GCC 精彩之旅

在为 Linux 开发应用程序时，绝大多数情况下使用的都是 C 语言，因此几乎每一位 Linux 程序员面临的首要问题都是如何灵活运用 C 编译器。目前 Linux 下最常用的 C 语言编译器是 GCC（GNU CompilerCollection），它是 GNU 项目中符合 ANSI C 标准的编译系统，能够编译用 C、C++和 Object C 等语言编写的程序。GCC 不仅功能非常强大，结构也异常灵活。最值得称道的一点就是它可以通过不同的前端模块来支持各种语言，如 Java、Fortran、Pascal、Modula-3 和 Ada 等。

开放、自由和灵活是 Linux 的魅力所在，而这一点在 GCC 上的体现就是程序员通过它能够更好地控制整个编译过程。在使用 GCC 编译程序时，编译过程可以被细分为四个阶段：

- 预处理（Pre-Processing）
- 编译（Compiling）
- 汇编（Assembling）
- 链接（Linking）

Linux 程序员可以根据自己的需要让 GCC 在编译的任何阶段结束，以便检查或使用编译器在该阶段的输出信息，或者对最后生成的二进制文件进行控制，以便通过加入不同数量和种类的调试代码来为今后的调试做好准备。和其它常用的编译器一样，GCC 也提供了灵活而强大的代码优化功能，利用它可以生成执行效率更高的代码。

GCC 提供了 30 多条警告信息和三个警告级别，使用它们有助于增强程序的稳定性和可移植性。此外，GCC 还对标准的 C 和 C++语言进行了大量的扩展，提高程序的执行效率，有助于编译器进行代码优化，能够减轻编程的工作量。

导航：

- 代码优化
- 调试

GCC 起步

在学习使用 GCC 之前，下面的这个例子能够帮助用户迅速理解 GCC 的工作原理，并将其立即运用到实际的项目开发中去。首先用熟悉的编辑器输入清单 1 所示的代码：

view plaincopy to clipboardprint?

```
1.
2.    清单 1：hello.c
3.    #include <stdio.h>
4.    int main(void)
5.    {
6.    printf ("Hello world, Linux programming!¥n");
7.    return 0;
8.    }
```

然后执行下面的命令编译和运行这段程序：

view plaincopy to clipboardprint?

```
1.
2.    # gcc hello.c -o hello
3.    # ./hello
4.    Hello world, Linux programming!
```

从程序员的角度看，只需简单地执行一条 GCC 命令就可以了，但从编译器的角度来看，却需要完成一系列非常繁杂的工作。首先，GCC 需要调用预处理程序 cpp，由它负责展开在源文件中定义的宏，并向其中插入“#include”语句所包含的内容；接着，GCC 会调用 ccl 和 as 将处理后的源代码编译成目标代码；最后，GCC 会调用链接程序 ld，把生成的目标代码链接成一个可执行程序。整个编译过程

如图 1 所示：

为了更好地理解 GCC 的工作过程，可以把上述编译过程分成几个步骤单独进行，并观察每步的运行结果。第一步是进行预编译，使用-E 参数可以让 GCC 在预处理结束后停止编译过程：

```
# gcc -E hello.c -o hello.i
```

此时若查看 hello.cpp 文件中的内容，会发现 stdio.h 的内容确实都插到文件里去了，而其它应当被预处理的宏定义也都做了相应的处理。下一步是将 hello.i 编译为目标代码，这可以通过使用-c 参数来完成：

```
# gcc -c hello.i -o hello.o
```

GCC 默认将.i 文件看成是预处理后的 C 语言源代码，因此上述命令将自动跳过预处理步骤而开始执行编译过程，也可以使用-x 参数让 GCC 从指定的步骤开始编译。最后一步是将生成的目标文件链接成可执行文件：

```
# gcc hello.o -o hello
```

在采用模块化的设计思想进行软件开发时，通常整个程序是由多个源文件组成的，相应地也就形成了多个编译单元，使用 GCC 能够很好地管理这些编译单元。假设有一个由 foo1.c 和 foo2.c 两个源文件组成的程序，为了对它们进行编译，并最终生成可执行程序 foo，可以使用下面这条命令：

```
# gcc foo1.c foo2.c -o foo
```

