HZ BOOKS
华章科技

国内不可多得的全面系统介绍GCC设计与实现的书籍，对GCC的总体设计、主要代码架构及实现细节进行了深入的分析和总结

本书结合GCC4.4.0源代码，围绕GCC编译过程，以GCC中的中间表示AST、GIMPLE及RTL为主线，为读者描述了一条从源代码到目标机器汇编代码的清晰路线图

源码分析系列

深入分析GCC

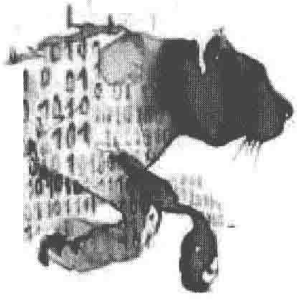
王亚刚◎编著

机械工业出版社
China Machine Press

源码分析系列

深入分析 GCC

王亚刚◎编著



机械工业出版社
China Machine Press

图书在版编目（CIP）数据

深入分析 GCC / 王亚刚编著 . —北京：机械工业出版社，2017.1
（源码分析系列）

ISBN 978-7-111-55632-9

I. 深… II. 王… III. 应用软件 IV. TP317

中国版本图书馆 CIP 数据核字（2016）第 317737 号

深入分析 GCC

出版发行：机械工业出版社（北京市西城区百万庄大街 22 号 邮政编码：100037）
责任编辑：张梦玲 责任校对：董纪丽
印 刷：北京诚信伟业印刷有限公司 版 次：2017 年 2 月第 1 版第 1 次印刷
开 本：186mm × 240mm 1/16 印 张：34.25
书 号：ISBN 978-7-111-55632-9 定 价：99.00 元

凡购本书，如有缺页、倒页、脱页，由本社发行部调换
客服热线：（010）88379426 88361066 投稿热线：（010）88379604
购书热线：（010）68326294 88379649 68995259 读者信箱：hzit@hzbook.com

前　言

GCC（GNU Compiler Collection，GNU 编译器套件）是一套由 GNU 开发的程序设计语言编译工具，是 GNU 工程中最重要的组成部分。经过近 30 年的发展，GCC 不仅支持众多的前端编程语言，还支持各种主流的处理器平台和操作系统平台，成为公认的跨平台编译器的事实标准，也成为编译器设计的成功典范。

作为一名 GCC 编译器的使用者和源码阅读的爱好者，我一直想写一本关于 GCC 的技术书。

2002 年，我开始在 Linux 系统上进行一些软件开发，GCC 就是我使用的编译工具。我一直对从源代码到目标代码的转换过程充满好奇，想知道在这个过程中 GCC 到底都做了些什么？GCC 是如何设计的，那些成千上万个 GCC 的源代码文件都表示了什么意义？那时我常常幻想，要是能透彻地分析和理解 GCC 源代码，多好！从那时起，在教学科研之余，我偶尔会翻阅一下 GCC 的相关源代码，可是看着繁多的 GCC 源代码，也常常感觉手足无措，真有一种“老虎吃天，无法下爪”的尴尬。于是分析 GCC 源代码的事情被搁置了，然而那种一探究竟的心情总是挥之不去。

2012 年开始，我有了较多的闲暇时间，在经过一段彷徨之后，分析 GCC 源代码的冲动又一次浮现在脑海。我知道，这次是要来真的了，我要做点自己喜欢的事。

Why?

我有空余时间了，我要干些自己感兴趣的事情。在我创建的 GCC 爱好者交流群中经常有朋友问，有没有介绍 GCC 的资料呀？大多人都会说，有——请看官方文档！我也去看了看，没错，GCC 有比较详细的官方文档，包括 gccinternal 及用户手册等。然而，这些文档的内容庞杂，缺乏系统分析 GCC 设计框架和工作流程的内容，并且大多的内容对读者来讲都是零散的，让初学者无所适从。于是我想，为什么不分析一下 GCC 系统，把 GCC 的设计实现用一种更清晰明了、更系统的方法介绍给 GCC 的爱好者呢？

What?

本书将围绕 GCC 编译过程，详细介绍从源代码到 AST、从 AST 到 GIMPLE、从 GIMPLE

到RTL，以及从RTL到最终的目标机器汇编代码的详细过程，涉及各个阶段中间表示的详细分析、生成过程。本书提供了大量的图表和实例，展示了GCC编译系统的总体工作流程和工作细节。本书的另外一个特点是结合GCC 4.4.0的源代码进行分析，使读者在了解编译原理的基础上进一步掌握其实现的总体流程和细节，让更多读者对编译技术的认识不再只停留在理论层面，而是向其展示一个编译系统实例的实现过程。

How?

