### 什么是VDBE

VDBE全称为虚拟数据库引擎(virtual database engine)，它是对真实计算机资源环境的一个抽象。它的原理类似于Java中的JVM（java虚拟机）和.NET平台的CLR（公共语言运行库），它们实现了这两种语言跨平台性。

### SQLite为什么要使用VDBE

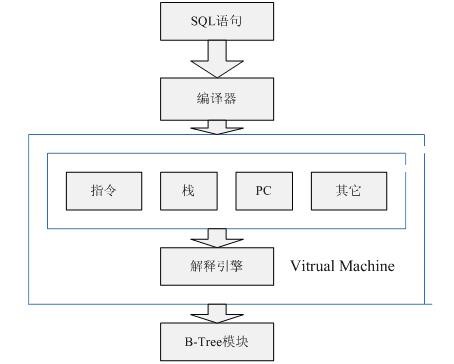
跨平台（个人理解）

流程：1.将SQL语句编译为虚拟机指令 2.虚拟机执行指令（调用B-tree相关的接口）

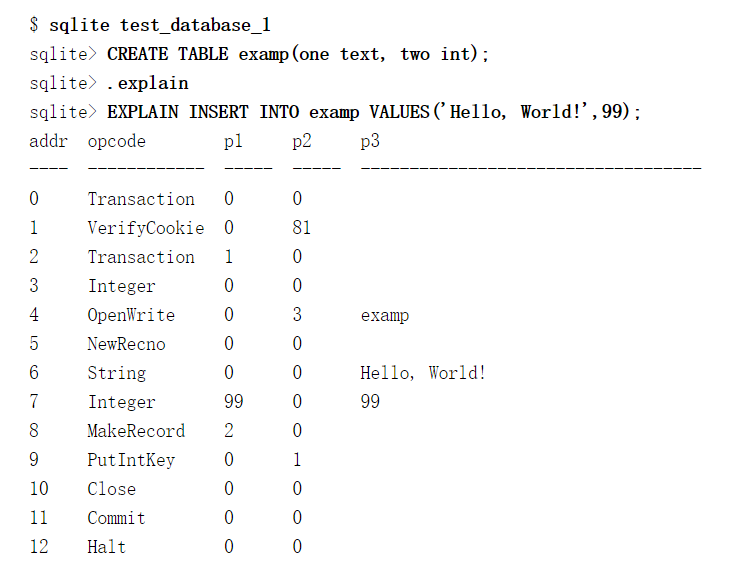
sqLite会根据不同平台实现虚拟机，例如根据Android和iOS平台实现虚拟机的功能。VDBE有自己完善的硬体架构，如处理器、堆栈、寄存器等，还具有相应的指令系统，VDBE屏蔽了与具体操作系统平台相关的信息，使得程序只需生成在虚拟机上运行的目标代码（字节码），就可以在多种平台上不加修改地运行。

B树的主要工作是实现数据库文件的存储。它将所有的页面以树形结构组织起 来,高度优化了索引查找机制。页面缓存服务于B树,主要负责传输以及提高工 作的效率,在磁盘上以B树形式传输页面。

### VDBE的运行



aOp数组保存有SQL经过编译后生成的所有指令。指令计数器(PC)每一个vdbe都有一个程序计数器，用来保存初始的计数器值。经过编译器生成的vdbe指令最终都是由解释引擎解释执行的



上图是一个SQL语句在通过命令成成的字节码指令。在指令中可以看到Transaction指令表示开始一个事务，Commit或Rollback结束一个事务，VerifyCookie检验数据库模式的版本等等。 OpenWrite指令在P1（这儿为0）上即examp表上打开一个读写指针。

备注：B-tree和pager的使用 sqLite的数据都存放在数据库页当中。数据库页当中存储着大量信息（包括记录，字段和索引等）。B-tree负责将页组织为树状结构，方便搜索。Pager根据B-tree的要求对磁盘进行读写操作。（另一种说法：每一个B-tree都对应一个表或者索引。） 总之，每一条SQL语句都会被编译为一系列操作码供VDBE执行。

### VDBE执行方法

vdbe.c文件中存放着VDBE的执行方法（sqlite3VdbeExec），这是VDBE的核心，也是SQLite的核心，SQL解析器生成一个程序然后由VDBE执行SQL语句的工作。VDBE程序在形式上类似于汇编语言。VDBC程序由一系列线性操作组成，每个操作都有1个操作码和5个操作数，操作数P1,P2,P3是整数，操作数P4是一个以null结尾的字符串，操作数P5是一个无符号字符。sqlite3VdbeExec函数用于解析VDBE程序指令,但 是要建立一个程序指令还需要其它文件里的函数的帮助和支撑。 下面是sqlite3VdbeExec函数的说明：

