**2. 数据库机器(Database Machine)**

2.1 数据流

在本项目中，数据库机是实际操作数据库文件的部分，DBM相当于一个cpu，用于逐条处理dbm指令，操作寄存器和游标来返回sql语句的结果。

dbm对用户透明，用户使用SQL操作数据库，而真正操作具体数据的是dbm指令。

其输入输出如图2-1-1所示

2.2 DBM中的数据结构,

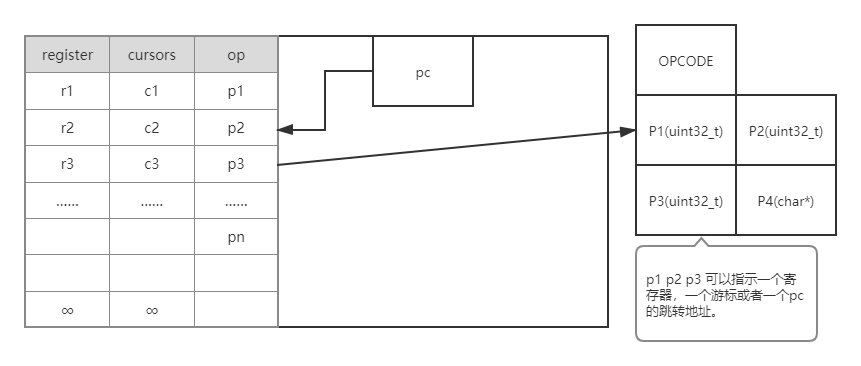
图2-1-1



如图2-2-1所示，一个dbm中含有无穷的寄存器和游标，在dbm运行时pc从op这个数组中逐条取出命令并执行，每条指令都可能对某个寄存器或游标进行操作

1. 寄存器可以储存整数，字符串或者无类型的比特串
2. 游标指向某个表的具体项，并且可以操作cursor快速移动，

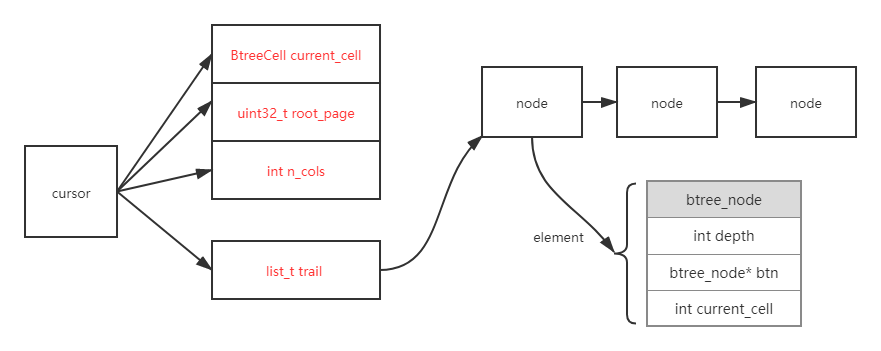
图2-2-1



游标的结构如图2-2-2所示，每个游标都具体指向表中某元组(current\_cell),并且储存该表对应B树的根节点(root\_page)和表所含列数(n\_cols)

最后是一个用于记录从根节点到含有该游标所指元组的路径的一个链表trail，在trail中，其元素记录了当前B树深度(depth),所在的B树节点以及用于指示路径的current\_cell标识

