Android - Cards de Referência

SQLite

Bem vindo ao nosso nono card sobre Android! :-) Qual aplicação não precisa persistir dados, hein? Quase todas. Uma forma de armazenar dados no Android é usando o SQLite. Vamos ver como usar ele em nossa aplicação?

SQLITE

A persistência de dados é algo que volta e meia nós temos que nos preocupar. Praticamente toda aplicação precisa guardar informações. No Android, temos o SQLite para nos socorrer. Sim, isso mesmo. Você armazena, recupera e gerencia os dados usando instruções SQL. Claro, as instruções SQL que usaremos estão limitadas conforme o que o SQLite nos permite fazer.

Por que o SQLite? Bom, os engenheiros do Android devem ter seus motivos, mas eu gosto de enumerar alguns que me levam a crer que o SQLite é uma boa escolha:

1. É opensource;
2. Não requer um monte de configurações;
3. Os dados são guardados em um arquivo e a segurança fica a cargo do próprio sistema de arquivos;
4. Não é um servidor e toda sobrecarga proveniente disto.

Embora o SQLite use SQL, a API do Android para usar o SQLite provê alguns utilitários que facilitam a manipulação do banco de dados.

ESTENDENDO SQLITEOPENHELPER

A forma mais conveniente e prática para se iniciar com o uso do SQLite é criar uma classe que estende de **android.database.sqlite.SQLiteOpenHelper**. Uma vez que estendemos esta classe, percebemos que ela é abstrata e pede que você implemente dois métodos: **onCreate()** e **onUpgrade()**.

Ela também vai pedir que você implemente um construtor padrão. Feito isto, teremos a classe da próxima listagem.

A partir de agora, toda interação com o banco de dados será realizada a partir de uma instância desta classe! O que fazemos no método **onCreate()**? Ele é chamado apenas UMA vez: no momento da criação do banco. Aqui, o ideal é que você crie suas tabelas.

```
public class DBOpenHelper extends SQLiteOpenHelper {

   public DBOpenHelper(Context context, String name,
         CursorFactory factory, int version) {
      super(context, name, factory, version);
   }

   public void onCreate(SQLiteDatabase database) {
         database.execSQL("CREATE TABLE tabela ( _id
INTEGER PRIMARY KEY AUTOINCREMENT, nome TEXT)");
   }

   public void onUpgrade(SQLiteDatabase db, int oldVersion,
         int newVersion) {
      if (newVersion > 10) {
         // Atualizar alguma tabela?
      }
   }
}
```

O parâmetro para este método é exatamente uma instância do banco de dados, representado pela classe **SQLiteDatabase**. E o método **onUpgrade()**? Ele é chamado sempre que há uma nova versão do banco de dados. Eu escrevi do BANCO DE DADOS e não da sua aplicação. Observe que você recebe dois parâmetros a mais: a versão antiga e a nova.

Que versão é essa? É uma versão que você mesmo define. E para que isso? Vamos supor que você lança a versão 1 do seu aplicativo. Aí, cria o banco de dados na versão 1 também. Depois lança a versão 2 do aplicativo. Mas o banco de dados fica intacto. Nada muda.

Agora, você lança a versão 3 do aplicativo e atualiza duas tabelas do banco de dados e coloca o banco de dados como sendo a versão 2. Opa! Uma nova atualização do aplicativo, agora para a versão 4. E mais uma nova atualização nas tabelas, que leva seu banco para a versão 3. Entendidos?

Agora um usuário baixa seu aplicativo na versão 1. Depois de muito tempo sem usar, ele resolve atualizar o aplicativo e vai direto para a versão 4. Como garantir que o aplicativo continua funcionando com o mesmo banco de dados? Você precisa atualizar o esquema do banco de dados deste usuário, não é? E você usa o método **onUpgrade()** para isso!

Você define a versão do banco de dados toda vez que cria uma instância de **SQLiteOpenHelper**, no nosso caso, uma instância de **DBOpenHelper**. Observe que passamos quatro parâmetros para ele. Um contexto, nome do banco de dados, fábrica de cursor e a versão. O **CursorFactory** é necessário somente quando você tem sua própria implementação de Cursor. São casos raros e não vamos tratar eles aqui!