|  |
| --- |
| int sqlite3VdbeExec( Vdbe \*p) { /\* The VDBE \*/  int pc=0; /\* 程序计数器 \*/  Op \*aOp = p->aOp; /\* p->aOp的复本 \*/  Op \*pOp; /\* 当前操作码 \*/  int rc = SQLITE\_OK; /\* 函数返回值 \*/  sqlite3 \*db = p->db; /\* 数据库 \*/  Mem \*aMem = p->aMem; /\* p->aMem的复本 \*/  Mem \*pIn1 = 0; /\* 第一次输入的参数 \*/  Mem \*pIn2 = 0; /\* 第二次输入的参数 \*/  Mem \*pIn3 = 0; /\* 第三次输入的参数 \*/  Mem \*pOut = 0; /\* 输出的参数\*/  int iCompare = 0; /\*存放操作码OP\_Compare的操作结果\*/  int \*aPermute = 0; /\*操作码OP\_Compare使用的数组。\*/  sqlite3VdbeEnter(p); /\*初始化虚拟机程序p的环境\*/  p->rc = SQLITE\_OK;  p->pResultSet = 0;  db->busyHandler.nBusy = 0;  CHECK\_FOR\_INTERRUPT;  sqlite3VdbeIOTraceSql(p);  for(pc=p->pc; rc==SQLITE\_OK; pc++){  pOp = &aOp[pc];  if( pOp->opflags & OPFLG\_OUT2\_PRERELEASE ){  assert( pOp->p2>0 );  assert( pOp->p2<=p->nMem );  pOut = &aMem[pOp->p2];  memAboutToChange(p, pOut);  VdbeMemRelease(pOut);  pOut->flags = MEM\_Int;  }  switch( pOp->opcode ){  //switch语句,每一个case都是在VDBE里执行一个单独的指令，一共155个。  ……  }  } |

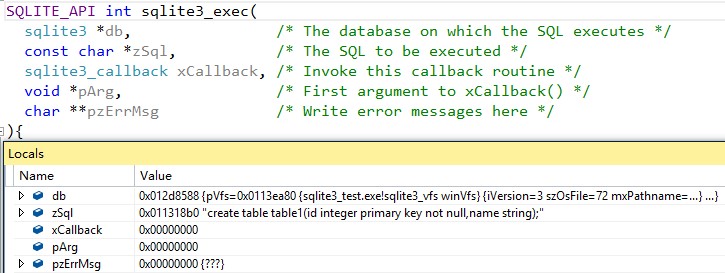
### SQLite执行与VDBE

为了对SQLite执行过程有一个更清晰的认识，我们使用接口部分提供的API函数，编写了一段基本的数据库操作代码，在visual studio环境下，结合着八个模块的顺序进行单步调试，在sqlite3VdbeExec函 数体内，我们也看到了switch语句的传值为pOp->opcode，所以我们重点观察opcode的生成过程和VDBE核心函数sqlite3VdbeExec的执行过程，代码如下：

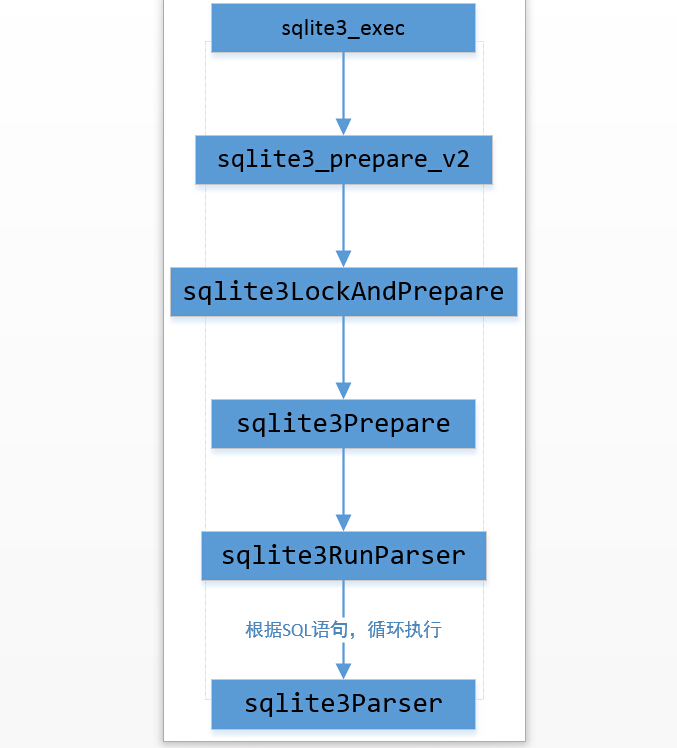
|  |
| --- |
| int main()  {  sqlite3 \*db = NULL;  int result;  result = sqlite3\_open("test.db", &db);  if (SQLITE\_OK != result)  {  printf("Create/Open test.db error! \n");  }  printf("Create/Open test.db success!! \n");  const char \*sqlStr1 = "create table table2(sid integer primary key not null,age string);";  result = sqlite3\_exec(db, sqlStr1, 0, 0, 0);  if (SQLITE\_OK != result)  {  printf("create table table1 error! \n");  return 0;  }  printf("create table table1 success! \n");  const char\* sqlStr2 = "insert into table1() values(1,'name1');";  result = sqlite3\_exec(db, sqlStr2, 0, 0, 0);  if (SQLITE\_OK != result)  {  printf("insert table table1 error! \n");  return 0;  }  printf("insert table table1 success! \n");  sqlite3\_close(db);  return 0;  } |

