sqlite常量的定义：  
const  
  SQLITE\_OK           = 0;  返回成功  
  SQLITE\_ERROR        = 1;  SQL错误或错误的数据库  
  SQLITE\_INTERNAL     = 2;  An internal logic error in SQLite  
  SQLITE\_PERM         = 3;  拒绝访问  
  SQLITE\_ABORT        = 4;  回调函数请求中断  
  SQLITE\_BUSY         = 5;  数据库文件被锁  
  SQLITE\_LOCKED       = 6;  数据库中的一个表被锁  
  SQLITE\_NOMEM        = 7;  内存分配失败  
  SQLITE\_READONLY     = 8;  试图对一个只读数据库进行写操作  
  SQLITE\_INTERRUPT    = 9;  由sqlite\_interrupt()结束操作  
  SQLITE\_IOERR        = 10; 磁盘I/O发生错误  
  SQLITE\_CORRUPT      = 11; 数据库磁盘镜像畸形  
  SQLITE\_NOTFOUND     = 12; (Internal Only)表或记录不存在  
  SQLITE\_FULL         = 13; 数据库满插入失败  
  SQLITE\_CANTOPEN     = 14; 不能打开数据库文件  
  SQLITE\_PROTOCOL     = 15; 数据库锁定协议错误  
  SQLITE\_EMPTY        = 16; (Internal Only)数据库表为空  
  SQLITE\_SCHEMA       = 17; 数据库模式改变  
  SQLITE\_TOOBIG       = 18; 对一个表数据行过多  
  SQLITE\_CONSTRAINT   = 19; 由于约束冲突而中止  
  SQLITE\_MISMATCH     = 20; 数据类型不匹配  
  SQLITE\_MISUSE       = 21; 数据库错误使用  
  SQLITE\_NOLFS        = 22; 使用主机操作系统不支持的特性  
  SQLITE\_AUTH         = 23; 非法授权  
  SQLITE\_FORMAT       = 24; 辅助数据库格式错误  
  SQLITE\_RANGE        = 25; 2nd parameter to sqlite\_bind out of range  
  SQLITE\_NOTADB       = 26; 打开的不是一个数据库文件  
  SQLITE\_ROW          = 100; sqlite\_step() has another row ready  
  SQLITE\_DONE         = 101; sqlite\_step() has finished executing

(源码网整理：[www.codepub.com](http://www.codepub.com/))

[**SQLITE3 使用总结 (ZZ)**](http://lostmyself.yo2.cn/2008/11/22/631814/)

Posted by [**LostMyself**](http://lostmyself.yo2.cn/author/)

十一月 22, 2008

发信人: lih (未醒), 信区: VC  
标 题: SQLITE3 使用总结  
发信站: BBS 侏罗纪站 (Wed Jul 11 09:02:02 2007)

前序：  
Sqlite3 的确很好用。小巧、速度快。但是因为非微软的产品，帮助文档总觉得不够。这  
些天再次研究它，又有一些收获，这里把我对 sqlite3 的研究列出来，以备忘记。

这里要注明，我是一个跨平台专注者，并不喜欢只用 windows 平台。我以前的工作就是为  
unix 平台写代码。下面我所写的东西，虽然没有验证，但是我已尽量不使用任何 windo  
ws 的东西，只使用标准 C 或标准C++。但是，我没有尝试过在别的系统、别的编译器下编  
译，因此下面的叙述如果不正确，则留待以后修改。

下面我的代码仍然用 VC 编写，因为我觉得VC是一个很不错的IDE，可以加快代码编写速度  
（例如配合 Vassist ）。下面我所说的编译环境，是VC2003。如果读者觉得自己习惯于  
unix 下用 vi 编写代码速度较快，可以不用管我的说明，只需要符合自己习惯即可，因为  
我用的是标准 C 或 C++ 。不会给任何人带来不便。

一、 版本  
从 www.sqlite.org 网站可下载到最新的 sqlite 代码和编译版本。我写此文章时，最新  
代码是 3.3.17 版本。

很久没有去下载 sqlite 新代码，因此也不知道 sqlite 变化这么大。以前很多文件，现  
在全部合并成一个 sqlite3.c 文件。如果单独用此文件，是挺好的，省去拷贝一堆文件还  
担心有没有遗漏。但是也带来一个问题：此文件太大，快接近7万行代码，VC开它整个机器  
都慢下来了。如果不需要改它代码，也就不需要打开 sqlite3.c 文件，机器不会慢。但是  
，下面我要写通过修改 sqlite 代码完成加密功能，那时候就比较痛苦了。如果个人水平  
较高，建议用些简单的编辑器来编辑，例如 UltraEdit 或 Notepad 。速度会快很多。

二、 基本编译  
这个不想多说了，在 VC 里新建 dos 控制台空白工程，把 sqlite3.c 和 sqlite3.h 添加  
到工程，再新建一个 main.cpp 文件。在里面写:

extern "C"

{

#include "./sqlite3.h"

};

int main( int , char\*\* )

{

return 0;

}

为什么要 extern “C” ？如果问这个问题，我不想说太多，这是C++的基础。要在 C++  
里使用一段 C 的代码，必须要用 extern “C” 括起来。C++跟 C虽然语法上有重叠，但  
是它们是两个不同的东西，内存里的布局是完全不同的，在C++编译器里不用extern “C”  
括起C代码，会导致编译器不知道该如何为 C 代码描述内存布局。

