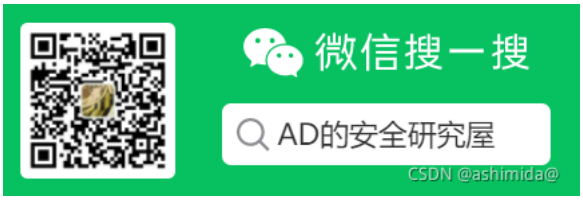
GCC源码分析(七) — 语法/语义分析之声明符解析(下)

更多内容可关注微信公众号



在<GCC源码分析(五) — 语法/语义分析之声明符解析(上)> 中已经大体介绍了声明符的解析流程，其中没有完成的一部分就是参数列表的解析，也就是下面产生式中的 parameter-type-list的解析:

```
1.    declarator:
2.      pointer[opt] direct-declarator
3.
4.    direct-declarator:
5.      identifier T4
6.      (attributes[opt] declarator) T4
7.
8.    T4:
9.      array-declarator T4
10.     ( parameter-type-list ) T4
11.     ( identifier-list[opt] ) T4
12.     empty
13.
14.   parameter-type-list:
15.     parameter-list
16.     parameter-list , ...
```

如前所述,在gcc中是通过c_parser_parms_list_declarator函数来解析parameter-type-list的产生式的,此函数如下:

```
1. static struct c_arg_info *
2. c_parser_parms_list_declarator (c_parser *parser, tree attrs, tree expr)
3. {
4.   /* 若参数列表解析直接遇到close paren则代表其中没有参数，如 int func();, 故直接返回一个内容为空的 c_arg_info结构体 */
5.   if (c_parser_next_token_is (parser, CPP_CLOSE_PAREN))
6.     {
7.       struct c_arg_info *ret = build_arg_info ();
8.       c_parser_consume_token (parser);
9.       return ret;
10.    }
11.  /* 若下一个符号为 ellipsis 即省略号 ...，则按照不定参数处理，...必须是此参数类型类表的最后一个参数(见产生式) */
12.  if (c_parser_next_token_is (parser, CPP_ELLIPSIS))
13.    {
14.      struct c_arg_info *ret = build_arg_info ();
15.      if (flag_allow_parameterless_variadic_functions)
16.      {
17.          ret->types = NULL_TREE;          /* 若允许F (...)的形式,则types链表直接设置为空(代表没参数列表)即可 */
18.      }
19.      else
20.      {
21.          ret->types = error_mark_node;    /* 不允许此形式则直接报错 */
22.          error_at (c_parser_peek_token (parser)->location, "ISO C requires a named argument before %<...%>");
23.      }
24.      c_parser_consume_token (parser);
25.      if (c_parser_next_token_is (parser, CPP_CLOSE_PAREN))    /* 如果... 后面直接是闭括号，则符合产生式，解析完毕直接返回一个内容为空的c_arg_info结构体
26.      {
27.          c_parser_consume_token (parser);
28.          return ret;
29.      } else {
30.          /* 否则属于语法错误，...后面是不能再加参数列表的，故消耗掉下一个闭括号之前的所有符号并报错 */
31.          c_parser_skip_until_found (parser, CPP_CLOSE_PAREN, "expected %<)%>");
32.          return NULL;
33.      }
34.    }
35.  /*
36.     这里循环处理paramter-list中的每个参数声明，每一次循环处理一个参数声明，这里会为参数声明生成声明树节点，并绑定到当前scope，
37.    当退出循环时(代表paramter-list解析完毕),会将当前scope的所有绑定信息记录到一个c_arg_info结构体中，并删除scope中的所有绑定
38.    最终返回的c_arg_info结构体就代表了参数列表解析的结果.
39.  */
40.  while (true)
41.    {
```

```
      /* 此函数用来解析一个参数声明，返回的c_parm结构体中记录了参数声明的 声明说明符,声明符，属性，位置信息(见下)  */
      struct c_parm *parm = c_parser_parameter_declaration (parser, attrs);
           .......
      /*
         根据parm中的信息，为当前参数声明生成声明节点(decl),并调用pushdecl函数将当前声明节点绑定到其标识符的对应binding队列和当前scope中
        pushdecl实际上是对bind函数的封装，在此函数的基础上做了一些参数检查(声明节点的构建函数之后分析).
      */
      push_parm_decl (parm, &expr);

      /* 若解析到 close paren ),则代表整个参数类型列表()解析完毕了 */
      if (c_parser_next_token_is (parser, CPP_CLOSE_PAREN))
      {
              ......
          /*
            这里对应前面的step 5)，将当前scope中所有的 PARM_DECL 信息链接为一个c_arg_info结构体并返回
           链接的内容包括PARM_DECL声明信息和其对应的类型信息。
            此scope中所有的 PARM_DECL 都通过其tree_node.chain链接起来，第一个PARM_DECL挂接到
           arg_info->parms中。
            此函数同时为此scope中每一个参数声明(PARM_DECL)生成一个tree_list节点，每个PARM_DECL
           的tree_list节点同样通过其chian链接起来，每个tree_list.value 指向此 PARM_DECL的类型节点
          */
        /*
           到这里代表解析完毕，则此函数将处理前面循环调用push_parm_decl在当前scope的所有绑定，此函数类似pop_scope会将当前scope中的所有c_binding都
          释放掉，但释放的同时会将所有c_binding的信息都记录到一个c_arg_info结构体中，并返回此结构体.
        */
        return get_parm_info (false, expr);
      }

      /* 除以上情况外，如果没有遇到 comma(逗号)，则代表语法错误，直接报错 如果遇到了逗号则c_parser_require会将其直接消耗掉 */
      if (!c_parser_require (parser, CPP_COMMA, "expected %<;%>, %<,%> or %<)%>", UNKNOWN_LOCATION, false))
      {
        /* 若没有遇到逗号，则直接消耗掉下一个闭括号之前的所有符号并报错 */
        c_parser_skip_until_found (parser, CPP_CLOSE_PAREN, NULL);
        return NULL;
      }

      /* 再遇到 ...，处理流程和上面类似，先省略 */
      if (c_parser_next_token_is (parser, CPP_ELLIPSIS)) ......

           /* 到这里代表遇到的是 逗号，则循环继续解析下一个参数声明 */
    }
}
```

