

个人简历  
专业打杂程序员  
联系方式  
新浪微博 腾讯微博

IT新闻:  
苹果新Retina MacBook Pro ( 2014年中 ) 开箱图+SSD简单测试 [8分钟前](#)  
网吧里玩出的世界冠军 打场游戏赚了400万 [10分钟前](#)  
Twitter收购深度学习创业公司Madbits [35分钟前](#)  
昵称: YY哥  
园龄: 7年2个月  
粉丝: 342  
关注: 2  
[+加关注](#)

< 2009年3月 >						
日	一	二	三	四	五	六
22	23	24	25	26	27	28
<a href="#">1</a>	2	3	4	5	6	7
8	9	<a href="#">10</a>	11	12	13	14
15	16	17	<a href="#">18</a>	19	20	21
22	23	24	25	26	27	28
29	30	31	1	2	3	4

搜索

找找看

谷歌搜索

常用链接

我的随笔  
我的评论  
我的参与  
最新评论  
我的标签  
[更多链接](#)

随笔分类

c/c++(9)  
Linux相关(24)  
MySQL(11)  
Others(2)  
Web技术(12)  
数据结构与算法(15)  
数据库技术(30)  
系统相关(3)  
云计算与虚拟化(3)

随笔档案

2014年7月 (4)

SQLite入门与分析(七)---浅谈SQLite的虚拟机

写在前面：虚拟机技术在现在是一个非常热的技术，它的历史也很悠久。最早的虚拟机可追溯到IBM的VM/370，到上个世纪90年代，在计算机程序设计语言领域又出现一件革命性的事情——Java语言的出现，它与c++最大的不同在于它必须在Java虚拟机上运行。Java虚拟机掀起了虚拟机技术的热潮，随后，Microsoft也不甘落后，雄心勃勃的推出了.Net平台。由于在这里主要讨论SQLite的虚拟机，不打算对这些做过多评论，但是作为对比，我会先对Java虚拟机作一个概述。好了，下面进入正题。

1、概述

所谓虚拟机是指对真实计算机资源环境的一个抽象，它为解释性语言程序提供了一套完整的计算机接口。虚拟机的思想对现在的编译有很大影响，其思路是先编译成虚拟机指令，然后针对不同计算机实现该虚拟机。

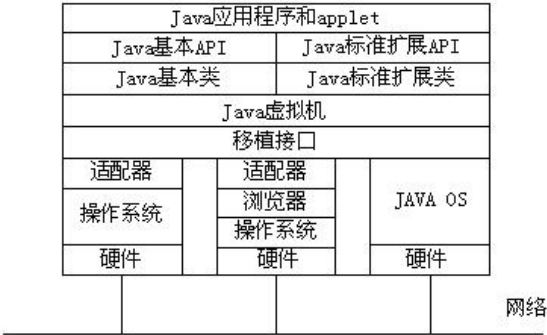
虚拟机定义了一组抽象的逻辑组件，这些组件包括寄存器组、数据栈和指令集等等。虚拟机指令的解释执行包括3步：

1. 获取指令参数；
2. 执行该指令对应的功能；
3. 分派下一条指令。

其中第一步和第三步构成了虚拟机的执行开销。

很多语言都采用了虚拟机作为运行环境。作为下一代计算平台的竞争者，Sun的Java和微软的.NET平台都采用了虚拟机技术。Java的支撑环境是Java虚拟机（Java Virtual Machine, JVM），.NET的支撑环境是通用语言运行库（Common Language Runtime, CLR）。JVM是典型的虚拟机架构。

Java平台结构如图所示。从图中可以看出，JVM处于核心位置，它的下方是移植接口。移植接口由依赖平台的不依赖平台的两部分组成，其中依赖于平台的部分称为适配器。JVM通过移植接口在具体的操作系统上实现。如果在Java操作系统（Java Operation System, JOS）上实现，则不需要依赖于平台的适配器，因为这部分工作已由JOS完成。因此对于JVM来说，操作系统和更低的硬件层是透明的。在JVM的上方，是Java类和Java应用程序接口（Java API）。在Java API上可以编写Java应用程序和Java小程序（applet）。所以对于Java应用程序和applet这一层次来说，操作系统和硬件就更是透明的了。我们编写的Java程序，可以在任何Java平台上运行而无需修改。



JVM定义了独立于平台的类文件格式和字节码形式的指令集。在任何Java程序的字节码表示形式中，变量和方法的引用都是使用符号，而不是使用具体的数字。由于内存的布局要在运行时才确定，所以类的变量和方法的改变不会影响现存的字节码。例如，一个Java程序引用了其他系统中的某个类，该系统中那个类的更新不会使这个Java程序崩溃。这也提高了Java的平台独立性。

虚拟机一般都采用了基于栈的架构，这种架构易于实现。虚拟机方法显著提高了程序语言的可移植性和安全性，但同时也导致了执行效率的下降。

2、Java虚拟机

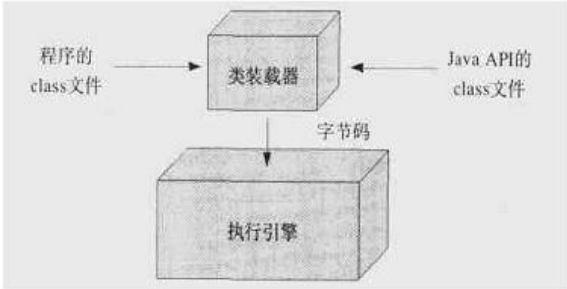
2.1、概述

Java虚拟机的主要任务是装载Class文件并执行其中的字节码。Java虚拟机包含一个类装载器(class loader)，它从程序和API中装载class文件，Java API中只有程序运行时需要的那些类才会被装载，字节码由执行引擎来执行。

