其次，你应该以 g++ 命令代替 gcc 来编译 C++ 程序，实际上 g++ 只是一个简单的批处理文件，它会在启动 gcc 时帮你加上一些参数（例如指定连接标准 C++ 函数库），所以 g++ 和 gcc 所接受的参数是一样的。

如果你不想使用 g++，那你必须确定你连接了 C++ 的函数库，这样你才能使用基本的 C++ 类，例如 cout 和 cin 这两个 I/O 对象。同时你也必须确定你已经安装好 C++ 的函数库和头文件。有些 Linux 软件包只提供标准的 C 函数库，所以虽然 gcc 可以很正常的编译完 C++ 程序，但是若没有 C++ 函数库，一样无法正确通过连接（link）。

Makefiles

有时候你会在 Linux 上使用到 *make*，即使你并不打算设计任何程序，例如你在更新和重建你的内核时，也会执行 *make*。如果你很幸运，或许不需要为了 makefile 而奋斗，但是我们也要帮助那些不幸的人们，所以在这节中，我们打算详细的介绍 *make* 语法，使你撰写 makefile 时能驾轻就熟。

后面有些范例会引用到 Linux 内核的 makefile，因为里头使用了许多 GNU 版本的 *make* 程序之扩展功能，所以我们除了标准的 *make* 功能之外也要讨论其中的一些扩展功能。有一本介绍 *make* 的好书，名为《Managing Projects with make》，作者是 Andrew Oram 和 Steve Talbott，而 GNU make 的扩展功能在其使用说明中（manual）也有详细的说明。

许多人认为 *make* 只是把源文件转成目标文件和函数库，以及将目标文件连接成执行文件的方式而已。其实 *make* 可以看成是一种通用型工具，可以有计划地把原先的文件名（prerequisite）转成另一种形式的文件名（target）。转换的结果可以是可执行文件、PostScript 文件或其他各类文件名。原先的文件名可能是 C 程序代码、TEX 文本文件之类的文件名。

尽管你可以使用很简单的批处理文件来执行 gcc 编译程序，但是 *make* 会判断原来的文件名有无必要转换，所以 *make* 仅会将更改过的原始程序文件编译成目标文件。

假设你有一个程序是由三个源程序文件所组成的。如果你使用下面这样的命令编译它们：

```
papaya$ gcc -o foo foo.c bar.c gaz.c
```

每当你改变其中任一个文件名时，你都必须要重新编译、连接所有的文件名。倘若你只改了一个文件名，这样做是很浪费时间的（尤其是当这文件名只是整个大程序中的一个小文件名时）。这时你便会想要只编译这个文件名，然后再连接其他文件名以组成可执行文件，而 *make* 正好可以自动做到这些功能。

Make 能做什么

make 基本上能让你以很简单的步骤完成工作。如果有一个可执行文件是由许多的源文件组成，你可以更改其中一个文件名而不需要编译所有文件名。为了达到这种弹性，*make* 会记录所有文件名的关联性。

这里有一个简单的 makefile。我们通常用 *makefile* 或 *Makefile* 作为文件名，并将之放在和源文件相同的目录内。

```
edimh: main.o edit.o
        gcc -o edimh main.c edit.o

main.o: main.c
        gcc -c main.c

edit.o: edit.c
        gcc -c edit.c
```

这个文件名说明了 *edimh* 执行文件是由 *main.c* 及 *edit.c* 所构成的。你不必局限于将 makefile 用于 C 程序设计这方面，里面所下的命令可以是任何命令。

这个 makefile 有三个进入点（entry），每一个进入点都包含一行 *dependency line*（相依性），说明如何建立一个文件名。因此，第一行说明了 *edimh*（就是在冒号前的文件名）是由 *main.o* 及 *edit.o* 组成的（写在冒号之后）。这一行告诉 *make*：如果这些目标文件被更改过，就执行后面 *gcc* 的命令，包含 *gcc* 命令的行必须以定位符号（tab）开头（而不是一连串的空白）。

