

№10 Хранение информации. SQLite, адаптеры, AsyncTask. ContentProvider

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.

SQLite доступен на любом Android-устройстве, его не нужно устанавливать отдельно. SQLite представляет из себя реляционную СУБД, обладающую следующими характерными особенностями: свободно распространяемая (open source), поддерживающая стандартный язык запросов и транзакции, легковесная, одноуровневая (встраиваемая), отказоустойчивая.

Для работы большинства систем управления базами данных необходим специальный процесс сервера базы данных. SQLite обходится без сервера; база данных SQLite представляет собой обычный файл. Когда база данных не используется, она не расходует процессорное время. Это особенно важно на мобильных устройствах, чтобы избежать разрядки аккумулятора

С базой данных взаимодействует только наше приложение, поэтому можно обойтись без идентификации с именем пользователя и паролем.

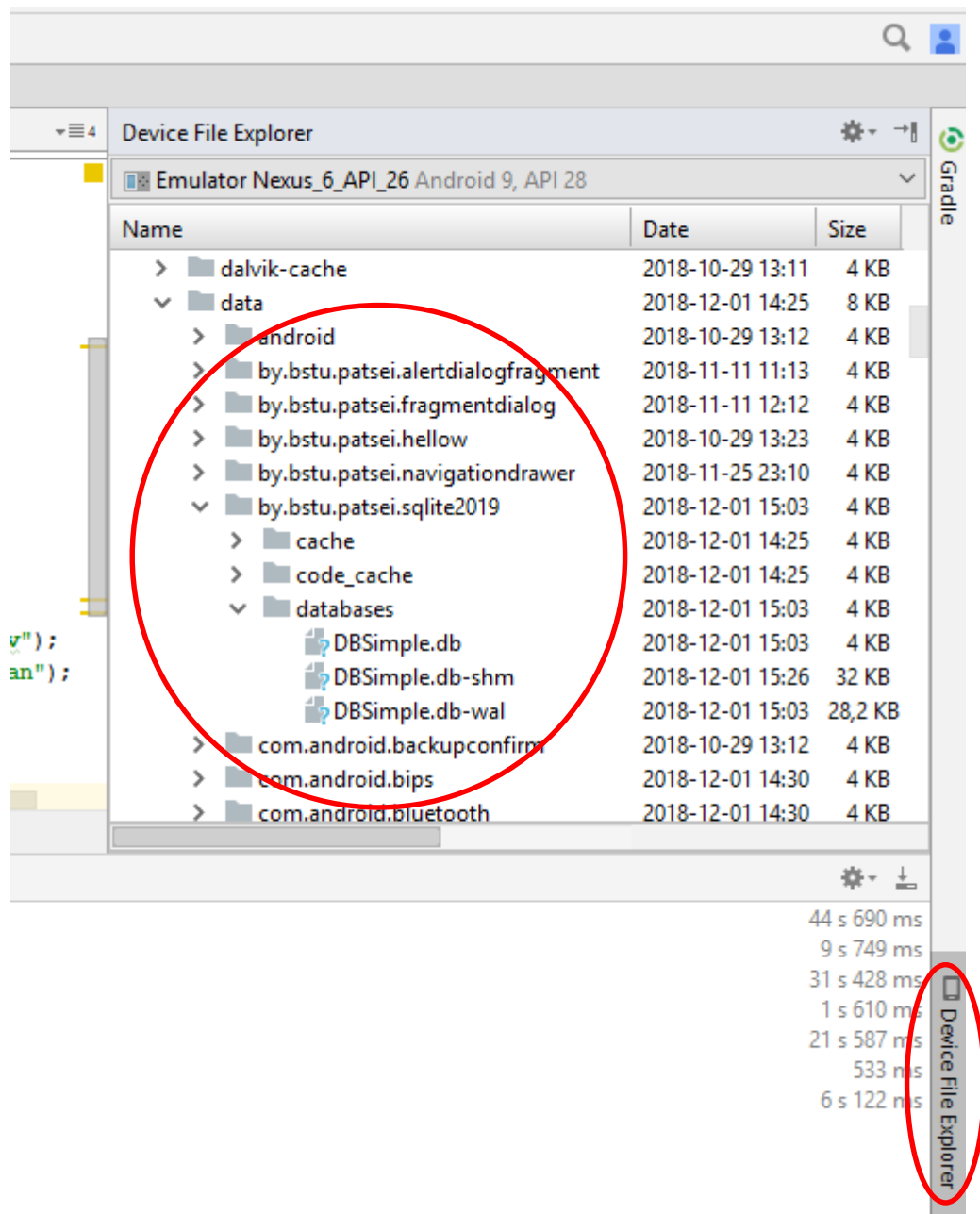
Хранение

Android хранит базы данных в каталоге `/data/data/<имя_вашего_пакета>/databases` на устройстве, на эмуляторе, путь может отличаться. Метод **`Environment.getDataDirectory()`** возвращает путь к каталогу **DATA**.

По умолчанию все базы данных закрытые, доступ к ним могут получить только те приложения, которые их создали.

Каждая база данных состоит из двух файлов. Имя первого файла базы данных соответствует имени базы данных. Это основной файл баз данных SQLite, в нём хранятся все данные. Вы будете создавать его программно. Второй файл — файл журнала. В файле журнала хранится информация обо всех изменениях, внесенных в базу данных. Если в работе с данными возникнет проблема, Android использует данные журнала для отмены (или отката) последних изменений.

Для просмотра можете воспользоваться Device File Explorer.



