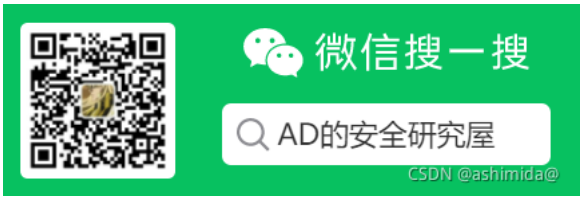
GCC源码分析(十一) — 函数节点的gimple低端化



一、gimple低端化

在gimple高端化时已经说过,整个编译单元所有外部声明的AST树节点生成完毕后会遍历符号表中的所有符号节点进行分析,而对于函数节点最终会执行到 cgraph_node::analyze对其进行gimple高端化和gimple低端化分析,**gimple高端化的分析是通过函数gimplify_function_tree(decl)来完成的**,而同样是在此流程中,**gimple低端化则是通过一系列pass的执行完成的**,代码如下:

```
1. void cgraph_node::analyze (void)
2. {
3.   tree decl = this->decl; /* 从node节点中获取其对应的函数声明节点 */
4.   ......
5.   else
6.   {
7.       push_cfun (DECL_STRUCT_FUNCTION (decl));
8.
9.       if (!gimple_has_body_p (decl))   /* 若函数没有gimple_body,则对此函数进行gimple低端化 */
10.        gimplify_function_tree (decl);
11.
12.      if (!lowered)    /* 若当前函数尚未做过低端化处理则对齐进行低端化 */
13.    {
14.      gimple_register_cfg_hooks ();    /* 注册控制流图(cfg)被修改的hook */
15.      bitmap_obstack_initialize (NULL);
16.      execute_pass_list (cfun, g->get_passes ()->all_lowering_passes);    /* gimple低端化的所有pass均在这里执行 */
17.      free_dominance_info (CDI_POST_DOMINATORS);
18.      free_dominance_info (CDI_DOMINATORS);
19.      lowered = true;    /* 整个all_lowering_passes中所有pass都执行完毕了，则标记gimple低端化完毕 */
20.    }
21.      pop_cfun ();
22.    }
23. }
```

其中 execute_pass_list (cfun, g->get_passes ()->all_lowering_passes); 实际上是执行了一个pass链表,此链表中每个pass都有一个处理函数，这些处理函数按照顺序依次对当前函数执行完毕后才代表gimple低端化的完毕,此pass链表内容如下:

```
1. //pass.def
2.   INSERT_PASSES_AFTER (all_lowering_passes)
3.   NEXT_PASS (pass_warn_unused_result);        //此pass用来处理编译选项warn_unused_result
4.   NEXT_PASS (pass_diagnose_omp_blocks);
5.   NEXT_PASS (pass_diagnose_tm_blocks);
6.   NEXT_PASS (pass_lower_omp);
7.   NEXT_PASS (pass_lower_cf);                  //此pass负责gimple低端化的主要内容(去除gbind节点,将所有greturn节点放到函数最后
8.   NEXT_PASS (pass_lower_tm);
9.   NEXT_PASS (pass_refactor_eh);
10.  NEXT_PASS (pass_lower_eh);
11.  NEXT_PASS (pass_build_cfg);                 //此pass负责cfg的生成
12.  NEXT_PASS (pass_warn_function_return);
13.  NEXT_PASS (pass_expand_omp);
14.  NEXT_PASS (pass_sprintf_length, false);
15.  NEXT_PASS (pass_walloca, /*strict_mode_p=*/true);
16.  NEXT_PASS (pass_build_cgraph_edges);        //此pass负责cfg边的生成
17.  TERMINATE_PASS_LIST (all_lowering_passes)
```

以上是gimple低端化过程中所有的pass链表,但这些pass并不是每个都会执行的，具体和编译选项以及pass的gate函数是否通过有关,这里主要介绍4个pass,其中pass_lower_cf/pass_build_cfg/pass_build_cgraph_edges是gimple低端化中最重要的三个pass.