在代码中我们可以看到几个重要的API函数的调用(sqlite3\_open、sqlite3\_exec、sqlite3\_close)，sqlite3\_exec函数执行我们输入的SQL语句，这些语句包括表的创建和表数据的增删查改，下面就是我们 的调试过程：

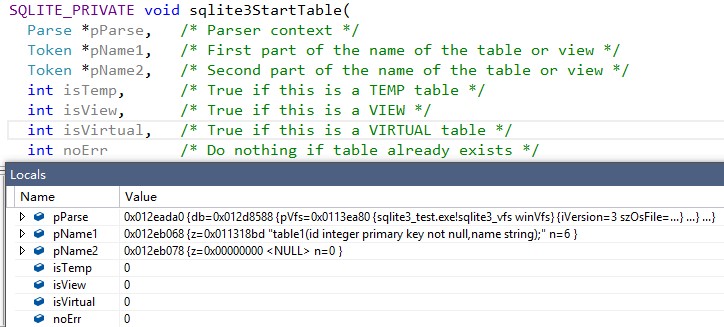
1. 通过在result = sqlite3\_exec(db, sqlStr1, 0, 0, 0)设置断点，进入sqlite3\_exec函数体内部，通过下图4-1可以看到(db, sqlStr1, 0, 0, 0)已经传入：



1. 继续执行sqlite3\_exec函数，由于中间过程非常复杂我们使用下图4-2表示从sqlite3\_exec开始执行到opcode生成之前的过程，也就是准备阶段：

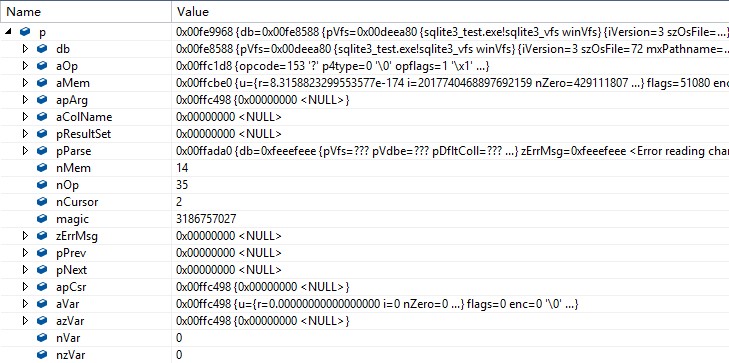


1. 在sqlite3RunParser函数中有分词器功能函数对SQL语句进行分析，分析到SQL语句中的create命令或者表名等，就会进入sqlite3Parser函数，去解析这个命令或者语句，由于我们使用的是create table命令，所以在解析器里会执行sqlite3StartTable函数，图4-3就是sqlite3StartTable函数参数 信息：



从图中我们可以看到表名table1已经在pName1中，虚表、视图和临时表标识都为0，pParse中保存着被分析解释过的SQL语句。sqlite3StartTable函数的功能就是创建表，在函数体中通过调用sqlite3FindTable函数在main数据库中查找到这个表，然后创建了一个VDBE实例v，把OP\_ReadCookie、OP\_If、OP\_Integer、OP\_SetCookie、OP\_CreateTable、OP\_OpenWrite、OP\_NewRowid、OP\_Null、OP\_Insert、OP\_Close等然后执行sqlite3FinishCoding函数，并在sqlite3FinishCoding函数中创建了一个VDBE实例v，把操作符OP\_Init、OP\_Transaction、OP\_TableLock、OP\_Goto和OP\_Halt添加到了v的aOp中，到此为止准备阶段结束，释放占用内存，清除各种结构，代码也从sqlite3\_prepare\_v2跳出。

1. 下面进入sqlite3\_step函数，准备阶段处理好的VDBE实例v也被传入，而进入sqlite3Step后，就会调用VDBE处理函数sqlite3VdbeExec，这个时候v的结构如图4-4所示：



1. 进入sqlite3VdbeExec后，先进行一些变量的初始化和配置，下面进入到for循环中，pc为循环因子，初始为0。然后进入switch语句中，来执行之前生成的操作符。

虚拟数据库引擎(VDBE)

### VDBE SQLite权威指南

VDBE是SQLite的核心，它的上层模块和下层模块本质上都是为它服务的，它的实现位于vbde.c、vdbe.h、vdbeapi.c、vdbeInt.h和vdbemem.c等几个文件中。如第5章所述，一个语句(statement)会编译为一个完整的VDBE程序，执行一条单独的SQL命令。它通过底层的基础设施B-tree执行由编译器(Compiler)生成的字节代码，这种字节代码程序语言是为了进行查询、读取和修改数据库而专门设计的。

