SQLite之SQL解析-词法分析-6 - 简书

start_time: 2024-09-08 20:59:45 +0800

SQLite之SQL解析-词法分析-6

includes IP属地: 浙江
0.1 2019.07.14 14:53 字数 3322

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.

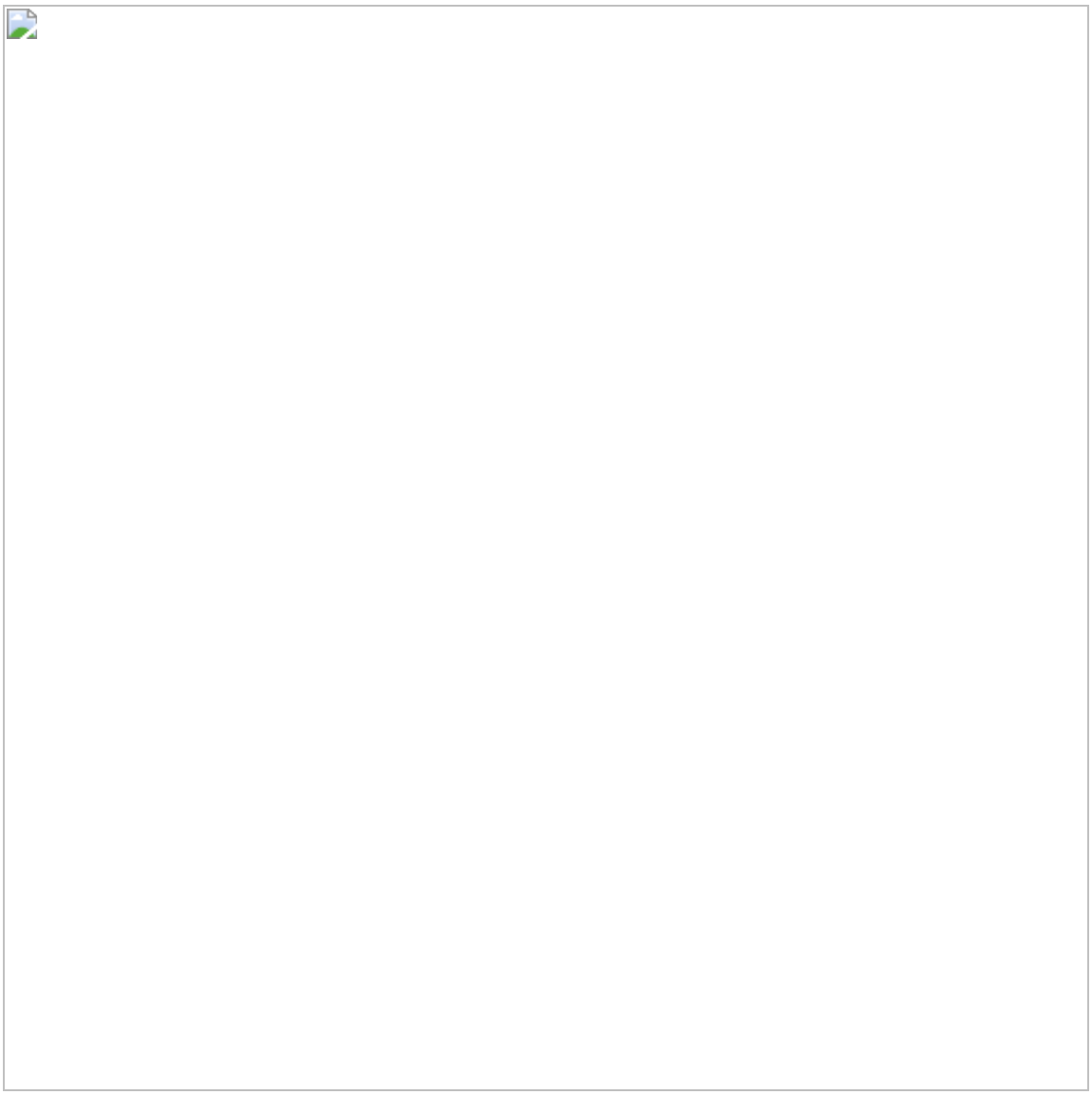
SQLite词法分析过程

词法分析入口

如前面分析，SQLite是前端+虚拟机+后端形式的架构，拿到SQL语句后，需要先在前端编译，得到虚拟机代码指令。编译过程开始的首要任务即是词法分析。SQLite的词法分析比较简单，这块代码是作者手工编写的，而不是使用像Lex的词法分析生成器工具自动生成的。要阅读其源码，不得不提到几个重要函数：

```
int sqlite3_prepare(xxxx);
int sqlite3_prepare_v2(xxxx);
int sqlite3_prepare_v3(xxxx);
```

