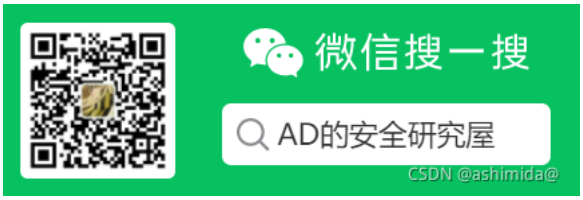
GCC源码分析(五) — 语法/语义分析之声明符解析(上)

更多内容可关注微信公众号



c_parser_declaration_or_fndef在**解析完声明说明符之后接着要解析的就是声明符**，对于函数和声明来说，虽然产生式不一样但这一步基本是一样的(除了断言在之前已经处理过了),函数定义和声明的组成都是 "声明说明符 声明符 ... "开头的,所以在声明说明符解析完毕后不论是函数还是声明的解析，下一步都是要解析声明符, 而声明符的解析函数是:

```
c_parser_declaration_or_fndef =>  c_parser_declarator
```

声明符实际上指的是如具体的变量或函数名，举例如下:

```
1. static int x, y=0;    // 这里的x和y都是声明符
2. int func(void x);     //这里的func和x都是声明符
3. int func(void x)       //这里的func和x都是声明符
4. {
5.   ......
6. }
7. // 声明说明符的意思是声明的说明符，其本质是多个说明符的组合，如这里的static，int都是一个单独的说明符
8. // 而"声明的"中的声明，指的就是这里的func,x,y,这些才是真正的声明符.
```

c_parser_declarator函数是声明符的解析函数， **此函数的返回就代表一个声明符解析的结束**，在声明和函数定义的解析过程中(c_parser_declaration_or_fndef),由于二者的产生式不完全相同(在一个声明中产生式是初始声明符列表，而函数定义中是单一的声明符),所以此函数默认是需循环解析多个声明符的，而如果解析完一个声明符发现是函数定义，则无需再解析下一个，其源码简化如下:

```
1. static void c_parser_declaration_or_fndef (c_parser *parser, bool fndef_ok, bool static_assert_ok, bool empty_ok,
2.                 bool nested, bool start_attr_ok, tree *objc_foreach_object_declaration, vec<c_token> omp_declare_simd_clauses,
3.                 struct oacc_routine_data *oacc_routine_data, bool *fallthru_attr_p)
4. {
5.   struct c_declspecs *specs;
6.   ......
7.   specs = build_null_declspecs ();    /* 为声明说明符结构体(struct c_declspecs)分配空间 */
8.   ......
9.
10.  /* 解析声明说明符,所有的说明符(如static int x, y; 中的static int分别是两个说明符)的信息最终都会被记录到同一个 c_declspecs 结构体中返回(到specs)  */
11.  c_parser_declspecs (parser, specs, true, true, start_attr_ok, true, true, cla_nonabstract_decl);
12.
13.    /* 代表声明说明符解析完毕，此函数只对部分关键字修正类型 */
14.  finish_declspecs (specs);
15.
16.  /* 不论声明还是函数定义，声明说明符后面至少要有一个声明符，这里的while循环用来解析一个或多个声明符 */
17.  while (true)
18.    {
19.      /*
20.        c_declarator指针代表一个声明符解析的结果，其通常是一个链表，链表中的每个元素都是一 个c_declarator,称为c声明符，如 int * p; 实际上是由两个c声明符链
21.        第一个c声明符代表指针(这里的*)，第二个c声明符代表标识符p,二者链接后的c_declarator结构体，才代表int *p;中*p整个声明符
22.      */
23.      struct c_declarator *declarator;
24.      bool dummy = false;
25.      tree fnbody = NULL_TREE;
26.      /* 解析一个声明符，  如 int x,y; 中的x */
27.      declarator = c_parser_declarator (parser, specs->typespec_kind != ctsk_none, C_DTR_NORMAL, &dummy);
28.
29.      /* 声明说明符 声明符 解析完毕，如果其后面接的是 =/;/,/或者一些c汇编等其他的，则走这里，除了函数定义(后面接函数体)，基本都走这里 */
30.      if (c_parser_next_token_is (parser, CPP_EQ)
31.      || c_parser_next_token_is (parser, CPP_COMMA)
32.      || c_parser_next_token_is (parser, CPP_SEMICOLON)
33.      || c_parser_next_token_is_keyword (parser, RID_ASM)
34.      || c_parser_next_token_is_keyword (parser, RID_ATTRIBUTE)
35.      || c_parser_next_token_is_keyword (parser, RID_IN))
36.    {
37.      /* 如果已经识别到一个声明了，那么后续不可能再出现函数定义了 */
38.      fndef_ok = false;
39.      /* 解析到等号则说明此声明符为初始声明符,需要赋初值,如 int x = 0; */
40.      if (c_parser_next_token_is (parser, CPP_EQ))
41.        {
42.          /* 为当前的声明符构建一个声明节点，除了赋值外其他操作这里都完成了 */
43.          d = start_decl (declarator, specs, true, chainon (postfix_attrs, all_prefix_attrs));
44.          start_init (d, asm_name, global_bindings_p (), &richloc);
45.          init = c_parser_initializer (parser);
46.          flag_sanitize = flag_sanitize_save;
47.          finish_init ();
48.        }
```

```
      else
        {
          /* 非等号的情况下不用赋初值，但是也同样要构建声明节点 */
          tree d = start_decl (declarator, specs, false, chainon (postfix_attrs, all_prefix_attrs));
          if (d & ...)
            DECL_ARGUMENTS (d) = declarator->u.arg_info->parms;
          if (d)
            finish_decl (d, UNKNOWN_LOCATION, NULL_TREE,NULL_TREE, asm_name);
        }

      /* 到这里代表声明说明符 声明符均解析完毕，且声明节点也生成完毕了,若后续为逗号，则要循环解析下一个声明符 */
      if (c_parser_next_token_is (parser, CPP_COMMA))
        {
          c_parser_consume_token (parser);
          ......
          continue;
        }
      else if (c_parser_next_token_is (parser, CPP_SEMICOLON))
        {
          /* 这里是分号的情况，直接消耗掉且代表当前声明解析完毕了 */
          c_parser_consume_token (parser);
          return;
        }
      else error; return;
    }

    /* 到这里(没有匹配到;/=/,等符号),则说明这是一个函数定义，开始解析函数体 */
      if (!start_function (specs, declarator, all_prefix_attrs))
    ......
}
```

