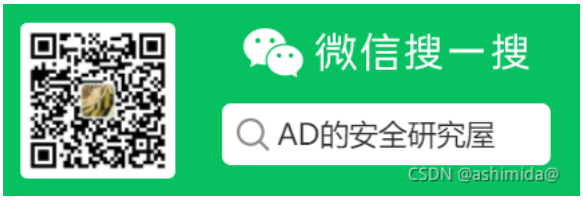
GCC源码分析(六) — 符号绑定,作用域与block树节点



gcc中的符号绑定(c_binding)作用于(scope)和block树节点，根据c语言语法可知在不同的作用于中是可以对标识符重新声明的:

```
1. int x;
2. void fun()
3. {
4.     int x;
5. }
```

如上的定义中函数func内部的x和全局变量x是两个不同的声明，但在gcc中二者的标识符实际上都是x. 在gcc源码中相同字符串的标识符节点是全局唯一的(只有一个), 因为仅仅在词法和语法层面是没法区分出相同字符串的不同含义的.

而在解析到源码不同位置时则需要区分出一个标识符到底代表的哪一个声明, **符号绑定和作用域的作用就是用来区分同一个标识符在不同位置对应到哪个具体声明的，而block树节点则用来保存解析的结果。**

在前面已知gcc中通过一个tree_identifier结构体来代表一个标识符的树节点，但实际分配时会为标识符分配一个扩展的lang_identifier节点，其结构如下:

```
1. struct   lang_identifier {
2.   struct c_common_identifier common_id;               //兼容tree_identifier
3.   struct c_binding *symbol_binding; /* vars, funcs, constants, typedefs */
4.   struct c_binding *tag_binding;    /* struct/union/enum tags */
5.   struct c_binding *label_binding;  /* labels */
6. };
```

lang_identifier结构体除了记录标识符的字符串、长度、hash。链表等信息外(tree_identifier本身的作用)还多了三个c_binding指针，这3个指针分别用来将此标识符绑定三种不同类型的声明(符号、tag、label),每个c_binding指针都是一个链表，其按照标识符声明出现的逆序记录在源码解析过程中同名的所有声明节点.

如上面例子中解析到func中的 int x, 那么在标识符x的symbol_binding中就会顺序链接两个绑定信息(一个c_binding结构体只记录一个标识符和一个声明的绑定关系):

- 第一个绑定信息绑定的是函数内x的声明(*symbol_binding)
- 第二个绑定信息绑定的是全局变量x的声明(*symbol_binding->shadowed)

而对标识符x之所以会有三个c_binding链表是因为c语言允许同一个标识符的符号,tag(如结构体)和标签共存，故如下代码是正确的:

```
1. void main()
2. {
3.     int x;                    //符号
4.     struct x{};               //结构体(tag)
5. x:                            //标签(label)
6.     return;
7. }
```

记录标识符与声明的绑定的结构体c_binding定义如下:

```
1. /*  一个c_binding结构体的主要作用就是记录一个标识符节点和一个声明/类型节点的对应关系 */
2. struct   c_binding {
3.   union   {        /* first so GTY desc can use decl */
4.     tree   type; /* the type in this scope */
5.     struct c_label_vars *   label; /* for warnings */
6.   }   u;
7.     /*  当前标识符绑定的节点，对于普通符号和标签，这里是对应的声明节点(_DECL),对于对于枚举/结构体/联合体则是对应的 UNION_TYPE/ENUMERAL_TYPE/RECORD_TYPE节
8.   tree decl;            /* the decl bound */
9.   /* 当前声明/类型节点绑定到的标识符节点 */
10.  tree id;            /* the identifier it's bound to */
11.  /* 指向这个scope中的下一个声明的c_binding，同一个scope中的c_binding要链接起来，这样在解析完当前scope后才可以批量释放以避免副作用 */
12.  struct c_binding *prev;    /* the previous decl in this scope */
13.  /*
14.     同名的标识符，只有在最内层的才有效，这里的shadowed是c_binding用来链接同名 标识符声明的c_binding的,
15.    shadowed链接的是其外层声明的c_binding,因为外层在当前不可见，所以叫做shadowed.
16.  */
```

```
  struct c_binding *shadowed;    /* the innermost decl shadowed by this one */
  /* 当前c_binding 的深度，c_binding的深度就是其绑定到的那个c_scope的深度，这样在标识符节点中可通过深度来确定此c_binding绑定的声明是属于哪个scope的 */
  unsigned int depth : 28;      /* depth of this scope */
  BOOL_BITFIELD invisible : 1;  /* normal lookup should ignore this binding */
  BOOL_BITFIELD nested : 1;     /* do not set DECL_CONTEXT when popping */
  BOOL_BITFIELD inner_comp : 1; /* incomplete array completed in inner scope */
  BOOL_BITFIELD in_struct : 1;    /* currently defined as struct field */
  /* 此声明的源码位置 */
  location_t locus;        /* location for nested bindings */
};
```