可能在 sqlite3.c 里人家已经把整段代码都 extern “C” 括起来了，但是你遇到一个  
.c 文件就自觉的再括一次，也没什么不好。

基本工程就这样建立起来了。编译，可以通过。但是有一堆的 warning。可以不管它。

三、 SQLITE操作入门  
sqlite提供的是一些C函数接口，你可以用这些函数操作数据库。通过使用这些接口，传递  
一些标准 sql 语句（以 char \* 类型）给 sqlite 函数，sqlite 就会为你操作数据库。

sqlite 跟MS的access一样是文件型数据库，就是说，一个数据库就是一个文件，此数据库  
里可以建立很多的表，可以建立索引、触发器等等，但是，它实际上得到的就是一个文件  
。备份这个文件就备份了整个数据库。

sqlite 不需要任何数据库引擎，这意味着如果你需要 sqlite 来保存一些用户数据，甚至  
都不需要安装数据库(如果你做个小软件还要求人家必须装了sqlserver 才能运行，那也太  
黑心了)。

下面开始介绍数据库基本操作。

（1） 基本流程  
i.1 关键数据结构

sqlite 里最常用到的是 sqlite3 \* 类型。从数据库打开开始，sqlite就要为这个类型准  
备好内存，直到数据库关闭，整个过程都需要用到这个类型。当数据库打开时开始，这个  
类型的变量就代表了你要操作的数据库。下面再详细介绍。

i.2 打开数据库

int sqlite3\_open( 文件名, sqlite3 \*\* );

用这个函数开始数据库操作。

需要传入两个参数，一是数据库文件名，比如：c:\\DongChunGuang\_Database.db。

文件名不需要一定存在，如果此文件不存在，sqlite 会自动建立它。如果它存在，就尝试  
把它当数据库文件来打开。

sqlite3 \*\* 参数即前面提到的关键数据结构。这个结构底层细节如何，你不要关它。

函数返回值表示操作是否正确，如果是 SQLITE\_OK 则表示操作正常。相关的返回值sqlit  
e定义了一些宏。具体这些宏的含义可以参考 sqlite3.h 文件。里面有详细定义（顺便说  
一下，sqlite3 的代码注释率自称是非常高的，实际上也的确很高。只要你会看英文，sq  
lite 可以让你学到不少东西）。

下面介绍关闭数据库后，再给一段参考代码。

i.3 关闭数据库

int sqlite3\_close(sqlite3 \*);

前面如果用 sqlite3\_open 开启了一个数据库，结尾时不要忘了用这个函数关闭数  
据库。

下面给段简单的代码：

extern "C"

{

#include "./sqlite3.h"

};

int main( int , char\*\* )

{

sqlite3 \* db = NULL; //声明sqlite关键结构指针

int result;

//打开数据库

//需要传入 db 这个指针的指针，因为 sqlite3\_open 函数要为这个指针分配内存，还要  
让db指针指向这个内存区

result = sqlite3\_open( “c:\\Dcg\_database.db”, &db );

if( result != SQLITE\_OK )

{

//数据库打开失败

return -1;

}

//数据库操作代码

//…

//数据库打开成功

//关闭数据库

sqlite3\_close( db );

return 0;

}

这就是一次数据库操作过程。

（2） SQL语句操作

本节介绍如何用sqlite 执行标准 sql 语法。

i.1 执行sql语句

int sqlite3\_exec(sqlite3\*, const char \*sql, sqlite3\_callback, void \*, char \*\*  
errmsg );

这就是执行一条 sql 语句的函数。

第1个参数不再说了，是前面open函数得到的指针。说了是关键数据结构。

第2个参数const char \*sql 是一条 sql 语句，以\0结尾。

第3个参数sqlite3\_callback 是回调，当这条语句执行之后，sqlite3会去调用你提供的这  
个函数。（什么是回调函数，自己找别的资料学习）

第4个参数void \* 是你所提供的指针，你可以传递任何一个指针参数到这里，这个参数最  
终会传到回调函数里面，如果不需要传递指针给回调函数，可以填NULL。等下我们再看回  
调函数的写法，以及这个参数的使用。

第5个参数char \*\* errmsg 是错误信息。注意是指针的指针。sqlite3里面有很多固定的错  
误信息。执行 sqlite3\_exec 之后，执行失败时可以查阅这个指针（直接 printf(“%s\n  
”,errmsg)）得到一串字符串信息，这串信息告诉你错在什么地方。sqlite3\_exec函数通  
过修改你传入的指针的指针，把你提供的指针指向错误提示信息，这样sqlite3\_exec函数  
外面就可以通过这个 char\*得到具体错误提示。

说明：通常，sqlite3\_callback 和它后面的 void \* 这两个位置都可以填 NULL。填NULL  
表示你不需要回调。比如你做 insert 操作，做 delete 操作，就没有必要使用回调。而  
当你做 select 时，就要使用回调，因为 sqlite3 把数据查出来，得通过回调告诉你查出  
了什么数据。

i.2 exec 的回调

typedef int (\*sqlite3\_callback)(void\*,int,char\*\*, char\*\*);

你的回调函数必须定义成上面这个函数的类型。下面给个简单的例子：

//sqlite3的回调函数

// sqlite 每查到一条记录，就调用一次这个回调

int LoadMyInfo( void \* para, int n\_column, char \*\* column\_value, char \*\*  
column\_name )

{

//para是你在 sqlite3\_exec 里传入的 void \* 参数

//通过para参数，你可以传入一些特殊的指针（比如类指针、结构指针），然  
后在这里面强制转换成对应的类型（这里面是void\*类型，必须强制转换成你的类型才可用  
）。然后操作这些数据

//n\_column是这一条记录有多少个字段 (即这条记录有多少列)

// char \*\* column\_value 是个关键值，查出来的数据都保存在这里，它实际  
上是个1维数组（不要以为是2维数组），每一个元素都是一个 char \* 值，是一个字段内  
容（用字符串来表示，以\0结尾）

//char \*\* column\_name 跟 column\_value是对应的，表示这个字段的字段名称

//这里，我不使用 para 参数。忽略它的存在.

int i;

printf( “记录包含 %d 个字段\n”, n\_column );

for( i = 0 ; i < n\_column; i ++ )