当执行这个命令时

```
papaya$ make edimh
```

如果 *edimh* 这个文件名不存在，或者是其中一个目标文件被更改过，*make* 会执行 *gcc* 来编译。在这里我们把 *edimh* 叫做 *target*（译注 1），而冒号后的文件名则叫做 *dependent* 或 *prerequiste*。

后面几个进入点也有同样的功能，*main.o* 当 *main.o* 不存在或 *main.c* 更改时会重新编译，*edit.o* 也是在类似情形下重编。

到底 *make* 是如何知道文件名的新旧呢？事实上 *gcc* 是根据每个文件名系统都有的文件名更新日期决定的，你可以利用 *ls -l* 来查看每个文件名的更新日期。文件名的日期字段精确度是一秒，所以 *make* 可以很准确地利用它来判断你是否在编译后改变过源文件名，或者是在建立好执行文件后，曾经编译过目标文件。

我们可以试试这个 makefile，看看它是如何运作的：

```
papaya$ make edimh
gcc -c main.c
gcc -c edit.c
gcc -o edimh main.o edit.o
```

如果我们编辑 *main.c* 然后再重做一次，会发现它只会重新建立需要的文件名，以节省我们的时间：

```
papaya$ make edimh
gcc -c main.c
gcc -o edimh main.o edit.o
```

在 makefile 里的三个进入点的前后顺序是不重要的，*make* 会分析这些文件名之间的关联性，并且以正确的顺序执行这些命令。把 *edimh* 这个进入点摆在一开始较好，因为 *make* 的默认是会建立第一个进入点。也就是说我们可以用 make 来代替 make edimh 这个命令而不会影响结果。

译注 1：由于 target 有目标的意味，为避免和 object file 弄混，所以不译。

让我们来看看一个较大的范例吧，看看你能否了解其中的关联：

```
install: all
        mv edimh /usr/local
        mv readimh /usr/local

all: edimh readimh

readimh: read.o edit.o
        gcc -o readimh main.o read.o

edimh: main.o edit.o
        gcc -o edimh main.o edit.o

main.o: main.c
        gcc -c main.c

edit.o: edit.c
        gcc -c edit.c

read.o: read.c
        gcc -c read.c
```

首先我们看到进入点 install，因为它不会产生任何的文件名，所以我们称之为假的 target（*phony target*），它的存在只是为了执行它下面的命令。但在 install 执行前，all 必须先执行，因为 install 必须依赖 all（记住！在 makefile 中进入点的顺序是不重要的）。

所以 *make* 会先执行 all，虽然 all 下面没有任何命令（这样的语法也是正确的），但 all 必须依赖 edimh 和 readimh 两个进入点。这两个进入点都是属于真正的执行文件，所以 *make* 会依照相关性，从底层的 .c 文件名开始往回重建文件名。

下面是执行结果（你可能要有 root 权限才能将文件名安装在 */usr/local* 目录下）：

```
papaya$ make install
gcc -c main.c
gcc -c edit.c
gcc -o edimh main.o edit.o
gcc -c read.c
```

```
gcc -o readimh main.o read.o
mv edimh /usr/local
mv readimh /usr/local
```

所以这个 makefile 能够完整地编译、连接和安装程序。首先它会先建立好 *edimh* 所需要的文件名，然后建立好 *readmh* 所需要的文件名中尚未建立的部分，等这两个执行文件都建立好了，all 便会完成。接着 *make* 会再执行 install 部分，将两个执行文件放在正确的位置。

很多 makefile（包括编译 Linux 内核所用的 makefile）都含有许多假 target（phony target），以做一些例行的工作。例如 Linux 核心的 makefile 就包含用以删除临时文件的命令：

```
clean:  archclean
        rm -f kernel/ksyms.lst
        rm -f core `find . -name '*.[oas]' -print`
        .
        .
        .
```

内核的 makefile 也会建立所有目标文件和相关头文件的列表（这是件很复杂但又很重要的事，因为一旦头文件改变了，你必须确定所有引用到该头文件的文件名都被重新编译过）：

```
depend dep:
        touch tools/version.h
        for i in init/*.c;do echo -n "init/";$(CPP) -M $$i;done > .tmpdep
        .
        .
        .
```