二、pass_warn_unused_result

gcc的编译选项warn_unused_result的作用是,当某个标记了此属性的函数被调用后,调用者如果没记录此函数的返回值则报错, 而实际的报错工作就是此pass报出的.

```
class pass_warn_unused_result : public gimple_opt_pass
{
public:
  pass_warn_unused_result (gcc::context *ctxt) : gimple_opt_pass (pass_data_warn_unused_result, ctxt)
  {}

  virtual bool gate (function *) { return flag_warn_unused_result; }
  virtual unsigned int execute (function *)
    {
      do_warn_unused_result (gimple_body (current_function_decl));
      return 0;
    }
};

void do_warn_unused_result (gimple_seq seq)
{
  tree fdecl, ftype;
  gimple_stmt_iterator i;

  for (i = gsi_start (seq); !gsi_end_p (i); gsi_next (&i))
    {
      gimple *g = gsi_stmt (i);

      switch (gimple_code (g))
    {
      /* 对于 GIMPLE_BIND/GIMPLE_TRY/GIMPLE_CATCH/GIMPLE_EH_FILTER 这四种节点,都是递归处理 */
    case GIMPLE_BIND:
      do_warn_unused_result (gimple_bind_body (as_a <gbind *>(g)));
      break;
    ......
    case GIMPLE_CALL:    /* 此函数主要分析的就是 GIMPLE_CALL指令*/
      if (gimple_call_lhs (g))    /* 若gcall指令的左值节点非空,则代表此函数调用的返回值是有人接收的,不需额外检查了,直接跳过 */
        break;
      if (gimple_call_internal_p (g))    /* 若当前调用的是gcc内置函数,则没保存返回值也没关系,因为内置函数不会有用户添加的warn_unused_result属性, 也直接跳
        break;
      fdecl = gimple_call_fndecl (g);     /* 获取被调用函数的声明节点(若有) */
      ftype = gimple_call_fntype (g);    /* 获取被调用函数[指针]在源码中指定的类型 */

      /* 到这里说明当前被调用函数非内置函数,且返回值也没有保存, 若此时发现被调用函数类型节点属性中有 warn unused属性,则说明此函数
        的返回值是需要被接受的,故报warning, 注意这里的依据是被调用函数的类型节点,而非声明节点,因为对于间接调用可能找不到声明节点,但类型节点一定是存在的 */
      if (lookup_attribute ("warn_unused_result", TYPE_ATTRIBUTES (ftype)))
        {
          location_t loc = gimple_location (g);
          if (fdecl)
            warning_at (loc, OPT_Wunused_result,"ignoring return value of %qD, "
                "declared with attribute warn_unused_result", fdecl);
          else
            warning_at (loc, OPT_Wunused_result,"ignoring return value of function "
                "declared with attribute warn_unused_result");
        }
      break;
    default:
      break;
    }
    }
}
```

此函数实现检查函数返回值是否被存储的思路实际上是**遍历每个函数的每条gcall指令,若此gcall指令中有左侧值节点非空,则函数的返回值一定是被接受了的**,反之则如果被调用函数指定了warn_unused_result属性则报错.

三、pass_lower_cf

pass_lower_cf是低端化的主要内容之一,其作用是将伪一维(实际上因为gbind等节点可嵌套,还是二维的)的gimple高端化后的指令序列真正转换为一维的gimple指令序列,其中的主要操作有两点:

1. 将所有gbind节点都删除掉,gbind节点的子指令序列链接到gbind所在的指令序列(实际上就是树节点的深度优先遍历)
2. 将所有greturn语句都用一条goto label语句替换,最后在整个gimple指令序列的末尾为这些label插入 glabel表达式