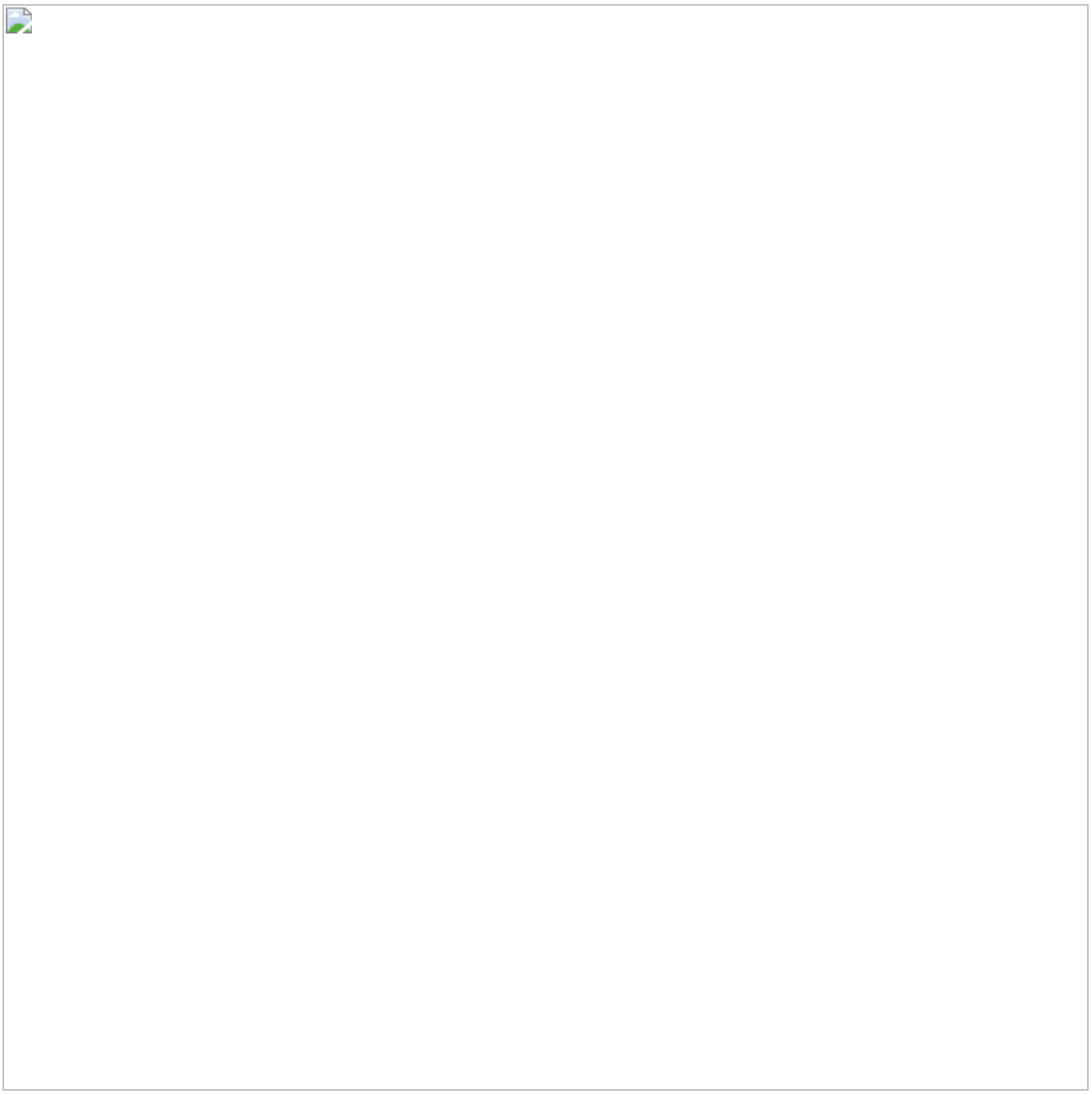
这几个接口均接受输入一组SQL语句，并将编译后的结果记录到sqlite3_stmt结构中。
经过几个核心函数调用后，调用到编译的核心函数sqlite3RunParser：