GCC源代码涉及的内容非常庞杂，很难在一本书中全面描述，因此本书以GCC中间表示为主线，详细分析GCC从源代码开始，直到生成目标机器汇编代码的整个过程中所使用的三种中间表示（AST、GIMPLE及RTL），并对这三种中间表示的基本概念、生成过程进行详细的描述，对基于GIMPLE和RTL的优化处理进行介绍，从而描述一条从源代码到目标机器汇编代码的清晰路线图。

Who?

本书以热爱编译系统理论及其实现的在校大学生、研究生为主要读者对象，也可以作为企业中研发编译系统以及进行编译系统移植的研发工程师的有益参考。

在编写这本书的时候，有一种精神支撑着我，我相信“兴趣”加上“坚持”就是胜利！分析GCC不是一年半载的事情，需要3年、5年，甚至更长时间，不过我可以坚持，我要用我的坚持换来对GCC的深入分析，让更多的GCC爱好者熟悉它、接触它、了解它，更多地参与GCC的开发与维护。

感谢我的爱人和孩子，给了我家的温暖和亲情。感谢病榻上的父亲，虽然他不能和我说话，但他那一双大手，依然经常抚摸在我的头上。感谢年老体弱的母亲，感谢她一直照顾我的父亲，让我知道什么是坚持，什么是不离不弃！感谢西安邮电大学GPU项目组的各位同事在本书的写作中提出的宝贵建议。

本书的写作得到国家自然科学基金重点项目（项目编号：61136002）以及陕西省教育厅科研计划项目（项目编号：14JK1674）资助。

鉴于作者水平有限，在分析和写作本书的过程中也引入了一些个人观点，因此难免有一些理解的偏差和错误，敬请读者批评指正并不吝赐教，如有意见和建议，请联系作者lazy_linux@126.com，在此一并感谢！

王亚刚

2016年10月于西安邮电大学

目　录

前言

第 1 章　GCC 概述 …… 1

1.1　GCC 的产生与发展 …… 1

1.2　GCC 的特点 …… 2

1.3　GCC 代码分析 …… 3

第 2 章　GCC 源代码分析工具 …… 4

2.1　vim+ctags 代码阅读工具 …… 4

2.2　GNU gdb 调试工具 …… 6

2.3　GNU binutils 工具 …… 8

2.4　shell 工具及 graphviz 绘图工具 …… 11

2.5　GCC 调试选项 …… 13

第 3 章　GCC 总体结构 …… 16

3.1　GCC 的目录结构 …… 16

3.2　GCC 的逻辑结构 …… 18

3.3　GCC 源代码编译 …… 20

3.3.1　配置 …… 21

3.3.2　编译 …… 23

3.3.3　安装 …… 25

第 4 章　从源代码到 AST/GENERIC …… 26

4.1　抽象语法树 …… 26

4.2　树节点的声明 …… 28

4.3　树节点结构 …… 33

4.3.1　struct tree_base …… 35

4.3.2　struct tree_common …… 36

4.3.3　常量节点 …… 38

4.3.4　标识符节点 …… 42

4.3.5　声明节点 …… 44

4.3.6　struct tree_decl_minimal …… 46

4.3.7　struct tree_decl_common …… 46

4.3.8　struct tree_field_decl …… 49

4.3.9　struct tree_decl_with_rtl …… 55

4.3.10　struct tree_label_decl …… 55

4.3.11　struct tree_result_decl …… 56

4.3.12　struct tree_const_decl …… 57

4.3.13　struct tree_parm_decl …… 57

4.3.14　struct tree_decl_with_vis …… 59

4.3.15　struct tree_var_decl …… 59

4.3.16　struct tree_decl_non_common …… 62

4.3.17　struct tree_function_decl …… 62

4.3.18　struct tree_type_decl …… 64

4.3.19　类型节点 …… 67

4.3.20　tree_list 节点 …… 68

4.3.21　表达式节点 …… 71

4.3.22　语句节点 …… 73

4.3.23 其他树节点 …… 75
4.4 AST 输出及图示 …… 76
4.5 AST 的生成 …… 83
4.5.1 词法分析 …… 84
4.5.2 词法分析过程 …… 90
4.5.3 语法分析 …… 98
4.5.4 语法分析过程 …… 99
4.5.5 c_parse_file …… 103
4.5.6 c_parser_translation_unit …… 105
4.5.7 c_parser_external_declaration …… 105
4.5.8 c_parser_declaration_or_fndef …… 107
4.5.9 c_parser_declspecs …… 112
4.6 小结 …… 114

