Cflow 使用详解

朝歌 译

2014-11-8

目录

1、cflow 简介

cflow 工具用于分析 C 语言实现的源文件集并输出各个函数之间的依赖关系图。cflow 可以生成两种类型的图：正向图和逆向图。正向图从 main()函数开始，递归显示 main()函数调用的所有函数。相反，逆向图是一个子图的集合，使用递归命令时，为每个函数列出它的调用者。

除了这两种输出模式，cflow 也可以为输入文件中的所有遇到的符号生成一个交叉引用列表。

该程序还提供了对将出现在输出中出现的符号的详细控制，允许忽略用户不感兴趣的符号。详细的输出格式也是可配置的。

2、使用 cflow 分析程序的简要方法

我们先从一个例子开始熟悉 GNU cflow 的用法。假设你已经有了一个 whoami 命令的简单实现，并且你想获得它的函数依赖关系图。下面的是程序：

```
/* whoami.c - a simple implementation of whoami utility */
        #include <pwd.h>
        #include <sys/types.h>
        #include <stdio.h>
        #include <stdlib.h>

        int
        who_am_i (void)
        {
           struct passwd *pw;
           char *user = NULL;

           pw = getpwuid (geteuid ());
           if (pw)
              user = pw->pw_name;
           else if ((user = getenv ("USER")) == NULL)
              {
```

```
            fprintf (stderr, "I don't know!\n");
            return 1;
          }
        printf ("%s\n", user);
        return 0;
      }

      int
      main (int argc, char **argv)
      {
        if (argc > 1)
          {
            fprintf (stderr, "usage: whoami\n");
            return 1;
          }
        return who_am_i ();
      }
```

运行 cflow 产生下面的输出：

```
$ cflow whoami.c
      main() <int main (int argc,char **argv) at whoami.c:26>:
          fprintf()
          who_am_i() <int who_am_i (void) at whoami.c:8>:
              getpwuid()
              geteuid()
              getenv()
              fprintf()
              printf()
```

这是一个正向调用图显示了输入文件中的调用者-被调用者依赖关系。每一行以一个函数名开始，紧跟后面有一对括号来指明这是一个函数。如果这个函数在某一个输入文件中定义，这一行会使用一对尖括号，其中显示函数的原型和它被定义的位置。这一行将会以：结尾来表示这个函数调用了其它函数。比如，这一行

```
main() <int main (int argc,char **argv) at whoami.c:25>:
```

显示 main 函数在源文件 whoami.c 中的第 25 行定义，原型是 int main (int argc,char **argv)，最后的冒号表示 main 函数调用了其他函数。

这一行后面的是被 main 函数调用的函数。每一行都会根据嵌套关系有相应的缩进。

通常 cflow 会显示完整的函数原型。然而有时你希望忽略原型的一部分。一些选项可以用来完成这个功能。使用--omit-symbol-names 选项来打印没有函数名的原型。使用--omit-arguments 选项来忽略参数列表。这些选项可以被用做很多用途，其中一个就是使结果图更加的紧凑。为了显示他们的作用，下面是使用上述两种--omit-选项的结果：main() <int () at whoami.c:25>:

默认情况下，cflow 从 main()函数开始输出正向图。当分析一个完整的 C 程序集的时候这是非常的便利的。但是有时候用户可能只想看到从特定函数出发的部分图。使用一main(-m)选项，Cflow 允许选择这样的功能调用。因此，运行$cflow --main who_am_i whoami.c，将会得到下面的结果：

```
who_am_i() <int who_am_i (void) at whoami.c:8>:
        getpwuid()
        geteuid()
        getenv()
        fprintf()
        printf()
```

3、两种类型的流图

在前面我们讨论了正向图，用于展示调用者-被调用者的依赖关系。另一种 cflow 输出是逆向图，列举被调用者-调用者的依赖关系。,为了生成逆向图，运行 cflow 的时候要使用--reverse(-r)选项。比如使用下面的例子：

```
$ cflow --reverse whoami.c
    fprintf():
        who_am_i() <int who_am_i (void) at whoami.c:8>:
            main() <int main (int argc,char **argv) at whoami.c:26>
        main() <int main (int argc,char **argv) at whoami.c:26>
    getenv():
        who_am_i() <int who_am_i (void) at whoami.c:8>:
            main() <int main (int argc,char **argv) at whoami.c:26>
    geteuid():
        who_am_i() <int who_am_i (void) at whoami.c:8>:
```

```
            main() <int main (int argc,char **argv) at whoami.c:26>
    getpwuid():
        who_am_i() <int who_am_i (void) at whoami.c:8>:
            main() <int main (int argc,char **argv) at whoami.c:26>
    main() <int main (int argc,char **argv) at whoami.c:26>
    printf():
        who_am_i() <int who_am_i (void) at whoami.c:8>:
            main() <int main (int argc,char **argv) at whoami.c:26>
    who_am_i() <int who_am_i (void) at whoami.c:8>:
        main() <int main (int argc,char **argv) at whoami.c:26>
```