SQL词法分析-prepare

词法分析的层次

sqlite3RunParser函数是个非常重要的函数，其实现了对SQL语句的编译，即词法、语法分析的全过程。根据前面文章介绍，词法分析器从输入的字符串自左向右逐个将输入的字串流分割成一个个单词符号，并输入给语法分析器进行语法分析。显然，sqlite3RunParser的执行过程也应该是这样的。



Parse过程

sqlite3RunParser函数实现：

```
while( 1 ){
      ...
      //词法分析得到token
      n = sqlite3GetToken((u8*)zSql, &tokenType);
      ...
      //将一个个token输入到语法分析器，驱动语法分析器进行语法分析
      sqlite3Parser(pEngine, tokenType, pParse->sLastToken, pParse);
      ...
    }
  }
```

SQLite词法分析源码

词法分析函数的输入输出

从源码注释上看，词法分析器函数sqlite3GetToken接受输入一个SQL语句，不断的调用该函数，逐步的返回token距SQL语句起始点的长度，并且每次调用将得到的token类型保存到第二个参数tokenType中。

```
** Return the length (in bytes) of the token that begins at z[0].
** Store the token type in *tokenType before returning.
*/
int sqlite3GetToken(const unsigned char *z, int *tokenType)
{
  int i, c;
  switch( aiClass[*z] ){  /* Switch on the character-class of the first byte
                          ** of the token. See the comment on the CC_ defines
                          ** above. */
      //空格
    case CC_SPACE: {
     ...
      for(i=1; sqlite3Isspace(z[i]); i++){}
      *tokenType = TK_SPACE;
      return i;
    }
  //左括号
    case CC_LP: {
      *tokenType = TK_LP;
      return 1;
    }
  //右括号
    case CC_RP: {
      *tokenType = TK_RP;
      return 1;
    }
    ...
  //字母或下划线
    case CC_KYWD: {
      for(i=1; aiClass[z[i]]<=CC_KYWD; i++){}
      if( IdChar(z[i]) ){
        /* This token started out using characters that can appear in keywords,
        ** but z[i] is a character not allowed within keywords, so this must
        ** be an identifier instead */
        i++;
        break;
      }
      *tokenType = TK_ID;
      return keywordCode((char*)z, i, tokenType);
    }
...
  }
  while( IdChar(z[i]) ){ i++; }
  *tokenType = TK_ID;
  return i;
}
```

上面简化了的词法分析函数，代码比较简单，对于输入的字符串流逐个字符进行分析，字符被分为若干大类，通过查aiClass表，非常快速高效的得到当前字符类型。代码逻辑很清晰，比如： case CC_SPACE: 检测到空格输入，把相邻的空格全部过滤掉后，返回一个**空格的类**

型TK_SPACE(TK_SPACE是怎么得来的，后面介绍，go ahead);case CC_KYWD: 检测到字母或下划线，则继续读入，直到读到下一个特征字符后，将识别到的一个词组经keywordCode函数处理后返回。

Token识别知识铺垫

如上述对词法分析函数的解析，它从字符串流不断的提取出关键字、特征字、运算符等，转换成了一个叫做tokenType的数后返回，这个数字背后的逻辑基础是什么？单词符号和数字之间的对应关系是怎样的呢？

首先，可以观察到，sqlite3GetToken函数返回的tokenType取值是一组TK_开头的宏，这组宏定义在parse.h头文件中，意识到这点特征是非常重要的一步。而且TK_后接的字符串又正好是SQL语句的关键字，是否有想到些什么？

```
parse.h中定义的tokenType值
#define TK_SEMI                            1
#define TK_EXPLAIN                         2
#define TK_QUERY                           3
#define TK_PLAN                            4
#define TK_BEGIN                           5
#define TK_TRANSACTION                     6
#define TK_DEFERRED                        7
...
```

要详细解释这些现象背后的缘由，请关注后面语法分析章节。这里大致说明一下，SQLite使用的语法分析器生成器(lemon)会根据语法规则文件给所有的终结符、非终结符分配唯一的数字特征ID，在语法分析的过程中使用这些数字表达一些规则。parse.h即为其产物，它记录了SQL语句中的终结符的数字ID，而它的另一核心产物则是语法分析器parse.c,也就是说，语法分析器期望从词法分析器得到一组数组特征ID作为输入而非单词。因此词法分析器的核心任务除了分词外，还需要将分好的词转成语法分析器能认识的ID。