一个c_binding结构体的主要作用就是记录一个标识符节点和一个声明/类型节点的对应关系，在此之上其有三个成员 prev/shadowed/depth都与scope有关。

在源码解析中我们需要知道关于标识符的信息是：在当前作用于中此标识符到底代表了哪个声明，而标识符的绑定(c_binding)只能给出标识符定义的先后顺序关系，并不能体现出在哪个作用域中应该使用哪个声明这一要求，**所以在gcc中引入了一个scope的概念用来区分不同的作用域。**作用于之间是相互包含的，简单举例一个{}包裹的内容通常就是一个作用域，一个作用域内部可以由多个子作用域；一个作用域只能有一个父作用域；子作用域比父作用域深度+1; 如:

```
int func()
{                   //scope1, 深度为x
    {               //scope2,为 scope1的子作用域,深度为x+1
        int x;
    }
    {               //scope3,为 scope1的另一个子作用域, 深度为 x+1
        int y;
    }
}
```

在gcc编译期间，全局变量记录了以下几个作用域:

```
/* 当前作用域，代表当前源码解析到哪个作用域了，如解析到上面的int x,则在 scope1， 解析到int y则在 scope 2 */
static   struct c_scope *current_scope;
/* 当前正在编译的函数所在的作用域，若当前在一个函数内解析，则通过此全局变量可以快速确定当前函数在哪个作用域 */
static   struct c_scope *current_function_scope;
/* 正常编译单元所在的作用于，又叫做文件作用域，正常全局声明都是处于此作用域中，其depth = 1 */
static   struct c_scope *file_scope;
/* 外部声明的作用于，全局最外层的作用于， 其depth = 0 */
static   struct c_scope *external_scope;
```

在gcc中作用域是通过一个c_scope结构体表示的，其定义如下:

```
struct   c_scope {
  /* 记录当前scope的外层scope的指针 */
  struct c_scope *outer;
    /*
         若当前scope是一个函数的scope，那么这里记录其上层函数(若有)的函数scope的指针，这里若有值则代表是内层函数嵌套的解析(在gcc中好像是不允许函数嵌套定义的
         对于非函数scope，这里为空. 解析到函数 int func(...){} 中的大括号{时，当前新生成的scope就是函数的scope(也可以成为函数体的scope).
    */
  struct c_scope *outer_function;
  /*
     记录当前scope内，所有声明/类型和符号的绑定，在一个scope内解析到的所有符号，tag,label，都要为其生成一个c_binding结构体，通过c_binding->shadowed连接到
     通过 c_binding->prev连接到其所在的scope
  */
  struct c_binding *bindings;
  /*
     一个scope可能有多个子scope,这多个子scope都是顺序解析的，每个子scope解析完成之后,其scope本身不会被保留，
    而是将所有内容记录到一个block中,当前已经解析过的子scope的block信息就记录到这个blocks链表中.
  */
  tree blocks;
  tree blocks_last;
  /* 一个depth代表一个嵌套，由于编译单元是单线程的，在gcc编译过程中应该不会出现两个相同depth的scope同时存在,depth值越大代表约往内层. */
  unsigned int depth : 28;
  /* 在解析参数类型列表时，会设置此flag，代表当前scope用作参数列表的解析，如解析 int func(...),解析到...时就会设置此flags. */
  BOOL_BITFIELD parm_flag : 1;
  BOOL_BITFIELD had_vla_unspec : 1;
  BOOL_BITFIELD warned_forward_parm_decls : 1;
  /* 设置了此flag代表此scope是一个函数(block)的最外层的scope.如 int func(...) { ... }; 此scope就是{}内元素的block */
  BOOL_BITFIELD function_body : 1;
  /* 代表总是要为当前scope构建一个block */
  BOOL_BITFIELD keep : 1;
  BOOL_BITFIELD float_const_decimal64 : 1;
  /* 若当前scope有标签的binding则这里标记为true */
  BOOL_BITFIELD has_label_bindings : 1;
  BOOL_BITFIELD has_jump_unsafe_decl : 1;
};
```