这个输出包含了几个子图，每一个字图描述了一个特定的函数的调用者。因此，第一个子图说明函数 fprintf 被两个函数调用:who_am_i 和 main。同时他也被 main 函数直接调用。

第一个值得注意的地方是在输出中 who_am_i 重复出现了多次。这是一个详细的输出，为了让输出显得更加简洁，可以使用--brief(-b)选项。比如：

```
$ cflow --brief --reverse whoami.c
    fprintf():
        who_am_i() <int who_am_i (void) at whoami.c:8>:
            main() <int main (int argc,char **argv) at whoami.c:26>
        main() <int main (int argc,char **argv) at whoami.c:26> [see 3]
    getenv():
        who_am_i() <int who_am_i (void) at whoami.c:8>: [see 2]
    geteuid():
        who_am_i() <int who_am_i (void) at whoami.c:8>: [see 2]
    getpwuid():
        who_am_i() <int who_am_i (void) at whoami.c:8>: [see 2]
    main() <int main (int argc,char **argv) at whoami.c:26> [see 3]
    printf():
        who_am_i() <int who_am_i (void) at whoami.c:8>: [see 2]
    who_am_i() <int who_am_i (void) at whoami.c:8>: [see 2]
```

在简要输出中，一旦某个给定的函数被写入，随后的关于该函数的调用实例中将会只包含它的定义和第一次输出行的引用。

如果输出图比较大，查找需要的行数就会比较麻烦（除非你使用 Emacs 的 cflow-mode）。这种情况下可以使用特殊的选项--number (-n)，这样就可以在输出时显示每一行的顺序标号。使用这个选项，上面的输出将变成这样：

```
$ cflow --number --brief --reverse whoami.c
        1 fprintf():
        2       who_am_i() <int who_am_i (void) at whoami.c:8>:
        3               main() <int main (int argc,char **argv) at whoami.c:26>
        4       main() <int main (int argc,char **argv) at whoami.c:26> [see 3]
        5 getenv():
        6       who_am_i() <int who_am_i (void) at whoami.c:8>: [see 2]
        7 geteuid():
        8       who_am_i() <int who_am_i (void) at whoami.c:8>: [see 2]
        9 getpwuid():
       10       who_am_i() <int who_am_i (void) at whoami.c:8>: [see 2]
       11 main() <int main (int argc,char **argv) at whoami.c:26> [see 3]
       12 printf():
       13       who_am_i() <int who_am_i (void) at whoami.c:8>: [see 2]
       14 who_am_i() <int who_am_i (void) at whoami.c:8>: [see 2]
```

当然，--brief 和--number 选项对正向图和逆向图都起作用。

4、各种输出格式

前面所描述的输出格式被称为 GNU 类型。除此之外，cflow 也可以使用 POSIX 产生格式化的输出。这种格式，输出的每一行都以一个参考数字开始，比如，最开始是输出行的顺序号，后面跟随每一个嵌套层的固定长度的缩进。然后如果有的话依次是函数的名字、冒号、函数的原型。紧跟在函数原型后面的是定义的位置（包括文件名和行号）。函数顶一个位置都被尖括号围住。如果函数的定义没有找到，该行将会以一个空的尖括号结尾。

使用格式化输出要么在命令行中通过--format=posix (-f posix)选项指定，要么设置环境变量 POSIXLY_CORRECT 。

使用 POSIX 格式处理我们的样例文件，如下：

```
$ cflow --format=posix whoami.c
        1 main: int (int argc,char **argv), <whoami.c 26>
        2       fprintf: <>
        3       who_am_i: int (void), <whoami.c 8>
        4               getpwuid: <>
```

```
5       geteuid: <>
6       getenv: <>
7       fprintf: <>
8       printf: <>
```

是否在输出中要包含函数的参数列表现在并不清楚。默认情况下 cflow 将会全部打印它们。然而一些程序使用 cflow 分析时希望省略参数列表，这可以使用 --omit-arguments 选项实现。

cflow 未来的版本中将会提供更多的输出格式，包括 XML 和 HTML 输出。目前你可以使用 VCG 工具来创建典型的图。根据 xvcg 来转变输出格式可以使用 cflow2vcg 程序在 GPL 下都是可用的。

Cflow2vcg 期望使用 POSIX 格式图，每层嵌套的缩进为一个水平制表符，第 0 层使用额外的 tab 字符，在函数声明中没有参数列表。这样用可兼容 cflow2vcg 产生的输出格式，调用 cflow 如下：

```
cflow --format=posix --omit-arguments \
        --level-indent='0=\t' --level-indent='1=\t' \
        --level-indent=start='\t'
```

你可以使用下面的脚本来虚拟调用这三个工具:

```
#! /bin/sh

    cflow --format=posix --omit-arguments \
        --level-indent='0=\t' --level-indent='1=\t' \
        --level-indent=start='\t' $* |
      cflow2vcg | xvcg -
```

5、处理递归调用

有时候程序中包含调用自己的函数。GNU 输出格式为这种函数提供了特殊的表示。在结束字符-冒号之前，会有一个标号‘(R)’作为递归函数的标志。随后递归调用处会在行结尾使用‘(recursive: see refline)’标识出。这里的 refline 表示递