Token的Type转换方法

对于特征符号的 type转换直接代码写死，非常容易。对于词组的tokenType怎么获取，需要依赖核心函数keywordCode。下面分析该函数的源码实现：

```
/* Check to see if z[0..n-1] is a keyword. If it is, write the
** parser symbol code for that keyword into *pType.  Always
** return the integer n (the length of the token). */
static int keywordCode(const char *z, int n, int *pType){
  int i, j;
  const char *zKW;
  if( n>=2 ){
    i = ((charMap(z[0])*4) ^ (charMap(z[n-1])*3) ^ n) % 127;
    for(i=((int)aKWHash[i])-1; i>=0; i=((int)aKWNext[i])-1){
      if( aKWLen[i]!=n ) continue;
      j = 0;
      zKW = &zKWText[aKWOffset[i]];
#ifdef SQLITE_ASCII
      while( j<n && (z[j]&~0x20)==zKW[j] ){ j++; }
#endif
#ifdef SQLITE_EBCDIC
      while( j<n && toupper(z[j])==zKW[j] ){ j++; }
#endif
      if( j<n ) continue;
      ...
      *pType = aKWCode[i];
      break;
    }
  }
  return n;
}
```

什么鬼？(charMap(z[0])*4) ^ (charMap(z[n-1])*3) ^ n) % 127；x&^K%&$%*@#....?完全无法理解！！！这是什么原理？
冷静一下，看下源文件注释：

> /***** This file contains automatically generated code ******
> ** 这个文件中包含了自动生成的代码
> ** The code in this file has been automatically generated by
> ** 这段代码是通过mkkeywordhash.c自动生成的
> ** sqlite/tool/mkkeywordhash.c

mkkeywordhash.c黑幕

编译调试

这个文件直接编译可以得到完整的可执行程序，使用命令：

> gcc mkkeywordhash.c -o mkkeywordhash

从Main函数开始阅读，下面分析其实现。

1. 初步压缩，删除不支持的关键字

```
/* Remove entries from the list of keywords that have mask==0 */
  for(i=j=0; i<nKeyword; i++){
    if( aKeywordTable[i].mask==0 ) continue;
    if( j<i ){
      aKeywordTable[j] = aKeywordTable[i];
    }
    j++;
  }
  nKeyword = j;
```

该段函数实现对关键字表的初步压缩，对于mask==0的数组字段进行删除，之所以有这样的操作，是因为SQLite支持编译时裁剪优化，当有些SQL功能不需要支持时，没必要保留在关键字查找表中，影响表的空间大小从而影响到内存和执行效率。该部分代码执行完毕后，将得到一个长度为nKeyword 的数组aKeywordTable，这部分包含的key即当前编译的SQL支持的命令范围。

2. 对明确支持的关键字初始化len、hash、id

```
 /* Fill in the lengths of strings and hashes for all entries. */
  for(i=0; i<nKeyword; i++){
    Keyword *p = &aKeywordTable[i];
    p->len = (int)strlen(p->zName);
    assert( p->len<sizeof(p->zOrigName) );
    memcpy(p->zOrigName, p->zName, p->len+1);
    totalLen += p->len;
    p->hash = (charMap(p->zName[0])*4) ^
              (charMap(p->zName[p->len-1])*3) ^ (p->len*1);
    p->id = i+1;
  }
```

aKeywordTable数组中的每一项代表SQL的一个关键字，关键字的名字记录在zName字段中，作者使用该哈希函数为关键字进行哈希(**算法：关键字的第一个字符*4)^(关键字的最后一个字符*3)^(关键字名字的长度)**)：

> (charMap(p->zName[0])*4) ^(charMap(p->zName[p->len-1])*3) ^ (p->len*1)
> 从aKeywordTable中的关键字看到，首尾字符已经可以获得很少的哈希冲突了

同时，在代码里有一句断言需要提一下： assert(p->len<sizeof(p->zOrigName)); zOrigName和zName之间是有什么关联？这里暂时看不出来，其实zName存着的是关键字字串压缩后的结果，而zOrigName存着的是原始的key串，下面会看到压缩的过程。

