SQLITE3 基础教程

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
```

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 windows 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。二是 sqlite3 **，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 sqlite 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，sqlite 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_value[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 autoincrement, 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, errmsg );
//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

sqlite3_get_table 函数做到。

int sqlite3_get_table(sqlite3*, const char *sql, char ***resultp, int *nrow, int *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], dbResult [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 stat 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，sqlite 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 SQLITE_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： sqlite3_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 sqlite3.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 char * 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 char * 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 md5 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 sqlite3_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 database”。

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_rekey 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, LPCryptBlock 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 nKeySize);
//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 nKeySize)
{
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。