Linux 环境中使用 Flex、Bison 进行 SQL 语法分析

孙兆玉　朱鸿宇　黄宇光

摘　要　本文以查询语句分析为例，从问题描述、语法范式构建、词法分析、语法分析和应用接口设计等方面详细阐述了一种 SQL 语句解析的通用策略，并介绍了与之相关的冲突消解、可重入策略和错误处理三个方面的技术。

关键词　BNF 范式，词法分析，语法分析，冲突消解，可重入

一、 引言

SQL 是面向关系数据库操作的一门成熟的高级语言，它是数据库管理系统强大的管理操作接口。每个数据库管理系统（DBMS）都包含有自己的 SQL 语法、语义分析模块，但通常都不向外提供具体的分析功能函数以及数据结构。然而实际应用中，却有很多地方需要进行 SQL 语法分析，如：不同标准 SQL 互相翻译、面向 SQL 编辑器的报错提示、基于 SQL 语句的任务分析统计、SQL 操作图形化等。

本文介绍了一种采用 Flex、Bison 分析工具进行 SQL 语法分析的通用策略，解决了 SQL 分析过程中的冲突消解、可重入策略和错误处理三大问题，实现了对 Oracle 的 SQL 语句进行语法分析并构建完整的 SQL 语法树的目标，为进一步的处理提供了方便。

二、 Flex、Bison 工具

Flex、Bison 是上个世纪 80 年代左右出现的 Unix 环境下分析编译工具 lex、yacc 的 GNU 版本。当时贝尔实验室的 Stephen C. Johnson 是 yacc 最主要的作者，他采纳了很多先进算法思想，将 yacc 打造成为一个成功的语法分析编译工具。其后，M. E. Lesk and E. Schmidt 成功推出了词法扫描工具 lex，它与 yacc 是天生兄弟，lex 架构上采用了 yacc 模式，字符串扫描采用了 A. V. Aho 的算法机制，因此，Schmidt 常谦称 Johnson 和 Aho 才是 lex 的元老作者。后来，GNU 社区在 lex、yacc 基础上，推出了 Flex、Bison，然后随着不断的技术改进又推出了许多版本，每个版本都有自己的特色，主要针对以前版本功能缺陷的一些修改等。本文推荐使用 Flex2.5.31 版就具有生成多线程词法扫描器能力。

Flex 主要功能是根据用户定制的构词规则，生成面向字符流自动扫描分词的程序。与用户自己动手编写扫描分词程序相比，它的速度和准确度一般都要高。并且采用它，用户可以减少大量的编码。

Bison 主要功能是根据用户定义的 LALR(1)上下文无关文法系统描述规则，生成自动进行语法分析的程序。它的通用性使其能在各种编译器、解释工具的设计上发挥优势。并且 Bison 兼容 yacc 的语法规则。

三、SQL 语法分析的特点与目标

从生成语言的角度看，SQL 语言系统的设计遵循标准文法规则，具有完善的词法和语法体系。通常一个 SQL 语法分析系统应该包括语言实例、语法规则、定制规则、分析器、分析结果（语法语义树表达形式）几部分组成如图 1 所示。SQL 语法规则和定制规则是构造分析器的直接依据，语言实例和结果分别是分析器的输入和输出。

