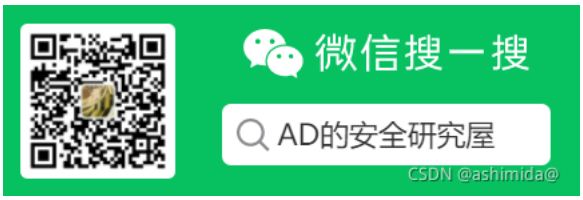
GCC源码分析(二) — 词法分析



GCC的词法分析是在伴随语法分析完成的，当语法分析过程中找不到下一个token时，就会调用词法分析来解析后续的token。**词法分析的代码在./gcc/c-family目录;而C语言语法分析的目录在./gcc/c目录下，词法分析是给所有C家族的语言共用的。**GCC的词法分析的主要代码是从**cc1(如./gcc/c-family/c-lex.c) => libcpp(如./gcc/libcpp/lex.c)**中的，对于cc1来说这些代码都编译进其二进制了。

lex是LEXical compiler的缩写，LEX是UNIX下著名的词法分析器,GCC早期的版本使用Lex/Flex工具进行C语言的词法分析，较新的版本则使用专门的词法分析代码，主要位于gcc/c-lex.c中，其API接口函数为接口函数_cpp_lex_token，真正解析词法正则表达式的函数为_cpp_lex_direct。

```
## _cpp_lex_token => _cpp_lex_direct
```

_cpp_lex_token的一次调用就会从文件中解析出一个词法元素，此词法元素可以是一个标识符，数字，字符串，或者是一个操作符号(如 + 代表加法)，此函数中会自动忽略空格和回车，此函数返回的是一个cpp_token，代表一个新解析出来的符号。

在整个词法分析的过程中涉及了很多个结构体，理清这些结构体的关系有助于后续的分析。**在整个源码=>词法分析结束过程中，和词法分析相关的信息都记录在parse_in结构体中，或者可以通过此结构体索引到，所以下面的所有结构体分析，都是从parse_in开始依次展开的。**

1. struct cpp_reader

cpp_reader在全局就只有一个全局变量 parse_in, 从源码开始到整个词法分析结束过程中的都有信息基本都可以通过此结构体索引到，如在此过程中所有打开的文件信息,文件内容,文件当前词法分析到哪里了,所有解析出的词法元素的存储位置,所有识别出的标识符(及其hash表与节点),字符串,数字的存储位置等。

```
## ./gcc/c-family/c-common.c
cpp_reader *parse_in;

##./libcpp/internal.h    //这里只记录部分结构体成员
struct cpp_reader
{
  /*
     buffer是一个链表，每一个元素都记录了一个文件内容存储位置,此文件当前解析到哪里了和文件结构体指针等信息.
     cc1每次只解析一个编译单元，这里之所以会是个链表，是因为当代码中遇到如#include之类的会递归解析，
     此时会push新的文件到buffer,新的文件会先被解析,直到解析完成再返回原有buffer继续解析，因此这里会出现buffer列表.
  */
  cpp_buffer *buffer;
  struct line_maps *line_table;    //所有源码的行号信息都可以通过这里索引
  /* The line of the '#' of the current directive.  */
  location_t directive_line;
  /* If in_directive, the directive if known.  */
  const struct directive *directive;
  /* Token generated while handling a directive, if any. */
  cpp_token directive_result;
  /* When expanding a macro at top-level, this is the location of the macro invocation.  */
  location_t invocation_location;
  struct _cpp_file *all_files;    //词法分析过程中,所有打开的文件都会记录到这里
  struct _cpp_file *main_file;    //当前cc1编译的主文件
  /* Lexing. 以下是具体词法分析相关成员 */
  cpp_token *cur_token;    //当前词法解析最新解析出的符号(在词法分析中，每个符号都存储为一个cur_token结构体)
  /*
    一个tokenrun中主要存储了一个cpp_token的数组(默认250个元素),这个数组是一次性分配的,每次解析一个新的token，就会让cur_token++,指向下一个位置.
    当token_run不够时，就会分配一个新的token_run，所有的token_run都是通过自身的链表链接的。
    * base_run记录系统中第一个分配的cpp_token[]数组的信息
    * cur_run指向当前正在使用的token_run结构体的指针,实际上也是最后一个分配(并正在使用)的token_run的指针.
  */
  tokenrun base_run, *cur_run;
  /*
    之前词法分析预先解析出多少个cpp_token,这些预先解析出来的cpp_token，实际上就存在
    cur_token[0] - cur_token[lookaheads] 中,下次解析如果有预解析的token，直接cur_token++即可
  */
  unsigned int lookaheads;
  /*
     此成员代表标识符的hash表，其以hash表为结构，记录了词法分析中分析出的所有标识符的指针(相同标识符系统内是只一份存储的)
     - 其entries[]是一个记录标识符地址指针的数组，这个数组就是这里的hash表。
     - 其alloc_node函数是用来为标识符分配存储空间的
     - 标识符的hash是根据其字符串的每个字符和字符串长度来确定的
     若hash冲突，则会循环用二次hash算法找下一个位置
    注意这里只是个指针，通常指向全局变量 ident_hash
  */
  struct ht *hash_table;
};
```

