IT ШКОЛА SAMSUNG

Курс обучения преподавателей

Работа с БД в Android

Методические материалы

Варакин Михаил

Август 2015

Оглавление

Работа с базами данных в Android

Механизм работы с базами данных в Android позволяет хранить и обрабатывать структурированную информацию. Любое приложение может создавать свои собственные базы данных, над которыми оно будет иметь полный контроль.
В Android используется библиотека *SQLite*, представляющую из себя реляционную СУБД, обладающую следующими характерными особенностями: свободно распространяемая (open source), поддерживающая стандартный язык запросов и транзакции, легковесная, одноуровневая (встраиваемая), отказоустойчивая.

Курсоры (Cursor) и ContentValues

Запросы к СУБД возвращают объекты типа *Cursor*. Для экономии ресурсов используется подход, когда при извлечении данных не возвращаются копии их значений из СУБД, а создается *Cursor*, обеспечивающий навигацию и доступ к запрошенному набору исходных данных. Методы объекта Cursor предоставляют различные возможности навигации, назначение которых, как правило, понятно из названия:

- `moveToFirst()`
- `moveToNext()`
- `moveToPrevious()`
- `getCount()`
- `getColumnIndexOrThrow()`
- `getColumnName()`
- `getColumnNames()`
- `moveToPosition()`
- `getPosition()`

При добавлении данных в таблицы СУБД применяются объекты класса *ContentValues*. Каждый такой объект содержит данные одной строки в таблице и, по сути, является ассоциативным массивом с именами столбцов и соответствующими значениями.

Работа с СУБД SQLite

При создании приложений, использующих СУБД, во многих случаях удобно применять инструменты, называющиеся ORM (Object-Relationship Mapping), отображающие данные из одной или нескольких таблиц на объекты используемого языка программирования. Кроме того, ORM позволяют абстрагироваться от конкретной реализации и структуры таблиц и берут на себя обязанности по взаимодействию с СУБД. К сожалению, в силу ограниченности ресурсов мобильной платформы ORM в настоящий момент в Android практически не применяется. Тем не менее, разумным подходом при разработке

приложения будет инкапсуляция всех взаимодействий с СУБД в одном классе, методы которого будут предоставлять необходимые услуги остальным компонентам приложения. В простых случаях для этих целей можно расширить класс *CursorWrapper*, добавив в него методы для манипуляции объектами нужного типа.

Также хорошей практикой является создание вспомогательного класса, берущего на себя работу с СУБД. Данный класс обычно инкапсулирует взаимодействия с базой данных, предоставляя интуитивно понятный строго типизированный способ удаления, добавления и изменения объектов. Такой *Адаптер базы данных* также должен обрабатывать запросы к БД и переопределять методы для открытия, закрытия и создания базы данных. Его также обычно используют как удобное место для хранения статических констант, относящихся к базе данных, таких, например, как имена таблиц и полей. Для управления состоянием «соединения» с БД и обработки изменения её состояния (например, когда при развитии приложения требуется соответственно модернизировать уже существующие БД без потери хранящихся данных) обычно применяется класс *SQLiteOpenHelper* или его наследники:

```
class DbOpenHelper extends SQLiteOpenHelper {
    static final String DB_TABLE = "notes";
    static final String COLUMN_NOTE = "note";
    private static final String DB_NAME = "notes.db";
    private static final int DB_VERSION = 1;
    private static final String DB_CREATE_TABLE = "CREATE TABLE " + DB_TABLE +
            " ( _id INTEGER PRIMARY KEY AUTOINCREMENT, "
            + COLUMN_NOTE + " TEXT NOT NULL);";

    public DbOpenHelper() {
        super(MainActivity.this, DB_NAME, null, DB_VERSION);
    }

    @Override
    public void onCreate(SQLiteDatabase db) {
        db.execSQL(DB_CREATE_TABLE);
        ContentValues cv = new ContentValues(1);
    }

    @Override
    public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) {
    }
}
```

Открытие и закрытие БД

Перед началом работы с БД её необходимо открыть, обычно с помощью метода *getWritableDatabase()* класса *SQLiteOpenHelper*:

```
mDb = mHelper.getWritableDatabase();
```

Данный метод может вызывать исключение *SQLiteException* в случае невозможности открыть БД. В этом случае можно попытаться использовать метод *getReadableDatabase(),* который откроет БД без возможности изменения (или, в свою очередь, тоже вызовет исключение *SQLiteException*). Полученный объект БД будет валидным до тех пор, пока для него явно не будет вызван метод *close()*. Стоит заметить, что при открытой БД методы *getWritableDatabase()* и *getReadableDatabase()* возвращают закэшированный объект, так что их вызовы обходятся очень дешево. В то же время, при необходимости сделать обновление БД (с помощью метода *onUpgrade()* класса *SQLiteOpenHelper*) методы *get*Database()* могут работать достаточно долго, так что при наличии такой опасности БД нельзя открывать в главном потоке (*main thread*), в том числе и внутри контент-провайдера.

Особенности работы с БД в Android

При работе с базами данных в Android *следует избегать* хранения *BLOB*'ов с таблицах из-за резкого падения эффективности работы.
Как показано в примере адаптера БД, для каждой таблицы *рекомендуется* создавать автоикрементное поле **_id**, которое будет уникальным индексом для строк. Если же планируется делегировать доступ к БД с помощью контент-провайдеров, такое поле является *обязательным*.

Для экономии ресурсов устройства стоит рассмотреть возможности синхронизации открытия и закрытия БД с жизненным циклом активностей (как правило, это методы *onStart()/onStop()*). Также необходимо помнить, что в реальном приложении операции с БД могут выполняться достаточно долго, так что выполнять в главном потоке их также крайне не рекомендуется.