不同的Java虚拟机，执行引擎的实现可能不同。在软件实现的虚拟机中，一般有几下几种实现方式：

- （1）解释执行：实现简单，但速度较慢，这是Java最初阶段的实现方式。
- （2）即时编译(just-in-time)：执行较快，但消耗内存。在这种情况下，第一次执行的字节码会编译成本地机器代码，然后被缓存，以后可以重用。
- （3）自适应优化器：虚拟机开始的时候解释字节码，但是会监视程序的运行，并记录下使用最频繁的代码，然后把这些代码编译成本地代码，而其它的代码仍保持为字节码。该方法既提高的运行速度，又减少了内存开销。

同样，虚拟机也可由硬件来实现，它用本地方法执行Java字节码。



2.2、Java虚拟机

Java虚拟机的结构分为：类装载子系统，运行时数据区，执行引擎，本地方法接口。其中运行时数据区又分为：方法区，堆，Java栈，PC寄存器，本地方法栈。

- 2014年3月 (1)
- 2013年9月 (1)
- 2013年8月 (1)
- 2013年2月 (1)
- 2012年11月 (4)
- 2012年1月 (1)
- 2011年12月 (1)
- 2011年10月 (1)
- 2011年3月 (1)
- 2010年9月 (1)
- 2010年8月 (1)
- 2010年7月 (3)
- 2010年6月 (2)
- 2010年5月 (7)
- 2010年4月 (1)
- 2010年3月 (1)
- 2010年1月 (1)
- 2009年12月 (2)
- 2009年10月 (2)
- 2009年9月 (14)
- 2009年8月 (4)
- 2009年6月 (14)
- 2009年5月 (3)
- 2009年4月 (1)
- 2009年3月 (3)
- 2009年2月 (11)
- 2008年10月 (7)
- 2008年8月 (5)
- 2008年7月 (1)
- 2008年6月 (2)
- 2008年5月 (2)
- 2008年4月 (5)

kernel

kernel中文社区  
LDN  
The Linux Document Project  
The Linux Kernel Archives

manual

cppreference  
gcc manual  
mysql manual

sites

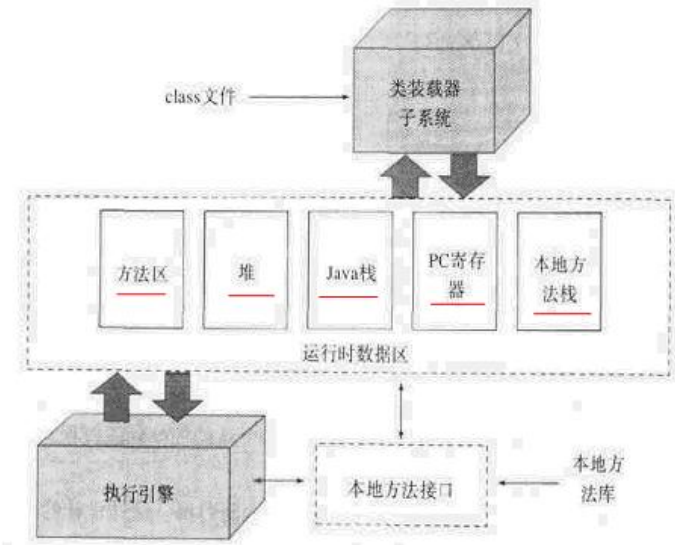
Database Journal  
Fedora镜像  
highscalability  
KFUPM ePrints  
Linux docs  
Linux Journal  
NoSQL  
SQLite

技术社区

apache  
CSDN  
IBM-developerworks  
lucene中国  
nutch中国  
oldlinux  
oracle's forum

最新评论

1. Re:理解MySQL——架构与概念  
我试验了下.数据 5 9 10 13 18begin;select \* from asf\_execution where num> 5 and num 5 and INSTANCE\_ID\_<18 lock in share mode;会有 1.行锁 2.间隙锁 [5 18]插



关于Java虚拟机就介绍到此,由于Java虚拟机内容庞大,在这里不可能一一介绍,如果想更多了解Java虚拟机,参见《深入Java虚拟机》。

3、SQLite虚拟机

在SQLite的后端 ( backend ) 的上一层,通常叫做虚拟数据库引擎(virtual database engine),或者叫做虚拟机(virtual machine)。从作用上来说,它是SQLite的核心。用户程序发出的SQL语句请求,由前端(frontend)编译器 ( 以后会继续介绍 ) 处理,生成字节代码程序 ( bytecode programs ), 然后由VM解释执行。VM执行时,又会调用B-tree模块的相关的接口,并输出执行的结果 ( 本节将以一个具体的查询过程来描述这一过程 )。