parse_in结构体被使用的主要三个场景:

```
toplev::main
    =>lang_hooks.init_options = c_common_init_options
    => parse_in = cpp_create_reader()                          //1) 这里主要是对全局变量parse_in的初始化
  => do_compile
    => process_options
      => lang_hooks.post_options = c_common_post_options
        => cpp_read_main_file (parse_in, in_fnames[0])         //2) 这里主要负责打开并读入编译单元文件
    => compile_file();
      => lang_hooks.parse_file = c_common_parse_file()
        => c_parse_file ();                                    //3) 这里每一次的词法分析(获取一个token)都要用到parse_in
```

2. struct _cpp_file/struct cpp_buffer

这两个结构体通常是一起分配的：一个_cpp_file结构体代表一个打开的文件的信息，一个cpp_buffer结构体则代表当前此文件在词法解析中的状态信息,一般这两个结构体的分配有两个场景:

```
toplev::main
    =>lang_hooks.init_options = c_common_init_options
    => parse_in = cpp_create_reader()                                  //场景1，cc1初始化时读入主文件(编译单元文件)时为其分配_cpp_file和cpp_buffer
      => pfile->main_file = _cpp_find_file(pfile, fname,...)
        => file = make_cpp_file (pfile, start_dir, fname);               //分配__cpp_file
      => _cpp_stack_file (pfile, pfile->main_file, false, loc);          //分配cpp_buffer,并读入文件内容
  => do_compile
    => compile_file();
      => lang_hooks.parse_file = c_common_parse_file()
        => c_parse_file ();
          .... => _cpp_stack_include                                   //场景2，当语法解析过程中发现include了新文件时，同样为其分配_cpp_file和cpp_buffer
                => file = _cpp_find_file (...)
                   => file = make_cpp_file (pfile, start_dir, fname);          //分配__cpp_file
                => stacked = _cpp_stack_file (pfile, file, type == IT_IMPORT, loc);    //分配cpp_buffer,并读入文件内容
```

这两个结构体内容如下:

```
// ./libcpp/files.c    //省略部分成员
struct _cpp_file
{
  const char *name;    /* 文件的base name */
  const char *path;    /* 文件全路径名，打开文件时用这个名 */
  struct _cpp_file *next_file;  //此链表链接所有打开的文件，最终链接到parse_in.all_files上
  const uchar *buffer; /* 这里存的是真正的文件内容的字符串数组，在 _cpp_stack_file读入文件内容时分配的 */
  int fd;    /* File descriptor.  Invalid if -1, otherwise open.  */
  unsigned short stack_count;   //文件在预处理过程中已经stack的次数
  bool main_file;        /* 标记此文件是否为编译单元的主文件 */
  bool buffer_valid;    //buffer中是否包含真正的文件内容
};

// ./libcpp/internal.h
struct _cpp_file
struct cpp_buffer
{
  const unsigned char *cur;        /* 指向在词法分析过程中当前已经解析到了哪个字符 */
  const unsigned char *line_base;  /* 当前行的起始地址，同样是 buf中的某个偏移 */
  /*
     在cpp_buffer初始化时(parse_in场景1,cpp_read_main_file => _cpp_stack_file)，指向文件内容字符串数组的首地址
     在buffer使用时，会一行一行的解析，碰到\r或 \r\n就代表一行的结束，那么next_line就会指向逻辑上 下一行的起始字符串
  */
  const unsigned char *next_line;  /* Start of to-be-cleaned logical line.  */
  const unsigned char *buf;        /* 指向文件内容字符串首地址  */
  const unsigned char *rlimit;     /* 指向文件内容字符串的尾地址 */
  const unsigned char *to_free;    /* Pointer that should be freed when popping the buffer.  */
  struct cpp_buffer *prev;         /* 此buffer代表的文件打开前正在解析的那个文件的buffer */
  struct _cpp_file *file;          /* 指向代表此文件的file结构体，文件内容字符串指针是从这里获取的 */
  bool need_line;                  /* 代表当前这一行代码已经处理完，需要读入一行新的数据，其初始值为true,见 _cpp_get_fresh_line */
  ......
};
```

