GNU Make 使用手冊（中譯版）

翻譯：於鳳昌

譯者注：本人在閱讀 Linux 源代碼過程中發現如果要全面瞭解 Linux 的結構、理解 Linux 的編程總體設計及思想必須首先全部讀通 Linux 源代碼中各級的 Makefile 檔。目前，在網上雖然有一些著作，但都不能全面的解釋 Linux 源代碼中各級的 Makefile 檔，因此本人認真閱讀了 GNU Make 使用手冊（3.79）版原文，在此基礎上翻譯了該手冊，以滿足對 Linux 源代碼有興趣或者希望採用 GCC 編寫程式但對缺乏 GNU Make 全面瞭解之人士的需要。本人是業餘愛好不是專業翻譯人士，如果有問題請通過電子信箱與我聯繫共同商討，本人的 E-mail 為：yfc70@public2.lyptt.ha.cn 。注意在文章中出現的斜體加粗字表示章節。

GNU make Version 3.79

April 2000

Richard M. Stallman and Roland McGrath

目錄

1 make 概述
- 1.1 怎樣閱讀本手冊
- 1.2 問題和 BUG

2 Makefile 檔介紹
- 2.1 規則的格式
- 2.2 一個簡單的 Makefile 檔
- 2.3make 處理 Makefile 檔的過程
- 2.4 使用變數簡化 Makefile 檔
- 2.5 讓 make 推斷命令
- 2.6 另一種風格的 Makefile 檔
- 2.7 在目錄中刪除檔的規則

3 3 編寫 Makefile 檔
- 3.1Makefile 檔的內容
- 3.2Makefile 檔的命名
- 3.3 包含其他的 Makefile 文件
- 3.4 變數 MAKEFILES

3.5Makefile 檔重新生成的過程
3.6 重載其他 Makefile 文件
3.7make 讀取 Makefile 檔的過程
4 編寫規則
4.1 規則的語法
4.2 在檔案名中使用通配符
4.2.1 通配符例子
4.2.2 使用通配符的常見錯誤
4.2.3 函數 wildcard
4.3 在目錄中搜尋依賴
4.3.1VPATH:所有依賴的搜尋路徑
4.3.2vpath 指令
4.3.3 目錄搜尋過程
4.3.4 編寫搜尋目錄的 shell 命令
4.3.5 目錄搜尋和隱含規則
4.3.6 連接庫的搜尋目錄
4.4 假想目標
4.5 沒有命令或依賴的規則
4.6 使用空目錄檔記錄事件
4.7 內建的特殊目標名
4.8 具有多個目標的規則
4.9 具有多條規則的目標
4.10 靜態格式規則
4.10.1 靜態格式規則的語法
4.10.2 靜態格式規則和隱含規則
4.11 雙冒號規則
4.12 自動生成依賴
5 在規則中使用命令
5.1 命令回顯
5.2 執行命令
5.3 並行執行
5.4 命令錯誤
5.5 中斷或關閉 make
5.6 遞迴調用 make
5.6.1 變數 MAKE 的工作方式
5.6.2 與子 make 通訊的變數
5.6.3 與子 make 通訊的選項
5.6.4`--print-directory'選項
5.7 定義固定次序命令
5.8 使用空命令
6 6 使用變數
6.1 變數引用基礎
6.2 變數的兩個特色
6.3 變數高級引用技術
6.3.1 替換引用
6.3.2 嵌套變數引用
6.4 變數取值
6.5 設置變數
6.6 為變數值追加文本
6.7override 指令
6.8 定義多行變數

6.9 環境變數
6.10 特定目標變數的值
6.11 特定格式變數的值
7 Makefile 檔的條件語句
7.1 條件語句的例子
7.2 條件語句的語法
7.3 測試標誌的條件語句
8 文本轉換函數
8.1 函數調用語法
8.2 字串替換和分析函數
8.3 檔案名函數
8.4 函數 foreach
8.5 函數 if
8.6 函數 call
8.7 函數 origin
8.8 函數 shell
8.9 控制 Make 的函數
9 9 運行 make
9.1 指定 Makefile 檔的參數
9.2 指定最終目標的參數
9.3 代替執行命令
9.4 避免重新編譯檔
9.5 變數重載
9.6 測試編譯程序
9.7 選項概要
10 使用隱含規則
10.1 使用隱含規則
10.2 隱含規則目錄
10.3 隱含規則使用的變數
10.4 隱含規則鏈
10.5 定義與重新定義格式規則
10.5.1 格式規則簡介
10.5.2 格式規則的例子
10.5.3 自動變數
10.5.4 格式匹配
10.5.5 萬用規則
10.5.6 刪除隱含規則
10.6 定義最新類型的缺省規則
10.7 過時的尾碼規則
10.8 隱含規則搜尋演算法
11 使用 make 更新檔案檔
11.1 檔案成員目標
11.2 檔案成員目標的隱含規則
11.2.1 更新檔案成員的符號索引表
11.3 使用檔案的危險
11.4 檔案檔的尾碼規則
12 GNU make 的特點
13 不相容性和失去的特點
14 Makefile 文件慣例
14.1makefile 檔的通用慣例
14.2makefile 文件的工具

14.3 指定命令的變數
14.4 安裝路徑變數
14.5 用戶標準目標
14.6 安裝命令分類
15 快速參考
16make 產生的錯誤
17 複雜的 Makefile 檔例子
附錄 名詞翻譯對照表

1 Make 概述

Make 可自動決定一個大程式中哪些檔需要重新編譯，並發佈重新編譯它們的命令。本版本GNU Make使用手冊由Richard M. Stallman and Roland McGrath編著，是從Paul D. Smith撰寫的V3.76版本發展過來的。

GNU Make符合 *IEEE Standard 1003.2-1992* (POSIX.2) 6.2章節的規定。

因為C語言程式更具有代表性，所以我們的例子基於C語言程式，但Make並不是僅僅能夠處理C語言程式，它可以處理那些編譯器能夠在Shell命令下運行的的各種語言的程式。事實上，GNU Make不僅僅限於程式，它可以適用於任何如果一些檔變化導致另外一些檔必須更新的任務。

如果要使用Make，必須先寫一個稱為Makefile的檔，該檔描述程式中各個檔之間的相互關係，並且提供每一個檔的更新命令。在一個程式中，可執行程式檔的更新依靠OBJ檔，而OBJ檔是由原始檔案編譯得來的。

一旦合適的Makefile文件存在，每次更改一些原始檔案，在shell命令下簡單的鍵入：

```
make
```

就能執行所有的必要的重新編譯任務。Make程式根據Makefile檔中的資料和每個檔更改的時間戳決定哪些檔需要更新。對於這些需要更新的檔，Make基於Makefile檔發佈命令進行更新，進行更新的方式由提供的命令行參數控制。具體操作請看*運行* ***Make*** 章節。

1.1 怎樣閱讀本手冊

如果您現在對Make一無所知或者您僅需要瞭解對make 的普通性介紹，請查閱前幾章內容，略過後面的章節。前幾章節是普通介紹性內容，後面的章節是具體的專業、技術內容。

如果您對其他Make程式十分熟悉，請參閱 ***GNU Make*** *的特點和不相容性和失去的特點*部分，***GNU Make*** *的特點*這一章列出了GNU Make對make程式的擴展，*不相容和失去的特點*一章解釋了其他Make程式有的特徵而GNU Make缺乏的原因。

對於快速流覽者，請參閱*選項概要*、*快速參考*和*內建的特殊目標名*部分。

1.2 問題和 BUG

如果您有關於GNU Make的問題或者您認為您發現了一個BUG，請向開發者報告；我們不能許諾我們能幹什麼，但我們會盡力修正它。在報告BUG之前，請確定您是否真正發現了 BUG，仔細研究文檔後確認它是否真的按您的指令運行。如果文檔不能清楚的告訴您怎麼做，也要報告它，這是文檔的一個BUG。

在您報告或者自己親自修正BUG之前，請把它分離出來，即在使問題暴露的前提下盡可能的縮小Makefile文件。然後把這個Makefile檔和Make給出的精確結果發給我們。同時請說明您希望得到什麼，這可以幫助我們確定問題是否出在文檔上。

一旦您找到一個精確的問題，請給我們發 E-mail，我們的 E-mail 地址是：
bug-make@gnu.org
在郵件中請包含您使用的 GNU Make 的版本號。您可以利用命令‘make--version’得到版本號。同時希望您提供您的機器型號和作業系統類型，如有可能的話，希望同時提供 config.h 檔（該檔有配置過程產生）。

2 Makefile 檔介紹

Make 程式需要一個所謂的 Makefile 檔來告訴它幹什麼。在大多數情況下，Makefile 檔告訴 Make 怎樣編譯和連接成一個程式。

本章我們將討論一個簡單的 Makefile 檔，該檔描述怎樣將 8 個 C 根源程式檔和 3 個頭檔編譯和連接成為一個文本編輯器。Makefile 檔可以同時告訴 Make 怎樣運行所需要的雜亂無章的命令（例如，清除操作時刪除特定的檔）。如果要看更詳細、複雜的 Makefile 檔例子，請參閱*複雜的* ***Makefile*** *檔例子*一章。

當 Make 重新編譯這個編輯器時，所有改動的 C 語言原始檔案必須重新編譯。如果一個頭檔改變，每一個包含該頭檔的 C 語言原始檔案必須重新編譯，這樣才能保證生成的編輯器是所有原始檔案更新後的編輯器。每一個 C 語言原始檔案編譯後產生一個對應的 OBJ 檔，如果一個原始檔案重新編譯，所有的 OBJ 檔無論是剛剛編譯得到的或原來編譯得到的必須從新連接，形成一個新的可執行檔。

2.1 規則的格式

一個簡單的 Makefile 檔包含一系列的“規則”，其樣式如下：

目標(target)…: 依賴(prerequiries)…
<tab>命令(command)
…
…

目標(target)通常是要產生的檔的名稱，目標的例子是可執行檔或 OBJ 檔。目標也可是一個執行的動作名稱，諸如‘clean’（詳細內容請參閱*假想目標*一節）。

依賴是用來輸入從而產生目標的檔，一個目標經常有幾個依賴。

命令是 Make 執行的動作，一個規則可以含有幾個命令，每個命令占一行。注意：每個命令行前面必須是一個 **Tab** 字元，即命令行第一個字元是 **Tab**。這是不小心容易出錯的地方。

通常，如果一個依賴發生變化，則需要規則調用命令對相應依賴和服務進行處理從而更新或創建目標。但是，指定命令更新目標的規則並不都需要依賴，例如，包含和目標‘clern’相聯繫的刪除命令的規則就沒有依賴。

規則一般是用於解釋怎樣和何時重建特定檔的，這些特定檔是這個詳盡規則的目標。Make 需首先調用命令對依賴進行處理，進而才能創建或更新目標。當然，一個規則也可以是用於解釋怎樣和何時執行一個動作，詳見*編寫規則*一章。

一個 Makefile 檔可以包含規則以外的其他文本，但一個簡單的 Makefile 檔僅僅需要包含規則。雖然真正的規則比這裏展示的例子複雜，但格式卻是完全一樣。

2.2 一個簡單的 Makefile 檔

一個簡單的 Makefile 檔，該檔描述了一個稱為文本編輯器（edit）的可執行檔生成方法，該檔依靠 8 個 OBJ 檔（.o 檔），它們又依靠 8 個 C 根源程式檔和 3 個頭檔。

在這個例子中，所有的 C 語言原始檔案都包含‘defs.h’ 頭檔，但僅僅定義編輯命令的原始檔案包含‘command.h’頭檔，僅僅改變編輯器緩衝區的低層檔包含‘buffer.h’頭文件。

```
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
```

我們把每一個長行使用反斜杠-新行法分裂為兩行或多行，實際上它們相當於一行，這樣做的意圖僅僅是為了閱讀方便。

使用 Makefile 檔創建可執行的稱為‘edit’的檔，鍵入：make

使用 Makefile 檔從目錄中刪除可執行檔和目標，鍵入：make clean

在這個 Makefile 檔例子中，目標包括可執行檔‘edit’和 OBJ 檔‘main.o’及‘kdb.o’。依賴是 C 語言原始檔案和 C 語言頭檔如‘main.c’和‘def.h’等。事實上，每一個 OBJ 檔即是目標也是依賴。所以命令行包括‘cc -c main.c’和‘cc -c kbd.c’。

當目標是一個檔時，如果它的任一個依賴發生變化，目標必須重新編譯和連接。任何命令行的第一個字元必須是‘Tab’字元，這樣可以把 Makefile 檔中的命令行與其他行分別開來。（一定要牢記：**Make** 並不知道命令是如何工作的，它僅僅能向您提供保證目標的合適更新的命令。**Make** 的全部工作是當目標需要更新時，按照您制定的具體規則執行命令。）

目標‘clean’不是一個檔，僅僅是一個動作的名稱。正常情況下，在規則中‘clean’這個動作並不執行，目標‘clean’也不需要任何依賴。一般情況下，除非特意告訴 make 執行‘clean’命令，否則‘clean’命令永遠不會執行。注意這樣的規則不需要任何依賴，它們存在的目的僅僅是執行一些特殊的命令。象這些不需要依賴僅僅表達動作的目標稱為假想目標。詳細內容參見*假想目標*；參閱*命令錯誤*可以瞭解 rm 或其他命令是怎樣導致 make 忽略錯誤的。

2.3 make 處理 makefile 檔的過程

缺省情況下，make 開始於第一個目標（假想目標的名稱前帶‘.’）。這個目標稱為缺省最終目標（即 make 最終更新的目標，具體內容請*看指定最終目標的參數*一節）。

在上節的簡單例子中，缺省最終目標是更新可執行檔‘edit’，所以我們將該規則設為第一規則。這樣，一旦您給出命令：

```
make
```

make 就會讀當前目錄下的 makefile 檔，並開始處理第一條規則。在本例中，第一條規則是連接生成‘edit’，但在 make 全部完成本規則工作之前，必須先處理‘edit’所依靠的 OBJ 檔。這些 OBJ 檔按照各自的規則被處理更新，每個 OBJ 檔的更新規則是編譯其原始檔案。重新編譯根據其依靠的原始檔案或頭檔是否比現存的 OBJ 檔更‘新’，或者 OBJ 檔是否存在來判

斷。

其他規則的處理根據它們的目標是否和缺省最終目標的依賴相關聯來判斷。如果一些規則和缺省最終目標無任何關聯則這些規則不會被執行，除非告訴 Make 強制執行（如輸入執行 make clean 命令）。

在 OBJ 檔重新編譯之前，Make 首先檢查它的依賴 C 語言原始檔案和 C 語言頭檔是否需要更新。如果這些 C 語言原始檔案和 C 語言頭檔不是任何規則的目標，make 將不會對它們做任何事情。Make 也可以自動產生 C 語言根源程式，這需要特定的規則，如可以根據 Bison 或 Yacc 產生 C 語言根源程式。

在 OBJ 檔重新編譯（如果需要的話）之後，make 決定是否重新連接生成 edit 可執行檔。如果 edit 可執行檔不存在或任何一個 OBJ 檔比存在的 edit 可執行檔'新'，則 make 重新連接生成 edit 可執行檔。

這樣，如果我們修改了'insert.c'檔，然後運行 make，make 將會編譯'insert.c'檔更新'insert.o'檔，然後重新連接生成 edit 可執行檔。如果我們修改了'command.h'檔，然後運行 make，make 將會重新編譯'kbd.o'和'command.o'檔，然後重新連接生成 edit 可執行檔。

2.4 使用變數簡化 **makefile** 檔

在我們的例子中，我們在'edit'的生成規則中把所有的 OBJ 檔列舉了兩次，這裏再重複一遍：

```
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
```

這樣的兩次列舉有出錯的可能，例如在系統中加入一個新的 OBJ 檔，我們很有可能在一個需要列舉的地方加入了，而在另外一個地方卻忘記了。我們使用變數可以簡化 makefile 檔並且排除這種出錯的可能。變數是定義一個字串一次，而能在多處替代該字串使用（具體內容請閱讀*使用變數*一節）。

在 makefile 檔中使用名為 objects, OBJECTS, objs, OBJS, obj, 或 OBJ 的變數代表所有 OBJ 檔已是約定成俗。在這個 makefile 檔我們定義了名為 objects 的變數，其定義格式如下：

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

然後，在每一個需要列舉 OBJ 檔的地方，我們使用寫為`$(objects)'形式的變數代替（具體內容請閱讀*使用變數*一節）。下面是使用變數後的完整的 makefile 檔：

```
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 讓 make 推斷命令

編譯單獨的 C 語言根源程式並不需要寫出命令，因為 make 可以把它推斷出來：make 有一個使用‘CC –c’命令的把 C 語言根源程式編譯更新為相同檔案名的 OBJ 檔的隱含規則。例如 make 可以自動使用‘cc -c main.c -o main.o’命令把‘main.c’編譯 ‘main.o’。因此，我們可以省略 OBJ 檔的更新規則。詳細內容請看*使用隱含規則*一節。

如果 C 語言根源程式能夠這樣自動編譯，則它同樣能夠自動加入到依賴中。所以我們可在依賴中省略 C 語言根源程式，進而可以省略命令。下面是使用隱含規則和變數 objects 的完整 makefile 檔的例子：

```
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)
```

這是我們實際編寫 makefile 檔的例子。（和目標‘clean’聯繫的複雜情況在別處闡述。具體參見*假想目標*及*命令錯誤*兩節內容。）因為隱含規則十分方便，所以它們非常重要，在 makefile 文件中經常使用它們。

2.6 另一種風格的 makefile 檔

當時在 makefile 檔中使用隱含規則創建 OBJ 檔時，採用另一種風格的 makefile 檔也是可行的。在這種風格的 makefile 檔中，可以依據依賴分組代替依據目標分組。下面是採用這種風格的 makefile 檔：

```
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
```

這裏的 defs.h 是所有 OBJ 檔的共同的一個依賴；command.h 和 bufffer.h 是具體列出的 OBJ 檔的共同依賴。

雖然採用這種風格編寫 makefile 檔更具風味：makefile 檔更加短小，但一部分人以為把每一個目標的資訊放到一起更清晰易懂而不喜歡這種風格。

2.7 在目錄中刪除檔的規則

編譯程序並不是編寫 make 規則的唯一事情。Makefile 檔可以告訴 make 去完成編譯程序以外的其他任務，例如，怎樣刪除 OBJ 檔和可執行檔以保持目錄的‘乾淨’等。下面是刪除利用 make 規則編輯器的例子：

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

在實際應用中，應該編寫較為複雜的規則以防不能預料的情況發生。更接近實用的規則樣式如下：

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

這樣可以防止 make 因為存在名為’clean’的檔而發生混亂，並且導致它在執行 rm 命令時發生錯誤（具體參見*假想目標*及*命令錯誤*兩節內容）。

諸如這樣的規則不能放在 makefile 檔的開始，因為我們不希望它變為缺省最終目標。應該象我們的 makefile 檔例子一樣，把關於 edit 的規則放在前面，從而把編譯更新 edit 可執行程式定為缺省最終目標。

3 編寫 **makefile** 檔

make 編譯系統依據的資訊來源於稱為 makefile 檔的資料庫。

3.1 **makefile** 檔的內容

makefile 檔包含 5 方面內容：具體規則、隱含規則、定義變數、指令和注釋。規則、變數和指令將在後續章節介紹。

- ! 具體規則用於闡述什麼時間或怎樣重新生成稱為規則目標的一個或多個檔的。它列舉了目標所依靠的檔，這些檔稱為該目標的依賴。具體規則可能同時提供了創建或更新該目標的命令。詳細內容參閱*編寫規則*一章。
- ! 隱含規則用於闡述什麼時間或怎樣重新生成同一檔案名的一系列檔的。它描述的目標是根據和它名字相同的檔進行創建或更新的，同時提供了創建或更新該目標的命令。詳細內容參閱*使用隱含規則*一節。
- ! 定義變數是為一個變數賦一個固定的字串值，從而在以後的檔中能夠使用該變數代替這個字串。注意在 makefile 檔中定義變數占一獨立行。在上一章的 makefile 檔例子中我們定義了代表所有 OBJ 檔的變數 objects（詳細內容參閱*使用變數簡化* ***makefile*** *檔*一節）。
- ! 指令是 make 根據 makefile 檔執行一定任務的命令。這些包括如下幾方面:
  - " 讀其他 makefile 檔（詳細內容參見*包含其他的* ***makefile*** *檔*）。
  - " 判定（根據變數的值）是否使用或忽略 makefile 檔的部分內容（詳細內容參閱 ***makefile*** *檔的條件語句*一節）。
  - " 定義多行變數，即定義變數值可以包含多行字元的變數（詳細內容參見*定義多行變數*一節）。
- ! 以‘#’開始的行是注釋行。注釋行在處理時將被 make 忽略，如果一個注釋行在行尾是‘\’則表示下一行繼續為注釋行，這樣注釋可以持續多行。除在 define 指令內部外，注釋可以出現在 makefile 檔的任何地方，甚至在命令內部（這裏 shell 決定什麼是注釋內容）。

3.2 makfile 檔的命名

缺省情況下，當 make 尋找 makefile 檔時，它試圖搜尋具有如下的名字的檔，按順序：‘GNUmakefile’、‘makefile’和‘Makefile’。

通常情況下您應該把您的 makefile 檔命名為‘makefile’或‘Makefile’。(我們推薦使用‘Makefile’，因為它基本出現在目錄列表的前面，後面挨著其他重要的文件如‘README’等。)。雖然首先搜尋‘GNUmakefile’，但我們並不推薦使用。除非您的 makefile 檔是特為 GNU make 編寫的，在其他 make 版本上不能執行，您才應該使用‘GNUmakefile’作為您的 makefile 的檔案名。

如果 make 不能發現具有上面所述名字的檔，它將不使用任何 makefile 檔。這樣您必須使用命令參數給定目標，make 試圖利用內建的隱含規則確定如何重建目標。詳細內容參見*使用隱含規則*一節。

如果您使用非標準名字 makefile 檔，您可以使用‘-f’或‘--file’參數指定您的 makefile 檔。參數‘-f name’或‘--file=name’能夠告訴 make 讀名字為‘name’的檔作為 makefile 檔。如果您使用 ‘-f’或‘--file’參數多於一個，意味著您指定了多個 makefile 檔，所有的 makefile 檔按具體的順序發生作用。一旦您使用了‘-f’或‘--file’參數，將不再自動檢查是否存在名為‘GNUmakefile’、‘makefile’或‘Makefile’的 makefile 文件。

3.3 包含其他的 makefile 文件

include 指令告訴 make 暫停讀取當前的 makefile 檔，先讀完 include 指令指定的 makefile 檔後再繼續。指令在 makefile 檔占單獨一行，其格式如下：

include *filenames...*

filenames 可以包含 shell 檔案名的格式。

在 include 指令行，行開始處的多餘的空格是允許的，但 make 處理時忽略這些空格，注意該行不能以 Tab 字元開始（因為，以 Tab 字元開始的行，make 認為是命令行）。include 和檔案名之間以空格隔開，兩個檔案名之間也以空格隔開，多餘的空格 make 處理時忽略，在該行的尾部可以加上以‘#’為起始的注釋。檔案名可以包含變數及函數調用，它們在處理時由 make 進行擴展（具體內容參閱*使用變數*一節）。

例如，有三個‘.mk’文件：‘a.mk’、‘b.mk’和‘c.mk’，變數$(bar)擴展為 bish bash，則下面的表達是：

include foo *.mk $(bar)

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

當 make 遇見 include 指令時， make 就暫停讀取當前的 makefile 文件，依次讀取列舉的 makefile 檔，讀完之後，make 再繼續讀取當前 makefile 檔中 include 指令以後的內容。

使用 include 指令的一種情況是幾個程式分別有單獨的 makefile 檔，但它們需要一系列共同的變數定義（詳細內容參閱*設置變數*），或者一系列共同的格式規則（詳細內容參閱*定義與重新定義格式規則*）。

另一種使用 include 指令情況是需要自動從原始檔案為目標產生依賴的情況，此時，依賴在主 makefile 檔包含的檔中。這種方式比其他版本的 make 把依賴附加在主 makefile 檔後部的傳統方式更顯得簡潔。具體內容參閱*自動產生依賴*。

如果 makefile 檔案名不以‘/’開頭，並且在當前目錄下也不能找到，則需搜尋另外的目錄。首先，搜尋以‘-I’或‘--include-dir’參數指定的目錄，然後依次搜尋下面的目錄（如果它們存在的話）：‘*prefix*/include' (通常為 ‘/usr/local/include') ‘/usr/gnu/include', ‘/usr/local/include', ‘/usr/include'。

如果指定包含的 makefile 檔在上述所有的目錄都不能找到，make 將產生一個警告資訊，注意這不是致命的錯誤。處理完 include 指令包含的 makefile 檔之後，繼續處理當前的 makefile 檔。一旦完成 makefile 檔的讀取操作，make 將試圖創建或更新過時的或不存在的 makefile 文件。詳細內容參閱 ***makefile*** *檔重新生成的過程*。只有在所有 make 尋求丟失的 makefile 檔的努力失敗後，make 才能斷定丟失的 makefile 檔是一個致命的錯誤。

如果您希望對不存在且不能重新創建的 makefile 檔進行忽略，並且不產生錯誤資訊，則使用-include 指令代替 include 指令，格式如下：
-include *filenames...*
這種指令的作用就是對於任何不存在的 makefile 檔都不會產生錯誤(即使警告資訊也不會產生)。如果希望保持和其他版本的 make 相容，使用 sinclude 指令代替-include 指令。

3.4 變數 MAKEFILES

如果定義了環境變數 MAKEFILES，make 認為該變數的值是一列附加的 makefile 檔案名，檔案名之間由空格隔開，並且這些 makefile 檔應首先讀取。Make 完成這個工作和上節完成 include 指令的方式基本相同，即在特定的目錄中搜尋這些檔。值得注意的是，缺省最終目標不會出現在這些 makefile 檔中，而且如果一些 makefile 檔沒有找到也不會出現任何錯誤資訊。

環境變數 MAKEFILES 主要在 make 遞迴調用過程中起通訊作用（詳細內容參閱*遞迴調用* ***make***）。在 make 頂級調用之前設置環境變數並不是十分好的主意，因為這樣容易將 makefile 檔與外界的關係弄的更加混亂。然而如果運行 make 而缺少 makefile 檔時，環境變數 MAKEFILES 中 makefile 檔可以使內置的隱含規則更好的發揮作用，如搜尋定義的路徑等（詳細內容參閱*在目錄中搜尋依賴*）。

一些用戶喜歡在登錄時自動設置臨時的環境變數 MAKEFILES，而 makefile 檔在該變數指定的檔無效時才使用。這是非常糟糕的主意，應為許多 makefile 檔在這種情況下運行失效。最好的方法是直接在 makefile 檔中寫出具體的 include 指令(詳細內容參看上一節)。

3.5 makefile 檔重新生成的過程

有時 makefile 檔可以由其他檔重新生成，如從 RCS 或 SCCS 文件生成等。如果一個 makefile 檔可以從其他檔重新生成，一定注意讓 make 更新 makefile 檔之後再讀取 makefile 檔。

完成讀取所有的 makefile 檔之後，make 檢查每一個目標，並試圖更新它。如果對於一個 makefile 檔有說明它怎樣更新的規則(無論在當前的 makefile 文件中或其他 makefile 文件中)，或者存在一條隱含規則說明它怎樣更新（具體內容參見*使用隱含規則*），則在必要的時候該 makefile 檔將會自動更新。在所有的 makefile 檔檢查之後，如果發現任何一個 makefile 檔發生變化，make 就會清空所有記錄，並重新讀入所有 makefile 檔。(然後再次試圖更新這些 makefile 檔，正常情況下，因為這些 makefile 檔已被更新，make 將不會再更改它們。)

如果您知道您的一個或多個 makefile 檔不能重新創建，也許由於執行效率緣故，您不希望 make 按照隱含規則搜尋或重建它們，您應使用正常的方法阻止按照隱含規則檢查它們。例如，您可以寫一個具體的規則，把這些 makefile 檔當作目標，但不提供任何命令（詳細內容參閱*使用空命令*）。

如果在 makefile 檔中指定依據雙冒號規則使用命令重建一個檔，但沒有提供依賴，則一旦 make 運行就會重建該檔（詳細內容參見*雙冒號規則*）。同樣，如果在 makefile 檔中指定依據雙冒號規則使用命令重建的一個 makefile 檔，並且不提供依賴，則一旦 make 運行就會重建該 makefile 檔，然後重新讀入所有 makefile 檔，然後再重建該 makefile 檔，再重新讀入所有 makefile 檔，如此往復陷入無限迴圈之中，致使 make 不能再完成別的任務。如果要避免上述情況的發生，一定注意不要依據雙冒號規則使用命令並且不提供依賴重建任何 **makefile** 檔。

如果您沒有使用‘-f’或‘--file’指定 makefile 檔，make 將會使用缺省的 makefile 檔案名(詳細內容參見 3.2 節內容)。不象使用‘-f’或‘--file’選項指定具體的 makefile 檔，這時 make 不能確定 makefile 檔是否存在。如果缺省的 makefile 檔不存在，但可以由運行的 make 依據規則創建，您需要運行這些規則，創建要使用的 makefile 檔。

如果缺省的 makefile 檔不存在，make 將會按照搜尋的次序將它們試著創建，一直到將 makefile 檔成功創建或 make 將所有的檔案名都試過來。注意 make 不能找到或創建 makefile

檔不是錯誤，makefile 檔並不是運行 make 必須的。

因為即使您使用‘-t’特別指定，‘-t’或‘--touch’選項對更新 makefile 檔不產生任何影響，makefile 檔仍然會更新，所以當您使用‘-t’或‘--touch’選項時，您不要使用過時的 makefile 文件來決定‘touch’哪個目標（具體含義參閱*代替執行命令*）。同樣，因為‘-q' (或 ‘--question') 和‘-n' (或 ‘--just-print')也能不阻止更新 makefile 文件，所以過時的 makefile 檔對其他的目標將產生錯誤的輸出結果。如，‘make -f mfile -n foo’命令將這樣執行：更新‘mfile’，然後讀入，再輸出更新‘foo’的命令和依賴，但並不執行更新‘foo’，注意，所有回顯的更新‘foo’的命令是在更新後的‘mfile’中指定的。

在實際使用過程中，您一定會遇見確實希望阻止更新 makefile 檔的情況。如果這樣，您可以在 makefile 檔命令行中將需要更新的 makefile 檔指定為目標，如此則可阻止更新 makefile 檔。一旦 makefile 檔案名被明確指定為一個目標，選項‘-t’等將會對它發生作用。如這樣設定，‘make -f mfile -n foo’命令將這樣執行：讀入‘mfile’，輸出更新‘foo’的命令和依賴，但並不執行更新‘foo’。回顯的更新‘foo’的命令包含在現存的‘mfile’中。

3.6 重載其他 makefile 文件

有時一個 makefile 檔和另一個 makefile 檔相近也是很有用的。您可以使用‘include’指令把更多的 makefile 檔包含進來，如此可加入更多的目標和定義的變數。然而如果兩個 makefile 檔對相同的目標給出了不同的命令，make 就會產生錯誤。

在主 makefile 檔（要包含其他 makefile 檔的那個）中，您可以使用通配符格式規則說明只有在依靠當前 makefile 檔中的資訊不能重新創建目標時，make 才搜尋其他的 makefile 檔，詳細內容參見*定義與重新定義格式規則*。

例如：如果您有一個說明怎樣創建目標‘foo’（和其他目標）的 makefile 檔稱為‘Makefile’，您可以編寫另外一個稱為‘GNUmakefile’的 makefile 檔包含以下語句：

```
foo:
        frobnicate > foo

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

如果鍵入‘make foo’，make 就會找到‘GNUmakefile’，讀入，然後運行‘frobnicate > foo’。如果鍵入‘make bar’，make 發現無法根據‘GNUmakefile’創建‘bar’，它將使用格式規則提供的命令：‘make –f Makefile bar’。如果在‘Makefile’中提供了‘bar’更新的規則，make 就會使用該規則。對其他‘GNUmakefile’不提供怎樣更新的目標 make 也會同樣處理。這種工作的方式是使用了格式規則中的格式匹配符‘%’，它可以和任何目標匹配。該規則指定了一個依賴‘force’，用來保證命令一定要執行，無論目標檔是否存在。我們給出的目標‘force’時使用了空命令，這樣可防止 make 按照隱含規則搜尋和創建它，否則，make 將把同樣的匹配規則應用到目標‘force’本身，從而陷入創建依賴的迴圈中。

3.7 make 讀取 makefile 檔的過程

GNU make 把它的工作明顯的分為兩個階段。在第一階段，make 讀取 makefile 檔，包括 makefile 檔本身、內置變數及其值、隱含規則和具體規則、構造所有目標的依靠圖表和它們的依賴等。在第二階段，make 使用這些內置的組織決定需要重新構造的目標以及使用必要的規則進行工作。

瞭解 make 兩階段的工作方式十分重要，因為它直接影響變數、函數擴展方式；而這也是編寫 makefile 檔時導致一些錯誤的主要來源之一。下面我們將對 makefile 檔中不同結構的擴展方式進行總結。我們稱在 make 工作第一階段發生的擴展是立即擴展：在這種情況下，make 對 makefile 檔進行語法分析時把變數和函數直接擴展為結構單元的一部分。我們把不能立即執行的擴展稱為延時擴展。延時擴展結構直到它已出現在上下文結構中或 make 已進入到了第二工作階段時才執行展開。

您可能對這一部分內容不熟悉。您可以先看完後面幾章對這些知識熟悉後再參考本節內容。

變數賦值

變數的定義語法形式如下：

immediate = *deferred*
immediate ?= *deferred*
immediate := *immediate*
immediate += *deferred* or *immediate*

define *immediate*
  deferred
endef

對於附加操作符'+='，右邊變數如果在前面使用（:=）定義為簡單擴展變數則是立即變數，其他均為延時變數。

條件語句

整體上講，條件語句都按語法立即分析，常用的有：ifdef、ifeq、ifndef 和 inneq。

定義規則

規則不論其形式如何，都按相同的方式擴展。

immediate : *immediate* ; *deferred*
        deferred

目標和依賴部分都立即擴展，用於構造目標的命令通常都是延時擴展。這個通用的規律對具體規則、格式規則、尾碼規則、靜態格式規則和簡單依賴定義都適用。

4 編寫規則

makefile 檔中的規則是用來說明何時以及怎樣重建特定檔的，這些特定的檔稱為該規則的目標（通常情況下，每個規則只有一個目標）。在規則中列舉的其他檔稱為目標的依賴，同時規則還給出了目標創建、更新的命令。一般情況下規則的次序無關緊要，但決定缺省最終目標時卻是例外。缺省最終目標是您沒有另外指定最終目標時，make 認定的最終目標。缺省最終目標是 makefile 檔中的第一條規則的目標。如果第一條規則有多個目標，只有第一個目標被認為是缺省最終目標。有兩種例外的情況：以句點（'.'）開始的目標不是缺省最終目標（如果該目標包含一個或多個斜杠'/'，則該目標也可能是缺省最終目標）；另一種情況是格式規則定義的目標不是缺省最終目標（參閱*定義與重新定義格式規則*）。

所以，我們編寫 makefile 檔時，通常將第一個規則的目標定為編譯全部程式或是由 makefile 檔表述的所有程式（經常設定一個稱為'all'的目標）。參閱*指定最終目標的參數*。

4.1 規則的語法

通常一條規則形式如下：

targets : *prerequisites*
        command
        ...

