

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。

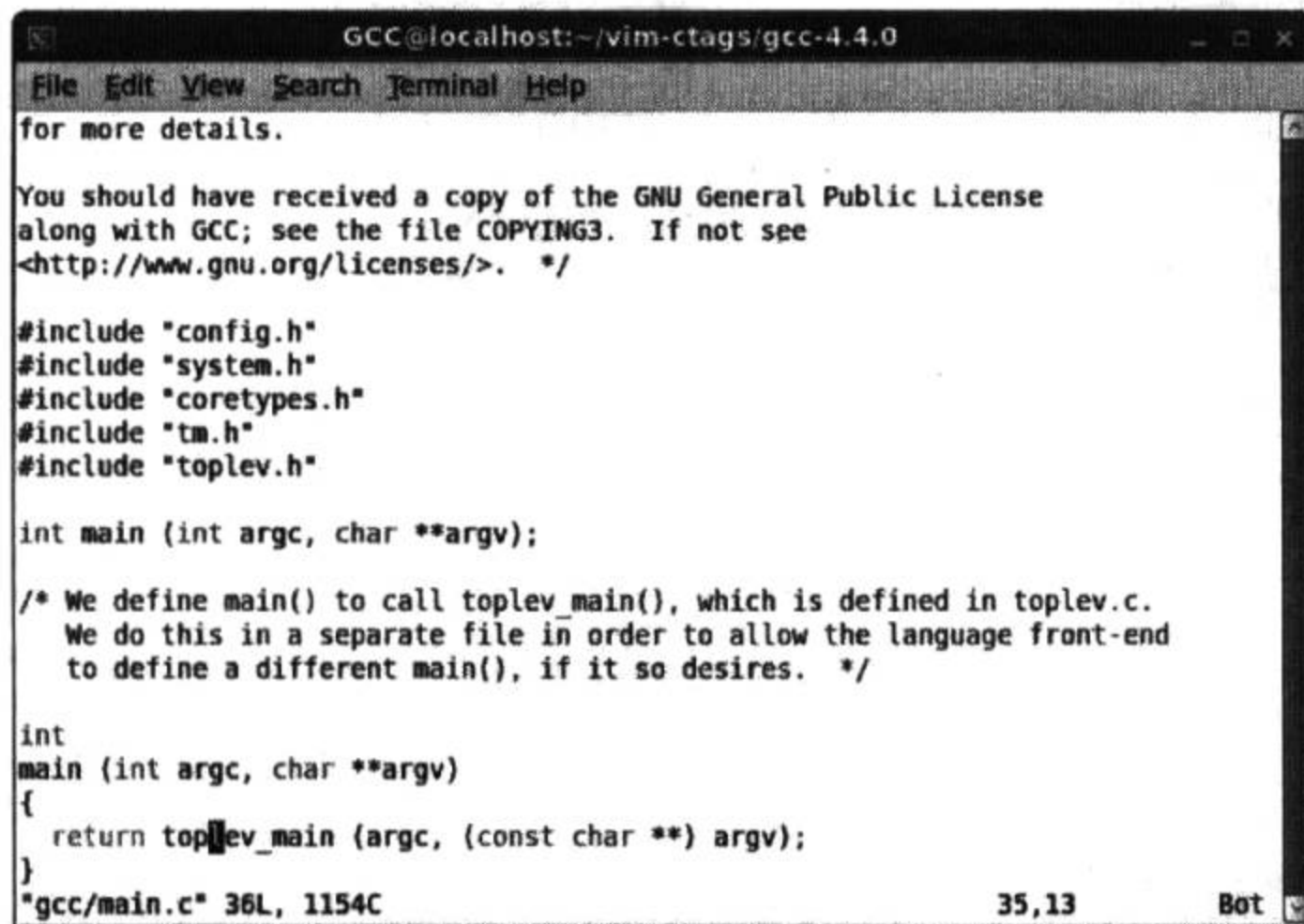
显然，在该文件中，读者感兴趣的是 main 函数中调用的 `toplev_main` 函数的实现。此时，只需要将光标移动到 `toplev_main` 函数名称上，并按 `Ctrl+]` 组合键，此时 vim 会根据 tags 中提供的信息，自动打开函数 `toplev_main` 所在的文件 `gcc/toplev.c`，并且让光标停留在此函数的开始，如图 2-2 所示。

