SQLite3 源程序分析

作者：空转

本文分析的 SQLite 版本为 3.6.18。现在已经变成 3.6.20 了，但本文中所涉及的内容变化不大。读者最好能下载一个源程序，然后将本文与源程序对照阅读。这样也有利于发现本文的错误，说实话吧，我写的时候是连分析带猜的，错误肯定很多。

参考文献：
1-The Definitive Guide to SQLite . Michael Owens：比较经典的 SQLite 著作。我边看边翻译了其中的部分内容，但翻得不好，大家还是看原文吧。
2-SQLite 文件格式分析_v102 . 空转：我写的，写得特好。现在是 v102 版，跟前面的版本相比增加了不少背景知识，对文件格式的介绍算是很全面了。看本文之前，应该先浏览一下此参考文献。

1. SQLite3 程序分析

1.1. 主程序流程

所谓“主程序”是指 SQLite 所提供的命令行处理程序（CLP）。通过对它的分析可以对 SQLite 源程序建立整体概念，比一上来就直接分析单独 API 的处理过程要容易。CLP 的主要程序都在 shell.c 中。
CLP 的执行流程很简单：循环接受用户输入的 SQL 命令，处理 SQL 命令。命令的执行都是调用 sqlite3_exec()函数完成，也就是采用的是“执行封装的 Query”的形式[1]。
程序定义了一个功能比较强大的回叫函数来处理 SQL 命令执行的返回结果：

```
static int callback(void *pArg, int nArg, char **azArg, char **azCol);
```

程序定义了 9 种回显的形式，通过一个 callback_data 结构来对回显参数进行配置。

1.1.1. 程序主函数

程序的 main()函数在 shell.c 的尾部，简化后的 main()函数的执行过程主要分为 5 步：

1. 设置回显参数
2. 取数据库文件名
3. 打开数据库
4. 循环处理 SQL 命令
5. 关闭数据库

如下：

```
int main(int argc, char **argv){
    struct callback_data data;      //回显参数
    int rc = 0;
```

```
    Argv0 = argv[0];
    main_init(&data);                //设置默认的回显形式

    //取数据库文件名，如没有，默认为内存数据库
    data.zDbFilename = argv[1];

    data.out = stdout;

    /* 如果数据库文件存在，则打开它。
    ** 如果不存在，先不打开（现在什么都不做），
    ** 可以防止用户因错误的输入而创建空文件。
    */
    if( access(data.zDbFilename, 0)==0 ){
        open_db(&data);
    }

    printf(
        "SQLite version %s\n"
        "Enter \".help\" for instructions\n"
        "Enter SQL statements terminated with a \";\"\n",
        sqlite3_libversion()
        );
    rc = process_input(&data, 0);

    if( db ){                        //关闭数据库
        if( sqlite3_close(db)!=SQLITE_OK ){
            fprintf(stderr,"error closing database: %s\n", sqlite3_errmsg(db));
        }
    }
    return rc;
}
```

说明：上述函数与源程序相比做了很大的简化，去掉的部分不是不重要的，而是“可以不解释”的。实用程序的流程一般都是复杂的，SQLite 也不例外。本文按照自己的主线进行介绍，只求能说明问题（自圆其说），主线之外的东西，不管重不重要，都尽量忽略。后面的函数也存在这样情况，就不再说明了。

回显参数的设置就不再介绍了，参考源程序的 callback()函数和 callback_data 结构，有关回叫函数的使用见参考文献一。下面介绍数据库的打开过程。

1.1.2. 打开数据库

数据库文件的打开过程在 SQLite 的权威文档中有介绍，过程如下图：

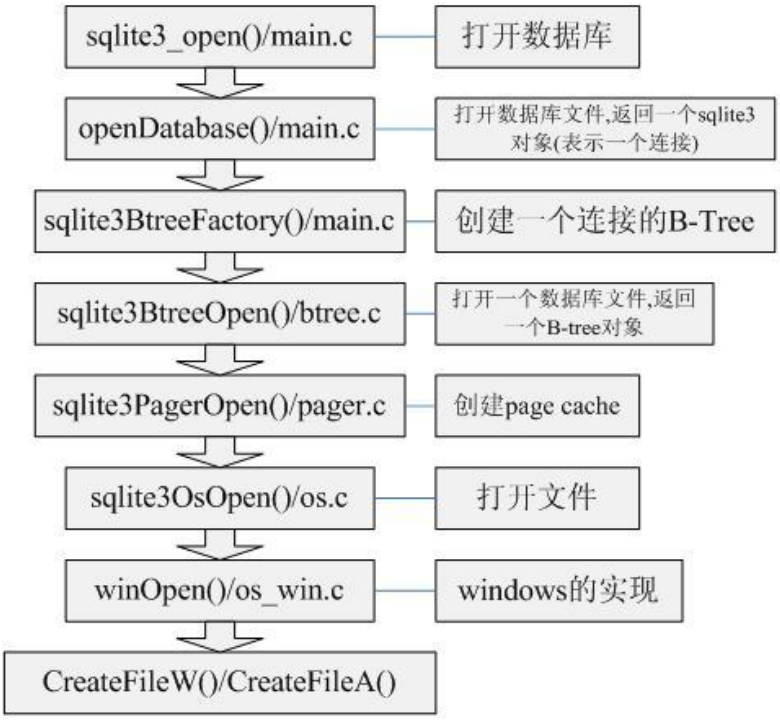

图 1-1 数据库文件的打开过程

在 CLP 中打开数据库，比上图又多了两层，其调用层次如下：
1-main()：
位于 shell.c。
从命令行参数中得到数据库名，如果数据库文件存在，则打开它。

2-open_db()：
位于 shell.c。
功能：确认数据库是否已经打开。如果已打开，则什么都不做。如果没有，则打开它。如果打开失败，输出一个错误信息。

3-sqlite3_open()：
位于 main.c。
功能：打开一个数据库。
该函数中只包含对 opendatabase()的调用，但调用的参数与 sqlite3_open_v2()所使用的参数不同。

4-opendatabase()：
位于 main.c。
功能：这个函数为 sqlite3_open()和 sqlite3_open16()工作，打开一个数据库。数据库文件名"zFilename"采用 UTF-8 编码。
先生成各类标志什么的，然后生成默认的排序法。当需要生成数据库后台驱动时，调用 sqlite3BtreeFactory()。

在此函数中真正分配 sqlite 结构的空间：db = sqlite3MallocZero(sizeof(sqlite3))。

在调用 sqlite3BtreeFactory()之前，需要对 db 的一些域进行设置。

5-sqlite3BtreeFactory()
位于 main.c。
功能：本函数创建到数据库 BTree 驱动的连接。如果 zFilename 是文件名，则打开并使用它。
如果 zFilename 是":memory:"，则使用内存数据库（在连接断开时释放）。
如果 zFilename 为空且数据库是虚拟（virtual）的，则只是暂时使用，在连接断开时被删除。
虚拟数据库可以是磁盘文件或就在内存中，由 sqlite3TempInMemory()函数来决定是哪一种情况。

6-sqlite3BtreeOpen()：
位于 btree.c。
功能：打开一个数据库文件。
由于在 sqlite3BtreeFactory()中已经调用过 sqlite3TempInMemory()函数，所以此处逻辑稍简单了一些。
zFilename 是数据库文件名。如果 zFilename 为空，创建一个具有随机文件名的数据库，这个数据库会在调用 sqlite3BtreeClose()时被删除。
如果 zFilename 是":memory:"，创建内存数据库，并在关闭时被释放。
如果此 Btree 是共享缓冲区的候选者，则尝试寻找一个已存在的 BtShared 来共享。（参本文后面关于内存数据结构的介绍）
如果不是共享缓冲区的候选者或未找到已存在的 BtShared，则调用 sqlite3PagerOpen()函数打开文件。
文件打开之后，调用 sqlite3PagerReadFileheader()来读文件头中的配置信息。

7-sqlite3PagerOpen()：
位于 pager.c。
功能：分配并初始化一个新 Pager 对象，将其指针放到*ppPager。
该 pager 会在调用 sqlite3PagerClose()时被释放。
zFilename 参数是要打开的数据库文件的路径。
如果 zFilename 为空，创建随机文件名的文件。
如果 zFilename 为":memory:"，所有信息都放到缓冲区中，不会被写入磁盘。这用来实现内存数据库。
如果 pager 对象已分配且指定文件打开成功，返回 SQLITE_OK 并将*ppPager 指向新 pager 对象。
如果有错误发生，*ppPager 置空并返回错误代码。
执行过程是：先申请空间，再调用 sqlite3OsOpen()打开文件（如果需要），再根据打开的文件设置内存。

8-sqlite3OsOpen()：
位于 os.c。
功能：打开一个文件，与具体的操作系统无关。
是一种 VFS 封装。VFS 的意思是"virtual file system"，虚拟文件系统。
本函数只有几条语句，只有一条关键语句：
rc = pVfs->xOpen(pVfs, zPath, pFile, flags & 0x7f1f, pFlagsOut);

对于 Win32 操作系统，该语句实际调用的是 winOpen()函数。

9-winOpen()：
位于 os_win.c。
功能：打开一个 Windows 操作系统文件。
先将文件名转换为操作系统所使用的编码。再设置一系列参数。
最终调用操作系统函数 CreateFileA()打开文件。

10-CreateFileA()：
位于 WINBASE.H。
功能：
打开文件名所指定的文件。如果文件不存在，则创建。

1.1.3. 循环处理 SQL 命令

SQL 命令的处理是由 process_input()函数完成的。该函数还完成”.”命令的处理，这我们就不管了。简化后的 process_input()函数如下：

```
static int process_input(struct callback_data *p, FILE *in){
    while( 1 ){
        zLine = one_input_line(zSql, in);
        if( zLine && zLine[0]=='.' && nSql==0 ){
            rc = do_meta_command(zLine, p);
            continue;
        }
        rc = sqlite3_exec(p->db, zSql, callback, p, &zErrMsg);
        if( rc || zErrMsg ){
            处理出错信息;
        }
    }
    return errCnt;
}
```

这么简化应该就不用解释了。

1.2. SQL 命令编译与执行的过程

1.2.1. sqlite3_exec()函数

函数 sqlite3_exec()位于文件 legacy.c 的尾部，其函数头为：

```
int sqlite3_exec(
  sqlite3 *db,                /* 一个打开的数据库连接 */
  const char *zSql,           /* 要执行的 SQL 语句 */
  sqlite3_callback xCallback, /* 回叫函数 */
```

```
  void *pArg,                  /* 传递给 xCallback()的第一个参数 */
  char **pzErrMsg              /* 将错误信息写到*pzErrMsg 中 */
)
```

sqlite3_exec()函数一次可以执行多条 SQL 命令。执行完成后返回一个 SQLITE_ success/failure 代码，还会将错误信息写到*pzErrMsg 中。如果 SQL 是查询，查询结果中的每一行都会调用 xCallback()函数。pArg 为传递给 xCallback()的第一个参数。如果 xCallback==NULL，即使对查询命令也没有回叫调用。

sqlite3_exec()函数的实现体现了一个典型的、实用的 SQL 语句处理过程，我认为对应用程序的开发很有借鉴意义，所以就不过多简化了，去掉一些测试代码，增加一些注释，源程序基本如下：

```
int sqlite3_exec(
  sqlite3 *db,                 /* 一个打开的数据库连接 */
  const char *zSql,            /* 要执行的 SQL 语句 */
  sqlite3_callback xCallback, /* 回叫函数 */
  void *pArg,                  /* 传递给 xCallback()的第一个参数 */
  char **pzErrMsg              /* 将错误信息写到*pzErrMsg 中 */
){
  int rc = SQLITE_OK;          /* 返回码 */
  const char *zLeftover;       /* 未处理的 SQL 串尾部。zSql 中可能包含多个 SQL
                               语句，一次处理一个，此变量为剩下的还未处理的
                               语句。 */
  sqlite3_stmt *pStmt = 0;     /* 当前 SQL 语句（对象） */
  char **azCols = 0;           /* 结果字段(s)的名称 */
  int nRetry = 0;              /* 重试的次数 */
  int callbackIsInit;          /* 如果初始化了回叫函数，为 true */

  if( zSql==0 ) zSql = "";

  sqlite3Error(db, SQLITE_OK, 0);    /* 清除 db 中的错误信息 */
  while( (rc==SQLITE_OK || (rc==SQLITE_SCHEMA && (++nRetry)<2)) && zSql[0] ){
    int nCol;
    char **azVals = 0;

    pStmt = 0;
    rc = sqlite3_prepare(db, zSql, -1, &pStmt, &zLeftover);/* 编译一条语句 */
    if( rc!=SQLITE_OK ){
      continue;
    }
    if( !pStmt ){
      /* 遇到注释时会执行此分支 */
      zSql = zLeftover;
      continue;
    }
```

```
callbackIsInit = 0;
nCol = sqlite3_column_count(pStmt); /* 取字段数 */

while( 1 ){
  int i;
  rc = sqlite3_step(pStmt);     /* 执行语句 */

  /* 如果有回叫函数并且需要，则调用回叫函数 */
  if( xCallback && (SQLITE_ROW==rc ||
      (SQLITE_DONE==rc && !callbackIsInit
                       && db->flags&SQLITE_NullCallback)) ){
    /* 1-如果回叫函数未初始化，则初始化之 */
    if( !callbackIsInit ){
      /* 此分支只执行一次 */
      azCols = sqlite3DbMallocZero(db, 2*nCol*sizeof(const char*) + 1);
      if( azCols==0 ){
        goto exec_out;
      }
      for(i=0; i<nCol; i++){
        /* 取各字段的名称 */
        azCols[i] = (char *)sqlite3_column_name(pStmt, i);
      }
      callbackIsInit = 1;
    }
    /* 2-如果返回的是记录 */
    if( rc==SQLITE_ROW ){
      azVals = &azCols[nCol];
      for(i=0; i<nCol; i++){
        /* 取各字段的值 */
        azVals[i] = (char *)sqlite3_column_text(pStmt, i);
        if( !azVals[i] && sqlite3_column_type(pStmt, i)!=SQLITE_NULL ){
          db->mallocFailed = 1;
          goto exec_out;
        }
      }
    }
    /* 3-调用回叫函数对返回的记录进行处理 */
    if( xCallback(pArg, nCol, azVals, azCols) ){
      rc = SQLITE_ABORT;
      sqlite3VdbeFinalize((Vdbe *)pStmt);
      pStmt = 0;
      sqlite3Error(db, SQLITE_ABORT, 0);
      goto exec_out;
    }
```

```
      }

      /*
      如果返回的不是记录，有两种情况：一种是到达结果记录集的结尾，
      第二种是执行 create table 一类的不返回记录集的命令。
      无论哪种情况，此处都需要“定案”。
      */
      if( rc!=SQLITE_ROW ){
        rc = sqlite3VdbeFinalize((Vdbe *)pStmt);
        pStmt = 0;
        if( rc!=SQLITE_SCHEMA ){
          nRetry = 0;
          zSql = zLeftover;
          while( sqlite3Isspace(zSql[0]) ) zSql++;
        }
        break;
      }
    }

    sqlite3DbFree(db, azCols);
    azCols = 0;
  }

exec_out:
  if( pStmt ) sqlite3VdbeFinalize((Vdbe *)pStmt);
  sqlite3DbFree(db, azCols);

  rc = sqlite3ApiExit(db, rc);
  /* 对出错信息进行处理 */
  if( rc!=SQLITE_OK && ALWAYS(rc==sqlite3_errcode(db)) && pzErrMsg ){
    int nErrMsg = 1 + sqlite3Strlen30(sqlite3_errmsg(db));
    *pzErrMsg = sqlite3Malloc(nErrMsg);
    if( *pzErrMsg ){
      memcpy(*pzErrMsg, sqlite3_errmsg(db), nErrMsg);
    }else{
      rc = SQLITE_NOMEM;
      sqlite3Error(db, SQLITE_NOMEM, 0);
    }
  }else if( pzErrMsg ){
    *pzErrMsg = 0;
  }

  return rc;
}
```