3.1、虚拟机的内部结构

先来看一个简单的例子:

```
int main(int argc, char **argv)
{
    int rc, i, id, cid;
    char *name;
    char *sql;
    char *zErr;
    sqlite3 *db; sqlite3_stmt *stmt;
    sql="select id,name,cid from episodes";
    //打开数据库
    sqlite3_open("test.db", &db);
    //编译sql语句
    sqlite3_prepare(db, sql, strlen(sql), &stmt, NULL);
    //调用VM, 执行VDBE程序
    rc = sqlite3_step(stmt);

    while(rc == SQLITE_ROW) {
        id = sqlite3_column_int(stmt, 0);
        name = (char *)sqlite3_column_text(stmt, 1);
        cid = sqlite3_column_int(stmt, 2);
        if(name != NULL){
            fprintf(stderr, "Row: id=%i, cid=%i, name='%s'\n", id,cid,name);
        } else {
            /* Field is NULL */
            fprintf(stderr, "Row: id=%i, cid=%i, name=NULL\n", id,cid);
        }
        rc = sqlite3_step(stmt);
    }
    //释放资源
    sqlite3_finalize(stmt);
    //关闭数据库
    sqlite3_close(db);
    return 0;
}
```

这段程序很简单,它的功能就是遍历整个表,并把查询结果输出。  
在SQLite中,用户发出的SQL语句,都会由编译器生成一个虚拟机实例。在上面的例子中,变量sql代表的SQL语句经过sqlite3\_prepare()处理后,便生成一个虚拟机实例——stmt。虚拟机实例从外部看到的结构是sqlite3\_stmt所代表的的数据结构,而在内部,是一个vdbe数据结构代表的实例。  
关于这点可以看看它们的定义:  
//sqlite3.h

入INSERT I.....

--麒麟飞

2. Re:理解MySQL——架构与概念

例1-5

insert into t(i) values(1);

这句话应该是可以插入的。

不会被阻塞

--麒麟飞

3. Re:理解MySQL——架构与概念

注：SELECT ... FOR UPDATE仅在自动提交关闭(即手动提交)时才会对元组加锁，而在自动提交时，符合条件的元组不会被加锁。

这个是错误的。自动提交的，也会尝试获取排它锁。

你可以试验下。

--麒麟飞

4. Re:浅谈mysql的两阶段提交协议

YY哥 偶像啊!细腻文笔 配有说服力的代码和图 我崇拜你 !!!

之前sqlite的深入分析帮了我大忙..

现在做mysql相关 有来你的博客找东西 哈哈!!

--hark.perfe

5. Re:(i++)+(i++)与(++i)+(++i)

@arrowcat

这类语句本身没什么意义，但是楼主思考的角度让我豁然开朗。

--HJWAJ

阅读排行榜

1. 理解MySQL——索引与优化(77627)

2. SQLite入门与分析(一)---简介(48610)

3. 理解MySQL——复制(Replication)(26209)

4. libevent源码分析(19048)

5. SQLite入门与分析(二)---设计与概念(16977)

评论排行榜

1. (i++)+(i++)与(++i)+(++i)(40)

2. SQLite入门与分析(一)---简介(30)

3. 浅谈SQLite——实现与应用(20)

4. 一道算法题,求更好的解法(18)

5. 理解MySQL——索引与优化(16)

推荐排行榜

1. SQLite入门与分析(一)---简介(12)

2. 理解MySQL——索引与优化(12)

3. 浅谈SQLite——查询处理及优化(10)

4. 乱谈服务器编程(9)

5. libevent源码分析(6)

typedef struct sqlite3\_stmt sqlite3\_stmt;

