Persistencia. SQLite

SQLite es el sistema de bases de datos nativo de Android. Se trata de un gestor de código abierto para base de datos relacionales, que cumple con todos los estándares y que además es extremadamente ligero. Otra de sus características es que guarda toda la base de datos en un único fichero. Es útil en aplicaciones pequeñas y en dispositivos embebidos con recursos limitados ya que no requiere ninguna instalación adicional. La explicación de la sintaxis SQL y de todas las características de SQLite se sale de los objetivos de este libro, ya que son lenguajes estándar ampliamente conocidos. Por lo que nos centremos únicamente en cómo utilizar desde Android este gestor de base de datos.

Crear, actualizar y conectar.

Android nos proporciona la clase abstracta `SQLiteOpenHelper` para ayudarnos con la creación, conexión y actualización de la base de datos. Puede parecer que esta es una tarea fácil, pero cuando trabajemos con una aplicación que ya esté en producción hay que tener mucho cuidado con la actualización de la base de datos. Si por ejemplo lanzamos una nueva versión que contiene más campos. ¿Cómo hacemos para que los usuarios que tenían la versión anterior se puedan actualizar sin perder sus datos? Gracias a esta clase podremos gestionar estos casos de una forma sencilla.

Para utilizarla tenemos que crear una clase que herede de `SQLiteOpenHelper`. Esto nos obligará a implementar los métodos `onCreate` y `onUpgrade`. Opcionalmente podemos sobrescribir el constructor o añadir el nuestro propio. A continuación, se muestra un ejemplo:

```
1. public class DBAccess extends SQLiteOpenHelper {
2.
3.
4.  public DBAccess(Context context, String name,
5. SQLiteDatabase.CursorFactory cursor, int version) {
6.
7.   super(context, name, cursor, version);
8.
9.  }
10.
11. @Override
12. public void onCreate(SQLiteDatabase sqLiteDatabase) {
13.
14.       String CREATE_DB = "CREATE TABLE ciudades ("
15.        + "id INTEGER PRIMARY KEY AUTOINCREMENT,"
16.        + " nombre TEXT)";
17.
18.  sqLiteDatabase.execSQL(CREATE_DB);
19.
20. }
21.
22. @Override
23. public void onUpgrade(SQLiteDatabase sqLiteDatabase,
24.int oldVersion, int newVersion) {
25.
26.  db.execSQL("DROP TABLE IF EXISTS ciudades");
27.  onCreate(db);
28.
29. }
30.
31.}
```

El constructor de esta clase lo único que hace es llamar al constructor base con el contexto, el nombre de la base de datos, un tercer parámetro para configurar el tipo de cursor (podemos indicar null para utilizar el cursor por defecto) y un último parámetro con la versión de la base de datos. Cuando se instancie esta clase, de forma automática se determinará si la base de datos (con el nombre que le hemos indicado) está creada o no. En caso de que no esté creada se llamará al método onCreate para crearle. En caso de que ya exista se comprobará la versión, y si es una versión anterior a la que hemos indicado en el constructor se llamará al método onUpgrade, mientras que, si tiene la misma versión, como no hay que actualizarla ni crearla no llamará a ninguno de los dos métodos, simplemente se abrirá la conexión.

El método `onCreate` recibe como parámetro una instancia de la clase `SQLiteDatabase` a través de la cual tendremos acceso a la base de datos para lectura y escritura. Esta clase dispone de multitud de métodos, podéis obtener información en la documentación oficial.