如果同时处理的文件不止一个，GCC 仍然会按照预处理、编译和链接的过程依次进行。如果深究起来，上面这条命令大致相当于依次执行如下三条命令：

```
# gcc -c foo1.c -o foo1.o
# gcc -c foo2.c -o foo2.o
# gcc foo1.o foo2.o -o foo
```

在编译一个包含许多源文件的工程时，若只用一条 GCC 命令来完成编译是非常浪费时间的。假设项目中有 100 个源文件需要编译，并且每个源文件中都包含 10000 行代码，如果像上面那样仅用一条 GCC 命令来完成编译工作，那么 GCC 需要将每个源文件都重新编译一遍，然后再全部连接起来。很显然，这样浪费的时间相当多，尤其是当用户只是修改了其中某一个文件的时候，完全没有必要将每个文件

都重新编译一遍，因为很多已经生成的目标文件是不会改变的。要解决这个问题，关键是要灵活运用 GCC，同时还要借助像 Make 这样的工具。

警告提示功能

GCC 包含完整的出错检查和警告提示功能，它们可以帮助 Linux 程序员写出更加专业和优美的代码。

先来读读清单 2 所示的程序，这段代码写得很糟糕，仔细检查一下不难挑出很多毛病：

◆main 函数的返回值被声明为 void，但实际上应该是 int；

◆使用了 GNU 语法扩展，即使用 long long 来声明 64 位整数，不符合 ANSI/ISO C 语言标准；

◆main 函数在终止前没有调用 return 语句。

view plaincopy to clipboardprint?

```
1.
2.    清单 2：illcode.c
3.    #include <stdio.h>
4.    void main(void)
5.    {
6.    long long int var = 1;
7.    printf("It is not standard C code!¥n");
8.    }
```

下面来看看 GCC 是如何帮助程序员来发现这些错误的。当 GCC 在编译不符合 ANSI/ISO C 语言标准的源代码时，如果加上了-pedantic 选项，那么使用了扩展语法的地方将产生相应的警告信息：

```
# gcc -pedantic illcode.c -o illcode
illcode.c: In function `main' :
illcode.c:9: ISO C89 does not support `long long'
illcode.c:8: return type of `main'  is not `int'
```

需要注意的是，-pedantic 编译选项并不能保证被编译程序与 ANSI/ISO C 标准的完全兼容，它仅仅只能用来帮助 Linux 程序员离这个目标越来越近。或者换句话说，-pedantic 选项能够帮助程序员发现一些不符合 ANSI/ISO C 标准的代码，但不是全部，事实上只有 ANSI/ISO C 语言标准中要求进行编译器诊断的那些情况，才有可能被 GCC 发现并提出警告。除了-pedantic 之外，GCC 还有一些其它编译选项也能够产生有用的警告信息。这些选项大多以-W 开头，其中最有价值的当数-Wall 了，使用它能够使 GCC 产生尽可能多的警告信息：

view plaincopy to clipboardprint?

```
1.
2.    # gcc -Wall illcode.c -o illcode
3.    illcode.c:8: warning: return type of `main' is not `int'
4.    illcode.c: In function `main':
5.    illcode.c:9: warning: unused variable `var'
```

GCC 给出的警告信息虽然从严格意义上说不能算作是错误，但却很可能成为错误的栖身之所。一个优秀的 Linux 程序员应该尽量避免产生警告信息，使自己的代码始终保持简洁、优美和健壮的特性。在处理警告方面，另一个常用的编译选项是-Werror，它要求 GCC 将所有的警告当成错误进行处理，这在使用自动编译工具（如 Make 等）时非常有用。如果编译时带上-Werror 选项，那么 GCC 会在所有产生警告的地方停止编译，迫使程序员对自己的代码进行修改。只有当相应的警告信息消除时，才可能将编译过程继续朝前推进。执行情况如下：

```
# gcc -Wall -Werror illcode.c -o illcode
cc1: warnings being treated as errors
illcode.c:8: warning: return type of `main’  is not `int’
illcode.c: In function `main’ :
illcode.c:9: warning: unused variable `var’
```