归函数的跟定义所处的位置。

为了说明这个，我们考虑下面的程序，他打印递归目录列表，允许在任意嵌套级

截断：

```
#include <sys/types.h>
#include <sys/stat.h>
#include <dirent.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <stdlib.h>
#include <string.h>

/* Return true if file NAME is a directory. */
static int
isdir (char *name)
{
  struct stat st;

  if (stat (name, &st))
    {
      perror (name);
      return 0;
    }
  return S_ISDIR (st.st_mode);
}

static char *ignored_names[] = { ".", "..", NULL };

/* Return true if NAME should not be recursed into */
int
ignorent (char *name)
{
  char **p;
  for (p = ignored_names; *p; p++)
    if (strcmp (name, *p) == 0)
      return 1;
  return 0;
}

int max_level = -1;
```

```
/* Print contents of the directory PREFIX/NAME.
     Prefix each output line with LEVEL spaces. */
void
printdir (int level, char *name)
{
   DIR *dir;
   struct dirent *ent;
   char cwd[512];

   if (!getcwd(cwd, sizeof cwd))
      {
         perror ("cannot save cwd\n");
         _exit (1);
      }
   chdir (name);
   dir = opendir (".");
   if (!dir)
      {
         perror (name);
         _exit (1);
      }

   while ((ent = readdir (dir)))
      {
         printf ("%*.*s%s", level, level, "", ent->d_name);
         if (ignorent (ent->d_name))
            printf ("\n");
         else if (isdir (ent->d_name))
            {
               printf ("/");
               if (level + 1 == max_level)
                  putchar ('\n');
               else
                  {
                     printf (" contains:\n");
                     printdir (level + 1, ent->d_name);
                  }
            }
         else
            printf ("\n");
      }
   closedir (dir);
   chdir (cwd);
}
```

```
int
main (int argc, char **argv)
{
  if (argc < 2)
    {
      fprintf (stderr, "usage: d [-MAX] DIR [DIR...]\n");
      return 1;
    }

  if (argv[1][0] == '-')
    {
      if (!(argv[1][1] == '-' && argv[1][2] == 0))
        max_level = atoi (&argv[1][1]);
      --argc;
      ++argv;
    }

  while (--argc)
    printdir (0, *++argv);

  return 1;
}
```

运行 cflow 处理这个程序生成如下的图：

```
$ cflow --number d.c
    1 main() <int main (int argc,char **argv) at d.c:85>:
    2         fprintf()
    3         atoi()
    4         printdir() <void printdir (int level,char *name) at d.c:42> (R):
    5                 getcwd()
    6                 perror()
    7                 chdir()
    8                 opendir()
    9                 readdir()
   10                 printf()
   11                 ignorent() <int ignorent (char *name) at d.c:28>:
   12                         strcmp()
   13                 isdir() <int isdir (char *name) at d.c:12>:
   14                         stat()
   15                         perror()
   16                         S_ISDIR()
   17                 putchar()
   18                 printdir()
```

```
                    <void printdir (int level,char *name) at d.c:42>
                    (recursive: see 4)
        19          closedir()
```

第 4 行的描述显示 printdir 是递归函数。递归调用是在第 18 行。

6、控制符号类型

有人也许注意到了输出中奇怪的现象：函数_exit 丢失了，虽然它在源文件中被 printdir 调用了两次。这是因为默认情况下 cflow 忽略所有的一下划线开头的符号。为了将这样的符号包含进来，我们使用-i _ (or --include _)命令行选项。继续我们的例子：

```
$ cflow --number -i _ d.c
        1 main() <int main (int argc,char **argv) at d.c:85>:
        2       fprintf()
        3       atoi()
        4       printdir() <void printdir (int level,char *name) at d.c:42> (R):
        5           getcwd()
        6           perror()
        7           _exit()
        8           chdir()
        9           opendir()
       10           readdir()
       11           printf()
       12           ignorent() <int ignorent (char *name) at d.c:28>:
       13               strcmp()
       14           isdir() <int isdir (char *name) at d.c:12>:
       15               stat()
       16               perror()
       17               S_ISDIR()
       18           putchar()
       19           printdir()
                      <void printdir (int level,char *name) at d.c:42>
                      (recursive: see 4)
       20           closedir()
```

通常情况下，参数--include 制定了一个符号类列表。默认的选项行为是输出中包含被请求的类。如果参数是以减号或插入符号开始的，则处理方式正好相反，是

在输出结果中排除这种符号类。

符号类 '_' 包含了所有已下划线开头的符号。另一个有用的符号类是's'，它表示静态函数或数据。默认情况下 ,静态函数是被包含在输出中的。为了省略他们，可以使用所给的-i ^s (or -i -s)选项。我们的样例程序 d.c 中定义了静态函数 isdir，运行 cflow -i ^s，可以在结果图中完全忽略这个函数和它的调用者。

```
$ cflow --number -i ^s d.c
    1 main() <int main (int argc,char **argv) at d.c:85>:
    2       fprintf()
    3       atoi()
    4       printdir() <void printdir (int level,char *name) at d.c:42> (R):
    5           getcwd()
    6           perror()
    7           chdir()
    8           opendir()
    9           readdir()
   10           printf()
   11           ignorent() <int ignorent (char *name) at d.c:28>:
   12               strcmp()
   13           putchar()
   14           printdir()
                  <void printdir (int level,char *name) at d.c:42>
                  (recursive: see 4)
   15           closedir()
```