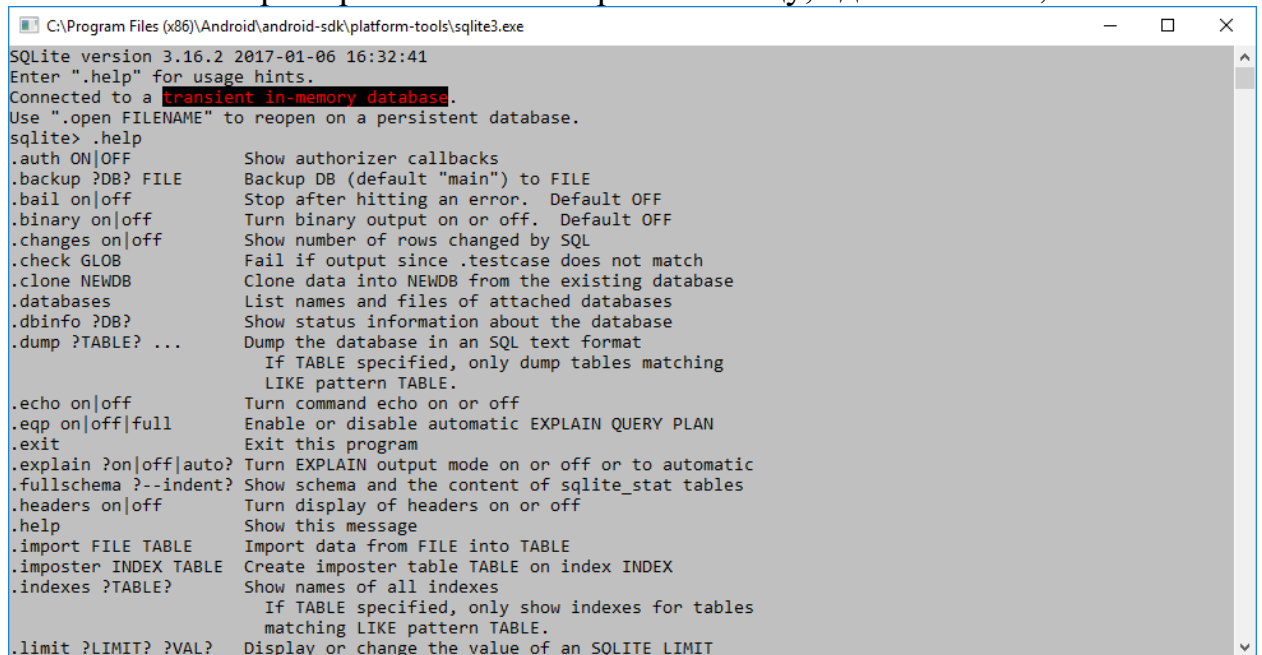
Программное обеспечение для работы с бд

В Android SDK включена оболочка `sqlite3.exe` для работы с командной строки.

Android\android-sdk\platform-tools\

```
C:\Program Files (x86)\Android\android-sdk\platform-tools\sqlite3.exe
SQLite version 3.16.2 2017-01-06 16:32:41
Enter ".help" for usage hints.
Connected to a transient in-memory database.
Use ".open FILENAME" to reopen on a persistent database.
sqlite>
```

Вызовите команду `.help` и посмотрите все возможности работы с оболочкой. Например можно посмотреть таблицу, сделать `select`,



```
C:\Program Files (x86)\Android\android-sdk\platform-tools\sqlite3.exe
SQLite version 3.16.2 2017-01-06 16:32:41
Enter ".help" for usage hints.
Connected to a transient in-memory database.
Use ".open FILENAME" to reopen on a persistent database.
sqlite> .help
.auth ON|OFF          Show authorizer callbacks
.backup ?DB? FILE      Backup DB (default "main") to FILE
.bail on|off           Stop after hitting an error.  Default OFF
.binary on|off         Turn binary output on or off.  Default OFF
.changes on|off        Show number of rows changed by SQL
.check GLOB            Fail if output since .testcase does not match
.clone NEWDB           Clone data into NEWDB from the existing database
.databases             List names and files of attached databases
.dbinfo ?DB?          Show status information about the database
.dump ?TABLE? ...      Dump the database in an SQL text format
                        If TABLE specified, only dump tables matching
                        LIKE pattern TABLE.
.echo on|off           Turn command echo on or off
.eqp on|off|full       Enable or disable automatic EXPLAIN QUERY PLAN
.exit                 Exit this program
.explain ?on|off|auto? Turn EXPLAIN output mode on or off or to automatic
.fullschema ?--indent? Show schema and the content of sqlite_stat tables
.headers on|off        Turn display of headers on or off
.help                 Show this message
.import FILE TABLE    Import data from FILE into TABLE
.imposter INDEX TABLE Create imposter table TABLE on index INDEX
.indexes ?TABLE?       Show names of all indexes
                        If TABLE specified, only show indexes for tables
                        matching LIKE pattern TABLE.
.limit ?LIMIT? ?VAL?   Display or change the value of an SQLITE_LIMIT
```

Существуют также дополнительные инструменты для работы с бд :

DbBrowser for SQLite
SQLite Expert Professional
SQLite Studio

Определение схемы и контракта

При работе с базой данных принято создавать новый пакет **data** внутри основного пакета.

Google рекомендует создавать класс-контракт. Будем придерживаться этого правила. Мы как бы подписываем контракт на работу с базой данных и предоставляем все нужные данные

Одним из основных принципов баз данных SQL является схема: формальное объявление о том, как организована база данных. Схема отражена в операторах SQL, которые вы используете для создания своей базы данных.

Класс контракта - это контейнер для констант, который определяет имена для *URI, таблиц и столбцов*. Класс контракта позволяет использовать одни и те же константы для всех других классов в одном пакете. Это позволяет вам изменять имя столбца в одном месте и распространять его во всем коде.

Хороший способ организовать контрактный класс - поставить определения, которые являются глобальными для всей вашей базы данных на корневом уровне класса. Затем создайте внутренний класс для каждой таблицы. Каждый внутренний класс перечисляет столбцы соответствующей таблицы.

```

public final class DBContract {

    public DBContract() {}

    /* Внутренний класс определяет контент таблицы */
    public static abstract class DBEntry implements BaseColumns {
        public static final String TABLE_NAME = "student";
        public static final String COLUMN_NAME_NAME = "name";
        public static final String COLUMN_NAME_INFO = "info";
        public static final String COLUMN_NAME_RATE = "rate";
    }
}

```

В этом фрагменте кода определяются имя таблицы и имена столбцов для одной таблицы. В классе используется реализация интерфейса **BaseColumn**:

```

public static final class DBEntry implements implements BaseColumns {

```

В большинстве случаев работа с базой данных происходит через специальные объекты **Cursor**, которые требуют наличия в таблице колонки с именем **_id**. Вы можете создать столбец вручную в коде, а можно положиться на **BaseColumn**, который создаст столбец с нужным именем автоматически. Если вы не будете работать с курсорами, то можете использовать и стандартное наименование **id** или вообще не использовать данный столбец. Но это не желательно.