1.2.2. SQL 语句编译的调用层次

当调用 sqlite3_prepare()函数时，编译一条 SQL 语句。编译过程的调用层次如下：
1- sqlite3_prepare()
在 prepare.c 中。
SQLite 现在提供两个版本的编译 API 函数：遗留的和现在使用的。
在遗留版本中，原始 SQL 文本没有保存在编译后的语句（sqlite3_stmt 结构）中，因此，如果 schema 发生改变，sqlite3_step()会返回 SQLITE_SCHEMA。在新版本中，编译后的语句中保存原始 SQL 文本，当遇到 schema 改变时自动重新编译。
sqlite3_prepare()函数中其实只包含一条对 sqlite3LockAndPrepare（）的调用语句：

```
  rc = sqlite3LockAndPrepare(db,zSql,nBytes,0,ppStmt,pzTail);
```

其中第 4 个参数为 0，表示不将 SQL 文本复制到 ppStmt 中。
空注：源程序中紧跟此函数的 sqlite3_prepare_v2()函数中在调用 sqlite3LockAndPrepare（）时第 4 个参数为 1，不知与上述解释是否矛盾。

2- sqlite3LockAndPrepare()
在 prepare.c 中。结合注释，很简单，也很清晰。

```
static int sqlite3LockAndPrepare(
  sqlite3 *db,              /* 数据库句柄 */
  const char *zSql,         /* UTF-8 编码的 SQL 语句 */
  int nBytes,               /* zSql 的字节数 */
  int saveSqlFlag,          /* 如果为 True，将 SQL 文本复制到 sqlite3_stmt 中。 */
  sqlite3_stmt **ppStmt,    /* OUT: 指向语句句柄 */
  const char **pzTail       /* OUT: 未处理的 SQL 串 */
){
  int rc;
  *ppStmt = 0;
  if( !sqlite3SafetyCheckOk(db) ){  /* 确定 db 指针的合法性。 */
    return SQLITE_MISUSE;
  }

  /* 将 UTF-8 编码的 SQL 语句 zSql 编译成。 */
  rc = sqlite3Prepare(db, zSql, nBytes, saveSqlFlag, ppStmt, pzTail);
  if( rc==SQLITE_SCHEMA ){  /* 如果遇到 SCHEMA 改变，定案，再编译 */
    sqlite3_finalize(*ppStmt);
    rc = sqlite3Prepare(db, zSql, nBytes, saveSqlFlag, ppStmt, pzTail);
  }

  return rc;
}
```

3- sqlite3Prepare()
在 prepare.c 中。

很长的函数，在其中调用 sqlite3RunParser()函数，在给定的 SQL 字符串上执行分析器。
函数中，先创建 Parse 结构、加锁什么的，到调用 sqlite3RunParser()函数时参数反而很简单了：

```
sqlite3RunParser(pParse, zSql, &zErrMsg);
```

此处 zSql 是一个完整的 SQL 语句串。
调用返回后还要做一系列处理，略。

4- sqlite3RunParser()
在 tokenize.c 中。
功能：在给定的 SQL 字符串上执行分析器。传入一个 parser 结构。返回一个 SQLITE_状态码。如果有错误发生，将错误信息写入*pzErrMsg。
本函数内部是一个循环语句，每次循环处理一个词，根据词的类型做出不同的处理。如果是正经的词（不是空格什么的），都会调用 sqlite3Parser()函数对其进行分析。

5- sqlite3Parser()
在 parse.c 中。
本函数为分析器主程序。
parse.c 中的程序好象都是自动生成的，我反正是看不懂，也就不想看了。摘一段与兄弟们共享：

```
if( yypParser->yyidx < 0 || yymajor==0 ){
  yy_destructor(yypParser,(YYCODETYPE)yymajor,&yyminorunion);
  yy_parse_failed(yypParser);
  yymajor = YYNOCODE;
}else if( yymx!=YYERRORSYMBOL ){
  YYMINORTYPE u2;
  u2.YYERRSYMDT = 0;
  yy_shift(yypParser,yyact,YYERRORSYMBOL,&u2);
}
```

1.2.3. 查询的执行过程

前一小节介绍的编译调用层次看起来还是很清晰的，但实际执行时情况要复杂得多。
比如 Oracle 一类的数据库，以服务器的形式供客户端访问。在服务器启动的过程中可以完成所有必要的初始化工作，在解析 SQL 语句时逻辑可能反而简单一些。（Oracle 好像也有一些东西在第一次调用时才加载，比如 Java 虚拟机什么的）。
SQLite 这样的数据库主要是提供 API 供应用程序调用，这就要求在一次单独的调用中要完成所有相关工作。另外，SQLite 好像更倾向于将工作留到不得不做时再做（即使不一定非得这样），所以在 SQLite 中经常会看到“如果还没创建，则创建”或“如果还没打开，则打开”一类的代码。这样，程序的旁枝就会比较多，有时读起来会有一定困难。比如，在 SQLite 启动后第 1 次执行 select 语句时，在编译该语句的过程中需要完成 schema 信息内存初始化的全部工作。
下面我们就跟踪一条最简单的 select 语句的执行过程，从中可以了解 SQLite 的运行机制。
首先要准备数据库。
创建一个新的数据库，创建一个表：

```
create table d (
   id integer primary key,
   name text,
   loca text );
```
向表中插入 4 条记录：
```
insert into d (name,loca) values ('accounting','Beijing');
insert into d (name,loca) values ('research','Nanjing');
insert into d (name,loca) values ('marketing','Xining');
insert into d (name,loca) values ('operation','Baoding');
```
执行下列命令：
```
.m col
.h on
.w 4 15 3 3 3 20 3
explain select * from d;
```
返回结果如下：

addr	opcode	p1	p2	p3	p4	p5	comment
0	Trace	0	0	0		00	
1	Goto	0	11	0		00	
2	OpenRead	0	2	0	3	00	
3	Rewind	0	9	0		00	
4	Rowid	0	1	0		00	
5	Column	0	1	2		00	
6	Column	0	2	3		00	
7	ResultRow	1	3	0		00	
8	Next	0	4	0		01	
9	Close	0	0	0		00	
10	Halt	0	0	0		00	
11	Transaction	0	0	0		00	
12	VerifyCookie	0	2	0		00	
13	TableLock	0	2	0	d	00	
14	Goto	0	2	0		00	

上面显示了一条 select 语句经编译后所生成的 VDBE 程序。有关 VDBE 程序的介绍请参考《SQLite 权威指南》[1]。其中相关介绍好像有些过时，主要是由于这部分程序变化比较快，但还是很有参考价值的，反正我看了那部分内容之后，上面的程序就能看懂个大概意思了。

下面，我们就在此数据库基础上跟踪查询语句 select * from d 的处理过程。主要是罗列在处理过程所执行过的函数。每次调用的相关说明并不多，有的只说明关键变量的值，有的简单说明执行过程。主要是调用的太多了，实在没法对每次调用都详细说明。读者最好按上面的方法创建示例数据库，然后边看边跟踪执行。
注意：函数前面的数字表示调用的层次，而不是序号。

1-sqlite3_exec:
zSql="select * from d;"

2-sqlite3Prepare:
zSql="select * from d;"
调用 sqlite3RunParser。

3-sqlite3RunParser：
每处理一个单词，调用一次 sqlite3Parser。
当语句处理完毕，语句串变为""，最后一次调用 sqlite3Parser。
在 sqlite3Parser 中，后部有一个 do while 循环。循环了好多遍，下面一句也执行了好多遍：
    yy_reduce(yypParser,yyact-YYNSTATE);
yy_reduce 中有一个大的 switch 语句，每次调用执行的分支不同。终于有一遍中调用了 sqlite3Select。
sqlite3Select 是 select 语句的处理主程序，在其中又经过如下调用层次（太多，这些层次就没编号了）：
sqlite3Select(在 select.c 中)
↓
sqlite3SelectPrep(在 select.c 中)
↓
sqlite3SelectExpand(在 select.c 中)
↓
sqlite3WalkSelect(在 walker.c 中)
↓
selectExpander(在 select.c 中)
↓
sqlite3LocateTable(在 build.c 中)
↓
sqlite3ReadSchema(在 prepare.c 中)
↓
sqlite3Init(在 prepare.c 中)。

4-sqlite3Init：
功能：
初始化所有数据库文件——主数据库、临时数据库和所有附加的数据库。返回成功码。如果有错误发生，将错误信息写入*pzErrMsg。
执行：
进入第一个循环语句。在循环语句中调用 sqlite3InitOne。
需要注意的是程序中有一小句很重要：
  db->init.busy = 1;
当 db->init.busy 被设为 1 时，就不会再有 VDBE 代码生成或执行。后面就可以在回叫函数中通过执行系统表中的 create 语句的方式为对象创建内部数据结构而又不会实际地执行这些创建语句。

5-sqlite3InitOne：
(在 prepare.c 中)

功能：
读入一个单独数据库文件的 schema，并初始化内部的数据结构。
执行：
调用回叫函数 sqlite3InitCallback，执行系统表的创建语句，为系统表创建内部数据结构。

6-sqlite3InitCallback：
(在 prepare.c 中)
功能：
本函数是初始化数据库时的回叫程序。
执行：
调用 sqlite3_exec。

7-sqlite3_exec:
zSql=CREATE TABLE sqlite_master(
  type text,
  name text,
  tbl_name text,
  rootpage integer,
  sql text
)

8-sqlite3Prepare:
zSql=CREATE TABLE sqlite_master(
  type text,
  name text,
  tbl_name text,
  rootpage integer,
  sql text
)

7-sqlite3_exec:
从 sqlite3Prepare 返回后，执行到 sqlite3_step 一句。

8-sqlite3_step：
sqlite3_step 是顶层函数，它调用 sqlite3Step 完成主要工作。

9-sqlite3Step：
此函数中调用了 sqlite3VdbeExec。

10-sqlite3VdbeExec:
p->zSql=""
p->nOp=2
p->aOp:
0 21 OP_Trace

1　40　OP_Halt
可见，不执行实际的创建功能，直接返回。

9-sqlite3Step：
回到 sqlite3Step 后，发现 sqlite3VdbeExec 调用的返回结果为 101。
rc 又与另一个数“与”操作后，值为 21，返回。

8-sqlite3_step：
rc 值在其后的运算中变为 101，返回。

7-sqlite3_exec:
rc!=SQLITE_ROW，不回显，退出循环。返回。

6-sqlite3InitCallback：
返回。

5-sqlite3InitOne：
继续执行，很多语句之后，遇到调用 sqlite3_exec，查询系统表的内容。

6-sqlite3_exec:
zSql=SELECT name, rootpage, sql FROM 'main'.sqlite_master

7-sqlite3Prepare:
zSql=SELECT name, rootpage, sql FROM 'main'.sqlite_master
正常返回。

6-sqlite3_exec:
从 sqlite3Prepare 返回后，在 sqlite3_step 中经过几层（略）调用到 sqlite3VdbeExec。

7-sqlite3VdbeExec:
p->zSql=""
p->nOp=14
p->aOp[]的内容为：
0　21　OP_Trace
1　96　OP_Goto
2　13　OP_OpenRead
3　120 OP_Rewind
4　3　OP_Column
5　3　OP_Column
6　3　OP_Column
7　90　OP_ResultRow
8　104 OP_Next
9　34　OP_Close
10　40　OP_Halt

```
11  101 OP_Transaction
12  97  OP_TableLock
13  96  OP_Goto
```
可见，这次要真正地执行查询功能。