其中函数c_parser_parameter_declaration定义如下:

```
/* 此函数是用来解析一个参数声明的 */
static struct c_parm *
c_parser_parameter_declaration (c_parser *parser, tree attrs)
{
  struct c_declspecs *specs;
  struct c_declarator *declarator;
    ......
  while (c_parser_next_token_is (parser, CPP_PRAGMA))        /* 发现编译制导符号(pragma)，则先解析编译制导符号,参数声明的开头是声明说明符 */
    c_parser_pragma (parser, pragma_param, NULL);

  if (!c_parser_next_token_starts_declspecs (parser))        /* 若当前第一个字符不是声明说明符，则不满足产生式,error返回 */
    {
          .....
      c_parser_error (parser,"expected declaration specifiers or %<...%>");
      c_parser_skip_to_end_of_parameter (parser);
      return NULL;
    }

  location_t start_loc = c_parser_peek_token (parser)->location;    /* 第一个字符是声明说明符,则获取声明说明符中第一个说明符的位置 */
  specs = build_null_declspecs ();    /* 创建一个新的声明说明符 */
  if (attrs)                          /* 若有额外的属性，先直接添加到声明说明符结构体中 */
    {
      declspecs_add_attrs (input_location, specs, attrs);
      attrs = NULL_TREE;
    }

  /* 调用声明说明符分析函数分析参数声明中的声明说明符(declaration-specifiers) */
  c_parser_declspecs (parser, specs, true, true, true, true, false, cla_nonabstract_decl);
  finish_declspecs (specs);        /* 同样分析完毕这里按需确定下是否需设置specs.type */

  prefix_attrs = specs->attrs;    /* 声明说明符中已有的属性算作参数声明的前缀属性 */
  specs->attrs = NULL_TREE;

  /* 递归声明符解析函数解析出一个声明符，这里传入了C_DTR_PARM代表解析参数声明，其和普通声明的区别在于参数声明中可以省略声明符 */
  declarator = c_parser_declarator (parser, specs->typespec_kind != ctsk_none, C_DTR_PARM, &dummy);

  /* 若没有解析到声明符，则跳过逗号之前的所有符号(因为参数类型列表是以逗号分隔的),并返回空(并不代表错误,如 int func(void);) */
  if (declarator == NULL)
    {
      c_parser_skip_until_found (parser, CPP_COMMA, NULL);
      return NULL;
    }

  if (c_parser_next_token_is_keyword (parser, RID_ATTRIBUTE))             /* 解析完声明符后尝试解析后缀属性 */
    postfix_attrs = c_parser_attributes (parser);
```

```
46.
47.   location_t end_loc = parser->last_token_location;        /* 记录分析过的最后一个token的源码位置 */
48.   /* 遍历整个声明符的所有inner的内容，找到其标识符节点；整个声明符是由多个c_declarator结构体构成的，但其最内层一定是本质的那个标识符，如 int * p; 最后一层的
49.   c_declarator *id_declarator = declarator;
50.   while (id_declarator && id_declarator->kind != cdk_id)
51.     id_declarator = id_declarator->declarator;
52.
53.   /* 如果此声明符有标识符，则记录标识符位置，如 int *p; 中p的位置；如果省略了标识符，则使用此参数声明中第一个符号的位置如 int func(char [10]);则使用char的
54.   location_t caret_loc = (id_declarator->u.id ? id_declarator->id_loc : start_loc);
55.   /* 最终返回位置信息，位置是 caret_loc，start/end信息作为附加信息也记录在其中 */
56.   location_t param_loc = make_location (caret_loc, start_loc, end_loc);
57.
58.   /*
59.     最终将声明说明符，属性，声明符(c_decalrator链表),位置信息记录到一个c_parm结构体返回， 此函数就单纯记录了四个指针，没其他内容，省略不写
60.     struct c_parm {
61.      struct c_declspecs *specs;
62.       tree attrs;
63.       struct c_declarator *declarator;
64.       location_t loc;
65.     };
66.   */
67.   return build_c_parm (specs, chainon (postfix_attrs, prefix_attrs), declarator, param_loc);
68. }
```