这里头有些 shell 命令过于复杂，我们会在后面“多行命令”这节里讨论到 makefile 的命令写法。

一些语法规则

在编写 makefile 时，最困难的事莫过于语法的正确性（尤其是对于新用户而言更是如此）。让我们直接来讨论它吧，makefile 的语法相当的死板，比方说你想

使用空格来代替定位符号（tab）或是用定位符号来代替空格，那么 *make* 将会看不懂，而且 *make* 的错误信息也是模糊不清的。

所以记得命令一定要以定位符号为开头，不要用空格，而且不要在其他地方任意加上定位符号。

你可以在一行的任何地方使用 # 来告知 *make* 从这个符号之后都是注解，所有在 # 符号后面的字都会被忽略。

你可以在行尾加上 \，此时 *make* 会把下一行也视为同一行。这个符号在 makefile 任何一行都可以使用，而不仅限于很长的命令。

现在让我们来看看 *make* 较具威力的功能：*make* 定义了一种属于自己的语言。

宏（macro）

当人们在 makefile 内要使用同样的文件名或字符串一次以上，通常会将它定义成宏，所谓宏就是以一个字符串代替另一字符串的功能。例如你可以这样修改我们刚开始的范例：

```
OBJECTS = main.o edit.o

edimh: $(OBJECTS)
        gcc -o edimh $(OBJECTS)
```

当 *make* 执行时，它会很直接的将 main.o edit.o 这个字符串取代你写 $(OBJECTS) 的地方。如果你想在这个设计中增加另一个目标文件，只要在第一行的地方加上即可，而其他相关的部分会被自动更新。

当你使用 $(OBJECTS) 时不要忘记加上小括弧，宏可能很像 shell 里的环境变量（例如 $HOME 和 $PATH），但它们是不一样的。

宏可以在定义时用到另一宏，例如下面的范例：

```
ROOT = /usr/local
```

```
HEADERS = $(ROOT)/include
SOURCES = $(ROOT)/src
```

上面范例中的 HEADERS 会等于 */usr/local/include* 这个目录，而 SOURCES 则是 */usr/local/src*。如果你不想将这个软件安装在 */usr/local*，你只要改变定义 ROOT 的该行文字即可。

事实上宏不必一定要用大写来定义，但通常大家还是习惯用大写。

GNU 的 *make* 提供一项新功能，它允许你增加已经定义过的宏内容，你只须使用 := 这符号来取代等号就可以了。

```
DRIVERS         =drivers/block/block.a

ifdef CONFIG_SCSI
DRIVERS := $(DRIVERS) drivers/scsi/scsi.a
endif
```

第一行是普通的宏定义，用来定义 DRIVERS 为 drivers/block/block.a 这个文件名，后面的几行会判断 CONFIG_SCSI 是否被定义，如果被定义则 DRIVERS 会被添上 drivers/scsi/scsi.a 的文件名。所以这个例子中完整的定义是：

```
drivers/block/block.a drivers/scsi/scsi.a
```

那么你要如何定义 CONFIG_SCSI 呢？你可以把定义放在 makefile 内，并分派任何字符串给它：

```
CONFIG_SCSI = yes
```

但是你可能会发现在 *make* 的命令行上指定它会比较容易，下面是在命令行指定的做法：

```
papaya$ make CONFIG_SCSI=yes target_name
```

使用宏的微妙之处就是你可以不加任何定义，倘若你不加任何定义，那该字符串将会被空字符串替代（也就是原本宏所在的位置将会什么都没有）。另外你也可

以把宏定义成环境变量，举例来说，假如你在 makefile 中没有定义 CONFIG_SCSI，你可以更改你的 *.bashrc* 文件（倘若你使用 *bash* shell）：

```
export CONFIG_SCSI=yes
```