图2-2-2

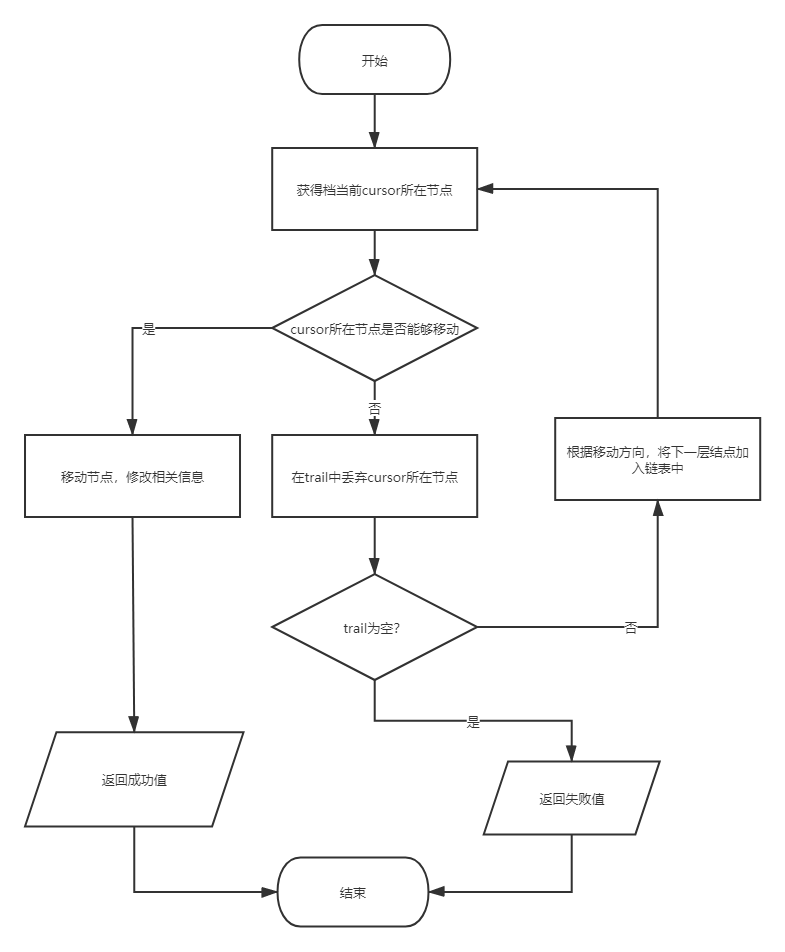


2.3 cursor移动的实现

虽然在一个表中的每一项记录在逻辑上是连续的，但在物理储存结构上两条记录可能不相连，所以在对游标进行移动操作时会涉及到切换结点的问题，在游标中使用链表记录从根节点到对应B树叶节点的路径。根据链表中的记录，能够快速找到其父节点，从而能够快速完成叶节点的切换。