vdbe的定义：

␣

//虚拟机数据结构 vdbeInt.h

struct Vdbe {

sqlite3 \*db; /\* The whole database \*/

Vdbe \*pPrev,\*pNext; /\* Linked list of VDBEs with the same Vdbe.db \*/

FILE \*trace; /\* Write an execution trace here, if not NULL \*/

int nOp; /\* Number of instructions in the program(指令的条数) \*/

int nOpAlloc; /\* Number of slots allocated for aOp[] \*/

Op \*aOp; /\* Space to hold the virtual machine's program(指令) \*/

int nLabel; /\* Number of labels used \*/

int nLabelAlloc; /\* Number of slots allocated in aLabel[] \*/

int \*aLabel; /\* Space to hold the labels \*/

Mem \*aStack; /\* The operand stack, except string values(栈空间) \*/

Mem \*pTos; /\* Top entry in the operand stack(栈顶指针) \*/

Mem \*\*apArg; /\* Arguments to currently executing user function \*/

Mem \*aColName; /\* Column names to return \*/

int nCursor; /\* Number of slots in apCsr[] \*/

Cursor \*\*apCsr; /\* One element of this array for each open cursor(游标数组) \*/

int nVar; /\* Number of entries in aVar[] \*/

Mem \*aVar; /\* Values for the OP\_Variable opcode \*/

char \*\*azVar; /\* Name of variables \*/

int okVar; /\* True if azVar[] has been initialized \*/

int magic; /\* Magic number for sanity checking \*/

int nMem; /\* Number of memory locations currently allocated \*/

Mem \*aMem; /\* The memory locations(保存临时变量的Mem) \*/

int nCallback; /\* Number of callbacks invoked so far(回调的次数) \*/

int cacheCtr; /\* Cursor row cache generation counter \*/

Fifo sFifo; /\* A list of ROWIDs \*/

int contextStackTop; /\* Index of top element in the context stack \*/

int contextStackDepth; /\* The size of the "context" stack \*/

Context \*contextStack; /\* Stack used by opcodes ContextPush & ContextPop \*/

int pc; /\* The program counter(初始程序计数器) \*/

int rc; /\* Value to return(返回结果) \*/

unsigned uniqueCnt; /\* Used by OP\_MakeRecord when P2!=0 \*/

int errorAction; /\* Recovery action to do in case of an error \*/

int inTempTrans; /\* True if temp database is transactioned \*/

int returnStack[100]; /\* Return address stack for OP\_Gosub & OP\_Return \*/

int returnDepth; /\* Next unused element in returnStack[] \*/

int nResColumn; /\* Number of columns in one row of the result set \*/

char \*\*azResColumn; /\* Values for one row of result \*/

int popStack; /\* Pop the stack this much on entry to VdbeExec()(出栈的项数) \*/

char \*zErrMsg; /\* Error message written here \*/

u8 resOnStack; /\* True if there are result values on the stack(有结果在栈上则为真) \*/

u8 explain; /\* True if EXPLAIN present on SQL command \*/

u8 changeCntOn; /\* True to update the change-counter \*/

u8 aborted; /\* True if ROLLBACK in another VM causes an abort \*/

u8 expired; /\* True if the VM needs to be recompiled \*/

u8 minWriteFileFormat; /\* Minimum file format for writable database files \*/

int nChange; /\* Number of db changes made since last reset \*/

i64 startTime; /\* Time when query started - used for profiling \*/

#ifdef SQLITE\_SSE

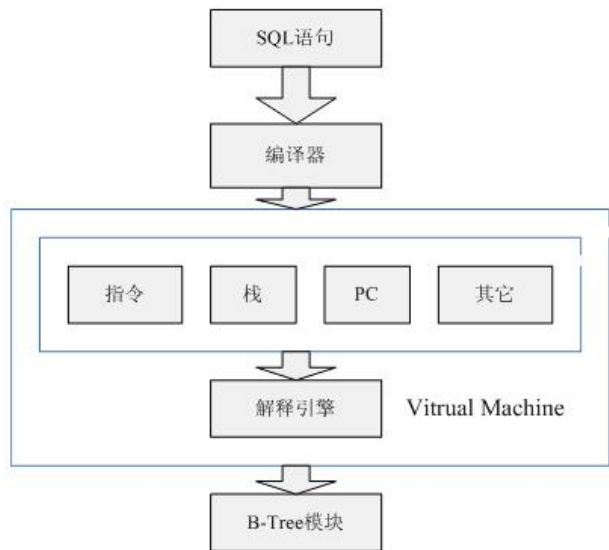
int fetchId; /\* Statement number used by sqlite3\_fetch\_statement \*/

int lru; /\* Counter used for LRU cache replacement \*/

#endif

};

由vdbe的定义，可以总结出SQLite虚拟机的内部结构：



### 3.2、指令

```

int nOp;          /* Number of instructions in the program(指令的条数) */
Op *aOp;          /* Space to hold the virtual machine's program(指令)*/
  
```

aOp数组保存有SQL经过编译后生成的所有指令，对于上面的例子为：

```

0、Goto(0x5b-91)    |0|0c
1、Integer(0x2d-45) |0|0
2、OpenRead(0x0c-12)|0|2
3、SetNumColumns(0x64-100)|0|03
4、Rewind(0x77-119) |0|0a
5、Rowid(0x23-35)   |0|0
6、Column(0x02-2)   |0|1
7、Column(0x02-2)   |0|2
8、Callback(0x36-54)|3|0
9、Next(0x68)       |0|5
10、Close
11、Halt
12、Transaction(0x66-102)|0|0
13、VerifyCookie(0x61-97)|0|1
14、Goto(0x5b-91)    |0|1|
  
```

sqlite3\_step()引起VDBE解释引擎执行这段代码，下面来分析该段指令的执行过程：

Goto：这是一条跳转指令，它的作用仅仅是跳到第12条指令；

Transaction：开始一个事务（读事务）；

Goto：跳到第1条指令；

Integer：把操作数P1入栈，这里的0表示OpenRead指令打开的数据库的编号；

OpenRead：打开表的游标,数据库的编号从栈顶中取得，P1为游标的编号，P2为root page。

如果P2<=0,则从栈中取得root page no；

SetNumColumns：对P1确定的游标的列数设置为P2（在这里为3），在OP\_Column指令执行前,该指令应该被调用来

设置表的列数；

Rewind：移动当前游标（P1）移到表或索引的第一条记录；

Rowid：把当前游标（P1）指向的记录的关键字压入栈；

Column：解析当前游标指定的记录的数据，p1为当前游标索引号，p2为列号，并将结果压入栈中；

Callback：该指令执行后，PC将指向下一条指令。该指令的执行会结束sqlite3\_step()的运行，并向其返回

SQLITE\_ROW——如果存在记录的话；并将VDBE的PC指针指向下一条指令——即Next指令，所以当

重新调用sqlite3\_step()执行VDBE程序时，会执行Next指令（具体的分析见后面的指令实例分析）；

Next：将游标移到下一条记录，并将PC指向第5条指令；

Close：关闭数据库。

### 3.3、栈

```

Mem *aStack;      /* The operand stack, except string values(栈空间) */
Mem *pTos;        /* Top entry in the operand stack(栈顶指针) */
  