或：

targets : *prerequisites* ; *command*
        command
        ...

目標（target）是檔的名稱，中間由空格隔開。通配符可以在檔案名中使用（參閱*在檔案名中使用通配符*），‘a（m）’形式的檔案名表示成員 m 在檔 a 中（參閱*檔案成員目標*）。一般情況下，一條規則只有一個目標，但偶爾由於其他原因一條規則有多個目標（參閱*具有多個目標的規則*）。

命令行以 Tab 字元開始，第一個命令可以和依賴在一行，命令和依賴之間用分號隔開，也可以在依賴下一行，以 Tab 字元為行的開始。這兩種方法的效果一樣，參閱*在規則中使用命令*。

因為美元符號已經用為變數引用的開始符，如果您真希望在規則中使用美元符號，您必須連寫兩次，‘$$’（參閱*使用變數*）。您可以把一長行在中間插入‘\’使其分為兩行，也就是說，一行的尾部是’\’的話，表示下一行是本行的繼續行。但這並不是必須的，make 沒有對 makefile 檔中行的長度進行限制。一條規則可以告訴 make 兩件事情：何時目標已經過時，以及怎樣在必要時更新它們。

判斷目標過時的準則和依賴關係密切，依賴也由檔案名構成，檔案名之間由空格隔開，通配符和檔案成員也允許在依賴中出現。一個目標如果不存在或它比其中一個依賴的修改時間早，則該目標已經過時。該思想來源於目標是根據依賴的資訊計算得來的，因此一旦任何一個依賴發生變化，目標檔也就不再有效。目標的更新方式由命令決定。命令由 shell 解釋執行，但也有一些另外的特點。參閱*在規則中使用命令*。

4.2 在檔案名中使用通配符

一個簡單的檔案名可以通過使用通配符代表許多檔。Make 中的通配符和 Bourne shell 中的通配符一樣是‘*’、‘？’和‘[…]’。例如：‘*.C’指在當前目錄中所有以‘.C’結尾的文件。

字元‘~’在檔案名的前面也有特殊的含義。如果字元‘~’單獨或後面跟一個斜杠‘/’，則代表您的 home 目錄。如‘~/bin’擴展為‘/home/bin’。如果字元‘~’後面跟一個字，它擴展為 home 目錄下以該字為名字的目錄，如‘~John/bin’表示‘home/John/bin’。在一些作業系統（如 ms-dos，ms-windows）中不存在 home 目錄，可以通過設置環境變數 home 來類比。

在目標、依賴和命令中的通配符自動擴展。在其他上下文中，通配符只有在您明確表明調用通配符函數時才擴展。

通配符另一個特點是如果通配符前面是反斜杠‘\’，則該通配符失去通配能力。如‘foo*bar’表示一個特定的檔其名字由‘foo’、‘*’和‘bar’構成。

4.2.1 通配符例子

通配符可以用在規則的命令中，此時通配符由 shell 擴展。例如，下面的規則刪除所有 OBJ 檔：

```
clean：
        rm –f   *.o
```

通配符在規則的依賴中也很有用。在下面的 makefile 規則中，‘make print’將列印所有從上次您列印以後又有改動的‘.c’文件：

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

本規則使用‘ptint’作為一個空目標檔（參看*使用空目標檔記錄事件*）；自動變數‘$?’用來列印那些已經修改的檔，參看*自動變數*。

當您定義一個變數時通配符不會擴展，如果您這樣寫：

```
objects = *.o
```

變數 objects 的值實際就是字串‘*.o’。然而，如果您在一個目標、依賴和命令中使用變數 objects 的值，通配符將在那時擴展。使用下面的語句可使通配符擴展：

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

詳細內容參閱*函數* ***wildcard***。

4.2.2 使用通配符的常見錯誤

下面有一個幼稚使用通配符擴展的例子，但實際上該例子不能完成您所希望完成的任務。假設可執行檔‘foo’由在當前目錄的所有 OBJ 檔創建，其規則如下：

```
objects = *.o

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

由於變數 objects 的值為字串‘*.o’，通配符在目標‘foo’的規則下擴展，所以每一個 OBJ 檔都會變為目標‘foo’的依賴，並在必要時重新編譯自己。

但如果您已刪除了所有的 OBJ 檔，情況又會怎樣呢？因沒有和通配符匹配的檔，所以目標‘foo’就依靠了一個有著奇怪名字的檔‘*.o’。因為目錄中不存在該檔，make 將發出不能創建‘*.o’的錯誤資訊。這可不是所要執行的任務。

實際上，使用通配符獲得正確的結果是可能的，但您必須使用稍微複雜一點的技術，該技術包括使用函數 wildcard 和替代字串等。詳細內容將在下一節論述。

微軟的作業系統（MS-DOS、MS-WINDOWS）使用反斜杠分離目錄路徑，如：

C:\foo\bar\bar.c

這和 Unix 風格‘c:/foo/bar/bar.c’等價（‘c:’是驅動器字母）。當 make 在這些系統上運行時，不但支援在路徑中存在反斜杠也支持 Unix 風格的前斜杠。但是這種對反斜杠的支持不包括通配符擴展，因為通配符擴展時，反斜杠用作引用字元。所以，在這些場合您必須使用 Unix 風格的前斜杠。

4.2.3 函數 wildcard

通配符在規則中可以自動擴展，但設置在變數中或在函數的參數中通配符一般不能正常擴展。如果您需要在這些場合擴展通配符，您應該使用函數 wildcard，格式如下：

```
$(wildcard pattern...)
```

可以在 makefile 檔的任何地方使用該字串，應用時該字串被一列在指定目錄下存在的並且檔案名和給出的檔案名的格式相符合的檔所代替，檔案名中間由空格隔開。如果沒有和指定格式一致的檔，則函數 wildcard 的輸出將會省略。注意這和在規則中通配符擴展的方式不同，在規則中使用逐字擴展方式，而不是省略方式（參閱上節）。

使用函數 wildcard 得到指定目錄下所有的 C 語言根源程式檔案名的命令格式為：

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

我們可以把所獲得的 C 語言根源程式檔案名的字元串通過將‘.c’尾碼變為‘.o’轉換為 OBJ 檔案名的字串，其格式為：

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

這裏我們使用了另外一個函數：patsubst，詳細內容參閱*字串替換和分析函數*。

這樣，一個編譯特定目錄下所有 C 語言根源程式並把它們連接在一起的 makefile 檔可以寫成如下格式：

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

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

這裏使用了編譯 C 語言根源程式的隱含規則，因此沒有必要為每個檔寫具體編譯規則。‘:=’是‘=’的變異，對‘:=’的解釋，參閱*兩種風格的變數*。

4.3 在目錄中搜尋依賴

對於大型系統，把原始檔案安放在一個單獨的目錄中，而把二進位檔放在另一個目錄中是十分常見的。Make 的目錄搜尋特性使自動在幾個目錄搜尋依賴十分容易。當您在幾個目

錄中重新安排您的檔，您不必改動單獨的規則，僅僅改動一下搜尋路徑即可。

4.3.1 VPATH：所有依賴的搜尋路徑

make 變數 VPATH 的值指定了 make 搜尋的目錄。經常用到的是那些包含依賴的目錄，並不是當前的目錄；但 VPATH 指定了 make 對所有檔都適用的目錄搜尋序列，包括了規則的目標所需要的檔。

如果一個作為目標或依賴的檔在當前目錄中不存在，make 就會在 VPATH 指定的目錄中搜尋該檔。如果在這些目錄中找到要尋找的檔，則就象這些檔在當前目錄下存在一樣，規則把這些檔指定為依賴。參閱*編寫搜尋目錄的* ***shell*** *命令*。

在 VPATH 變數定義中，目錄的名字由冒號或空格分開。目錄列舉的次序也是 make 搜尋的次序。在 MS-DOS、MS-WINDOWS 系統中，VPATH 變數定義中的目錄的名字由分號分開，因為在這些系統中，冒號用為路徑名的一部分（通常在驅動器字母後面）。例如：

VPATH = src:../headers

指定了兩個目錄，‘src’和‘…/headers’，make 也按照這個次序進行搜尋。使用該 VPATH 的值，下面的規則，

foo.o : foo.c

在執行時就象如下寫法一樣會被中斷：

foo.o : src/foo.c

然後在 src 目錄下搜尋 foo.c。

4.3.2 vpath 指令

vpath 指令（注意字母是小寫）和 VPATH 變數類似，但卻更具靈活性。vpath 指令允許對符合一定格式類型的檔案名指定一個搜尋路徑。這樣您就可以對一種格式類型的檔案名指定一個搜尋路徑，對另外格式類型的檔案名指定另外一個搜尋路徑。總共由三種形式的 vpath 指令：

vpath ***pattern directories***

對一定格式類型的檔案名指定一個搜尋路徑。搜尋的路徑由一列要搜尋的目錄構成，目錄由冒號（在 **MS-DOS**、**MS-WINDOWS** 系統中用分號）或空格隔開，和 **VPATH** 變數定義要搜尋的路徑格式一樣。

vpath ***pattern***

清除和一定類型格式相聯繫的搜尋路徑。

vpath

清除所有前面由 **vapth** 指令指定的搜尋路徑。

一個 vpath 的格式 pattern 是一個包含一個’%’的字串。該字串必須和正搜尋的一個依賴的檔案名匹配，字元%可和任何字串匹配（關於格式規則，參閱*定義與重新定義格式規則*）。例如，%.h 和任何檔案名以.h 結尾的文件匹配。如果不使用‘%’，格式必須與依賴精確匹配，這種情況很少使用。

在 vpath 指令格式中的字元‘%’可以通過前面的反斜杠被引用。引用其他字元‘%’的反斜杠也可以被更多的反斜杠引用。引用字元‘%’和其他反斜杠的反斜杠在和檔案名比較之前和格式是分開的。如果反斜杠所引用的字元‘%’沒有錯誤，則該反斜杠不會運行帶來任何危害。

如果 vpath 指令格式和一個依賴的檔案名匹配，並且在當前目錄中該依賴不存在，則 vpath 指令中指定的目錄和 VPATH 變數中的目錄一樣可以被搜尋。例如：

vpath %.h ../headers

將告訴 make 如果在當前目錄中以‘.h’結尾文件不存在，則在‘../headers’目錄下搜尋任何以‘.h’結尾依賴。

如果有幾個 vpath 指令格式和一個依賴的檔案名匹配，則 make 一個接一個的處理它們，搜尋所有在指令中指定的目錄。Make 按它們在 makefile 檔中出現的次序控制多個 vpath 指

令，多個指令雖然有相同的格式，但它們是相互獨立的。以下代碼：
vpath %.c foo
vpath % blish
vpath %.c bar
表示搜尋`.c'檔先搜尋目錄`foo'、然後`blish'，最後`bar'；如果是如下代碼：
vpath %.c foo:bar
vpath % blish
表示搜尋`.c'檔先搜尋目錄‘foo'、然後‘bar'，最後‘blish'。

4.3.3 目錄搜尋過程

當通過目錄搜尋找到一個檔，該檔有可能不是您在依賴列表中所列出的依賴；有時通過目錄搜尋找到的路徑也可能被廢棄。Make 決定對通過目錄搜尋找到的路徑保存或廢棄所依據的演算法如下：

1、1、如果一個目標檔在 makefile 檔所在的目錄下不存在，則將會執行目錄搜尋。
2、2、如果目錄搜尋成功，則路徑和所得到的檔暫時作為目標檔儲存。
3、3、所有該目標的依賴用相同的方法考察。
4、4、把依賴處理完成後，該目標可能需要或不需要重新創建：
   1、1、如果該目標不需要重建，目錄搜尋時所得到的檔的路徑用作該目標所有依賴的路徑，同時包含該目標檔。簡而言之，如果 make 不必重建目標，則您使用通過目錄搜尋得到的路徑。
   2、2、如果該目標需要重建，目錄搜尋時所得到的檔的路徑將廢棄，目標檔在 makefile 檔所在的目錄下重建。簡而言之，如果 make 要重建目標，是在 makefile 檔所在的目錄下重建目標，而不是在目錄搜尋時所得到的檔的路徑下。

該演算法似乎比較複雜，但它卻可十分精確的解釋實際您所要的東西。

其他版本的 make 使用一種比較簡單的演算法：如果目標檔在當前目錄下不存在，而它通過目錄搜尋得到，不論該目標是否需要重建，始終使用通過目錄搜尋得到的路徑。

實際上，如果在 GNU make 中使您的一些或全部目錄具備這種行為，您可以使用 GPATH 變數來指定這些目錄。

GPATH 變數和 VPATH 變數具有相同的語法和格式。如果通過目錄搜尋得到一個過時的目標，而目標存在的目錄又出現在 GPATH 變數，則該路徑將不廢棄，目標將在該路徑下重建。

4.3.4 編寫目錄搜尋的 shell 命令

即使通過目錄搜尋在其他目錄下找到一個依賴，不能改變規則的命令，這些命令同樣按照原來編寫的方式執行。因此，您應該小心的編寫這些命令，以便它們可以在 make 能夠在發現依賴的目錄中處理依賴。

借助諸如‘$^’的自動變數可更好的使用 shell 命令（參閱*自動變數*）。例如，‘$^’的值代表所有的依賴列表，並包含尋找依賴的目錄；‘$@’的值是目標。
foo.o : foo.c
	cc -c $(CFLAGS) $^ -o $@

變數 CFLAGS 存在可以方便您利用隱含規則指定編譯 C 語言根源程式的旗標。我們這裏使用它是為了保持編譯 C 語言根源程式一致性。參閱*隱含規則使用的變數*。

依賴通常情況下也包含頭檔，因自動變數‘$<’的值是第一個依賴，因此這些頭檔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 目錄搜尋和隱含規則

搜尋的目錄是由變數 VPATH 或隱含規則引入的 vpath 指令指定的（詳細參閱*使用隱含規則*）。例如，如果檔‘foo.o’沒有具體的規則，make 則使用隱含規則：如文件 foo.c 存在，make 使用內置的規則編譯它；如果檔 foo.c 不在當前目錄下，就搜尋適當的目錄，如在別的目錄下找到 foo.c，make 同樣使用內置的規則編譯它。

隱含規則的命令使用自動變數是必需的，所以隱含規則可以自然地使用目錄搜尋得到的檔。

4.3.6 連接庫的搜尋目錄

對於連接庫檔，目錄搜尋採用一種特別的方式。這種特別的方式來源於個玩笑：您寫一個依賴，它的名字是‘-|name’的形式。（您可以在這裏寫一些奇特的字元，因為依賴正常是一些檔案名，庫檔案名通常是‘libname.a’ 的形式，而不是‘-|name’ 的形式。）

當一個依賴的名字是‘-|name’的形式時，make 特別地在當前目錄下、與 vpath 匹配的目錄下、VPATH 指定的目錄下以及‘/lib’, ‘/usr/lib', 和 ‘*prefix*/lib'（正常情況為`/usr/local/lib'，但是 MS-DOS、MS-Windows 版本的 make 的行為好像是 *prefix* 定義為 DJGPP 安裝樹的根目錄的情況）目錄下搜尋名字為‘lib*name*.so'的文件然後再處理它。如果沒有搜尋到‘lib*name*.so'檔，然後在前述的目錄下搜尋‘lib*name*.a'文件。

例如，如果在您的系統中有‘/usr/lib/libcurses.a'的庫文件，則：

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

如果‘foo’比‘foo.c’更舊，將導致命令‘cc foo.c /usr/lib/libcurses.a -o foo'執行。

缺省情況下是搜尋‘lib*name*.so' 和‘lib*name*.a'文件，具體搜尋的文件及其類型可使用.LIBPATTERNS 變數指定，這個變數值中的每一個字都是一個字串格式。當尋找名為‘-|name’的依賴時，make 首先用 *name* 替代列表中第一個字中的格式部分形成要搜尋的庫檔案名，然後使用該庫檔案名在上述的目錄中搜尋。如果沒有發現庫檔，則使用列表中的下一個字，其餘以此類推。

.LIBPATTERNS 變數缺省的值是"‘lib%.so lib%.a'"，該值對前面描述的缺省行為提供支援。您可以通過將該值設為空值從而徹底關閉對連接庫的擴展。

4.4 假想目標

假想目標並不是一個真正的檔案名，它僅僅是您制定的一個具體規則所執行的一些命令的名稱。使用假想目標有兩個原因：避免和具有相同名稱的檔衝突和改善性能。

如果您寫一個其命令不創建目標檔的規則，一旦由於重建而提及該目標，則該規則的命令就會執行。這裏有一個例子：

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

因為 rm 命令不創建名為‘clean’的文件，所以不應有名為‘clean’的檔存在。因此不論何時您發佈`make clean'指令，rm 命令就會執行。

假想目標能夠終止任何在目錄下創建名為‘clean’的文件工作。但如在目錄下存在檔 clean，因為該目標 clean 沒有依賴，所以檔 clean 始終會認為已經該更新，因此它的命令將永不會執行。為了避免這種情況，您應該使用象如下特別的.PHONY 目標格式將該目標具體的聲明為一個假想目標：

```
.PHONY : clean
```

一旦這樣聲明，‘make clean’命令無論目錄下是否存在名為‘clean’的檔，該目標的命令都會執行。

因為 make 知道假想目標不是一個需要根據別的檔重新創建的實際檔，所以它將跳過隱含規則搜尋假想目標的步驟（詳細內容參閱*使用隱含規則*）。這是把一個目標聲明為假想目標可以提高執行效率的原因，因此使用假想目標您不用擔心在目錄下是否有實際檔存在。這樣，對前面的例子可以用假想目標的寫出，其格式如下：

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

另外一個使用假想目標的例子是使用 make 的遞迴調用進行連接的情況：此時，makefile 檔常常包含列舉一系列需要創建的子目錄的變數。不用假想目標完成這種任務的方法是使用一條規則，其命令是一個在各個子目錄下迴圈的 shell 命令，如下面的例子：

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

但使用這個方法存在下述問題：首先，這個規則在創建子目錄時產生的任何錯誤都不及時發現，因此，當一個子目錄創建失敗時，該規則仍然會繼續創建剩餘的子目錄。雖然該問題可以添加監視錯誤產生並退出的 shell 命令來解決，但非常不幸的是如果 make 使用了'-k'選項，這個問題仍然會產生。第二，也許更重要的是您使用了該方法就失去使用 make 並行處理的特點能力。

使用假想目標（如果一些子目錄已經存在，您則必須這樣做，否則，不存在的子目錄將不會創建）則可以避免上述問題：

```
SUBDIRS = foo bar baz

.PHONY: subdirs $(SUBDIRS)

subdirs: $(SUBDIRS)

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

foo: baz
```

此時，如果子目錄'baz'沒有創建完成，子目錄'foo'將不會創建；當試圖使用並行創建時這種關係的聲明尤其重要。

一個假想目標不應該是一個實際目標檔的依賴，如果這樣，make 每次執行該規則的命令，目標檔都要更新。只要假想目標不是一個真實目標的依賴，假想目標的命令只有在假想目標作為特別目標時才會執行（參閱*指定最終目標的參數*）。

假想目標也可以有依賴。當一個目錄下包含多個程式時，使用假想目標可以方便的在一個 makefile 檔中描述多個程式的更新。重建的最終目標缺省情況下是 makefile 檔的第一個規則的目標，但將多個程式作為假想目標的依賴則可以輕鬆的完成在一個 makefile 檔中描述多個程式的更新。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
```

這樣，您可以重建所有程式，也可以參數的形式重建其中的一個或多個（如'make prog1 prog3'）。

當一個假想目標是另一個假想目標的依賴，則該假想目標將作為一個假想目標的子常式。例如，這裏'make cleanall'用來刪除 OBJ 檔、diff 檔和程式檔：

```
.PHONY: cleanall cleanobj cleandiff

cleanall : cleanobj cleandiff
        rm program

cleanobj :
        rm *.o

cleandiff :
        rm *.diff
```

4.5 沒有命令或依賴的規則

如果一個規則沒有依賴、也沒有命令，而且這個規則的目標也不是一個存在的檔，則 make 認為只要該規則運行，該目標就已被更新。這意味著，所有以這種規則的目標為依賴的目標，它們的命令將總被執行。這裏舉一個例子：

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

這裏的目標‘FORCR’滿足上面的特殊條件，所以以其為依賴的目標‘clean’將總強制它的命令執行。關於‘FORCR’的名字沒有特別的要求，但‘FORCR’是習慣使用的名字。

也許您已經明白，使用‘FORCR’的方法和使用假想目標（.PHONY: clean）的結果一樣，但使用假想目標更具體更靈活有效，由於一些別的版本的 make 不支援假想目標，所以‘FORCR’出現在許多 makefile 檔中。參閱*假想目標*。

4.6 使用空目標檔記錄事件

空目標是一個假想目標變數，它用來控制一些命令的執行，這些命令可用來完成一些經常需要的具體任務。但又不象真正的假想目標，它的目標檔可以實際存在，但檔的內容與此無關，通常情況下，這些檔沒有內容。

空目標檔的用途是用來記錄規則的命令最後一次執行的時間，也是空目標檔最後更改的時間。它之所以能夠這樣執行是因為規則的命令中有一條用於更新目標檔的‘touch’命令。另外，空目標檔應有一些依賴（否則空目標檔沒有存在的意義）。如果空目標比它的依賴舊，當您命令重建空目標檔時，有關的命令才會執行。下面有一個例子：

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

按照這個規則，如果任何一個原始檔案從上次執行‘make print'以來發生變化，鍵入‘make print'則執行 lpr 命令。自動變數‘$?’用來列印那些發生變化的檔（參閱*自動變數*）。

4.7 內建的特殊目標名

一些名字作為目標使用則含有特殊的意義：

- ! **.PHONY**

  特殊目標.PHONY 的依賴是假想目標。假想目標是這樣一些目標，make 無條件的執行它命令，和目錄下是否存在該檔以及它最後一次更新的時間沒有關係。詳細內容參閱*假想目標*。

- ! **.SUFFIXES**

  特殊目標.SUFFIXES 的依賴是一列用於尾碼規則檢查的尾碼。詳細內容參閱*過時的尾碼規則*。

- ! **.DEFAULT**

  .DEFAULT 指定一些命令，這些命令用於那些沒有找到規則（具體規則或隱含規則）更

新的目標。詳細內容參閱*定義最新類型的-缺省規則*。如果.DEFAULT 指定了一些命令，則所有提及到的檔只能作為依賴，而不能作為任何規則的目標；這些指定的命令也只按照他們自己的方式執行。詳細內容參閱*隱含規則搜尋演算法*。

- ! **.PRECIOUS**
  特殊目標.PRECIOUS 的依賴將按照下面給定的特殊方式進行處理：如果在執行這些目標的命令的過程中，make 被關閉或中斷，這些目標不能被刪除，詳細內容參閱*關閉和中斷* ***make***；如果目標是中間檔，即使它已經沒有任何用途也不能被刪除，具體情況和該目標正常完成一樣，參閱*隱含規則鏈*；該目標的其他功能和特殊目標.SECONDARY 的功能重疊。如果規則的目標格式與依賴的檔案名匹配，您可以使用隱含規則的格式（如‘%.O’）列舉目標作為特殊目標.PRECIOUS 的依賴檔來保存由這些規則創建的中間檔。
- ! **.INTERMEDIATE**
  特殊目標.INTERMEDIATE 的依賴被處理為中間檔。詳細內容參見*隱含規則鏈*。.INTERMEDIATE 如果沒有依賴檔，它將不會發生作用。
- ! **.SECONDARY**
  特殊目標.SECONDARY 的依賴被處理為中間檔，但它們永遠不能自動刪除。詳細內容參見*隱含規則鏈*。.SECONDARY 如果沒有依賴檔，則所有的 makefile 檔中的目標都將被處理為中間檔。
- ! **.DELETE_ON_ERROR**
  如果在 makefile 檔的某處.DELETE_ON_ERROR 作為一個目標被提及，則如果該規則發生變化或它的命令沒有正確完成而退出，make 將會刪除該規則的目標，具體行為和它受到了刪除信號一樣。詳細內容參閱*命令錯誤*。
- ! .IGNORE
  如果您特別為目標.IGNORE 指明依賴，則 MAKE 將會忽略處理這些依賴檔時執行命令產生的錯誤。如果.IGNORE 作為一個沒有依賴的目標提出來，MAKE 將忽略處理所有檔時產生的錯誤。.IGNORE 命令並沒有特別的含義，.IGNORE 的用途僅是為了和早期版本的相容。因為.IGNORE 影響所有的命令，所以它的用途不大；我們推薦您使用其他方法來忽略特定命令產生的錯誤。詳細內容參閱*命令錯誤*。
- ! **.SILENT**
  如果您特別為.SILENT 指明依賴，則在執行之前 MAKE 將不會回顯重新構造檔的命令。如果.SILENT 作為一個沒有依賴的目標提出來，任何命令在執行之前都不會列印。.SILENT 並沒有特別的含義，其用途僅是為了和早期版本的相容。我們推薦您使用其他方法來處理那些不列印的命令。詳細內容參閱*命令回顯*。如果您希望所有的命令都不列印，請使用‘-s’或‘-silent’選項(詳細參閱*選項概要*)。
- ! **.EXPORT_ALL_VARIABLES**
  如該特殊目標簡單的作為一個目標被提及，MAKE 將缺省地把所有變數都傳遞到子進程中。參閱使與子 **MAKE** 通信的變數。
- ! **.NOTPARALLEL**
  如果.NOTPARALLEL 作為一個目標提及，即使給出‘-j’選項，make 也不使用並行執行。但遞迴調用的 make 命令仍可並行執行（在調用的 makefile 檔中包含.NOTPARALLEL 的目標的例外）。.NOTPARALLEL 的任何依賴都將忽略。

任何定義的隱含規則尾碼如果作為目標出現都會視為一個特殊規則，即使兩個尾碼串聯起來也是如此，例如‘.c.o’。這些目標稱為*尾碼規則*，這種定義方法是過時的定義隱含規則的方法（目前仍然廣泛使用的方法）。原則上，如果您要把它分為兩個並把它們加到尾碼列表中，任何目標名都可採用這種方法指定。實際上，尾碼一般以‘.’開始，因此，這些特別的目標同樣以‘.’開始。具體參閱*過時的尾碼規則*。

4.8 具有多個目標的規則

具有多個目標的規則等同於寫多條規則，這些規則除了目標不同之外，其餘部分完全相同。相同的命令應用於所有目標，但命令執行的結果可能有所差異，因此您可以在命令中使用‘$@’分配不同的實際目標名稱。這條規則同樣意味著所有的目標有相同的依賴。

在以下兩種情況下具有多個目標的規則相當有用：

- ！　　您僅僅需要依賴，但不需要任何命令。例如：

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

為三個提及的目標檔給出附加的共同依賴。

- ！　　所有的目標使用相同的命令。但命令的執行結果未必完全相同，因為自動變數‘$@’可以在重建時指定目標（參閱自動變數）。例如：

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

等同於：

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

這裏我們假設程式可以產生兩種輸出檔類型：一種給出‘-big’，另一種給出‘-little’。參閱*字串代替和分析函數*，對函數 subst 的解釋。

如果您喜歡根據目標變換依賴，象使用變數‘$@’變換命令一樣。您不必使用具有多個目標的規則，您可以使用*靜態格式規則*。詳細內容見下文。

4.9 具有多條規則的目標

一個目標檔可以有多個規則。在所有規則中提及的依賴都將融合在一個該目標的依賴列表中。如果該目標比任何一個依賴‘舊’，所有的命令將執行重建該目標。

但如果一條以上的規則對同一檔給出多條命令，make 將使用最後給出的規則，同時列印錯誤資訊。（作為特例，如果檔案名以點‘.’開始，不列印出錯資訊。這種古怪的行為僅僅是為了和其他版本的 make 相容）。您沒有必要這樣編寫您的 makefile 檔，這正是 make 給您發出錯誤資訊的原因。

一條特別的依賴規則可以用來立即給多條目標檔提供一些額外的依賴。例如，使用名為‘objects’的變數，該變數包含系統產生的所有輸出檔列表。如果‘congfig.h’發生變化所有的輸出檔必須重新編譯，可以採用下列簡單的方法編寫：

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

這些可以自由插入或取出而不影響實際指定的目標檔生成規則，如果您希望斷斷續續的為目標添加依賴，這是非常方便的方法。

另外一個添加依賴的方法是定義一個變數，並將該變數作為 make 命令的參數使用。詳細內容參閱*變數重載*。例如：

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

命令`make extradeps=foo.h'含義是將‘foo.h’作為所有 OBJ 檔的依賴，如果僅僅輸入‘make’命令則不是這樣。

如果沒有具體的規則為目標的生成指定命令，那麼 make 將搜尋合適的隱含規則進而確定一些命令來完成生成或重建目標。詳細內容參閱*使用隱含規則*。

4.10 靜態格式規則

靜態格式規則是指定多個目標並能夠根據每個目標名構造對應的依賴名的規則。靜態格式規則在用於多個目標時比平常的規則更常用，因為目標可以不必有完全相同的依賴；也就是說，這些目標的依賴必須類似，但不必完全相同。

4.10.1 靜態格式規則的語法

這裏是靜態格式規則的語法格式：

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

目標列表指明該規則應用的目標。目標可以含有通配符，具體使用和平常的目標規則基本一樣（參閱*在檔案名中使用通配符*）。

目標的格式和依賴的格式是說明如何計算每個目標依賴的方法。從匹配目標格式的目標名中依據格式抽取部分字串，這部分字串稱為徑。將徑分配到每一個依賴格式中產生依賴名。

每一個格式通常包含字元‘%’。目標格式匹配目標時，‘%’可以匹配目標名中的任何字串；這部分匹配的字串稱為徑；剩下的部分必須完全相同。如目標‘foo.o’匹配格式‘%.o’，字串‘foo’稱為徑。而目標‘foo.c’和‘foo.out’不匹配格式。

每個目標的依賴名是使用徑代替各個依賴中的‘%’產生。如，如果一個依賴格式為‘%.c’，把徑‘foo’代替依賴格式中的‘%’生成依賴的檔案名‘foo.c’。在依賴格式中不包含‘%’也是合法的，此時對所有目標來說，依賴是相同的。

在格式規則中字元‘%’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‘%’、‘\’的反斜杠在和檔案名比較或由徑代替它之前從格式中移走。反斜杠不會因為引用‘%’而混亂。如，格式`the\%weird\\%pattern\\'是`the%weird\' 加上字元‘%'，後面再和字串 ‘pattern\\'連接。最後的兩個反斜杠由於不能影響任何統配符‘%’所以保持不變。

這裏有一個例子，它將對應的‘.c’檔編譯成‘foo.o’和‘bar.o’。

```
objects = foo.o bar.o

all: $(objects)

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

這裏‘$<’是自動變數，控制依賴的名稱，‘$@’也是自動變數，掌握目標的名稱。詳細內容參閱*自動變數*。

每一個指定目標必須和目標格式匹配，如果不符則產生警告。如果您有一列檔，僅有其中的一部分和格式匹配，您可以使用 **filter** 函數把不符合的檔移走（參閱*字串替代和分析函數*）：