字节代码在内存中被封装成sqlite3\_stmt对象(内部叫做Vdbe，见vdbeInt.h)，Vdbe(或者说statement)包含执行程序所需要的一切，包括：

VDBE程序

程序计数器

结果字段的名称和类型

参数的绑定值

运行栈和固定数量的编号的内在单元

其它的运行时状态信息，如B-tree游标

VDBE是一个虚拟机，它的字节代码指令和汇编程序十分类似，每一条指令由操作码和三个操作数构成：<opcode, P1, P2, P3>。Opcode为一定功能的操作码，为了理解，可以看成一个函数。p1是32位的有符号整数，p2是31位的无符号整数，它通常是跳转(jump)指令的目标地址(destination)，当然还有其它用途；p3为一个以null结尾的字符串或者其它结构体的指针。目前SQLite中有128个操作码。和C API不同的是，VDBE操作码经常变化，所以不应该用字节码编写自己的程序。

下面的几个C API直接和VDBE交互：

sqlite3\_bind\_xxx() functions

sqlite3\_step()

sqlite3\_reset()

sqlite3\_column\_xxx() functions

sqlite3\_finalize()

一般情况下，所有的API都是用来执行一个查询并在VDBE相关的结果集中步进操作。它们有一个共同点：都以一个语句句柄做参数。这是因为它们都需要句柄中的VDBE代码或相关资源来完成任务。注意：sqlite3\_prepare()工作于开始阶段，用于产生VDBE代码，它不参与执行。

所有SQL命令的VDBE程序都可以通过EXPLAIN命令得到，如：

|  |
| --- |
| sqlite> .m col  sqlite> .h on  sqlite> .w 4 15 3 3 15  sqlite> explain select \* from episodes;  addr        opcode                  p1    p2    p3  0            Goto                    0     12  1            Integer                 0     0  2            OpenRead             0     2     # episodes  3            SetNumColumns   0     3  4            Rewind                 0     10  5            Recno                   0     0  6            Column                0     1  7            Column                0     2  8            Callback                3     0  9            Next                     0     5  10          Close                    0     0  11          Halt                     0     0  12          Transaction           0     0  13          VerifyCookie         0     10  14          Goto                    0     1  15          Noop                    0     0 |

上面使用了4条命令，前面的命令用于调试和格式化。另外，我在编译SQLite时使用了SQLITE\_DEBUG选项，这个选择可以提供运行栈更多的信息，比如包含在p3里面的表名。

空注：当前版本的SQLite(3.6.18)确实有较大变化，现在执行EXPLAIN命令的结果如下。

|  |
| --- |
| 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    40   0                  00  13    TableLock        0    2    0      episodes    00  14    Goto             0    2    0                  00 |

#### 栈(Stack)

一个VDBE程序通常由几个完成特定任务的段(section)构成，每一个段中都有一些操作栈的指令。这么做是因为不同的指令有不同数量的参数，有些指令只有一个参数；有些指令没有参数；有些指令有好几个参数，这时三个操作数就不够了。

考虑到这些情况，指令采用栈来传递参数。而这些指令本身不会做这些工作，所以在它们之前需要其它一些指令的帮助，以取得需要的参数。VDBE把计算的中间结果保存到内存单元(memory cell)中，其实堆栈和内存单元都基于Mem结构(见vdbeInt.h)。

#### 程序体

让我们来看前面打开episodes表的例子。它的第一个段主要包括指令1~3。

第一条指令(Integer)是为第二条指令作准备的，也就是把第二条指令执行需要的参数压入堆栈，OpenRead从堆栈中取出参数值然后执行。

SQLite可以通过ATTACH命令在一个连接中打开多个数据库文件，每当SQLite打开一个数据库，它就为之赋一个索引号(index)，主数据库的索引为0，附加的第一个数据库为1，依次类推。Integer指令将数据库索引的值压入栈(本例为0，代表主数据库)，而OpenRead从中取出值，并决定操作哪个数据库。它用P2来确定需要打开表的根页(root page)。然后它打开一个指定数据库中指定表的B-tree游标。所有这些在VDBE代码文档中都有解释，例如，OpenRead命令在SQLite文档中的解释如下：

为数据库表打开一个只读游标，这个表的根页在数据库文件的P2处。数据库文件由栈顶的一个整数指定。0表示主数据库，1表示用于存放临时表的数据库。新打开游标的标识符在P1中。P1的值不必是相邻的，但应该是一个小整数。如果其值为负，表示错误。If P2==0 then take the root page number from off of the stack.