Классы для работы с SQLite

Система Android включает набор классов для управления базой данных SQLite. Основная часть этой работы выполняется тремя типами объектов.

Помощник SQLite Помощник SQLite создается расширением класса **SQLiteOpenHelper**. Он предоставляет средства для создания и управления базами данных. **SQLiteOpenHelper** содержит два обязательных абстрактных метода:

- **onCreate()** — вызывается при первом создании базы данных
- **onUpgrade()** — вызывается при модификации базы данных

Также используются другие методы класса:

- **onDowngrade(SQLiteDatabase, int, int)**
- **onOpen(SQLiteDatabase)**
- **getReadableDatabase()**
- **getWritableDatabase()**

В приложении необходимо создать собственный класс, наследуемый от **SQLiteOpenHelper**. В этом классе необходимо реализовать указанные обязательные методы, описав в них логику создания и модификации вашей базы.

Класс **SQLiteDatabase** предоставляет доступ к базе данных. Его можно сравнить с классом **SQLConnection** в JDBC. Он позволяет выполнять запросы к бд, выполнять с ней различные манипуляции. Например, есть следующие методы:

- **query()** чтение данных
- **insert()** вставка записей
- **delete()** удаление записей
- **update()** обновление записей
- **execSQL()** выполнение допустимого кода SQL
- **rawQuery()** выполнение «сырого» запроса

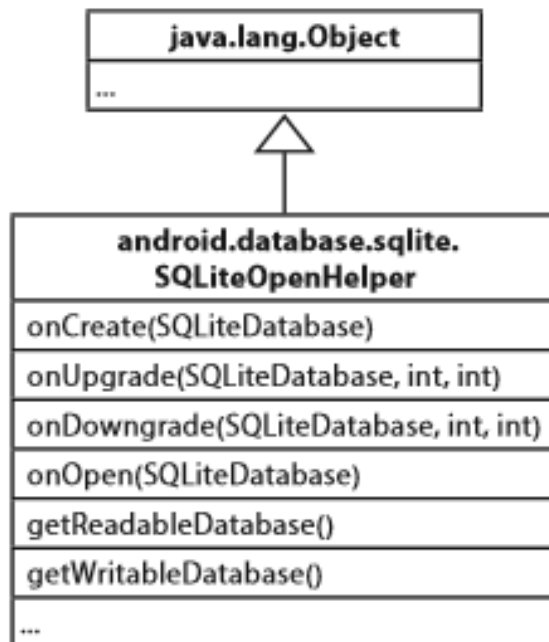
Класс **SQLiteCursor** предназначен для чтения и записи в базу данных. Его можно сравнить с классом **ResultSet** в JDBC. Вместо того чтобы извлекать данные и возвращать копию значений, курсоры ссылаются на результирующий набор исходных данных. Курсоры позволяют управлять текущей позицией (строкой) в результирующем наборе данных, возвращаемом при запросе

Класс **SQLiteQueryBuilder** позволяет создавать SQL-запросы.

Сами sql-выражения представлены классом **SQLiteStatement**, которые позволяют с помощью плейсхолдеров вставлять в выражения динамические данные.

Класс **SQLiteOpenHelper**

Класс **SQLiteOpenHelper** упрощает задачи создания и сопровождения баз данных. Считайте, что это своего рода личный ассистент, который берет на себя служебные операции по управлению базами данных. Рассмотрим некоторые типичные задачи, в решении которых вам поспособствует помощник SQLite.



Создание базы данных При первой установке приложения файл базы данных не существует. Помощник SQLite проследит за тем, чтобы файл базы данных был создан с правильным именем и с правильной структурой таблиц.

Обеспечение доступа к базе данных Нашему приложению не обязательно знать все подробности о том, где хранится файл базы данных. Помощник SQLite предоставляет удобный объект, представляющий базу данных, и приложение работает с базой через этот объект — тогда, когда сочтет нужным

Сопровождение баз данных Может случиться так, что структура базы данных изменится со временем. Положитесь на помощника SQLite — он преобразует старую версию в новую, с учетом самых последних изменений в структуре базы данных

Далее можно создать выражения для обслуживания - создания и удаления:

```

private static final String SQL_CREATE_ENTRIES =
    "CREATE TABLE " + DBEntry.TABLE_NAME + " (" +
        DBEntry._ID + " INTEGER PRIMARY KEY," +
        DBEntry.COLUMN_NAME_NAME + "TEXT" + "," +
        DBEntry.COLUMN_NAME_INFO + "TEXT" + "," +
        DBEntry.COLUMN_NAME_RATE + "INTEGER) ";

private static final String SQL_DELETE_ENTRIES =
    "DROP TABLE IF EXISTS " + DBEntry.TABLE_NAME;

```

Ну и собственно класс самого хэлпера

```

public class DbHelper extends SQLiteOpenHelper {

    public static final int DATABASE_VERSION = 2;
    public static final String DATABASE_NAME = "DBSimple.db";

    public DbHelper(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) {
        db.execSQL(SQL_DELETE_ENTRIES);
        onCreate(db);
    }
}

```

Как вы уже догадались, константа **DATABASE_NAME** отвечает за имя файла, в котором будет храниться база данных приложения. Можно придумать любое имя и обойтись без расширения.

Вторая константа **DATABASE_VERSION** требует дополнительных объяснений. Она отвечает за номер версии базы, начинается с 1. Принцип её работы схож с номером версий самого приложения. Как только программа заметила обновление номера базы, она запускает метод **onUpgrade()**, который у нас сформировался автоматически. В этом методе размещен код, который должен сработать при обновлении базы и обновить структуру базы данных у старых пользователей.

При первой установке приложения базы данных ещё не существует. Тут проверять пока нечего. При установке новой версии приложения система проверит номер версии базы данных. Если он больше, чем было, то вызовется метод **onUpgrade()**. Если наоборот, то вызовется метод **onDowngrade()** (обычно так происходит редко).

Метод **onCreate()** вопросов не вызывает - здесь создаётся сама база данных с необходимыми данными для работы.

Метод вызывается, если в устройстве нет базы данных и наш класс должен создать его. У метода есть параметр **db**, который относится к классу **SQLiteDatabase**. У класса есть специальный метод **execSQL()**, которому нужно передать запрос (SQL-скрипт) для создания таблицы.