其代码如下:

```
class pass_lower_cf : public gimple_opt_pass
{
public:
  pass_lower_cf (gcc::context *ctxt) : gimple_opt_pass (pass_data_lower_cf, ctxt) {}

  virtual unsigned int execute (function *) { return lower_function_body (); }
};

unsigned int lower_function_body (void)
{
  struct lower_data data;
  gimple_seq body = gimple_body (current_function_decl);    /* 获取gimple高端化最后生成的指令序列 */
```

```
13.   gimple_seq lowered_body;      /* 这里记录gimple低端化后的指令序列 */
14.
15.   bind = gimple_seq_first_stmt (body);    /* 从body里面获取其gbind指令(唯一指令,其余都是gbind的子指令) */
16.   lowered_body = NULL;
17.   gimple_seq_add_stmt (&lowered_body, bind);    /* 将此唯一指令加到 lowered_body序列 */
18.   i = gsi_start (lowered_body); /* 获取指令序列的迭代器 */
19.
20.   lower_gimple_bind (&i, &data);    /* 对此gbind指令做低端化处理,其内部会递归处理此gbind节点子指令队列中所有指令 */
21.
22.     i = gsi_last (lowered_body);      /* i指向指令序列最后一条指令 */
23.
24.   /* gbind低端化过程中没解析到一个greturn语句都可能会在此队列中插入一个<label,greturn>对,代表此函数的一种返回,
25.      在低端化过程中greturn被 goto label代替,这里为label确定位置(也就是生成glabel指令,且在glabel指令之后插入真正的greturn指令 */
26.   while (!data.return_statements.is_empty ())
27.     {
28.       return_statements_t t = data.return_statements.pop ();   /* 逆序获取一个<label,greturn>对 */
29.       x = gimple_build_label (t.label);                        /* 为LABEL_DECL生成glabel指令 */
30.       gsi_insert_after (&i, x, GSI_CONTINUE_LINKING);          /* glabel指令插入到整个指令序列的末尾 */
31.       gsi_insert_after (&i, t.stmt, GSI_CONTINUE_LINKING);     /* glabel指令后插入greturn 指令 */
32.     }
33.
34.   gimple_set_body (current_function_decl, lowered_body);        /* 将低端化后的结果重写入func.gimple_body 中，替换原有的高端gimples生成的语句序列 */
35.     ......
36.   return 0;
37. }
38.
39. void lower_gimple_bind (gimple_stmt_iterator *gsi, struct lower_data *data)
40. {
41.     ......
42.   gbind *stmt = as_a <gbind *> (gsi_stmt (*gsi));    /* 从迭代器中获取要处理的 gbind节点 */
43.
44.   /*
45.      将此gbind节点中的所有编译器生成的临时变量和源码中显式声明的局部变量都加入到当前函数的function.local_decls中，因为gbind节点都要消除了,
46.     故将所有gbind包括其嵌套gbind中的变量都记录到当前函数中,后续占空间分配也是根据function.local_decls来的
47.      * block.vars记录的是源码中当前block中显示声明的变量.
48.      * gbind.vars和block.vars实际上是同一个队列,只不过其后续链接了gimple高端化过程中动态生成的临时变量.
49.      * 而function.local_decls则记录了当前函数中所有block(gbind)中所有的显示声明和动态生成的临时变量.
50.   */
51.   record_vars (gimple_bind_vars (stmt));
52.   lower_sequence (gimple_bind_body_ptr (stmt), data);    /* 低端化gbind节点中的整个子指令序列 */
53.   gsi_insert_seq_before (gsi, gimple_bind_body (stmt), GSI_SAME_STMT);    /* 将子指令序列低端化的结果插入到gbind指令前面 */
54.   gsi_remove (gsi, false);    /* 删除此gbind节点，gbind节点中的vars/body/block在前面都分别处理过了,gbind节点已无用了 */
55. }
56.
57. void lower_sequence (gimple_seq *seq, struct lower_data *data)
58. {
59.   gimple_stmt_iterator gsi;
60.
61.   for (gsi = gsi_start (*seq); !gsi_end_p (gsi); )    /* 递归处理指令序列中的每一条statement,直到结束; statement的划分是在lower_stmt中确定的 */
62.     lower_stmt (&gsi, data);    /* 此函数不但要对gimple的指令序列做低端化,还同时划分了statement, 每次返回时都代表一个statement解析完毕,gsi指向下一个stat
63. }
64.
65. void lower_stmt (gimple_stmt_iterator *gsi, struct lower_data *data)
66. {
67.   gimple *stmt = gsi_stmt (*gsi);
68.
69.   switch (gimple_code (stmt))            /* 根据不同的gimple节点类型做不同的处理 */
70.     {
71.     case GIMPLE_BIND:
72.       lower_gimple_bind (gsi, data);        /* 遇到子gbind节点,则递归处理 */
73.       return;
74.     case GIMPLE_COND:
75.     case GIMPLE_GOTO:
76.     case GIMPLE_SWITCH:
77.       data->cannot_fallthru = true;
78.       gsi_next (gsi);
79.       return;
80.     case GIMPLE_RETURN:
81.       if (data->cannot_fallthru)              /* 若前一句导致当前return语句无法执行到,则当前return语句被删除,如 */
82.           gsi_remove (gsi, false);            /* return; return; 则第二句return会走到这里; return 0; return 1;则不会,因为前面会先生成一条assgin语句 *
83.       else
84.       {
85.           lower_gimple_return (gsi, data);    /* 正常return的处理,这里将return语句替换为goto语句，并在data.return_statements 中增加此greturn和LABLE_D
86.           data->cannot_fallthru = true;
87.       }
88.        return;
89.     case GIMPLE_TRY:
90.             ......
91.     case GIMPLE_NOP:
92.     case GIMPLE_ASM:
93.     case GIMPLE_ASSIGN:
94.            ......
95.       break;    /* 对于大部分case来说,gimple低端化实际上并没有做处理，如出现最多的GIMPLE_ASSIGN(如加减法等)，注意对于间接调用最终会变为一个gassign +一个g
96.
97.     case GIMPLE_CALL:
98.       {
99.     tree decl = gimple_call_fndecl (stmt);    /* 尝试获取被调用函数 */
100.    for (i = 0; i < gimple_call_num_args (stmt); i++)
101.      {
102.
103.       tree arg = gimple_call_arg (stmt, i);    /* 获取第i个参数的树节点 */
104.       if (EXPR_P (arg))
105.         TREE_SET_BLOCK (arg, data->block);       /* 若参数是表达式,则为其设置block */
```

```
      }

    if (decl && (flags_from_decl_or_type (decl) & ECF_NORETURN))    /* 若被调用函数是一个不可返回的函数,则为下一条指令设置不可进入 */
      {
        data->cannot_fallthru = true;
        gsi_next (gsi);
        return;
      }
    ......
    default:
      gcc_unreachable ();
    }

  gsi_next (gsi);        /* 当前statement解析完毕,默认指向下一条gimple语句的开始后才返回, */
}
```

在此函数中需要注意的有两点:

1) 关于gbind节点:

在gimple高端化结束后, 指令序列中是有gbind节点的

- 此时整个函数高端化过程中生成的所有临时变量都被添加到函数体的gbind.vars链表的最前面
- 而函数内部的每个子gbind节点的.vars中只链接了其作用域中显示声明的变量

也就是说除了函数体的gbind节点外,其他子gbind节点上链接的变量应该和AST代码生成时block中的vars是一致的，实际上二者就是共享同一个链表的.