只要有游标打开，就会有一个读锁加载到数据库上。如果数据库本来是未加锁的，此命令的部分工作包括获得一个读锁。读锁允许其它进程读数据库，但是禁止任何进程改数据库。读锁在所有游标都关闭时释放。如果此指令在申请读锁时失败，程序结束并返回SQLITE\_BUSY错误码。

P3的值是指向一个结构的指针，该结构定义索引的内容和排序序列的关键信息。当不指向索引时，P3的内容为空。

这个关于OpenRead的文档与其它指令的文档一样，可以直接在源程序文件中找到，特别是vdbe.c中。

最终，SetNumColumns指令设置游标需要处理的列的数量，这是由所要处理的表包含的列数决定的。P1为游标的索引(这里为0，是刚刚打开的游标的索引号)。P2为列的数目，episodes表有三列。

继续本例，Rewind指令将游标设置到表的开始，它会检查表是否为空(“空”即没有记录)。如果没有记录，它会导致指令指针跳转到P2指定的指令处。此处P2为10，即Close指令。一旦Rewind设置游标，接下来就会执行下一段(指令5~9)的几条指令。它们的主要功能是遍历结果集，Recno把由游标P1指定的记录的关键字段值压入堆栈。Column指令从由P1指定的游标，P2指定的列取值。5,6,7三条指令分别把id(primary key)、season和name字段(游标0所指明的表episodes的全部3个字段)的值压入栈。接下来，Callback指令从栈中取出三个值(由P1指定)，然后形成一个记录数组，存储在内存单元 (memory cell) 中。然后，Callback会挂起VDBE的执行，把控制权交给sqlite3\_step()，该函数将返回SQLITE\_ROW。

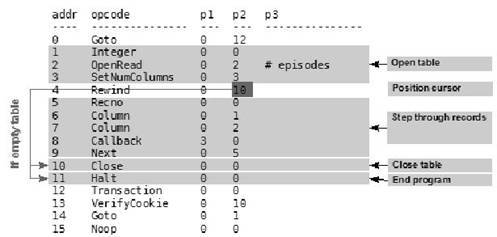


图9-1 VDBE的步骤：Open和Read

一旦VDBE创建了记录结构(该结构同样关联于语句(statement)句柄)，程序就可以通过sqlite3\_column\_xxx() 函数从记录结构内取出字段值。当下次调用sqlite3\_step()时，指令指针会指向Next指令。Next指令会把游标移向表的下一行，如果有其它的记录，它会跳到由P2指定的指令，在这里为指令5(Recno指令)，创建一个新的记录结构，进入下一次循环。如果已经没有其它记录可读，Next不跳转，而是执行下一条指令，这里是Close指令。Close指令会关闭游标，然后执行Halt指令，结束VDBE程序，并且sqlite3\_step()函数会返回SQLITE\_DONE。

#### 程序开始与停止

前面介绍了程序的核心部分，现在来看看其余的指令，这些指令与启动和初始化有关，见图9-2。第一条指令是Goto指令，它是一条跳转指令，跳到P2处，本例中是跳到第12条指令。

指令12是Transaction，它开始一个新的事务；然后执行下一条指令VerifyCookie，它的主要功能是确定VDBE程序编译后，数据库schema是否改变(即是否进行过更新操作)。这在SQLite中是一个很重要的概念，在SQL被sqlite3\_prepare()编译成VDBE代码至程序调用sqlite3\_step()执行字节码的这段时间内，另一个SQL命令可能会改变数据库模式(比如ALTER TABLE、DROP TABLE或CREATE TABLE)。一旦发生这种情况，schema版本就会改变，之前编译的语句(statement)就会变得无效。当前的数据库schema信息记录在数据库文件的根页中。类似地，每个语句都有一份用于比较的在编译时刻该模式的备份，VerifyCookie的功能就是检查它们是否匹配，如果不匹配，就要采取适当的措施。

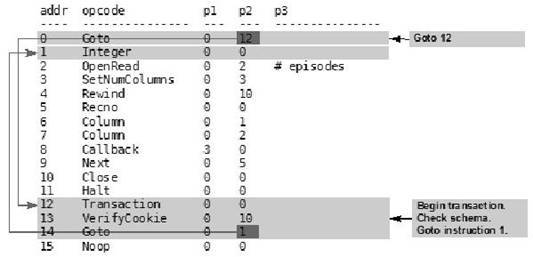


图9-2 VDBE的步骤：程序开始

语句的版本号由VerifyCookie的P2参数指定，将它与磁盘上的数据库schema版本号进行比较。如果schema没有改变，两个版本号应该一致。如果不一致，则VDBE程序失效。在此情况下，VerifyCookie将会终止程序并返回SQLITE\_SCHEMA错误。在此情况下，应用程序需要重新编译SQL语句，基于新的schema版本生成新的VDBE程序。

如果两者匹配，会执行下一条指令Goto；它会跳到程序的主要部分，即第一条指令，打开表读取记录。这里有两点值得注意：

(1)Transaction指令本身不会获取锁。它的功能相当于BEGIN，而共享锁实际是由OpenRead指令获取的。当事务关闭时释放锁，由Halt指令完成，它会进行扫尾工作。