```
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 $<
```

在這個例子中，‘$(filter %.o,$(files))'的結果是‘bar.o lose.o'，第一個靜態格式規則是將相應的 C 語言原始檔案編譯更新為 OBJ 檔，‘$(filter %.elc,$(files))' 的結果是‘foo.elc'，它由‘foo.el’構造。

另一個例子是闡明怎樣在靜態格式規則中使用‘$*’：

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

當命令 generate 執行時，$*擴展為徑，即‘big’或‘little’二者之一。

4.10.2 靜態格式規則和隱含規則

靜態格式規則和定義為格式規則的隱含規則有很多相同的地方（詳細參閱*定義與重新定義格式規則*）。雙方都有對目標的格式和構造依賴名稱的格式，差異是 make 使用它們的時

機不同。

隱含規則可以應用于任何於它匹配的目標，但它僅僅是在目標沒有具體規則指定命令以及依賴可以被搜尋到的情況下應用。如果有多條隱含規則適合，僅有執行其中一條規則，選擇依據隱含規則的定義次序。

相反，靜態格式規則用於在規則中指明的目標。它不能應用於其他任何目標，並且它的使用方式對於各個目標是固定不變的。如果使用兩個帶有命令的規則發生衝突，則是錯誤。

靜態格式規則因為如下原因可能比隱含規則更好：

- ！ 對一些檔案名不能按句法分類的但可以給出列表的檔，使用靜態格式規則可以重載隱含規則鏈。
- ！ 如果不能精確確定使用的路徑，您不能確定一些無關緊要的檔是否導致 make 使用錯誤的隱含規則（因為隱含規則的選擇根據其定義次序）。使用靜態格式規則則沒有這些不確定因素：每一條規則都精確的用於指定的目標上。

4.11 雙冒號規則

雙冒號規則是在目標名後使用‘：：’代替‘：’的規則。當同一個目標在一條以上的規則中出現時，雙冒號規則和平常的規則處理有所差異。

當一目標在多條規則中出現時，所有的規則必須是同一類型：要麼都是雙冒號規則，要麼都是普通規則。如果他們都是雙冒號規則，則它們之間都是相互獨立的。如果目標比一個雙冒號規則的依賴‘舊’，則該雙冒號規則的命令將執行。這可導致具有同一目標雙冒號規則全部或部分執行。

雙冒號規則實際就是將具有相同目標的多條規則相互分離，每一條雙冒號規則都獨立的運行，就像這些規則的目標不同一樣。

對於一個目標的雙冒號規則按照它們在 makefile 檔中出現的順序執行。然而雙冒號規則真正有意義的場合是雙冒號規則和執行順序無關的場合。

雙冒號規則有點模糊難以理解，它僅僅提供了一種在特定情況下根據引起更新的依賴檔不同，而採用不同方式更新目標的機制。實際應用雙冒號規則的情況非常罕見。

每一個雙冒號規則都應該指定命令，如果沒有指定命令，則會使用隱含規則。詳細內容參閱*使用隱含規則*。

4.12 自動生成依賴

在為一個程式編寫的 makefile 檔中，常常需要寫許多僅僅是說明一些 OBJ 檔依靠頭檔的規則。例如，如果‘main.c’通過一條#include 語句使用‘defs.h’，您需要寫入下的規則：

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

您需要這條規則讓 make 知道如果‘defs.h’一旦改變必須重新構造‘main.o’。由此您可以明白對於一個較大的程式您需要在 makefile 檔中寫很多這樣的規則。而且一旦添加或去掉一條#include 語句您必須十分小心地更改 makefile 檔。

為避免這種煩惱，現代 C 編譯器根據原程式中的#include 語句可以為您編寫這些規則。如果需要使用這種功能，通常可在編譯根源程式時加入‘-M’開關，例如，下面的命令：

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

產生如下輸出：

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

這樣您就不必再親自寫這些規則，編譯器可以為您完成這些工作。

注意，由於在 makefile 檔中提及構造‘main.o’，因此‘main.o’將永遠不會被隱含規則認為是中間檔而進行搜尋，這同時意味著 make 不會在使用它之後自動刪除它；參閱*隱含規則鏈*。

對於舊版的 make 程式，通過一個請求命令，如‘make depend’，利用編譯器的特點生成依賴是傳統的習慣。這些命令將產生一個‘depend’檔，該檔包含所有自動生成的依賴；然後 makefile 檔可以使用 include 命令將它們讀入（參閱*包含其他* ***makefile*** *文件*）。

在 GNU make 中，重新構造 makefile 檔的特點使這個慣例成為了過時的東西——您永

遠不必具體告訴 make 重新生成依賴，因為 GNU make 總是重新構造任何過時的 makefile 檔。參閱 ***Makefile** 檔的重新生成的過程*。

我們推薦使用自動生成依賴的習慣是把 makefile 檔和根源程式檔一一對應起來。如，對每一個根源程式檔‘name.c’有一名為‘name.d’的 makefile 檔和它對應，該 makefile 檔中列出了名為‘name.o’的 OBJ 檔所依賴的檔。這種方式的優點是僅在根源程式檔改變的情況下才有必要重新掃描生成新的依賴。

這裏有一個根據 C 語言根源程式‘name.c’生成名為‘name.d’依賴檔的格式規則：

```
%.d: %.c
        set -e; $(CC) -M $(CPPFLAGS) $< \
                  | sed 's/\($*\)\.o[ :]*/\1.o $@ : /g' > $@; \
                [ -s $@ ] || rm -f $@
```

關於定義格式規則的資訊參閱*定義與重新定義格式規則*。‘-e’開關是告訴 shell 如果$(CC)命令運行失敗（非零狀態退出）立即退出。正常情況下，shell 退出時帶有最後一個命令在管道中的狀態（sed）,因此 make 不能注意到編譯器產生的非零狀態。

對於 GNU C 編譯器您可以使用‘-MM’開關代替‘-M’，這是省略了有關系統頭檔的依賴。詳細內容參閱《GNU CC 使用手冊》中*控制預處理選項*。

命令 Sed 的作用是翻譯（例如）：

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

到：

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

這使每一個‘.d’檔和與之對應的‘.o’檔依靠相同的根源程式檔和頭檔，據此，Make 可以知道如果任一個根源程式檔和頭檔發生變化，則必須重新構造依賴檔。

一旦您定義了重新構造‘.d’檔的規則，您可以使用使用 include 命令直接將它們讀入，（參閱*包含其他 **makefile** 文件*），例如：

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

（這個例子中使用一個代替變數參照從根源程式檔列表‘foo.c bar.c'翻譯到依賴檔列表‘foo.d bar.d'。詳細內容參閱*替換引用*。）所以，‘.d’的 makefile 文件和其他 makefile 文件一樣，即使沒用您的任何進一步的指令，make 同樣會在必要的時候重新構建它們。參閱 ***Makefile** 檔的重新生成過程*。

5 在規則中使用命令

規則中的命令由一系列 shell 命令行組成，它們一條一條的按順序執行。除第一條命令行可以分號為開始附屬在目標-依賴行後面外，所有的命令行必須以 TAB 開始。空白行與注釋行可在命令行中間出現，處理時它們被忽略。（但是必須注意，以 TAB 開始的‘空白行’不是空白行，它是空命令，參閱*使用空命令*。）

用戶使用多種不同的 shell 程式，如果在 makefile 檔中沒有指明其他的 shell，則使用缺省的‘/bin/sh’解釋 makefile 檔中的命令。參閱*命令執行*。

使用的 shell 種類決定了是否能夠在命令行上寫注釋以及編寫注釋使用的語法。當使用‘/bin/sh’作為 shell，以‘#’開始的注釋一直延伸到該行結束。‘#’不必在行首，而且‘#’不是注釋的一部分。

5.1 命令回顯

正常情況下 make 在執行命令之前首先列印命令行，我們因這樣可將您編寫的命令原樣

輸出故稱此為回顯。

以‘@’起始的行不能回顯，‘@’在傳輸給 shell 時被丟棄。典型的情況，您可以在 makefile 檔中使用一個僅僅用於列印某些內容的命令，如 echo 命令來顯示 makefile 檔執行的進程：
@echo About to make distribution files

當使用 make 時給出‘-n’或‘--just-print’標誌，則僅僅回顯命令而不執行命令。參閱*選項概要*。在這種情況下也只有在這種情況下，所有的命令行都回顯，即使以‘@’開始的命令行也回顯。這個標誌對於在不使用命令的情況下發現 make 認為哪些是必要的命令非常有用。

‘-s’或‘--silent’標誌可以使 make 阻止所有命令回顯，好像所有的行都以‘@’開始一樣。在 makefile 檔中使用不帶依賴的特別目標‘.SILENT’的規則可以達到相同的效果（參閱*內建的特殊目標名*）。因為‘@’使用更加靈活以至於現在已基本不再使用特別目標.SILENT。

5.2 執行命令

需要執行命令更新目標時，每一命令行都會使用一個獨立的子 shell 環境，保證該命令行得到執行。（實際上，make 可能走不影響結果的捷徑。）

請注意：這意味著設置局部變數的 **shell** 命令如 **cd** 等將不影響緊跟著的命令行；如果您需要使用 **cd** 命令影響到下一個命令，請把這兩個命令放到一行，它們中間用分號隔開，這樣 **make** 將認為它們是一個單一的命令行，把它們放到一起傳遞給 **shell**，然後按順序執行它們。例如：

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

如果您喜歡將一個單一的命令分割成多個文本行，您必須用反斜杠作為每一行的結束，最後一行除外。這樣，多個文本行通過刪除反斜杠按順序組成一新行，然後將它傳遞給 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 的程式是由變數 SHELL 指定，缺省情況下，使用程式‘/bin/sh’作為 shell。

在 MS_DOS 上運行，如果變數 SHELL 沒有指定，變數 COMSPEC 的值用來代替指定 shell。

在 MS_DOS 上運行和在其他系統上運行，對於 makefile 檔中設置變數 SHELL 的行的處理也不一樣。因為 MS_DOS 的 shell，‘command.com’，功能十分有限，所以許多 make 用戶傾向於安裝一個代替的 shell。因此，在 MS_DOS 上運行，make 檢測變數 SHELL 的值，並根據它指定的 Unix 風格或 DOS 風格的 shell 變化它的行為。即使使用變數 SHELL 指向‘command.com’，make 依然檢測變數 SHELL 的值。

如果變數 SHELL 指定 Unix 風格的 shell，在 MS_DOS 上運行的 make 將附加檢查指定的 shell 是否能真正找到；如果不能找到，則忽略指定的 shell。在 MS_DOS 上，GNU make 按照下述步驟搜尋 shell:

1、在變數 SHELL 指定的目錄中。例如，如果 makefile 指明`SHELL = /bin/sh'，make 將在當前路徑下尋找子目錄‘/bin’。

2、在當前路徑下。

3、按順序搜尋變數 PATH 指定的目錄。

在所有搜尋的目錄中，make 首先尋找指定的檔（如例子中的‘sh’）。如果該檔沒有存在，make 將在上述目錄中搜尋帶有確定的可執行檔擴展的檔。例如：‘.exe', ‘.com', ‘.bat', ‘.btm', ‘.sh' 文件和其他文件等。

如果上述過程中能夠成功搜尋一個 shell，則變數 SHELL 的值將設置為所發現 shell 的全路徑檔案名。然而如果上述努力全部失敗，變數 SHELL 的值將不改變，設置 shell 的行的有效性將被忽略。這是在 make 運行的系統中如果確實安裝了 Unix 風格的 shell，make 僅支持指明的 Unix 風格 shell 特點的原因。

注意這種對 shell 的擴展搜尋僅僅限制在 makefile 檔中設置變數 SHELL 的情況。如果在環境或命令行中設置，希望您指定 shell 的全路徑檔案名，而且全路徑檔案名需象在 Unix 系

統中運行的一樣準確無誤。

經過上述的 DOS 特色的處理，而且您還把 ‘sh.exe’安裝在變數 PATH 指定的目錄中，或在 makefile 檔內部設置`SHELL = /bin/sh'（和多數 Unix 的 makefile 檔一樣），則在 MS_DOS 上的運行效果和在 Unix 上運行完全一樣。

不像其他大多數變數，變數 SHELL 從不根據環境設置。這是因為環境變數 SHELL 是用來指定您自己選擇交互使用的 shell 程式。如果變數 SHELL 在環境中設置，它將影響 makefile 檔的功能，這是非常不划算的，參閱*環境變數*。然而在 MS-DOS 和 MS-WINDOWS 中在環境中設置變數 SHELL 的值是要使用的，因為在這些系統中，絕大多數用戶並不設置該變數的值，所以 make 很可能特意為該變數指定要使用的值。在 MS-DOS 上，如果變數 SHELL 的設置對於 make 不合適，您可以設置變數 MAKESHELL 用來指定 make 使用的 shell；這種設置將使變數 SHELL 的值失效。

5.3 並行執行

GNU make 可以同時執行幾條命令。正常情況下，make 一次執行一個命令，待它完成後在執行下一條命令。然而，使用‘-j’和‘--jobs’選項將告訴 make 同時執行多條命令。在 MS-DOS 上，‘-j’選項沒有作用，因為該系統不支援多進程處理。

如果‘-j’選項後面跟一個整數，該整數表示一次執行的命令的條數；這稱為 job slots 數。如果‘-j’選項後面沒有整數，也就是沒有對 job slots 的數目限制。缺省的 job slots 數是一，這意味著按順序執行（一次執行一條命令）。同時執行多條命令的一個不太理想的結果是每條命令產生的輸出與每條命令發送的時間對應，即命令產生的消息回顯可能較為混亂。

另一個問題是兩個進程不能使用同一設備輸入，所以必須確定一次只能有一條命令從終端輸入，make 只能保證正在運行的命令的標準輸入流有效，其他的標準輸入流將失效。這意味著如果有幾個同時從標準輸入設備輸入的話，對於絕大多數子進程將產生致命的錯誤（即產生一個‘Broken pipe’信號）。

命令對一個有效的標準輸入流（它從終端輸入或您為 make 改造的標準輸入設備輸入）的需求是不可預測的。第一條運行的命令總是第一個得到標準輸入流，在完成一條命令後第一條啟動的另一條命令將得到下一個標準輸入流，等等。

如果我們找到一個更好替換方案，我們將改變 make 的這種工作方式。在此期間，如果您使用並行處理的特點，您不應該使用任何需要標準輸入的命令。如果您不使用該特點，任何需要標準輸入的命令將都能正常工作。

最後，make 的遞迴調用也導致出現問題。更詳細的內容參閱*與子* ***make*** *通訊的選項*。

如果一個命令失敗（被一個信號中止，或非零退出），且該條命令產生的錯誤不能忽略（參閱*命令錯誤*），剩餘的構建同一目標的命令行將會停止工作。如果一條命令失敗，而且‘-k’或‘--keep-going’選項也沒有給出（參閱*選項概要*），make 將放棄繼續執行。如果 make 由於某種原因（包括信號）要中止，此時又子進程正在運行，它將等到這些子進程結束之後再實際退出。

當系統正滿負荷運行時，您或許希望在負荷輕的時再添加任務。這時，您可以使用‘-l’選項告訴 make 根據平均負荷限制同一時刻運行的任務數量。‘-l’或‘--max-load’選項一般後跟一個浮點數。例如：

```
-l 2.5
```

將不允許 make 在平均負荷高於 2.5 時啟動一項任務。‘-l’選項如果沒有跟資料，則取消前面‘-l’給定的負荷限制。

更精確地講，當 make 啟動一項任務時，而它此時已經有至少一項任務正在運行，則它將檢查當前的平均負荷；如果不低於‘-l’選項給定的負荷限制時，make 將等待直到平均負荷低於限制或所有其他任務完成後再啟動其他任務。

缺省情況下沒有負荷限制。

5.4 命令錯誤

在每一個 shell 命令返回後，make 檢查該命令退出的狀態。如果該命令成功地完成，下一個命令行就會在新的子 shell 環境中執行，當最後一個命令行完成後，這條規則也宣告完成。如果出現錯誤（非零退出狀態），make 將放棄當前的規則，也許是所有的規則。

有時一個特定的命令失敗並不是出現了問題。例如：您可能使用 mkdir 命令創建一個目錄存在，如果該目錄已經存在，mkdir 將報告錯誤，但您此時也許要 make 繼續執行。

要忽略一個命令執行產生的錯誤，請使用字元'-'（在初始化 TAB 的後面）作為該命令行的開始。字元'-'在命令傳遞給 shell 執行時丟棄。例如：

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

這條命令即使在不能刪除一個檔時也強制 rm 繼續執行。

在運行 make 時使用'-i'或'--ignore-errors'選項，將會忽略所有規則的命令運行產生的錯誤。在 makefile 檔中使用如果沒有依賴的特殊目標.IGNORE 規則，也具有同樣的效果。但因為使用字元'-'更靈活，所以該條規則已經很少使用。

一旦使用'-'或'-i'選項，運行命令時產生的錯誤被忽略，此時 make 象處理成功運行的命令一樣處理具有返回錯誤的命令，唯一不同的地方是列印一條消息，告訴您命令退出時的編碼狀態，並說明該錯誤已經被忽略。如果發生錯誤而 make 並不說明其被忽略，則暗示當前的目標不能成功重新構造，並且和它直接相關或間接相關的目標同樣不能重建。因為前一個過程沒有完成，所以不會進一步執行別的命令。

在上述情況下，make 一般立即放棄任務，返回一個非零的狀態。然而，如果指定'-k'或'--keep-goning'選項，make 則繼續考慮這個目標的其他依賴，如果有必要在 make 放棄返回非零狀態之前重建它們。例如，在編譯一個 OBJ 檔發生錯誤後，即使 make 已經知道將所有 OBJ 檔連接在一起是不可能的，'make -k'選項也繼續編譯其他 OBJ 文件。詳細內容參閱*選項概要*。通常情況下，make 的行為基於假設您的目的是更新指定的目標，一旦 make 得知這是不可能的，它將立即報告失敗。'-k'選項是告訴 make 真正的目的是測試程式中所有變化的可行性，或許是尋找幾個獨立的問題以便您可以在下次編譯之前糾正它們。這是 Emacs 編譯命令缺省情況下傳遞'-k'選項的原因。

通常情況下，當一個命令運行失敗時，如果它已經改變了目標檔，則該檔很可能發生混亂而不能使用或該檔至少沒有完全得到更新。但是，檔的時間戳卻表明該檔已經更新到最新，因此在 make 下次運行時，它將不再更新該檔。這種狀況和命令被發出的信號強行關閉一樣，參閱*中斷或關閉* ***make***。因此，如果在開始改變目標檔後命令出錯，一般應該刪除目標檔。如果.DELETE_ON_ERROR 作為目標在 makefile 檔中出現，make 將自動做這些事情。這是您應該明確要求 make 執行的動作，不是以前的慣例；特別考慮到相容性問題時，您更應明確提出這樣的要求。

5.5 中斷或關閉 **make**

如果 make 在一條命令運行時得到一個致命的信號， 則 make 將根據第一次檢查的時間戳和最後更改的時間戳是否發生變化決定它是否刪除該命令要更新的目標檔。

刪除目標檔的目的是當 make 下次運行時確保目標檔從原文件得到更新。為什麼？假設正在編譯檔時您鍵入 Ctrl-c，而且這時已經開始寫 OBJ 檔'foo.o'，Ctrl-c 關閉了該編譯器，結果得到不完整的 OBJ 檔'foo.o'的時間戳比根源程式'foo.c'的時間戳新，如果 make 收到 Ctrl-c 的信號而沒有刪除 OBJ 檔'foo.o'，下次請求 make 更新 OBJ 檔'foo.o'時，make 將認為該檔已更新到最新而沒有必要更新，結果在 linker 將 OBJ 檔連接為可執行檔時產生奇怪的錯誤資訊。

您可以將目標檔作為特殊目標.PRECIOUS 的依賴從而阻止 make 這樣刪除該目標檔。在重建一個目標之前，make 首先檢查該目標檔是否出現在特殊目標.PRECIOUS 的依賴列表中，從而決定在信號發生時是否刪除該目標檔。您不刪除這種目標檔的原因可能是：目標更新是一種原子風格，或目標檔存在僅僅為了記錄更改時間（其內容無關緊要），或目標檔必須一直存在，用來防止其他類型的錯誤等。

5.6 遞迴調用 **make**

遞迴調用意味著可以在 makefile 檔中將 make 作為一個命令使用。這種技術在包含大的系統中把 makefile 分離為各種各樣的子系統時非常有用。例如，假設您有一個子目錄‘subdir’，該目錄中有它自己的 makefile 檔，您希望在該子目錄中運行 make 時使用該 makefile 檔，則您可以按下述方式編寫：

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

或，等同於這樣寫 (參閱*選項概要*):

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

您可以僅僅拷貝上述例子實現 make 的遞迴調用，但您應該瞭解它們是如何工作的，它們為什麼這樣工作，以及子 make 和上層 make 的相互關係。

為了使用方便，GNU make 把變數 CURDIR 的值設置為當前工作的路徑。如果‘-C’選項有效，它將包含的是新路徑，而不是原來的路徑。該值和它在 makefile 中設置的值有相同的優先權（缺省情況下，環境變數 CURDIR 不能重載）。注意，操作 make 時設置該值無效。

5.6.1 變數 **MAKE** 的工作方式

遞迴調用 make 的命令總是使用變數 MAKE，而不是明確的命令名‘make’，如下所示：

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

該變數的值是調用 make 的檔案名。如果這個檔案名是‘/bin/make’，則執行的命令是`cd subdir && /bin/make'。如果您在上層 makefile 檔時用特定版本的 make，則執行遞迴調用時也使用相同的版本。

在命令行中使用變數 MAKE 可以改變‘-t' (‘--touch'), ‘-n' (‘--just-print'), 或 ‘-q' (‘--question')選項的效果。如果在使用變數 MAKE 的命令行首使用字元‘+’也會起到相同的作用。參閱*代替執行命令*。

設想一下在上述例子中命令‘make -t’的執行過程。(‘-t’選項標誌目標已經更新，但卻不執行任何命令，參閱*代替執行命令*。）按照通常的定義，命令‘make –t’在上例中僅僅創建名為‘subsystem’的檔而不進行別的工作。您實際要求運行‘cd subdir && make –t’幹什麼？是執行命令或是按照‘-t’的要求不執行命令？

Make 的這個特點是這樣的：只要命令行中包含變數 MAKE，標誌`-t', `-n' 和 `-q'將不對本行起作用。雖然存在標誌不讓命令執行，但包含變數 MAKE 的命令行卻正常運行，make 實際上是通過變數 MAKEFLAGS 將標誌值傳遞給了子 make(參閱*與子 **make** 通訊的選項*)。所以您的驗證檔、列印命令的請求等都能傳遞給子系統。

5.6.2 與子 **make** 通訊的變數

通過明確要求，上層 make 變數的值可以借助環境傳遞給子 make，這些變數能在子 make 中缺省定義，在您不使用‘-e’開關的情況下，傳遞的變數的值不能代替子 make 使用的 makefile 檔中指定的值（參閱*命令概要*）。

向下傳遞、或輸出一個變數時，make 將該變數以及它的值添加到運行每一條命令的環境中。子 make，作為回應，使用該環境初始化它的變數值表。參閱*環境變數*。

除了明確指定外，make 僅向下輸出在環境中定義並初始化的或在命令行中設置的變數，而且這些變數的變數名必須僅由字母、數位和下劃線組成。一些 shell 不能處理名字中含有字母、數位和下劃線以外字元的環境變數。特殊變數如 SHELL 和 MAKEFLAGS 一般

總要向下輸出（除非您不輸出它們）。即使您把變數 MAKEFILE 設為其他的值，它也向下輸出。

Make 自動傳遞在命令行中定義的變數的值，其方法是將它們放入 MAKEFLAGS 變數中。詳細內容參閱下節。Make 缺省創造的變數的值不能向下傳遞，子 make 可以自己定義它們。如果您要將指定變數輸出給子 make，請用 export 指令，格式如下：

export *variable* ...

您要將阻止一些變數輸出給子 make，請用 unexport 指令，格式如下：

unexport *variable* ...

為方便起見，您可以同時定義並輸出一個變數：

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*

同樣,

export *variable* += value

亦同樣:

variable += value
export *variable*
參閱*為變數值追加文本*。

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

如果您要將所有的變數都輸出，您可以單獨使用 export：

export

這告訴 make 把 export 和 unexport 沒有提及的變數統統輸出，但任何在 unexport 提及的變數仍然不能輸出。如果您單獨使用 export 作為缺省的輸出變數方式，名字中含有字母、數位和下劃線以外字元的變數將不能輸出，這些變數除非您明確使用 export 指令提及才能輸出。

單獨使用 export 的行為是老闆本 GNU make 缺省定義的行為。如果您的 makefile 依靠這些行為，而且您希望和老闆本 GNU make 相容，您可以為特殊目標.EXPORT_ALL_VARIABLES 編寫一條規則代替 export 指令，它將被老闆本 GNU make 忽略，但如果同時使用 export 指令則報錯。

同樣，您可以單獨使用 unexport 告訴 make 缺省不要輸出變數，因為這是缺省的行為，只有前面單獨使用了 export（也許在一個包括的 makefile 中）您才有必要這樣做。您不能同時單獨使用 export 和 unexport 指令實現對某些命令輸出對其他的命令不輸出。最後面的一條指令（export 或 unexport）將決定 make 的全部運行結果。

作為一個特點，變數 MAKELEVEL 的值在從一個層次向下層傳遞時發生變化。該變數的值是字元型，它用十進位數字表示層的深度。‘0’代表頂層 make，‘1’代表子 make，‘2’代表子-子-make，以此類推。Make 為一個命令建立一次環境，該值增加 1。

該變數的主要作用是在一個條件指令中測試（參閱 ***makefile** 檔的條件語句*）；採用這種方法，您可以編寫一個 makefile，如果遞迴調用採用一種運行方式，由您控制直接執行採用

另一種運行方式。

您可以使用變數 MAKEFILES 使所有的子 make 使用附加的 makefile 檔。變數 MAKEFILES 的值是 makefile 檔案名的列表，檔案名之間用空格隔開。在外層 makefile 中定義該變數，該變數的值將通過環境向下傳遞；因此它可以作為子 make 的額外的 makefile 檔，在子 make 讀正常的或指定的 makefile 檔前，將它們讀入。參閱*變數* ***MAKEFILES***。

5.6.3 與子 **make** 通訊的選項

諸如‘-s’和‘-k’標誌通過變數 MAKEFLAGS 自動傳遞給子 make。該變數由 make 自動建立，並包含 make 收到的標誌字母。所以，如果您是用‘make –ks’變數 MAKEFLAGS 就得到值‘ks’。

作為結果，任一個子 make 都在它的運行環境中為變數 MAKEFLAGS 賦值；作為回應，make 使用該值作為標誌並進行處理，就像它們作為參數被給出一樣。參閱*選項概要*。

同樣，在命令行中定義的變數也將借助變數 MAKEFLAGS 傳遞給子 make。變數 MAKEFLAGS 值中的字可以包含‘=’，make 將它們按變數定義處理，其過程和在命令行中定義的變數一樣。參閱*變數重載*。

選項`-C', `-f', `-o', 和 ‘-W’不能放入變數 MAKEFLAGS 中；這些選項不能向下傳遞。

‘-j’選項是一個特殊的例子（參閱*並行執行*）。如果您將它設置為一些數值‘N’，而且您的作業系統支援它（大多數 Unix 系統支援，其他作業系統不支援），父 make 和所有子 make 通訊保證在它們中間同時僅有‘N’個任務運行。注意，任何包含遞迴調用的任務（參閱*代替執行命令*）不能計算在總任務數內（否則，我們僅能得到‘N’個子 make 運行，而沒有多餘的時間片運行實在的工作）。

如果您的作業系統不支援上述通訊機制，那麼‘-j 1’將放到變數 MAKEFLAGS 中代替您指定的值。這是因為如果‘-j’選項傳遞給子 make，您可能得到比您要求多很多的並行運行的任務數。如果您給出‘-j’選項而沒有數位參數，意味著盡可能並行處理多個任務，這樣向下傳遞，因為倍數的無限制性所以至多為 1。

如果您不希望其他的標誌向下傳遞，您必須改變變數 MAKEFLAGS 的值，其改變方式如下：

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

該命令行中定義變數的實際上出現在變數 MAKEOVERRIDES 中，而且變數 MAKEFLAGS 包含了該變數的引用值。如果您要向下傳遞標誌，而不向下傳遞命令行中定義的變數，這時，您可以將變數 MAKEOVERRIDES 的值設為空，格式如下：

```
MAKEOVERRIDES =
```

這並不十分有用。但是，一些系統對環境的大小有固定限制，而且該值較小，將這麼多的資訊放到變數 MAKEFLAGS 的值中可能超過該限制。如果您看到‘Arg list too long'的錯誤資訊，很可能就是由於該問題造成的。（按照嚴格的 POSIX.2 的規定，如果在 makefile 檔定義特殊目標‘.POSIX’，改變變數 MAKEOVERRIDES 的值並不影響變數 MAKEFLAGS。也許您並不關心這些。）

為了和早期版本相容，具有相同功能的變數 MFLAGS 也是存在的。除了它不能包含命令行定義變數外，它和變數 MAKEFLAGS 有相同的值，而且除非它是空值，它的值總是以短線開始（MAKEFLAGS 只有在和多字元選項一起使用時才以短線開始，如和‘--warn-undefined-variables’連用）。變數 MFLAGS 傳統的使用在明確的遞迴調用 make 的命令中，例如：

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

但現在，變數 MAKEFLAGS 使這種用法變得多餘。如果您要您的 makefile 檔和老版本的 make 程式相容，請使用這種方式；這種方式在現代版本 make 中也能很好的工作。

如果您要使用每次運行 make 都要設置的特定選項，例如‘-k’選項（參閱*選項概要*），變數 MAKEFLAGS 十分有用。您可以簡單的在環境中將給變數 MAKEFLAGS 賦值，或在

makefile 檔中設置變數 MAKEFLAGS，指定的附加標誌可以對整個 makefile 檔都起作用。（注意：您不能以這種方式使用變數 MFLAGS，變數 MFLAGS 存在僅為和早期版本相容，採用其他方式設置該變數 make 將不予解釋。）

當 make 解釋變數 MAKEFLAGS 值的時候（不管在環境中定義或在 makefile 檔中定義），如果該值不以短線開始，則 make 首先為該值假設一個短線；接著將該值分割成字，字與字間用空格隔開，然後將這些字進行語法分析，好像它們是在命令行中給出的選項一樣。（'-C', '-f','-h','-o','-W'選項以及它們的長名字版本都將忽略，對於無效的選項不產生錯誤資訊。）

如果您在環境中定義變數 MAKEFLAGS，您不要使用嚴重影響 make 運行，破壞 makefile 檔的意圖以及 make 自身的選項。例如'-t', '-n', 和'-q'選項，如果將它們中的一個放到變數 MAKEFLAGS 的值中，可能產生災難性的後果，或至少產生讓人討厭的結果。

5.6.4 '--print-directory'選項

如果您使用幾層 make 遞迴調用，使用'-w'或'--print-directory'選項，通過顯示每個 make 開始處理以及處理完成的目錄使您得到比較容易理解的輸出。例如，如果使用'make –w'命令在目錄'/u/gnu/make'中運行 make，則 make 將下面格式輸出資訊：

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

說明進入目錄中，還沒有進行任何任務。下面的資訊：

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

說明任務已經完成。

通常情況下，您不必具體指明這個選項，因為 make 已經為您做了：當您使用'-C'選項時，'-w'選項已經自動打開，在子 make 中也是如此。如果您使用'-s'選項，'-w'選項不會自動打開，因為'-s'選項是不列印資訊，同樣使用`--no-print-directory'選項'-w'選項也不會自動打開。

5.7 定義固定次序命令

在創建各種目標時，相同次序的命令十分有用時，您可以使用 define 指令定義固定次序的命令，並根據這些目標的規則引用固定次序。固定次序實際是一個變數，因此它的名字不能和其他的變數名衝突。

下面是定義固定次序命令的例子：

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

run-yacc 是定義的變數的名字；endef 標誌定義結束；中間的行是命令。define 指令在固定次序中不對變數引用和函數調用擴展；字元'$'、圓括號、變數名等等都變成您定義的變數的值的一部分。*定義多行變數*一節對指令 define 有詳細解釋。

在該例子中，對於任何使用該固定次序的規則，第一個命令是對其第一個依賴運行 Yacc 命令，Yacc 命令執行產生的輸出檔一律命名為'y.tab.c'；第二條命令，是將該輸出檔的內容移入規則的目標檔中。

在使用固定次序時，規則中命令使用的變數應被確定的值替代，您可以象替代其他變數一樣替代這些變數（詳細內容參閱*變數引用基礎*）。因為由 define 指令定義的變數是遞迴擴展的變數，所以在使用時所有變數引用才擴展。例如：

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

當固定次序'run-yacc'運行時，'foo.y'將代替變數'$^'，'foo.c'將代替變數'$@'。

這是一個現實的例子，但並不是必要的，因為 make 有一條隱含規則可以根據涉及的檔案名的類型確定所用的命令。參閱*使用隱含規則*。

在命令執行時，固定次序中的每一行被處理為和直接出現在規則中的命令行一樣，前面加上一個 Tab，make 也特別為每一行請求一個獨立的子 shell。您也可以在固定次序的每一

行上使用影響命令行的首碼字元(`@', `-',和 `+')，參閱*在規則中使用命令*。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 將不回顯第一行，但要回顯後面的兩個命令行。

另一方面，如果首碼字元在引用固定次序的命令行中使用，則該首碼字元將應用到固定次序的每以行中。例如這個規則：

```
frob.out: frob.in
        @$(frobnicate)
```

將不回顯固定次序的任何命令。具體內容參閱*命令回顯*。

5.8 使用空命令

定義什麼也不幹的命令有時很有用，定義空命令可以簡單的給出一個僅僅含有空格而不含其他任何東西的命令即可。例如：

```
target: ;
```

為字串‘target’定義了一個空命令。您也可以使用以 Tab 字元開始的命令行定義一個空命令，但這由於看起來空白容易造成混亂。

也許您感到奇怪，為什麼我們定義一個空命令？唯一的原因是為了阻止目標更新時使用隱含規則提供的命令。（參閱*使用隱含規則*以及*定義最新類型的缺省規則*）

也許您喜愛為實際不存在的目標檔定義空命令，因為這樣它的依賴可以重建。然而這樣做並不是一個好方法，因為如果目標檔實際存在，則依賴有可能不重建，使用假想目標是較好的選擇，參閱*假想目標*。

6 使用變數

變數是在 makefile 中定義的名字，其用來代替一個文本字串，該文本字串稱為該變數的值。在具體要求下，這些值可以代替目標、依賴、命令以及 makefile 檔中其他部分。（在其他版本的 make 中，變數稱為巨集（macros）。）

在 makefile 檔讀入時，除規則中的 shell 命令、使用‘=’定義的‘=’右邊的變數、以及使用 define 指令定義的變數體此時不擴展外，makefile 檔其他各個部分的變數和函數都將擴展。

變數可以代替檔列表、傳遞給編譯器的選項、要執行的程式、查找原始檔案的目錄、輸出寫入的目錄，或您可以想像的任何文本。

變數名是不包括‘:’,‘#’,‘=’、前導或結尾空格的任何字串。然而變數名包含字母、數位以及下劃線以外的其他字元的情況應儘量避免，因為它們可能在將來被賦予特別的含義，而且對於一些 shell 它們也不能通過環境傳遞給子 make（參閱*與子* ***make*** *通訊的變數*）。變數名是大小寫敏感的，例如變數名‘foo’, ‘FOO’, 和 ‘Foo’代表不同的變數。

使用大寫字母作為變數名是以前的習慣，但我們推薦在 makefile 內部使用小寫字母作為變數名，預留大寫字母作為控制隱含規則參數或用戶重載命令選項參數的變數名。參閱*變數重載*。

一少部分的變數使用一個標點符號或幾個字元作為變數名，這些變數是自動變數，它們又特定的用途。參閱*自動變數*。

6.1 變數引用基礎

寫一個美元符號後跟用圓括號或大括弧括住變數名則可引用變數的值：‘$(foo)' 和‘${foo}'都是對變數‘foo’的有效引用。‘$’的這種特殊作用是您在命令或檔案名中必須寫‘$$’才有單個‘$’的效果的原因。

變數的引用可以用在上下文的任何地方：目標、依賴、命令、絕大多數指令以及新變數的值等等。這裏有一個常見的例子，在程式中，變數保存著所有 OBJ 檔的檔案名：

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

$(objects) : defs.h
```

