GNU Make 使用手册（中译版）

翻译：于凤昌

译者注：本人在阅读 **Linux** 源代码过程中发现如果要全面了解 **Linux** 的结构、理解 **Linux** 的编程总体设计及思想必须首先全部读通 **Linux** 源代码中各级的 **Makefile** 文件。目前，在网上虽然有一些著作，但都不能全面的解释 **Linux** 源代码中各级的 **Makefile** 文件，因此本人认真阅读了 **GNU Make** 使用手册（**3.79**）版原文，在此基础上翻译了该手册，以满足**Linux** 源代码有兴趣或者希望采用 **GCC** 编写程序但对缺乏 **GNU Make** 全面了解之人士的需要。本人是业余爱好 不是专业翻译人士，如果有 问题请通过电子信箱与我联系共同商讨 ，本人的 **E-mail** 为：**yfc70@public2.lyptt.ha.cn** 。**注意在文章中出现的斜体加粗字表示章节。**

GNU make Version 3.79

April 2000

Richard M. Stallman and Roland McGrath

目录

1 Make 概述

Make 可自动决定一个大程序中哪些文件需要重新编译，并发布重新编译它们的命令。本版本 GNU Make 使用手册由 Richard M. Stallman and Roland McGrath 编著，是从 Paul D. Smith 撰写的 V3.76 版本发展过来的。

GNU Make 符合 *IEEE Standard 1003.2-1992* (POSIX.2) 6.2 章节的规定。

因为 C 语言程序更具有代表性，所以我们的例子基于 C 语言程序，但 Make 并不是仅仅能够处理 C 语言程序，它可以处理那些编译器能够在 Shell 命令下运行的的各种语言的程序。事实上，GNU Make 不仅仅限于程序，它可以适用于任何如果一些文件变化导致另外一些文件必须更新的任务。

如果要使用 Make，必须先写一个称为 Makefile 的文件，该文件描述程序中各个文件之间的相互关系，并且提供每一个文件的更新命令。在一个程序中，可执行程序文件的更新依靠 OBJ 文件，而 OBJ 文件是由源文件编译得来的。

一旦合适的 Makefile 文件存在，每次更改一些源文件，在 shell 命令下简单的键入：

```
make
```

就能执行所有的必要的重新编译任务。Make 程序根据 Makefile 文件中的数据和每个文件更改的时间戳决定哪些文件需要更新。对于这些需要更新的文件，Make 基于 Makefile 文件发布命令进行更新，进行更新的方式由提供的命令行参数控制。具体操作请看*运行 Make* 章节。

1.1 怎样阅读本手册

如果您现在对 Make 一无所知或者您仅需要了解对 make 的普通性介绍，请查阅前几章内容，略过后面的章节。前几章节是普通介绍性内容，后面的章节是具体的专业、技术内容。

如果您对其它 Make 程序十分熟悉，请参阅 *GNU Make 的特点*和*不兼容性和失去的特点*部分，*GNU Make 的特点*这一章列出了 GNU Make 对 make 程序的扩展，*不兼容和失去的特点*一章解释了其它 Make 程序有的特征而 GNU Make 缺乏的原因。

对于快速浏览者，请参阅*选项概要*、*快速参考*和*内建的特殊目标名*部分。

1.2 问题和 BUG

如果您有关于 GNU Make 的问题或者您认为您发现了一个 BUG，请向开发者报告；我们不能许诺我们能干什么，但我们会尽力修正它。在报告 BUG 之前，请确定您是否真正发现了 BUG，仔细研究文档后确认它是否真的按您的指令运行。如果文档不能清楚的告诉您怎么做，也要报告它，这是文档的一个 BUG。

在您报告或者自己亲自修正 BUG 之前，请把它分离出来，即在使问题暴露的前提下尽可能的缩小 Makefile 文件。然后把这个 Makefile 文件和 Make 给出的精确结果发给我们。同

时请说明您希望得到什么，这可以帮助我们确定问题是否出在文档上。

一旦您找到一个精确的问题，请给我们发 E-mail，我们的 E-mail 地址是：

bug-make@gnu.org

在邮件中请包含您使用的 GNU Make的版本号。您可以利用命令‘make--version’得到版本号。同时希望您提供您的机器型号和操作系统类型，如有可能的话，希望同时提供 config.h 文件（该文件有配置过程产生）。

2 Makefile 文件介绍

Make 程序需要一个所谓的 Makefile 文件来告诉它干什么。在大多数情况下，Makefile 文件告诉 Make 怎样编译和连接成一个程序。

本章我们将讨论一个简单的 Makefile 文件，该文件描述怎样将 8 个 C 源程序文件和 3 个头文件编译和连接成为一个文本编辑器。Makefile 文件可以同时告诉 Make 怎样运行所需要的杂乱无章的命令（例如，清除操作时删除特定的文件）。如果要看更详细、复杂的 Makefile 文件例子，请参阅*复杂的 Makefile 文件例子*一章。

当 Make 重新编译这个编辑器时，所有改动的 C 语言源文件必须重新编译。如果一个头文件改变，每一个包含该头文件的 C 语言源文件必须重新编译，这样才能保证生成的编辑器是所有源文件更新后的编辑器。每一个 C 语言源文件编译后产生一个对应的 OBJ 文件，如果一个源文件重新编译，所有的 OBJ 文件无论是刚刚编译得到的或原来编译得到的必须从新连接，形成一个新的可执行文件。

2.1 规则的格式

一个简单的 Makefile 文件包含一系列的“规则”，其样式如下：

目标(target)…: 依赖(prerequiries)…

<tab>命令(command)

…

…

目标(target)通常是要产生的文件的名称，目标的例子是可执行文件或 OBJ 文件。目标也可是一个执行的动作名称，诸如‘clean’（详细内容请参阅*假想目标*一节）。

依赖是用来输入从而产生目标的文件，一个目标经常有几个依赖。

命令是 Make 执行的动作，一个规则可以含有几个命令，每个命令占一行。**注意：每个命令行前面必须是一个 Tab 字符，即命令行第一个字符是Tab。**这是不小心容易出错的地方。

通常，如果一个依赖发生变化，则需要规则调用命令对相应依赖和服务进行处理从而更新或创建目标。但是，指定命令更新目标的规则并不都需要依赖，例如，包含和目标‘clern’相联系的删除命令的规则就没有依赖。

规则一般是用于解释怎样和何时重建特定文件的，这些特定文件是这个详尽规则的目标。Make 需首先调用命令对依赖进行处理，进而才能创建或更新目标。当然，一个规则也可以是用于解释怎样和何时执行一个动作，详见*编写规则*一章。

一个 Makefile 文件可以包含规则以外的其它文本，但一个简单的 Makefile 文件仅仅需要包含规则。虽然真正的规则比这里展示的例子复杂，但格式却是完全一样。

2.2 一个简单的 Makefile 文件

一个简单的 Makefile 文件，该文件描述了一个称为文本编辑器（edit）的可执行文件生成方法，该文件依靠 8 个 OBJ 文件（.o 文件），它们又依靠 8 个 C 源程序文件和 3 个头文

件。

在这个例子中，所有的 C 语言源文件都包含‘defs.h’ 头文件，但仅仅定义编辑命令的源文件包含‘command.h’头文件，仅仅改变编辑器缓冲区的低层文件包含‘buffer.h’头文件。

```
edit : main.o kbd.o command.o display.o \
       insert.o search.o files.o utils.o
        cc -o edit main.o kbd.o command.o display.o \
                   insert.o search.o files.o utils.o

main.o : main.c defs.h
        cc -c main.c
kbd.o : kbd.c defs.h command.h
        cc -c kbd.c
command.o : command.c defs.h command.h
        cc -c command.c
display.o : display.c defs.h buffer.h
        cc -c display.c
insert.o : insert.c defs.h buffer.h
        cc -c insert.c
search.o : search.c defs.h buffer.h
        cc -c search.c
files.o : files.c defs.h buffer.h command.h
        cc -c files.c
utils.o : utils.c defs.h
        cc -c utils.c
clean :
        rm edit main.o kbd.o command.o display.o \
           insert.o search.o files.o utils.o
```

我们把每一个长行使用反斜杠-新行法分裂为两行或多行，实际上它们相当于一行，这样做的意图仅仅是为了阅读方便。

使用 Makefile 文件创建可执行的称为‘edit’的文件，键入：make

使用 Makefile 文件从目录中删除可执行文件和目标，键入：make clean

在这个 Makefile 文件例子中，目标包括可执行文件‘edit’和 OBJ 文件‘main.o’及‘kdb.o’。依赖是 C 语言源文件和 C 语言头文件如‘main.c’和‘def.h’等。事实上，每一个 OBJ 文件即是目标也是依赖。所以命令行包括‘cc -c main.c’和‘cc -c kbd.c’。

当目标是一个文件时，如果它的任一个依赖发生变化，目标必须重新编译和连接。任何命令行的第一个字符必须是‘Tab’字符，这样可以把 Makefile 文件中的命令行与其它行分别开来。（**一定要牢记：Make 并不知道命令是如何工作的，它仅仅能向您提供保证目标的合适更新的命令。Make 的全部工作是当目标需要更新时，按照您制定的具体规则执行命令。**）

目标‘clean’不是一个文件，仅仅是一个动作的名称。正常情况下，在规则中‘clean’这个动作并不执行，目标‘clean’也不需要任何依赖。一般情况下，除非特意告诉 make 执行‘clean’命令，否则‘clean’命令永远不会执行。注意这样的规则不需要任何依赖，它们存在的目的仅仅是执行一些特殊的命令。象这些不需要依赖仅仅表达动作的目标称为假想目标。详细内容参见*假想目标*；参阅*命令错误*可以了解 rm 或其它命令是怎样导致 make 忽略错误的。

2.3 make 处理 makefile 文件的过程

缺省情况下，make 开始于第一个目标（假想目标的名称前带‘.’）。这个目标称为缺省最终目标（即 make 最终更新的目标，具体内容请看*指定最终目标的参数*一节）。

在上节的简单例子中，缺省最终目标是更新可执行文件‘edit’，所以我们将该规则设

为第一规则。这样，一旦您给出命令：

```
make
```

make 就会读当前目录下的 **makefile** 文件，并开始处理第一条规则。在本例中，第一条规则是连接生成‘**edit**’，但在 **make** 全部完成本规则工作之前，必须先处理‘**edit**’所依靠的 **OBJ** 文件。这些 **OBJ** 文件按照各自的规则被处理更新，每个 **OBJ** 文件的更新规则是编译其源文件。重新编译根据其依靠的源文件或头文件是否比现存的 **OBJ** 文件更‘新’，或者 **OBJ** 文件是否存在来判断。

其它规则的处理根据它们的目标是否和缺省最终目标的依赖相关联来判断。如果一些规则和缺省最终目标无任何关联则这些规则不会被执行，除非告诉 **Make** 强制执行（如输入执行 **make clean** 命令）。

在 **OBJ** 文件重新编译之前，**Make** 首先检查它的依赖 **C** 语言源文件和 **C** 语言头文件是否需要更新。如果这些 **C** 语言源文件和 **C** 语言头文件不是任何规则的目标，**make** 将不会对它们做任何事情。**Make** 也可以自动产生 **C** 语言源程序，这需要特定的规则，如可以根据 **Bison** 或 **Yacc** 产生 **C** 语言源程序。

在 **OBJ** 文件重新编译（如果需要的话）之后，**make** 决定是否重新连接生成 **edit** 可执行文件。如果 **edit** 可执行文件不存在或任何一个 **OBJ** 文件比存在的 **edit** 可执行文件‘新’，则 **make** 重新连接生成 **edit** 可执行文件。

这样，如果我们修改了‘**insert.c**’文件，然后运行 **make**，**make** 将会编译‘**insert.c**’文件更新‘**insert.o**’文件，然后重新连接生成 **edit** 可执行文件。如果我们修改了‘**command.h**’文件，然后运行 **make**，**make** 将会重新编译‘**kbd.o**’和‘**command.o**’文件，然后重新连接生成 **edit** 可执行文件。

2.4 使用变量简化 makefile 文件

在我们的例子中，我们在‘**edit**’的生成规则中把所有的 **OBJ** 文件列举了两次，这里再重复一遍：

```
edit : main.o kbd.o command.o display.o \
       insert.o search.o files.o utils.o
        cc -o edit main.o kbd.o command.o display.o \
                   insert.o search.o files.o utils.o
```

这样的两次列举有出错的可能，例如在系统中加入一个新的 **OBJ** 文件，我们很有可能在一个需要列举的地方加入了，而在另外一个地方却忘记了。我们使用变量可以简化 **makefile** 文件并且排除这种出错的可能。变量是定义一个字符串一次，而能在多处替代该字符串使用（具体内容请阅读***使用变量***一节）。

在 **makefile** 文件中使用名为 **objects, OBJECTS, objs, OBJS, obj,** 或 **OBJ** 的变量代表所有 **OBJ** 文件已是约定成俗。在这个 **makefile** 文件我们定义了名为 **objects** 的变量，其定义格式如下：

```
objects = main.o kbd.o command.o display.o \
          insert.o search.o files.o utils.o
```

然后，在每一个需要列举 **OBJ** 文件的地方，我们使用写为**`$(objects)'**形式的变量代替（具体内容请阅读***使用变量***一节）。下面是使用变量后的完整的 **makefile** 文件：

```
objects = main.o kbd.o command.o display.o \
          insert.o search.o files.o utils.o

edit : $(objects)
        cc -o edit $(objects)
main.o : main.c defs.h
        cc -c main.c
kbd.o : kbd.c defs.h command.h
        cc -c kbd.c
command.o : command.c defs.h command.h
```

```
        cc -c command.c
display.o : display.c defs.h buffer.h
        cc -c display.c
insert.o : insert.c defs.h buffer.h
        cc -c insert.c
search.o : search.c defs.h buffer.h
        cc -c search.c
files.o : files.c defs.h buffer.h command.h
        cc -c files.c
utils.o : utils.c defs.h
        cc -c utils.c
clean :
        rm edit $(objects)
```

2.5 让 make 推断命令

编译单独的 C 语言源程序并不需要写出命令，因为 make 可以把它推断出来：make 有一个使用‘CC －c’命令的把 C 语言源程序编译更新为相同文件名的 OBJ 文件的隐含规则。例如 make 可以自动使用‘cc -c main.c -o main.o’命令把‘main.c’编译 ‘main.o’。因此，我们可以省略 OBJ 文件的更新规则。详细内容请看*使用隐含规则*一节。

如果 C 语言源程序能够这样自动编译，则它同样能够自动加入到依赖中。所以我们可在依赖中省略 C 语言源程序，进而可以省略命令。下面是使用隐含规则和变量 objects 的完整 makefile 文件的例子：

```
objects = main.o kbd.o command.o display.o \
          insert.o search.o files.o utils.o

edit : $(objects)
        cc -o edit $(objects)

main.o : defs.h
kbd.o : defs.h command.h
command.o : defs.h command.h
display.o : defs.h buffer.h
insert.o : defs.h buffer.h
search.o : defs.h buffer.h
files.o : defs.h buffer.h command.h
utils.o : defs.h

.PHONY : clean
clean :
        -rm edit $(objects)
```

这是我们实际编写 makefile 文件的例子。（和目标‘clean’联系的复杂情况在别处阐述。具体参见*假想目标*及*命令错误*两节内容。）因为隐含规则十分方便，所以它们非常重要，在 makefile 文件中经常使用它们。

2.6 另一种风格的 makefile 文件

当时在 makefile 文件中使用隐含规则创建 OBJ 文件时，采用另一种风格的 makefile 文件也是可行的。在这种风格的 makefile 文件中，可以依据依赖分组代替依据目标分组。下面是采用这种风格的 makefile 文件：

```
objects = main.o kbd.o command.o display.o \
          insert.o search.o files.o utils.o

edit : $(objects)
        cc -o edit $(objects)

$(objects) : defs.h
kbd.o command.o files.o : command.h
display.o insert.o search.o files.o : buffer.h
```

这里的 **defs.h** 是所有 **OBJ** 文件的共同的一个依赖；**command.h** 和 **bufffer.h** 是具体列出的 **OBJ** 文件的共同依赖。

虽然采用这种风格编写 **makefile** 文件更具风味：**makefile** 文件更加短小，但一部分人以为把每一个目标的信息放到一起更清晰易懂而不喜欢这种风格。

2.7 在目录中删除文件的规则

编译程序并不是编写 **make** 规则的唯一事情。**Makefile** 文件可以告诉 **make** 去完成编译程序以外的其它任务，例如，怎样删除 **OBJ** 文件和可执行文件以保持目录的‘干净’等。下面是删除利用 **make** 规则编辑器的例子：

```
clean:
            rm edit $(objects)
```

在实际应用中，应该编写较为复杂的规则以防不能预料的情况发生。更接近实用的规则样式如下：

```
.PHONY : clean
clean :
            -rm edit $(objects)
```

这样可以防止 **make** 因为存在名为’**clean**’的文件而发生混乱，并且导致它在执行 **rm** 命令时发生错误（具体参见*假想目标*及*命令错误*两节内容）。

诸如这样的规则不能放在 **makefile** 文件的开始，因为我们不希望它变为缺省最终目标。应该象我们的 **makefile** 文件例子一样，把关于 **edit** 的规则放在前面，从而把编译更新 **edit** 可执行程序定为缺省最终目标。

3 编写 makefile 文件

make 编译系统依据的信息来源于称为 **makefile** 文件的数据库。

3.1 makefile 文件的内容

makefile 文件包含 **5** 方面内容：具体规则、隐含规则、定义变量、指令和注释。规则、变量和指令将在后续章节介绍。

- **具体规则**用于阐述什么时间或怎样重新生成称为规则目标的一个或多个文件的。它列举了目标所依靠的文件，这些文件称为该目标的依赖。具体规则可能同时提供了创建或更新该目标的命令。详细内容参阅*编写规则*一章。
- **隐含规则**用于阐述什么时间或怎样重新生成同一文件名的一系列文件的。它描述的目标是根据和它名字相同的文件进行创建或更新的，同时提供了创建或更新该目标的命令。详细内容参阅*使用隐含规则*一节。

- 定义变量是为一个变量赋一个固定的字符串值，从而在以后的文件中能够使用该变量代替这个字符串。注意在 **makefile** 文件中定义变量占一独立行。在上一章的 **makefile** 文件例子中我们定义了代表所有 **OBJ** 文件的变量 **objects**(详细内容参阅 ***使用变量简化 makefile 文件***一节）。
- 指令是 **make** 根据 **makefile** 文件执行一定任务的命令。这些包括如下几方面**:**
  - 读其它 **makefile** 文件（详细内容参见***包含其它的 makefile 文件***）。
  - 判定（根据变量的值）是否使用或忽略 **makefile** 文件的部分内容（详细内容参阅 ***makefile 文件的条件语句***一节）。
  - 定义多行变量，即定义变量值可以包含多行字符的变量（详细内容参见***定义多行变量***一节）。
- 以‘**#**’开始的行是注释行。注释行在处理时将被 **make** 忽略，如果一个注释行在行尾是‘****’则表示下一行继续为注释行，这样注释可以持续多行。除在 **define** 指令内部外，注释可以出现在 **makefile** 文件的任何地方，甚至在命令内部（这里 **shell** 决定什么是注释内容）。

3.2 makfile 文件的命名

缺省情况下，当 **make** 寻找 **makefile** 文件时，它试图搜寻具有如下的名字的文件，按顺序：‘**GNUmakefile**’、‘**makefile**’ 和 ‘**Makefile**’。

通常情况下您应该把您的 **makefile** 文件命名为 ‘**makefile**’ 或 ‘**Makefile**’。**(**我们推荐使用 ‘**Makefile**’，因为它基本出现在目录列表的前面， 后面挨着其它重要的文件如 ‘**README**’等。**)**。虽然首先搜寻‘**GNUmakefile**’，但我们并不推荐使用。除非您的 **makefile** 文件是特为 **GNU make**编写的，在其它 **make** 版本上不能执行，您才应该使用 ‘**GNUmakefile**’作为您的 **makefile** 的文件名。

如果 **make** 不能发现具有上面所述名字的文件，它将不使用任何 **makefile** 文件。这样您必须使用命令参数给定目标，**make** 试图利用内建的隐含规则确定如何重建目标。详细内容参见***使用隐含规则***一节。

如果您使用非标准名字 **makefile** 文件，您可以使用 ‘**-f**’ 或 ‘**--file**’ 参数指定您的 **makefile** 文件。参数 ‘**-f name**’ 或 ‘**--file=name**’ 能够告诉 **make** 读名字为 ‘**name**’ 的文件作为 **makefile** 文件。如果您使用 ‘**-f**’ 或 ‘**--file**’ 参数多于一个，意味着您指定了多个 **makefile** 文件，所有的 **makefile** 文件按具体的顺序发生作用。一旦您使用了 ‘**-f**’ 或 ‘**--file**’ 参数，将不再自动检查是否存在名为 ‘**GNUmakefile**’、‘**makefile**’ 或 ‘**Makefile**’ 的 **makefile** 文件。

3.3 包含其它的 makefile 文件

include 指令告诉 **make** 暂停读取当前的 **makefile** 文件，先读完 **include** 指令指定的 **makefile** 文件后再继续。指令在 **makefile** 文件占单独一行，其格式如下：

include ***filenames...***

filenames 可以包含 **shell** 文件名的格式。

在 **include** 指令行，行开始处的多余的空格是允许的，但 **make** 处理时忽略这些空格，注意该行不能以 **Tab** 字符开始（因为，以**Tab** 字符开始的行，**make** 认为是命令行）。**include** 和文件名之间以空格隔开，两个文件名之间也以空格隔开，多余的空格 **make** 处理时忽略，在该行的尾部可以加上以 ‘**#**’ 为起始的注释。文件名可以包含变量及函数调用，它们在处理时由 **make** 进行扩展（具体内容参阅***使用变量***一节）。

例如，有三个 ‘**.mk**’ 文件：‘**a.mk**’、‘**b.mk**’ 和 ‘**c.mk**’，变量**$(bar)**扩展为 **bish bash** 则下面的表达是：

include foo *.mk $(bar)

和 ‘**include foo a.mk b.mk c.mk bish bash**’ 等价。

当 **make** 遇见 **include** 指令时， **make** 就暂停读取当前的 **makefile** 文件，依次读取列举

的 makefile 文件，读完之后，make 再继续读取当前 makefile 文件中 include 指令以后的内容。

使用 include 指令的一种情况是几个程序分别有单独的 makefile 文件，但它们需要一系列共同的变量定义（详细内容参阅*设置变量*），或者一系列共同的格式规则（详细内容参阅*定义与重新定义格式规则*）。

另一种使用 include 指令情况是需要自动从源文件为目标产生依赖的情况，此时，依赖在主 makefile 文件包含的文件中。这种方式比其它版本的 make 把依赖附加在主 makefile 文件后部的传统方式更显得简洁。具体内容参阅*自动产生依赖*。

如果 makefile 文件名不以‘/’开头，并且在当前目录下也不能找到，则需搜寻另外的目录。首先，搜寻以‘-|’或‘--include-dir’参数指定的目录，然后依次搜寻下面的目录（如果它们存在的话）：‘*prefix*/include' (通常为 ‘/usr/local/include') ‘/usr/gnu/include', ‘/usr/local/include', ‘/usr/include'。

如果指定包含的 makefile 文件在上述所有的目录都不能找到，make 将产生一个警告信息，注意这不是致命的错误。处理完 include 指令包含的 makefile 文件之后，继续处理当前的 makefile 文件。一旦完成 makefile 文件的读取操作，make 将试图创建或更新过时的或不存在的 makefile 文件。详细内容参阅 *makefile 文件重新生成的过程*。只有在所有 make 寻求丢失的 makefile 文件的努力失败后，make 才能断定丢失的 makefile 文件是一个致命的错误。

如果您希望对不存在且不能重新创建的 makefile 文件进行忽略，并且不产生错误信息，则使用-include 指令代替 include 指令，格式如下：

-include *filenames...*

这种指令的作用就是对于任何不存在的 makefile 文件都不会产生错误（即使警告信息也不会产生）。如果希望保持和其它版本的 make 兼容，使用 sinclude 指令代替-include 指令。

3.4 变量 MAKEFILES

如果定义了环境变量 MAKEFILES，make 认为该变量的值是一列附加的 makefile 文件名，文件名之间由空格隔开，并且这些 makefile 文件应首先读取。Make 完成这个工作和上节完成 include 指令的方式基本相同，即在特定的目录中搜寻这些文件。值得注意的是，缺省最终目标不会出现在这些 makefile 文件中，而且如果一些 makefile 文件没有找到也不会出现任何错误信息。

环境变量 MAKEFILES 主要在 make 递归调用过程中起通讯作用（详细内容参阅*递归调用 make*）。在 make 顶级调用之前设置环境变量并不是十分好的主意，因为这样容易将 makefile 文件与外界的关系弄的更加混乱。然而如果运行 make 而缺少 makefile 文件时，环境变量 MAKEFILES 中 makefile 文件可以使内置的隐含规则更好的发挥作用，如搜寻定义的路径等（详细内容参阅*在目录中搜寻依赖*）。

一些用户喜欢在登录时自动设置临时的环境变量 MAKEFILES，而 makefile 文件在该变量指定的文件无效时才使用。这是非常糟糕的主意，应为许多 makefile 文件在这种情况下运行失效。最好的方法是直接在 makefile 文件中写出具体的 include 指令(详细内容参看上一节)。

3.5 makefile 文件重新生成的过程

有时 makefile 文件可以由其它文件重新生成，如从 RCS 或 SCCS 文件生成等。如果一个 makefile 文件可以从其它文件重新生成，一定注意让 make 更新 makefile 文件之后再读取 makefile 文件。

完成读取所有的 makefile 文件之后，make 检查每一个目标，并试图更新它。如果对于一个 makefile 文件有说明它怎样更新的规则（无论在当前的 makefile 文件中或其它 makefile 文件中），或者存在一条隐含规则说明它怎样更新（具体内容参见*使用隐含规则*），则在必要的时候该 makefile 文件将会自动更新。在所有的 makefile 文件检查之后，如果发

现任何一个 **makefile** 文件发生变化，**make** 就会清空所有记录，并重新读入所有 **makefile** 文件。（然后再次试图更新这些 **makefile** 文件，正常情况下，因为这些 **makefile** 文件已被更新，**make** 将不会再更改它们。）

如果您知道您的一个或多个 **makefile** 文件不能重新创建，也许由于执行效率缘故，您不希望 **make** 按照隐含规则搜寻或重建它们，您应使用正常的方法阻止按照隐含规则检查它们。例如，您可以写一个具体的规则，把这些 **makefile** 文件当作目标，但不提供任何命令（详细内容参阅*使用空命令*）。

如果在 **makefile** 文件中指定依据双冒号规则使用命令重建一个文件，但没有提供依赖，则一旦 **make** 运行就会重建该文件（详细内容参见*双冒号规则*）。同样，如果在 **makefile** 文件中指定依据双冒号规则使用命令重建的一个 **makefile** 文件，并且不提供依赖，则一旦 **make** 运行就会重建该 **makefile** 文件，然后重新读入所有 **makefile** 文件，然后再重建该 **makefile** 文件，再重新读入所有 **makefile** 文件，如此往复陷入无限循环之中，致使 **make** 不能再完成别的任务。如果要避免上述情况的发生，**一定注意不要依据双冒号规则使用命令并且不提供依赖重建任何 makefile 文件。**

如果您没有使用‘**-f**’或‘**--file**’指定 **makefile** 文件，**make** 将会使用缺省的 **makefile** 文件名（详细内容参见 **3.2** 节内容）。假象使用‘**-f**’或‘**--file**’选项指定具体的 **makefile** 文件，这时 **make** 不能确定 **makefile** 文件是否存在。如果缺省的 **makefile** 文件不存在，但可以由运行的 **make** 依据规则创建，您需要运行这些规则，创建要使用的 **makefile** 文件。

如果缺省的 **makefile** 文件不存在，**make** 将会按照搜寻的次序将它们试着创建，一直到将 **makefile** 文件成功创建或 **make** 将所有的文件名都试过来。注意 **make** 不能找到或创建 **makefile** 文件不是错误，**makefile** 文件并不是运行 **make** 必须的。

因为即使您使用‘**-t**’特别指定，‘**-t**’或‘**--touch**’选项对更新 **makefile** 文件不产生任何影响， **makefile** 文件仍然会更新，所以当您使用‘**-t**’或‘**--touch**’选项时，您不要使用过时的 **makefile** 文件来决定‘**touch**’哪个目标（具体含义参阅*代替执行命令*）。同样，因为‘**-q' (**或 ‘**--question') 和 ‘-n' (**或 ‘**--just-print')**也能不阻止更新 **makefile** 文件，所以过时的 **makefile** 文件对其它的目标将产生错误的输出结果。如，‘**make -f mfile -n foo**’命令将这样执行：更新‘**mfile**’，然后读入，再输出更新‘**foo**’的命令和依赖，但并不执行更新‘**foo**’，注意，所有回显的更新‘**foo**’的命令是在更新后的‘**mfile**’中指定的。

在实际使用过程中，您一定会遇见确实希望阻止更新 **makefile** 文件的情况。如果这样，您可以在 **makefile** 文件命令行中将需要更新的 **makefile** 文件指定为目标，如此则可阻止更新 **makefile** 文件。一旦 **makefile** 文件名被明确指定为一个目标，选项‘**-t**’等将会对它发生作用。如这样设定，‘**make -f mfile -n foo**’命令将这样执行：读入‘**mfile**’，输出更新‘**foo**’的命令和依赖，但并不执行更新‘**foo**’。回显的更新‘**foo**’的命令包含在现存的‘**mfile**’中。

3.6 重载其它 makefile 文件

有时一个 **makefile** 文件和另一个 **makefile** 文件相近也是很有用的。您可以使用‘**include**’指令把更多的 **makefile** 文件包含进来，如此可加入更多的目标和定义的变量。然而如果两个 **makefile** 文件对相同的目标给出了不同的命令，**make** 就会产生错误。

在主 **makefile** 文件（要包含其它 **makefile** 文件的那个）中，您可以使用通配符格式规则说明只有在依靠当前 **makefile** 文件中的信息不能重新创建目标时，**make** 才搜寻其它的 **makefile** 文件，详细内容参见*定义与重新定义格式规则*。

例如：如果您有一个说明怎样创建目标‘**foo**’（和其它目标）的 **makefile** 文件称为‘**Makefile**’，您可以编写另外一个称为‘**GNUmakefile**’的 **makefile** 文件包含以下语句：

```
foo:
        frobnicate > foo

%: force
        @$(MAKE) -f Makefile $@
```

```
force: ;
```

如果键入‘**make foo**’，**make** 就会找到‘**GNUmakefile**’，读入，然后运行‘**frobnicate > foo**’。如果键入‘**make bar**’，**make** 发现无法根据‘**GNUmakefile**’创建‘**bar**’，它将使用格式规则提供的命令：‘**make -f Makefile bar**’。如果在‘**Makefile**’中提供了‘**bar**’更新的规则，**make** 就会使用该规则。对其它‘**GNUmakefile**’不提供怎样更新的目标 **make** 也会同样处理。这种工作的方式是使用了格式规则中的格式匹配符‘**%**’，它可以和任何目标匹配。该规则指定了一个依赖‘**force**’，用来保证命令一定要执行，无论目标文件是否存在。我们给出的目标‘**force**’时使用了空命令，这样可防止 **make** 按照隐含规则搜寻和创建它，否则，**make** 将把同样的匹配规则应用到目标‘**force**’本身，从而陷入创建依赖的循环中。

3.7 make 读取 makefile 文件的过程

GNU make把它的工作明显的分为两个阶段。在第一阶段，**make** 读取 **makefile** 文件，包括 **makefile** 文件本身、内置变量及其值、隐含规则和具体规则、构造所有目标的依靠图表和它们的依赖等。在第二阶段，**make** 使用这些内置的组织决定需要重新构造的目标以及使用必要的规则进行工作。

了解 **make** 两阶段的工作方式十分重要，因为它直接影响变量、函数扩展方式；而这也是编写 **makefile** 文件时导致一些错误的主要来源之一。下面我们将对 **makefile** 文件中不同结构的扩展方式进行总结。我们称在 **make** 工作第一阶段发生的扩展是立即扩展：在这种情况下，**make** 对 **makefile** 文件进行语法分析时把变量和函数直接扩展为结构单元的一部分。我们把不能立即执行的扩展称为延时扩展。延时扩展结构直到它已出现在上下文结构中或 **make** 已进入到了第二工作阶段时才执行展开。

您可能对这一部分内容不熟悉。您可以先看完后面几章对这些知识熟悉后再参考本节内容。

变量赋值

变量的定义语法形式如下：

```
immediate = deferred
immediate ?= deferred
immediate := immediate
immediate += deferred or immediate

define immediate
  deferred
endef
```

对于附加操作符‘**+=**’，右边变量如果在前面使用（**:=**）定义为简单扩展变量则是立即变量，其它均为延时变量。

条件语句

整体上讲，条件语句都按语法立即分析，常用的有：**ifdef**、**ifeq**、**ifndef** 和 **inneq**。

定义规则

规则不论其形式如何，都按相同的方式扩展。

```
immediate : immediate ; deferred
        deferred
```

目标和依赖部分都立即扩展，用于构造目标的命令通常都是延时扩展。这个通用的规律对具体规则、格式规则、后缀规则、静态格式规则和简单依赖定义都适用。

4 编写规则

makefile 文件中的规则是用来说明何时以及怎样重建特定文件的，这些特定的文件称为该规则的目标（通常情况下，每个规则只有一个目标）。在规则中列举的其它文件称为目标的依赖，同时规则还给出了目标创建、更新的命令。一般情况下规则的次序无关紧要，但决定缺省最终目标时却是例外。缺省最终目标是您没有另外指定最终目标时，**make** 认定的最终目标。缺省最终目标是 **makefile** 文件中的第一条规则的目标。如果第一条规则有多个目标，只有第一个目标被认为是缺省最终目标。有两种例外的情况：以句点（‘**.**’）开始的目标不是缺省最终目标（如果该目标包含一个或多个斜杠‘**/**’，则该目标也可能是缺省最终目标）；另一种情况是格式规则定义的目标不是缺省最终目标（参阅***定义与重新定义格式规则***）。

所以，我们编写 **makefile** 文件时，通常将第一个规则的目标定为编译全部程序或是由 **makefile** 文件表述的所有程序（经常设定一个称为‘**all**’的目标）。参阅***指定最终目标的参数***。

4.1 规则的语法

通常一条规则形式如下：

```
targets : prerequisites
        command
        ...
```

或：

```
targets : prerequisites ; command
        command
        ...
```

目标（**target**）是文件的名称，中间由空格隔开。通配符可以在文件名中使用（参阅***在文件名中使用通配符***），‘**a**(**m**)’形式的文件名表示成员 **m** 在文件 **a** 中（参阅***档案成员目标***）。一般情况下，一条规则只有一个目标，但偶尔由于其它原因一条规则有多个目标（参阅***具有多个目标的规则***）。

命令行以 **Tab** 字符开始，第一个命令可以和依赖在一行，命令和依赖之间用分号隔开，也可以在依赖下一行，以 **Tab** 字符为行的开始。这两种方法的效果一样，参阅***在规则中使用命令***。

因为美元符号已经用为变量引用的开始符，如果您真希望在规则中使用美元符号，您必须连写两次，‘**$$**’（参阅***使用变量***）。您可以把一长行在中间插入‘****’使其分为两行，也就是说，一行的尾部是’****’的话，表示下一行是本行的继续行。但这并不是必须的，**make** 没有对 **makefile** 文件中行的长度进行限制。一条规则可以告诉 **make** 两件事情：何时目标已经过时，以及怎样在必要时更新它们。

判断目标过时的准则和依赖关系密切，依赖也由文件名构成，文件名之间由空格隔开，通配符和档案成员也允许在依赖中出现。一个目标如果不存在或它比其中一个依赖的修改时间早，则该目标已经过时。该思想来源于目标是根据依赖的信息计算得来的，因此一旦任何一个依赖发生变化，目标文件也就不再有效。目标的更新方式由命令决定。命令由 **shell** 解释执行，但也有一些另外的特点。参阅***在规则中使用命令***。

4.2 在文件名中使用通配符

一个简单的文件名可以通过使用通配符代表许多文件。**Make** 中的通配符和 **Bourne shell** 中的通配符一样是‘*****’、‘？’和‘**[…]**’。例如：‘***.C**’指在当前目录中所有以‘**.C**’结尾的文件。

字符‘**~**’在文件名的前面也有特殊的含义。如果字符‘**~**’单独或后面跟一个斜杠‘**/**’，则代表您的 **home** 目录。如‘**~/bin**’扩展为‘**/home/bin**’。如果字符‘**~**’后面跟一个字，

它扩展为 home 目录下以该字为名字的目录，如‘~John/bin’表示‘home/John/bin’。在一些操作系统（如 ms-dos，ms-windows）中不存在 home 目录，可以通过设置环境变量 home 来模拟。

在目标、依赖和命令中的通配符自动扩展。在其它上下文中，通配符只有在您明确表明调用通配符函数时才扩展。

通配符另一个特点是如果通配符前面是反斜杠‘\’，则该通配符失去通配能力。如‘foo*bar’表示一个特定的文件其名字由‘foo’、‘*’和‘bar’构成。

4.2.1 通配符例子

通配符可以用在规则的命令中，此时通配符由 shell 扩展。例如，下面的规则删除所有 OBJ 文件：

```
clean:
 rm          -f  *.o
```

通配符在规则的依赖中也很有用。在下面的 makefile 规则中，‘make print’将打印所有从上次您打印以后又有改动的‘.c’文件：

```
print: *.c
        lpr -p $?
        touch print
```

本规则使用‘ptint’作为一个空目标文件（参看*使用空目标文件记录事件*）；自动变量‘$?’用来打印那些已经修改的文件，参看*自动变量*。

当您定义一个变量时通配符不会扩展，如果您这样写：

```
objects = *.o
```

变量 objects 的值实际就是字符串‘*.o’。然而，如果您在一个目标、依赖和命令中使用变量 objects 的值，通配符将在那时扩展。使用下面的语句可使通配符扩展：

```
objects=$(wildcard *.o)
```

详细内容参阅*函数 wildcard*。

4.2.2 使用通配符的常见错误

下面有一个幼稚使用通配符扩展的例子，但实际上该例子不能完成您所希望完成的任务。假设可执行文件‘foo’由在当前目录的所有 OBJ 文件创建，其规则如下：

```
objects = *.o

foo : $(objects)
        cc -o foo $(CFLAGS) $(objects)
```

由于变量 objects 的值为字符串‘*.o’，通配符在目标‘foo’的规则下扩展，所以每一个 OBJ 文件都会变为目标‘foo’的依赖，并在必要时重新编译自己。

但如果您已删除了所有的 OBJ 文件，情况又会怎样呢？因没有和通配符匹配的文件，所以目标‘foo’就依靠了一个有着奇怪名字的文件‘*.o’。因为目录中不存在该文件，make 将发出不能创建‘*.o’的错误信息。这可不是所要执行的任务。

实际上，使用通配符获得正确的结果是可能的，但您必须使用稍微复杂一点的技术，该技术包括使用函数 wildcard 和替代字符串等。详细内容将在下一节论述。

微软的操作系统（MS-DOS、MS-WINDOWS）使用反斜杠分离目录路径，如：

```
C:\foo\bar\bar.c
```

这和 Unix 风格‘c:/foo/bar/bar.c’等价（‘c:’是驱动器字母）。当 make 在这些系统上运行时，不但支持在路径中存在反斜杠也支持 Unix 风格的前斜杠。但是这种对反斜杠的支持不包括通配符扩展，因为通配符扩展时，反斜杠用作引用字符。所以，在这些场合您必须使用 Unix 风格的前斜杠。

4.2.3 函数 wildcard

通配符在规则中可以自动扩展，但设置在变量中或在函数的参数中通配符一般不能正常扩展。如果您需要在这些场合扩展通配符，您应该使用函数 **wildcard**，格式如下：

$(wildcard *pattern*...)