若你是使用 *csh* 或 *tcsh* 你可以改变 *.cshrc* 文件：

```
setenv CONFIG_SCSI yes
```

这样一来 CONFIG_SCSI 就会被定义好。

文件名后缀、模式匹配规则

在编写 makefile 时常有些例行的事项，例如，将源代码编译成目标文件这项工作，你可能会懒得分别定义其关联性。事实上，你的确可以不必如此。因为 UNIX 的编译器有一个很简单的标准（就是将文件尾为 *.c* 的文件名编译成文件尾为 *.o* 的目标文件），所以 *make* 提供一个叫做文件名后缀规则（suffix rule）的功能来支持这些文件名。

下面简单的语法可以用来编译 C 原始程序，你可以把它加在你的 makefile 中。

```
.c.o:
        gcc -c ${CFLAGS} $<
```

.c.o: 这行告诉 *make* 将 *.c* 的输入文件名转成 *.o* 的输出文件，CFLAGS 则是先前所定义的宏，你可以将所要执行的编译参数定义在 CFLAGS 中（例如 *-g* 启动调试功能、*-O* 启动最优化功能等等），而 $< 则代表了整个输入文件，所以在 *make* 执行时会自动以 *.c* 结尾的文件名来取代这字符串。

我们来执行一个简单的范例，其中命令行送出了 *-g* 和 *-O* 两个参数：

```
papaya$ make CFLAGS="-O -g" edit.o
gcc -c -O -g edit.c
```

事实上你可以不必指定上面的 *.c.o* 语法，因为 *make* 已经内置了一些常用的语法了，*make* 甚至使用了 CFLAGS，所以你只需要设定该变量以指定编译的参数

即可。例如用来建立 Linux 内核的 makefile 中就有下面这一连串 *gcc* 参数的定义：

```
CFLAGS = -Wall -Wstrict-prototypes -O2 -fomit-frame-pointer -pipe
```

让我们来介绍一下其他值得一提的编译参数：*-D* 参数是用来定义 C 源文件名中用到的符号（Symbol）。因为 #ifdefs 中会出现各式各样常用的符号，所以你可能会需要传入许多这类的参数到 makefile 中，例如 *-DDEBUG* 或 *-DBSD* 等等。如果你想要在 *make* 命令列使用这一语法，一定要用单引号或双引号括起来，因为你会希望 shell 把整个东西当成一个参数传至 makefile：

```
papaya$ make CFLAGS="-DDEBUG -DBSD" ...
```

GNU *make* 还提供了比文件名后缀判断还好用的功能，这个功能就是以模式（pattern）来判断。模式规则（*pattern rule*）使用百分比符号 % 来代表任何字符串，所以 C 原始文件名的编译可以用下列的语法来完成：

```
%.o: %.c
        gcc -c -o $@ $(CFLAGS) $<
```

在冒号前面的是输出文件 *%.o* 而后面是代表输入文件 *%.c*。简单的说，模式匹配就好像前面所叙述的一般规则一样，但它使用百分比符号来代替文件名的一部分。

你应该会注意到 $< 代表输入文件，但是我们也看到了 $@，它代表输出文件，所以所有 *.o* 的文件名都会被替换进去。这两个功能都是内置的宏，所以 *make* 在每次执行一个进入点时都会定义它们。

另一个常用的内置宏就是 $*，用来代表不包含结尾字符串的输入文件名。举例来说如果有个输入文件叫 *edit.c*，那么 $* 就代表了 *edit*，而 $*.s 就代表 *edit.s*（*.s* 常常用来表示汇编语言源程序文件）。

模式匹配有一项功能是原先文件后缀匹配所做不到的，比方说你想要在产生的输出文件文件名中间加上 _dbg 字符串来分辨出包含有调试信息的目标文件：

```
%_dbg.o: %.c
        gcc -c -g -o $@ $(CFLAGS) $<

DEBUG_OBJECTS = main_dbg.o edit_dbg.o

edimh_dbg: $(DEBUG_OBJECTS)
        gcc -o $@ $(DEBUG_OBJECTS)
```

所以你可以用两种方式来建立你的目标文件，一种含有调试信息，另一种则没有。而且这些文件的文件名皆不同，你可以把它们统统放在同一个目录内。

```
papaya$ make edimh_dbg
gcc -c -g -o main_dbg.o  main.c
gcc -c -g -o edit_dbg.o  edit.c
gcc -o edimh_dbg  main_dbg.o edit_dbg.o
```