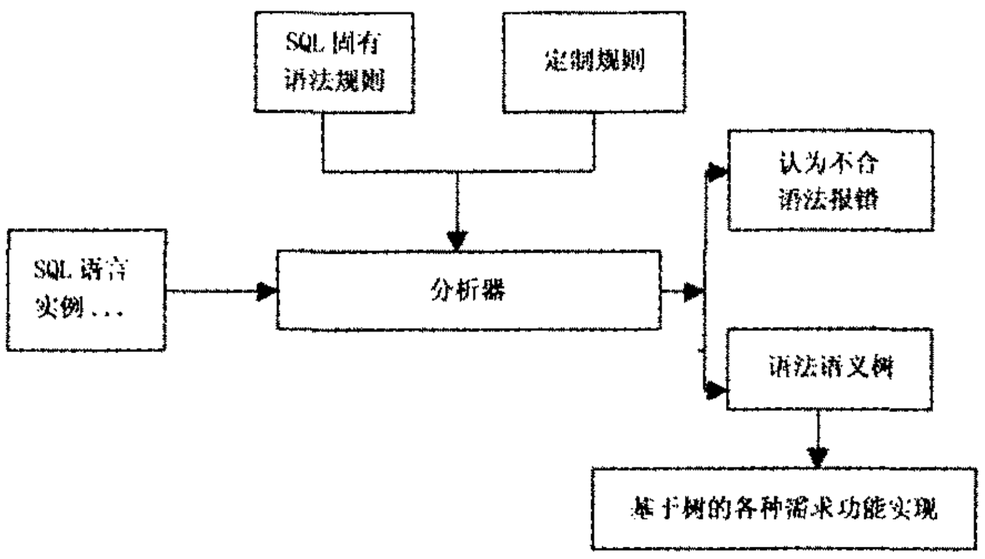

图 1　SQL 语言分析系统

SQL 固有的语法规则采用流图模式层层展开，是用来描述其中顺序、蕴涵等各种结构的，如图 2 所示。这种图示表达与 BNF 范式表达有内在的一致性，它们都遵循自上而下层层展开的表示框架、采用递归嵌套方式、逻辑中隐含着有穷自动机，并且由语法流图到 BNF 范式的转换直观易行——流图中的每个通路对应 BNF 范式中对表达式左侧的非终结符一个展开方案。例如，图 2 对应 BNF 表达式为 expression_list : : = ({expr},) + | “(“({expr},) +”)”。

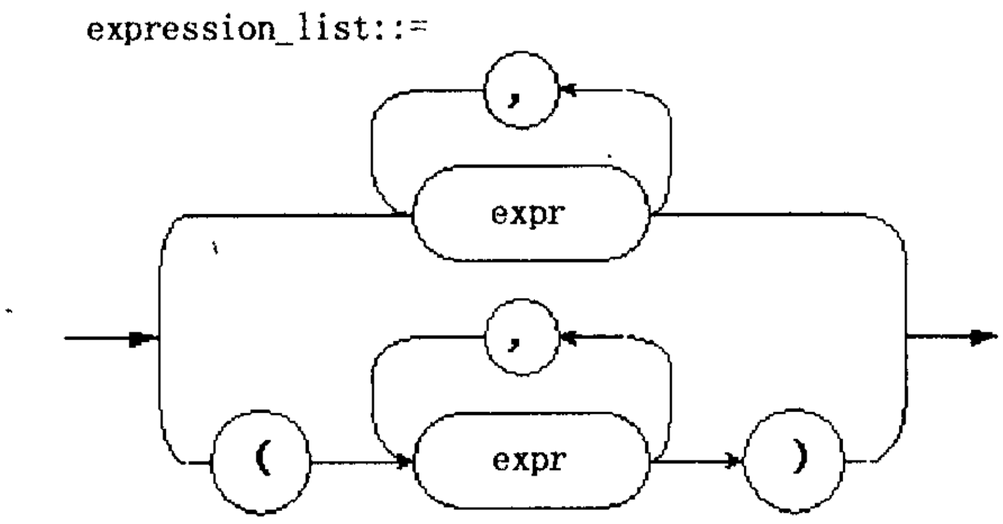

图 2　SQL 语法流图

BNF 范式是描述语法逻辑的强大工具，Bison 这些工具也都识别 BNF 范式。但是 expression_list : : = ({expr},) + | “(“({expr},) + ”)”（称之为幂式）这样的表达形式需要展开成左递归文法形式 expression_list : : = expr_list | ‘(‘ expr_list’)’ expr_list : : = expr | expr_list ‘,’ expr(称之为左递归式）才能被 Bison 所接受，因此进行 SQL 语法分析的预备工作就是将语法流图转换成为分析工具接受的左递归式。

四、SQL 词法分析

从本节开始，所有的词法和语法规则都按照 Oracle 10g 的 SQL 标准进行设计，以后不再说明。词法分析是语法分析的基础，它把 SQL 语句分解聚合成词块 token 流，提供给语法分析器进一步解析。词法分析器构建借助于工具 Flex 将正则式描述的构词规则转换成词法分析器如图 3 所示。