实际上，非包含符号('^' 或 '-')可以在-i 的参数中的任何地方使用，不仅限于开始。因此，选项-i _^s 表示包含除了静态函数以外的以下划线开头的符号。

-i 选项可以累积使用，所以这个例子可以写为-i _ -i ^s。

一个很重要的一点是，默认情况下 cflo 图中只包含函数。然而你可以使用符号类 "x" 来将变量输出。这个类包含所有的数据符号，包括全局变量和静态变量。

如下例：

```
$ cflow --number -i x d.c
    1 main() <int main (int argc,char **argv) at d.c:85>:
    2       fprintf()
    3       stderr
```

```
 4      max_level <int max_level at d.c:37>
 5      atoi()
 6      printdir() <void printdir (int level,char *name) at d.c:42> (R):
 7          DIR
 8          dir
 9          getcwd()
10          perror()
11          chdir()
12          opendir()
13          readdir()
14          printf()
15          ignorent() <int ignorent (char *name) at d.c:28>:
16              ignored_names <char *ignored_names[] at d.c:24>
17              strcmp()
18          isdir() <int isdir (char *name) at d.c:12>:
19              stat()
20              perror()
21              S_ISDIR()
22              NULL
23          max_level <int max_level at d.c:37>
24          putchar()
25          printdir()
              <void printdir (int level,char *name) at d.c:42>
              (recursive: see 6)
26          closedir()
```

现在，第 3、4、16、23 行显示了数据标号，同时显示了可用的定义。然而请注意第 7、8 行。为什么类型名 DIR 和自动变量 dir 也被作为数据列出？

为了回答这个问题，我们首先描述一线 cflow 对符号的概念定义。程序维持着符号表，使用 C 语言预处理关键字初始化。当解析输入文件时，cflow 更新了这些表。特别的，当遇到一个 typedef 时，它就将这个定义符号注册为数据类型。

现在，DIR 在 d.c 中没有声明，所以 cflow 无法知道这是一个数据类型。所以他认为这是一个变量，这样一来输入：DIR *dir;就被解析为一个表达式，意思是“DIR 乘以 dir”。

当然这是错误的。有两种方式可以帮助 cflow 排除这种混淆。要么详细的声明 DIR 为一个数据类型，要么让 cflow 运行预处理器，这样一来 cflow 就能看到头文件

的内容，并自己做出决定。运行预处理器在下一章中。这一章我们主要集中精力在第一种方法。

命令行选项--symbol (-s)声明这个符号的语法类。这个参数包含了被冒号隔开的两个字符串：--symbol sym:class。

第一个字符串，sym 是 C 语言的符号表的识别码。第二个字符串，class，指定了与这个符号结合的类。特别的如果 class 是‘type’，那么符号 sym 就被记录为 C 语言类型定义。因此，修改上面的输出，运行：

```
$ cflow --number -i x --symbol DIR:type d.c
```

另一个重要的符号类型是参数包装，这是宏的一种，经常用于兼容 ANSI 之前的编译器来保护函数原型中的参数声明的源。比如下面的声明，从/usr/include/resolv.h 中获得，其中__P 是参数包装：

```
void res_npquery __P((const res_state, const u_char *, int, FILE *));
```

为了能让 cflow 处理这样的声明，声明__P 为一个包装，例如：

```
cflow --symbol __P:wrapper *.c
```

在所有的必须使用--symbol 选项的例子中，都是 cflow 不能识别给定的符号的意义，这要么是因为 cflow 不能看到类型的定义，就像‘DIR’的例子；要么是因为宏定义没有展开。这两种情况都可以用下一章藐视的预处理模式来解决。

虽然有了预处理模式，但--symbol 选项还是有用的，我们会在下面的一节中看到：

6.1、语法类

总的来说，符号定义的语法类在 C 语言代码中是可以合法存在的。有如下的类：

关键字；关键字，比如‘if’、‘when’等

修改器；类型修改器，比如这个符号在数据类型后出现，可以修改数据的意义，比如指针‘*’。

修饰符；声明修饰符。能在 C 数据类型的前面和后面声明。你也许会经常声明 gcc 关键字 ‘__extension__’作为修饰符：--symbol __extension__:qualifier

识别码；C 语言识别码。

类型；C 语言数据类型，比如 ‘char’，’int’。

包装器;他有两个用途,第一个是当没有运行预处理的时候声明一个参数包装器。这种用法在前面已经声明了。第二种,他表示任何能出现在声明符前或作为结尾的分号之前和后面可以跟一个括号表达式列表中的任何符号。