En caso de que introduzcamos algún cambio (añadamos una tabla, un campo a una tabla, cambiemos un tipo, etc.) deberíamos incrementar el número de versión para que se llame al método `onUpgrade` y así podamos actualizar los cambios. En el ejemplo anterior, el cambio de versión se implementa de una manera un poco drástica: borrar la tabla y volver a crearla. Pero esto no se debería hacer así porque se perderían todos los datos que tuviera el usuario. En su lugar sería más conveniente realizar únicamente las modificaciones necesarias sobre las tablas existentes, por ejemplo, añadir una columna, cambiar un tipo de dato, etc. Además, es importante que controlemos la versión en la que está la base de datos y la versión a la que se va a actualizar (ambos datos se reciben como parámetros en onUpgrade). Es común que se vayan añadiendo cambios de versión tras versión, y además también es posible que algún usuario tenga que aplicar varios cambios de versión seguidos. Por ejemplo, si un usuario no actualiza su aplicación desde la versión 1 y actualmente está en la 4, se le tendrían que aplicar las modificaciones de las versiones 2, 3 y 4 para que la base de datos de su aplicación esté completamente actualizada. A esta forma de controlar las versiones de una base de datos también se le conoce con el nombre de migraciones. Un ejemplo de cómo podríamos controlar esto sería:

```
1. @Override
2. public void onUpgrade(SQLiteDatabase sqLiteDatabase, int oldVersion,
3. int newVersion) {
4.
5.  switch (oldVersion) {
6.   case 1:
7.    db.execSQL("ALTER TABLE ciudades ADD COLUMN poblacion INTEGER");
8.   case 2:
9.    db.execSQL("ALTER TABLE ciudades ADD COLUMN superficie REAL");
10.  case 3:
11.   db.execSQL("ALTER TABLE ciudades ADD COLUMN gentilicio TEXT");
12.   break;
13.  default:
14.   throw new IllegalStateException("Version desconocida " + oldversion
   );
15.  }
16.}
```

Acceso a la base de datos

Para acceder a la base de datos solo necesitamos instanciar un objeto de la clase con los cuatros parámetros del constructor. A partir del objeto generado es necesario llamar a `getReadableDatabase()` o `getWritableDatabase()` para obtener una referencia a la base de datos de solo escritura o de escritura y lectura respectivamente. Los dos métodos devolverán un objeto del tipo `SQLiteDatabase` que nos permitirá lanzar nuestras consultas y operaciones.

```
1. String DBNAME = "mibasededatos";
2.
3. int DBVERSION = 1;
4.
5. MiOpenHelper oHelper = new MiOpenHelper(context, DBNAME, null,
6. DBVERSION);
7. SQLiteDatabase db = oHelper.getWritableDatabase();
8.
9. db.execSQL(" INSERT INTO ciudades (nombre) VALUES ('Alicante')");
```

Con este código sería suficiente para poder trabajar en las distintas actividades, pero sería mejor si separásemos el código con una capa de abstracción superior.

```
1. public class DataHelper {
2.
3.    private static final DBNAME = "mibasededatos";
4.
5.    private static final int DBVERSION = 1;
6.
7.    private SQLiteDatabase mDB;
8.
9.    public DataHelper(Context context) {
10.
11.       MiOpenHelper oHelper = new MiOpenHelper(context, DBNAME, null, D
   BVERSION);
12.
13.       mDB = oHelper.getWritableDatabase();
14.
15.   }
16.
17.   public List<String> selectAll(){  /*   ...   */};
18.
19.   public long insert(String nombre){  /*   ...   */};
20.
21.   public int deleteAll() {  /*   ...   */};
22.
23.   private static class MiOpenHelper extends SQLiteOpenHelper{
24.
25.       // [ .... Codigo de SQLiteOpenHelper .... ]
26.   }
27.}
```

Inserción de datos

Para insertar dato se puede usar el método `execSQL()` que ya hemos visto antes. El problema es que es propenso a fallos en la construcción o a inyección SQL.

La segunda alternativa es utilizar el método insert de la clase `SQLiteDatabase`, que de la misma forma que el método `query`, nos permitirá construir la SQL a partir de los parámetros del método. En este caso recibe solo tres parámetros: el nombre de la tabla, las columnas que pueden ser `nullables` y los valores a insertar, que los definiremos ayudándonos de la clase `ContentValues`. Como salida devuelve el identificador del nuevo registro o -1 en caso de error. A continuación, se incluye un ejemplo:

```
1. public int insert(String nombre){
2.          SQLiteDatabase db = this.getWritableDatabase();
3.
4.          ContentValues values = new ContentValues();
5.
6.           values.put("nombre", nombre);
7.
8.           return mDB.insert("ciudades", null, values);
9.
10.}
```

Consulta de datos

Para realizar consultas a la base de datos se encuentran dos métodos: `rawQuery` y `query`, ambas ejecutarán sobre la base de datos y nos devolverán un objeto de tipo `Cursor` apuntando a los resultados.