可以在 **makefile** 文件的任何地方使用该字符串，应用时该字符串被一列在指定目录下存在的并且文件名和给出的文件名的格式相符合的文件所代替，文件名中间由空格隔开。如果没有和指定格式一致的文件，则函数 **wildcard** 的输出将会省略。注意这和在规则中通配符扩展的方式不同，在规则中使用逐字扩展方式，而不是省略方式（参阅上节）。

使用函数 **wildcard** 得到指定目录下所有的 **C** 语言源程序文件名的命令格式为：

```
$(wildcard *.c)
```

我们可以把所获得的 **C** 语言源程序文件名的字符串通过将‘**.c**’后缀变为‘**.o**’转换为 **OBJ** 文件名的字符串，其格式为：

```
$(patsubst %.c,%.o,$(wildcard *.c))
```

这里我们使用了另外一个函数：**patsubst**，详细内容参阅*字符串替换和分析函数*。

这样，一个编译特定目录下所有 **C** 语言源程序并把它们连接在一起的 **makefile** 文件可以写成如下格式：

```
objects := $(patsubst %.c,%.o,$(wildcard *.c))

foo : $(objects)
        cc -o foo $(objects)
```

这里使用了编译 **C** 语言源程序的隐含规则，因此没有必要为每个文件写具体编译规则。‘**:=**’是‘**=**’的变异，对‘**:=**’的解释，参阅*两种风格的变量*。

4.3 在目录中搜寻依赖

对于大型系统，把源文件安放在一个单独的目录中，而把二进制文件放在另一个目录中是十分常见的。**Make** 的目录搜寻特性使自动在几个目录搜寻依赖十分容易。当您在几个目录中重新安排您的文件，您不必改动单独的规则，仅仅改动一下搜寻路径即可。

4.3.1 VPATH：所有依赖的搜寻路径

make 变量 **VPATH** 的值指定了 **make** 搜寻的目录。经常用到的是那些包含依赖的目录，并不是当前的目录；但 **VPATH** 指定了 **make** 对所有文件都适用的目录搜寻序列，包括了规则的目标所需要的文件。

如果一个作为目标或依赖的文件在当前目录中不存在，**make** 就会在 **VPATH** 指定的目录中搜寻该文件。如果在这些目录中找到要寻找的文件，则就象这些文件在当前目录下存在一样，规则把这些文件指定为依赖。参阅*编写搜寻目录的 shell 命令*。

在 **VPATH** 变量定义中，目录的名字由冒号或空格分开。目录列举的次序也是 **make** 搜寻的次序。在 **MS-DOS**、**MS-WINDOWS** 系统中，**VPATH** 变量定义中的目录的名字由分号分开，因为在这些系统中，冒号用为路径名的一部分（通常在驱动器字母后面）。例如：

```
VPATH = src:../headers
```

指定了两个目录，‘**src**’和‘**…/headers**’，**make** 也按照这个次序进行搜寻。使用该 **VPATH** 的值，下面的规则，

```
foo.o : foo.c
```

在执行时就象如下写法一样会被中断：

```
foo.o : src/foo.c
```

然后在 **src** 目录下搜寻 **foo.c**。

4.3.2 vpath 指令

vpath 指令（注意字母是小写）和 **VPATH** 变量类似，但却更具灵活性。**vpath** 指令允许对符合一定格式类型的文件名指定一个搜寻路径。这样您就可以对一种格式类型的文件名指定一个搜寻路径，对另外格式类型的文件名指定另外一个搜寻路径。总共由三种形式的 **vpath** 指令：

vpath *pattern directories*

对一定格式类型的文件名指定一个搜寻路径。搜寻的路径由一列要搜寻的目录构成，目录由冒号（在 MS-DOS、MS-WINDOWS 系统中用分号）或空格隔开，和 VPATH 变量定义要搜寻的路径格式一样。

vpath *pattern*

清除和一定类型格式相联系的搜寻路径。

vpath

清除所有前面由 vapth 指令指定的搜寻路径。

一个 **vpath** 的格式 **pattern** 是一个包含一个’%’的字符串。该字符串必须和正搜寻的一个依赖的文件名匹配，字符%可和任何字符串匹配（关于格式规则，参阅***定义与重新定义格式规则***）。例如，**%.h** 和任何文件名以**.h** 结尾的文件匹配。如果不使用‘%’，格式必须与依赖精确匹配，这种情况很少使用。

在 **vpath** 指令格式中的字符‘%’可以通过前面的反斜杠被引用。引用其它字符‘%’的反斜杠也可以被更多的反斜杠引用。引用字符‘%’和其它反斜杠的反斜杠在和文件名比较之前和格式是分开的。如果反斜杠所引用的字符‘%’没有错误，则该反斜杠不会运行带来任何危害。

如果 **vpath** 指令格式和一个依赖的文件名匹配，并且在当前目录中该依赖不存在，则 **vpath** 指令中指定的目录和 **VPATH** 变量中的目录一样可以被搜寻。例如：

```
vpath %.h ../headers
```

将告诉 **make** 如果在当前目录中以‘**.h**’结尾文件不存在，则在‘**../headers**’目录下搜寻任何以‘**.h**’结尾依赖。

如果有几个 **vpath** 指令格式和一个依赖的文件名匹配，则 **make** 一个接一个的处理它们，搜寻所有在指令中指定的目录。**Make** 按它们在 **makefile** 文件中出现的次序控制多个 **vpath** 指令，多个指令虽然有相同的格式，但它们是相互独立的。以下代码：

```
vpath %.c foo
vpath %   blish
vpath %.c bar
```

表示搜寻**`.c'**文件先搜寻目录**`foo'**、然后**`blish'**，最后**`bar'**；如果是如下代码：

```
vpath %.c foo:bar
vpath %   blish
```

表示搜寻**`.c'**文件先搜寻目录‘**foo'**、然后‘**bar'**，最后‘**blish'**。

4.3.3 目录搜寻过程

当通过目录搜寻找到一个文件，该文件有可能不是您在依赖列表中所列出的依赖；有时通过目录搜寻找到的路径也可能被废弃。**Make** 决定对通过目录搜寻找到的路径保存或废弃所依据的算法如下：

1、如果一个目标文件在 **makefile** 文件所在的目录下不存在，则将会执行目录搜寻。

2、如果目录搜寻成功，则路径和所得到的文件暂时作为目标文件储存。

3、所有该目标的依赖用相同的方法考察。

4、把依赖处理完成后，该目标可能需要或不需要重新创建：

1、如果该目标不需要重建，目录搜寻时所得到的文件的路径用作该目标所有依赖的路径，同时包含该目标文件。简而言之，如果 **make** 不必重建目标，则您使用通过目

录搜寻得到的路径。

2、如果该目 标需要重建，目录搜寻 时所得到的文件的 路径将废弃，目标文件在 makefile 文件所在的目录下重建。简而言之，如果make 要重建目标，是在 makefile 文件所在的目录下重建目标，而不是在目录搜寻时所得到的文件的路径下。

该算法似乎比较复杂，但它却可十分精确的解释实际您所要的东西。

其它版本的 make 使用一种比较简单的算法：如果目标文件在当前目录下不存在，而它通过目录搜寻得到，不论该目标是否需要重建，始终使用通过目录搜寻得到的路径。

实际上，如果在 GNU make 中使您的一些或全部目录具备这种行为，您可以使用 GPATH 变量来指定这些目录。

GPATH 变量和 VPATH 变量具有相同的语法和格式。如果通过目录搜寻得到一个过时的目标，而目标存在的目录又出现在 GPATH 变量，则该路径将不废弃，目标将在该路径下重建。

4.3.4 编写目录搜寻的 shell 命令

即使通过目录搜寻在其它目录下找到一个依赖，不能改变规则的命令，这些命令同样按照原来编写的方式执行。因此，您应该小心的编写这些命令，以便它们可以在 make 能够在发现依赖的目录中处理依赖。

借助诸如‘$^’的自动变量可更好的使用 shell 命令（参阅*自动变量*）。例如，‘$^’的值代表所有的依赖列表，并包含寻找依赖的目录；‘$@’的值是目标。

```
foo.o : foo.c
        cc -c $(CFLAGS) $^ -o $@
```

变量 CFLAGS 存在可以方便您利用隐含规则指定编译 C 语言源程序的旗标。我们这里使用它是为了保持编译 C 语言源程序一致性。参阅*隐含规则使用的变量*。

依赖通常情况下也包含头文件，因自动变量‘$<’的值是第一个依赖，因此这些头文件您可以不必在命令中提及，例如：

```
VPATH = src:../headers
foo.o : foo.c defs.h hack.h
        cc -c $(CFLAGS) $< -o $@
```

4.3.5 目录搜寻和隐含规则

搜寻的目录是由变量 VPATH 或隐含规则引入的 vpath 指令指定的（详细参阅*使用隐含规则*）。例如，如果文件‘foo.o’没有具体的规则，make 则使用隐含规则：如文件 foo.c 存在，make 使用内置的规则编译它；如果文件 foo.c 不在当前目录下，就搜寻适当的目录，如在别的目录下找到 foo.c，make 同样使用内置的规则编译它。

隐含规则的命令使用自动变量是必需的，所以隐含规则可以自然地使用目录搜寻得到的文件。

4.3.6 连接库的搜寻目录

对于连接库文件，目录搜寻采用一种特别的方式。这种特别的方式来源于个玩笑：您写一个依赖，它的名字是‘-|name’的形式。 您可以在这里写一些奇特的字符，因为依赖正常是一些文件名，库文件名通常是‘libname.a’ 的形式，而不是‘-|name’ 的形式。）

当一个依赖的名字是‘-|name’的形式时，make 特别地在当前目录下、与 vpath 匹配的目录下、VPATH 指定的目录下以及‘/lib’，‘/usr/lib'，和 ‘*prefix*/lib'（正常情况为`/usr/local/lib'，但是 MS-DOS、MS-Windows 版本的 make 的行为好像是 *prefix* 定义为 DJGPP 安装树的根目录的情况）目录下搜寻名字为‘lib*name*.so'的文件然后再处理它。

如果没有搜寻到‘**lib*name*.so**'文件，然后在前述的目录下搜寻‘**lib*name*.a**'文件。

例如，如果在您的系统中有‘**/usr/lib/libcurses.a**'的库文件，则：

```
foo : foo.c -lcurses
        cc $^ -o $@
```

如果‘**foo**’比‘**foo.c**’更旧，将导致命令‘**cc foo.c /usr/lib/libcurses.a -o foo**'执行。

缺省情况下是搜寻‘**lib*name*.so**' 和‘**lib*name*.a**'文件，具体搜寻的文件及其类型可使用**.LIBPATTERNS** 变量指定，这个变量值中的每一个字都是一个字符串格式。当寻找名为‘**-|name**’的依赖时，**make** 首先用 ***name*** 替代列表中第一个字中的格式部分形成要搜寻的库文件名，然后使用该库文件名在上述的目录中搜寻。如果没有发现库文件，则使用列表中的下一个字，其余以此类推。

.LIBPATTERNS 变量缺省的值是**"‘lib%.so lib%.a'"**，该值对前面描述的缺省行为提供支持。您可以通过将该值设为空值从而彻底关闭对连接库的扩展。

4.4 假想目标

假想目标并不是一个真正的文件名，它仅仅是您制定的一个具体规则所执行的一些命令的名称。使用假想目标有两个原因：避免和具有相同名称的文件冲突和改善性能。

如果您写一个其命令不创建目标文件的规则，一旦由于重建而提及该目标，则该规则的命令就会执行。这里有一个例子：

```
clean:
        rm *.o temp
```

因为 **rm** 命令不创建名为‘**clean**’的文件，所以不应有名为‘**clean**’的文件存在。因此不论何时您发布`**make clean**'指令，**rm** 命令就会执行。

假想目标能够终止任何在目录下创建名为‘**clean**’的文件工作。但如在目录下存在文件 **clean**，因为该目标 **clean** 没有依赖，所以文件 **clean** 始终会认为已经该更新，因此它的命令将永不会执行。为了避免这种情况，您应该使用象如下特别的**.PHONY** 目标格式将该目标具体的声明为一个假想目标：

```
.PHONY : clean
```

一旦这样声明，‘**make clean**’命令无论目录下是否存在名为‘**clean**’的文件，该目标的命令都会执行。

因为 **make** 知道假想目标不是一个需要根据别的文件重新创建的实际文件，所以它将跳过隐含规则搜寻假想目标的步骤（详细内容参阅***使用隐含规则***）。这是把一个目标声明为假想目标可以提高执行效率的原因，因此使用假想目标您不用担心在目录下是否有实际文件存在。这样，对前面的例子可以用假想目标的写出，其格式如下：

```
.PHONY: clean
clean:
        rm *.o temp
```

另外一个使用假想目标的例子是使用 **make** 的递归调用进行连接的情况：此时，**makefile** 文件常常包含列举一系列需要创建的子目录的变量。不用假想目标完成这种任务的方法是使用一条规则，其命令是一个在各个子目录下循环的 **shell** 命令，如下面的例子：

```
subdirs:
        for dir in $(SUBDIRS); do \
          $(MAKE) -C $$dir; \
        done
```

但使用这个方法存在下述问题：首先，这个规则在创建子目录时产生的任何错误都不及时发现，因此，当一个子目录创建失败时，该规则仍然会继续创建剩余的子目录。虽然该问题可以添加监视错误产生并退出的 **shell** 命令来解决，但非常不幸的是如果 **make** 使用了‘**-k**’选项，这个问题仍然会产生。第二，也许更重要的是您使用了该方法就失去使用 **make** 并行处理的特点能力。

使用假想目标（如果一些子目录已经存在，您则必须这样做，否则，不存在的子目录将

不会创建）则可以避免上述问题：

```
SUBDIRS = foo bar baz

.PHONY: subdirs $(SUBDIRS)

subdirs: $(SUBDIRS)

$(SUBDIRS):
        $(MAKE) -C $

foo: baz
```

此时，如果子目录‘**baz**’没有创建完成，子目录’**foo**’将不会创建；当试图使用并行创建时这种关系的声明尤其重要。

一个假想目标不应该是一个实际目标文件的依赖，如果这样，**make** 每次执行该规则的命令，目标文件都要更新。只要假想目标不是一个真实目标的依赖，假想目标的命令只有在假想目标作为特别目标时才会执行（参阅***指定最终目标的参数***）。

假想目标也可以有依赖。当一个目录下包含多个程序时，使用假想目标可以方便的在一个 **makefile** 文件中描述多个程序的更新。重建的最终目标缺省情况下是 **makefile** 文件的第一个规则的目标，但将多个程序作为假想目标的依赖则可以轻松的完成在一个 **makefile** 文件中描述多个程序的更新。如下例：

```
all : prog1 prog2 prog3
.PHONY : all

prog1 : prog1.o utils.o
        cc -o prog1 prog1.o utils.o

prog2 : prog2.o
        cc -o prog2 prog2.o

prog3 : prog3.o sort.o utils.o
        cc -o prog3 prog3.o sort.o utils.o
```

这样，您可以重建所有程序，也可以参数的形式重建其中的一个或多个（如‘**make prog1 prog3'**）。

当一个假想目标是另一个假想目标的依赖，则该假想目标将作为一个假想目标的子例程。例如，这里‘**make cleanall'**用来删除 **OBJ** 文件、**diff** 文件和程序文件：

```
.PHONY: cleanall cleanobj cleandiff

cleanall : cleanobj cleandiff
        rm program

cleanobj :
        rm *.o

cleandiff :
        rm *.diff
```

4.5 没有命令或依赖的规则

如果一个规则没有依赖、也没有命令，而且这个规则的目标也不是一个存在的文件，则 **make** 认为只要该规则运行，该目标就已被更新。这意味着，所有以这种规则的目标为依赖的目标，它们的命令将总被执行。这里举一个例子：

```
clean: FORCE
```

```
        rm $(objects)
FORCE:
```

这里的目标‘**FORCR**’满足上面的特殊条件，所以以其为依赖的目标‘**clean**’将总强制它的命令执行。关于‘**FORCR**’的名字没有特别的要求，但‘**FORCR**’是习惯使用的名字。

也许您已经明白，使用‘**FORCR**’的方法和使用假想目标（**.PHONY: clean**）的结果一样，但使用假想目标更具体更灵活有效，由于一些别的版本的 **make** 不支持假想目标，所以‘**FORCR**’出现在许多 **makefile** 文件中。参阅***假想目标***。

4.6 使用空目标文件记录事件

空目标是一个假想目标变量，它用来控制一些命令的执行，这些命令可用来完成一些经常需要的具体任务。但又不象真正的假想目标，它的目标文件可以实际存在，但文件的内容与此无关，通常情况下，这些文件没有内容。

空目标文件的用途是用来记录规则的命令最后一次执行的时间，也是空目标文件最后更改的时间。它之所以能够这样执行是因为规则的命令中有一条用于更新目标文件的‘**touch**’命令。另外，空目标文件应有一些依赖（否则空目标文件没有存在的意义）。如果空目标比它的依赖旧，当您命令重建空目标文件时，有关的命令才会执行。下面有一个例子：

```
print: foo.c bar.c
        lpr -p $?
        touch print
```

按照这个规则，如果任何一个源文件从上次执行‘**make print**'以来发生变化，键入‘**make print**'则执行 **lpr** 命令。自动变量‘**$?**’用来打印那些发生变化的文件（参阅***自动变量***）。

4.7 内建的特殊目标名

一些名字作为目标使用则含有特殊的意义：

- **.PHONY**

特殊目标**.PHONY** 的依赖是假想目标。假想目标是这样一些目标，**make** 无条件的执行它命令，和目录下是否存在该文件以及它最后一次更新的时间没有关系。详细内容参阅***假想目标***。

- **.SUFFIXES**

特殊目标**.SUFFIXES** 的依赖是一列用于后缀规则检查的后缀。详细内容参阅***过时的后缀规则***。

- **.DEFAULT**

.DEFAULT 指定一些命令，这些命令用于那些没有找到规则（具体规则或隐含规则）更新的目标。详细内容参阅***定义最新类型的-缺省规则***。如果**.DEFAULT** 指定了一些命令，则所有提及到的文件只能作为依赖，而不能作为任何规则的目标；这些指定的命令也只按照他们自己的方式执行。详细内容参阅***隐含规则搜寻算法***。

- **.PRECIOUS**

特殊目标**.PRECIOUS** 的依赖将按照下面给定的特殊方式进行处理：如果在执行这些目标的命令的过程中，**make** 被关闭或中断，这些目标不能被删除，详细内容参阅***关闭和中断 make***；如果目标是中间文件，即使它已经没有任何用途也不能被删除，具体情况和该目标正常完成一样，参阅***隐含规则链***；该目标的其它功能和特殊目标**.SECONDARY** 的功能重叠。如果规则的目标格式与依赖的文件名匹配，您可以使用隐含规则的格式（如‘**%.o**’）列举目标作为特殊目标**.PRECIOUS** 的依赖文件来保存由这些规则创建的中间文件。

- **.INTERMEDIATE**

特殊目标**.INTERMEDIATE** 的依赖被处理为中间文件。详细内容参见***隐含规则链***。**.INTERMEDIATE** 如果没有依赖文件，它将不会发生作用。

- **.SECONDARY**

特殊目标**.SECONDARY** 的依赖被处理为中间文件，但它们永远不能自动删除。详细内容参见***隐含规则链***。**.SECONDARY** 如果没有依赖文件，则所有的 **makefile** 文件中的目标都将被处理为中间文件。

- **.DELETE_ON_ERROR**

  如果在 **makefile** 文件的某处**.DELETE_ON_ERROR** 作为一个目标被提及，则如果该规则发生变化或它的命令没有正确完成而退出，**make** 将会删除该规则的目标，具体行为和它受到了删除信号一样。详细内容参阅***命令错误***。

- **.IGNORE**

  如果您特别为目标**.IGNORE** 指明依赖，则 **MAKE** 将会忽略处理这些依赖文件时执行命令产生的错误。如果**.IGNORE** 作为一个没有依赖的目标提出来，**MAKE** 将忽略处理所有文件时产生的错误。**.IGNORE** 命令并没有特别的含义，**.IGNORE** 的用途仅是为了和早期版本的兼容。因为**.IGNORE** 影响所有的命令，所以它的用途不大；我们推荐您使用其它方法来忽略特定命令产生的错误。详细内容参阅***命令错误***。

- **.SILENT**

  如果您特别为**.SILENT** 指明依赖，则在执行之前 **MAKE** 将不会回显重新构造文件的命令。如果 **.SILENT** 作为一个没有依赖的目标提出来，任何命令在执行之前都不会打印。**.SILENT** 并没有特别的含义，其用途仅是为了和早期版本的兼容。我们推荐您使用其它方法来处理那些不打印的命令。详细内容参阅***命令回显***。如果您希望所有的命令都不打印，请使用‘**-s**’或‘**-silent**’选项**(**详细参阅***选项概要*****)**。

- **.EXPORT_ALL_VARIABLES**

  如该特殊目标简单的作为一个目标被提及，**MAKE** 将缺省地把所有变量都传递到子进程中。参阅使**与子 MAKE 通信的变量**。

- **.NOTPARALLEL**

  如果**.NOTPARALLEL** 作为一个目标提及，即使给出‘**-j**’选项，**make** 也不使用并行执行。但递归调用的 **make** 命令仍可并行执行（在调用的 **makefile** 文件中包含**.NOTPARALLEL** 的目标的例外）。**.NOTPARALLEL** 的任何依赖都将忽略。

任何定义的隐含规则后缀如果作为目标出现都会视为一个特殊规则，即使两个后缀串联起来也是如此，例如‘**.c.o**’。这些目标称为后缀规则，这种定义方法是过时的定义隐含规则的方法（目前仍然广泛使用的方法）。原则上，如果您要把它分为两个并把它们加到后缀列表中，任何目标名都可采用这种方法指定。实际上，后缀一般以‘**.**’开始，因此，这些特别的目标同样以‘**.**’开始。具体参阅***过时的后缀规则***。

4.8 具有多个目标的规则

具有多个目标的规则等同于写多条规则，这些规则除了目标不同之外，其余部分完全相同。相同的命令应用于所有目标，但命令执行的结果可能有所差异，因此您可以在命令中使用‘**$@**’分配不同的实际目标名称。这条规则同样意味着所有的目标有相同的依赖。

在以下两种情况下具有多个目标的规则相当有用：

- 您仅仅需要依赖，但不需要任何命令。例如：

  ```
  kbd.o command.o files.o: command.h
  ```

  为三个提及的目标文件给出附加的共同依赖。

- 所有的目标使用相同的命令。但命令的执行结果未必完全相同，因为自动变量‘**$@**’可以在重建时指定目标（参阅自动变量）。例如：

  ```
  bigoutput littleoutput : text.g
          generate text.g -$(subst output,,$@) > $@
  ```

  等同于：

  ```
  bigoutput : text.g
          generate text.g -big > bigoutput
  littleoutput : text.g
          generate text.g -little > littleoutput
  ```

  这里我们假设程序可以产生两种输出文件类型：一种给出‘**-big**’，另一种给出

‘-little’。参阅*字符串代替和分析函数*，对函数 subst 的解释。

如果您喜欢根据目标变换依赖，象使用变量‘$@’变换命令一样。您不必使用具有多个目标的规则，您可以使用*静态格式规则*。详细内容见下文。

4.9 具有多条规则的目标

一个目标文件可以有多个规则。在所有规则中提及的依赖都将融合在一个该目标的依赖列表中。如果该目标比任何一个依赖‘旧’，所有的命令将执行重建该目标。

但如果一条以上的规则对同一文件给出多条命令，make 将使用最后给出的规则，同时打印错误信息。（作为特例，如果文件名以点‘.’开始，不打印出错信息。这种古怪的行为仅仅是为了和其它版本的 make 兼容）您没有必要这样编写您的 makefile 文件，这正是 make 给您发出错误信息的原因。

一条特别的依赖规则可以用来立即给多条目标文件提供一些额外的依赖。例如，使用名为‘objects’的变量，该变量包含系统产生的所有输出文件列表。如果‘congfig.h’发生变化所有的输出文件必须重新编译，可以采用下列简单的方法编写：

```
objects = foo.o bar.o
foo.o : defs.h
bar.o : defs.h test.h
$(objects) : config.h
```

这些可以自由插入或取出而不影响实际指定的目标文件生成规则，如果您希望断断续续的为目标添加依赖，这是非常方便的方法。

另外一个添加依赖的方法是定义一个变量，并将该变量作为 make 命令的参数使用。详细内容参阅*变量重载*。例如：

```
extradeps=
$(objects) : $(extradeps)
```

命令`make extradeps=foo.h'含义是将‘foo.h’作为所有 OBJ 文件的依赖，如果仅仅输入‘make’命令则不是这样。

如果没有具体的规则为目标的生成指定命令，那么 make 将搜寻合适的隐含规则进而确定一些命令来完成生成或重建目标。详细内容参阅*使用隐含规则*。

4.10 静态格式规则

静态格式规则是指定多个目标并能够根据每个目标名构造对应的依赖名的规则。静态格式规则在用于多个目标时比平常的规则更常用，因为目标可以不必有完全相同的依赖；也就是说，这些目标的依赖必须类似，但不必完全相同。

4.10.1 静态格式规则的语法

这里是静态格式规则的语法格式：

```
targets ...: target-pattern: dep-patterns ...
        commands
        ...
```

目标列表指明该规则应用的目标。目标可以含有通配符，具体使用和平常的目标规则基本一样（参阅*在文件名中使用通配符*）。

目标的格式和依赖的格式是说明如何计算每个目标依赖的方法。从匹配目标格式的目标名中依据格式抽取部分字符串，这部分字符串称为径。将径分配到每一个依赖格式中产生依赖名。

每一个格式通常包含字符‘%’。目标格式匹配目标时，‘%’可以匹配目标名中的任何字符串；这部分匹配的字符串称为径；剩下的部分必须完全相同。如目标‘foo.o’匹配格式

‘%.o’，字符串‘foo’称为径。而目标‘foo.c’和‘foo.out’不匹配格式。

每个目标的依赖名是使用径代替各个依赖中的‘%’产生。如，如果个依赖格式为‘%.c’，把径‘foo’代替依赖格式中的‘%’生成依赖的文件名‘foo.c’。在依赖格式中不包含‘%’也是合法的，此时对所有目标来说，依赖是相同的。

在格式规则中字符‘%’可以用前面加反斜杠‘\’方法引用。引用‘%’的反斜杠也可以由更多的反斜杠引用。引用‘%’、‘\’的反斜杠在和文件名比较或由径代替它之前从格式中移走。反斜杠不会因为引用‘%’而混乱。如格式`the\%weird\\%pattern\\'是`the%weird\'加上字符‘%'，后面再和字符串 ‘pattern\\'连接。最后的两个反斜杠由于不能影响任何统配符‘%’所以保持不变。

这里有一个例子，它将对应的‘.c’文件编译成‘foo.o’和‘bar.o’。

```
objects = foo.o bar.o

all: $(objects)

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

这里‘$<’是自动变量，控制依赖的名称，‘$@’也是自动变量，掌握目标的名称。详细内容参阅*自动变量*。

每一个指定目标必须和目标格式匹配，如果不符则产生警告。如果您有一列文件，仅有其中的一部分和格式匹配，您可以使用 **filter** 函数把不符合的文件移走（参阅*字符串替代和分析函数*）：

```
files = foo.elc bar.o lose.o

$(filter %.o,$(files)): %.o: %.c
        $(CC) -c $(CFLAGS) $< -o $@
$(filter %.elc,$(files)): %.elc: %.el
        emacs -f batch-byte-compile $<
```

在这个例子中，‘$(filter %.o,$(files))'的结果是‘bar.o lose.o'，第一个静态格式规则是将相应的 C 语言源文件编译更新为 OBJ 文件，‘$(filter %.elc,$(files))'的结果是‘foo.elc'，它由‘foo.el’构造。

另一个例子是阐明怎样在静态格式规则中使用‘$*’：

```
bigoutput littleoutput : %output : text.g
        generate text.g -$* > $@
```

当命令 generate 执行时，$*扩展为径，即‘big’或‘little’二者之一。

4.10.2 静态格式规则和隐含规则

静态格式规则和定义为格式规则的隐含规则有很多相同的地方（详细参阅*定义与重新定义格式规则*）。双方都有对目标的格式和构造依赖名称的格式，差异是 make 使用它们的时机不同。

隐含规则可以应用于任何于它匹配的目标，但它仅仅是在目标没有具体规则指定命令以及依赖可以被搜寻到的情况下应用。如果有多条隐含规则适合，仅有执行其中一条规则，选择依据隐含规则的定义次序。

相反，静态格式规则用于在规则中指明的目标。它不能应用于其它任何目标，并且它的使用方式对于各个目标是固定不变的。如果使用两个带有命令的规则发生冲突，则是错误。

静态格式规则因为如下原因可能比隐含规则更好：

- 对一些文件名不能按句法分类的但可以给出列表的文件，使用静态格式规则可以重载隐含规则链。
- 如果不能精确确定使用的路径，您不能确定一些无关紧要的文件是否导致 make 使用错误的隐含规则（因为隐含规则的选择根据其定义次序）。使用静态格式规则则没有这些

不确定因素：每一条规则都精确的用于指定的目标上。

4.11 双冒号规则

双冒号规则是在目标名后使用‘：：’代替‘：’的规则。当同一个目标在一条以上的规则中出现时，双冒号规则和平常的规则处理有所差异。

当一目标在多条规则中出现时，所有的规则必须是同一类型：要么都是双冒号规则，要么都是普通规则。如果他们都是双冒号规则，则它们之间都是相互独立的。如果目标比一个双冒号规则的依赖‘旧’，则该双冒号规则的命令将执行。这可导致具有同一目标双冒号规则全部或部分执行。

双冒号规则实际就是将具有相同目标的多条规则相互分离，每一条双冒号规则都独立的运行，就像这些规则的目标不同一样。

对于一个目标的双冒号规则按照它们在 **makefile** 文件中出现的顺序执行。然而双冒号规则真正有意义的场合是双冒号规则和执行顺序无关的场合。

双冒号规则有点模糊难以理解，它仅仅提供了一种在特定情况下根据引起更新的依赖文件不同，而采用不同方式更新目标的机制。实际应用双冒号规则的情况非常罕见。

每一个双冒号规则都应该指定命令，如果没有指定命令，则会使用隐含规则。详细内容参阅*使用隐含规则*。

4.12 自动生成依赖

在为一个程序编写的 **makefile** 文件中，常常需要写许多仅仅是说明一些 **OBJ** 文件依靠头文件的规则。例如，如果‘**main.c**’通过一条**#include** 语句使用‘**defs.h**’，您需要写入下的规则：

```
main.o: defs.h
```

您需要这条规则让 **make** 知道如果‘**defs.h**’一旦改变必须重新构造‘**main.o**’。由此您可以明白对于一个较大的程序您需要在 **makefile** 文件中写很多这样的规则。而且一旦添加或去掉一条**#include** 语句您必须十分小心地更改 **makefile** 文件。

为避免这种烦恼，现代**C** 编译器根据原程序中的**#include** 语句可以为您编写这些规则。如果需要使用这种功能，通常可在编译源程序时加入‘**-M**’开关，例如，下面的命令：

```
cc -M main.c
```

产生如下输出：

```
main.o : main.c defs.h
```

这样您就不必再亲自写这些规则，编译器可以为您完成这些工作。

注意，由于在 **makefile** 文件中提及构造‘**main.o**’，因此‘**main.o**’将永远不会被隐含规则认为是中间文件而进行搜寻，这同时意味着 **make** 不会在使用它之后自动删除它；参阅*隐含规则链*。

对于旧版的 **make** 程序，通过一个请求命令，如‘**make depend**’，利用编译器的特点生成依赖是传统的习惯。这些命令将产生一个‘**depend**’文件，该文件包含所有自动生成的依赖；然后 **makefile** 文件可以使用 **include** 命令将它们读入（参阅*包含其它 makefile 文件*）。

在 **GNU make**中，重新构造 **makefile** 文件的特点使这个惯例成为了过时的东西——您永远不必具体告诉 **make** 重新生成依赖，因为 **GNU make** 总是重新构造任何过时的 **makefile** 文件。参阅 ***Makefile* 文件的重新生成的过程**。

我们推荐使用自动生成依赖的习惯是把 **makefile** 文件和源程序文件一一对应起来。如，对每一个源程序文件‘**name.c**’有一名为‘**name.d**’的 **makefile** 文件和它对应，该**makefile** 文件中列出了名为‘**name.o**’的 **OBJ** 文件所依赖的文件。这种方式的优点是仅在源程序文件改变的情况下才有必要重新扫描生成新的依赖。

这里有一个根据 **C** 语言源程序‘**name.c**’生成名为‘**name.d**’依赖文件的格式规则：

```
%.d: %.c
        set -e; $(CC) -M $(CPPFLAGS) $< \
```

```
                | sed 's/\($*\)\.o[ :]*/\1.o $@ : /g' > $@; \
              [ -s $@ ] || rm -f $@
```

关于定义格式规则的信息参阅*定义与重新定义格式规则*。‘**-e**’开关是告诉 **shell** 如果**$(CC)** 命令运行失败（非零状态退出）立即退出。正常情况下，**shell** 退出时带有最后一个命令在管道中的状态（**sed**）,因此 **make** 不能注意到编译器产生的非零状态。

对于 **GNU C**编译器您可以使用‘**-MM**’开关代替‘**-M**’，这是省略了有关系统头文件的依赖。详细内容参阅《**GNU CC** 使用手册》中*控制预处理选项*。

命令 **Sed** 的作用是翻译（例如）：

```
main.o : main.c defs.h
```

到：

```
main.o main.d : main.c defs.h
```

这使每一个‘**.d**’文件和与之对应的‘**.o**’文件依靠相同的源程序文件和头文件，据此，**Make** 可以知道如果任一个源程序文件和头文件发生变化，则必须重新构造依赖文件。

一旦您定义了重新构造‘**.d**’文件的规则，您可以使用使用 **include** 命令直接将它们读入， 参阅*包含其它 **makefile** 文件*），例如：