对 Linux 程序员来讲，GCC 给出的警告信息是很有价值的，它们不仅可以帮助程序员写出更加健壮的程序，而且还是跟踪和调试程序的有力工具。建议在用 GCC

编译源代码时始终带上-Wall 选项，并把它逐渐培养成为一种习惯，这对找出常见的隐式编程错误很有帮助。

库依赖

在 Linux 下开发软件时，完全不使用第三方函数库的情况是比较少见的，通常来讲都需要借助一个或多个函数库的支持才能够完成相应的功能。从程序员的角度看，函数库实际上就是一些头文件（.h）和库文件（.so 或者.a）的集合。虽然 Linux 下的大多数函数都默认将头文件放到/usr/include/目录下，而库文件则放到/usr/lib/目录下，但并不是所有的情况都是这样。正因如此，GCC 在编译时必须有自己的办法来查找所需要的头文件和库文件。

GCC 采用搜索目录的办法来查找所需要的文件，-I 选项可以向 GCC 的头文件搜索路径中添加新的目录。例如，如果在/home/xiaowp/include/目录下有编译时所需要的头文件，为了让 GCC 能够顺利地找到它们，就可以使用-I 选项：

```
# gcc foo.c -I /home/xiaowp/include -o foo
```

同样，如果使用了不在标准位置的库文件，那么可以通过-L 选项向 GCC 的库文件搜索路径中添加新的目录。例如，如果在/home/xiaowp/lib/目录下有链接时所需要的库文件 libfoo.so，为了让 GCC 能够顺利地找到它，可以使用下面的命令：

```
# gcc foo.c -L /home/xiaowp/lib -lfoo -o foo
```

值得好好解释一下的是-l 选项，它指示 GCC 去连接库文件 libfoo.so。Linux 下的库文件在命名时有一个约定，那就是应该以 lib 三个字母开头，由于所有的库文件都遵循了同样的规范，因此在用-l 选项指定链接的库文件名时可以省去 lib 三个字母，也就是说 GCC 在对-lfoo 进行处理时，会自动去链接名为 libfoo.so 的文件。

Linux 下的库文件分为两大类分别是动态链接库（通常以.so 结尾）和静态链接库（通常以.a 结尾），两者的差别仅在程序执行时所需的代码是在运行时动态加载的，还是在编译时静态加载的。默认情况下，GCC 在链接时优先使用动态链接库，只有当动态链接库不存在时才考虑使用静态链接库，如果需要的话可以在

编译时加上-static 选项，强制使用静态链接库。例如，如果/home/xiaowp/lib/目录下有链接时所需要的库文件 libfoo.so 和 libfoo.a，为了让 GCC 在链接时只用到静态链接库，可以使用下面的命令：

```
# gcc foo.c -L /home/xiaowp/lib -static -lfoo -o foo
```

代码优化

代码优化指的是编译器通过分析源代码，找出其中尚未达到最优的部分，然后对其重新进行组合，目的是改善程序的执行性能。GCC 提供的代码优化功能非常强大，它通过编译选项-On 来控制优化代码的生成，其中 n 是一个代表优化级别的整数。对于不同版本的 GCC 来讲，n 的取值范围及其对应的优化效果可能并不完全相同，比较典型的范围是从 0 变化到 2 或 3。

编译时使用选项-O 可以告诉 GCC 同时减小代码的长度和执行时间，其效果等价于-O1。在这一级别上能够进行的优化类型虽然取决于目标处理器，但一般都会包括线程跳转（Thread Jump）和延迟退栈（Deferred Stack Pops）两种优化。选项-O2 告诉 GCC 除了完成所有-O1 级别的优化之外，同时还要进行一些额外的调整工作，如处理器指令调度等。选项-O3 则除了完成所有-O2 级别的优化之外，还包括循环展开和其它一些与处理器特性相关的优化工作。通常来说，数字越大优化的等级越高，同时也就意味着程序的运行速度越快。许多 Linux 程序员都喜欢使用-O2 选项，因为它在优化长度、编译时间和代码大小之间，取得了一个比较理想的平衡点。