关于scope,c_binding,标识符和声明之间关系总结如下:

1. 一个scope代表一个作用域，标识符在不同作用域中可以拥有不同的声明，这些声明会通过一个c_binding结构体与标识符绑定，并通过c_binding->shadowed字段链接到当前标识符的3个绑定链表之一中，同时通过c_binding->prev字段连接到当前scope->binding链表中(以便于后续释放)

2. 每个scope都有其不同的深度，整个编译期间最外层的scope为external_scope(depth = 0),第二层为file_scope(depth = 1),之后是代码中的各个scope
3. 解析到每个声明/类型定义时，都要为其绑定标识符(创建c_binding)，此绑定也是有深度的，其深度就是当前声明所在的scope的深度
4. 在源码解析使用到一个符号时，会从符号的绑定链表中根据深度查找当前scope及当前scope外层的符号绑定，>=当前符号深度的深度最小的绑定中的声明，即为当前符号的声明.
5. 在一个scope释放时，会释放当前scope内生成的所有绑定节点，同时在被绑定的标识符节点也删除此绑定信息，并最终将所有此scope内的声明节点和子scope的信息(已经记录到block节点中了)记录到一个block节点并返回，这样可以确保声明除了当前scope则无效。

前面的符号绑定和作用域机制保证了在代码中可以正确识别标识符的作用域，**但最终遗留的问题就是，当一个scope解析完毕后当前scope内的信息要保存到哪里，在gcc中的tree_block节点就是用来保存一个scope解析完毕后的信息的**，其定义如下:

```
struct tree_block {
  struct tree_base base;
  /* 连接一个scope中的所有同level的 block(也就是说所有的子block) */
  tree chain;
  unsigned block_num;
  location_t locus;
  location_t end_locus;
  /* vars指向当前scope中的第一个decl节点，此decl->chain又会链接此scope中下一个decl节点,以此类推，这里链接了当前block内部的所有声明(以其在源码中的出现顺序)
  tree vars;
  vec<tree, va_gc> *nonlocalized_vars;
  /* 指向当前block中的第一个子block，一个block中所有的子block都是通过block->chain链接的，通过第一个子block可以遍历所有子block. */
  tree subblocks;
  /* 记录当前block的父block，对于函数最外层的block,指向函数的FUNCTION_DECL节点;多层block包裹的情况则指向上层block;file_scope中则指向编译单元的TRANSLATIO
  tree supercontext;
  tree abstract_origin;
  tree fragment_origin;
  tree fragment_chain;
  /* Pointer to the DWARF lexical block.  */
  struct die_struct *die;
};
```

一个scope在解析过程中会依赖于其所能看到的所有标识符的绑定(代表其声明)，其中此scope内层的标识符绑定只在当前scope中有效，故在当前scope解析完毕后当前scope内在标识符上的绑定应该被删除，同时当前scope内生成的所有声明以及子scope的信息均会被保存到一个tree_block节点并返回，其中:

1. tree_block->vars顺序链接当前scope内所有的声明节点
2. tree_block->subblocks顺序链接当前scope内所有子scope生成的block节点

一个scope是在销毁的时候才最终生成了block，生成了block也同时表示此scope中所有的绑定已不再可用,此scope中的内容已经解析完毕.