```
sources = foo.c bar.c
include $(sources:.c=.d)
```

（这个例子中使用一个代替变量参照从源程序文件列表‘**foo.c bar.c'**翻译到依赖文件列表‘**foo.d bar.d'**。详细内容参阅*替换引用*。所以，‘**.d**’的 **makefile** 文件和其它 **makefile** 文件一样，即使没用您的任何进一步的指令，**make** 同样会在必要的时候重新构建它们。参阅 ***Makefile** 文件的重新生成过程*。

5 在规则中使用命令

规则中的命令由一系列 **shell** 命令行组成，它们一条一条的按顺序执行。除第一条命令行可以分号为开始附属在目标-依赖行后面外，所有的命令行必须以 **TAB** 开始。空白行与注释行可在命令行中间出现，处理时它们被忽略。（但是必须注意，以 **TAB** 开始的‘空白行’不是空白行，它是空命令，参阅*使用空命令*。）

用户使用多种不同的 **shell** 程序，如果在 **makefile** 文件中没有指明其它的 **shell**，则使用缺省的‘**/bin/sh**’解释 **makefile** 文件中的命令。参阅*命令执行*。

使用的 **shell** 种类决定了是否能够在命令行上写注释以及编写注释使用的语法。当使用‘**/bin/sh**’作为 **shell**，以‘**#**’开始的注释一直延伸到该行结束。‘**#**’不必在行首，而且‘**#**’不是注释的一部分。

5.1 命令回显

正常情况下 **make** 在执行命令之前首先打印命令行，我们因这样可将您编写的命令原样输出故称此为回显。

以‘**@**’起始的行不能回显，‘**@**’在传输给 **shell** 时被丢弃。典型的情况，您可以在 **makefile** 文件中使用一个仅仅用于打印某些内容的命令，如**echo** 命令来显示 **makefile** 文件执行的进程：

```
@echo About to make distribution files
```

当使用 **make** 时给出‘**-n**’或‘**--just-print**’标志，则仅仅回显命令而不执行命令。参阅*选项概要*。在这种情况下也只有在这种情况下，所有的命令行都回显，即使以‘**@**’开始的命令行也回显。这个标志对于在不使用命令的情况下发现 **make** 认为哪些是必要的命令

非常有用。

‘-s’或‘--silent’标志可以使 make 阻止所有命令回显，好像所有的行都以‘@’开始一样。在makefile 文件中使用不带依赖的特别目标‘.SILENT’的规则可以达到相同的效果（参阅*内建的特殊目标名*）。因为‘@’使用更加灵活以至于现在已基本不再使用特别目标.SILENT。

5.2 执行命令

需要执行命令更新目标时，每一命令行都会使用一个独立的子 shell 环境，保证该命令行得到执行。 实际上，make 可能走不影响结果的捷径。）

请注意：这意味着设置局部变量的 shell 命令如 cd 等将不影响紧跟着的命令行；如果您需要使用 cd 命令影响到下一个命令，请把这两个命令放到一行，它们中间用分号隔开，这样 make 将认为它们是一个单一的命令行，把它们放到一起传递给 shell，然后按顺序执行它们。例如：

```
foo : bar/lose
        cd bar; gobble lose > ../foo
```

如果您喜欢将一个单一的命令分割成多个文本行，您必须用反斜杠作为每一行的结束，最后一行除外。这样，多个文本行通过删除反斜杠按顺序组成一新行，然后将它传递给 shell。如此，下面的例子和前面的例子是等同的：

```
foo : bar/lose
        cd bar;  \
        gobble lose > ../foo
```

用作 shell 的程序是由变量 SHELL 指定，缺省情况下，使用程序‘/bin/sh’作为 shell。

在 MS_DOS 上运行，如果变量 SHELL 没有指定，变量 COMSPEC 的值用来代替指定 shell。

在 MS_DOS 上运行和在其它系统上运行，对于makefile 文件中设置变量 SHELL 的行的处理也不一样。因为 MS_DOS 的 shell，‘command.com’，功能十分有限，所以许多 make 用户倾向于安装一个代替的 shell。因此，在 MS_DOS 上运行，make 检测变量 SHELL 的值，并根据它指定的 Unix 风格或 DOS 风格的 shell 变化它的行为。即使使用变量 SHELL 指向‘command.com’ ，make 依然检测变量 SHELL 的值。

如果变量 SHELL 指定 Unix 风格的 shell，在 MS_DOS 上运行的 make 将附加检查指定的 shell 是否能真正找到；如果不能找到，则忽略指定的 shell。在 MS_DOS 上，GNU make 按照下述步骤搜寻 shell:

1、在变量 SHELL 指定的目录中。例如，如果 makefile 指明`SHELL = /bin/sh'，make 将在当前路径下寻找子目录‘/bin’。

2、在当前路径下。

3、按顺序搜寻变量 PATH 指定的目录。

在所有搜寻的目录中，make 首先寻找指定的文件（如例子中的‘sh’）。如果该文件没有存在，make 将在上述目录中搜寻带有确定的可执行文件扩展的文件。例如：‘.exe’，‘.com’，‘.bat’，‘.btm’，‘.sh'文件和其它文件等。

如果上述过程中能够成功搜寻一个 shell，则变量 SHELL 的值将设置为所发现 shell 的全路径文件名。然而如果上述努力全部失败，变量 SHELL 的值将不改变，设置 shell 的行的有效性将被忽略。这是在 make 运行的系统中如果确实安装了 Unix 风格的 shell，make 仅支持指明的 Unix 风格 shell 特点的原因。

注意这种对 shell 的扩展搜寻仅仅限制在 makefile 文件中设置变量 SHELL 的情况。如果在环境或命令行中设置，希望您指定 shell 的全路径文件名，而且全路径文件名需象在 Unix 系统中运行的一样准确无误。

经过上述的 DOS 特色的处理，而且您还把 ‘sh.exe’安装在变量 PATH 指定的目录中，或在 makefile 文件内部设置`SHELL = /bin/sh'（和多数 Unix 的 makefile 文件一样），则在 MS_DOS 上的运行效果和在 Unix 上运行完全一样。

不像其它大多数变量，变量 SHELL 从不根据环境设置。这是因为环境变量 SHELL 是用来指定您自己选择交互使用的 shell 程序。如果变量 SHELL 在环境中设置，它将影响 makefile

文件的功能，这是非常不划算的，参阅*环境变量*。然而在 MS-DOS 和 MS-WINDOWS 中在环境中设置变量 SHELL 的值是要使用的，因为在这些系统中，绝大多数用户并不设置该变量的值，所以 make 很可能特意为该变量指定要使用的值。在MS-DOS 上，如果变量 SHELL 的设置对于 make 不合适，您可以设置变量 MAKESHELL 用来指定 make 使用的 shell；这种设置将使变量 SHELL 的值失效。

5.3 并行执行

GNU make 可以同时执行几条命令。正常情况下，make 一次执行一个命令，待它完成后在执行下一条命令。然而，使用‘-j’和‘--jobs’选项将告诉 make 同时执行多条命令。在 MS-DOS 上，‘-j’选项没有作用，因为该系统不支持多进程处理。

如果‘-j’选项后面跟一个整数，该整数表示一次执行的命令的条数；这称为 job slots 数。如果‘-j’选项后面没有整数，也就是没有对 job slots的数目限制。缺省的 job slots 数是一，这意味着按顺序执行（一次执行一条命令）。同时执行多条命令的一个不太理想的结果是每条命令产生的输出与每条命令发送的时间对应，即命令产生的消息回显可能较为混乱。

另一个问题是两个进程不能使用同一设备输入，所以必须确定一次只能有一条命令从终端输入，make 只能保证正在运行的命令的标准输入流有效，其它的标准输入流将失效。这意味着如果有几个同时从标准输入设备输入的话，对于绝大多数子进程将产生致命的错误（即产生一个‘Broken pipe’信号）。

命令对一个有效的标准输入流（它从终端输入或您为 make 改造的标准输入设备输入）的需求是不可预测的。第一条运行的命令总是第一个得到标准输入流，在完成一条命令后第一条启动的另一条命令将得到下一个标准输入流，等等。

如果我们找到一个更好替换方案，我们将改变 make 的这种工作方式。在此期间，如果您使用并行处理的特点，您不应该使用任何需要标准输入的命令。如果您不使用该特点，任何需要标准输入的命令将都能正常工作。

最后，make 的递归调用也导致出现问题。更详细的内容参阅*与子 make 通讯的选项*。

如果一个命令失败（被一个信号中止，或非零退出），且该条命令产生的错误不能忽略（参阅*命令错误*），剩余的构建同一目标的命令行将会停止工作。如果一条命令失败，而且‘-k’或‘--keep-going’选项也没有给出（参阅*选项概要*），make 将放弃继续执行。如果 make 由于某种原因（包括信号）要中止，此时又子进程正在运行，它将等到这些子进程结束之后再实际退出。

当系统正满负荷运行时，您或许希望在负荷轻的时再添加任务。这时，您可以使用‘-|’选项告诉 make 根据平均负荷限制同一时刻运行的任务数量。‘-|’或‘--max-load’选项一般后跟一个浮点数。例如：

```
-| 2.5
```

将不允许 make 在平均负荷高于 2.5 时启动一项任务。‘-|’选项如果没有跟数据，则取消前面‘-|’给定的负荷限制。

更精确地讲，当 make 启动一项任务时，而它此时已经有至少一项任务正在运行，则它将检查当前的平均负荷；如果不低于‘-|’选项给定的负荷限制时，make 将等待直到平均负荷低于限制或所有其它任务完成后再启动其它任务。

缺省情况下没有负荷限制。

5.4 命令错误

在每一个 shell 命令返回后，make 检查该命令退出的状态。如果该命令成功地完成，下一个命令行就会在新的子 shell 环境中执行，当最后一个命令行完成后，这条规则也宣告完成。如果出现错误（非零退出状态），make 将放弃当前的规则，也许是所有的规则。

有时一个特定的命令失败并不是出现了问题。例如：您可能使用 mkdir 命令创建一个目录存在，如果该目录已经存在，mkdir 将报告错误，但您此时也许要 make 继续执行。

要忽略一个命令执行产生的错误，请使用字符‘-’（在初始化 TAB 的后面）作为该命令行的开始。字符‘-’在命令传递给 shell 执行时丢弃。例如：

```
clean:
        -rm -f *.o
```

这条命令即使在不能删除一个文件时也强制 rm 继续执行。

在运行 make 时使用‘-i’或‘--ignore-errors’选项，将会忽略所有规则的命令运行产生的错误。在 makefile 文件中使用如果没有依赖的特殊目标.IGNORE 规则，也具有同样的效果。但因为使用字符‘-’更灵活，所以该条规则已经很少使用。

一旦使用‘-’或‘-i’选项，运行命令时产生的错误被忽略，此时 make 象处理成功运行的命令一样处理具有返回错误的命令，唯一不同的地方是打印一条消息，告诉您命令退出时的编码状态，并说明该错误已经被忽略。如果发生错误而 make 并不说明其被忽略，则暗示当前的目标不能成功重新构造，并且和它直接相关或间接相关的目标同样不能重建。因为前一个过程没有完成，所以不会进一步执行别的命令。

在上述情况下，make 一般立即放弃任务，返回一个非零的状态。然而，如果指定‘-k’或‘--keep-goning’选项，make 则继续考虑这个目标的其它依赖，如果有必要在 make 放弃返回非零状态之前重建它们。例如，在编译一个 OBJ 文件发生错误后，即使 make 已经知道将所有 OBJ 文件连接在一起是不可能的，‘make -k’选项也继续编译其它 OBJ 文件。详细内容参阅*选项概要*。通常情况下，make 的行为基于假设您的目的是更新指定的目标，一旦 make 得知这是不可能的，它将立即报告失败。‘-k’选项是告诉 make 真正的目的是测试程序中所有变化的可行性，或许是寻找几个独立的问题以便您可以在下次编译之前纠正它们。这是 Emacs 编译命令缺省情况下传递‘-k’选项的原因。

通常情况下，当一个命令运行失败时，如果它已经改变了目标文件，则该文件很可能发生混乱而不能使用或该文件至少没有完全得到更新。但是，文件的时间戳却表明该文件已经更新到最新，因此在 make 下次运行时，它将不再更新该文件。这种状况和命令被发出的信号强行关闭一样，参阅*中断或关闭 make*。因此，如果在开始改变目标文件后命令出错，一般应该删除目标文件。如果.DELETE_ON_ERROR 作为目标在 makefile 文件中出现，make 将自动做这些事情。这是您应该明确要求 make 执行的动作，不是以前的惯例；特别考虑到兼容性问题时，您更应明确提出这样的要求。

5.5 中断或关闭 make

如果 make 在一条命令运行时得到一个致命的信号， 则 make 将根据第一次检查的时间戳和最后更改的时间戳是否发生变化决定它是否删除该命令要更新的目标文件。

删除目标文件的目的是当 make 下次运行时确保目标文件从原文件得到更新。为什么？假设正在编译文件时您键入 Ctrl-c，而且这时已经开始写 OBJ 文件‘foo.o’，Ctrl-c 关闭了该编译器，结果得到不完整的 OBJ 文件‘foo.o’的时间戳比源程序‘foo.c’的时间戳新，如果 make 收到 Ctrl-c 的信号而没有删除 OBJ 文件‘foo.o’，下次请求 make 更新 OBJ 文件‘foo.o’时，make 将认为该文件已更新到最新而没有必要更新，结果在 linker 将 OBJ 文件连接为可执行文件时产生奇怪的错误信息。

您可以将目标文件作为特殊目标.PRECIOUS 的依赖从而阻止 make 这样删除该目标文件。在重建一个目标之前，make 首先检查该目标文件是否出现在特殊目标.PRECIOUS 的依赖列表中，从而决定在信号发生时是否删除该目标文件。您不删除这种目标文件的原因可能是：目标更新是一种原子风格，或目标文件存在仅仅为了记录更改时间（其内容无关紧要），或目标文件必须一直存在，用来防止其它类型的错误等。

5.6 递归调用 make

递归调用意味着可以在 makefile 文件中将 make 作为一个命令使用。这种技术在包含大的系统中把 makefile 分离为各种各样的子系统时非常有用。例如，假设您有一个子目录‘subdir’，该目录中有它自己的 makefile 文件，您希望在该子目录中运行 make 时使用该

makefile 文件，则您可以按下述方式编写：

```
  subsystem:
 cd subdir && $(MAKE)
```

或，等同于这样写（参阅*选项概要*）：

```
subsystem:
        $(MAKE) -C subdir
```

您可以仅仅拷贝上述例子实现 make 的递归调用，但您应该了解它们是如何工作的，它们为什么这样工作，以及子 make 和上层 make 的相互关系。

为了使用方便，GNU make把变量 CURDIR 的值设置为当前工作的路径。如果‘-C’选项有效，它将包含的是新路径，而不是原来的路径。该值和它在 makefile 中设置的值有相同的优先权（缺省情况下，环境变量 CURDIR 不能重载）。注意，操作 make 时设置该值无效。

5.6.1 变量 MAKE 的工作方式

递归调用 make 的命令总是使用变量 MAKE，而不是明确的命令名‘make’，如下所示：

```
subsystem:
 cd subdir && $(MAKE)
```

该变量的值是调用 make 的文件名。如果这个文件名是‘/bin/make’，则执行的命令是`cd subdir && /bin/make'。如果您在上层 makefile 文件时用特定版本的 make，则执行递归调用时也使用相同的版本。

在命令行中使用变量 MAKE 可以改变‘-t' (‘--touch'), ‘-n' (‘--just-print'), 或 ‘-q' (‘--question')选项的效果。如果在使用变量 MAKE 的命令行首使用字符‘+’也会起到相同的作用。参阅*代替执行命令*。

设想一下在上述例子中命令‘make -t’的执行过程。（-t’选项标志目标已经更新，但却不执行任何命令，参阅*代替执行命令*。)按照通常的定义，命令‘make –t’在上例中仅仅创建名为‘subsystem’的文件而不进行别的工作。您实际要求运行‘cd subdir && make –t’干什么？是执行命令或是按照‘-t’的要求不执行命令？

Make 的这个特点是这样的：只要命令行中包含变量 MAKE，标志`-t', `-n' 和 `-q'将不对本行起作用。虽然存在标志不让命令执行，但包含变量 MAKE 的命令行却正常运行，make 实际上是通过变量 MAKEFLAGS 将标志值传递给了子 make（参阅*与子 make 通讯的选项*）。所以您的验证文件、打印命令的请求等都能传递给子系统。

5.6.2 与子 make 通讯的变量

通过明确要求，上层 make 变量的值可以借助环境传递给子 make，这些变量能在子 make 中缺省定义，在您不使用‘-e’开关的情况下，传递的变量的值不能代替子 make 使用的 makefile 文件中指定的值（参阅*命令概要*）。

向下传递、或输出一个变量时，make 将该变量以及它的值添加到运行每一条命令的环境中。子 make，作为响应，使用该环境初始化它的变量值表。参阅*环境变量*。

除了明确指定外，make 仅向下输出在环境中定义并初始化的或在命令行中设置的变量，而且这些变量的变量名必须仅由字母、数字和下划线组成。一些shell 不能处理名字中含有字母、数字和下划线以外字符的环境变量。特殊变量如 SHELL 和 MAKEFLAGS 一般总要向下输出（除非您不输出它们）。即使您把变量 MAKEFILE 设为其它的值，它也向下输出。

Make 自动传递在命令行中定义的变量的值，其方法是将它们放入 MAKEFLAGS 变量中。详细内容参阅下节。Make 缺省创造的变量的值不能向下传递，子 make 可以自己定义它们。如果您要将指定变量输出给子 make，请用 export 指令，格式如下：

```
export variable ...
```

您要将阻止一些变量输出给子 **make**，请用 **unexport** 指令，格式如下：

unexport *variable ...*

为方便起见，您可以同时定义并输出一个变量：

export *variable* = value

下面的格式具有相同的效果**:**

***variable* = value**
export *variable*

以及

export *variable* := value

具有相同的效果：

***variable* := value**
export *variable*

同样**,**

export *variable* += value

亦同样**:**

***variable* += value**
export *variable*

参阅*为变量值追加文本*。

您可能注意到 **export** 和 **unexport** 指令在 **make** 与 **shell** 中的工作方式相同，如 **sh**。

如果您要将所有的变量都输出，您可以单独使用 **export**：

export

这告诉 **make** 把 **export** 和 **unexport** 没有提及的变量统统输出，但任何在 **unexport** 提及的变量仍然不能输出。如果您单独使用 **export** 作为缺省的输出变量方式，名字中含有字母、数字和下划线以外字符的变量将不能输出，这些变量除非您明确使用 **export** 指令提及才能输出。

单独使用 **export** 的行为是老板本 **GNU make** 缺省定义的行为。如果您的 **makefile** 依靠这些行为，而且您希望和老板本 **GNU make**兼容，您可以为特殊目标**.EXPORT_ALL_VARIABLES** 编写一条规则代替 **export** 指令，它将被老板本 **GNU make** 忽略，但如果同时使用 **export** 指令则报错。

同样，您可以单独使用 **unexport** 告诉 **make** 缺省不要输出变量，因为这是缺省的行为，只有前面单独使用了 **export**（也许在一个包括的 **makefile** 中）您才有必要这样做。您不能同时单独使用 **export** 和 **unexport** 指令实现对某些命令输出对其它的命令不输出。最后面的一条指令（**export** 或 **unexport**）将决定 **make** 的全部运行结果。

作为一个特点，变量 **MAKELEVEL** 的值在从一个层次向下层传递时发生变化。该变量的值是字符型，它用十进制数表示层的深度。‘**0**’代表顶层 **make**，‘**1**’代表子 **make**，‘**2**’代表子**-**子**-make**，以此类推。**Make** 为一个命令建立一次环境，该值增加 **1**。

该变量的主要作用是在一个条件指令中测试（参阅 ***makefile* 文件的条件语句**）；采用这种方法，您可以编写一个 **makefile**，如果递归调用采用一种运行方式，由您控制直接执行采用另一种运行方式。

您可以使用变量 **MAKEFILES** 使所有的子 **make** 使用附加的 **makefile** 文件。变量 **MAKEFILES** 的值是 **makefile** 文件名的列表，文件名之间用空格隔开。在外层 **makefile** 中定义该变量，该变量的值将通过环境向下传递；因此它可以作为子 **make** 的额外的 **makefile** 文件，在子 **make** 读正常的或指定的 **makefile** 文件前，将它们读入。参阅***变量 MAKEFILES*。**

5.6.3 与子 make 通讯的选项

诸如‘-s’和‘-k’标志通过变量 MAKEFLAGS 自动传递给子 make。该变量由 make 自动建立，并包含 make 收到的标志字母。所以，如果您是用‘make -ks’变量 MAKEFLAGS 就得到值‘ks’。

作为结果，任一个子 make 都在它的运行环境中为变量 MAKEFLAGS 赋值；作为响应，make 使用该值作为标志并进行处理，就像它们作为参数被给出一样。参阅*选项概要*。

同样，在命令行中定义的变量也将借助变量 MAKEFLAGS 传递给子 make。变量 MAKEFLAGS 值中的字可以包含‘=’，make 将它们按变量定义处理，其过程和在命令行中定义的变量一样。参阅*变量重载*。

选项`-C', `-f', `-o', 和 ‘-W’不能放入变量 MAKEFLAGS 中；这些选项不能向下传递。

‘-j’选项是一个特殊的例子（参阅*并行执行*）。如果您将它设置为一些数值‘N’，而且您的操作系统支持它（大多数 Unix 系统支持，其它操作系统不支持），父 make 和所有子 make 通讯保证在它们中间同时仅有‘N’个任务运行。注意，任何包含递归调用的任务（参阅*代替执行命令*）不能计算在总任务数内（否则，我们仅能得到‘N’个子 make 运行，而没有多余的时间片运行实在的工作）。

如果您的操作系统不支持上述通讯机制，那么‘-j 1’将放到变量 MAKEFLAGS 中代替您指定的值。这是因为如果‘-j’选项传递给子 make，您可能得到比您要求多很多的并行运行的任务数。如果您给出‘-j’选项而没有数字参数，意味着尽可能并行处理多个任务，这样向下传递，因为倍数的无限制性所以至多为 1。

如果您不希望其它的标志向下传递，您必须改变变量 MAKEFLAGS 的值，其改变方式如下：

```
subsystem:
        cd subdir && $(MAKE) MAKEFLAGS=
```

该命令行中定义变量的实际上出现在变量 MAKEOVERRIDES 中，而且变量 MAKEFLAGS 包含了该变量的引用值。如果您要向下传递标志，而不向下传递命令行中定义的变量，这时，您可以将变量 MAKEOVERRIDES 的值设为空，格式如下：

```
MAKEOVERRIDES =
```

这并不十分有用。但是，一些系统对环境的大小有固定限制，而且该值较小，将这么多的信息放到变量 MAKEFLAGS 的值中可能超过该限制。如果您看到‘Arg list too long’的错误信息，很可能就是由于该问题造成的。（按照严格的 POSIX.2 的规定，如果在 makefile 文件定义特殊目标‘.POSIX’，改变变量 MAKEOVERRIDES 的值并不影响变量 MAKEFLAGS。也许您并不关心这些。）

为了和早期版本兼容，具有相同功能的变量 MFLAGS 也是存在的。除了它不能包含命令行定义变量外，它和变量 MAKEFLAGS 有相同的值，而且除非它是空值，它的值总是以短线开始（MAKEFLAGS 只有在和多字符选项一起使用时才以短线开始，如和‘--warn-undefined-variables’连用）。变量 MFLAGS 传统的使用在明确的递归调用 make 的命令中，例如：

```
subsystem:
        cd subdir && $(MAKE) $(MFLAGS)
```

但现在，变量 MAKEFLAGS 使这种用法变得多余。如果您要您的 makefile 文件和老版本的 make 程序兼容，请使用这种方式；这种方式在现代版本 make 中也能很好的工作。

如果您要使用每次运行 make 都要设置的特定选项，例如‘-k’选项（参阅*选项概要*），变量 MAKEFLAGS 十分有用。您可以简单的在环境中将给变量 MAKEFLAGS 赋值，或在 makefile 文件中设置变量 MAKEFLAGS，指定的附加标志可以对整个 makefile 文件都起作用。（注意：您不能以这种方式使用变量 MFLAGS，变量 MFLAGS 存在仅为和早期版本兼容，采用其它方式设置该变量 make 将不予解释。）

当 make 解释变量 MAKEFLAGS 值的时候（不管在环境中定义或在 makefile 文件中定义），如果该值不以短线开始，则 make 首先为该值假设一个短线；接着将该值分割成字，字与字间用空格隔开，然后将这些字进行语法分析，好像它们是在命令行中给出的选项一样。（-C',

‘**-f**’, ‘**-h**’, ‘**-o**’, ‘**-W**’选项以及它们的长名字版本都将忽略，对于无效的选项不产生错误信息。）

如果您在环境中定义变量 **MAKEFLAGS**，您不要使用严重影响 **make** 运行，破坏 **makefile** 文件的意图以及 **make** 自身的选项。例如‘**-t**’，‘**-n**’，和‘**-q**’选项，如果将它们中的一个放到变量 **MAKEFLAGS** 的值中，可能产生灾难性的后果，或至少产生让人讨厌的结果。

5.6.4 ‘--print-directory’选项

如果您使用几层 **make** 递归调用，使用‘**-w**’或‘**--print-directory**’选项，通过显示每个 **make** 开始处理以及处理完成的目录使您得到比较容易理解的输出。例如，如果使用‘**make** －w’命令在目录‘**/u/gnu/make**’中运行 **make**，则 **make** 将下面格式输出信息：

```
make: Entering directory `/u/gnu/make'.
```

说明进入目录中，还没有进行任何任务。下面的信息：

```
make: Leaving directory `/u/gnu/make'.
```

说明任务已经完成。

通常情况下，您不必具体指明这个选项，因为 **make** 已经为您做了：当您使用‘**-C**’选项时，‘**-w**’选项已经自动打开，在子 **make** 中也是如此。如果您使用‘**-s**’选项，‘**-w**’选项不会自动打开，因为‘**-s**’选项是不打印信息，同样使用`**--no-print-directory**'选项‘**-w**’选项也不会自动打开。

5.7 定义固定次序命令

在创建各种目标时，相同次序的命令十分有用时，您可以使用 **define** 指令定义固定次序的命令，并根据这些目标的规则引用固定次序。固定次序实际是一个变量，因此它的名字不能和其它的变量名冲突。

下面是定义固定次序命令的例子：

```
define run-yacc
yacc $(firstword $^)
mv y.tab.c $@
endef
```

run-yacc 是定义的变量的名字；**endef** 标志定义结束；中间的行是命令。**define** 指令在固定次序中不对变量引用和函数调用扩展；字符‘**$**’、圆括号、变量名等等都变成您定义的变量的值的一部分。***定义多行变量***一节对指令 **define** 有详细解释。

在该例子中，对于任何使用该固定次序的规则，第一个命令是对其第一个依赖运行 **Yacc** 命令，**Yacc** 命令执行产生的输出文件一律命名为‘**y.tab.c**’；第二条命令，是将该输出文件的内容移入规则的目标文件中。

在使用固定次序时，规则中命令使用的变量应被确定的值替代，您可以象替代其它变量一样替代这些变量（详细内容参阅***变量引用基础***）。因为由 **define** 指令定义的变量是递归扩展的变量，所以在使用时所有变量引用才扩展。例如：

```
foo.c : foo.y
        $(run-yacc)
```

当固定次序‘**run-yacc**’运行时，‘**foo.y**’将代替变量‘**$^**’，‘**foo.c**’将代替变量‘**$@**’。这是一个现实的例子，但并不是必要的，因为 **make** 有一条隐含规则可以根据涉及的文件名的类型确定所用的命令。参阅***使用隐含规则***。

在命令执行时，固定次序中的每一行被处理为和直接出现在规则中的命令行一样，前面加上一个 **Tab**，**make** 也特别为每一行请求一个独立的子 **shell**。您也可以在固定次序的每一行上使用影响命令行的前缀字符(**`@'**, **`-'**,和 **`+'**)，参阅***在规则中使用命令***。例如使用下述的固定次序：

```
@echo "frobnicating target $@"
frob-step-1 $< -o $@-step-1
```

```
frob-step-2 $@-step-1 -o $@
endef
```
make 将不回显第一行，但要回显后面的两个命令行。

另一方面，如果前缀字符在引用固定次序的命令行中使用，则该前缀字符将应用到固定次序的每以行中。例如这个规则：

```
frob.out: frob.in
        @$(frobnicate)
```
将不回显固定次序的任何命令。具体内容参阅*命令回显*。

5.8 使用空命令

定义什么也不干的命令有时很有用，定义空命令可以简单的给出一个仅仅含有空格而不含其它任何东西的命令即可。例如：

```
target: ;
```
为字符串‘target’定义了一个空命令。您也可以使用以 Tab 字符开始的命令行定义一个空命令，但这由于看起来空白容易造成混乱。

也许您感到奇怪，为什么我们定义一个空命令？唯一的原因是为了阻止目标更新时使用隐含规则提供的命令。 参阅*使用隐含规则*以及*定义最新类型的缺省规则*）

也许您喜爱为实际不存在的目标文件定义空命令，因为这样它的依赖可以重建。然而这样做并不是一个好方法，因为如果目标文件实际存在，则依赖有可能不重建，使用假想目标是较好的选择，参阅*假想目标*。

6 使用变量

变量是在 makefile 中定义的名字，其用来代替一个文本字符串，该文本字符串称为该变量的值。在具体要求下，这些值可以代替目标、依赖、命令以及 makefile 文件中其它部分。（在其它版本的 make 中，变量称为宏（macros）。）

在 makefile 文件读入时，除规则中的 shell 命令、使用‘=’定义的‘=’右边的变量、以及使用 define 指令定义的变量体此时不扩展外，makefile 文件其它各个部分的变量和函数都将扩展。

变量可以代替文件列表、传递给编译器的选项、要执行的程序、查找源文件的目录、输出写入的目录，或您可以想象的任何文本。

变量名是不包括‘:’,‘#’,‘=’、前导或结尾空格的任何字符串。然而变量名包含字母、数字以及下划线以外的其它字符的情况应尽量避免，因为它们可能在将来被赋予特别的含义，而且对于一些 shell 它们也不能通过环境传递给子 make（参阅*与子 make 通讯的变量*）。变量名是大小写敏感的，例如变量名‘foo’, ‘FOO’, 和 ‘Foo’代表不同的变量。

使用大写字母作为变量名是以前的习惯，但我们推荐在 makefile 内部使用小写字母作为变量名，预留大写字母作为控制隐含规则参数或用户重载命令选项参数的变量名。参阅*变量重载*。

一少部分的变量使用一个标点符号或几个字符作为变量名，这些变量是自动变量，它们又特定的用途。参阅*自动变量*。

6.1 变量引用基础

写一个美元符号后跟用圆括号或大括号括住变量名则可引用变量的值：‘$(foo)’ 和‘${foo}’都是对变量‘foo’的有效引用。‘$’的这种特殊作用是您在命令或文件名中必

须写‘**$$**’才有单个‘**$**’的效果的原因。

变量的引用可以用在上下文的任何地方：目标、依赖、命令、绝大多数指令以及新变量的值等等。这里有一个常见的例子，在程序中，变量保存着所有 **OBJ** 文件的文件名：

```
objects = program.o foo.o utils.o
program : $(objects)
        cc -o program $(objects)

$(objects) : defs.h
```

变量的引用按照严格的文本替换进行，这样该规则

```
foo = c
prog.o : prog.$(foo)
        $(foo)$(foo) -$(foo) prog.$(foo)
```

可以用于编译 **C** 语言源程序‘**prog.c**’。因为在变量分配时，变量值前面的空格被忽略，所以变量 **foo** 的值是‘**C**’。（不要在您的 **makefile** 文件这样写！）

美元符号后面跟一个字符但不是美元符号、圆括号、大括号，则该字符将被处理为单字符的变量名。因此可以使用‘**$x**’引用变量 **x**。然而，这除了在使用自动变量的情况下，在其它实际工作中应该完全避免。参阅*自动变量*。

6.2 变量的两个特色

在 **GNU make** 中可以使用两种方式为变量赋值，我们将这两种方式称为变量的两个特色（**two flavors**）。两个特色的区别在于它们的定义方式和扩展时的方式不同。

变量的第一个特色是递归调用扩展型变量。这种类型的变量定义方式：在命令行中使用‘**=**’定义（参阅*设置变量*）或使用 **define** 指令定义（参阅*定义多行变量*）。变量替换对于您所指定的值是逐字进行替换的；如果它包含对其它变量的引用，这些引用在该变量替换时（或在扩展为其它字符串的过程中）才被扩展。这种扩展方式称为递归调用型扩展。例如：

```
foo = $(bar)
bar = $(ugh)
ugh = Huh?

all:;echo $(foo)
```

将回显‘**Huh?**’：‘**$(foo)**’扩展为‘**$(bar)**’，进一步扩展为‘**$(ugh)**’，最终扩展为‘**Huh?**’。

这种特色的变量是其它版本 **make** 支持的变量类型，有缺点也有优点。大多数人认为的该类型的变量的优点是：

```
CFLAGS = $(include_dirs) -O
include_dirs = -Ifoo -Ibar
```

即能够完成希望它完成的任务：当‘**CFLAGS**’在命令中扩展时，它将最终扩展为‘**-Ifoo -Ibar**’。其最大的缺点是不能在变量后追加内容，如在：

```
CFLAGS = $(CFLAGS) -O
```

在变量扩展过程中可能导致无穷循环（实际上 **make** 侦测到无穷循环就会产生错误信息）。

它的另一个缺点是在定义中引用的任何函数时（参阅*文本转换函数*）变量一旦展开函数就会立即执行。这可导致 **make** 运行变慢，性能变坏；并且导致通配符与 **shell** 函数（因不能控制何时调用或调用多少次）产生不可预测的结果。

为避免该问题和递归调用扩展型变量的不方便性，出现了另一个特色变量：简单扩展型变量。

简单扩展型变量在命令行中用‘**:=**’定义（参阅*设置变量*）。简单扩展型变量的值是一次扫描永远使用，对于引用的其它变量和函数在定义的时候就已经展开。简单扩展型变量的值实际就是您写的文本扩展的结果。因此它不包含任何对其它变量的引用；在该变量定义时就包含了它们的值。所以：

```
x := foo
```

```
y := $(x) bar
x := later
```

等同于：

```
y := foo bar
x := later
```

引用一个简单扩展型变量时，它的值也是逐字替换的。这里有一个稍复杂的例子，说明了‘**:=**’和 **shell** 函数连接用法（参阅*函数 **shell***）。该例子也表明了变量 **MAKELEVEL** 的用法，该变量在层与层之间传递时值发生变化。（参阅*与子 **make** 通讯的变量*，可获得变量 **MAKELEVEL** 关于的信息。）

```
ifeq (0,${MAKELEVEL})
cur-dir   := $(shell pwd)
whoami    := $(shell whoami)
host-type := $(shell arch)
MAKE := ${MAKE} host-type=${host-type} whoami=${whoami}
endif
```

按照这种方法使用‘**:=**’的优点是看起来象下述的典型的‘下降到目录’的命令：

```
${subdirs}:
      ${MAKE} cur-dir=${cur-dir}/$@ -C $@ all
```

简单扩展型变量因为在绝大多数程序设计语言中可以象变量一样工作，因此它能够使复杂的 **makefile** 程序更具有预测性。它们允许您使用它自己的值重新定义（或它的值可以被一个扩展函数以某些方式处理）它们还允许您使用更有效的扩展函数（参阅*文本转换函数*）。

您可以使用简单扩展型变量将控制的前导空格引入到变量的值中。前导空格字符一般在变量引用和函数调用时被丢弃。简单扩展型变量的这个特点意味着您可以在一个变量的值中包含前导空格，并在变量引用时保护它们。象这样：

```
nullstring :=
space := $(nullstring) # end of the line
```

这里变量 **space** 的值就是一个空格，注释‘**# end of the line**’包括在这里为了让人更易理解。因为尾部的空格不能从变量值中分离出去，仅在结尾留一个空格也有同样的效果（但是此时相当难读），如果您在变量值后留一个空格，象这样在行的结尾写上注释清楚表明您的打算是很不错的主意。相反，如果您在变量值后不要空格，您千万记住不要在行的后面留下几个空格再随意放入注释。例如：

```
dir := /foo/bar    # directory to put the frobs in
```

这里变量 **dir** 的值是‘**/foo/bar** ’（四个尾部空格）这不是预期的结果。假设‘**/foo/bar**’是预期的值）。

另一个给变量赋值的操作符是‘**?=**’，它称为条件变量赋值操作符，因为它仅仅在变量还没有定义的情况下有效。这声明：