游标进行next或prev的流程如图2-3-1所示

图2-3-1



对cursor进行seek操作的流程图如图2-3-2所示

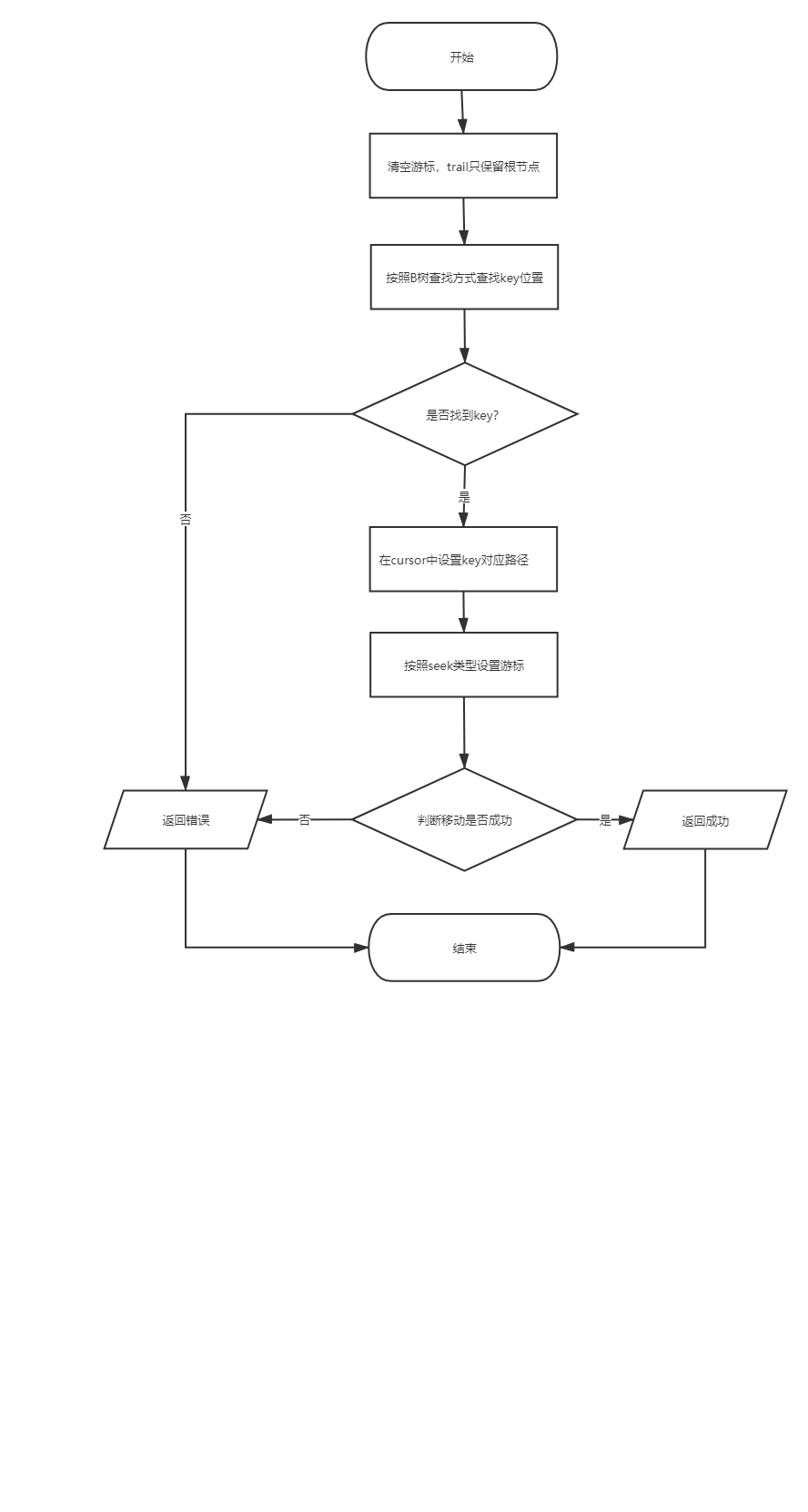


图2-3-2

2.4 dbm向上提供的接口函数及参数

所有的dbm操作函数都只有两个参数stmt 和 op但不同的指令格式不同

**//通过op->opcode 返回执行对应命令的函数**

int chidb\_dbm\_op\_handle (chidb\_stmt \*stmt, chidb\_dbm\_op\_t \*op)

**/\*寄存器赋值操作类**

**\*Integer String Null Copy SCopy**

**\*与数据库无关，仅仅是根据命令参数向对应的寄存器赋值**

**\*Integer op->p1 value op->p2 r**

**\*将p1 中的值放入 指示的寄存器中**

**\*String op->p1 length op->p2 r op->p4 string**

**\*将长度为Length的字符串存入寄存器r**

**\*/**

int chidb\_dbm\_op\_Integer (chidb\_stmt \*stmt, chidb\_dbm\_op\_t \*op)

int chidb\_dbm\_op\_String (chidb\_stmt \*stmt, chidb\_dbm\_op\_t \*op)

int chidb\_dbm\_op\_Null (chidb\_stmt \*stmt, chidb\_dbm\_op\_t \*op)

**/\*寄存器操作函数**

**\*op->p1 r1 op->p2 r2**

**\*将寄存器r1 中的内容拷贝到r2 中 Scopy 是浅拷贝\*/**

int chidb\_dbm\_op\_Copy (chidb\_stmt \*stmt, chidb\_dbm\_op\_t \*op)

int chidb\_dbm\_op\_SCopy (chidb\_stmt \*stmt, chidb\_dbm\_op\_t \*op)

**/\*指令跳转相关函数**

**\*Eq Ne Lt Le Gt Ge**

**\*与数据库无关，仅仅是对于指令是否按照给定的值跳转**

**\*op->p1 r1 op->p3 r2 op->p2 address\_j**

**\*比较r1 r2 两个寄存器中的值，如果比较结果与对应命令相同，则将指令跳转到address\_j 的位置\*/**

int chidb\_dbm\_op\_Eq (chidb\_stmt \*stmt, chidb\_dbm\_op\_t \*op)

int chidb\_dbm\_op\_Ne (chidb\_stmt \*stmt, chidb\_dbm\_op\_t \*op)

int chidb\_dbm\_op\_Lt (chidb\_stmt \*stmt, chidb\_dbm\_op\_t \*op)

int chidb\_dbm\_op\_Le (chidb\_stmt \*stmt, chidb\_dbm\_op\_t \*op)

int chidb\_dbm\_op\_Gt (chidb\_stmt \*stmt, chidb\_dbm\_op\_t \*op)

int chidb\_dbm\_op\_Ge (chidb\_stmt \*stmt, chidb\_dbm\_op\_t \*op)