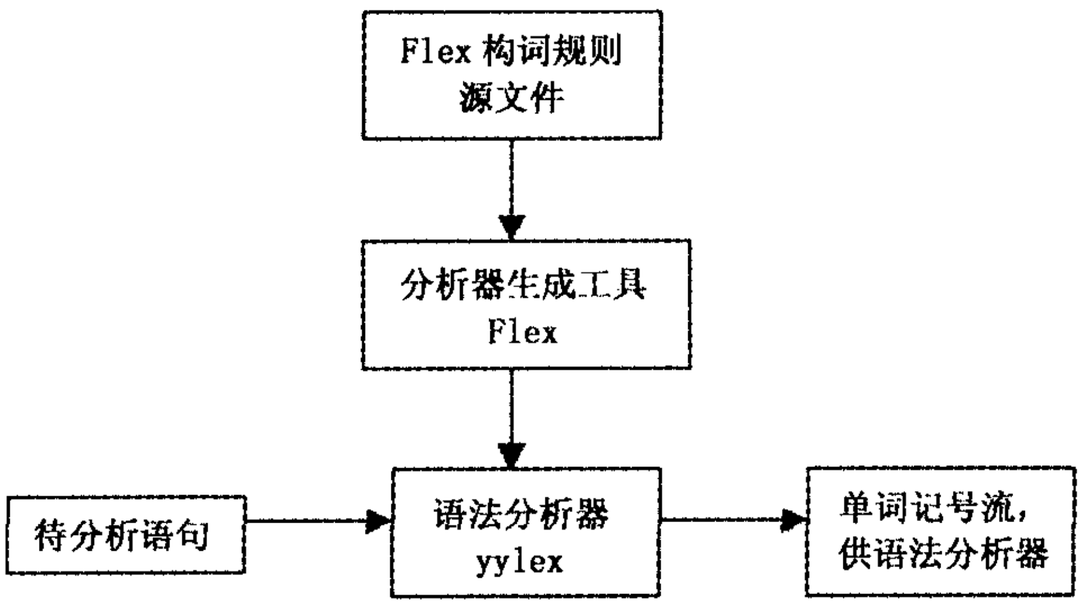

图 3　词法分析器生成流程

词法分析器设计的主要工作是设计词块 token 和其属性值，以及确定构词规则表达（正则式描述）。SQL 语句的构词种类可以分为保留字、语法符号、用户定义的标志和常量（数值或者字符串）。

1. 保留字

保留字是 SQL 语法中承担特殊功能有特别意义的词，如：SELECT WHERE 等。它们分别当作一个词 token 向语法分析传递，本身的记号特征就足够语法分析的信息，所以不需要设计属性值。

2. 符号类词

语言设计的符号词有比较类，运算类，以及断句限定范围类。比较类符号（“=”“<>”“<”“>”“<=”“>=”）在语句构成中可以互相替代（位置），发挥相同比较功能，所以，把比较类符号归结为一个词记号用 COMPARISON 名称来表示，同时把比较的符号本身作为记号的属性传递给上层分析；运算符号（“+”“-”“*”“/”）由于“+-”表示两种运算意义，并且“+-*/”有运算级别的区别，所以见于数量也比较少，可以分别当作一个词记号；断句限定范围的标点符号，则功能互相无法替代，所以各自为词。

3. 用户定义的标志、常量

标志符号是由字母开头的字母、数字串。在 SQL 语句中，它能表示授权标识符、表 table 标识符、列名、相关名、模块名、游标名、过程名、参数名与嵌入式变量名；常量有字符串常量、整数、实数。标志符和常量都用正则表达式来描述。

设计好的词 token 将会被词法分析器和语法分析器共同识别，通常这些 token 定义在 Bison 源程序中，然后 Bison 生成含有这些 token 定义的头文件提供给 flex 程序。Flex 将包含上述构词规则的源文件转换成 SQL 词法分析 C 程序。

Flex 源文件组织使用两次%%分割成为三部分，第一部分为头文件、变量声明等；第二部分为词法规则定义部分；第三部分为用户定义函数，可包括 main 函数。

```
%{
#include <string.h>
#include "queryopt.h"
/* 包含 bison 工具生成的词记号宏定义头文件 */
#include "speed.tab.h"
int lineno = 1;
void yyerror(char *s);
int yywrap();
%}
/* 与 bison 生成的语法分析器桥接 */
%option bison-bridge
/* 生成多线程可重入词法分析器 */
%option reentrant
%%      //节分隔符
//保留字
ALL     {return ALL;}
AND     {return AND;}
......
//符号类
"="|"<>"|"<"|">"     |"<="|">="
{
yylval.str = strdup(yytext);
return COMPARISON;
}
[-+*/:(),.;] {return yytext[0];}
......
//用户定义的标志、常量
[A-Za-z][A-Za-z0-9_]* {/*up the char*/
            int i;
```

```
                    yylval. str =strdup(yytext);
/ * 将所有字母转换为大写 * /
                        for(i =0; i <yyleng; i + +) {
if((yylval. str[i]>=´a´)
& & (yylval. str[i] < =´z´))
                        yylval. str[i] - =32;
                    }
                    return NAME;
                }
[0 -9] +    | [0 -9] +´. ´[0 -9] * | ´. ´[0 -9] *
{ yylval. str =strdup(yytext);    return INTNUM; }
.....
%%          //节分隔符
.....       //定义自己的函数
```