我们建议这样对 gcc 使用这个类：‘__attribute__’。

6.2、符号别名

另一个--symbol 选项的用法是定义符号别名。别名是一个与被它引用的符号完全一样的标识。别名可以这样声明：--symbol newsym:=oldsym

这样一来，符号 newsym 就被声明为和 oldsym 完全一样的类型了。

符号别名也能在其他例子中作为定义符号类使用。对一些特殊的关键字是非常有用的，例如 ‘__restrict’： --symbol __restrict:=restrict。

6.3、GCC 初始化

下面的引用集是使用 gcc 时 cflow 的初始化选项。我们建议将他们放到 cflow.rc 文件中：

```
--symbol __inline:=inline
        --symbol __inline__:=inline
        --symbol __const__:=const
        --symbol __const:=const
        --symbol __restrict:=restrict
        --symbol __extension__:qualifier
        --symbol __attribute__:wrapper
```

```
--symbol __asm__:wrapper
--symbol __nonnull:wrapper
--symbol __wur:wrapper
```

7、运行预处理器

cflow 可以在分析之前预处理输入文件，与 cc 在编译之前做的工作一样。这样做可以允许 cflow 准确的处理所有的符号声明，这样就避免了使用--symbol 对特使符号做必要的定义。使用--cpp (--preprocess)选项可以使能预处理功能。对我们的样例 d.c，这种模式输出如下：

```
$ cflow --cpp -n d.c
    1 main() <int main (int argc,char **argv) at d.c:85>:
    2     fprintf()
    3     atoi()
    4     printdir() <void printdir (int level,char *name) at d.c:42> (R):
    5         getcwd()
    6         perror()
    7         chdir()
    8         opendir()
    9         readdir()
   10         printf()
   11         ignorent() <int ignorent (char *name) at d.c:28>:
   12             strcmp()
   13         isdir() <int isdir (char *name) at d.c:12>:
   14             stat()
   15             perror()
   16         putchar()
   17         printdir()
                <void printdir (int level,char *name) at d.c:42>
                (recursive: see 4)
   18         closedir()
```

对比这个图和第 5 章没有使用--cpp 选项的结果。就像你看到的一样，S_ISDIR 不见了。宏被展开了。现在试着运行 cflow --cpp --number -i x d.c 并比较与之对应的没有使用预处理的图。你将会看到不使用--symbol 选项也能生成正确的结果。

默认情况下--cpp 运行/usr/bin/cpp。如果你希望运行其他预处理命令，可以在参

数中用后接等号后的符号指定它。比如，cflow --cpp='cc -E'将会运行 C 编译器来作为预处理器。

8、使用 ASCII 码来生成流图

你可以使用--level-indent 选项来配置 cflow 输出的精确格式。最简单的用法是使用这个选项个边每个嵌套级别的缩进长度 ,他可以指定一个参数值做为每一个嵌套级别的缩进列数。比如下面的指令设置缩进长度为 2，是默认长度的一般：

cflow --level-indent 2 d.c

比如，这样可以保证图在页面的边界之内。

然而--level-indent 能做的不仅仅是这些。每一行都包含下面的图形元素：一个开始标识、一个结束标识和他们之间的几个缩进符。默认情况下，开始标识和结束标识是空的，每个缩进填充包含 4 个空格。

如果--level-indent 选项的参数有 element=string 的形式，它指定一个字符串来代替输出的指定图元素。元素的命名为：

start 开始标识

0 缩进填充 0

1 缩进填充 1

end0 结束标识 0

end1 结束标识 1

为审美会有两种类型的缩进填充和结束标识呢？记住这个流图代表了一个调用树，所以它包含终端节点（叶子），比如这是一个调用结束函数。还有非终端结点（这个调用后接另一个嵌套级的调用）。end0 表示非终端结点，end1 表示终

端节点。

对于缩进填充，缩进填充 1 用于代表图的边界，缩进填充 0 用于保持输出图的格式对齐。

为了论证这个，我们考虑下面的例子程序：

```
/* foo.c */
        int
        main()
        {
                f();
                g();
                f();
        }

        int
        f()
        {
                i = h();
        }
```

我们用下面的字符串表示行元素：

```
start      '::'
0     ' ' (two spaces)
1     '| ' (a vertical bar and a space)
end0      '+-'
end1      '\-'
```

这个对应的命令行是：

```
cflow --level begin=:: --level '0= ' --level '1=| ' --level end0='+-' --level end1='\\-' foo.c
```

注意转义处理 end1 的反斜杠：一般来说，在--level-option 选项中的字符串可以包含常用的转义字符序列，所以反斜杠自己必须被转义。另一个捷径是，云溪在字符串中出现 CxN，C 表示一个单字符、N 表示一个十进制数字。这个符号的意思是“重复字符 C N 次”。然而当字符'x'用在字符串的开头时就失去了它的专用意义。这个命令产生下面的结果：

```
::+-main() <int main () at foo.c:3>:
        ::    +-f() <int f () at foo.c:11>:
```

```
::    | \-h()
::    \-g()
```