而gimple低端化时候gbind节点是都要被删除的,此时就需要将所有vars都放到函数中记录了,**最终是通过function.local_decls记录了此函数中所有动态生成和代码中显示声明的变量,而最终函数栈分配也是依赖于此中的变量个数**，需要注意的是local_decls是个vector,故动态生成的临时变量放入后最后会将其自身的链接指向都清空,而源码中显式声明的变量则不会去链，因为其在block中还会被用到.

2) 关于greturn节点:

在gimple高端化的过程中,实际上就是按照源码中greturn语句出现的顺序直接翻译的，greturn贯穿整个指令序列中;

而在gimple低端化后,则是将所有的greturn语句都统一放到了指令序列的最末尾，而在原有的greturn的位置使用goto label跳转到某个label,而在指令序列的末尾则为每个label分别添加一个glabel + 原有的那个greturn语句, *.008t.lower记录此pass处理后的指令序列.

四、pass_build_cfg

pass_build_cfg的作用是将**低端化后的指令序列(seq)转换为函数的控制流图**(CFG,Control Flow Graph**,函数的控制流图代表的是一个函数内部的控制流转移关系,与函数间的控制流转译无关**). 控制流图中的两个主要概念是基本块(bb,basic_block)和边(edge),在此函数中**既要将seq划分成一个个基本块(BB)**,还要**根据seq中控制流转移语句设置各个BB之间的转移关系(BB之间的边)。**

在分析此pass之前,需要先了解控制流图中三个最主要的结构体,分别是:

- **struct control_flow_graph:** 此结构体记录了一个函数的控制流图, 控制流图是由多个基本块和多个边构成的,在pass_build_cfg的开始,系统会为每个函数分配一个cfg结构体记录在function->cfg中
- **struct basic_block_def :** 此结构体记录函数中的一个基本块(BB)的信息
- **struct edge_def:** 此结构体记录此函数中的一条边(e)的信息

此三个结构体的详细定义如下:

```
//这里需要注意的是,CFG中各个结构体通常都是通过宏来访问的,如ENTRY_BLOCK是通过ENTRY_BLOCK_PTR_FOR_FN宏来访问的,其余可参考源码.
struct control_flow_graph {
  /* 指向当前函数的入口基本块(ENTRY_BLOCK), 一个函数只有唯一的入口BB,此基本块是控制流的开始,是在cfg初始化时(init_flow)生成的, 不论函数内是否有指令序列,都会
  basic_block x_entry_block_ptr;
  /* 指向当前函数的出口基本块(EXIT_BLOCK), 一个函数只有唯一的出口BB,此基本块是控制流的结束,函数返回前总是要跳转到此基本块, 不论函数内是否有指令序列,都会生成
  basic_block x_exit_block_ptr;
  /* x_basic_block_info 是一个可动态扩展的数组(初始元素20个),其中每一个元素都记录一个基本块的指针,基本块的指针按照基本块的index记录在此数组对应的下标中 */
  vec<basic_block, va_gc> *x_basic_block_info;
  /* 记录当前CFG中已有的基本块的个数,初值为2,即有出入口两个BB */
  int x_n_basic_blocks;
  /* 记录此CFG中已有的边的个数, CFG中的边是分散记录在每一个BB中的 */
  int x_n_edges;
  /* 记录当前CFG中下一个可用的基本块编号,初值同样为2,0/1分别是入口/出口BB */
  int x_last_basic_block;

  /* 记录当前CFG中为下一个标签分配的UID, label_decl的uid是在函数内唯一的, 在划分基本块的过程中(make_blocks)若发现glabel语句,则会根据last_label_uid
     为其对应的label_decl设置函数内唯一的uid(见gimple_set_bb => LABEL_DECL_UID), 这个uid会作为 x_label_to_block_map 的索引,用来确定此label_decl所在的B
  int last_label_uid;
  /* 此数组记录当前函数中标签和其所在基本块的映射关系, 这里map[i]=bb; 其中i是某个标签的UID, BB是此标签所在BB的指针,
     在划分基本块后,如果一个基本块中有标签,那么此标签一定在此基本块gimple语句序列的第一句(或者前N句都是标签),向标签的跳转实际上就是向此基本块的跳转,
     在基本块划分的过车用中(make_blocks)这里会保存标签与基本块的映射关系,以便于在后续标签清除和创建边时提供标签(LABEL_DECL)到BB的索引 */
  vec<basic_block, va_gc> *x_label_to_block_map;

  enum profile_status_d x_profile_status;
```

```
  enum dom_state x_dom_computed[2];
  unsigned x_n_bbs_in_dom_tree[2];
  int max_jumptable_ents;
  profile_count count_max;    /* 记录函数中可以出现的最大BB个数,是个常量 */
  int edge_flags_allocated;
  int bb_flags_allocated;
}

struct basic_block_def {
  vec<edge, va_gc> *preds;    /* 此数组记录当前BB所有的入边 */
  vec<edge, va_gc> *succs;    /* 此数组记录当前BB所有的出边 */
  PTR aux;
  struct loop *loop_father;
  struct et_node * dom[2];

  /* 这里是两个单项链表,在整个链表中ENTRY_BLOCK总是第一个元素,EXIT_BLOCK总是最后一个元素,后续每新增一个BB,都会插入到 EXIT_BLOCK的前面,其余BB的后面,
     这个链表中BB的顺序并不代表函数代码中控制流的转移(BB之间的跳转关系是通过edge来体现的,并不是此链表),而是代表各个BB对应的gimple指令序列在源码中出现的顺序,
    一个BB中若最后一条语句不是控制流转移语句(如ggoto/gcond),则其在make_edges创建边时会默认fallthru到->next BB中,在源码中相当于继续下一条指令执行.
  */
  basic_block prev_bb;
  basic_block next_bb;

  union basic_block_il_dependent {
      struct gimple_bb_info  gimple;      /* gimple.seq 记录从函数整个指令序列中拆分出来的属于当前BB的这部分指令序列的首个元素地址 */
      struct {
        rtx_insn *head_;
        struct rtl_bb_info * rtl;
      }  x;
    }  il;
  int flags;
  int index;   /* 记录当前基本块的编号,ENTRY_BLOCK编号为0, EXIT_BLOCK编号为1,其余基本块按照创建顺序依次编号,此编号可作为CFG x_basic_block_info 数组下标对
  profile_count count;
  int discriminator;
}

struct edge_def {
  basic_block src;    /* 记录当前边起始自哪个BB */
  basic_block dest;   /* 记录当前边指向哪个BB */
  union edge_def_insns {
    gimple_seq g;
    rtx_insn *r;
  } insns;
  PTR aux;
  location_t goto_locus;    /* 记录当前edge所代表的转移语句的源码位置 */
  unsigned int dest_idx;
  int flags;                  /* 记录当前边的属性,如 EDGE_FALLTHRU 代表无条件进入dest BB,EDGE_TRUE_VALUE代表在src BB最后一条语句为true时候进入dest BB
  profile_probability probability;
  inline profile_count count () const;
}
```

