基于G++前端的C++源代码分析系统的初步研究①

彭四伟 辛丽娟 (北京化工大学 信息科学与技术学院 北京 100029)

摘 要: 利用 G++(GCC)的结构特点，将其前端的语言分析部分提取出来，对其前端部分进行初步研究，重构成独立的应用程序，用来对输入的源代码进行分析，并生成分析报告。代码分析的目的是为程序员提供代码的统计信息、结构信息，甚至更深层次的模式信息，帮助程序员更好地了解和把握程序的结构、框架和模式，提供改进、重构代码的参考依据。

关键词: G++; 前端语言; 重构代码; 代码分析; 参考依据

Preliminary Research of C++ Source Code Analysis System Based on G++ Front End

PENG Si-Wei, XIN Li-Juan

(College of Information Science and Technology, Beijing University of Chemical Technology, Beijing 100029, China)

Abstract: This article utilizes G++ (GCC) feature, extracts its front–end language analysis part, researches its front-end preliminarily, and restructs the independent application programme to analyze the input source code and produce the report. The purpose of code analysis is to supply the statistical information, the structure information, even the deeper level's pattern information of code for programmers, to help them understand and master the structure, the frame and the pattern of programme well, so as to provide reference of improving and restructuring code.

Keywords: G++; front–end language; restructuring code; code analysis; reference

GCC(GNU Compiler Collection) 是在 UNIX 以及类 UNIX 平台上广泛使用的编译器集合，它能够支持多种语言前端，包括 C, C++, Objective-C, Ada, Fortran, Java 和 treelang 等[1]。目前，关于 GCC(G++)的研究和开发工作侧重于 GCC(G++)后端代码优化方面，而本文所关注的目标是 GCC(G++)的编译过程中前端是如何工作的。

由于 G++是开源产品，可以直接使用其源代码，这就可以省去自行开发源代码分析程序的工作，而可以通过剖析和提取 G++前端部分(由于源代码分析与物理平台无关，因此不需要使用 G++的后端部分)，获得源代码的语法级分析功能[2]，设计出一个基于 G++前端的 C++源代码分析系统。所谓源代码分析系统就是对源代码进行自动分析，并自动生成一系列分析、评估报告或文档，例如，函数关系图、函数流程图、类系结构图等等。

1 G++编译器初步研究

本文采用一种新的方法对新安装的 G++编译器验证其是否可用。INSTALL 文件建议用 make check 运行测试套件，但是本文中采用的是将编译成功后的将 g++-4.3.3 中的 g++命令与系统 g++命令做符号连接，经过验证之后，说明此办法是可行的。下面就进行 G++编译器的内部结构的分析，这是对 G++编译器前端进行一步分析的基础。

从编译器发展的历史上看，编译器前端的技术现在已经相当成熟，不论从理论基础，还是应用的方面上来讲，都是计算技术中的典范。GCC/G++为语言前端提供了四层意义上的接口[3]，首先是在语言层面上，为不同语言的基本构造提供统一形式

① 收稿时间:2009-08-18;收到修改稿时间:2009-09-15

的接口(树)，其次语言前端接入 GCC/G++编译主体的接口，三是 GCC/G++给语言前端提供访问编译器主体，以及后端的函数调用接口，四是配置层面上，编译器的构造过程中如何整合所有的程序得到一个针对特定语言的编译器。下面图 1 就是 GCC 编译器集合一个整体的结构[4,5]，其中 G++编译器的整体结构包含其中。