```
FOO ?= bar
```

和下面的语句严格等同（参阅*函数 **origin***）

```
ifeq ($(origin FOO), undefined)
  FOO = bar
endif
```

注意，一个变量即使是空值，它仍然已被定义，所以使用‘**?=**’定义无效。

6.3 变量引用高级技术

本节内容介绍变量引用的高级技术。

6.3.1 替换引用

替换引用是用您指定的变量替换一个变量的值。它的形式‘**$(*var*:*a*=*b*)**’（或‘**${*var*:*a*=*b*}**’），它的含义是把变量 **var** 的值中的每一个字结尾的 **a** 用 **b** 替换。

我们说‘在一个字的结尾’，我们的意思是 **a** 一定在一个字的结尾出现，且 **a** 的后面要么是空格要么是该变量值的结束，这时的 **a** 被替换，值中其它地方的 **a** 不被替换。例如：

```
foo := a.o b.o c.o
bar := $(foo:.o=.c)
```

将变量‘**bar**’的值设为‘**a.c b.c c.c**’。参阅***变量设置***。

替换引用实际是使用扩展函数 **patsubst** 的简写形式（参阅***字符串替换和分析函数***）。我们提供替换引用也是使扩展函数 **patsubst** 与 **make** 的其它实现手段兼容的措施。

另一种替换引用是使用强大的扩展函数 **patsubst**。它的形式和上述的‘**$(*var*:*a*=*b*)**’一样，不同在于它必须包含单个‘**%**’字符，其实这种形式等同于‘**$(patsubst *a*,*b*,$(*var*))**’。有关于函数 **patsubst** 扩展的描述参阅***字符串替换和分析函数***。例如：

```
foo := a.o b.o c.o
bar := $(foo:%.o=%.c)
```

社值变量‘**bar'**的值为‘**a.c b.c c.c'**。

6.3.2 嵌套变量引用（计算的变量名）

嵌套变量引用（计算的变量名）是一个复杂的概念，仅仅在十分复杂的 **makefile** 程序中使用。绝大多数情况您不必考虑它们，仅仅知道创建名字中含有美元标志的变量可能有奇特的结果就足够了。然而，如果您是要把一切搞明白的人或您实在对它们如何工作有兴趣，请认真阅读以下内容。

变量可以在它的名字中引用其它变量，这称为嵌套变量引用（计算的变量名）。例如：

```
x = y
y = z
a := $($(x))
```

定义阿 **a** 为‘**z**’：‘**$(x)**’在‘**$($(x))**’中扩展为‘**y**’，因此‘**$($(x))**’扩展为‘**$(y)**’，最终扩展为‘**z**’。这里对引用的变量名的陈述不太明确；它根据‘**$(x)**’的扩展进行计算，所以引用‘**$(x)**’是嵌套在外层变量引用中的。

前一个例子表明了两层嵌套，但是任何层次数目的嵌套都是允许的，例如，这里有一个三层嵌套的例子：

```
x = y
y = z
z = u
a := $($($(x)))
```

这里最里面的‘**$(x)**’扩展为‘**y**’，因此‘**$($(x))**’扩展为‘**$(y)**’，‘**$(y)**’扩展为‘**z**’，最终扩展为‘**u**’。

在一个变量名中引用递归调用扩展型变量，则按通常的风格再扩展。例如：

```
x = $(y)
y = z
z = Hello
a := $($(x))
```

定义的 **a** 是‘**Hello**’：‘**$($(x))**’扩展为‘**$($(y))**’，‘**$($(y))**’变为‘**$(z)**’，**$(z)**’最终扩展为‘**Hello**’。

嵌套变量引用和其它引用一样也可以包含修改引用和函数调用（参阅*文本转换函数*）。例如，使用函数 **subst**（参阅*字符串替换和分析函数*）：

```
x = variable1
variable2 := Hello
y = $(subst 1,2,$(x))
z = y
a := $($($(z)))
```

定义的 **a** 是‘**Hello**’。任何人也不会写象这样令人费解的嵌套引用程序，但它确实可以工作：‘**$($($(z)))**’ 扩展为‘**$($(y))**’，‘**$($(y))**’变为‘**$(subst 1,2,$(x))**’。它从变量‘**x**’得到值‘**variable1**’,变换替换为‘**variable2**’，所以整个字符串变为‘**$(variable2)**’，一个简单的变量引用，它的值为‘**Hello**’。

嵌套变量引用不都是简单的变量引用，它可以包含好几个变量引用，同样也可包含一些固定文本。例如，

```
a_dirs := dira dirb
1_dirs := dir1 dir2

a_files := filea fileb
1_files := file1 file2

ifeq "$(use_a)" "yes"
a1 := a
else
a1 := 1
endif

ifeq "$(use_dirs)" "yes"
df := dirs
else
df := files
endif

dirs := $($(a1)_$(df))
```

根据设置的 **use_a** 和 **use_dirs** 的输入可以将 **dirs** 这个相同的值分别赋给 **a_dirs, 1_dirs, a_files** 或 **1_files**。

嵌套变量引用也可以用于替换引用：

```
a_objects := a.o b.o c.o
1_objects := 1.o 2.o 3.o

sources := $($(a1)_objects:.o=.c)
```

根据 **a1** 的值，定义的 **sources** 可以是`**a.c b.c c.c**' 或 `**1.c 2.c 3.c**'。

使用嵌套变量引用唯一的限制是它们不能只部分指定要调用的函数名，这是因为用于识别函数名的测试在嵌套变量引用扩展之前完成。例如：

```
ifdef do_sort
func := sort
else
func := strip
endif

bar := a d b g q c

foo := $($(func) $(bar))
```

则给变量‘foo’的值赋为‘sort a d b g q c'或 ‘strip a d b g q c'，而不是将‘a d b g q c’作为函数 sort 或 strip 的参数。如果在将来去掉这种限制是一个不错的主意。

您也可以变量赋值的左边使用嵌套变量引用，或在 define 指令中。如：

```
dir = foo
$(dir)_sources := $(wildcard $(dir)/*.c)
define $(dir)_print
lpr $($(dir)_sources)
endef
```

该例子定义了变量‘dir',‘foo_sources',和‘foo_print'。

注意：虽然嵌套变量引用和递归调用扩展型变量都是用在复杂的 makefile 文件中，但二者不同（参阅*变量的两个特色*）。

6.4 变量取值

变量有以下几种方式取得它们的值：

- 您可以在运行 make 时为变量指定一个重载值。参阅*变量重载*。
- 您可以在 makefile 文件中指定值，即变量赋值（参阅*设置变量*）或使用逐字定义变量（参阅*定义多行变量*）。
- 把环境变量变为 make 的变量。参阅*环境变量*。
- 自动变量可根据规则提供值，它们都有简单的习惯用法，参阅*自动变量*。
- 变量可以用常量初始化。参阅*隐含规则使用的变量*。

6.5 设置变量

在 makefile 文件中设置变量，编写以变量名开始后跟‘=’或‘:=’的一行即可。任何跟在‘=’或‘:=’后面的内容就变为变量的值。例如：

```
objects = main.o foo.o bar.o utils.o
```

定义一个名为 objects 的变量，变量名前后的空格和紧跟‘=’的空格将被忽略。

使用‘=’定义的变量是递归调用扩展型变量；以‘:=’定义的变量是简单扩展型变量。简单扩展型变量定义可以包含变量引用，而且变量引用在定义的同时就被立即扩展。参阅*变量的两种特色*。

变量名中也可以包含变量引用和函数调用，它们在该行读入时扩展，这样可以计算出能够实际使用的变量名。

变量值的长度没有限制，但受限于计算机中的实际交换空间。当定义一个长变量时，在合适的地方插入反斜杠，把变量值分为多个文本行是不错的选择。这不影响 make 的功能，但可使 makefile 文件更加易读。

绝大多数变量如果您不为它设置值，空字符串将自动作为它的初值。虽然一些变量有内建的非空的初始化值，但您可随时按照通常的方式为它们赋值（参阅*隐含规则使用的变量*。）另外一些变量可根据规则自动设定新值，它们被称为自动变量。参阅*自动变量*。

如果您喜欢仅对没有定义过的变量赋给值，您可以使用速记符‘?=’代替‘=’。下面两种设置变量的方式完全等同（参阅*函数 origin*）：

```
FOO ?= bar
```

和

```
ifeq ($(origin FOO), undefined)
FOO = bar
endif
```

6.6 为变量值追加文本

为已经定以过的变量的值追加更多的文本一般比较有用。您可以在独立行中使用‘**+=**’来实现上述设想。如：

```
objects += another.o
```

这为变量 **objects** 的值添加了文本‘**another.o**’（其前面有一个前导空格）。这样：

```
objects = main.o foo.o bar.o utils.o
objects += another.o
```

变量 **objects** 设置为‘**main.o foo.o bar.o utils.o another.o**'。

使用 **`+='** 相同于**:**

```
objects = main.o foo.o bar.o utils.o
objects := $(objects) another.o
```

对于使用复杂的变量值，不同方法的差别非常重要。如变量在以前没有定义过，则‘**+=**’的作用和‘**=**’相同：它定义一个递归调用型变量。然而如果在以前有定义，‘**+=**’的作用依赖于您原始定义的变量的特色，详细内容参阅***变量的两种特色***。

当您使用‘**+=**’为变量值附加文本时，**make** 的作用就好象您在初始定义变量时就包含了您要追加的文本。如果开始您使用‘**:=**’定义一个简单扩展型变量，再用‘**+=**’对该简单扩展型变量值追加文本，则该变量按新的文本值扩展，好像在原始定义时就将追加文本定义上一样，详细内容参阅***设置变量***。实际上，

```
variable := value
variable += more
```

等同于：

```
variable := value
variable := $(variable) more
```

另一方面，当您把‘**+=**’和首次使用无符号‘**=**’定义的递归调用型变量一起使用时，**make** 的运行方式会有所差异。在您引用递归调用型变量时，**make** 并不立即在变量引用和函数调用时扩展您设定的值；而是将它逐字储存起来，将变量引用和函数调用也储存起来，以备以后扩展。当您对于一个递归调用型变量使用‘**+=**’时，相当于对一个不扩展的文本追加新文本。

```
variable = value
variable += more
```

粗略等同于：

```
temp = value
variable = $(temp) more
```

当然，您从没有定义过叫做 **temp** 的变量，如您在原始定义变量时，变量值中就包含变量引用，此时可以更为深刻地体现使用不同方式定义的的重要性。拿下面常见的例子，

```
CFLAGS = $(includes) -O
...
CFLAGS += -pg # enable profiling
```

第一行定义了变量 **CFLAGS**，而且变量 **CFLAGS** 引用了其它变量，**includes**。（变量 **CFLAGS** 用于 **C** 编译器的规则，参阅***隐含规则目录***。）由于定义时使用‘**=**’，所以变量 **CFLAGS** 是递归调用型变量，意味着‘**$(includes) -O**’在 **make** 处理变量 **CFLAGS** 定义时是不扩展的；也就是变量 **includes** 在生效之前不必定义，它仅需要在任何引用变量 **CFLAGS** 之前定义即可。如果我们试图不使用‘**+=**’为变量 **CFLAGS** 追加文本，我们可能按下述方式：

CFLAGS := $(CFLAGS) -pg # enable profiling
这似乎很好，但结果绝不是我们所希望的。使用‘**:=**’重新定义变量 **CFLAGS** 为简单扩展型变量，意味着 **make** 在设置变量 **CFLAGS** 之前扩展了‘**$(CFLAGS) -pg**’。如果变量 **includes** 此时没有定义，我们将得到‘**-O -pg**’，并且以后对变量 **includes** 的定义也不会有效。相反，使用‘**+=**’ 设置变量 **CFLAGS** 我们得到没有扩展的‘**$(CFLAGS) –O -pg**’，这样保留了对变量 **includes** 的引用，在后面一个地方如果变量 **includes** 得到定义，‘**$(CFLAGS)**’仍然可以使用它的值。

6.7 override 指令

如果一个变量设置时使用了命令参数（参阅***变量重载***），那么在 **makefile** 文件中通常的对该变量赋值不会生效。此时对该变量进行设置，您需要使用 **override** 指令，其格式如下：
override *variable* = *value*

或

override *variable* := *value*
为该变量追加更多的文本，使用：
override *variable* += *more text*
参阅***为变量值追加文本***。

override 指令不是打算扩大 **makefile** 和命令参数冲突，而是希望用它您可以改变和追加哪些设置时使用了命令参数的变量的值。

例如，假设您在运行 **C** 编译器时总是使用‘**-g**’开关，但您允许用户像往常一样使用命令参数指定其它开关，您就可以使用 **override** 指令**:**
override CFLAGS += -g

您也可以在 **define** 指令中使用 **override** 指令，下面的例子也许就是您想要得：

override define foo
bar
endef
关于 **define** 指令的信息参阅下节。

6.8 定义多行变量

设置变量值的另一种方法时使用 **define** 指令。这个指令有一个特殊的用法，既可以定义包含多行字符的变量。这使得定义命令的固定次序十分方便（参阅***定义固定次序命令***）。

在 **define** 指令同一行的后面一般是变量名，当然，也可以什么也没有。变量的值由下面的几行给出，值的结束由仅仅包含 **endef** 的一行标示出。除了上述在语法上的不同之外，**define** 指令象‘**=**’一样工作：它创建了一个递归调用型变量（参阅***变量的两个特色***）。变量的名字可以包括函数调用和变量引用，它们在指令读入时扩展，以便能够计算出实际的变量名。
define two-lines
echo foo
echo $(bar)
endef

变量的值在通常的赋值语句中只能在一行中完成，但在**define** 指令中在 **define** 指令行以后 **endef** 行之前中间所有的行都是变量值的一部分（最后一行除外，因为标示 **endef** 那一行不能认为是变量值的一部分）。前面的例子功能上等同于：

```
two-lines = echo foo; echo $(bar)
```

因为两命令之间用分号隔开，其行为很接近于两个分离的 **shell** 命令。然而，注意使用两个分离的行，意味着 **make** 请求 **shell** 两次，每一行都在独立的子 **shell** 中运行。参阅***执行命令***。

如果您希望使用 **define** 指令的变量定义比使用命令行定义的变量优先，您可以把 **define** 指令和 **override** 指令一块使用：

```
override define two-lines
foo
$(bar)
endef
```

参阅 ***override 指令***。

6.9 环境变量

make 使用的变量可以来自 **make** 的运行环境。任何 **make** 能够看见的环境变量，在 **make** 开始运行时都转变为同名同值的 **make** 变量。但是，在 **makefile** 文件中对变量的具体赋值，或使用带有参数的命令，都可以对环境变量进行重载（如果明确使用‘**-e**’标志，环境变量的值可以对 **makefile** 文件中的赋值进行重载，参阅***选项概要***，但是这在实际中不推荐使用。）

这样，通过在环境中设置变量 **CFLAGS**，您可以实现在绝大多数 **makefile** 文件中的所有 **C** 源程序的编译使用您选择的开关。因为您知道没有 **makefile** 将该变量用于其它任务，所以这种使用标准简洁含义的变量是安全的（但这也是不可靠的，一些 **makefile** 文件可能设置变量 **CFLAGS**，从而使环境中变量 **CFLAGS** 的值失效）。当使用递归调用的 **make** 时，在外层 **make** 环境中定义的变量，可以传递给内层的 **make**（参阅***递归调用 make***）。缺省方式下，只有环境变量或在命令行中定义的变量才能传递给内层 **make**。您可以使用 **export** 指令传递其它变量，参阅***与子 make 通讯的变量***。

环境变量的其它使用方式都不推荐使用。将 **makefile** 的运行完全依靠环境变量的设置、超出 **makefile** 文件的控制范围，这种做法是不明智的，因为不同的用户运行同一个 **makefile** 文件有可能得出不同的结果。这和大部分 **makefile** 文件的意图相违背。

变量 **SHELL** 在环境中存在，用来指定用户对交互的 **shell** 的选择，因此使用变量 **SHELL** 也存字类似的问题。这种根据选定值影响 **make** 运行的方式是很不受欢迎的。所以，**make** 将忽略环境中变量 **SHELL** 的值（在 **MS-DOS** 和 **MS-Windows** 中运行例外，但此时变量 **SHELL** 通常不设置值，参阅***执行命令***）。

6.10 特定目标变量的值

make 中变量的值一般是全局性的；既，无论它们在任何地方使用，它们的值是一样的（当然，您重新设置除外）；自动变量是一个例外（参阅***自动变量***）。

另一个例外是特定目标变量的值，这个特点允许您可以根据 **make** 建造目标的变化改变变量的定义。象自动变量一样，这些值只能在一个目标的命令脚本的上下文起作用。

可以象这样设置特定目标变量的值：

target ... **:** ***variable-assignment***

或这样：

target ... **:** override ***variable-assignment***

‘***target ...***’中可含有多个目标，如此，则设置的特定目标变量的值可在目标列表中的任一个目标中使用。‘***variable-assignment***’使用任何赋值方式都是有效的：递归调用型（‘**=**’）、静态（‘**:=**’）、追加（‘**+=**’）或条件（‘**?=**’）。所有出现在‘***variable-assignment***’中的变量能够在特定目标 ***target ...*** 的上下文中使用：也就是任何以前为特定目标 ***target ...***

定义的特定目标变量的值在这些特定目标中都是有效的。注意这种变量值和全局变量值相比是局部的值：这两种类型的变量不必有相同的类型（递归调用 vs.静态）。

特定目标变量的值和其它 makefile 变量具有相同的优先权。一般在命令行中定义的变量（和强制使用‘-e’情况下的环境变量）的值占据优先的地位，而使用 override 指令定义的特定目标变量的值则占据优先地位。

特定目标变量的值有另外一个特点：当您定义一个特定目标变量时，该变量的值对特定目标 ***target ...***的所有依赖有效，除非这些依赖用它们自己的特定目标变量的值将该变量重载。例如：

```
prog : CFLAGS = -g
prog : prog.o foo.o bar.o
```

将在目标 prog 的命令脚本中设置变量 CFLAGS 的值为‘-g’，同时在创建`prog.o', `foo.o', 和 `bar.o'的命令脚本中变量 CFLAGS 的值也是‘-g’，以及prog.o', ‘foo.o', 和 ‘bar.o' 的依赖的创建命令脚本中变量 CFLAGS 的值也是‘-g’。

6.11 特定格式变量的值

除了特定目标变量的值（参阅上小节）外，GNU make 也支持特定格式变量的值。使用特定格式变量的值，可以为匹配指定格式的目标定义变量。在为目标定义特定目标变量后将搜寻按特定格式定义的变量，在为该目标的父目标定义的特定目标变量前也要搜寻按特定格式定义的变量。

设置特定格式变量格式如下：

pattern ... : variable-assignment

或这样：

pattern ... : override ***variable-assignment***

这里的‘pattern’是%-格式。象特定目标变量的值一样，‘pattern ...’中可含有多个格式，如此，则设置的特定格式变量的值可在匹配列表中的任一个格式中的目标中使用。‘***variable-assignment***’使用任何赋值方式都是有效的，在命令行中定义的变量的值占据优先的地位，而使用 override 指令定义的特定格式变量的值则占据优先地位。例如：

```
%.o : CFLAGS = -O
```

搜寻所有匹配格式%.o 的目标，并将它的变量 CFLAGS 的值设置为‘-O’。

7 makefile 文件的条件语句

一个条件语句可以导致根据变量的值执行或忽略 makefile 文件中一部分脚本。条件语句可以将一个变量与其它变量的值相比较，或将一个变量与一字符串常量相比较。条件语句用于控制 make 实际看见的 makefile 文件部分，不能用于在执行时控制 shell 命令。

7.1 条件语句的例子

下述的条件语句的例子告诉 make 如果变量 CC 的值是‘gcc’时使用一个数据库，如不是则使用其它数据库。它通过控制选择两命令行之一作为该规则的命令来工作。‘CC=gcc’作为 make 改变的参数的结果不仅用于决定使用哪一个编译器，而且决定连接哪一个数据库。

```
libs_for_gcc = -lgnu
normal_libs =
```

```
foo: $(objects)
ifeq ($(CC),gcc)
        $(CC) -o foo $(objects) $(libs_for_gcc)
else
        $(CC) -o foo $(objects) $(normal_libs)
endif
```

该条件语句使用三个指令：**ifeq**、**else** 和 **endif**。

Ifeq 指令是条件语句的开始，并指明条件。它包含两个参数，它们被逗号分开，并被扩在圆括号内。运行时首先对两个参数变量替换，然后进行比较。在 **makefile** 中跟在 **ifeq** 后面的行是符合条件时执行的命令；否则，它们将被忽略。

如果前面的条件失败，**else** 指令将导致跟在其后面的命令执行。在上述例子中，意味着当第一个选项不执行时，和第二个选项连在一起的命令将执行。在条件语句中，**else** 指令是可选择使用的。

Endif 指令结束条件语句。任何条件语句必须以 **endif** 指令结束，后跟 **makefile** 文件中的正常内容。

上例表明条件语句工作在原文水平：条件语句的行根据条件要么被处理成 **makefile** 文件的一部分或要么被忽略。这是 **makefile** 文件重大的语法单位（例如规则）可以跨越条件语句的开始或结束的原因。

当变量 **CC** 的值是 **gcc**，上例的效果为：

```
foo: $(objects)
        $(CC) -o foo $(objects) $(libs_for_gcc)
```

当变量 **CC** 的值不是 **gcc** 而是其它值的时候，上例的效果为：

```
foo: $(objects)
        $(CC) -o foo $(objects) $(normal_libs)
```

相同的结果也能使用另一种方法获得：先将变量的赋值条件化，然后再使用变量：

```
libs_for_gcc = -lgnu
normal_libs =

ifeq ($(CC),gcc)
  libs=$(libs_for_gcc)
else
  libs=$(normal_libs)
endif

foo: $(objects)
        $(CC) -o foo $(objects) $(libs)
```

7.2 条件语句的语法

对于没有 **else** 指令的条件语句的语法为：

```
conditional-directive
text-if-true
endif
```

‘***text-if-true***’可以是任何文本行，在条件为‘真’时它被认为是 **makefile** 文件的一部分；如果条件为‘假’，将被忽略。完整的条件语句的语法为：

```
conditional-directive
text-if-true
else
text-if-false
endif
```

如果条件为‘真’，使用‘***text-if-true***’；如果条件为‘假’，使用‘***text-if-false***’。‘***text-if-false***’可以是任意多行的文本。

关于‘*conditional-directive*’的语法对于简单条件语句和复杂条件语句完全一样。有四种不同的指令用于测试不同的条件。下面是指令表：

ifeq (*arg1*, *arg2*)
ifeq '*arg1*' '*arg2*'
ifeq "*arg1*" "*arg2*"
ifeq "*arg1*" '*arg2*'
ifeq '*arg1*' "*arg2*"

扩展参数 **arg1**、**arg2** 中的所有变量引用，并且比较它们。如果它们完全一致，则使用‘***text-if-true***’，否则使用‘***text-if-false***’（如果存在的话）。您经常要测试一个变量是否有非空值，当经过复杂的变量和函数扩展得到一个值，对于您认为是空值，实际上有可能由于包含空格而被认为不是空值，由此可能造成混乱。对于此，您可以使用 **strip** 函数从而避免空格作为非空值的干扰。例如：

ifeq ($(strip $(foo)),)
text-if-empty
endif

即使**$(foo)**中含有空格，也使用‘***text-if-empty***’。

ifneq (*arg1*, *arg2*)
ifneq '*arg1*' '*arg2*'
ifneq "*arg1*" "*arg2*"
ifneq "*arg1*" '*arg2*'
ifneq '*arg1*' "*arg2*"

扩展参数 **arg1**、**arg2** 中的所有变量引用，并且比较它们。如果它们不同，则使用‘***text-if-true***’，否则使用‘***text-if-false***’（如果存在的话）。

ifdef *variable-name*

如果变量‘***variable-name***’是非空值，‘***text-if-true***’有效，否则，‘***text-if-false***’有效（如果存在的话）。变量从没有被定义过则变量是空值。**注意 ifdef 仅仅测试变量是否有值。它不能扩展到看变量是否有非空值。因而，使用 ifdef 测试所有定义过的变量都返回‘真’，但那些象‘foo=’情况除外。测试空值请使用 ifeq($(foo),)。例如：**

```
bar =
foo = $(bar)
ifdef foo
frobozz = yes
else
frobozz = no
endif
```

设置‘**frobozz**'的值为‘**yes**',而：**:**

```
foo =
ifdef foo
frobozz = yes
else
frobozz = no
endif
```

设置‘**frobozz**' 为‘**no**'。

ifndef *variable-name*

如果变量‘***variable-name***’是空值，‘***text-if-true***’有效，否则，‘***text-if-false***’有效（如果存在的话）。

在指令行前面允许有多余的空格，它们在处理时被忽略，但是不允许有 **Tab**（如果一行以 **Tab** 开始，那么该行将被认为是规则的命令行）。除此之外，空格和 **Tab** 可以插入到行的任何地方，当然指令名和参数中间除外。以‘**#**’开始的注释可以在行的结尾。

在条件语句中另两个有影响的指令是 **else** 和 **endif**。这两个指令以一个单词的形式出现，没有任何参数。在指令行前面允许有多余的空格，空格和 **Tab** 可以插入到行的中间，以‘**#**’开始的注释可以在行的结尾。

条件语句影响 make 使用的 makefile 文件。如果条件为‘真’，make 读入‘*text-if-true*’包含的行；如果条件为‘假’，make 读入‘*text-if-false*’包含的行（如果存在的话）；makefile 文件的语法单位，例如规则，可以跨越条件语句的开始或结束。

当读入 makefile 文件时，Make 计算条件的值。因而您不能在测试条件时使用自动变量，因为他们是命令执行时才被定义（参阅*自动变量*）。

为了避免不可忍受的混乱，在一个 makefile 文件中开始一个条件语句，而在另外一个 makefile 文件中结束这种情况是不允许的。然而如果您试图引入包含的 makefile 文件不中断条件语句，您可以在条件语句中编写 include 指令。

7.3 测试标志的条件语句

您可以使用变量 MAKEFLAGS 和 findstring 函数编写一个条件语句，用它来测试例如‘-t’等的 make 命令标志（参阅*字符串替换和分析的函数*）。这适用于仅使用 touch 标志不能完全更改文件的时间戳的场合。

findstring 函数检查一个字符串是否为另一个字符串的子字符串。如果您要测试‘-t’标志，使用‘-t’作为第一个字符串，将变量 MAKEFLAGS 的值作为另一个字符串。例如下面的例子是安排使用‘ranlib -t’完成一个档案文件的更新：

```
archive.a: ...
ifneq (,$(findstring t,$(MAKEFLAGS)))
        +touch archive.a
        +ranlib -t archive.a
else
        ranlib archive.a
endif
```

前缀‘+’表示这些命令行是递归调用行，即使是用‘-t’标志它们一样要执行。参阅*递归调用 make*。

8 文本转换函数

函数允许您在 makefile 文件中处理文本、计算文件、操作使用命令等。在函数调用时您必须指定函数名以及函数操作使用的参数。函数处理的结果将返回到 makefile 文件中的调用点，其方式和变量替换一样。

8.1 函数调用语法

函数调用和变量引用类似，它的格式如下：

```
$(function arguments)
```

或这样：

```
${function arguments}
```

这里‘function’是函数名，是 make 内建函数列表中的一个。当然您也可以使用创建函数 call 创建的您自己的函数。‘*arguments*’是该函数的参数。参数和函数名之间是用空格或 Tab 隔开，如果有多个参数，它们之间用逗号隔开。这些空格和逗号不是参数值的一部分。包围函数调用的定界符，无论圆括号或大括号，可以在参数中成对出现，在一个函数调用中只能有一种定界符。如果在参数中包含变量引用或其它的函数调用，最好使用同一种定

界符，如写为‘**$(subst a,b,$(x))'**，而不是‘**`$(subst a,b,${x})'**。这是因为这种方式不但比较清楚，而且也有在一个函数调用中只能有一种定界符的规定。

为每一个参数写的文本经过变量替换或函数调用处理，最终得到参数的值，这些值是函数执行必须依靠的文本。另外，变量替换是按照变量在参数中出现的次序进行处理的。

逗号和不成对出现的圆括号、大括号不能作为文本出现在参数中，前导空格也不能出现在第一个参数中。这些字符不能被变量替换处理为参数的值。如果需要使用这些字符，首先定义变量 **comma** 和 **space**，它们的值是单独的逗号和空格字符，然后在需要的地方因用它们，如下例：

```
comma:= ,
empty:=
space:= $(empty) $(empty)
foo:= a b c
bar:= $(subst $(space),$(comma),$(foo))
# bar is now `a,b,c'.
```

这里函数 **subst** 的功能是将变量 **foo** 中的空格用逗号替换，然后返回结果。

8.2 字符串替换和分析函数

这里有一些用于操作字符串的函数：

$(subst *from*,*to*,*text*)

在文本‘***text***’中使用‘***to***’替换每一处‘***from***’。例如：

```
$(subst ee,EE,feet on the street)
```

结果为‘**fEEt on the street**’。

$(patsubst *pattern*,*replacement*,*text*)

寻找‘**text**’中符合格式‘**pattern**’的字，用‘**replacement**’替换它们。这里‘**pattern**’中包含通配符‘**%**’，它和一个字中任意个数的字符相匹配。如果‘**replacement**’中也含有通配符‘**%**’，则这个‘**%**’被和‘**pattern**’中通配符‘**%**’匹配的文本代替。在函数 **patsubst** 中的‘**%**’可以用反斜杠**(** ‘****’ **)**引用。引用字符‘**%**’的反斜杠可以被更多反斜杠引用。引用字符‘**%**’和其它反斜杠的反斜杠在比较文件名或有一个 **stem**（径）代替它之前从格式中移出。使用反斜杠引用字符‘**%**’不会带来其它麻烦。例如，格式‘**the\%weird\\%pattern\\'** 是‘**the%weird\'** 加上通配符‘**%'**然后和字符串‘**pattern\\'**连接。最后的两个反斜杠由于不能影响任何统配符‘**%**’所以保持不变。在字之间的空格间被压缩为单个空格，前导以及结尾空格被丢弃。例如：

```
$(patsubst %.c,%.o,x.c.c bar.c)
```

的结果为：‘**x.c.o bar.o'**。替换引用是实现函数 **patsubst** 功能一个简单方法：

$(*var*:*pattern*=*replacement*)

等同于 ：

$(patsubst *pattern*,*replacement*,$(*var*))

另一个通常使用的函数 **patsubst** 的简单方法是：替换文件名的后缀。

$(*var*:*suffix*=*replacement*)

等同于：

$(patsubst %*suffix*,%*replacement*,$(*var*))

例如您可能有一个 **OBJ** 文件的列表：

```
objects = foo.o bar.o baz.o
```

要得到这些文件的源文件，您可以简单的写为：

```
$(objects:.o=.c)
```

代替规范的格式：

```
$(patsubst %.o,%.c,$(objects))
```

$(strip *string*)

去掉前导和结尾空格，并将中间的多个空格压缩为单个空格。这样，‘**$(strip a b c)'** 结果为‘**a b c**’。函数 **strip** 和条件语句连用非常有用。当使用 **ifeq** 或 **ifneq** 把一些值和

空字符串‘’比较时，您通常要将一些仅由空格组成的字符串认为是空字符串（参阅 ***makefile 中的条件语句***）。如此下面的例子在实现预期结果时可能失败：

```
.PHONY: all
ifneq   "$(needs_made)" ""
all: $(needs_made)
else
all:;@echo 'Nothing to make!'
endif
```

在条件指令 **ifneq** 中用函数调用‘**$(strip $(needs_made))**'代替变量引用‘**$(needs_made)**'将不再出现问题。

$(findstring *find*,*in*)

在字符串‘**in**’中搜寻‘**find**’，如果找到，则返回值是‘**find**’，否则返回值为空。您可以在一个条件中使用该函数测试给定的字符串中是否含有特定的子字符串。这样，下面两个例子：

```
$(findstring a,a b c)
$(findstring a,b c)
```

将分别产生值‘**a**’和‘’。对于函数 **findstring** 的特定用法参阅***测试标志的条件语句***。

$(filter *pattern...*,*text*)

返回在‘**text**’中由空格隔开且匹配格式‘**pattern...**’的字，对于不符合格式‘**pattern...**’的字移出。格式用‘%’写出，和前面论述过的函数 **patsubst** 的格式相同。函数 **filter** 可以用来变量分离类型不同的字符串。例如：

```
sources := foo.c bar.c baz.s ugh.h
foo: $(sources)
        cc $(filter %.c %.s,$(sources)) -o foo
```

表明‘**foo**'依靠‘**foo.c**'，‘**bar.c**'，‘**baz.s**'和‘**ugh.h**'；但仅有‘**foo.c**'，‘**bar.c**'和‘**baz.s**'指明用命令编译。

$(filter-out *pattern...*,*text*)

返回在‘**text**’中由空格隔开且不匹配格式‘**pattern...**’的字，对于符合格式‘**pattern...**’的字移出。只是函数 **filter** 的反函数。例如：

```
objects=main1.o foo.o main2.o bar.o
mains=main1.o main2.o
```

下面产生不包含在变量‘**mains**’中的 **OBJ** 文件的文件列表：

```
$(filter-out $(mains),$(objects))
```

$(sort *list*)

将‘**list**’中的字按字母顺序排序，并取掉重复的字。输出是由单个空格隔开的字的列表。

```
$(sort foo bar lose)
```

返回值是‘**bar foo lose**’。顺便提及，由于函数 **sort** 可以取掉重复的字，您就是不关心排序也可以使用它的这个特点。

这里有一个实际使用函数 **subst** 和 **patsubst** 的例子。假设一个 **makefile** 文件使用变量 **VPATH** 指定 **make** 搜寻依赖文件的一系列路径（参阅 ***VPATH:依赖搜寻路径***）。这个例子表明怎样告诉 **C** 编译器在相同路径列表中搜寻头文件。

变量 **VPATH** 的值是一列用冒号隔开的路径名，如‘**src:../headers**'。首先，函数 **subst** 将冒号变为空格：

$(subst :, ,$(VPATH))

这产生值‘**src ../headers**'。然后，函数 **patsubst** 为每一个路径名加入‘**-I**’标志，这样这些路径可以加到变量 **CFLAGS** 中，就可以自动传递给 **C** 编译器：

override CFLAGS += $(patsubst %,-I%,$(subst :, ,$(VPATH)))

结果是在以前给定的变量 **CFLAGS** 的值后追加文本‘**-Isrc -I../headers**’。**Override** 指令的作用是即使以前使用命令参数指定变量 **CFLAGS** 的值，新值也能起作用。参阅 ***override 指令***。

8.3 文件名函数

其中几个内建的扩展函数和拆分文件名以及列举文件名相关联。下面列举的函数都能执行对文件名的特定转换。函数的参数是一系列的文件名，文件名之间用空格隔开（前导和结尾空格被忽略）。列表中的每一个文件名都采用相同的方式转换，而且结果用单个空格串联在一起。

$(dir *names*...)

抽取‘**names**’中每一个文件名的路径部分，文件名的路径部分包括从文件名的开始到最后一个斜杠（含斜杠）之前的一切字符。如果文件名中没有斜杠，路径部分是‘**./**’。如：

```
$(dir src/foo.c hacks)
```

产生的结果为 ‘**src/ ./**’。

$(notdir *names*...)

抽取‘**names**’中每一个文件名中除路径部分外一切字符（真正的文件名）。如果文件名中没有斜杠，则该文件名保持不变，否则，将路径部分移走。一个文件名如果仅包含路径部分（以斜杠结束的文件名）将变为空字符串。这是非常不幸的，因为这意味着在结果中如果有这种文件名存在，两文件名之间的空格将不是由相同多的空格隔开。但现在我们并不能看到其它任何有效的代替品。例如：

```
$(notdir src/foo.c hacks)
```

产生的结果为‘**foo.c hacks**’。

$(suffix *names*...)

抽取‘**names**’中每一个文件名的后缀。如果文件名中（或含有斜杠，且在最后一个斜杠后）含有句点，则后缀是最后那个句点以后的所有字符，否则，后缀是空字符串。如果结果为空意味着‘**names**’没有带后缀文件名，如果文件中含有多个文件名，则结果列出的后缀数很可能比原文件名数目少。例如：

```
$(suffix src/foo.c src-1.0/bar.c hacks)
```

产生的结果是‘**.c .c**’。

$(basename *names*...)

抽取‘**names**’中每一个文件名中除后缀外一切字符。如果文件名中（或含有斜杠，且在最后一个斜杠后）含有句点，则基本名字是从开始到最后一个句点（不包含）间的所有字符。如果没有句点，基本名字是整个文件名。例如：

```
$(basename src/foo.c src-1.0/bar hacks)
```

产生的结果为‘**src/foo src-1.0/bar hacks**’。

$(addsuffix *suffix*,*names*...)

参数‘**names**’作为一系列的文件名，文件名之间用空格隔开；**suffix** 作为一个单位。将 **Suffix**（后缀）的值附加在每一个独立文件名的后面，完成后将文件名串联起来，它们之间用单个空格隔开。例如：

```
$(addsuffix .c,foo bar)
```

结果为‘**foo.c bar.c**’。

$(addprefix *prefix*,*names*...)

参数‘**names**’作为一系列的文件名，文件名之间用空格隔开；**prefix** 作为一个单位。将 **preffix**（前缀）的值附加在每一个独立文件名的前面，完成后将文件名串联起来，它们之间用单个空格隔开。例如：

