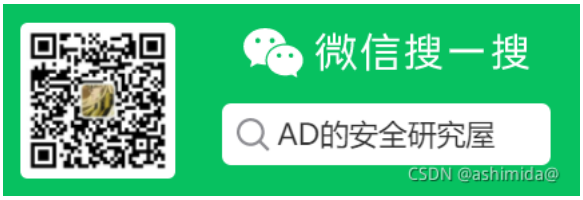
GCC源码分析(十) — 函数节点的gimple高端化

更多内容可关注微信公众号



一、gimple高端化

前面提到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,对于函数节点最终会执行到cgraph_node::analyze对其进行gimple高端化和低端化分析,其中 **gimple高端化是通过函数gimplify_function_tree(decl)来完成的**,此函数用来分析一个具体的函数声明节点,其代码大体如下:

```
/*
   此函数负责整个gimple高端化,其创建了一个gbind节点,其中:
   .body记录当前函数形参和函数体gimplie高端化生成的语句序列
   .vars记录gimplify过程中编译器自动生成的和函数体内显示声明的所有变量声明节点
   .block记录函数体的tree_block
  最终此gbind节点被记录到fndecl->function->gimple_body中,且同时清空其AST树节点指针(fndecl->saved_tree),代表从此此函数只能继续通过gimple指令序列来分析,而
*/
void gimplify_function_tree (tree fndecl)
{
  gimple_seq seq;
  gbind *bind;

  bind = gimplify_body (fndecl, true);          /* gimplify函数参数链表和函数体,最终返回一个gbind结果记录此函数所有语义 */
  seq = NULL;
  gimple_seq_add_stmt (&seq, bind);             /* 在seq指令序列中加入此gbind节点,实际上最终seq = bind */
  gimple_set_body (fndecl, seq);                /* gimplify_body返回的gbind节点记录了当前函数AST树节点gimple高端化的结果,此结果被记录到fndecl->functic

  if (flag_instrument_function_entry_exit && ... ) { ... }    /* 若命令行指定了-finstrument-functions,则会为此函数进行gimple级别的插桩 */
  DECL_SAVED_TREE (fndecl) = NULL_TREE;         /* gimplify_body生成的gbind节点代表了此函数AST树节点的所有语义,后续AST树节点不会再被使用,这里清空 */
  dump_function (TDI_gimple, fndecl);           /* 将函数信息dump到 TDI_gimple对应的文件中去(通常是*.c.005t.gimple) */
}
```

在源码解析->AST树节点后,一个AST树节点即可代表源码中此外部声明的所有语义,不论是函数中代码的顺序执行、分支、循环, 在树节点中都可以表示出来.

源码本质上是顺序的,其转化为的树节点默认是需要一种遍历顺序才可以与源码对应的, gimple高端化的过程实际上就是**通过深度优先算法(自上而下,从左到右)，遍历AST树节点,将树节点中的参数链表,语句链表解析为一个顺序的gimple指令序列. 解析后的gimple指令序列是一个一维的序列,**但更严谨的说应该是一个**趋于一维的指令序列,**因为此时 gimple指令序列中还存在类似gbind的节点,一般的gimple节点仅仅代表一个语句,而一个gbind节点又包括一个子语句序列,故存在gbind等节点时不能认为其完全是一维的。

二、gimplify_body

gimplify_body实现了函数参数列表和函数体的gimple高端化,其代码如下:

```
gbind * gimplify_body (tree fndecl, bool do_parms)
{
  gimple_seq parm_stmts, parm_cleanup = NULL, seq;
  gimple *outer_stmt;
  gbind *outer_bind;

  init_tree_ssa (cfun);            /* 此函数主要初始化 function->gimple_df结构体,gimplify过程中生成的SSA_NAME节点都保存在 gimple_df.ssanames中 */
  push_gimplify_context (true);     /* 初始化一个 gimplify上下文,保存在 全局变量gimplify_ctxp中, 在gimplify过程中新生成的变量节点等都会临时记录在这里 */

  parm_stmts = do_parms ? gimplify_parameters (&parm_cleanup) : NULL;    /* 若需要,则这里gimplify 函数的参数链表, 此过程中若产生了gimple指令序列则记录

  seq = NULL;
  /*
     此函数负责gimplify 函数的函数体节点, 函数体节点通常是一个 BIND_EXPR,代表函数{}内的定义,但对于main函数来说这里是一个statement_list,不论是什么都属于sta
     函数体的解析结果最终形成了一个gimple指令序列,此指令序列通过seq返回(通常是gbind节点),而解析过程中编译器新创建的变量则记录在全局变量gimplify_ctxp->temps
  */
  gimplify_stmt (&DECL_SAVED_TREE (fndecl), &seq);
```

```
18.
19.   outer_stmt = gimple_seq_first_stmt (seq);                       /* 获取seq指令序列中第一条指令节点(是一个gimple的指针) */
20.
21.   if (gimple_code (outer_stmt) == GIMPLE_BIND                     /* 若当前outer_stmt本身已经是 gbind节点了,则outer_bind直接使用outer_stmt */
22.       && gimple_seq_first (seq) == gimple_seq_last (seq))
23.     outer_bind = as_a <gbind *> (outer_stmt);
24.   else
25.     outer_bind = gimple_build_bind (NULL_TREE, seq, NULL);        /* 否则新创建一个gbind节点包裹原有的指令序列(如main函数) */
26.
27.   /* 清空 fndecl.saved_tree 也就是生成gbind节点后清除了原有AST树节点的指针,后续针对此函数的分析都应该基于此gbind节点，这也是cgraph_node::analyze时是否执行
28.   DECL_SAVED_TREE (fndecl) = NULL_TREE;
29.
30.   /* 若解析函数参数链表时产生了gimple语句,则此语句添加到outer_bind这个gbind节点的.body的最前面，gimplify之后 gbind.body记录了此block中生成的所有gimple的i
31.   if (!gimple_seq_empty_p (parm_stmts)) { ... }
32.
33.   /* pop 整个 gimplify上下文,主要将gimplify过程中生成的所有临时变量都保存到gbind->vars中(此时outer_bind必然是一个gbind节点) */
34.   pop_gimplify_context (outer_bind);
35.
36.   return outer_bind;    /* 返回最终的 gbind节点 */
37. }
```

gimplify_body可以总结为3步:

1. gimplify参数链表
2. gimplify函数体
3. 将此过程中生成的所有gimple指令序列和临时变量记录到一个gbind节点中并返回

三、gimplify_stmt

gimplify_stmt负责一个语句的gimplify,此函数定义如下:

```
1. bool gimplify_stmt (tree *stmt_p, gimple_seq *seq_p)
2. {
3.   gimple_seq_node last;
4.
5.   last = gimple_seq_last (*seq_p);    /* 获取gimple指令序列 seq_p中最后一条gimple指令地址 */
6.   gimplify_expr (stmt_p, seq_p, NULL, is_gimple_stmt, fb_none);    /* gimplify stmt_p代表的语句,其返回结果顺序写入seq_p指令序列的末尾 */
7.   return last != gimple_seq_last (*seq_p);    /* 若seq_p指令序列中新增了语句则返回true，未新增则返回false */
8. }
```

gimplify_stmt的逻辑很简单,但需要注意的是在AST中实际上并没有一个statement节点类型,而**判断一个AST节点是否是statement(语句)的标准则是is_gimple_stmt函数**，此函数就是个swith case, 此函数中**判定为语句(statement)的节点包括**:

1. 非空的NOP_EXPR节点
2. 非VOID类型的BIND_EXPR/COND_EXPR节点
3. SWITCH_EXPR/GOTO_EXPR/RETURN_EXPR/LABEL_EXPR/.../ASM_EXPR/STATEMENT_LIST/CALL_EXPR/MODIFY_EXPR /PREDICT_EXPR节点

四、gimplify_expr

所有节点都被认为是表达式，故gimplify_expr也是gimple高端化中最常用的一个函数,此函数逻辑比较复杂,其定义如下:

```
1. /*
2.    此函数用来分析expr_p代表的一个表达式树(包括递归分析其所有的子树),分析完毕后expr_p代表的表达式生成的所有gimple指令序列都会被添加到pre_p和post_p中,如果exp
3.    * 如果当前分析的是一条语句(statement),或者调用gimplify_expr时显示指定不需要返回值(fb_none,目前看到的只有statment会传入fb_none),那么expr_p被设置为NULL,
4.    * 如果当前gimplify_expr需要返回值,且当前解析的非语句(statment):
5.      - 若解析完毕后的表达式结果 expr_p满足gimple_test_f这个条件,则直接返回*expr_p作为返回值即可.
6.      - 若解析完毕后的表达式结果 expr_p不满足gimple_test_f条件:
7.        -- 若当前需要返回一个左值(fb_lvalue),且 expr_p是可获取地址的,则构建一个对expr_p的MEM_REF引用节点并返回.
8.        -- 若当前需要返回一个右值(fb_rvalue),且 expr_p是可作为右值的节点,则构建一个临时变量(或SSA_NAME)来记录当前的右值并返回.
9.        -- 其他情况均报错
10. */
11. enum gimplify_status gimplify_expr (tree *expr_p, gimple_seq *pre_p, gimple_seq *post_p, bool (*gimple_test_f) (tree), fallback_t fallback)
12. {
13.   tree tmp;
14.   gimple_seq internal_pre = NULL;
15.   gimple_seq internal_post = NULL;
16.   tree save_expr;
17.   bool is_statement;
18.   location_t saved_location;
19.   enum gimplify_status ret;
20.   gimple_stmt_iterator pre_last_gsi, post_last_gsi;
21.   tree label;
22.
23.   save_expr = *expr_p;         /* 若expr_p传入一个空的AST树节点,则直接返回 GS_ALL_DONE,代表当前expr_p表达式树节点gimplify完毕 */
24.   if (save_expr == NULL_TREE) return GS_ALL_DONE;
25.
26.   is_statement = gimple_test_f == is_gimple_stmt;    /* 记录当前分析的是否是一个语句表达式 */
27.   if (pre_p == NULL) pre_p = &internal_pre;
28.   if (post_p == NULL) post_p = &internal_post;
29.
```

```
do
{
    save_expr = *expr_p;    /* 再次记录当前正在处理的表达式的AST树节点 */
    switch (TREE_CODE (*expr_p))    /* 根据不同的TREE_CODE，执行不同的 gimplify转换 */
  {
  case POSTINCREMENT_EXPR:    /* ++, -- 表达式的转换 */
  ......
    ret = gimplify_self_mod_expr (expr_p, pre_p, post_p, fallback != fb_none, TREE_TYPE (*expr_p));
    break;
  ......
  case CALL_EXPR:    /* 函数调用的gimplify转换，如函数体中解析printf的调用会走到这里 */
    ret = gimplify_call_expr (expr_p, pre_p, fallback != fb_none);
    /* CALL_EXPR的执行结果肯定是一个右值,而如果此时需要返回一个左值节点,那么肯定是要构建一个临时变量记录CALL_EXPR的执行结果的. */
    if (fallback == fb_lvalue)
      {
        *expr_p = get_initialized_tmp_var (*expr_p, pre_p, post_p, false);
        mark_addressable (*expr_p);
        ret = GS_OK;
      }
    break;
  case COMPOUND_EXPR:    /* 复合表达式的转换 */
    ret = gimplify_compound_expr (expr_p, pre_p, fallback != fb_none);
    break;
  case MODIFY_EXPR:    /* 赋值表达式的转换，如x=1;就是个MODIFY_EXPR; int x=1;就是个INIT_EXPR，此外临时变量的初始化也会新增INIT_EXPR */
  case INIT_EXPR:
    ret = gimplify_modify_expr (expr_p, pre_p, post_p,
                    fallback != fb_none);
    break;
  case ADDR_EXPR:         /* 地址表达式的处理(一个 CALL_EXPR 节点的op[1]记录其调用的函数,而这个函数记录通常会在一个ADDR_EXPR中) */
    ret = gimplify_addr_expr (expr_p, pre_p, post_p);
    break;
  case NOP_EXPR:    /* 代表类型转换的表达式 */
  case CONVERT_EXPR:
        ......
    ret = gimplify_expr (&TREE_OPERAND (*expr_p, 0), pre_p, post_p, is_gimple_val, fb_rvalue);
    recalculate_side_effects (*expr_p);
    break;
  case INDIRECT_REF:      /* 间接引用的处理 */
    ......
  case MEM_REF:           /* 内存引用的处理 */
        ......
  case INTEGER_CST:       /* 常量表达式的处理 */
  case REAL_CST:
  case FIXED_CST:
  case STRING_CST:
  case COMPLEX_CST:
  case VECTOR_CST:
    if (TREE_OVERFLOW_P (*expr_p))  *expr_p = drop_tree_overflow (*expr_p);
    ret = GS_ALL_DONE;
    break;
  case CONST_DECL:        /* 常量声明的处理 */
    if (fallback & fb_lvalue) ret = GS_ALL_DONE;
    else
      {
        *expr_p = DECL_INITIAL (*expr_p); ret = GS_OK;
      }
    break;
  case DECL_EXPR:         /* 声明表达式的处理，block中的声明会以声明表达式的形式出现在语句列表中,如int x;*/
    ret = gimplify_decl_expr (expr_p, pre_p);
    break;
  case BIND_EXPR:        /* BIND_EXPR的转换(也就是block的转换) */
    ret = gimplify_bind_expr (expr_p, pre_p);
    break;
  case LOOP_EXPR:         /* 循环表达式的处理 */
    ret = gimplify_loop_expr (expr_p, pre_p);
    break;
  case SWITCH_EXPR:       /* switch的处理 */
    ret = gimplify_switch_expr (expr_p, pre_p);
    break;
  case EXIT_EXPR:
    ret = gimplify_exit_expr (expr_p);
    break;
  case GOTO_EXPR:         /* goto表达式的处理 */
      ......
    gimplify_seq_add_stmt (pre_p, gimple_build_goto (GOTO_DESTINATION (*expr_p)));
    ret = GS_ALL_DONE;
    break;
  case PREDICT_EXPR:      /* 分支预测 */
    gimplify_seq_add_stmt (pre_p, gimple_build_predict (PREDICT_EXPR_PREDICTOR (*expr_p), PREDICT_EXPR_OUTCOME (*expr_p)));
    ret = GS_ALL_DONE;
    break;
  case LABEL_EXPR:        /* label表达式处理函数 */
    ret = gimplify_label_expr (expr_p, pre_p);
    label = LABEL_EXPR_LABEL (*expr_p);
          ......
    break;
  case CASE_LABEL_EXPR:   /* case表达式处理函数 */
    ret = gimplify_case_label_expr (expr_p, pre_p);
    if (gimplify_ctxp->live_switch_vars) asan_poison_variables (gimplify_ctxp->live_switch_vars, false, pre_p);
    break;
  case RETURN_EXPR:       /* 返回表达式的处理，如一个return 0; */
    ret = gimplify_return_expr (*expr_p, pre_p);
    break;
  case CONSTRUCTOR:       /* 构造函数的处理 */
        ......
```

```
    ......
    case ASM_EXPR:                  /* ASM表达式的处理 */
      ret = gimplify_asm_expr (expr_p, pre_p, post_p);
      break;
    ......
    case STATEMENT_LIST:            /* 语句链表的处理,如处理到一个{}后，里面会出现一个语句链表 */
      ret = gimplify_statement_list (expr_p, pre_p);
      break;
    case VAR_DECL:
    case PARM_DECL:         /* 变量或参数声明节点表达式的处理 */
      ret = gimplify_var_or_parm_decl (expr_p);    /* 这里基本直接返回 GS_ALL_DONE即可,不需要做什么处理 */
      break;
    case RESULT_DECL:       /* 返回值声明节点的处理 */
      if (gimplify_omp_ctxp) omp_notice_variable (gimplify_omp_ctxp, *expr_p, true);
      ret = GS_ALL_DONE;
      break;
    case SSA_NAME:          /* SSA_NAME */
      ret = GS_ALL_DONE;
      break;
    ......
    default:                /* 其他的case都在这里处理 */
      switch (TREE_CODE_CLASS (TREE_CODE (*expr_p)))
        {
        case tcc_comparison:    /* 比较类表达式的处理 */
                    ......
        case tcc_unary:         /* 单目运算的处理(实际上就是gimplify单目运算对应的树节点,然后返回一个右值) */
          ret = gimplify_expr (&TREE_OPERAND (*expr_p, 0), pre_p, post_p, is_gimple_val, fb_rvalue);
          break;
        case tcc_binary:        /* 双目运算的处理 */
                  ......
          break;
        case tcc_declaration:  /* 其他声明和常量节点均不处理 */
        case tcc_constant:
          ret = GS_ALL_DONE;
          goto dont_recalculate;
        default:                /* 其他case报错 */
          gcc_unreachable ();
        }
      recalculate_side_effects (*expr_p);
    dont_recalculate:
      break;
    }
  } while (ret == GS_OK);  /* 如果处理结果返回的是GS_OK,那么需要循环再次处理; 如果返回GS_ALL_DONE则代表处理完毕 */

  /* 如果当前解析过程不需要返回值,而解析结果(重写入expr_p中了) 还是个有返回值的非语句节点,那么递归解析此节点后expr_p返回NULL_TREE */
  if (fallback == fb_none && *expr_p && !is_gimple_stmt (*expr_p))
    {
        ......
        *expr_p = NULL;
    }

  if (fallback == fb_none || is_statement)    /* 若当前正在处理一条语句,或当前调用没有要求返回值 */
    {
      *expr_p = NULL_TREE;     /* 直接置空 expr_p */
      goto out;
    }

    /* 到这里说明当前解析需要返回值,且当前解析的不是一个statement语句, 此时若当前解析的返回值(expr_p)满足gimple_test_f测试条件则直接返回(前提是没有post_p)
  if (gimple_seq_empty_p (internal_post) && (*gimple_test_f) (*expr_p))
    goto out;
  /*
      如果表达式要求返回一个左值节点,且当前表达式又是可以被获取地址的(如VAR_DECL)
     这个case编译了部分kernel都没看到过..先pass
      这里也说明能为其生成左值节点的表达式必须满足 is_gimple_addressable
  */
    /* 如当前需要返回一个左值节点,且返回的结果表达式(expr_p)是可获取地址的(is_gimple_addressable),则为其构建一个内存引用节点(MEM_REF)作为左值并返回 */
  if ((fallback & fb_lvalue) && gimple_seq_empty_p (internal_post) && is_gimple_addressable (*expr_p))
    {
      ......
      *expr_p = build2 (MEM_REF, ref_type, tmp, build_zero_cst (ref_alias_type));
    }
  /* 若当前需要返回一个右值节点,且返回的结果表达式(expr_p)可以作为右值 (is_gimple_reg_rhs_or_call),
    则将expr_p的当前值赋值给一个临时变量记录expr_p的当前值, 并返回此临时变量节点到expr_P作为新的右值 */
  else if ((fallback & fb_rvalue) && is_gimple_reg_rhs_or_call (*expr_p))
    {
      ......
      *expr_p = get_formal_tmp_var (*expr_p, pre_p);
    }
  else {     /* 其他情况均错误  */
     ret = GS_ERROR;
     goto out;
  }
  gcc_assert ((*gimple_test_f) (*expr_p));            /* 确保返回值节点是满足 gimple_test_f的 */

  if (!gimple_seq_empty_p (internal_post))        /* 将最后的post加到 pre_p指令序列中,并为这些指令添加源码位置 */
    {
      annotate_all_with_location (internal_post, input_location);
      gimplify_seq_add_seq (pre_p, internal_post);
    }

out:
  return ret;
}
```