{

printf( “字段名:%s ß> 字段值:%s\n”, column\_name[i], column\_valu  
e[i] );

}

printf( “------------------\n“ );

return 0;

}

int main( int , char \*\* )

{

sqlite3 \* db;

int result;

char \* errmsg = NULL;

result = sqlite3\_open( “c:\\Dcg\_database.db”, &db );

if( result != SQLITE\_OK )

{

//数据库打开失败

return -1;

}

//数据库操作代码

//创建一个测试表，表名叫 MyTable\_1，有2个字段： ID 和 name。其中ID是一个自动增  
加的类型，以后insert时可以不去指定这个字段，它会自己从0开始增加

result = sqlite3\_exec( db, “create table MyTable\_1( ID integer primary key au  
toincrement, name nvarchar(32) )”, NULL, NULL, errmsg );

if(result != SQLITE\_OK )

{

printf( “创建表失败，错误码:%d，错误原因:%s\n”, result, &errmsg );

}

//插入一些记录

result = sqlite3\_exec( db, “insert into MyTable\_1( name ) values ( ‘走路’ )  
”, 0, 0, errmsg );

if(result != SQLITE\_OK )

{

printf( “插入记录失败，错误码:%d，错误原因:%s\n”, result, &errmsg );

}

result = sqlite3\_exec( db, “insert into MyTable\_1( name ) values ( ‘骑单车’  
)”, 0, 0, errmsg );

if(result != SQLITE\_OK )

{

printf( “插入记录失败，错误码:%d，错误原因:%s\n”, result, &errmsg );

}

result = sqlite3\_exec( db, “insert into MyTable\_1( name ) values ( ‘坐汽车’  
)”, 0, 0, &errmsg );

if(result != SQLITE\_OK )

{

printf( “插入记录失败，错误码:%d，错误原因:%s\n”, result,& errmsg );

}

//开始查询数据库

result = sqlite3\_exec( db, “select \* from MyTable\_1”, LoadMyInfo, NULL, &errm  
sg );

//关闭数据库

sqlite3\_close( db );

return 0;

}

通过上面的例子，应该可以知道如何打开一个数据库，如何做数据库基本操作。

有这些知识，基本上可以应付很多数据库操作了。

i.3 不使用回调查询数据库

上面介绍的 sqlite3\_exec 是使用回调来执行 select 操作。还有一个方法可以直接查询  
而不需要回调。但是，我个人感觉还是回调好，因为代码可以更加整齐，只不过用回调很  
麻烦，你得声明一个函数，如果这个函数是类成员函数，你还不得不把它声明成 static  
的（要问为什么？这又是C++基础了。C++成员函数实际上隐藏了一个参数：this，C++调用  
类的成员函数的时候，隐含把类指针当成函数的第一个参数传递进去。结果，这造成跟前  
面说的 sqlite 回调函数的参数不相符。只有当把成员函数声明成 static 时，它才没有  
多余的隐含的this参数）。

虽然回调显得代码整齐，但有时候你还是想要非回调的 select 查询。这可以通过 sqlit  
e3\_get\_table 函数做到。

int sqlite3\_get\_table(sqlite3\*, const char \*sql, char \*\*\*resultp, int \*nrow, i  
nt \*ncolumn, char \*\*errmsg );

第1个参数不再多说，看前面的例子。

第2个参数是 sql 语句，跟 sqlite3\_exec 里的 sql 是一样的。是一个很普通的以\0结尾  
的char \*字符串。

第3个参数是查询结果，它依然一维数组（不要以为是二维数组，更不要以为是三维数组）  
。它内存布局是：第一行是字段名称，后面是紧接着是每个字段的值。下面用例子来说事  
。

第4个参数是查询出多少条记录（即查出多少行）。

第5个参数是多少个字段（多少列）。

第6个参数是错误信息，跟前面一样，这里不多说了。

下面给个简单例子:

int main( int , char \*\* )

{

sqlite3 \* db;

int result;

char \* errmsg = NULL;

char \*\*dbResult; //是 char \*\* 类型，两个\*号

int nRow, nColumn;

int i , j;

int index;

result = sqlite3\_open( “c:\\Dcg\_database.db”, &db );

if( result != SQLITE\_OK )

{

//数据库打开失败

return -1;

}

//数据库操作代码

//假设前面已经创建了 MyTable\_1 表

//开始查询，传入的 dbResult 已经是 char \*\*，这里又加了一个 & 取地址符，传递进去  
的就成了 char \*\*\*

result = sqlite3\_get\_table( db, “select \* from MyTable\_1”, &dbResult, &nRow,  
&nColumn, &errmsg );

if( SQLITE\_OK == result )

{

//查询成功

index = nColumn; //前面说过 dbResult 前面第一行数据是字段名称，从 nColumn  
索引开始才是真正的数据

printf( “查到%d条记录\n”, nRow );

for( i = 0; i < nRow ; i++ )

{

printf( “第 %d 条记录\n”, i+1 );

for( j = 0 ; j < nColumn; j++ )

{

printf( “字段名:%s ß> 字段值:%s\n”, dbResult[j], dbRes  
ult [index] );

++index; // dbResult 的字段值是连续的，从第0索引到第 nColumn - 1  
索引都是字段名称，从第 nColumn 索引开始，后面都是字段值，它把一个二维的表（传统  
的行列表示法）用一个扁平的形式来表示

}

printf( “-------\n” );

}

}

//到这里，不论数据库查询是否成功，都释放 char\*\* 查询结果，使用 sqlite 提供的功  
能来释放

sqlite3\_free\_table( dbResult );

//关闭数据库

sqlite3\_close( db );

return 0;

}

到这个例子为止，sqlite3 的常用用法都介绍完了。

用以上的方法，再配上 sql 语句，完全可以应付绝大多数数据库需求。

但有一种情况，用上面方法是无法实现的：需要insert、select 二进制。当需要处理二进  
制数据时，上面的方法就没办法做到。下面这一节说明如何插入二进制数据