第 5 章 从 AST/GENERIC 到 GIMPLE …… 115

5.1 GIMPLE …… 115
5.2 GIMPLE 语句 …… 119
5.3 GIMPLE 的表示与存储 …… 122
5.4 GIMPLE 语句的操作数 …… 128
5.5 GIMPLE 语句序列的基本操作 …… 132
5.6 GIMPLE 的生成 …… 135
5.6.1 gimplify_function_tree …… 136
5.6.2 gimplify_body …… 138
5.6.3 gimlify_parameters …… 139
5.6.4 gimplify_stmt …… 144
5.6.5 gimplify_expr …… 144
5.7 GIMPLE 转换实例 …… 157
5.7.1 BIND_EXPR 节点的 GIMPLE 生成 …… 158
5.7.2 STATEMENT_LIST_EXPR 节点的 GIMPLE 生成 …… 159
5.7.3 MODIFY_EXPR 节点的 GIMPLE 生成 …… 160
5.7.4 POSTINCREMENT_EXPR 节点的 GIMPLE 生成 …… 162
5.8 实例分析 …… 172
5.9 小结 …… 176

第 6 章 GIMPLE 处理及其优化 … 177

6.1 GCC Pass …… 177
6.1.1 核心数据结构 …… 177
6.1.2 Pass 的类型 …… 179
6.1.3 Pass 链的初始化 …… 182
6.1.4 Pass 的执行 …… 184
6.2 Pass 列表 …… 187
6.3 GIMPLE Pass 实例 …… 193
6.3.1 pass_remove_useless_stmts … 193
6.3.2 pass_lower_cf …… 195
6.3.3 pass_build_cfg …… 197
6.3.4 pass_build_cgraph_edges …… 203
6.3.5 pass_build_ssa …… 205
6.3.6 pass_all_optimizations …… 206
6.3.7 pass_expand …… 207
6.4 小结 …… 207

第 7 章 RTL …… 208

7.1 RTL 中的对象类型 …… 209
7.2 RTX_CODE …… 210
7.3 RTX 类型 …… 210
7.4 RTX 输出格式 …… 212
7.5 RTX 操作数 …… 213
7.6 RTX 的机器模式 …… 216

7.7 RTX 的存储 …… 219
7.8 RTX 表达式 …… 222
7.8.1 常量 …… 225
7.8.2 寄存器和内存 …… 227
7.8.3 算术运算 …… 228
7.8.4 比较运算 …… 230
7.8.5 副作用 …… 230
7.9 IR-RTL …… 232
7.9.1 INSN …… 233
7.9.2 JUMP_INSN …… 234
7.9.3 CALL_INSN …… 235
7.9.4 BARRIER …… 235
7.9.5 CODE_LABEL …… 236
7.9.6 NOTE …… 237
7.10 小结 …… 238

第 8 章 机器描述文件 ${target}.md …… 239

8.1 机器描述文件 …… 240
8.2 指令模板 …… 241
8.2.1 模板名称 …… 242
8.2.2 RTL 模板 …… 246
8.2.3 条件 …… 256
8.2.4 输出模板 …… 256
8.2.5 属性 …… 256
8.3 定义 RTL 序列 …… 257
8.4 指令拆分 …… 263
8.5 枚举器 …… 266
8.5.1 mode 枚举器 …… 266
8.5.2 code 枚举器 …… 268
8.6 窥孔优化 …… 269
8.6.1 define_peephole …… 269
8.6.2 define_peephole2 …… 270
8.7 小结 …… 271

第 9 章 机器描述文件 ${target}.[ch] …… 272

9.1 targetm …… 272
9.1.1 struct gcc_target 的定义 …… 273
9.1.2 targetm 的初始化 …… 277
9.2 编译驱动及选项 …… 279
9.2.1 编译选项 …… 280
9.2.2 SPEC 语言及 SPEC 文件 …… 281
9.2.3 机器相关的编译选项 …… 285
9.3 存储布局 …… 286
9.3.1 位顺序和字节顺序 …… 286
9.3.2 类型宽度 …… 287
9.3.3 机器模式提升 …… 287
9.3.4 存储对齐 …… 288
9.3.5 编程语言中数据类型的存储布局 …… 289
9.4 寄存器使用 …… 290
9.4.1 寄存器的基本描述 …… 290
9.4.2 寄存器分配顺序 …… 297
9.4.3 机器模式 …… 298
9.4.4 寄存器类型 …… 300
9.5 堆栈及函数调用规范描述 …… 307
9.5.1 堆栈的基本特性 …… 309
9.5.2 寄存器消除 …… 313
9.5.3 函数栈帧的管理 …… 315
9.5.4 参数传递 …… 316
9.5.5 函数返回值 …… 318
9.5.6 i386 机器栈帧 …… 318
9.6 寻址方式 …… 325
9.7 汇编代码分区 …… 326
9.8 定义输出的汇编语言 …… 333

