Linux/UNIX 高级编程

中科红旗软件技术有限公司　编著

清 华 大 学 出 版 社

（京）新登字 158 号

内 容 简 介

本书全面系统地介绍了 Linux/UNIX 高级编程的方法。其中第一章介绍了 Linux 的软件开发工具；第二章至第五章讲述了 Linux 的文件系统，I/O 系统调用、系统数据文件等；第六至第九章讲述了 Linux 进程模型等方面的内容；第十章讲述终端 I/O 编程；第十一章讲述网络套接字编程；第十二章讲述多线程、系统服务等方面的内容。第十三章讲述多线程编程等内容。

本书结合大量实例，概念清晰，特色鲜明，实用性强。

本书适合于 Linux/UNIX 高级编程人员。

书　　名：Linux/UNIX 高级编程
作　　者：中科红旗软件技术有限公司
出 版 者：清华大学出版社（北京清华大学学研大厦，邮编 100084）
　　http://www.tup.tsinghua.edu.cn
责任编辑：柴文强
印 刷 者：清华大学印刷厂
发 行 者：新华书店总店北京发行所
开　　本：787×960 1/16　印张：31.5　字数：703 千字
版　　次：2001 年 7 月第 1 版　2001 年 7 月第 1 次印刷
书　　号：ISBN 7-302-04605-0/TP・2728
印　　数：0001~5000
定　　价：60.00 元

前　言

Linux 是运行在个人计算机和工作站上的 UNIX 操作系统，它继承了 UNIX 的全部优点，是真正的多用户、多任务、多平台的操作系统。在个人计算机和工作站上使用 Linux，能充分发挥硬件的功能，使个人计算机能够作为工作站和服务器使用，提高工作站效率；Linux 符合 POSIX 标准，能同时支持 i386、Alpha、SPARC 和 PowerPC 等处理器；对于在 UNIX 上运行的软件，不用改动或稍加改动就可以运行在 Linux 操作系统上。

同 UNIX 相比，Linux 要灵活得多。Linux 的用户界面非常友好，用户使用很方便。Linux 采用市集开发方式，受自由软件委员会指定的 GNU（General Public License）公用许可证保护，Linux 系统及其应用软件的源代码都是向用户公开的，允许用户使用 Linux 自带的 C/C++语言编译器修改并编译自己喜欢的程序，同时也可以在 GNU 的许可下使用这些源程序的部分或全部代码；这种利用 Linux 源程序的可能性大大激发了世界范围内热衷于计算机事业用户的兴趣和创造力，吸引了众多程序员为 Linux 工作；目前，所有 UNIX 的主要功能都已经有相应的 Linux 工具和程序，大量的自由软件被移植到 Linux 操作系统上，世界各地流行着越来越多的 Linux 发行版本，如 Red Hat Linux，Red Flag Linux 和 Turbo Linux 等；在这些发行版本中，不仅包含 Linux 操作系统的内核，还包含各种编程语言的编译程序、数据库管理系统、图形用户界面、通信联网工具以及大量的其他应用软件。可见，Linux 已经是一个相当优秀的应用软件开发平台。

同 Windows 相比，Linux 也一样具有优势。Linux 强大的网络功能使 Internet 以及许许多多局域网中的服务器都被 Linux 占据，这不仅是由于 Linux 的稳定性相当出色，也是由于 Linux 的系统效率极高；与同类型的 Windows NT 服务器相比，Linux 对硬件的要求可以降低一到两个档次，一台普通的 PC 机就能胜任复杂的网络服务工作；诸如基于文本的 BBS 这样的电子公告牌的讨论性网络服务是 Windows NT 无法实现的，因为迄今为止，只有基于 UNIX 系统的使用文本方式的 BBS 系统；另外，Linux 下的 TCP/IP 网络方面的应用软件更是应有尽有，能实现许多在普通系统中无法实现的网络功能。

不可否认，Linux 具有较 UNIX 和 Windows 更卓越的性能；但是，价格却低得多。因此，如果您想提高自己的计算机使用和编程水平，学习 Linux 确实是一条捷径。

为促进 Linux 在我国的推广应用，使您能更快地步入 Linux 的奇妙世界，少走一些弯路。针对目前市场上缺乏全面、翔实的 Linux / Unix 编程书籍，红旗 Linux 教育培训部拟组织经验丰富的 Linux 编程人员，参考国际经典 Linux、Unix 编程书籍，编写了这本《Linux / Unix 高级编程》。选择在自由软件日趋成熟之际发行，旨在充分发挥我国软件工作人员的积极性和创造力，振兴我国的软件事业。本书内容涵盖了 Linux / Unix 编程的主要方面，

