Simple ...ite

Chilap, 30 Ma...

Implement an ...base.

Download source code - 116 KB

Introduction

SQLite is a very good database engine for small applications to store relational data. However, most of the time, we may still want to store data in the traditional way using a file system: having different directories with different data files saved in those directories.

This way, your data is scattered on the file system as well as a database file. May be you have already gotten used to it but I want to gather all the data in just one place. When backing up data from your Android app, it would be very nice if they are all grouped into a single database file. So why not combine the SQLite tables and the traditional file system together? This project is going to 'create' a simple file system inside a SQLite database file.

Background

As you all know, any file system needs to allocate some room to store the metadata about the file system itself, the directories, and the files. In this project, only three tables are created. Let's have a look at the table schemas:

FsInfo (table to store information about the file system itself):

Hide Copy Code

```
CREATE TABLE FsInfo (infoName varchar(128) primary key,infoVal varchar(128))
```

This table is just a simple key-value pair structure.

FsBlock (table to store metadata about the directories and files)

Hide Copy Code

```
CREATE TABLE FsBlock (fsID integer primary key autoincrement,
                      fsType integer,
                      fsCreateTime integer,
                      fsLastModTime integer,
                      fsFileSize integer,
                      fsName varchar(512),
                      fsParent integer,
                      fsChild blob)
```

Each row in this table describes a directory or a file:

- **fsID** -- an auto increment ID to uniquely identify a directory or a file (no two directories or files will have the same ID) (ID number '1' is reserved for root directory)
- **fsType** -- '0' indicates a directory while '1' is a file
- **fsCreateTime** -- time of creation, which is stored as the format of the Windows file time (i.e., number of 100 nano seconds since 1601 Jan, 1)
- **fsLastModTime** -- time of last modification
- **fsFileSize** -- for a directory, it is '0'; for a file, it is the number of bytes stored in the 'DataBlock' table
- **fsName** -- name of file or directory (not full path)
- **fsParent** -- fsID of the parent directory
- **fsChild** -- for a directory, it is an array of fsIDs of its children (each ID is 4 bytes in size but can be changed to 8 bytes); for a file, it is a single 'dID', which is the primary key in the 'DataBlock' table

DataBlock (table to store the real data of all files)

Hide Copy Code

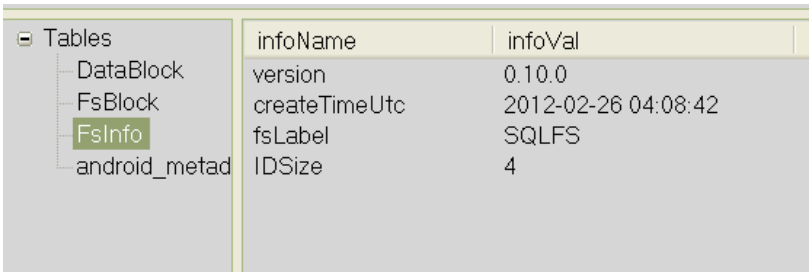
```
CREATE TABLE DataBlock (dID integer primary key autoincrement,
                        dFileType integer,
                        dTextData text,
                        dRawBinData blob)
```

Each row in this table contains the real data of a file:

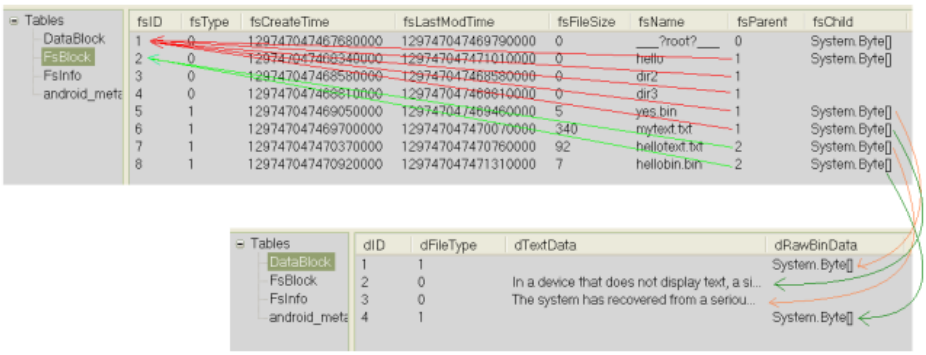
- **dID** -- an auto increment to uniquely identify the file content
- **dFileType** -- '0' means it is a text file and file data should be fetched from 'dTextData'; '1' means it is a binary file and file data should be fetched from 'dRawBinData'
- **dTextData** -- contains the text content of a file if the corresponding 'dFileType' is '0'
- **dRawBinData** -- contains the binary content of a file if the corresponding 'dFileType' is '1'

Real example

Let's take a real example of the tables above:



Only some simple data is stored in FsInfo. Actually, you can store anything you like.