變數的引用按照嚴格的文本替換進行，這樣該規則

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

可以用於編譯 C 語言根源程式‘prog.c’。因為在變數分配時，變數值前面的空格被忽略，所以變數 foo 的值是‘C’。（不要在您的 makefile 檔這樣寫！）

美元符號後面跟一個字元但不是美元符號、圓括號、大括弧，則該字元將被處理為單字元的變數名。因此可以使用‘$x’引用變數 x。然而，這除了在使用自動變數的情況下，在其他實際工作中應該完全避免。參閱*自動變數*。

6.2 變數的兩個特色

在 GNU make 中可以使用兩種方式為變數賦值，我們將這兩種方式稱為變數的兩個特色（two flavors）。兩個特色的區別在於它們的定義方式和擴展時的方式不同。

變數的第一個特色是遞迴調用擴展型變數。這種類型的變數定義方式：在命令行中使用‘=’定義（參閱*設置變數*）或使用 define 指令定義（參閱*定義多行變數*）。變數替換對於您所指定的值是逐字進行替換的；如果它包含對其他變數的引用，這些引用在該變數替換時（或在擴展為其他字串的過程中）才被擴展。這種擴展方式稱為遞迴調用型擴展。例如：

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

all:;echo $(foo)
```

將回顯‘Huh?'：‘$(foo)’擴展為‘$(bar)’，進一步擴展為‘$(ugh)’，最終擴展為‘Huh?’。

這種特色的變數是其他版本 make 支援的變數類型，有缺點也有優點。大多數人認為的該類型的變數的優點是：

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

即能夠完成希望它完成的任務：當‘CFLAGS’在命令中擴展時，它將最終擴展為‘-Ifoo -Ibar’。其最大的缺點是不能在變數後追加內容，如在：

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

在變數擴展過程中可能導致無窮迴圈（實際上 make 偵測到無窮迴圈就會產生錯誤資訊）。

它的另一個缺點是在定義中引用的任何函數時（參閱*文本轉換函數*）變數一旦展開函數就會立即執行。這可導致 make 運行變慢，性能變壞；並且導致通配符與 shell 函數（因不能控制何時調用或調用多少次）產生不可預測的結果。

為避免該問題和遞迴調用擴展型變數的不方便性，出現了另一個特色變數：簡單擴展型變數。

簡單擴展型變數在命令行中用‘:=’定義（參閱*設置變數*）。簡單擴展型變數的值是一次掃描永遠使用，對於引用的其他變數和函數在定義的時候就已經展開。簡單擴展型變數的值實際就是您寫的文本擴展的結果。因此它不包含任何對其他變數的引用；在該變數定義時就包含了它們的值。所以：

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

```
x := later
```

等同於：

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

引用一個簡單擴展型變數時，它的值也是逐字替換的。這裏有一個稍複雜的例子，說明了‘:=’和 shell 函數連接用法（參閱*函數* ***shell***）。該例子也表明了變數 MAKELEVEL 的用法，該變數在層與層之間傳遞時值發生變化。（參閱*與子* ***make*** *通訊的變數*，可獲得變數 MAKELEVEL 關於的資訊。）

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

按照這種方法使用‘:=’的優點是看起來象下述的典型的‘下降到目錄’的命令：

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

簡單擴展型變數因為在絕大多數程式設計語言中可以象變數一樣工作，因此它能夠使複雜的 makefile 程式更具有預測性。它們允許您使用它自己的值重新定義（或它的值可以被一個擴展函數以某些方式處理），它們還允許您使用更有效的擴展函數（參閱*文本轉換函數*）。

您可以使用簡單擴展型變數將控制的前導空格引入到變數的值中。前導空格字元一般在變數引用和函數調用時被丟棄。簡單擴展型變數的這個特點意味著您可以在一個變數的值中包含前導空格，並在變數引用時保護它們。象這樣：

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

這裏變數 space 的值就是一個空格，注釋‘# end of the line’包括在這裏為了讓人更易理解。因為尾部的空格不能從變數值中分離出去，僅在結尾留一個空格也有同樣的效果（但是此時相當難讀），如果您在變數值後留一個空格，象這樣在行的結尾寫上注釋清楚表明您的打算是很不錯的主意。相反，如果您在變數值後不要空格，您千萬記住不要在行的後面留下幾個空格再隨意放入注釋。例如：

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

這裏變數 dir 的值是‘/foo/bar ’（四個尾部空格），這不是預期的結果。（假設‘/foo/bar’是預期的值）。

另一個給變數賦值的操作符是‘?=’，它稱為條件變數賦值操作符，因為它僅僅在變數還沒有定義的情況下有效。這聲明：

```
FOO ?= bar
```

和下面的語句嚴格等同（參閱*函數* ***origin***）

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

注意，一個變數即使是空值，它仍然已被定義，所以使用‘?=’定義無效。

6.3 變數引用高級技術

本節內容介紹變數引用的高級技術。

6.3.1 替換引用

替換引用是用您指定的變數替換一個變數的值。它的形式‘$(*var*:*a*=*b*)’（或‘${*var*:*a*=*b*}’），它的含義是把變數 var 的值中的每一個字結尾的 a 用 b 替換。

我們說‘在一個字的結尾’，我們的意思是 a 一定在一個字的結尾出現，且 a 的後面要麼是空格要麼是該變數值的結束，這時的 a 被替換，值中其他地方的 a 不被替換。例如：

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

將變數‘bar’的值設為‘a.c b.c c.c’。參閱*變數設置*。

替換引用實際是使用擴展函數 patsubst 的簡寫形式（參閱*字串替換和分析函數*）。我們提供替換引用也是使擴展函數 patsubst 與 make 的其他實現手段相容的措施。

另一種替換引用是使用強大的擴展函數 patsubst。它的形式和上述的‘$(*var*:*a*=*b*)’一樣，不同在於它必須包含單個‘%’字元，其實這種形式等同於‘$(patsubst *a*,*b*,$(*var*))’。有關於函數 patsubst 擴展的描述參閱*字串替換和分析函數*。例如：

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

社值變數‘bar'的值為‘a.c b.c c.c'。

6.3.2 嵌套變數引用（計算的變數名）

嵌套變數引用（計算的變數名）是一個複雜的概念，僅僅在十分複雜的 makefile 程式中使用。絕大多數情況您不必考慮它們，僅僅知道創建名字中含有美元標誌的變數可能有奇特的結果就足夠了。然而，如果您是要把一切搞明白的人或您實在對它們如何工作有興趣，請認真閱讀以下內容。

變數可以在它的名字中引用其他變數，這稱為嵌套變數引用（計算的變數名）。例如：

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

定義阿 a 為‘z’：‘$(x)’在‘$($(x))’中擴展為‘y’，因此‘$($(x))’擴展為‘$(y)’，最終擴展為‘z’。這裏對引用的變數名的陳述不太明確；它根據‘$(x)’的擴展進行計算，所以引用‘$(x)’是嵌套在外層變數引用中的。

前一個例子表明了兩層嵌套，但是任何層次數目的嵌套都是允許的，例如，這裏有一個三層嵌套的例子：

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

這裏最裏面的‘$(x)’ 擴展為‘y’，因此‘$($(x))’擴展為‘$(y)’，‘$(y)’ 擴展為‘z’，最終擴展為‘u’。

在一個變數名中引用遞迴調用擴展型變數，則按通常的風格再擴展。例如：

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

定義的 a 是‘Hello’：‘$($(x))’擴展為‘$($(y))’，‘$($(y))’變為‘$(z)’, $(z)’最終擴展為‘Hello’。

嵌套變數引用和其他引用一樣也可以包含修改引用和函數調用（參閱*文本轉換函數*）。例如，使用函數 subst（參閱*字串替換和分析函數*）：

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

```
z = y
a := $($($(z)))
```
定義的 a 是‘Hello’。任何人也不會寫象這樣令人費解的嵌套引用程式，但它確實可以工作：‘$($($(z)))’ 擴展為‘$($(y))’，‘$($(y))’變為‘$(subst 1,2,$(x))’。它從變數‘x’得到值‘variable1’, 變換替換為‘variable2’，所以整個字串變為‘$(variable2)’，一個簡單的變數引用，它的值為‘Hello’。

嵌套變數引用不都是簡單的變數引用，它可以包含好幾個變數引用，同樣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))
```
根據設置的 use_a 和 use_dirs 的輸入可以將 dirs 這個相同的值分別賦給 a_dirs, 1_dirs, a_files 或 1_files。

嵌套變數引用也可以用於替換引用：

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

sources := $($(a1)_objects:.o=.c)
```
根據 a1 的值，定義的 sources 可以是`a.c b.c c.c' 或 `1.c 2.c 3.c'。

使用嵌套變數引用唯一的限制是它們不能只部分指定要調用的函數名，這是因為用於識別函數名的測試在嵌套變數引用擴展之前完成。例如：

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

bar := a d b g q c

foo := $($(func) $(bar))
```
則給變數‘foo’的值賦為‘sort a d b g q c' 或 ‘strip a d b g q c'，而不是將‘a d b g q c’作為函數 sort 或 strip 的參數。如果在將來去掉這種限制是一個不錯的主意。

您也可以變數賦值的左邊使用嵌套變數引用，或在 define 指令中。如：

```
dir = foo
$(dir)_sources := $(wildcard $(dir)/*.c)
define $(dir)_print
lpr $($(dir)_sources)
endef
```
該例子定義了變數‘dir’,‘foo_sources', 和‘foo_print'。

注意：雖然嵌套變數引用和遞迴調用擴展型變數都是用在複雜的 makefile 檔中，但二者不同（參閱*變數的兩個特色*）。

6.4 變數取值

變數有以下幾種方式取得它們的值：

- ! 您可以在運行 make 時為變數指定一個重載值。參閱*變數重載*。
- ! 您可以在 makefile 檔中指定值，即變數賦值（參閱*設置變數*）或使用逐字定義變數（參閱*定義多行變數*）。
- ! 把環境變數變為 make 的變數。參閱*環境變數*。
- ! 自動變數可根據規則提供值，它們都有簡單的習慣用法，參閱*自動變數*。
- ! 變數可以用常量初始化。參閱*隱含規則使用的變數*。

6.5 設置變數

在 makefile 檔中設置變數，編寫以變數名開始後跟‘=’或‘:=’的一行即可。任何跟在‘=’或‘:=’後面的內容就變為變數的值。例如：

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

定義一個名為 objects 的變數，變數名前後的空格和緊跟‘=’的空格將被忽略。

使用‘=’定義的變數是遞迴調用擴展型變數；以‘:=’定義的變數是簡單擴展型變數。簡單擴展型變數定義可以包含變數引用，而且變數引用在定義的同時就被立即擴展。參閱*變數的兩種特色*。

變數名中也可以包含變數引用和函數調用，它們在該行讀入時擴展，這樣可以計算出能夠實際使用的變數名。

變數值的長度沒有限制，但受限於電腦中的實際交換空間。當定義一個長變數時，在合適的地方插入反斜杠，把變數值分為多個文本行是不錯的選擇。這不影響 make 的功能，但可使 makefile 檔更加易讀。

絕大多數變數如果您不為它設置值，空字串將自動作為它的初值。雖然一些變數有內建的非空的初始化值，但您可隨時按照通常的方式為它們賦值（參閱*隱含規則使用的變數*。）另外一些變數可根據規則自動設定新值，它們被稱為自動變數。參閱*自動變數*。

如果您喜歡僅對沒有定義過的變數賦給值，您可以使用速記符‘?=’代替‘=’。下面兩種設置變數的方式完全等同（參閱*函數* ***origin***）：

FOO ?= bar

和

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

6.6 為變數值追加文本

為已經定以過的變數的值追加更多的文本一般比較有用。您可以在獨立行中使用‘+=’來實現上述設想。如：

objects += another.o

這為變數 objects 的值添加了文本‘another.o’（其前面有一個前導空格）。這樣：

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

變數 objects 設置為‘main.o foo.o bar.o utils.o another.o'。

使用 `+=' 相同於:

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

對於使用複雜的變數值，不同方法的差別非常重要。如變數在以前沒有定義過，則‘+=’的作用和‘=’相同：它定義一個遞迴調用型變數。然而如果在以前有定義，‘+=’的作用依賴于您原始定義的變數的特色，詳細內容參閱*變數的兩種特色*。

當您使用‘+=’為變數值附加文本時，make 的作用就好象您在初始定義變數時就包含了您要追加的文本。如果開始您使用‘:=’定義一個簡單擴展型變數，再用‘+=’對該簡單擴展型變數值追加文本，則該變數按新的文本值擴展，好像在原始定義時就將追加文本定義上一樣，詳細內容參閱*設置變數*。實際上，
variable := value
variable += more

等同於：

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

另一方面，當您把‘+=’和首次使用無符號‘=’定義的遞迴調用型變數一起使用時，make 的運行方式會有所差異。在您引用遞迴調用型變數時，make 並不立即在變數引用和函數調用時擴展您設定的值；而是將它逐字儲存起來，將變數引用和函數調用也儲存起來，以備以後擴展。當您對於一個遞迴調用型變數使用‘+=’時，相當於對一個不擴展的文本追加新文本。
variable = value
variable += more

粗略等同於：

temp = value
variable = $(temp) more
當然，您從沒有定義過叫做 temp 的變數，如您在原始定義變數時，變數值中就包含變數引用，此時可以更為深刻地體現使用不同方式定義的的重要性。拿下面常見的例子，
CFLAGS = $(includes) -O
...
CFLAGS += -pg # enable profiling
第一行定義了變數 CFLAGS，而且變數 CFLAGS 引用了其他變數，includes。（變數 CFLAGS 用於 C 編譯器的規則，參閱*隱含規則目錄*。）由於定義時使用‘=’，所以變數 CFLAGS 是遞迴調用型變數，意味著‘$(includes) -O’在 make 處理變數 CFLAGS 定義時是不擴展的；也就是變數 includes 在生效之前不必定義，它僅需要在任何引用變數 CFLAGS 之前定義即可。如果我們試圖不使用‘+=’為變數 CFLAGS 追加文本，我們可能按下述方式：
CFLAGS := $(CFLAGS) -pg # enable profiling
這似乎很好，但結果絕不是我們所希望的。使用‘:=’重新定義變數 CFLAGS 為簡單擴展型變數，意味著 make 在設置變數 CFLAGS 之前擴展了‘$(CFLAGS) -pg’。如果變數 includes 此時沒有定義，我們將得到‘-0 -pg’，並且以後對變數 includes 的定義也不會有效。相反，使用‘+=’ 設置變數 CFLAGS 我們得到沒有擴展的‘$(CFLAGS) –0 -pg’，這樣保留了對變數 includes 的引用，在後面一個地方如果變數 includes 得到定義，‘$(CFLAGS)’仍然可以使用它的值。

6.7 override 指令

如果一個變數設置時使用了命令參數（參閱*變數重載*），那麼在 makefile 檔中通常的對該變數賦值不會生效。此時對該變數進行設置，您需要使用 override 指令，其格式如下：
override *variable = value*

或

override *variable* := *value*
為該變數追加更多的文本，使用：
override *variable* += *more text*
參閱*為變數值追加文本*。

override 指令不是打算擴大 makefile 和命令參數衝突，而是希望用它您可以改變和追加哪些設置時使用了命令參數的變數的值。

例如，假設您在運行 C 編譯器時總是使用'-g'開關，但您允許用戶像往常一樣使用命令參數指定其他開關，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，下面的例子也許就是您想要得：

override define foo
bar
endef
關於 define 指令的資訊參閱下節。

6.8 定義多行變數

設置變數值的另一種方法時使用 define 指令。這個指令有一個特殊的用法，既可以定義包含多行字元的變數。這使得定義命令的固定次序十分方便（參閱*定義固定次序命令*）。

在 define 指令同一行的後面一般是變數名，當然，也可以什麼也沒有。變數的值由下麵的幾行給出，值的結束由僅僅包含 endef 的一行標示出。除了上述在語法上的不同之外，define 指令象'='一樣工作：它創建了一個遞迴調用型變數（參閱*變數的兩個特色*）。變數的名字可以包括函數調用和變數引用，它們在指令讀入時擴展，以便能夠計算出實際的變數名。
define two-lines
echo foo
echo $(bar)
endef

變數的值在通常的賦值語句中只能在一行中完成，但在 define 指令中在 define 指令行以後 endef 行之前中間所有的行都是變數值的一部分（最後一行除外，因為標示 endef 那一行不能認為是變數值的一部分）。前面的例子功能上等同於：
two-lines = echo foo; echo $(bar)

因為兩命令之間用分號隔開，其行為很接近於兩個分離的 shell 命令。然而，注意使用兩個分離的行，意味著 make 請求 shell 兩次，每一行都在獨立的子 shell 中運行。參閱*執行命令*。

如果您希望使用 define 指令的變數定義比使用命令行定義的變數優先，您可以把 define 指令和 override 指令一塊使用：
override define two-lines
foo
$(bar)
endef
參閱 ***override** 指令*。

6.9 環境變數

make 使用的變數可以來自 make 的運行環境。任何 make 能夠看見的環境變數，在 make 開始運行時都轉變為同名同值的 make 變數。但是，在 makefile 檔中對變數的具體賦值，或使用帶有參數的命令，都可以對環境變數進行重載（如果明確使用'-e'標誌，環境變數的值可以對 makefile 檔中的賦值進行重載，參閱*選項概要*，但是這在實際中不推薦使用。）

這樣，通過在環境中設置變數 CFLAGS，您可以實現在絕大多數 makefile 檔中的所有 C 根源程式的編譯使用您選擇的開關。因為您知道沒有 makefile 將該變數用於其他任務，所以這種使用標準簡潔含義的變數是安全的（但這也是不可靠的，一些 makefile 檔可能設置變數 CFLAGS，從而使環境中變數 CFLAGS 的值失效）。當使用遞迴調用的 make 時，在外層 make 環境中定義的變數，可以傳遞給內層的 make（參閱*遞迴調用* ***make***）。缺省方式下，只有環境變數或在命令行中定義的變數才能傳遞給內層 make。您可以使用 export 指令傳遞其他變數，參閱*與子* ***make*** *通訊的變數*。

環境變數的其他使用方式都不推薦使用。將 makefile 的運行完全依靠環境變數的設置、超出 makefile 檔的控制範圍，這種做法是不明智的，因為不同的用戶運行同一個 makefile 檔有可能得出不同的結果。這和大部分 makefile 檔的意圖相違背。

變數 SHELL 在環境中存在，用來指定用戶對交互的 shell 的選擇，因此使用變數 SHELL 也存字類似的問題。這種根據選定值影響 make 運行的方式是很不受歡迎的。所以，make 將忽略環境中變數 SHELL 的值（在 MS-DOS 和 MS-Windows 中運行例外，但此時變數 SHELL 通常不設置值，參閱*執行命令*）。

6.10 特定目標變數的值

make 中變數的值一般是全局性的；既，無論它們在任何地方使用，它們的值是一樣的（當然，您重新設置除外）；自動變數是一個例外（參閱*自動變數*）。

另一個例外是特定目標變數的值，這個特點允許您可以根據 make 建造目標的變化改變變數的定義。象自動變數一樣，這些值只能在一個目標的命令腳本的上下文起作用。

可以象這樣設置特定目標變數的值：

target ... : *variable-assignment*

或這樣：

target ... : override *variable-assignment*

‘*target ...*’中可含有多個目標，如此，則設置的特定目標變數的值可在目標列表中的任一個目標中使用。‘*variable-assignment*’使用任何賦值方式都是有效的：遞迴調用型（‘=’）、靜態（‘:=’）、追加（‘+=’）或條件（‘?=’）。所有出現在‘*variable-assignment*’中的變數能夠在特定目標 *target ...*的上下文中使用：也就是任何以前為特定目標 *target ...*定義的特定目標變數的值在這些特定目標中都是有效的。注意這種變數值和總體變數值相比是局部的值：這兩種類型的變數不必有相同的類型（遞迴調用 vs.靜態）。

特定目標變數的值和其他 makefile 變數具有相同的優先權。一般在命令行中定義的變數（和強制使用‘-e’情況下的環境變數）的值佔據優先的地位，而使用 override 指令定義的特定目標變數的值則佔據優先地位。

特定目標變數的值有另外一個特點：當您定義一個特定目標變數時，該變數的值對特定目標 *target ...*的所有依賴有效，除非這些依賴用它們自己的特定目標變數的值將該變數重載。例如：

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

將在目標 prog 的命令腳本中設置變數 CFLAGS 的值為‘-g’，同時在創建`prog.o', `foo.o', 和`bar.o'的命令腳本中變數 CFLAGS 的值也是‘-g’，以及 prog.o',‘foo.o', 和‘bar.o'的依賴的創建命令腳本中變數 CFLAGS 的值也是‘-g’。

6.11 特定格式變數的值

除了特定目標變數的值（參閱上小節）外，GNU make 也支援特定格式變數的值。使用特定格式變數的值，可以為匹配指定格式的目標定義變數。在為目標定義特定目標變數後將

搜尋按特定格式定義的變數，在為該目標的父目標定義的特定目標變數前也要搜尋按特定格式定義的變數。

設置特定格式變數格式如下：

pattern ... : *variable-assignment*

或這樣：

pattern ... : override *variable-assignment*

這裏的‘pattern’是%-格式。象特定目標變數的值一樣，‘pattern ...’中可含有多個格式，如此，則設置的特定格式變數的值可在匹配列表中的任一個格式中的目標中使用。

‘*variable-assignment*’使用任何賦值方式都是有效的，在命令行中定義的變數的值佔據優先的地位，而使用 override 指令定義的特定格式變數的值則佔據優先地位。例如：

%.o : CFLAGS = -O

搜尋所有匹配格式%.o 的目標，並將它的變數 CFLAGS 的值設置為‘-0’。

7 makefile 檔的條件語句

一個條件語句可以導致根據變數的值執行或忽略 makefile 檔中一部分腳本。條件語句可以將一個變數與其他變數的值相比較，或將一個變數與一字串常量相比較。條件語句用於控制 make 實際看見的 makefile 檔部分，不能用於在執行時控制 shell 命令。

7.1 條件語句的例子

下述的條件語句的例子告訴 make 如果變數 CC 的值是‘gcc’時使用一個資料庫，如不是則使用其他資料庫。它通過控制選擇兩命令行之一作為該規則的命令來工作。‘CC=gcc’作為 make 改變的參數的結果不僅用於決定使用哪一個編譯器，而且決定連接哪一個資料庫。

```
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
```

該條件語句使用三個指令：ifeq、else 和 endif。

Ifeq 指令是條件語句的開始，並指明條件。它包含兩個參數，它們被逗號分開，並被擴在圓括號內。運行時首先對兩個參數變數替換，然後進行比較。在 makefile 中跟在 ifeq 後面的行是符合條件時執行的命令；否則，它們將被忽略。

如果前面的條件失敗，else 指令將導致跟在其後面的命令執行。在上述例子中，意味著當第一個選項不執行時，和第二個選項連在一起的命令將執行。在條件語句中，else 指令是可選擇使用的。

Endif 指令結束條件語句。任何條件語句必須以 endif 指令結束，後跟 makefile 檔中的正常內容。

上例表明條件語句工作在原文水準：條件語句的行根據條件要麼被處理成 makefile 文件的一部分或要麼被忽略。這是 makefile 檔重大的語法單位（例如規則）可以跨越條件語句的開始或結束的原因。