```
$(addprefix src/,foo bar)
```

结果为‘**src/foo src/bar**’。

$(join *list1*,*list2*)

将两个参数串联起来：两个参数的第一个字串联起来形成结果的第一个字，两个参数的第二个字串联起来形成结果的第二个字，以此类推。如果一个参数比另一个参数的字多，则多余的字原封不动的拷贝到结果上。例如，‘**$(join a b,.c .o)**’产生‘**a.c b.o**’。字之间多余的空格不再保留，它们由单个空格代替。该函数可将函数 **dir**、**notdir** 的结果合并，产生原始给定的文件列表。

$(word *n*,*text*)

返回‘**text**’中的第 **n** 个字。**N** 的合法值从 **1** 开始。如果 **n** 比‘**text**’中的字的数目大，则返回空值。例如：

$(word 2, foo bar baz)

返回 ‘**bar**’。

$(wordlist *s*,*e*,*text*)

返回‘**text**’中的从第 **s** 个字开始到第 **e** 个字结束的一列字。**S**、**e** 的合法值从 **1** 开始。如果 **s** 比‘**text**’中的字的数目大，则返回空值；如果 **e** 比‘**text**’中的字的数目大，则返回从第 **s** 个字开始到‘**text**’结束的所有字；如果 **s** 比 **e** 大，不返回任何值。例如：

$(wordlist 2, 3, foo bar baz)

返回`**bar baz**'。

$(words *text*)

返回‘**text**’中字的数目。这样‘**text**’中的最后一个字是‘**$(word $(words*text*),*text*)**’。

$(firstword *names...*)

参数‘**names**’作为一系列的文件名，文件名之间用空格隔开；返回第一个文件名，其余的忽略。例如：

$(firstword foo bar)

产生结果‘**foo**’。 虽然 **$(firstword *text*)** 和 **$(word 1,*text*)**的作用相同，但第一个函数因为简单而保留下来。

$(wildcard *pattern*)

参数‘**pattern**’是一个文件名格式，典型的包含通配符（和 **shel** 中的文件名一样）。函数 **wildcard** 的结果是一列和格式匹配的且文件存在的文件名，文件名之间用一个空格隔开，参阅*在文件名中使用通配符*。

8.4 函数 foreach

函数 **foreach** 和其它函数非常不同，它导致一个文本块重复使用，而且每次使用该文本块进行不同的替换；它和 **shell sh** 中的命令 **for** 及 **C-shell csh** 中的命令 **foreach** 类似。

函数 **foreach** 语法如下：

$(foreach *var*,*list*,*text*)

前两个参数，‘**var**’和‘**list**’，将首先扩展，注意最后一个参数‘**text**’此时不扩展；接着，对每一个‘**list**’扩展产生的字，将用来为‘**var**’扩展后命名的变量赋值；然后‘**text**’引用该变量扩展；因此它每次扩展都不相同。

结果是由空格隔开的‘**text**’ 在‘**list**’中多次扩展的字组成的新的‘**list**’。‘**text**’多次扩展的字串联起来，字与字之间由空格隔开，如此就产生了函数 **foreach** 的返回值。

这是一个简单的例子，将变量‘**files**’的值设置为 ‘**dirs**’中的所有目录下的所有文件的列表：

```
dirs := a b c d
files := $(foreach dir,$(dirs),$(wildcard $(dir)/*))
```

这里‘**text**’是‘**$(wildcard $(dir)/*)**’。第一个为变量 **dir** 发现的值是‘**a**’，所以产生函数 **foreach** 结果的第一个字为‘**$(wildcard a/*)**’；第二个重复的值是‘**b**’，所以产生函数 **foreach** 结果的第二个字为‘**$(wildcard b/*)**’；第三个重复的值是‘**c**’，所以产生函数 **foreach** 结果的第三个字为‘**$(wildcard c/*)**’；等等。该例子和下例有共同的结果：

```
files := $(wildcard a/* b/* c/* d/*)
```

如果‘**text**’比较复杂，您可以使用附加变量为它命名，这样可以提高程序的可读性：

```
find_files = $(wildcard $(dir)/*)
dirs := a b c d
files := $(foreach dir,$(dirs),$(find_files))
```

这里我们使用变量 **find_file**。我们定义变量 **find_file** 时，使用了‘**=**’，因此该变量为递归调用型变量，这样变量 **find_file** 所包含的函数调用将在函数 **foreach** 控制下在扩展；对于简单扩展型变量将不是这样，在变量 **find_file** 定义时就调用函数 **wildcard**。

函数 **foreach** 对变量‘**var**’没有长久的影响，它的值和变量特色在函数 **foreach** 调用结束后将和前面一样，其它从‘**list**’得到的值仅在函数 **foreach** 执行时起作用，它们是暂时的。变量‘**var**’在函数 **foreach** 执行期间是简单扩展型变量，如果在执行函数 **foreach** 之前变量‘**var**’没有定义，则函数 **foreach** 调用后也没有定义。参阅***变量的两个特色***。

当使用复杂变量表达式产生变量名时应特别小心，因为许多奇怪的字符作为变量名是有效的，但很可能不是您所需要的，例如：

```
files := $(foreach Esta escrito en espanol!,b c ch,$(find_files))
```

如果变量 **find_file** 扩展引用名为‘**Esta escrito en espanol!**’变量，上例是有效的，但它极易带来错误。

8.5 函数 if

函数 **if** 对在函数上下文中扩展条件提供了支持（相对于 **GNU make makefile** 文件中的条件语句，例如 **ifeq** 指令，参阅***条件语句的语法***）。

一个函数 **if** 的调用，可以包含两个或三个参数：

$(if *condition*,*then-part*[,*else-part*])

第一个参数‘**condition**’，首先把前导、结尾空格去掉，然后扩展。如果扩展为非空字符串，则条件‘**condition**’为‘真’；如果扩展为空字符串，则条件‘**condition**’为‘假’。

如果条件‘**condition**’为‘真’**,**那么计算第二个参数‘**then-part**’的值，并将该值作为整个函数 **if** 的值。

如果条件‘**condition**’为‘假’**,**第三个参数如果存在，则计算第三个参数‘**else-part**’的值，并将该值作为整个函数 **if** 的值；如果第三个参数不存在，函数 **if** 将什么也不计算，返回空值。

注意仅能计算‘**then-part**’和‘**else-part**’二者之一，不能同时计算。这样有可能产生副作用（例如函数 **shell** 的调用）。

8.6 函数 call

函数 **call** 是唯一的创建新的带有参数函数的函数。您可以写一个复杂的表达是作为一个变量的值，然后使用函数 **call** 用不同的参数调用它。

函数 **call** 的语法为：

$(call *variable*,*param*,*param*,...)

当 **make** 扩展该函数时，它将每一个参数‘**param**’赋值给临时变量**$(1)**、**$(2)**等；变量**$(0)**的值是变量‘**variable**’。对于参数‘**param**’的数量无没有最大数目限制，也没有最小数目限制，但是如果使用函数 **call** 而没有任何参数，其意义不大。

变量‘**variable**’在这些临时变量的上下文中被扩展为一个 **make** 变量，这样，在变量‘**variable**’中对变量‘**$(1)**’的引用决定了调用函数 **call** 时对第一个参数‘**param**’的使用。

注意变量‘**variable**’是一个变量的名称，不是对该变量的引用，所以，您不能采用‘**$**’和圆括号的格式书写该变量，当然，如果您需要使用非常量的文件名，您可以在文件名中使用变量引用。

如果变量名是内建函数名，则该内建函数将被调用（即使使用该名称的 **make** 变量已经存在）。函数 **call** 在给临时变量赋值以前首先扩展参数，这意味着，变量‘**variable**’对内建函数的调用采用特殊的规则进行扩展，象函数 **foreach** 或 **if**，它们的扩展结果和您预期的结果可能不同。下面的一些例子能够更清楚的表达这一点。

该例子时使用宏将参数的顺序翻转：

```
reverse = $(2) $(1)
```

```
foo = $(call reverse,a,b)
```
这里变量 **foo** 的值是“**b a**”。

下面是一个很有意思的例子：它定义了一个宏，使用该宏可以搜寻变量 **PATH** 包含的所有目录中的第一个指定类型的程序：

```
pathsearch = $(firstword $(wildcard $(addsufix /$(1),$(subst :, ,$(PATH)))))

LS := $(call pathsearch,ls)
```
现在变量 **LS** 的值是“**/bin/ls**”或其它的类似的值。

在函数 **call** 中可以使用嵌套。每一次递归调用都可以为它自己的局部变量“**$(1)**”等赋值，从而代替上一层函数 **call** 赋的值。例如：这实现了映像函数功能。

```
map = $(foreach a,$(2),$(call $(1),$(a)))
```
现在您可以映像（**map**）仅有一个参数的函数，如函数 **origin**，一步得到多个值：

```
o = $(call map,origin,o map MAKE)
```
最后变量 **o** 包含诸如“**file file default**”这样的值。

警告：在函数 call 的参数中使用空格一定要十分小心。因为在其它函数中，第二个或接下来的参数中的空格是不删除的，这有可能导致非常奇怪的结果。当您使用函数 **call** 时，去掉参数中任何多余的空格才是最安全的方法。

8.7 函数 origin

函数 **origin** 不想一般函数，它不对任何变量的值操作；它仅仅告诉您一些关于一个变量的信息；它特别的告诉您变量的来源。

函数 **origin** 的语法：

```
$(origin variable)
```
注意变量“**variable**”是一个查询变量的名称，不是对该变量的引用所以，您不能采用“**$**”和圆括号的格式书写该变量，当然，如果您需要使用非常量的文件名，您可以在文件名中使用变量引用。

函数 **origin** 的结果是一个字符串，该字符串变量是怎样定义的：

‘**undefined**’

如果变量“**variable**”从没有定义。

‘**default**’

变量“**variable**”是缺省定义，通常和命令 **CC** 等一起使用，参阅*隐含规则使用的变量*。注意如果您对一个缺省变量重新进行了定义，函数 **origin** 将返回后面的定义。

‘**environment**’

变量“**variable**”作为环境变量定义，选项“**-e**”没有打开（参阅*选项概要*）。

‘**environment override**’

变量“**variable**”作为环境变量定义，选项“**-e**”已打开（参阅*选项概要*）。

‘**file**’

变量“**variable**”在 **makefile** 中定义。

‘**command line**’

变量“**variable**”在命令行中定义。

‘**override**’

变量“**variable**”在 **makefile** 中用 **override** 指令定义（参阅 ***override* 指令**）。

‘**automatic**’

变量“**variable**”是自动变量，定义它是为了执行每个规则中的命令（参阅*自动变量*）。

这种信息的基本用途（其它用途是满足您的好奇心）是使您要了解变量值的依据。例如，假设您有一个名为“**foo**”的 **makefile** 文件，它包含了另一个名为“**bar**”的 **makefile** 文件，如果在环境变量中已经定义变量“**bletch**”，您希望运行命令“**make -f bar**”在 **makefile** 文件“**bar**”中重新定义变量“**bletch**”。但是 **makefile** 文件“**foo**”在包括 **makefile** 文件“**bar**”之前已经定义了变量“**bletch**”，而且您也不想使用 **override** 指令定义，那么您可

以在 **makefile** 文件‘**foo**’中使用 **override** 指令，因为 **override** 指令将会重载任何命令行中的定义，所以其定义的优先权超越以后在 **makefile** 文件‘**bar**’中的定义。因此**makefile** 文件‘**bar**’可以包含：

```
ifdef bletch
ifeq "$(origin bletch)" "environment"
bletch = barf, gag, etc.
endif
endif
```

如果变量‘**bletch**’在环境中定义，这里将重新定义它。

即使在使用选项‘**-e**’的情况下，您也要对来自环境的变量‘**bletch**’重载定义，则您可以使用如下内容：

```
ifneq "$(findstring environment,$(origin bletch))" ""
bletch = barf, gag, etc.
endif
```

如果‘**$(origin bletch)**’返回‘**environment**’或‘**environment override**’，这里将对变量‘**bletch**’重新定义。参阅*字符串替换和分析函数*。

8.8 函数 shell

除了函数 **wildcard** 之外，函数 **shell** 和其它函数不同，它是 **make** 与外部环境的通讯工具。函数 **shell** 和在大多数 **shell** 中后引号（’）执行的功能一样：它用于命令的扩展。这意味着它起着调用 **shell** 命令和返回命令输出结果的参数的作用。**Make** 仅仅处理返回结果，再返回结果替换调用点之前，**make** 将每一个换行符或者一对回车/换行符处理为单个空格；如果返回结果最后是换行符（和回车符），**make** 将把它们去掉。由函数 **shell** 调用的命令，一旦函数调用展开，就立即执行。在大多数情况下，当 **makefile** 文件读入时函数 **shell** 调用的命令就已执行。例外情况是在规则命令行中该函数的调用，因为这种情况下只有在命令运行时函数才能扩展，其它调用函数 **shell** 的情况和此类似。

这里有一些使用函数 **shell** 的例子：

```
contents := $(shell cat foo)
```

将含有文件 **foo** 的目录设置为变量 **contents** 的值，是用空格（而不是换行符）分离每一行。

```
files := $(shell echo *.c)
```

将‘***.c**’的扩展设置为变量 **files** 的值。除非 **make** 使用非常怪异的 **shell**，否则这条语句和‘**wildcard *.c**’的结果相同。

8.9 控制 make 的函数

这些函数控制 **make** 的运行方式。通常情况下，它们用来向用户提供 **makefile** 文件的信息或在侦测到一些类型的环境错误时中断 **make** 运行。

```
$(error text...)
```

通常‘**text**’是致命的错误信息。注意错误是在该函数计算时产生的，因此如果您将该函数放在命令的脚本中或递归调用型变量赋值的右边，它直到过期也不能计算。‘**text**’将在错误产生之前扩展，例如：

```
ifdef ERROR1
$(error error is $(ERROR1))
endif
```

如果变量 **ERROR01** 已经定义，在将 **makefile** 文件读入时产生致命的错误。或，

```
ERR = $(error found an error!)

.PHONY: err
err: ; $(ERR)
```

如果 **err** 目标被调用，在 **make** 运行时产生致命错误。

$(warning *text*...)

该函数和函数 **error** 工作的方式类似，但此时 **make** 不退出，即虽然‘**text**’扩展并显示结果信息，但 **make** 仍然继续执行。扩展该函数的结果是空字符串。

9 运行 make

讲述编译程序的 **makefile** 文件，可以由多种方式实现。最简单的方式是编译所有过期的文件，对于通常所写的 **makefile** 文件，如果不使用任何参数运行 **make**，那么将这样执行。

但是您也许仅仅更新一部分文件；您也许需要使用不同的编译器或不同的编译选项；您也许仅仅希望找出过时的文件而不更新它们。这些只有通过在运行 **make** 时给出参数才能实现。退出 **make** 状态有三种情况：

0

表示 **make** 成功完成退出。

2

退出状态为 **2** 表示 **make** 运行中遇到错误，它将打印信息描述错误。

1

退出状态为 **1** 表示您运行 **make** 时使用了‘**-q**’标志，并且 **make** 决定一些文件没有更新。参阅*代替执行命令*。

9.1 指定 makefile 文件的参数

指定 **makefile** 文件名的方法是使用‘**-f**’或‘**--file**’选项（‘**--makefile**’也能工作）。例如，‘**-f altmake**’说明名为‘**altmake**’的文件作为 **makefile** 文件。

如果您连续使用‘**-f**’标志几次，而且每一个‘**-f**’后面都带有参数，则所有指定的文件将连在一起作为 **makefile** 文件。

如果您不使用‘**-f**’或‘**--file**’选项，缺省的是按次序寻找‘**GNUmakefile**’，‘**makefile**’，和 ‘**Makefile**’，使用这三个中第一个能够找到的存在文件或能够创建的文件，参阅*编写 makefile 文件*。

9.2 指定最终目标的参数

最终目标（**gaol**）是 **make** 最终努力更新的目标。其它更新的目标是因为它们作为最终目标的依赖，或依赖的依赖，等等以此类推。

缺省情况下，**makefile** 文件中的第一个目标是最终目标（不计算那些以句点开始的目标）。因此，**makefile** 文件的第一个编译目标是对整个程序或程序组描述。如果第一个规则同时拥有几个目标，只有该规则的第一个目标是缺省的最终目标。

您可以使用 **make** 的参数指定最终目标。方法是使用目标的名字作为参数。如果您指定几个最终目标，**make** 按您命名时的顺序一个接一个的处理它们。

任何在 **makefile** 文件中出现的目标都能作为最终目标（除了以‘**-**’开始或含有‘**=**’的目标，它们一种解析为开关，另一种是变量定义）。即使在 **makefile** 文件中没有出现的目标，按照隐含规则可以说明怎样生成，也能指定为最终目标。

Make 将在命令行中使用特殊变量 **MAKECMGOALS** 设置您指定的最终目标。如果没有在命令行指定最终目标，该变量的值为空值。注意该变量值能在特殊场合下使用。

一个合适的例子是在清除规则中避免删除包括‘**.d**’的文件（参阅*自动产生依赖*），因这样 **make** 不会一创建它们，就立即又删除它们：

```
sources = foo.c bar.c

ifneq ($(MAKECMDGOALS),clean)
include $(sources:.c=.d)
endif
```

指定最终目标的一个用途是仅仅编译程序的一部分或程序组中的几个程序。如是这样，您可以将您希望变异的文件指定为最终目标。例如，在一个路径下包含几个程序，一个 **makefile** 文件以下面的格式开始：

```
.PHONY: all
all: size nm ld ar as
```

如果您仅对程序 **size** 编译，则您可以使用‘**make size**’命令，这样就只有您指定的程序才重新编译。

指定最终目标的另一个用途是编译产生哪些没有正常生成的文件。例如，又一个文件需要调试，或一个版本的程序需要编译进行测试，然而该文件不是 **makefile** 文件规则中缺省最终目标的依赖，此时，可以使用最终目标参数指定它们。

指定最终目标的另一个用途是运行和一个假想目标（参阅*假想目标*）或空目标（*使用空目标记录事件*）相联系的命令。许多 **makefile** 文件包含一个假想目标‘**clean**’删除除了原文件以外的所有文件。正常情况下，只有您具体指明使用‘**make clean**’命令，**make** 才能执行上述任务。下面列出典型的假想目标和空目标的名称。对 **GNU make** 软件包使用的所有标准目标名参阅*用户标准目标*：

‘**all**'
: 创建 **makefile** 文件的所有顶层目标。

`**clean**'
: 删除所有 **make** 正常创建的文件。

`**mostlyclean**'

象假象目标‘**clean**’，但避免删除人们正常情况下不重新建造的一少部分文件。例如，用于 **GCC** 的目标‘**mostlyclean**’不删除‘**libgcc.a**’，因为重建它的情况十分稀少，而且创建它又需要很多时间。

`**distclean**'

`**realclean**'

`**clobber**'

这些目标可能定义为比目标‘**clean**’ 删除更多的文件。例如，删除配置文件或为编译正常创建的准备文件，甚至 **makefile** 文件自身不能创建的文件。

‘**install**’

向命令搜寻目录下拷贝可执行文件；向可执行文件寻找目录下拷贝可执行文件使用的辅助文件。

‘**print**’

打印发生变化的文件列表。

‘**tar**’

创建源文件的压缩‘**tar**’文件。

‘**shar**’

为源文件创建一个 **shell** 的档案文件。

‘**dist**’

为源文件创建一个发布文件。这可能是‘**tar**’文件， ‘**shar**’文件，或多个上述的压缩版本文件。

‘**TAGS**’

更新该程序的‘**tags**’标签。

`**check**'

`**test**'

对该 **makefile** 文件创建的程序执行自我测试。

9.3 代替执行命令

makefile 文件告诉 **make** 怎样识别一个目标是否需要更新以及怎样更新每一个目标。但是更新目标并不是您一直需要的，一些特定的选项可以用来指定 **make** 的其它活动：

`**-n**'
`**--just-print**'
`**--dry-run**'
`**--recon**'
‘**No-op**’。**make** 的这项活动是打印用于创建目标所使用的命令，但并不执行它们。

`**-t**'
`**--touch**'
‘**touch**’。这项活动是做更新标志，实际却不更改它们。换句话说，**make** 假装编译了目标，但实际对它们没有一点儿改变。

`**-q**'
`**--question**'
‘**question**’。这项活动是暗中察看目标是否已经更新；但是任何情况下也不执行命令。换句话说，即不编译也不输出。

`**-W *file***'
`**--what-if=*file***'
`**--assume-new=*file***'
`**--new-file=*file***'
‘**What if** 。每一个‘**-W**’标志后跟一个文件名。所有文件名的更改时间被 **make** 记录为当前时间，但实际上更改时间保持不变。如果您要更新文件，您可以使用‘**-W**’标志和‘**-n**’标志连用看看将发生什么。

使用标志‘**-n**’，**make** 打印那些正常执行的命令，但却不执行它们。

使用标志‘**-t**’，**make** 忽略规则中的命令，对那些需要更新的目标使用‘**touch**’命令。如果不使用‘**-s**’或**.SILENT**，‘**touch**’命令同样打印。为了提高执行效率，**make** 并不实际调用程序 **touch**，而是使 **touch** 直接运行。

使用标志‘**-q**’，**make** 不打印输出也不执行命令，如果所有目标都已经更新到最新，**make** 的退出状态是 **0**；如果一部分需要更新，退出状态是 **1**；如果 **make** 遇到错误，退出状态是 **2**，因此您可以根据没有更新的目标寻找错误。

在运行 **make** 时对以上三个标志如果同时两个或三个将产生错误。标志‘**-n**’、‘**-t**’和‘**-s**’对那些以字符‘**+**’开始的命令行和包含字符串‘**$(MAKE)**'或‘**${MAKE}**'命令行不起作用。注意仅有这些以字符‘**+**’开始的命令行和包含字符串‘**$(MAKE)**'或‘**${MAKE}**'命令行运行时不注意这些选项。参阅***变量 MAKE 的工作方式***。

‘**-W**’标志有一下两个特点：

- 如果同时使用标志‘**-n**’或‘**-q**’，如果您更改一部分文件，看看 **make** 将会做什么。
- 没有使用标志‘**-n**’或‘**-q**’，如果 **make** 运行时采用标志‘**-W**’，则 **make** 假装所有文件已经更新，但实际上不更改任何文件。

注意选项‘**-p**’和‘**-v**’允许您得到更多的 **make** 信息或正在使用的 **makefile** 文件的信息（参阅***选项概要***）。

9.4 避免重新编译文件

有时您可能改变了一个源文件，但您并不希望编译所有依靠它的文件。例如，假设您在一个许多文件都依靠的头文件种添加了一个宏或一个声明，按照保守的规则，**make** 认为任何对于该头文件的改变，需要编译所有依靠它的文件，但是您知道那是不必要的，并且您没有等待它们完全编译的时间。

如果您提前了解改变头文件以前的问题，您可以使用‘**-t**’选项。该标志告诉 **make** 不

运行规则中的命令，但却将所有目标的时间戳改到最新。您可按下述步骤实现上述计划：
1、用 make 命令重新编译那些需要编译的源文件；
2、更改头文件；
3、使用‘make －t’命令改变所有目标文件的时间戳，这样下次运行 make 时就不会因为头文件的改变而编译任何一个文件。

如果在重新编译那些需要编译的源文件前已经改变了头文件，则按上述步骤做已显得太晚了；作为补救措施，您可以使用‘-o file’标志，它能将指定的文件的时间戳假装改为以前的时间戳（参阅*选项概要*）。这意味着该文件没有更改，因此您可按下述步骤进行：

1、使用‘make -o file’命令重新编译那些不是因为改变头文件而需要更新的文件。如果涉及几个头文件，您可以对每个头文件都使用‘-o’标志进行指定。

2、使用‘make －t’命令改变所有目标文件的时间戳。

9.5 变量重载

使用‘=’定义的变量：‘v=x’将变量 v 的值设为 x。如果您用该方法定义了一个变量，在 makefile 文件后面任何对该变量的普通赋值都将被 make 忽略，要使它们生效应在命令行将它们重载。

最为常见的方法是使用传递附加标志给编译器的灵活性。例如，在一个 makefile 文件中，变量 CFLAGS 已经包含了运行 C 编译器的每一个命令，因此，如果仅仅键入命令 make 时，文件‘foo.c’将按下面的方式编译：

```
cc -c $(CFLAGS) foo.c
```

这样您在 makefile 文件中对变量 CFALAGS 设置的任何影响编译器运行的选项都能生效，但是每次运行 make 时您都可以将该变量重载，例如：如果您说‘make CFLAGS='-g -O''，任何 C 编译器都将使用‘cc -c -g -O 编译程序。这还说明了在重载变量时，怎样使用 shell 命令中的引用包括空格和其它特殊字符在内的变量的值。

变量 CFALAGS 仅仅是您可以使用这种方式重载的许多标准变量中的一个，这些标准变量的完整列表见*隐含规则使用的变量*。

您也可以编写 makefile 察看您自己的附加变量，从而使用户可通过更改这些变量控制 make 运行时的其它面貌。

当您使用命令参数重载变量时，您可以定义递归调用扩展型变量或简单扩展型变量。上例中定义的是递归调用扩展型变量，如果定义简单扩展型变量，请使用‘:=’代替‘=’。注意除非您在变量值中使用变量引用或函数调用，这两种变量没有任何差异。

利用这种方式也可以改变您在 makfile 文件中重载的变量。在makfile 文件中重载的变量是使用 override 指令，是和‘override *variable* = *value*’相似的命令行。详细内容参阅 *override 指令*。

9.6 测试编译程序

正常情况下，在执行 shell 命令时一旦有错误发生，make 立即退出返回非零状态；不会为任何目标继续运行命令。错误表明 make 不能正确的创建最终目标，并且 make 一发现错误就立即报告。

当您编译您修改过的程序时，这不是您所要的结果。您希望 make 能够经可能的试着编译每一个程序，并尽可能的显示每一个错误。

这种情况下，您可以使用‘-k’或‘--keep-going’选项。这种选项告诉 make 遇到错误返回非零状态之前，继续寻找该目标的依赖，如果有必要则重新创建它们。例如，在编译一个目标文件时发现错误，即使 make 已经知道连接它们已是不可能的，‘make －k’也将继续编译其它目标文件。除在 shell 命令失败后继续运行外，即使发在 make 不知道如何创建的目标和依赖文件以后，‘make －k’也将尽可能的继续运行。在没有‘-k’选项时，这些错误将是致命的（参阅*选项概要*）。

通常情况下，make 的行为是基于假设您的目标是使最终目标更新；一旦它发现这是不

可能的它就立即报告错误。选项‘**-k**’说真正的目标是尽可能测试改变对程序的影响，发现存在的问题，以便在下次运行之前您可以纠正它们。这是 **Emacs M-x compile**命令缺省传递‘**-k**’选项的原因。

9.7 选项概要

下面是所有 **make** 能理解的选项列表：

`**-b**'
`**-m**'
和其它版本 **make** 兼容时，这些选项被忽略。

`**-C** ***dir***'
`**--directory=*dir***'
在将 **makefile** 读入之前，把路径切换到‘**dir**’下。如果指定多个‘**-C**’选项，每一个都是相对于前一个的解释：‘**-C/-C etc**’ 等同于‘**-C/etc**’。该选项典型用在递归调用 **make** 过程中，参阅*递归调用 make*。

‘**-d**’
在正常处理后打印调试信息。调试信息说明哪些文件用于更新，哪个文件作为比较时间戳的标准以及比较的结果，哪些文件实际上需要更新，需要考虑、使用哪些隐含规则等等----一切和 **make** 决定最终干什么有关的事情。‘**-d**’选项等同于‘**--debug=a**’选项（参见下面内容）。

`**--debug[=*options*]**'
在正常处理后打印调试信息。可以选择各种级别和类型的输出。如果没有参数，打印‘基本’级别的调试信息。以下是可能的参数，仅仅考虑第一个字母，各个值之间使用逗号或空格隔开：

a (*all*)
显示所有调试信息，该选项等同于‘**-d**’选项。

b (*basic*)
基本调试信息打印每一个已经过时的目标，以及它们重建是否成功。

v (*verbose*)
比‘基本’级高一个的等级的调试信息。包括 **makefile** 文件的语法分析结果，没有必要更新的依赖等。该选项同时包含基本调试信息。

i (*implicit*)
打印隐含规则搜寻目标的信息。该选项同时包含基本调试信息。

j (*jobs*)
打印各种子命令调用的详细信息。

m (*makefile*)
以上选项不包含重新创建 **makefile** 文件的信息。该选项包含了这方面的信息。注意，选项‘**all**’也不包含这方面信息。该选项同时包含基本调试信息。

`**-e**'
`**--environment-overrides**'
设置从环境中继承来的变量的优先权高于 **makefile** 文件中的变量的优先权。参阅*环境变量*。

`**-f** ***file***'
`**--file=*file***'
`**--makefile=*file***'
将名为‘**file**’的文件设置为 **makefile** 文件。参阅*编写 makefile 文件*。

`**-h**'
`**--help**'
向您提醒 **make** 能够理解的选项，然后退出。

`**-i**'
`**--ignore-errors**'
忽略重建文件执行命令时产生的所有错误。

`-I *dir*'
`--include-dir=*dir*'
指定搜寻包含 makefile 文件的路径‘dir’。参阅*包含其它 makefile 文件*。如果同时使用几个‘-I’选项用于指定路径，则按照指定的次序搜寻这些路径。
`-j [*jobs*]'
`--jobs[=*jobs*]'
指定同时执行的命令数目。如果没有参数 make 将同时执行尽可能多的任务；如果有多个‘-j’选项，则仅最后一个选项有效。详细内容参阅*并行执行*。注意在 MS-DOS 下，该选项被忽略。
`-k'
`--keep-going'
在出现错误后，尽可能的继续执行。当一个目标创建失败，则所有依靠它的目标文件将不能重建，而这些目标的其它依赖则可继续处理。参阅*测试编译程序*。
`-l [*load*]'
`--load-average[=*load*]'
`--max-load[=*load*]'
指定如果有其它任务正在运行，并且平均负载已接近或超过‘load’（一个浮点数），则此时不启动新任务。如果没有参数则取消以前关于负载的限制。参阅*并行执行*。
`-n'
`--just-print'
`--dry-run'
`--recon'
打印要执行的命令，但却不执行它们。参阅*代替执行命令*。
`-o *file*'
`--old-file=*file*'
`--assume-old=*file*'
即使文件 file 比它的依赖‘旧’，也不重建该文件。不要因为文件 file 的改变而重建任何其它文件。该选项本质上是假装将该文件的时间戳改为旧的时间戳，以至于依靠它的规则被忽略。参阅*避免重新编译文件*。
`-p'
`--print-data-base'
打印数据库（规则和变量的值），这些数据来自读入 makefile 文件的结果；然后象通常那样执行或按照别的指定选项执行。如果同时给出‘-v’开关，则打印版本信息（参阅下面内容）。使用‘make -qp’则打印数据库后不试图重建任何文件。使用‘make -p -f/dev/null’则打印预定义的规则和变量的数据库。数据库输出中包含文件名，以及命令和变量定义的行号信息。它是在复杂环境中很好的调试工具。
`-q'
`--question'
‘问题模式’。不打印输出也不执行命令，如果所有目标都已经更新到最新，make 的退出状态是 0；如果一部分需要更新，退出状态是 1；如果 make 遇到错误，退出状态是 2，参阅*代替执行命令*。
`-r'
`--no-builtin-rules'
排除使用内建的隐含规则（参阅*使用隐含规则*）。您仍然可以定义您自己的格式规则（参阅*定义和重新定义格式规则*）。选项‘-r’同时也清除了缺省的后缀列表和后缀规则（参阅*过时的后缀规则*）。但是您可以使用.SUFFIXES 规则定义您自己的后缀。注意，使用选项‘-r’仅仅影响规则；缺省变量仍然有效（参阅*隐含规则使用的变量*）；参阅下述的选项‘-R’。
`-R'
`--no-builtin-variables'
排除使用内建的规则变量（参阅*隐含规则使用的变量*）。当然，您仍然可以定义自己的变量。选项‘-R’自动使选项‘-r’生效；因为它去掉了隐含规则所使用的变量的定义，所以隐含规则也就失去了存在的意义。

`-s'
`--silent'
`--quiet'
沉默选项。不回显那些执行的命令。参阅*命令回显*。
`-S'
`--no-keep-going'
`--stop'
使选项‘-k’失效。除非在递归调用 make 时，通过变量 MAKEFLAGS 从上层 make 继承选项‘-k’，或您在环境中设置了选项‘-k’，否则没有必要使用该选项。
`-t'
`--touch'
标志文件已经更新到最新，但实际没有更新它们。这是假装那些命令已经执行，用于愚弄将来的 make 调用。参阅*代替执行命令*。
`-v'
`--version'
打印 make 程序的版本信息，作者列表和没有担保的注意信息，然后退出。
`-w'
`--print-directory'
打印执行 makefile 文件时涉及的所有工作目录。这对于跟踪 make 递归调用时复杂嵌套产生的错误非常有用。参阅*递归调用 make*。实际上，您很少需要指定该选项，因为 make 已经替您完成了指定。参阅 *‘--print-directory’选项*。
`--no-print-directory'
在指定选项‘-w’的情况下，禁止打印工作路径。这个选项在选项‘-w’自动打开而且您不想看多余信息时比较有用。参阅 *‘--print-directory’选项*。
`-W *file*'
`--what-if=*file*'
`--new-file=*file*'
`--assume-new=*file*'
假装目标文件已经更新。在使用标志‘n’时，它将向您表明更改该文件会发生什么。如果没有标志‘n’它和在运行 make 之前对给定的文件使用 touch 命令的结果几乎一样，但使用该选项 make 只是在的想象中更改该文件的时间戳。参阅*代替执行命令*。
`--warn-undefined-variables'
当 make 看到引用没有定义的变量时，发布一条警告信息。如果您按照复杂方式使用变量，当您调试您的 makefile 文件时，该选项非常有用。

10 使用隐含规则

重新创建目标文件的一些标准方法是经常使用的。例如，一个传统的创建 OBJ 文件的方法是使用 C 编译器，如 cc，编译 C 语言源程序。

隐含规则能够告诉 make 怎样使用传统的技术完成任务，这样，当您使用它们时，您就不必详细指定它们。例如，有一条编译 C 语言源程序的隐含规则，文件名决定运行哪些隐含规则；另如，编译 C 语言程序一般是使用‘.c’文件，产生‘.o’文件。因此， make 据此和文件名的后缀就可以决定使用编译 C 语言源程序的隐含规则。一系列的隐含规则可按顺序应用；例如，make 可以从一个‘.y’文件，借助‘.c’文件，重建一个‘.o’文件，参阅*隐含规则链*。内建隐含规则的命令需要使用变量，通过改变这些变量的值，您就可以改变隐含规则的工作方式。例如，变量 CFLAGS 控制隐含规则用于编译 C 程序传递给 C 编译器的标志，参阅*隐含规则使用的变量*。通过编写格式规则，您可以创建您自己的隐含规则。参阅*定*

义和重新定义格式规则。

后缀规则是对定义隐含规则最有限制性。格式规则一般比较通用和清楚，但是后缀规则却要保持兼容性。参阅*过时的后缀规则*。

10.1 使用隐含规则

允许 **make** 对一个目标文件寻找传统的更新方法，您所有做的是避免指定任何命令。可以编写没有命令行的规则或根本不编写任何规则。这样，**make** 将根据存在的源文件的类型或要生成的文件类型决定使用何种隐含规则。

例如，假设 **makefile** 文件是下面的格式：

```
foo : foo.o bar.o
        cc -o foo foo.o bar.o $(CFLAGS) $(LDFLAGS)
```

因为您提及了文件‘**foo.o**’，但是您没有给出它的规则，**make** 将自动寻找一条隐含规则，该规则能够告诉 **make** 怎样更新该文件。无论文件‘**foo.o**’存在与否，**make** 都会这样执行。

如果能够找到一条隐含规则，则它就能够对命令和一个或多个依赖（源文件）提供支持。如果您要指定附加的依赖，例如头文件，但隐含规则不能支持，您需要为目标‘**foo.o**’写一条不带命令行的规则。

每一条隐含规则都有目标格式和依赖格式；也许多条隐含规则有相同的目标格式。例如，有数不清的规则产生‘**.o**’文件：使用 **C** 编译器编译‘**.C**’文件；使用 **Pascal** 编译器编译‘**.p**’文件；等等。实际应用的规则是那些依赖存在或可以创建的规则。所以，如果您有一个‘**.C**’文件，**make** 将运行 **C** 编译器；如果您有一个‘**.p**’文件，**make** 将运行 **Pascal** 编译器；等等。

当然，您编写一个 **makefile** 文件时，您知道您要 **make** 使用哪一条隐含规则，以及您知道 **make** 将选择哪一条规则，因为您知道那个依赖文件是假设存在的。预定义的隐含规则列表的详细内容参阅*隐含规则目录*。

首先，我们说一条隐含规则可以应用，该规则的依赖必须‘存在或可以创建’。一个文件‘可以创建’是说该文件在 **makefile** 中作为目标或依赖被提及，或者该文件可以经过一条隐含规则的递归调用后能够创建。如果一条隐含规则的依赖是另一条隐含规则的结果，我们说产生了‘链’。参阅*隐含规则链*。

总体上说，**make** 为每一个目标搜寻隐含规则，为没有命令行的双冒号规则搜寻隐含规则。仅作为依赖被提及的文件，将被认为是一个目标，如果该目标的规则没有指定任何内容，**make** 将为它搜寻隐含规则。对于详细的搜寻过程参阅*隐含规则的搜寻算法*。

注意，任何具体的依赖都不影响对隐含规则的搜寻。例如，认为这是一条具体的规则：