The above FsBlock shows a directory hierarchy as follows:

Hide Copy Code

```
/ (root dir)
|
+-- hello/
|     |
|     +-- hellotext.txt
|     |
|     +-- hellobin.bin
|
+-- dir2/
|
+-- dir3/
|
+-- yes.bin
|
+-- mytext.txt
```

DataBlock shows a mapping from the 'files' in FsBlock. (The IDs in the FsBlock table are stored in BLOB so they are not shown in the picture above.)

Hide Copy Code

```
/yes.bin -- dID 1
/mytext.txt -- dID 2
/hello/hellotext.txt -- dID 3
/hello/hellobin.bin -- dID 4
```

SqlFs library

Enough background and examples. I call the library which implements the above file system as 'SqlFs'. Let's have a look at the Java classes.

Download source code - 116 KB

Inside the zip archive *sqlfs.zip*, there are two projects -- SqlFs (a library project) and TestSqlFs (contains some test cases for the SqlFs library).

If you look at the directory *SqlFs/src/com/sss/sqlfs*, the most commonly used classes are:

Hide Copy Code

```
SqlFs
SqlFsNode
SqlDir (derived from SqlFsNode)
SqlFile (derived from SqlFsNode)
IFileData
SimpleFileData (derived from IFileData)
FsID
```

You can play with the above SQLite tables (create, delete, read, write, and update directories and files) with just these few classes listed. I will describe these classes in brief.

SqlFs

Everything starts from the class `SqlFs`. First of all, you need to create/open a database file:

Hide Copy Code

```
SqlFs fs = SqlFs.create("/sdcard/hello.db", appContext);
```

After obtaining an instance of `SqlFs`, you can read/write name-value pairs in the FsInfo table using these methods:

Hide Copy Code

```
String getInfo(String infoName);
void writeInfo(String infoName, String infoVal);
```

To create directories/files under the root directory, you need to retrieve the root directory first:

Hide Copy Code

```
SqlDir rootDir = fs.getRootDir();
```

or if you know the absolute path (not relative) of a directory or file, try:

Hide Copy Code

```
SqlDir getDir(String dirPath)
SqlFile getFile(String filePath)
```

When finished, you need to 'close' the file system:

Hide Copy Code

```
void close();
```

Take a look at SqlDir below to see how you can play with it.

Path separator, current directory, and parent directory

Before moving on, it is better to define some symbols used in SqlFs. If you take a look at '*SqlFsConst.java*', you can see:

Path separator is forward slash -- '/' (just like Unix). Current directory is a single dot -- '.' Parent directory is two dots -- '..'. Invalid characters for directory and file names are -- '\', '/', ':', '*', '?', '"', '<', '>', '|'.

So an absolute path can be written as */sdcard/hello.txt* while a relative one can be written as *../hello/hello.txt*. Just like what you do on Unix or Linux.

SqlFsNode

`SqlFsNode` is the super class of `SqlDir` and `SqlFile`. You will never instantiate this class directly. This class contains common methods shared by both `SqlDir` and `SqlFile`:

Hide Copy Code

```
Calendar getCreateTime()
Calendar getLastModTime()
int getFileSize()
String getName()
SqlDir getParent()
boolean rename(String newName)
boolean isAncestor(SqlDir dir) // check if 'dir' is one of its ancestor
boolean move(String destPath)
boolean move(SqlDir destDir)
```

Argument of '`move`' can be an absolute or relative path.

SqlDir

Normal operations that can be performed by SqlDir:

Hide Copy Code

```
int getChildCount()
boolean isAlreadyExist(String name) // check if there is any child with the same name
SqlDir addDir(String dirName)
SqlFile addFile(String fileName)
boolean delete()
SqlFsNode getChild(String name)
ArrayList<SqlFsNode> getChildList()
ArrayList<SqlDir> getSubDirs()
ArrayList<SqlFile> getFiles()
SqlFsNode getFsNode(String path)
SqlDir getDir(String dirPath)
SqlFile getFile(String filePath)
```

The methods are quite intuitive. All paths must be relative here.