具有一定的深度，具体内容如下：

- 第一章，为您详细介绍 Linux 的软件开发工具，包括编译、调试、维护工具以及广泛应用的集成开发环境。
- 第二、三、四、五章，讲述 Linux 的文件系统、I/O 系统调用、系统数据文件方面的内容。本部分不仅详细讲述了传统 Unix 的文件、I/O 系统调用，还专门介绍了 Linux 文件、I/O 系统的特点。
- 第六、七、八、九章，讲述 Linux 进程模型，包括进程关系、进程控制、进程间通信、信号处理等方面的内容。
- 第十章，讲述终端 I/O 编程，它的用途主要包括：终端、计算机之间的连接、调制解调器、打印机等方面，因此非常复杂。
- 第十一章，讲述网络套接字编程，socket 是最广泛使用和最成熟的网络编程接口规范，用来实现网络的互联。
- 第十二章，守护进程、系统服务等方面的内容，可以编写自己的守护进程来增加、增强系统服务。
- 第十三章，讲述多线程编程，在操作系统层面上讲述了线程的层次和模型，还包括线程同步、互斥等方面的内容。

本书结合大量实例，概念清晰，内容丰富翔实，特色鲜明，实用性强。本书不仅仅是一本手册，很多内容都在操作系统、核心中得到相互印证，真诚期待您的会心一笑。

本书由中科红旗软件技术有限公司编著，耿增强、宋立秋、张荣华、滕靖、朱昊、张焕强、时培植、黄小婉参与了写作工作，在此表示深深的感谢。由于本书涉及的内容丰富，加之篇幅、时间有限，书中难免有纰漏和不足，失误之处请务必谅解和指正。

中科红旗软件技术有限公司

2001 年 4 月

第一章　Linux 软件开发工具

1.1　gcc 和 g++

gcc 和 g++ 分别是 gnu 的 C 及 C++ 编译器，是 Linux 系统中将 C 及 C++ 语言源文件生成可执行程序的工具。

一个可执行的 C 或 C++ 程序需要经过如下四步生成：

第一步：由预处理程序对 C 或 C++ 语言源文件（*.c 或 *.cpp、*.C、*.cxx 等）进行宏扩展和条件处理，并导入前导文件，生成以 .i 为后缀的文件；

第二步：由编译程序将预处理后的文件（*.i）中的 C 或 C++ 语言源代码转换成汇编语言代码，生成以 .s 为后缀的汇编文件；

第三步：由汇编程序将汇编文件（*.s）中的汇编语言代码转换成目标代码（机器代码，*.o），生成以 .o 为后缀的目标文件；

第四步：由连接程序将目标文件（*.o）同指定的库文件进行连接，生成可执行程序。

若非特别指定，gcc 或 g++ 将自动调用上述工具，生成可执行程序。

gcc 和 g++命令的调用格式为：

```
gcc [options] filenames
g++ [options] filenames
```

其中：options 为 gcc 和 g++ 命令的选项，可以为空；多个选项必须分开写，如“-dr”不同于“-d -r”。filenames 是 gcc 和 g++ 命令的输入文件列表，多个文件之间以空格分隔。

gcc 和 g++ 命令的选项很多，它们规定着 gcc 和 g++ 命令的行为；掌握 gcc 和 g++ 命令的关键就是正确使用它们的选项。

gcc 和 g++ 命令的选项大致可分为如下几类：

- 全局选项，影响编译的全过程（从预处理到连接）；
- 语言选项；
- 预处理程序选项；
- 汇编程序选项；
- 连接程序选项；
- 目录选项；
- 优化选项；
- 调试选项；

- 警告选项。

下面分别介绍各类的常用选项（除非特殊声明，选项将同时适用于 gcc 和 g++ 命令）。

1.1.1 全局选项

1. -x language filenames

此选项指明 filenames 文件的编程语言（C 语言、C++语言等）及 gcc / g++ 编译的起始阶段（预处理、编译、汇编、连接）。

language 的可选值为 C、C++、assembler-with-cpp 等；其中：C、C++ 指明输入文件为 C 语言、C++ 语言文件，编译过程从预处理阶段开始；assembler、assembler-with-cpp 指明编译过程从汇编阶段开始。

-x language 选项对其后的多个文件都有效，直至遇到下一个-x 为止。

在不使用此选项时，gcc/g++ 利用文件名后缀来确定语言及编译的起始阶段。如：输入文件名以 .c 或 .cpp 为后缀时，从预处理阶段开始编译；以 .i 为后缀时，从编译阶段开始编译；以 .s 为后缀时，从汇编阶段开始编译；以 .o 为后缀时，从连接阶段开始编译。使用 -x 选项后，文件名后缀将不再对编译的起始阶段起作用。