```
foo.o: foo.p
```

文件 **foo.p** 不是首要条件，这意味着 **make** 按照隐含规则可以从一个 **Pascal** 源程序（‘**.p**’文件）创建 **OBJ** 文件，也就是说一个‘**.o**’文件可根据‘**.p**’文件进行更新。但文件 **foo.p** 并不是绝对必要的；例如，如果文件 **foo.c** 也存在，按照隐含规则则是从文件 **foo.c** 重建 **foo.o**，这是因为 **C** 编译规则在预定义的隐含规则列表中比 **Pascal** 规则靠前，参阅*隐含规则目录*。

如果您不希望使用隐含规则创建一个没有命令行的目标，您可以通过添加分号为该目标指定空命令。参阅*使用空命令*。

10.2 隐含规则目录

这里列举了预定义的隐含规则的目录，这些隐含规则是经常应用的，当然如果您在 **makefile** 文件中重载或删除后，这些隐含规则将会失去作用，详细内容参阅*删除隐含规则*。选项‘**-r**’或‘**--no-builtin-rules**’将删除所有预定义的隐含规则。

并不是所有的隐含规则都是预定义的，在 **make** 中很多预定义的隐含规则是后缀规则的扩展，因此，那些预定义的隐含规则和后缀规则的列表相关（特殊目标**.SUFFIXES** 的依赖列表）。缺省的后缀列表

为：.out, .a, .ln, .o, .c, .cc, .C, .p, .f, .F, .r, .y, .l, .s, .S, .mod, .sym, .def, .h, .info, .dvi, .tex, .texinfo, .texi, .txinfo, .w, .ch , .web, .sh, .elc, .el 。所有下面描述的隐含规则，如果它们的依赖中有一个出现在这个后缀列表中，则是后缀规则。如果您更改这个后缀列表，则只有那些由一个或两个出现在您指定的列表中的后缀命名的预定义后缀规则起作用；那些后缀没有出现在列表中的规则被禁止。对于详细的关于后缀规则的描述参阅*过时的后缀规则*。

Compiling C programs（编译 **C** 程序）

‘***n*.o'** 自动由‘***n*.c'** 使用命令 ‘**$(CC) -c $(CPPFLAGS) $(CFLAGS)'**生成 。

Compiling C++ programs （编译 **C++**程序）

‘***n*.o'**自动由‘***n*.cc'** 或‘***n*.C'**使用命令‘**$(CXX) -c $(CPPFLAGS) $(CXXFLAGS)**’生成。 我们鼓励您对 **C++**源文件使用后缀‘**.cc'** 代替后缀‘**.C**’。

Compiling Pascal programs （编译 **Pascal** 程序）

‘***n*.o'**自动由‘***n*.p'**使用命令‘**$(PC) -c $(PFLAGS)'**生成。

Compiling Fortran and Ratfor programs （编译 **Fortran** 和 **Ratfor** 程序）

‘***n*.o'**自动由‘***n*.r', ‘*n*.F'**或‘***n*.f'** 运行 **Fortran** 编译器生成。使用的精确命令如下：

```
`.f'
`$(FC) -c $(FFLAGS)'.
`.F'
`$(FC) -c $(FFLAGS) $(CPPFLAGS)'.
`.r'
`$(FC) -c $(FFLAGS) $(RFLAGS)'.
```

Preprocessing Fortran and Ratfor programs （预处理 **Fortran** 和 **Ratfor** 程序）

‘***n*.f'** 自动从 ‘***n*.r'**或‘***n*.F'**得到。该规则仅仅是与处理器把一个 **Ratfor** 程序或能够预处理的 **Fortran** 程序转变为标准的 **Fortran** 程序。使用的精确命令如下：

```
`.F'
`$(FC) -F $(CPPFLAGS) $(FFLAGS)'.
`.r'
`$(FC) -F $(FFLAGS) $(RFLAGS)'.
```

Compiling Modula-2 programs（编译 **Modula-2** 程序）

‘***n*.sym'**自动由‘***n*.def'**使用命令‘**$(M2C) $(M2FLAGS) $(DEFFLAGS)**’生成。 ‘***n*.o'** 从‘***n*.mod'**生成；命令为**:** ‘**$(M2C) $(M2FLAGS) $(MODFLAGS)'**。

Assembling and preprocessing assembler programs （汇编以及预处理汇编程序）

‘***n*.o'**自‘***n*.S'**运行 **C** 编译器，**cpp**，生成。命令为**:** ‘**$(CPP) $(CPPFLAGS)'**。

Linking a single object file （连接一个简单的 **OBJ** 文件）

‘***n*'** 自动由‘***n*.o'** 运行 **C** 编译器中的连接程序 **linker (**通常称为 **ld)**生成。命令为**:** ‘**$(CC) $(LDFLAGS) *n*.o $(LOADLIBES) $(LDLIBS)'**。该规则对仅有一个源程序的简单程序或对同时含有多个 **OBJ** 文件（可能来自于不同的源文件）的程序都能正常工作。如果同时含有多个 **OBJ** 文件，则其中必有一个 **OBJ** 文件的名字和可执行文件名匹配。例如：

```
x: y.o z.o
```

当‘**x.c', ‘y.c'** 和‘**z.c'** 都存在时则执行：

```
cc -c x.c -o x.o
cc -c y.c -o y.o
cc -c z.c -o z.o
cc x.o y.o z.o -o x
rm -f x.o
rm -f y.o
rm -f z.o
```

对于更复杂的情况，例如没有一个 **OBJ** 文件的名字和可执行文件名匹配，您必须为连接写一条具体的命令。每一种能自动生成‘**.o**’的文件，可以在没有‘**-c**’选项的情况下使用编译

器(‘$(CC)', ‘$(FC)'或‘$(PC)'； C编译器‘$(CC)'也适用于汇编程序)自动连接。当然也可以使用OBJ文件作为中间文件，但编译、连接一步完成速度将快很多。

Yacc for C programs （由Yacc生成C程序）

‘*n*.c'自动由‘*n*.y'使用命令‘$(YACC) $(YFLAGS)'运行 Yacc生成。

Lex for C programs （由Lex生成C程序）

‘*n*.c'自动由‘*n*.l' 运行 Lex生成。命令为：‘$(LEX) $(LFLAGS)'。

Lex for Ratfor programs （由Lex生成Rator程序）

‘*n*.r'自动由‘*n*.l' 运行 Lex生成。命令为：‘$(LEX) $(LFLAGS)'。 对于所有的Lex文件，无论它们产生C代码或Ratfor 代码，都使用相同的后缀‘.l’进行转换，在特定场合下，使用make自动确定您使用哪种语言是不可能的。如果make使用‘.l’文件重建一个OBJ文件，它必须猜想使用哪种编译器。它很可能猜想使用的是 C 编译器，因为C 编译器更加普遍。如果您使用 Ratfor语言，请确保在makefile文件中提及了‘*n*.r'，使 make知道您的选择。否则，如果您专用 Ratfor语言，不使用任何C 文件，请在隐含规则后缀列表中将‘.c’剔除：

```
.SUFFIXES:
.SUFFIXES: .o .r .f .l ...
```

Making Lint Libraries from C, Yacc, or Lex programs（由C, Yacc, 或 Lex程序创建Lint库）

‘*n*.ln' 可以从‘*n*.c' 运行 lint 产生。命令为：‘ $(LINT) $(LINTFLAGS) $(CPPFLAGS) -i’。用于C程序的命令和用于‘*n*.y'或‘*n*.l'程序相同。

TeX and Web（TeX 和 Web）

‘*n*.dvi'可以从‘*n*.tex' 使用命令‘$(TEX)'得到。‘*n*.tex'可以从‘*n*.web'使用命令‘$(WEAVE)'得到；或者从‘*n*.w' (和‘*n*.ch'，如果‘*n*.ch'存在或可以建造) 使用命令‘$(CWEAVE)'。‘*n*.p' 可以从‘*n*.web'使用命令‘$(TANGLE)'产生。‘*n*.c' 可以从‘*n*.w' (和‘*n*.ch'，如果‘*n*.ch'存在或可以建造) 使用命令‘$(CTANGLE)'得到。

Texinfo and Info（Texinfo和Info）

‘*n*.dvi'可以从‘*n*.texinfo', ‘*n*.texi', 或‘*n*.txinfo', 使用命令‘$(TEXI2DVI) $(TEXI2DVI_FLAGS)' 得到。‘*n*.info' 可以从‘*n*.texinfo', ‘*n*.texi', 或‘*n*.txinfo', 使用命令‘$(MAKEINFO) $(MAKEINFO_FLAGS)'创建。

RCS

文件‘*n*'必要时可以从名为‘*n*,v'或‘RCS/*n*,v'的 RCS 文件中提取。具体命令是：‘$(CO) $(COFLAGS)'。文件‘*n*'如果已经存在，即使 RCS 文件比它新，它不能从RCS 文件中提取。用于 RCS 的规则是最终的规则，参阅*万用规则*，所以 RCS 不能够从任何源文件产生，它们必须存在。

SCCS

文件‘*n*'必要时可以从名为‘s.*n*'或‘SCCS/s.*n*'的SCCS文件中提取。具体命令是：‘$(GET) $(GFLAGS)'。用于 SCCS 的规则是最终的规则，参阅*万用规则*，所以 SCCS不能够从任何源文件产生，它们必须存在。SCCS 的优点是，文件 ‘*n*' 可以从文件‘*n*.sh'拷贝并生成可执行文件(任何人都可以)。这用于 shell 的脚本，该脚本在SCCS 内部检查。因为 RCS 允许保持文件的可执行性，所以您没有必要将该特点用于RCS 文件。我们推荐您避免使用 SCCS，RCS 不但使用广泛，而且是免费的软件。选择自由软件代替相当的（或低劣的）收费软件，是您对自由软件的支持。

通常情况下，您要仅仅改变上表中的变量，需要参阅下面的文档。

隐含规则的命令实际使用诸如 COMPILE.c, LINK.p, 和 PREPROCESS.S 等等变量，它们的值包含以上列出的命令。Make 按照惯例进行处理，如，编译‘.x’源文件的规则使用变量‘COMPILE.*x*’；从‘.x’源文件生成可执行文件使用变量‘LINK.*x*’；预处理‘.x’源文件使用变量‘PREPROCESS.*x*’。

任何产生OBJ文件的规则都使用变量‘OUTPUT_OPTION’；make依据编译时间选项定义该变量的值是‘-o $@’或空值。当源文件分布在不同的目录中，您应该使用‘-O’选项保证输出到正确的文件中；使用变量VPATH时同样（参阅为*依赖搜寻目录*）。一些系统的编译器

不接受针对 OBJ 文件的‘-o’开关；如果您在这样的系统上运行，并使用了变量 VPATH，一些文件的编译输出可能会放到错误的地方。解决办法是将变量 OUTPUT_OPTION 值设为：‘; mv $*.o $@’。

10.3 隐含规则使用的变量

内建隐含规则的命令对预定义变量的使用是开放的；您可以在 makefile 文件中改变变量的值，也可以使用 make 的运行参数或在环境中改变，如此，在不对这些规则本身重新定义的情况下，就可以改变这些规则的工作方式。您还可以使用选项‘-R’或‘--no-builtin-variables’删除所有隐含规则使用的变量。

例如，编译 C 程序的命令实际是‘$(CC) -c $(CFLAGS) $(CPPFLAGS)’，变量缺省的值是‘cc’或空值，该命令实际是‘cc -c’。如重新定义变量‘CC’的值为‘ncc’，则所有隐含规则将使用‘ncc’作为编译 C 语言源程序的编译器。通过重新定义变量‘CFLAGS’的值为‘-g’，则您可将‘-g’选项传递给每个编译器。所有的隐含规则编译 C 程序时都使用‘$CC’获得编译器的名称，并且都在传递给编译器的参数中都包含‘$(CFLAGS)’。

隐含规则使用的变量可分为两类：一类是程序名变量（象 cc），另一类是包含程序运行参数的变量（象 CFLAGS）。(‘程序名’可能也包含一些命令参数，但是它必须以一个实际可以执行的程序名开始。) 如果一个变量值中包含多个参数，它们之间用空格隔开。

这里是内建规则中程序名变量列表：

AR
: 档案管理程序；缺省为：‘ar'.

AS
: 汇编编译程序；缺省为：‘as'.

CC
: C 语言编译程序；缺省为：‘cc'.

CXX
: C++编译程序；缺省为：‘g++'.

CO
: 从 RCS 文件中解压缩抽取文件程序；缺省为：‘co'.

CPP
: 带有标准输出的 C 语言预处理程序；缺省为：‘$(CC) -E'.

FC
: Fortran 以及 Ratfor 语言的编译和预处理程序；缺省为：‘f77'.

GET
: 从 SCCS 文件中解压缩抽取文件程序；缺省为：‘get'.

LEX
: 将 Lex 语言转变为 C 或 Ratfor 程序的程序；缺省为：‘lex'.

PC
: Pascal 程序编译程序；缺省为：‘pc'.

YACC
: 将 Yacc 语言转变为 C 程序的程序；缺省为：‘yacc'.

YACCR
: 将 Yacc 语言转变为 Ratfor 程序的程序；缺省为：‘yacc -r'.

MAKEINFO
: 将 Texinfo 源文件转换为信息文件的程序；缺省为：‘makeinfo'.

TEX
: 从 TeX 源产生 TeX DVI 文件的程序；缺省为：‘tex'.

TEXI2DVI
: 从 Texinfo 源产生 TeX DVI 文件的程序；缺省为：‘texi2dvi'.

WEAVE
: 将 Web 翻译成 TeX 的程序；缺省为：‘weave'.

CWEAVE
将 **CWeb** 翻译成 **TeX** 的程序；缺省为：‘**cweave'.**
TANGLE
将 **Web** 翻译成 **Pascal** 的程序；缺省为：‘**tangle'.**
CTANGLE
将 **Web** 翻译成 **C** 的程序；缺省为：‘**ctangle'.**
RM
删除文件的命令；缺省为：‘**rm -f'.**
这里是值为上述程序附加参数的变量列表。在没有注明的情况下，所有变量的值为空值。
ARFLAGS
用于档案管理程序的标志，缺省为：‘**rv'.**
ASFLAGS
用于汇编编译器的额外标志（当具体调用‘**.s'**或‘**.S'**文件时）。
CFLAGS
用于 **C** 编译器的额外标志。
CXXFLAGS
用于 **C++**编译器的额外标志。
COFLAGS
用于 **RCS co** 程序的额外标志。
CPPFLAGS
用于 **C** 预处理以及使用它的程序的额外标志（**C** 和 **Fortran** 编译器）。
FFLAGS
用于 **Fortran** 编译器的额外标志。
GFLAGS
用于 **SCCS get** 程序的额外标志。
LDFLAGS
用于调用 **linker**（‘**ld**’）的编译器的额外标志。
LFLAGS
用于 **Lex** 的额外标志。
PFLAGS
用于 **Pascal** 编译器的额外标志。
RFLAGS
用于处理 **Ratfor** 程序的 **Fortran** 编译器的额外标志。
YFLAGS
用于 **Yacc** 的额外标志。**Yacc**。

10.4 隐含规则链

有时生成一个文件需要使用多个隐含规则组成的序列。例如，从文件‘**n.y**’生成文件‘**n.o**’，首先运行隐含规则 **Yacc**，其次运行规则 **cc**。这样的隐含规则序列称为隐含规则链。

如果文件‘**n.c**’存在或在 **makefile** 文件中提及，则不需要任何特定搜寻：**make** 首先发现通过 **C** 编译器编译‘**n.c**’可生成该 **OBJ** 文件，随后，考虑生成‘**n.c**’时，则使用运行 **Yacc** 的规则。这样可最终更新‘**n.c**’和‘**n.o**’。

即使在文件‘**n.c**’不存在或在 **makefile** 文件中没有提及的情况下，**make** 也能想象出在文件‘**n.y**’和‘**n.o**’缺少连接！这种情况下，‘**n.c**’称为**中间文件**。一旦 **make** 决定使用中间文件，它将把中间文件输入数据库，好像中间文件在 **makefile** 文件中提及一样；按照隐含规则的描述创建中间文件。

中间文件和其它文件一样使用自己的规则重建，但是中间文件和其它文件相比有两种不同的处理方式。

第一个不同的处理方式是如果中间文件不存在 **make** 的行为不同：平常的文件 **b** 如果不存在，**make** 认为一个目标依靠文件 **b**，它总是创建文件 **b**，然后根据文件 **b** 更新目标；但是

文件 **b** 若是中间文件，**make** 很可能不管它而进行别的工作，即不创建文件 **b**，也不更新最终目标。只有在文件 **b** 的依赖比最终目标‘新’时或有其它原因时，才更新最终目标。

第二个不同点是 **make** 在更新目标创建文件 **b** 后，如果文件 **b** 不再需要，**make** 将把它删除。所以一个中间文件在 **make** 运行之前和 **make** 运行之后都不存在。**Make** 向您报告删除时打印一条‘**rm -f**’命令，表明有文件被删除。

通常情况下，任何在 **makefile** 文件中提及的目标和依赖都不是中间文件。但是，您可以特别指定一些文件 为中间文件，其方法为：将要指定为中间文件的文件作 为特殊目标 **.INTERMEDIATE** 的依赖。这种方法即使对采用别的方法具体提及的文件也能生效。

您通过将文件标志为 **secondary** 文件可以阻止自动删除中间文件。这时，您将您需要保留的中间文件指定为特殊目标 **.SECONDARY** 的依赖即可。对于 **secondary** 文件，**make** 不会因为它不存在而去创建它，也不会自动删除它。**secondary** 文件必须也是中间文件。

您可以列举一个隐含规则的目标格式（例如 **%.o**）作为特殊目标 **.PRECIOUS** 的依赖，这样您就可以保留那些由隐含规则创建的文件名匹配该格式的中间文件。参阅 ***中断和关闭 make***。

一个隐含规则链至少包含两个隐含规则。例如，从‘**RCS/foo.y,v**’创建文件‘**foo**’需要运行 **RCS**、**Yacc** 和 **cc**，文件 **foo.y** 和 **foo.c** 是中间文件，在运行结束后将被删掉。

没有一条隐含规则可以在隐含规则链中出现两次以上（含两次）。这意味着，**make** 不会简单的认为从文件‘**foo.o.o**’创建文件 **foo** 不是运行 **linker** 两次。这还可以强制 **make** 在搜寻一个隐含规则链时阻止无限循环。

一些特殊的隐含规则可优化隐含规则链控制的特定情况。例如，从文件 **foo.c** 创建文件 **foo** 可以被拥有编译和连接的规则链控制，它使用 **foo.o** 作为中间文件。但是对于这种情况存在一条特别的规则，使用简单的命令 **cc** 可以同时编译和连接。因为优化规则在规则表中的前面，所以优化规则和一步一步的规则链相比，优先使用优化规则。

10.5 定义与重新定义格式规则

您可以通过编写格式规则定义隐含规则。该规则看起来和普通规则类似，不同之处在于格式规则的目标中包含字符‘**%**’（只有一个）。目标是匹配文件名的格式；字符‘**%**’可以匹配任何非空的字符串，而其它字符仅仅和它们自己相匹配。依赖用‘**%**’表示它们的名字和目标名关联。

格式‘**%.o : %.c**’是说将任何‘**stem.c**’文件编译为‘**stem.o**’文件。

在格式规则中使用的‘**%**’扩展是在所有变量和函数扩展以后进行的，它们是在 **makefile** 文件读入时完成的。参阅 ***使用变量***和 ***转换文本函数***。

10.5.1 格式规则简介

格式规则是在目标中包含字符‘**%**’（只有一个）的规则，其它方面看起来和普通规则相同。目标是可以匹配文件名的格式，字符‘**%**’可以匹配任何非空的字符串，而其它字符仅仅和它们自己相匹配。

例如‘**%.c**’匹配任何以‘**.c**’结尾的文件名；‘**s.%.c**’匹配以‘**s.**’开始并且以‘**.c**’结尾的文件名，该文件名至少包含 **5** 个字符（因为‘**%**’至少匹配一个字符）。匹配‘**%**’的子字符串称为 **stem(**径**)**。依赖中使用‘**%**’表示它们的名字中含有和目标名相同的 **stem**。要使用格式规则，文件名必须匹配目标的格式，而且符合依赖格式的文件必须存在或可以创建。下面规则：

%.o : %.c ; *command*...

表明要创建文件‘**n.o**’，使用‘**n.c**’作为它的依赖，而且文件‘**n.c**’ 必须存在或可以创建。

在格式规则中，依赖有时不含有‘**%**’。这表明采用该格式规则创建的所有文件都是采用相同的依赖。这种固定依赖的格式规则在有些场合十分有用。

格式规则的依赖不必都包含字符‘%’，这样的规则是一个有力的常规通配符，它为任何匹配该目标格式规则的文件提供创建方法。参阅*定义最新类型的缺省规则*。

格式规则可以有多个目标，不象正常的规则，这种规则不能扮演具有相同依赖和命令的多条不同规则。如果一格式规则具有多个目标，**make** 知道规则的命令对于所有目标来说都是可靠的，这些命令只有在创建所目标时才执行。当为匹配一目标搜寻格式规则时，规则的目标格式和规则要匹配的目标不同是十分罕见的，所以 **make** 仅仅担心目前对文件给出命令和依赖是否有问题。注意该文件的命令一旦执行，所有目标的时间戳都会更新。

格式规则在 **makefile** 文件中的次序很重要，因为这也是考虑它们的次序。对于多个都能使用的规则，使用最先出现的规则。您亲自编写的规则比内建的规则优先。注意依赖存在或被提及的规则优先于依赖需要经过隐含规则链生成的规则。

10.5.2 格式规则的例子

这里有一些实际在 **make** 中预定义的格式规则例子，第一个，编译‘**.c**’文件生成‘**.o**’文件的规则：

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

定义了一条编译‘**x.c**’文件生成‘**x.o**’文件的规则，命令使用自动变量‘**$@**’和‘**$<**’ 替换任何情况使用该规则的目标文件和源文件。参阅*自动变量*。

第二个内建的例子：

```
% :: RCS/%,v
        $(CO) $(COFLAGS) $<
```

定义了在子目录‘**RCS**’中根据相应文件‘**x.v**’生成文件‘**x**’的规则。因为目标是‘%’，只要相对应的依赖文件存在，该规则可以应用于任何文件。双冒号表示该规则是最终规则，它意味着不能是中间文件。参阅*万用规则*。

下面的格式规则有两个目标：

```
%.tab.c %.tab.h: %.y
        bison -d $<
```

这告诉 **make** 执行命令‘**bison -d *x*.y**’将创建‘**x.tab.c**’和‘**x.tab.h**’。如果文件 **foo** 依靠文件‘**parse.tab.o**’和‘**scan.o**’，而文件‘**scan.o**’又依靠文件‘**parse.tab.h**’，当‘**parse.y**’发生变化，命令‘**bison -d parse.y**’执行一次。‘**parse.tab.o**’和‘**scan.o**’的依赖也随之更新。假设文件‘**parse.tab.o**’由文件‘**parse.tab.c**’编译生成，文件‘**scan.o**’由文件‘**scan.c**’生成，当连接‘**parse.tab.o**’、‘**scan.o**’和其它依赖生成文件 **foo** 时，上述规则能够很好执行。）

10.5.3 自动变量

假设您编写一个编译‘**.c**’文件生成‘**.o**’文件的规则：您怎样编写命令‘**CC**’，使它能够操作正确的文件名？您当然不能将文件名直接写进命令中，因为每次使用隐含规则操作的文件名都不一样。

您应该使用 **make** 的另一个特点，自动变量。这些变量在规则每次执行时都基于目标和依赖产生新值。例如您可以使用变量‘**$@**’代替目标文件名，变量‘**$<**’代替依赖文件名。

下面是自动变量列表：

$@

规则的目标文件名。如果目标是一个档案成员，则变量‘**$@**’ 档案文件的文件名。对于有多个目标的格式规则（参阅*格式规则简介*），变量‘**$@**’是那个导致规则命令运行的目标文件名。

$%

当目标是档案成员时，该变量是目标成员名，参阅*使用 make 更新档案文件*。例如，

如果目标是‘**foo.a(bar.o)'**，则‘**$%'**的值是‘**bar.o'**，‘**$@'**的值是‘**foo.a'**。如果目标不是档案成员，则‘**$%'**是空值。

$<

第一个依赖的文件名。如果目标更新命令来源于隐含规则，该变量的值是隐含规则添加的第一个依赖。参阅*使用隐含规则*。

$?

所有比目标‘新’的依赖名，名字之间用空格隔开。对于为档案成员的依赖，只能使用已命名的成员。参阅*使用 make 更新档案文件*。

$^

所有依赖的名字，名字之间用空格隔开。对于为档案成员的依赖，只能使用已命名的成员。参阅*使用 make 更新档案文件*。对同一个目标来说，一个文件只能作为一个依赖，不管该文件的文件名在依赖列表中出现多少次。所以，如果在依赖列表中，同一个文件名出现多次，变量‘**$^**’的值仍然仅包含该文件名一次。

$+

该变量象‘**$^'**，但是，超过一次列出的依赖将按照它们在 **makefile** 文件中出现的次序复制。这主要的用途是对于在按照特定顺序重复库文件名很有意义的地方使用连接命令。

$*

和隐含规则匹配的 **stem(**径**)**，参阅*格式匹配*。如果一个目标为‘**dir/a.foo.b'**，目标格式规则为：‘**a.%.b'** ，则 **stem** 为‘**dir/foo'**。在构建相关文件名时 **stem** 十分有用。在静态格式规则中，**stem** 是匹配目标格式中字符‘%’的文件名中那一部分。在一个没有 **stem** 具体规则中；变量‘**$*'** 不能以该方法设置。如果目标名以一种推荐的后缀结尾（参阅*过时的后缀规则*）,变量‘**$*'**设置为目标去掉该后缀后的部分。例如，如果目标名是‘**foo.c'**，则变量‘**$*'** 设置为‘**foo',** 因为‘**.c'** 是一个后缀。**GNU make** 处理这样奇怪的事情是为了和其它版本的 **make** 兼容。在隐含规则和静态格式规则以外，您应该尽量避免使用变量‘**$*'**。在具体规则中如果目标名不以推荐的后缀结尾，则变量‘**$***’在该规则中设置为空值。

当您希望仅仅操作那些改变的依赖，变量‘**$?'** 即使在具体的规则中也很有用。例如，假设名为‘**lib**’的档案文件包含几个 **OBJ** 文件的拷贝，则下面的规则仅将发生变化的 **OBJ** 文件拷贝到档案文件：

```
lib: foo.o bar.o lose.o win.o
        ar r lib $?
```

在上面列举的变量中，有四个变量的值是单个文件名。三个变量的值是文件名列表。这七个变量都有仅仅存放文件的路径名或仅仅存放目录下文件名的变体。变量的变体名是由变量名追加字母‘**D**’或‘**F**’构成。这些变体在 **GNU make**中处于半废状态，原因是使用函数 **T dir** 和 **notdir** 能够得到相同的结果。参阅*文件名函数*。注意，‘**F'**变体省略所有在 **dir** 函数中总是输出的结尾斜杠这里是这些变体的列表：

`$(@D)'

目标文件名中的路径部分，结尾斜杠已经移走。如果变量**`$@'**的值是**`dir/foo.o'**，变体 **`$(@D)'**的值是**`dir'**。 如果变量**`$@'**的值不包含斜杠，则变体的值是**`.'**。

`$(@F)'

目标文件名中的真正文件名部分。如果变量**`$@'**的值是**`dir/foo.o'**,变体 **`$(@F)'** 的值是**` foo.o '**。**`$(@F)'** 等同于 **`$(notdir $@)'**。

`$(*D)'
`$(*F)'

stem（径）中的路径名和文件名；在这个例子中它们的值分别为：**`dir'** 和 **`foo'** 。

`$(%D)'

`$(%F)'
档案成员名中的路径名和文件名；这仅对采用‘*archive*(*member*)’形式的档案成员目标有意义，并且当成员包含路径名时才有用。参阅*档案成员目标*。

`$(<D)'
`$(<F)'
第一个依赖名中的路径名和文件名。

`$(^D)'
`$(^F)'
所有依赖名中的路径名和文件名列表。

`$(?D)'
`$(?F)'
所有比目标‘新’的依赖名中的路径名和文件名列表。

注意，在我们讨论自动变量时，我们使用了特殊格式的惯例：我们写"the value of‘$<'",而不是"the variable <"；和我们写普通变量，例如变量 objects 和 CFLAGS 一样。我们认为这种惯例在这种情况下看起来更加自然。这并没有其它意义，变量‘$<'的变量名为 < 和变量‘$(CFLAGS)' 实际变量名为 CFLAGS 一样。您也可以使用‘$(<)'代替‘$<'。

10.5.4 格式匹配

目标格式是由前缀、后缀和它们之间的通配符%组成，它们中的任一个或两个都可以是空值。格式匹配一个文件名只有该文件名是以前缀开始，后缀结束，而且两者不重叠的条件下，才算匹配。前缀、后缀之间的文本成为径（stem）。当格式‘%.o’匹配文件名‘test.o’时，径（stem）是‘test’。格式规则中的依赖将径（stem）替换字符%，从而得出文件名。对于上例中，如果一个依赖为‘%.c’，则可扩展为‘test.c’。

当目标格式中不包含斜杠（实际并不是这样），则文件名中的路径名首先被去除，然后，将其和格式中的前缀和后缀相比较。在比较之后，以斜杠结尾的路径名，将会加在根据格式规则的依赖规则产生的依赖前面。只有在寻找隐含规则时路径名才被忽略，在应用时路径名绝不能忽略。例如，‘e%t’和文件名‘src/eat’匹配，stem(径)是‘src/a’。当依赖转化为文件名时，stem 中的路径名将加在前面，stem(径)的其余部分替换‘%’。使用stem（径）‘src/a’和依赖格式规则‘c%r’匹配得到文件名‘src/car’。

10.5.5 万用规则

一个格式规则的目标仅仅包含‘%’，它可以匹配任何文件名，我们称这些规则为万用规则。它们非常有用，但是 make 使用它们的耗时也很多，因为 make 必须为作为目标和作为依赖列出的每一个文件都考虑这样的规则。

假设 makefile 文件提及了文件 foo.c。为了创建该目标，make 将考虑是通过连接一个 OBJ 文件‘foo.c.o’创建，或是通过使用一步的 C 编译连接程序从文件 foo.c.c 创建，或是编译连接 Pascal 程序 foo.c.p 创建，以及其它的可能性等。

我们知道 make 考虑的这些可能性是很可笑的，因为 foo.c 就是一个 C 语言源程序，不是一个可执行程序。如果make 考虑这些可能性，它将因为这些文件诸如 foo.c.o 和 foo.c.p 等都不存在最终拒绝它们。但是这些可能性太多，所以导致 make 的运行速度极慢。

为了加快速度，我们为 make 考虑匹配万用规则的方式设置了限制。有两种不同类型的可以应用的限制，在您每次定义一个万用规则时，您必须为您定义的规则在这两种类型中选择一种。

一种选择是标志该万用规则是最终规则，即在定义时使用双冒号定义。一个规则为最终规则时，只有在它的依赖存在时才能应用，即使依赖可以由隐含规则创建也不行。换句话说，在最终规则中没有进一步的链。

例如，从 **RCS** 和 **SCCS** 文件中抽取原文件的内建的隐含规则是最终规则，则如果文件‘**foo.c,v'** 不存在，**make** 绝不会试图从一个中间文件‘**foo.c,v.o**’或‘**RCS/SCCS/s.foo.c,v**’在创建它。 **RCS** 和 **SCCS** 文件一般都是最终源文件，它不能从其它任何文件重新创建，所以，**make** 可以记录时间戳，但不寻找重建它们的方式。

如果您不将万用规则标志为最终规则，那么它就是非最终规则。一个非最终万用规则不能用于指定特殊类型数据的文件。如果存在其它规则（非万用规则）的目标匹配一文件名，则该文件名就是指定特殊类型数据的文件名。

例如，文件名‘**foo.c'** 和格式规则 **`%.c : %.y' (**该规则运行 **Yacc)**。无论该规则是否实际使用**(**如果碰巧存在文件‘**foo.y**’，该规则将运行**)**，和目标匹配的事实就能足够阻止任何非最终万用规则在文件 **foo.c** 上使用。这样，**make** 考虑就不试图从文件‘**foo.c.o', ‘foo.c.c', ‘foo.c.p'**等创建可执行的‘**foo.c'**。

内建的特殊伪格式规则是用来认定一些特定的文件名，处理这些文件名的文件时不能使用非最终万用规则。这些伪格式规则没有依赖和命令，它们用于其它目的时被忽略。例如，内建的隐含规则：

%.p :

存在可以保证 **Pascal** 源程序如‘**foo.p'** 匹配特定的目标格式，从而阻止浪费时间寻找‘**foo.p.o'** 或‘**foo.p.c'**。

在后缀规则中，为后缀列表中的每一个有效后缀都创建了伪格式规则，如‘**%.p'** 。参阅***过时的后缀规则***。

10.5.6 删除隐含规则

通过定义新的具有相同目标和依赖但不同命令的规则，您可以重载内建的隐含规则（或重载您自己定义的规则）。一旦定义新的规则，内建的规则就被代替。 新规则在隐含规则次序表中的位置由您编写规则的地方决定。

通过定义新的具有相同目标和依赖但不含命令的规则，您可以删除内建的隐含规则。例如，下面的定义规则将删除运行汇编编译器的隐含规则：

%.o : %.s

10.6 定义最新类型的缺省规则

您通过编写不含依赖的最终万用格式规则，您可以定义最新类型的缺省规则。参阅***万用规则***。这和其它规则基本一样，特别之处在于它可以匹配任何目标。因此，这样的规则的命令可用于所有没有自己的命令的目标和依赖，以及用于那些没有其它隐含规则可以应用的目标和依赖。

例如，在测试 **makefile** 时，您可能不关心源文件是否含有真实数据，仅仅关心它们是否存在。那么，您可以这样做：

```
%::
        touch $@
```

这导致所有必需的源文件（作为依赖）都自动创建。

您可以为没有规则的目标以及那些没有具体指定命令的目标定义命令。要完成上述任务，您需要为特殊目标**.DEFAULT** 编写规则。这样的规则可以在所有具体规则中用于没有作为目标出现以及不能使用隐含规则的依赖。自然，如果您不编写定义则没有特殊目标**.DEFAULT** 的规则。

如果您使用特殊目标**.DEFAULT** 而不带任何规则和命令**:**

```
.DEFAULT:
```

则以前为目标**.DEFAULT** 定义的命令被清除。如此 **make** 的行为和您从来没有定义目标**.DEFAULT** 一样。

如果您不需要一个目标从万用规则和目标**.DEFAULT** 中得到命令，也不想为该目标执行任何命令，您可以在定义时使用空命令。参阅***使用空命令***。

您可以使用最新类型规则重载另外一个 **makefile** 文件的一部分内容。参阅***重载其它 makefile 文件***。

10.7 过时的后缀规则

后缀规则是定义隐含规则的过时方法。后缀规则因为格式规则更为普遍和简洁而被废弃。它们在 **GNU make**中得到支持是为了和早期的 **makefile** 文件兼容。它们分为单后缀和双后缀规则。

双后缀规则被一对后缀定义：目标后缀和源文件后缀。它可以匹配任何文件名以目标后缀结尾的文件。相应的隐含依赖通过在文件名中将目标后缀替换为源文件后缀得到。一个目标和源文件后缀分别为‘**.o**’和‘**.c**’双后缀规则相当于格式规则`**%.o : %.c'**。

单后缀规则被单后缀定义，该后缀是源文件的后缀。它匹配任何文件名，其相应的依赖名是将文件名添加源文件后缀得到。源文件后缀为‘**.c**’的单后缀规则相当于格式规则‘**% : %.c**’。

通过比较规则目标和定义的已知后缀列表识别后追规则。当 **make** 见到一个目标后缀是已知后缀的规则时，该规则被认为是一个单后缀规则。当 **make** 见到一个目标后缀包含两个已知后缀的规则时，该规则被认为是一个双后缀规则。

例如，‘**.o**’和‘**.c**’都是缺省列表中的已知后缀。所以，如果您定义一个规则，其目标是‘**.c.o**’，则 **make** 认为是一个双后缀规则，源文件后缀是‘**.c**’，目标后缀是‘**.o**’。这里有一个采用过时的方法定义编译 **C** 语言程序的规则：