之后就是此pass的定义:

```
class pass_build_cfg : public gimple_opt_pass
{
public:
  pass_build_cfg (gcc::context *ctxt) : gimple_opt_pass (pass_data_build_cfg, ctxt) {}

  virtual unsigned int execute (function *) { return execute_build_cfg (); }
};

unsigned int execute_build_cfg (void)
{
  gimple_seq body = gimple_body (current_function_decl);    /* 获取前面gimple低端化后的指令序列 */
  build_gimple_cfg (body);    /* 此函数负责将低端化后的指令序列划分为各个基本块,并将gimple指令中的控制流转移转换为基本块(BB)之间的边(edge) */
  /* 划分基本块后, 此函数的整个gimple指令序列已经被拆分到各个基本块中了(BB.il.gimple.seq), 函数中fn->gimple_body的声明周期已结束, 后续此函数中所有指令只能
  gimple_set_body (current_function_decl, NULL);

  if (dump_file && (dump_flags & TDF_DETAILS))    /* 若指定了-fump-xxx-details 则这里会dump更多细节 */
    {
      fprintf (dump_file, "Scope blocks:\n");
      dump_scope_blocks (dump_file, dump_flags);
    }
  cleanup_tree_cfg ();
  loop_optimizer_init (AVOID_CFG_MODIFICATIONS);    /* 分析此cfg中的循环, 这里先pass  */
  replace_loop_annotate ();
  return 0;
}

void build_gimple_cfg (gimple_seq seq)
{
  /* 注册cfg的操作函数,不同阶段处理cfg时需要用到不同的函数(否则后面pass执行完,还按照前面的操作可能不兼容),这里注册的是 gimple_cfg_hooks */
  gimple_register_cfg_hooks ();
  memset ((void *) &cfg_stats, 0, sizeof (cfg_stats));    /* 此结构体记录当前cfg 创建过程中merge了多少个标签 */

  /* 为当前函数分配一个control_flow_graph结构体以记录其控制流,此结构体的指针记录在fn->cfg中; 同时为当前函数创建两个初始basic_block(ENTRY_BLOCK/EXIT_BLOCI
  init_empty_tree_cfg ();

  /* 此函数根据当前函数的指令序列seq,为此函数划分出一个个基本块,每个基本块通过BB.il.gimple.seq记录属于此基本块的指令序列,
     这些基本块会顺序插入到ENTRY BB链表的末尾,但EXIT BB总是最后一个基本块(ENTRY => BB1 => BB2 => ... => EXIT) */
  make_blocks (seq);