2. -x none filenames

此选项使 -x language 选项失去作用，其后文件以名字后缀决定语言及编译的起始阶段。

3. -c、-S、-E

这三个选项决定编译过程到哪一个阶段（预处理、编译、汇编、连接）结束，它们不能同时使用。

-c 选项激活预处理、编译及汇编程序，不执行连接程序；编译过程到汇编阶段结束；输出文件为目标文件（*.o），可用 -o 选项（见下文）另行指定输出文件名。

-S 选项激活预处理及编译程序，不执行汇编和连接程序；编译过程到编译阶段结束；输出文件为汇编文件（*.s），可用 -o 选项（见下文）另行指定输出文件名。

-E 选项激活预处理程序，不执行编译、汇编和连接程序；编译过程到预处理阶段结束；输出被送到标准输出，可用 -o 选项重新定向到文件输出。

4. -o filename

此选项以 filename 覆盖 gcc/g++ 默认的输出文件名；它可以同-c、-S、-E 中的任何一个选项一同使用，为各阶段的输出文件（可执行文件、目标文件、汇编文件、预处理后的 C/C++ 语言源文件）改名。

因为 -o 选项只能指定一个输出文件名，所以当有多个输出文件产生时，-o 选项不起作用。

如不使用此选项，默认情况下，可执行程序以 a.out 命名，目标文件以 *.o 命名，汇编文件以 *.s 命名，预处理后的 C/C++ 语言源文件定向到标准输出。

5. -pipe

此选项用管道代替临时文件来处理不同编译阶段（预处理、编译、汇编、连接）间的通信。这个选项在某些不支持管道操作的系统上不工作，但在 GNU 编译器上不会有问题。

1.1.2 语言选项

1. -ansi

此选项关闭 GNU C 中与 ANSI C 不兼容的特性，激活 ANSI C 的专有特性。当使用此选项时，预编译器定义了一个宏 _STRICT_ANSI_ ，您可在源程序中加入对此宏的判断来避免使用 ANSI 标准不兼容的特性。

比如：在 ANSI C 中，禁止使用 asm、inline 和 typeof 关键字及 UNIX、vax 等预定义宏，取而代之的是 _asm_ 、_inline_ 、_typeof_ 、_UNIX_ 和 _vax_ ；这些被释放出来的关键字和宏可由您赋予新的含义。

注意：-ansi 选项并不拒绝非 ansi 程序。

2. -fno-asm

此选项实现 -ansi 选项功能的一部分，它禁止将 asm、inline 和 typeof 用作关键字。使用此选项的目的是程序已经为 asm、inline 和 typeof 赋予新的含义，不希望编译器（gcc、g++）将它们解释为关键字。

3. -fno-strict-prototype

此选项只对 g++ 起作用，对 gcc 没有影响。

通常情况下，g++ 编译器将不带参数的函数声明理解为没有参数；使用此选项后，g++ 认为不带参数的函数声明是没有显式地对参数的个数及类型进行声明，而不是没有参数。

不论是否使用此选项，gcc 编译器总是将不带参数的函数声明理解成没有对参数的个数及类型进行声明。

4. -fthis-is-variable

此选项只对 g++ 有效。

默认情况下，C++ 语言中的类已经将其对象指定给 this 指针，因此不能再将 this 用作一般变量；为了同传统 C++ 语言兼容，此选项允许将 this 用作一般变量。

5. -fcond-mismatch

此选项允许条件表达式的第二和第三参数类型不匹配。这样的表达式的值为 void 类型。

6. -funsigned-char

```
-fno-signed-char
```

```
-fsigned-char
-fno-unsigned-char
```

这四个选项对 char 类型进行设置。char 类型可能为 signed-char，也可能为 unsigned char。每种机器对 char 类型都有默认设置，使用这四个选项可以重新指定 char 类型的设置。其中前两个选项将 char 类型指定为 unsigned char，后两个选项将 char 类型指定为 signed char。

1.1.3 预处理程序选项

此类选项用于设定 gcc/g++ 在预处理阶段的行为。前文曾经讲过，-E 选项使 gcc/g++ 只执行预处理程序；此类中的部分选项必须同-E 选项一起使用，因为它们使预处理程序的输出不适合继续编译。