Resumindo, na maioria das vezes: **onCreate()** é para CREATE TABLE e **onUpgrade()** para ALTER TABLE. :-)

INSERINDO DADOS

Agora que já temos nossa classe **DBOpenHelper** criada, podemos usar ela para inserir dados na nossa nova tabela. Temos duas formas de fazer isto. Uma para quem gosta muito de escrever comandos SQL e outra para quem gosta menos. Vamos analisar as duas formas.

Vamos começar com aqueles que gostam de SQL. Primeiro, precisamos instanciar nossa classe **DBOpenHelper** e chamar o método **getWritableDatabase()**. Teremos, então, uma instância de **SQLiteDatabase**. Vamos usar agora o método **execSQL(String)**.

```
public class MainActivity extends Activity {
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        DBOpenHelper dbOpenHelper =
                new DBOpenHelper(this, "banco.sqlite", null, 1);

        SQLiteDatabase database =
                dbOpenHelper.getWritableDatabase();

        database.execSQL(
                "insert into tabela (nome) values ('meu nome')");
    }
}
```

O trecho de código logo acima está demonstrando o que acabamos de comentar. Vamos ver agora a segunda forma de inserir informações. A mesma classe **SQLiteDatabase** fornece um método chamado **insert()** e que recebe alguns parâmetros. O primeiro deles é o nome da tabela. O segundo... bem, o segundo vamos deixar para explicar daqui a pouco. O terceiro é um **ContentValues**, que nada mais é do que uma estrutura igual a um **Map**, contendo chave=valor.

Vamos entender o tal do segundo parâmetro no método **insert()**? O nome dele é **nullColumnHack**. Ele existe por causa de uma peculiaridade do SQLite. Caso você tenha uma tabela com todas as colunas sem serem obrigatórias e queira inserir uma linha em branco, faria um **"insert into tabela"** em SQL, certo? Com SQLite, essa instrução é inválida.

É preciso informar pelo menos UM campo na instrução INSERT. E o nullColumnHack é exatamente para isto. Você informa qual coluna será usada para colocar na instrução INSERT que ele vai criar para você automaticamente. Entendeu? Então, ali você coloca uma String com o nome da coluna.

```
public class MainActivity extends Activity {
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        DBOpenHelper dbOpenHelper =
            new DBOpenHelper(this, "banco.sqlite", null, 1);

        SQLiteDatabase database =
            dbOpenHelper.getWritableDatabase();

        ContentValues values = new ContentValues();
        values.put("nome", "meu nome");
        database.insert("tabela", null, values);
    }
}
```

O código acima demonstra como fazer um INSERT usando o método insert(). Por que é interessante usar ele? Observe que você usa o **ContentValues** sem se importar em concatenar strings. Evita também possíveis SQL Injections ou até mesmo erros normais quando se constrói Queries.

ATUALIZANDO DADOS

As coisas aqui são bem parecidas a como fazemos para inserir dados. Também temos dois métodos que podem nos ajudar. O primeiro continua sendo o **execSQL()**. Bom, não vamos discutir ele novamente, certo? Basta você criar uma instrução SQL com o comando UPDATE e ser feliz.

Vamos ver o segundo método, chamado **update()**. Ele também está na classe **SQLiteDatabase** mas tem mais parâmetros. O primeiro continua sendo o nome da tabela. O segundo agora é o **ContentValues** com os campos que deverão ser atualizados. O terceiro é uma String e o quarto é um Array de Strings.

Este método ainda retorna um inteiro com a quantidade de registros que foram atualizados. Bom, para entender, lembre que em um UPDATE você normalmente informa uma clásula WHERE, lembra? Você quer atualizar, por exemplo o nome de uma pessoa com ID igual a 2.

As combinações são variadas. No parâmetro da String, você cria uma instrução do tipo **nome = ? and outrocampo=? and maisoutrocampo=?**. Agora, você precisa informa os valores que serão substituídos nas interrogações. E como faz isso? Acertou! Passando eles no Array de Strings, que é o próximo parâmetro.

```
public class MainActivity extends Activity {
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        DBOpenHelper dbOpenHelper =
            new DBOpenHelper(this, "banco.sqlite", null, 1);

        SQLiteDatabase database =
            dbOpenHelper.getWritableDatabase();

        ContentValues values = new ContentValues();
        values.put("nome", "meu nome");

        String whereClause = "_id=?";
        String[] whereArgs = new String[] { "1" };
        database.update("tabela", values, whereClause, whereArgs);
    }
}
```