6-sqlite3_exec:
注：仍然在 SELECT name, rootpage, sql FROM 'main'.sqlite_master 的 STEP 循环中。
从 sqlite3_step 返回后，rc==SQLITE_ROW，继续处理返回的记录。
在“if(xCallback(pArg, nCol, azVals, azCols)){”一句中调用 sqlite3InitCallback，然后，又调用 sqlite3_exec。
这里调用 sqlite3_exec 是要执行系统表中每个对象的创建语句，目的是为这些对象创建内存数据结构。

7-sqlite3_exec:
zSql=
```
create table d (
   id integer primary key,
   name text,
   loca text )
```

8-sqlite3Prepare:
zSql=
```
create table d (
   id integer primary key,
   name text,
   loca text )
```

7-sqlite3_exec:
从 sqlite3Prepare 返回后，在 sqlite3_step 中经过几层（略）调用 sqlite3VdbeExec。

8-sqlite3VdbeExec:
p->zSql=""
p->nOp=2

7-sqlite3_exec:
rc!=SQLITE_ROW，不回显，退出循环。返回。

6-sqlite3_exec:
注：仍然在 SELECT name, rootpage, sql FROM 'main'.sqlite_master 的 STEP 循环中。
循环处理下一条记录。
从 sqlite3Prepare 返回后，在 sqlite3_step 中经过几层（略）调用 sqlite3VdbeExec。

7-sqlite3VdbeExec:
p->zSql=""

p->nOp=14
p->aOp[]

6-sqlite3_exec:
继续，sqlite3_step 的返回值为 101
rc!=SQLITE_ROW，不回显，退出循环。返回。
注：此时 sqlite_master 表中只有一条记录，至此，已经处理完了。

5-sqlite3InitOne：
返回。

4-sqlite3Init：
退出第一个循环。
又调用 sqlite3InitOne。

5-sqlite3InitOne：
调用 sqlite3InitCallback，创建临时数据库的系统表。

6-sqlite3InitCallback：
调用 sqlite3_exec。

7-sqlite3_exec:
zSql=CREATE TEMP TABLE sqlite_temp_master ...

8-sqlite3Prepare:
zSql=CREATE TEMP TABLE sqlite_temp_master ...

7-sqlite3_exec:
从 sqlite3Prepare 返回后，在 sqlite3_step 中经过几层（略）调用 sqlite3VdbeExec。

8-sqlite3VdbeExec:
p->zSql=""
p->nOp=2

7-sqlite3_exec:
rc!=SQLITE_ROW，不回显，退出循环。返回。

6-sqlite3InitCallback：
返回。

5-sqlite3InitOne：
继续执行，调用 sqlite3_exec。

4-sqlite3Init：

从 sqlite3InitOne 返回。

3-sqlite3RunParser：
从 sqlite3Init 经多级返回。

2-sqlite3Prepare:
zSql="select * from d;"
从 sqlite3RunParser 返回。

1-sqlite3_exec:
zSql="select * from d;"
从 sqlite3Prepare 返回。至此，在第一次 sqlite3Prepare 调用中所做的一系列准备工作终于做完了，又回到了对"select * from d;"语句的处理过程中。
顺便说明一下，本次"select * from d;"语句的处理过程完成后，如果马上再次执行"select * from d;"语句，上述准备工作就不需要再做一次了。流程会简单很多。
从 sqlite3Prepare 返回后，在 sqlite3_step 中调用 sqlite3VdbeExec。

2-sqlite3VdbeExec:
p->zSql="select * from d"
p->nOp=15
p->aOp[]的内容为：

```
0   21  OP_Trace
1   96  OP_Goto
2   13  OP_OpenRead
3   120 OP_Rewind
4   3   OP_Column
5   3   OP_Column
6   3   OP_Column
7   90  OP_ResultRow
8   104 OP_Next
9   34  OP_Close
10  40  OP_Halt
11  101 OP_Transaction
12  99  OP_VerifyCookie
13  97  OP_TableLock
14  96  OP_Goto
```

可见，这次要真正地执行查询功能。

1-sqlite3_exec:
从 sqlite3VdbeExec 返回，rc==100==SQLITE_ROW。
在“if(xCallback(pArg, nCol, azVals, azCols)){”一句中显示一条记录。

2-sqlite3VdbeExec:
p->zSql="select * from d"

p->nOp=15

1-sqlite3_exec:
从 sqlite3VdbeExec 返回，rc==100==SQLITE_ROW。
显示第 2 条记录。

2-sqlite3VdbeExec:
p->zSql="select * from d"
p->nOp=15

1-sqlite3_exec:
从 sqlite3VdbeExec 返回，rc==100==SQLITE_ROW。
显示第 3 条记录。

2-sqlite3VdbeExec:
p->zSql="select * from d"
p->nOp=15

1-sqlite3_exec:
从 sqlite3VdbeExec 返回，rc==100==SQLITE_ROW。
显示第 4 条记录。

2-sqlite3VdbeExec:
p->zSql="select * from d"
p->nOp=15

1-sqlite3_exec:
从 sqlite3VdbeExec 返回，rc!=SQLITE_ROW。定案并退出循环。
处理过程结束，累死了。

2. SQLite3 的内存数据结构

2.1. SQLite 体系结构介绍

我一向认为，水平越高的程序员编的程序越简单，越容易看懂，就好比水平越高的领导越能将问题简单化，而不是先把单位搞得鸡犬不宁然后自己也累得吐血。（这篇东西写得这么晦涩，作者水平也就可见一斑了）
SQLite 的原作者们水平都很高，程序技巧性很强，但我看源程序时还是经常遇到困难，除了我真笨外，还有以下原因：
1-SQLite 是实用程序，不是教学程序。一般实用程序在其功能主线之外都会附加一些代码，以保证程序的健壮性或完成测试功能等，SQLite 在这方面尤其过分。
2-缺少 SQLite 实现的背景知识，缺少数据库实现的背景知识，总之，我没文化。

3-缺少对 SQLite 体系结构的深入了解。有这方面的介绍，都很笼统，本章就是对这类介绍的一个补充。
其实，我费了这么大劲读程序，也就是搞清了本章的内容，搞清本章的内容之后再读程序，就变得比较简单了。“数据结构”很重要。

先来看下面代码，相信兄弟们都看得懂。该代码例示了在 SQLite 上执行一个查询的一般过程。

```
#include<stdio.h>
#include<stdlib.h>
#include"sqlite3.h"
#include<string.h>
#pragma comment(lib, "sqlite3.lib")

int main(int argc,char **argv)
{
    int rc,i,ncols;
    sqlite3 *db;
    sqlite3_stmt *stmt;
    char *sql;
    const char*tail;
    //打开数据
    rc=sqlite3_open("foods.db",&db);
    if(rc){
        fprintf(stderr,"Can'topendatabase:%sn",sqlite3_errmsg(db));
        sqlite3_close(db);
        exit(1);
    }

    sql="select * from episodes";
    //预处理
    rc=sqlite3_prepare(db,sql,(int)strlen(sql),&stmt,&tail);
    if(rc!=SQLITE_OK){
        fprintf(stderr,"SQLerror:%sn",sqlite3_errmsg(db));
    }

    rc=sqlite3_step(stmt);
    ncols=sqlite3_column_count(stmt);
    while(rc==SQLITE_ROW){
        for(i=0;i<ncols;i++){
            fprintf(stderr,"'%s'",sqlite3_column_text(stmt,i));
        }
        fprintf(stderr,"\n");
        rc=sqlite3_step(stmt);
    }
```

```
    //释放 statement
    sqlite3_finalize(stmt);
    //关闭数据库
    sqlite3_close(db);

    printf("\n");
    return(0);
}
```

程序就不解释了。在上述程序中，使用了两个结构的指针，分别是 sqlite3 和 sqlite3_stmt。SQLite 中有数不清的结构，而这两个结构正是 SQLite 数据处理的两大主线。学《数据结构》时讲究先研究存储结构再研究算法，对 SQLite 来说，搞清这两个结构之后，再读程序就会变得很容易了。

下图是 SQLite 官方网站对 SQLite 体系结构的描述，详细说明见参考文献 1。

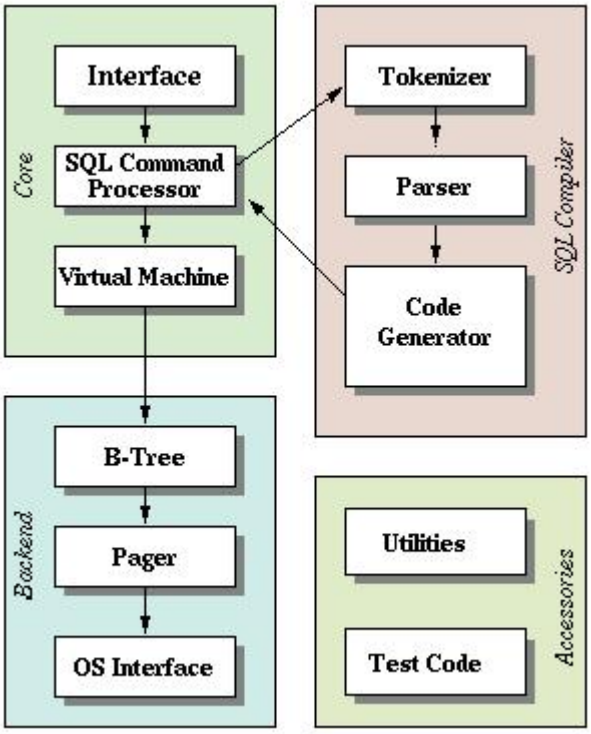

图 2-1 SQLite 体系结构

在 SQLite 的实现中，无论是代码的设计还是数据结构的设计都是按上述体系结构完成的，层次非常清晰。SQLite 中有数不清的结构，我们只按照两大主线介绍其中的不到 20 种，如下图所示。图中的箭头代表包含关系，可以看出，这些结构也是按照上述体系结构组织的。

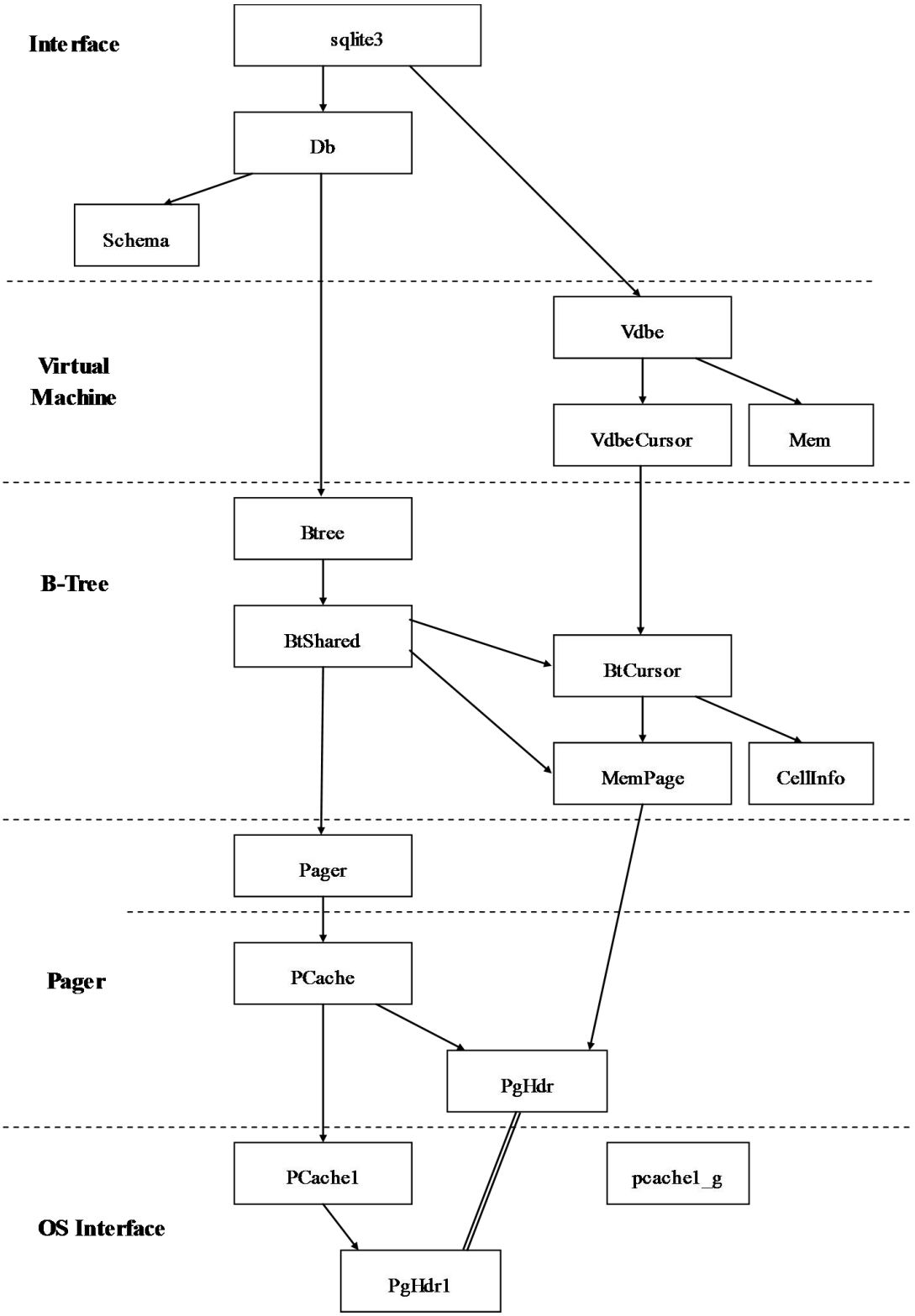

图 2-2 SQLite 的内存数据结构

关于“结构”与“对象”：

C 中只有结构，没有对象，但现在谁又能在编程时完全不受面向对象概念的影响呢？SQLite 的好多结构中都有函数指针，注册后我看跟对象的成员函数差不多吧。本文中有时把结构变量称为“结构的一个实例”或“对象”，不影响理解。