1. -include file

```
-imacros file
-Dmacro
-Dmacro=defn
-Umacro
```

上述五个选项的作用相当于 C/C++ 语言源文件中的 #include、#define 和 #undef 语句，但是优先于 gcc/g++ 的输入文件被处理。当 gcc/g++ 的命令行中有上述选项出现时，gcc/g++ 不论 -Dmacro、-Umacro 选项的位置如何，总是最先处理它们；然后再按照命令行中的顺序处理 -include 和 -imacros 选项；只有在命令行中的预处理选项都被处理完之后，gcc/g++ 才去处理输入文件中的预处理语句。

在上述五个选项中：-include file 选项的作用相当于 C/C++ 语言源文件中的“#incude”语句，导出 file 文件内容到 gcc/g++ 的输入文件中；-imacros file 选项的作用是将 file 文件中的宏定义扩展到 gcc/g++ 的输入文件中，宏定义本身并不出现在输入文件中；-Dmacro 选项的作用相当于 C/C++ 语言源文件中的“#define macro”语句，定义 macro 宏为串“1”或“TRUE”；-Dmacro=defn 选项的作用相当于 C/C++ 语言源文件中的“#define macro defn”语句，定义 macro 宏为串 defn；-Umacro 选项的作用相当于 C/C++ 语言源文件中的“#undef macro”语句，取消对 macro 宏的预定义。

2. -undef

此选项取消对任何非标准宏的定义。

```
-Idir
-I-
-idirafter dir
-iprefix prefix
-iwithprefix dir
```

这些选项限定 gcc/g++ 搜索头文件（前导文件）的路径。

C/C++ 语言源文件中约定使用 #include <file.h> 语句来包含任何由 C 编译系统提供的标准前导文件，使用 #incldue "file.h" 语句来包含您自己目录中的前导文件。

在不使用上述选项的情况下，gcc/g++ 按照如下顺序搜索前导文件：

对于在 #include <file.h> 中给出的前导文件：gcc/g++ 只到系统的标准位置（通常是/usr/include）中搜索；

对于在 #include "file.h" 中给出的前导文件：gcc/g++ 先在包含 #include "file.h" 语句的 C/C++ 源文件所在的目录中搜索（这个目录通常是您的当前目录），若未找到，再到标准位置（通常为/usr/include）下搜索。

如在命令行中使用 -Idir 选项，gcc/g++ 将按如下顺序搜索头文件：

对于在 #include <file.h> 中给出的头文件：先在 -I 选项所指定的 dir 中查找，再到标准位置中查找；

对于在 #include "file.h" 中给出的头文件：先在含此语句的源文件所在的目录下查找，再到 -I 选项所指定的 dir 中查找，最后到标准位置下查找。

如在命令行中使用 -I- 选项，且有 -Idir 选项出现在 -I- 之前，则 gcc/g++ 按如下顺序搜索头文件：

对于在 #include <file.h> 中给出的头文件：只在标准位置中查找，不到-I 选项所指定的 dir 中查找，除非 -Idir 选项出现在-I-选项之后；

对于在#include "file.h"中给出的头文件：先在含此语句的源文件所在的目录下查找，再到-I 选项所指定的 dir 中查找，最后到标准位置下查找；不受 -I- 选项的影响。

如在命令行中使用 -Idirafter dir 选项，则当 gcc/g++ 在 -I 选项指定的 dir 中搜索失败后，将继续在此选项指定的 dir 中搜索。

-iprefix 选项和 -iwithprefix 选项总是在一起使用，它们规定当 gcc/g++ 在 -I 选项指定的 dir 中搜索失败后，继续在由 -iprefix prefix 和 -iwithprefix dir 共同指定的路径 "prefix+dir" 下搜索。

3. -nostdinc

此选项规定 gcc/g++ 不在系统标准位置（通常为/usr/include）下搜索头文件。

联合使用-nostdinc 选项和-I 选项，您可以明确限定头文件的搜索路径。

4. -nostdin C++

此选项规定不在 g++ 指定的标准路径中搜索，但仍在其他标准路径中搜索。

此选项在创建 libg++ 库时使用。

5. -C

此选项使预处理程序不删除 C/C++ 语言源文件中的注释语句；此选项必须同-E 选项一起使用。

6. -M

-MM

```
-MD
-MMD
```

这四个选项为每一目标文件生成依赖关系，以方便 make（参见本章第二节）使用。

-M 选项使预处理程序为每一 C/C++ 源码文件输出一条规则，规则的目标是源码文件（*.c, *.cpp）的目标文件（*.o），依赖关系是所有包含在 #include 语句中的头文件。每条规则可能只有一行，也可能用续行符分写成多行，规则清单被送到标准输出。此选项包含 -E 选项的功能。

-MM 选项类似于 -M 选项，但依赖关系中只包含在 #include "file.h" 语句中的头文件，而忽略在 #include <file.h> 中的头文件。