9.8.1 汇编代码文件的框架 …… 333
9.8.2 数据输出 …… 336
9.8.3 未初始化数据输出 …… 336
9.8.4 标签输出 …… 338
9.8.5 指令输出 …… 342
9.9 机器描述信息的提取 …… 343
9.9.1 gencode.c …… 347
9.9.2 genattr.c …… 348
9.9.3 genattrtab.c …… 348
9.9.4 genrecog.c …… 349
9.9.5 genflag.c …… 352
9.9.6 genemit.c …… 353
9.9.7 genextract.c …… 354
9.9.8 genopinit.c …… 356
9.9.9 genoutput.c …… 360
9.9.10 genpreds.c …… 362
9.9.11 其他 …… 363
9.10 小结 …… 364

第 10 章 从 GIMPLE 到 RTL … 365

10.1 GIMPLE 序列 …… 365
10.2 典型数据结构 …… 366
10.3 RTL 生成的基本过程 …… 367
10.3.1 变量展开 …… 370
10.3.2 参数及返回值处理 …… 380
10.3.3 初始块的处理 …… 395
10.3.4 基本块的 RTL 生成 …… 398
10.3.5 退出块的处理 …… 410
10.3.6 其他处理 …… 411
10.4 GIMPLE 语句转换成 RTL …… 411
10.4.1 GIMPLE 语句转换的一般过程 …… 412
10.4.2 GIMPLE_GOTO 语句的 RTL 生成 …… 415
10.4.3 GIMPLE_ASSIGN 语句的 RTL 生成 …… 417
10.5 小结 …… 432

第 11 章 RTL 处理及优化 …… 433

11.1 RTL 处理过程 …… 433
11.2 特殊虚拟寄存器的实例化 …… 435
11.3 指令调度 …… 437
11.3.1 指令调度算法 …… 439
11.3.2 GCC 指令调度的实现 …… 440
11.3.3 指令调度实例 1 …… 442
11.3.4 指令调度实例 2 …… 459
11.4 统一寄存器分配 …… 460
11.4.1 基本术语 …… 461
11.4.2 寄存器分配的主要流程 … 463
11.4.3 代码分析 …… 466
11.4.4 寄存器分配实例 1 …… 468
11.4.5 寄存器分配实例 2 …… 483
11.5 汇编代码生成 …… 494
11.5.1 汇编代码文件的结构 …… 495
11.5.2 从 RTL 到汇编代码 …… 499
11.6 小结 …… 502

第 12 章 支持新的目标处理器 … 503

12.1 GCC 移植 …… 503
12.2 PAAG 处理器 …… 504
12.2.1 PAAG 处理器指令集 …… 505
12.2.2 应用二进制接口 …… 505
12.3 GCC 移植的基本步骤 …… 506
12.4 PAAG 机器描述文件 (paag.md) …… 507

12.5 paag.[ch] 文件 …… 512
12.5.1 存储布局 …… 512
12.5.2 寄存器使用规范 …… 513
12.5.3 堆栈布局及堆栈指针 …… 514
12.5.4 函数调用规范 …… 515
12.5.5 寻址方式 …… 519
12.5.6 汇编代码输出 …… 521
12.5.7 杂项 …… 523
12.6 PAAG 后端注册 …… 523
12.7 GCC 移植测试 …… 524
12.8 小结 …… 526

参考文献 …… 527

索引 …… 529

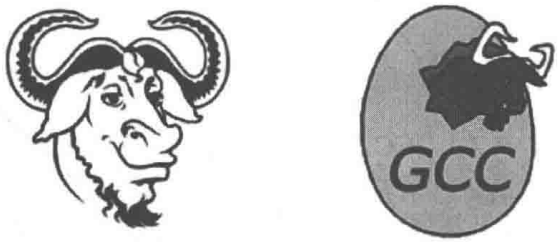
第 1 章
GCC 概述

本章主要对 GCC 的发展过程及 GCC 的特点进行简介，并给出了本书的主要内容简介。

1.1 GCC 的产生与发展

GCC（GNU Compiler Collection）是 GNU 工程（GNU Project）中的核心工具软件，其官方网址为 https://gcc.gnu.org/。GCC 支持多种前端的编程语言，包括 C、C++、Java、Ada 和 Fortran 等，其编译生成的目标代码可以在几乎所有的处理器平台上运行，是目前使用最广泛的编译系统之一。GCC 遵循 GNU GPL（GNU Public License）协议，由 FSF（Free Software Foundation）发布。GNU 和 GCC 的图标如图 1-1 所示。