下面通过具体实例来感受一下 GCC 的代码优化功能，所用程序如清单 3 所示。

view plaincopy to clipboardprint?

```
1.
2.    清单 3：optimize.c
3.    #include <stdio.h>
4.    int main(void)
5.    {
6.    double counter;
```

```
7.  double result;
8.  double temp;
9.  for (counter = 0;
10. counter < 2000.0 * 2000.0 * 2000.0 / 20.0 + 2020;
11. counter += (5 - 1) / 4) {
12. temp = counter / 1979;
13. result = counter;
14. }
15. printf("Result is %lf¥n", result);
16. return 0;
17. }
```

首先不加任何优化选项进行编译：

```
# gcc -Wall optimize.c -o optimize
```

借助 Linux 提供的 time 命令，可以大致统计出该程序在运行时所需要的时间：

```
# time ./optimize
Result is 400002019.000000
real 0m14.942s
user 0m14.940s
sys 0m0.000s
```

接下去使用优化选项来对代码进行优化处理：

```
# gcc -Wall -O optimize.c -o optimize
```

在同样的条件下再次测试一下运行时间：

```
# time ./optimize
Result is 400002019.000000
real 0m3.256s
user 0m3.240s
sys 0m0.000s
```

对比两次执行的输出结果不难看出，程序的性能的确得到了很大幅度的改善，由

原来的 14 秒缩短到了 3 秒。这个例子是专门针对 GCC 的优化功能而设计的，因此优化前后程序的执行速度发生了很大的改变。尽管 GCC 的代码优化功能非常强大，但作为一名优秀的 Linux 程序员，首先还是要力求能够手工编写出高质量的代码。如果编写的代码简短，并且逻辑性强，编译器就不会做更多的工作，甚至根本用不着优化。

优化虽然能够给程序带来更好的执行性能，但在如下一些场合中应该避免优化代码：

◆ 程序开发的时候优化等级越高，消耗在编译上的时间就越长，因此在开发的时候最好不要使用优化选项，只有到软件发行或开发结束的时候，才考虑对最终生成的代码进行优化。

◆ 资源受限的时候一些优化选项会增加可执行代码的体积，如果程序在运行时能够申请到的内存资源非常紧张（如一些实时嵌入式设备），那就不要对代码进行优化，因为由这带来的负面影响可能会产生非常严重的后果。

◆ 跟踪调试的时候在对代码进行优化的时候，某些代码可能会被删除或改写，或者为了取得更佳的性能而进行重组，从而使跟踪和调试变得异常困难。

调试

一个功能强大的调试器不仅为程序员提供了跟踪程序执行的手段，而且还可以帮助程序员找到解决问题的方法。对于 Linux 程序员来讲，GDB（GNU Debugger）通过与 GCC 的配合使用，为基于 Linux 的软件开发提供了一个完善的调试环境。

默认情况下，GCC 在编译时不会将调试符号插入到生成的二进制代码中，因为这样会增加可执行文件的大小。如果需要在编译时生成调试符号信息，可以使用 GCC 的-g 或者-ggdb 选项。GCC 在产生调试符号时，同样采用了分级的思路，开发人员可以通过在-g 选项后附加数字 1、2 或 3 来指定在代码中加入调试信息的多少。默认的级别是 2（-g2），此时产生的调试信息包括扩展的符号表、行号、局部或外部变量信息。级别 3（-g3）包含级别 2 中的所有调试信息，以及源代码中定义的宏。级别 1（-g1）不包含局部变量和与行号有关的调试信息，因此只能够用于回溯跟踪和堆栈转储之用。回溯跟踪指的是监视程序在运行过程中的

函数调用历史，堆栈转储则是一种以原始的十六进制格式保存程序执行环境的方法，两者都是经常用到的调试手段。

GCC 产生的调试符号具有普遍的适应性，可以被许多调试器加以利用，但如果使用的是 GDB，那么还可以通过-ggdb 选项在生成的二进制代码中包含 GDB 专用的调试信息。这种做法的优点是可以方便 GDB 的调试工作，但缺点是可能导致其它调试器（如 DBX）无法进行正常的调试。选项-ggdb 能够接受的调试级别和-g 是完全一样的，它们对输出的调试符号有着相同的影响。