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

6 深入分析GCC

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。



```
GCC@localhost:~/vim-ctags/gcc-4.4.0
File Edit View Search Terminal Help
for more details.

You should have received a copy of the GNU General Public License
along with GCC; see the file COPYING3. If not see
<http://www.gnu.org/licenses/>. */

#include "config.h"
#include "system.h"
#include "coretypes.h"
#include "tm.h"
#include "toplev.h"

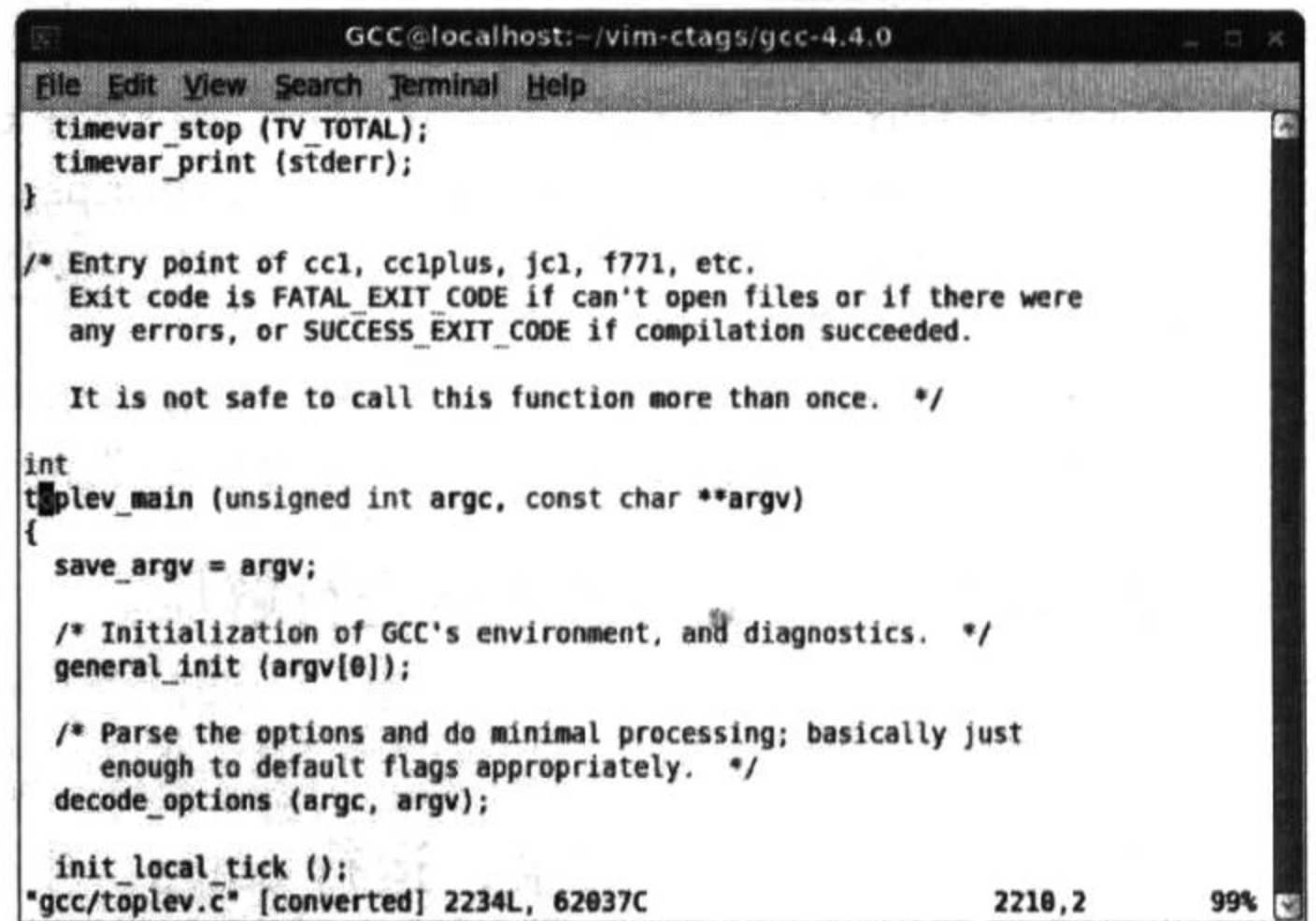
int main (int argc, char **argv);

/* We define main() to call toplev_main(), which is defined in toplev.c.
   We do this in a separate file in order to allow the language front-end
   to define a different main(), if it so desires. */

int
main (int argc, char **argv)
{
  return toplev_main (argc, (const char **) argv);
}

"gcc/main.c" 36L, 1154C Bot
```

图 2-1 使用 vim 编辑查看文件



```
GCC@localhost:~/vim-ctags/gcc-4.4.0
File Edit View Search Terminal Help
timevar_stop (TV_TOTAL);
timevar_print (stderr);
}

/* Entry point of cc1, cc1plus, jcl, f771, etc.
   Exit code is FATAL_EXIT_CODE if can't open files or if there were
   any errors, or SUCCESS_EXIT_CODE if compilation succeeded.

   It is not safe to call this function more than once. */

int
tplev_main (unsigned int argc, const char **argv)
{
  save_argv = argv;

  /* Initialization of GCC's environment, and diagnostics. */
  general_init (argv[0]);

  /* Parse the options and do minimal processing; basically just
     enough to default flags appropriately. */
  decode_options (argc, argv);

  init_local_tick ();

  "gcc/toplev.c" [converted] 2234L, 62037C 2210,2 99%
```