Вы также можете реализовать методы **onDowngrade ()** или **onOpen ()**, но они не требуются.

Чтобы использовать реализацию вспомогательного класса, создайте новый экземпляр, передайте его конструктору контекст, можно имя базы данных, можно текущую версию. Вызовите метод **getReadableDatabase()** или

getWritableDatabase(), чтобы открыть и вернуть экземпляр базы данных, с которой мы имеем дело (он будет доступен как для чтения, так и для записи). Вызов метода **getWritableDatabase()** может завершиться неудачно из-за проблем с полномочиями или нехваткой места на диске, поэтому лучше предусмотреть откат к методу **getReadableDatabase()**.

Если база данных не существует, вспомогательный класс вызывает свой обработчик **onCreate()**; если версия базы данных изменилась, вызовется обработчик **onUpgrade()**. В любом случае вызов методов **getWritableDatabase/getReadableDatabase**, в зависимости от ситуации, вернет существующую, только что созданную или обновленную базу данных.

Оба метода имеют разные названия, но возвращают один и тот же объект. Только метод для чтения **getReadableDatabase()** сначала проверит доступность на чтение/запись. В случае ошибки метод проверит доступность на чтение и только потом уже вызовет исключение. Второй метод сразу проверяет доступ к чтению/записи и вызывает исключение при отказе в доступе.

Если вы планируете использовать несколько таблиц, то рекомендуется создавать для каждой таблицы отдельный класс.

Типы данных SQLite

В SQLite применяется следующая система типов данных:

- **NULL** (пустое значение)
- **INTEGER**: представляет целое число, аналог типу `int` в java
- **REAL**: представляет число с плавающей точкой, аналог `float` и `double` в java
- **TEXT**: представляет набор символов (UTF-8, UTF-16), аналог `String` и `char` в java
- **NUMERIC**
- **BLOB**: представляет массив бинарных данных, например, изображение (При работе с базами данных в Android следует избегать хранения BLOB'ов с таблицами из-за резкого падения эффективности работы)

Сохраняемые данные должны представлять соответствующие типы в java с конвертацией и без.

SQLite сама по себе не проверяет типы данных, поэтому вы можете записать целое число в колонку, предназначенную для строк, и наоборот.

В отличие от многих систем баз данных, в SQLite не нужно указывать размер столбца. Во внутренней реализации тип данных преобразуется в намного более универсальный класс хранения. Это означает, что вы можете в общих чертах указать, какие данные собираетесь хранить, но не обязаны указывать их конкретный размер

Размещение информации в базе данных (вставка)

1) Создаем экземпляр

```
DbHelper dbHelper = new DbHelper(this);
```

2) Вставляем данные в бд через ContentValues

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.

Поэтому для вставки сначала подготавливаются данные с помощью класса **ContentValues**. Вы указываете имя колонки таблицы и значение для неё, т.е. работает по принципу "ключ-значение".

Когда подготовите все данные во все столбцы, то вызывайте метод **insert()**, который сразу раскидает данные по столбцам.

Способ добен, требует мало кода и легко читаем. Вы создаёте экземпляр класса, а затем с помощью метода **put()** записываете в нужную колонку нужные данные. После чего вызывается метод **insert()**, который помещает подготовленные данные в таблицу.

У метода **insert()** три аргумента. В первом указывается имя таблицы, в которую будут добавляться записи. В третьем указывается объект **ContentValues**, созданный ранее. Вторым аргументом используется для указания колонки. SQL не позволяет вставлять пустую запись, и если будет использоваться пустой **ContentValue**, то укажите во втором аргументе **null** во избежание ошибки.

Итак:

- **table** — имя таблицы, в которую будет вставлена запись;
- **nullColumnHack** — в базе данных SQLite не разрешается вставлять полностью пустую строку, и если строка, полученная от клиента контент-провайдера, будет пустой, то только этому столбцу явно будет назначено значение **null**;
- **values** — карта отображений (класс **Map** и его наследники), передаваемая клиентом контент-провайдера, которая содержит пары ключ-значение. Ключи в карте должны быть названиями столбцов таблицы, значения — вставляемыми данными.

```

// получаем репозиторий в режиме записи
SQLiteDatabase db = mDbHelper.getWritableDatabase();

// создаем новую запись
ContentValues values = new ContentValues();
values.put(DBContract.DBEntry.COLUMN_NAME_NAME, "Ivanov");
values.put(DBContract.DBEntry.COLUMN_NAME_INFO, "BSTU");
values.put(DBContract.DBEntry.COLUMN_NAME_RATE, "8");

// вставляем, возвращает primary key
long newRowId;
newRowId = db.insert(
    DBContract.DBEntry.TABLE_NAME,
    null,
    values);

```

Метод **insert()** возвращает идентификатор `_id` вставленной строки или `-1` в случае ошибки.

Второй способ. SQL-запрос

Существует также другой способ вставки через метод **execSQL()**, когда подготавливается нужная строка и запускается скрипт.

В этом варианте используется традиционный SQL-запрос **INSERT INTO....** Основное неудобство при этом способе - не запутаться в кавычках. Если что-то не вставляется, то смотрите логи сообщений.

Изменение данных

Если запись уже существует, но вам нужно изменить какое-то значение, то вместо **insert()** используйте метод **update()**. В остальном принцип тот же.

```

SQLiteDatabase db = mDbHelper.getWritableDatabase();

// Новое значение
String title = "Murkov";
ContentValues values = new ContentValues();
values.put(DBEntry.COLUMN_NAME_NAME, title);

// Столбец, коорый надо обновлять
String selection = DBEntry.COLUMN_NAME_NAME + " LIKE ?";
String[] selectionArgs = { "Ivanov" };

int count = db.update(
    DbHelper.DBEntry.TABLE_NAME,
    values,

```

```
selection,  
selectionArgs);
```

Чтение данных

Считывать данные также можно двумя способами. В любом случае результат возвращается в виде объекта **Cursor**.