**/\*根据给定的page打开\关闭游标**

**\*op->p1 cursor 1 op->p2 column\_number**

**\*打开指定游标，并记录对应的行数\*/**

int chidb\_dbm\_op\_OpenRead (chidb\_stmt \*stmt, chidb\_dbm\_op\_t \*op)

int chidb\_dbm\_op\_OpenWrite (chidb\_stmt \*stmt, chidb\_dbm\_op\_t \*op)

int chidb\_dbm\_op\_Close (chidb\_stmt \*stmt, chidb\_dbm\_op\_t \*op)

**/\*重置游标,使其指向第一个单元**

**\*Rewind**

**\*如果游标对应的B树为空，则执行跳转**

**\*op->p1 cursor c op->p2 address\_j\*/**

int chidb\_dbm\_op\_Rewind (chidb\_stmt \*stmt, chidb\_dbm\_op\_t \*op)

**/\*移动游标**

**\*Next Prev Seek SeekGt SeekLt SeekGe SeekLe**

**\*op->p1 cursor c op->p2 address\_j op->p3 key**

**\*按照给定的key对游标进行一次移动操作**

**\*移动失败则进行指令跳转\*/**

int chidb\_dbm\_op\_Next (chidb\_stmt \*stmt, chidb\_dbm\_op\_t \*op)

int chidb\_dbm\_op\_Prev (chidb\_stmt \*stmt, chidb\_dbm\_op\_t \*op)

int chidb\_dbm\_op\_Seek (chidb\_stmt \*stmt, chidb\_dbm\_op\_t \*op)

int chidb\_dbm\_op\_SeekGt (chidb\_stmt \*stmt, chidb\_dbm\_op\_t \*op)

int chidb\_dbm\_op\_SeekGe (chidb\_stmt \*stmt, chidb\_dbm\_op\_t \*op)

int chidb\_dbm\_op\_SeekLt (chidb\_stmt \*stmt, chidb\_dbm\_op\_t \*op)

int chidb\_dbm\_op\_SeekLe (chidb\_stmt \*stmt, chidb\_dbm\_op\_t \*op)

**/\*以索引为判断条件的跳转函数**

**\*op->p1 cursor c op->p2 address\_j op->p3 Idkey**

**\*判断成功后执行跳转\*/**

int chidb\_dbm\_op\_IdxGt (chidb\_stmt \*stmt, chidb\_dbm\_op\_t \*op)

int chidb\_dbm\_op\_IdxGe (chidb\_stmt \*stmt, chidb\_dbm\_op\_t \*op)

int chidb\_dbm\_op\_IdxLt (chidb\_stmt \*stmt, chidb\_dbm\_op\_t \*op)

int chidb\_dbm\_op\_IdxLe (chidb\_stmt \*stmt, chidb\_dbm\_op\_t \*op)

int chidb\_dbm\_op\_IdxPKey (chidb\_stmt \*stmt, chidb\_dbm\_op\_t \*op)

**/\*返回表中的结果**

**\*Column op->p1 cursor c op->p2 column n op->p3 register r**

**\*将游标c所指的某个单元格中第n列数据存入寄存器r中 (n从0开始)\*/**

int chidb\_dbm\_op\_Column (chidb\_stmt \*stmt, chidb\_dbm\_op\_t \*op)

**/\*返回cursor所指项的关键字值**

**\*op->p1 cursor c op->p3 register r**

**\*如果cursor所指向的项是索引项，则返索引对应的值Pkey\*/**

int chidb\_dbm\_op\_Key (chidb\_stmt \*stmt, chidb\_dbm\_op\_t \*op)

**/\***

**\*记录要返回的其实行号和行数**

**\*op->p1 StartRow op->p2 nRR\*/**

int chidb\_dbm\_op\_ResultRow (chidb\_stmt \*stmt, chidb\_dbm\_op\_t \*op)

**/\***

**\*向表中添加记录**

**\*op->p1 startregister r op->p2 number n op->p3 register r2**

**\*向表中添加从r1 到 r1+n-1 中所记录的数据并且将这些记录存入r2中\*/**

int chidb\_dbm\_op\_MakeRecord (chidb\_stmt \*stmt, chidb\_dbm\_op\_t \*op)