2.2. sqlite3 主线

2.2.1. sqlite3 结构

该结构有 70 多个域，我们只关心下面几个域。

```
struct sqlite3 {
  int nDb;              /* 当前后台（数据库）数量，初始后为 2，一个 main，一个临时 */
  Db *aDb;              /* 所有的后台，初始后为 db->aDb[0]和 db->aDb[1] */
  …
  struct Vdbe *pVdbe;               /* 活动的虚拟机列表 */
  int activeVdbeCnt;                /* 活动的虚拟机数量 */
  int writeVdbeCnt;                 /* 活动的正在写数据库的虚拟机数量 */
  …
};
```

在本主线中只讨论前两个域，后面三个域将在 sqlite3_stmt 主线一节讨论。

每个 sqlite3 结构在应用程序中代表一个“数据库连接”。每个数据库连接中可以连接多个数据库（文件），初始时有两个。编号为 0 的为主数据库，名称为 main。编号为 1 的为临时数据库。临时数据库可能是内存数据库，也可能是临时文件，在本文 1.1.2 节有介绍。临时数据库在连接关闭时被删除。从编号 2 开始为附加的数据库。

在结构中将 aDb 定义为 Db 结构的指针，实际使用时是 Db 结构指针的数组，使用下标存取。nDb 为 aDb 中元素的数量。

再次说明：上述结构与源程序相比做了很大的简化，去掉的部分不是不重要的，而是“可以不解释”的。后面的结构也存在这样情况，就不再说明了。

2.2.2. Db 结构

```
/*
数据库文件
为每个打开的数据库文件建立一个此结构的实例。
在 sqlite.aDb[]数组中通常包含两个本结构的实例。
aDb[0]是主数据库文件。aDb[1]是存放临时表的数据库文件。
另外，还可能附加其他数据库。
*/
struct Db {
  char *zName;          /* 此数据库名：main or temp */
  Btree *pBt;           /* 此数据库文件的 B*Tree 结构 */
  Schema *pSchema;      /* Pointer to database schema (possibly shared) */
};
```

关于上述两个结构的命名，我想猜测一下：SQLite 以前肯定是不支持在一个连接中打开多个数据库文件的，因此上述两个结构原来肯定是一回事，所以直到现在编写应用程序时还是用：

```
sqlite3 *db;
```

按现在的定义，是否用下面语句更合适呢？

sqlite3 *conn;

历史问题，不深究了，反正也没多少人附加数据库文件吧。

Db 结构的域倒是很少，关键就是每个 Db 结构中包含一个 Btree 结构（结构指针，下面在不影响理解的情况下，就不再对结构和其指针加以区分了）的域 pBt。也就是说，应用程序为连接中的每个数据库文件建立一个 Btree 对象。

2.2.3. Schema 结构

```
/*
本结构的实例用于存放数据库 schema。
*/
struct Schema {
  int schema_cookie;   /* Schema 版本：每次 schema 改变时，此值+1。 */
  Hash tblHash;        /* 所有表名，按名称排序 */
  Hash idxHash;        /* 所有索引名，按名称排序 */
  Hash trigHash;       /* 所有触发器名，按名称排序 */
  Table *pSeqTab;      /* AUTOINCREMENT 所使用的 sqlite_sequence 表 */
  u8 file_format;      /* 文件格式版本 */
  u8 enc;              /* 此数据库使用的编码类型 */
  u16 flags;           /* 与此 schema 相关的标志 */
  int cache_size;      /* 缓冲区中的页数 */
};
```

应用程序还为连接中的每个数据库文件建立一个 Schema 对象。从上述注释中就可以看出该结构的作用，不再详细说明了。沿此结构研究下去又是一条主线，本文从略。

2.2.4. Btree 结构

在进一步说明之前请先注意以下事实：

- 对数据库文件的读写都是通过 Btree 来完成的。
- 可能同时有多个连接访问相同的数据库文件。
- 数据库并发控制的责任不可能交给 OS，也不可能交给更高的一级，只能在 Btree 一级完成。
- 因此，Btree 中必然有多个连接所共享的部分。

所以，SQLite 在设计数据结构时将 Btree 分为两个层次：

- Btree 结构，为每个连接所独有，也就是说一个数据库文件在每个连接中都有一个 Btree 结构的实例。
- BtShared 结构，为所有连接所共享，也就是说无论有多少个连接同时访问一个数据库，该数据库只有一个 BtShared 的实例。

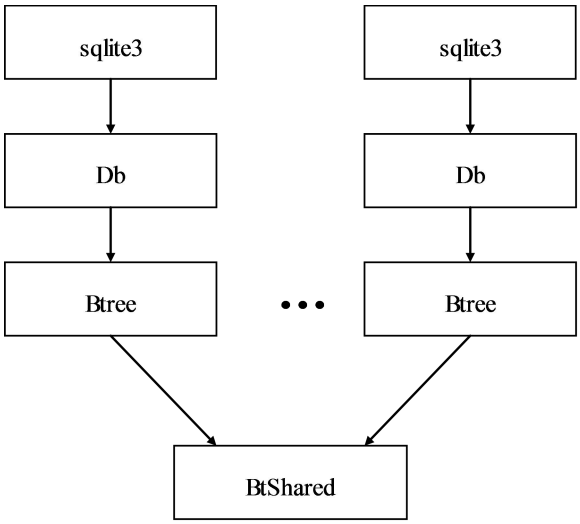

图 2-3 Btree 结构与 BtShared 结构的关系

```
/*
在数据库连接中，为每一个打开的数据库文件保持一个指向本对象实例的指针。
这个结构对数据库连接是透明的。数据库连接不能看到此结构的内部，只能通过指针来操作此结构。
有些数据库文件，其缓冲区可能被多个连接所共享。在这种情况下，每个连接都单独保持到此对象的指针。
但此对象的每个实例指向相同的 BtShared 对象。数据库缓冲区和 schema 都包含在 BtShared 对象中。
*/
struct Btree {
  sqlite3 *db;       /* 拥有此 btree 的数据库连接 */
  BtShared *pBt;     /* 此 btree 的可共享内容 */
  Btree *pNext;      /* List of other sharable Btrees from the same db */
  Btree *pPrev;      /* Back pointer of the same list */
};
```

关于 db 域：指向拥有此 btree 的数据库连接。这是一个由子孙结构指向祖先结构的指针。SQLite 的各类结构中有大量此类指针，可以实现子孙结构与祖先结构之间方便的相互引用。下面所介绍的结构中此类域就忽略不再介绍了。

2.2.5. BtShared 结构

```
/*
此对象的一个实例描述一个单独的数据库文件。
一个数据库文件同时可以被多个数据库连接使用。
当多个数据库连接共享相同的数据库文件时，每个连接有它自己的文件 Btree，这些 Btree 指向同一个本 BtShared 对象。
BtShared.nRef 是共享此数据库文件的连接的个数。
*/
struct BtShared {
  Pager *pPager;        /* 页缓冲区 */
```

```
  BtCursor *pCursor;    /* 包含当前打开的所有游标的链表 */
  MemPage *pPage1;      /* 数据库的 page 1 */
  u16 pageSize;         /* 每页的字节数 */
  u16 usableSize;       /* 每页可用的字节数。pageSize-每页尾部保留空间的大小，
                           在文件头偏移为 20 处设定。 */
  void *pSchema;        /* 指向由 sqlite3BtreeSchema()所申请空间的指针 */
  void (*xFreeSchema)(void*);  /* BtShared.pSchema 的析构函数 */
  int nRef;             /* 共享此数据库文件的连接的个数 */
};
```

BtShared 主要是管理页缓冲区，包括一个 Pager 结构和多个描述数据库文件全局参数的域。
关于 BtCursor 结构和 MemPage 结构的介绍见 2.4 节。

2.2.6. Pager 结构

```
/*
一个打开的页缓冲区是本结构的一个实例。
*/
struct Pager {
  sqlite3_file *fd;          /* 数据库文件描述符 */
  sqlite3_file *jfd;         /* 主日志文件描述符 */
  sqlite3_file *sjfd;        /* 子日志文件描述符 */
  char *zFilename;           /* 数据库文件名 */
  char *zJournal;            /* 日志文件名 */
  PCache *pPCache;           /* 页缓冲对象的指针。 */
};
```

包含相关文件的指针和一个页缓冲区。
关于 SQLite 所使用的各种临时文件的介绍见参考文献二。

2.2.7. PCache 结构

在 SQLite 权威文档（原文：http://www.sqlite.org/custombuild.html）中有如下描述：
绝大多数应用程序与使用默认编译选项编译的 SQLite 配合工作得很好。
尽管如此，还是有些特殊的应用程序可能想从 SQLite 中去掉一些内置的系统接口，代替以更适合应用程序的实现。SQLite 很容易在编译时重新配置，以适应特殊项目的需求。SQLite 在编译时可以：

- 用一个不同的实现来代替内置的互斥子系统。
- 在单线程应用程序中完全去掉所有的互斥机制。
- 重新配置内存分配子系统，使用一个非 malloc()的内存分配器。
- 重新调整内存分配子系统，使它根本就不再调用 malloc()。所有的内存请求都在 SQLite 启动时分配的固定大小的内存缓冲区中获得。
- 用另一个设计来替换文件系统接口。换句话说，用一组完全不同的系统调用来替换 SQLite 现有的操作磁盘的系统调用。
- 覆盖其它的操作系统接口。

一般来说，在 SQLite 中有 3 个子系统可以被修改或覆盖。互斥子系统用来在多线程间将对 SQLite 资源的存取序列化。内存分配子系统。虚拟文件系统(Virtual File System)子系统在 SQLite 和底层操作系统(特别是文件系统)间提供一个统一的(portable)接口。我们称这 3 个子系统为 SQLite 的接口子系统。

关于上面的描述我有两点说明：

- 咱不是“绝大多数开发者”，咱是要分析 SQLite 源程序，所以要了解 SQLite 的详细实现。
- 关于操作系统接口的说明在“打开数据库”一节有涉及，互斥子系统本文不涉及，本节涉及内存分配子系统的内容。

SQLite 将内存分配子系统实现为两层：

- PCache 结构为内存分配子系统的接口结构，与具体的实现无关。
- PCache1 为内存分配子系统的默认实现。

```
/*
** 一个完整的页缓冲区是本结构的一个实例。
*/
struct PCache {
  PgHdr *pDirty, *pDirtyTail;         /* 按 LRU 次序排列的脏页链表 */
  PgHdr *pSynced;                     /* 脏页链表中最近同步过的页 */
  int nRef;                           /* 引用的页数 */
  int nMax;                           /* 配置的缓冲区大小 */
  int szPage;                         /* 此缓冲区中每页的大小 */
  int szExtra;                        /* 每页扩展空间的大小 */
  sqlite3_pcache *pCache;             /* Pluggable cache module */
  PgHdr *pPage1;                      /* 指向 page 1 */
};
```

SQLite 允许加载用户自定义的缓冲区模块，默认的缓冲区模块是 pcache1.c。
在 pcache1.c 中，sqlite3_pcache *被映射为指向 PCache1 结构的指针。

2.2.8. PgHdr 结构

```
/*
缓冲区中的每个页由此结构的一个实例来控制。
此结构在 pcache.c 中定义。
*/
struct PgHdr {
  void *pData;                   /* 此页的内容 */
  void *pExtra;                  /* 扩充内容 */
  Pgno pgno;                     /* 本页的页号 */
  /**********************************************************************
  ** 上面的域是公共的。下面的域是 pcache.c 私有的，其它模块不能存取。
  */
  i16 nRef;                      /* 此页的用户数 */
```

```
  PCache *pCache;                 /* 拥有此页的缓冲区 */

  PgHdr *pDirtyNext;              /* 脏页链表中的 Next 指针 */
  PgHdr *pDirtyPrev;              /* 脏页链表中的 Previous 指针 */
};
```

2.2.9. PCache1 结构

```
/* 指向此结构的指针作为不透明的 sqlite3_pcache*句柄返回。
*/
struct PCache1 {
  /* 缓冲区配置参数。页大小(szPage)和可净化标志(bPurgeable)在本结构创建时设置。
  nMax 有可能在任何时候被 pcache1CacheSize()方法的调用所修改。
  */
  int szPage;                         /* 分配页的大小（字节数） */
  int bPurgeable;                     /* 如果缓冲区可净化，为 True */
  unsigned int nMin;                  /* Minimum number of pages reserved */
  unsigned int nMax;                  /*配置的缓冲区大小*/

  /* 所有页的 Hash 表。
  */
  unsigned int nRecyclable;           /* LRU 链表中总的页数 */
  unsigned int nPage;                 /* HASH 表 apHash 中总的页数 */
  unsigned int nHash;                 /* apHash[]的槽位数 */
  PgHdr1 **apHash;                    /* Hash 表，用于按照键值快速查找 */

  unsigned int iMaxKey;               /* 从上一次 xTruncate()之后的最大键值 */
};
```

本结构代表一个缓冲区，这个缓冲区中有多个页，这些页存储在 Hash 表 apHash 中。从定义来看，apHash 是 PgHdr1 结构指针的指针，实际编程时，apHash 是 PgHdr1 结构的指针数组。可见，apHash 是一个采用链接法解决地址冲突的 Hash 表。apHash 有 nHash 个槽位，每个槽位上是一个链表。