gimplify_expr做为表达式gimple高端化的case分配函数,其中包含了各种表达式case的处理方式,而**此函数的整体逻辑就是**gimplify **expr_p**指向的表达式,此过程中生成的指令序列全部记录到pre_p gimple语句序列中; 若表达式最终有返回值则重新返回到expr_p中,解析状态以函数返回值的形式返回. 这里需要注意的是关于expr_p这个返回值节点:

- 若当前gimplify_expr分析的是一条语句(statement),或者调用时没有要求返回值(fb_none)(目前看到二者是同时出现的，没有单独出现的case), 那么最终即使解析完毕后*expr_p非空,则也要将其置空后返回.
- 若当前gimplify_expr分析的不是语句且要求了返回值:
    - 若解析后的expr_p本身就满足 gimple_test_f,那么直接返回expr_p即可.
    - 若解析后的expr_p不满足gimple_test_f:
        - 若当前要求返回一个左值(fb_lvalue),且expr_p节点可以被获取到地址,则构建一个对expr_p的MEM_REF引用并返回.
        - 若当前要求返回一个右值(fb_rvalue),且expr_p可作为一个右值节点,则构建一个临时变量(或SSA_NAME)记录expr_p的当前值作为右值返回.
        - 其他情况均报错.

后续的代码则依次分析几个gimplify_expr中主要表达式的处理流程

五、gimplify_bind_expr

在gimplify_expr中,若解析到BIND_EXPR,则会调用gimplify_bind_expr函数,其定义大致如下:

```
gimplify_status gimplify_expr (tree *expr_p, gimple_seq *pre_p, gimple_seq *post_p, bool (*gimple_test_f) (tr`ee), fallback_t fallback)
{
    ......
    case ADDR_EXPR:
      ret = gimplify_addr_expr (expr_p, pre_p, post_p);
      break;
    ......
}

struct gbind : public gimple    /* gbind继承自gimple类型,此外只额外多了三个元素 */
{
  tree vars;                /* vars 正常继承自 block.vars, 而如果当前block是一个函数的函数体(而不是嵌套subblock)时, 函数gimplify完毕后会将当前
                               block和subblock中所有动态生成的变量都加入到这里面(gimplify_ctxp->temps), 这里会加入subblock的原因是因为栈分配时
                               在函数入口出口完成的,故subblock的分配也算在这里 */
  tree block;               /* 当前gbind 所对应的那个{}对应的block结构体 */
  gimple_seq body;          /* 当前gbind对应的{}内部语句链表解析出的gimple 语句序列 */
};

enum gimplify_status gimplify_bind_expr (tree *expr_p, gimple_seq *pre_p)
{
  tree bind_expr = *expr_p;
  tree t;
  gbind *bind_stmt;

  /* 遍历BIND_EXPR所代表的的当前block的变量声明链表(block.vars) */
  for (t = BIND_EXPR_VARS (bind_expr); t ; t = DECL_CHAIN (t)) {
      if (VAR_P (t)) {
          DECL_SEEN_IN_BIND_EXPR_P (t) = 1;     /* 标记已经看到了当前变量,此值主要是为了区分后续动态创建的变量的  */
      /* 若局部变量指定了特殊的寄存器则会标记此位,  如 register int x asm("x18"); */
      if (DECL_HARD_REGISTER (t) && !is_global_var (t) && cfun)
        cfun->has_local_explicit_reg_vars = true;
      }
  }

   /* 创建一个GIMPLE_BIND节点(gbind),代表gimplify后的BIND_EXPR表达式, 其变量链表和对应block已经随创建初始化了 */
  bind_stmt = gimple_build_bind (BIND_EXPR_VARS (bind_expr), NULL, BIND_EXPR_BLOCK (bind_expr));
  body = NULL;
  /* gimplify BIND_EXPR中的body节点,通常 BIND_EXPR代表一个{}(block)内的代码,而BIND_EXPR的body则记录了其中所有的语句列表,通常应该是一个
     tree_statement_list,而此函数最终则将是解析后生成的gimple指令序列返回到 body变量中.
  */
  gimplify_stmt (&BIND_EXPR_BODY (bind_expr), &body);

  /* body中记录 BIND_EXPR.body节点gimplify后生成的指令序列,将其记录到gbind.body中 */
  gimple_bind_set_body (bind_stmt, body);

  //到这里这个gbind节点是已经生成好了
  ......
  /* 如果当前函数使用了可变长数组,但没调用__buildin_alloca函数,则为其增加前后的桩代码 */
  if (gimplify_ctxp->save_stack && !gimplify_ctxp->keep_stack)
    {
         .......
    }

  gimplify_seq_add_stmt (pre_p, bind_stmt);    /* 将 BIND_EXPR解析出的语句序列 插在 pre_p gimple的后面 */
  *expr_p = NULL_TREE;    /* BIND_EXPR是不需要返回值的,故解析玩不后无条件将expr_p设置为空,并返回全局解析完成(GS_ALL_DONE) */
  return GS_ALL_DONE;
}
```

此函数主要为BIND_EXPR(expr_p)创建了一个gbind语句并添加到pre_p指令序列中,此过程中会将:

- BIND_EXPR中显式声明的变量(vars)记录到gbind.vars上
- BIND_EXPR对应的block记录到gbind.block上
- BIND_EXPR整个树节点(包括子树节点)gimplify后的结果记录到gbind.body中.

BIND_EXPR的gimple高端化举例如下:

```
int func()
{
   int x;
   x = 1;
   {
       int y;
       y = 2;
   }
}

//gimplie高端化后的函数在 *.005t.gimple中可见(设置 -fdump-tree-all-raw参数)
func ()
gimple_bind <            //每一个{} 对应一个gbind节点
  int x;
  gimple_assign <integer_cst, x, 1, NULL, NULL>
  gimple_bind <          //内层的gbind节点
    int y;
    gimple_assign <integer_cst, y, 2, NULL, NULL>
  >
>
```

六、gimplify_statement_list

通常来说BIND_EXPR的子节点是一个STATEMENT_LIST,代表一个语句链表,在gimplify_expr中则通过函数gimplify_statement_list处理:

```
gimplify_status gimplify_expr (tree *expr_p, gimple_seq *pre_p, gimple_seq *post_p, bool (*gimple_test_f) (tr`ee), fallback_t fallback)
{
    ......
    case STATEMENT_LIST:
      ret = gimplify_statement_list (expr_p, pre_p);
      break;
    ......
}

enum gimplify_status gimplify_statement_list (tree *expr_p, gimple_seq *pre_p)
{
  ......
  tree_stmt_iterator i = tsi_start (*expr_p);
  while (!tsi_end_p (i))    /* 判断是否遍历到了tree_statement_list的最后一条语句 */
    {
      /* 此函数处理一条 AST语句,并将处理后生成的指令序列记录到pre_p中, 语句不需要返回值,故这里也不用关心返回值节点 */
      gimplify_stmt (tsi_stmt_ptr (i), pre_p);
      tsi_delink (&i);    /* 从statment list中将此tree节点下链 */
    }
  return GS_ALL_DONE;
}

/* stmt(语句)并非某个类型的节点,而是多种节点的集合,其是表达式的一部分,故这里使用gimplify_expr通过传入 is_gimple_stmt + fb_none 可以用来解析一条语句 */
bool gimplify_stmt (tree *stmt_p, gimple_seq *seq_p)
{
  gimple_seq_node last;
  last = gimple_seq_last (*seq_p);     /* 获取已有gimple指令序列 seq_p中最后一条gimple指令地址 */
  /* 此函数负责解析一条语句(stmt), 此函数返回后stmt_p应该为空, seq_p指令序列的末尾会附加当前stmt的解析结果,期间生成的:
    * 所有临时变量都被添加到全局变量gimplify_ctxp->temps中
    * 所有ssa_name都被添加到当前函数的fn->gimplf_df->ssa_names中 */
  gimplify_expr (stmt_p, seq_p, NULL, is_gimple_stmt, fb_none);
  return last != gimple_seq_last (*seq_p);    /* 如果新增了指令序列则返回true, 未新增则返回false */
}
```

此函数主要负责对一个语句链表(STATEMENT_LIST)的gimplify,由于语句链表是不需要返回值的，故最终**只需要将所有gimplify的解析结果记录到seq_p这个gimple指令序列中返回即可**.