-MD 选项类似于 -M 选项，但规则清单不送到标准输出，而是写入 .d 文件中，以方便 md 工具将多个 .d 文件合并成一个文件，提供给 make 使用。

-MMD 选项综合了 -MD 选项和 -MM 选项的功能。

1.1.4 汇编程序选项

-Wa，option

此选项传递 option 给汇编程序；如果 option 中含有逗号，就将 option 分隔成多个选项后传递给汇编程序。

1.1.5 连接程序选项

1. -Wl，option

此选项传递 option 给连接程序；如果 option 中含有逗号，就将 option 分隔成多个选项后传递给连接程序。

2. -dy

```
-Bstatic
-Bdynamic
```

这些选项用于指明库的连接方式——静态连接方式或动态连接方式；您选择的连接方式决定了库函数何时被连接到程序中。

在静态连接方式下，您的程序的外部引用同它们的定义在创建可执行文件的时候就被指定内存地址，连接到程序中。在此方式下，可供连接的库有静态连接库（也称归档库）和静态共享库。静态连接库是一组目标文件，其中每个目标文件都包含了一个函数或一组相关函数；连接时，库中含有您程序中尚未解析的外部引用的目标文件副本将被结合到您的可执行文件中。相比之下，静态共享库所包含的目标代码可以由多个进程在运行时共享；当您的程序连接一个静态共享库时，定义您程序中外部引用的那部分库代码并没有被复制

到程序的目标文件中，而是在目标文件中创建了一个识别库代码的被称作 .lib 的特殊段；直到程序执行时，.lib 段中的信息才将所需的静态共享库代码引入到进程的地址空间中；因此，对于所有使用静态共享库的程序来说，静态共享库代码只有一个目标副本在内存中存在。

在动态连接方式下，您的程序的外部引用同它们的定义是在程序运行时才被连接的。在此方式下，可供连接的库为动态连接库，也称共享目标库。动态连接库是一个独立的目标文件，其中包含了库中每一个函数的代码；动态连接库中的内容只有在程序运行时才被映射到进程的虚拟地址空间中；同静态连接相比，动态连接程序节省磁盘空间和运行时被共享库占用的系统进程内存，而且动态连接代码可以方便地修改和升级，无需对依赖于它的应用程序进行重新连接。

按照约定，动态连接库的文件名前缀为 lib，后缀为 .so；静态连接库的文件名前缀为 lib，后缀为 .a；静态共享库的文件名前缀为 lib，后缀为 _s.a。所以，标准 C 库的静态连接版本为 libc.a；静态共享版本为 libc_s.a；动态连接版本为 libc.so。

默认情况下，gcc/g++ 使用指定库的静态连接版本；而在 -dy 选项下，连接程序首先搜索库的动态连接版本，搜索失败后，才使用静态连接版本；静态共享库不受 -dy 选项的影响，它是在指定库文件时通过库文件名后缀（-s）来标识的。

如果您想明确限定某些库的连接方式，可以使用 -Bstatic 和-Bdynamic 选项。-Bstatic 选项关闭动态连接方式，-Bdynamic 选项打开动态连接方式。一旦指定了-Bstatic 选项，连接程序将不再接受动态连接库，直到再次指定-Bdynamic 选项；反之亦然。

3. -llibrary

此选项用于指定要连接的库，参数 library 是库文件名去掉前缀和后缀剩下的部分。如：-lsocket 指明连接 socket 库，其静态连接版本为 libsocket.a，动态连接版本为 libsocket.so；具体连接哪一个版本，由 -dy、-Bstatic 和 -Bdynamic 选项决定；但若要连接其静态共享版本 libsocket_s.a，则应用 -lsocket_s 选项给出，此选项不受-dy、-Bstatic 和 -Bdynamic 选项的影响。

不论是否使用 -l 选项，连接程序都将连接标准 C 库；如果使用 -l 选项，连接程序将在搜索完 -l 选项所指定的库文件之后，再在标准 C 库中寻找尚未解析的外部引用的定义。

注意：由于连接程序在库中仅仅为它所找到的未解析的外部引用搜索定义，所以 -l 选项在 gcc/g++ 命令行出现的位置很重要，通常放在命令行的末尾。

4. -Ldir

此选项用于指定库的搜索路径。

在未使用此选项的情况下，连接程序直接从库的标准路径（通常为/usr/lib/）中搜索指定的库（包括在 -l 选项中指定的库和标准 C 库）。

在使用此选项的情况下，连接程序首先在 -L 指定的路径中搜索，然后再到标准路径下搜索。

另外，您还可以使用环境变量 LD_LIBRARY_PATH 来增加连接程序搜索路径的列表。LD_LIBRARY_PATH 必须是用冒号隔开的路径名称列表（限定使用绝对路径），可选的第二个列表同第一个列表之间用分号隔开，如：

```
LD_LIBRARY_PATH=dir1 : dir2; dir3: dir4 export LD_LIBRARY_PATH
```