**/\*向表中添加一个新的记录**

**\*op->p1 cursor op->p2 register r1 op->p3 r2**

**\*获得r1所存的数据，并将其存入r2所指示的关键字的表中，并且cursor指向新的单元\*/**

int chidb\_dbm\_op\_Insert (chidb\_stmt \*stmt, chidb\_dbm\_op\_t \*op)

int chidb\_dbm\_op\_IdxInsert (chidb\_stmt \*stmt, chidb\_dbm\_op\_t \*op)

**/\*创建表**

**\*op->p1 register r1**

**\*在B树中添加一个新的节点，并且将这个节点对应的页号存入r1 中\*/**

int chidb\_dbm\_op\_CreateTable (chidb\_stmt \*stmt, chidb\_dbm\_op\_t \*op)

int chidb\_dbm\_op\_CreateIndex (chidb\_stmt \*stmt, chidb\_dbm\_op\_t \*op)

**//中断DBM，退出程序**

int chidb\_dbm\_op\_Halt (chidb\_stmt \*stmt, chidb\_dbm\_op\_t \*op)

**3. 代码生成(Database Machine)**

完成以下功能:

1. 实现读取 Schema 表
2. 实现简单 Select 语句到 DBM 指令的代码生成
3. 实现简单 Insert 语句到 DBM 指令的代码生成
4. 实现 Create Table 语句到 DBM 指令的代码生成

**3.1 例子**

CREATE TABLE products(code INTEGER PRIMARY KEY, name TEXT, price INTEGER);

翻译为

Integer 1 0 \_ \_

OpenWrite 0 0 5 \_

CreateTable 4 \_ \_ \_

String 5 1 \_ "table"

String 8 2 \_ "products"

String 8 3 \_ "products"

String 73 5 \_ "CREATE TABLE products(code INTEGER PRIMARY KEY, name TEXT, price INTEGER)"

MakeRecord 1 5 6 \_

Integer 1 7 \_ \_

Insert 0 6 7 \_

Close 0 \_ \_ \_

**3.2 数据流图**



其中词法分析器和语法分析器由 chidb 提供

**3.3 具体实现**

**3.3.1. 读取 Schema 表**

**3.3.1.1 定义 Schema Item 类型**

在 chidbInt.h 中定义 chidb\_schema\_item\_t 类型, 结构定义如下

typedef struct

{

char \*type;

char \*name;

char \*assoc;

int root\_page;

chisql\_statement\_t \*stmt;

} chidb\_schema\_item\_t;

其包含了 Schema 表中每一行的记录信息

**3.3.1.2 修改 chidb 结构体**

在 chidbInt.h 中修改 struct chidb 的定义

从

struct chidb

{

BTree \*bt;

};

修改为

typedef list\_t chidb\_schema\_t;

struct chidb

{

BTree \*bt;

chidb\_schema\_t schema;

int need\_refresh;

};

在 chidb 结构体中增加了一个包含记录的列表和表示是否需要更新的成员 need\_refresh, 其中 need\_refresh 会在 Create Table 语句之后置为 1, 会导致重新读取 Schema 表

**3.3.1.3 实现读取 Schema 表**

在 src/libchidb/api.c 中声明函数

int load\_schema(chidb \*db, npage\_t nroot);

函数定义步骤如下:

1. 读取页码为 nroot 的页, 遍历页中所有的 cells
2. 如果当前结点是表内部结点, 对其 child\_page 调用 load\_schema
3. 如果当前结点是叶子结点, 则解析其包含的数据, 加入到 schema 中
4. 遍历完成之后, 如果结点非叶子结点, 则对其 right\_page 调用 load\_schema

**3.3.1.4 修改打开和关闭文件时的过程**

修改 src/libchidb/api.c 中函数 chidb\_open 和 chidb\_close 的定义

分别添加读取 Schema 和释放 Schema 的步骤

**3.3.2. 简单 Select 语句的代码生成**

chisql 中, sql 语句会被解析成 chisql\_statement\_t 类型的数据, 其中 Select 语句的 type 字段值为 STMT\_SELECT, 而其结构被解析成递归定义的 SRA 结构, 存储在 chisql\_statement\_t.select 中, SRA 结构的递归定义如下

data SRA = Table TableReference

| Project SRA [Expression]

| Select SRA Condition

| NaturalJoin [SRA]

| Join [SRA] (Maybe JoinCondition)

| OuterJoin [SRA] OJType (Maybe JoinCondition)

| Union SRA SRA

| Except SRA SRA

| Intersect SRA SRA

data OJType = Left

| Right

| Full

data ColumnReference = ColumnReference (Maybe String) String

data TableReference = TableName String (Maybe String)

data JoinCondition = On Condition

| Using [String]