上述将“...”出现的地方类似补充完整后，就可以使用 Flex 将之转换为一个完整的 SQL 词法分析器。

五、SQL 语法分析

Bison 可以将用户描述的 SQL 语法规则转化成能进行 SQL 自动语法分析的程序工具（如图 4 所示）。如何分析 SQL 语法规则，构造 Bison 语法源文件是进行 SQL 语法分析的重点。

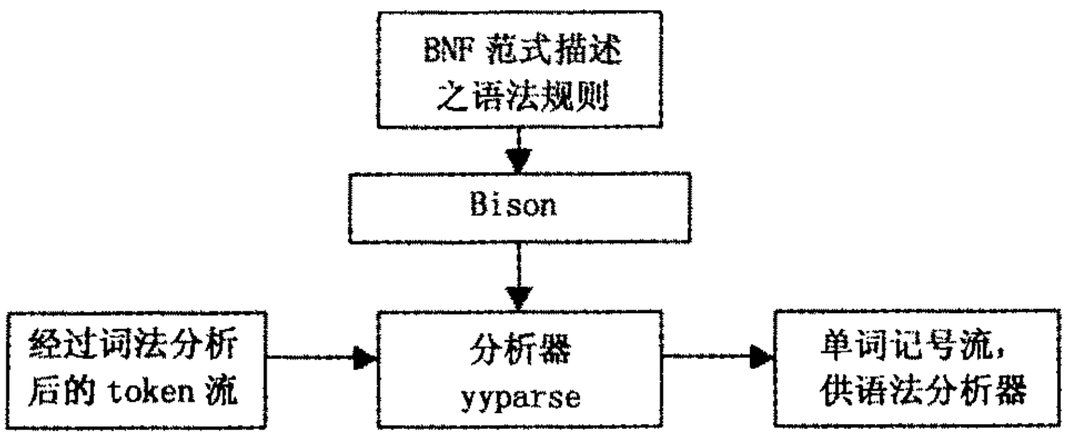

图 4　Bison 工作原理

下面以查询语句为例，遵照 SELECT...FROM...WHERE... 查询语句的语法构成和语法功能单位特点，按照图 5 划分分析模块。

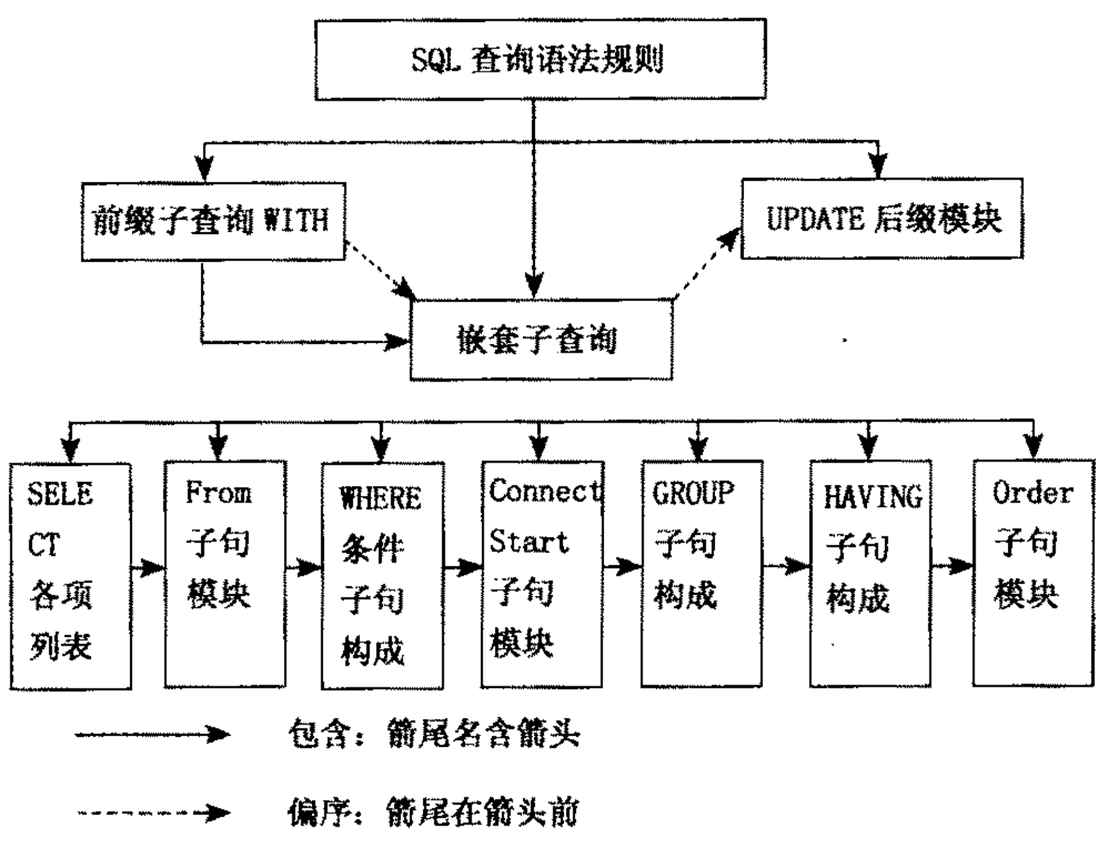

图 5　SQL 查询语法分析模块

除了图示的模块分割外，模块之间还存在着一些嵌套的关系，比如条件子句（where、having 等）中嵌套子查询。所有的模块关系都将体现在语法范式表达和语法树结构设计中。

在“SQL 语法分析的特点与目标”部分，已经介绍 Bison 接收定义的“左递归式”。这里就具体介绍下 Bison 源文件是如何组织这些规则的。

Bison 源文件的布局与 Flex 相似，采用%%将文件分割成三部分，第一部分包含头文件声明，结构定义，函数声明，宏声明等；第二部分是主要内容，它是由所有的语法规则表达式列表而成，开发人员在此修改规则式以及编码处理分析结果；第三部分是开发人员自定义函数处，可以是一些通用工具函数，也可以是对功能函数的封装，无具体限制。

第一部分

头文件声明和数据结构的定义和一般 C 程序相同，不过要使用%{、%}封闭这一部分（Bison 规则），例如：

```
%{
#include <string. h>
#include ˝queryopt. h˝
typedef struct{
SQLPATTERN * query_stru;
QueryTime * query_time;
TimeCol  * pat_tcol;
}MyExtraType;
#define YYLEX_PARAM scanner
#define YYPARSE_PARAM scanner
%}
```

其次，使用%union、%type、%token 声明分析结果的数据类型。

```
%union  {
char * u_str;
WITHSTRU * u_withsub;
SQLPATTERN * u_qs;
.....
};
%type  <u_str> atom col_ref sign expr...