在分号之前的第一个列表中指定的目录将在用 -L 指定的路径之前按顺序搜索，分号之后的第二个列表中指定的目录将在用 -L 指定的路径之后按顺序搜索。在使用环境变量 LD_LIBRARY_PATH 指定库的搜索路径时应注意两点：一是所有在此环境下执行的连接程序都将受到此变量的影响；二是当您将某动态连接库移动到另一个目录中时，只需修改 LD_LIBRARY_PATH 变量值，不必重新编译程序（用 -L 选项设定库的搜索路径时，必须在修改 -L 选项的参数后，重新编译程序，才能正确执行）。

5. -dy -G 选项

此选项用于创建动态连接库；您可以在 -G 选项后面用 -o 选项指定要创建的动态连接库的名称；否则，将创建名为 a.out 的动态连接库。

当所创建的库不遵循 lib 前缀和 .so 后缀时，无法用 -L 选项连接，为此可在创建库后用 mv 命令重新命名库。

如：
```
gcc -dy -G file1.o file2.o file3.o
mv a.out libx.a
```

等价于：gcc -dy -G -o libx.a file1.o file2.o file3.o

5. 介绍两个与连接程序密切相关的命令

- ar 命令：

此命令用于创建动态连接库；

如： ar -r libx.a file1.o file2.o file3.o

- ldd 命令：

此命令指示连接程序输出程序所依赖的动态连接库的路径及名称。可以用 -d 选项指示连接程序输出运行程序时遇到的所有未被解析的数据引用的诊断信息；也可以用 -r 选项指示连接程序输出运行程序时遇到的所有未被解析的数据或函数引用诊断信息。

1.1.6 目录选项

此类选项包括 -Idir、 -I- 和 -Ldir 选项等；详细内容在前面的预处理程序选项和连接程序选项中已经讲过，此处不再重复。

1.1.7 优化选项

此类选项用于提高 gcc/g++ 编译器产生的汇编语言代码的效率，缩短目标代码的执行

时间。若不使用这些选项，编译器将尽可能减少编译开销。

此类选项包括：

```
-O0
-O1（等价于-O 选项）
-O2
-O3
```

其中：-O0 选项关闭编译器的优化功能；-O1、-O2、-O3 选项分别指示编译器对程序进行不同层次的优化，从 -O1 到 -O2、再到 -O3，优化程度依次加深，编译时间越来越长，生成代码的质量逐渐提高，程序运行的速度不断加快。

当在同一 gcc/g++ 命令行使用多个 -O 选项时，最后一个优化选项起作用。

1.1.8 调试选项

此类选项用于指示编译器产生与程序变量和语句相关的信息，供调试工具使用。

1. -g

-g 选项指示编译器以操作系统的本地格式（如：stabs、COFF、XCOFF、DUARF 等）产生调试信息，提供给调试程序（如：gdb）使用。在大多数使用 stabs 格式的系统上，-g 选项能够产生额外的调试信息，但这些信息只能由 gdb 使用，其他的调试程序可能会因此发生异常；若想禁止额外信息的产生，可使用 -gstabs 选项。

