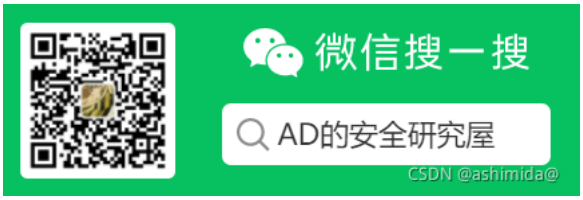
GCC源码分析(三) — 词法符号转语法符号

更多内容可关注微信公众号



语法分析要解决的主要问题是对词法分析得到的词法符号序列进行语法的推导，如果推到成功则源代码就是满足语法规范的源代码。

常见的语法分析工具包括Yacc, Bison等，早期的GCC中使用Yacc及Bison进行C语言的语法分析，而在较高的版本中不再使用二者，而是使用gcc/c-parser.c中定义的专门函数完成c语言语法分析。

C语言发展至今，其语法经历了多次修订，包含多个版本，gcc中均予以支持， 总体来讲GCC对C语言的语法分析采用一种【自顶向下】的语法推导过程，由于部分推导式有时可能会产生冲突，因此需要对下一个或两个词法符号进行预读，从而消除冲突（如=/ ==）。

因此GCC中C语言的语法分析过程可以简述为：最多提前预读两个词法符号的自顶向下的语法推导过程

在词法分析中会将源码解析为一个个词法符号，而在语法分析的一开始, 会先将此词法符号转换为语法符号，其实际的改动并不多，大体只包括3点:

1. 将词法符号中的节点值转换为具体AST树节点，如
    - 会为词法分析中的字符串生成一个tree_string节点
    - 会为词法分析中的常数生成一个如tree_int_cst节点
    - 标识符在词法分析中已经分配了树节点，这里只通过contain_of获取指针
2. 将词法符号中的标识符(CPP_NAME)更一步细分:
    - 所有匹配c_common_reswords中保留字(如if)的标识符被标记为保留字（CPP_NAME=>CPP_KEYWORD），且确定具体是哪个保留字(id_kind)
3. 确定标识符的类型，如类型/地址空间/类标识符或普通标识符

语法符号的结构体struct c_token定义如下:

```
// ./gcc/c/c-parser.h

/* 在GCC中使用struct c_token结构体来描述一个C语言中的语法符号，词法符号使用cpp_token描述 */

struct c_token {

  /*

     除了保留字会由CPP_NAME => CPP_KEYWORD外,c_token中的符号类型和cpp_token中的类型=基本相同

     在c_lex_one_token函数中则对cpp_token的返回类型做了进一步判断，如果此token和 c_common_reswords中的字符串匹配，

    那么此标识符就是一个保留字(也就是关键字),其类型会被设置为 CPP_KEYWORD，同时keyword字段会记录此关键字具体是哪个关键字。

  */

  ENUM_BITFIELD (cpp_ttype) type : 8;

  /*

    此字段记录的是标识符的类型，其只分为:

    * C_ID_ID: 普通的标识符

      除了 TYPE_DECL 之外的其他 CPP_NAME/CPP_KEYWORD，都属于普通标识符

    * C_ID_TYPENAME: typedef的类型标识符

      - 如当前的c_token 是 int，那么id_kind 就是 C_ID_TYPENAME

      - 如之前有typedef int x;定义，那么当前c_token 是X，则 也同样是C_ID_TYPENAME

    * C_ID_CLASSNAME: 类标识符