```

```
  /* 若当前函数中指令序列为空,则make_blocks中有可能没有新增基本块,若make_blocks函数结束后还是只有两个基本块(ENTRY/EXIT),则无条件创建一个新的BB,也就是说一个
  if (n_basic_blocks_for_fn (cfun) == NUM_FIXED_BLOCKS)
    create_empty_bb (ENTRY_BLOCK_PTR_FOR_FN (cfun));

  /* x_basic_block_info 数组中保存各个基本块的指针,若此数组已满,则动态扩展此数组大小 */
  if (basic_block_info_for_fn (cfun)->length () < (size_t) n_basic_blocks_for_fn (cfun))
    vec_safe_grow_cleared (basic_block_info_for_fn (cfun), n_basic_blocks_for_fn (cfun));

  /* 根据基本块划分(make_blocks)逻辑可知:
     * 函数的每个基本块中若有glabel语句,则这些glabel语句必须全部都在此BB指令序列的最开始(可以是连续多个,但不会出现在此BB的其他非调试语句之后)
     * 函数的每个基本块中只有最后一条语句可能是控制流转移语句(如ggoto/gcond),但最后一条也有可能不是控制流转移语句(此时代码逻辑会直接fallthru到下一个基本块)
     而此函数会:
     * 先分析每个BB的指令序列中最开始的glabel语句,并为每个BB确认一个主标签(每个glabel语句都是为一个标签确定源码位置,故也对应一个标签),源码中的glabel语句优先
       否则选择第一个出现的glabel语句,若没有glabel语句则无标签;
     * 然后将每个BB的指令序列最后一句中的跳转到的目标标签,切换为目标标签所在BB的主标签(实际上是修改ggoto/gcond中跳转到的LABEL_DECL为其所在BB中主标签的LABEL
       确定一个LABEL_DECL属于哪个BB,在make_blocks中解析到glabel指令是就会在x_label_to_block_map中增加标签和BB的映射关系)
     * 最后将每个BB中无用的glabel语句删除(源码中标签定义导致的glabel语句,BB中被引用到的主标签对应的glabel语句和non_local/foreced标签对应的glabel语句保留,其
    简言之此函数分析当前函数cfg中无用的glabel语句并将其删除
  */
  cleanup_dead_labels ();

  group_case_labels ();    /* 合并switch 中的case  */
  discriminator_per_locus = new hash_table<locus_discrim_hasher> (13);

  /* 此函数负责分析将每个BB中最后一条语句,并为此BB设置指向下一个BB的控制流转移的边(edge),对于:
     * 最后一条为控制流转移语句的(如ggoto/gcond),则找到指令label_decl所在的BB,建立当前BB到目标BB的边,同时清空指令中的label_decl(对于ggoto语句,由于没有额外
     * 最后一条非空指令转移语句的,则直接构建当前BB到BB->next(源码中下一个BB)的边,这种属于默认fallthru进入下一个BB
     此函数结束后,BB之间的跳转关系则完全由BB之间的边决定了(preds/succes),函数所有BB的gimple指令序列中不应该再有对任何label_decl的引用(因为具体标签都应该转
  make_edges ();

  assign_discriminators ();

  /* 将label跳转转换为边后再次清除无用的glabel语句,由于BB最后一条语句中的label都转换为边了(在指令中也清空了), 故在此分析时很多之前给label_decl定位的glabel
     故这里第二次处理会删除大量的glabel语句 */
  cleanup_dead_labels ();

  delete discriminator_per_locus;
  discriminator_per_locus = NULL;
}
```

此pass执行完毕后,**低端化后当前函数的整个指令序列(fn->gimple_body)就被拆分为一个个小的指令序列,记录在不同BB的.il.gimple.seq中了**,当前函数的语义已不再由fn->gimple_body代表(已清空),而是由fn->cfg这个control_flow_graph结构体代表.**而cfg结构体中主要记录的就是各个BB的指针,而各个BB中又记录其指向当前cfg其他BB的边(edges)**.

五、pass_build_cgraph_edges

pass_build_cgraph_edges是整个gimple低端化的最后一个pass,其作用是分析当前编译单元内所有的函数,并为这些函数之间**构建调用关系图(Call Graph)**.

调用关系图(Call Graph)和控制流图(Control Flow Graph)的区别是: **调用关系图分析的是函数之间的调用关系,而控制流图分析的是一个函数内部各个基本块之间的跳转关系**. 但二者的基本操作都是将节点和节点之间的关系构建成图.

Call Graph分析的是函数之间的调用关系,故**Call Graph中的每一个节点**都代表一个函数,在gcc中**则是由一个cgraph_node结构体来表示的**. 在gcc中函数和变量都属于符号(基类都是symtab_node), 而具体来说,函数是由cgraph_node来表示的,而全局变量是由varpool_node来表示的,二者都是symtab_node的子类.

gcc中的所有符号都是通过symtab_node.previous/next链接起来的,其中包括函数符号(cgraph_node)和全局变量符号(varpool_node), **故Call Graph中的所有节点并没有单独成链表,若需遍历则需要通过全局符号表(symtab)来遍历**.