2. -gstabs

此选项以 stabs 格式生成调试信息，但不包含 gdb 的额外调试信息。

6. -gstabs+

此选项以 stabs 格式生成调试信息，并含有仅供 gdb 使用的额外调试信息。

7. -ggdb

此选项生成调试信息，且尽可能包含 gdb 的额外调试信息。

注意：不同于其他许多 C/C++ 语言编译器，gcc/g++ 允许调试选项与优化选项一起使用。

1.1.9 警告选项

警告作为诊断信息，报告程序中的非致命错误。

警告选项控制由 gcc/g++ 产生警告的数量和种类。

1. -pedantic

此选项禁止某些 GNU 扩充特性和传统 C 语言特性。通常不论有无此选项，ANSI 标准 C 程序都能正确编译，所以一般情况下无需使用该选项。

2. -pedantic-errors

此选项类似于 -pedantic，只是不产生警告信息而产生错误信息。

3. -fsyntax-only

此选项只检查代码的语法错误，不产生任何输出。

4. -Wimplict

此选项当函数或参数被隐式声明时，产生警告信息。

5. -Wunused

此选项当某局部变量未被使用过；或某函数声明被声明为静态函数，却从未被定义过；或者当某表达式的计算结果从未被使用过时，产生警告信息。

6. -Wcomment

此选项当有“/”出现在注释行中时，产生警告信息。

7. -Wformat

此选项当 printf 和 scanf 中的格式出错时，产生警告信息。

以上只分类介绍了 gcc/g++ 编译器的常用选项，另外还有许多选项，请查看 gcc/g++ 的参考手册或相关文档。

1.2 gnu make

同 gcc/g++ 一样，gnu make（简称为 gmake；在 Linux 系统中，gmake 是 make 的符号连接，所以本节不区分 make 与 gmake）也是 Linux 系统上的软件开发人员所必须掌握的工具之一，二者都能将 C/C++ 语言源文件生成可执行程序。比较它们的关系，可以说：gcc/g++ 是 make 的基础，没有 gcc/g++ （不考虑其他编译器），make 就无法将 C/C++ 语言源文件生成可执行程序；而 make 则好比是 gcc/g++ 命令的调度器，它通过读进一个文本文件（通常称之为 Makefile 或 makefile 文件，其内定义有 C/C++ 语言源文件的依赖关系和要执行的命令序列），执行一组以 gcc/g++ 为主的 shell 命令序列，在创建/更新程序的同时，将不必要的编译减到最少。

通过使用 make 命令，您可以从 C/C++ 软件的维护工作中解脱出来；您不必再因为某些文件的改变，而去手工编译/连接依赖于它们的文件。

make 命令可以自动帮您记住以下内容：

- 文件之间的依赖关系；
- 哪些文件是最近修改的；
- 在源文件修改后哪些文件需要重新编译；
- 产生新版本程序所需的精确的操作序列。

make 具有如此突出的优势，以至于如果您的程序不是用 make 命令生成的，那么这个

程序就只能算作是您的个人程序，而不具有丝毫的通用性。

make 命令的基本工作流程如图 1-1 所示。

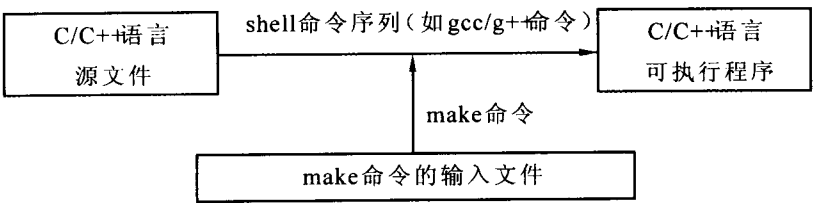

图 1-1 make 命令基本工作流程图

下面先介绍 make 命令的输入文件，再介绍 make 命令的使用方法。

1.2.1 make 命令的输入文件

make 命令的输入文件默认命名为 Makefile 或 makefile。除非您在 make 命令行中特别指定，make 将首先查找 Makefile 文件，找到后就去执行其中的命令序列；找不到再去查找 makefile 文件（为方便说明，以下称 make 命令的输入文件为 Makefile 文件）。

make 命令的输入文件主要包含规则和变量，它们负责记录文件之间的依赖关系和命令的执行顺序。

1.2.1.1 规则

1. 规则构成

Makefile 文件中的每条规则由一个带冒号的“依赖行”和一条或多条以 tab 开头的“命令行”组成。

规则语法为：

```
目标 1 [目标 2...]:   [依赖文件列表]
[\t  命令]
...
```

其中：第一行为依赖行，其下各行为命令行，带方括号的项为可选项。依赖行中，冒号左边是目标，冒号右边是依赖文件；目标和依赖文件均是由字母、数字、句点和斜杠组成的字符串，串中允许包含 shell 元字符（如：*和?），这些字符在对行求值时展开；当目标或依赖文件的数目多于一个时，以空格分隔。命令行是不包含'#'（除非出现在引号之间）的任意字符构成的字符串；多个命令可以用分号分隔后写在同一命令行中，也可分成多个命令行；命令行描述如何由依赖文件生成规则目标，所有的命令行都必须以 tab 字符（命令行中以 '\t' 代表 tab 字符）开头。

【例 1-1】 下面是由某 Makefile 文件中抽取出的一条规则。

```
xyz:  x.c y.c z.c assemble.o object.a
\t  gcc -o xyz x.c y.c z.c assemble.o /usr/local/lib/object.a
```

其中：第一行为依赖行，表明规则目标 xyz 依赖于 C 语言源文件 x.c、y.c、z.c 及目标代码文件 assemble.o 和库文件 object.a。第二行为命令行，以 tab 开头，指定由依赖文件生成规则目标的编译/连接命令。