七、gimplify_decl_expr

gimplify_decl_expr函数用来gimplify一个声明表达式,声明表达式(DECL_EXPR)实际上代表的是非file_scope(函数体内部以及其subblock内部)的一个声明,如:

```
int func()
```

```
{
   int x;
}

// 见 *.004t.orginal
// bind_expr的body就是一个 decl_expr(因为就一条语句所以没有statementlist), 代表一个声明表达式.
@1      bind_expr       type: @2       vars: @3       body: @4
@2      void_type       name: @5       algn: 8
@3      var_decl        name: @6       type: @7       scpe: @8
                        srcp: 1.c:3                   size: @9
                        algn: 32       used: 0
@4      decl_expr       type: @2        //这里没显示,实际上这里DECL_EXPR_DECL (stmt)记录的是一个VAR_DECL节点,也就是x的声明节点.
```

和外部声明中的变量声明不同(外部声明中的变量声明会直接导致全局变量节点varpool_node的创建), block中的变量声明除了会使变量被添加到block.vars中，还会在语句链表中添加一条DECL_EXPR,因为与全局变量不同,**局部变量在每次函数执行时都要赋初值或动态计算空间大小,故需要在语句链表中主线一个DECL_EXPR表达式，以确保在gimplify阶段生成对应的初始化语句**,其代码如下:

```
/* 此函数用来处理函数内部的声明表达式,主要工作包括将变量的初值转化为对变量的赋值gimple语句,以及处理变长数组等 */
static enum gimplify_status gimplify_decl_expr (tree *stmt_p, gimple_seq *seq_p)
{
  tree stmt = *stmt_p;
  tree decl = DECL_EXPR_DECL (stmt);    /* DECL_EXPR中只在.operands[0]中记录一个 XXX_DECL声明节点,这边是其全部内容 */
  *stmt_p = NULL_TREE;                  /* 清空指针,声明表达式不需要返回表达式节点 */

  /* !external的VAR_DECL代表局部变量，此函数只对局部变量有处理,非局部变量不处理直接返回 GS_ALL_DONE */
  if (VAR_P (decl) && !DECL_EXTERNAL (decl))
  {
      tree init = DECL_INITIAL (decl);
      bool is_vla = false;

      if (TREE_CODE (DECL_SIZE_UNIT (decl)) != INTEGER_CST ....    /*  可变长数组的gimplify */
    {
      gimplify_vla_decl (decl, seq_p);
      is_vla = true;
    }
        ......

      if (init && init != error_mark_node)    /* 若变量有初值节点 */
    {
      if (!TREE_STATIC (decl))      /* 若非静态局部变量 */
        {
          DECL_INITIAL (decl) = NULL_TREE;    /* 将声明节点的初值节点清空 */
          init = build2 (INIT_EXPR, void_type_node, decl, init);    /* 构建一个INIT_EXPR代表此局部变量的初始化 */
          gimplify_and_add (init, seq_p);    /* gimplify 此INIT_EXPR表达式,并将结果添加到seq_p队列中 */
          ggc_free (init);    /* 转化为gassign语句后, free 掉 INIT_EXPR */
        }
              ......
    }
  }
  return GS_ALL_DONE;
}
```

声明表达式处理过程中一个主要的操作是将**声明节点(若有初值)转化为一条gimple赋值语句(gassign)**,如:

```
x = 1;
//最终会被转化为
//gimple_assign <integer_cst, x, 1, NULL, NULL>
```

八、gimplify_var_or_parm_decl

gimplify_var_or_parm_decl负责对VAR_DECL/PARM_DECL节点的处理,VAR_DECL和PARM_DECL通常都是末端节点，gimplify_var_or_parm_decl函数基本可以认为是空函数,**当递归gimplify到VAR_DECL/PARM_DECL时通常也就代表着当前表达式的gimplify结束了**(前面的DECL_EXPR在没有初值的时候也是作为末端节点存在的)