```
.c.o:
        $(CC) -c $(CFLAGS) $(CPPFLAGS) -o $@ $<
```

后缀规则不能有任何属于它们自己的依赖。如果它们有依赖，它们将不是作为后缀规则使用，而是以令人啼笑皆非的方式处理正常的文件。例如，规则：

```
.c.o: foo.h
        $(CC) -c $(CFLAGS) $(CPPFLAGS) -o $@ $<
```

告诉从依赖 **foo.h** 生成文件名为‘**.c.o**’的文件，并不是象格式规则：

```
%.o: %.c foo.h
        $(CC) -c $(CFLAGS) $(CPPFLAGS) -o $@ $<
```

告诉从‘**.c'**文件生成 ‘**.o'** 文件‘**.c'**的方法：创建所有‘**.o'** 文件使用该格式规则，而且同时使用依赖文件‘**foo.h'**。

没有命令的后缀规则也没有意义。它们并不没有命令的格式规则那样移去以前的规则（参阅***删除隐含规则***）。他们仅仅简单的在数据库中加入后缀或双后缀作为一个目标。

已知的后缀是特殊目标‘**.SUFFIXES**’简单的依赖名。通过为特殊目标‘**.SUFFIXES**’编写规则加入更多的依赖，您可以添加您自己的已知后缀。例如：

```
.SUFFIXES: .hack .win
```

把‘.hack'和‘.win'添加到了后缀列表中。

如果您希望排除缺省的已知后缀而不是仅仅的添加后缀，那么您可以为特殊目标‘.SUFFIXES’编写没有依赖的规则。通过这种方式，可以完全排除特殊目标‘.SUFFIXES’存在的依赖。接着您可以编写另外一个规则添加您要添加的后缀。例如，

```
.SUFFIXES:            # 删除缺省后缀
.SUFFIXES: .c .o .h   # 定义自己的后缀列表
```

标志‘-r'或‘--no-builtin-rules'也能把缺省的后缀列表清空。

变量 SUFFIXES 在 make 读入任何 makefile 文件之前定义缺省的后缀列表。您可以使用特殊目标‘.SUFFIXES’改变后缀列表，但这不能改变变量 SUFFIXES 的值。

10.8 隐含规则搜寻算法

这里是 make 为一个目标‘t’搜寻隐含规则的过程。这个过程用于任何没有命令的双冒号规则，用于任何不含命令的普通规则的目标，以及用于任何不是其它规则目标的依赖。这个过程也能用于来自隐含规则的依赖递归调用该过程搜寻规则链。

在本算法中不提及任何后缀规则，因为后缀规则在 makefile 文件读入时转化为了格式规则。

对于个是‘*archive*(*member*)’的档案成员目标，下述算法重复两次，第一次使用整个目标名‘t’，如果第一次运行没有发现规则，则第二次使用‘(*member*)’作为目标‘t’。

1、 在‘t’中分离出路径部分，称为‘d’，剩下部分称为‘n’。例如如果‘t’是‘src/foo.o’，那么‘d’是‘src/’；‘n’是‘foo.o’。

2、 建立所有目标名匹配‘t’和‘n’的格式规则列表。如果目标格式中含有斜杠，则匹配‘t’，否则，匹配‘n’。

3、 如果列表中有一个规则不是万用规则，则从列表中删除所有非最终万用规则。

4、 将没有命令的规则也从列表中移走。

5、 对每个列表中的格式规则：

   1、 寻找 stem‘s’，也就是和目标格式中%匹配的‘t’或‘n’部分。

   2、 使用 stem‘s’计算依赖名。如果目标格式不包含斜杠，则将‘d’添加在每个依赖的前面。

   3、 测试所有的依赖是否存在或能够创建。（如果一个文件在 makefile 中作为目标或依赖被提及，则我们说它应该存在。）如果所有依赖存在或能够创建，或没有依赖，则可使用该规则。

6、 如果到现在还没有发现能使用的规则，进一步试。对每一个列表中的规则：

   1、 如果规则是最终规则，则忽略它，继续下一条规则。

   2、 象上述一样计算依赖名。

   3、 测试所有的依赖是否存在或能够创建。

   4、 对于不存在的依赖，按照该算法递归调用查找是否能够采用隐含规则创建。

   5、 如果所有依赖存在或能使用隐含规则创建，则应用该规则。

7、 如果没有隐含规则，则如有用于目标‘.DEFAULT’规则，则应用该规则。在这种情况下，将目标‘.DEFAULT’的命令给与‘t’。

一旦找到可以应用的规则，对每一个匹配的目标格式（无论是‘t’或‘n’）使用 stem‘s’替换%，将得到的文件名储存起来直到执行命令更新目标文件‘t’。在这些命令执行以后，把每一个储存的文件名放入数据库，并且标志已经更新，其时间戳和目标文件‘t’一样。

如果格式规则的命令为创建‘**t**’执行，自动变量将设置为相应的目标和依赖（参阅*自动变量*）。

11 使用 make 更新档案文件

档案文件是包含子文件的文件，这些子文件有各自的文件名，一般将它们称为成员；档案文件和程序 **ar** 一块被提及，它们的主要用途是作为连接的例程库。

11.1 档案成员目标

独立的档案文件成员可以在 **make** 中用作目标或依赖。按照下面的方式，您可以在档案文件‘**archive**’中指定名为‘**member**’的成员：

***archive*(*member*)**

这种结构仅仅在目标和依赖中使用，绝不能在命令中应用！绝大多数程序都不在命令中支持这个语法，而且也不能对档案成员直接操作。只有程序 **ar** 和那些为操作档案文件设计的程序才能这样做。所以合法的更新档案成员的命令一定使用 **ar**。例如，下述规则表明借助拷贝文件‘**hack.o**’在档案‘**foolib**’中创建成员‘**hack.o**’**:**

```
foolib(hack.o) : hack.o
        ar cr foolib hack.o
```

实际上，几乎所有的档案成员目标是采用这种方式更新的，并且有一条隐含规则为您专门更新档案成员目标。**注意：**如果档案文件没有直接存在，程序 **ar** 的‘**c**’标志是需要的。

在相同的档案中同时指定几个成员，您可以在圆括号中一起写出所有的成员名。例如：

```
foolib(hack.o kludge.o)
```

等同于：

```
foolib(hack.o) foolib(kludge.o)
```

您还可以在档案成员引用中使用 **shell** 类型的通配符。参阅*在文件名中使用通配符*。例如，‘**foolib(*.o)**’扩展为在档案‘**foolib**’中所有存在以‘**.o**’结尾的成员。也许相当于：‘**foolib(hack.o) foolib(kludge.o)**’。

11.2 档案成员目标的隐含规则

对目标‘**a(m)**’表示名为‘**m**’的成员在档案文件‘**a**’中。

Make 为这种目标搜寻隐含规则时，是用它另外一个的特殊特点：**make** 认为匹配‘**(m)**’的隐含规则也同时匹配‘**a(m)**’。

该特点导致一个特殊的规则，它的目标是‘**(%)**’。该规则通过将文件‘**m**’拷贝到档案中更新目标‘**a(m)**’。例如，它通过将文件‘**bar.o**’拷贝到档案‘**foo.a**’中更新档案成员目标‘**foo.a(bar.o)**’。

如果该规则和其它规则组成链，功能十分强大。‘**make "foo.a(bar.o)"**’（注意使用双引号是为了保护圆括号可被 **shell** 解释）即使没有 **makefile** 文件仅存在文件‘**bar.c**’就可以保证以下命令执行：

```
cc -c bar.c -o bar.o
ar r foo.a bar.o
rm -f bar.o
```

这里 **make** 假设文件‘**bar.o**’是中间文件。参阅*隐含规则链*。

诸如这样的隐含规则是使用自动变量‘**$%**’编写的，参阅*自动变量*。

档案成员名不能包含路径名，但是在 **makefile** 文件中路径名是有用的。如果您写一个档案成员规则‘**foo.a(dir/file.o)**’，**make** 将自动使用下述命令更新：

```
ar r foo.a dir/file.o
```

它的结果是拷贝文件‘**dir/file.o**’进入名为‘**file.a**’的档案中。在完成这样的任务时使用自动变量**%D** 和**%F**。

11.2.1 更新档案的符号索引表

用作库的档案文件通常包含一个名为‘**__.SYMDEF**’ 特殊的成员，成员‘**__.SYMDEF**’包含由所有其它成员定义的外部符号名的索引表。在您更新其它成员后，您必须更新成员‘**__.SYMDEF**’，从而使成员‘**__.SYMDEF**’可以合适的总结其它成员。要完成成员‘**__.SYMDEF**’的更新需要运行 **ranlib** 程序：

```
ranlib archivefile
```

正常情况下，您应该将该命令放到档案文件的规则中，把所有档案文件的成员作为该规则的依赖。例如：

```
libfoo.a: libfoo.a(x.o) libfoo.a(y.o) ...
        ranlib libfoo.a
```

上述程序的结果是更新档案成员‘**x.o**’，‘**y.o**’，等等，然后通过运行程序 **ranlib** 更新符号索引表表成员‘**__.SYMDEF**’。更新成员的规则这里没有列出，多数情况下，您可以省略它们，使用隐含规则把文件拷贝到档案中，具体描述见以前的内容。

使用 **GNU ar** 程序时这不是必要的，因为它自动更新成员‘**__.SYMDEF**’。

11.3 使用档案的危险

同时使用并行执行（**-j** 开关，参阅*并行执行*）和档案应该十分小心。如果多个命令同时对相同的档案文件操作，它们相互不知道，有可能破坏文件。将来的 **make** 版本可能针对

该问题提供一个机制，即将所有操作相同档案文件的命令串行化。但是现在，您必须在编写您自己的 **makefile** 文件时避免该问题，或者采用其它方式，或者不使用选项**-j**。

11.4 档案文件的后缀规则

为处理档案文件，您可以编写一个特殊类型的后缀规则。关于所有后缀的扩展请参阅***过时的后缀规则***。档案后缀规则在 **GNU make** 中已被废弃，因为用于档案的格式规则更加通用（参阅***档案成员目标的隐含规则***），但是为了和其它版本的 **make** 兼容，它们仍然被保留。

编写用于档案的后缀规则，您可以简单的编写一个用于目标后缀‘**.a**’的后缀规则即可。例如，这里有一个用于从 **C** 语言源文件更新档案库的过时后缀规则：

```
.c.a:
        $(CC) $(CFLAGS) $(CPPFLAGS) -c $< -o $*.o
        $(AR) r $@ $*.o
        $(RM) $*.o
```

这和下面的格式规则工作完全一样：

```
(%.o): %.c
        $(CC) $(CFLAGS) $(CPPFLAGS) -c $< -o $*.o
        $(AR) r $@ $*.o
        $(RM) $*.o
```

实际上，这仅仅是 **make** 看到一个以‘**.a**’作为后缀的后缀规则时，它所做的工作。任何双后缀规则‘***.x*.a'** 被转化为一个格式规则，该格式规则的目标格式是‘**(%.o)'** ，依赖格式是‘**%.*x*'.**。

因为您可能要使用‘**.a'** 作为一个文件类型的后缀，**make** 也以正常方式转换档案后缀规则为格式规则，参阅***过时的后缀规则***。这样一个双后缀规则‘***.x*.a'** 产生两个格式规则：‘**(%.o): %.*x*'** 和‘**%.a: %.*x*'.**

12 GNU make 的特点

这里是 **GNU make**的特点的总结，用于比较其它版本的 **make**。我们以 **4.2 BSD** 中的 **make** 的特点为基准。如果您要编写一个可移植的 **makefile** 文件，您不要使用这里列出的 **make** 的特点，也不要使用***不兼容性和失去的特点***中列出的内容。

许多特点在 **System V** 中的 **make** 也存在。

- 变量 **VPATH** 以及它特殊的意义。参阅***在目录中搜寻依赖***。这个特点存在于 **System V** 中的 **make**，但没有事实证明。**4.3 BSD make** 也含有该特点（据说是模仿 **System V** 中变量 **VPATFH** 的特点）。
- 包含其它 **makefile** 文件。参阅***包含其它 makefile 文件***。允许使用一个指令包含多个文件是 **GNU** 的扩展。
- 通过环境，变量可以读入和通讯，参阅***环境变量***。

- 通过变量 MAKEFLAGS 在递归调用 make 时可以传递选项。参阅*和子 make 通讯选项*。
- 在档案引用中自动变量$% 设置为成员名。参阅*自动变量*。
- 自动变量$@, $*, $<, $%, 和 $? 有变体形式如$(@F)和$(@D)。我们把此概念化，并使用它对自动变量$^ 进行了明显扩展。参阅*自动变量*。
- 变量引用。参阅*变量引用基础*。
- 命令行选项‘-b'和‘-m'，接受和忽略。在 System V make 中，这些选项实际起作用。
- 即使指定选项‘-n',‘-q'或‘-t'，也能通过变量 MAKE 执行地归调用 make 的命令。参阅*递归调用 make*。
- 在后缀规则中支持后缀‘.a'。参阅*用于档案文件的后缀规则*。这个特点在 GNU make 中几乎不用，因为规则链更加通用的特点（参阅*隐含规则链*）允许一个格式规则用于在档案中安装成员已经足够（参阅*用于档案成员目标的隐含规则*）。
- 在命令中行排列和反斜杠-新行结合依旧保留，当命令打印时，它们出现的格式和它们在 makefile 文件中基本一样，不同之处是去掉了初始化空白。

下面的特点被各种不同版本的 make 吸收，但哪些版本吸收了哪些特点并不十分清楚。

- 在格式规则中使用‘%’。已经有几个不同版本的 make 使用了该特点。我们不能确认是谁发明了它，但它发展很快。参阅*定义与重新定义格式规则*。
- 规则链以及隐含中间文件。这个特点首先由 Stu Feldman 在它的 make 版本中实现，并用于 AT&T 第八版 Unix 研究中。后来 AT&T 贝尔实验室的 Andrew Hume 在它的 mk 程序中应用（这里称为“传递闭合”）。我们并不清楚是从他们那里得到这个特点或是同时我们自己开发出来的。参阅*隐含规则链*。
- 自动变量包含当前目标的所有依赖的列表。我们一点也不知道是谁做的。参阅*自动变量*。自动变量$+ 是变量$^的简单扩展。
- "what if" 标志(GNU make 中的‘-W') 是 Andrew Hume 在 mk 中发明的。参阅*代替执行命令*。
- 并行执行的概念在许多版本的 make 中存在，尽管 System V 或 BSD 并没有实现。参阅*执行命令*。
- 使用格式替换改变变量引用来自于 SunOS 4。参阅*变量引用基础*。在 GNU make 中，这个功能在变换语法和 SunOS 4 兼容之前由函数 patsubst 提供。不知道谁是权威，因为 GNU make 使用函数 patsubst 在 SunOS 4 发布之前。
- 在命令行前面的‘+’字符有特殊重要的意义（参阅*代替执行命令*）。这是由 *IEEE Standard 1003.2-1992* (POSIX.2)定义的。
- 使用‘+=语法为变量追加值来自于 SunOS 4 make。参阅*为变量值追加文本*。
- 语法‘*archive*(*mem1 mem2...*)'在单一档案文件中列举多个成员来自于 SunOS 4 make.。参阅*档案成员目标*。
- -include 指令包括 makefile 文件，并且对于不存在的文件也不产生错误。该特点 with 来自于 SunOS 4 make。(但是 SunOS 4 make 在单个指令中指定多个 makefile 文件。) 该特点和 SGI make 的 sinclude 相同，

剩余的特点是由 GNU make 发明的：

- 使用‘-v'或`--version'选项打印版本和拷贝权信息。
- 使用‘-h' 或‘--help' 选项总结 make 的选项。
- 简单扩展型变量。参阅*变量的两特特色*。
- 在递归调用 make 时，通过变量 MAKE 自动传递命令行变量。参阅*递归调用 make*。
- 使用命令选项‘-C' 或‘--directory'改变路径。参阅*选项概要*。
- 定义多行变量。参阅*定义多行变量*。
- 使用特殊目标.PHONY 声明假想目标。AT&T 贝尔实验室 Andrew Hume 使用不同的语法在它的 mk 程序中也实现了该功能。这似乎是并行的发现。参阅*假想目标*。
- 调用函数操作文本。参阅*用于转换文本的函数*。
- 使用‘-o'或‘--old-file'选项假装文件是旧文件。参阅*避免重新编译文件*。

- 条件执行。该特点已在不同版本 make 中已经实现很长时间了；它似乎是 C 与处理程序和类似的宏语言的自然扩展，而不是革命性的概念。参阅*makefile 文件中的条件语句*。
- 指定包含的 makefile 文件的搜寻路径。参阅*包含其它 makefile 文件*。
- 使用环境变量指定额外的 makefile 文件。参阅*变量 MAKEFILES*。
- 从文件名中去除前导斜杠`./' ，因此，‘./*file*' 和‘*file*' 是指同一个文件。
- 使用特别搜寻方法搜寻形式如‘-l*name*’的库依赖。参阅*连接库搜寻目录*。
- 允许后缀规则中的后缀包含任何字符（参阅*过时的后缀规则*）。在其它版本的 make 中后缀必须以‘.’开始，并且不能包含‘/’字符。
- 包吹跟踪当前 make 级别适用的变量 MAKWFILES 的值，参阅*递归调用 make*。
- 将任何在命令行中给出的目标放入变量 MAKECMDGOALS。参阅*指定最终目标的参数*。
- 指定静态格式规则。参阅*静态格式规则*。
- 提供选择性 vpath 搜寻。参阅*在目录中搜寻依赖*。
- 提供可计算的变量引用。参阅*变量引用基础*。
- 更新 makefile 文件。参阅重建 makefile 文件。System V make 中有非常非常有限的来自于该功能的形式，它用于为 make 检查 SCCS 文件。
- 各种新建的隐含规则。参阅*隐含规则目录*。
- 内建变量`MAKE_VERSION' 给出 make 的版本号。

13 不兼容性和失去的特点

其它版本的 make 程序也有部分特点在 GNU make 中没有实现。POSIX.2 标准 (*IEEE Standard 1003.2-1992*)规定不需要这些特点。

- ‘*file*((*entry*))' 形式的目标代表一个档案文件的成员 file。选择该成员不使用文件名，而是通过一个定义连接符号 enty 的 OBJ 文件。该特点没有被 GNU make 吸收因为该非标准组件将为 make 加入档案文件符号表的内部知识。参阅*更新档案符号索引表*。
- 在后缀规则中以字符‘~’结尾的后缀在 System V make中有特别的含义；它们指和文件名中没有‘~’的文件通讯的 SCCS 文件。例如，后缀规则‘.c~.o'将从名为‘s.*n*.c'的 SCCS 文件中抽取文件‘*n*.o'。为了完全覆盖，需要这种整系列的后缀规则，参阅*过时的后缀规则*。在 GNU make 中，这种整系列的后缀规则由勇于从 SCCS 文件抽取的两个格式规则掌管，它们可和通用的规则结合成规则链，参阅*隐含规则链*。
- 在 System V make 中，字符串‘$$@'又奇特的含义，在含有多个规则的依赖中，它代表正在处理的特殊目标。这在 GNU make 没有定义，因为字符串‘$$'代表一个平常的字符‘$'。使用静态格式规则可以实现该功能的一部分（参阅*静态格式规则*）。
  System V make 中的规则：

```
$(targets): $$@.o lib.a
```

  在 GNU make 中可以用静态格式规则代替：

```
$(targets): %: %.o lib.a
```

- 在 System V 和 4.3 BSD make 中， 通过 VPATH 搜寻（参阅*为依赖搜寻目录*）发现的文件，它们的文件名改变后加入到命令字符串中。我们认为使用自动变量更简单明了，所以不引进该特点。
- 在一些 Unix make 中，自动变量$*出现在规则的依赖中有令人惊奇的特殊特点：扩展为该规则的目标全名。我们不能明白 Unix make 在心中对这是怎样考虑的，它和正常的变量$*定义完全不同。

- 在一些 Unix make中，隐含规则搜寻（参阅*使用隐含规则*）明显是为所有目标做的，而不仅仅为那些没有命令的目标。这意味着：

```
foo.o:
```

```
    cc -c foo.c
```

在 Unix make 有直觉知道‘foo.o' 依靠‘foo.c'。我们认为这样的用法易导致混乱。Make 中依赖的属性已经定义好（至少对于 GNU make 是这样），再做这样的事情不合规矩。

- GNU make 不包含任何编译以及与处理 EFL 程序的隐含规则。如果我们听说谁使用 EFL，我们乐意把它们加入。
- 在 SVR4 make 中，一条后缀规则可以不含命令，它的处理方式和它含有空命令的处理方式一样（参阅*使用空命令*）。例如：

```
.c.a:
```

将重载内建的后缀规则‘.c.a' 。我们觉得对没有命令的规则简单的为目标添加依赖更为简洁。上述例子和在 GNU make 中下例的行为相同。

```
.c.a: ;
```

- 一些版本的 make 调用 shell 使用‘-e'标志，而不是‘-k'标志 (参阅*测试程序编译*)。标志‘-e'告诉 shell 一旦程序运行返回非零状态就立即退出。我们认为根据每一命令行是否需要需要特殊处理直接写入命令中更为清楚。

14 makefile 文件惯例

本章描述为 GNU make 编写 makefile 文件的惯例。使用 Automake 将帮助您按照这些惯例编写 makefile 文件。

14.1 makefile 文件的通用惯例

任何 makefile 文件都应该包含这行：

```
SHELL = /bin/sh
```

避免在系统中变量 SHELL 可能继承环境中值的麻烦。（在GNU make 中这从来不是问题。）

不同的 make 程序有不同的后缀列表和隐含规则，这有可能造成混乱或错误的行为。因此最好的办法是设置后缀列表，在该列表中，仅仅包含您在特定 makefile 文件中使用的后缀。例如：

```
.SUFFIXES:
.SUFFIXES: .c .o
```

第一行清除了后缀列表，第二行定义了在该 makefile 中可能被隐含规则使用的后缀。

不要假设‘.' 是命令执行的路径。当您在创建程序过程中，需要运行仅是您程序包中一部分的程序时，请确认如果该程序是要创建程序的一部分使用‘./’，如果该程序是源代码中不变的部分使用‘$(srcdir)’。没有这些前缀，仅仅在当前路径下搜索。

建造目录（build directory）‘./’和源代码目录(source directory) ‘$(srcdir)’的区别是很重要的，因为用户可以在‘configure’中使用‘--srcdir’选项建造一个单独的目录。下面的规则：

```
foo.1 : foo.man sedscript
        sed -e sedscript foo.man > foo.1
```

如果创建的目录不是源代码目录将失败，因为文件‘foo.man’和‘sedscript’在源代码目录下。

在使用 GNU make时，依靠变量‘VPATH’搜寻源文件在单个从属性文件存在情况下可以很好地工作，因为 make 中自动变量‘$<’中含有源文件的存在路径。（许多版本的 make 仅在隐含规则中设值变量‘$<’。）例如这样的 makefile 文件目标：

```
foo.o : bar.c
        $(CC) -I. -I$(srcdir) $(CFLAGS) -c bar.c -o foo.o
```

将被替换为：

```
foo.o : bar.c
        $(CC) -I. -I$(srcdir) $(CFLAGS) -c $< -o $@
```

这是为了保证变量‘VPATH’正确的工作。目标含有多个依赖时，使用名了的‘$(srcdir)’是最容易的保证该规则很好工作的方法。例如，以上例子中的目标‘foo.1’最好写为：

```
foo.1 : foo.man sedscript
        sed -e $(srcdir)/sedscript $(srcdir)/foo.man > $@
```

GNU 的分类中通常包含一些不是源文件的文件——例如，‘Info’文件、从 Autoconf, Automake, Bison 或 Flex 中输出的文件等。这些文件在源文件目录下，它们也应该在源文件目录下，不应该在建造目录下。因此 makefile 规则应在源文件目录下更新它们。

然而，如果一个文件没有在分类中出现，makefile 文件不应把它们放到源文件目录下，因为按照通常情况创建一个程序，不应该以任何方式更改源文件目录。

试图建造的创建和安装目标，至少（以及它们的子目标）可在并行的 make 中正确的工作。

14.2 makefile 文件的工具

编写在 shell sh 中运行而不在 csh 中运行的 makefile 文件命令(以及 shell 的脚本，例如‘configure’)，不要使用任何 ksh 或 bash 的特殊特点。

用于创建和安装的‘configure’脚本和 Makefile 规则除下面所列出工具外不应该直接使用其它的任何工具：

```
cat cmp cp diff echo egrep expr false grep install-info
ln ls mkdir mv pwd rm rmdir sed sleep sort tar test touch true
```

压缩程序 gzip 可在 dist 规则中使用。

坚持使用用于这些程序的通用选项，例如，不要使用‘mkdir -p'，它可能比较方便，但是其它大多数系统却不支持它。

避免在 makefile 中创造符号连接是非常不错的注意，因为一些系统不支持这种做法。

用于创建和安装的 Makefile 规则可以使用编译器以及相关的程序，但应该通过 make 变量使用它们,这样可以方便用户使用别的进行替换。这里有按照我们的观念编写一些程序：

```
ar bison cc flex install ld ldconfig lex
make makeinfo ranlib texi2dvi yacc
```

请使用下述 make 变量运行这些程序:

```
$(AR) $(BISON) $(CC) $(FLEX) $(INSTALL) $(LD) $(LDCONFIG) $(LEX)
$(MAKE) $(MAKEINFO) $(RANLIB) $(TEXI2DVI) $(YACC)
```

使用 ranlib 或 ldconfig，您应该确定如果系统中不存在要使用的程序不会引起任何副作用。安排忽略这些命令产生的错误，并且打印信息告诉用户该命令运行失败并不意味着存在问题。（Autoconf ‘AC_PROG_RANLIB'宏可在这方面帮助您。）如果您使用符号连接，对于不支持符号连接的系统您应该有一个低效率运行方案。

附加的工具也可通过 make 变量使用：

```
chgrp chmod chown mknod
```

它在 makefile 中（或脚本中），您知道包含这些工具的特定系统中它都可以很好的工作。

14.3 指定命令的变量

Makefile 文件应该为重载的特定命令、选项等提供变量。

特别在您运行大部分工具时都应该应用变量，如果您要使用程序 Bison，名为 BISON 的变量它的缺省值设置为：‘BISON = bison’，在您需要使用程序 Bison 时，您可以使用$(BISON)引用。

文件管理器工具如 ln，rm，mv 等等，不必要使用这种方式引用，因为用户不可能使用别的程序替换它们。

每一个程序变量应该和用于向该程序提供选项的选项变量一起提供。在程序名变量后添加‘FLAGS'表示向该程序提供选项的选项变量--例如，BISONFLAGS。（名为 CFLAGS 的变量向 C 编译器提供选项，名为 YFLAGS 的变量向 yacc 提供选项，名为 LFLAGS 的变量向 lex 提供选项等是这个规则例外，但因为它们是标准所以我们保留它们。）在任何进行预处理的编译命令中使用变量 CPPFLAGS，在任何进行连接的编译命令中使用变量 LDFLAGS 和直接使用程序 ld 一样。

对于 C 编译器在编译特定文件时必须使用的选项，不应包含在变量 CFLAGS 中，因为用户希望他们能够自由的指定变量 CFLAGS。 要独立于变量 CFLAGS 安排向 C 编译器传递这些必要的选项，可以将这些选项写入编译命令行中或隐含规则的定义中，如下例：

```
CFLAGS = -g
ALL_CFLAGS = -I. $(CFLAGS)
.c.o:
        $(CC) -c $(CPPFLAGS) $(ALL_CFLAGS) $<
```

变量 CFLAGS 中包括选项‘-g’，因为它对于一些编译并不是必需的，您可以认为它是缺省推荐的选项。如果数据包创建使用 GCC 作为编译器，则变量 CFLAGS 中包括选项‘-o’，而且以它为缺省值。

将变量 CFLAGS 放到编译命令的最后，在包含编译选项其它变量的后边，因此用户可以使用变量 CFLAGS 对其它变量进行重载。

每次调用 C 编译器都用到变量 CFLAGS，无论进行编译或连接都一样。

任何 Makefile 文件都定义变量 INSTALL，变量 INSTALL 是将文件安装到系统中的基本命令。

任何 Makefile 文件都定义变量 INSTALL_PROGRAM 和 INSTALL_DATA，（它们的缺省值都是 $(INSTALL)。）在实际安装程序时，不论可执行程序或非可执行程序，一般都使用它们作为命令。下面是使用这些变量的例子：

```
$(INSTALL_PROGRAM) foo $(bindir)/foo
$(INSTALL_DATA) libfoo.a $(libdir)/libfoo.a
```

您可以随意将变量 DESTDIR 预先设置为目标文件名。这样做允许安装程序创建随后在实际目标文件系统中安装文件的快照。不要再 makefile 文件中设置变量 DESTDIR，也不要包含在安装文件中。用变量 DERSTDIR 改变上述例子：

```
$(INSTALL_PROGRAM) foo $(DESTDIR)$(bindir)/foo
$(INSTALL_DATA) libfoo.a $(DESTDIR)$(libdir)/libfoo.a
```

在安装命令中一般使用文件名而不是路径名作为第二个参数。对每一个安装文件都使用单独的命令。

14.4 安装路径变量

安装目录经常以变量命名，所以在非标准地方安装也很容易，这些变量的标准名字将在下面介绍。安装目录依据标准文件系统布局，变量的变体已经在 SVR4, 4.4BSD, Linux, Ultrix v4, 以及其它现代操作系统中使用。

以下两个变量设置安装文件的根目录，所有的其它安装目录都是它们其中一个的子目录，没有任何文件可以直接安装在这两个根目录下。

`prefix'
前缀是用于构造以下列举变量的缺省值。变量 prefix 缺省值是‘/usr/local'。建造完整的 GNU 系统时，变量 prefix 的缺省值是空值，‘/usr' 是符号连接符‘/'。(如果您使用 Autoconf，应将它写为‘@prefix@'。)使用不同于创建程序时变量 prefix 的值运行‘make install'，不会重新编译程序。

`exec_prefix'
前缀是用于构造以下列举变量的缺省值。变量 exec_prefix 缺省值是$(prefix). (如果您使用 Autoconf，应将它写为`@exec_prefix@'。) 一般情况下。变量 $(exec_prefix) 用于存放包含机器特定文件的目录，(例如可执行文件和例程库)，变量$(prefix) 直接存放其它目录。使用不同于创建程序时变量 exec_prefix 的值运行‘make install'，不会重新编译程序。

可执行程序安装在以下目录中：

`bindir'
这个目录下用于安装用户可以运行的可执行程序。其正常的值是‘/usr/local/bin'，但是使用时应将它写为‘$(exec_prefix)/bin'。(如果您使用 Autoconf, 应将它写为‘@bindir@'。)

`sbindir'
这个目录下用于安装从 shell 中调用执行的可执行程序。它仅仅对系统管理员有作用。它的正常的值是‘/usr/local/sbin'，但是使用时应将它写为‘$(exec_prefix)/sbin'。(如果您使用 Autoconf, 应将它写为‘@sbindir@'。)

`libexecdir'
这个目录下用于安装其它程序调用的可执行程序。其正常的值是‘/usr/local/libexec'，但是使用时应将它写为‘$(exec_prefix)/libexec'。(如果您使用 Autoconf, 应将它写为‘@libexecdir@'。)

程序执行时使用的数据文件可分为两类：

- 程序可以正常更改的文件和不能正常更改的文件（虽然用户可以编辑其中的一部分文件）。
- 体系结构无关文件，指这些文件可被所有机器共享；体系相关文件，指仅仅可以被相同类型机器、操作系统共享的文件；其它是永远不能被两个机器共享的文件。

这可产生六种不同的可能性。我们极力反对使用体系相关的文件，当然 OBJ 文件和库文件除外。使用其它体系无关的数据文件更加简洁，并且，这样做也不是很难。

所以，这里有 Makefile 变量用于指定路径：

`datadir'
这个目录下用于安装只读型体系无关数据文件。其正常的值是‘/usr/local/share'，但是使用时应将它写为‘$(prefix)/share'。(如果您使用 Autoconf, 应将它写为‘@datadir@'。) 作为例外，参阅下述的变量‘$(infodir)' 和‘$(includedir)'。

`sysconfdir'
这个目录下用于安装从属于单个机器的只读数据文件，这些文件是：用于配置主机的文件。邮件服务、网络配置文件，‘/etc/passwd'，等等都属于这里的文件。所有该目录下的文件都是平常的 ASCII 文本文件。其正常的值是‘/usr/local/etc',

但是使用时应将它写为‘$(prefix)/etc'.(如果您使用 Autoconf,应将它写为‘@sysconfdir@'.)不要在这里安装可执行文件(它们可能属于‘$(libexecdir)'或‘$(sbindir)')。也不要在这里安装那些在使用时要更改的文件(这些程序用于改变系统拒绝的配置)。它们可能属于‘$(localstatedir)'。

`sharedstatedir'

这个目录下用于安装程序运行中要发生变化的体系无关数据文件。其正常的值是‘/usr/local/com',但是使用时应将它写为‘$(prefix)/com'。(如果您使用 Autoconf,应将它写为‘@sharedstatedir@'。)

`localstatedir'

这个目录下用于安装程序运行中要发生变化的数据文件。但他们属于特定的机器。用户永远不需要在该目录下更改文件配置程序包选项;将这些配置信息放在分离的文件中,这些文件将放入‘$(datadir)'或‘$(sysconfdir)'中,‘$(localstatedir)'正常的值是‘/usr/local/var',但是使用时应将它写为‘$(prefix)/var'。(如果您使用 Autoconf,应将它写为‘@localstatedir@'。)

`libdir'

这个目录下用于存放 OBJ 文件和库的 OBJ 代码。不要在这里安装可执行文件,它们可能应属于‘$(libexecdir)'。变量 libdir 正常的值是‘/usr/local/lib',但是使用时应将它写为‘$(exec_prefix)/lib'。(如果您使用 Autoconf,应将它写为‘@libdir@'。)

`infodir'

这个目录下用于安装软件包的 Info 文件。缺省情况下其值是‘/usr/local/info',但是使用时应将它写为‘$(prefix)/info'.(如果您使用 Autoconf,应将它写为‘@infodir@'.)

`lispdir'

这个目录下用于安装软件包的 Emacs Lisp 文件。缺省情况下其值是‘/usr/local/share/emacs/site-lisp',但是使用时应将它写为‘$(prefix)/share/emacs/site-lisp'。如果您使用 Autoconf,应将它写为‘@lispdir@'。为了保证‘@lispdir@'工作,您需要将以下几行加入到您的‘configure.in'文件中:

```
lispdir='${datadir}/emacs/site-lisp'
AC_SUBST(lispdir)
```

`includedir'

这个目录下用于安装用户程序中 C‘#include'预处理指令包含的头文件。其正常的值是‘/usr/local/include',但是使用时应将它写为‘$(prefix)/include'。(如果您使用 Autoconf,应将它写为‘@includedir@'。)除 GCC 外的大多数编译器不在目录‘/usr/local/include'搜寻头文件,因此这种安装方式仅仅适用于 GCC。有时,这也不是问题,因为一部分库文件仅仅依靠 GCC 才能工作。但也有一部分库文件依靠其他编译器,它们将它们的头文件安装到两个地方,一个由变量 includedir 指定,另一个由变量 oldincludedir 指定。

`oldincludedir'

这个目录下用于安装‘#include'的头文件,这些头文件用于除 GCC 外的其它 C 语言编译器。其正常的值是‘/usr/include'。(如果您使用 Autoconf,应将它写为‘@oldincludedir@'。) Makefile 命令变量 oldincludedir 的值是否为空,如果是空值,它们不在试图使用它,它们还删除第二次安装的头文件。一个软件包在该目录下替换已经存在的头文件,除非头文件来源于同一个软件包。例如,如果您的软件包 Foo 提供一个头文件‘foo.h',则它在变量 oldincludedir 指定的目录下安装的条件是 (1) 这里没有投文件‘foo.h' 或 (2) 来源于软件包 Foo 的头文件‘foo.h'已经在该目录下存在。要检查头文件‘foo.h'是否来自于软件包 Foo,将一个 magic 字符串放到文件中--作为命令的一部分--然后使用正则规则(grep)查找该字符串。

Unix 风格的帮助文件安装在以下目录中:

`mandir'
安装该软件包的顶层帮助（如果有）目录。其正常的值是‘/usr/local/man'，但是使用时应将它写为‘$(prefix)/man'。（如果您使用 Autoconf，应将它写为‘@mandir@'。）

`man1dir'
这个目录下用于安装第一层帮助。其正常的值是‘$(mandir)/man1'。

`man2dir'
这个目录下用于安装第一层帮助。其正常的值是‘$(mandir)/man2'。

`...'
不要将任何 GNU 软件的主要文档作为帮助页。应该编写使用手册。帮助页仅仅是为了人们在 Unix 上方便运行 GNU 软件，它是附属的运行程序。

`manext'
文件名表示对已安装的帮助页的扩展。它包含一定的周期，后跟适当的数字，正常为‘1’。

`man1ext'
文件名表示对已安装的帮助页第一部分的扩展。

`man2ext'
文件名表示对已安装的帮助页第二部分的扩展。

`...'
使用这些文件名代替`manext'。如果该软件包的帮助页需要安装使用手册的多个章节。

最后您应该设置一下变量：

`srcdir'
这个目录下用于安装要编译的原文件。该变量正常的值由 shell 脚本 configure 插入。(如果您使用 Autoconf，应将它写为‘srcdir = @srcdir@'.)

例如：