作用域相关的代码可以分为4部分,分别是作用域的初始化,作用域的生成(push_scope),作用域的释放(pop_scope)与作用域内的符号绑定(bind),代码分析如下:

一、作用域的初始化

作用域的初始化发生在真正编译之前，其初始化全局的external_scope和file_scope两个作用域,这两个是整个编译单元最外层的scope，其depth分别为0和1，初始化流程为:

```
toplev::main
  => do_compile
    => lang_dependent_init
      => c_objc_common_init
        => c_init_decl_processing
          => push_scope();                        //创建程序中的第一个scope(默认记录到current_scope中)
          => external_scope = current_scope;      //将最先创建的整个scope作为external_scope
    => compile_file ();
      => c_common_parse_file
        => push_file_scope                        //创建程序中第二个scope作为file_scope，并将所有的buildin的函数声明绑定到此scope中
          => push_scope ();                       //创建程序中第二个scope
          => file_scope = current_scope;          //第二个scope作为file_scope
          => for (decl = visible_builtins; decl; decl = DECL_CHAIN (decl))
                bind (DECL_NAME (decl), ......);  //将所有buildin的函数都绑定到file_scope中，builtin函数是gcc内置的，在gcc中可以调用函数add_builtin_fu
                                                  //一个buildin函数(add_builtin_function => lang_hooks.builtin_function => c_builtin_function)
        => c_parse_file ();                       //开始解析编译单元
        => pop_file_scope ();                     //解析完毕后 pop file_scope
```

可见在真正的代码解析(c_parse_file)之前作用域的初始化实际上只是创建了两层的scope(push_scope中会自动增加depth),并将其记录到全局变量中.

二、push_scope

push_scope实际上只是新创建了一个scope结构体并初始化为下一层的深度，然后将其放到全局指针current_scope中(符号绑定时都是向current_scope中绑定的); 对于函数参数和函数体则是个例外，二者会共用同一个scope:

```
void push_scope (void)
{
  /*
     若当前是因为解析到函数体的{而导致执行的push_scope,那么全局变量next_is_function_body会暂时为1,push_scope过程中将其重新设置为0,
    其含义是当前函数参数和函数体共用scope，因为参数和函数体可以看做是同一个作用域的.
  */
  if (next_is_function_body)
    {
      current_scope->parm_flag         = false;       /* 标记当前scope已经不再用于参数解析了，参数解析已经完成了，目前在解析函数体了 */
      current_scope->function_body     = true;        /* 代表此scope是一个函数体的最外层scope */
      current_scope->keep              = true;        /* 对于函数来说，分析完成后是要默认构建block树节点的 */
      /* 当前函数体的scope(因为当前不新创建scope了,current就是当前函数体的),就是当前的全局函数scope */
      current_scope->outer_function    = current_function_scope;
      current_function_scope           = current_scope;
      keep_next_level_flag = false;     /* 下一个level是否强制生成block的全局变量，清零*/
      next_is_function_body = false;    /* 后续解析是否为函数体的全局变量，清零*/

      if (current_scope->outer)
        current_scope->float_const_decimal64 = current_scope->outer->float_const_decimal64;
      else
        current_scope->float_const_decimal64 = false;
    }
  else
    {
      /* 若scope_freelist中有可重用的scope结构体，就复用一个，否则重新分配 */
      struct c_scope *scope;
      if (scope_freelist)
      {
          scope = scope_freelist;
          scope_freelist = scope->outer;
      }
      else
        scope = ggc_cleared_alloc<c_scope> ();    /* 分配一个新的scope作为新的scope */

      if (current_scope)
        scope->float_const_decimal64 = current_scope->float_const_decimal64;
      else
        scope->float_const_decimal64 = false;

      scope->keep          = keep_next_level_flag;    /* 记录上层传下来的是否无条件 make a block节点 */
      /* 新scope的外层scope是 curent_scope，深度是 current_scope + 1 */
      scope->outer         = current_scope;
      scope->depth       = current_scope ? (current_scope->depth + 1) : 0;

      if (current_scope && scope->depth == 0)    /* 这是检查scope溢出 */
      {
          scope->depth--;
          sorry ("GCC supports only %u nested scopes", scope->depth);
      }
     /* 当前scope指向新分配的scope，也就是最内层的scope */
      current_scope        = scope;
      keep_next_level_flag = false;
    }
}
```