O trecho acima está demonstrando o uso do update(). As vantagens do método **update()** são as mesmas que discutimos no método **insert()**. Você não precisa ficar se preocupando em concatenar uma String gigantesca. :-)

SELECIONANDO DADOS

Já criamos as tabelas, inserimos e atualizamos dados. Agora vamos fazer **SELECTS** nessa tabela. Mais uma vez, temos duas formas de fazer isso. E, de novo, um para cada gosto! Podemos fazer a seleção usando Queries SQL ou usar o método **query()**. Bom, só que aqui talvez você prefira escrever queries SQL mesmo!

```
public class MainActivity extends Activity {
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        DBOpenHelper dbOpenHelper =
            new DBOpenHelper(this, "banco.sqlite", null, 1);

        SQLiteDatabase database =
            dbOpenHelper.getReadableDatabase();

        Cursor cursor =
            database.rawQuery("select * from tabela where _id=?",
            new String[] { "1" });

        while (cursor.moveToNext()) {
            int index = cursor.getColumnIndex("nome");
            Log.d("DB", cursor.getString(index));
        }
        cursor.close();
        database.close();
    }
}
```

De cara, já temos uma mudança inicial. Antes fazíamos **getWritableDatabase()** e agora vamos fazer **getReadableDatabase()**. Tem uma diferença imensa em usar um método ou o outro? Na verdade, não tem. Mas, semanticamente, fica mais bonito, não acha?

Observe a listagem anterior. Estamos usando o método **rawQuery()**, que recebe dois parâmetros apenas. Uma String com a query e um Array contendo os valores dos campos na cláusula **WHERE**. Fizemos isso com o método **update()**, lembra? O **rayQuery()** vai lhe retornar um objeto do tipo **Cursor**.

O cursor serve para você acessar os dados que foram retornados da consulta. Para ir um a um, normalmente, fazemos um **cursor.moveToNext()** até que este método retorne **FALSE**.

Vamos obter os dados de cada posição do cursor chamando os métodos no formato **getXXX()**, que recebem como parâmetro o índice da coluna que queremos ter o valor. Nessa linha, temos **getString()**, **getInt()**, **getFloat()** e por aí vai. Mas, e se tivermos uma Query com dezenas de campos? Vai ficar chato chamar um getString(0), getString(1)... então, é melhor fazer um **cursor.getColumnIndex(String)** passando o nome da coluna como parâmetro. Esse método retorna o índice que, em seguida, você pode usar nos métodos **getXXX()**.

Vamos usar o método **query()** para fazer uma consulta?

```
public class MainActivity extends Activity {
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        DBOpenHelper dbOpenHelper =
            new DBOpenHelper(this, "banco.sqlite", null, 1);
            SQLiteDatabase database =
                dbOpenHelper.getReadableDatabase();

            String[] columns = new String[] {"nome"};
            String selection = "_id=?";
            String[] selectionArgs = new String[] {"1"};
            String groupBy = null;
            String having = null;
            String orderBy = null;

            Cursor cursor = database.query("tabela", columns,
                    selection, selectionArgs, groupBy, having, orderBy);
            while (cursor.moveToNext()) {
                int index = cursor.getColumnIndex("nome");
                Log.d("DB", cursor.getString(index));
            }
            cursor.close();
            database.close();
    }
}
```

Entendeu agora o motivo de eu ter dito que talvez você preferisse escrever sua Query diretamente? O método tem muitos parâmetros e nem todos gostam disso. Mas, vamos aos parâmetros. Em sequência, temos:

Nome da Tabela - Uma String onde você informa qual o nome da tabela.

Colunas - Um Array contendo quais colunas da tabela devem ser retornadas pela consulta.

Critérios de Seleção - Quais os critérios para selecionar ou filtrar os dados retornados? Imagine que você está construíndo uma consulta e informa a cláusula WHERE. Aqui, temos uma String contendo o que teria no WHERE, por exemplo: _id=? and nome=? and valor=?

Valor dos Critérios de Seleção - No parâmetro anterior você definiu os critérios, mas e os valores? Você lá só colocou interrogações onde deveriam ir os valores. Você informa quais valores entrarão no lugar das interrogações usando um Array de Strings.

Agrupamento - Uma String informando como ocorrerá o agrupamento dos dados na consulta. Lembra a instrução HAVING do SQL? Então, é a mesma coisa! Aqui, por exemplo, você pode colocar uma String "**SUM(campo) > 1000**".