3. 找出某些关键字是另外关键字的一部分的情况，为再次压缩准备

```
/* Sort the table from shortest to longest keyword */
  qsort(aKeywordTable, nKeyword, sizeof(aKeywordTable[0]), keywordCompare1);

  /* Look for short keywords embedded in longer keywords */
  for(i=nKeyword-2; i>=0; i--){
    Keyword *p = &aKeywordTable[i];
    //从最长的key串开始（数组下标nKeyword-1），依次检查该子串是否完全包含
    // Keyword  p,显然pOther 必须要比p长，所以j>i。
    for(j=nKeyword-1; j>i && p->substrId==0; j--){
      Keyword *pOther = &aKeywordTable[j];
      if( pOther->substrId ) continue;
      if( pOther->len<=p->len ) continue;
      for(k=0; k<=pOther->len-p->len; k++){
        //pOther的name长度大于p，要检查p是不是pOther完全子串，只需要从
        //pOther起始点到pOther->len-p->len(==k)比较是否相等即可
        if( memcmp(p->zName, &pOther->zName[k], p->len)==0 ){
          p->substrId = pOther->id;
          p->substrOffset = k;
          break;
        }
      }
    }
  }
```

首先，将key数组排序，排序算法非常简单，就是按长度排序，长度相同的按字符串比较大小排序。keywordCompare1函数不再赘述。
然后，从字符串最长的倒叙查找比其短的的字符串，是否是它的子串。比如SET是OFFSET子串，则将SET的substrId 设置为OFFSET的id，并记录SET在OFFSET中的偏移量substrOffset=3；这样在后面的字符串压缩中，SET和OFFSET就可以完全合并了。合并算法后面介绍，go ahead！

3. 找出关键字名字的最长后缀，为更进一步的压缩准备

```
  /* Compute the longestSuffix value for every word */
  for(i=0; i<nKeyword; i++){
    Keyword *p = &aKeywordTable[i];
    //已经是别的字串的子串了，后面会被合并掉，不需要寻找最长后缀
    if( p->substrId ) continue;
    for(j=0; j<nKeyword; j++){
      Keyword *pOther;
      if( j==i ) continue;
      pOther = &aKeywordTable[j];
       //已经是别的字串的子串了，也不需要用于比对最长后缀
      if( pOther->substrId ) continue;
      for(k=p->longestSuffix+1; k<p->len && k<pOther->len; k++){
        if( memcmp(&p->zName[p->len-k], pOther->zName, k)==0 ){
          p->longestSuffix = k;
        }
      }
    }
  }
/* Sort the table into reverse order by length */
  qsort(aKeywordTable, nKeyword, sizeof(aKeywordTable[0]), keywordCompare2);
```

最长后缀的意图，是找出有相交但不完全包含的两个字符串的集和中，相交最长值为最长后缀，比如：abcde 和defgh、cdefg、bcdefgei三个字串都是相交但又非完全包含，和defgh相交的长度为2,即de;和cdefg相交的长度为3,即cde;和bcdefgei相交的长度为4,即bcde。因此，abcde 的最长后缀值为4。
达到这样的意图，只需要对每个关键字，都要遍历其他所有的关键字，寻找该关键字的最长后缀值。找完后排序。

4. 字符串压缩核心算法

将所有的关键字紧密排列到一起，并计算每个关键字在整个字符串中的偏移

```
//运行到这里，aKeywordTable是按照最长后缀排序过的数组
  /* Fill in the offset for all entries */
  nChar = 0;
  for(i=0; i<nKeyword; i++){
    Keyword *p = &aKeywordTable[i];
    //已经计算好了offset或是其他字串的子串，暂不处理
    if( p->offset>0 || p->substrId ) continue;
    p->offset = nChar;
    nChar += p->len;
    //p为当前有最长后缀的子串，子串长度为p->len，寻找和后缀重叠但不全包含的字
    //符串进行拼接，因此比对从p->len-1开始，不断寻找后缀匹配的串，找不到则k--,
    //因为aKeywordTable是排序的最长后缀串，因此该循环中一般一定能找到一个串
    for(k=p->len-1; k>=1; k--){
      //遍历比该串还短的子串，寻找最长后缀匹配
      for(j=i+1; j<nKeyword; j++){
        Keyword *pOther = &aKeywordTable[j];
        if( pOther->offset>0 || pOther->substrId ) continue;
        if( pOther->len<=k ) continue;
        //找打p串后缀和pOther前缀重合的串，进行合并
        if( memcmp(&p->zName[p->len-k], pOther->zName, k)==0 ){
          //other将合入p，下次循环找other的后缀重叠串，因此other赋值p
          p = pOther;
          //other串开始offset等于当前串尾向前减去重叠部分
          p->offset = nChar - k;
          //pOther合并后总串的长度改变
          nChar = p->offset + p->len;
          //合并压缩后，zName就只剩未重叠部分，重叠部分被压缩到前面的串中
          //压缩后，zName变短，长度等于原长度-重叠部分，重叠长度k记录到prefix
          p->zName += k;
          p->len -= k;
          p->prefix = k;
          j = i;
          k = p->len;
        }
      }
    }
  }
  //对于完全子串，其offset等于父串(完全包含它的串)在整个字串中的offset+该子串在父串中的offset
  for(i=0; i<nKeyword; i++){
    Keyword *p = &aKeywordTable[i];
    if( p->substrId ){
      p->offset = findById(p->substrId)->offset + p->substrOffset;
    }
  }
//按offset偏移从小到大排序
/* Sort the table by offset */
  qsort(aKeywordTable, nKeyword, sizeof(aKeywordTable[0]), keywordCompare3);
```