(2) 语句对象(VDBE程序)所需的存储空间在程序执行前就已经确定。这缘于两个重要事实：首先，栈的深度不会比指令的数目还多。其次，内存单元(memory cell)的数量永远不会多于指令的数量(通常少得多)。在执行VDBE程序之前，SQLite可以计算出分配资源所需要的内存。

#### 指令的类型

VDBE同时只会执行一条指令。每条指令都完成一项简单的任务，而且通常和该指令前面、后面的指令有关。大体上来说，指令可分为三类：

(1)处理值：这些指令通常完成算术运算，比如加、减和除；逻辑运算，比如与和或；还有字符串操作。

(2)数据管理：这些指令操作在内存和磁盘上的数据。内存指令进行栈操作或者在内存单元之间传递数据。磁盘操作指令控制B-tree和pager打开或操作游标，开始或结束事务，等等。

(3)流程控制：控制指令主要是有条件地或无条件地移动指令指针。

一旦熟悉了指令集，就不难明白VDBE程序是如何工作的。至少你可以了解如何使用栈来为后面指令的执行做准备。

#### B-Tree和Pager模型

B-tree使VDBE执行查找、插入和删除的效率达到O(logN)，以及在O(1)的效率下双向遍历结果集。它是自平衡的，可自动地执行碎片整理和空间回收。B-tree本身不会直接读写磁盘，它仅仅维护着页(page)之间的关系。当B-tree需要页或者修改页时，它就会调用pager。当修改页时，pager保证原始页首先写入日志文件。当它完成写操作时，pager根据事务状态决定如何做。

#### B-Tree API

B-Tree模块有它自己的API，它可以独立于C API使用。也就是说，如果你愿意，你可以把它当作一个独立的运行库来使用，或在SQLite中直接存取库表。SQLite B-tree模块的另一个好处就是它本身支持事务。由pager处理的事务、锁和日志都是为B-tree服务的。根据功能，可将B-Tree API分为以下几类：

访问和事务函数

包括：

sqlite3BtreeOpen: 打开一个新的数据库文件，返回一个B-tree对象。

sqlite3BtreeClose: 关闭一个数据库。

sqlite3BtreeBeginTrans: 开始一个新的事务。

sqlite3BtreeCommit: 提交当前事务。

sqlite3BtreeRollback: 回卷当前事务。

sqlite3BtreeBeginStmt: 开始一个statement事务。

sqlite3BtreeCommitStmt: 提交一个statement事务。

sqlite3BtreeRollbackStmt: 回卷一个statement事务。

表函数

包括：

sqlite3BtreeCreateTable: 在数据库文件中创建一个新的、空的B-tree。其参数决定是采用表格式(B+tree)还是索引格式(B-tree)。

sqlite3BtreeDropTable: 从数据库中删除一个B-tree。

sqlite3BtreeClearTable: 从B-tree中删除所有数据，但保持B-tree的结构。

游标函数

包括：

sqlite3BtreeCursor: Creates a new cursor pointing to a particular B-tree. Cursors can be either a read cursor or a write cursor. Read and write cursors may not exist in the same B-tree at the same time.

sqlite3BtreeCloseCursor: Closes the B-tree cursor.

sqlite3BtreeFirst: Moves the cursor to the first element in a B-tree.

sqlite3BtreeLast: Moves the cursor to the last element in a B-tree.

sqlite3BtreeNext: Moves the cursor to the next element after the one it is currently pointing to.

sqlite3BtreePrevious: Moves the cursor to the previous element before the one it is currently pointing to.

sqlite3BtreeMoveto: Moves the cursor to an element that matches the key value passed in as a parameter. If there is no match, leaves the cursor pointing to an element that would be on either side of the matching element, had it existed.

记录函数

包括：

sqlite3BtreeDelete: Deletes the record that the cursor is pointing to.

sqlite3BtreeInsert: Inserts a new element in the appropriate place of the B-tree.

sqlite3BtreeKeySize: Returns the number of bytes in the key of the record that the cursor is pointing to.

sqlite3BtreeKey: Returns the key of the record the cursor is currently pointing to.

sqlite3BtreeDataSize: Returns the number of bytes in the data record that the cursor is currently pointing to.

sqlite3BtreeData: Returns the data in the record the cursor is currently pointing to.

配置函数

包括：

sqlite3BtreeSetCacheSize: Controls the page cache size as well as the synchronous writes (as defined in the synchronous pragma).

sqlite3BtreeSetSafetyLevel: Changes the way data is synced to disk in order to increase or decrease how well the database resists damage due to OS crashes and power failures. Level 1 is the same as asynchronous (no syncs() occur and there is a high probability of damage). This is the equivalent to pragma synchronous=OFF. Level 2 is the default. There is a very low but non-zero probability of damage. This is the equivalent to pragma synchronous=NORMAL. Level 3 reduces the probability of damage to near zero but with a write performance reduction. This is the equivalent to pragma synchronous=FULL.

sqlite3BtreeSetPageSize: Sets the database page size.

sqlite3BtreeGetPageSize: Returns the database page size.

sqlite3BtreeSetAutoVacuum: Sets the autovacuum property of the database.

sqlite3BtreeGetAutoVacuum: Returns whether the database uses autovacuum.

sqlite3BtreeSetBusyHandler: Sets the busy handler.