因此，我们可以用 ASCII 码表来表示调用树。GNU cflow 提供了一个特殊的选项 --tree (-T)，它是--level '0= ' --level '1=| ' --level end0='+-' --level end1='\\-'参数的快捷方式。下面这个例子是由这个选项生成的流图。源文件 wc.c 是 UNIX 的 wc 命令的简单实现，附加在附录中。

```
cflow --tree --brief --cpp wc.c
        +-main() <int main (int argc,char **argv) at wc.c:127>
           +-errf() <void errf (char *fmt,...) at wc.c:34>
           | \-error_print()
           |          <void error_print (int perr,char *fmt,va_list ap) at wc.c:22>
           |      +-vfprintf()
           |      +-perror()
           |      +-fprintf()
           |      \-exit()
           +-counter() <void counter (char *file) at wc.c:108>
           | +-fopen()
           | +-perrf() <void perrf (char *fmt,...) at wc.c:46>
           | | \-error_print()
           | |        <void error_print (int perr,char *fmt,va_list ap)
           | |        at wc.c:22> [see 3]
           | +-getword() <int getword (FILE *fp) at wc.c:78>
           | | +-feof()
           | | \-isword() <int isword (unsigned char c) at wc.c:64>
           | |      \-isalpha()
           | +-fclose()
           | \-report()
           |          <void report (char *file,count_t ccount,
           |             count_t wcount,count_t lcount) at wc.c:57>
           |      \-printf()
           \-report()
                    <void report (char *file,count_t ccount,
                       count_t wcount,count_t lcount) at wc.c:57> [see 17]
```

9、交叉引用输出

cflow 也可以生成交叉引用列表。这个模式使用-xref(-x)使能。交叉引用输出在

单独的一行列出了每一个符号。每一行显示了标示符和和他出现的源位置。如果这个位置是该符号定义的地方，那么这一行用星号额外标识，并在后面附加它的定义。例如，下面是 d.c 程序的一个交叉引用输出段：

```
printdir * d.c:42 void printdir (int level,char *name)
        printdir      d.c:74
        printdir      d.c:102
```

它显示 printdir 函数在第 42 行定义，被引用两次分别在第 74 和 102 行。

出现在交叉引用列表中的符号受到--include 选项的控制。除此之外在“控制符号类型”一节中讲述的字符类，可用一个额外的符号类 t 控制背 typedef 关键自定义的类型名。

10、配置文件和变量

就像前面看到的一样，cflow 是高度可配置的。不同的命令行选项会有不同规格的影响，像指定显得操作模式或修改一些输出的问题。也许你会经常使用一些选项，而另一些偶尔会使用甚至从不使用。

CFLOW_OPTIONS 环境变量选项是放置在任何具体选项之前的默认选项。例如，你在配置文件中设置 CFLOW_OPTIONS="--format=posix --cpp"，cflow 将会在任何具体选项之前加上这两个选项，--format=posix 和--cpp。

也有一些其他的默认选项可以指定。在最终合并了 CFLOW_OPTIONS 变量的内容之后，cflow 检查环境变量 CFLOWRC 的值。这个只如果不为空，那么就指定将要加载的配置文件的名字。如果 CFLOWRC 为空或者没有定义，cflow 会尝试读取位于用户 home 目录下的.cflowrc 文件。如果这个文件不存在不会有任何错误，但是当这个文件存在但是无法处理时，cflow 将会生成具体的错误消息。

配置文件是按行读取的，空白行和以 shell 注释符'#'开头的行将会被忽略。此外，cflow 将会像 shell 一样将行进行字分割，位于 CFLOW_OPTIONS 后的字将被插入到命令选项中，并且位于其他选项之前。

另外请注意配置文件中的-D 选项，-D 选项的值将会被添加到预处理命令行并被 shell 处理，所以要小心这个引用参数的使用。使用规则是：使用你会在命令行中使用的相同的引用。例如运行 cc -E 作为预处理器，你可以使用下面的配置文件：

```
--cpp='cc -E'
-DHAVE_CONFIG_H
-D__extension__\\\(c\\\)=
```

此外，上面的例子展示了使用 gcc 复制 '__extension__()'构造的一种方式，比如定义一个空字符串。

有时候需要取消某个选项的功能。比如在配置文件中指定--brief 选项时，也许偶然会需要详细的图。取消任何 GNU cflow 的选项不需要参数，只需要在相应的选项名前加上' no-'即可。因此指定--no-brief 可以取消前面--brief 选项的影响。

11、在 Makefiles 中使用 cflow

如果你希望使用 cflow 分析你的项目源，Makefile 或 Makefile.am 是正确的选择。这一章我们将会描述 Makefile.am 的通用用法。如果你不使用 automake,可以从中这一章中减去关于 Makefile 的部分。