```

```
    * C_ID_ADDRSPACE: 地址标识符??

    * C_ID_NONE:    非标识符

  */

  ENUM_BITFIELD (c_id_kind) id_kind : 8;

  /*

     若一个标识符是关键字，那么此enum用来区分到底是哪个关键字,如 static关键字的keyword编号为 RID_STATIC,

     非关键字的keyword默认为为 RID_MAX

    注：关键字和指令标识符不同，关键字(保留字)是在任何情况下都要特殊处理的,如if,而保留字只有在#后才会被特殊处理，如#define

     c_common_reswords中记录了系统的所有保留字，dtable中记录大部分指令标识符

  */

  ENUM_BITFIELD (rid) keyword : 8;

  ENUM_BITFIELD (pragma_kind) pragma_kind : 8;        //编译制导的标识符

  location_t location;    /* 记录token在源代码中的位置 */

  /*

    在词法符号(cpp_token)中，保存的都是标识符的字符串信息，而在语法符号中，则要将这些字符串信息构建到一个对应的AST tree中。

    而value节点就是用来保存不同的语法符号构建的那个AST tree节点的(见 c_lex_one_token):

    * 对于 CPP_NAME，即标识符来说，直接从cpp_token中container_of就可以返回一个tree_identifer树节点(词法分析虽然返回了一个hashnode，但实际上分配了一个tree

    * 对于 CPP_STRING，即字符串来说,在c_lex_one_token函数中会为字符串生成一个tree_string节点并返回

    * 对于 CPP_CHAR/CPP_NUMBER 来说,在c_lex_one_token函数中会为常量创建一个常量节点如tree_int_cst/tree_poly_int_cst并返回

  */

  tree value;

  unsigned char flags;    /* Token flags.  */

  ......
};
```

在词法分析中其接口函数只有一个，就是_cpp_lex_token, 此函数最终返回的cpp_token就是词法分析的结果；而在语法分析中实际上有多个API组成了其接口函数,主要包括:

1. c_parser_peek_token: 预读一个语法符号c_token
2. c_parser_peek_2nd_token: 预读当前未分析的第二个语法符号
3. c_parser_peek_nth_token: 预读当前未分析的第n个语法符号(n<4)
4. c_parser_consume_token: 消耗掉当前第一个语法符号

其中1-3是用来获取一个新的语法符号的，而4是用来消耗掉一个已经使用完毕的语法符号的, 和词法分析不同的是:

- **词法分析最终会保留所有的cpp_token**到全局的parse_in的buffer中，每次调用_cpp_lex_token都会从源码中解析出一个新的词法符号
- 而语法分析多次调用c_parse_peek_*_token获取的是同一个语法符号，直到调用c_parser_consume_token消耗掉此语法符号后再次调用才会获取到下一个语法符号。

在1-3中，最终实际上都是**通过c_lex_one_token函数获取真正的语法符号**，并将其保存到c_parse结构体中，这里仅以c_parse_peek_token为例:

```
//./gcc/c/c-parse.c
c_token *c_parser_peek_token (c_parser *parser)

{

  if (parser->tokens_avail == 0)

    {

      /* 若parse中没有可用的语法符号了，则调用语法解析函数，解析出一个语法符号 */

      c_lex_one_token (parser, &parser->tokens[0]);

      parser->tokens_avail = 1;

    }

  /* 若当前parse中有未消耗的符号，则直接拿来用 */

  return &parser->tokens[0];
```

```
21.
22. }
```

这里的c_parser传入的是全局变量the_parser,此结构体定义如下:

```
1. struct c_parser {
2.
3.   c_token * tokens;    /* 当前正在处理的语法符号c_token的地址，这里除了初始化时，应该指向 tokens_buf[0] */
4.
5.   c_token tokens_buf[4];  /* c_token预读缓存，按照gcc的语法分析原理，预读不会超过4个语法符号 */
6.
7.   unsigned int tokens_avail;    /* tokens_buf中可用的预读词法符号的数目 */
8.
9.   BOOL_BITFIELD error : 1;            //是否已经从语法分析错误中回复
10.
11.  BOOL_BITFIELD in_pragma : 1;        //是否在进行编译制导的处理
12.
13.  BOOL_BITFIELD in_if_block : 1;    //是否在处理最顶层的if语句
14.
15.  /* True if we want to lex an untranslated string.  */
16.
17.  BOOL_BITFIELD lex_untranslated_string : 1;
18.
19.    ......
20.
21.  /* Location of the last consumed token.  */
22.
23.  location_t last_token_location;    //已经分析过的(consumed)最后一个token在源码中的位置
24.
25. };
```

此结构体中除了一些状态位之外，基本只有一个大小为4的预读数组，在语法分析中不保留历史的语法符号(c_token)，所以大小为4的数组就够了。

c_lex_one_token的具体实现可参考源码.