%type  <u_qs>    oraselect subquery
.....
%token <u_str> NAME AMMSC INTNUM STRING APPROX-
NUM
%token SELECT UNIQUE DISTINCT...
.....
```

在下面第二部分中，每条规则左侧出现的被展开标记都得有自己对应的数据类型，从而明确保存分析结果的数据类型，%union 就是把所有使用到的数据类型、变量声明罗列在一起，然后供%type 和%token 来具体声明每个标记对应的变量类型。上述，atom_col 是个被展开的描述列名称的标记，被定义了变量 u_str，实际上是指定了一个类型为 char * 的变量；oraselect 被指定变量类型为 SQLPATTERN *，用于保存分析获得的模板；%token 声明是与 Flex 文件紧密相关的部分，它所声明的是词 to-

ken 标记（词法分析的结果表示），“词法分析”部分已经介绍，token 有赋值和非赋值两种类型，这里 NAME 就定义了可以被赋 char * 值，而 SELECT 就是简单记号，不被赋值。% token 定义的部分，通过 Bison 生成专门供 Flex 源文件使用的头文件 speed. tab. h（定义词法分析结果 token 标记）。

此外，该部分还包括 Bison 相关定制的宏声明 % pure_parser、% left 等。

第二部分

所有定义规则项的罗列，每一项都遵循如下格式：

```
expanded_item:
emember1 emember2 . . .
                {          . . .          }
            |   em1 em2
                {     . . . . .    }
            . . . . .
            ;
```

冒号：左侧为被展开的语法记号，右侧是用‘|’分开的多项展开方式，{. . .}是具体每项展开对应的用户编程内容，然后以分号结束。

示例查询语法最上层的展开，

```
oraselect:
        subquery for_update     {
            $1 ->sp_updateclause =$2;
            $$ =$1;
            QUERY_STRU =$$;      }
;
```