（2） 操作二进制

sqlite 操作二进制数据需要用一个辅助的数据类型：sqlite3\_stmt \* 。

这个数据类型记录了一个“sql语句”。为什么我把 “sql语句” 用双引号引起来？因为  
你可以把 sqlite3\_stmt \* 所表示的内容看成是 sql语句，但是实际上它不是我们所熟知  
的sql语句。它是一个已经把sql语句解析了的、用sqlite自己标记记录的内部数据结构。

正因为这个结构已经被解析了，所以你可以往这个语句里插入二进制数据。当然，把二进  
制数据插到 sqlite3\_stmt 结构里可不能直接 memcpy ，也不能像 std::string 那样用  
+ 号。必须用 sqlite 提供的函数来插入。

i.1 写入二进制

下面说写二进制的步骤。

要插入二进制，前提是这个表的字段的类型是 blob 类型。我假设有这么一张表：

create table Tbl\_2( ID integer, file\_content blob )

首先声明

sqlite3\_stmt \* stat;

然后，把一个 sql 语句解析到 stat 结构里去：

sqlite3\_prepare( db, “insert into Tbl\_2( ID, file\_content) values( 10, ? )”,  
-1, &stat, 0 );

上面的函数完成 sql 语句的解析。第一个参数跟前面一样，是个 sqlite3 \* 类型变量，  
第二个参数是一个 sql 语句。

这个 sql 语句特别之处在于 values 里面有个 ? 号。在sqlite3\_prepare函数里，?号表  
示一个未定的值，它的值等下才插入。

第三个参数我写的是-1，这个参数含义是前面 sql 语句的长度。如果小于0，sqlite会自  
动计算它的长度（把sql语句当成以\0结尾的字符串）。

第四个参数是 sqlite3\_stmt 的指针的指针。解析以后的sql语句就放在这个结构里。

第五个参数我也不知道是干什么的。为0就可以了。

如果这个函数执行成功（返回值是 SQLITE\_OK 且 stat 不为NULL ），那么下面就可以开  
始插入二进制数据。

sqlite3\_bind\_blob( stat, 1, pdata, (int)(length\_of\_data\_in\_bytes), NULL ); //  
pdata为数据缓冲区，length\_of\_data\_in\_bytes为数据大小，以字节为单位

这个函数一共有5个参数。

第1个参数：是前面prepare得到的 sqlite3\_stmt \* 类型变量。

第2个参数：?号的索引。前面prepare的sql语句里有一个?号，假如有多个?号怎么插入？  
方法就是改变 bind\_blob 函数第2个参数。这个参数我写1，表示这里插入的值要替换 st  
at 的第一个?号（这里的索引从1开始计数，而非从0开始）。如果你有多个?号，就写多个  
bind\_blob 语句，并改变它们的第2个参数就替换到不同的?号。如果有?号没有替换，sq  
lite为它取值null。

第3个参数：二进制数据起始指针。

第4个参数：二进制数据的长度，以字节为单位。

第5个参数：是个析够回调函数，告诉sqlite当把数据处理完后调用此函数来析够你的数据  
。这个参数我还没有使用过，因此理解也不深刻。但是一般都填NULL，需要释放的内存自  
己用代码来释放。

bind完了之后，二进制数据就进入了你的“sql语句”里了。你现在可以把它保存到数据库  
里：

int result = sqlite3\_step( stat );

通过这个语句，stat 表示的sql语句就被写到了数据库里。

最后，要把 sqlite3\_stmt 结构给释放：

sqlite3\_finalize( stat ); //把刚才分配的内容析构掉

i.2 读出二进制

下面说读二进制的步骤。

跟前面一样，先声明 sqlite3\_stmt \* 类型变量：

sqlite3\_stmt \* stat;

然后，把一个 sql 语句解析到 stat 结构里去：

sqlite3\_prepare( db, “select \* from Tbl\_2”, -1, &stat, 0 );

当 prepare 成功之后（返回值是 SQLITE\_OK ），开始查询数据。

int result = sqlite3\_step( stat );

这一句的返回值是 SQLITE\_ROW 时表示成功（不是 SQLITE\_OK ）。

你可以循环执行 sqlite3\_step 函数，一次 step 查询出一条记录。直到返回值不为 SQL  
ITE\_ROW 时表示查询结束。

然后开始获取第一个字段：ID 的值。ID是个整数，用下面这个语句获取它的值：

int stat, 0 ); //第2个参数表示获取第几个字段内容，从0开始计算，因为我的表的ID字  
段是第一个字段，因此这里我填0

下面开始获取 file\_content 的值，因为 file\_content 是二进制，因此我需要得到它的  
指针，还有它的长度：

const void \* pFileContent = sqlite3\_column\_blob( stat, 1 );

int len = sqlite3\_column\_bytes( stat, 1 );

这样就得到了二进制的值。

把 pFileContent 的内容保存出来之后，不要忘了释放 sqlite3\_stmt 结构：

sqlite3\_finalize( stat ); //把刚才分配的内容析构掉

i.3 重复使用 sqlite3\_stmt 结构

如果你需要重复使用 sqlite3\_prepare 解析好的 sqlite3\_stmt 结构，需要用函数： sq  
lite3\_reset。

result = sqlite3\_reset(stat);

这样， stat 结构又成为 sqlite3\_prepare 完成时的状态，你可以重新为它 bind 内容。

www.sqlite.org 网上down下来的 sqlite3.c 文件，直接摸索出这些接口的实现，我认为  
我还没有这个能力。

好在网上还有一些代码已经实现了这个功能。通过参照他们的代码以及不断编译中vc给出  
的错误提示，最终我把整个接口整理出来。

实现这些预留接口不是那么容易，要重头说一次怎么回事很困难。我把代码都写好了，直  
接把他们按我下面的说明拷贝到 sqlite3.c 文件对应地方即可。我在下面也提供了sqlit  
e3.c 文件，可以直接参考或取下来使用。