这点代码比较难理解，下面用事例描述这个过程，以帮助理解这段代码：

1. 假设有6个字符串，分别为“abc”、“bcd”、“cde”、“def”、“efg”、“fgh”，这6个字符串之间两两之间不互相完全包含。要把这些字符串存起来，需要占用abcbcdcdedefefgfgh共18个字节，这是非常浪费空间的。因为我们发现他们之间两两重叠的部分，实际可以压缩起来的。
2. 将6个字符串前后重合部分合并起来，字符串变成了tmp = "abcdefgh"，这时我们只需要知道abc是tmp中开始0偏移3的串，bcd是tmp中开始1偏移3的串，cde是tmp中开始2偏移3的串...依次类推。
3. 对于完全包含的子串，比如有字符串bc,要用tmp描述，只需要指导bc完全包含在abc或者bcd之1中，假定选择abc，那bc是abc的子串，其在abc中偏移1。此时，bc在tmp中的描述就变成了abc在tmp中的偏移0加bc在abc中的偏移1，即在tmp中偏移1，长度2。

上述代码的核心逻辑就是按照这样的原理进行压缩的。

经过这种压缩算法压缩后，aKeywordTable数组中的keywork的zName就变小了，而aKeywordTable又是一个按照offset排序过得数组，此时只要for循环，把非完全字串（substrId==0）的zName拼接在一起就得到了压缩后的总字符串zKWText:

```
/* zKWText[] encodes 834 bytes of keyword text in 554 bytes */
/*   REINDEXEDESCAPEACHECKEYBEFOREIGNOREGEXPLAINSTEADDATABASELECT       */
/*   ABLEFTHENDEFERRABLELSEXCEPTRANSACTIONATURALTERAISEXCLUSIVE         */
/*   XISTSAVEPOINTERSECTRIGGEREFERENCESCONSTRAINTOFFSETEMPORARY         */
/*   UNIQUERYWITHOUTERELEASEATTACHAVINGROUPDATEBEGINNERECURSIVE         */
/*   BETWEENOTNULLIKECASCADELETECASECOLLATECREATECURRENT_DATEDETACH     */
/*   IMMEDIATEJOINSERTMATCHPLANALYZEPRAGMABORTVALUESVIRTUALIMITWHEN     */
/*   WHERENAMEAFTEREPLACEANDEFAULTAUTOINCREMENTCASTCOLUMNCOMMIT         */
/*   CONFLICTCROSSCURRENT_TIMESTAMPRIMARYDEFERREDISTINCTDROPFAIL        */
/*   FROMFULLGLOBYIFISNULLORDERESTRICTRIGHTROLLBACKROWUNIONUSING        */
/*   VACUUMVIEWINITIALLY                                                */
```

5. 计算最优哈希表大小及哈希值

```
/* Figure out how big to make the hash table in order to minimize the
** number of collisions */
bestSize = nKeyword;
bestCount = nKeyword*nKeyword;
for(i=nKeyword/2; i<=2*nKeyword; i++){
  for(j=0; j<i; j++) aKWHash[j] = 0;
  for(j=0; j<nKeyword; j++){
    h = aKeywordTable[j].hash % i;
    aKWHash[h] *= 2;
    aKWHash[h]++;
  }
  for(j=count=0; j<i; j++) count += aKWHash[j];
  if( count<bestCount ){
    bestCount = count;
    bestSize = i;
  }
}

/* Compute the hash */
for(i=0; i<bestSize; i++) aKWHash[i] = 0;
for(i=0; i<nKeyword; i++){
  h = aKeywordTable[i].hash % bestSize;
  aKeywordTable[i].iNext = aKWHash[h];
  aKWHash[h] = i+1;
}
```