```

aStack是VDBE执行时使用的栈，它主要用来保存指令执行需要的参数，以及指令执行时产生的中间结果(参见后面的指令实例分析)。

在计算机硬件领域，基于寄存器的架构已经压倒基于栈的架构成为当今的主流，但是在解释性的虚拟机领域，基于栈架构的实现

占了上风。

1. 从编译的角度来看,许多编程语言可以很容易地被编译成栈架构机器语言。如果采用寄存器架构,编译器为了获得好的性能必须进行优化,如全局寄存器分配(这需要对数据流进行分析)。这种复杂的优化工作使虚拟机的便捷性大打折扣。
2. 如果采用寄存器架构, **虚拟机必须经常保存和恢复寄存器中的内容**。与硬件计算机相比,这些操作在虚拟机中的开销要大得多。因为每一条虚拟机指令都需要进行很费时的指令分派操作。虽然其它的指令也要分派,但是它们的语义内容更丰富。
3. 采用寄存器架构时,指令对应的操作数位于不同寄存器中,对操作数的寻址也是一个问题。而在基于栈的虚拟机中,操作数位于栈顶或紧跟在虚拟机指令之后。由于基于栈的架构的简便性,一些查询语言的实现也采用了此种架构。

SQLite的虚拟机就是基于栈架构的实现。每一个vdbe都有一个栈顶指针,它保存着vdbe的初始栈顶值。而在解释引擎中也有一个pTos, **它们是有区别的**:

(1) vdbe的pTos: 在一趟vdbe执行的过程中不会变化,直到相应的指令修改它为止,在上面的例子中,Callback指令会修改其值(见指令分析)。

(2) 而解释引擎中的pTos是随着指令的执行而动态变化的,在上面的例子中,Integer.Column指令的执行都会引起解释引擎pTos的改变。

### 3.4、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。和pTos一样,解释引擎也有一个pc,它用来指向VM下一条要执行的指令。

### 3.5、解释引擎

经过编译器生成的vdbe最终都是由解释引擎解释执行的,SQLite的解释引擎实现的原理非常简单,本质上就是一个包含大量case语句的for循环,但是由于SQLite的指令较多(在version 3.3.6中是139条),所以代码比较庞大。

SQLite的解释引擎是在一个方法中实现的:

```
int sqlite3VdbeExec(
    Vdbe *p          /* The VDBE */
)
```

具体代码如下(为了阅读,去掉了一些不影响阅读的代码,具体见SQLite的源码):



```
/*执行VDBE程序.当从数据库中取出一行数据时,该函数会调用回调函数(如果有的话),
**或者返回SQLITE_ROW.
**/
int sqlite3VdbeExec(
    Vdbe *p          /* The VDBE */
){
    //指令计数器
    int pc;           /* The program counter */
    //当前指令
    Op *pOp;          /* Current operation */
    int rc = SQLITE_OK; /* Value to return */
    //数据库
    sqlite3 *db = p->db; /* The database */

    u8 encoding = ENC(db); /* The database encoding */
    //栈顶
    Mem *pTos;         /* Top entry in the operand stack */

    if( p->magic!=VDBE_MAGIC_RUN ) return SQLITE_MISUSE;

    //当前栈顶指针
    pTos = p->pTos;

    if( p->rc==SQLITE_NOMEM ){
        /* This happens if a malloc() inside a call to sqlite3_column_text() or
        ** sqlite3_column_text16() failed. */
        goto no_mem;
    }
    p->rc = SQLITE_OK;
    //如果需要进行出栈操作,则进行出栈操作
    if( p->popStack ){
        popStack(&pTos, p->popStack);
        p->popStack = 0;
    }
    //表明栈中没有结果
    p->resOnStack = 0;
    db->busyHandler.nBusy = 0;

    //执行指令
    for(pc=p->pc; rc==SQLITE_OK; pc++){
        //取出操作码
        pOp = &p->aOp[pc];

        switch( pOp->opcode ){
            //跳到操作数P2指向的指令
            case OP_Goto: { /* no-push */
                CHECK_FOR_INTERRUPT;
```



```

        //设置pc
        pc = pOp->p2 - 1;
        break;
    }

    //P1入栈
    case OP_Integer: {
        //当前栈顶指针上移
        pTos++;
        //设为整型
        pTos->flags = MEM_Int;
        //取操作数P1,并赋值
        pTos->i = pOp->p1;
        break;
    }

    //其它指令的实现...
} //end switch
} //end for
}