Hash 函数很简单：PgHdr1->iKey%nHash。

注：关于 SQLite 内存分配系统的设计可参考 SQLite 的官方文档（原文：http://www.sqlite.org/malloc.html）。本文的附录一中有我翻译的该文档的部分内容。

2.2.10. PgHdr1 结构

```
/*
每个缓冲区入口由此结构的一个实例来表示。
一个 PgHdr1.pCache->szPage 字节大小的缓冲区在内存中创建此结构之前被分配。(参后面的
PGHDR1_TO_PAGE()宏）
```

```
*/
struct PgHdr1 {
  unsigned int iKey;              /* Key 值(页号) */
  PgHdr1 *pNext;                  /* hash 表链中的下一个元素 */
  PgHdr1 *pLruNext;               /* 如果是未钉住的页，指向 LRU 链表中的后一个 */
  PgHdr1 *pLruPrev;               /* 如果是未钉住的页，指向 LRU 链表中的前一个 */
};
```

我理解所谓“缓冲区入口”就是内存中对一个数据库页的存取入口。
上面所列出的几个域中，iKey 的含义是好理解的，其它几个域和注释中所提到宏的含义将在 2.3.4 节中结合其它内容一起介绍。

2.3. 内在分配子系统的实现方法

无法全面分析，只介绍我注意到的几个问题吧。
上节仅按照结构包含的主线介绍了多个结构的定义，虽然对每个结构都做了相当的简化，但要一下子看明白还是很困难的。看了下面这几个小节之后，有些问题可能就明白了。

2.3.1. 数据库页的原始数据到底在哪儿

SQLite 数据库文件由固定大小的“页(page)”组成。页的默认大小为 1024 个字节(1KB)。页是数据库读写和在内存中进行管理的基本单位。(有关 SQLite 数据库文件格式的内容见参考文献二)
数据从文件读到内存以后，总得有个地方存吧，但无论从 PgHdr1 结构还是从 PgHdr 结构中，都找不到这样的指向页缓存的指针。
实际的实现是这样的：当为 PgHdr1 结构变量分配内存时，不是按照 PgHdr1 结构的大小来分配，而是分配一块更大的内存，包括 4 个部分，如下图：

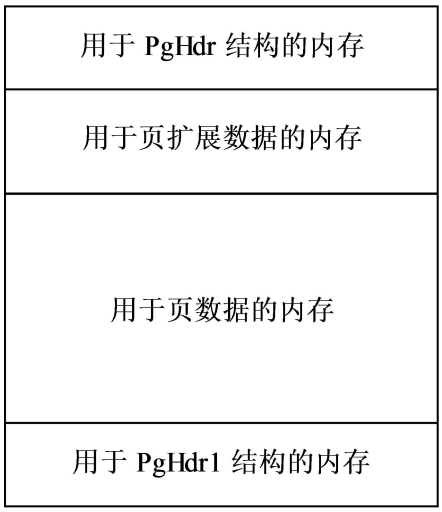

图 2-4 为 PgHdr1 结构分配的内存
在 PCache1 中，PCache1.szPage 指的是前 3 部分空间的大小之和。
从源程序中对下面两个宏的定义及注释中可以看出页内存分配与使用的方法。

```
/*
当 PgHdr1 结构被分配时，PCache1.szPage 字节大小的数据空间也会同时分配，即分配的总空间为 sizeof(PgHdr1)+PCache1.szPage 个字节。
（空注：从下面宏定义可以看出，在总空间中，页数据在前，PgHdr1 结构在后。）
宏 PGHDR1_TO_PAGE()将指向 PgHdr1 结构的指针转换为指向相关的页数据。
宏 PAGE_TO_PGHDR1()的作用正好相反。
*/
#define PGHDR1_TO_PAGE(p)      (void*)(((char*)p) - p->pCache->szPage)
#define PAGE_TO_PGHDR1(c, p)   (PgHdr1*)(((char*)p) + c->szPage)
```

2.3.2. PgHdr1 结构与 PgHdr 结构的关系

这两个结构其实是一回事，都代表一个数据库页，都能操作数据库中一页的数据。但它们在 SQLite 的不同层次上实现，互相之间是不透明的。它们在内存中的状态见图 2-4，它们在 SQLite 中的不同层次见图 2-2。
从下面 3 个函数（的部分程序中）可以看出这两个结构之间的关系。

```
/*
函数 1：
尝试从缓冲区中获得一个页。
本函数在 pcache.c 中。
*/
int sqlite3PcacheFetch(
  PCache *pCache,       /* Obtain the page from this cache */
  Pgno pgno,            /* Page number to obtain */
  int createFlag,       /* 如果为 true，当一个页不存在时则创建它 */
  PgHdr **ppPage        /* Write the page here */
){
  PgHdr *pPage = 0;
  int eCreate;

  /* 如果可填充缓冲区(sqlite3_pcache*)还没有分配，则现在分配。 */
  if( !pCache->pCache && createFlag ){
    sqlite3_pcache *p;
    int nByte;
    /* 经过映射，xCreate 方法调用的是 pcache1.c 中的 pcache1Create()函数。
    分析下面两句，结合被调函数，可以看出：
    struct PCache 和 struct PCache1 中的 szPage 值是不同的，这一点需要注意。
    */
    nByte = pCache->szPage + pCache->szExtra + sizeof(PgHdr);
    p = sqlite3GlobalConfig.pcache.xCreate(nByte, pCache->bPurgeable);
    if( !p ){
      return SQLITE_NOMEM;
    }
```

```
    sqlite3GlobalConfig.pcache.xCachesize(p, pCache->nMax);
    pCache->pCache = p;
  }

  eCreate = createFlag * (1 + (!pCache->bPurgeable || !pCache->pDirty));
  if( pCache->pCache ){
    /* 从缓冲区取第 pgno 页 */
    经过映射，xFetch 方法调用的是 pcache1.c 中的 pcache1Fetch()函数。
    pPage  =  sqlite3GlobalConfig.pcache.xFetch(pCache->pCache,  pgno,  eCreate);

  }

  if( pPage ){
    if( !pPage->pData ){
      memset(pPage, 0, sizeof(PgHdr) + pCache->szExtra);
      pPage->pExtra = (void*)&pPage[1];
      /* 注意此处要求返回的是一个 PgHdr 结构指针 */
      pPage->pData = (void *)&((char *)pPage)[sizeof(PgHdr) + pCache->szExtra];
      pPage->pCache = pCache;
      pPage->pgno = pgno;
    }
  }

  *ppPage = pPage;
  return (pPage==0 && eCreate) ? SQLITE_NOMEM : SQLITE_OK;
}

/*
函数 2：
实现 sqlite3_pcache.xCreate 方法。
分配一个 cache。
本函数在 pcache1.c 中。
*/
static sqlite3_pcache *pcache1Create(int szPage, int bPurgeable){
  PCache1 *pCache;

  pCache = (PCache1 *)sqlite3_malloc(sizeof(PCache1));
  if( pCache ){
    memset(pCache, 0, sizeof(PCache1));
    /*
    可以看出，struct PCache 和 struct PCache1 中的 szPage 值是不同的。
    */
    pCache->szPage = szPage;
    pCache->bPurgeable = (bPurgeable ? 1 : 0);
```

```
    if( bPurgeable ){
      pCache->nMin = 10;
      pcache1EnterMutex();
      pcache1.nMinPage += pCache->nMin;
      pcache1LeaveMutex();
    }
  }
  return (sqlite3_pcache *)pCache;
}

/*
函数 3：
实现 sqlite3_pcache.xFetch 方法。
按键值取一个页。
本函数在 pcache1.c 中。
*/
static void *pcache1Fetch(sqlite3_pcache *p, unsigned int iKey, int createFlag){
  PCache1 *pCache = (PCache1 *)p;
  PgHdr1 *pPage = 0;
  /* 注意此处返回的是一个 PgHdr1 结构指针，但经过了换算。 */
  return (pPage ? PGHDR1_TO_PAGE(pPage) : 0);
}
```

上述 3 个函数分别处于两个不同的层次，其中函数 1 处于 Pager 层，函数 2、3 处于 OS Interface 层。

2.3.3. sqlite3GlobalConfig 全局变量

在函数 1 中使用了一个全局变量 sqlite3GlobalConfig。在 sqliteInt.h 中有如下宏定义：

```
#define sqlite3GlobalConfig sqlite3Config
```

在 global.c 中为该全局变量赋初值，可见该变量是一个结构变量：

```
/*
** 这个单元包含了 SQLite 的全局配置信息。
*/
SQLITE_WSD struct Sqlite3Config sqlite3Config = {
    SQLITE_DEFAULT_MEMSTATUS,   /* bMemstat */
    0,                          /* bCoreMutex */
    …
};
```

结构 struct Sqlite3Config 的定义在 sqliteInt.h 中，有好多个域，此处我们只关心一个：

```
/*
保存 SQLite 的全局配置信息。
也包含一些状态信息。
```

```
*/
struct Sqlite3Config {
  …
  sqlite3_pcache_methods pcache;    /* 底层页缓冲接口 */
  …
};
```

用户可在该域中注册自定义的内存分配函数，默认的注册程序在 pcache1.c 中，如下：

```
/*
本函数在初始化期间(sqlite3_initialize())调用，注册默认的页缓冲模块。
*/
void sqlite3PCacheSetDefault(void){
  static sqlite3_pcache_methods defaultMethods = {
    0,                        /* pArg */
    …
    pcache1Create,            /* xCreate */
    pcache1Fetch,             /* xFetch */
  };
  sqlite3_config(SQLITE_CONFIG_PCACHE, &defaultMethods);
}
```

2.3.4. pcache1_g 全局结构变量

在函数 2、3 中使用了全局变量 pcache1，该变量在 OS_Interface 层起作用。该变量的定义如下：

```
/*
本内存分配子系统所使用的全局数据。
*/
static SQLITE_WSD struct PCacheGlobal {
  sqlite3_mutex *mutex;               /* static mutex MUTEX_STATIC_LRU */

  int nMaxPage;                       /* Sum of nMaxPage for purgeable caches */
  int nMinPage;                       /* Sum of nMinPage for purgeable caches */
  int nCurrentPage;                   /* Number of purgeable pages allocated */
  PgHdr1 *pLruHead, *pLruTail;        /* LRU 双向链表的表头和表尾指针 */

  /* 与 SQLITE_CONFIG_PAGECACHE 选项有关的设置。 */
  int szSlot;                         /* 空闲单元的大小 */
  void *pStart, *pEnd;                /* 整个页缓冲区的边界 */
  PgFreeslot *pFree;                  /* 空闲单元链表 */
  int isInit;                         /* 如果已经初始化则为 True */
} pcache1_g;

/*
本文件中存取上述全局结构变量时总是通过别名"pcache1"来完成。
```

*/
#define pcache1 (GLOBAL(struct PCacheGlobal, pcache1_g))

该全局变量管理一个全局的、很大的静态内存缓冲区。应用程序在 SQLite 启动时为它提供几个大的缓冲区，这样，在 SQLite 的运行过程中，所有的内存申请都尽量使用这些缓冲区，而不需要调用 malloc()和 free()（参本文附录一）。该全局变量在下面的函数中初始化：

```
/*
此函数用于初始化静态缓冲区。
如果页缓冲区是通过将 SQLITE_CONFIG_PAGECACHE 传递给 sqlite3_config()来建立，会调用此函数。
参数 pBuf 指向一个已分配的足够大的内存空间，包含'n'个页缓冲，每个页缓冲为'sz'字节。
*/
void sqlite3PCacheBufferSetup(void *pBuf, int sz, int n){
  if( pcache1.isInit ){
    PgFreeslot *p;
    sz = ROUNDDOWN8(sz);
    pcache1.szSlot = sz;
    pcache1.pStart = pBuf;
    pcache1.pFree = 0;
    while( n-- ){
      p = (PgFreeslot*)pBuf;
      p->pNext = pcache1.pFree;
      pcache1.pFree = p;
      pBuf = (void*)&((char*)pBuf)[sz];
    }
    pcache1.pEnd = pBuf;
  }
}
```

可见，SQLite 对静态内存的管理用的就是数据结构中静态链表的方法。在初始化时将大缓冲区分成若干个大小为 pcache1.szSlot 字节的单元，并将所有这些空闲单元链接起来构成空闲单元链表（数据结构中也叫自由队列）。初始化的结果如下图：