还有其它的函数，所有这些函数在btree.h和btree.c中都有很完备的文档，但上面列出的函数可以使你建立一个总体印象。

#### 编译器

前面已经介绍了VDBE以下直到OS层的各层次。下面介绍VDBE程序是怎么来的。编译器的输入是一个单独的SQL命令，输出是最终经过优化的VDBE程序，这些工作在3个阶段上完成：分词器(Tokenizer)、分析器(Parser)和代码生成器(Code Generator)。

#### 分词器(Tokenizer)

编译的第一步是对SQL命令分词。分词器将一个命令分解成一串单独的词汇(token)。词可以是有特定含义的一个字符或一个字符序列。每个词都有其关联的词类(token class)，词类是一个数字标识，表明这个词是什么。例如左括号的词类是TK\_LP，保留字SELECT的词类是TK\_SELECT。所有词类在parse.h中定义。例如下面的SQL语句：

SELECT rowid FROM foo where name='bar' LIMIT 1 ORDER BY rowid;

经分词器处理之后的部分结果在表9-2中给出。

表9-2 一个分词后SELECT语句

| **文本** | **词类** | **动作** |
| --- | --- | --- |
| SELECT | TK\_SELECT | 发给分析器 |
| " " | TK\_SPACE | 丢弃 |
| Rowid | TK\_ID | 发给分析器 |
| " " | TK\_SPACE | 丢弃 |
| FROM | TK\_FROM | 发给分析器 |
| " " | TK\_SPACE | 丢弃 |
| foo | TK\_ID | 发给分析器 |
| " " | TK\_SPACE | 丢弃 |
| WHERE | TK\_WHERE | 发给分析器 |
| " " | TK\_SPACE | 丢弃 |
| name | TK\_ID | 发给分析器 |
| = | TK\_EQ | 发给分析器 |
| … |  |  |

总之，分词器按照SQL的词法定义把它切分为一个一个的词，并传递给分析器(Parser)进行语法分析(忽略空格)。

##### 保留字

分词器是手工编写的(hand-coded)，主要在Tokenize.c中实现。因为是手工代码，不是用自动生成的代码来对SQL保留字分类。保留字在keywordhash.h文件中定义。这个文件是一个最优化的、将所有SQL保留字压缩到可能最小的缓冲区，方法是公共的字符序列重叠存放。SQLite使用指明了每个保留字偏移量和大小的数组来识别保留字入口。这种方法是一种空间优化的方法，有利于内嵌式的应用程序。一个生成了的缓冲区的例子如下：

static int keywordCode(const char \*z, int n){

static const char zText[537] =

"ABORTABLEFTEMPORARYADDATABASELECTHENDEFAULTRANSACTIONATURALTER"

"AISEACHECKEYAFTEREFERENCESCAPELSEXCEPTRIGGEREGEXPLAINITIALLYANALYZE"

"XCLUSIVEXISTSTATEMENTANDEFERRABLEATTACHAVINGLOBEFOREIGNOREINDEX"

"AUTOINCREMENTBEGINNERENAMEBETWEENOTNULLIKEBYCASCADEFERREDELETE"

"CASECASTCOLLATECOLUMNCOMMITCONFLICTCONSTRAINTERSECTCREATECROSS"

"CURRENT\_DATECURRENT\_TIMESTAMPLANDESCDETACHDISTINCTDROPRAGMATCH"

"FAILIMITFROMFULLGROUPDATEIFIMMEDIATEINSERTINSTEADINTOFFSETISNULL"

"JOINORDEREPLACEOUTERESTRICTPRIMARYQUERYRIGHTROLLBACKROWHENUNION"

"UNIQUEUSINGVACUUMVALUESVIEWHERE";

The keywordhash.h file includes a routine sqlite3KeywordCode(), which allows the tokenizer to quickly match the keyword with its appropriate token class with minimal space. So, the tokenizer first tries to match a token with what it knows, and failing that, it resorts to sqlite3KeywordCode(), which will return either a keyword token class or a generic TK\_ID.

The tokenizer and parser work hand in hand, one token at a time. As the tokenizer resolves each token, it passes the token to the parser. The parser takes the tokens and builds a parse tree, which is a hierarchical representation of the statement.

