```

SQLiteOpenHelper

- **SQLiteOpenHelper**: si tratta di una classe astratta utilizzata per implementare le funzioni di creazione, apertura ed upgrade di undatabase di SQLite.
- L'implementazione di questa classe permette di nascondere la logica utilizzata nella gestione del database in termini appunto di creazione o upgrade prima della relativa apertura.
- Per implementare tale classe bisogna estendere la classe **SQLiteOpenHelper** e ridefinirne il costruttore ed i metodi onUpgrade ed onCreate.

Esempio

- Per utilizzare l'implementazione di questa classe è necessario crearne una nuova istanza e passare come *parametro* l'oggetto **context**, il **nome del database**, la **versione del db** ed un oggetto **CursorFactory**. Quindi bisogna chiamare le funzioni getReadableDatabase e/o getWritableDatabase per aprire e restituire una istanza (solo lettura o scrivibile) del database in questione:

```
dbHelper = new myDbHelper(context, DATABASE_NAME, null, DATABASE_VERSION);

SQLiteDatabase db;
try {
  db = dbHelper.getWritableDatabase();
}
catch (SQLiteException ex){
  db = dbHelper.getReadableDatabase();
}
```

openOrCreateDatabase

- E' possibile creare ed aprire un database anche senza la classe helper utilizzando il metodo **openOrCreateDatabase** del Context dell'applicazione.

```
private static final String DATABASE_NAME = "myDatabase.db";
private static final String DATABASE_TABLE = "mainTable";

private static final String DATABASE_CREATE =
  "create table " + DATABASE_TABLE + " ( _id integer primary key autoincrement," +
  "column_one text not null);";

SQLiteDatabase myDatabase;

private void createDatabase() {
  myDatabase = openOrCreateDatabase(DATABASE_NAME, Context.MODE_PRIVATE, null);
  myDatabase.execSQL(DATABASE_CREATE);
}
```

Query

- Attraverso la funzione db.query è possibile ottenere un oggetto Cursor che contiene i risultati dell'interrogazione
- Il metodo query che riceve come parametri:
  - un valore opzionale Boolean, per specificare se l'insieme di ritorno dovrebbe contenere solo valori univoci
  - il nome della tabella da interrogare
  - una proiezione ovvero un array di stringhe che elencano le colonne da includere nell'insieme di ritorno
  - una valore per la clausola "where" che definisce la condizione sulle righe da restituire (può includere ? ovvero parametri da sositutire con valori successivamente)
  - una array di argomenti stringa da sostiturire ai ? in where
  - una valore per clausola "group by" che definisce il tipo di raggruppamento da usare sulla selezione
  - una valore per clausola "having" che permette di filtrare sui raggruppamenti
  - una stringa che permette di definire l'ordine delle stringhe restitute
  - una stringa opzionale che definisce il numero massimo di righe da restitutire

Esempio

```
// Return all rows for columns one and three, no duplicates
String[] result_columns = new String[] {KEY_ID, KEY_COL1, KEY_COL3};

Cursor allRows = myDatabase.query(true, DATABASE_TABLE, result_columns,
                                  null, null, null, null, null, null);

// Return all columns for rows where column 3 equals a set value
// and the rows are ordered by column 5.
String where = KEY_COL3 + "=" + requiredValue;
String order = KEY_COL5;
Cursor myResult = myDatabase.query(DATABASE_TABLE, null, where,
                                   null, null, null, order);
```

SQLiteDatabase

- La classe SQLiteDatabase espone i metodi **insert**, **delete** ed **update** che di fatto incapsulano i corrispondenti comandi SQL.
- In aggiunta si può utilizzare il metodo **execSQL** che permette di eseguire un qualsiasi comando SQL sulle tabelle del database.
- Ogni qual volta si modificano i valori di un database si può (dovrebbe) richiamare la funzione **refreshQuery** per ogni Cursor che si riferisce alle tabelle interessate dalle modifiche.

insert

- Per inserire una nuova riga si deve costruire un oggetto **ContentValues** ed utilizzare il suo metodo **put** per inserire un valore per ogni colonna. Quindi si richiama il metodo **insert** del database target passando l'oggetto **ContentValues** creato.

```
// Create a new row of values to insert.
ContentValues newValues = new ContentValues();

// Assign values for each row.
newValues.put(COLUMN_NAME, newValue);
[ ... Repeat for each column ... ]

// Insert the row into your table
myDatabase.insert(DATABASE_TABLE, null, newValues);
```

update

- Anche l'aggiornamento è realizzato con l'oggetto **ContentValues**; si crea quindi un oggetto **ContentValues** come visto per l'inserimento. Si richiama quindi il metodo **update** passando il nome della tabella, l'oggetto **ContentValues**, e la clausola **where**:

```
// Define the updated row content.
ContentValues updatedValues = new ContentValues();

// Assign values for each row.
newValues.put(COLUMN_NAME, newValue);
[ ... Repeat for each column ... ]

String where = KEY_ID + "=" + rowId;

// Update the row with the specified index with the new values.
myDatabase.update(DATABASE_TABLE, newValues, where, null);
```

delete

- Per cancellare una riga basta chiamare la funzione **delete** sul database interessato specificando la tabella e la clausola where:

```
myDatabase.delete(DATABASE_TABLE, KEY_ID + "=" + rowId, null);
```