在本实践中, 只实现一个受限的 Select 语句的子集, 包含以下限制:

1. 查询只包含一个单独的表
2. 查询包含若干个列或者是 '\*'
3. 查询可以包含 where 子句, 但是 where 只会包含一个单独的条件, 并且其格式为 column op value, 其中运算符只可能是 =, >, >=, < 或者 <=, 以及值的类型只能是整型或字符串类型

由于上述限制的存在, 本实践中可解析得到的 SRA 结构只会是以下两种形式:

1. 不包含 where 子句的查询

Project([columns or '\*'],

Table(table\_name)

)

1. 包含 where 子句的查询

Project([columns or '\*'],

Select(column op value,

Table(table\_name)

)

)

**3.3.2.1 错误检查**

在进行具体的代码生成之前, 需要先对解析后的结果进行检查, 检查包含三个步骤:

1. 要查询的表必须存在
2. 要查询的列必须存在于要查询的表中
3. 如果 where 子句存在, 列的类型必须和所给的值的类型相同

**3.3.2.2 代码生成**

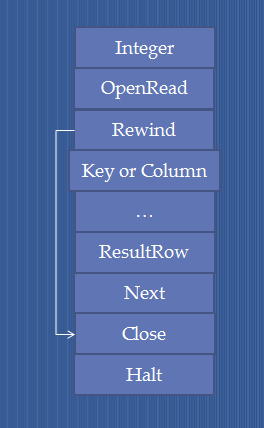


Figure 1

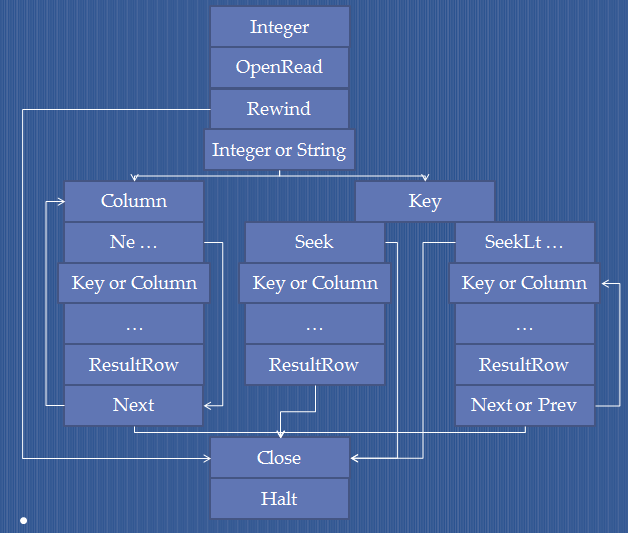


Figure 2

Figure1 为无Where子句时生成的示例代码，Figure2为有Where子句时示意图。

1. 根据表名查找到表所在的根页码 nroot, 生成 Integer 指令存储 nroot 的值到寄存器并生成 OpenRead 指令读取出 nroot 并以只读模式打开其对应的页与游标相对应
2. 创建 Rewind 指令, 当表为空时跳转到结束指令 ( Close ), 但目前其跳转目标暂时不确定, 记为 n1
3. 若 where 子句存在, 将比较中的值通过相应指令 ( Integer 或 String ) 存储到寄存器中, 若列为第一列则生成 Key 指令, 并通过 Seek 指令簇 完成比较, 跳转目标记为 n2, 标记 after\_next 为 1, 否则通过 Column 指令获取对应的列, 并通过 Eq 等指令完成比较, 生成的比较指令的跳转目标记为 n2, 需要跳转到 Next 或者 Prev 指令, 同时将后续可能生成的 Next 或 Prev 指令的跳转目标记为 n3, 当列为第一列时 n3 为比较指令的下一条, 否则 n3 指向 Column 指令, 特殊的, 当列为第一列且比较为相等比较时, 只产生 Seek 指令且不后续不产生 Next 或 Prev 指令, 故 n3 设为 -1
4. 遍历要查询的列名, 并获取其在表中的位置, 通过 Column 指令存储到寄存器中
5. 通过 ResultRow 指令将上一步存储到寄存器中的值生成一条记录存储在寄存器中
6. 当 n3 为 -1 时, 不需要 Next 或 Prev 指令, 当第 3 步产生的比较指令是 SeekLe 或 SeekLt 时产生一条 Prev 指令, 否则产生 Next 指令, 其跳转目标为 n3, 当 after\_next 值为 1 时, n2 指向 Next 或 Prev 指令之后, 否则指向 Next 或 Prev 指令
7. 生成 Close 指令关闭游标关联的页
8. 生成 Halt 指令停机