这里要说一点的是，我另外新建了两个文件：crypt.c和crypt.h。

其中crypt.h如此定义：

#ifndef DCG\_SQLITE\_CRYPT\_FUNC\_

#define DCG\_SQLITE\_CRYPT\_FUNC\_

/\*\*\*\*\*\*\*\*\*\*\*

董淳光写的 SQLITE 加密关键函数库

\*\*\*\*\*\*\*\*\*\*\*/

/\*\*\*\*\*\*\*\*\*\*\*

关键加密函数

\*\*\*\*\*\*\*\*\*\*\*/

int My\_Encrypt\_Func( unsigned char \* pData, unsigned int data\_len, const char  
\* key, unsigned int len\_of\_key );

/\*\*\*\*\*\*\*\*\*\*\*

关键解密函数

\*\*\*\*\*\*\*\*\*\*\*/

int My\_DeEncrypt\_Func( unsigned char \* pData, unsigned int data\_len, const cha  
r \* key, unsigned int len\_of\_key );

#endif

其中的 crypt.c 如此定义：

#include "./crypt.h"

#include "memory.h"

/\*\*\*\*\*\*\*\*\*\*\*

关键加密函数

\*\*\*\*\*\*\*\*\*\*\*/

int My\_Encrypt\_Func( unsigned char \* pData, unsigned int data\_len, const char  
\* key, unsigned int len\_of\_key )

{

return 0;

}

/\*\*\*\*\*\*\*\*\*\*\*

关键解密函数

\*\*\*\*\*\*\*\*\*\*\*/

int My\_DeEncrypt\_Func( unsigned char \* pData, unsigned int data\_len, const cha  
r \* key, unsigned int len\_of\_key )

{

return 0;

}

这个文件很容易看，就两函数，一个加密一个解密。传进来的参数分别是待处理的数据、  
数据长度、密钥、密钥长度。

处理时直接把结果作用于 pData 指针指向的内容。

你需要定义自己的加解密过程，就改动这两个函数，其它部分不用动。扩展起来很简单。

这里有个特点，data\_len 一般总是 1024 字节。正因为如此，你可以在你的算法里使用一  
些特定长度的加密算法，比如AES要求被加密数据一定是128位（16字节）长。这个1024不  
是碰巧，而是 Sqlite 的页定义是1024字节，在sqlite3.c文件里有定义:

# define SQLITE\_DEFAULT\_PAGE\_SIZE 1024

你可以改动这个值，不过还是建议没有必要不要去改它。

上面写了两个扩展函数，如何把扩展函数跟 Sqlite 挂接起来，这个过程说起来比较麻烦  
。我直接贴代码。

分3个步骤。

首先，在 sqlite3.c 文件顶部，添加下面内容：

#ifdef SQLITE\_HAS\_CODEC

#include "./crypt.h"

/\*\*\*\*\*\*\*\*\*\*\*

用于在 sqlite3 最后关闭时释放一些内存

\*\*\*\*\*\*\*\*\*\*\*/

void sqlite3pager\_free\_codecarg(void \*pArg);

#endif

这个函数之所以要在 sqlite3.c 开头声明，是因为下面在 sqlite3.c 里面某些函数里要  
插入这个函数调用。所以要提前声明。

其次，在sqlite3.c文件里搜索“sqlite3PagerClose”函数，要找到它的实现代码（而不  
是声明代码）。

实现代码里一开始是：

#ifdef SQLITE\_ENABLE\_MEMORY\_MANAGEMENT

/\* A malloc() cannot fail in sqlite3ThreadData() as one or more calls to

\*\* malloc() must have already been made by this thread before it gets

\*\* to this point. This means the ThreadData must have been allocated already

\*\* so that ThreadData.nAlloc can be set.

\*/

ThreadData \*pTsd = sqlite3ThreadData();

assert( pPager );

assert( pTsd && pTsd->nAlloc );

#endif

需要在这部分后面紧接着插入：

#ifdef SQLITE\_HAS\_CODEC

sqlite3pager\_free\_codecarg(pPager->pCodecArg);

#endif

这里要注意，sqlite3PagerClose 函数大概也是 3.3.17版本左右才改名的，以前版本里是  
叫 “sqlite3pager\_close”。因此你在老版本sqlite代码里搜索“sqlite3PagerClose”  
是搜不到的。

类似的还有“sqlite3pager\_get”、“sqlite3pager\_unref”、“sqlite3pager\_write”  
、“sqlite3pager\_pagecount”等都是老版本函数，它们在 pager.h 文件里定义。新版本  
对应函数是在 sqlite3.h 里定义（因为都合并到 sqlite3.c和sqlite3.h两文件了）。所  
以，如果你在使用老版本的sqlite，先看看 pager.h 文件，这些函数不是消失了，也不是  
新蹦出来的，而是老版本函数改名得到的。

最后，往sqlite3.c 文件下找。找到最后一行：

/\*\*\*\*\*\*\*\*\*\*\*\*\*\* End of main.c \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
/

在这一行后面，接上本文最下面的代码段。

这些代码很长，我不再解释，直接接上去就得了。

唯一要提的是 DeriveKey 函数。这个函数是对密钥的扩展。比如，你要求密钥是128位，  
即是16字节，但是如果用户只输入 1个字节呢？2个字节呢？或输入50个字节呢？你得对密  
钥进行扩展，使之符合16字节的要求。