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   file ../.././gcc/main.c, line 35.
```



```
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)。

8 深入分析GCC

```
(gdb) bt                                ← 显示函数调用的堆栈
#0 toplev_main (argc=1, argv=0xbffff434) at ../../gcc/toplev.c:2212
#1 main (argc=1, argv=0xbffff434) at ../../gcc/main.c:35
```

可以看出当前执行的函数为 `toplev_main` 函数，其调用者为函数 `main`，并且这两个函数所在的文件及位置信息也在 `bt` 的输出中给出。`bt` 命令的输出可以很详细地展示当前函数的调用关系，对于理解程序的执行流程非常有帮助。

另外，`gdb` 在输入命令时，如果输入命令的开始部分可以完全确定一个命令时，则可以简写该命令，例如，一般用户经常将命令 `run` 简写为 `r`，`step` 命令简写为 `s`，`next` 命令简写为 `n`，`continue` 命令简写为 `c` 等，如果用户没有输入命令，直接按回车键，则 `gdb` 默认会执行上一次输入的命令。例如在单步跟踪时，如果输入了命令 `next`，后续单步跟踪则可以只需要按 `[Enter]` 键就可以了。这些规律，读者可以在使用过程中不断总结，提高调试效率。

另外，还有其他众多的调试工具，这些工具大都对 `gdb` 程序进行了封装，例如 `cgdb`，可以提供一些方便地实现源代码查看等其他很有特色的功能，其官网地址为 <http://cgdb.sourceforge.net/>。可以通过以下代码进行 `cgdb` 程序的安装。

```
[root@localhost ~]# wget -c http://prdownloads.sourceforge.net/cgdb/cgdb-0.6.6.tar.gz?download
[root@localhost cgdb-0.6.6]# tar xzvf cgdb-0.6.6.tar.gz
[root@localhost cgdb-0.6.6]# cd cgdb
[root@localhost cgdb-0.6.6]# yum install readline*
[root@localhost cgdb-0.6.6]# ./configure
[root@localhost cgdb-0.6.6]# make; make install
```

例如，可以使用 `cgdb` 对 `cc1` 进行调试。

```
[GCC@localhost gcc-4.4.0]$ cgdb ~/paag-gcc/host-i686-pc-linux-gnu/gcc/cc1
```

界面如图 2-3 所示，可以看到 `cgdb` 中能够很方便地查看源代码。关于 `cgdb` 的使用请查阅相关文档，不再赘述。

2.3 GNU binutils 工具

在分析 GCC 代码时，尤其是后端代码生成的过程中，经常需要对编译生成的目标文件进行分析，包括编译生成的汇编代码、目标文件等，此时，如果能够熟练使用 GNU binutils 工具链中的工具，无疑将对分析非常有用。GNU binutils 工具的源代码及介绍参见 GNU 的官网：<http://www.gnu.org/software/binutils/>，其中主要工具如表 2-1 所示。