##### 分析器(Parser)

SQLite的语法分析器是用Lemon生成的(Lemon是一个开源的LALR(1)语法分析器的生成器，SQLite在使用时进行了定制)。该分析器用parse.c内定义的语法规则将一串词组织成层次结构的分析树(parse tree)。

The parse tree is primarily composed of expressions and lists of expressions. An expression itself is a recursive structure that can contain subexpressions under it. For example, the WHERE clause in a SELECT parse tree is represented by a single expression. The SELECT clause, on the other hand, is represented as a list of expressions; each expression is a column that will be returned in the result set. 例如，如下简单的SQL语句：

SELECT rowid, name, season FROM episodes WHERE rowid=1 LIMIT 1

可以组织成如图9-8所示的分析树。

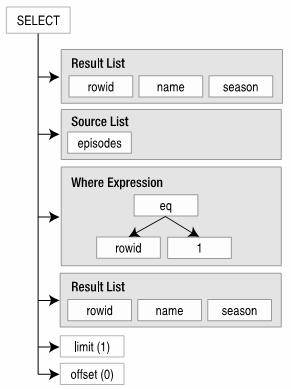


图9-8 简化了的分析树

一旦语句经过分词和分析，分析树作为一种结果会传送给代码生成器。

#### 代码生成器(Code Generator)

代码生成器是SQLite中最庞大、最复杂的部分。与其它模块不同，代码生成器没有定义明确的接口，但它与分析器关系紧密。代码生成器由多个源文件组成，这些源文件大多针对特定的SQL操作。例如，生成SELECT语句的代码在select.c中，其它的源文件包括update.c、insert.c、delete.c和trigger.c等等。

代码生成器根据语法分析树生成VDBE程序。树的每一部分生成一个VDBE指令序列来完成特定的任务。The values for the operands are taken from the data structures associated with the parse tree. 例如，下面是一个读操作中打开表的代码的生成实现：

|  |
| --- |
| /\* Generate code that will open a table for reading.\*/  void sqlite3OpenTableForReading(    Vdbe \*v,        /\* Generate code into this VDBE \*/    int iCur,       /\* The cursor number of the table \*/    Table \*pTab     /\* The table to be opened \*/  ){    sqlite3VdbeAddOp(v, OP\_Integer, pTab->iDb, 0);    sqlite3VdbeAddOp(v, OP\_OpenRead, iCur, pTab->tnum);    VdbeComment((v, "# %s", pTab->zName));    sqlite3VdbeAddOp(v, OP\_SetNumColumns, iCur, pTab->nCol);  } |

Sqlite3vdbeAddOp函数有三个参数：(1)VDBE实例(它将添加指令)，(2)操作码(一条指令)，(3)两个操作数。它增加一条指令到VDBE程序。上例中，sqlite3OpenTableForReading增加了3条指令，即图9-1中的指令1~3，功能是打开一个表的B-tree用于读。为方便起见，将图9-1中的指令序列重列于此：

|  |
| --- |
| sqlite> explain select \* from episodes;  addr        opcode                  p1    p2    p3  0            Goto                    0     12  1            Integer                 0     0  2            OpenRead             0     2     # episodes  3            SetNumColumns   0     3  4            Rewind                 0     10  5            Recno                   0     0  6            Column                0     1  7            Column                0     2  8            Callback                3     0  9            Next                     0     5  10          Close                    0     0  11          Halt                     0     0  12          Transaction           0     0  13          VerifyCookie         0     10  14          Goto                    0     1  15          Noop                    0     0 |

#### 优化

代码生成器不仅负责生成代码，也负责进行查询优化。优化是代码生成的一部分，主要的实现位于where.c中。生成的WHERE子句通常被其它模块共享，比如select.c、update.c和delete.c。这些模块调用sqlite3WhereBegin()开始WHERE语句块的指令生成，然后将返回的代码加入到它们自己的VDBE代码中，最后调用sqlite3WhereEnd()，生成结束WHERE子句代码的VDBE指令。程序的一般结构如图9-9所示：

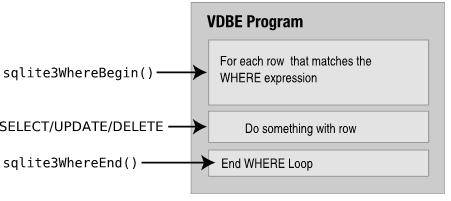


图9-9 WHERE子句的VDBE代码的生成 优化发生在sqlite3WhereBegin()阶段。

它在已完成工作的基础上，寻找可以使用的索引，寻找可以重写的表达式，等等。 为了能对优化先有一个感觉，

我们从一个不含WHERE子句的简单SELECT语句开始，如图9-10所示：

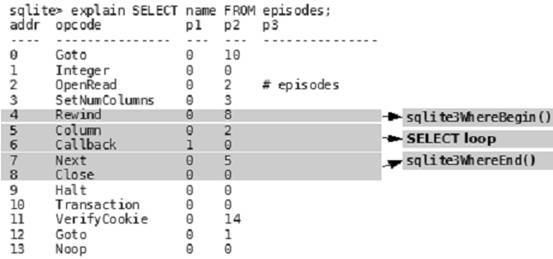


图9-10 一个不含WHERE子句的SELECT语句