Ordenamento - Uma String informando como ocorrerá o ordenamento dos dados retornados. Lembra da instrução ORDER BY do SQL? Igualzinho! Você pode colocar uma String igual a "**NOME, SOBRENOME ASC**", por exemplo.

O uso do Cursor é sempre igual, não muda no caso do uso de uma forma ou outra.

REMOVENDO DADOS

Outra vez! Temos duas opções. Usando o comando **execSQL()** e escrevendo um comando SQL de **DELETE**, ou usar o método **delete()**. Vamos a um exemplo.

```
public class MainActivity extends Activity {
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        DBOpenHelper dbOpenHelper =
            new DBOpenHelper(this, "banco.sqlite", null, 1);

        SQLiteDatabase database =
            dbOpenHelper.getWritableDatabase();

        database.delete("tabela", "_id=?", new String[] { "1" });
    }
}
```

Nada muito diferente do que já vimos, certo? Temos como parâmetros o nome da tabela, as cláusulas de seleção dos registros que serão removidos e um Array com os valores.

Só temos uma diferença para notar. Esse método retorna um inteiro, que informa a quantidade de registros que foram removidos.

TRANSAÇÕES

Tem como fazer blocos de transações no Android com SQLite? Tem sim.

```
public class MainActivity extends Activity {
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        DBOpenHelper dbOpenHelper =
            new DBOpenHelper(this, "banco.sqlite", null, 1);

        SQLiteDatabase database =
            dbOpenHelper.getWritableDatabase();

        database.beginTransaction();
        // Atualizações diversas!
        database.setTransactionSuccessful();
        database.endTransaction();
    }
}
```

No trecho de código acima, temos três métodos para observar. O primeiro é para iniciar a transação: **beginTransaction()**. Com a transação iniciada, podemos fazer diversas operações, inclusive em tabelas diferentes.

Você quer que seja feito um COMMIT? Então, chame o método **setTransactionSuccessful()**. Mas, não é só isso. Ainda falta mais um método para a transação ser finalizada. Chame o método **endTransaction()** e pronto.

Ah! Por algum motivo você queria um ROLLBACK? Então, esqueça de chamar o método **setTransactionSuccessful()** e chame **endTransaction()** logo após. Entendeu? O que define um COMMIT ou ROLLBACK é ter chamado ou não o método **setTransactionSuccessful()**. Mas lembre-se de sempre terminar com um **endTransaction()**.

Como se usa isso na prática? Normalmente, você vai fazer as instruções que deseja e fazer alguma checagem em um IF para definir se chama **setTransactionSuccessful()** ou não. Ou, pode colocar em um **TRY CATCH FINALLY** como estamos fazendo no próximo trecho de código!

```
public class MainActivity extends Activity {
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        DBOpenHelper dbOpenHelper =
            new DBOpenHelper(this, "banco.sqlite", null, 1);

        SQLiteDatabase database =
            dbOpenHelper.getWritableDatabase();

        database.beginTransaction();
        try {
            // Atualizações diversas!
            // Se lançar uma exceção, não chama o método abaixo.
            database.setTransactionSuccessful();
        } finally {
            database.endTransaction();
        }
    }
}
```

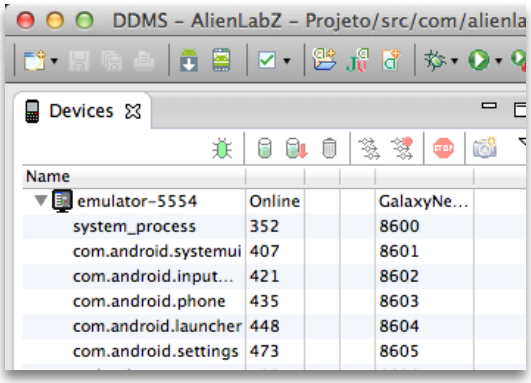
ACESSANDO O BANCO DE DADOS

Você tem uma forma de acessar o banco de dados através de um terminal. Independente se você está no Linux, Windows ou Mac, basta abrir um terminal e executar alguns poucos comandos.