Методы объекта **Cursor** предоставляют различные возможности навигации, назначение которых, как правило, понятно из названия:

- *moveToFirst*
- *moveToNext*
- *moveToPrevious*
- *getCount*
- *getColumnIndexOrThrow*
- *getColumnName*
- *getColumnNames*
- *moveToPosition*
- *getPosition*

Первый способ. Метод query()

Для чтения из базы данных используйте метод **query()** с передачей критериев выделения и желаемых столбцов. Метод сочетает элементы **insert()** и **update()**, за исключением того, что список столбцов определяет данные, которые вы хотите получить, а не данные для вставки..

```

    // определяем названия столбцов
    // которые нужны для выполнения запроса
String[] projection = {
    BaseColumns._ID,
    DBEntry.COLUMN_NAME_NAME,
    DBEntry.COLUMN_NAME_INFO,
    ...
};

String selection = DBEntry.COLUMN_NAME_NAME + " = ?";
String[] selectionArgs = { "Ivanov" };

    // Возвращается Cursor
String sortOrder =
    DBEntry.COLUMN_NAME_INFO + " DESC";

Cursor cursor = db.query(
    DBEntry.TABLE_NAME, // имя таблицы
    projection,           // столбцы
    selection,             // столбцы для WHERE
    selectionArgs,        // значения для WHERE
    null,                 // не группировать строки
    null,                 // не фильтровать
    sortOrder             // порядок сортировки
);

```

```

Cursor query (String table,
    String[] columns,
    String selection,
    String[] selectionArgs,
    String groupBy,
    String having,
    String sortOrder)

```

В метод **query()** передают семь параметров. Если какой-то параметр для запроса вас не интересует, то оставляете *null*:

- **table** — имя таблицы, к которой передается запрос;
- **String[] columnNames** — список имен возвращаемых полей (массив). При передаче *null* возвращаются все столбцы;
- **String whereClause** — параметр, формирующий выражение WHERE (исключая сам оператор WHERE). Значение *null* возвращает все строки. Например: *_id = 19 and summary = ?*
- **String[] selectionArgs** — значения аргументов фильтра. Вы можете включить ? в "whereClause". Подставляется в запрос из заданного массива;
- **String[] groupBy** - фильтр для группировки, формирующий выражение GROUP BY (исключая сам оператор GROUP BY). Если GROUP BY не нужен, передается *null*;

- **String[] having** — фильтр для группировки, формирующий выражение HAVING (исключая сам оператор HAVING). Если не нужен, передается null;
- **String[] orderBy** — параметр, формирующий выражение ORDER BY (исключая сам оператор ORDER BY). При сортировке по умолчанию передается null.

Чтобы получить все записи из нужных столбцов без условий, достаточно указать имя таблицы в первом параметре и строчный массив во втором. В остальных параметрах оставляем null.

Чтобы посмотреть на строку в месте курсора, используйте один из методов перемещения **Cursor**, которые всегда нужно вызывать перед считыванием значений. Обычно следует начинать с вызова **moveToFirst()**, который помещает "позицию чтения" на первую запись в результатах.

После завершения работы с курсором мы закрываем все связанные объекты:

```
List itemIds = new ArrayList<>();
while(cursor.moveToNext()) {
    long itemId = cursor.getLong(
        cursor.getColumnIndexOrThrow(DBEntry._ID));
    itemIds.add(itemId);
}
cursor.close();
```

Для перемещения между записями в курсорах используются четыре основных метода: **moveToFirst()**, **moveToLast()**, **moveToPrevious()** и **moveToNext()**.

Для чтения данных из текущей записи курсора используются методы **get*()**. Точное название метода зависит от типа значения, которое вы собираетесь прочитать. Например, метод **getString()** возвращает значение столбца в формате String, а метод **getInt()** возвращает значение столбца в формате int. Каждый из методов получает один параметр — индекс столбца.

```
String name = cursor.getString(0);
```

Второй способ. Метод rawQuery()

Второй способ использует сырой (raw) SQL-запрос. Сначала формируется строка запроса и отдаётся методу **rawQuery()**.

Удаление данных

Метод **delete()** класса **SQLiteDatabase** работает по тому же принципу, как и метод **update()**.

```
// Определение 'where' части запроса
String selection = DBEntry.COLUMN_NAME_NAME + " LIKE ?";
// Определение аргументов в placeholder
String[] selectionArgs = { "Ivanov" };
// SQL statement.
int deletedRows = db.delete(DBEntry.TABLE_NAME, selection,
selectionArgs);
```

Возвращает количество удаленных столбцов.

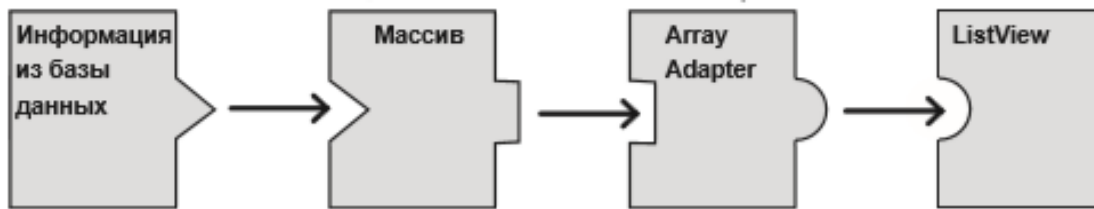
Метод **openOrCreateDatabase**: Открытие и создание баз данных без использования **SQLiteOpenHelper**

Вы можете открывать и создавать базы данных без помощи класса **SQLiteOpenHelper**, используя метод **openOrCreateDatabase()**, принадлежащий объекту **Context** вашего приложения. Получите доступ к базе данных в два шага. Сначала вызовите метод **openOrCreateDatabase()**, чтобы создать новую базу данных. Затем из полученного экземпляра базы данных вызовите **execSQL()**, чтобы выполнять команды на языке SQL, с помощью которых будут созданы таблицы и установлены отношения между ними.