可以看到在声明或函数定义的解析过程中默认是循环解析声明符的， **c_parser_declarator**一次只解析一个声明符, 并最终返回一个c_declarator指针(若存在嵌套则嵌套的内部会完全解析完毕，如 int func(int x, int y); 在返回函数func(int x, int y)的c_declarator节点时(注意这个整体叫做函数,func叫做标识符),int x, int y的声明节点都已经构建完毕，并连接到c_declarator->arg_info->parms中了), 此函数的大体流程如下:

```
/* 此函数实际上只递归处理了声明符开头的多个指针部分，其余部分(直接声明符),是通过子函数c_parser_direct_declarator处理的，每个c声明符的结果通过c_declarator结
struct c_declarator *
c_parser_declarator (c_parser *parser, bool type_seen_p, c_dtr_syn kind,
             bool *seen_id)
{
  /* 对于指针开头的声明符,单独处理指针，剩余部分递归处理 */
  if (c_parser_next_token_is (parser, CPP_MULT))
    {
      /* 指针和直接声明符之间可能会有类型限定符和属性，这里先处理下类型限定符和属性(若有)，然后再递归处理，如 int * const p; 中的const */
      struct c_declspecs *quals_attrs = build_null_declspecs ();
      struct c_declarator *inner;
      c_parser_consume_token (parser);        /* 消耗掉此 * 语法元素 */
      /* 复用声明说明符解析函数解析类型限定符列表,只解析attribute和类型限定符 两类说明符 */
      c_parser_declspecs (parser, quals_attrs, false, false, true, false, false, cla_prefer_id);

      /* *之后还有可能有*，产生式中没有*个数限制，故这里再按照正常的声明符递归解析 */
      inner = c_parser_declarator (parser, type_seen_p, kind, seen_id);
      if (inner == NULL)
        return NULL;    /* NULL代表解析出错 */
      else
      /* 为*生成一个新的c_declarator,包裹内部的解析结果返回，内部(inner)则是递归后最终通过 c_parser_direct_declarator解析出来的 */
    return make_pointer_declarator (quals_attrs, inner);
    }
  /* 非指针开头的情况，调用此函数解析直接声明符，最终返回一个c_declarator链表 */
  return c_parser_direct_declarator (parser, type_seen_p, kind, seen_id);
}
```