初期的 GCC 仅仅作为 C 语言的编译器，即 GNU C Compiler。1987 年 GCC 1.0 发布，同年 12 月，GCC 开始支持 C++ 语言，随后，GCC 开始支持 Objective-C、Objective-C++、Fortran、Java 和 Ada 等语言。与此同时，GCC 也被逐渐移植到各种各样的主流处理器体系结构上，包括 i386、ix86_64、SPARCE、ARM 和 MIPS 等处理器平台。



a）GNU 图标　　b）GCC 图标

图 1-1　GNU 及 GCC 的图标

自从 1987 年 Richard Stallman 和 Len Tower 发布 GCC 的第一个版本 GCC 1.0 以来，目前 GCC 的最新版本已经更新到 GCC 6.0，https://gcc.gnu.org/releases.html 给出了 GCC 在各个时期推出的 GCC 版本，其中最重大的变化是在 1999 年 7 月，GCC 与 EGCS（Experimental/Enhanced GNU Compiler System）重新融合并发布了 GCC 2.95 版本。

相关的资料可以查阅以下官方网站信息：

```
GNU Compiler Collection: https://gcc.gnu.org/
Free Software Foundation: http://www.fsf.org/
GNU Project: https://gnu.org/
GNU Public License: https://www.gnu.org/licenses/licenses.en.html#GPL
```

1.2 GCC 的特点

GCC 作为目前较为成功的编译系统之一，具有非常突出的优点，主要包括：

（1）GCC 编译系统支持众多的前端编程语言，GCC 4.4.0 中 ${GCC_SOURCE}/gcc/ 目录下包含了前端编程语言处理的目录及其代码（其中，${GCC_SOURCE} 表示 GCC 源代码的主目录，下同），主要包括 C、C++、Ada、Fortran、Java、Objective-C、Objective-C++ 等语言的前端处理，可以使用如下命令查看这些目录：

```
[GCC@localhost gcc-4.4.0]$ ls -l gcc
drwxrwxr-x.  3 GCC GCC   69632 Apr 21  2009 ada
drwxrwxr-x.  2 GCC GCC    4096 Nov 27  2013 cp
drwxrwxr-x.  2 GCC GCC    4096 Nov  6 15:14 fortran
drwxrwxr-x.  2 GCC GCC    4096 Oct  9 17:34 java
drwxrwxr-x.  2 GCC GCC    4096 Apr 21  2009 objc
drwxrwxr-x.  2 GCC GCC    4096 Apr 21  2009 objcp
```

（2）GCC 支持众多的目标机器体系结构，具有良好的可移植性，GCC 4.4.0 的 ${GCC_SOURCE}/gcc/config/ 目录下包含了 GCC 对目标处理器的支持情况，其中包括了各种主流的处理器，例如，arm、i386、mips 以及 alpha 等，以下是 GCC 4.4.0 代码所支持的处理器列表：

```
alpha       arc         arm         avr         cris
crx         fr30        frv         h8300       i386
ia64        iq2000      m32c        m32r        m68hc11
m68k        mcore       mips        mmix        mn10300
pa          pdp11       picochip    rs6000      s390
score       sh          sparc       spu         stormy16
v850        vax         xtensa
```

（3）GCC 具有丰富的配套工具链支持。

GCC 不是一个孤立的编译工具，而是整个 GNU 工程中的一个组成部分。GNU 工程中的其他软件，包括 GNU C 库 glibc、GNU 的调试工具 gdb，以及 GNU 二进制工具链 binutils（GNU Binutils Toolchains，例如汇编工具 as，连接工具 ld，目标文件分析工具 objdump、objcopy 等）等都与 GCC 关系密切，互相依赖。

可以使用下述的 shell 命令查看 GNU 二进制工具链中主要包括的工具：

```
[GCC@localhost paag-gcc]$ rpm -ql binutils  | xargs ls -l | grep "/usr/bin"
-rwxr-xr-x. 1 root root  24352 Oct 15  2014 /usr/bin/addr2line
-rwxr-xr-x. 1 root root  54444 Oct 15  2014 /usr/bin/ar
-rwxr-xr-x. 1 root root 527220 Oct 15  2014 /usr/bin/as
-rwxr-xr-x. 1 root root  26356 Oct 15  2014 /usr/bin/c++filt
-rwxr-xr-x. 1 root root  99212 Oct 15  2014 /usr/bin/gprof
-rwxr-xr-x. 1 root root 588116 Oct 15  2014 /usr/bin/ld
-rwxr-xr-x. 1 root root  38800 Oct 15  2014 /usr/bin/nm
-rwxr-xr-x. 1 root root 212216 Oct 15  2014 /usr/bin/objcopy
-rwxr-xr-x. 1 root root 276528 Oct 15  2014 /usr/bin/objdump
-rwxr-xr-x. 1 root root  54448 Oct 15  2014 /usr/bin/ranlib
-rwxr-xr-x. 1 root root 288560 Oct 15  2014 /usr/bin/readelf
```

```
-rwxr-xr-x. 1 root root  27196 Oct 15  2014 /usr/bin/size
-rwxr-xr-x. 1 root root  25832 Oct 15  2014 /usr/bin/strings
-rwxr-xr-x. 1 root root 212244 Oct 15  2014 /usr/bin/strip
```