声明符的解析接口函数实际上就是c_parser_declarator,此函数最终返回一个 c_declarator结构体的指针的链表代表解析出的一个声明符.

和声明说明符的结果保存不同，**声明说明符解析的全部结果都记录在一个c_declspecs结构体中，而声明符返回的c_declarator实际上只是一个解析结果的第一个元素，c_declarator自身是一个链表，这个链表中每个元素都是一个c_declarator，每个元素均代表声明符的一部分。**如 int const * x; 中， int 和const分别是两个说明符，int const合在一起组成了声明说明符， 整个声明说明符中所有信息都记录在一个c_declspecs结构体中，也就是说 int 和const的信息都记录在此结构体中； 而这里的声明符(*x)实际上是由两个c声明符(c_declarator)构成的，第一个c声明符代表的是一个指针节点，第二个声明符代表的是标识符节点x.

c_declarator结构体的定义如下:

```
1. enum c_declarator_kind {
2.   cdk_id,           /* 代表普通c声明符 */
3.   cdk_function,     /* 代表函数c声明符 */
4.   cdk_array,        /* 代表数组c声明符 */
5.   cdk_pointer,      /* 代表指针c声明符 */
6.   cdk_attrs         /* 代表属性c声明符 */
7. };
8.
9. struct c_declarator {
10.   enum c_declarator_kind kind;              /* 记录当前声明符的类型 */
11.   location_t id_loc;                        /* 记录此c_declarator代表的符号(未必是标识符，如*)的源码位置 */
12.   /*
13.     c_declarator代表一个c声明符，如果一个c声明符有inner则innerc声明符的信息记录到这里(c_declarator链表就是通过此指针链接的)，如:
14.     * 若kind = cdk_id,则代表当前c声明符为一个普通c声明符，其declarator(inner)为空，如int x; ，见build_id_declarator
15.     * 若kind = cdk_function,则代表当前c声明符为一个函数c声明符,其declarator(inner)指向函数名的那个普通c声明符节点，如 int func(...); func(...)是一个函数
16.     * 若kind = cdk_array,则代表当前声明符为一个数组c声明符,其 declarator(inner)指向数组名的那个普通c声明符节点,如 int p[...]中的p,而...的内容记录在下面的
17.     * 若kind = cdk_pointer,则代表当前声明符为一个指针c声明符,其 decalrator(inner)指向除去指针(和属性)外，其内部的c声明符，如:
18.       - int *p; 则内部的声明符就是个普通c声明符 p;
19.       - int *func(...); 则其内部声明符就是个函数c声明符;
20.     * 若kind = cdk_attrs,则代表当前声明符为一个属性c声明符,其 decalrator(inner)指向内部的内容
21.   */
22.   struct c_declarator *declarator;
23.   /* 不同类型的c声明符则使用union联合体的不同部分记录其信息 */
24.   union {
25.     tree id;            /* 对于普通c声明符(kind=cdk_id),使用此字段记录普通c声明符的标识符节点(lang_identifier) */
26.     struct c_arg_info *arg_info;    /* 对于函数c声明符(kind=cdk_function),则此字段记录函数的参数类型列表或表示符列表信息(见后) */
27.
28.     struct {                        /* 对于数组c声明符(kind=cdk_array),此结构体记录数组相关信息 */
29.       tree dimen;                   /* The array dimension, or NULL for [] and [*].  */
30.       int quals;                    /* The qualifiers inside [].  */
31.       tree attrs;                   /* The attributes (currently ignored) inside [].  */
32.       BOOL_BITFIELD static_p : 1;   /* Whether [static] was used.  */
33.       BOOL_BITFIELD vla_unspec_p : 1;    /* Whether [*] was used.  */
34.     } array;
35.     int pointer_quals;              /* 对于指针c声明符(kind=cdk_pointer),此结构体记录指针和后续c声明符之间的类型限定符的内容(若有)  */
36.     tree attrs;                     /* 对于属性c声明符(kind=cdk_attrs)，这里记录一个属性链表 */
37.   } u;
38. };
```