DeriveKey 函数就是做这个扩展的。有人把接收到的密钥求md5，这也是一个办法，因为m  
d5运算结果固定16字节，不论你有多少字符，最后就是16字节。这是md5算法的特点。但是  
我不想用md5，因为还得为它添加包含一些 md5 的.c或.cpp文件。我不想这么做。我自己  
写了一个算法来扩展密钥，很简单的算法。当然，你也可以使用你的扩展方法，也而可以  
使用 md5 算法。只要修改 DeriveKey 函数就可以了。

在 DeriveKey 函数里，只管申请空间构造所需要的密钥，不需要释放，因为在另一个函数  
里有释放过程，而那个函数会在数据库关闭时被调用。参考我的 DeriveKey 函数来申请内  
存。

这里我给出我已经修改好的 sqlite3.c 和 sqlite3.h 文件。

如果太懒，就直接使用这两个文件，编译肯定能通过，运行也正常。当然，你必须按我前  
面提的，新建 crypt.h 和 crypt.c 文件，而且函数要按我前面定义的要求来做。

i.3 加密使用方法：

现在，你代码已经有了加密功能。

你要把加密功能给用上，除了改 sqlite3.c 文件、给你工程添加 SQLITE\_HAS\_CODEC 宏，  
还得修改你的数据库调用函数。

前面提到过，要开始一个数据库操作，必须先 sqlite3\_open 。

加解密过程就在 sqlite3\_open 后面操作。

假设你已经 sqlite3\_open 成功了，紧接着写下面的代码：

int i;

//添加、使用密码

i = sqlite3\_key( db, "dcg", 3 );

//修改密码

i = sqlite3\_rekey( db, "dcg", 0 );

用 sqlite3\_key 函数来提交密码。

第1个参数是 sqlite3 \* 类型变量，代表着用 sqlite3\_open 打开的数据库（或新建数据  
库）。

第2个参数是密钥。

第3个参数是密钥长度。

用 sqlite3\_rekey 来修改密码。参数含义同 sqlite3\_key。

实际上，你可以在sqlite3\_open函数之后，到 sqlite3\_close 函数之前任意位置调用 sq  
lite3\_key 来设置密码。

但是如果你没有设置密码，而数据库之前是有密码的，那么你做任何操作都会得到一个返  
回值：SQLITE\_NOTADB，并且得到错误提示：“file is encrypted or is not a databas  
e”。

只有当你用 sqlite3\_key 设置了正确的密码，数据库才会正常工作。

如果你要修改密码，前提是你必须先 sqlite3\_open 打开数据库成功，然后 sqlite3\_key  
设置密钥成功，之后才能用 sqlite3\_rekey 来修改密码。

如果数据库有密码，但你没有用 sqlite3\_key 设置密码，那么当你尝试用 sqlite3\_reke  
y 来修改密码时会得到 SQLITE\_NOTADB 返回值。

如果你需要清空密码，可以使用：

//修改密码

i = sqlite3\_rekey( db, NULL, 0 );

来完成密码清空功能。

i.4 sqlite3.c 最后添加代码段

/\*\*\*

董淳光定义的加密函数

\*\*\*/

#ifdef SQLITE\_HAS\_CODEC

/\*\*\*

加密结构

\*\*\*/

#define CRYPT\_OFFSET 8

typedef struct \_CryptBlock

{

BYTE\* ReadKey; // 读数据库和写入事务的密钥

BYTE\* WriteKey; // 写入数据库的密钥

int PageSize; // 页的大小

BYTE\* Data;

} CryptBlock, \*LPCryptBlock;

#ifndef DB\_KEY\_LENGTH\_BYTE /\*密钥长度\*/

#define DB\_KEY\_LENGTH\_BYTE 16 /\*密钥长度\*/

#endif

#ifndef DB\_KEY\_PADDING /\*密钥位数不足时补充的字符\*/

#define DB\_KEY\_PADDING 0x33 /\*密钥位数不足时补充的字符\*/

#endif

/\*\*\* 下面是编译时提示缺少的函数 \*\*\*/

/\*\* 这个函数不需要做任何处理，获取密钥的部分在下面 DeriveKey 函数里实现 \*\*/

void sqlite3CodecGetKey(sqlite3\* db, int nDB, void\*\* Key, int\* nKey)

{

return ;

}

/\*被sqlite 和 sqlite3\_key\_interop 调用, 附加密钥到数据库.\*/

int sqlite3CodecAttach(sqlite3 \*db, int nDb, const void \*pKey, int nKeyLen);

/\*\*

这个函数好像是 sqlite 3.3.17前不久才加的，以前版本的sqlite里没有看到这个函数

这个函数我还没有搞清楚是做什么的，它里面什么都不做直接返回，对加解密没有影响

\*\*/

void sqlite3\_activate\_see(const char\* right )

{

return;

}

int sqlite3\_key(sqlite3 \*db, const void \*pKey, int nKey);

int sqlite3\_rekey(sqlite3 \*db, const void \*pKey, int nKey);

/\*\*\*

下面是上面的函数的辅助处理函数

\*\*\*/

// 从用户提供的缓冲区中得到一个加密密钥

// 用户提供的密钥可能位数上满足不了要求，使用这个函数来完成密钥扩展

static unsigned char \* DeriveKey(const void \*pKey, int nKeyLen);

//创建或更新一个页的加密算法索引.此函数会申请缓冲区.

static LPCryptBlock CreateCryptBlock(unsigned char\* hKey, Pager \*pager, LPCryp  
tBlock pExisting);

//加密/解密函数, 被pager调用

void \* sqlite3Codec(void \*pArg, unsigned char \*data, Pgno nPageNum, int nMode)  
;

//设置密码函数

int \_\_stdcall sqlite3\_key\_interop(sqlite3 \*db, const void \*pKey, int nKeySize)  
;

// 修改密码函数

int \_\_stdcall sqlite3\_rekey\_interop(sqlite3 \*db, const void \*pKey, int nKeySiz  
e);

//销毁一个加密块及相关的缓冲区,密钥.