多行命令

任何的 shell 命令都可以在 makefile 内执行，但是因为 *make* 会把每一行命令在不同的 shell 里执行，所以有时候会有一些复杂的情形发生，例如下面的例子执行时会发生错误：

```
target:
        cd obj
        HOST_DIR=/home/e
        mv *.o $HOST_DIR
```

此时不论是 *cd* 或是定义 HOST_DIR 变量都不会对之后的命令产生影响，所以你势必要将它们放在同一行来执行。shell 有一功能可以让你使用分号来隔开每个命令，如下面的例子：

```
target:
        cd obj ; HOST_DIR=/home/e ; mv *.o $$HOST_DIR
```

你会发现有另一个地方改变了，那就是 $ 变成 $$。这是为了让 *make* 知道此为 shell 变量而不是宏。

你可以用反斜线 \ 来将它们分成多行以便于阅读，使用反斜线时 *make* 还是会认为它们是同一行：

```
target:
        cd obj ; \
        HOST_DIR=/home/e ; \
        mv *.o $$HOST_DIR
```

makefile 有时候也会再用到 *make* 命令，我们称之为递归式 *make*，其语法如下所示：

```
linuxsubdirs: dummy
        set -e; for i in $(SUBDIRS); do $(MAKE) -C $$i; done
```

其中宏 $(MAKE) 会启动 *make*。有几个理由会让你必须用到这个功能：一是上面的例子里所示范的，要编译多层子目录，而且每个子目录都有一个 makefile 文件名，另一个理由是在命令行中定义宏，如此一来你可以根据不同的宏而有不同的编译方式。

GNU *make* 提供了另一个强大的 shell 界面以延伸其功能，你可以执行一个 shell 命令并将其输出内容定义给宏。在 Linux 内核的 makefile 中有几个例子，但是我们仅列出一个简单的例子：

```
HOST_NAME = $(shell uname -n)
```

这行会将 HOST_NAME 的内容定义成你所用的网络节点名称（也就是 *uname-n* 的输出）。

除此之外，*make* 还提供了两个方便的功能：一是你可以在你的命令前放一个 @ 符号，当 *make* 执行到这命令时，将不会把这行命令显示出来：

```
@if [ -x /bin/dnsdomainname ]; then \
  echo #define LINUX_COMPILE_DOMAIN \"`dnsdomainname`\"; \
 else \
  echo #define LINUX_COMPILE_DOMAIN \"`domainname`\"; \
 fi >> tools/version.h
```

另一个功能是你可以放一个破折号在你的命令前面，告诉 *make* 即使命令发生错误也不要停下来。这也是相当常用的，尤其是用 *mv* 或 *cp* 命令时：

```
- mv edimh /usr/local
- mv readimh /usr/local
```

引入其他的 Makefile

当有一个大的软件开发设计时，人们往往将其 makefile 分成不同的文件名。这样做可以很容易地让不同目录下的 makefile 共用一些信息，尤其是一些宏定义。例如：

```
include filename
```

会读取 *filename* 这个文件名的内容，并将之载入。你可以在 Linux 内核的 makefile 中看到类似的例子，举例来说：

```
include .depend
```

如果你看了 *.depend* 这个文件名，你会看到一堆 makefile 的进入点。说的更清楚一点，应该说是有许多目标文件和头文件之间关系的声明（事实上你可能会找不到 *.depend*，因为它是由 makefile 中的另一个进入点产生的）。

有时候 include 后面可以接宏而不是文件名：

```
include $(INC_FILE)
```

上例中的 INC_FILE 可能是环境变量或宏，这让我们能更容易地控制要使用哪个文件名。

autoconf 与 automake

为一个大型项目编写 Makefile 是一项繁杂又浪费时间的工作，尤其当程序必须在多个操作系统平台上编译时。在 GNU 中有两个实用的工具：*autoconf* 与 *automake*，他们具备陡峭的学习曲线，一旦熟悉之后，它们可以大幅简化建立跨