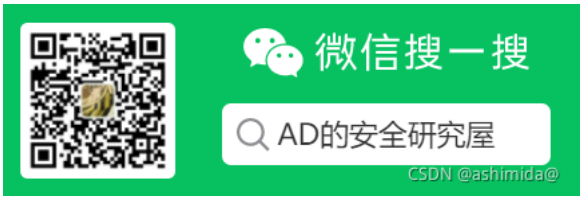
GCC源码分析(九) — gcc全局符号表与符号的分析(gimplify)

更多内容可关注微信公众号



一、全局符号表

在gcc中有一个全局变量 symbol_table *symtab; 此变量是用来记录整个编译过程中产生的所有函数和符号的, symbol_table类的关键元素记录如下:

```
class symbol_table
{
public:
  friend class symtab_node;
  friend class cgraph_node;
  friend class cgraph_edge;

  int cgraph_count;    /* 记录当前全局符号表中的函数节点个数(不算变量节点) */
  int cgraph_max_uid;
  int cgraph_max_summary_id;
  int edges_count;
  int edges_max_uid;
  int edges_max_summary_id;

  /*
     这里是一个链表,其中链接当前编译单元的所有符号节点,对于函数节点实际上是一个 cgraph_node结构体, 对于变量节点实际上是一个 varpool_node结构体.
     nodes指向的是最后一个插入的节点, 见symbol_table::register_symbol (symtab_node *node)
  */
  symtab_node*   nodes;

  asm_node*   asmnodes;
  asm_node*   asm_last_node;
  cgraph_node*   free_nodes;
  cgraph_edge *   free_edges;
  int order;

  bool global_info_ready;
  /*
    * 在 analyze_functions中(gimplify之前)状态为 CONSTRUCTION 此状态代表当前正在处于 cgraph构建中,此时新增函数是安全的,
    * 在 execute_build_ssa_passes 中状态变为 IPA_SSA
  */
  enum symtab_state state;
};
```

这里主要需要关注的就是 symtab->nodes成员,此成员中记录了当前编译单元所有的函数和变量节点信息:

- 对于函数来说,其节点信息是通过一个 cgraph_node结构体来表示的
- 对于变量来说,其节点信息是通过一个 varpool_node结构体来表示的

因为二者的基类都是symtab_node,故两种节点可以同时链接到nodes上.

在gcc源码解析的过程中，当解析一个函数时(c_parser_declaration_or_fndef)最后会调用**cgraph_node::get_create**为当前函数创建一个函数节点(cgraph_node),当解析到一个变量时(同样c_parser_declaration_or_fndef)会调用**varpool_node::get_create**为变量创建一个变量节点(varpool_node)，二者创建的新节点都会被链接到全局符号表symtab->nodes中,流程如下:

```
toplev::main => do_compile => compile_file => c_common_parse_file => c_parse_file
 => c_parser_translation_unit
   => 循环直到文件结束
     => c_parser_external_declaration
       => c_parser_declaration_or_fndef     //解析一个声明或函数定义
         => 若当前解析到一个声明则调用finish_decl
           => 若当前声明不是函数声明,则调用 varpool_node::finalize_decl 为变量构建变量节点
             => varpool_node::get_create (decl);
         => 若当前解析到一个函数定义,则调用finish_function
           => cgraph_node::finalize_function (fndecl, false);    //为函数定义构建函数节点
             => cgraph_node::get_create (decl);
```

也就是说**在源码解析的AST语法树生成的过程中gcc已经为所有的函数定义和变量声明构建了对应的函数节点和变量节点并连接到了全局符号表symtab->nodes中，这个构建过程也同时将函数/变量的树节点和其node节点绑定到了一起**，最终:

- 函数定义和变量声明树节点均可以通过 decl->decl_with_vis.symtab_node 找到其对应的符号节点
- 一个符号节点也可以通过node->decl找到其对应的声明节点