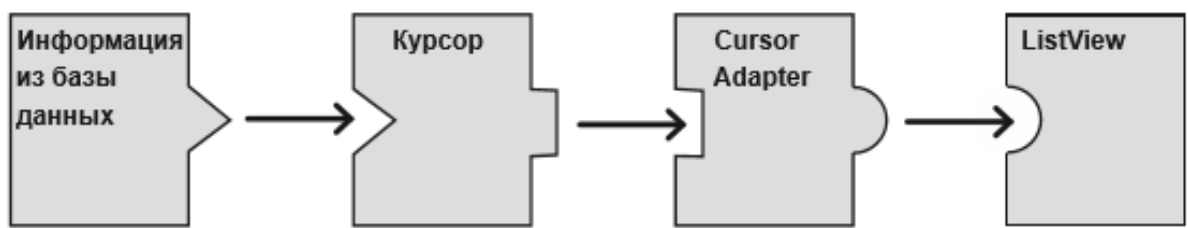
```
private static final String DATABASE_NAME = "myDatabase.db";
private static final String DATABASE_TABLE = "mainTable";
private static final String DATABASE_CREATE =
    "create table " + DATABASE_TABLE + " ( _id integer primary key
autoincrement," + "column_one text not null);";
SQLiteDatabase myDatabase;
private void createDatabase() {
    myDatabase = openOrCreateDatabase(DATABASE_NAME,
Context.MODE_PRIVATE,
    null);
    myDatabase.execSQL(DATABASE_CREATE);
}
```

Представление прочитанных данных

Одно из возможных решений выглядит так: прочитать список (студентов, заголовков) из базы данных и сохранить информацию в массиве, передаваемом адаптеру массива.



Такое решение работает, но... При такой маленькой базе данных, как у нас, чтение всей информации и хранение ее в массиве (то есть в памяти) не создает проблем. Но если приложение работает с очень большим объемом информации, ее чтение из базы данных займет некоторое время. Кроме того, для хранения массива потребуется много памяти. Вместо этого мы перейдем с класса адаптера массива **ArrayAdapter** на адаптер курсора **CursorAdapter**



Адаптер курсора читает столько данных, сколько нужно.

Допустим, наша база данных намного больше — например, это списки для университета. Может оказаться, что вместо трех студентов в базе данных придется хранить 2000. Однако в списке при этом будет отображаться лишь малая часть записей. В представлении **ListView** одновременно может отображаться лишь небольшая часть записей. На устройствах с маленьким экраном изначально могут выводиться, скажем, первые 11. Если бы данные хранились в массиве, нам пришлось бы загрузить все 2000 записей из базы данных в массив и только потом вывести часть данных. С **CursorAdapter** все работает немного иначе.

Когда списковое представление отображается впервые, оно масштабируется по размерам экрана. Предположим, в нем достаточно места для отображения пяти элементов.

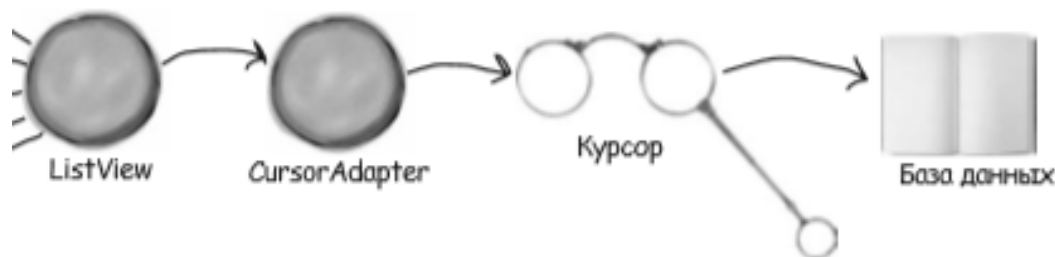
Компонент **ListView** не знает, где именно хранятся данные — в массиве или базе данных, — но знает, что он получит данные от адаптера. Соответственно, он обращается к адаптеру и запрашивает первые пять записей.

Объект **CursorAdapter** получает курсор при создании, но обращается к курсору за данными только тогда, когда потребуется.

Несмотря на то что таблица базы данных содержит 2000 записей, курсор должен прочитать только первые пять. Такой подход намного эффективнее, и он означает, что данные появятся на экране намного быстрее.

Когда пользователь прокручивает список, **CursorAdapter** приказывает курсору прочитать другие записи из базы данных. Если пользователь

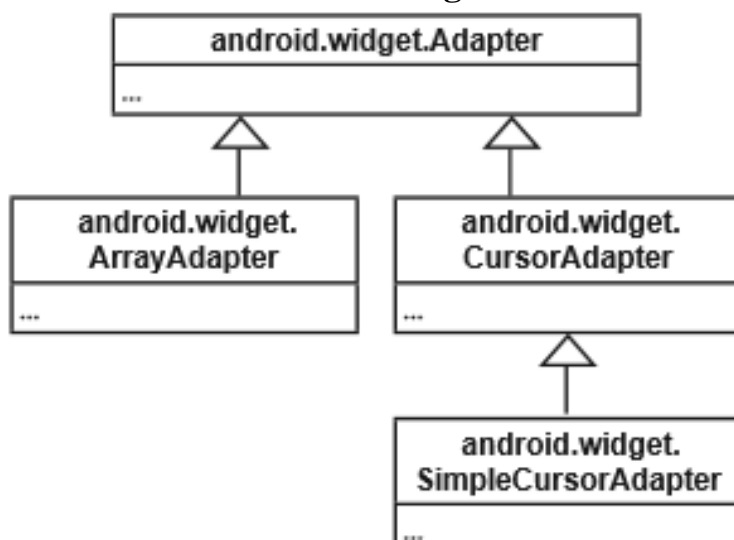
ограничивается минимальной прокруткой и открывает только одну новую строку, курсор читает из базы данных одну запись.



Итак, класс **CursorAdapter** в данном случае работает намного эффективнее **ArrayAdapter**: он читает данные только тогда, когда в них возникнет необходимость.

SimpleCursorAdapter

Класс **SimpleCursorAdapter** — специализация **CursorAdapter**, которая может использоваться в большинстве ситуаций с выводом данных курсора в списковом представлении. Он получает столбцы из курсора и связывает их с компонентом **TextViews** или **ImageViews**



Первое, что необходимо учесть при создании курсора для адаптера, — какие столбцы должен содержать курсор. В курсор следует включить все столбцы, которые должны отображаться в списковом представлении, вместе со столбцом с именем `_id`. Столбец `_id` должен быть включен в курсор, иначе адаптер курсора работать не будет. Почему? В Android принято присваивать столбцу первичного ключа таблицы имя `_id`. Это правило настолько закрепилось в Android, что адаптер курсора предполагает, что этот столбец всегда присутствует, и использует его для однозначной идентификации каждой строки в курсоре. При использовании адаптера курсора со списковым представлением этот столбец используется для определения строки, на которой щелкнул пользователь. Так как адаптер курсора нужен нам, курсор должен содержать столбцы `_id` и `COLUMN_NAME_NAME`.