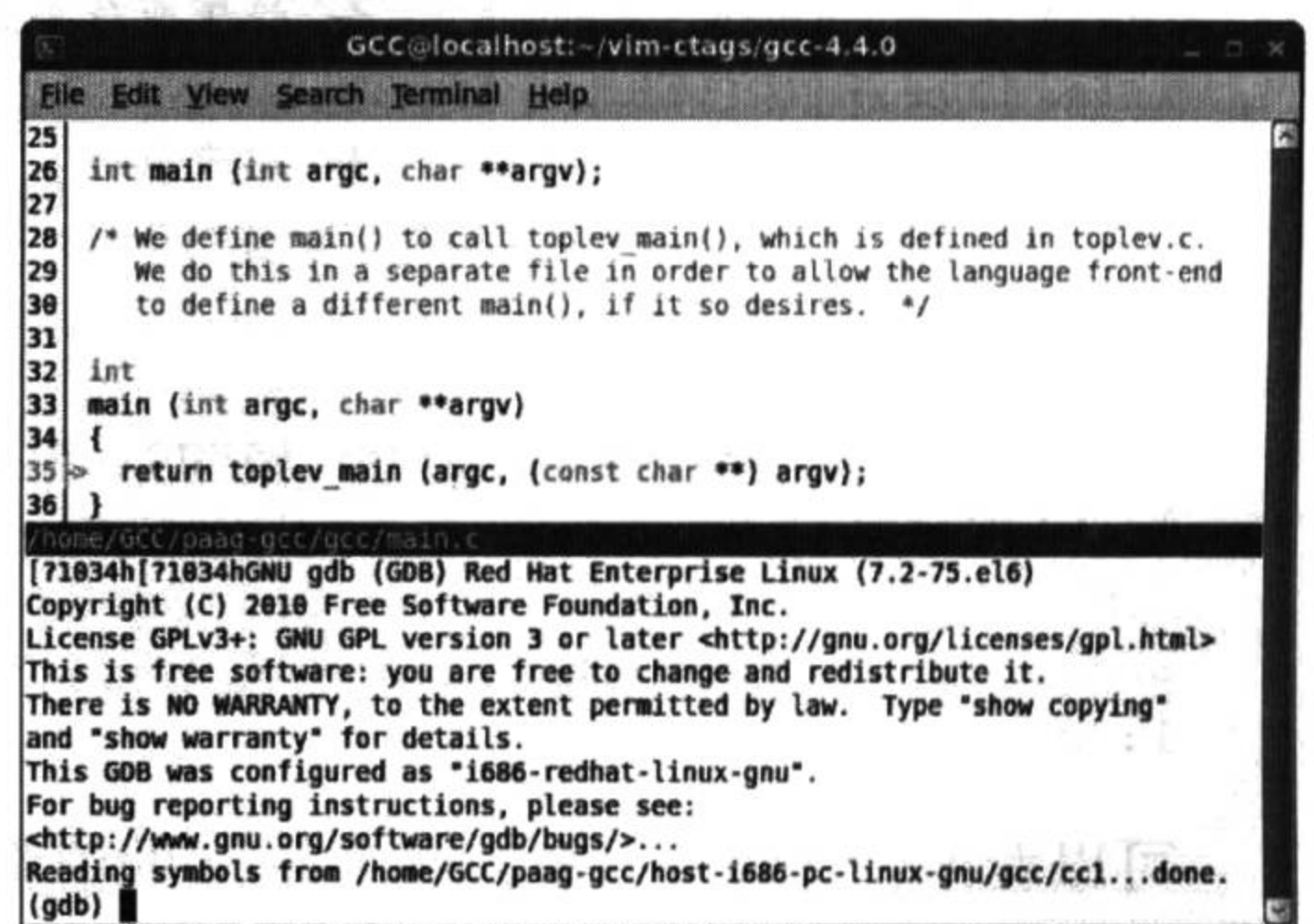


图 2-3 cgdb 界面

表 2-1 GNU binutils 中的主要工具

工具名称	作 用
ld	GNU 链接器
as	GNU 汇编器
ar	归档文件 (archives) 的打包工具, 用来生成静态或者动态链接库
ranlib	生成归档文件内容的索引
nm	显示目标文件中的符号信息
objdump	显示目标文件信息, 可以用目标文件的反汇编等
objcopy	目标文件的复制, 可以完成目标文件格式的转换等
readelf	显示 ELF 格式目标文件的信息
size	显示目标文件或者归档文件中节区 (Section) 的大小
strings	显示文件中的可打印字符串的信息
strip	去除目标文件中的符号信息

例如, 对于如下的源代码:

```
[GCC@localhost test]$ cat test.c
int main(){
    int i=0, sum=0;
    sum = sum + i;
    return sum;
}
```

可以使用 objdump 进行目标代码的反汇编:

```
[GCC@localhost test]$ gcc -c -o test.o test.c
[GCC@localhost test]$ objdump -d test.o
test.o:      file format elf32-i386
```

Disassembly of section .text:

```
00000000 <main>:
   0:      55                push    %ebp
   1:      89 e5             mov     %esp,%ebp
   3:      83 ec 10          sub     $0x10,%esp
   6:      c7 45 f8 00 00 00 00 movl    $0x0,-0x8(%ebp)
   d:      c7 45 fc 00 00 00 00 movl    $0x0,-0x4(%ebp)
  14:      8b 45 f8             mov     -0x8(%ebp),%eax
  17:      01 45 fc             add     %eax,-0x4(%ebp)
  1a:      8b 45 fc             mov     -0x4(%ebp),%eax
  1d:      c9                leave
  1e:      c3                ret
```

可以使用 nm 查看目标文件中的符号信息:

```
[GCC@localhost test]$ nm test.o
00000000 T main
```

也可以使用 readelf 工具查看目标文件的 ELF 信息。

```
[GCC@localhost test]$ readelf -a test.o
ELF Header:
  Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
```


10 深入分析GCC

```
Class: ELF32
Data: 2's complement, little endian
Version: 1 (current)
OS/ABI: UNIX - System V
ABI Version: 0
Type: REL (Relocatable file)
Machine: Intel 80386
Version: 0x1
Entry point address: 0x0
Start of program headers: 0 (bytes into file)
Start of section headers: 200 (bytes into file)
Flags: 0x0
Size of this header: 52 (bytes)
Size of program headers: 0 (bytes)
Number of program headers: 0
Size of section headers: 40 (bytes)
Number of section headers: 9
Section header string table index: 6
```

Section Headers:

[Nr]	Name	Type	Addr	Off	Size	ES	Flg	Lk	Inf	Al
[0]		NULL	00000000	000000	000000	00		0	0	0
[1]	.text	PROGBITS	00000000	000034	00001f	00	AX	0	0	4
[2]	.data	PROGBITS	00000000	000054	000000	00	WA	0	0	4
[3]	.bss	NOBITS	00000000	000054	000000	00	WA	0	0	4
[4]	.comment	PROGBITS	00000000	000054	00002e	01	MS	0	0	1
[5]	.note.GNU-stack	PROGBITS	00000000	000082	000000	00		0	0	1
[6]	.shstrtab	STRTAB	00000000	000082	000045	00		0	0	1
[7]	.symtab	SYMTAB	00000000	000230	000080	10		8	7	4
[8]	.strtab	STRTAB	00000000	0002b0	00000d	00		0	0	1

Key to Flags:

W (write), A (alloc), X (execute), M (merge), S (strings)
I (info), L (link order), G (group), x (unknown)
O (extra OS processing required) o (OS specific), p (processor specific)

There are no section groups in this file.