c_parser_declarator递归处理语法符号*后，最终调用其子函数c_parser_direct_declarator解析直接说明符，其代码简化如下:

```
/*
  direct-declarator:
    identifier T4
    (attributes[opt] declarator) T4
  此函数只是用来解析上面的产生式中非T4的部分的,而T4的部分则是通过其子函数c_parser_direct_declarator_inner解析的,
 非T4的部分是声明符的标识符部分，T4部分是声明符[],或()内部包裹的部分，故其解析函数中有一个inner.
  故此函数中只是识别了声明符中的标识符部分(若有)，包括int x;中的x，或 int func(...)中的func,或int p[...]中的p;
*/
static struct c_declarator *
c_parser_direct_declarator (c_parser *parser, bool type_seen_p, c_dtr_syn kind,
                bool *seen_id)
{
   /* 正常除了 int (*p)(...);这种，以标识符开头的case基本都走这里 */
  if (kind != C_DTR_ABSTRACT
      && c_parser_next_token_is (parser, CPP_NAME)
      && ((type_seen_p
       && (c_parser_peek_token (parser)->id_kind == C_ID_TYPENAME
           || c_parser_peek_token (parser)->id_kind == C_ID_CLASSNAME))
      || c_parser_peek_token (parser)->id_kind == C_ID_ID))
    {
      /* 因为匹配到标识符了，这里纤维标识符构建一个c声明符(c_declarator)节点 */
      struct c_declarator *inner = build_id_declarator (c_parser_peek_token (parser)->value);  //1)
      /* 这里并没有区分标识符列表的解析，在 grokparms 函数中才会真正的检查  */
```

```
      *seen_id = true;
      inner->id_loc = c_parser_peek_token (parser)->location;
      c_parser_consume_token (parser);
      /*
         这里传入的inner是标识符节点，如果当前是一个普通标识符定义，那么inner直接返回，如果是函数或数组定义,以函数int func(int x, int y); 为例，最终返回的
        func(int x, int y); 而其中的func是一个声明符，int x, int y是一个参数类型列表，二者都属于此函数的inner，这里将标识符func当做inner传入，而此函数内部
        最终返回的c_declarator则代表此函数.
      */
      return c_parser_direct_declarator_inner (parser, *seen_id, inner);
    }

  /* 参数列表解析过程中会递归到声明符解析，此时会调用到这里,如 int func(int [10])中 int [10]的解析会进入这里，在参数列表中声明符可以省略标识符 */
  if (kind != C_DTR_NORMAL && c_parser_next_token_is (parser, CPP_OPEN_SQUARE))
    {
      /* 参数声明中的声明符可以省略标识符，此时设置标识符节点为空树,其他不变 */
      struct c_declarator *inner = build_id_declarator (NULL_TREE);
      inner->id_loc = c_parser_peek_token (parser)->location;
      return c_parser_direct_declarator_inner (parser, *seen_id, inner);
    }

  /* 若直接声明符的第一个符号不是标识符或参数解析时的方括号，那么就必须是圆括号,否则语法错误,这里解析的是圆括号的情况 */
  if (c_parser_next_token_is (parser, CPP_OPEN_PAREN))
    {
      ......
      /* 对于 int (*p)(...); 这种，先匹配 *p这个声明符  */
      inner = c_parser_declarator (parser, type_seen_p, kind, seen_id);
      /* *p的位置只能是一个声明符，然后必须就是闭括号了，否则报错 */
      if (c_parser_next_token_is (parser, CPP_CLOSE_PAREN))
      {
        ......
        /* 解析后面(...)的内容，并返回 */
        return c_parser_direct_declarator_inner (parser, *seen_id, inner);
      }
      else {
        /* (*p 后面没有直接闭括号，语法错误 */
        c_parser_skip_until_found (parser, CPP_CLOSE_PAREN, "expected %<)%>");
        return NULL;
      }
    }
  else
    {
      /* 如果不是 open paren，且是在解析正常的直接声明符，则报错，没有这样的产生式,如 int , */
      if (kind == C_DTR_NORMAL)
          c_parser_error (parser, "expected identifier or %<(%>"); return NULL;
      else
          return build_id_declarator (NULL_TREE);
    }
}
```