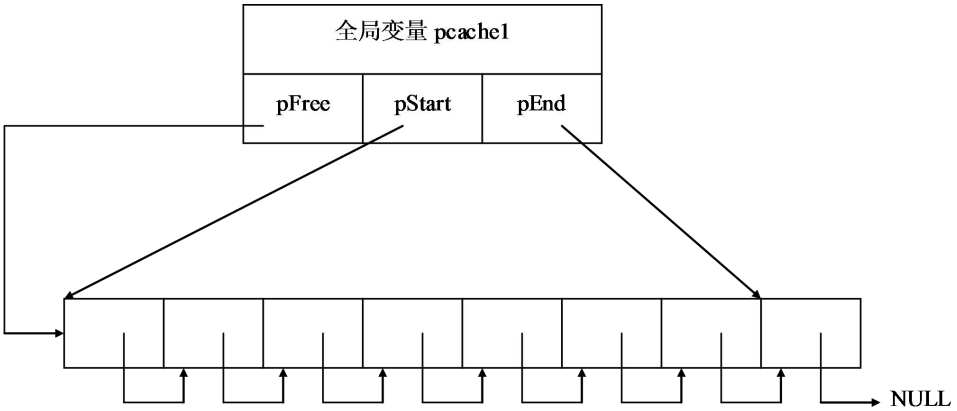

图 2-5 SQLite 的静态缓冲区和 pcache1 全局变量

其中，pcache1 的 pFree 为空闲单元链表首指针。另外两个域 pStart 和 pEnd 分别指向第一个

和最后一个单元，将来用于边界检查。
了解这些内容之后，空间的分配与回收就好理解了。

```
/*
在静态缓冲区中分配 nByte 大小的空间。
如果没有合适大小的空间或这样的空间已经用完，调用 sqlite3Malloc()。
*/
static void *pcache1Alloc(int nByte){
  void *p;
  if( nByte<=pcache1.szSlot && pcache1.pFree ){
    /* 从空闲单元链表中删除头结点 */
    p = (PgHdr1 *)pcache1.pFree;
    pcache1.pFree = pcache1.pFree->pNext;
  }else{
    /* 使用 sqlite3Malloc 分配一个新的缓冲区。 */
    p = sqlite3Malloc(nByte);
  }
  return p;
}

/*
释放一个由 pcache1Alloc()分配的缓冲空间。
*/
static void pcache1Free(void *p){
  if( p==0 ) return;
  if( p>=pcache1.pStart && p<pcache1.pEnd ){
    /* 在静态缓冲区范围内，加入到空闲单元链表的头部 */
    PgFreeslot *pSlot;
    pSlot = (PgFreeslot*)p;
    pSlot->pNext = pcache1.pFree;
    pcache1.pFree = pSlot;
  }else{
    /* 不在静态缓冲区范围内，释放空间 */
    sqlite3_free(p);
  }
}
```

现在，可以再来深入讨论 PCache1 结构的组织了（前面仅介绍了该结构的定义，见 2.2.9 节，还很不够）。
每个 SQLite 进程只有一个 pcache1 全局变量，也就是说只有一个静态缓冲区，但会为每个打开的数据库文件创建一个 PCache1 结构的页缓冲区，如下图：

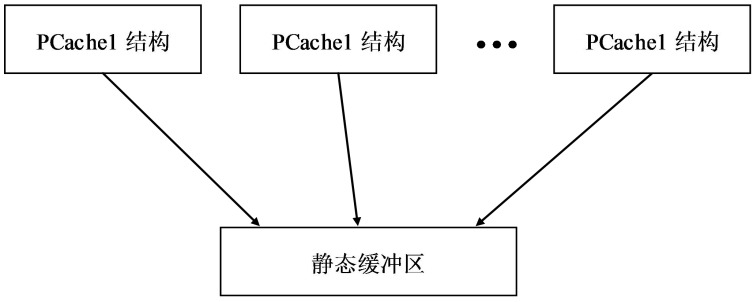

图 2-6 页缓冲区和静态缓冲区

当页缓冲区需要一个页空间时，就从静态缓冲区中申请；当页缓冲区中的页不再使用时，就释放回静态缓冲区。

下面是内存分配子系统中为页缓冲区分配一个页的程序：

```
/*
为缓冲区 pCache 分配一个新页。
*/
static PgHdr1 *pcache1AllocPage(PCache1 *pCache){
  /* 计算需要分配的字节数：PgHdr1 结构大小+页大小 */
  int nByte = sizeof(PgHdr1) + pCache->szPage;
  void *pPg = pcache1Alloc(nByte);
  PgHdr1 *p;
  if( pPg ){
    p = PAGE_TO_PGHDR1(pCache, pPg);
    if( pCache->bPurgeable ){
      pcache1.nCurrentPage++;
    }
  }else{
    p = 0;
  }
  return p;
}
```

每个页缓冲区中维护一个 Hash 表，见 PCache1 结构的下面 3 个域：

```
  unsigned int nPage;                   /* HASH 表 apHash 中总的页数 */
  unsigned int nHash;                   /* apHash[]的槽位数 */
  PgHdr1 **apHash;                      /* Hash 表，用于按照键值快速查找 */
```

页缓冲区中所有的页都存储在 Hash 表 apHash 中，有关该 Hash 表的定义与算法见 2.2.9 节。

下面是在静态缓冲区中申请一个新页的程序段：

```
static void *pcache1Fetch(sqlite3_pcache *p, unsigned int iKey, int createFlag){
  PCache1 *pCache = (PCache1 *)p;
  PgHdr1 *pPage = 0;

  /* 第 5 步 */
  if( !pPage ){
    /* 申请新页 */
```

```
    pPage = pcache1AllocPage(pCache);
  }

  if( pPage ){
    /*将新页插入到页缓冲区 Hash 表相应槽链的头部 */
    unsigned int h = iKey % pCache->nHash;
    pCache->nPage++;
    pPage->iKey = iKey;
    pPage->pNext = pCache->apHash[h];
    pPage->pCache = pCache;
    pPage->pLruPrev = 0;
    pPage->pLruNext = 0;
    pCache->apHash[h] = pPage;
  }

  return (pPage ? PGHDR1_TO_PAGE(pPage) : 0);
}
```

现在就可以理解 PgHdr1 结构中 pNext 域的含义了，它是指向 Hash 表相应槽链中下一个页的指针。

上述程序段中红色的两句是什么意思呢？下面来解释。

SQLite 加载到页缓冲区中的页，有些随时可以释放，如仅为读数据库操作服务，读完了就可以释放了。有些不能释放，如已经加锁的页或已经被修改的页。对于不能释放的页，SQLite 会把它“钉住(pin)”。对于可以释放的页，SQLite 不到必要（这里“必要”包括多种情况，不详细讨论）时也不会释放，这样，下次如果再使用该页时就不需要再次从磁盘读取了。

如果静态缓冲区中已无空闲单元，当再要向静态缓冲区申请空间时，SQLite 会释放一些“可以释放”的页，以满足新的空间申请要求。SQLite 的这种空间使用机制是通过 LRU 链表来实现的。

在 pcache1 全局变量中维护着一个 LRU（最近最少使用算法）双向链表，SQLite 将所有未钉住的页都加入到此链表中，pLruHead 和 pLruTail 域分别指向该双向链表的表头和表尾。当需要释放一个页时从表尾删除，某个页使用了之后就会移到表头处，新解除“钉住”的页也加入到表头处，这样就实现了页缓冲区的 LRU 功能。

如果一个页是不钉住的（即该页在 LRU 链表中），则 PgHdr1 结构中的两个域 pLruNext 和 pLruPrev 分别指向 LRU 链表中的前一个页和后一个页。如果一个页是钉住的，这两个域的值为 0。

理解了这个机制，好多功能的实现就好理解了。比如，所谓“钉住”一个页，其实就是把它从 LRU 表中删除。所谓“不钉住”一个页，就是把它加到 LRU 表中去。刚分配的页都是钉住的，不加到 LRU 链表中（现在就可以理解上面程序段中红色两句的含义了）。

将一个页钉住的程序如下，典型的双向链表的删除算法：

```
/*
此函数将 pPage 页从全局 LRU 链表中移除。如果 pPage 不在 LRU 链表中，此函数什么也不做。
*/
static void pcache1PinPage(PgHdr1 *pPage){
```

```
  if( pPage && (pPage->pLruNext || pPage==pcache1.pLruTail) ){
    /* 从双向链表中删除 */
    if( pPage->pLruPrev ){
      pPage->pLruPrev->pLruNext = pPage->pLruNext;
    }
    if( pPage->pLruNext ){
      pPage->pLruNext->pLruPrev = pPage->pLruPrev;
    }
    /* 如果是头结点，改头指针 */
    if( pcache1.pLruHead==pPage ){
      pcache1.pLruHead = pPage->pLruNext;
    }
    /* 如果是尾结点，改尾指针 */
    if( pcache1.pLruTail==pPage ){
      pcache1.pLruTail = pPage->pLruPrev;
    }
    pPage->pLruNext = 0;
    pPage->pLruPrev = 0;
    pPage->pCache->nRecyclable--;   /* 修改可重用页数 */
  }
}
```

使一个页不被钉住的程序如下，比典型的双向链表的插入算法多一点儿操作：

```
/*
实现 sqlite3_pcache.xUnpin 方法。
使页不被钉住（使其可被回收）
*/
static void pcache1Unpin(sqlite3_pcache *p, void *pPg, int reuseUnlikely){
  PCache1 *pCache = (PCache1 *)p;
  PgHdr1 *pPage = PAGE_TO_PGHDR1(pCache, pPg);

  if( reuseUnlikely || pcache1.nCurrentPage>pcache1.nMaxPage ){
    /*如果该页看起来不会再被使用了，且 LRU 中页数已经太多，直接释放该页。*/
    pcache1RemoveFromHash(pPage);   /* 从该页所在的页缓冲区中删除 */
    pcache1FreePage(pPage); /* 释放该页 */
    /* 当然就不需要再向 LRU 链表中插入了 */
  }else{
    /* 将该页插入到 LRU 链表的头部。 */
    if( pcache1.pLruHead ){     /* LRU 链表不为空 */
      pcache1.pLruHead->pLruPrev = pPage;
      pPage->pLruNext = pcache1.pLruHead;
      pcache1.pLruHead = pPage;
    }else{  /* LRU 链表为空 */
      pcache1.pLruTail = pPage;
```

```
      pcache1.pLruHead = pPage;
    }
    pCache->nRecyclable++;
  }
}
```

写到这里还有两个问题我也没搞清楚：
一是“净化”机制没搞清楚；
二是我在跟踪 SQLite 运行时，静态缓冲区的大小总是为 0，而不是传说中的“默认为 2000 页”。
没搞清楚就算了，我不怕丢人。

2.4. sqlite3_stmt 主线

现在开始讨论 SQLite 内存数据结构的第 2 条主线。

2.4.1. Vdbe 结构

记得网上有人问过，在 SQLite 源程序中为什么找不到 sqlite3_stmt 结构的定义。在 SQLite 对外的接口中都是使用 sqlite3_stmt，而其实在 SQLite 的实现中，"sqlite3_stmt *"是一个指向 VDBE 结构的不透明的指针。

```
/*
虚拟机的一个实例。
此结构包含了虚拟机的全部状态。
*/
struct Vdbe {
  Vdbe *pPrev,*pNext;     /* 数据库连接中虚拟机链表的指针域 */
  int nOp;                /* 程序中的指令数 */
  int nOpAlloc;           /* 为 aOp[]分配的槽位数 */
  Op *aOp;                /* 存放虚拟机程序的空间 */
  Mem *aColName;          /* 返回的各字段的名称 */
  Mem *pResultSet;        /* 指向一个结果数组的指针 */
  u16 nResColumn;         /* 结果集中每行的字段数 */
  u16 nCursor;            /* apCsr[]中的元素数 */
  VdbeCursor **apCsr;     /* 此数组中的每个元素为一个打开的游标 */
  int pc;                 /* 程序计数器 */
};
```

每个数据库连接中可能有多个活动的虚拟机，这些虚拟机被组织成一个双向链表，pPrev 和 pNext 域分别指向链表中的前趋和后继。
在 sqlite3 结构中有一个域 pVdbe，为指向此双向链表的头指针。新创建的虚拟机插在该双向链表的头部，创建虚拟机的程序如下：

```
/*
```

创建一个新的虚拟机。
*/
```
Vdbe *sqlite3VdbeCreate(sqlite3 *db){
  Vdbe *p;
  p = sqlite3DbMallocZero(db, sizeof(Vdbe) );
  if( p==0 ) return 0;
  p->db = db;
  if( db->pVdbe ){
    db->pVdbe->pPrev = p;
  }
  p->pNext = db->pVdbe;
  p->pPrev = 0;
  db->pVdbe = p;
  return p;
}
```

2.4.2. Mem 结构

/*
Mem 结构
在 SQLite 内部，vdbe 用 Mem 结构来操作几乎所有的 SQL 值。
Mem 结构会给出同一个值的多种表示法(string、integer 等)。
一个值（也就是 Mem 结构）具有以下属性：
每个值具有一个弱类型。存储在 Mem 结构中的值的弱类型由 MemType(Mem*)返回。
类型可以是 SQLITE_NULL、SQLITE_INTEGER、SQLITE_REAL、SQLITE_TEXT 或 SQLITE_BLOB。
*/
```
struct Mem {
  union {
    i64 i;              /* 整数值 */
    int nZero;          /* 当 MEM_Zero 位设置时有效 */
    FuncDef *pDef;      /* Used only when flags==MEM_Agg */
    RowSet *pRowSet;    /* Used only when flags==MEM_RowSet */
    VdbeFrame *pFrame;  /* Used when flags==MEM_Frame */
  } u;
  double r;         /* 实数值 */
  sqlite3 *db;      /* 相关的数据库连接 */
  char *z;          /* 字符串或 BLOB 值 */
  int n;            /* 字符串值中的字节数，不包括'\0' */
  u16 flags;        /* 由 MEM_Null, MEM_Str, MEM_Dyn 等标志位组成，详见后面的注释 */
  u8  type;         /* SQLITE_NULL, SQLITE_TEXT, SQLITE_INTEGER 等值中的一个 */
  u8  enc;          /* 编码：SQLITE_UTF8, SQLITE_UTF16BE, SQLITE_UTF16LE */
};
```