```
# 用于安装路径的普通前缀。
# 注意：该路经在您开始安装时必须存在
prefix = /usr/local
exec_prefix = $(prefix)
# 这里放置`gcc'命令调用的可执行文件。
bindir = $(exec_prefix)/bin
# 这里放置编译器使用的目录。
libexecdir = $(exec_prefix)/libexec
#这里放置 Info 文件。
infodir = $(prefix)/info
```

如果您的程序要在标准用户指定的目录中安装大量的文件，将该程序的文件放入到特意指定的子目录中是很有必要的。如果您要这样做，您应该写安装规则创建这些子目录。

不要期望用户在上述列举的变量值中包括这些子目录，对于安装目录使用一套变量名的办法使用户能够对于不同的 GNU 软件包指定精确的值，为了使这种做法有用，所有的软件包必须设计为当用户使用时它们能够聪明的工作。

14.5 用户标准目标

所有的 GNU 程序中，在 makefile 中都有下列目标：

`all'

编译整个程序。这应该是缺省的目标。该目标不必重建文档文件，Info 文件已正常情况下应该包括在各个发布的文件中，DVI 文件只有在明确请求情况下才重建。缺省时，make 规则编译和连接使用选项‘-g’，所以程序调试只是象征性的。对于不介意缺少帮助的用户如果他们希望将可执行程序和帮助分开，可以从中剥离出可执行程序。

`install'

编译程序并将可执行程序、库文件等拷贝到为实际使用保留的文件名下。如果是证实程序是否适合安装的简单测试，则该目标应该运行该测试程序。不要在安装时剥离可执行程序，魔鬼很可能关心那些使用 install-strip 目标来剥离可执行程序的人。如果这是可行的，编写的 install 目标规则不应该更改程序建造的目录下的任何东西，仅仅提供‘make all’一切都能完成。这是为了方便用户命名和在其它系统安装建造程序，如果要安装程序的目录不存在，该命令应能创建所有这些目录，这包括变量 prefix 和 exec_prefix 特别指定的目录和所有必要的子目录。完成该任务的方法是借助下面描述的目标 installdirs。在所有安装帮助页的命令前使用‘-’使 make 能够忽略这些命令产生的错误，这可以确保在没有 Unix 帮助页的系统上安装该软件包时能够顺利进行。安装 Info 文件的方法是使用变量$(INSTALL_DATA)将 Info 文件拷贝到变量‘$(infodir)’中（参阅*指定命令的变量*），如果 install-info 程序存在则运行它。 install-info 是一个编辑 Info ‘dir’文件的程序，它可以为 Info 文件添加或更新菜单；它是 Texinfo 软件包的一部分。这里有一个安装 Info 文件的例子：

```
$(DESTDIR)$(infodir)/foo.info: foo.info
        $(POST_INSTALL)
# 可能在‘.’下有新的文件，在 srcdir 中没有。
        -if test -f foo.info; then d=.; \
         else d=$(srcdir); fi; \
        $(INSTALL_DATA) $$d/foo.info $(DESTDIR)$@; \
#如果 install-info 程序存在则运行它。
# 使用‘if’代替在命令行前的‘-’
# 这样，我们可以注意到运行 install-info 产生的真正错误。
# 我们使用‘$(SHELL) -c’ 是因为在一些 shell 中
# 遇到未知的命令不会运行失败。
        if $(SHELL) -c 'install-info --version' \
           >/dev/null 2>&1; then \
          install-info --dir-file=$(DESTDIR)$(infodir)/dir \
                       $(DESTDIR)$(infodir)/foo.info; \
        else true; fi
```

在编写 install 目标时，您必须把所有的命令归位三类：正常的命令、 安装前命令和安装后命令。参阅*安装命令分类*。

`uninstall'

删除所有安装的文件--有‘install’目标拷贝的文件。该规则不应更改编译产生的目录，仅仅删除安装文件的目录。反安装命令象安装命令一样分为三类，参阅*安装命令分类*。

`install-strip'

和目标 install 类似，但在安装时仅仅剥离出可执行文件。 在许多情况下，该目标的定义非常简单：

```
install-strip:
        $(MAKE) INSTALL_PROGRAM='$(INSTALL_PROGRAM) -s' \
                install
```

正常情况下我们不推荐剥离可执行程序进行安装，只有您确信这些程序不会产生问

题时才能这样。剥离安装一个实际执行的可执行文件同时保存那些在这种场合存在 BUG 的可执行文件是显而易见的。

`clean'

删除所有当前目录下的文件，这些文件正常情况下是那些‘建立程序’创建的文件。不要删除那些记录配置的文件，同时也应该保留那些‘建立程序’能够修改的文件，正常情况下要删除的那些文件不包括这些文件，因为发布文件是和这些文件一起创建的。如果‘.dvi'文件不是文件发布文件的一部分，则使用目标‘clean’将同时删除‘.dvi'文件。

`distclean'

删除所有当前目录下的文件，这些文件正常情况下是那些‘建立程序’或‘配置程序’创建的文件。如果您不解包源程序，‘建立程序’不会创建任何其它文件，‘make distclean'将仅在文件发布文件中留下原有的文件。

`mostlyclean'

和目标‘clean'类似，但是避免删除人们正常情况下不编译的文件。例如，用于GCC 的目标‘mostlyclean 不删除文件‘libgcc.a'，因为在绝大多数情况下它都不需要重新编译。

`maintainer-clean'

几乎在当前目录下删除所有能够使用该 makefile 文件可以重建的文件。使用该目标删除的文件包括使用目标 distclean,删除的文件加上从 Bison 产生的 C 语言源文件和标志列表、 Info 文件等等。我们说“几乎所有文件”的原因是运行命令‘make maintainer-clean'不删除脚本‘configure'，即使脚本‘configure'可以使用 Makefile 文件创建。更确切地说，运行‘make maintainer-clean'不删除为了运行脚本‘configure'以及开始建立程序的涉及的所有文件。 这是运行‘make maintainer-clean'删除所有能够重新创建文件时唯一不能删除的一类文件。目标‘maintainer-clean'由该软件包的养护程序使用，不能被普通用户使用。您可以使用特殊的工具重建被目标‘make maintainer-clean'删除的文件。因为这些文件正常情况下包含在发布的文件中，我们并不关心它们是否容易重建。如果您发现您需要对全部发布的文件重新解包，您不能责怪我们。要帮助 make 的用户意识到这一点，用于目标 maintainer-clean 应以以下两行为开始：

```
@echo ‘该命令仅仅用于养护程序；’
@echo ‘它删除的所有文件都能使用特殊工具重建。’
```

`TAGS'

更新该程序的标志表。

`info'

产生必要的 Info 文件。最好的方法是编写象下面规则：

```
info: foo.info

foo.info: foo.texi chap1.texi chap2.texi
        $(MAKEINFO) $(srcdir)/foo.texi
```

您必须在 makefile 文件中定以变量 MAKEINFO。它将运行 makeinfo 程序，该程序是发布程序中 Texinfo 的一部分。正常情况下，一个 GNU 发布程序和 Info 文件一起创建，这意味着 Info 文件存在于源文件的目录下。当用户建造一个软件包，一般情况下，make 不更新 Info 文件，因为它们已经更新到最新了。

`dvi'

创建 DVI 文件用于更新 Texinfo 文档。例如：

```
dvi: foo.dvi

foo.dvi: foo.texi chap1.texi chap2.texi
        $(TEXI2DVI) $(srcdir)/foo.texi
```

您必须在 makefile 文件中定义变量 TEXI2DVI。它将运行程序 texi2dvi，该程序是发布的 Texinfo 一部分。要么仅仅编写依靠文件，要么允许 GNU make提供命令，二者

必选其一。

`dist'

为程序创建一个 tar 文件。创建 tar 文件可以将其中的文件名以子目录名开始，这些子目录名可以是用于发布的软件包名。另外，这些文件名中也可以包含版本号，例如，发布的 GCC 1.40 版的 tar 文件解包的子目录为‘gcc-1.40'。最方便的方法是创建合适的子目录名，如使用 in 或 cp 等作为子目录，在它们的下面安装适当的文件，然后把 tar 文件解包到这些子目录中。使用 gzip 压缩这些 tar 文件，例如，实际的 GCC 1.40版的发布文件叫‘gcc-1.40.tar.gz'。目标 dist 明显的依靠所有的发布文件中不是源文件的文件，所以你应确保发布中的这些文件已经更新。参阅 *GNU 标准编码*中*创建发布文件*。

`check'

执行自我检查。用户应该在运行测试之前，应该先建立程序，但不必安装这些程序；您应该编写一个自我测试程序，在程序已建立但没有安装时执行。

以下目标建议使用习惯名，对于各种程序它们很有用：

installcheck

执行自我检查。用户应该在运行测试之前，应该先建立、安装这些程序。您不因该假设‘$(bindir)'在搜寻路径中。

installdirs

添加名为‘installdirs'目标对于创建文件要安装的目录以及它们的父目录十分有用。脚本‘mkinstalldirs'是专为这样处理方便而编写的；您可以在 Texinfo 软件包中找到它，您可以象这样使用规则：

```
# 确保所有安装目录(例如 $(bindir))
# 都实际存在，如果没有则创建它们。
installdirs: mkinstalldirs
        $(srcdir)/mkinstalldirs $(bindir) $(datadir) \
                                $(libdir) $(infodir) \
                                $(mandir)
```

该规则并不更改编译时创建的目录，它仅仅创建安装目录。

14.6 安装命令分类

编写已安装目标，您必须将所有命令分为三类：正常的命令、安装前命令和安装后命令。

正常情况下，命令把文件移动到合适的地方，并设置它们的模式。它们不会改变任何文件，仅仅把它们从软件包中完整地抽取出来。

安装前命令和安装后命令可能更改一些文件，如，它们编辑配置文件后数据库文件。

安装前命令在正常命令之前执行，安装后命令在正常命令执行后执行。

安装后命令最普通的用途是运行 install-info 程序。 这种工作不能由正常命令完成，因为它更改了一个文件（Info 目录），该文件不能全部、单独从软件包中安装。它是一个安装后命令，因为它需要在正常命令安装软件包中的 Info 文件后才能执行。

许多程序不需要安装前命令，但是我们提供这个特点，以便在需要时可以使用。

要将安装规则的命令分为这三类，应在命令中间插入 category lines（分类行）。 分类行指定了下面叙述的命令的类别。

分类行包含一个 **Tab**、一个特殊的 **make** 变量引用，以及行结尾的随机注释。您可以使用三个变量，每一个变量对应一个类别；变量名指定了类别。分类行不能出现在普通的执行文件中，因为这些 **make** 变量被由正常的定义（您也不应在 **makefile** 文件中定义）。

这里有三种分类行，后面的注释解释了它的含义：

```
$(PRE_INSTALL)     # 以下是安装前命令
$(POST_INSTALL)    # 以下是安装后命令
$(NORMAL_INSTALL)  # 以下是正常命令
```

如果在安装规则开始您没有使用分类行，则在第一个分类行出现之前的所有命令都是正常命令。如果您没有使用任何分类行，则所有命令都是正常命令。

这是反安装的分类行

```
$(PRE_UNINSTALL)     #以下是反安装前命令
$(POST_UNINSTALL)    #以下是反安装后命令
$(NORMAL_UNINSTALL)  #以下是正常命令
```

反安装前命令的典型用法是从 **Info** 目录删除全部内容。

如果目标 **install** 或 **uninstall** 有依赖作为安装程序的子程序，那么您应该使用分类行先启动每一个依赖的命令，再使用分类行启动主目标的命令。无论哪一个依赖实际执行，这种方式都能保证每一条命令都放置到了正确的分类中。

安装前命令和安装后命令除了对于下述命令外，不能运行其它程序：

```
basename bash cat chgrp chmod chown cmp cp dd diff echo
egrep expand expr false fgrep find getopt grep gunzip gzip
hostname install install-info kill ldconfig ln ls md5sum
mkdir mkfifo mknod mv printenv pwd rm rmdir sed sort tee
test touch true uname xargs yes
```

按照这种方式区分命令的原因是为了创建二进制软件包。典型的二进制软件包包括所有可执行文件、必须安装的其它文件以及它自己的安装文件——所以二进制软件包不需要运行任何正常命令。但是安装二进制软件包需要执行安装前命令和安装后命令。

建造二进制软件包的程序通过抽取安装前命令和安装后命令工作。这里有一个抽取安装前命令的方法：

```
make -n install -o all \
      PRE_INSTALL=pre-install \
      POST_INSTALL=post-install \
      NORMAL_INSTALL=normal-install \
  | gawk -f pre-install.awk
```

这里文件 ‘**pre-install.awk**'可能包括：

```
$0 ~ /^\t[ \t]*(normal_install|post_install)[ \t]*$/ {on = 0}
on {print $0}
```

$0 ~ /^\t[\t]*pre_install[\t]*$/ {on = 1}
安装前命令的结果文件是象安装二进制软件包的一部分 **shell** 脚本一样执行。

15 快速参考

这是对指令、文本操作函数以及 **GNU make** 能够理解的变量等的汇总。对于其他方面的总结参阅*特殊的内建目标名，隐含规则目录，选项概要*。

这里是 **GNU make** 是别的指令的总结：

define *variable*
endef
定义多行递归调用扩展型变量。参阅*定义固定次序的命令*。
ifdef *variable*
ifndef *variable*
ifeq (*a*,*b*)
ifeq "*a*" "*b*"
ifeq '*a*' '*b*'
ifneq (*a*,*b*)
ifneq "*a*" "*b*"
ifneq '*a*' '*b*'
else
endif
makefile 文件中的条件扩展，参阅 ***makefile* 文件中的条件语句**。
include *file*
-include *file*
sinclude *file*
包含其它 **makefile** 文件，参阅*包含其它 **makefile** 文件*。
override *variable* = *value*
override *variable* := *value*
override *variable* += *value*
override *variable* ?= *value*
override define *variable*
endef
定义变量、对以前的定义重载、以及对在命令行中定义的变量重载。参阅 ***override* 指令**。
export
告诉 **make** 缺省向子过程输出所有变量，参阅*与子 **make** 通讯的变量*。
export *variable*
export *variable* = *value*
export *variable* := *value*
export *variable* += *value*
export *variable* ?= *value*
unexport *variable*
告诉 **make** 是否向子过程输出一个特殊的变量。参业*与子 **make** 通讯的变量*。
vpath *pattern path*
制定搜寻匹配‘%’格式的文件的路径。参阅 ***vpath* 指令**。
vpath *pattern*
去除以前为‘***pattern***’指定的所有搜寻路径。

vpath
　　去除以前用 **vpath** 指令指定的所有搜寻路径。

这里是操作文本函数的总结，参阅*文本转换函数*：

$(subst *from*,*to*,*text*)
　　在‘**text**’中用‘**to**’代替‘**from**’，参阅*字符串替换与分析函数*。
$(patsubst *pattern*,*replacement*,*text*)
　　在‘**text**’中用‘***replacement***’代替匹配‘***pattern***’字，参阅*字符串替换与分析函数*。
$(strip *string*)
　　从字符串中移去多余的空格。参阅*字符串替换与分析函数*。
$(findstring *find*,*text*)
　　确定‘***find***’在‘***text***’中的位置。参阅*字符串替换与分析函数*。
$(filter *pattern...*,*text*)
　　在‘**text**’中选择匹配‘***pattern***’的字。参阅*字符串替换与分析函数*。
$(filter-out *pattern...*,*text*)
　　在‘**text**’中选择不匹配‘***pattern***’的字。参阅*字符串替换与分析函数*。
$(sort *list*)
　　将‘**list**’中的字按字母顺序排序，并删除重复的字。参阅*字符串替换与分析函数*。
$(dir *names...*)
　　从文件名中抽取路径名。参阅*文件名函数*。
$(notdir *names...*)
　　从文件名中抽取路径部分。参阅*文件名函数*。
$(suffix *names...*)
　　从文件名中抽取非路径部分。参阅*文件名函数*。
$(basename *names...*)
　　从文件名中抽取基本文件名。参阅*文件名函数*。
$(addsuffix *suffix*,*names...*)
　　为‘**names**’中的每个字添加后缀。参阅*文件名函数*。
$(addprefix *prefix*,*names...*)
　　为‘**names**’中的每个字添加前缀。参阅*文件名函数*。
$(join *list1*,*list2*)
　　连接两个并行的字列表。参阅*文件名函数*。
$(word *n*,*text*)
　　从‘**text**’中抽取第 **n** 个字。参阅*文件名函数*。
$(words *text*)
　　计算‘**text**’中字的数目。参阅*文件名函数*。
$(wordlist *s*,*e*,*text*)
　　返回‘**text**’中 **s** 到 **e** 之间的字。参阅*文件名函数*。
$(firstword *names...*)
　　在‘**names…**’中的第一个字。参阅*文件名函数*。
$(wildcard *pattern...*)
　　寻找匹配 **shell** 文件名格式的文件名。参阅 ***wildcard* 函数**。
$(error *text...*)
　　该函数执行时，**make** 产生信息为‘**text**’的致命错误。参阅*控制 make 的函数*。
$(warning *text...*)
　　该函数执行时，**make** 产生信息为‘**text**’的警告。参阅*控制 make 的函数*。
$(shell *command*)
　　执行 **shell** 命令并返回它的输出。参阅*函数 **shell***。
$(origin *variable*)
　　返回 **make** 变量‘***variable***’的定义信息。参阅*函数 **origin***。

$(foreach *var*,*words*,*text*)

将列表列表 **words** 中的每一个字对应后接 **var** 中的每一个字，将结果放在 **text** 中。参阅*函数 foreach*。

$(call *var*,*param*,...)

使用对**$(1)**, **$(2)...**对变量计算变量 ***var*** ，变量**$(1)**, **$(2)...**分别代替参数 ***param*** 第一个、第二个…的值。参阅*函数 call*。

这里是对自动变量的总结，完整的描述参阅*自动变量*。

$@

目标文件名。

$%

当目标是档案成员时，表示目标成员名。

$<

第一个依赖名。

$?

所有比目标‘新’的依赖的名字，名字之间用空格隔开。对于为档案成员的依赖，只能使用命名的成员。参阅*使用 make 更新档案文件*。

$^

$+

所有依赖的名字，名字之间用空格隔开。对于为档案成员的依赖，只能使用命名的成员。参阅*使用 make 更新档案文件*。变量 **$^** 省略了重复的依赖，而变量 **$+** 则按照原来次序保留重复项，

$*

和隐含规则匹配的 **stem**(径)。参阅*格式匹配*。

$(@D)

$(@F)

变量**$@.**中的路径部分和文件名部分。

$(*D)

$(*F)

变量**$***中的路径部分和文件名部分。

$(%D)

$(%F)

变量**$%**中的路径部分和文件名部分。

$(<D)

$(<F)

变量**$<**中的路径部分和文件名部分。

$(^D)

$(^F)

变量**$^**中的路径部分和文件名部分。

$(+D)

$(+F)

变量**$+**中的路径部分和文件名部分。

$(?D)

$(?F)

变量**$?**中的路径部分和文件名部分。

以下是 **GNU make** 使用变量：

MAKEFILES

每次调用 **make** 要读入的 **Makefiles** 文件。参阅*变量 MAKEFILES*。

VPATH

对在当前目录下不能找到的文件搜索的路径。参阅 ***VPATH：所有依赖的搜寻路径。***

SHELL

系统缺省命令解释程序名，通常是`**/bin/sh'**。您可以在 **makefile** 文件中设值变量 **SHELL** 改变运行程序使用的 **shell**。参阅***执行命令***。

MAKESHELL

改变量仅用于 **MS-DOS**，**make** 使用的命令解释程序名，该变量的值比变量 **SHELL** 的值优先。参阅***执行命令***。

MAKE

调用的 **make** 名。在命令行中使用该变量有特殊的意义。参阅变量 **MAKE** 的工作方式。

MAKELEVEL

递归调用的层数**(**子 **makes)**。参阅***与子 make 通讯的变量***。

MAKEFLAGS

向 **make** 提供标志。您可以在环境或 **makefile** 文件中使用该变量设置标志。参阅***与子 make 通讯的变量***。在命令行中不能直接使用该变量，应为它的内容不能在 **shell** 中正确引用，但总是允许递归调用 **make** 时通过环境传递给子 **make**。

MAKECMDGOALS

该目标是在命令行中提供给 **make** 的。设置该变量对 **make** 的行为没有任何影响。参阅***特别目标的参数***。

CURDIR

设置当前工作目录的路径名，参阅***递归调用 make***。

SUFFIXES

在读入任何 **makefile** 文件之前的后缀列表。

.LIBPATTERNS

定义 **make** 搜寻的库文件名，以及搜寻次序。参阅***连接库搜寻目录***。

16 make 产生的错误

这里是您可以看到的由 **make** 产生绝大多数普通错误列表，以及它们的含义和修正它们信息。

有时 **make** 产生的错误不是致命的，如一般在命令脚本行前面存在前缀的情况下，和在命令行使用选向‘**-k**’的情况下产生的错误几乎都不是致命错误。使用字符串*******作前缀将产生致命的错误。

错误信息前面都使用前缀，前缀的内容是产生错误的程序名或 **makefile** 文件中存在错误的文件名和包含该错误的行的行号和。

在下述的错误列表中，省略了普通的前缀：

`[*foo*] Error *NN*'

`[*foo*] *signal description*'

这些错误并不是真的 **make** 的错误。它们意味着 **make** 调用的程序返回非零状态值，错误码（**Error *NN***），这种情况 **make** 解释为失败，或非正常方式退出（一些类型信号），参阅***命令错误***。如果信息中没有附加*******，则是子过程失败，但在 **makefile** 文件中的这条规则有特殊前缀，因此 **make** 忽略该错误。

`missing separator. Stop.'

`missing separator (did you mean TAB instead of 8 spaces?). Stop.'

这意味着 **make** 在读取命令行时遇到不能理解的内容。**GNU make** 检查各种分隔符**(:, =,** 字符 **TAB**，等**)** 从而帮助确定它在命令行中遇到了什么类型的错误。这意味着，**make** 不能发现一个合法的分隔符。出现该信息的最可能的原因是您（或许您的编辑

器，绝大部分是 ms-windows 的编辑器）在命令行缩进使用了空格代替了字符 Tab。这种情况下，make 将使用上述的第二种形式产生错误信息。**一定切记，任何命令行都以字符 Tab 开始，八个空格也不算数**。参阅*规则的语法*。

`commands commence before first target. Stop.'

`missing rule before commands. Stop.'

这意味着在 makefile 中似乎以命令行开始：以Tab 字符开始，但不是一个合法的命令行（例如，一个变量的赋值）。任何命令行必须和一定的目标相联系。产生第二种的错误信息是一行的第一个非空白字符为分号，make 对此的解释是您遗漏了规则中的"target: prerequisite" 部分，参阅*规则的语法*。

`No rule to make target `*xxx*'.'

`No rule to make target `*xxx*', needed by `*yyy*'.'

这意味着 make 决定必须建立一个目标，但却不能在 makefile 文件中发现任何用于创建该目标的指令，包括具体规则和隐含规则。如果您希望创建该目标，您需要另外为改目标编写规则。其它关于该问题产生原因可能是 makefile 文件是草稿（如文件名错）或破坏了源文件树（一个文件不能按照计划重建，仅仅由于一个依赖的问题）。

`No targets specified and no makefile found. Stop.'

`No targets. Stop.'

前者意味着您没有为命令行提供要创建的目标，make 不能读入任何 makefile 文件。后者意味着一些 makefile 文件被找到，但没有包含缺省目标以及命令行等。GNU make 在这种情况下无事可做。参阅*指定 makefile 文件的参数*。

`Makefile `*xxx*' was not found.'

`Included makefile `*xxx*' was not found.'

在命令行中指定一个 makefile 文件（前者）或包含的 makefile 文件（后者）没有找到。

`warning: overriding commands for target `*xxx*''

`warning: ignoring old commands for target `*xxx*''

GNU make 允许命令在一个规则中只能对一个命令使用一次(双冒号规则除外)。如果您为一个目标指定一个命令，而该命令在目标定义是已经定义过，这种警告就会产生；第二个信息表明后来设置的命令将改写以前对该命令的设置。参阅*具有多条规则的目标*。

`Circular *xxx* <- *yyy* dependency dropped.'

这意味着 make 检测到一个相互依靠一个循环：在跟踪目标 xxx 的依赖 yyy 时发现，依赖 yyy 的依赖中一个又以 xxx 为依赖。

`Recursive variable `*xxx*' references itself (eventually). Stop.'

这意味着您定义一个正常（递归调用性）make 变量 xxx，当它扩展时，它将引用它自身。无论对于简单扩展型变量(:=)或追加定义（+=）,这也都是不能允许的，参阅*使用变量*。

`Unterminated variable reference. Stop.'

这意味着您在变量引用或函数调用时忘记写右括号。

`insufficient arguments to function `*xxx*'. Stop.'

这意味着您在调用函数是您密友提供需要数目的参数。关于函数参数的详细描述参阅*文本转换函数*。

`missing target pattern. Stop.'

`multiple target patterns. Stop.'

`target pattern contains no `%'. Stop.'

这些错误信息是畸形的静态格式规则引起的。第一条意味着在规则的目标部分没有规则，第二条意味着在目标部分有多个规则，第三条意味着没有包含格式符%。参阅*静态格式规则语法*。

`warning: -jN forced in submake: disabling jobserver mode.'

该条警告和下条警告是在 make 检测到在能与子 make 通讯的系统中包含并行处理的

错误（参阅*与子 make 通讯选项*）。警告信息是如果递归调用一个 make 过程，而且还使用了‘-jn’选项（这里 n 大于 1）。这种情况很可能发生，例如，如果您设置环境变量 MAKE 为‘make –j2’。这种情况下，子 make 不能与其它 make 过程通讯，而且还简单假装它由两个任务。

`warning: jobserver unavailable: using -j1. Add `+' to parent make rule.'

为了保证 make 过程之间通讯，父层 make 将传递信息给子 make。这可能导致问题，因为子过程有可能不是实际的一个 make，而父过程仅仅认为子过程是一个 make 而将所有信息传递给子过程。父过程是采用正常的算法决定这些的（参阅*变量 MAKE 的工作方式*）。如果makefile 文件构建了这样的父过程，它并不知道子过程是否为 make，那么，子过程将拒收那些没有用的信息。这种情况下，子过程就会产生该警告信息，然后按照它内建的次序方式进行处理。

17 复杂的 makfile 文件例子

这是一个用于 GNU tar 程序的 makefile 文件；这是一个中等复杂的 makefile 文件。

因为‘all’是第一个目标，所以它是缺省目标。该 makefile 文件一个有趣的地方是‘testpad.h'是由 testpad 程序创建的源文件，而且该程序自身由‘testpad.c'编译得到的。

如果您键入‘make'或`make all',则 make 创建名为‘tar'可执行文件，提供远程访问磁带的进程‘rmt’，和名为‘tar.info'的 Info 文件。

如果您键入‘make install’，则 make 不但创建‘tar’,‘rmt',和‘tar.info'，而且安装它们。

如果您键入‘make clean',则 make 删除所有‘.o'文件，以及‘tar',‘rmt',‘testpad’, ‘testpad.h',和‘core’文件。

如果您键入‘make distclean',则 make 不仅删除‘make clean'删除的所有文件，而且包括文件‘TAGS', ‘Makefile',和‘config.status' 文件。(虽然不明显，但该 makefile (和‘config.status')是用户用 configure 程序产生的，该程序是由发布的 tar 文件提供，但这里不进行说明。)

如果您键入‘make realclean',则 make 删除‘make distclean ’删除的所有文件，而且包括由‘tar.texinfo'产生的 Info 文件。

另外，目标 shar 和 dist 创造了发布文件的核心。

```
# 自动从 makefile.in 产生
# 用于 GNU tar 程序的 Unix Makefile
# Copyright (C) 1991 Free Software Foundation, Inc.

# 本程序是自由软件；在遵照 GNU 条款的情况下
# 您可以重新发布它或更改它
# 普通公众许可证 ...
...
...

SHELL = /bin/sh

#### 启动系统配置部分 ####

srcdir = .

# 如果您使用 gcc，您应该在运行
# 和它一起创建的固定包含的脚本程序以及
# 使用-traditional 选项运行 gcc 中间选择其一。
# 另外的 ioctl 调用在一些系统上不能正确编译
```

```
CC = gcc -O
YACC = bison -y
INSTALL = /usr/local/bin/install -c
INSTALLDATA = /usr/local/bin/install -c -m 644

# 您应该在 DEFS 中添加的内容：
# -DSTDC_HEADERS        如果您有标准 C 的头文件和
#                       库文件。
# -DPOSIX               如果您有 POSIX.1 的头文件和
#                       库文件。
# -DBSD42               如果您有 sys/dir.h (除非
#                       您使用-DPOSIX), sys/file.h,
#                       和 st_blocks 在`struct stat'中。
# -DUSG                 如果您有 System V/ANSI C
#                       字符串和内存控制函数
#                       和头文件， sys/sysmacros.h,
#                       fcntl.h, getcwd, no valloc,
#                       和 ndir.h (除非
#                       您使用-DDIRENT)。
# -DNO_MEMORY_H         如果 USG 或 STDC_HEADERS 但是不
#                       包括 memory.h.
# -DDIRENT              如果 USG 而且您又用 dirent.h
#                       代替 ndir.h。
# -DSIGTYPE=int         如果您的信号控制器
#                       返回 int,非 void.
# -DNO_MTIO             如果您缺少 sys/mtio.h
#                       (magtape ioctls).
# -DNO_REMOTE           如果您没有一个远程 shell
#                       或 rexec.
# -DUSE_REXEC           对远程磁带使用 rexec
#                       操作代替
#                       分支 rsh 或 remsh.
# -DVPRINTF_MISSING     如果您缺少函数 vprintf
#                       (但是有_doprnt).
# -DDOPRNT_MISSING      如果您缺少函数 _doprnt.
#                       同样需要定义
#                       -DVPRINTF_MISSING.
# -DFTIME_MISSING       如果您缺少系统调用 ftime
# -DSTRSTR_MISSING      如果您缺少函数 strstr。
# -DVALLOC_MISSING      如果您缺少函数 valloc。
# -DMKDIR_MISSING       如果您缺少系统调用 mkdir 和
#                       rmdir。
# -DRENAME_MISSING      如果您缺少系统调用 rename。
# -DFTRUNCATE_MISSING   如果您缺少系统调用 ftruncate。
#
# -DV7                  在 Unix 版本 7 (没有
#                       进行长期测试).
# -DEMUL_OPEN3          如果您缺少 3-参数版本
#                       的 open, 并想通过您有的系统调用
#                       仿真它。
# -DNO_OPEN3            如果您缺少 3-参数版本的 open
#                       并要禁止 tar -k 选项
#                       代替仿真 open.
```

```
# -DXENIX                如果您有 sys/inode.h
#                        并需要它包含 94

DEFS =  -DSIGTYPE=int -DDIRENT -DSTRSTR_MISSING \
        -DVPRINTF_MISSING -DBSD42
# 设置为 rtapelib.o ，除非使它为空时
# 您定义了 NO_REMOTE,
RTAPELIB = rtapelib.o
LIBS =
DEF_AR_FILE = /dev/rmt8
DEFBLOCKING = 20

CDEBUG = -g
CFLAGS = $(CDEBUG) -I. -I$(srcdir) $(DEFS) \
        -DDEF_AR_FILE=\"$(DEF_AR_FILE)\" \
        -DDEFBLOCKING=$(DEFBLOCKING)
LDFLAGS = -g

prefix = /usr/local
# 每一个安装程序的前缀。
#正常为空或`g'。
binprefix =

# 安装 tar 的路径
bindir = $(prefix)/bin

# 安装 info 文件的路径.
infodir = $(prefix)/info

#### 系统配置结束部分 ####

SRC1 =  tar.c create.c extract.c buffer.c \
        getoldopt.c update.c gnu.c mangle.c
SRC2 =  version.c list.c names.c diffarch.c \
        port.c wildmat.c getopt.c
SRC3 =  getopt1.c regex.c getdate.y
SRCS =  $(SRC1) $(SRC2) $(SRC3)
OBJ1 =  tar.o create.o extract.o buffer.o \
        getoldopt.o update.o gnu.o mangle.o
OBJ2 =  version.o list.o names.o diffarch.o \
        port.o wildmat.o getopt.o
OBJ3 =  getopt1.o regex.o getdate.o $(RTAPELIB)
OBJS =  $(OBJ1) $(OBJ2) $(OBJ3)
AUX =   README COPYING ChangeLog Makefile.in  \
        makefile.pc configure configure.in \
        tar.texinfo tar.info* texinfo.tex \
        tar.h port.h open3.h getopt.h regex.h \
        rmt.h rmt.c rtapelib.c alloca.c \
        msd_dir.h msd_dir.c tcexparg.c \
        level-0 level-1 backup-specs testpad.c

all:    tar rmt tar.info
```

```
tar:    $(OBJS)
        $(CC) $(LDFLAGS) -o $@ $(OBJS) $(LIBS)

rmt:    rmt.c
        $(CC) $(CFLAGS) $(LDFLAGS) -o $@ rmt.c

tar.info: tar.texinfo
        makeinfo tar.texinfo

install: all
        $(INSTALL) tar $(bindir)/$(binprefix)tar
        -test ! -f rmt || $(INSTALL) rmt /etc/rmt
        $(INSTALLDATA) $(srcdir)/tar.info* $(infodir)

$(OBJS): tar.h port.h testpad.h
regex.o buffer.o tar.o: regex.h
# getdate.y 有 8 个变换/减少冲突。

testpad.h: testpad
        ./testpad

testpad: testpad.o
        $(CC) -o $@ testpad.o

TAGS:   $(SRCS)
        etags $(SRCS)

clean:
        rm -f *.o tar rmt testpad testpad.h core

distclean: clean
        rm -f TAGS Makefile config.status

realclean: distclean
        rm -f tar.info*

shar: $(SRCS) $(AUX)
        shar $(SRCS) $(AUX) | compress \
          > tar-`sed -e '/version_string/!d' \
                     -e 's/[^0-9.]*\([0-9.]*\).*/\1/' \
                     -e q
                     version.c`.shar.Z

dist: $(SRCS) $(AUX)
        echo tar-`sed \
             -e '/version_string/!d' \
             -e 's/[^0-9.]*\([0-9.]*\).*/\1/' \
             -e q
             version.c` > .fname
        -rm -rf `cat .fname`
        mkdir `cat .fname`
        ln $(SRCS) $(AUX) `cat .fname`
        tar chZf `cat .fname`.tar.Z `cat .fname`
```

```
        -rm -rf `cat .fname` .fname

tar.zoo: $(SRCS) $(AUX)
        -rm -rf tmp.dir
        -mkdir tmp.dir
        -rm tar.zoo
        for X in $(SRCS) $(AUX) ; do \
            echo $$X ; \
            sed 's/$$/^M/' $$X \
            > tmp.dir/$$X ; done
        cd tmp.dir ; zoo aM ../tar.zoo *
        -rm -rf tmp.dir
```

脚注

(1)

为 MS-DOS 和 MS-Windows 编译的 GNU Make 和将前缀定义为 DJGPP 树体系的根的行为相同。

(2)

在 MS-DOS 上，当前工作目录的值是 global，因此改变它将影响这些系统中随后的命令行。

(3)

texi2dvi 使用 TeX 进行真正的格式化工作。TeX 没有和 Texinfo 一块发布。

本文档使用版本 1.54 的 texi2html 翻译器于 2000 年 7 月 19 日产生。

名词翻译对照表

archive	档案
archive member targets	档案成员目标
arguments of functions	函数参数
automatic variables	自动变量
backslash (\)	反斜杠
basename	基本文件名
binary packages	二进制包

compatibility		兼容性
data base		数据库
default directries		缺省目录
default goal		缺省最终目标
defining variables verbatim		定义多行变量
directive		指令
dummy pattern rule		伪格式规则
echoing of commands		命令回显
editor		编辑器
empty commands		空命令
empty targets		空目标
environment		环境
explicit rule		具体规则
file name functions		文件名函数
file name suffix		文件名后缀
flags		标志
flavors of variables		变量的特色
functions		函数
goal		最终目标
implicit rule		隐含规则
incompatibilities		不兼容性
intermediate files		中间文件
match-anything rule		万用规则
options		选项
parallel execution		并行执行
pattern rule		格式规则
phony targets		假想目标
prefix		前缀
prerequisite		依赖
recompilation		重新编译
rule		规则
shell command	shell	命令
static pattern rule		静态格式规则
stem		径
sub-make		子 make
subdirectories		子目录
suffix		后缀
suffix rule		后缀规则
switches		开关
target		目标
value		值
variable		变量
wildcard		通配符
word		字