需要注意的是，使用任何一个调试选项都会使最终生成的二进制文件的大小急剧增加，同时增加程序在执行时的开销，因此调试选项通常仅在软件的开发和调试阶段使用。调试选项对生成代码大小的影响从下面的对比过程中可以看出来：

```
# gcc optimize.c -o optimize
# ls optimize -l
-rwxrwxr-x 1 xiaowp xiaowp 11649 Nov 20 08:53 optimize (未加调试选项)
# gcc -g optimize.c -o optimize
# ls optimize -l
-rwxrwxr-x 1 xiaowp xiaowp 15889 Nov 20 08:54 optimize (加入调试选项)
```

虽然调试选项会增加文件的大小，但事实上 Linux 中的许多软件在测试版本甚至最终发行版本中仍然使用了调试选项来进行编译，这样做的目的是鼓励用户在发现问题时自己动手解决，是 Linux 的一个显著特色。

下面还是通过一个具体的实例说明如何利用调试符号来分析错误，所用程序见清单 4 所示。

view plaincopy to clipboardprint?

```
1.
2.    清单 4：crash.c
3.    #include <stdio.h>
4.    int main(void)
5.    {
6.    int input =0;
7.    printf("Input an integer:");
```

```
8.    scanf("%d", input);
9.    printf("The integer you input is %d¥n", input);
10.   return 0;
11.   }
```

编译并运行上述代码，会产生一个严重的段错误（Segmentation fault）如下：

```
# gcc -g crash.c -o crash
# ./crash

Input an integer:10
Segmentation fault
```

为了更快速地发现错误所在，可以使用 GDB 进行跟踪调试，方法如下：

```
# gdb crash
GNU gdb Red Hat Linux (5.3post-0.20021129.18rh)
……
(gdb)
```

当 GDB 提示符出现的时候，表明 GDB 已经做好准备进行调试了，现在可以通过 run 命令让程序开始

在 GDB 的监控下运行：

```
(gdb) run
Starting program: /home/xiaowp/thesis/gcc/code/crash
Input an integer:10
Program received signal SIGSEGV, Segmentation fault.
0×4008576b in _IO_vfscanf_internal () from /lib/libc.so.6
```

仔细分析一下 GDB 给出的输出结果不难看出，程序是由于段错误而导致异常中止的，说明内存操作

出了问题，具体发生问题的地方是在调用_IO_vfscanf_internal ( )的时候。为了得到更加有价值的信息，可

以使用 GDB 提供的回溯跟踪命令 backtrace，执行结果如下：

```
(gdb) backtrace
#0 0×4008576b in _IO_vfscanf_internal () from /lib/libc.so.6
#1 0xbffff0c0 in ?? ()
#2 0×4008e0ba in scanf () from /lib/libc.so.6
#3 0×08048393 in main () at crash.c:11
#4 0×40042917 in __libc_start_main () from /lib/libc.so.6
```

跳过输出结果中的前面三行，从输出结果的第四行中不难看出，GDB 已经将错误定位到 crash.c 中的第 11 行了。现在仔细检查一下：

```
(gdb) frame 3
#3 0×08048393 in main () at crash.c:11
11 scanf(”%d”, input);
```

使用 GDB 提供的 frame 命令可以定位到发生错误的代码段，该命令后面跟着的数值可以在 backtrace 命令输出结果中的行首找到。现在已经发现错误所在了，应该将

```
scanf(”%d”, input);
```

改为

```
scanf(”%d”, &input);
```

完成后就可以退出 GDB 了，命令如下：

```
(gdb) quit
```

GDB 的功能远远不止如此，它还可以单步跟踪程序、检查内存变量和设置断点等。

调试时可能会需要用到编译器产生的中间结果，这时可以使用-save-temps 选项，让 GCC 将预处理代码、汇编代码和目标代码都作为文件保存起来。如果想检查生成的代码是否能够通过手工调整的办法来提高执行性能，在编译过程中生成的中间文件将会很有帮助，具体情况如下：

```
# gcc -save-temps foo.c -o foo
# ls foo*
foo foo.c foo.i foo.s
```