（4）GCC 提供可靠、高效、高质量的目标代码。

GCC 是目前使用的最为广泛的编译器系统之一，众多工业级应用的实践证明，GCC 编译系统生成的代码具有很高的可靠性和运行效率。

（5）GCC 对于并行编译的支持。

在 GCC 4.4.0 中，已经提供了对 OpenMP 的完整支持。

1.3 GCC 代码分析

GCC 作为目前 GNU 项目中应用最广泛的工具软件之一，是工程师设计编译系统最典型、最成功的范例，是高校学生学习编译系统最生动、最权威的设计实例，同时也是程序员进行高质量代码设计的有益参考。本书以 GCC 4.4.0 的源代码为例，对 GCC 的设计和实现进行分析和解读，主要涉及以下内容：

（1）GCC 的发展历史及特点；

（2）GCC 的总体结构；

（3）GCC 中各种中间表示（包括抽象语法树、GIMPLE、寄存器传输语言）的生成技术；

（4）GCC 中基于 GIMPLE 的优化处理，这一部分主要实现一些与目标机器无关的性能优化；

（5）GCC 中基于 RTL 的优化处理，这一部分主要实现一些与目标机器相关的性能优化；

（6）GCC 的移植技术，即如何让 GCC 支持新的目标机器。

本书将紧密围绕编译系统中的中间表示（IR，Intermediate Representation）这一核心概念，重点介绍 GCC 中的三种中间表示：抽象语法树（AST，Abstract Syntax Tree）、GIMPLE 和寄存器传输语言（RTL，Register Transfer Language），对其基本概念、存储结构及其生成过程等进行深入分析。由于 GCC 基于 GIMPLE 和 RTL 的优化处理数量非常多，每种优化处理都涉及一个比较独立的优化问题，很难在本书中一一详述，因此，本书只简单地介绍了 GCC 中基于 GIMPLE 及 RTL 的优化处理的基本组织方式，并对其中一些非常典型的优化处理进行了简介。最后，本书也给出了将 GCC 成功移植到西安邮电大学自主研发的阵列处理器上的一个实例。

限于篇幅，书中的大部分代码只给出了简化版本，读者在分析时需要结合源代码仔细研读。

第 2 章
GCC 源代码分析工具

代码分析是一件烦琐的事情。在分析 GCC 源代码时，几乎所有的人都会说："这么多的代码，怎么看？"是的，面对 GCC 4.4.0 如此庞大的代码量，原始的、徒手的做法显然是不足以应付的。在阅读 GCC 代码时，通常遇到的典型问题包括：

（1）如何跟踪函数调用；

（2）如何查看一个变量的定义；

（3）如何查看一个函数被哪些函数调用过；

（4）如何分析函数之间的调用关系；

（5）如何理解某个函数的工作过程。

当然，除了理解这些表面的问题，更深层的问题就是 GCC 到底是如何设计的？ GCC 这么庞大的代码是如何组织的？ GCC 在进行源代码编译的过程中都包括哪些主要的处理阶段，每个阶段完成了哪些工作，这些阶段之间又是如何相互联系起来的？

这些问题的回答，都需要对 GCC 的代码进行详细分析。笔者认为，没有好的工具作为辅助，分析 GCC 代码几乎是不可能的！本章主要介绍一些作者在分析 GCC 4.4.0 代码时所使用的一些常用工具，供大家参考。这部分内容仅仅是点到为止，详细内容请参阅其用户文档。

本书介绍的所有代码分析工具均基于 Centos Linux 系统。

2.1 vim+ctags 代码阅读工具

vim 是 Linux 中应用最广泛的编辑器，也是阅读 GCC 4.4.0 源代码的首选工具。ctags 是一种标签工具，可以配合 vim 编辑器，帮助用户很方便地实现代码中的符号跟踪。

下面简单介绍使用 vim + ctags 对 GCC 4.4.0 源代码分析的过程。为了描述方便，全书使用 ${GCC_SOURCE} 来表示 GCC 4.4.0 代码所在的顶层目录。