3. struct tokenrun/struct cpp_token/lookahead

词法分析中分析出的所有词法符号在编译过程中都会一直保存着，一个cpp_token结构体就代表词法分析中分析出的一个符号; 由于每个词法元素都需要一个cpp_token, 在整个编译过程中通常需要大量的cpp_token结构体，在cc1中的做法是一次分配250个cpp_token[]数组,等到用尽后再分配250个，而一个tokenrun就是用来记录一个包含250个cpp_token的数组的信息的。

```
// ./libcpp/internal.h
struct tokenrun
{
  tokenrun *next, *prev;    /* tokenrun是一个双向链表 (但不是循环链表),next/prev是后/前向指针，系统中第一个tokenrun的指针保存在 parse_in->base_run中*/
  /*
     每次分配一个tokenrun(next_tokenrun 函数)，都会分配250个cpp_token[]的位置base和limit分别指向首尾的位置;
     tokenrun的分配是在正常的词法解析过程中发现未知不够了调用next_tokenrun分配的，见 _cpp_lex_token
  */
  cpp_token *base, *limit;
};

// ./libcpp/include/cpplib.h
/*
   cpp_token指的是 C preprocessing token,是用来保存一个词法元素的结构体(语法元素是用c_token结构体保存的)，词法元素
  可以是标识符，数字，字符串，或操作符如+等(见 _cpp_lex_direct 中的分类),在词法分析过程中每确认一个词法元素就会生成一个
```

```
 cpp_token结构体来保存此词法元素的信息。
*/

struct cpp_token {
  location_t src_loc;    /* 记录词法元素中第一个字符的源码位置 */
  /*
    记录词法分析解析出的符号的属性(见_cpp_lex_direct),如:
    * 若当前从源码中解析出了一个数字，则此数字的cpp_token.type = CPP_NUMBER.
    * 若当前从源码中解析出了一个字符/字符串/头文件(如<stdio.h>),其对应的 cpp_token.type = CPP_WSTRING/CPP_WCHAR/CPP_HEADER_NAME.
    * 若当前从源码中解析出了一个标识符，则此标识符对应的cpp_token.type = CPP_NAME (关键字在词法分析中被认为是标识符)
  */
  ENUM_BITFIELD(cpp_ttype) type : CHAR_BIT;  /* token type */
  /*
     这里记录的是获取当前词法元素过程中，发生的一些事情(见 _cpp_lex_direct)，如:
     * BOL，代表此token是一行的第一个token
     * PREV_WHITE 代表在解析到此token之前，遇到了空白字符
  */
  unsigned short flags;        /* flags - see above */
  union cpp_token_u    /* 这个cpp_token_u 代表的是为各个词法元素最终建立的值节点，不同的词法元素使用不同的结构体。 */
  {
    /*
       若词法元素是标识符，则此结构体记录标识符的字符串内容，hash等信息(对于标识符是不使用cpp_string的),此时的type = CPP_NAME
    */
    struct cpp_identifier node;
    cpp_token * source;
    /*
        对于数字或字符串，都是用cpp_string来存储
         * 若词法元素是数字，则此处保存的是源码中的这个数字字符串的内容和长度，此时对应的type为CPP_NUMBER
           如字符串内容"1234",就是这个数字字符串，见_cpp_lex_direct => lex_number
         * 若词法元素是字符串，此处保存的是源码中的这个字符串的内容和长度，此时对应的type为CPP_WSTRING/CPP_WCHAR/CPP_HEADER_NAME
           见 _cpp_lex_direct => lex_string => create_literal
    */
    struct cpp_string str;
        ......
  } val;
};
```