static void DestroyCryptBlock(LPCryptBlock pBlock);

static void \* sqlite3pager\_get\_codecarg(Pager \*pPager);

void sqlite3pager\_set\_codec(Pager \*pPager,void \*(\*xCodec)(void\*,void\*,Pgno,int  
),void \*pCodecArg );

//加密/解密函数, 被pager调用

void \* sqlite3Codec(void \*pArg, unsigned char \*data, Pgno nPageNum, int nMode)

{

LPCryptBlock pBlock = (LPCryptBlock)pArg;

unsigned int dwPageSize = 0;

if (!pBlock) return data;

// 确保pager的页长度和加密块的页长度相等.如果改变,就需要调整.

if (nMode != 2)

{

PgHdr \*pageHeader;

pageHeader = DATA\_TO\_PGHDR(data);

if (pageHeader->pPager->pageSize != pBlock->PageSize)

{

CreateCryptBlock(0, pageHeader->pPager, pBlock);

}

}

switch(nMode)

{

case 0: // Undo a "case 7" journal file encryption

case 2: //重载一个页

case 3: //载入一个页

if (!pBlock->ReadKey) break;

dwPageSize = pBlock->PageSize;

My\_DeEncrypt\_Func(data, dwPageSize, pBlock->ReadKey, DB\_KEY\_LENGTH\_BYTE )  
; /\*调用我的解密函数\*/

break;

case 6: //加密一个主数据库文件的页

if (!pBlock->WriteKey) break;

memcpy(pBlock->Data + CRYPT\_OFFSET, data, pBlock->PageSize);

data = pBlock->Data + CRYPT\_OFFSET;

dwPageSize = pBlock->PageSize;

My\_Encrypt\_Func(data , dwPageSize, pBlock->WriteKey, DB\_KEY\_LENGTH\_BYTE )  
; /\*调用我的加密函数\*/

break;

case 7: //加密事务文件的页

/\*在正常环境下, 读密钥和写密钥相同. 当数据库是被重新加密的,读密钥和写密钥  
未必相同.

回滚事务必要用数据库文件的原始密钥写入.因此,当一次回滚被写入,总是用数据库  
的读密钥,

这是为了保证与读取原始数据的密钥相同.

\*/

if (!pBlock->ReadKey) break;

memcpy(pBlock->Data + CRYPT\_OFFSET, data, pBlock->PageSize);

data = pBlock->Data + CRYPT\_OFFSET;

dwPageSize = pBlock->PageSize;

My\_Encrypt\_Func( data, dwPageSize, pBlock->ReadKey, DB\_KEY\_LENGTH\_BYTE );  
/\*调用我的加密函数\*/

break;

}

return data;

}

//销毁一个加密块及相关的缓冲区,密钥.

static void DestroyCryptBlock(LPCryptBlock pBlock)

{

//销毁读密钥.

if (pBlock->ReadKey){

sqliteFree(pBlock->ReadKey);

}

//如果写密钥存在并且不等于读密钥,也销毁.

if (pBlock->WriteKey && pBlock->WriteKey != pBlock->ReadKey){

sqliteFree(pBlock->WriteKey);

}

if(pBlock->Data){

sqliteFree(pBlock->Data);

}

//释放加密块.

sqliteFree(pBlock);

}

static void \* sqlite3pager\_get\_codecarg(Pager \*pPager)

{

return (pPager->xCodec) ? pPager->pCodecArg: NULL;

}

// 从用户提供的缓冲区中得到一个加密密钥

static unsigned char \* DeriveKey(const void \*pKey, int nKeyLen)

{

unsigned char \* hKey = NULL;

int j;

if( pKey == NULL || nKeyLen == 0 )

{

return NULL;

}

hKey = sqliteMalloc( DB\_KEY\_LENGTH\_BYTE + 1 );

if( hKey == NULL )

{

return NULL;

}

hKey[ DB\_KEY\_LENGTH\_BYTE ] = 0;

if( nKeyLen < DB\_KEY\_LENGTH\_BYTE )

{

memcpy( hKey, pKey, nKeyLen ); //先拷贝得到密钥前面的部分

j = DB\_KEY\_LENGTH\_BYTE - nKeyLen;

//补充密钥后面的部分

memset( hKey + nKeyLen, DB\_KEY\_PADDING, j );

}

else

{ //密钥位数已经足够,直接把密钥取过来

memcpy( hKey, pKey, DB\_KEY\_LENGTH\_BYTE );

}

return hKey;

}

//创建或更新一个页的加密算法索引.此函数会申请缓冲区.

static LPCryptBlock CreateCryptBlock(unsigned char\* hKey, Pager \*pager, LPCryp  
tBlock pExisting)

{

LPCryptBlock pBlock;

if (!pExisting) //创建新加密块

{

pBlock = sqliteMalloc(sizeof(CryptBlock));

memset(pBlock, 0, sizeof(CryptBlock));

pBlock->ReadKey = hKey;

pBlock->WriteKey = hKey;

pBlock->PageSize = pager->pageSize;

pBlock->Data = (unsigned char\*)sqliteMalloc(pBlock->PageSize + CRYPT\_OFFS  
ET);

}

else //更新存在的加密块

{

pBlock = pExisting;

if ( pBlock->PageSize != pager->pageSize && !pBlock->Data){

sqliteFree(pBlock->Data);

pBlock->PageSize = pager->pageSize;

pBlock->Data = (unsigned char\*)sqliteMalloc(pBlock->PageSize + CRYPT  
\_OFFSET);

}

}

memset(pBlock->Data, 0, pBlock->PageSize + CRYPT\_OFFSET);

return pBlock;

}

/\*

\*\* Set the codec for this pager

\*/

void sqlite3pager\_set\_codec(

Pager \*pPager,

void \*(\*xCodec)(void\*,void\*,Pgno,int),

void \*pCodecArg

)