三、bind

```
/*
    name: 要绑定的标识符节点
    decl: 要绑定的声明或类型节点
    scope: 要绑定到哪个作用域中
    locus: decl的源码位置
  此函数负责将一个声明绑定到其标识符(lang_identifier)的三个绑定队列之一中，根据声明的类型不同，会绑定到不同的队列中.
  scope参数提供了绑定的灵活性，使得此函数可以向任意一个scope中绑定声明，最终此绑定会根据scope的深度链接到标识符绑定队列的对应位置。
  当前scope会记录自身的所有绑定，在scope释放时需要释放所有绑定，以消除子作用域对外层的影响.
*/
static void bind (tree name, tree decl, struct c_scope *scope, bool invisible, bool nested, location_t locus)
{
  struct c_binding *b, **here;
  /* binding_freelist是可重用的c_binding结构体链表，若其中有可用元素则直接拿一个可用元素来用 */
  if (binding_freelist)
    {
      b = binding_freelist;
      binding_freelist = b->prev;
    }
  else
    /* 否则分配一个c_binding结构体，用来绑定一个声明和一个标识符 */
    b = ggc_alloc<c_binding> ();

  b->shadowed = 0;              /* 此绑定后面才具体绑定到标识符，这里先初始化为0 */
  b->decl = decl;               /* 记录此标识符当前要绑定到哪个声明 */
  b->id = name;                 /* 记录此c_binding结构体为哪个标识符绑定声明 */
  b->depth = scope->depth;      /* 当前scope的深度，也就是此绑定的深度 */
  b->invisible = invisible;     /* normal lookup时此声明是否可见 */
  b->nested = nested;
  b->inner_comp = 0;
  b->in_struct = 0;
  b->locus = locus;             /* 记录声明的源码位置 */
```

```
  b->u.type = NULL;

  /* 整个c_binding 结构体会被绑定到scope的bindings队里上,后续scope释放时好一并释放 */
  b->prev = scope->bindings;
  scope->bindings = b;

  /* 给goto语句用的，先pass */
  if (decl_jump_unsafe (decl))
    scope->has_jump_unsafe_decl = 1;

  /* 若标识符节点为空，则只需要将当前的 c_binding 放到scope中就可以返回了,这种情况可能发生在如参数声明中， int func(char [10]); 中的char [10];  */
  if (!name)
    return;

  /* 若有标识符节点，则根据声明类型的不同，绑定到标识符的不同c_binding链表中，绑定到链表最优先被获取的一侧,相当于代码中的最近作用域的定义优先. */
  switch (TREE_CODE (decl))
    {
    /* 对于标签定义来说，使用 label_binding 字段; 对于结构体,枚举,union，则使用 tag_binding 字段; 其他常量,变量，参数，函数，类型定义，则都是用 symbol_bi
    case LABEL_DECL:     here = &I_LABEL_BINDING (name);   break;
    case ENUMERAL_TYPE:
    case UNION_TYPE:
    case RECORD_TYPE:    here = &I_TAG_BINDING (name);     break;
    case VAR_DECL:
    case FUNCTION_DECL:
    case TYPE_DECL:
    case CONST_DECL:
    case PARM_DECL:
    case ERROR_MARK:     here = &I_SYMBOL_BINDING (name);  break;
    default:
      gcc_unreachable ();
    }
  /*
     标识符的c_binding的顺序应该是按照深度从大到小的，这里要根据当前scope的深度，将此c_binding插入到标识符绑定队列的相应位置
    这里没有直接插入到最后一个，是提供了灵活性，当前scope传入的可以不是current_scope
  */
  while (*here && (*here)->depth > scope->depth)
    here = &(*here)->shadowed;
  b->shadowed = *here;
  *here = b;
}
```