當變數 CC 的值是 gcc，上例的效果為：

```
foo: $(objects)
```

```
	$(CC) -o foo $(objects) $(libs_for_gcc)
```
當變數 CC 的值不是 gcc 而是其他值的時候，上例的效果為：
```
foo: $(objects)
	$(CC) -o foo $(objects) $(normal_libs)
```
相同的結果也能使用另一種方法獲得：先將變數的賦值條件化，然後再使用變數：
```
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 條件語句的語法

對於沒有 else 指令的條件語句的語法為：
conditional-directive
text-if-true
endif
‘*text-if-true*’可以是任何文本行，在條件為‘真’時它被認為是 makefile 檔的一部分；如果條件為‘假’，將被忽略。完整的條件語句的語法為：
conditional-directive
text-if-true
else
text-if-false
endif
如果條件為‘真’，使用‘*text-if-true*’；如果條件為‘假’，使用‘*text-if-false*’。‘*text-if-false*’可以是任意多行的文本。

關於‘*conditional-directive*’的語法對於簡單條件語句和複雜條件語句完全一樣。有四種不同的指令用於測試不同的條件。下麵是指令表：
ifeq (*arg1*, *arg2*)
ifeq '*arg1*' '*arg2*'
ifeq "*arg1*" "*arg2*"
ifeq "*arg1*" '*arg2*'
ifeq '*arg1*' "*arg2*"

擴展參數 arg1、arg2 中的所有變數引用，並且比較它們。如果它們完全一致，則使用‘*text-if-true*’，否則使用‘*text-if-false*’（如果存在的話）。您經常要測試一個變數是否有非空值，當經過複雜的變數和函數擴展得到一個值，對於您認為是空值，實際上有可能由於包含空格而被認為不是空值，由此可能造成混亂。對於此，您可以使用 strip 函數從而避免空格作為非空值的干擾。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*"
擴展參數 arg1、arg2 中的所有變數引用，並且比較它們。如果它們不同，則使用‘*text-if-true*’，否則使用‘*text-if-false*’（如果存在的話）。
ifdef *variable-name*

如果變數‘*variable-name*’是非空值，‘*text-if-true*’有效，否則，‘*text-if-false*’有效（如果存在的話）。變數從沒有被定義過則變數是空值。注意 **ifdef** 僅僅測試變數是否有值。它不能擴展到看變數是否有非空值。因而，使用 **ifdef** 測試所有定義過的變數都返回**‘真’**，但那些象**‘foo=’**情況除外。測試空值請使用 **ifeq($(foo),)**。例如：

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

設置‘frobozz'的值為‘yes', 而：:

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

設置‘frobozz' 為‘no'。

ifndef *variable-name*

如果變數‘*variable-name*’是空值，‘*text-if-true*’有效，否則，‘*text-if-false*’有效（如果存在的話）。

在指令行前面允許有多餘的空格，它們在處理時被忽略，但是不允許有 Tab（如果一行以 Tab 開始，那麼該行將被認為是規則的命令行）。除此之外，空格和 Tab 可以插入到行的任何地方，當然指令名和參數中間除外。以‘#’開始的注釋可以在行的結尾。

在條件語句中另兩個有影響的指令是 else 和 endif。這兩個指令以一個單詞的形式出現，沒有任何參數。在指令行前面允許有多餘的空格，空格和 Tab 可以插入到行的中間，以‘#’開始的注釋可以在行的結尾。

條件語句影響 make 使用的 makefile 檔。如果條件為‘真’，make 讀入‘*text-if-true*’包含的行；如果條件為‘假’，make 讀入‘*text-if-false*’包含的行（如果存在的話）；makefile 檔的語法單位，例如規則，可以跨越條件語句的開始或結束。

當讀入 makefile 檔時，Make 計算條件的值。因而您不能在測試條件時使用自動變數，因為他們是命令執行時才被定義（參閱*自動變數*）。

為了避免不可忍受的混亂，在一個 makefile 檔中開始一個條件語句，而在另外一個 makefile 檔中結束這種情況是不允許的。然而如果您試圖引入包含的 makefile 檔不中斷條件語句，您可以在條件語句中編寫 include 指令。

7.3 測試標誌的條件語句

您可以使用變數 MAKEFLAGS 和 findstring 函數編寫一個條件語句，用它來測試例如‘-t’等的 make 命令標誌（參閱*字串替換和分析的函數*）。這適用於僅使用 touch 標誌不能完全更改檔的時間戳的場合。

findstring 函數檢查一個字串是否為另一個字串的子字串。如果您要測試‘-t’標誌，使用‘-t’作為第一個字串，將變數 MAKEFLAGS 的值作為另一個字串。例如下面的例子是安排使用‘ranlib –t’完成一個檔案檔的更新：

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

首碼‘+’表示這些命令行是遞迴調用行，即使是用‘-t’標誌它們一樣要執行。參閱*遞迴調用* ***make***。

8 文本轉換函數

函數允許您在 makefile 檔中處理文本、計算檔、操作使用命令等。在函數調用時您必須指定函數名以及函數操作使用的參數。函數處理的結果將返回到 makefile 檔中的調用點，其方式和變數替換一樣。

8.1 函數調用語法

函數調用和變數引用類似，它的格式如下：
$(*function arguments*)

或這樣：

${*function arguments*}

這裏‘function’是函數名，是 make 內建函數列表中的一個。當然您也可以使用創建函數 call 創建的您自己的函數。‘*arguments*’是該函數的參數。參數和函數名之間是用空格或 Tab 隔開，如果有多個參數，它們之間用逗號隔開。這些空格和逗號不是參數值的一部分。包圍函數調用的定界符，無論圓括號或大括弧，可以在參數中成對出現，在一個函數調用中只能有一種定界符。如果在參數中包含變數引用或其他的函數調用，最好使用同一種定界符，如寫為‘$(subst a,b,$(x))', 而不是 `$(subst a,b,${x})'。這是因為這種方式不但比較清楚，而且也有在一個函數調用中只能有一種定界符的規定。

為每一個參數寫的文本經過變數替換或函數調用處理，最終得到參數的值，這些值是函數執行必須依靠的文本。另外，變數替換是按照變數在參數中出現的次序進行處理的。

逗號和不成對出現的圓括號、大括弧不能作為文本出現在參數中，前導空格也不能出現在第一個參數中。這些字元不能被變數替換處理為參數的值。如果需要使用這些字元，首先定義變數 comma 和 space，它們的值是單獨的逗號和空格字元，然後在需要的地方因用它們，如下例：

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

這裏函數 subst 的功能是將變數 foo 中的空格用逗號替換，然後返回結果。

8.2 字串替換和分析函數

這裏有一些用於操作字串的函數：
$(subst *from*,*to*,*text*)
在文本‘*text*’中使用‘*to*’替換每一處‘*from*’。例如：

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

結果為‘fEEt on the street’。
$(patsubst *pattern*,*replacement*,*text*)
尋找‘text’中符合格式‘pattern’的字，用‘replacement’替換它們。這裏‘pattern’中包含通配符‘%’，它和一個字中任意個數的字元相匹配。如果‘replacement’中也含有通配符‘%’，則這個‘%’被和‘pattern’中通配符‘%’匹配的文本代替。在函數 patsubst 中的‘%’可以用反斜杠(‘\’)引用。引用字元‘%’的反斜杠可以被更多反斜杠引用。引用字元‘%’和其他反斜杠的反斜杠在比較檔案名或有一個 stem（徑）代替它之前從格式中移出。使用反斜杠引用字元‘%’不會帶來其他

麻煩。例如，格式‘the\%weird\\%pattern\\’是‘the%weird\’ 加上通配符‘%’然後和字串‘pattern\\’連接。最後的兩個反斜杠由於不能影響任何統配符‘%’所以保持不變。在字之間的空格間被壓縮為單個空格，前導以及結尾空格被丟棄。例如：

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

的結果為：‘x.c.o bar.o’。替換引用是實現函數 patsubst 功能一個簡單方法：

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

等同於 ：

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

另一個通常使用的函數 patsubst 的簡單方法是：替換檔案名的尾碼。

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

等同於：

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

例如您可能有一個 OBJ 檔的列表：

objects = foo.o bar.o baz.o

要得到這些檔的原始檔案，您可以簡單的寫為：

$(objects:.o=.c)

代替規範的格式：

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

$(strip *string*)

去掉前導和結尾空格，並將中間的多個空格壓縮為單個空格。這樣，‘$(strip a b c)’結果為‘a b c’。函數 strip 和條件語句連用非常有用。當使用 ifeq 或 ifneq 把一些值和空字串‘’比較時，您通常要將一些僅由空格組成的字串認為是空字串（參閱 ***makefile*** *中的條件語句*）。如此下面的例子在實現預期結果時可能失敗：

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

在條件指令 ifneq 中用函數調用‘$(strip $(needs_made))’代替變數引用‘$(needs_made)’將不再出現問題。

$(findstring *find*,*in*)

在字串‘in’中搜尋‘find’，如果找到，則返回值是‘find’，否則返回值為空。您可以在一個條件中使用該函數測試給定的字串中是否含有特定的子字串。這樣，下面兩個例子：

$(findstring a,a b c)

$(findstring a,b c)

將分別產生值‘a’和‘’。對於函數 findstring 的特定用法參閱*測試標誌的條件語句*。

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

返回在‘text’中由空格隔開且匹配格式‘pattern...’的字，對於不符合格式‘pattern...’的字移出。格式用‘%’寫出，和前面論述過的函數 patsubst 的格式相同。函數 filter 可以用來變數分離類型不同的字串。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'；但僅有‘foo.c',‘bar.c' 和 ‘baz.s' 指明用命令編譯。

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

返回在‘text’中由空格隔開且不匹配格式‘pattern...’的字，對於符合格式‘pattern...’的字移出。只是函數 filter 的反函數。例如：

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

下面產生不包含在變數‘mains’中的 OBJ 檔的檔列表：

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

$(sort *list*)

將‘list’中的字按字母順序排序，並取掉重複的字。輸出是由單個空格隔開的字的列表。

$(sort foo bar lose)

返回值是‘bar foo lose’。順便提及，由於函數 sort 可以取掉重複的字，您就是不關心排序也可以使用它的這個特點。

這裏有一個實際使用函數 subst 和 patsubst 的例子。假設一個 makefile 檔使用變數 VPATH 指定 make 搜尋依賴檔的一系列路徑（參閱 ***VPATH:****依賴搜尋路徑*）。這個例子表明怎樣告訴 C 編譯器在相同路徑列表中搜尋頭文件。

變數 VPATH 的值是一列用冒號隔開的路徑名，如‘src:../headers'。首先，函數 subst 將冒號變為空格：

$(subst :, ,$(VPATH))

這產生值‘src ../headers'。然後，函數 patsubst 為每一個路徑名加入‘-I’標誌，這樣這些路徑可以加到變數 CFLAGS 中，就可以自動傳遞給 C 編譯器：

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

結果是在以前給定的變數 CFLAGS 的值後追加文本‘-Isrc -I../headers’。Override 指令的作用是即使以前使用命令參數指定變數 CFLAGS 的值，新值也能起作用。參閱 ***override*** *指令*。

8.3 檔案名函數

其中幾個內建的擴展函數和拆分檔案名以及列舉檔案名相關聯。下面列舉的函數都能執行對檔案名的特定轉換。函數的參數是一系列的檔案名，檔案名之間用空格隔開（前導和結尾空格被忽略）。列表中的每一個檔案名都採用相同的方式轉換，而且結果用單個空格串聯在一起。

$(dir *names*...)

抽取‘names’中每一個檔案名的路徑部分，檔案名的路徑部分包括從檔案名的開始到最後一個斜杠（含斜杠）之前的一切字元。如果檔案名中沒有斜杠，路徑部分是‘./’。如：

$(dir src/foo.c hacks)

產生的結果為 ‘src/ ./’。

$(notdir *names*...)

抽取‘names’中每一個檔案名中除路徑部分外一切字元（真正的檔案名）。如果檔案名中沒有斜杠，則該檔案名保持不變，否則，將路徑部分移走。一個檔案名如果僅包含路徑部分（以斜杠結束的檔案名）將變為空字串。這是非常不幸的，因為這意味著在結果中如果有這種檔案名存在，兩檔案名之間的空格將不是由相同多的空格隔開。但現在我們並不能看到其他任何有效的代替品。例如：

$(notdir src/foo.c hacks)

產生的結果為‘foo.c hacks’。

$(suffix *names*...)

抽取‘names’中每一個檔案名的尾碼。如果檔案名中（或含有斜杠，且在最後一個斜杠後）含有句點，則尾碼是最後那個句點以後的所有字元，否則，尾碼是空字串。如果結果為空意味著‘names’沒有帶尾碼檔案名，如果檔中含有多個檔案名，則結果列出的尾碼數很可能比原檔案名數目少。例如：

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

產生的結果是‘.c .c’。

$(basename *names*...)

抽取‘names’中每一個檔案名中除尾碼外一切字元。如果檔案名中（或含有斜杠，且在最後一個斜杠後）含有句點，則基本名字是從開始到最後一個句點（不包含）間的所有字元。如果沒有句點，基本名字是整個檔案名。例如：

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

產生的結果為‘src/foo src-1.0/bar hacks’。

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

參數‘names’作為一系列的檔案名，檔案名之間用空格隔開；suffix 作為一個單位。將 Suffix（尾碼）的值附加在每一個獨立檔案名的後面，完成後將檔案名串聯起來，它們之間用單個空格隔開。例如：

$(addsuffix .c,foo bar)
結果為‘foo.c bar.c’。

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

參數‘names’作為一系列的檔案名，檔案名之間用空格隔開；prefix 作為一個單位。將 preffix（首碼）的值附加在每一個獨立檔案名的前面，完成後將檔案名串聯起來，它們之間用單個空格隔開。例如：

$(addprefix src/,foo bar)

結果為‘src/foo src/bar’。

$(join *list1*,*list2*)

將兩個參數串聯起來：兩個參數的第一個字串聯起來形成結果的第一個字，兩個參數的第二個字串聯起來形成結果的第二個字，以此類推。如果一個參數比另一個參數的字多，則多餘的字原封不動的拷貝到結果上。例如，‘$(join a b,.c .o)'產生‘a.c b.o'。字之間多餘的空格不再保留，它們由單個空格代替。該函數可將函數 dir、notdir 的結果合併，產生原始給定的檔列表。

$(word *n,text*)

返回‘text’中的第 n 個字。N 的合法值從 1 開始。如果 n 比‘text’中的字的數目大，則返回空值。例如：

$(word 2, foo bar baz)
返回 ‘bar’。

$(wordlist *s,e,text*)

返回‘text’中的從第 s 個字開始到第 e 個字結束的一列字。S、e 的合法值從 1 開始。如果 s 比‘text’中的字的數目大，則返回空值；如果 e 比‘text’中的字的數目大，則返回從第 s 個字開始到‘text’結束的所有字；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’中字的數目。這樣‘text’中的最後一個字是‘$(word $(words *text*),*text*)’。

$(firstword *names...*)

參數‘names’作為一系列的檔案名，檔案名之間用空格隔開；返回第一個檔案名，其餘的忽略。例如：

$(firstword foo bar)
產生結果‘foo’。 雖然 $(firstword *text*) 和 $(word 1,*text*)的作用相同，但第一個函數因為簡單而保留下來。

$(wildcard *pattern*)

參數‘pattern’是一個檔案名格式，典型的包含通配符（和 shel 中的檔案名一樣）。函數 wildcard 的結果是一列和格式匹配的且檔存在的檔案名，檔案名之間用一個空格隔開，參閱*在檔案名中使用通配符*。

8.4 函數 foreach

函數 foreach 和其他函數非常不同，它導致一個文本塊重複使用，而且每次使用該文本塊進行不同的替換；它和 shell sh 中的命令 for 及 C-shell csh 中的命令 foreach 類似。

函數 foreach 語法如下：

$(foreach *var,list,text*)

前兩個參數，‘var’和‘list’，將首先擴展，注意最後一個參數‘text’此時不擴展；接著，對每一個‘list’擴展產生的字，將用來為‘var’擴展後命名的變數賦值；然後‘text’引用該變數擴展；因此它每次擴展都不相同。

結果是由空格隔開的‘text’ 在‘list’中多次擴展的字組成的新的‘list’。‘text’多次擴展的字串聯起來，字與字之間由空格隔開，如此就產生了函數 foreach 的返回值。

這是一個簡單的例子，將變數‘files’的值設置為 ‘dirs’中的所有目錄下的所有檔的列表：

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

這裏‘text’是‘$(wildcard $(dir)/*)’。第一個為變數 dir 發現的值是‘a’,所以產生函數 foreach 結果的第一個字為‘$(wildcard a/*)’；第二個重複的值是‘b’，所以產生函數 foreach 結果的第二個字為‘$(wildcard b/*)’；第三個重複的值是‘c’，所以產生函數 foreach 結果的第三個字為‘$(wildcard c/*)’；等等。該例子和下例有共同的結果：

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

如果‘text’比較複雜，您可以使用附加變數為它命名，這樣可以提高程式的可讀性：

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

這裏我們使用變數 find_file。我們定義變數 find_file 時，使用了‘=’，因此該變數為遞迴調用型變數，這樣變數 find_file 所包含的函數調用將在函數 foreach 控制下在擴展；對於簡單擴展型變數將不是這樣，在變數 find_file 定義時就調用函數 wildcard。

函數 foreach 對變數‘var’沒有長久的影響,它的值和變數特色在函數 foreach 調用結束後將和前面一樣，其他從‘list’得到的值僅在函數 foreach 執行時起作用，它們是暫時的。變數‘var’在函數 foreach 執行期間是簡單擴展型變數,如果在執行函數 foreach 之前變數‘var’沒有定義，則函數 foreach 調用後也沒有定義。參閱*變數的兩個特色*。

當使用複雜變數運算式產生變數名時應特別小心,因為許多奇怪的字元作為變數名是有效的，但很可能不是您所需要的，例如：

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

如果變數 find_file 擴展引用名為‘Esta escrito en espanol!’變數，上例是有效的，但它極易帶來錯誤。

8.5 函數 if

函數 if 對在函數上下文中擴展條件提供了支援（相對於 GNU make makefile 檔中的條件語句，例如 ifeq 指令，參閱*條件語句的語法*）。

一個函數 if 的調用，可以包含兩個或三個參數：

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

第一個參數‘condition’，首先把前導、結尾空格去掉，然後擴展。如果擴展為非空字串，則條件‘condition’為‘真’；如果擴展為空字串，則條件‘condition’為‘假’。

如果條件‘condition’為‘真’,那麼計算第二個參數‘then-part’的值，並將該值作為整個函數 if 的值。

如果條件‘condition’為‘假’,第三個參數如果存在，則計算第三個參數‘else-part’的值，並將該值作為整個函數 if 的值；如果第三個參數不存在，函數 if 將什麼也不計算，返回空值。

注意僅能計算‘then-part’和‘else-part’二者之一，不能同時計算。這樣有可能產生副作用（例如函數 shell 的調用）。

8.6 函數 call

函數 call 是唯一的創建新的帶有參數函數的函數。您可以寫一個複雜的表達是作為一個變數的值，然後使用函數 call 用不同的參數調用它。

函數 call 的語法為：

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

當 make 擴展該函數時，它將每一個參數‘param’賦值給臨時變數$(1)、$(2)等；變數$(0)的值是變數‘variable’。對於參數‘param’的數量無沒有最大數目限制，也沒有最小數目限制，但是如果使用函數 call 而沒有任何參數，其意義不大。

變數‘variable’在這些臨時變數的上下文中被擴展為一個 make 變數，這樣，在變數‘variable’中對變數‘$(1)’的引用決定了調用函數 call 時對第一個參數‘param’的使用。

注意變數‘variable’是一個變數的名稱，不是對該變數的引用，所以，您不能採用‘$’和圓括號的格式書寫該變數，當然，如果您需要使用非常量的檔案名，您可以在檔案名中使用變數引用。

如果變數名是內建函數名，則該內建函數將被調用（即使使用該名稱的 make 變數已經存在）。函數 call 在給臨時變數賦值以前首先擴展參數，這意味著，變數‘variable’對內建函數的調用採用特殊的規則進行擴展，象函數 foreach 或 if，它們的擴展結果和您預期的結果可能不同。下面的一些例子能夠更清楚的表達這一點。

該例子時使用巨集將參數的順序翻轉：

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

foo = $(call reverse,a,b)
```

這裏變數 foo 的值是‘b a’。

下面是一個很有意思的例子：它定義了一個巨集，使用該巨集可以搜尋變數 PATH 包含的所有目錄中的第一個指定類型的程式：

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

LS := $(call pathsearch,ls)
```

現在變數 LS 的值是‘/bin/ls’或其他的類似的值。

在函數 call 中可以使用嵌套。每一次遞迴調用都可以為它自己的局部變數‘$(1)’等賦值，從而代替上一層函數 call 賦的值。例如：這實現了映射函數功能。

```
map = $(foreach a,$(2),$(call $(1),$(a)))
```

現在您可以映射（map）僅有一個參數的函數，如函數 origin，一步得到多個值：

```
o = $(call map,origin,o map MAKE)
```

最後變數 o 包含諸如‘file file default’這樣的值。

警告：在函數 **call** 的參數中使用空格一定要十分小心。因為在其他函數中，第二個或接下來的參數中的空格是不刪除的，這有可能導致非常奇怪的結果。當您使用函數 call 時，去掉參數中任何多餘的空格才是最安全的方法。

8.7 函數 origin

函數 origin 不想一般函數，它不對任何變數的值操作；它僅僅告訴您一些關於一個變數的資訊；它特別的告訴您變數的來源。

函數 origin 的語法：

```
$(origin variable)
```

注意變數‘variable’是一個查詢變數的名稱，不是對該變數的引用所以，您不能採用‘$’和圓括號的格式書寫該變數，當然，如果您需要使用非常量的檔案名，您可以在檔案名中使用變數引用。

函數 origin 的結果是一個字串，該字串變數是怎樣定義的：

‘undefined'

如果變數‘variable’從沒有定義。

‘default'

變數‘variable’是缺省定義，通常和命令 CC 等一起使用，參閱*隱含規則使用的變數*。注意如果您對一個缺省變數重新進行了定義，函數 origin 將返回後面的定義。

‘environment'

變數‘variable’作為環境變數定義，選項‘-e’沒有打開（參閱*選項概要*）。

‘environment override'

變數‘variable’作為環境變數定義，選項‘-e’已打開（參閱*選項概要*）。

‘file'

變數‘variable’在 makefile 中定義。

‘command line'

變數‘variable’在命令行中定義。

‘override'

變數‘variable’在 makefile 中用 override 指令定義（參閱 ***override** 指令*）。

‘automatic'

變數‘variable’是自動變數，定義它是為了執行每個規則中的命令（參閱*自動變數*）。

這種資訊的基本用途（其他用途是滿足您的好奇心）是使您要瞭解變數值的依據。例如，假設您有一個名為‘foo’的 makefile 檔，它包含了另一個名為‘bar’的 makefile 檔，如果在環境變數中已經定義變數‘bletch’，您希望運行命令‘make –f bar’在 makefile 檔‘bar’中重新定義變數‘bletch’。但是 makefile 檔‘foo’在包括 makefile 檔‘bar’之前已經定義了變數‘bletch’，而且您也不想使用 override 指令定義，那麼您可以在 makefile 檔‘foo’中使用 override 指令，因為 override 指令將會重載任何命令行中的定義，所以其定義的優先權超越以後在 makefile 檔‘bar’中的定義。因此 makefile 檔‘bar’可以包含：

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

如果變數‘bletch’在環境中定義，這裏將重新定義它。

即使在使用選項‘-e’的情況下，您也要對來自環境的變數‘bletch’重載定義，則您可以使用如下內容：

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

如果‘$(origin bletch)’返回‘environment’或‘environment override’，這裏將對變數‘bletch’重新定義。參閱*字串替換和分析函數*。

8.8 函數 shell

除了函數 wildcard 之外，函數 shell 和其他函數不同，它是 make 與外部環境的通訊工具。函數 shell 和在大多數 shell 中後引號（’）執行的功能一樣：它用於命令的擴展。這意味著它起著調用 shell 命令和返回命令輸出結果的參數的作用。Make 僅僅處理返回結果，再返回結果替換調用點之前，make 將每一個換行符或者一對回車/換行符處理為單個空格；如果返回結果最後是換行符（和回車符），make 將把它們去掉。由函數 shell 調用的命令，一旦函數調用展開，就立即執行。在大多數情況下，當 makefile 檔讀入時函數 shell 調用的命令就已執行。例外情況是在規則命令行中該函數的調用，因為這種情況下只有在命令運行時函數才能擴展，其他調用函數 shell 的情況和此類似。

這裏有一些使用函數 shell 的例子：

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

將含有檔 foo 的目錄設置為變數 contents 的值，是用空格（而不是換行符）分離每一行。

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

將‘*.c’的擴展設置為變數 files 的值。除非 make 使用非常怪異的 shell，否則這條語句和‘wildcard *.c’的結果相同。

8.9 控制 make 的函數

這些函數控制 make 的運行方式。通常情況下，它們用來向用戶提供 makefile 檔的資訊或在偵測到一些類型的環境錯誤時中斷 make 運行。

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

通常‘text’是致命的錯誤資訊。注意錯誤是在該函數計算時產生的，因此如果您將該函數放在命令的腳本中或遞迴調用型變數賦值的右邊，它直到過期也不能計算。‘text’將在錯誤產生之前擴展，例如：

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

如果變數 ERROR01 已經定義，在將 makefile 檔讀入時產生致命的錯誤。或，

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

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

如果 err 目標被調用，在 make 運行時產生致命錯誤。

$(warning *text*...)

該函數和函數 error 工作的方式類似，但此時 make 不退出，即雖然‘text’擴展並顯示結果資訊，但 make 仍然繼續執行。擴展該函數的結果是空字串。

9 運行 make

講述編譯程序的 makefile 檔，可以由多種方式實現。最簡單的方式是編譯所有過期的檔，對於通常所寫的 makefile 檔，如果不使用任何參數運行 make，那麼將這樣執行。

但是您也許僅僅更新一部分檔；您也許需要使用不同的編譯器或不同的編譯選項；您也許僅僅希望找出過時的檔而不更新它們。這些只有通過在運行 make 時給出參數才能實現。

退出 make 狀態有三種情況：

0

表示 make 成功完成退出。

2

退出狀態為 2 表示 make 運行中遇到錯誤，它將列印資訊描述錯誤。

1

退出狀態為 1 表示您運行 make 時使用了‘-q’標誌，並且 make 決定一些檔沒有更新。參閱*代替執行命令*。

9.1 指定 makefile 檔的參數

指定 makefile 檔案名的方法是使用‘-f’或‘--file’選項（‘--makefile’也能工作）。例如，‘-f altmake’說明名為‘altmake’的檔作為 makefile 檔。

如果您連續使用‘-f’標誌幾次，而且每一個‘-f’後面都帶有參數，則所有指定的檔將連在一起作為 makefile 檔。

如果您不使用‘-f’或‘--file’選項，缺省的是按次序尋找‘GNUmakefile', ‘makefile', 和‘Makefile'，使用這三個中第一個能夠找到的存在檔或能夠創建的檔，參閱*編寫* ***makefile*** *檔*。

9.2 指定最終目標的參數

最終目標（gaol）是 make 最終努力更新的目標。其他更新的目標是因為它們作為最終目標的依賴，或依賴的依賴，等等以此類推。

缺省情況下，makefile 檔中的第一個目標是最終目標（不計算那些以句點開始的目標）。因此，makefile 檔的第一個編譯目標是對整個程式或程式組描述。如果第一個規則同時擁有幾個目標，只有該規則的第一個目標是缺省的最終目標。

您可以使用 make 的參數指定最終目標。方法是使用目標的名字作為參數。如果您指定幾個最終目標，make 按您命名時的順序一個接一個的處理它們。

任何在 makefile 檔中出現的目標都能作為最終目標（除了以‘-’開始或含有‘=’的目標，它們一種解析為開關，另一種是變數定義）。即使在 makefile 檔中沒有出現的目標，按照隱含規則可以說明怎樣生成，也能指定為最終目標。

Make 將在命令行中使用特殊變數 MAKECMGOALS 設置您指定的最終目標。如果沒有在命令行指定最終目標，該變數的值為空值。注意該變數值能在特殊場合下使用。

一個合適的例子是在清除規則中避免刪除包括‘.d’的檔（參閱*自動產生依賴*），因這樣 make 不會一創建它們，就立即又刪除它們：

```
sources = foo.c bar.c

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

指定最終目標的一個用途是僅僅編譯程序的一部分或程式組中的幾個程式。如是這樣，您可以將您希望變異的檔指定為最終目標。例如，在一個路徑下包含幾個程式，一個 makefile 檔以下面的格式開始：

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

如果您僅對程式 size 編譯，則您可以使用‘make size’命令，這樣就只有您指定的程式才重新編譯。

指定最終目標的另一個用途是編譯產生哪些沒有正常生成的檔。例如，又一個檔需要調試，或一個版本的程式需要編譯進行測試，然而該檔不是 makefile 檔規則中缺省最終目標的依賴，此時，可以使用最終目標參數指定它們。

指定最終目標的另一個用途是運行和一個假想目標（參閱*假想目標*）或空目標（*使用空目標記錄事件*）相聯繫的命令。許多 makefile 檔包含一個假想目標‘clean’刪除除了原文件以外的所有檔。正常情況下，只有您具體指明使用‘make clean’命令，make 才能執行上述任務。下面列出典型的假想目標和空目標的名稱。對 GNU make 套裝軟體使用的所有標準目標名參閱*用戶標準目標*：

‘all'
　　創建 makefile 檔的所有頂層目標。

`clean'
　　刪除所有 make 正常創建的檔。

`mostlyclean'
　　象假像目標‘clean’，但避免刪除人們正常情況下不重新建造的一少部分檔。例如，用於 GCC 的目標‘mostlyclean’不刪除‘libgcc.a’，因為重建它的情況十分稀少，而且創建它又需要很多時間。

`distclean'
`realclean'
`clobber'
這些目標可能定義為比目標‘clean’ 刪除更多的檔。例如，刪除配置檔或為編譯正常創建的準備檔，甚至 makefile 檔自身不能創建的文件。

‘install’
向命令搜尋目錄下拷貝可執行檔；向可執行檔尋找目錄下拷貝可執行檔使用的輔助檔。

‘print’
列印發生變化的檔列表。

‘tar’
創建原始檔案的壓縮‘tar’檔。

‘shar’
為原始檔案創建一個 shell 的檔案檔。

‘dist’
為原始檔案創建一個發佈檔。這可能是‘tar’檔， ‘shar’檔，或多個上述的壓縮版本檔。

‘TAGS’
更新該程式的‘tags’標籤。

`check'
`test'
對該 makefile 檔創建的程式執行自我測試。

9.3 代替執行命令

makefile 檔告訴 make 怎樣識別一個目標是否需要更新以及怎樣更新每一個目標。但是更新目標並不是您一直需要的，一些特定的選項可以用來指定 make 的其他活動：

`-n'
`--just-print'
`--dry-run'
`--recon'
‘No-op’。make 的這項活動是列印用於創建目標所使用的命令，但並不執行它們。
`-t'
`--touch'
‘touch’。這項活動是做更新標誌，實際卻不更改它們。換句話說，make 假裝編譯了目標，但實際對它們沒有一點兒改變。
`-q'
`--question'
‘question’。這項活動是暗中察看目標是否已經更新；但是任何情況下也不執行命令。換句話說，即不編譯也不輸出。
`-W *file*'
`--what-if=*file*'
`--assume-new=*file*'
`--new-file=*file*'
‘What if’。每一個‘-W’標誌後跟一個檔案名。所有檔案名的更改時間被 make 記錄為當前時間，但實際上更改時間保持不變。如果您要更新檔，您可以使用‘-W’標誌和‘-n’標誌連用看看將發生什麼。

使用標誌‘-n’，make 列印那些正常執行的命令，但卻不執行它們。

使用標誌‘-t’，make 忽略規則中的命令，對那些需要更新的目標使用‘touch’命令。如果不使用‘-s’或.SILENT，‘touch’命令同樣列印。為了提高執行效率，make 並不實際調用程式 touch，而是使 touch 直接運行。

使用標誌‘-q’，make 不列印輸出也不執行命令，如果所有目標都已經更新到最新，make 的退出狀態是 0；如果一部分需要更新，退出狀態是 1；如果 make 遇到錯誤，退出狀態是 2，因此您可以根據沒有更新的目標尋找錯誤。

在運行 make 時對以上三個標誌如果同時兩個或三個將產生錯誤。標誌‘-n’、‘-t’和‘-s’對那些以字元‘+’開始的命令行和包含字串‘$(MAKE)' 或‘${MAKE}'命令行不起作用。注意僅有這些以字元‘+’開始的命令行和包含字串‘$(MAKE)' 或‘${MAKE}'命令行運行時不注意這些選項。參閱*變數* ***MAKE*** *的工作方式*。

‘-W’標誌有一下兩個特點：

- ！ 如果同時使用標誌‘-n’或‘-q’，如果您更改一部分檔，看看 make 將會做什麼。
- ！ 沒有使用標誌‘-n’或‘-q’，如果 make 運行時採用標誌‘-W’，則 make 假裝所有檔已經更新，但實際上不更改任何檔。

注意選項‘-p’和‘-v’允許您得到更多的 make 資訊或正在使用的 makefile 檔的資訊（參閱*選項概要*）。

9.4 避免重新編譯檔

有時您可能改變了一個原始檔案，但您並不希望編譯所有依靠它的檔。例如，假設您在一個許多檔都依靠的頭檔種添加了一個宏或一個聲明，按照保守的規則，make 認為任何對於該頭檔的改變，需要編譯所有依靠它的檔，但是您知道那是不必要的，並且您沒有等待它們完全編譯的時間。

如果您提前瞭解改變頭檔以前的問題，您可以使用‘-t’選項。該標誌告訴 make 不運行規則中的命令，但卻將所有目標的時間戳改到最新。您可按下述步驟實現上述計畫：
1、用 make 命令重新編譯那些需要編譯的原始檔案；
2、更改頭文件；
3、使用‘make –t’命令改變所有目標檔的時間戳，這樣下次運行 make 時就不會因為頭檔的改變而編譯任何一個檔。

如果在重新編譯那些需要編譯的原始檔案前已經改變了頭檔，則按上述步驟做已顯得太晚了；作為補救措施，您可以使用‘-o file’標誌，它能將指定的檔的時間戳假裝改為以前的

時間戳（參閱*選項概要*）。這意味著該檔沒有更改，因此您可按下述步驟進行：

1、使用‘make -o file’命令重新編譯那些不是因為改變頭檔而需要更新的檔。如果涉及幾個頭檔，您可以對每個頭檔都使用‘-o’標誌進行指定。

2、使用‘make –t’命令改變所有目標檔的時間戳。

9.5 變數重載

使用‘=’定義的變數：‘v=x’將變數 v 的值設為 x。如果您用該方法定義了一個變數，在 makefile 檔後面任何對該變數的普通賦值都將被 make 忽略，要使它們生效應在命令行將它們重載。

最為常見的方法是使用傳遞附加標誌給編譯器的靈活性。例如，在一個 makefile 檔中，變數 CFLAGS 已經包含了運行 C 編譯器的每一個命令，因此，如果僅僅鍵入命令 make 時，檔‘foo.c’將按下面的方式編譯：

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

這樣您在 makefile 檔中對變數 CFALAGS 設置的任何影響編譯器運行的選項都能生效，但是每次運行 make 時您都可以將該變數重載，例如：如果您說‘make CFLAGS='-g -O'’，任何 C 編譯器都將使用‘cc -c -g -O’編譯程序。這還說明了在重載變數時，怎樣使用 shell 命令中的引用包括空格和其他特殊字元在內的變數的值。

變數 CFALAGS 僅僅是您可以使用這種方式重載的許多標準變數中的一個，這些標準變數的完整列表見*隱含規則使用的變數*。

您也可以編寫 makefile 察看您自己的附加變數，從而使用戶可通過更改這些變數控制 make 運行時的其他面貌。

當您使用命令參數重載變數時，您可以定義遞迴調用擴展型變數或簡單擴展型變數。上例中定義的是遞迴調用擴展型變數，如果定義簡單擴展型變數，請使用‘:=’代替‘=’。注意除非您在變數值中使用變數引用或函數調用，這兩種變數沒有任何差異。

利用這種方式也可以改變您在 makfile 檔中重載的變數。在 makfile 檔中重載的變數是使用 override 指令，是和‘override *variable* = *value*’相似的命令行。詳細內容參閱 ***override** 指令*。

9.6 測試編譯程序

正常情況下，在執行 shell 命令時一旦有錯誤發生，make 立即退出返回非零狀態；不會為任何目標繼續運行命令。錯誤表明 make 不能正確的創建最終目標，並且 make 一發現錯誤就立即報告。

當您編譯您修改過的程式時，這不是您所要的結果。您希望 make 能夠經可能的試著編譯每一個程式，並盡可能的顯示每一個錯誤。

這種情況下，您可以使用‘-k’或‘--keep-going’選項。這種選項告訴 make 遇到錯誤返回非零狀態之前，繼續尋找該目標的依賴，如果有必要則重新創建它們。例如，在編譯一個目標檔時發現錯誤，即使 make 已經知道連接它們已是不可能的，‘make –k’也將繼續編譯其他目標檔。除在 shell 命令失敗後繼續運行外，即使發在 make 不知道如何創建的目標和依賴檔以後，‘make –k’也將盡可能的繼續運行。在沒有‘-k’選項時，這些錯誤將是致命的（參閱*選項概要*）。

通常情況下，make 的行為是基於假設您的目標是使最終目標更新；一旦它發現這是不可能的它就立即報告錯誤。選項‘-k’說真正的目標是盡可能測試改變對程式的影響，發現存在的問題，以便在下次運行之前您可以糾正它們。這是 Emacs M-x compile 命令缺省傳遞‘-k’選項的原因。

9.7 選項概要

下面是所有 make 能理解的選項列表：

`-b'

`-m'

和其他版本 make 相容時，這些選項被忽略。

`-C *dir*'

`--directory=*dir*'

在將 makefile 讀入之前，把路徑切換到‘dir’下。如果指定多個‘-C’選項，每一個都是相對於前一個的解釋：‘-C/-C etc’等同於‘-C/etc’。該選項典型用在遞迴調用 make 過程中，參閱*遞迴調用* ***make***。

‘-d’

在正常處理後列印調試資訊。調試資訊說明哪些檔用於更新，哪個檔作為比較時間戳的標準以及比較的結果，哪些檔實際上需要更新，需要考慮、使用哪些隱含規則等等----一切和 make 決定最終幹什麼有關的事情。‘-d’選項等同於‘--debug=a’選項（參見下面內容）。

`--debug[=*options*]'

在正常處理後列印調試資訊。可以選擇各種級別和類型的輸出。如果沒有參數，列印‘基本’級別的調試資訊。以下是可能的參數，僅僅考慮第一個字母，各個值之間使用逗號或空格隔開：

a (*all*)

顯示所有調試資訊，該選項等同於‘-d’選項。

b (*basic*)

基本調試資訊列印每一個已經過時的目標，以及它們重建是否成功。

v (*verbose*)

比‘基本’級高一個的等級的調試資訊。包括 makefile 檔的語法分析結果，沒有必要更新的依賴等。該選項同時包含基本調試資訊。

i (*implicit*)

列印隱含規則搜尋目標的資訊。該選項同時包含基本調試資訊。

j (*jobs*)

列印各種子命令調用的詳細資訊。

m (*makefile*)

以上選項不包含重新創建 makefile 檔的資訊。該選項包含了這方面的資訊。注意，選項‘all’也不包含這方面資訊。該選項同時包含基本調試資訊。

`-e'

`--environment-overrides'

設置從環境中繼承來的變數的優先權高於 makefile 檔中的變數的優先權。參閱*環境變數*。

`-f *file*'

`--file=*file*'

`--makefile=*file*'

將名為‘file’的檔設置為 makefile 檔。參閱*編寫* ***makefile*** *檔*。

`-h'

`--help'

向您提醒 make 能夠理解的選項，然後退出。

`-i'

`--ignore-errors'

忽略重建檔執行命令時產生的所有錯誤。

`-I *dir*'

`--include-dir=*dir*'

指定搜尋包含 makefile 檔的路徑‘dir’。參閱*包含其他* ***makefile*** *文件*。如果同時使用幾個‘-I’選項用於指定路徑，則按照指定的次序搜尋這些路徑。

`-j [*jobs*]'

`--jobs[=*jobs*]'

指定同時執行的命令數目。如果沒有參數 make 將同時執行盡可能多的任務；如果有多個‘-j’選項，則僅最後一個選項有效。詳細內容參閱*並行執行*。注意在 MS-DOS 下，該選項被忽略。

`-k'

`--keep-going'
在出現錯誤後，盡可能的繼續執行。當一個目標創建失敗，則所有依靠它的目標檔將不能重建，而這些目標的其他依賴則可繼續處理。參閱*測試編譯程序*。
`-l [*load*]'
`--load-average[=*load*]'
`--max-load[=*load*]'
指定如果有其他任務正在運行，並且平均負載已接近或超過‘load’（一個浮點數），則此時不啟動新任務。如果沒有參數則取消以前關於負載的限制。參閱*並行執行*。
`-n'
`--just-print'
`--dry-run'
`--recon'
列印要執行的命令，但卻不執行它們。參閱*代替執行命令*。
`-o *file*'
`--old-file=*file*'
`--assume-old=*file*'
即使檔 file 比它的依賴‘舊’，也不重建該檔。不要因為文件 file 的改變而重建任何其他文件。該選項本質上是假裝將該檔的時間戳改為舊的時間戳，以至於依靠它的規則被忽略。參閱*避免重新編譯檔*。
`-p'
`--print-data-base'
列印資料庫（規則和變數的值），這些資料來自讀入 makefile 檔的結果；然後象通常那樣執行或按照別的指定選項執行。如果同時給出‘-v’開關，則列印版本資訊（參閱下面內容）。使用‘make –qp’則列印資料庫後不試圖重建任何檔。使用‘make –p –f/dev/null’則列印預定義的規則和變數的資料庫。資料庫輸出中包含檔案名，以及命令和變數定義的行號資訊。它是在複雜環境中很好的調試工具。
`-q'
`--question'
‘問題模式’。不列印輸出也不執行命令，如果所有目標都已經更新到最新，make 的退出狀態是 0；如果一部分需要更新，退出狀態是 1；如果 make 遇到錯誤，退出狀態是 2，參閱*代替執行命令*。
`-r'
`--no-builtin-rules'
排除使用內建的隱含規則（參閱*使用隱含規則*）。您仍然可以定義您自己的格式規則（參閱*定義和重新定義格式規則*）。選項‘-r’同時也清除了缺省的尾碼列表和尾碼規則（參閱*過時的尾碼規則*）。但是您可以使用.SUFFIXES 規則定義您自己的尾碼。注意，使用選項‘-r’僅僅影響規則；缺省變數仍然有效（參閱*隱含規則使用的變數*）；參閱下述的選項‘-R’。
`-R'
`--no-builtin-variables'
排除使用內建的規則變數（參閱*隱含規則使用的變數*）。當然，您仍然可以定義自己的變數。選項‘-R’自動使選項‘-r’生效；因為它去掉了隱含規則所使用的變數的定義，所以隱含規則也就失去了存在的意義。
`-s'
`--silent'
`--quiet'
沉默選項。不回顯那些執行的命令。參閱*命令回顯*。
`-S'
`--no-keep-going'
`--stop'
使選項‘-k’失效。除非在遞迴調用 make 時，通過變數 MAKEFLAGS 從上層 make 繼承選項‘-k’，或您在環境中設置了選項‘-k’，否則沒有必要使用該選項。
`-t'
`--touch'
標誌檔已經更新到最新，但實際沒有更新它們。這是假裝那些命令已經執行，用於愚弄將來