Cursor es un iterador que nos permite recorrer los resultados devueltos de forma secuencial. Para esto tenemos una serie de métodos, como `moveToFirst`() para mover el cursor al primer elemento y `moveToNext`() para mover el cursor al siguiente registro. Ambos métodos devuelven un booleano para indicar si existe o no elementos en la posición actual. Además, la clase cursor también incorpora una serie de métodos tipo `getXXX`(), donde XXX es el tipo de dato, para recuperar un valor del registro actual. Como cada registro puede tener muchas columnas, al llamar a `getXXX`() tenemos que indicar como parámetro el índice (empezando en cero) de la columna que queremos obtener. Opcionalmente podemos recuperar el índice de la columna mediante el método `getColumnIndexOrThrow(nombre_columna).` A continuación, se muestra un ejemplo de cómo podríamos realizar una consulta a la base de datos para recuperar los nombres de todas las ciudades y devolverlos en un `ArrayList:`

```
1.  public List<String> selectAll(){
2.
3.          List<String>  list = new ArrayList<String>();
4.          String QUERY = "SELECT * FROM " + DB_TABLE_NAME;
5.
6.          Cursor c = db.rawQuery(QUERY,null);
7.
8.          if(c.moveToFirst()){
9.
10.             do{
11.                list.add(cursor.getString(1); //Columna nombre
12.                //Tambien podríamos haber hecho...
13.                // list.add(cursor.getString(cursor.getColumnIndexOrThrow("nomb
    re")));
14.
15.             }while(c.moveToNext());
16.         }
17.
18.         if(c != null){
19.              c.close();
20.         }
21.
22.         return list;
23.
24. }
```

La otra opción comentada anteriormente es el uso del método query, quedando como sigue:

```
1. public List<String> selectAll(){
2.
3.          List<String> list = new ArrayList<String>();
4.         Cursor c = db.query("ciudades", null, null, null, null,
5.         null, null);
6.
7.          if(c.moveToFirst()){
8.
9.              do{
10.                list.add(cursor.getString(1); //Columna nombre
11.                //Tambien podríamos haber hecho...
12.                // list.add(cursor.getString(cursor.getColumnIndexOr
   Throw("nombre")));
13.
14.             }while(c.moveToNext());
15.         }
16.
17.         if(c != null){
18.              c.close();
19.         }
20.
21.         return list;
22.
23.}
```

Este método tiene siete parámetros diferentes:

- `Table`: Nombre de la tabla a consultar.
- `Columns`: Array con los nombres de las columnas que queremos obtener. Si indicamos `null` se devolverán todas las columnas.
- Selection: En este parámetro podemos indicar una cadena con la cláusula WHERE de la consulta. Los argumentos variables de esta cláusula los tendremos que marcar con el símbolo ?.
- `SelectionArgs`[]. Array con los valores que se usarán para reemplazar las incógnitas (marcadas con ?).
- `groupBy`: Permite indicar la cláusula GROUP BY.
- `Having`: Permite establecer la sentencia HAVING
- `orderBy`: Permite indicar el orden.

¿Dónde se guarda la base de datos?

SQLite guarda la base de datos completa, con todas las tablas en un fichero con el nombre que le hayamos asignado en el constructor de la clase SQLiteOpenHelper. Pero ¿dónde podemos encontrar este fichero= Por defecto Android lo guarda junto a los datos de las aplicaciones en la siguiente ruta:

```
/data/data/<nombre-del-paquete>/databases/<nombre-de-la-bd>
```

Para explorar su contenido tenemos varias opciones. Podemos utilizar la herramienta DDMS(Dalvik Debug Monitor Server), ir a la pestaña *"File Explorer*", buscar la ruta correspondiente de la base de datos y descargar el fichero a nuestro ordenador usando el botón *Pull file from the device* . Una vez tenemos una copia local ya podemos abrirla usando cualquier herramienta externa.