2.4.3. VdbeCursor 结构

游标在虚拟机一层的表现形式。

```
/*
游标是数据库中特定的 BTree 指针。
游标可以用特定的键值在 BTree 中搜寻入口，或循环处理所有的入口。
你可以向 BTree 中插入新的入口或从游标当前指向的入口中取出键值或数据值。
虚拟机打开的每个游标为此结构的一个实例。
*/
struct VdbeCursor {
  BtCursor *pCursor;    /* 后台游标结构 */
  Btree *pBt;           /* Separate file holding temporary table */
  int nField;           /* 字段数量 */
  /* 游标当前指向的数据记录的信息。
  仅在 cacheStatus 与 Vdbe.cacheCtr 匹配时有效。
  */
  u32 cacheStatus;      /* 如果此值与 Vdbe.cacheCtr 匹配，说明缓冲区可用 */
  int payloadSize;      /* Payload 大小，以字节为单位。 */
  u32 *aType;           /* 各字段的数据类型值 */
  u32 *aOffset;         /* 各字段数据的偏移量 */
  u8 *aRow;             /* 当前行的数据（如果它们都在同一个页中） */
};
```

注：各域的含义解释起来挺麻烦的，就不解释了。看看参考文献二，就全明白了。下面几个结构也存在此情况，不再说明。

2.4.4. BtCursor 记录

```
/*
游标是指向一个特定入口的指针，这个入口在一个数据库文件的特定 b-tree 中。
入口由 MemPage 和 MemPage.aCell[]的下标确定。
一个数据库文件可被多个数据库连接共享，但游标不能被共享。
*/
struct BtCursor {
  Pgno pgnoRoot;            /* 此 Btree 的根页页号 */
  CellInfo info;            /* 当前指向的单元（cell）的解析结果 */
  void *pKey;               /* 游标最后已知的位置的键值 */
  i64 nKey;                 /* pKey 的大小或最后的整数键值 */
  i16 iPage;                            /* 当前页在 apPage 中的索引 */
  MemPage *apPage[BTCURSOR_MAX_DEPTH];  /* 从根页到本页的所有页 */
  u16 aiIdx[BTCURSOR_MAX_DEPTH];        /* apPage[i]中的当前索引。空注：单元
                                        指针数组中的当前下标。 */
};
```

2.4.5. CellInfo 结构

```
/*
此结构的实例用来保存单元头信息。
parseCellPtr()负责根据从原始磁盘页中取得的信息填写此结构。
*/
struct CellInfo {
  u8 *pCell;     /* 指向单元内容的指针 */
  i64 nKey;      /* 关键字的字节数。如果 intkey 标志被设置，此域即为关键字本身。*/
  u32 nData;     /* 数据的字节数 */
  u16 nHeader;   /* 记录头的字节数 */
  u16 iOverflow; /* 溢出页链表中第 1 个溢出页的页号。如果没有溢出页，无此域。 */
  u16 nSize;     /* 单元数据的大小（不包括溢出页上的内容）*/
};
```

2.4.6. MemPage 结构

```
/*
** 每当文件的一个页加载到内存，下面结构的一个实例也会增加并被初始化为 0。
** 这个结构存储有页的信息，这些信息从原始的文件页中解码得到的。
*/
struct MemPage {
  u8 intKey;           /* 如果 intkey 标志设置则为 True */
  u8 leaf;             /* 如果是叶子，则为 True */
  u8 hasData;          /* 如果有数据则为 True */
  u8 hdrOffset;        /* 对 page 1 为 100，对其它页为 0 */
  u8 childPtrSize;     /* 如果是叶子则为 0，如果不是叶子则为 4 */
  u16 cellOffset;      /* 单元指针数组的偏移量，aData 中第 1 个单元的指针 */
  u16 nFree;           /* 可使用空间的总和（字节数） */
  u16 nCell;           /* 本页的单元数, local and ovfl */
  u8 *aData;           /* 指向页数据的磁盘映像 */
  DbPage *pDbPage;     /* Pager 的页句柄 */
  Pgno pgno;           /* 本页的页号 */
};
```

页数据在不同的层次上有不同的表现形式。前面已经介绍过了：页数据在 OS Interface 层的表现形式是 PgHdr1 结构，在 Pager 层的表现形式是 PgHdr 结构。在 B-Tree 层的表现形式就是 MemPage 结构。

PgHdr 结构在 Pager 层还有一个别名，定义如下：

```
/*
** 页句柄类型
*/
typedef struct PgHdr DbPage;
```

通过下面程序可以了解 MemPage 结构与 PgHdr 结构的关系：

```
/*
将从 pager 中获得的 DbPage 转化为 btree 中使用的 MemPage。
*/
static MemPage *btreePageFromDbPage(DbPage *pDbPage, Pgno pgno, BtShared *pBt){
  MemPage *pPage = (MemPage*)sqlite3PagerGetExtra(pDbPage);
  pPage->aData = sqlite3PagerGetData(pDbPage);
  pPage->pDbPage = pDbPage;
  pPage->pBt = pBt;
  pPage->pgno = pgno;
  pPage->hdrOffset = pPage->pgno==1 ? 100 : 0;
  return pPage;
}
```

3. SQLite 的锁机制与实现

要想全面分析 SQLite 源程序还是挺难的。人家算上注释才 10 万行代码，我要是写一个 20 万行的“分析”也没劲，讨论几个小的专题吧。

先讨论一下 SQLite 的锁机制吧。网上讨论 SQLite 锁的文章不少，但一般都是讲原理，本章讲实现。

3.1. SQLite 的锁机制介绍

SQLite 采用粗放型的锁。当一个连接要写数据库，所有其他的连接被锁住，直到写连接结束了它的事务。SQLite 有一个加锁表，来帮助不同的写数据库者能够在最后一刻再加锁，以保证最大的并发性。

SQLite 使用锁逐步上升机制，为了写数据库，连接需要逐步地获得排它锁。SQLite 有 5 个不同的锁状态：未加锁(UNLOCKED)、共享(SHARED)、保留(RESERVED)、未决(PENDING)和排它(EXCLUSIVE)。每个数据库连接在同一时刻只能处于其中一个状态。每种状态(未加锁状态除外)都有一种锁与之对应。锁的状态以及状态的转换如下图所示：

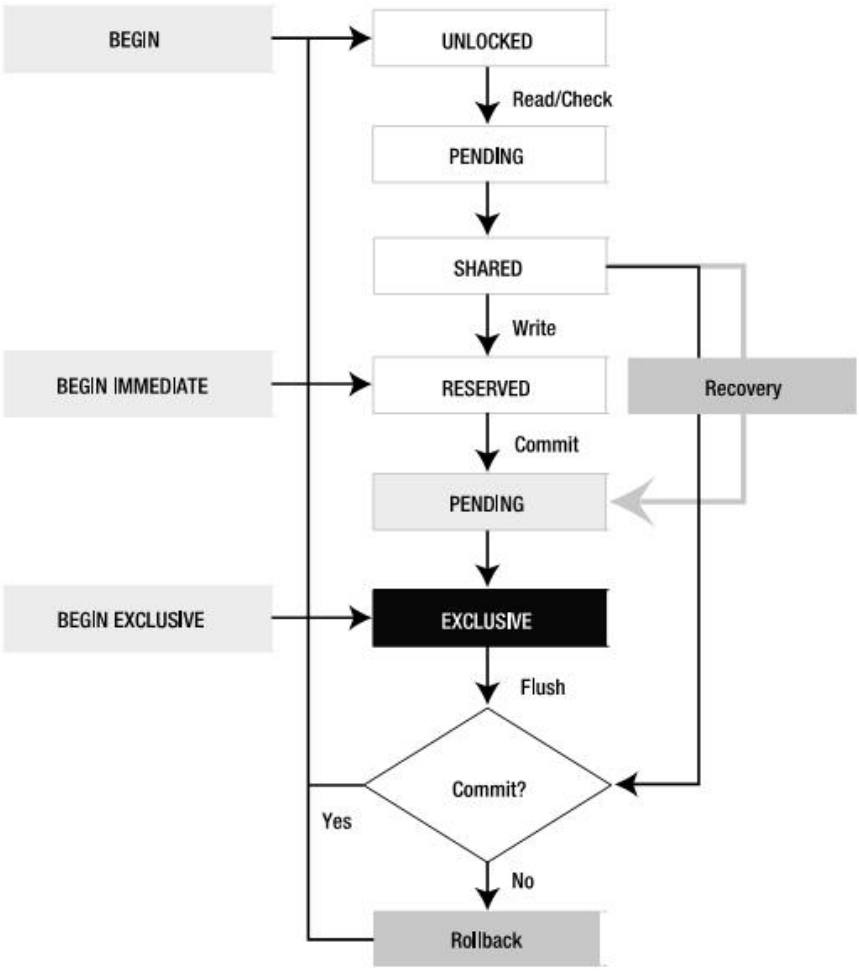

图 3-1 SQLite 锁的状态以及状态的转换

最初的状态是未加锁状态，在此状态下，连接还没有存取数据库。当连接到了一个数据库，甚至已经用 BEGIN 开始了一个事务时，连接都还处于未加锁状态。

未加锁状态的下一个状态是共享状态。为了能够从数据库中读(不写)数据，连接必须首先进入共享状态，也就是首先要获得一个共享锁。多个连接可以同时获得并保持共享锁，也就是说多个连接可以同时从同一个数据库中读数据。但即使仅有一个共享锁没有释放，也不允许任何连接写数据库。

如果一个连接想要写数据库，它必须首先获得一个保留锁。一个数据库上同时只能有一个保留锁。保留锁可以与共享锁共存，保留锁是写数据库的第 1 阶段。保留锁即不阻止其它拥有共享锁的连接继续读数据库，也不阻止其它连接获得新的共享锁。

一旦一个连接获得了保留锁，它就可以开始处理数据库修改操作了，尽管这些修改只能在缓冲区中进行，而不是实际地写到磁盘。对读出内容所做的修改保存在内存缓冲区中。

当连接想要提交修改(或事务)时，需要将保留锁提升为排它锁。为了得到排它锁，还必须首先将保留锁提升为未决锁。获得未决锁之后，其它连接就不能再获得新的共享锁了，但已经拥有共享锁的连接仍然可以继续正常读数据库。此时，拥有未决锁的连接等待其它拥有共享锁的连接完成工作并释放其共享锁。

一旦所有其它共享锁都被释放，拥有未决锁的连接就可以将其锁提升至排它锁，此时就可以自由地对数据库进行修改了。所有以前对缓冲区所做的修改都会被写到数据库文件。

3.2. SQLite 锁的实现

3.2.1. 操作系统基础

SQLite 锁的实现与具体的操作系统有关。SQLite 提供了 Windows、Unix 和 OS2 三种操作系统的实现，下面以 Windows 操作为例进行介绍。
Windows 允许对文件中的部分内容加共享锁或排它锁，相关的底层函数为：LockFile()、LockFileEx()和 UnlockFile()。这些函数的详细说明此处从略，可到 MSDN 中去查。前两个函数可对文件的指定区域加锁，需要特别指出的是，加锁的区域甚至可以是文件以外（超过文件尾的不实际存在的地方）。使用 LockFileEx()可以对文件加共享锁或排它锁。

3.2.2. 实现的说明

仅仅利用操作系统的机制显然是不够的，功能也不够，还降低系统的可移植性。SQLite 必须要有自己的机制来管理数据库锁。
SQLite 加锁的单位是数据库文件。

SQLite 在 os.h 中对锁类型有如下定义：

```
#define NO_LOCK          0
#define SHARED_LOCK      1
#define RESERVED_LOCK    2
#define PENDING_LOCK     3
#define EXCLUSIVE_LOCK   4
```

可见，随着锁级别的升高，其类型值也变大。

SQLite 在 os.h 中对加锁区域有如下定义：

```
#define PENDING_BYTE       sqlite3PendingByte
#define RESERVED_BYTE      (PENDING_BYTE+1)
#define SHARED_FIRST       (PENDING_BYTE+2)
#define SHARED_SIZE        510
```

SQLite 在 global.c 中定义了如下全局变量：

```
int sqlite3PendingByte = 0x40000000;
```

上面，为 4 种锁定义了 3 个区域，SQLite 通过对数据库文件的这 3 个区域加锁来实现其锁机制。
(1)PENDING 锁：由于同时只能有一个进程拥有 PENDING 锁，所以哪个进程能够在 PENDING_BYTE 区域上加排它锁(使用 LockFile()函数)，即获得了 PENDING 锁。
(2)RESERVED 锁：(与 PENDING 锁相同的机制)由于同时只能有一个进程拥有 RESERVED 锁，所以哪个进程能够在 RESERVED_BYTE 区域上加排它锁(使用 LockFile()函数)，即获得了 RESERVED 锁。
(3)SHARED 锁：由于可以同时有多个进程获得 SHARED 锁，因此，哪个进程能够在 SHARED_FIRST 区域加共享锁(使用 LockFileEx()函数)，即获得了 SHARED 锁。
(4)EXCLUSIVE 锁：虽然同时只能有一个进程拥有 EXCLUSIVE 锁，但由于在该进程拥有

EXCLUSIVE 锁时不允许其他进程拥有 SHARED 锁，因此 EXCLUSIVE 锁与 SHARED 锁使用相同的文件区域。哪个进程能够在 SHARED_FIRST 区域上加排它锁(使用 LockFile()函数，这样操作系统就保证了不会有其他 SHARED 锁)，即获得了 PENDING 锁。

为什么上述 3 个文件区域大小不一呢？PENDING_BYTE 和 RESERVED_BYTE 都只有 1 个字节，而 SHARED_FIRST 区域却有 SHARED_SIZE(510)个字节。我觉着都只要 1 个字节就可以了，但据说在 SQLite 的历史上，曾经用 SHARED_FIRST ~ SHARED_FIRST + SHARED_SIZE 范围内的每个字节表示不同的 SHARED 锁（最多 510 个），所以就这么传承下来了，这种说法我没有考证。