只有一种展开方式，在展开式右侧$1 表示第一个记号对应变量，$2 表示第二个，以此类推，然后$$表示左侧被展开记号对应的变量。

第三部分

开发者定义自己的函数，格式与 C 语言定义函数相同。

实际上，Bison 文件形式比较简单，规则描述部分的设计才是最重要的。这部分由于语法表达冲突的存在，往往需要大量表达式重写或者调整，所以建议在明确设计出完整的无冲突语法范式后再进行语法树构建等其他的用户编程。

分析结果的处理是嵌入在每一项规则描述设计中的，它的信息主要存储在内存中建立的一棵语法树中，语法树的结构特点是由模块划分和语法规则共同决定的。父结点是子结点们的上层抽象表示，子结点是父结点的顺序展开表达。树的结构对存储嵌套结构的管理实现也很方便，使用指针和标识值就可以确定子结点是嵌套子查询还是普通字段。语法树的生成是通过在语法分析过程中层层申请结点结构并上传指针，建立起父子结点关系，进而建立起一棵树的。这棵树并非规则树，每个结点的子结点数目也不相同。

1. 无冲突语法规则的调试

根据图 5 的基本框架和 Oracle 文档提供的语法流图，能编写出粗略的语法规则文件，但它必然包含了许多将被 LALR(1) 文法辨识为冲突的部分。

Bison 只能向前识别一个字符，如果下个字符相同但是却有多个可选后续状态（其中有一个要“归约”），文法就出现了冲突，bison 因此也无从得知该选择哪个范式进行“移进”或者“归约”。当接收一个字符后，既可以选择“移进”，又可以选择“归约”，就是“移进 归约”（shift/reduce）冲突，若都是“归约”，不过一个通过 A 式，一个通过 B 式，就是“归约 - 归约”（reduce/reduce）冲突。分析冲突出现的原因，主要是语言文法本身的特点，以及范式层次化设计的疏忽所致。针对前者，需要结合语言实际使用情况，在能接收全部实用语言的基础上对语言进行进一步的限制；后者则较简单，分析清楚逻辑层次后重新设计即可。

按照图 5 所示的“嵌套子查询”所包括的各个子模块来一一设计、测试，然后联合来测试。其中有很多可以归为 A: : = A atail 一类，像 select、from、group、order；另有 where、having 可以归为条件表达；Connect Start 子句属单句类，其句长有限。下面列举典型的运算表达式范式的分析和冲突消解，

```
expr:
        expr ´ + ´ expr
    |   expr ´ - ´ expr
    |   expr ´ * ´ expr
    |   expr ´ / ´ expr
    |     ´ + ´ expr
    |     ´ - ´ expr
```

这种表达会出现冲突，比如：3 + 2 * . . . 在扫描至 * 号时就遇到了“归约 - 移进”冲突，原因是运算级别表述不清晰，“+ -”既可以作双操作数运算符，也可以当作单操作数运算符。解决办法有两个，一是重写表达式；另外，bison 提供了一个更简单的途径，它通过使用% left 宏直接指定运算级别。

```
%left ´ + ´ ´ - ´
%left ´ * ´ ´ / ´
%nonassoc UMINUS
. . . . . .
|     ´ - ´ expr %prec UMINUS
. . . . . .
```

上述指定了‘-’ ‘*’ ‘/’ ‘+’ ‘-’从高到低的运算级别。

同理，可以用 % left OR、% left AND、% left NOT、% left COMPARISON（= <> < > < = > =）来解决逻辑判别表达式的冲突问题。

基本语法分析单元使用混乱，常导致“归约 - 归约”冲突产生，例如：

```
col_str: expr | normal | . . . ;
expr: NAME | expr ´ + ´ expr | . . . ;
normal: NAME | NAME ´ . ´ NAME | NAME ´ . ´ NAME | . . . ;
```

上面是在编写规则时曾遇到的一个问题，当遇到记号NAME时，就出现了归约冲突，因为它有两条路走，归约至“expr”和“normal”都可以，这样是无法继续执行下去的。如此冲突的解决是比较简单的，将2条以上的归约减少至1条路就可以了，比如，改写 normal: NAME '.' NAME | NAME '.' NAME|...。最有效的办法是重新设计 col_str 的各种不交叉展开方式。