（1）使用 yum 工具安装 ctags 程序。

```
[root@localhost ~]# sudo yum install ctags
```

（2）使用 wget 工具从 GCC 源代码的镜像站点下载 GCC 4.4.0 的源代码文件。

```
[GCC@localhost ~]$ wget -c http://mirror1.babylon.network/gcc/releases/gcc-
4.4.0/gcc-4.4.0.tar.bz2
--2015-05-19 10:06:52--  http://mirror1.babylon.network/gcc/releases/gcc-4.4.0/
gcc-4.4.0.tar.bz2
Resolving mirror1.babylon.network... 5.135.162.176, 2001:41d0:8:e5b0::1
Connecting to mirror1.babylon.network|5.135.162.176|:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 62708198 (60M) [application/octet-stream]
Saving to: "gcc-4.4.0.tar.bz2"
100%[======================================>] 62,708,198   211K/s   in 4m 51s
2015-05-19 10:11:45 (210 KB/s) - "gcc-4.4.0.tar.bz2" saved [62708198/62708198]
```

（3）使用 tar 工具对源代码进行解压。

```
[GCC@localhost vim-ctags]$ tar xjvf gcc-4.4.0.tar.bz2
```

（4）进入 gcc-4.4.0 目录，运行 ctags，生成 tags 文件。

```
[GCC@localhost vim-ctags]$ cd gcc-4.4.0
[GCC@localhost gcc-4.4.0]$ ctags -R
[GCC@localhost gcc-4.4.0]$ ls -l tags
-rw-rw-r--. 1 GCC GCC 52296910 May 19 10:14 tags
```

可以看出，生成的 tags 文件的大小为 52 296 910 字节，包含的 tags 信息非常多，有兴趣的读者可以使用文本工具打开该 tags 文件，查看其中的内容。

（5）使用 vim 查看 GCC 4.4.0 源代码。

在查看源代码时，需要先对代码的结构进行大致了解，从合适的入口开始分析。一般来讲，按照程序的执行流程来分析代码的结构及其运行过程是一个不错的选择，因此，笔者选择从 ${GCC_SOURCE}/gcc/main.c 文件入手，使用 vim 来查看该文件。

这里需要特别说明的是，执行 vim 命令时的当前工作目录应该和 tags 文件所在的目录相同，这样才能在 vim 中使用 tags 文件。上面执行 ctags 命令产生的 tags 文件在 ${GCC_SOURCE} 目录中，因此，运行 vim 时，当前工作目录应该切换到 ${GCC_SOURCE} 目录中。

```
[GCC@localhost vim-ctags]$ cd gcc-4.4.0
[GCC@localhost gcc-4.4.0]$ vim gcc/main.c
```

系统显示如图 2-1 所示。

显然，在该文件中，读者感兴趣的是 main 函数中调用的 toplev_main 函数的实现。此时，只需要将光标移动到 toplev_main 函数名称上，并按 Ctrl+] 组合键，此时 vim 会根据 tags 中提供的信息，自动打开函数 toplev_main 所在的文件 gcc/toplev.c，并且让光标停留在该函数的开始，如图 2-2 所示。