其子函数c_parser_direct_declarator_inner则负责解析声明符中的T4表达式:

```
/*
   T4:
   array-declarator T4
   ( parameter-type-list ) T4
   ( identifier-list[opt] ) T4
   empty
   T4表达式实际上代表一个声明符括号"内部"的东西，如 int func(...);或 int p[...];中的 ...
   此函数中的inner通常是标识符的c_declarator结构体，如这里的func,或p的c_declarator，func/p对于函数func(...)或数组p[...]来说，是其内部的标识符元素，且在解
   已经解析完毕了，故会被当做此函数的inner传进来
*/
static struct c_declarator *
c_parser_direct_declarator_inner (c_parser *parser, bool id_present,
                  struct c_declarator *inner)
{
  /* 若 int fun 后面跟的是一个 open square "[" ，则说明是一个数组定义*/
  if (c_parser_next_token_is (parser, CPP_OPEN_SQUARE))
    {
      ......
      /* 解析数组 [...]内部的 ... */
      dimen = c_parser_expr_no_commas (parser, NULL);
      /* 右值转左值 */
      dimen = convert_lvalue_to_rvalue (brace_loc, dimen, true, true);
      /* 创建一个c声明符记录 [...]内部的内容 */
      declarator = build_array_declarator (brace_loc, dimen.value, quals_attrs, static_seen, star_seen);
      /* 再创建一个数组c声明符，将[...] 和数组标识符都记录其中，此声明符代表整个数组 */
      inner = set_array_declarator_inner (declarator, inner);
      /* 继续递归解析T4表达式，如果后面是empty就直接返回inner 了 */
      return c_parser_direct_declarator_inner (parser, id_present, inner);
    }
  /* 如果跟着的是 open paren "(", 则代表是函数声明/定义/调用 */
  else if (c_parser_next_token_is (parser, CPP_OPEN_PAREN))
    {           ......
      /*
         遇到括号，则这里解析参数类型列表(parameter-type-list),或标识符列表(identifier-list[opt]) ,此函数最终返回
        一个c_arg_info结构体分别代表解析到的形参/实参列表.
      */
      args = c_parser_parms_declarator (parser, id_present, attrs);

      /* 遇到括号，则一定是函数声明或函数定义或函数调用，这里都会先生成一个函数声明的c声明符,参数分别为函数名(inner)和形参/实参列表(args)*/
```

```
      inner = build_function_declarator (args, inner);
      /* 这里是括号解析完毕后，递归重新解析T4表达式 */
      return c_parser_direct_declarator_inner (parser, id_present, inner);
    }
  /*  非括号的情况直接返回，这也是递归的最终终结位置 */
  return inner;
}
```