至此，提到的规则目标均是“真实目标”，它们在命令执行后确实能被创建/更新。与此相对的还有一种目标，称为“假象目标”；此种目标既不出现在规则的命令行中，也不会因为命令行的执行而被创建/更新。

“假象目标”通常有两个用处。一是创建多个互不相关的目标。

【例 1-2】 Makefile 文件中经常会有类似于下面的语句。

```
all: object1 object2
object1:...
\t ...
object2:...
\t  ...
```

此例中：all 为“假象目标”，object1 与 object2 是两个互不相关的“真实目标”。执行 make all 命令后，因为 all 是“假象目标”，没有任何规则生成它，所以 make 会以为 all 需要创建而去执行此条规则；执行这条规则时，make 要先创建/更新规则中的依赖文件 object1 和 object2；于是分别执行以 object1 和 object2 为目标的规则，如此就由一条规则创建/更新了两个互不相关的目标 object1 和 object2。

“假象目标”的第二个用处是描述一组不生成固定目标的动作。

【例 1-3】 某 Makefile 文件中含有如下内容：

```
clean:
\t  rm  *.o
\t  rm  *.a
```

此例中：clean 为“假象目标”，规则执行结果会删除 Makefile 文件所在路径下的所有以.o 和.a 为后缀的文件。

正确使用“假象目标”的前提是“假象目标”不存在。倘若由于某种原因，“假象目标”存在，且是最新（在“假象目标”的第二种用法中，由于没有依赖文件，所以只要“假象目标”存在，就一定是最新的），make 就会拒绝执行此条规则，因而既无法生成多个互不相关的目标，也不能执行命令行规定的动作。为确保此种情况不会发生，解决办法是在 Makefile 文件中用.PHONY 关键字标示出所有的“假象目标”；这样，make 就不会检查“假象目标”是否存在，而总是认为它们需要被创建/更新。

【例 1-4】 .PHONY: clean

指定 clean 为“假象目标”，且总需要被创建/更新。

2. 规则编写

规则编写包括“依赖行”和“命令行”的编写。

首先介绍依赖行的编写。最明显的办法是逐个检查源文件，将它们的目标文件（*.o）作为规则目标，将源文件中包含的头文件列成依赖文件。此种方法的优点是简单易行，但当项目很大时，会因为涉及的文件众多、目录结构复杂，而导致工作量剧增，出错率上升。为克服这个缺点，您可以借助 gcc/g++ 的 -M 选项（具体参见本章第一节）来创建依赖关系；此选项为 gcc/g++ 编译器的每个输入文件生成一条规则，规则的目标是输入文件的目标代码文件（*.o），依赖文件是输入文件及其所包含的全部头文件（包括系统头文件和用户定制的头文件）。因为系统头文件（在 C/C++ 语言源文件中以 #include <file.h> 包含）一般不会被修改，所以您也可以用 -MM 选项代替 -M 选项，使编译器生成的依赖关系中只包含您自己定制的头文件（在 C/C++ 语言源文件中以 #include "file.h" 包含）。

编写完依赖关系后，就该编写命令行了。对此，您可在依赖行的下一行插入要执行的命令（注意：命令行必须以 tab 开头），也可什么都不写，按照隐含规则来创建目标。

3. 隐含规则

隐含规则也称为后缀转换规则，它们决定如何将某种后缀的文件转换成另一种后缀的文件。每条隐含规则都有名称，其名称是将转换前后的后缀连接得到的。

如：将 C 语言源文件（*.c）转换成目标文件（*.o）的隐含规则称为.c.o 规则；

将汇编语言文件（*.s）转换成目标文件（*.o）的隐含规则称为.s.o 规则；

将 C 语言源文件（*.c）生成可执行程序（无后缀）的规则称为.c 规则；此规则属于空后缀规则。

当 make 命令在相应的 Makefile 文件中没有找到生成某目标的命令行时，就会求助于隐含规则。

【例 1-5】 某 Makefile 文件中含有如下规则：

```
x.o: x.c x.h
y.o: ...
...
```

此例中，x.o 规则只含有依赖行，而没有命令行。make 在创建 x.o 目标时，就会根据隐含规则中的 .c.o 规则来创建 x.o。

除了能提供隐含的命令行外，隐含规则还能提供隐含的依赖信息。

【例 1-6】 当 Makefile 文件中含有如下规则时：

```
x.o:  x.h
y.o:  ...
```

为创建 x.o 目标，make 先按一定的顺序查找隐含规则，当找到第一条以 .o 为目标后缀的规则（假定为 .c.o 规则）后，就去查找 Makefile 文件所在路径下的文件，若存在此隐