的 make 調用。參閱*代替執行命令*。
`-v'
`--version'
列印 make 程式的版本資訊，作者列表和沒有擔保的注意資訊，然後退出。
`-w'
`--print-directory'
列印執行 makefile 檔時涉及的所有工作目錄。這對於跟蹤 make 遞迴調用時複雜嵌套產生的錯誤非常有用。參閱*遞迴調用* ***make***。實際上，您很少需要指定該選項，因為 make 已經替您完成了指定。參閱***'--print-directory'****選項*。
`--no-print-directory'
在指定選項'-w'的情況下，禁止列印工作路徑。這個選項在選項'-w'自動打開而且您不想看多餘資訊時比較有用。參閱***'--print-directory'****選項*。
`-W *file*'
`--what-if=*file*'
`--new-file=*file*'
`--assume-new=*file*'
假裝目標檔已經更新。在使用標誌'n'時，它將向您表明更改該檔會發生什麼。如果沒有標誌'n'它和在運行 make 之前對給定的檔使用 touch 命令的結果幾乎一樣，但使用該選項 make 只是在的想像中更改該檔的時間戳。參閱*代替執行命令*。
`--warn-undefined-variables'
當 make 看到引用沒有定義的變數時，發佈一條警告資訊。如果您按照複雜方式使用變數，當您調試您的 makefile 檔時，該選項非常有用。

10 使用隱含規則

重新創建目標檔的一些標準方法是經常使用的。例如，一個傳統的創建 OBJ 檔的方法是使用 C 編譯器，如 cc，編譯 C 語言根源程式。

隱含規則能夠告訴 make 怎樣使用傳統的技術完成任務，這樣，當您使用它們時，您就不必詳細指定它們。例如，有一條編譯 C 語言根源程式的隱含規則，檔案名決定運行哪些隱含規則；另如，編譯 C 語言程式一般是使用'.c'檔，產生'.o'文件。因此， make 據此和檔案名的尾碼就可以決定使用編譯 C 語言根源程式的隱含規則。一系列的隱含規則可按順序應用；例如，make 可以從一個'.y'檔，借助'.c'檔，重建一個'.o'檔，參閱*隱含規則鏈*。內建隱含規則的命令需要使用變數，通過改變這些變數的值，您就可以改變隱含規則的工作方式。例如，變數 CFLAGS 控制隱含規則用於編譯 C 程式傳遞給 C 編譯器的標誌，參閱*隱含規則使用的變數*。通過編寫格式規則，您可以創建您自己的隱含規則。參閱*定義和重新定義格式規則*。

尾碼規則是對定義隱含規則最有限制性。格式規則一般比較通用和清楚，但是尾碼規則卻要保持相容性。參閱*過時的尾碼規則*。

10.1 使用隱含規則

允許 make 對一個目標檔尋找傳統的更新方法，您所有做的是避免指定任何命令。可以編寫沒有命令行的規則或根本不編寫任何規則。這樣，make 將根據存在的原始檔案的類型或要生成的文件類型決定使用何種隱含規則。

例如，假設 makefile 檔是下麵的格式：

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

因為您提及了檔'foo.o'，但是您沒有給出它的規則，make 將自動尋找一條隱含規則，該規

則能夠告訴 make 怎樣更新該檔。無論檔‘foo.o’存在與否，make 都會這樣執行。

如果能夠找到一條隱含規則，則它就能夠對命令和一個或多個依賴（原始檔案）提供支援。如果您要指定附加的依賴，例如頭檔，但隱含規則不能支援，您需要為目標‘foo.o’寫一條不帶命令行的規則。

每一條隱含規則都有目標格式和依賴格式；也許多條隱含規則有相同的目標格式。例如，有數不清的規則產生‘.o’文件：使用 C 編譯器編譯‘.C’文件；使用 Pascal 編譯器編譯‘.p’文件；等等。實際應用的規則是那些依賴存在或可以創建的規則。所以，如果您有一個‘.C’檔，make 將運行 C 編譯器；如果您有一個‘.p’檔，make 將運行 Pascal 編譯器；等等。

當然，您編寫一個 makefile 檔時，您知道您要 make 使用哪一條隱含規則，以及您知道 make 將選擇哪一條規則，因為您知道那個依賴檔是假設存在的。預定義的隱含規則列表的詳細內容參閱*隱含規則目錄*。

首先，我們說一條隱含規則可以應用，該規則的依賴必須‘存在或可以創建’。一個檔‘可以創建’是說該檔在 makefile 中作為目標或依賴被提及，或者該檔可以經過一條隱含規則的遞迴調用後能夠創建。如果一條隱含規則的依賴是另一條隱含規則的結果，我們說產生了‘鏈’。參閱*隱含規則鏈*。

總體上說，make 為每一個目標搜尋隱含規則，為沒有命令行的雙冒號規則搜尋隱含規則。僅作為依賴被提及的檔，將被認為是一個目標，如果該目標的規則沒有指定任何內容，make 將為它搜尋隱含規則。對於詳細的搜尋過程參閱*隱含規則的搜尋演算法*。

注意，任何具體的依賴都不影響對隱含規則的搜尋。例如，認為這是一條具體的規則：
foo.o: foo.p
文件 foo.p 不是首要條件，這意味著 make 按照隱含規則可以從一個 Pascal 根源程式（‘.p’檔）創建 OBJ 檔，也就是說一個‘.o’檔可根據‘.p’檔進行更新。但文件 foo.p 並不是絕對必要的；例如，如果檔 foo.c 也存在，按照隱含規則則是從檔 foo.c 重建 foo.o，這是因為 C 編譯規則在預定義的隱含規則列表中比 Pascal 規則靠前，參閱*隱含規則目錄*。

如果您不希望使用隱含規則創建一個沒有命令行的目標，您可以通過添加分號為該目標指定空命令。參閱*使用空命令*。

10.2 隱含規則目錄

這裏列舉了預定義的隱含規則的目錄，這些隱含規則是經常應用的，當然如果您在 makefile 檔中重載或刪除後，這些隱含規則將會失去作用，詳細內容參閱*刪除隱含規則*。選項‘-r’或‘--no-builtin-rules’將刪除所有預定義的隱含規則。

並不是所有的隱含規則都是預定義的，在 make 中很多預定義的隱含規則是尾碼規則的擴展，因此，那些預定義的隱含規則和尾碼規則的列表相關（特殊目標.SUFFIXES 的依賴列表）。缺省的尾碼列表為：.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。所有下面描述的隱含規則，如果它們的依賴中有一個出現在這個尾碼列表中，則是尾碼規則。如果您更改這個尾碼列表，則只有那些由一個或兩個出現在您指定的列表中的尾碼命名的預定義尾碼規則起作用；那些尾碼沒有出現在列表中的規則被禁止。對於詳細的關於尾碼規則的描述參閱*過時的尾碼規則*。

Compiling C programs（編譯 C 程式）

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

Compiling C++ programs （編譯 C++程式）

‘*n*.o'自動由‘*n*.cc' 或‘*n*.C'使用命令‘$(CXX) -c $(CPPFLAGS) $(CXXFLAGS)’生成。 我們鼓勵您對 C++原始檔案使用尾碼‘.cc' 代替尾碼‘.C’。

Compiling Pascal programs （編譯 Pascal 程式）

‘*n*.o'自動由‘*n*.p'使用命令‘$(PC) -c $(PFLAGS)'生成。

Compiling Fortran and Ratfor programs （編譯 Fortran 和 Ratfor 程式）

‘*n*.o'自動由‘*n*.r', ‘*n*.F'或‘*n*.f' 運行 Fortran 編譯器生成。使用的精確命令如下：

`.f'
`$(FC) -c $(FFLAGS)'.
`.F'
`$(FC) -c $(FFLAGS) $(CPPFLAGS)'.
`.r'
`$(FC) -c $(FFLAGS) $(RFLAGS)'.

Preprocessing Fortran and Ratfor programs （預處理 Fortran 和 Ratfor 程式）

‘*n*.f' 自動從 ‘*n*.r'或‘*n*.F'得到。該規則僅僅是與處理器把一個 Ratfor 程式或能夠預處理的 Fortran 程式轉變為標準的 Fortran 程式。使用的精確命令如下：

`.F'
`$(FC) -F $(CPPFLAGS) $(FFLAGS)'.
`.r'
`$(FC) -F $(FFLAGS) $(RFLAGS)'.

Compiling Modula-2 programs（編譯 Modula-2 程式）

‘*n*.sym'自動由‘*n*.def'使用命令‘$(M2C) $(M2FLAGS) $(DEFFLAGS)'生成。 ‘*n*.o' 從‘*n*.mod'生成；命令為:‘$(M2C) $(M2FLAGS) $(MODFLAGS)'。

Assembling and preprocessing assembler programs （彙編以及預處理組合語言程式）

‘*n*.o'自‘*n*.S'運行 C 編譯器，cpp，生成。命令為:‘$(CPP) $(CPPFLAGS)'。

Linking a single object file （連接一個簡單的 OBJ 檔）

‘*n*' 自動由‘*n*.o' 運行C編譯器中的連接程式 linker (通常稱為 ld)生成。命令為: ‘$(CC) $(LDFLAGS) *n*.o $(LOADLIBES) $(LDLIBS)'。該規則對僅有一個根源程式的簡單程式或對同時含有多個 OBJ 檔（可能來自於不同的原始檔案）的程式都能正常工作。如果同時含有多個 OBJ 檔，則其中必有一個 OBJ 檔的名字和可執行檔案名匹配。例如：

```
x: y.o z.o
```

當‘x.c', ‘y.c' 和‘z.c' 都存在時則執行：

```
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
```

對於更複雜的情況，例如沒有一個 OBJ 檔的名字和可執行檔案名匹配，您必須為連接寫一條具體的命令。每一種能自動生成‘.o’的檔，可以在沒有‘-c’選項的情況下使用編譯器(‘$(CC)',‘$(FC)'或‘$(PC)'； C 編譯器‘$(CC)'也適用於組合語言程式)自動連接。當然也可以使用 OBJ 檔作為中間檔，但編譯、連接一步完成速度將快很多。

Yacc for C programs （由 Yacc 生成 C 程式）

‘*n*.c'自動由‘*n*.y'使用命令‘$(YACC) $(YFLAGS)'運行 Yacc 生成。

Lex for C programs （由 Lex 生成 C 程式）

‘*n*.c'自動由‘*n*.l' 運行 Lex 生成。命令為：‘$(LEX) $(LFLAGS)'。

Lex for Ratfor programs （由 Lex 生成 Rator 程式）

‘*n*.r'自動由‘*n*.l' 運行 Lex 生成。命令為：‘$(LEX) $(LFLAGS)'。 對於所有的 Lex 檔，無論它們產生 C 代碼或 Ratfor 代碼，都使用相同的尾碼‘.l’進行轉換，在特定場合下，使用 make 自動確定您使用哪種語言是不可能的。如果 make 使用‘.l’檔重建一個 OBJ 檔，它必須猜想使用哪種編譯器。它很可能猜想使用的是 C 編譯器，因為 C 編譯器更加普遍。如果您使用 Ratfor 語言，請確保在 makefile 檔中提及了‘*n*.r',使 make 知道您的選擇。否則，如果您專用 Ratfor 語言，不使用任何 C 檔，請在隱含規則尾碼列表中將‘.c’剔除：

```
.SUFFIXES:
.SUFFIXES: .o .r .f .l ...
```

Making Lint Libraries from C, Yacc, or Lex programs（由 C, Yacc, 或 Lex 程式創建 Lint 庫）

‘*n*.ln' 可以從‘*n*.c' 運行 lint 產生。命令為：‘ $(LINT) $(LINTFLAGS)

$(CPPFLAGS) –i’。用於 C 程式的命令和用於‘*n*.y'或‘*n*.l'程式相同。

TeX and Web（TeX 和 Web）

‘*n*.dvi'可以從‘*n*.tex' 使用命令‘$(TEX)'得到。‘*n*.tex'可以從‘*n*.web'使用命令‘$(WEAVE)'得到；或者從‘*n*.w' (和‘*n*.ch'，如果‘*n*.ch'存在或可以建造) 使用命令‘$(CWEAVE)'。‘*n*.p'可以從‘*n*.web'使用命令‘$(TANGLE)'產生。‘*n*.c' 可以從‘*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' 可以從‘*n*.texinfo',‘*n*.texi', 或‘*n*.txinfo', 使用命令‘$(TEXI2DVI) $(TEXI2DVI_FLAGS)'得到。‘*n*.info'可以從‘*n*.texinfo',‘*n*.texi', 或‘*n*.txinfo', 使用命令‘$(MAKEINFO) $(MAKEINFO_FLAGS)'創建。

RCS

檔‘*n*'必要時可以從名為‘*n*,v'或‘RCS/*n*,v'的 RCS 文件中提取。具體命令是：‘$(CO) $(COFLAGS)'。檔‘*n*'如果已經存在，即使 RCS 檔比它新，它不能從 RCS 檔中提取。用於 RCS 的規則是最終的規則，參閱*萬用規則*，所以 RCS 不能夠從任何原始檔案產生，它們必須存在。

SCCS

檔‘*n*'必要時可以從名為‘s.*n*'或‘SCCS/s.*n*'的 SCCS 文件中提取。具體命令是：‘$(GET) $(GFLAGS)'。用於 SCCS 的規則是最終的規則，參閱*萬用規則*，所以 SCCS 不能夠從任何原始檔案產生，它們必須存在。SCCS 的優點是，檔 ‘*n*' 可以從檔 ‘*n*.sh'拷貝並生成可執行檔(任何人都可以)。這用於 shell 的腳本，該腳本在 SCCS 內部檢查。因為 RCS 允許保持檔的可執行性，所以您沒有必要將該特點用於 RCS 檔。我們推薦您避免使用 SCCS，RCS 不但使用廣泛，而且是免費的軟體。選擇自由軟體代替相當的（或低劣的）收費軟體，是您對自由軟體的支援。

通常情況下，您要僅僅改變上表中的變數，需要參閱下面的文檔。

隱含規則的命令實際使用諸如 COMPILE.c, LINK.p, 和 PREPROCESS.S 等等變數，它們的值包含以上列出的命令。Make 按照慣例進行處理，如，編譯‘.x’原始檔案的規則使用變數‘COMPILE.*x*’；從‘.x’原始檔案生成可執行檔使用變數‘LINK.*x*’；預處理‘.x’原始檔案使用變數‘PREPROCESS.*x*’。

任何產生 OBJ 檔的規則都使用變數‘OUTPUT_OPTION’；make 依據編譯時間選項定義該變數的值是‘-o $@’或空值。當原始檔案分佈在不同的目錄中，您應該使用‘-O’選項保證輸出到正確的檔中；使用變數 VPATH 時同樣（參閱為*依賴搜尋目錄*）。一些系統的編譯器不接受針對 OBJ 檔的‘-o’開關；如果您在這樣的系統上運行，並使用了變數 VPATH，一些檔的編譯輸出可能會放到錯誤的地方。解決辦法是將變數 OUTPUT_OPTION 值設為：‘; mv $*.o $@’。

10.3 隱含規則使用的變數

內建隱含規則的命令對預定義變數的使用是開放的；您可以在 makefile 檔中改變變數的值，也可以使用 make 的運行參數或在環境中改變，如此，在不對這些規則本身重新定義的情況下，就可以改變這些規則的工作方式。您還可以使用選項‘-R’或‘--no-builtin-variables’刪除所有隱含規則使用的變數。

例如，編譯 C 程式的命令實際是‘$(CC) -c $(CFLAGS) $(CPPFLAGS)’，變數缺省的值是‘cc’或空值，該命令實際是‘cc –c’。如重新定義變數‘CC’的值為‘ncc’，則所有隱含規則將使用‘ncc’作為編譯 C 語言根源程式的編譯器。通過重新定義變數‘CFLAGS’的值為‘-g’，則您可將‘-g’選項傳遞給每個編譯器。所有的隱含規則編譯 C 程式時都使用‘$CC’獲得編譯器的名稱，並且都在傳遞給編譯器的參數中都包含‘$(CFLAGS)’。

隱含規則使用的變數可分為兩類：一類是程式名變數（象 cc），另一類是包含程式運行參數的變數（象 CFLAGS）。(‘程式名’可能也包含一些命令參數，但是它必須以一個實際可以執行的程式名開始。) 如果一個變數值中包含多個參數，它們之間用空格隔開。

這裏是內建規則中程式名變數列表：

AR

檔案管理程式；缺省為：‘ar'.

AS

彙編編譯程序；缺省為：‘as'.

CC

C 語言編譯程序；缺省為：‘cc'.

CXX

C++編譯程序；缺省為：‘g++'.

CO

從 RCS 檔中解壓縮抽取檔程式；缺省為：‘co'.

CPP

帶有標準輸出的 C 語言預處理程式；缺省為：‘$(CC) -E'.

FC

Fortran 以及 Ratfor 語言的編譯和預處理程式；缺省為：‘f77'.

GET

從 SCCS 檔中解壓縮抽取檔程式；缺省為：‘get'.

LEX

將 Lex 語言轉變為 C 或 Ratfor 程式的程式；缺省為：‘lex'.

PC

Pascal 程式編譯程序；缺省為：‘pc'.

YACC

將 Yacc 語言轉變為 C 程式的程式；缺省為：‘yacc'.

YACCR

將 Yacc 語言轉變為 Ratfor 程式的程式；缺省為：‘yacc -r'.

MAKEINFO

將 Texinfo 原始檔案轉換為資訊檔的程式；缺省為：‘makeinfo'.

TEX

從 TeX 源產生 TeX DVI 檔的程式；缺省為：‘tex'.

TEXI2DVI

從 Texinfo 源產生 TeX DVI 檔的程式；缺省為：‘texi2dvi'.

WEAVE

將 Web 翻譯成 TeX 的程式；缺省為：‘weave'.

CWEAVE

將 CWeb 翻譯成 TeX 的程式；缺省為：‘cweave'.

TANGLE

將 Web 翻譯成 Pascal 的程式；缺省為：‘tangle'.

CTANGLE

將 Web 翻譯成 C 的程式；缺省為：‘ctangle'.

RM

刪除檔的命令；缺省為：‘rm -f'.

這裏是值為上述程式附加參數的變數列表。在沒有注明的情況下，所有變數的值為空值。

ARFLAGS

用於檔案管理程式的標誌，缺省為：‘rv'.

ASFLAGS

用於彙編編譯器的額外標誌 (當具體調用‘.s'或‘.S'文件時)。

CFLAGS

用於 C 編譯器的額外標誌。

CXXFLAGS

用於 C++編譯器的額外標誌。

COFLAGS

用於 RCS co 程式的額外標誌。

CPPFLAGS

用於 C 預處理以及使用它的程式的額外標誌 (C 和 Fortran 編譯器)。

FFLAGS

用於 Fortran 編譯器的額外標誌。

GFLAGS

用於 SCCS get 程式的額外標誌。

LDFLAGS

用於調用 linker（'ld'）的編譯器的額外標誌。

LFLAGS

用於 Lex 的額外標誌。

PFLAGS

用於 Pascal 編譯器的額外標誌。

RFLAGS

用於處理 Ratfor 程式的 Fortran 編譯器的額外標誌。

YFLAGS

用於 Yacc 的額外標誌。Yacc。

10.4 隱含規則鏈

有時生成一個檔需要使用多個隱含規則組成的序列。例如，從檔'n.y'生成檔'n.o'，首先運行隱含規則 Yacc，其次運行規則 cc。這樣的隱含規則序列稱為隱含規則鏈。

如果檔'n.c'存在或在 makefile 檔中提及，則不需要任何特定搜尋：make 首先發現通過 C 編譯器編譯'n.c'可生成該 OBJ 檔，隨後，考慮生成'n.c'時，則使用運行 Yacc 的規則。這樣可最終更新'n.c'和'n.o'。

即使在檔'n.c'不存在或在 makefile 檔中沒有提及的情況下，make 也能想像出在檔'n.y'和'n.o'缺少連接！這種情況下，'n.c'稱為中間檔。一旦 make 決定使用中間檔，它將把中間檔輸入資料庫，好像中間檔在 makefile 檔中提及一樣；按照隱含規則的描述創建中間檔。

中間檔和其他檔一樣使用自己的規則重建，但是中間檔和其他檔相比有兩種不同的處理方式。

第一個不同的處理方式是如果中間檔不存在 make 的行為不同：平常的檔 b 如果不存在，make 認為一個目標依靠檔 b，它總是創建檔 b，然後根據檔 b 更新目標；但是檔 b 若是中間檔，make 很可能不管它而進行別的工作，即不創建檔 b，也不更新最終目標。只有在檔 b 的依賴比最終目標'新'時或有其他原因時，才更新最終目標。

第二個不同點是 make 在更新目標創建檔 b 後，如果檔 b 不再需要，make 將把它刪除。所以一個中間檔在 make 運行之前和 make 運行之後都不存在。Make 向您報告刪除時列印一條'rm –f'命令，表明有檔被刪除。

通常情況下，任何在 makefile 檔中提及的目標和依賴都不是中間檔。但是，您可以特別指定一些檔為中間檔，其方法為：將要指定為中間檔的檔作為特殊目標 .INTERMEDIATE 的依賴。這種方法即使對採用別的方法具體提及的檔也能生效。

您通過將檔標誌為 secondary 檔可以阻止自動刪除中間檔。這時，您將您需要保留的中間檔指定為特殊目標 .SECONDARY 的依賴即可。對於 secondary 檔，make 不會因為它不存在而去創建它，也不會自動刪除它。secondary 檔必須也是中間檔。

您可以列舉一個隱含規則的目標格式（例如%.o）作為特殊目標 .PRECIOUS 的依賴，這樣您就可以保留那些由隱含規則創建的檔案名匹配該格式的中間檔。參閱 *中斷和關閉* ***make***。

一個隱含規則鏈至少包含兩個隱含規則。例如，從'RCS/foo.y,v'創建檔'foo'需要運行 RCS、Yacc 和 cc，檔 foo.y 和 foo.c 是中間檔，在運行結束後將被刪掉。

沒有一條隱含規則可以在隱含規則鏈中出現兩次以上（含兩次）。這意味著，make 不會簡單的認為從檔'foo.o.o'創建檔 foo 不是運行 linker 兩次。這還可以強制 make 在搜尋一個隱含規則鏈時阻止無限迴圈。

一些特殊的隱含規則可優化隱含規則鏈控制的特定情況。例如，從檔 foo.c 創建檔 foo 可以被擁有編譯和連接的規則鏈控制，它使用 foo.o 作為中間檔。但是對於這種情況存在一條特別的規則，使用簡單的命令 cc 可以同時編譯和連接。因為優化規則在規則表中的前面，所以優化規則和一步一步的規則鏈相比，優先使用優化規則。

10.5 定義與重新定義格式規則

您可以通過編寫格式規則定義隱含規則。該規則看起來和普通規則類似，不同之處在於格式規則的目標中包含字元‘%’（只有一個）。目標是匹配檔案名的格式；字元‘%’可以匹配任何非空的字串，而其他字元僅僅和它們自己相匹配。依賴用‘%’表示它們的名字和目標名關聯。

格式‘%.o : %.c’是說將任何‘stem.c’檔編譯為‘stem.o’文件。

在格式規則中使用的‘%’擴展是在所有變數和函數擴展以後進行的，它們是在 makefile 檔讀入時完成的。參閱*使用變數*和*轉換文本函數*。

10.5.1 格式規則簡介

格式規則是在目標中包含字元‘%’（只有一個）的規則，其他方面看起來和普通規則相同。目標是可以匹配檔案名的格式，字元‘%’可以匹配任何非空的字串，而其他字元僅僅和它們自己相匹配。

例如‘%.c’匹配任何以‘.c’結尾的檔案名；‘s.%.c’匹配以‘s.’開始並且以‘.c’結尾的檔案名，該檔案名至少包含 5 個字元（因為‘%’至少匹配一個字元）。匹配‘%’的子字串稱為 stem(徑)。依賴中使用‘%’表示它們的名字中含有和目標名相同的 stem。要使用格式規則，檔案名必須匹配目標的格式，而且符合依賴格式的檔必須存在或可以創建。下面規則：

%.o : %.c ; *command...*

表明要創建檔‘n.o’，使用‘n.c’作為它的依賴，而且檔‘n.c’ 必須存在或可以創建。

在格式規則中，依賴有時不含有‘%’。這表明採用該格式規則創建的所有檔都是採用相同的依賴。這種固定依賴的格式規則在有些場合十分有用。

格式規則的依賴不必都包含字元‘%’，這樣的規則是一個有力的常規通配符，它為任何匹配該目標格式規則的檔提供創建方法。參閱*定義最新類型的缺省規則*。

格式規則可以有多個目標，不象正常的規則，這種規則不能扮演具有相同依賴和命令的多條不同規則。如果一格式規則具有多個目標，make 知道規則的命令對於所有目標來說都是可靠的，這些命令只有在創建所目標時才執行。當為匹配一目標搜尋格式規則時，規則的目標格式和規則要匹配的目標不同是十分罕見的，所以 make 僅僅擔心目前對檔給出命令和依賴是否有問題。注意該檔的命令一旦執行，所有目標的時間戳都會更新。

格式規則在 makefile 檔中的次序很重要，因為這也是考慮它們的次序。對於多個都能使用的規則，使用最先出現的規則。您親自編寫的規則比內建的規則優先。注意依賴存在或被提及的規則優先於依賴需要經過隱含規則鏈生成的規則。

10.5.2 格式規則的例子

這裏有一些實際在 make 中預定義的格式規則例子，第一個，編譯‘.c’檔生成‘.o’檔的規則：

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

定義了一條編譯‘x.c’檔生成‘x.o’檔的規則，命令使用自動變數‘$@’和‘$<’ 替換任何情況使用該規則的目標檔和原始檔案。參閱*自動變數*。

第二個內建的例子：

```
% :: RCS/%,v
        $(CO) $(COFLAGS) $<
```

定義了在子目錄‘RCS’中根據相應檔‘x.v’生成檔‘x’的規則。因為目標是‘%’，只要相對應的依賴檔存在，該規則可以應用於任何檔。雙冒號表示該規則是最終規則，它意味著不能是中間檔。參閱*萬用規則*。

下面的格式規則有兩個目標：

```
%.tab.c %.tab.h: %.y
        bison -d $<
```

這告訴 make 執行命令‘bison -d *x*.y’將創建‘x.tab.c’和‘x.tab.h’。如果檔 foo 依靠檔‘parse.tab.o’和‘scan.o’，而文件‘scan.o’又依靠檔‘parse.tab.h’，當‘parse.y’發生變化，命令‘bison -d parse.y’執行一次。‘parse.tab.o’和‘scan.o’的依賴也隨之更新。（假設文件‘parse.tab.o’由文件‘parse.tab.c’編譯生成，檔‘scan.o’由文件‘scan.c’生成，當連接‘parse.tab.o’、‘scan.o’和其他依賴生成檔 foo 時，上述規則能夠很好執行。）

10.5.3 自動變數

假設您編寫一個編譯‘.c’檔生成‘.o’檔的規則：您怎樣編寫命令‘CC’，使它能夠操作正確的檔案名？您當然不能將檔案名直接寫進命令中，因為每次使用隱含規則操作的檔案名都不一樣。

您應該使用 make 的另一個特點，自動變數。這些變數在規則每次執行時都基於目標和依賴產生新值。例如您可以使用變數‘$@’代替目標檔案名，變數‘$<’代替依賴檔案名。

下麵是自動變數列表：

$@

規則的目標檔案名。如果目標是一個檔案成員，則變數‘$@’ 檔案檔的檔案名。對於有多個目標的格式規則（參閱*格式規則簡介*），變數‘$@’是那個導致規則命令運行的目標檔案名。

$%

當目標是檔案成員時，該變數是目標成員名，參閱*使用 **make** 更新檔案檔*。例如，如果目標是‘foo.a(bar.o)'，則‘$%'的值是‘bar.o'，‘$@'的值是‘foo.a'。如果目標不是檔案成員，則‘$%'是空值。

$<

第一個依賴的檔案名。如果目標更新命令來源於隱含規則，該變數的值是隱含規則添加的第一個依賴。參閱*使用隱含規則*。

$?

所有比目標‘新’的依賴名，名字之間用空格隔開。對於為檔案成員的依賴，只能使用已命名的成員。參閱*使用 **make** 更新檔案檔*。

$^

所有依賴的名字，名字之間用空格隔開。對於為檔案成員的依賴，只能使用已命名的成員。參閱*使用 **make** 更新檔案檔*。對同一個目標來說，一個檔只能作為一個依賴，不管該檔的檔案名在依賴列表中出現多少次。所以，如果在依賴列表中，同一個檔案名出現多次，變數‘$^’的值仍然僅包含該檔案名一次。

$+

該變數象‘$^'，但是，超過一次列出的依賴將按照它們在 makefile 檔中出現的次序複製。這主要的用途是對於在按照特定順序重複庫檔案名很有意義的地方使用連接命令。

$*

和隱含規則匹配的 stem(徑)，參閱*格式匹配*。如果一個目標為‘dir/a.foo.b'，目標格式規則為：‘a.%.b' ，則 stem 為‘dir/foo'。在構建相關檔案名時 stem 十分有用。在靜態格式規則中，stem 是匹配目標格式中字元‘%’的檔案名中那一部分。在一個沒有 stem 具體規則中；變數‘$*' 不能以該方法設置。如果目標名以一種推薦的尾碼結尾（參閱*過時的尾碼規則*），變數‘$*'設置為目標去掉該尾碼後的部分。例如，如果目標名是‘foo.c'，則變數‘$*' 設置為‘foo', 因為‘.c' 是一個尾碼。GNU make 處理這樣奇怪的事情是為了和其他版本的 make 相容。在隱含規則和靜態格式規則以外，您應該儘量避免使用變數‘$*'。在具體規則中如果目標名不以推薦的尾碼結尾，則變數‘$*’在該規則中設置為空值。

當您希望僅僅操作那些改變的依賴，變數‘$?' 即使在具體的規則中也很有用。例如，假設名為‘lib’的檔案檔包含幾個 OBJ 檔的拷貝，則下面的規則僅將發生變化的 OBJ 檔拷貝到檔案檔：

```
lib: foo.o bar.o lose.o win.o
        ar r lib $?
```

在上面列舉的變數中，有四個變數的值是單個檔案名。三個變數的值是檔案名列表。這七個變數都有僅僅存放檔的路徑名或僅僅存放目錄下檔案名的變體。變數的變體名是由變數名追加字母‘D’或‘F’構成。這些變體在 GNU make 中處於半廢狀態，原因是使用函數 T dir 和 notdir 能夠得到相同的結果。參閱*檔案名函數*。注意，‘F'變體省略所有在 dir 函數中總是輸出的結尾斜杠這裏是這些變體的列表：

`$(@D)'
: 目標檔案名中的路徑部分，結尾斜杠已經移走。如果變數`$@'的值是`dir/foo.o'，變體 `$(@D)'的值是`dir'。 如果變數`$@'的值不包含斜杠，則變體的值是`.'。

`$(@F)'
: 目標檔案名中的真正檔案名部分。如果變數`$@'的值是`dir/foo.o'，變體 `$(@F)'的值是` foo.o '。`$(@F)' 等同於 `$(notdir $@)'。

`$(*D)'
`$(*F)'
: stem（徑）中的路徑名和檔案名；在這個例子中它們的值分別為：`dir' 和 `foo' 。

`$(%D)'
`$(%F)'
: 檔案成員名中的路徑名和檔案名；這僅對採用‘*archive*(*member*)’形式的檔案成員目標有意義，並且當成員包含路徑名時才有用。參閱*檔案成員目標*。

`$(<D)'
`$(<F)'
: 第一個依賴名中的路徑名和檔案名。

`$(^D)'
`$(^F)'
: 所有依賴名中的路徑名和檔案名列表。

`$(?D)'
`$(?F)'
: 所有比目標‘新’的依賴名中的路徑名和檔案名列表。

注意，在我們討論自動變數時，我們使用了特殊格式的慣例；我們寫"the value of‘$<'", 而不是"the variable <" ；和我們寫普通變數，例如變數 objects 和 CFLAGS 一樣。我們認為這種慣例在這種情況下看起來更加自然。這並沒有其他意義，變數‘$<'的變數名為 < 和變數‘$(CFLAGS)' 實際變數名為 CFLAGS 一樣。您也可以使用‘$(<)'代替‘$<'。

10.5.4 格式匹配

目標格式是由首碼、尾碼和它們之間的通配符%組成，它們中的任一個或兩個都可以是空值。格式匹配一個檔案名只有該檔案名是以首碼開始，尾碼結束，而且兩者不重疊的條件下，才算匹配。首碼、尾碼之間的文本成為徑（stem）。當格式‘%.o’匹配檔案名‘test.o’時，徑（stem）是‘test’。格式規則中的依賴將徑（stem）替換字元%，從而得出檔案名。對於上例中，如果一個依賴為‘%.c’，則可擴展為‘test.c’。

當目標格式中不包含斜杠（實際並不是這樣），則檔案名中的路徑名首先被去除，然後，將其和格式中的首碼和尾碼相比較。在比較之後，以斜杠結尾的路徑名，將會加在根據格式規則的依賴規則產生的依賴前面。只有在尋找隱含規則時路徑名才被忽略，在應用時路徑名絕不能忽略。例如，‘e%t’和檔案名‘src/eat’匹配，stem(徑)是‘src/a’。當依賴轉化為檔案名時，

stem 中的路徑名將加在前面，stem(徑)的其餘部分替換‘%’。使用 stem（徑） ‘src/a’和依賴格式規則‘c%r’匹配得到檔案名‘src/car’。

10.5.5 萬用規則

一個格式規則的目標僅僅包含‘%’，它可以匹配任何檔案名，我們稱這些規則為萬用規則。它們非常有用，但是 make 使用它們的耗時也很多，因為 make 必須為作為目標和作為依賴列出的每一個檔都考慮這樣的規則。

假設 makefile 文件提及了文件 foo.c。為了創建該目標，make 將考慮是通過連接一個 OBJ 檔‘foo.c.o’創建，或是通過使用一步的 C 編譯連接程式從檔 foo.c.c 創建，或是編譯連接 Pascal 程式 foo.c.p 創建，以及其他的可能性等。