GCC 支持的其它调试选项还包括-p 和-pg，它们会将剖析（Profiling）信息加入

到最终生成的二进制代码中。剖析信息对于找出程序的性能瓶颈很有帮助，是协助 Linux 程序员开发出高性能程序的有力工具。
在编译时加入-p 选项会在生成的代码中加入通用剖析工具（Prof）能够识别的统计信息，而-pg 选项则生成只有 GNU 剖析工具（Gprof）才能识别的统计信息。

最后提醒一点，虽然 GCC 允许在优化的同时加入调试符号信息，但优化后的代码对于调试本身而言将是一个很大的挑战。代码在经过优化之后，在源程序中声明和使用的变量很可能不再使用，控制流也可能会突然跳转到意外的地方，循环语句有可能因为循环展开而变得到处都有，所有这些对调试来讲都将是一场噩梦。建议在调试的时候最好不使用任何优化选项，只有当程序在最终发行的时候才考虑对其进行优化。

加速

在将源代码变成可执行文件的过程中，需要经过许多中间步骤，包含预处理、编译、汇编和连接。这些过程实际上是由不同的程序负责完成的。大多数情况下 GCC 可以为 Linux 程序员完成所有的后台工作，自动调用相应程序进行处理。
这样做有一个很明显的缺点，就是 GCC 在处理每一个源文件时，最终都需要生成好几个临时文件才能完成相应的工作，从而无形中导致处理速度变慢。例如，GCC 在处理一个源文件时，可能需要一个临时文件来保存预处理的输出、一个临时文件来保存编译器的输出、一个临时文件来保存汇编器的输出，而读写这些临时文件显然需要耗费一定的时间。当软件项目变得非常庞大的时候，花费在这上面的代价可能会变得很沉重。解决的办法是，使用 Linux 提供的一种更加高效的通信方式—管道。它可以用来同时连接两个程序，其中一个程序的输出将被直接作为另一个程序的输入，这样就可以避免使用临时文件，但编译时却需要消耗更多的内存。
在编译过程中使用管道是由 GCC 的-pipe 选项决定的。下面的这条命令就是借助 GCC 的管道功能来提高编译速度的：

```
# gcc -pipe foo.c -o foo
```

在编译小型工程时使用管道，编译时间上的差异可能还不是很明显，但在源代码非常多的大型工程中，差异将变得非常明显。

文件扩展名

在使用 GCC 的过程中，用户对一些常用的扩展名一定要熟悉，并知道其含义。为了方便大家学习使用 GCC，在此将这些扩展名罗列如下：

.c C 原始程序；

.C C++原始程序；

.cc C++原始程序；

.cxx C++原始程序；

.m Objective-C 原始程序；

.i 已经过预处理的 C 原始程序；

.ii 已经过预处理之 C++原始程序；

.s 组合语言原始程序；

.S 组合语言原始程序；

.h 预处理文件(标头文件)；

.o 目标文件；

.a 存档文件。

GCC 常用选项

GCC 作为 Linux 下 C/C++重要的编译环境，功能强大，编译选项繁多。为了方便大家日后编译方便，在此将常用的选项及说明罗列出来如下：

-c 通知 GCC 取消链接步骤，即编译源码并在最后生成目标文件；

-Dmacro 定义指定的宏，使它能够通过源码中的#ifdef 进行检验；

-E 不经过编译预处理程序的输出而输送至标准输出；

-g3 获得有关调试程序的详细信息，它不能与-o 选项联合使用；

-Idirectory 在包含文件搜索路径的起点处添加指定目录；

-llibrary 提示链接程序在创建最终可执行文件时包含指定的库；

-O、-O2、-O3 将优化状态打开，该选项不能与-g 选项联合使用；

-S 要求编译程序生成来自源代码的汇编程序输出；

-v 启动所有警报；

-Wall 在发生警报时取消编译操作，即将警报看作是错误；

-Werror 在发生警报时取消编译操作，即把报警当作是错误；

-w 禁止所有的报警。

小结

GCC 是在 Linux 下开发程序时必须掌握的工具之一。本文对 GCC 做了一个简要的介绍，主要讲述了如何使用 GCC 编译程序、产生警告信息、调试程序和加快 GCC 的编译速度。对所有希望早日跨入 Linux 开发者行列的人来说，GCC 就是成为一名优秀的 Linux 程序员的起跑线。