分词器得到一个字符串key后，要查字符串对应的tokenType，直接遍历整个表查询并不够优化。为了提高查询效率，作者使用哈希索引实现常数级的复杂度，以提高查询效率。那SQL支持的关键字数目是用户可裁剪的，如何取哈希表的大小，以求得比较小的哈希碰撞，从而获取较高的综合查询效率？

作者将哈希表的范围限定到关键字数量的一半到其平方值之间，以这个区间内的值取模，哪个数值会更优？哈希碰撞会更小？设置一个“惩罚因子”——此处是取值2（也可以是3,4,5，实际上任意除1之外的正整数都可以），每次取模后的结果都要乘以这个“惩罚因子”，重复的次数与2的幂次方成正比，增长速度是非常快的。对所有模值求和，意味着重复项越多，最终求得的和越大。**怎么理解？**比如当哈希表大小取值为n1时，如果完全没有发生碰撞，aKWHash[n]初始值是0，那么哈希表中所有的值求和count += aKWHash[j];就等于n1；而当有一个元素发生1次碰撞后，求和应该等于n1+2，一个元素发生2次碰撞后求和等于n1+6，三次n1+14....按2的指数次方增长。因此，和的大小很大程度反映了碰撞的激烈程度。随着哈希表的大小i的增加，最终计算得到的哈希值会变得一样的“稀疏”，也就意味着求和的结果都趋向于i,当i使得和最小，则认为其实最优的哈希模值。

为aKeywordTable中留下来的每一个key计算哈希值，如果没有哈希冲突，意味着每个哈希值都对应唯一的aKWHash[h]，由于aKWHash[]数组默认值是0，因此每个关键字的iNext值都是0。如果存在哈希冲突，则aKWHash[h]中是非零值。该非零值减1即表示与当前关键字产生哈希冲突的关键字的索引值。

6. 代码自动生成

有了上面这些分析，代码自动生成做的工作就比较清晰了，它生成了让我们看不懂的函数int keywordCode(const char *z, int n, int *pType)及其计算使用的那些数组、哈希表等。即zKWText为压缩后的字符串；aKWHash为keyword的哈希值表；aKWLen为第i个key的长度；aKWOffset为第i个key在zKWText的offset值，aKWCode是第i个key的tokenType;在回过头来看keywordCode实现，意图就十分清晰了。

```
static int keywordCode(const char *z, int n, int *pType){
  int i, j;
  const char *zKW;
  if( n>=2 ){
    //根据词法分析到的关键字串，计算其哈希值
    i = ((charMap(z[0])*4) ^ (charMap(z[n-1])*3) ^ n) % 127;
   /*i=((int)aKWHash[i])-1，即i值应该是keywork的索引或和keywork索引发生哈希碰撞的keyword索引。*/
   //1.哈希无冲突时，其值就是keywork的索引;
   //2.当存在哈希冲突时，其值是和keyword冲突的索引值+1
   //有没有哈希碰撞，可以检测aKWNext[i]是不是等于0，等于0无冲突
  //在此基础上检查len和字符串（全转大写比较）是否相等
    for(i=((int)aKWHash[i])-1; i>=0; i=((int)aKWNext[i])-1){
      if( aKWLen[i]!=n ) continue;
      j = 0;
      zKW = &zKWText[aKWOffset[i]];
#ifdef SQLITE_ASCII
      while( j<n && (z[j]&~0x20)==zKW[j] ){ j++; }
#endif
#ifdef SQLITE_EBCDIC
      while( j<n && toupper(z[j])==zKW[j] ){ j++; }
#endif
      if( j<n ) continue;
     ...
    //条件验证通过后，type则可直接通过索引i拿到并赋值
    *pType = aKWCode[i];
      break;
    }
  }
  return n;
```

至此，SQLite的词法分析部分的核心代码分析完毕，可以看到，作者为了提高执行效率和降低内存使用率方面，做了非常深入的优化。

最后编辑于 ：2019-07-14 14:53:40

Sqlite3源码剖析

举报文章

小礼物走一走，来简书关注我

赞赏支持

还没有人赞赏，支持一下


includes 探寻本质、追根溯源！
总资产8 共写了22193字 获得24个赞 共10个粉丝

人面猴
序言：七十年代末，一起剥皮案震惊了整个滨河市，随后出现的几起案子，更是在滨河造成了极大的恐慌，老刑警刘岩，带你破解...

 沈念sama 阅读 191823 评论 5 赞 459
死咒
序言：滨河连续发生了三起死亡事件，死亡现场离奇诡异，居然都是意外死亡，警方通过查阅死者的电脑和手机，发现死者居然都...