```



### 3.6、指令实例分析

由于篇幅限制,仅给出几条的指令的实现,其它具体实现见源码。

#### 1、Callback指令

```

/*
 * 该指令执行后,PC将指向下一条指令.
 * 栈中栈顶的P1个值为查询的结果.该指令会导致sqlite3_step()函数将以SQLITE_ROW为返回码
 * 而结束运行.此时用户程序就可以通过sqlite3_column_XXX读取位于栈中的数据了.
 * 当sqlite3_step()再一次运行时,栈顶的P1个值会在执行Next指令前自动出栈.
 */
case OP_Callback: {
    /* no-push */
    Mem *pMem;
    Mem *pFirstColumn;
    assert( p->nResColumn==pOp->p1 );

    /* Data in the pager might be moved or changed out from under us
     * in between the return from this sqlite3_step() call and the
     * next call to sqlite3_step(). So deephemeralize everything on
     * the stack. Note that ephemeral data is never stored in memory
     * cells so we do not have to worry about them.
     */
    pFirstColumn = &pTos[0-pOp->p1];
    for(pMem = p->aStack; pMem<pFirstColumn; pMem++){
        Deephemeralize(pMem);
    }

    /* Invalidate all ephemeral cursor row caches */
    p->cacheCtr = (p->cacheCtr + 2)|1;

    /* Make sure the results of the current row are \000 terminated
     * and have an assigned type. The results are deephemeralized as
     * as side effect.
     */
    for( ; pMem<=pTos; pMem++ ){
        sqlite3VdbeMemNulTerminate(pMem);
        //设置结果集中的数据类型
        storeTypeInfo(pMem, encoding);
    }

    /* Set up the statement structure so that it will pop the current
     * results from the stack when the statement returns.
     */
    p->resOnStack = 1; //栈上有结果
    p->nCallback++; //回调次数加1
    //出栈的数据个数,在下次执行VDBE时,会先进行出栈操作
    p->popStack = pOp->p1;
    //程序计数器加1
    p->pc = pc + 1;

    //设置vdb的栈顶指针,此时,栈中保存有结果
    p->pTos = pTos;
}

```

```

/*注意:这里不是break,而是return; 向sqlite3_step()返回SQLITE_ROW.
**当用户程序重新调用sqlite3_step()时,重新执行VDBE.
*/
return SQLITE_ROW;
}

```

## 2、Rewind指令

```

☐
/*移动当前游标到表或索引的第一条记录.
**如果表为空且p2>0,则跳到p2处;如果p2为0且表不空,则执行下一条指令.
*/
case OP_Rewind: {          /* no-push */
    int i = pOp->p1;
    Cursor *pC;
    BtCursor *pCrsr;
    int res;

    assert( i>=0 && i<p->nCursor );
    //取得当前游标
    pC = p->apCsr[i];
    assert( pC!=0 );
    if( (pCrsr = pC->pCursor)!=0 ){
        //调用B-tree模块,移动游标到第一条记录
        rc = sqlite3BtreeFirst(pCrsr, &res);
        pC->atFirst = res==0;
        pC->deferredMoveto = 0;
        pC->cacheStatus = CACHE_STALE;
    }else{
        res = 1;
    }
    pC->nullRow = res;
    if( res && pOp->p2>0 ){
        pc = pOp->p2 - 1;
    }
    break;
}

```

## 3、Column指令

```

☐
/*解析当前游标指定的记录的数据
**p1为当前游标索引号,p2为列号
*/
case OP_Column: {
    u32 payloadSize; /* Number of bytes in the record */
    int p1 = pOp->p1; /* P1 value of the opcode */
    //列号
    int p2 = pOp->p2; /* column number to retrieve */
    //VDBE游标
    Cursor *pC = 0; /* The VDBE cursor */
    char *zRec; /* Pointer to complete record-data */
    //btree游标
    BtCursor *pCrsr; /* The BTree cursor */
    u32 *aType; /* aType[i] holds the numeric type of the i-th column */
    u32 *aOffset; /* aOffset[i] is offset to start of data for i-th column */
    //列数
    u32 nField; /* number of fields in the record */
    int len; /* The length of the serialized data for the column */
    int i; /* Loop counter */
    char *zData; /* Part of the record being decoded */
    Mem sMem; /* For storing the record being decoded */

    sMem.flags = 0;
    assert( p1<p->nCursor );
    //栈顶指针上移
    pTos++;
    pTos->flags = MEM_Null;

    /* This block sets the variable payloadSize to be the total number of
    ** bytes in the record.
    **
    ** zRec is set to be the complete text of the record if it is available.
    ** The complete record text is always available for pseudo-tables
    ** If the record is stored in a cursor, the complete record text
    ** might be available in the pC->aRow cache. Or it might not be.
    ** If the data is unavailable, zRec is set to NULL.
    */
}