在上述步骤完成之后定义结果集, 设置结果集的起始寄存器和寄存器的个数为查找的列数, 为列申请空间并拷贝列名

**3.3.3. 简单 Insert 语句的代码生成**

在本实践中, 只实现一个受限的 Insert 语句的子集, 包含以下限制:

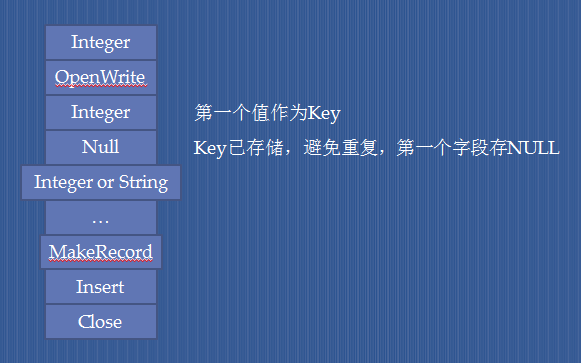
1. Insert 语句总是包含所有列的值, 没有默认值
2. 表名之后不跟随若干列名, 换而言之, 只支持 Insert Into table\_name Values(values...);
3. 插入的值只支持整型和字符串类型

**3.3.3.1 错误检查**

在进行具体的代码生成之前, 需要先对解析后的结果进行检查, 检查包含两个步骤:

1. 要插入的值的表必须存在
2. 值的类型必须和列的类型相匹配

**3.3.3.2 代码生成**



1. 生成 Integer 和 OpenWrite 指令以读写模式打开要插入值的表所在的根页
2. 第一列为Key，生成Integer指令，因为Key与第一列重复，故为记录中第一列值生成Null指令
3. 遍历剩余的值, 通过与值类型对应的指令( Integer 或 String )存储在连续的寄存器上
4. 将上述的值通过 MakeRecord 指令生成一行记录存储在寄存器中
5. 生成 Insert 指令将记录与Key插入到对应的表上
6. 生成 Close 指令关闭打开的页

**3.3.4. Create Table 语句的代码生成**

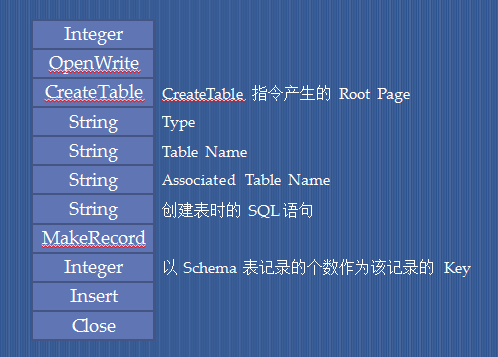
在本实践中, 只实现一个受限的 Create Table 语句的子集, 包含以下限制:

1. 每一列都以 列名 类型 的格式定义, 类型可以是整型或字符串类型
2. 第一列总是主键约束的整型 INTEGER PRIMARY KEY
3. 表中不再有其他的约束

**3.3.4.1 错误检查**

只检查一项, 即要创建的表名是否已经存在

**3.3.4.2 代码生成**



1. 生成 Integer 和 OpenWrite 指令以读写模式打开 Schema 表
2. 通过 CreateTable 指令新建一个表, 并将其所在页码存储在寄存器 4 上
3. 按照顺序分别将创建的类型("table"), 表名, 关联名, SQL 语句 存储在寄存器 1, 2, 3, 5 上
4. 生成 MakeRecord 指令将寄存器 1-5 上的值生成一条记录
5. 生成 Integer 指令存储 Schema 记录的个数作为新插入记录的 Key
6. 生成 Insert 指令将记录和 Key 插入进 Schema 表中
7. 生成 Close 指令关闭打开的页