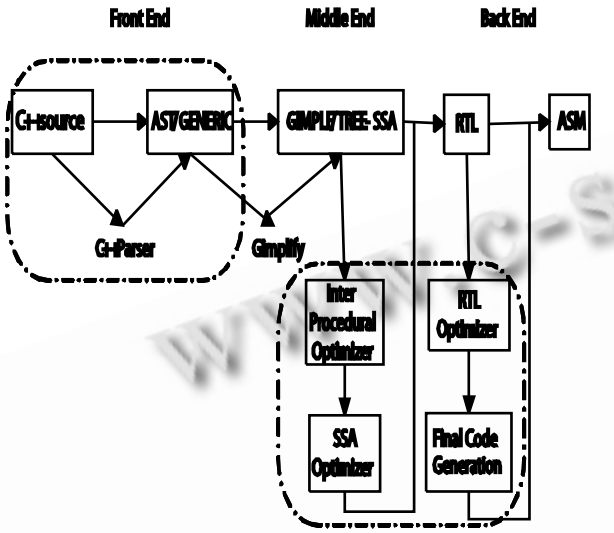

图 1　编译器整体结构

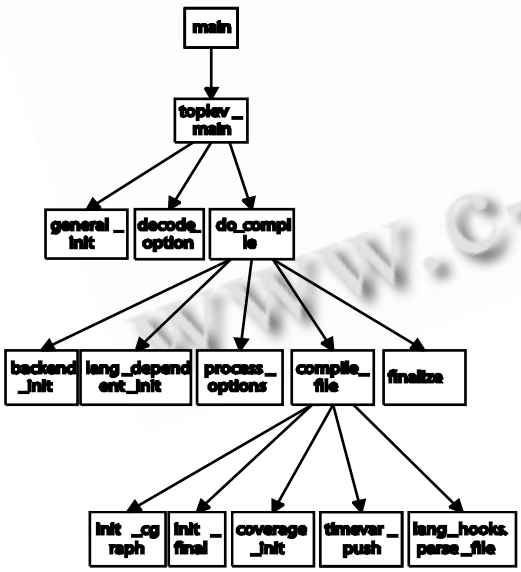

图 2　GCC 内部调用流程驱动部分，初始化部分

虽然 G++的源代码要比 GCC 的源代码小很多，因为 G++编译器在编译 C++的程序时还会用到 C 编译器的一些东西，所以 G++前端的源代码不可避免的会调用 C 语言的一些东西。在下面的论述中，将会就 G++前端是如何被主程序调用的进行研究。

gcc 是一个驱动程序,除了 main 函数,所提到的函数都 toplev.c 中被调用。toplev.c 文件是 cc1 和 c++的最高层控制流。它所做的工作大致是分析命令行参数，打开文件，调用编译器的各种遍，并且可以记录每一遍使用的时间。错误消息和 malloc 的低级接口也是在这里处理的[6]。

2　回调机制

本节将介绍 GCC 主体如何调用前端，进而得知 G++编译器的前端的工作机制。

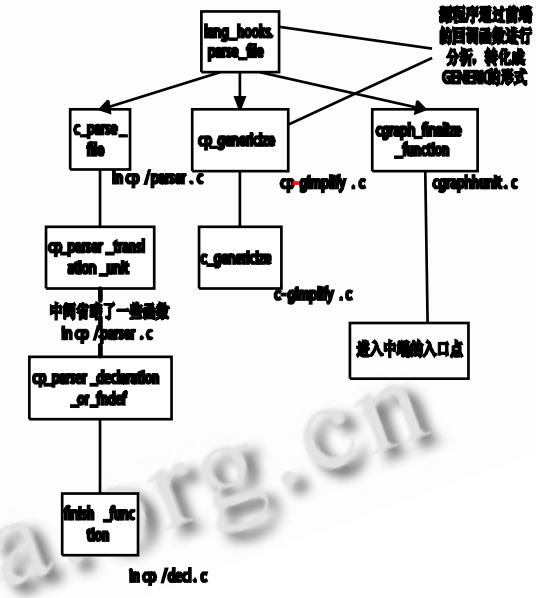

图 3　GCC 内部调用 G++前端部分

通过对源程序的理解以及流程图的分析，可以看出来 lang_hooks 中的 parse_file 函数完成了大部分的编译工作,这个 lang_hooks 结构体就是 GCC 主体和编译器前端的接口，而编译器的前端又通过定义这个结构体中的函数来向 GCC 注册自己，GCC 主体代码部分通过这个接口中的函数来处理和语言相关的东西。在 langhooks-def.h 和 langhooks.c, langhooks.h 中这个钩子结构的具体说明。在 langhooks.h 中，下面我们来分析下这个 lang_hook 结构。在这个结构中列出几个比较关键的函数。而和语言相关的所有的回调函数“钩子函数”都是在这个结

构中定义的。

```
struct lang_hooks
{
const char *name; //不同语言前端的名称，例如“GNU C++”
size_t identifier_size; //lang_identifier 结构的大小
.....
unsigned int (*init_options) (unsigned int argc, const char **argv); //主体调用的第一个前端函数
.....
void (*parse_file) (int); //主体来调用前端分析被编译文件的入口函数
void (*clear_binding_stack) (void); //分析完成之后，马上调用，清理堆栈绑定
```

下面是一些和语言相关的回调函数：

```
struct lang_hooks_for_functions function; //进入或离开函数时，管理具体的与语言相关的数据或状态
struct lang_hooks_for_tree_inlining tree_inlining; //树的内联函数
struct lang_hooks_for_callgraph callgraph ; //callgraph 的函数，一定不能为空
struct lang_hooks_for_tree_dump tree_ dump; //树的 dump 输出函数
struct lang_hooks_for_decls decls; //树结点 decls 函数和符号表都定义在此结构里
struct lang_hooks_for_types types; //相关的钩子类型函数
};
```