```

```
**
** We also compute the number of columns in the record. For cursors,
** the number of columns is stored in the Cursor.nField element. For
** records on the stack, the next entry down on the stack is an integer
** which is the number of records.
**/
//设置游标
pC = p->apCsr[p1];

assert( pC!=0 );
if( pC->pCursor!=0 ){
    /* The record is stored in a B-Tree */
    //移到当前游标
    rc = sqlite3VdbeCursorMoveto(pC);
    if( rc ) goto abort_due_to_error;
    zRec = 0;
    pCrsr = pC->pCursor;
    if( pC->>nullRow ){
        payloadSize = 0;
    }else if( pC->cacheStatus==p->cacheCtr ){
        payloadSize = pC->payloadSize;
        zRec = (char*)pC->aRow;
    }else if( pC->isIndex ){
        i64 payloadSize64;
        sqlite3BtreeKeySize(pCrsr, &payloadSize64);
        payloadSize = payloadSize64;
    }else{
        //解析数据,payloadSize保存cell的数据字节数
        sqlite3BtreeDataSize(pCrsr, &payloadSize);
    }
    nField = pC->nField;
}else if( pC->pseudoTable ){
    /* The record is the sole entry of a pseudo-table */
    payloadSize = pC->nData;
    zRec = pC->pData;
    pC->cacheStatus = CACHE_STALE;
    assert( payloadSize==0 || zRec!=0 );
    nField = pC->nField;
    pCrsr = 0;
}else{
    zRec = 0;
    payloadSize = 0;
    pCrsr = 0;
    nField = 0;
}

/* If payloadSize is 0, then just push a NULL onto the stack. */
if( payloadSize==0 ){
    assert( pTos->flags==MEM_Null );
    break;
}

assert( p2<nField );

/* Read and parse the table header. Store the results of the parse
** into the record header cache fields of the cursor.
**/
if( pC && pC->cacheStatus==p->cacheCtr ){
    aType = pC->aType;
    aOffset = pC->aOffset;
}else{
    u8 *zIdx;          /* Index into header */
    u8 *zEndHdr;        /* Pointer to first byte after the header(指向header之后的第一个字节)*/
    u32 offset;         /* Offset into the data */
    int szHdrSz;        /* Size of the header size field at start of record */
    int avail;          /* Number of bytes of available data */

    //数据类型数组
    aType = pC->aType;
    if( aType==0 ){
        //每个数据类型分配8字节---sizeof(aType)==4
        pC->aType = aType = sqliteMallocRaw( 2*nField*sizeof(aType) );
    }
    if( aType==0 ){
        goto no_mem;
    }
    //每列数据的偏移
```



```

pC->aOffset = aOffset = &aType[nField];
pC->payloadSize = payloadSize;
pC->cacheStatus = p->cacheCtr;

/* Figure out how many bytes are in the header */
if( zRec ){
    zData = zRec;
}else{
    if( pC->isIndex ){
        zData = (char*)sqlite3BtreeKeyFetch(pCrsr, &avail);
    }else{
        //获取数据
        zData = (char*)sqlite3BtreeDataFetch(pCrsr, &avail);
    }
    /* If KeyFetch()/DataFetch() managed to get the entire payload,
    ** save the payload in the pC->aRow cache. That will save us from
    ** having to make additional calls to fetch the content portion of
    ** the record.
    */
    if( avail>=payloadSize ){
        zRec = zData;
        pC->aRow = (u8*)zData;
    }else{
        pC->aRow = 0;
    }
}
assert( zRec!=0 || avail>=payloadSize || avail>=9 );
//获得header size
szHdrSz = GetVarint((u8*)zData, offset);

/* The KeyFetch() or DataFetch() above are fast and will get the entire
** record header in most cases. But they will fail to get the complete
** record header if the record header does not fit on a single page
** in the B-Tree. When that happens, use sqlite3VdbeMemFromBtree() to
** acquire the complete header text.
*/
if( !zRec && avail<offset ){
    rc = sqlite3VdbeMemFromBtree(pCrsr, 0, offset, pC->isIndex, &sMem);
    if( rc!=SQLITE_OK ){
        goto op_column_out;
    }
    zData = sMem.z;
}

/* 一个记录的例子:
** 08 | 08 |04 00 13 01 | 63 61 74 01
** 08: nSize,payload总的大小—后面8个字节
** 08: 关键字大小,对于整型则为关键字本身
** 04: header size, 包括本身共4个字节—04 00 13 01
** 00: 第一列的数据类型—空类型
** 13: 第二列的数据类型—字符串,长为(19-13)/2=3—"cat"
** 01: 第三列的数据类型—整型, 占一个字节—1
** 对于这里的zData保存的数据为:04 00 13 01 63 61 74 01
*/
//header之后的数据,对于上例为:63 61 74 01
zEndHdr = (u8 *)&zData[offset];
//header数据的索引号,对于上例为:00 13 01
zIdx = (u8 *)&zData[szHdrSz];

/* Scan the header and use it to fill in the aType[] and aOffset[]
** arrays. aType[i] will contain the type integer for the i-th
** column and aOffset[i] will contain the offset from the beginning
** of the record to the start of the data for the i-th column
*/
/*扫描header, 然后设置aType[]和aOffset[]数组; aType[i]为第i列的数据类型,
**aOffset[i]为第i列数据相对于记录的开始的偏移.
*/
for(i=0; i<nField; i++){
    if( zIdx<zEndHdr ){
        //计算每一列数据的偏移
        aOffset[i] = offset;
        //计算每一列的数据类型
        zIdx += GetVarint(zIdx, aType[i]);
        //offset指向下一列
        offset += sqlite3VdbeSerialTypeLen(aType[i]);
    }else{
        /* If i is less than nField, then there are less fields in this
        ** record than SetNumColumns indicated there are columns in the