在parse_in结构体中还有一个lookahead字段记录在当前的tokenrun中有多少个预读的cpp_token, 因为cpp_token是数组顺序排列的,所以获取下一个token只需 cur_token ++

4. struct ht

这个结构体虽然叫ht(hash table),但其实际上只能作为字符串的hash table,因为其存储的元素固定为struct ht_identifier(虽然叫标识符，但本质上是记录字符串和其对应hash的)

ht表中的entries字段指向一个hashnode[]数组，其每个元素都是一个hashnode,其之所以叫hashnode,是因为其实际上只记录了一个hash节点的指针，hashnode这个指针真正指向的才是一个struct ht_identifier结构体.

```
// ./libcpp/include/symtab.h
struct   ht_identifier {
  const unsigned char *str;    //标识符的字符串名的指针
  unsigned int len;            //字符串名的长度
  unsigned int hash_value;    //字符串的hash
};

typedef struct ht_identifier *hashnode;

struct ht
{
  /* Identifiers are allocated from here.  */
  struct obstack stack;        //负责此hash表中的内存分配
  /*
    指向一个hashnode[nslots]数组的首地址，这个数组就是所谓的hash桶，数组中的每个元素都记录了一个具体元素的指针(所以每个元素叫做一个hashnode)
    而hashnode具体的元素则是一个 ht_identifer,其只能代表一个字符串的内容，长度和hash.
    此hash桶是自动扩展的，在ht搜索函数ht_lookup_with_hash中，若发现整个hash table超过3/4都满了，就会主动扩展此hash table(重新分配，复制，释放原有的)。
  */
  hashnode *entries;
  /*
     整个gcc源码中有两个alloc_node函数，一个定义在./gcc/stringpool.c中，一个定义在libcpp/identifiers.c中(libcpp这个目录是负责预处理和词法分析的).
     * 对于cc1来说，其有自己的alloc_node函数，调用的总是 gcc/stringpools.c:alloc_node
     * 而对于使用libcpp.a的其他程序，如果自己没有实现alloc_node函数，那么会默认使用./libcpp/identifier.c:alloc_node函数
     alloc_node函数是用来分配节点内存的，分配后hashnode[]数组中的指针，也就指向这个内存中的元素，在ht搜索过程中(ht_lookup_with_hash),若发现需要新
     插入一个元素，则就会调用alloc_node来分配内存，最终其返回值会被记录到hashnode[]中。
    注: alloc_node可以为节点分配任意类型的结构体，只要最终返回此结构体中的一个ht_identifer结构体即可(./gcc/stringpool.c真正分配的是一个 lang_identifier校
  */
  hashnode (*alloc_node) (cpp_hash_table *);
  /* Call back, allocate something that hangs off a node like a cpp_macro.  NULL means use the usual allocator.  */
  void * (*alloc_subobject) (size_t);
  unsigned int nslots;          /* hash桶中总共能存多少个指针(也就是entires 数组大小 */
  unsigned int nelements;       /* 当前entries中已使用的位置个数 */
  struct cpp_reader *pfile;     /* 指向对应的cpp_reader(即parse_in),这里说明ht和reader是绑定的 */
  unsigned int searches;        /* 记录当前table(ht结构体)被搜索过的次数 */
  unsigned int collisions;      /* 记录hash冲突的次数 */
  /* Should 'entries' be freed when it is no longer needed?  */
  bool entries_owned;
};
```

在parse_in的初始化过程中:

```
toplev::main
```

```
2.   => general_init (argv[0], m_init_signals);
3.     => init_stringpool (void)                              //1) 为全局变量struct ht* ident_hash分配空间并初始化alloc_node
4.   =>lang_hooks.init_options = c_common_init_options
5.     => parse_in = cpp_create_reader()
6.       => _cpp_init_hashtable (pfile, table);               //2)设置parse_in->hash_table = ident_hash, 作为词法解析过程中的全局标识符hash表
```

整个词法分析是伴随着语法分析进行的,当语法分析需要新的符号时,_cpp_lex_token函数就会解析并返回一个词法分析中的token(struct cpp_token)供语法分析使用.