由定义也可以看出gcc中的c声明符一共有5中类型，产生式中的声明符(declarator)就是由这5种类型的c声明符组合(链接)起来的, 在声明符解析过程中解析到不同类型的c声明符时也会调用不同的函数分别构建这5种c声明符之一，故一共有5个c声明符的构建函数:

```
1.    * build_id_declarator
2.    * build_function_declarator
3.    * build_array_declarator + set_array_declarator_inner
4.    * make_pointer_declarator
5.    * build_attrs_declarator
```

而函数get_parm_info的定义如下：

```
struct c_arg_info *
get_parm_info (bool ellipsis, tree expr)
{
  struct c_binding *b = current_scope->bindings;      /* 获取当前scope上所有的符号绑定 */
  struct c_arg_info *arg_info = build_arg_info ();    /* 分配并初始化一个c_arg_info结构体保存返回结果 */
  tree parms = NULL_TREE;                             /* parms 是一个链表，用来链接当前scope上所有绑定的decl节点 */
  vec<c_arg_tag, va_gc> *tags = NULL;
  tree types = NULL_TREE;                             /* types也是一个链表，用来链接当前scope上所有绑定的decl的类型节点 */
  tree others = NULL_TREE;

  current_scope->bindings = 0;  /* 当前scope的所有binding都被摘走了，后续scope消除时不必再消除这些binding了(当前函数处理完就顺便消除了) */

  /* c_parser_parms_list_declarator调用到 get_parm_info 说明参数列表中是一定有参数声明，则current_scope一定有符号绑定,所以这里会assert*/
  gcc_assert (b);

  /* b是当前scope绑定的符号链表，参数类型类表中所有参数声明都在push_parm_decl 函数中创建并绑定到了此列表中 */
  if (b->prev == 0 && TREE_CODE (b->decl) == PARM_DEC && !DECL_NAME (b->decl) && VOID_TYPE_P (TREE_TYPE (b->decl)))
    {
      /* 如果是就void一个绑定信息，也就是 int func(void);的形式 */
          ......
      /* 则这里只设置types为 void_list_node 返回此arg_info节点即可 */
      arg_info->types = void_list_node;
      return arg_info;
    }
    ......
  while (b)         /* 遍历当前scope上所有的声明绑定信息,将相关信息链接到arg_info结构体中，并删除此binding节点 */
    {
      tree decl = b->decl;    /* 此声明绑定上绑定的声明节点 */
      tree type = TREE_TYPE (decl);    /* 此声明的类型节点 */
      c_arg_tag tag;
      const char *keyword;
      /* 判断第一个参数的声明树的 TREE_CODE,对于参数声明来说，其TREE_CODE都是PARM_DECL，类型就是参数声明中的类型 */
      switch (TREE_CODE (decl))
    {
    case PARM_DECL:    /* 如果此声明是一个参数声明 */
      if (b->id)       /* 若声明绑定了标识符，则从标识符节点的绑定链表中删除此绑定 */
        {
          gcc_assert (I_SYMBOL_BINDING (b->id) == b);
          I_SYMBOL_BINDING (b->id) = b->shadowed;
        }
              ......
          /* 将当前声明链接到parms链表中,声明链接顺序已经变为其在代码中出现顺序了(负负得正) */
          DECL_CHAIN (decl) = parms;
          parms = decl;
          /* DECL_ARG_TYPE 保存的应该是调用时的实参类型，这里先初始化为形参类型,后面碰到具体调用时应该会改？？？   */
          DECL_ARG_TYPE (decl) = type;
          /* 创建tree_list节点，将所有的类型节点串联起来(类型节点是共用的，这里代表一种关系，故必须新建tree_list串联) */
          types = tree_cons (0, type, types);
        }
      break;

    case ENUMERAL_TYPE: keyword = "enum"; goto tag;
    case UNION_TYPE:    keyword = "union"; goto tag;
    case RECORD_TYPE:   keyword = "struct"; goto tag;
    tag:
      if (b->id)    /* 同样结构体之类的也是要去除绑定 */
        {
          gcc_assert (I_TAG_BINDING (b->id) == b);
          I_TAG_BINDING (b->id) = b->shadowed;
        }

      tag.id = b->id;
      tag.type = decl;
      vec_safe_push (tags, tag);    /* 所有结构体和联合体的绑定信息都放到tags中 */
      break;

    case FUNCTION_DECL:
        ......
        goto set_shadowed;
    case CONST_DECL:
    case TYPE_DECL:
       ......
      gcc_assert (!b->nested);
      DECL_CHAIN (decl) = others;    /* 常量声明和类型声明链接到others链表 */
      others = decl;
      /* fall through */
    case ERROR_MARK:
    set_shadowed:
      if (b->id)                    /* 同样这些声明节点也要去绑定 */
        {
          gcc_assert (I_SYMBOL_BINDING (b->id) == b);
          I_SYMBOL_BINDING (b->id) = b->shadowed;
        }
      break;
    case LABEL_DECL:
    case VAR_DECL:
    default:
      gcc_unreachable ();
    }
      /* 释放此binding节点到binding_freelist，并使用当前scope上的下一个binding节点 */
      b = free_binding_and_advance (b);
    }
```

```
  /* 将上述所有信息记录到arg_info结构体中并返回 */
  arg_info->parms = parms;
  arg_info->tags = tags;
  arg_info->types = types;
  arg_info->others = others;
  arg_info->pending_sizes = expr;
  return arg_info;
}
```