我們知道 make 考慮的這些可能性是很可笑的，因為 foo.c 就是一個 C 語言根源程式，不是一個可執行程式。如果 make 考慮這些可能性，它將因為這些檔諸如 foo.c.o 和 foo.c.p 等都不存在最終拒絕它們。但是這些可能性太多，所以導致 make 的運行速度極慢。

為了加快速度，我們為 make 考慮匹配萬用規則的方式設置了限制。有兩種不同類型的可以應用的限制，在您每次定義一個萬用規則時，您必須為您定義的規則在這兩種類型中選擇一種。

一種選擇是標誌該萬用規則是最終規則，即在定義時使用雙冒號定義。一個規則為最終規則時，只有在它的依賴存在時才能應用，即使依賴可以由隱含規則創建也不行。換句話說，在最終規則中沒有進一步的鏈。

例如，從 RCS 和 SCCS 檔中抽取原文件的內建的隱含規則是最終規則，則如果檔‘foo.c,v' 不存在，make 絕不會試圖從一個中間檔‘foo.c,v.o’或‘RCS/SCCS/s.foo.c,v’在創建它。 RCS 和 SCCS 檔一般都是最終原始檔案，它不能從其他任何檔重新創建，所以，make 可以記錄時間戳，但不尋找重建它們的方式。

如果您不將萬用規則標誌為最終規則，那麼它就是非最終規則。一個非最終萬用規則不能用於指定特殊類型資料的檔。如果存在其他規則（非萬用規則）的目標匹配一檔案名，則該檔案名就是指定特殊類型資料的檔案名。

例如，檔案名‘foo.c' 和格式規則 `%.c : %.y' (該規則運行 Yacc)。無論該規則是否實際使用(如果碰巧存在檔‘foo.y’，該規則將運行)，和目標匹配的事實就能足夠阻止任何非最終萬用規則在檔 foo.c 上使用。這樣，make 考慮就不試圖從檔‘foo.c.o',‘foo.c.c', ‘foo.c.p'等創建可執行的‘foo.c'。

內建的特殊偽格式規則是用來認定一些特定的檔案名，處理這些檔案名的檔時不能使用非最終萬用規則。這些偽格式規則沒有依賴和命令，它們用於其他目的時被忽略。例如，內建的隱含規則：

%.p :

存在可以保證 Pascal 根源程式如‘foo.p' 匹配特定的目標格式，從而阻止浪費時間尋找‘foo.p.o' 或‘foo.p.c'。

在尾碼規則中，為尾碼列表中的每一個有效尾碼都創建了偽格式規則，如‘%.p' 。參閱 *過時的尾碼規則*。

10.5.6 刪除隱含規則

通過定義新的具有相同目標和依賴但不同命令的規則，您可以重載內建的隱含規則（或重載您自己定義的規則）。一旦定義新的規則，內建的規則就被代替。 新規則在隱含規則次序表中的位置由您編寫規則的地方決定。

通過定義新的具有相同目標和依賴但不含命令的規則，您可以刪除內建的隱含規則。例如，下面的定義規則將刪除運行彙編編譯器的隱含規則：

%.o : %.s

10.6 定義最新類型的缺省規則

您通過編寫不含依賴的最終萬用格式規則，您可以定義最新類型的缺省規則。參閱*萬用規則*。這和其他規則基本一樣，特別之處在於它可以匹配任何目標。因此，這樣的規則的命令可用於所有沒有自己的命令的目標和依賴，以及用於那些沒有其他隱含規則可以應用的目標和依賴。

例如，在測試 makefile 時，您可能不關心原始檔案是否含有真實資料，僅僅關心它們是否存在。那麼，您可以這樣做：

```
%::
        touch $@
```

這導致所有必需的原始檔案（作為依賴）都自動創建。

您可以為沒有規則的目標以及那些沒有具體指定命令的目標定義命令。要完成上述任務，您需要為特殊目標.DEFAULT 編寫規則。這樣的規則可以在所有具體規則中用於沒有作為目標出現以及不能使用隱含規則的依賴。自然，如果您不編寫定義則沒有特殊目標.DEFAULT 的規則。

如果您使用特殊目標.DEFAULT 而不帶任何規則和命令:

```
.DEFAULT:
```

則以前為目標.DEFAULT 定義的命令被清除。如此 make 的行為和您從來沒有定義目標.DEFAULT 一樣。

如果您不需要一個目標從萬用規則和目標.DEFAULT 中得到命令，也不想為該目標執行任何命令，您可以在定義時使用空命令。參閱*使用空命令*。

您可以使用最新類型規則重載另外一個 makefile 檔的一部分內容。參閱*重載其他 **makefile** 文件*。

10.7 過時的尾碼規則

尾碼規則是定義隱含規則的過時方法。尾碼規則因為格式規則更為普遍和簡潔而被廢棄。它們在 GNU make 中得到支持是為了和早期的 makefile 檔相容。它們分為單尾碼和雙尾碼規則。

雙尾碼規則被一對尾碼定義：目標尾碼和原始檔案尾碼。它可以匹配任何檔案名以目標尾碼結尾的檔。相應的隱含依賴通過在檔案名中將目標尾碼替換為原始檔案尾碼得到。一個目標和原始檔案尾碼分別為‘.o’和‘.c’雙尾碼規則相當於格式規則`%.o : %.c'。

單尾碼規則被單尾碼定義，該尾碼是原始檔案的尾碼。它匹配任何檔案名，其相應的依賴名是將檔案名添加原始檔案尾碼得到。原始檔案尾碼為‘.c’的單尾碼規則相當於格式規則‘% : %.c’。

通過比較規則目標和定義的已知尾碼列表識別後追規則。當 make 見到一個目標尾碼是已知尾碼的規則時，該規則被認為是一個單尾碼規則。當 make 見到一個目標尾碼包含兩個已知尾碼的規則時，該規則被認為是一個雙尾碼規則。

例如，‘.o’和‘.c’都是缺省列表中的已知尾碼。所以，如果您定義一個規則，其目標是‘.c.o’，則 make 認為是一個雙尾碼規則，原始檔案尾碼是‘.c’，目標尾碼是‘.o’。這裏有一個採用過時的方法定義編譯 C 語言程式的規則：

.c.o:
	$(CC) -c $(CFLAGS) $(CPPFLAGS) -o $@ $<

尾碼規則不能有任何屬於它們自己的依賴。如果它們有依賴，它們將不是作為尾碼規則使用，而是以令人啼笑皆非的方式處理正常的檔。例如，規則：

.c.o: foo.h
	$(CC) -c $(CFLAGS) $(CPPFLAGS) -o $@ $<

告訴從依賴 foo.h 生成檔案名為‘.c.o’的檔，並不是象格式規則：

%.o: %.c foo.h
	$(CC) -c $(CFLAGS) $(CPPFLAGS) -o $@ $<

告訴從‘.c'檔生成 ‘.o' 文件‘.c'的方法：創建所有‘.o' 檔使用該格式規則，而且同時使用依賴檔‘foo.h'。

沒有命令的尾碼規則也沒有意義。它們並不沒有命令的格式規則那樣移去以前的規則（參閱*刪除隱含規則*）。他們僅僅簡單的在資料庫中加入尾碼或雙尾碼作為一個目標。

已知的尾碼是特殊目標‘.SUFFIXES’簡單的依賴名。通過為特殊目標‘.SUFFIXES’編寫規則加入更多的依賴，您可以添加您自己的已知尾碼。例如：

.SUFFIXES: .hack .win

把‘.hack' 和‘.win'添加到了尾碼列表中。

如果您希望排除缺省的已知尾碼而不是僅僅的添加尾碼，那麼您可以為特殊目標‘.SUFFIXES’編寫沒有依賴的規則。通過這種方式，可以完全排除特殊目標‘.SUFFIXES’存在的依賴。接著您可以編寫另外一個規則添加您要添加的尾碼。例如，

.SUFFIXES:		# 刪除缺省尾碼
.SUFFIXES: .c .o .h	# 定義自己的尾碼列表

標誌‘-r'或‘--no-builtin-rules'也能把缺省的尾碼列表清空。

變數 SUFFIXES 在 make 讀入任何 makefile 檔之前定義缺省的尾碼列表。您可以使用特殊目標‘.SUFFIXES’改變尾碼列表，但這不能改變變數 SUFFIXES 的值。

10.8 隱含規則搜尋演算法

這裏是 make 為一個目標‘t’搜尋隱含規則的過程。這個過程用於任何沒有命令的雙冒號規則，用於任何不含命令的普通規則的目標，以及用於任何不是其他規則目標的依賴。這個過程也能用於來自隱含規則的依賴遞迴調用該過程搜尋規則鏈。

在本演算法中不提及任何尾碼規則，因為尾碼規則在 makefile 檔讀入時轉化為了格式規則。

對於個是‘*archive*(*member*)’的檔案成員目標，下述演算法重複兩次，第一次使用整個目標名‘t’，如果第一次運行沒有發現規則，則第二次使用‘(*member*)’作為目標‘t’。

1、 1、 在‘t’中分離出路徑部分，稱為‘d’，剩下部分稱為‘n’。例如如果‘t’是‘src/foo.o’，那麼‘d’是‘src/’；‘n’是‘foo.o’。

2、 2、 建立所有目標名匹配‘t’和‘n’的格式規則列表。如果目標格式中含有斜杠，則匹配‘t’，否則，匹配‘n’。

3、 3、 如果列表中有一個規則不是萬用規則，則從列表中刪除所有非最終萬用

規則。

4、 4、 將沒有命令的規則也從列表中移走。

5、 5、 對每個列表中的格式規則：

1、1、 尋找 stem‘s’，也就是和目標格式中%匹配的‘t’或‘n’部分。

2、2、 使用 stem‘s’計算依賴名。如果目標格式不包含斜杠，則將‘d’添加在每個依賴的前面。

3、3、 測試所有的依賴是否存在或能夠創建。（如果任何檔在 makefile 中作為目標或依賴被提及，則我們說它應該存在。）如果所有依賴存在或能夠創建，或沒有依賴，則可使用該規則。

6、 6、 如果到現在還沒有發現能使用的規則，進一步試。對每一個列表中的規則：

1、1、 如果規則是最終規則，則忽略它，繼續下一條規則。

2、2、 象上述一樣計算依賴名。

3、3、 測試所有的依賴是否存在或能夠創建。

4、4、 對於不存在的依賴，按照該演算法遞迴調用查找是否能夠採用隱含規則創建。

5、5、 如果所有依賴存在或能使用隱含規則創建，則應用該規則。

7、 7、 如果沒有隱含規則，則如有用於目標‘.DEFAULT’規則，則應用該規則。在這種情況下，將目標‘.DEFAULT’的命令給與‘t’。

一旦找到可以應用的規則，對每一個匹配的目標格式（無論是‘t’或‘n’）使用 stem‘s’替換%，將得到的檔案名儲存起來直到執行命令更新目標檔‘t’。在這些命令執行以後，把每一個儲存的檔案名放入資料庫，並且標誌已經更新，其時間戳和目標檔‘t’一樣。

如果格式規則的命令為創建‘t’執行，自動變數將設置為相應的目標和依賴（參閱*自動變數*）。

11 使用 make 更新檔案檔

檔案檔是包含子檔的檔，這些子檔有各自的檔案名，一般將它們稱為成員；檔案檔和程式 ar 一塊被提及，它們的主要用途是作為連接的常式庫。

11.1 檔案成員目標

獨立的檔案檔成員可以在 make 中用作目標或依賴。按照下面的方式，您可以在檔案檔‘archive’中指定名為‘member’的成員：

archive(*member*)

這種結構僅僅在目標和依賴中使用，絕不能在命令中應用！絕大多數程式都不在命令中支援這個語法，而且也不能對檔案成員直接操作。只有程式 ar 和那些為操作檔案檔設計的程式才能這樣做。所以合法的更新檔案成員的命令一定使用 ar。例如，下述規則表明借助拷貝檔‘hack.o’在檔案‘foolib’中創建成員‘hack.o’:

```
foolib(hack.o) : hack.o
        ar cr foolib hack.o
```

實際上，幾乎所有的檔案成員目標是採用這種方式更新的，並且有一條隱含規則為您專門更新檔案成員目標。注意：如果檔案檔沒有直接存在，程式 ar 的‘c’標誌是需要的。

在相同的檔案中同時指定幾個成員，您可以在圓括號中一起寫出所有的成員名。例如：

foolib(hack.o kludge.o)

等同於：

foolib(hack.o) foolib(kludge.o)

您還可以在檔案成員引用中使用 shell 類型的通配符。參閱*在檔案名中使用通配符*。例如，‘foolib(*.o)' 擴展為在檔案‘foolib’中所有存在以‘.o’結尾的成員。也許相當於：‘foolib(hack.o) foolib(kludge.o)'。

11.2 檔案成員目標的隱含規則

對目標‘a(m)’表示名為‘m’的成員在檔案檔‘a’中。

Make 為這種目標搜尋隱含規則時，是用它另外一個的特殊特點：make 認為匹配‘(m)’的隱含規則也同時匹配‘a(m)’。

該特點導致一個特殊的規則，它的目標是‘(%)’。該規則通過將檔‘m’拷貝到檔案中更新目標‘a(m)’。例如，它通過將檔‘bar.o’拷貝到檔案‘foo.a’中更新檔案成員目標‘foo.a(bar.o)’。

如果該規則和其他規則組成鏈，功能十分強大。‘make "foo.a(bar.o)"’（注意使用雙引號是為了保護圓括號可被 shell 解釋）即使沒有 makefile 檔僅存在檔‘bar.c’就可以保證以下命令執行：

```
cc -c bar.c -o bar.o
ar r foo.a bar.o
rm -f bar.o
```

這裏 make 假設檔‘bar.o’是中間檔。參閱*隱含規則鏈*。

諸如這樣的隱含規則是使用自動變數‘$%’編寫的，參閱*自動變數*。

檔案成員名不能包含路徑名，但是在 makefile 檔中路徑名是有用的。如果您寫一個檔案成員規則‘foo.a(dir/file.o)’，make 將自動使用下述命令更新：

```
ar r foo.a dir/file.o
```

它的結果是拷貝檔‘dir/file.o’進入名為‘file.a’的檔案中。在完成這樣的任務時使用自動變數%D 和%F。

11.2.1 更新檔案的符號索引表

用作庫的檔案檔通常包含一個名為‘__.SYMDEF’ 特殊的成員，成員‘__.SYMDEF’包含由所有其他成員定義的外部符號名的索引表。在您更新其他成員後，您必須更新成員‘__.SYMDEF’，從而使成員‘__.SYMDEF’可以合適的總結其他成員。要完成成員‘__.SYMDEF’的更新需要運行 ranlib 程式：

ranlib *archivefile*

正常情況下，您應該將該命令放到檔案檔的規則中，把所有檔案檔的成員作為該規則的依賴。例如：

```
libfoo.a: libfoo.a(x.o) libfoo.a(y.o) ...
        ranlib libfoo.a
```

上述程式的結果是更新檔案成員‘x.o',‘y.o', 等等, 然後通過運行程式 ranlib 更新符號索引表表成員‘__.SYMDEF’。更新成員的規則這裏沒有列出，多數情況下，您可以省略它們，使用隱含規則把檔拷貝到檔案中，具體描述見以前的內容。

使用 GNU ar 程式時這不是必要的，因為它自動更新成員‘__.SYMDEF’。

11.3 使用檔案的危險

同時使用並行執行（-j 開關，參閱*並行執行*）和檔案應該十分小心。如果多個命令同時對相同的檔案檔操作，它們相互不知道，有可能破壞檔。將來的 make 版本可能針對該問題提供一個機制，即將所有操作相同檔案檔的命令串列化。但是現在，您必須在編寫您自己的 makefile 檔時避免該問題，或者採用其他方式，或者不使用選項-j。

11.4 檔案檔的尾碼規則

為處理檔案檔，您可以編寫一個特殊類型的尾碼規則。關於所有尾碼的擴展請參閱*過時的尾碼規則*。檔案尾碼規則在 GNU make 中已被廢棄，因為用於檔案的格式規則更加通用（參閱*檔案成員目標的隱含規則*），但是為了和其他版本的 make 相容，它們仍然被保留。

編寫用於檔案的尾碼規則，您可以簡單的編寫一個用於目標尾碼‘.a’的尾碼規則即可。例如，這裏有一個用於從 C 語言原始檔案更新檔案庫的過時尾碼規則：

```
.c.a:
        $(CC) $(CFLAGS) $(CPPFLAGS) -c $< -o $*.o
        $(AR) r $@ $*.o
        $(RM) $*.o
```

這和下面的格式規則工作完全一樣：

```
(%.o): %.c
        $(CC) $(CFLAGS) $(CPPFLAGS) -c $< -o $*.o
        $(AR) r $@ $*.o
        $(RM) $*.o
```

實際上，這僅僅是 make 看到一個以‘.a’作為尾碼的尾碼規則時，它所做的工作。任何雙尾碼規則‘.*x*.a' 被轉化為一個格式規則，該格式規則的目標格式是‘(%.o)' ，依賴格式是‘%.*x*'.。

因為您可能要使用‘.a' 作為一個檔類型的尾碼，make 也以正常方式轉換檔案尾碼規則為格式規則，參閱*過時的尾碼規則*。這樣一個雙尾碼規則‘.*x*.a' 產生兩個格式規則：‘(%.o): %.*x*' 和‘%.a: %.*x*'.

12 GNU make 的特點

這裏是 GNU make 的特點的總結，用於比較其他版本的 make。我們以 4.2 BSD 中的 make 的特點為基準。如果您要編寫一個可移植的 makefile 檔，您不要使用這裏列出的 make 的特點，也不要使用*不相容性和失去的特點*中列出的內容。

許多特點在 System V 中的 make 也存在。

- 變數 VPATH 以及它特殊的意義。參閱*在目錄中搜尋依賴*。這個特點存在於 System V 中的 make，但沒有事實證明。4.3 BSD make 也含有該特點（據說是模仿 System V 中變數 VPATFH 的特點）。
- 包含其他 makefile 文件。參閱*包含其他* ***makefile*** *文件*。允許使用一個指令包含多個檔是 GNU 的擴展。
- 通過環境，變數可以讀入和通訊，參閱*環境變數*。
- 通過變數 MAKEFLAGS 在遞迴調用 make 時可以傳遞選項。參閱*和子* ***make*** *通訊選項*。
- 在檔案引用中自動變數$% 設置為成員名。參閱*自動變數*。
- 自動變數$@, $*, $<, $%, 和 $? 有變體形式如$(@F)和$(@D)。我們把此概念化，並使用它對自動變數$^ 進行了明顯擴展。參閱*自動變數*。
- 變數引用。參閱*變數引用基礎*。
- 命令行選項‘-b'和‘-m'，接受和忽略。在 System V make 中，這些選項實際起作用。
- 即使指定選項‘-n',‘-q'或‘-t'，也能通過變數 MAKE 執行地歸調用 make 的命令。參閱*遞迴調用* ***make***。
- 在尾碼規則中支援尾碼‘.a'。參閱*用於檔案檔的尾碼規則*。這個特點在 GNU make 中幾乎不用，因為規則鏈更加通用的特點（參閱*隱含規則鏈*）允許一個格式規則用於在檔案中安裝成員已經足夠（參閱*用於檔案成員目標的隱含規則*）。
- 在命令中行排列和反斜杠-新行結合依舊保留，當命令列印時，它們出現的格式和它們在 makefile 檔中基本一樣，不同之處是去掉了初始化空白。

下面的特點被各種不同版本的 make 吸收，但哪些版本吸收了哪些特點並不十分清楚。

- 在格式規則中使用‘%’。已經有幾個不同版本的 make 使用了該特點。我們不能確認是誰發明了它，但它發展很快。參閱*定義與重新定義格式規則*。
- 規則鏈以及隱含中間檔。這個特點首先由 Stu Feldman 在它的 make 版本中實現，並用於 AT&T 第八版 Unix 研究中。後來 AT&T 貝爾實驗室的 Andrew Hume 在它的 mk 程式中應用（這裏稱為“傳遞閉合”）。我們並不清楚是從他們那裏得到這個特點或是同時我們自己開發出來的。參閱*隱含規則鏈*。
- 自動變數包含當前目標的所有依賴的列表。我們一點也不知道是誰做的。參閱*自動變數*。自動變數$+ 是變數$^的簡單擴展。
- "what if" 標誌(GNU make 中的‘-W') 是 Andrew Hume 在 mk 中發明的。參閱*代替執行命令*。
- 並行執行的概念在許多版本的 make 中存在，儘管 System V 或 BSD 並沒有實現。參閱*執行命令*。
- 使用格式替換改變變數引用來自於 SunOS 4。參閱*變數引用基礎*。在 GNU make 中，這個功能在變換語法和 SunOS 4 相容之前由函數 patsubst 提供。不知道誰是權威，因為 GNU make 使用函數 patsubst 在 SunOS 4 發佈之前。
- 在命令行前面的‘+’字元有特殊重要的意義（參閱*代替執行命令*）。這是由 *IEEE Standard 1003.2-1992* (POSIX.2)定義的。
- 使用‘+=語法為變數追加值來自於 SunOS 4 make。參閱*為變數值追加文本*。

- 語法‘*archive*(*mem1 mem2*...)'在單一檔案檔中列舉多個成員來自于 SunOS 4 make.。參閱*檔案成員目標*。
- -include 指令包括 makefile 檔，並且對於不存在的檔也不產生錯誤。該特點 with 來自於 SunOS 4 make。(但是 SunOS 4 make 在單個指令中指定多個 makefile 檔。) 該特點和 SGI make 的 sinclude 相同，

剩餘的特點是由 GNU make 發明的：

- 使用‘-v'或`--version'選項列印版本和拷貝權資訊。
- 使用‘-h' 或‘--help' 選項總結 make 的選項。
- 簡單擴展型變數。參閱*變數的兩特特色*。
- 在遞迴調用 make 時，通過變數 MAKE 自動傳遞命令行變數。參閱*遞迴調用* ***make***。
- 使用命令選項‘-C' 或‘--directory'改變路徑。參閱*選項概要*。
- 定義多行變數。參閱*定義多行變數*。
- 使用特殊目標.PHONY 聲明假想目標。AT&T 貝爾實驗室 Andrew Hume 使用不同的語法在它的 mk 程式中也實現了該功能。這似乎是並行的發現。參閱*假想目標*。
- 調用函數操作文本。參閱*用於轉換文本的函數*。
- 使用‘-o'或‘--old-file'選項假裝檔是舊檔。參閱*避免重新編譯檔*。
- 條件執行。該特點已在不同版本 make 中已經實現很長時間了；它似乎是 C 與處理程式和類似的巨集語言的自然擴展，而不是革命性的概念。參閱 ***makefile*** *檔中的條件語句*。
- 指定包含的 makefile 檔的搜尋路徑。參閱*包含其他* ***makefile*** *文件*。
- 使用環境變數指定額外的 makefile 檔。參閱*變數* ***MAKEFILES***。
- 從檔案名中去除前導斜杠`./' ，因此，‘*./file*' 和‘*file*' 是指同一個文件。
- 使用特別搜尋方法搜尋形式如‘-l*name*’的庫依賴。參閱*連接庫搜尋目錄*。
- 允許尾碼規則中的尾碼包含任何字元（參閱*過時的尾碼規則*）。在其他版本的 make 中尾碼必須以‘.’開始，並且不能包含‘/’字元。
- 包吹跟蹤當前 make 級別適用的變數 MAKWFILES 的值，參閱*遞迴調用* ***make***。
- 將任何在命令行中給出的目標放入變數 MAKECMDGOALS。參閱*指定最終目標的參數*。
- 指定靜態格式規則。參閱*靜態格式規則*。
- 提供選擇性 vpath 搜尋。參閱*在目錄中搜尋依賴*。
- 提供可計算的變數引用。參閱*變數引用基礎*。
- 更新 makefile 檔。參閱重建 makefile 文件。System V make 中有非常非常有限的來自於該功能的形式，它用於為 make 檢查 SCCS 檔。
- 各種新建的隱含規則。參閱*隱含規則目錄*。
- 內建變數`MAKE_VERSION' 給出 make 的版本號。

13 不相容性和失去的特點

其他版本的 make 程式也有部分特點在 GNU make 中沒有實現。POSIX.2 標準 (*IEEE Standard 1003.2-1992*)規定不需要這些特點。

- ‘*file*((*entry*))' 形式的目標代表一個檔案檔的成員 file。選擇該成員不使用檔案名，而是通過一個定義連接符號 enty 的 OBJ 檔。該特點沒有被 GNU make 吸收因為該非標準元件將為 make 加入檔案檔符號表的內部知識。參閱*更新檔案符號索引表*。
- 在尾碼規則中以字元‘~’結尾的尾碼在 System V make 中有特別的含義；它們指和檔

案名中沒有‘~’的檔通訊的 SCCS 檔。例如，尾碼規則‘.c~.o'將從名為‘s.*n*.c'的 SCCS 文件中抽取文件‘*n*.o'。為了完全覆蓋，需要這種整系列的尾碼規則，參閱*過時的尾碼規則*。在 GNU make 中，這種整系列的尾碼規則由勇於從 SCCS 檔抽取的兩個格式規則掌管，它們可和通用的規則結合成規則鏈，參閱*隱含規則鏈*。

- 在 System V make 中，字串‘$$@'又奇特的含義，在含有多個規則的依賴中，它代表正在處理的特殊目標。這在 GNU make 沒有定義，因為字串‘$$'代表一個平常的字元‘$'。使用靜態格式規則可以實現該功能的一部分（參閱*靜態格式規則*）。System V make 中的規則：

$(targets): $$@.o lib.a

在 GNU make 中可以用靜態格式規則代替：

$(targets): %: %.o lib.a

- 在 System V 和 4.3 BSD make 中， 通過 VPATH 搜尋（參閱*為依賴搜尋目錄*）發現的檔，它們的檔案名改變後加入到命令字串中。我們認為使用自動變數更簡單明瞭，所以不引進該特點。
- 在一些 Unix make 中，自動變數$*出現在規則的依賴中有令人驚奇的特殊特點：擴展為該規則的目標全名。我們不能明白 Unix make 在心中對這是怎樣考慮的，它和正常的變數$*定義完全不同。
- 在一些 Unix make 中，隱含規則搜尋（參閱*使用隱含規則*）明顯是為所有目標做的，而不僅僅為那些沒有命令的目標。這意味著：

foo.o:

cc -c foo.c

在 Unix make 有直覺知道‘foo.o' 依靠‘foo.c'。我們認為這樣的用法易導致混亂。Make 中依賴的屬性已經定義好（至少對於 GNU make 是這樣），再做這樣的事情不合規矩。

- GNU make 不包含任何編譯以及與處理 EFL 程式的隱含規則。如果我們聽說誰使用 EFL，我們樂意把它們加入。
- 在 SVR4 make 中，一條尾碼規則可以不含命令，它的處理方式和它含有空命令的處理方式一樣（參閱*使用空命令*）。例如：

.c.a:

將重載內建的尾碼規則‘.c.a' 。我們覺得對沒有命令的規則簡單的為目標添加依賴更為簡潔。上述例子和在 GNU make 中下例的行為相同。

.c.a: ;

- 一些版本的 make 調用 shell 使用‘-e'標誌，而不是‘-k'標誌 (參閱*測試程式編譯*)。標誌‘-e'告訴 shell 一旦程式運行返回非零狀態就立即退出。我們認為根據每一命令行是否需要需要特殊處理直接寫入命令中更為清楚。

14 makefile 文件慣例

本章描述為 GNU make 編寫 makefile 檔的慣例。使用 Automake 將幫助您按照這些慣例編寫 makefile 檔。

14.1 makefile 檔的通用慣例

任何 makefile 檔都應該包含這行：

SHELL = /bin/sh

避免在系統中變數 SHELL 可能繼承環境中值的麻煩。（在 GNU make 中這從來不是問題。）

不同的 make 程式有不同的尾碼列表和隱含規則，這有可能造成混亂或錯誤的行為。因此最好的辦法是設置尾碼列表，在該列表中，僅僅包含您在特定 makefile 檔中使用的尾碼。

例如：
.SUFFIXES:
.SUFFIXES: .c .o
第一行清除了尾碼列表，第二行定義了在該 makefile 中可能被隱含規則使用的尾碼。

不要假設‘.' 是命令執行的路徑。當您在創建程式過程中，需要運行僅是您套裝程式中一部分的程式時，請確認如果該程式是要創建程式的一部分使用‘./’，如果該程式是源代碼中不變的部分使用‘$(srcdir)’。沒有這些首碼，僅僅在當前路徑下搜索。

建造目錄（build directory ）‘./’和源代碼目錄(**source directory**) ‘$(srcdir)’的區別是很重要的，因為用戶可以在‘configure’中使用‘--srcdir’選項建造一個單獨的目錄。下面的規則：
foo.1 : foo.man sedscript
sed -e sedscript foo.man > foo.1
如果創建的目錄不是源代碼目錄將失敗，因為檔‘foo.man’和‘sedscript’在源代碼目錄下。

在使用 GNU make 時，依靠變數‘VPATH’搜尋原始檔案在單個從屬性檔存在情況下可以很好地工作，因為 make 中自動變數‘$<’中含有原始檔案的存在路徑。（許多版本的 make 僅在隱含規則中設值變數‘$<’。）例如這樣的 makefile 檔目標：
foo.o : bar.c
$(CC) -I. -I$(srcdir) $(CFLAGS) -c bar.c -o foo.o
將被替換為：
foo.o : bar.c
$(CC) -I. -I$(srcdir) $(CFLAGS) -c $< -o $@
這是為了保證變數‘VPATH’正確的工作。目標含有多個依賴時，使用名了的‘$(srcdir)’是最容易的保證該規則很好工作的方法。例如，以上例子中的目標‘foo.l’最好寫為：
foo.1 : foo.man sedscript
sed -e $(srcdir)/sedscript $(srcdir)/foo.man > $@

GNU 的分類中通常包含一些不是原始檔案的檔——例如，‘Info’文件、從 Autoconf, Automake, Bison 或 Flex 中輸出的檔等。這些檔在原始檔案目錄下，它們也應該在原始檔案目錄下，不應該在建造目錄下。因此 makefile 規則應在原始檔案目錄下更新它們。

然而，如果一個檔沒有在分類中出現，makefile 檔不應把它們放到原始檔案目錄下，因為按照通常情況創建一個程式，不應該以任何方式更改原始檔案目錄。

試圖建造的創建和安裝目標，至少（以及它們的子目標）可在並行的 make 中正確的工作。

14.2 makefile 文件的工具

編寫在 shell sh 中運行而不在 csh 中運行的 makefile 檔命令(以及 shell 的腳本，例如‘configure’)，不要使用任何 ksh 或 bash 的特殊特點。

用於創建和安裝的‘configure’腳本和 Makefile 規則除下面所列出工具外不應該直接使用其他的任何工具：
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

壓縮程式 gzip 可在 dist 規則中使用。

堅持使用用於這些程式的通用選項，例如，不要使用‘mkdir -p'，它可能比較方便，但是其他大多數系統卻不支援它。

避免在 makefile 中創造符號連接是非常不錯的注意，因為一些系統不支援這種做法。

用於創建和安裝的 Makefile 規則可以使用編譯器以及相關的程式，但應該通過 make 變數使用它們，這樣可以方便用戶使用別的進行替換。這裏有按照我們的觀念編寫一些程式：
ar bison cc flex install ld ldconfig lex
make makeinfo ranlib texi2dvi yacc
請使用下述 make 變數運行這些程式:
$(AR) $(BISON) $(CC) $(FLEX) $(INSTALL) $(LD) $(LDCONFIG) $(LEX)
$(MAKE) $(MAKEINFO) $(RANLIB) $(TEXI2DVI) $(YACC)

使用 ranlib 或 ldconfig，您應該確定如果系統中不存在要使用的程式不會引起任何副作用。安排忽略這些命令產生的錯誤，並且列印資訊告訴用戶該命令運行失敗並不意味著存在問題。（Autoconf‘AC_PROG_RANLIB'宏可在這方面幫助您。）如果您使用符號連接，對於不支援符號連接的系統您應該有一個低效率運行方案。

附加的工具也可通過 make 變數使用：

chgrp chmod chown mknod

它在 makefile 中（或腳本中），您知道包含這些工具的特定系統中它都可以很好的工作。

14.3 指定命令的變數

Makefile 檔應該為重載的特定命令、選項等提供變數。

特別在您運行大部分工具時都應該應用變數，如果您要使用程式 Bison, 名為 BISON 的變數它的缺省值設置為：‘BISON = bison’，在您需要使用程式 Bison 時，您可以使用$(BISON)引用。

檔管理器工具如 ln, rm, mv 等等，不必要使用這種方式引用，因為用戶不可能使用別的程式替換它們。

每一個程式變數應該和用於向該程式提供選項的選項變數一起提供。在程式名變數後添加‘FLAGS'表示向該程式提供選項的選項變數--例如, BISONFLAGS。(名為 CFLAGS 的變數向 C 編譯器提供選項, 名為 YFLAGS 的變數向 yacc 提供選項，名為 LFLAGS 的變數向 lex 提供選項等是這個規則例外，但因為它們是標準所以我們保留它們。) 在任何進行預處理的編譯命令中使用變數 CPPFLAGS ，在任何進行連接的編譯命令中使用變數 LDFLAGS 和直接使用程式 ld 一樣。

對於 C 編譯器在編譯特定檔時必須使用的選項，不應包含在變數 CFLAGS 中，因為用戶希望他們能夠自由的指定變數 CFLAGS。 要獨立於變數 CFLAGS 安排向 C 編譯器傳遞這些必要的選項, 可以將這些選項寫入編譯命令行中或隱含規則的定義中，如下例：

```
CFLAGS = -g
ALL_CFLAGS = -I. $(CFLAGS)
.c.o:
        $(CC) -c $(CPPFLAGS) $(ALL_CFLAGS) $<
```

變數 CFLAGS 中包括選項‘-g’，因為它對於一些編譯並不是必需的，您可以認為它是缺省推薦的選項。如果資料包創建使用 GCC 作為編譯器，則變數 CFLAGS 中包括選項‘-o’,而且以它為缺省值。