```

```

    /** table. Set the offset for any extra columns not present in
    ** the record to 0. This tells code below to push a NULL onto the
    ** stack instead of deserializing a value from the record.
    */
    aOffset[i] = 0;
}
}
Release(&sMem);
sMem.flags = MEM_Null;

/* If we have read more header data than was contained in the header,
** or if the end of the last field appears to be past the end of the
** record, then we must be dealing with a corrupt database.
*/
if( zIdx>zEndHdr || offset>payloadSize ){
    rc = SQLITE_CORRUPT_BKPT;
    goto op_column_out;
}
}

/* Get the column information. If aOffset[p2] is non-zero, then
** deserialize the value from the record. If aOffset[p2] is zero,
** then there are not enough fields in the record to satisfy the
** request. In this case, set the value NULL or to P3 if P3 is
** a pointer to a Mem object.
*/
//获取P2指定的列的数据
if( aOffset[p2] ){
    assert( rc==SQLITE_OK );
    if( zRec ){
        //取得该列的数据
        zData = &zRec[aOffset[p2]];
    }else{
        len = sqlite3VdbeSerialTypeLen(aType[p2]);
        rc = sqlite3VdbeMemFromBtree(pCrsr, aOffset[p2], len, pC->isIndex,&sMem);
        if( rc!=SQLITE_OK ){
            goto op_column_out;
        }
        zData = sMem.z;
    }
    //解析zData, 并将结果保存在pTos中
    sqlite3VdbeSerialGet((u8*)zData, aType[p2], pTos);
    pTos->enc = encoding;
}else{
    if( pOp->p3type==P3_MEM ){
        sqlite3VdbeMemShallowCopy(pTos, (Mem *) (pOp->p3), MEM_Static);
    }else{
        pTos->flags = MEM_Null;
    }
}

/* If we dynamically allocated space to hold the data (in the
** sqlite3VdbeMemFromBtree() call above) then transfer control of that
** dynamically allocated space over to the pTos structure.
** This prevents a memory copy.
*/
if( (sMem.flags & MEM_Dyn)!=0 ){
    assert( pTos->flags & MEM_Ephem );
    assert( pTos->flags & (MEM_Str|MEM_Blob) );
    assert( pTos->z==sMem.z );
    assert( sMem.flags & MEM_Term );
    pTos->flags &= ~MEM_Ephem;
    pTos->flags |= MEM_Dyn|MEM_Term;
}

/* pTos->z might be pointing to sMem.zShort[]. Fix that so that we
** can abandon sMem */
rc = sqlite3VdbeMemMakeWriteable(pTos);

op_column_out:
break;
}

```

#### 4、Next指令

```

☐
/*移动游标, 使其指向表的下一个记录

```

```
*/
case OP_Prev:          /* no-push */
case OP_Next: {        /* no-push */
    Cursor *pC;
    BtCursor *pCrshr;

    CHECK_FOR_INTERRUPT;
    assert( pOp->p1>=0 && pOp->p1<p->nCursor );
    pC = p->apCsr[pOp->p1];
    assert( pC!=0 );
    if( (pCrshr = pC->pCursor)!=0 ){
        int res;
        if( pC->>nullRow ){
            res = 1;
        }else{
            assert( pC->deferredMoveto==0 );
            //调用B-tree模块, 移动游标指向下一条记录
            rc = pOp->opcode==OP_Next ? sqlite3BtreeNext(pCrshr, &res) :
                                     sqlite3BtreePrevious(pCrshr, &res);

            pC->>nullRow = res;
            pC->cacheStatus = CACHE_STALE;
        }
        if( res==0 ){
            pc = pOp->p2 - 1;
            sqlite3_search_count++;
        }
    }else{
        pC->>nullRow = 1;
    }
    pC->rowidIsValid = 0;
    break;
}
```

分类: 数据库技术

绿色通道:


好文要顶

关注我

收藏该文

与我联系



YY哥

关注 - 2

粉丝 - 342

+加关注

00

(请您对文章做出评价)

« 上一篇: [SQLite入门与分析\(六\)---再谈SQLite的锁](#)  
» 下一篇: [c++中的const与指针](#)

posted @ 2009-03-18 18:59 YY哥 阅读(6929) 评论(6) 编辑 收藏

评论列表

- #1楼 2009-03-18 23:42 u+ajax[未注册用户]  
终于看到你分析虚拟机了，谢谢你的分享，可能这种东西只有知音才可以有共鸣。你写得很认真。非常感谢。
- #2楼 2009-03-19 12:48 Soli  
很长  
支持(0) 反对(0)
- #3楼 2009-03-19 12:53 !A.Z[未注册用户]  
虚拟“数据库”？
- #4楼 2009-03-19 18:53 neoragex2002  
有点意思。请楼主也分析一下如何将SQL编译成对应的基于栈的中间代码形式，及sqlite是如何进行编译优化的，多谢。  
支持(0) 反对(0)
- #5楼 2009-05-16 11:03 阿超—  
楼主是否能把这个虚拟机提出来,作为单独的可重用的虚拟机来用呢？  
支持(0) 反对(0)
- #6楼 2010-12-12 01:00 阿超—

?

支持(0) 反对(0)

[刷新评论](#) [刷新页面](#) [返回顶部](#)

注册用户登录后才能发表评论，请 [登录](#) 或 [注册](#)，[访问网站首页](#)。

[博客园首页](#) [博问](#) [新闻](#) [闪存](#) [程序员招聘](#) [知识库](#)



**最新IT新闻:**

- 苹果新Retina MacBook Pro（2014年中）开箱图+SSD简单测试
  - 网吧里玩出的世界冠军 打场游戏赚了400万
  - Twitter收购深度学习创业公司Madbits
  - 这两个前亚马逊员工要把亚马逊赶出印度
  - Twitter财报中你不能错过的6个数据
- » [更多新闻...](#)

**最新知识库文章:**

- 如何在网页中使用留白
  - SQL/NoSQL两大阵营激辩：谁更适合大数据
  - 如何获取（GET）一杯咖啡——星巴克REST案例分析
  - 为什么程序员的工作效率跟他们的工资不成比例
  - 我眼里的DBA
- » [更多知识库文章...](#)