При создании простого адаптера необходимо указать, как должны выводиться данные, какой курсор следует использовать и какие столбцы должны связываться с теми или иными представлениями.

```
Cursor cursor = db.query(  
    DBEntry.TABLE_NAME, // имя таблицы  
    projection,          // столбцы  
    selection,           // столбцы для WHERE  
    selectionArgs,       // значения для WHERE  
    null,               // не группировать строки  
    null,               // не фильтровать  
    sortOrder           // порядок сортировки  
);
```

```
CursorAdapter listAdapter = new SimpleCursorAdapter(this,  
    android.R.layout.simple_list_item_1,  
    cursor,  
    new String[]{DBEntry.COLUMN_NAME_NAME}  
    new int[]{android.R.id.text1},  
    0);  
listStudent.setAdapter(listAdapter);
```

Как и в случае с адаптером массива, мы используем **android.R.layout.simple_list_item_1**, чтобы сообщить, что каждая строка в курсоре должна отображаться в виде одного текстового представления в списковом представлении. Этому текстовому представлению назначен идентификатор **android.R.id.text1**. Общая форма конструктора **SimpleCursorAdapter** выглядит так:

```
SimpleCursorAdapter adapter =  
    new SimpleCursorAdapter(  
        Context context,  
        int layout, ← Как должны отображаться данные  
        Cursor cursor, ← курсор → включает id и данные  
        String[] fromColumns,  
        int[] toViews, ← Соответствие между  
                        столбцами курсора и  
                        представлениями  
        int flags); ← определение поведения  
                     курсора
```

Параметры **context** и **layout** — те же, которые использовались при создании адаптера массива. В параметре **context** передается текущий контекст,

а параметр **layout** сообщает, как должны отображаться данные. Вместо массива, из которого должны загружаться данные, нужно задать курсор с данными при помощи параметра **cursor**. Параметр **fromColumns** указывает, какие столбцы в курсоре должны использоваться, а параметр **toViews** — в каких представлениях они должны отображаться. В параметре **flags** обычно передается 0 (значение по умолчанию). Также можно передать значение **FLAG_REGISTER_CONTENT_OBSERVER** для регистрации наблюдателя, который будет оповещаться об изменении данных. Здесь эта возможность не рассматривается, так как она может привести к утечке.

Курсоры не отслеживают изменения информации в базе данных. Если информация изменится после создания курсора, то курсор обновлен не будет. Он по-прежнему содержит исходные записи без каких-либо изменений

```
public void onRestart() {
    super.onRestart();
    ...
    db =
    Cursor cursor =
    ListView list =
    CursorAdapter =
    //Замена курсора, используемого CursorAdapter, только что созданным
    adapter.changeCursor(cursor);
```

Курсор должен оставаться открытым на тот случай, если из него потребуется прочитать дополнительные данные. Например, это может произойти, если пользователь прокрутит списковое представление для просмотра новой порции данных.

Это означает, что курсор и базу данных не удастся закрыть сразу же после вызова метода **setAdapter()** для связывания со списковым представлением. Вместо этого для закрытия можно воспользоваться методом **onDestroy()** активности. Так как активность готовится к уничтожению, сохранять связь с курсором или базой данных не нужно, и их можно закрыть

```
public void onDestroy() {
    super.onDestroy();
    cursor.close();
    db.close();
}
```

Работа с потоками

Основная проблема с обращениями к медленной базе данных заключается в том, что она может замедлить реакцию приложения на действия пользователя.

Чтобы понять, почему это происходит, необходимо задуматься над тем, как работают программные потоки в Android. Начиная с версии Lollipop, существуют три вида потоков, которые необходимо учитывать.

1) **Основной поток событий (UI)** Этот поток прослушивает интенты, получает сообщения о касаниях от экрана и вызывает все методы внутри ваших активностей. Т.е. отвечает за все что происходит с экраном.

2) **Поток визуализации.** Этот поток, с которым вы обычно не взаимодействуете, читает список запросов на обновление экрана, а затем выдает команды низкоуровневому графическому оборудованию на перерисовку экрана.

3) **Все остальные потоки, созданные вами**

Приложение выполняет почти всю свою работу в основном потоке событий. Потому что основной поток событий выполняет методы событий приложения. Если включить код базы данных в метод **onCreate()**, то основной поток событий будет занят работой с базой данных вместо того, чтобы заниматься обработкой событий от экрана или других приложений. Если выполнение кода базы данных занимает много времени, у пользователя может возникнуть ощущение, что приложение забыло о его существовании. Итак, необходимо *вынести код базы данных из основного потока событий* и выполнить его в отдельном потоке в фоновом режиме.

Подготовку интерфейса, загрузку контент-view и т.п. оставим в *основном потоке*.

Код обновления представлений из базы данных обязательно выполняется *в главном потоке*, иначе произойдет исключение.

Для этого существует ряд классов:

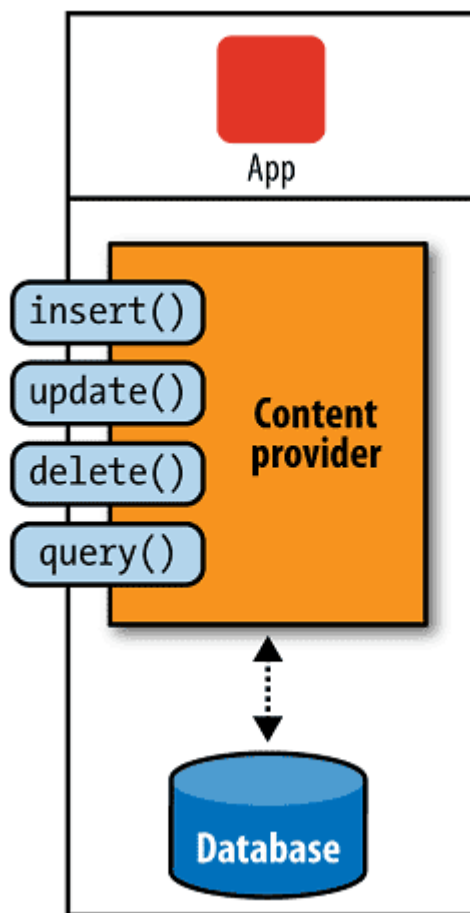
Класс **AsyncTask** – помощник над *Thread* и *Handler*. Подходит для небольших операций (считается устаревшим с API 30).