cgraph_node/varpool_node是在语法分析阶段解析出每个外部声明后直接生成的,其路径为:

```
toplev::main => do_compile => compile_file => lang_hooks.parse_file () => c_common_parse_file => c_parse_file
 => c_parser_translation_unit
  => c_parser_external_declaration
   => c_parser_declaration_or_fndef (parser, true, true, true, false, true, NULL, vNULL);
      => finish_decl
         =>  varpool_node::finalize_decl (decl);            /* 为全局变量创建varpool_node */
      => finish_function
         => c_genericize (fndecl)
            => cgn = cgraph_node::get_create (fndecl);      /* 为全局函数定义创建cgraph_node */
```

除了cgraph_node代表Call Graph中的一个节点外, 在gcc中还使用cgraph_edge代表Call Graph中的一条边,这两个结构体此处的关键成员定义如下:

```
//注,此两个结构体中成员和函数较多,这里只记录Call Graph生成的相关部分
struct cgraph_node : public symtab_node {
  /* 此链表中每一个元素都记录从当前函数(this,也就是caller)到目标函数(callee)的一条边(callee未决的记录在indirect_calls中);
     在为当前函数(this)构建Call graph时(如pass_build_cgraph_edges pass),每解析到一条gcall指令,且被调用函数是确定的情况下都会在此队列增加一条边.
     这个队列之所以叫做callees,是因为这里的边代表的都是当前函数对不同子函数(callee)的调用. */
  cgraph_edge *callees;
  /* 此链表中记录的是所有确定的调用了当前函数(this为callee)的边,其代表的是在整个编译单元内,当前函数确定被调用的位置.
     这个队列之所以叫做callers,是因为这里的边代表的是当前函数被不同父函数(caller)的调用. */
```

```
  cgraph_edge *callers;
  /* 当前函数中解析出的所有目标函数(callee)确定的gcall指令生成的边都链接在了callees队列中，而目标函数未决的边都链接在此队列中，
    (目标未决则一定是间接调用,但反之未必,如指针常量目标就可能是确定的) */
  cgraph_edge *indirect_calls;
  /* 标记此函数是否已经做了gimple低端化处理，当低端化的all_lowering_passes中的全部passes之后,才会标记lowerd为1 */
  unsigned lowered : 1;
private:
  /* 当前cgraph_node在全局符号表symtab中的唯一uid,见symbol_table::allocate_cgraph_symbol */
  int m_uid;
}

struct cgraph_edge {
  cgraph_node *caller;    /* 此edge中的caller */
  cgraph_node *callee;    /* 此edge中的callee */

  /* prev/next_caller是此edge的两个单项链表,其作用是链接和当前edge的callee相同的其他所有边,相当于链接了系统中所有调用了此callee的边.  */
  cgraph_edge *prev_caller;
  cgraph_edge *next_caller;

  /* prev/next_callee是此edge的另外两个单项链表,其作用是链接和当前edge的caller相同的其他所有边,相当于链接了系统中一个函数(caller)中所有确定的子函数调用相关
  cgraph_edge *prev_callee;
  cgraph_edge *next_callee;

  gcall *call_stmt;      /* 记录导致此边生成的caller中某个BB中的那条gcall语句 */
  cgraph_indirect_call_info *indirect_info;    /* 对于callee未决的边，使用此结构体记录间接调用信息 */
}
```

而此pass的主体部分定义如下:

```
class pass_build_cgraph_edges : public gimple_opt_pass
{
public:
  pass_build_cgraph_edges (gcc::context *ctxt): gimple_opt_pass (pass_data_build_cgraph_edges, ctxt){}
  virtual unsigned int execute (function *);
};

unsigned int pass_build_cgraph_edges::execute (function *fun)
{
  basic_block bb;
  cgraph_node *node = cgraph_node::get (current_function_decl);     /* 从当前函数的FUNCTION_DECL节点中找到此函数符号对应的cgraph_node节点 */

  FOR_EACH_BB_FN (bb, fun)    /* pass_build_cfg之后 fn->gimple_body 就被划分为各个基本块记录到fn->cfg中了，这里一次遍历此函数中所有的基本块 */
    {
      for (gsi = gsi_start_bb (bb); !gsi_end_p (gsi); gsi_next (&gsi))    /* 遍历此BB中的每一条gimple指令 */
    {
      gimple *stmt = gsi_stmt (gsi);
      tree decl;

      if (is_gimple_debug (stmt))        /* debug 指令直接pass */
        continue;

      if (gcall *call_stmt = dyn_cast <gcall *> (stmt))    /* 此pass的主要作用就是分析此函数中的每一条gcall指令,并为其生成call graph 中的边 */
        {
          /* 若gcall指令能直接定位到目标函数(如直接调用或指针常量的间接调用)则decl为被调用函数的声明节点，定位不到则返回NULL  */
          decl = gimple_call_fndecl (call_stmt);
          if (decl)
            /* 若被调用函数确定了,那么当前边caller/callee就都确定了,这里直接构建一条确定的边，此边被链接到caller的callees队列,也同时链接到callee的callers
            node->create_edge (cgraph_node::get_create (decl), call_stmt, bb->count);
          else if (gimple_call_internal_p (call_stmt)) ;    /* 如果是对gcc内部函数的调用,则不创建边,直接返回(估计是内联了?) */
          else
            /* 若被调用函数未决,则创建一个间接调用的边,其记录在caller的indirect_calls队列中,因为callee未决故不会在记录在callee相关队列中 */
            node->create_indirect_edge (call_stmt, gimple_call_flags (call_stmt), bb->count);
        }
      node->record_stmt_references (stmt);
          ......
    }
  /* 遍历当前函数中所有的临时变量(gcc动态生成)或局部变量(源码中的)，对于其中static的全局变量,将其添加到queued_nodes中,
     等下一次循环到 analyze_functions时按照全局变量来做分析 */
  FOR_EACH_LOCAL_DECL (fun, ix, decl)
    if (VAR_P (decl) && (TREE_STATIC (decl) && !DECL_EXTERNAL (decl))
    && !DECL_HAS_VALUE_EXPR_P (decl) && TREE_TYPE (decl) != error_mark_node)
      varpool_node::finalize_decl (decl);     /* 在 CONSTRUCTION阶段,这里只是将此节点加入到 queued_nodes队列中 */

  return 0;
}
```