There are no program headers in this file.

There are no relocations in this file.

There are no unwind sections in this file.

Symbol table '.symtab' contains 8 entries:

Num:	Value	Size	Type	Bind	Vis	Ndx	Name
0:	00000000	0	NOTYPE	LOCAL	DEFAULT	UND	
1:	00000000	0	FILE	LOCAL	DEFAULT	ABS	test.c
2:	00000000	0	SECTION	LOCAL	DEFAULT	1	
3:	00000000	0	SECTION	LOCAL	DEFAULT	2	
4:	00000000	0	SECTION	LOCAL	DEFAULT	3	
5:	00000000	0	SECTION	LOCAL	DEFAULT	5	
6:	00000000	0	SECTION	LOCAL	DEFAULT	4	
7:	00000000	31	FUNC	GLOBAL	DEFAULT	1	main

No version information found in this file.

2.4 shell 工具及 graphviz 绘图工具

为了更好地分析 GCC 的运行过程,可以使用 GCC 支持的一些编译选项,例如, `-fdump-tree-all`、`-fdump-ipa-all`、`-fdump-rtl-all` 等,这样编译过程中将产生大量的中间运行结果信息,帮助用户理解 GCC 的处理细节。另外,用户也可以根据需要在源代码中增加适当的调试代码,从而输出一些运行时的中间信息。对这些输出结果进行高效分析,从中提取有价值的信息是 GCC 分析过程中非常关键的一种途径。

笔者认为,借助于 Linux shell 命令的强大字符串处理功能,可以极大地提高信息处理的效率。例如,可以使用 `grep` 对输出中的特定模式进行匹配,利用 `sed` 对输出的信息进行各种强大的编辑处理,包括替换、修改等,利用 `awk` 可以对输出结果进行进一步的处理。建议读者熟练使用 `grep`、`sed`、`awk` 等工具,并能熟练编写一些简单的处理脚本。

另一方面,图形直观生动,擅长展示逻辑关系,因此,为了说明问题,往往需要对处理结果进行图形方式的展示, `graphviz` 提供的绘图工具 (<http://www.graphviz.org/>) 就是笔者进行 GCC 分析时常用的图形生成工具。

例如,对于如下的源代码 `test.c`:

```
[GCC@host2 g2r]$ cat test.c
int global_int = 0;
int main(int argc, char *argv[])
{
    int i;
    static int static_sum=0;
    int array[10]={0,1,2,3,4,5,6,7,8,9};

    for(i=global_int; i<10; i++){
        int j=i*2;
        static_sum = static_sum + j + array[i];
        if(static_sum>1000) goto Label_RET;
    }
Label_RET:
    return static_sum;
}
```

通过在 GCC 中增加调试代码,可以生成 `main` 函数的控制流图文件 `Control_Flow.dot`。

```
[GCC@host2 g2r]$ cat Control_Flow.dot
digraph G {
    node [shape = record];
    0 [label = "{ENTRY}"];
    0 -> 2 [style=solid, label=fallthru];
    2 [label = "{BB-2}"];
    2 -> 6 [style=solid, label=fallthru];
    3 [label = "{BB-3}"];
    3 -> 4 [style=solid, label=true];
    3 -> 5 [style=solid, label=false];
    4 [label = "{BB-4}"];
```



```
4 -> 7 [style=solid, label=fallthru];
5 [label = "{BB-5}"];
5 -> 6 [style=solid, label=fallthru];
6 [label = "{BB-6}"];
6 -> 3 [style=solid, label=true];
6 -> 7 [style=solid, label=false];
7 [label = "{BB-7}"];
7 -> 8 [style=solid, label=fallthru];
8 [label = "{BB-8}"];
8 -> 1 [style=solid];
1 [label = "{EXIT}"];
}
```

显然，该控制流图是不直观、不容易理解的，然而通过将 Control_Flow.dot 中描述的逻辑关系转换成 graphviz 的图形脚本，就可以利用 graphviz 中 dot 工具生成其图示结果 Control_Flow.png，如图 2-4 所示。

```
dot -Tpng -o Control_Flow.png Control_Flow.dot
```