Выполнение запросов для доступа к данным

Для повышения эффективности использования ресурсов мобильной платформы запросы к БД возвращают объект типа *Cursor*, используемый в дальнейшем для навигации и получения значений полей. Выполнение запросов осуществляется с помощью метода *query* экземпляра БД, параметры которого позволяют гибко управлять критериями выборки:

```
Cursor cursor = mDb.query(DbOpenHelper.DB_TABLE, null, null, null, null,
      null, "_id DESC");
```

Доступ к результатам с помощью курсора

Для получения результатов запроса необходимо установить курсор на нужную строку с помощью методов вида moveToМестоположение, перечисленных выше. После этого используются типизированные методы *getTun*, получающие

в качестве параметра индекс (номер) поля в строке. Этот индекс, в свою очередь, можно получить от курсора с помощью методов *getColumnIndex()* или *getColumnIndexOrThrow()*, параметром для которых служит имя поля, индекс которого нужно получить:

```
cursor.getColumnIndexOrThrow(DbOpenHelper.COLUMN_NOTE);
```

Изменение данных в БД

В классе *SQLiteDatabase*, содержащем методы для работы с БД, имеются методы *insert()*, *update()* и *delete()*, которые инкапсулируют операторы SQL, необходимые для выполнения соответствующих действий. Кроме того, метод *execSQL()* позволяет выполнить любой допустимый код SQL, не возвращающий результатов (а метод *rawQuery()* – любой запрос).

Вставка строк

Метод *insert()* хочет получать (кроме других параметров) объект *ContentValues*, содержащий значения полей вставляемой строки и возвращает значение индекса:

```
ContentValues cv = new ContentValues(1);
for (int i = 0; i < 10; i++) {
      cv.put(COLUMN_NOTE, "Note Number " + i);
      db.insert(DB_TABLE, null, cv);
}
```

Обновление строк

Также используется *ContentValues*, содержащий подлежащие изменению поля, а для указания, какие именно строки нужно изменить, используется параметр *where*, имеющий стандартный для SQL вид:

```
ContentValues row = new ContentValues(1);
row.put(DbOpenHelper.COLUMN_NOTE, noteText);
String where = "_id = " + mNoteId;
mDb.update(DbOpenHelper.DB_TABLE, row, where, null);
```

Метод *update* возвращает количество измененных строк. Если в качестве параметра *where* передать *null*, будут изменены все строки таблицы.

Удаление строк

Выполняет похожим на *update* образом:

```
mDb.delete(DbOpenHelper.DB_TABLE, "_id = " + id, null);
```

Использование SimpleCursorAdapter

Для отображения результатов запросов к БД обычно используется *ListView*, для работы которого необходим какой-либо *Adapter*. Класс *SimpleCursorAdapter* позволяет привязать *курсор* к *ListView*. используя описание разметки для отображения строк и полей. Для однозначного определения того, в каких элементах разметки какие поля, получаемые через курсор, следует отображать, используются два массива: строковый с именами полей строк, и целочисленный с идентификаторами элементов разметки:

```
	SimpleCursorAdapter mAdapter;

	void showNotes() {
		Cursor cursor = mDb.query(DbOpenHelper.DB_TABLE, null, null, null,
			null, null, "_id DESC");
		if (mAdapter == null) {
			mAdapter = new SimpleCursorAdapter(this, R.layout.item_view,
				cursor, FROM, TO, 0);
		} else {
			mAdapter.swapCursor(cursor);
		}
	}
```

При наполнении *ListView* содержимым адаптер добавляет к каждому элементу списка (т. е. *ListView*) дополнительную информацию, позволяющую определить, например, позицию элемента в списке/массиве (для *ArrayAdapter*) и первичный ключ (то самое поле **_id**) **CursorAdapter*. Эта информация обычно используется, когда пользователь взаимодействует с конкретным элементом списка, например, с помощью контекстного меню:

```
	@Override
	public boolean onContextItemSelected(MenuItem item) {
		AdapterView.AdapterContextMenuInfo info;
		switch (item.getItemId()) {
			case R.id.action_delete:
				info = (AdapterView.AdapterContextMenuInfo)
					item.getMenuInfo();
				deleteNote(info.id);
				return true;

			case R.id.action_edit:
				info = (AdapterView.AdapterContextMenuInfo)
					item.getMenuInfo();
				CharSequence text = ((TextView) ((LinearLayout)
					info.targetView).getChildAt(0)).getText();
				editNote(info.id, text);
				return true;

			default:
				return super.onContextItemSelected(item);
		}
	}
```

Транзакции

Несмотря на поддержку транзакций в SQLite, из-за специфики архитектуры в мобильных приложениях транзакции применяются достаточно редко. Чаще всего они используются для ускорения массового добавления записей (*bulk insert*):

```
mDb.beginTransaction();
// There will be many inserts here. . .
mDb.setTransactionSuccessful();
mDb.endTransaction();
```

Без явного объявления начала (*beginTransaction()*) транзакции, тем не менее, выполняются – в режиме автозавершения (*autocomplete*), а *beginTransaction()* отключает этот режим. Для *указания* успешного завершения транзакции (*commit*) используется метод *setTransactionSuccesful()*, если необходимо транзакцию отменить (*rollback*), его просто не применяют. Метод *endTransaction()* завершает транзакцию.

Лабораторная работа «работа с SQLite»

Целью работы является создание простого приложения, позволяющего создавать, хранить, редактировать и удалять короткие заметки. Для простоты использования все действия с записями будут производиться в рамках одной Активности, поэтому реализуйте максимально упрощенный интерфейс, используя для редактирования новый фрагмент внутри одной активности.