c_parser_direct_declarator_inner的子函数c_parser_parms_declarator负责解析参数类型列表或标识符列表:

```
/*
    c_parser_parms_declarator负责解析的是( parameter-type-list )或( identifier-list[opt] )中括号内的两部分
  参数id_list_ok代表当前是否支持标识符列表的解析
  参数attrs是其父函数中解析出来的属性(对于参数列表可能非空，对于标识符列表一定为空)
*/
static struct c_arg_info *
c_parser_parms_declarator (c_parser *parser, bool id_list_ok, tree attrs)
{
  /* 新建一层scope */
  push_scope ();
  /* 标记当前scope是个参数解析用的 */
  declare_parm_level ();

  /* 这个if满足则确定当前语法符号满足标识符列表的条件，按照标识符列表解析 */
  if (id_list_ok && !attrs  && c_parser_next_token_is (parser, CPP_NAME) && ...)
    {
      tree list = NULL_TREE, *nextp = &list;
      /*
         循环解析标识符列表，其中的每个循环中的符号都必须是普通声明符，实际上就是解析identifier-list的产生式
        identifier-list:
            identifier
            identifier-list , identifier
      */
      while (c_parser_next_token_is (parser, CPP_NAME)
         && c_parser_peek_token (parser)->id_kind == C_ID_ID)
    {
      /* 对于CPP_NAME来说,value就是个标识符节点(lang_identifier),这里新建一个tree_list节点，此节点保存的内容为此标识符节点的指针 */
      *nextp = build_tree_list (NULL_TREE, c_parser_peek_token (parser)->value);
      /* 最终按照标识符列表的出现顺序，list链接了所有标识符， 如 func(x,y) 则 list = tree_list(x); list->chain = tree_list(y); */
      nextp = & TREE_CHAIN (*nextp);
      /* 解析标识符列表，若标识符之间不是逗号，则解析结束，如果是则消耗掉此逗号继续 */
      c_parser_consume_token (parser);
      if (c_parser_next_token_is_not (parser, CPP_COMMA)) break;
      c_parser_consume_token (parser);

      /* 逗号后面跟闭括号是错误的，如(a,) 正常应该是标识符后面跟括号，如 (a,b) */
      if (c_parser_next_token_is (parser, CPP_CLOSE_PAREN)) {
          c_parser_error (parser, "expected identifier"); break;
        }
    }

    /* 解析完之后，如果标识符列表后面是闭括号，则代表解析正确 */
      if (c_parser_next_token_is (parser, CPP_CLOSE_PAREN))
     {
      /* 构建并分配一个结构体 c_arg_info,此结构体用来返回结果 */
      struct c_arg_info *ret = build_arg_info ();
      /* 解析标识符列表的返回结果就是将这个tree_list记录到types中返回，其含义是实参(的标识符)列表 */
      ret->types = list;
      c_parser_consume_token (parser);
      /* 退出当前scope并返回 */
      pop_scope ();
      return ret;
     }
      else
      /* 如果标识符列表解析完成之后不是闭括号，则报错 */
      c_parser_skip_until_found (parser, CPP_CLOSE_PAREN,"expected %<)%>"); pop_scope (); return ret;
    }
  /* 到这个else则代表当前不满足标识符列表的解析条件，则按照参数类型类表解析 */
  else
    {
      /*
         此函数用来解析当前的参数类型列表，并将解析出来的多个参数声明信息和类型信息链接到c_arg_info中返回,细节后续分析，
        这里仅先需要知道参数类型列表的解析结果也是通过一个c_arg_info结构体返回的即可.
      */
      struct c_arg_info *ret = c_parser_parms_list_declarator (parser, attrs, NULL);
      pop_scope ();
      return ret;
    }
}
```

到这里就是声明符解析的大体流程，实际上梳理起来还是比较清晰的:

```
产生式:
   declarator:
     pointer[opt] direct-declarator

   direct-declarator:
     identifier T4
     (attributes[opt] declarator) T4
```

```
 8.
 9.    T4:
10.      array-declarator T4
11.      ( parameter-type-list ) T4
12.      ( identifier-list[opt] ) T4
13.      empty
14.
15. c_parser_declaration_or_fndef    /* 声明或函数定义解析 */
16. {
17.     /* 解析声明说明符 */
18.     c_parser_declspecs (parser, specs, true, true, start_attr_ok, true, true, cla_nonabstract_decl);
19.     while(true) {
20.         /* 解析一个声明符 */
21.         declarator = c_parser_declarator (parser, specs->typespec_kind != ctsk_none, C_DTR_NORMAL, &dummy);
22.         if(不继续解析)
23.             break;
24.     }
25. }
26.
27. c_parser_declarator    /* 负责解析声明符(declarator) */
28.   => c_parser_direct_declarator  /* 负责解析直接声明符(direct-declarator) */
29.      => c_parser_direct_declarator_inner  /* 负责解析T4表达式 */
30.         => c_parser_parms_declarator          /* 负责解析参数类型列表(parameter-type-list)或标识符列表(identifier-list[opt]) */
```

这里没有展开的细节主要是参数类型列表是如何解析的(也就是 **c_parser_parms_list_declarator**函数的解析流程),此外此函数的分析还涉及到整个scope的流程,故下一个分析中先介绍scope的工作原理.