可以看出，使用图形表示可以非常直观地展示程序中的控制流程，也为代码分析提供了最直观形象的辅助。

再举一例。在分析 GCC 的 AST 生成及 GIMPLE 生成等过程中，需要了解 AST 节点的具体内容及其相互关系，此时，也可以通过对 GCC 生成的 AST 中间结果进行脚本的处理，并生成其图示结果，例如图 2-5 给出了上述源代码中 sum=a+b 语句对应的关键 AST 节点及其相互关系，该结果形象直观，节点之间的关系清晰，对于分析 AST 的生成和 GIMPLE 转换等都具有非常重要的意义。

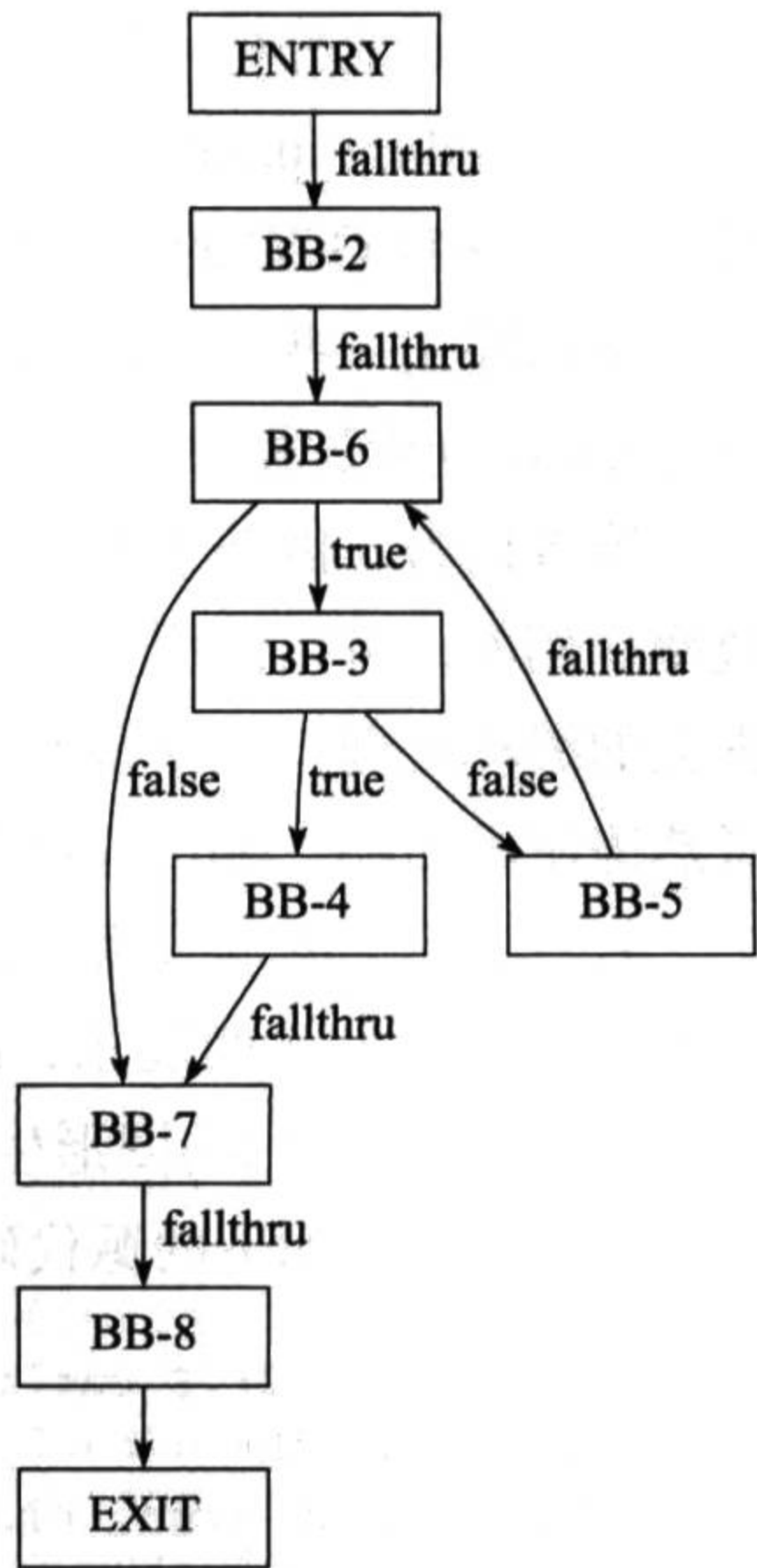


图 2-4 函数控制流程图的图示