为什么将 PENDING_BYTE 定位在 0X4000 0000（1GB）？ Windows 文件中，被加锁的区域不允许任何进程写，包括第一个持有该锁的进程。因此，SQLite 数据库文件中上述三个区域所在的页就不能再存储数据了。前面介绍过了，Windows 文件中加锁的区域甚至可以是文件以外（超过文件尾的不实际存在的地方），利用这一特性，将上述三个区域定位在很“遥远”的 1GB 处，只要文件不超过 1GB，上述三个区域就实际上是不存在的，也就不会影响数据的存储了。

3.2.3. 实现的程序

实现的程序在 os_win.c 中：

```
/*
为文件加锁。
加锁的类型由locktype参数指定，为以下之一：
      (1) SHARED_LOCK
      (2) RESERVED_LOCK
      (3) PENDING_LOCK
      (4) EXCLUSIVE_LOCK
*/
static int winLock(sqlite3_file *id, int locktype){
  int rc = SQLITE_OK;    /* 返回码 */
  int res = 1;           /* Windows锁操作的返回值 */
  int newLocktype;       /* 在退出前将pFile->locktype设为此值 */
  int gotPendingLock = 0;/* 如果此次申请的是一个PENDING锁，此值为True */
  winFile *pFile = (winFile*)id;
  DWORD error = NO_ERROR;

  /* 当前的锁>=locktype,则返回
  */
  if( pFile->locktype>=locktype ){
    return SQLITE_OK;
  }

  /* 如果申请的是PENDING锁或SHARED锁，需要将PENDING_LOCK字节锁住。
  如果申请的是SHARED锁，锁PENDING_LOCK字节只是暂时的。
```

```
*/
newLocktype = pFile->locktype;

/*区分两种情况: (1)如果当前文件处于无锁状态，获取读锁，
这是读事务和写事务在最初阶段都要经历的阶段。
(2)如果当前处于RESERVED_LOCK，且请求的锁为EXCLUSIVE_LOCK(写事务)，
则获取PENDING锁。
//////////////////(1)/////////////////
*/
if(   (pFile->locktype==NO_LOCK)
   || (   (locktype==EXCLUSIVE_LOCK)
       && (pFile->locktype==RESERVED_LOCK))
){
  int cnt = 3;
  /* 获取pending锁 */
  while( cnt-->0 && (res = LockFile(pFile->h, PENDING_BYTE, 0, 1, 0))==0 ){
    /* 如果获取失败，尝试3次。
    */
    Sleep(1);
  }
  /* 设置gotPendingLock为1，后面的程序根据此值可能会释放PENDING锁 */
  gotPendingLock = res;
  if( !res ){
    error = GetLastError();
  }
}

/*获取shared锁
此时,事务应该持有PENDING锁。PENDING锁作为事务从UNLOCKED到
SHARED_LOCKED的一个过渡，实际上此时锁处于两个状态:PENDING和SHARED，
直到后面释放PENDING锁后，才真正处于SHARED状态。
////////////////(2)///////////////////////////////////
*/
if( locktype==SHARED_LOCK && res ){
  res = getReadLock(pFile);
  if( res ){
    newLocktype = SHARED_LOCK;
  }else{
    error = GetLastError();
  }
}

/*获取RESERVED锁
此时事务应持有SHARED锁，变化过程为SHARED->RESERVED。
```

RESERVED锁的作用就是为了提高系统的并发性能。
//////////////////////(3)////////////////////////////////
*/
```
if( locktype==RESERVED_LOCK && res ){
  res = LockFile(pFile->h, RESERVED_BYTE, 0, 1, 0);
  if( res ){
    newLocktype = RESERVED_LOCK;
  }else{
    error = GetLastError();
  }
}
```

/*获取PENDING锁
此时事务持有RESERVED_LOCK锁，且事务申请EXCLUSIVE锁。
变化过程为:RESERVED->PENDING。
PENDING状态唯一的作用就是防止写饿死。
读事务不会执行此段代码，但写事务会执行该代码。
执行该代码后gotPendingLock设为0，后面就不会释放PENDING锁了。
//////////////////////////////(4)////////////////////////////////
*/
```
if( locktype==EXCLUSIVE_LOCK && res ){
  //这里没有实际的加锁操作（因为PENDING锁前面已经加过了），
  //只是把锁的状态改为PENDING状态
  newLocktype = PENDING_LOCK;
  //设置了gotPendingLock,后面就不会释放PENDING锁了,
  //相当于加了PENDING锁,实际上是在开始处加的PENDING锁
  gotPendingLock = 0;
}
```

/*获取EXCLUSIVE锁
当一个事务执行该代码时，它应该满足以下条件：
(1)锁的状态为PENDING; (2)是一个写事务。
变化过程:PENDING->EXCLUSIVE
//////////////////////(5)//
*/
```
if( locktype==EXCLUSIVE_LOCK && res ){
  res = unlockReadLock(pFile);
  res = LockFile(pFile->h, SHARED_FIRST, 0, SHARED_SIZE, 0);
  if( res ){
    newLocktype = EXCLUSIVE_LOCK;
  }else{
    error = GetLastError();
    getReadLock(pFile);
  }
```

```
  }

  /* 如果申请的是SHARED锁，此时需要释放在第1步中获得的PENDING锁。
  锁的变化为:PENDING->SHARED
  ////////////////////////(6)///////////////////////////////////
  */
  if( gotPendingLock && locktype==SHARED_LOCK ){
    UnlockFile(pFile->h, PENDING_BYTE, 0, 1, 0);
  }

  /* 改变文件的锁状态，返回适当的结果码 */
  if( res ){
    rc = SQLITE_OK;
  }else{
    pFile->lastErrno = error;
    rc = SQLITE_BUSY;
  }
  //在这里设置文件锁的状态
  pFile->locktype = (u8)newLocktype;
  return rc;
}
```

在加SHARED锁时调用了函数getReadLock(pFile)，这是由于前一小节中所介绍的历史原因造成的，程序如下：

```
/*
获取一个读锁。
*/
static int getReadLock(winFile *pFile){
  int res;
  if( isNT() ){
    OVERLAPPED ovlp;
    ovlp.Offset = SHARED_FIRST;
    ovlp.OffsetHigh = 0;
    ovlp.hEvent = 0;
    res = LockFileEx(pFile->h, LOCKFILE_FAIL_IMMEDIATELY,
                     0, SHARED_SIZE, 0, &ovlp);
  }
  if( res == 0 ){
    pFile->lastErrno = GetLastError();
  }
  return res;
}
```

看了这段程序就知道了，此函数根本没必要。

至此，意尽，不想再往下写。甚至写过的东西都不想再看一遍了。

4. 后记

第 1 次接触 SQLite 是 2009 年 10 月 1 日那天，上午边看国庆阅兵边上网，想找个开源数据库玩玩，选中了 SQLite。当天晚上就真的喜欢上了它，那一刻，缤纷的礼花照耀着南京美丽的夜空。

这两个月中我把所有的业余时间都用到了 SQLite 上，先学习，后分析，没钱，没成就，但有乐趣。没能将 SQLite 的源程序分析完，当我觉着分析得差不多时，兴趣也用得差不多了。SQLite 教会了我分享，所以总觉着在暂时离开 SQLite 之前应该留下点什么。尽管没有动力，在反复朗诵 Richard Hipp 的 3 句话后，还是决定写点东西吧，这一下又用了我近一周的时间。接触 SQLite 时间不长，成文非常仓促，所以本文难免会有很多错误，不是故意误导大家，是真的水平低。如果有兄弟想对我提出指导，我的邮箱是：njgaoyi@yahoo.com.cn。如果我没有回信，不是因为不想回，是因为我很少上网，在此先行谢过。

再朗诵一遍 Richard Hipp 的 3 句话吧：

May you do good and not evil.

May you find forgiveness for yourself and forgive others.

May you share freely, never taking more than you give.

空转

Ver 1.00: 2009-11-28 于南京

5. 附录一：SQLite 的动态内存分配

原文：http://www.sqlite.org/malloc.html
SQLite 通过动态内存分配获得内存来：存储各种对象(如数据库连接或编译后的语句)，建立数据库的内存缓冲区，存放查询的结果。SQLite 在动态内存分配子系统上做了很多努力，使其更加可靠、有预见性、强健和有效。
本文档提供对 SQLite 动态内存分配的概述，目标读者是想要在特定环境中使 SQLite 达到最佳性能的软件工程师。阅读本文档不需要使用 SQLite 的知识。SQLite 的默认设置在绝大多数应用程序中都能表现得很好，但本文档提供的信息对那些想在特殊环境中或为适应特殊需求而使用 SQLite 的工程师是有用的。

1.0 特色

SQLite 的内存分配子系统提供以下性能：

- 对抗分配失败的强健性。如果内存分配失败(malloc()或 realloc()返回了 NULL)，SQLite 将用温和的方法加以处理。首先尝试释放不再重要的(unpinned)缓冲页，而不是重试内存申请。如果还是失败，SQLite 或者停止想做的工作而返回 SQLITE_NOMEM 错误错误代码，或者在没有申请到内存的情况下继续工作。
- 无内存泄漏。应用程序负责释放它分配的任何对象。例如，应用程序必须使用 sqlite3_finalize()来释放编译的语句，使用 sqlite3_close()来关闭数据库连接。只要应用程序协作得当(在应用程序屋不出问题)，SQLite 就不会泄漏内存，即使是在内存分配失败或面临其它操作系统错误的情况下。
- 限定内存使用。sqlite3_soft_heap_limit()机制允许对 SQLite 所使用的内存做出一个限定，SQLite 会尽量不超过此限定。当 SQLite 接近此限定时，它会尽量重用自己缓冲区里的内存而不是申请新的。
- 零 malloc 操作。应用程序在 SQLite 启动时为它提供几个大的缓冲区，这样，在 SQLite 的运行过程中，所有的内存申请都尽量使用这些缓冲区，而不需要调用 malloc()和 free()。
- 应用程序支持的内存分配。应用程序提供方法选择使用启动时分配的内存，此方法用来代替 malloc()和 free()。
- 抗崩溃和碎裂。SQLite 可以通过配置保证不会在内存分配或堆碎裂中崩溃。这个属性对于需要长时间运行和高可靠性的内嵌式系统是重要的，在这样的系统中，一个内存分配错误就可能导致整个系统的失败。
- 内存使用统计。应用程序能够统计当前使用的内存并预测何时会接近或超过限定。
- 内存分配的最少调用。在很多系统中，malloc()和 free()的实现是低效的。SQLite 尽量将 malloc()和 free()的调用次数减到最少。
- 开放式的存取。通过 sqlite3_malloc()、sqlite3_realloc()和 sqlite3_free()接口，SQLite 扩展或应用程序本身都可以使用相同的 SQLite 的底层分配函数来使用内存。

2.0 测试

SQLite 中 75%的代码纯粹是为了测试和验证。可靠性对于 SQLite 非常重要。SQLite 的测试

机制是为了保证不会误用内存分配，不会泄漏内存并正确响应内存分配失败。
SQLite 通过使用一个特殊的工具化的内存分配器来保证不误用内存的动态分配。这个工具化的内存分配器可以通过编译时选项 SQLITE_MEMDEBUG 来激活。它比默认的内存分配器要慢得多，所以不推荐在产品中使用。但在测试阶段，它能提供如下检查：

- 边界检查
- 内存释放后处理
- 释放非 malloc 获得的内存
- 使用前将内存初始化

(此处略去若干字)

3.0 配置

SQLite 默认的内存分配方法可适应绝大多数应用程序。但有些应用程序还是想修改配置以严格适应特殊情况的需要。用编译时和启动时的配置选项都可以。

3.1 改变底层的内存分配器

SQLite 源码中包含了几种不同的内存分配器模块，可以在编译时选择，或者在启动时做有限的扩展。

3.1.1 默认的内存分配器

默认情况下，SQLite 使用标准 C 函数 malloc()、realloc()和 free()来进行内存分配。这些程序只经过简单封装，也提供一个 memsize()函数来返回已分配内存的大小。

3.1.2 The debugging memory allocator

3.1.3 Zero-malloc memory allocator

3.1.4 Experimental memory allocators

3.1.5 Application-defined memory allocators

3.1.6 Memory allocator overlays

3.1.7 No-op memory allocator stub

(此处略去若干字)

3.2 草稿(Scratch)内存

SQLite 偶尔会需要大量的草稿内存来完成一些瞬时计算。例如，在重新平衡一棵 Btree 时所需要的临时内存。这些草稿内存一般是几十 K 字节，只在一个单独的、执行时间很短的函数调用中使用。

在 SQLite 的老版本中，。。。所以，现在 SQLite 是从堆中分配草稿内存。

(此处略去若干字)

3.3 页缓冲区内存

在绝大多数应用程序中，SQLite 的页缓冲子系统都要比其它子系统使用更多(经常是多 10 倍以上)的动态内存。

SQLite 可能通过配置来从一个单独的、分离的、具有固定大小槽位的内存池中分配页缓冲区内存。这样做有两点好处：
因为每次分配内存的大小都相同，可以操作得很快。分配器不需要关心临接空间的接合及寻找合适大小的槽位等情况。所有未分配的空间存储在一个链表当中。分配就是将第一个入口从链表中删除，释放就是将内存入口加入到链表的表头位置。
With a single allocation size, the n parameter in the Robson proof is 1, and the total memory space required by the allocator (N) is exactly equal to maximum memory used (M). No additional memory is required to cover fragmentation overhead, thus reducing memory requirements. This is particularly important for the page cache memory since the page cache constitutes the largest component of the memory needs of SQLite.
3.4 Lookaside memory allocator
3.5 Memory status
3.6 Setting memory usage limits
4.0 Mathematical Guarantees Against Memory Allocation Failures
4.1 Computing and controlling parameters M and n
4.2 Ductile failure
5.0 Stability Of Memory Interfaces
6.0 Summary Of Memory Allocator Interfaces