在 langhooks-def.h 文件中，对于每一个 lang_hooks 的每一个最底层的成员，都有一个宏与之对应，这些宏目前初始化定义为一个默认的函数。其中 name 成员的在 langhooks-def.h 的默认定义为：

```
#define LANG_HOOKS_NAME “GNU unknown” //宏初始化的默认定义
```

最后，该文件利用这些宏定义了前面所提到的 LANG_HOOKS_INITIALIZER 宏，它是一个 struct lang_hooks 的初始化器，初始化在 langhooks-def.h 中的一些所需要的宏。在 G++ 前端中 cp-lang.c 是这样定义的：

```
#undef LANG_HOOKS_NAME //撤消以前的宏定义
#define LANG_HOOKS_NAME “GNU C++” //成员 name 的宏定义为“GNU C++”
#undef LANG_HOOKS_INIT //结束初始化宏
```

然后，就可以使用 LANG_HOOKS_INITIALIZER 来初始化自己的 lang_hooks 了，当然，没有被重新定义的，仍然使用默认的函数。语言前端模块中，以宏定义的方法将回调函数注册到 GCC 主体中，G++ 前端注册在 cp-lang.c 中：

```
const struct lang_hooks lang_hooks= LANG_HOOKS_INITIALIZER; //每个语言前端都提供自己的 lang_hooks 的初始化器
```

LANG_HOOKS_INITIALIZER 初始化定义在 langhooks-def.h 中，下面是一个 LANG_HOOKS_INITIALIZER 的形式：

```
#define LANG_HOOKS_INITIALIZER
{
LANG_HOOKS_NAME, //语言前端的宏名
LANG_HOOKS_IDENTIFIER_SIZE, //结构大小
.....
} // LANG_HOOKS_INITIALIZER 的初始化定义
```

在 langhooks-def.h 中定义了默认的语言回调函数，比如：

```
#define LANG_HOOKS_PARSE_FILE lhd_do_nothing_ //语言的语法分析的默认函数
```

这些默认的函数的实现在 langhooks.c 中。比如：

```
void lhd_do_nothing_i (int ARG_UNUSED (i))
{} //不做任何事情
```

在 cp-objcp-common.h 中，G++的语言前端通过如下形式，覆盖默认的回调函数定义。

```
#undef LANG_HOOKS_PARSE_FILE //取消以前的宏定义
#define LANG_HOOKS_PARSE_FILE c_common_parse_file //将此宏定义为此变量
```

在 cp-objcp-common.h 中，重新定义了大量的宏，这些函数是编写 G++ 前端的 C++语言和 OBJC++语言所共享的。而在 cp-lang.c 文件中定义的那两个宏是 C++语言自己的。c_common_parse_file 这个函数的定义在 c-opts.c 中。

```
/*初始化整个预处理器，循环每个输入文件*/
void c_common_parse_file (int set_ yydebug)
{
```

```
unsigned int i; //无符号整数 i
if (set_yydebug) //条件分支，如果全局变量 set_yydebug 非 0，表示在分析的过程中输出调试信息
switch (c_language) //枚举语言类型的分支
.....
{
 /*开始分析输入文件，如果调试器需要它*/
 If (debug_hooks->start_end_main_ source_file)
 (*debug_hooks->start_source_file) (0, this_ input_filename); //指向 gcc_debug_hooks 结构的指针 debug_hooks，如果调试器需要主源文件的开始和结束命令，则这个结构中的 start_end_ main_source_file 为 1；如果这个条件为真，则这个 debug_hooks 中的函数的参数分别为 0 和当前的输入文件
 finish_options (); //处理 -D, -U, -A, -imacros, 以及第一个 -include
 pch_init (); // 准备写一个 PCH 文件，在编译的开始阶段被调用
 push_file_scope (); //存根函数
 c_parse_file (); //分析一个源文件
 finish_file (); //最后的处理 file_scope 的中数据
 pop_file_scope (); //存根函数
 /* 结束分析输入文件，如果调试器需要它*/
 If (debug_hooks->start_end_main_ source_ file)
 (*debug_hooks->end_source_file) (0); //指向 gcc_debug_hooks 结构的指针 debug_hooks,如果调试器需要主源文件的开始和结束命令，则这个结构中的 start_end_main_source_file 为 1；如果这个条件为真，则这个结构中的 end_source_file 函数参数为 0
 if (++i >= num_in_fnames) break; //测试 i 是否大于等于输入的文件的数量，如果条件为真，则跳出循环
 cpp_undef_all (parse_in); //处理所有的未被定义的宏和断言，parse_in 为指向 cpp_reader 结构的指针
 cpp_clear_file_cache (parse_in); //确保分析器忘记以前的关于文件的操作，有利于对另一次的程序运行分析
 this_input_filename=cpp_read_main_file (parse_in, in_fnames[i]); // this_input_filename 为指向全局静态变量的指针，它存储的是输入文件的文件名，此文件名为 cpp_read_main_file 函数的返回值，此函数为 CPP 库中的处理文件的函数
 /* 如果输入文件丢失，放弃进一步的编译 */
 if (!this_input_filename) break;
 }
}
```