 沈念sama 阅读 80801 评论 2 赞 368
救了他两次的神仙让他今天三更去死
文/潘晓璐 我一进店门，熙熙楼的掌柜王于贵愁眉苦脸地迎上来，“玉大人，你说我怎么就摊上这事。”“怎么了？”我有些...

 开封第一讲书人 阅读 139194 评论 0 赞 314
道士缉凶录：失踪的卖姜人
文/不坏的土叔 我叫张陵，是天一观的道长。 经常有香客问我，道长，这世上最难降的妖魔是什么？ 我笑而不...

 开封第一讲书人 阅读 51571 评论 1 赞 262
港岛之恋（遗憾婚礼）
正文 为了忘掉前任，我火速办了婚礼，结果婚礼上，老公的妹妹穿的比我还像新娘。我一直安慰自己，他们只是感情好，可当我...

 茶点故事 阅读 60460 评论 4 赞 352
恶毒庶女顶嫁案：这布局不是一般人想出来的
文/花漫 我一把揭开白布。 她就那样静静地躺着，像睡着了一般。 火红的嫁衣衬着肌肤如雪。梳的纹丝不乱的头发上，一...

 开封第一讲书人 阅读 45667 评论 1 赞 268
城市分裂传说
那天，我揣着相机与录音，去河边找鬼。 笑死，一个胖子当着我的面吹牛，可吹牛的内容都是我干的。 我是一名探鬼主播，决...

 沈念sama 阅读 36208 评论 3 赞 379

双鸳鸯连环套：你想象不到人心有多黑
文/苍兰香墨 我猛地睁开眼，长吁一口气：“原来是场噩梦啊……”“哼！你这毒妇竟也来了？”一声冷哼从身侧响起，我...

 开封第一讲书人 阅读 34908 评论 0 赞 252
万荣杀人案实录
序言：老挝万荣一对情侣失踪，失踪者是张志新（化名）和其女友刘颖，没想到半个月后，有当地人在树林里发现了一具尸体，经...

 沈念sama 阅读 39097 评论 1 赞 284
护林员之死
正文 独居荒郊野岭守林人离奇死亡，尸身上长有42处带血的脓包…… 初始之章·张勋 以下内容为张勋视角 年9月15日...

 茶点故事 阅读 34277 评论 2 赞 304
白月光启示录
正文 我和宋清朗相恋三年，在试婚纱的时候发现自己被绿了。 大学时的朋友给我发了我未婚夫和他白月光在一起吃饭的照片。...

 茶点故事 阅读 36048 评论 1 赞 322
活死人
序言：一个原本活蹦乱跳的男人离奇死亡，死状恐怖，灵堂内的尸体忽然破棺而出，到底是诈尸还是另有隐情，我是刑警宁泽，带...

 沈念sama 阅读 31934 评论 3 赞 309
日本核电站爆炸内幕
正文 年R本政府宣布，位于F岛的核电站，受9级特大地震影响，放射性物质发生泄漏。R本人自食恶果不足惜，却给世界环境...

 茶点故事 阅读 37294 评论 3 赞 296
男人毒药：我在死后第九天来索命
文/蒙蒙 一、第九天 我趴在偏房一处隐蔽的房顶上张望。 院中可真热闹，春花似锦、人声如沸。这庄子的主人今日做“春日...

 开封第一讲书人 阅读 28694 评论 0 赞 17
一桩弑父案，背后竟有这般阴谋
文/苍兰香墨 我抬头看了看天上的太阳。三九已至，却和暖如春，着一层夹袄步出监牢的瞬间，已是汗流浃背。 一阵脚步声响...

 开封第一讲书人 阅读 29958 评论 1 赞 249
情欲美人皮
我被黑心中介骗来泰国打工， 没想到刚下飞机就差点儿被人妖公主榨干…… 1. 我叫王不留，地道东北人。 一个月前我还...

 沈念sama 阅读 41142 评论 2 赞 337
代替公主和亲
正文 我出身青楼，却偏偏与公主长得像，于是被迫代替她去往敌国和亲。 传闻我的和亲对象是个残疾皇子，可洞房花烛夜当晚...

 茶点故事 阅读 40367 评论 2 赞 335


includes
总资产 8

SQLite之SQL解析-词法分析-6
阅读：419

SQLite之SQL解析-词法分析-6
阅读：518

end_time: 2024-09-08 20:59:45 +0800 Completed in 228.592447ms