Классы **Executor**, **ThreadPoolExecutor** и **FutureTask** - для более длительных операций.

Итак: С очень маленькими базами данных — как та, которая используется в вашем приложении, — вы, вероятно, не заметите различий во времени работы с базой данных. Но это объясняется только малым размером базы данных. Если база данных достаточно велика или приложение работает на медленном устройстве, время обращения к базе данных будет значительным. Поэтому — *код работы с базой данных всегда должен выполняться в фоновом режиме*.

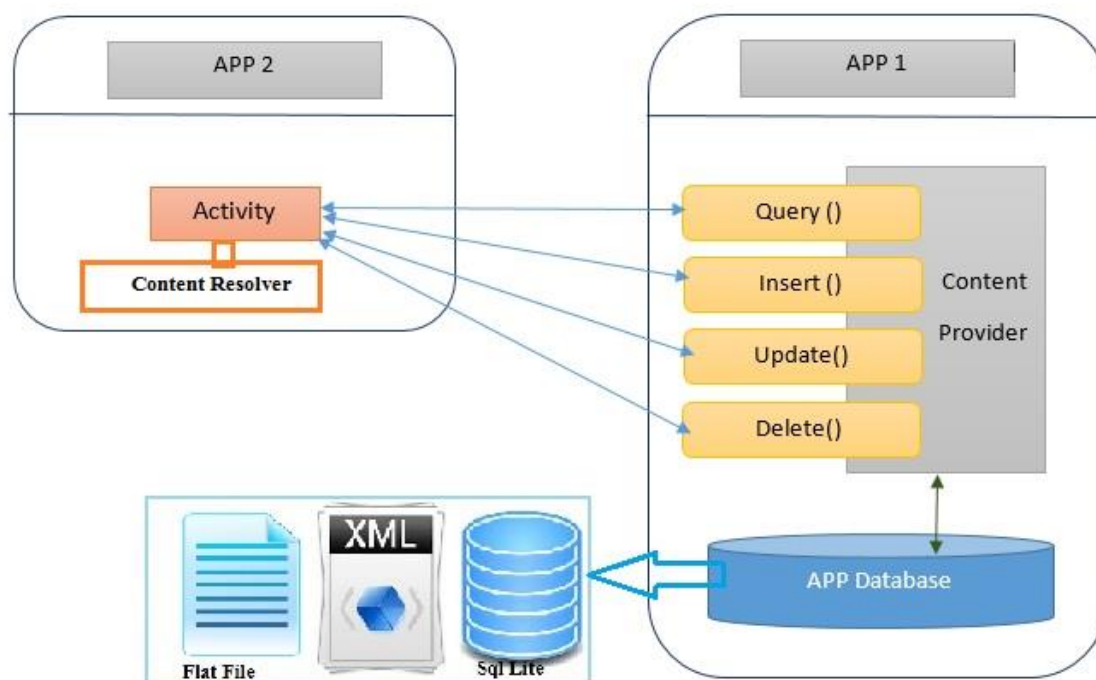
Контент-провайдеры

Контент-провайдеры предоставляют интерфейс для публикации и потребления структурированных наборов данных, основанный на URI с использованием простой схемы. Контент-провайдер или "Поставщик содержимого" (Content Provider) - это оболочка (wrapper), в которую заключены данные.



Если ваше приложение использует базу данных SQLite, то только ваше приложение имеет к ней доступ. Но бывают ситуации, когда данные желательно сделать общими. Простой пример - ваши контакты из телефонной книги тоже содержатся в базе данных, но вы хотите иметь доступ к данным, чтобы ваше приложение тоже могло выводить список контактов. Так как вы не имеете доступа к базе данных чужого приложения, был придуман специальный механизм, позволяющий делиться своими данными всем желающим.

В общем случае, контент-провайдеры следует создавать только тогда, когда требуется предоставить другим приложениям доступ к данным вашего приложения. В остальных случаях рекомендуется использовать СУБД (SQLite). Тем не менее, иногда контент-провайдеры используются внутри одного приложения для поиска и обработки специфических запросов к данным.



Базу данных SQLite можно заключить в поставщик содержимого. Чтобы получить данные или сохранить в нём новую информацию, нужно использовать набор REST-подобных идентификаторов URI. Например, если бы вам было нужно получить студента из списка студентов из поставщика содержимого, то вам понадобился бы такой URI :

```
content://by.belstu.fit/students/ivanov
```

Любая программа, работающая в устройстве, может использовать такие URI для доступа к данным и осуществления с ними определенных операций.

Встроенные поставщики

В Android используются встроенные поставщики содержимого (пакет **android.provider**). Неполный список поставщиков содержимого:

- Browser
- CallLog
- Contacts
 - People
 - Phones
 - Photos
 - Groups
- MediaStore
 - Audio
 - Albums
 - Artists

- Genres
 - Playlists
 - Images
 - Thumbnails
 - Video
- Settings

На верхних уровнях иерархии располагаются базы данных, на нижних - таблицы. Так, Browser, CallLog, Contacts, MediaStore и Settings - это отдельные базы данных SQLite, инкапсулированные в форме поставщиков. Обычно такие базы данных SQLite имеют расширение DB и доступ к ним открыт только из специальных пакетов реализации (implementation package). Любой доступ к базе данных из-за пределов этого пакета осуществляется через интерфейс поставщика содержимого.

Создание контент-провайдера

Для создания собственного контент-провайдера нужно унаследоваться от абстрактного класса `ContentProvider`:

```
public class MyContentProvider extends ContentProvider {
}
```

В классе необходимо реализовать абстрактные методы `query()`, `insert()`, `update()`, `delete()`, `getType()`, `onCreate()`.

Прослеживается некоторое сходство с созданием обычной базы данных.

`ContentResolver` используется для выполнения запросов и транзакций от активности к контент-провайдеру. `ContentResolver` включает в себя методы для запросов и транзакций, аналогичные тем, что содержит `ContentProvider`. Объекту `ContentResolver` не нужно знать о реализации контент-провайдера, с которым он взаимодействует - любой запрос всего лишь принимает путь URI, в котором указано, к какому объекту `ContentProvider` необходимо обращаться.

Content Provider Conceptual Model