在这个 c_common_parse_file 中,有一个 set_yydebug 变量，这个变量就是回调机制中前端和主体一个桥梁。在 toplev.c 中函数以下面这种形式来调用回调函数：

```
/*调用分析器来分析整个输入文件(对每一个函数调用 rest_of_compilation 函数) */
lang_hooks.parse_file (set_yydebug);
```

通过这样一种机制，结合由 configure 过程根据 C++语言选择源文件，从而实现正确的回调函数调用。

3 前端和后端的分离的理论研究

从严格意义上来讲，GCC(G++)编译器是分为前、中、后端的，如图 1 所示。所以如何分离 G++编译器的前后端就需要清楚中端在后端中所起的衔接作用。通过研究得知：

(1) 重构的程序，不仅是 G++源代码前端的，同时包括前中端的，中端主要是 GIMPLE-TREE/SSA 优化框架。因为程序不包括后端，所以可以把原来生成 RTL 以及后续的部分从源代码中去掉。前后端的分离并不需要设计一个新的中间表示，可以使用 GIMPLE-SSA 中间表示来做，GCC(G++)的 GIMPLE-SSA 优化框架就可以直接用作“分析框架”,而且 G++已有的那些分析都可以直接用。

(2) G++源代码中，pass 是一个经常被用到的的术语，源代码中的 passes.c 中的 init_optimization_ passes 函数负责调度所有需要执行的 pass[7]，它在 toplev_main 中的 general_init 函数的最后被调用。所有添加的 pass 都要在这个函数中注册。tree-optimize.c 是所有树优化 pass 的主要驱动。它包含函数 tree_rest_of_compilation,这个函数完成对于一个 FNDECL 树的所有的优化和编译工作，FNDECL 是前端生成的。在这个 init_optimization_ passes 函数中，将所有的中端分析、变换的 pass 都组织在一起了。其中有一个 pass 叫做 pass_expand，这个 pass 的作用就是将 GIMPLE 中间表示转化成后端的 RTL 表示。这个 pass 之后的 pass 就是在 RTL 上操作的了。 所以，分离后端就是首先将这个 pass 后面的所有 pass 去掉，

然后加上结束程序的 pass。经过理论分析与研究，下面是加入一个结束的 pass 的方法：①每一个 SSA 的 pass 都必须注册并被 init_optimization_pass 调度；②每一个 pass 都是由结构 tree_opt_pass 描述的；③增加一个新 pass。

增加一个新的 pass 的方法经过研究得知：创建一个全局的 struct tree_opt_pass 类型的变量，并在 tree_passes.h 中增加它的一个外部声明，在 passes.c 的 init_optimization_passes 中用 NEXT_PASS 注册的 pass。这个 pass 的具体实现代码会在以后的研究中逐步写出来。通过增加这个 pass 以及对源代码中相关的程序逐步做出相应的改变，便能成功分离出前后端。

4 结语

本文是对于基于 G++前端的 C++源代码系统的初步的研究，并且对其前后端分离做了一些理论研究，但 GCC/G++本身的源代码程序就很强大，很复杂，二者又有很多共用的部分。并且各个源程序文件的功能，以及之间的相互关系也很复杂。所以具体的需要代码的实现部分还需要进一步的研究。

参考文献

1 Josee lajoie.Gcc Home Page.http://gcc.gnu.org.2007-12

2 冯博琴,冯岚,等译.编译原理及实践.北京:机械工业出版社, 2000.

3 蔡杰. GCC 编译系统结构分析与后端移植实践[硕士学位论文].杭州:浙江大学, 2007.

4 丁松阳,张墨华. GCC 优化框架研究.光盘技术, 2006,4:28－29.

5 贺康,陈超,刘坚. C++编译器前端对函数重载的设计研究.电子科技,2009,22(2):28－32.

6 任珊红. GCC 的中间语言以及后端转换.计算机工程与科学, 1995,17:74－82.

7 朱少波.基于 GCC 开发 C 编译器的研究与实践[硕士学位论文].杭州:浙江大学, 2003. 9－11.