重写范式表达式通常是冲突问题最根本的解决办法，在语法整理分析的过程中遇到很多并非上述典型冲突类型，就会采用这种办法解决。重写表达式，一般是增加表达层次，将逻辑关系细化表达，从而达到冲突消解的目的。

2. 分析结果数据结构设计

图5揭示了分析组成部分，同时也提示了一条语句的分析结果应该是一棵如此结构的树，而树根与树枝结点的结构设计需要深究，既要存储完整的分析结果（非常复杂），又希望它尽量简单，从而减少编程量。

设计的思路应该比照分析模块的划分，遵照树结构特点，自上而下，逐一细化。

下面以 where 后跟的条件子句分析结果结构为例说明。

（1）基本的比较句结构设计

```
typedef struct _BASICSTRU  {
    char bs_colname[MAX_COL_LEN];
    //字段名,采用表名．字段名的格式
    char bs_oper[MAX_OPER_LEN]; //操作符号"">", "<",
">=", "<=", "<>", "LIKE", "BETWEEN", "IN", "NOT
IN", "IS NULLX", "IS NOT NULLX""
    int   bs_type;  //值类型,字符串或 SELECT 子句
    void *bs_value;   //值指针
} BASICSTRU;
```

（2）子结点为复杂或者基本的比较子句结构设计

```
typedef struct _COMBISTRU
{
    int cs_combi; //组合方式, NONE(即: 一个 basic 条件表达
//式) =0, ( ) =1, NOT =2, AND =3, OR =4
int cs_lefttype; //左子条件类型0基本的, 1组合的 -1 表示无
    union {
    BASICSTRU   cs_leftbs;      //基本类型的条件结构
    struct _COMBISTRU * cs_leftcs; //组合类型的条件结构
    };
    int         cs_righttype;  //右子条件类型
    union {
    BASICSTRU   * cs_rightbs;  //基本类型的条件结构
    struct _COMBISTRU * cs_rightcs; //组合类型的条件结构
    };
}COMBISTRU;
```

分析程序中的对应代码编写：

```
condition:
        condition OR condition
{ ..... }
|    condition AND condition
{ ..... }
|    NOT condition
{
    COMBISTRU * tmp;
    if( $2 ->cs_combi = =0 )
    {    /* NONE(即: 一个 basic 条件表达式) =0, '()' =
1, NOT =2, AND =3, OR =4 */
        $2 ->cs_combi =2;
        $$ =$2;
    }
    else if( $2 ->cs_combi = =2 )    /* not(not) =old */
    {
        if( $2 ->cs_lefttype = =0)
        {
$2 ->cs_combi =0;      /* not(not)basic =basic */
$$ =$2;
        }
        else /* left child is combined node */
        {
            $$ =$2 ->cs_leftcs;
            $2 ->cs_leftcs =NULL;
            FB_FREE($2);      /* free the left child */
        }
    }
else
    {
tmp =(COMBISTRU )FB_MALLOC(sizeof(COMBISTRU));
tmp ->cs_combi =2;   /* show as 'NOT' */
tmp ->cs_lefttype =1; /* say that is a combine condition
node */
        tmp ->cs_leftcs =$2;
        tmp ->cs_righttype = -1;
        $$ =tmp;
    }
}
```

上述主要是借助分析 condition: NOTcondition 时的编码来说明分析结果结构的使用是如何嵌入分析步骤中的，Bison 文件第二部分中每一条展开规则的编程都如上所述。最终从分析堆栈里退出，就意味着分析结束，获得分析结果模板。

3. 可重入分析器

保证代码的可重入性，可以将分析器应用到多线程环境下工作，它要求没有使用全局变量，错误情况能干净退出。

在语法规则文件（speed.y）的第一节中加入宏声明%pure_parser，即表示要求 bison 对应生成可重入的语法分析器，推荐 bison 的版本要新于1.35。

可重入语法分析程序的入口与普通的有所区别，外部程序通常把传入分析器 yyparse 的数据与 yyparse 传出的分析结果，组合起来定义为一个结构如：MyExtraType（见前述语法分析，bison 文件第一部分介绍）。

```
MyExtraType * my_data;
```

```
void * scanner;
my_data = (MyExtraType * )FB_MALLOC(
            sizeof(MyExtraType));
memset(my_data, 0, sizeof(MyExtraType));
//扫描分析
yylex_init ( & scanner); //词法分析程序的注册
yyset_extra ( my_data, scanner );
//设置分析器的传入、传出参数
yy_scan_string( sql, scanner); //扫描字符串 sql
if(!yyparse( scanner ))//上述定义分析任务进行分析
```