{

pPager->xCodec = xCodec;

pPager->pCodecArg = pCodecArg;

}

int sqlite3\_key(sqlite3 \*db, const void \*pKey, int nKey)

{

return sqlite3\_key\_interop(db, pKey, nKey);

}

int sqlite3\_rekey(sqlite3 \*db, const void \*pKey, int nKey)

{

return sqlite3\_rekey\_interop(db, pKey, nKey);

}

/\*被sqlite 和 sqlite3\_key\_interop 调用, 附加密钥到数据库.\*/

int sqlite3CodecAttach(sqlite3 \*db, int nDb, const void \*pKey, int nKeyLen)

{

int rc = SQLITE\_ERROR;

unsigned char\* hKey = 0;

//如果没有指定密匙,可能标识用了主数据库的加密或没加密.

if (!pKey || !nKeyLen)

{

if (!nDb)

{

return SQLITE\_OK; //主数据库, 没有指定密钥所以没有加密.

}

else //附加数据库,使用主数据库的密钥.

{

//获取主数据库的加密块并复制密钥给附加数据库使用

LPCryptBlock pBlock = (LPCryptBlock)sqlite3pager\_get\_codecarg(sqli  
te3BtreePager(db->aDb[0].pBt));

if (!pBlock) return SQLITE\_OK; //主数据库没有加密

if (!pBlock->ReadKey) return SQLITE\_OK; //没有加密

memcpy(pBlock->ReadKey, &hKey, 16);

}

}

else //用户提供了密码,从中创建密钥.

{

hKey = DeriveKey(pKey, nKeyLen);

}

//创建一个新的加密块,并将解码器指向新的附加数据库.

if (hKey)

{

LPCryptBlock pBlock = CreateCryptBlock(hKey, sqlite3BtreePager(db->aDb  
[nDb].pBt), NULL);

sqlite3pager\_set\_codec(sqlite3BtreePager(db->aDb[nDb].pBt), sqlite3Cod  
ec, pBlock);

rc = SQLITE\_OK;

}

return rc;

}

// Changes the encryption key for an existing database.

int \_\_stdcall sqlite3\_rekey\_interop(sqlite3 \*db, const void \*pKey, int nKeySiz  
e)

{

Btree \*pbt = db->aDb[0].pBt;

Pager \*p = sqlite3BtreePager(pbt);

LPCryptBlock pBlock = (LPCryptBlock)sqlite3pager\_get\_codecarg(p);

unsigned char \* hKey = DeriveKey(pKey, nKeySize);

int rc = SQLITE\_ERROR;

if (!pBlock && !hKey) return SQLITE\_OK;

//重新加密一个数据库,改变pager的写密钥, 读密钥依旧保留.

if (!pBlock) //加密一个未加密的数据库

{

pBlock = CreateCryptBlock(hKey, p, NULL);

pBlock->ReadKey = 0; // 原始数据库未加密

sqlite3pager\_set\_codec(sqlite3BtreePager(pbt), sqlite3Codec, pBlock);

}

else // 改变已加密数据库的写密钥

{

pBlock->WriteKey = hKey;

}

// 开始一个事务

rc = sqlite3BtreeBeginTrans(pbt, 1);

if (!rc)

{

// 用新密钥重写所有的页到数据库。

Pgno nPage = sqlite3PagerPagecount(p);

Pgno nSkip = PAGER\_MJ\_PGNO(p);

void \*pPage;

Pgno n;

for(n = 1; rc == SQLITE\_OK && n <= nPage; n ++)

{

if (n == nSkip) continue;

rc = sqlite3PagerGet(p, n, &pPage);

if(!rc)

{

rc = sqlite3PagerWrite(pPage);

sqlite3PagerUnref(pPage);

}

}

}

// 如果成功，提交事务。

if (!rc)

{

rc = sqlite3BtreeCommit(pbt);

}

// 如果失败，回滚。

if (rc)

{

sqlite3BtreeRollback(pbt);

}

// 如果成功，销毁先前的读密钥。并使读密钥等于当前的写密钥。

if (!rc)

{

if (pBlock->ReadKey)

{

sqliteFree(pBlock->ReadKey);

}

pBlock->ReadKey = pBlock->WriteKey;

}

else// 如果失败，销毁当前的写密钥，并恢复为当前的读密钥。

{

if (pBlock->WriteKey)

{

sqliteFree(pBlock->WriteKey);

}

pBlock->WriteKey = pBlock->ReadKey;

}

// 如果读密钥和写密钥皆为空，就不需要再对页进行编解码。

// 销毁加密块并移除页的编解码器

if (!pBlock->ReadKey && !pBlock->WriteKey)

{

sqlite3pager\_set\_codec(p, NULL, NULL);

DestroyCryptBlock(pBlock);

}

return rc;

}

/\*\*\*

下面是加密函数的主体

\*\*\*/

int \_\_stdcall sqlite3\_key\_interop(sqlite3 \*db, const void \*pKey, int nKeySize)

{

return sqlite3CodecAttach(db, 0, pKey, nKeySize);

}

// 释放与一个页相关的加密块

void sqlite3pager\_free\_codecarg(void \*pArg)

{

if (pArg)

DestroyCryptBlock((LPCryptBlock)pArg);

}

#endif //#ifdef SQLITE\_HAS\_CODEC

五、 后记  
写此教程，可不是一个累字能解释。

但是我还是觉得欣慰的，因为我很久以前就想写 sqlite 的教程，一来自己备忘，二而已  
造福大众，大家不用再走弯路。

本人第一次写教程，不足的地方请大家指出。

本文可随意转载、修改、引用。但无论是转载、修改、引用，都请附带我的名字：董淳光  
。以示对我劳动的肯定

(源码网整理：[www.codepub.com](http://www.codepub.com/))