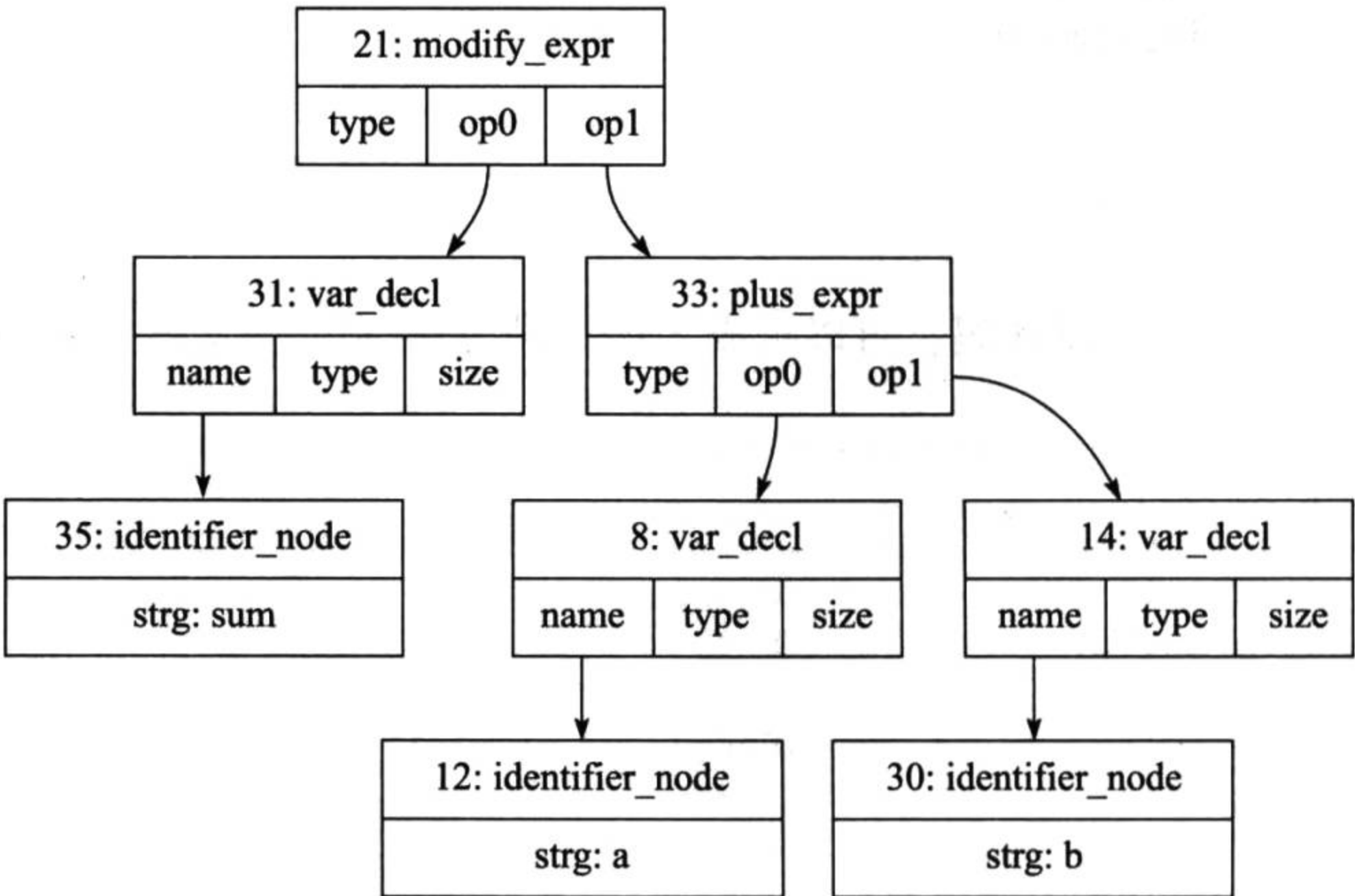


图 2-5 sum=a+b 对应的 AST 片段图示

2.5 GCC 调试选项

GCC 本身对包含了众多的调试选项，既可以为用户程序生成调试信息，也可以将 GCC 运行过程中的关键信息保存在文件或输出在终端上，常用的调试选项如表 2-2 所示。如果需要了解 GCC 在处理的各个阶段里中间表示的具体内容，或者需要了解 GCC 中某个处理过程对于中间表示的处理细节时，就可以使用表 2-2 中给出的各种 GCC 调试选项，输出 GCC 运行过程中所生成的中间表示的调试信息和处理过程细节，并结合 GCC 的代码，从而了解 GCC 的具体工作细节。