九、gimplify_call_expr

gimplify_call_expr负责对CALL_EXPR节点的gimplify,一个CALL_EXPR代表源码中一个函数调用(不论是直接的还是间接的函数调用),其代码如下:

```
gimplify_status gimplify_expr (tree *expr_p, gimple_seq *pre_p, gimple_seq *post_p, bool (*gimple_test_f) (tr`ee), fallback_t fallback)
{
    ......
    case CALL_EXPR:
      ret = gimplify_call_expr (expr_p, pre_p, fallback != fb_none);
      if (fallback == fb_lvalue) {    /* 函数表达式解析后要返回左值的case暂未遇到过 */
          *expr_p = get_initialized_tmp_var (*expr_p, pre_p, post_p, false);
          mark_addressable (*expr_p);
          ret = GS_OK;
      }
      break;
    ......
```

```
}

/*
   expr_p:需要解析的CALL_EXPR表达式
   pre_p: 存储结果的gimple指令序列
   want_value:代表此函数调用是否需要返回值
  此函数负责gimplify一条CALL_EXPR指令,其首先会gimplify此CALL_EXPR的被调用函数节点FN,然后依次gimplify当前调用的所有实参,之后则根据是否需要返回值来决定如何处
  * 若此CALL_EXPR不需要返回值,则直接将此CALL_EXPR gimplify为gcall节点并添加到pre_p队列中, expr_p返回NULL_TREE(代表不需要返回值,且解析完毕)
  * 若此CALL_EXPR需要返回值,则这里只是将为FN包裹一个NOP_EXPR,不生成gcall指令,然后expr_p原样返回此CALL_EXPR,真正的gcall指令则是在上一层gimplify中完成(通常
*/
static enum gimplify_status gimplify_call_expr (tree *expr_p, gimple_seq *pre_p, bool want_value)
{
  tree fndecl, parms, p, fnptrtype;
  enum gimplify_status ret;
  int i, nargs;
  gcall *call;
  bool builtin_va_start_p = false;

  location_t loc = EXPR_LOCATION (*expr_p);
  gcc_assert (TREE_CODE (*expr_p) == CALL_EXPR);

  if (CALL_EXPR_FN (*expr_p) == NULL_TREE)    /* CALL_EXPR_FN为空,则函数通过编号ID的形式记录在CALL_EXPR_IFN中, 这里则是对gcc内置函数的调用 */
    {
      if (want_value)                /* 若内置函数需要返回值,则到其父语句中处理(如MODIFY_EXPR) */
        return GS_ALL_DONE;
      /* 如果不需要返回值,则直接将CALL_EXPR转换为gcall指令并返回 */
      nargs = call_expr_nargs (*expr_p);     /* 根据CALL_EXPR结构体大小获取此函数调用的参数个数 */
      enum internal_fn ifn = CALL_EXPR_IFN (*expr_p);    /* 获取内置函数编号 */

      auto_vec<tree> vargs (nargs);
      for (i = 0; i < nargs; i++) {        /* 遍历所有参数并对所有参数节点进行gimplify */
          gimplify_arg (&CALL_EXPR_ARG (*expr_p, i), pre_p, EXPR_LOCATION (*expr_p));
          vargs.quick_push (CALL_EXPR_ARG (*expr_p, i));        /* push参数到vargs数组中 */
      }

      gcall *call = gimple_build_call_internal_vec (ifn, vargs);    /* 根据被调用函数ifn, 实参数组vargs,为此函数调用构建gcall指令 */
      ......
      gimplify_seq_add_stmt (pre_p, call);        /* 将gcall指令添加到当前pre_p指令序列中 */

      return GS_ALL_DONE;            /* 返回处理完毕,此时的expr_p还是只想当前CALL_EXPR */
    }
  /* 尝试获取当前CALL_EXPR调用的目标函数(若获取到,则此处的代码可以被展为直接调用),若目标函数未决或非函数(如某个内存地址)则这里通常会返回空. */
  fndecl = get_callee_fndecl (*expr_p);

  if (fndecl && fndecl_built_in_p (fndecl, BUILT_IN_NORMAL))     /* 若被调用函数是一个gcc builtin的函数(builtin和内置函数是两个东西) */
    /* 则根据不同的 builtin类型来做不同的处理,这里实际上只处理了va_start 和__builtin_alloca函数的调用(因为需要处理标记后续如何使用栈相关flag) */
    switch (DECL_FUNCTION_CODE (fndecl))
    {
      CASE_BUILT_IN_ALLOCA: .......
      case BUILT_IN_VA_START: ......
      default:  ;
    }
  }
  /*
     不论根据*expr_p是否找到了被调用函数, 此 CALL_EXPR中的FN节点的类型节点,一定记录着被调用函数的类型,如对于:
     * FN为 ADDR_EXPR ,则ADDR_EXPR就记录着函数指针的类型
     * FN为 VAR_DECL, 则此VAR_DECL也一定是个函数指针,其类型也是函数指针的类型.
  */
  fnptrtype = TREE_TYPE (CALL_EXPR_FN (*expr_p));

  /* 递归处理CALL_EXPR中代表被调用函数的表达式节点,此函数完成后&CALL_EXPR_FN (*expr_p)才可在gcall表达式中作为被调用函数出现,若在计算被调用函数的过程中需要生
    若&CALL_EXPR_FN (*expr_p)的返回值满足is_gimple_call_addr,则无需处理,否则要新建一个右值节点来作为被调用函数. */
  ret = gimplify_expr (&CALL_EXPR_FN (*expr_p), pre_p, NULL,is_gimple_call_addr, fb_rvalue);

  nargs = call_expr_nargs (*expr_p);    /* 根据此CALL_EXPR节点大小计算出此call表达式的【实参】个数 */

  fndecl = get_callee_fndecl (*expr_p);    /* 这里再次获取函数的声明节点(前面执行过gimplify_expr,可能导致被调用函数被算出来了) */
  parms = NULL_TREE;

  if (fndecl)    /* 如果被调用函数是确定的,则直接去函数声明的类型节点去找参数类型链表 */
    parms = TYPE_ARG_TYPES (TREE_TYPE (fndecl));
  else /* 如果被调用函数是不确定的,则去指针的类型节点中去找参数类型链表 */
    parms = TYPE_ARG_TYPES (TREE_TYPE (fnptrtype));
  /* 最终的p(形参链表)实际上是先尝试从函数声明中找(这里面可能会有默认值) 没有再从函数或函数指针(fnptrtype)的类型声明中找,若二者都没有则为空 */
  if (fndecl && DECL_ARGUMENTS (fndecl))
    p = DECL_ARGUMENTS (fndecl);
  else if (parms)
    p = parms;
  else p = NULL_TREE;