下面是设置 Makefile.am 框架的步骤：

- 使用 EXTRA_DIST 变量来添加你的配置文件
- 添加 CFLOW_FLAGS 来指定你要使用的任何 cflow 选项。这个变量可以为空，

它主要是通过运行 CFLOW_FLAGS=...来重载 cflow 选项。

- 对 dir_PROGRAMS 列表中你想为之生成图表的每一个程序，添加下面的描述：

如果你希望使用 cflow 分析你的项目源，Makefile 或 Makefile.am 是正确的选择。

这一章我们将会描述 Makefile.am 的通用用法。如果你不使用 automake,可以从中

这一章中减去关于 Makefile 的部分。

```
program_CFLOW_INPUT=$(program_OBJECTS:.$(OBJEXT)=.c)
program.cflow: program_CFLOW_INPUT cflow.rc Makefile
        CFLOWRC=path-to-your-cflow.rc \
          cflow -o$@ $(CFLOW_FLAGS) $(DEFS) \
                        $(DEFAULT_INCLUDES) $(INCLUDES) $(AM_CPPFLAGS)\
                              $(CPPFLAGS) \
                      $(program_CFLOW_INPUT)
```

使用时替换掉程序名和 path-to-your-cflow.rc 的文件名。如果你不希望使用预处

理，移除除 CFLOW_FLAGS 选项以外的其它选项。

- 如果有几个程序名通过 Makefile.am 建立，你也许会希望新增几个规则，那

  么可以使用一个命令创建所有的文件，比如：

  flowcharts: *prog1*.cflow *prog2*.cflow ...

下面是一个使用 cflow src/Makefile.am 的相关例子：

```
EXTRA_DIST=cflow.rc
      CFLOW_FLAGS=-i^s
      cflow_CFLOW_INPUT=$(cflow_OBJECTS:.$(OBJEXT)=.c)
      cflow.cflow: $(cflow_CFLOW_INPUT) cflow.rc Makefile
                      CFLOWRC=$(top_srcdir)/src/cflow.rc \
                      cflow -o$@ $(CFLOW_FLAGS) $(DEFS) \
                            $(DEFAULT_INCLUDES) $(INCLUDES) $(AM_CPPFLAGS) \
                                        $(CPPFLAGS) \
                $(cflow_CFLOW_INPUT)
```

12、cflow 选项的完整列表

这一章我们以字符序列出 cflow 的所有选项，包括简要的说明。所有的长选项和

短选项都被列出了，所以你可以将这个表作为快速参考。

大部分的选项都有一个相反意义的负选项对应，负选项的命名是对相应的长选项加前缀 no-.这个特性用于取消在配置文件中定义的选项。

-a （--ansi)

假设输入文件使用 ANSI C 编写。目前这意味着不能解析 K&R 声明的函数。这在某些情况下可以加快处理进度。

-b （--brief)

简要输出

--cpp[=command]

运行指定的预处理命令

-D name[=defn] (--define=name[=defn])

预定义名字作为宏。

-d number(--depth=number)

设置流图中嵌套的最大层数。

--debug[=number]

设置调试级别。默认值是 1，如果你开发或调试 cflow 时使用这个选项。

--emacs

让访问文件时告诉 Emacs 使用 cflow 模式输出。

-f name (--format=name)

使用给定的输出格式名。合法的名字是 gnu 和 posix。

-? (--help)

帮助，对每个选项作简要的说明。

-I dir (--include-dir=dir)

增加搜索头文件时，所需要的头文件所在目录。

-i spec (--include=spec)

控制包含符号的数量。spec 是一个字符串，指定了哪一类符号应该包含在输出里。合法字符如下：

- ^ 输出中排除后接字符

+ 输出中包含后接字符（缺省）

_ 以下划线开头的符号

s 静态符号

t 类型定义（只在交叉引用时使用）

x 所有的数据符号，包括外部符号和静态符号

-l

--level-indent=string 指定每个级别缩进时使用的字符串

-m name (--main=name) 设定最开始调用的函数名。

-n (--number) 打印行号

-o file (--output=file) 指定输出文件，默认是’-’,即标准输出

--ommit-arguments 不打印函数声明中的参数列表

--omit-symbol-names 不打印所指定的符号名字，在 posix 模式下可用。

-r (--reverse) 打印逆向调用图

-x (--xref) 只生成交叉引用列表

-p number(--pushdown=number) 初始化令牌栈的大小。默认值 64.令牌栈会自动增长，所以这个选项很少使用。

--preprocess[=command]　使用预处理

-s sym:class

--symbol=sym:class

--symbol=newsym:=oldsym

第一种形式，在语法类 class 中注册符号 sym。合法的额类名是‘keyword’(or ‘kw’), ‘modifier’, ‘qualifier’, ‘identifier’, ‘type’, ‘wrapper’。任何明确的缩写都是可接受的。

第二种形式(使用’:=’分割)，定义 newsym 作为 oldsym 的别名。

-S (--use-indentation)　使用文件缩进作为提示。目前这个意思是右大括号(‘}’)在第零列强制 cflow 结束当前的函数定义。使用这个选项解析可能会对某些远产生误解。

-U name (--undefine=name)　取消之前所做的 name 的定义

-l (--print-level) 打印嵌套层数。层数在输出行的最后打印(如果使用了 --number 或 --format=posix，层数会使用大括号括起来)。