其中struct c_arg_info结构体的定义如下:

```
struct c_arg_info {
  tree parms;            /* 参数类型列表中，所有参数声明节点都连接到此成员变量上；标识符列表中为空*/
  vec<c_arg_tag, va_gc> *tags;    /* 参数类型列表中，所有结构体等节点都链接到此成员变量上；标识符列表中为空 */
  /*
     对于参数类型列表还是标识符列表，这里都是一个tree_list的链表:
     * 对于参数类型列表，tree_list链接所有参数声明的类型节点(或者为void_list_node)
     * 对于标识符列表,tree_list链接所有的标识符节点
  */
  tree types;
  tree others;            /* 参数类型列表中的常量声明和类型声明链接到others */
  tree pending_sizes;
  BOOL_BITFIELD had_vla_unspec : 1;
};
```

函数的参数信息分为两种:

- 如果函数c声明符中解析出的是标识符列表，如 func(x,y),那么 arg_info结构体记录的相当于是实参，其只通过types字段中的tree_list链表记录各个标识符的树节点
- 如果函数c声明符中解析出的是参数类型列表，如int func(int x, int y),那么arg_info结构体记录的相当于是形参，其:
    - parms字段: 记录了参数类型列表中各个参数的声明(decl)节点的链表(按照代码顺序)
    - types字段: 记录了参数类型列表中各个参数声明中的类型节点的链表(按照代码顺序，若没有声明则为void_list_node)
    - tags字段: 记录了参数类型列表中所有结构体,联合体,枚举类型的定义
    - others字段: 记录了参数类型列表中定义的常量声明和类型声明

在解析函数参数列表的过程中，**参数声明解析完毕也就意味着其对应的声明节点已经创建成功了(内部的声明走完了完整的声明解析流程)，此过程是递归的，也同样可以简单描述为: 解析声明说明符 => 解析声明符 => 创建声明树节点 => 将所有声明树节点构建为c_arg_info结构体返回。**