  for (i = 0; i < nargs && p; i++, p = TREE_CHAIN (p)) ;   /* 这里是获取实参和形参中参数个数的最小值 */

  if (nargs > 0)    /* 对此次函数调用的所有实参做gimplify */
    {
      for (i = (PUSH_ARGS_REVERSED ? nargs - 1 : 0);  PUSH_ARGS_REVERSED ? i >= 0 : i < nargs;
           PUSH_ARGS_REVERSED ? i-- : i++)    /* 遍历所有参数 */
        {
          enum gimplify_status t;
          if ((i != 1) || !builtin_va_start_p)
            {
              /* gimplify 此函数的所有实参表达式,每个实参节点gimplify后的返回值保存在CALL_EXPR的参数链表中(而不是函数声明中) */
              t = gimplify_arg (&CALL_EXPR_ARG (*expr_p, i), pre_p, EXPR_LOCATION (*expr_p), ! returns_twice);
            }
```

```
        }
    }

  /*
     如当前CALL_EXPR并不需要返回值,那么此CALL_EXPR到这里就算解析完毕了,直接生成gcall指令,其对应的返回值节点(gcall.op[0])为空
     如当前CALL_EXPR需要返回值,那么这里就先不展开,而是到其上层的表达式中展开，因为如 m =func(); 实际上最终被展开为一条gcall指令,
    且返回值节点设置为m即可,不必为其单独创建一个返回值节点,并多一步赋值操作.
    */
  if (!want_value)
    {
      gimple_stmt_iterator gsi;
      call = gimple_build_call_from_tree (*expr_p, fnptrtype);     /* 为CALL_EXPR创建一个 gcall节点 */
      notice_special_calls (call);
      gimplify_seq_add_stmt (pre_p, call);        /* 将此gcall节点添加到 gimple语句链表 */
      gsi = gsi_last (*pre_p);
      maybe_fold_stmt (&gsi);
      *expr_p = NULL_TREE;                        /* 生成成功则expr_p设置为空 */
    }
  else
    /* 如果需要返回值则此处不生成gcall指令,而是将这条CALL_EXPR的FN包裹一层NOP_EXPR并还是返回输出的CALL_EXPR */
    CALL_EXPR_FN (*expr_p) = build1 (NOP_EXPR, fnptrtype, CALL_EXPR_FN (*expr_p));
  return ret;
}
```

此函数负责**gimplify一条CALL_EXPR指令,其首先会gimplify此CALL_EXPR的被调用函数节点FN,然后依次gimplify当前调用的所有实参,之后则根据是否需要返回值来决定如何处理**:

- 若此CALL_EXPR不需要返回值,则直接将此CALL_EXPR gimplify为gcall节点并添加到pre_p队列中, expr_p返回NULL_TREE(代表不需要返回值,且解析完毕)
- 若此CALL_EXPR需要返回值,则这里只是将为FN包裹一个NOP_EXPR,不生成gcall指令,然后expr_p原样返回此CALL_EXPR,真正的gcall指令则是在上一层gimplify中完成(通常是MODIFY_EXPR,主要原因是需要返回值时CALL_EXPR写入的返回值节点当前不确定,需要在上层确定)

十、gimplify_modify_expr

gimplify_modify_expr 用来解析一个MODIFY_EXPR/INIT_EXPR表达式,二者实际上都是赋值表达式,其代码如下:

```
gimplify_status gimplify_expr (tree *expr_p, gimple_seq *pre_p, gimple_seq *post_p, bool (*gimple_test_f) (tr`ee), fallback_t fallback)
{
    ......
    case MODIFY_EXPR:
    case INIT_EXPR:
      ret = gimplify_modify_expr (expr_p, pre_p, post_p, fallback != fb_none);
      break;
    ......
}

/*
    此函数负责gimplify一个MODIFY_EXPR/INIT_EXPR表达式节点,其首先分别gimplify赋值表达式的左侧节点(to_p)和右侧节点(from_p),然后根据右侧节点的类型决定如何产
    * 如果右侧节点是一个CALL_EXPR,则创建一个 gcall指令,并将gcall的返回值直接设置为to_p
    * 如果右侧节点非CALL_EXPR,则创建一个gassign指令,并将赋值指令的左操作数设置为to_p
   最终:
   * 若want_value为true,则此表达式需要返回返回值到expr_p,若to_p没有副作用则返回fork的to_p,若有副作用则返回from_p.
   * 若want_value为false,则此表达式不需要返回返回值,expr_p设置为NULL_TREE.
*/
static enum gimplify_status gimplify_modify_expr (tree *expr_p, gimple_seq *pre_p, gimple_seq *post_p, bool want_value)
{
  tree *from_p = &TREE_OPERAND (*expr_p, 1);    /* INIT_EXPR / MODIFY_EXPR节点的右操作数节点作为from_p,其含义是这个值来自哪里 */
  tree *to_p = &TREE_OPERAND (*expr_p, 0);      /* INIT_EXPR / MODIFY_EXPR节点的左操作数作为to_p,代表这个值要赋值给哪里 */
  gimple *assign;

  /* gimplify MODIFY_EXPR/INIT_EXPR的右值节点, 如右值是一个CALL_EXPR,则这里就返回此CALL_EXPR节点 */
  ret = gimplify_expr (from_p, pre_p, post_p, initial_pred, fb_rvalue);

  /* gimplify MODIFY_EXPR/INIT_EXPR的左值节点, 最终返回的to_p要能作为一个左值*/
  ret = gimplify_expr (to_p, pre_p, post_p, is_gimple_lvalue, fb_lvalue);

  /* 若右侧节点为一个CALL_EXPR, 则要将整个MODIFY_EXPR/INIT_EXP gimplify为 gcall指令 */
  if (TREE_CODE (*from_p) == CALL_EXPR)
    {
      gcall *call_stmt;
      if (CALL_EXPR_FN (*from_p) == NULL_TREE)  {   /* 如果是gcc内置函数则走这里处理 */
       ......
      }
      else
      {
        tree fnptrtype = TREE_TYPE (CALL_EXPR_FN (*from_p));
        CALL_EXPR_FN (*from_p) = TREE_OPERAND (CALL_EXPR_FN (*from_p), 0);  /* 这里通常会获取到一个NOP_EXPR */
        STRIP_USELESS_TYPE_CONVERSION (CALL_EXPR_FN (*from_p));            /* 去除外层NOP_EXPR*/
        tree fndecl = get_callee_fndecl (*from_p);      /* 尝试获取被调用函数,对于间接调用，这里可能获取为空 */
                ......
        call_stmt = gimple_build_call_from_tree (*from_p, fnptrtype);    /* 正常走这里,为CALL_EXPR构建 gcall节点并返回到 call_stmt */
      }