在分析了 toplev_main 函数的实现过程后，如果需要回到 main 函数处，只需要按 Ctrl+O 组合键即可。

当然，对于代码中所有的变量声明、类型声明、函数名称等标签，均可以使用上述方法

快速查看其定义及实现，避免了分析源代码中繁重的搜索工作，极大地提高了代码阅读和分析的效率。

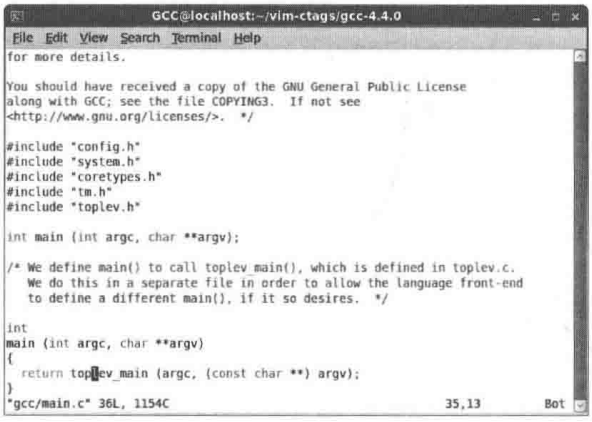


图 2-1 使用 vim 编辑查看文件

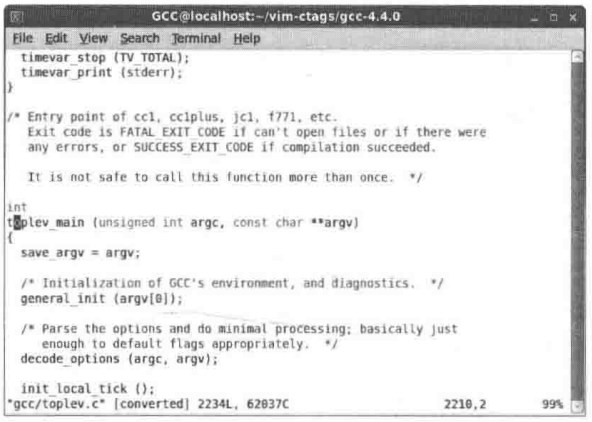


图 2-2 vim 中利用 tags 跳转到函数实现

2.2 GNU gdb 调试工具

调试工具是代码分析中至关重要的工具之一。在使用 vim+ctags 查看代码时，经常会遇到难以理解的部分，此时，可以借助调试工具，对代码的运行过程进行跟踪，通过跟踪运行过程以及关键数据的变化，可以从程序执行的过程中理解源代码的功能。

调试工具有很多种，最常用的是 GNU gdb 工具。下面通过一个例子，介绍如何使用 gdb，这些调试命令几乎就是笔者调试程序的所有命令，简单且实用。关于完整的 gdb 的使用，请参与 GNU gdb 文档，或者使用 man gdb 进行在线查询。

本例主要使用 gdb 来跟踪 GCC 的运行过程，因此，需要事先编译 GCC 源代码（编译时需要使用 -g 选项），生成可执行的编译程序 cc1，下面利用 gdb 对 cc1 程序的运行进行跟踪。

首先，可以在程序入口处设置断点（Break Point）：

```
[GCC@localhost paag-gcc]$ gdb host-i686-pc-linux-gnu/gcc/cc1
GNU gdb (GDB) Red Hat Enterprise Linux (7.2-75.el6)
Copyright (C) 2010 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "i686-redhat-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /home/GCC/paag-gcc/host-i686-pc-linux-gnu/gcc/cc1...done.
(gdb) b main                    ← 设置执行断点
Breakpoint 1 at 0x80c253d: file ../.././gcc/main.c, line 35.
(gdb) info break                ← 查看断点设置情况
Num     Type           Disp Enb Address    What
```

```
1       breakpoint     keep y   0x080c253d in main at ../.././gcc/main.c:35
(gdb)
```

执行程序，gdb 执行到断点处会自动停止，返回交互界面。

```
(gdb) run                    ← 运行程序
Starting program: /home/GCC/paag-gcc/host-i686-pc-linux-gnu/gcc/cc1
Breakpoint 1, main (argc=1, argv=0xbffff434) at ../.././gcc/main.c:35
35      return toplev_main (argc, (const char **) argv);
Missing separate debuginfos, use: debuginfo-install glibc-2.12-1.149.el6.i686
gmp-4.3.1-7.el6_2.2.i686 mpfr-2.4.1-6.el6.i686
```

单步跟踪程序的执行，step 命令和 next 命令均可以进行单步跟踪，二者的主要区别在于 step 在单步执行函数代码时，会进入被调用的函数，而 next 则会将函数调用看作“单步”，一次执行完一个函数的调用。对于其他代码，step 和 next 命令的功能基本相同。

此时可以看到，使用 run 命令执行程序后，程序执行到前面定义的断点处暂停执行。

如果此时需要查看 toplev_main 函数的执行细节，应该使用 step 命令进入该函数。

```
(gdb) step                   ← 单步跟踪
toplev_main (argc=1, argv=0xbffff434) at ../.././gcc/toplev.c:2212
```

对于程序执行过程中，需要查看某些变量的值，可以使用 print 命令。

```
(gdb) print argc             ← 打印变量值
$1 = 1
(gdb) print argv[0]          ← 打印变量值
$2 = 0xbffff5b5 "/home/GCC/paag-gcc/host-i686-pc-linux-gnu/gcc/cc1"
```

查看变量的值可以使用 print 命令，如果在每一条指令后都需要查看某些变量的值，使用 print 显得有些烦琐，可以使用 display 命令，设置显示的变量。

```
(gdb) disp argc              ← 设置变量查看
1: argc = 1
(gdb) next                   ← 单步执行
2215        general_init (argv[0]);
1: argc = 1
(gdb) next
2219        decode_options (argc, argv);
1: argc = 1
(gdb) next
2221        init_local_tick ();
1: argc = 1
```

可以看出，每执行一步，变量 argc 的值都会输出显示。

当需要连续执行程序时，可以使用 continue 命令，程序则恢复运行，直到下一个断点处再次暂停运行。

通常，在执行到某个断点处时，当需要了解当前函数的调用情况时，可以使用 bt 命令（backtrace）。