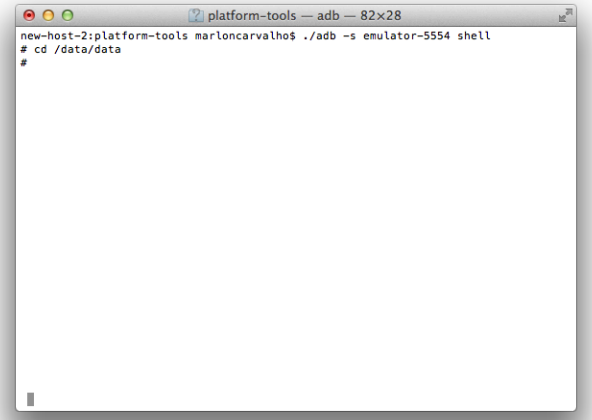
Caso você esteja no Windows, quando eu falo em Terminal, eu estou falando da Linha de Comando. No Linux e Mac é conhecido como Terminal mesmo.

Agora, lembra a pasta em que nós descompactamos a SDK do Android? Então, vamos nela usando o terminal. Seja Windows ou Linux/Mac, você chega até lá usando vários comando **CD /DIR**.

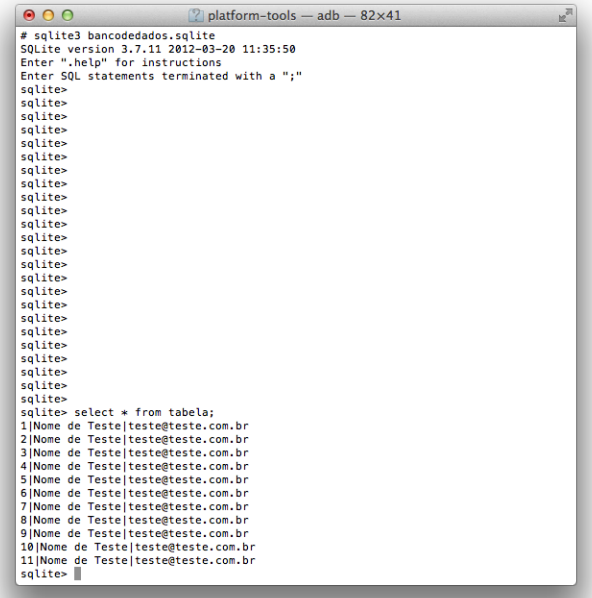
Entre na pasta **platform-tools**. Você terá um programa chamado **adb** nesta pasta. Aqui você precisa disparar um comando: **adb -s <emulador> shell**. No lugar do <emulador>, você precisa colocar o nome do emulador. E qual é esse nome? No Eclipse, na perspectiva DDMS, você terá uma aba para **Devices**.



No nosso caso, temos o emulador **emulator-5554**. Pra falar a verdade, se você tem sempre um único emulador, quase sempre será esse nome. :) Agora execute o comando **adb -s emulator-5554 shell**.



Navegue para o diretório **/data/data** usando o comando **cd /data/data/**. Execute o comando **ls**. Você verá aí um diretório para cada programa instalado em seu emulador. Lembra o nome do pacote e do seu programa? Então, navegue para esse diretório. Você terá um diretório **databases** com o arquivo de seu banco de dados. Execute **sqlite3 seuarquivo**.



Pronto! Agora é só disparar os comandos SQL para acessar suas tabelas!

Sobre o Autor

Marlon Silva Carvalho

É um programador de longa data. Já peregrinou por diversas linguagens e hoje se considera um programador agnóstico. Atualmente, está fascinado pelo mundo mobile e suas consequências para a humanidade. Está mais focado no desenvolvimento para Android, embora também goste de criar aplicativos para iOS.

Trabalha em projetos sobre mobilidade no SERPRO, tendo atuado no projeto Pessoa Física, para a Receita Federal do Brasil. Você pode encontrar ele no Twitter, no perfil @marlonscarvalho, em seu blog, no endereço http://marlon.silvacarvalho.net e através do e-mail marlon.carvalho@gmail.com.

Sobre os Cards

Caso você já tenha visto os excelentes RefCards da DZone, deve ter percebido a semelhança. E é proposital. A ideia surgiu após acompanhar estes RefCards por um bom tempo. Contudo, eles são mais gerais. O objetivo destes Cards é tratar assuntos mais específicos. Trazendo informações relevantes sobre eles.

Caso tenha gostado, continue nos acompanhando! Tentaremos trazer sempre novos cards sobre os mais variados assuntos. Inicialmente, focaremos no Android.

O ícone principal deste card é de autoria de Wallec e foi obtido em seu profile no DeviantArt.
http://wwalczyszyn.deviantart.com/