```

```
48.       if (!gimple_call_noreturn_p (call_stmt) || !should_remove_lhs_p (*to_p))
49.         gimple_call_set_lhs (call_stmt, *to_p);             /* 设置 to_p 为接受当前函数返回值的左值节点 */
50.       else if (TREE_CODE (*to_p) == SSA_NAME)
51.         SSA_NAME_DEF_STMT (*to_p) = gimple_build_nop ();
52.       assign = call_stmt;                   /* gcall指令最终记录到 assign 中 */
53.     }
54.   else
55.     {
56.       assign = gimple_build_assign (*to_p, *from_p);        /* 对于右值非CALL_EXPR的赋值指令则直接构建gassign节点 */
57.       .......
58.     }
59.
60.   gimplify_seq_add_stmt (pre_p, assign);    /* 将assign 这个 gimple节点插入到 pre_p的最后 */
61.
62.   if (want_value)
63.     {
64.       *expr_p = TREE_THIS_VOLATILE (*to_p) ? *from_p : unshare_expr (*to_p);    /* 若目标值有副作用,则返回原始值,否则返回目标值 的树节点*/
65.       return GS_OK;     /* 这里返回 GS_OK, 代表返回的expr_P也要递归走一遍 gimplify_expr的处理 */
66.     }
67.   else
68.     *expr_p = NULL;    /* 不需要返回值则直接清空expr_p */
69.   return GS_ALL_DONE;     /* 并返回ALL_DONE */
70. }
```


此函数负责gimplify一个MODIFY/INIT_EXPR表达式节点,其首先分别gimplify赋值表达式的左侧节点(to_p)和右侧节点(from_p),然后根据右侧节点的类型决定如何产生指令:

- 如果右侧节点是一个CALL_EXPR,则创建一个 gcall指令,并将gcall的返回值直接设置为to_p
- 如果右侧节点非CALL_EXPR,则创建一个gassign指令,并将赋值指令的左操作数设置为to_p

最终:

- 若want_value为true,则此表达式需要返回返回值到expr_p,若to_p没有副作用则返回fork的to_p,若有副作用则返回from_p.
- 若want_value为false,则此表达式不需要返回返回值,expr_p设置为NULL_TREE.

MODIFY_EXPR/INIT_EXPR/CALL_EXPR的调用举例如下:

```
1. int func()
2. {
3.     int x;
4.     x = 1;
5.     x = func1();
6.     func1();
7. }
8.
9. func ()
10. gimple_bind <
11.   int x;
12.   gimple_assign <integer_cst, x, 1, NULL, NULL>
13.   gimple_call <func1, x>
14.   gimple_call <func1, NULL>
15. >
```

十一、gimplify_return_expr

此函数负责gimplify一个RETURN_EXPR,在源码中的每一条return语句都会对应到一个RETURN_EXPR,如:

```
1. int func()
2. {
3.     return func1();
4.     return 1;
5.     return;
6. }
7.
8. func ()
9. gimple_bind <
10.   int D.3411;        /* 此变量实际上并没有名字,而是fdump时对于此类变量自动设置为D.变量的全局id */
11.   gimple_call <func1, D.3411>
12.   gimple_return <D.3411>
13.   gimple_assign <integer_cst, D.3411, 1, NULL, NULL>
14.   gimple_return <D.3411>
15.   gimple_return <NULL>
16. >
```


此函数的源码如下:

```
1. gimplify_status gimplify_expr (tree *expr_p, gimple_seq *pre_p, gimple_seq *post_p, bool (*gimple_test_f) (tr`ee), fallback_t fallback)
2. {
```

```
    ......
    case RETURN_EXPR:
      ret = gimplify_return_expr (*expr_p, pre_p);
      break;
    ......
}

/*
    此函数负责对返回表达式(RETURN_EXPR)进行gimplify, 其先gimplify 作为返回值的整个右侧值节点,然后生成一个临时的RESULT_DECL记录返回值,最终构建一个greturn语
*/
static enum gimplify_status gimplify_return_expr (tree stmt, gimple_seq *pre_p)
{
  greturn *ret;
  /* 在gimplify之前, RETURN_EXPR的op[0]代表返回值的表达式节点,其可以是一个RESULT_DECL/MODIFY_EXPR/INIT_EXPR或NULL_TREE(函数无返回值). */
  tree ret_expr = TREE_OPERAND (stmt, 0);
  tree result_decl, result;

  /* 对于 return; 这样的语句,其ret_expr为空,此时直接向序列中添加greturn语句即可 */
  if (!ret_expr || TREE_CODE (ret_expr) == RESULT_DECL)
    {
      greturn *ret = gimple_build_return (ret_expr);
      gimplify_seq_add_stmt (pre_p, ret);
      return GS_ALL_DONE;
    }

  /* 如果函数的返回值类型是 void,则不管返回语句,直接设置result_decl为 NULL_TREE */
  if (VOID_TYPE_P (TREE_TYPE (TREE_TYPE (current_function_decl))))
    result_decl = NULL_TREE;
  else
    {
      /* 否则return语句的值节点实际上是一个 MODIFY_EXPR/INIT_EXPR表达式这里获取赋值表达式的to_p(右侧值)节点作为result_decl */
      result_decl = TREE_OPERAND (ret_expr, 0);
      ......
    }

  if (!result_decl)    /* 如果没有result_decl节点,则代表无需返回值 */
    result = NULL_TREE;
  else if (aggregate_value_p (result_decl, TREE_TYPE (current_function_decl)))    /* 如果返回的是一个结构体 */
    {
      ......
      result = result_decl;
    }
  else if (gimplify_ctxp->return_temp)
    result = gimplify_ctxp->return_temp;    /* 如果当前gimplify过程中已经创建了返回值节点,则直接使用 */
  else
    {
      result = create_tmp_reg (TREE_TYPE (result_decl));  /* 否则创建一个无名的临时变量作为函数的返回值 */
      gimplify_ctxp->return_temp = result;    /* 返回值的声明节点被记录到 全局gimplify上下文中 */
    }

  if (result != result_decl)
    TREE_OPERAND (ret_expr, 0) = result;    /* 重新设置此 RESULT_DECL为 MODIFY_EXPR/INIT_EXPR的返回值节点 */

  gimplify_and_add (TREE_OPERAND (stmt, 0), pre_p);    /* 对return语句的MODIFY_EXPR/INIT_EXPR表达式做gimpify */
    .......
  ret = gimple_build_return (result);    /* 最终生成一条greturn语句,其只有一个操作数result代表返回值的RESULT_DECL节点 */

  gimplify_seq_add_stmt (pre_p, ret);    /* 将GIMPLE_RETURN 语句, 加到 语句链表中 */
  return GS_ALL_DONE;
}
```