这里需要注意的是,构建Call Graph相关的pass一共有三个,分别为:

- **pass_build_cgraph_edges:** 此pass只在all_lowering_passes中出现,是第一个为整个编译单元构建Call graph的pass
- **pass_rebuild_cgraph_edges:** 此pass在整个passes链表中出现了不止一次,其作用是**删除当前函数中所有指向callee的边,并重新分析当前函数中的gcall指令并重新构建边**(注,删除只针对当前函数的callees,其caller边需要到caller函数中处理,重构的过程和pass_build_cgraph_edges是类似的)
- **pass_remove_cgraph_callee_edges:** 此pass的作用是**删除当前函数中所有指向callee的边,但并不重构call graph**.

六、总结

gimplify一共分为两个步骤,这两个步骤都是针对每个函数依次完成的:

- **gimple高端化:** 其是在 gimplify_function_tree函数中完成的,其作用是将函数的AST树节点(function_decl.saved_tree)转换为一个gimple指令序列,并保存在此函数(struct function) fn.gimple_body中(之后AST树节点被清空).此过程是将二维的AST树结构体转化为了一个准一维的指令序列,说是准一维的原因是因为还有gbind等几个指令还是个二维结构.
- **gimple低端化:** gimple低端化是由多个pass共同完成的,这些pass都记录在all_lower_passes链表中,其中主要的pass有三个:

1. **pass_lower_cf:** 此pass负责将高端化后的gimple指令序列做低端化处理,其实际上主要处理了gbind语句(一个树的深度优先遍历将准一维彻底转变为一维,转换后的指令序列中不再存在gbind节点),并将greturn语句全部放到了指令序列的末尾.低端化后的指令序列同样保存在fn.gimple_body中
2. **pass_build_cfg:** 此pass负责将低端化后的一维gimple指令序列拆分为一个个基本块,每个基本块中只记录此指令序列中其对应部分的指令(bb.il.gimple.seq). 拆分后则根据每个bb的最后一条语句,分析bb之间的跳转关系(每个bb的最后一条语句决定其如何进入下一个bb,中间其他语句一定不会导致控制流转移),bb(struct basic_block)中的每一个跳转则由一个边(struct edge结构体)来体现
3. **pass_build_cgraph_edges:** pass_build_cfg是用来分析一个函数内部各个bb之间的跳转关系(也就是控制流图cfg), **而此pass则是用来分析当前编译单元内所有函数之间的调用关系(call cgraph, 函数调用关系图)**. 编译单元内的**每个函数都通过struct cgraph_node结构体表示**(继承自符号表结构体symbol_node),**而函数之间的调用关系则通过边(struct cgraph_edge结构体)表示**. 整个编译单元,除了gcc builtin的函数调用外,每个函数内出现的任意一个gcall指令最终都会对应一个边(cgraph_edge结构体),其他指令都不会导致边的产生,**cgraph_edge中的caller总是确定的(因为cgraph_edge是分析caller产生的),但其callee未必是确定的**:

    1. 对于目标函数确定的调用(直接调用,或间接常量函数指针调用),cgraph_edge的callee是确定的(指向被调用函数的cgraph_node节点)
    2. 对于目标函数不确定的调用(间接调用),cgraph_edge的callee为空(未决)

一个函数节点在gimplify的各个阶段,其关键字段变化如下:

当前函数相关字段\执行阶段	gimple高端化之前	gimple高端化之后	pass_lower_cf之后	pass_build_cfg之后	all_lower_passes后(整个低端化完成后)
tree_function_decl.saved_tree	非空,记录函数AST	NULL	NULL	NULL	NULL
function.gimple_body	NULL	非空,记录**准一维**gimple指令序列	非空,记录 **一维**gimple指令序列	NULL	NULL
function.cfg	NULL	NULL	NULL	非空,记录此函数的整个控制流图(cfg)	非空,记录此函数的整个控制流图(cfg)
cgraph_node.lowered	0	0	0	0	1,代表此函数的gimple低端化执行完毕