表 2-2 GCC 主要的调试选项

调试选项形式及作用	switch/pass 的主要取值			options 的主要取值	举例
-fdump-tree-switch -fdump-tree-switch-options 输出 GCC 编译过程中与 AST、GIMPLE 等树节点中间表示相关的调试信息	original	storeccp	dom	slim	-fdump-tree-all
	optimized	pre	dse	raw	-fdump-tree-original-raw
	gimple	fre	phiopt	details	fdump-tree-cfg-all
	cfg	copyprop	forwprop	stats	
	vsg	store_copyprop	copyrename	blocks	
	ch	dce	nrv	vops	
	ssa	mudflap	vect	lineno	
	alias	sra	vrp	uid	
	ccp	sink	all	verbose	
				all	
-fdump-ipa-switch 输出与 IPA 相关的调试信息	all	cgraph	inline		fdump-ipa-cgraph fdump-ipa-all
-fdump-rtl-pass 输出与 RTL 中间表示相关的调试信息	alignments	init	sibling		fdump-rtl-ira
	asmcons	initvals	split1		fdump-rtl-sched1
	auto_inc_dec	into_cfglayout	sms		fdump-rtl-expand
	barriers	ira	stack		fdump-rtl-all
	bbpart	jump	subreg1		
	bbro	loop2	subreg2		
	bt11	mach	subreg1		
	bypass	mode_sw	unshare		
	combine	rnreg	vartrack		
	compgotos	outof_cfglayout	vregs		
	cel	peephole2	web		
	cprop_hardreg	postreload	regclass		
	csa	pro_and_epilogue	subregs_of		
	cse1	regmove	mode_finish		
	dce	sched1	dfinit		
	eh_ranges	see	dfinish		
	expand	shorten	all		

例 2-1 GCC 调试选项的使用

假设有如下的源代码：

14 深入分析GCC

```
[GCC@localhost test]$ cat test.c
int main(){
    int i=0, sum=0;
    sum = sum + i;
    return sum;
}
```

为了了解 GCC 对该文件编译过程中的主要处理过程，可以使用如下命令输出 GCC 处理过程的主要调试信息和 workflows。

```
[GCC@localhost test]$ ~/paag-gcc/host-i686-pc-linux-gnu/gcc/cc1 -fdump-tree-all -fdump-rtl-all test.c
[GCC@localhost test]$ ls test.c*
test.c                                test.c.123t.optimized          test.c.168r.asmcons
test.c.001t.tu                        test.c.125t.blocks             test.c.171r.subregs_of_mode_init
test.c.003t.original                  test.c.126t.final_cleanup      test.c.172r.ira
test.c.004t.gimple                    test.c.128r.expand             test.c.173r.subregs_of_mode_finish
test.c.006t.vcg                       test.c.129r.sibling            test.c.176r.split2
test.c.007t.useless                   test.c.131r.initvals           test.c.178r.pro_and_epilogue
test.c.010t.lower                     test.c.132r.unshare            test.c.192r.stack
test.c.011t.ehopt                     test.c.133r.vregs              test.c.193r.alignments
test.c.012t.eh                        test.c.134r.into_cfglayout     test.c.196r.mach
test.c.013t.cfg                       test.c.135r.jump               test.c.197r.barriers
test.c.014t.cplxlower0                test.c.154r.reginfo            test.c.200r.eh_ranges
test.c.015t.veclower                  test.c.157r.outof_cfglayout    test.c.201r.shorten
test.c.021t.cleanup_cfg               test.c.163r.split1             test.c.202r.dfinish
test.c.023t.ssa                       test.c.165r.dfinit             test.c.203t.statistics
test.c.038t.release_ssa               test.c.166r.mode_sw
```

可以看出，此时输出的各种调试文件名称格式为：test.c.nnn[r/t].name，其中 nnn 为一个编号，t 表示该处理过程是基于 tree 的 GIMPLE 处理过程，r 表示该处理过程是基于 RTL 的处理过程。如果读者关注函数控制流图（CFG，Control Flow Graph）的信息，那么可以打开 test.c.013t.cfg 文件，查看其中的具体内容。内容如下：

```
[GCC@localhost test]$ cat test.c.013t.cfg
;; Function main (main)
Merging blocks 2 and 3
main ()
{
    int sum;
    int i;
    int D.1234;

<bb 2>:
    i = 0;
    sum = 0;
    sum = sum + i;
    D.1234 = sum;
    return D.1234;
}
```


其中就包含了例子中给出函数的控制流图，如果想了解更详细的 CFG 信息，也可以使用如下的编译形式：

```
[GCC@localhost test]$ ~/paag-gcc/host-i686-pc-linux-gnu/gcc/cc1 -fdump-tree-cfg-
all test.c
[GCC@localhost test]$ cat test.c.013t.cfg
;; Function main (main)
Scope blocks:
{ Scope block #0
  intD.0 iD.1232; (unused)
  intD.0 sumD.1233; (unused)
}
Pass statistics:
-----
Merging blocks 2 and 3
main ()
{
  intD.0 sumD.1233;
  intD.0 iD.1232;
  intD.0 D.1234;

  # BLOCK 2
  # PRED: ENTRY (fallthru)
  iD.1232 = 0;
  sumD.1233 = 0;
  sumD.1233 = sumD.1233 + iD.1232;
  D.1234 = sumD.1233;
  return D.1234;
  # SUCC: EXIT
}
```

可以看出，GCC 编译时会生成更加详细的 CFG 信息。

读者也可以根据自己的需要，合理地使用表 2-2 中的调试选项，输出 GCC 编译过程中感兴趣的调试信息，从而分析 GCC 的工作细节。