-T (--tree)　使用 ASCII 码打印，调用树。

--usage　提供简短的使用信息。

-v (--verbose) 详细的打印出所有的错误信息。cflow 中的错误信息与 c 编译器的错误信息是不一样的，所以这个选项默认是关闭的。

-V (--version) 打印程序的版本信息

13、GNU Emacs 使用 cflow

cflow 提供了一个 emacs 模块来支持 GNU Emacs 使用 cflow 的主要功能。如果你

机器上使用 emacs，应该将这个模块加载到你的 Emacs 的加载路径里。装载这个模块，你可以在你的.emacs 或 site-start.el 文件中加入下面的行：

```
(autoload 'cflow-mode "cflow-mode")
        (setq auto-mode-alist (append auto-mode-alist
                                      '(("\\.cflow$" . cflow-mode))))
```

第二个声明结合 cflow-mode 到任何有.cflow 的文件。如果你喜欢其他后缀作为流图文件，也可以取代它。你也可以忽略这个选项，如果你不希望在你的图文件中使用任何后缀。这种情况下我们建议使用--emacs 命令行选项。这个选项告诉 Emacs 访问文件的时候使用 cflow 的主要模块。

使用 cflow-mode 的缓冲区被认为是只读的。后接关键字绑定如下：

<E> cflow 模式临时退出并且允许你编辑图文件。恢复 cflow 模式使用 <M-x> cflow-mode <RET>. 这个选项主要用于调试用途，我们不建议编辑图文件。因为编辑图文件可能会改变行内容这样就会让交叉引用的值产生错误。

<x>

扩展当前图的行数。当光标在以‘[see N]’结尾的行时使用该关键字，将会直接转到引用行。使用 exchange-point-and-mark 返回你的检查行。

<R>

如果这个点在一个递归函数处，转到下一层递归，设定标志。

<r>

如果这个点在一个递归函数处，返回它的定义处，设定标志。

<s>

访问引用的源文件并找到函数定义。

14、附录 wc 命令的源文件

这是在 ASCII 码树中使用的文件 wc.c 的源文件：

```
/* Sample implementation of wc utility. */

        #include <stdlib.h>
        #include <stdio.h>
        #include <stdarg.h>

        typedef unsigned long count_t;    /* Counter type */

        /* Current file counters: chars, words, lines */
        count_t ccount;
        count_t wcount;
        count_t lcount;

        /* Totals counters: chars, words, lines */
        count_t total_ccount = 0;
        count_t total_wcount = 0;
        count_t total_lcount = 0;

        /* Print error message and exit with error status. If PERR is not 0,
             display current errno status. */
        static void
        error_print (int perr, char *fmt, va_list ap)
        {
           vfprintf (stderr, fmt, ap);
           if (perr)
              perror (" ");
           else
              fprintf (stderr, "\n");
           exit (1);
        }

        /* Print error message and exit with error status. */
        static void
        errf (char *fmt, ...)
        {
           va_list ap;

           va_start (ap, fmt);
           error_print (0, fmt, ap);
```

```
    va_end (ap);
}

/* Print error message followed by errno status and exit
     with error code. */
static void
perrf (char *fmt, ...)
{
    va_list ap;

    va_start (ap, fmt);
    error_print (1, fmt, ap);
    va_end (ap);
}

/* Output counters for given file */
void
report (char *file, count_t ccount, count_t wcount, count_t lcount)
{
    printf ("%6lu %6lu %6lu %s\n", lcount, wcount, ccount, file);
}

/* Return true if C is a valid word constituent */
static int
isword (unsigned char c)
{
    return isalpha (c);
}

/* Increase character and, if necessary, line counters */
#define COUNT(c)            \
          ccount++;          \
          if ((c) == '\n') \
             lcount++;

/* Get next word from the input stream. Return 0 on end
     of file or error condition. Return 1 otherwise. */
int
getword (FILE *fp)
{
    int c;
    int word = 0;

    if (feof (fp))
```

```
      return 0;

   while ((c = getc (fp)) != EOF)
      {
         if (isword (c))
            {
               wcount++;
               break;
            }
         COUNT (c);
      }

   for (; c != EOF; c = getc (fp))
      {
         COUNT (c);
         if (!isword (c))
            break;
      }

   return c != EOF;
}

/* Process file FILE. */
void
counter (char *file)
{
   FILE *fp = fopen (file, "r");

   if (!fp)
      perrf ("cannot open file `%s'", file);

   ccount = wcount = lcount = 0;
   while (getword (fp))
      ;
   fclose (fp);

   report (file, ccount, wcount, lcount);
   total_ccount += ccount;
   total_wcount += wcount;
   total_lcount += lcount;
}

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

```
{
    int i;

    if (argc < 2)
        errf ("usage: wc FILE [FILE...]");

    for (i = 1; i < argc; i++)
        counter (argv[i]);

    if (argc > 2)
        report ("total", total_ccount, total_wcount, total_lcount);
    return 0;
}
```