十二、gimplify_cond_expr

gimplify_cond_expr负责条件表达式的gimplify,在源码中的每一个条件表达式(如if语句)都会通过此函数解析, COND_EXPR最终会被转化为一条GIMPLE_COND语句,以及then,else分支的代码，GIMPLE_COND定义如下:

```
GIMPLE_COND <COND_CODE, OP1, OP2, TRUE_LABEL, FALSE_LABEL>
```

此GIMPLE指令共有5个操作数,其中:

- COND_CODE是比较操作的条件码, 如 EQ_EXPR/NE_EXPR(实际上是if(...)中...最终转化为的tree_code)
- OP1/OP2是此条件表达式最终比较的左侧值和右侧值节点
- TRUE_LABEL/FALSE_LABEL是两个标签声明(LABEL_DECL)节点, GIMPLE_COND在true或false分支触发时,并没有指定跳转到指令序列中的某条语句,而是指定了一个标签声明, 而标签声明在语句序列的什么位置则需要一条glabel语句来设置,在gimple指令序列中,一条glabel语句即是用来设置某个标签实际应该跳转到的位置的.

注: 条件表达式中,**true分支通常又叫做then分支(有的语言中是if..then的形式)，false分支通常称为else分支**.

gimplify_cond_expr的源码如下:

```
gimplify_status gimplify_expr (tree *expr_p, gimple_seq *pre_p, gimple_seq *post_p, bool (*gimple_test_f) (tr`ee), fallback_t fallback)
{
    ......
    case COND_EXPR:
      ret = gimplify_cond_expr (expr_p, pre_p, fallback);

```

```
      if (fallback == fb_lvalue)
        {
          *expr_p = get_initialized_tmp_var (*expr_p, pre_p, post_p, false);
          mark_addressable (*expr_p);
          ret = GS_OK;
        }
      break;
    ......
}

enum gimplify_status gimplify_cond_expr (tree *expr_p, gimple_seq *pre_p, fallback_t fallback)
{
  tree expr = *expr_p;
  tree tmp, arm1, arm2;
  tree label_true, label_false, label_cont;
  bool have_then_clause_p, have_else_clause_p;
  gcond *cond_stmt;
  gimple_seq seq = NULL;
    .......

  /* 先gimplify 条件表达式的内部条件，如 if(...) 中的... */
  ret = gimplify_expr (&TREE_OPERAND (expr, 0), pre_p, NULL, is_gimple_condexpr, fb_rvalue);
  have_then_clause_p = have_else_clause_p = false;

  /* 如果 if(...) 成立后执行的语句(除去调试指令)是一条goto label语句,那么if then后不需要单独创建一个新的标签,直接跳转到goto 后面的标签即可,相当于
    可以省略掉一条goto 语句,利用 gcond直接跳转到goto label对应的标签 */
  label_true = find_goto_label (TREE_OPERAND (expr, 1));
  if (label_true && ...
        ......
  else
    /* 正常情况下 then分支中的代码不是goto labelX,那么就会创建一个LABLE_DECL节点，在后面的GIMPLE_COND语句中若then分支成立则会跳转到此标签所在的位置
       LABEL_DECL只是记录了此标签的树节点,而一个标签的位置则是由glabel语句决定的 */
    label_true = create_artificial_label (UNKNOWN_LOCATION);

  label_false = find_goto_label (TREE_OPERAND (expr, 2));    /* else分支同理 */
  if (label_false && ...
    ...
  else
    label_false = create_artificial_label (UNKNOWN_LOCATION);

  /* 获取此条件表达式的比较代码，比较的左侧值(arm1)/右侧值(arm2) */
  gimple_cond_get_ops_from_tree (COND_EXPR_COND (expr), &pred_code, &arm1, &arm2);

  /* 构建 GIMPLE_COND语句,并插入到指令序列中，then分支会跳转到label_true标签，else分支会跳转到label_false标签，但需要注意的是此时二标签的位置还是不确定的
  cond_stmt = gimple_build_cond (pred_code, arm1, arm2, label_true, label_false);
  gimplify_seq_add_stmt (&seq, cond_stmt);

  gimple_stmt_iterator gsi = gsi_last (seq);
  label_cont = NULL_TREE;

  if (!have_then_clause_p)    /* 若then分支 中没找到goto label则走这里为gimple指令序列中添加 glabel语句 */
    {
          ......
      /* 前面构建GIMPLE_COND语句时已经确定了then分支,这里实际上就是在为then分支要跳转到的标签(label_true),构建glabel语句,此语句则代表label_true标签在gimp
      gimplify_seq_add_stmt (&seq, gimple_build_label (label_true));

      /* 将 then分支执行的代码 gimplify为指令序列并添加到seq中 */
      have_then_clause_p = gimplify_stmt (&TREE_OPERAND (expr, 1), &seq);

      /* then表达式分析完毕后,若当前COND_EXPR有else分支,则需要在then分支的结束增加一个goto label_cont的语句，而后续else分支gimplify之后的语句序列的末尾,
         即为label_cont要跳转的位置,需要添加一个glabel来标注 */
      if (!have_else_clause_p && ......
        {
          gimple *g;
          label_cont = create_artificial_label (UNKNOWN_LOCATION);
          g = gimple_build_goto (label_cont);
          gimplify_seq_add_stmt (&seq, g);
        }
    }

  if (!have_else_clause_p)    /* 若else 分支没找到label则继续为gimple指令序列添加glabel语句,这里是为label_false标记位置 */
    {
      /* then与执行完毕后,为else分支跳转的label_decl添加其对应的glabel指令，后续此glabel之后则继续添加 else中语句解析成的gimple指令序列 */
      gimplify_seq_add_stmt (&seq, gimple_build_label (label_false));
      have_else_clause_p = gimplify_stmt (&TREE_OPERAND (expr, 2), &seq);    /* gimplify else分支的代码 */
    }

  if (label_cont)     /* 当有else 时，else完事了还要再加上一个label，因为then语句最终要跳过else到此label上 */
    gimplify_seq_add_stmt (&seq, gimple_build_label (label_cont));

  gimple_seq_add_seq (pre_p, seq);    /* 将整个 then ...; goto label_cont; else ...; label_cont; gimplie指令序列添加到pre_p中 */

    ......
  *expr_p = NULL;
  return ret;
}
```