……

my_data 指向分析器传入、传出参数的内存块，scanner 是绑定 my_data 参量，联系词法分析器和语法分析器之间的一个句柄值，关于它的所有处理都在 Flex、Bison 生成的 API 函数中进行。

上述满足了用户定义的变量属于局部变量的特征，保证了生成语法分析器可重入，但是不可忽略词法分析器可重入的实现。Flex 生成可重入词法分析器，实际上从 2.5.31 版本才开始支持，它的调用方式如上述代码 yylex_init，yyset_extra，Flex 源文件相关要求见词法分析介绍。

六、 分析过程中错误处理

词法错误将在词法分析时发现，语法组织错误将会在语法分析发现，在错误处理方面语法分析和词法分析也没有明显的界限，有些错误可以在词法分析时不处理留给语法分析。遇到错误，语法树结构无法构建成功，但是在前面运行过的代码中有些构建结点已经申请，如果直接退出分析，就会造成泄漏内存，一个长期运行的服务器是不能存在内存泄漏的，所以错误处理一定要释放已经申请到的不用内存。

词法分析检错，针对的是疏忽导致的缺少符号或者写错符号，例如：字符串变量值要用单引号括起来（‘...’）会因为疏忽忘掉后面的引号。语法分析检错，能识别一个不恰当的词出现在被扫描的位置。Bison 的源文件中，用系统保留字 error 来表示某个规则出错，从而使得用户给出相应的处理动作（给出警告，丢掉错误 token，恢复处理等）。例：

```
oraselect:
        subqueryfor_update
    |   error
        {   yyerrok;    }
```

表示在分析遇到错误时，分析器还能返回到正常状态。内存泄漏的问题，可以自己管理内存的申请、释放来解决，在分析过程出错退出时，能通过一个可见句柄释放申请过而再无作用的内存块。

七、SQL 分析结果

按照设计，分析过程中会生成一棵记录语法和语义信息的树，为进行 sql 查询语句提取和修改各个子部分信息提供了基础。提取和修改在很多方面有需求，比如：格式化显示、报错、图形界面显示等。

1. 归一化 SQL 语句重构

归一化是指将具有相同查询效果而表达形式不同的查询语句归一化为同样的表述形式。譬如：Selecta, bfromtest 与 Select b, a FROM TEST 效果一样，但是形式上有顺序和大小写的不同，所以可以归一化为一种表达形式。再有如 select * fromtest where rtime>xxxxx 与 select * from test where rtime>yyyyy 效果可以有部分相同，它们可以将 rtime 给提取出来，进行其他部分的归一化；再者，wherea>band(c<dore=f)与 wherea>bandc<dore=fanda>b 等类似的例子。

2. 定制条件约束的 SQL 语句重构

对一个查询语句，如果要局限它的查询表空间和范围，通过加入条件限制的方式，使得用户无须在输入查询语句时特别指定范围条件。在查询一个以时间先后顺序为关键字的每条记录，可以将一个大时间段分割成几小段进行并行查询。

3. 替换 selectlist 的 SQL 语句重构

select 与 from 之间的字段列表（譬如：*、count(*)、a 等）可以根据需要替换，获得用户需求字段值。

上述三个实例在作者参与“数据库查询优化”项目中都有具体使用，并且运行效果较好。

八、结语

本文围绕分析 SQL 查询语句结构的全过程，展示了借助于 Flex、Bison 工具进行 SQL 语法分析的一个实践策略。虽然本文的具体实现面向的是 Oracle 的 SQL 标准，但其处理框架对于其他 DBMS 的 SQL 标准同样适用，其中 SQL 语法分析，构建无冲突语法范式和语法分析是它的核心，实践过程要遵循先范式后结果处理的顺序，具体情况下语法范式可以在不损伤无冲突条件下为方便分析结果处理、存储作些改动。可重入分析器是模块化设计和上层并行程序的需要，Flex 和 Bison 为此提供了很有效的实现接口。Flex、Bison 是强大的字符流扫描工具，它们在其他词法、语法分析方面都有广泛的应用。

参考文献

1. 陈意云．编译原理和技术．中国科学技术大学出版社，1997

2. Kenncg C. Louden. Comp iler construction p rincip les and practice[M]. 机械工业出版社，1999

3. 朱望归．数据库语言 SQL 的实现．计算机工程与应用，1994

(收稿日期:2007 年 1 月 12 日)