四、pop_scope

pop_scope不仅要将current_scope从scope链表中销毁，同时还要消除掉current_scope对整个符号绑定的副作用，也就是current_scope中所有在标识符上的绑定都要被清除，而current_scope范围内创建的声明节点则被记录到一个block结构体中保存(block->vars), 此block结构体可能记录到函数或全局编译单元声明的initial指针中或记录到其父scope的blocks链表中等待父block销毁时一并处理。

```
/* 函数返回最终创建的blocks树节点 */
tree pop_scope (void)
{
  struct c_scope *scope = current_scope;    /* pop_scope要pop出的一定是当前的scope */
  tree block, context, p;
  struct c_binding *b;

  bool functionbody = scope->function_body; /* 记录当期scope是否为一个函数体的scope */
  /* 如果当前是一个函数体的scope,或当前scope上只要有声明的绑定,或之前流程中指定了要keep,那么后面在消除当前scope时都要为其创建block */
  bool keep = functionbody || scope->keep || scope->bindings;
  /* 标签绑定的处理，先pass */
  update_label_decls (scope);

  block = NULL_TREE;
  if (keep)
    {
      block = make_node (BLOCK);    /* 构造一个 新的tree_block 树节点 */
      /*
         当前scope的子scope(若有)解析完毕后同样会走 pop_scope流程，其同样会生成一个block节点，当前scope的所有子scope销毁后生成的block节点都链接在scope->b
        这里将当前所有子scope解析结果的block链表记录到当前scope生成的block->subblocks链表中.
      */
      BLOCK_SUBBLOCKS (block) = scope->blocks;

      TREE_USED (block) = 1;    /* 标记当前block中存在被使用了的变量 */

      /* 遍历当前scope所有子scope的blocks链表，设置其父block为当前block(之前肯定没设置过，只有在当前scope也释放时才创建的父block) */
      for (p = scope->blocks; p; p = BLOCK_CHAIN (p))
        BLOCK_SUPERCONTEXT (p) = block;

      BLOCK_VARS (block) = NULL_TREE;     /* 当前block的 变量节点先设置为空，后面会以链表的形式记录当前scope中所有声明 */
    }

  /* 先记录当前scope的上下文节点，若当前scope是一个函数体{}的block，那么context就指向这个函数的声明节点作为上下文 */
  if (scope->function_body)
    context = current_function_decl;
  else if (scope == file_scope)
    {
      /* 若当前已经到了file_scope，则构建一个file_decl(TRANSLATION_UNIT_DECL)作为context,此decl中记录了当前文件名 */
      tree file_decl = build_translation_unit_decl (get_identifier (main_input_filename));
      context = file_decl;
      debug_hooks->register_main_translation_unit (file_decl);
```

```
    }
  else
    /* 否则context 就是当前新创建的这个block自身(或者为空) */
    context = block;
  /*
     遍历当前scope中的所有bind函数绑定的c_binding结构体，这些c_binding节点最终会被释放到binding_freelist并清空内容，但在释放
    前此循环中会分情况将c_binding中的声明节点等信息记录到block->vars中，部分情景下还会设置这些节点的上下文(但凡是bindings中有内容，当前scope一定是有block的
  */
  for (b = scope->bindings; b; b = free_binding_and_advance (b))
    {
      p = b->decl;        /* 获取当前c_binding 上的声明节点 */
      switch (TREE_CODE (p))
    {
    case LABEL_DECL:      /* 若此c_binding是一个标签声明的绑定 */
        ......
      DECL_CHAIN (p) = BLOCK_VARS (block);          /* 将此label的声明记录到block->vars声明链表中 */
      BLOCK_VARS (block) = p;
      gcc_assert (I_LABEL_BINDING (b->id) == b);    /* 标签是会绑定到某个标识符的，这个标识符上的最近的标签绑定也必须就是此绑定 */
      I_LABEL_BINDING (b->id) = b->shadowed;        /* 在标识符的标签绑定链表中中删除此标签的绑定(清空scope则清空其所有绑定的副作用) */
      /* 释放标签绑定中标签独有的数据结构,先pass */
      release_tree_vector (b->u.label->decls_in_scope);
      b->u.label = b->u.label->shadowed;
      break;
    case ENUMERAL_TYPE:   /* 对于结构体,联合体,枚举类型 */
    case UNION_TYPE:
    case RECORD_TYPE:
      /* 每个类型与类型变种上都要设置其上下文，也就是使用范围,对于标签没有设置使用范围，可能是因为标签默认只在当前scope中有效？？ */
      set_type_context (p, context);
      if (b->id)    /* 同样检查后解除结构体声明节点的绑定 */
        {
          gcc_assert (I_TAG_BINDING (b->id) == b);
          I_TAG_BINDING (b->id) = b->shadowed;
        }
      break;
    /* 以下声明的绑定，都属于符号绑定，若标识符绑定了函数声明或变量声明，则有额外的检查,最终都执行到common_symbol流程释放标识符的符号绑定 */
    case FUNCTION_DECL:
      ......
      goto common_symbol;
    case VAR_DECL:
      ......
    case TYPE_DECL:
    case CONST_DECL:
    common_symbol:
      if (!b->nested)    /* 所有符号绑定最后都在这里处理 */
        {
          DECL_CHAIN (p) = BLOCK_VARS (block);    /* 同样将声明节点记录到当前的block->vars链表中 */
          BLOCK_VARS (block) = p;
        }
      else if (VAR_OR_FUNCTION_DECL_P (p) && scope != file_scope)
        {
          /* 好像除了内联函数外还没看到嵌套定义的，这里先忽略嵌套定义的*/
                ......
        }

      if (scope == file_scope)
        {
          /* 若当前是文件scope，设置所有声明的上下文为 TRANSLATION_UNIT_DECL 节点 */
          DECL_CONTEXT (p) = context;
          /* 若当前是类型声明节点，那么其所有变种节点的作用于和当前类型节点(其主变种)的作用于是完全相同的 */
          if (TREE_CODE (p) == TYPE_DECL
          && TREE_TYPE (p) != error_mark_node)
        set_type_context (TREE_TYPE (p), context);
        }
      gcc_fallthrough ();
    case PARM_DECL:
    case ERROR_MARK:
      if (b->id)    /* 同样最终释放标识符上的符号绑定 */
        {
          gcc_assert (I_SYMBOL_BINDING (b->id) == b);
          I_SYMBOL_BINDING (b->id) = b->shadowed;
          if (b->shadowed && b->shadowed->u.type)
        TREE_TYPE (b->shadowed->decl) = b->shadowed->u.type;
        }
      break;

    default:
      gcc_unreachable ();
    }
    }

  /* 当前新生成的整个block总是要有地方记录的，要么就是记录到函数声明或编译单元声明的树节点的inital中,要么就是记录到其父block节点中. */
  if ((scope->function_body || scope == file_scope) && context)
    {
      /* 对于函数来说，block记录到函数声明decl的initial节点；对于文件scope来说，block记录到TRANSLATION_UNIT_DECL节点的initial节点 */
      DECL_INITIAL (context) = block;
      BLOCK_SUPERCONTEXT (block) = context;    /* 同时这两个节点也就是当前block的父 block了 */
    }
  else if (scope->outer)    /* else的情况是要记录到其父scope 的blocks中，随着父scope的销毁，跟随到一个新的blocks中 */
    {
      if (block)    /* 如果此scope生成了一个block，则将其block添加到外层scope的blocks链表中 */
        SCOPE_LIST_APPEND (scope->outer, blocks, block);
      else if (scope->blocks)    /* 如果当前scope没有生成新的block，则将当前scope的blocks链表合并到其父scope的blocks链表中 */
        SCOPE_LIST_CONCAT (scope->outer, blocks, scope, blocks);
    }
```

```
136.
137.   current_scope = scope->outer;     /* 当期scope变为上一层scope */
138.   if (scope->function_body)         /* 如果当前是函数scope，则新的函数scope也变为上一层 */
139.     current_function_scope = scope->outer_function;
140.
141.   memset (scope, 0, sizeof (struct c_scope));    /* 不再使用的这个scope清空，并释放到 scope_freelist中 */
142.   scope->outer = scope_freelist;
143.   scope_freelist = scope;
144.
145.   return block;                     /* 返回新创建的block  (tree_block)*/
146. }
```