二、全局符号表中结点的分析

全局符号表中的主要符号都是伴随源码解析过程中AST树的生成而生成的,在整个编译单元中所有外部声明都被解析为AST树节点后,所有的外部声明都总能对应到全局符号表中的一个函数节点或一个变量节点. 而此后编译的下一步操作就是解析全局符号表中的所有符号,此步的指令流程为:

```
toplev::main
  => do_compile
    => compile_file
      => c_common_parse_file => c_parse_file  => c_parser_translation_unit            //解析整个编译单元,并为每个外部声明在全局符号表中为其生成对应的函
    => if (!in_lto_p) symbol_table::finalize_compilation_unit
      => analyze_functions (/*first_time=*/true);
        => 遍历所有符号表节点
          => if (!cnode->analyzed) cnode->analyze ();                                 //对于函数节点,调用 cgraph_node::analyze
          => if (vnode && vnode->definition && !vnode->analyzed) vnode->analyze ();   //对于变量节点,调用 varpool_node::analyze
```

其中:

- 函数节点的分析,主要是对其进行gimplify(包括gimple高端化和gimple低端化)
- 变量节点的分析,主要是对其做对齐操作

而一个函数/变量节点分析完毕后,会设置其 symtab_node.analzed = true; 代表此节点已经分析完毕了.

三、函数节点的gimplify

对于变量节点的分析主要就是一个对齐操作(varpool_node::analyze),这里暂不分析,而对于函数节点的分析(cgraph_node::analyze)则比较麻烦, 其整个流程被称为函数节点的gimplify,按照步骤的不同，通常又被细分为gimple高端化和gimple低端化, cgraph_node::analyze函数大体如下:

```
void cgraph_node::analyze (void)
{
  if (native_rtl_p ())        /* __rtl节点不需要gimplify这一步,天然pass */
    {
      analyzed = true; return;
    }
  tree decl = this->decl;                           /* 从函数节点中找到此函数的声明树节点(FUNCTION_DECL) */
  ......
  if (alias)
    ......
  else
    {
      push_cfun (DECL_STRUCT_FUNCTION (decl));      /* 重置全局的cfun/current_function_decl到当前待分析函数 */
      assign_assembler_name_if_needed (decl);       /* 如果需要为此函数设置汇编名 */

      if (!gimple_has_body_p (decl))
        gimplify_function_tree (decl);              /* 若当前函数节点尚未做过gimple 高端化,则对其进行gimple 高端化 */

      if (!lowered)                                 /* 若当前函数节点尚未做过giomple低端化，则对其进行gimple 低端化 */
    {
      ......
      execute_pass_list (cfun, g->get_passes ()->all_lowering_passes);    /* gimple低端化并不是执行某个具体的函数,而是执行 all_lowering_passes链表中的
          ......
      lowered = true;                               /* 整个all_lowering_passes中所有pass都执行完毕，则在函数节点中标记gimple低端化结束 */
    }
      pop_cfun ();                                  /* 恢复之前的cfunc / current_function_decl */
    }
  analyzed = true;    /* 标记当前函数节点分析完毕 */
}
```

可见 函数节点的分析基本上就是gimple高端化+gimple低端化,其中:

- gimple 高端化通过函数gimplify_function_tree完成
- gimple低端化通过all_lowering_passes中的一系列pass完成,这些pass包括(是否使能取决于具体配置):

```
## ./gcc/passes.def
## 这里的pass是all_lowering_passes中所有可能被执行的pass,具体其是否被执行,则取决于gcc自身默认的编译选项和被编译代码的编译选项
  INSERT_PASSES_AFTER (all_lowering_passes)
  NEXT_PASS (pass_warn_unused_result);
  NEXT_PASS (pass_diagnose_omp_blocks);
  NEXT_PASS (pass_diagnose_tm_blocks);
  NEXT_PASS (pass_lower_omp);
  NEXT_PASS (pass_lower_cf);
  NEXT_PASS (pass_lower_tm);
  NEXT_PASS (pass_refactor_eh);
```

```
11.   NEXT_PASS (pass_lower_eh);
12.   NEXT_PASS (pass_build_cfg);
13.   NEXT_PASS (pass_warn_function_return);
14.   NEXT_PASS (pass_expand_omp);
15.   NEXT_PASS (pass_sprintf_length, false);
16.   NEXT_PASS (pass_walloca, /*strict_mode_p=*/true);
17.   NEXT_PASS (pass_build_cgraph_edges);
18.   TERMINATE_PASS_LIST (all_lowering_passes)
```