SqlFile and SimpleFileData

Let's take a look at what SqlFile can do:

Hide Copy Code

```
boolean delete()
boolean getFileData(IFileData fileData)
boolean saveFileData(IFileData fileData)
```

Example of `getFileData`:

Hide Copy Code

```
SqlFile file = rootDir.getFile("mytext.txt");
SimpleFileData fdRetrieve = new SimpleFileData();
file.getFileData(fdRetrieve);
if (fdRetrieve.isTextFile()) {
   String dataRetStr = fdRetrieve.getText();
    ...
}
else {
   byte[] dataRetrieve = fdRetrieve.getRawBinData();
   ...
}
```

Example of `saveFileData`:

Hide Copy Code

```
// save binary data
SqlFile file = rootDir.addFile("yes.bin");
SimpleFileData fd = new SimpleFileData();
byte[] dataBin = new byte[]{0x34, 0x12, 0x09, 0x11, 0x08};
fd.setRawBinData(dataBin);
file.saveFileData(fd);

// save text data
SqlFile file = rootDir.addFile("hellotext.txt");
SimpleFileData fd = new SimpleFileData();
String saveStr = "The system has recovered from a serious error.";
fd.setTextData(saveStr);
file.saveFileData(fd);
```

`SimpleFileData` is a reference implementation of `IFileData`. If you don't have special needs, just go with it. In fact, you can define your own implementation of `IFileData` but then you need to instantiate `SqlFs` using another '`SqlFs.create`':

Hide Copy Code

```
// assume MyFileData is your own implementation of IFileData
SqlFs fs = SqlFs.create("/sdcard/myfile.db", new MyFileData(), appContext);
```

Inside the test case project, TestSqlFs, there is an example of a user-defined `IFileData` implementation (`UrlFileData`). The schema of the 'DataBlock' table is also different from the one used by `SimpleFileData`.

FsID

It is a class to wrap around the fsID used in FsBlock. By default, it uses a 32 bit integer but can be changed to use 64 bit by changing the internal flag inside FsID:

Hide Copy Code

```
private static final boolean useLongID
```

Thread safety

I didn't test reading/writing the same database file with two different processes on Android but there is a test case (inside TestSqlFs -- `TestMultiReadWrite.testReadWrite`) to read/write the same DB with two different threads in the same process.

Internally, every public operation of `SqlFs`, `SqlDir`, and `SqlFile` is guarded by a lock:

Hide Copy Code

```
java.util.concurrent.locks.ReentrantLock
```

Thus, it is thread safe to work on the same DB in two different threads of the same process. Additionally, because many operations can't be performed in only one SQL statement, they are wrapped around by `beginTransaction` and `endTransaction`.

To play it safe, each thread should instantiate its own SqlFs (even accessing the same DB) and not pass the `SqlFs`, `SqlDir`, and `SqlFile` instances among threads.

How to run TestSqlFs.apk

It is not a normal Android apk with a GUI but need to be run under CmdConsole (http://www.codeproject.com/Articles/202996/Write-a-console-app-on-Android-using-Java).

Points of interest

This library is only good for small apps like those on Android. If you ask me about the performance of this library, um..., honestly, it won't be very good. But because those small Android apps won't do large amounts of read/write operations at any time, this library is fit to use.

License

Share

TWITTER FACEBOOK

About the Author

Chilap
Software Developer
Hong Kong
Like programming, reading, watching movies.
Wish to own a book store and a small cafe in the future.

You may also be interested in...

Building Reactive Apps

The Offine-First Approach to Mobile App Development

CppSQLite - C++ Wrapper for SQLite

SAPrefs - Netscape-like Preferences Dialog

A Simple Android SQLite Example

Window Tabs (WndTabs) Add-In for DevStudio

Comments and Discussions

You must Sign In to use this message board.

Search Comments

First Prev Next

My vote of 5
lvica76 8-Oct-13 3:45

My vote of 5
Sudhakar Shinde 2-May-13 20:06

My vote of 5
manoj kumar choubey 2-Apr-12 3:51

My vote of 5
eng. Plamen Kovandjiev 5-Mar-12 22:50

Refresh 1

General News Suggestion Question Bug Answer Joke Praise Rant Admin

Use Ctrl+Left/Right to switch messages, Ctrl+Up/Down to switch threads, Ctrl+Shift+Left/Right to switch pages.