將變數 CFLAGS 放到編譯命令的最後，在包含編譯選項其他變數的後邊，因此用戶可以使用變數 CFLAGS 對其他變數進行重載。

每次調用 C 編譯器都用到變數 CFLAGS ，無論進行編譯或連接都一樣。

任何 Makefile 檔都定義變數 INSTALL，變數 INSTALL 是將檔安裝到系統中的基本命令。

任何 Makefile 檔都定義變數 INSTALL_PROGRAM 和 INSTALL_DATA，(它們的缺省值都是$(INSTALL)。) 在實際安裝程式時，不論可執行程式或非可執行程式，一般都使用它們作為命令。下面是使用這些變數的例子：

```
$(INSTALL_PROGRAM) foo $(bindir)/foo
$(INSTALL_DATA) libfoo.a $(libdir)/libfoo.a
```

您可以隨意將變數 DESTDIR 預先設置為目標檔案名。這樣做允許安裝程式創建隨後在實際目標檔系統中安裝檔的快照。不要再 makefile 檔中設置變數 DESTDIR，也不要包含在安裝檔中。用變數 DERSTDIR 改變上述例子：

$(INSTALL_PROGRAM) foo $(DESTDIR)$(bindir)/foo
$(INSTALL_DATA) libfoo.a $(DESTDIR)$(libdir)/libfoo.a

在安裝命令中一般使用檔案名而不是路徑名作為第二個參數。對每一個安裝檔都使用單獨的命令。

14.4 安裝路徑變數

安裝目錄經常以變數命名，所以在非標準地方安裝也很容易，這些變數的標準名字將在下面介紹。安裝目錄依據標準檔系統佈局，變數的變體已經在 SVR4, 4.4BSD, Linux, Ultrix v4, 以及其他現代作業系統中使用。

以下兩個變數設置安裝檔的根目錄，所有的其他安裝目錄都是它們其中一個的子目錄，沒有任何檔可以直接安裝在這兩個根目錄下。

`prefix'

首碼是用於構造以下列舉變數的缺省值。變數 prefix 缺省值是‘/usr/local'。建造完整的 GNU 系統時，變數 prefix 的缺省值是空值，‘/usr' 是符號連接符‘/'。(如果您使用 Autoconf，應將它寫為‘@prefix@'。)使用不同于創建程式時變數 prefix 的值運行‘make install'，不會重新編譯程序。

`exec_prefix'

首碼是用於構造以下列舉變數的缺省值。變數 exec_prefix 缺省值是$(prefix). (如果您使用 Autoconf，應將它寫為`@exec_prefix@'。) 一般情況下。變數$(exec_prefix) 用於存放包含機器特定檔的目錄，(例如可執行檔和常式庫)，變數$(prefix) 直接存放其他目錄。使用不同于創建程式時變數 exec_prefix 的值運行‘make install'，不會重新編譯程序。

可執行程式安裝在以下目錄中：

`bindir'

這個目錄下用於安裝用戶可以運行的可執行程式。其正常的值是‘/usr/local/bin'，但是使用時應將它寫為‘$(exec_prefix)/bin'。 (如果您使用 Autoconf, 應將它寫為‘@bindir@'。)

`sbindir'

這個目錄下用於安裝從 shell 中調用執行的可執行程式。它僅僅對系統管理員有作用。它的正常的值是‘/usr/local/sbin'，但是使用時應將它寫為‘$(exec_prefix)/sbin'。 (如果您使用 Autoconf, 應將它寫為‘@sbindir@'。)

`libexecdir'

這個目錄下用於安裝其他程式調用的可執行程式。其正常的值是‘/usr/local/libexec'，但是使用時應將它寫為‘$(exec_prefix)/libexec'。(如果您使用 Autoconf, 應將它寫為‘@libexecdir@'。)

程式執行時使用的資料檔案可分為兩類：

- 程式可以正常更改的檔和不能正常更改的檔(雖然用戶可以編輯其中的一部分檔)。
- 體系結構無關檔，指這些檔可被所有機器共用；體系相關檔，指僅僅可以被相同類型機器、作業系統共用的檔；其他是永遠不能被兩個機器共用的檔。

這可產生六種不同的可能性。我們極力反對使用體系相關的檔，當然 OBJ 檔和庫檔除外。使用其他體系無關的資料檔案更加簡潔，並且，這樣做也不是很難。

所以，這裏有 Makefile 變數用於指定路徑：

`datadir'

這個目錄下用於安裝唯讀型體系無關資料檔案。其正常的值是‘/usr/local/share'，但是使用時應將它寫為‘$(prefix)/share'。(如果您使用 Autoconf, 應將它寫為‘@datadir@'。) 作為例外，參閱下述的變數‘$(infodir)'和‘$(includedir)'。

`sysconfdir'

這個目錄下用於安裝從屬於單個機器的唯讀資料檔案，這些檔是：用於配置主機的檔。郵件服務、網路配置檔，‘/etc/passwd'，等等都屬於這裏的檔。所有該目錄下的檔都是平常的 ASCII 文字檔案。其正常的值是‘/usr/local/etc', 但是使用時應將它寫為‘$(prefix)/etc'. (如果您使用 Autoconf, 應將它寫為‘@sysconfdir@'.) 不要在這裏安裝可執行檔（它們可能屬於‘$(libexecdir)'或‘$(sbindir)'）。也不要在這裏安裝那些在使用時要更改的檔（這些程式用於改變系統拒絕的配置）。它們可能屬於‘$(localstatedir)'。

`sharedstatedir'

這個目錄下用於安裝程式運行中要發生變化的體系無關資料檔案。其正常的值是‘/usr/local/com'，但是使用時應將它寫為‘$(prefix)/com'。 (如果您使用 Autoconf, 應將它寫為‘@sharedstatedir@'。)

`localstatedir'

這個目錄下用於安裝程式運行中要發生變化的資料檔案。但他們屬於特定的機器。用戶永遠不需要在該目錄下更改檔配置套裝程式選項；將這些配置資訊放在分離的檔中，這些檔將放入‘$(datadir)' 或‘$(sysconfdir)' 中，‘$(localstatedir)' 正常的值是‘/usr/local/var'，但是使用時應將它寫為‘$(prefix)/var'。 (如果您使用 Autoconf, 應將它寫為‘@localstatedir@'。)

`libdir'

這個目錄下用於存放 OBJ 檔和庫的 OBJ 代碼。不要在這裏安裝可執行檔，它們可能應屬於‘$(libexecdir)'。變數 libdir 正常的值是‘/usr/local/lib'，但是使用時應將它寫為‘$(exec_prefix)/lib'。(如果您使用 Autoconf, 應將它寫為 ‘@libdir@'。)

`infodir'

這個目錄下用於安裝套裝軟體的 Info 檔。缺省情況下其值是‘/usr/local/info'，但是使用時應將它寫為‘$(prefix)/info'. (如果您使用 Autoconf, 應將它寫為‘@infodir@'.)

`lispdir'

這個目錄下用於安裝套裝軟體的 Emacs Lisp 檔。缺省情況下其值是‘/usr/local/share/emacs/site-lisp'，但是使用時應將它寫為‘$(prefix)/share/emacs/site-lisp'。如果您使用 Autoconf, 應將它寫為‘@lispdir@'。為了保證‘@lispdir@'工作，您需要將以下幾行加入到您的‘configure.in'文件中：

```
lispdir='${datadir}/emacs/site-lisp'
AC_SUBST(lispdir)
```

`includedir'

這個目錄下用於安裝用戶程式中 C‘#include'預處理指令包含的頭檔。其正常的值是‘/usr/local/include'，但是使用時應將它寫為‘$(prefix)/include'。 (如果您使用 Autoconf, 應將它寫為‘@includedir@'。) 除 GCC 外的大多數編譯器不在目錄‘/usr/local/include' 搜尋頭檔，因此這種安裝方式僅僅適用於 GCC。有時，這也不是問題，因為一部分庫文件僅僅依靠 GCC 才能工作。但也有一部分庫檔依靠其他編譯器，它們將它們的頭檔安裝到兩個地方，一個由變數 includedir 指定，另一個由變數 oldincludedir 指定。

`oldincludedir'

這個目錄下用於安裝‘#include'的頭檔，這些頭檔用於除 GCC 外的其他 C 語言編譯器。其正常的值是‘/usr/include'。(如果您使用 Autoconf, 應將它寫為‘@oldincludedir@'。) Makefile 命令變數 oldincludedir 的值是否為空，如果是空值，它們不在試圖使用它，它們還刪除第二次安裝的頭檔。一個套裝軟體在該目錄下替換已經存在的頭檔，除非頭檔來源於同一個套裝軟體。例如，如果您的套裝軟體 Foo 提供一個頭檔‘foo.h'，則它在變數 oldincludedir 指定的目錄下安裝的條件是 (1) 這裏沒有投檔‘foo.h' 或 (2) 來源於套裝軟體 Foo 的頭文件‘foo.h'已經在該目錄下存在。要檢查頭檔‘foo.h'是否來自於套裝軟體 Foo，將一個 magic 字串放到檔中--作為命令的一部分--然後使用正則規則（grep）查找該字串。

Unix 風格的幫助檔安裝在以下目錄中：

`mandir'
安裝該套裝軟體的頂層幫助（如果有）目錄。其正常的值是‘/usr/local/man'，但是使用時應將它寫為‘$(prefix)/man'。 (如果您使用 Autoconf, 應將它寫為‘@mandir@'。)

`man1dir'
這個目錄下用於安裝第一層幫助。其正常的值是‘$(mandir)/man1'。

`man2dir'
這個目錄下用於安裝第一層幫助。其正常的值是‘$(mandir)/man2'。

`...'
不要將任何 **GNU** 軟體的主要文檔作為幫助頁。應該編寫使用手冊。幫助頁僅僅是為了人們在 **Unix** 上方便運行 **GNU** 軟體，它是附屬的運行程式。

`manext'
檔案名表示對已安裝的幫助頁的擴展。它包含一定的週期，後跟適當的數位，正常為‘1’。

`man1ext'
檔案名表示對已安裝的幫助頁第一部分的擴展。

`man2ext'
檔案名表示對已安裝的幫助頁第二部分的擴展。

`...'
使用這些檔案名代替`manext'。如果該套裝軟體的幫助頁需要安裝使用手冊的多個章節。

最後您應該設置一下變數：

`srcdir'
這個目錄下用於安裝要編譯的原文件。該變數正常的值由 shell 腳本 configure 插入。(如果您使用 Autoconf, 應將它寫為‘srcdir = @srcdir@'.)

例如：

```
# 用於安裝路徑的普通首碼。
# 注意：該路經在您開始安裝時必須存在
prefix = /usr/local
exec_prefix = $(prefix)
# 這裏放置`gcc'命令調用的可執行檔。
bindir = $(exec_prefix)/bin
# 這裏放置編譯器使用的目錄。
libexecdir = $(exec_prefix)/libexec
#這裏放置 Info 檔。
infodir = $(prefix)/info
```

如果您的程式要在標準用戶指定的目錄中安裝大量的檔，將該程式的檔放入到特意指定的子目錄中是很有必要的。如果您要這樣做，您應該寫安裝規則創建這些子目錄。

不要期望用戶在上述列舉的變數值中包括這些子目錄，對於安裝目錄使用一套變數名的辦法使用戶能夠對於不同的 GNU 套裝軟體指定精確的值，為了使這種做法有用，所有的套裝軟體必須設計為當用戶使用時它們能夠聰明的工作。

14.5 用戶標準目標

所有的 GNU 程式中，在 makefile 中都有下列目標：

`all'

編譯整個程式。這應該是缺省的目標。該目標不必重建文檔檔，Info 檔已正常情況下應該包括在各個發佈的檔中，DVI 檔只有在明確請求情況下才重建。缺省時，make 規則編譯和連接使用選項‘-g'，所以程式調試只是象徵性的。對於不介意缺少幫助的用戶如果他們希望將可執行程式和幫助分開，可以從中剝離出可執行程式。

`install'

編譯程序並將可執行程式、庫檔等拷貝到為實際使用保留的檔案名下。如果是證實程式是否適合安裝的簡單測試，則該目標應該運行該測試程式。不要在安裝時剝離可執行程式，魔鬼很可能關心那些使用 install-strip 目標來剝離可執行程式的人。如果這是可行的，編寫的 install 目標規則不應該更改程式建造的目錄下的任何東西，僅僅提供‘make all'一切都能完成。這是為了方便用戶命名和在其他系統安裝建造程式，如果要安裝程式的目錄不存在，該命令應能創建所有這些目錄，這包括變數 prefix 和 exec_prefix 特別指定的目錄和所有必要的子目錄。完成該任務的方法是借助下面描述的目標 installdirs。在所有安裝幫助頁的命令前使用‘-'使 make 能夠忽略這些命令產生的錯誤，這可以確保在沒有 Unix 幫助頁的系統上安裝該套裝軟體時能夠順利進行。安裝 Info 檔的方法是使用變數$(INSTALL_DATA)將 Info 檔拷貝到變數‘$(infodir)'中（參閱*指定命令的變數*），如果 install-info 程式存在則運行它。 install-info 是一個編輯 Info‘dir'檔的程式，它可以為 Info 檔添加或更新功能表；它是 Texinfo 套裝軟體的一部分。這裏有一個安裝 Info 檔的例子：

```
$(DESTDIR)$(infodir)/foo.info: foo.info
        $(POST_INSTALL)
# 可能在‘.’下有新的檔，在 srcdir 中沒有。
        -if test -f foo.info; then d=.; \
          else d=$(srcdir); fi; \
        $(INSTALL_DATA) $$d/foo.info $(DESTDIR)$@; \
#如果 install-info 程式存在則運行它。
# 使用‘if'代替在命令行前的‘-'
# 這樣，我們可以注意到運行 install-info 產生的真正錯誤。
# 我們使用‘$(SHELL) -c' 是因為在一些 shell 中
# 遇到未知的命令不會運行失敗。
        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
```

在編寫 install 目標時，您必須把所有的命令歸位三類：正常的命令、 安裝前命令和安裝後命令。參閱*安裝命令分類*。

`uninstall'

刪除所有安裝的檔--有‘install’目標拷貝的檔。該規則不應更改編譯產生的目錄，僅僅刪除安裝檔的目錄。反安裝命令象安裝命令一樣分為三類，參閱*安裝命令分類*。

`install-strip'

和目標 install 類似，但在安裝時僅僅剝離出可執行檔。 在許多情況下，該目標的定義非常簡單：

```
install-strip:
        $(MAKE) INSTALL_PROGRAM='$(INSTALL_PROGRAM) -s' \
                install
```

正常情況下我們不推薦剝離可執行程式進行安裝，只有您確信這些程式不會產生問題時才能這樣。剝離安裝一個實際執行的可執行檔同時保存那些在這種場合存在 BUG 的可執行檔是顯而易見的。

`clean'

刪除所有當前目錄下的檔，這些檔正常情況下是那些‘建立程式’創建的檔。不要刪除那些記錄配置的檔，同時也應該保留那些‘建立程式’能夠修改的檔，正常情況下要刪除的那些檔不包括這些檔，因為發佈檔是和這些檔一起創建的。如果‘.dvi'檔不是檔發佈檔的一部分，則使用目標‘clean’將同時刪除‘.dvi'文件。

`distclean'

刪除所有當前目錄下的檔，這些檔正常情況下是那些‘建立程式’或‘配置程式’創建的檔。如果您不解包根源程式，‘建立程式’不會創建任何其他文件，‘make distclean'將僅在文件發佈文件中留下原有的檔。

`mostlyclean'

和目標‘clean'類似，但是避免刪除人們正常情況下不編譯的檔。例如，用於 GCC 的目標‘mostlyclean 不刪除檔‘libgcc.a'，因為在絕大多數情況下它都不需要重新編譯。

`maintainer-clean'

幾乎在當前目錄下刪除所有能夠使用該 makefile 檔可以重建的檔。使用該目標刪除的檔包括使用目標 distclean,刪除的檔加上從 Bison 產生的 C 語言原始檔案和標誌列表、 Info 檔等等。我們說“幾乎所有檔”的原因是運行命令‘make maintainer-clean'不刪除腳本‘configure'，即使腳本‘configure'可以使用 Makefile 檔創建。更確切地說，運行‘make maintainer-clean'不刪除為了運行腳本‘configure'以及開始建立程式的涉及的所有檔。這是運行‘make maintainer-clean'刪除所有能夠重新創建檔時唯一不能刪除的一類檔。目標‘maintainer-clean'由該套裝軟體的養護程式使用，不能被普通用戶使用。您可以使用特殊的工具重建被目標‘make maintainer-clean'刪除的檔。因為這些檔正常情況下包含在發佈的檔中，我們並不關心它們是否容易重建。如果您發現您需要對全部發佈的檔重新解包，您不能責怪我們。要幫助 make 的用戶意識到這一點，用於目標 maintainer-clean 應以以下兩行為開始：

@echo‘該命令僅僅用於養護程式；’

@echo‘它刪除的所有檔都能使用特殊工具重建。’

`TAGS'

更新該程式的標誌表。

`info'

產生必要的 Info 檔。最好的方法是編寫象下面規則：

info: foo.info

foo.info: foo.texi chap1.texi chap2.texi

$(MAKEINFO) $(srcdir)/foo.texi

您必須在 makefile 檔中定以變數 MAKEINFO。它將運行 makeinfo 程式，該程式是發佈程式中 Texinfo 的一部分。正常情況下，一個 GNU 發佈程式和 Info 檔一起創建，這意味著 Info 檔存在於原始檔案的目錄下。當用戶建造一個套裝軟體，一般情況下，make 不更新 Info 檔，因為它們已經更新到最新了。

`dvi'

創建 DVI 文件用於更新 Texinfo 文檔。例如：

dvi: foo.dvi

foo.dvi: foo.texi chap1.texi chap2.texi

$(TEXI2DVI) $(srcdir)/foo.texi

您必須在 makefile 檔中定義變數 TEXI2DVI。它將運行程式 texi2dvi，該程式是發佈的 Texinfo 一部分。要麼僅僅編寫依靠檔，要麼允許 GNU make 提供命令，二者必選其一。

`dist'

為程式創建一個 tar 檔。創建 tar 檔可以將其中的檔案名以子目錄名開始，這些子目錄名可以是用於發佈的套裝軟體名。另外，這些檔案名中也可以包含版本號，例如，發佈的 GCC 1.40 版的 tar 檔解包的子目錄為‘gcc-1.40'。最方便的方法是創建合適的子目錄名，如使用 in 或 cp 等作為子目錄，在它們的下面安裝適當的檔，然後把 tar 檔解包到這些子目錄中。使用 gzip 壓縮這些 tar 檔，例如，實際的 GCC 1.40 版的發佈檔叫‘gcc-1.40.tar.gz'。目標 dist 明顯的依靠所有的發佈檔中不是原始檔案的檔，所以你應確保發佈中的這些檔已經更新。參閱 ***GNU*** *標準編碼*中*創建發佈檔*。

`check'

執行自我檢查。用戶應該在運行測試之前，應該先建立程式，但不必安裝這些程式；您應該編寫一個自我測試程式，在程式已建立但沒有安裝時執行。

以下目標建議使用習慣名，對於各種程式它們很有用：

installcheck

執行自我檢查。用戶應該在運行測試之前，應該先建立、安裝這些程式。您不因該假設‘$(bindir)'在搜尋路徑中。

installdirs

添加名為‘installdirs'目標對於創建檔要安裝的目錄以及它們的父目錄十分有用。腳本‘mkinstalldirs'是專為這樣處理方便而編寫的；您可以在 Texinfo 套裝軟體中找到它，您可以象這樣使用規則：

```
# 確保所有安裝目錄(例如 $(bindir))
# 都實際存在，如果沒有則創建它們。
installdirs: mkinstalldirs
        $(srcdir)/mkinstalldirs $(bindir) $(datadir) \
                                $(libdir) $(infodir) \
                                $(mandir)
```

該規則並不更改編譯時創建的目錄，它僅僅創建安裝目錄。

14.6 安裝命令分類

編寫已安裝目標，您必須將所有命令分為三類：正常的命令、安裝前命令和安裝後命令。

正常情況下，命令把檔移動到合適的地方，並設置它們的模式。它們不會改變任何檔，僅僅把它們從套裝軟體中完整地抽取出來。

安裝前命令和安裝後命令可能更改一些文件，如，它們編輯配置檔後資料庫檔。

安裝前命令在正常命令之前執行，安裝後命令在正常命令執行後執行。

安裝後命令最普通的用途是運行 install-info 程式。 這種工作不能由正常命令完成，因為它更改了一個檔（Info 目錄），該檔不能全部、單獨從套裝軟體中安裝。它是一個安裝後命令，因為它需要在正常命令安裝套裝軟體中的 Info 檔後才能執行。

許多程式不需要安裝前命令，但是我們提供這個特點，以便在需要時可以使用。

要將安裝規則的命令分為這三類，應在命令中間插入 **category lines**（分類行）。 分類行指定了下面敍述的命令的類別。

分類行包含一個 Tab、一個特殊的 make 變數引用，以及行結尾的隨機注釋。您可以使用三個變數，每一個變數對應一個類別；變數名指定了類別。分類行不能出現在普通的執行檔中，因為這些 make 變數被由正常的定義（您也不應在 makefile 檔中定義）。

這裏有三種分類行，後面的注釋解釋了它的含義：

```
$(PRE_INSTALL)        # 以下是安裝前命令
$(POST_INSTALL)       # 以下是安裝後命令
$(NORMAL_INSTALL)   # 以下是正常命令
```

如果在安裝規則開始您沒有使用分類行，則在第一個分類行出現之前的所有命令都是正常命令。如果您沒有使用任何分類行，則所有命令都是正常命令。

這是反安裝的分類行

```
$(PRE_UNINSTALL)        #以下是反安裝前命令
$(POST_UNINSTALL)       #以下是反安裝後命令
$(NORMAL_UNINSTALL)   #以下是正常命令
```

反安裝前命令的典型用法是從 Info 目錄刪除全部內容。

如果目標 install 或 uninstall 有依賴作為安裝程式的副程式，那麼您應該使用分類行先啟動每一個依賴的命令，再使用分類行啟動主目標的命令。無論哪一個依賴實際執行，這種方式都能保證每一條命令都放置到了正確的分類中。

安裝前命令和安裝後命令除了對於下述命令外，不能運行其他程式：

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

按照這種方式區分命令的原因是為了創建二進位套裝軟體。典型的二進位套裝軟體包括所有可執行檔、必須安裝的其他檔以及它自己的安裝檔——所以二進位套裝軟體不需要運行任何正常命令。但是安裝二進位套裝軟體需要執行安裝前命令和安裝後命令。

建造二進位套裝軟體的程式通過抽取安裝前命令和安裝後命令工作。這裏有一個抽取安裝前命令的方法：

```
make -n install -o all \
        PRE_INSTALL=pre-install \
        POST_INSTALL=post-install \
        NORMAL_INSTALL=normal-install \
  | gawk -f pre-install.awk
```

這裏檔'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}
```

安裝前命令的結果檔是象安裝二進位套裝軟體的一部分 shell 腳本一樣執行。

15 快速參考

這是對指令、文本操作函數以及 GNU make 能夠理解的變數等的匯總。對於其他方面的總結參閱*特殊的內建目標名*，*隱含規則目錄*，*選項概要*。

這裏是 GNU make 是別的指令的總結：

define *variable*
endef
　　定義多行遞迴調用擴展型變數。參閱*定義固定次序的命令*。
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 檔中的條件擴展，參閱 ***makefile** 檔中的條件語句*。

include *file*
-include *file*
sinclude *file*
包含其他 makefile 文件，參閱*包含其他 makefile 文件*。

override *variable* = *value*
override *variable* := *value*
override *variable* += *value*
override *variable* ?= *value*
override define *variable*
endef
定義變數、對以前的定義重載、以及對在命令行中定義的變數重載。參閱 ***override** 指令*。

export
告訴 make 缺省向子過程輸出所有變數，參閱*與子 **make** 通訊的變數*。

export *variable*
export *variable* = *value*
export *variable* := *value*
export *variable* += *value*
export *variable* ?= *value*
unexport *variable*
告訴 make 是否向子過程輸出一個特殊的變數。參業*與子 **make** 通訊的變數*。

vpath *pattern path*
制定搜尋匹配‘%’格式的檔的路徑。參閱 ***vpath** 指令*。

vpath *pattern*
去除以前為‘*pattern*’指定的所有搜尋路徑。

vpath
去除以前用 vpath 指令指定的所有搜尋路徑。

這裏是操作文本函數的總結，參閱*文本轉換函數*：

$(subst *from*,*to*,*text*)
在‘text’中用‘to’代替‘from’，參閱*字串替換與分析函數*。

$(patsubst *pattern*,*replacement*,*text*)
在‘text’中用‘*replacement*’代替匹配‘*pattern*’字，參閱*字串替換與分析函數*。

$(strip *string*)
從字串中移去多餘的空格。參閱*字串替換與分析函數*。

$(findstring *find*,*text*)
確定‘*find*’在‘*text*’中的位置。參閱*字串替換與分析函數*。

$(filter *pattern...*,*text*)
在‘text’中選擇匹配‘*pattern*’的字。參閱*字串替換與分析函數*。

$(filter-out *pattern...*,*text*)
在‘text’中選擇不匹配‘*pattern*’的字。參閱*字串替換與分析函數*。

$(sort *list*)
將‘list’中的字按字母順序排序，並刪除重複的字。參閱*字串替換與分析函數*。

$(dir *names...*)
從檔案名中抽取路徑名。參閱*檔案名函數*。

$(notdir *names*...)
　　從檔案名中抽取路徑部分。參閱*檔案名函數*。
$(suffix *names*...)
　　從檔案名中抽取非路徑部分。參閱*檔案名函數*。
$(basename *names*...)
　　從檔案名中抽取基本檔案名。參閱*檔案名函數*。
$(addsuffix *suffix*,*names*...)
　　為‘names’中的每個字添加尾碼。參閱*檔案名函數*。
$(addprefix *prefix*,*names*...)
　　為‘names’中的每個字添加首碼。參閱*檔案名函數*。
$(join *list1*,*list2*)
　　連接兩個並行的字列表。參閱*檔案名函數*。
$(word *n*,*text*)
　　從‘text’中抽取第 n 個字。參閱*檔案名函數*。
$(words *text*)
　　計算‘text’中字的數目。參閱*檔案名函數*。
$(wordlist *s*,*e*,*text*)
　　返回‘text’中 s 到 e 之間的字。參閱*檔案名函數*。
$(firstword *names*...)
　　在‘names…’中的第一個字。參閱*檔案名函數*。
$(wildcard *pattern*...)
　　尋找匹配 shell 檔案名格式的檔案名。參閱 ***wildcard** 函數*。
$(error *text*...)
　　該函數執行時，make 產生資訊為‘text’的致命錯誤。參閱*控制 **make** 的函數*。
$(warning *text*...)
　　該函數執行時，make 產生資訊為‘text’的警告。參閱*控制 **make** 的函數*。
$(shell *command*)
　　執行 shell 命令並返回它的輸出。參閱*函數 **shell***。
$(origin *variable*)
　　返回 make 變數‘*variable*’的定義資訊。參閱*函數 **origin***。
$(foreach *var*,*words*,*text*)
　　將列表列表 words 中的每一個字對應後接 var 中的每一個字，將結果放在 text 中。參閱*函數 **foreach***。
$(call *var*,*param*,...)
　　使用對$(1), $(2)...對變數計算變數 *var* ，變數$(1), $(2)...分別代替參數 *param* 第一個、第二個…的值。參閱*函數 **call***。

這裏是對自動變數的總結，完整的描述參閱*自動變數*。

$@
　　目標檔案名。
$%
　　當目標是檔案成員時，表示目標成員名。
$<
　　第一個依賴名。
$?
　　所有比目標‘新’的依賴的名字，名字之間用空格隔開。對於為檔案成員的依賴，只能使用命名的成員。參閱*使用 **make** 更新檔案檔*。
$^
$+
　　所有依賴的名字，名字之間用空格隔開。對於為檔案成員的依賴，只能使用命名的成員。參閱*使用 **make** 更新檔案檔*。變數 $^ 省略了重複的依賴，而變數 $+ 則按照原來次序保留重複項，
$*

和隱含規則匹配的 stem(徑)。參閱*格式匹配*。

$(@D)
$(@F)
: 變數$@.中的路徑部分和檔案名部分。

$(*D)
$(*F)
: 變數$*中的路徑部分和檔案名部分。

$(%D)
$(%F)
: 變數$%中的路徑部分和檔案名部分。

$(<D)
$(<F)
: 變數$<中的路徑部分和檔案名部分。

$(^D)
$(^F)
: 變數$^中的路徑部分和檔案名部分。

$(+D)
$(+F)
: 變數$+中的路徑部分和檔案名部分。

$(?D)
$(?F)
: 變數$?中的路徑部分和檔案名部分。

以下是 GNU make 使用變數：

MAKEFILES
: 每次調用 make 要讀入的 Makefiles 文件。參閱*變數* ***MAKEFILES***。

VPATH
: 對在當前目錄下不能找到的檔搜索的路徑。參閱 ***VPATH:*** *所有依賴的搜尋路徑*。

SHELL
: 系統缺省命令解釋程式名，通常是`/bin/sh'。您可以在 makefile 檔中設值變數 SHELL 改變運行程式使用的 shell。參閱*執行命令*。

MAKESHELL
: 改變量僅用於 MS-DOS，make 使用的命令解釋程式名，該變數的值比變數 SHELL 的值優先。參閱*執行命令*。

MAKE
: 調用的 make 名。在命令行中使用該變數有特殊的意義。參閱變數 MAKE 的工作方式。

MAKELEVEL
: 遞迴調用的層數(子 makes)。參閱*與子* ***make*** *通訊的變數*。

MAKEFLAGS
: 向 make 提供標誌。您可以在環境或 makefile 檔中使用該變數設置標誌。參閱*與子* ***make*** *通訊的變數*。在命令行中不能直接使用該變數，應為它的內容不能在 shell 中正確引用，但總是允許遞迴調用 make 時通過環境傳遞給子 make。

MAKECMDGOALS
: 該目標是在命令行中提供給 make 的。設置該變數對 make 的行為沒有任何影響。參閱*特別目標的參數*。

CURDIR
: 設置當前工作目錄的路徑名，參閱*遞迴調用* ***make***。

SUFFIXES
: 在讀入任何 makefile 檔之前的尾碼列表。

.LIBPATTERNS
: 定義 make 搜尋的庫檔案名，以及搜尋次序。參閱*連接庫搜尋目錄*。

16 make 產生的錯誤

這裏是您可以看到的由 make 產生絕大多數普通錯誤列表，以及它們的含義和修正它們資訊。

有時 make 產生的錯誤不是致命的，如一般在命令腳本行前面存在首碼的情況下，和在命令行使用選向‘-k’的情況下產生的錯誤幾乎都不是致命錯誤。使用字串***作首碼將產生致命的錯誤。

錯誤資訊前面都使用首碼，首碼的內容是產生錯誤的程式名或 makefile 檔中存在錯誤的檔案名和包含該錯誤的行的行號和。

在下述的錯誤列表中，省略了普通的首碼：

`[*foo*] Error *NN*'

`[*foo*] *signal description*'

這些錯誤並不是真的 make 的錯誤。它們意味著 make 調用的程式返回非零狀態值，錯誤碼（Error *NN*），這種情況 make 解釋為失敗，或非正常方式退出（一些類型信號），參閱*命令錯誤*。如果資訊中沒有附加***，則是子過程失敗，但在 makefile 檔中的這條規則有特殊首碼，因此 make 忽略該錯誤。

`missing separator. Stop.'

`missing separator (did you mean TAB instead of 8 spaces?). Stop.'

這意味著 make 在讀取命令行時遇到不能理解的內容。GNU make 檢查各種分隔符號(:, =, 字元 TAB，等) 從而幫助確定它在命令行中遇到了什麼類型的錯誤。這意味著，make 不能發現一個合法的分隔符號。出現該資訊的最可能的原因是您（或許您的編輯器，絕大部分是 ms-windows 的編輯器）在命令行縮進使用了空格代替了字元 Tab。這種情況下，make 將使用上述的第二種形式產生錯誤資訊。一定切記，任何命令行都以字元 **Tab** 開始，八個空格也不算數。參閱*規則的語法*。

`commands commence before first target. Stop.'

`missing rule before commands. Stop.'

這意味著在 makefile 中似乎以命令行開始：以 Tab 字元開始，但不是一個合法的命令行（例如，一個變數的賦值）。任何命令行必須和一定的目標相聯繫。產生第二種的錯誤資訊是一行的第一個非空白字元為分號，make 對此的解釋是您遺漏了規則中的"target: prerequisite" 部分，參閱*規則的語法*。

`No rule to make target `*xxx*'.'

`No rule to make target `*xxx*', needed by `*yyy*'.'

這意味著 make 決定必須建立一個目標，但卻不能在 makefile 檔中發現任何用於創建該目標的指令，包括具體規則和隱含規則。如果您希望創建該目標，您需要另外為改目標編寫規則。其他關於該問題產生原因可能是 makefile 文件是草稿（如檔案名錯）或破壞了原始檔案樹（一個檔不能按照計畫重建，僅僅由於一個依賴的問題）。

`No targets specified and no makefile found. Stop.'

`No targets. Stop.'

前者意味著您沒有為命令行提供要創建的目標，make 不能讀入任何 makefile 檔。後者意味著一些 makefile 檔被找到，但沒有包含缺省目標以及命令行等。GNU make 在這種情況下無事可做。參閱*指定* ***makefile*** *檔的參數*。

`Makefile `*xxx*' was not found.'

`Included makefile `*xxx*' was not found.'

在命令行中指定一個 makefile 檔（前者）或包含的 makefile 檔（後者）沒有找到。

`warning: overriding commands for target `*xxx*"

`warning: ignoring old commands for target `*xxx*"

GNU make 允許命令在一個規則中只能對一個命令使用一次(雙冒號規則除外)。如果您為一個目標指定一個命令，而該命令在目標定義是已經定義過，這種警告就會產生；第二個資訊表明後來設置的命令將改寫以前對該命令的設置。參閱*具有多條規*

則的目標。

`Circular *xxx* <- *yyy* dependency dropped.'

這意味著 make 檢測到一個相互依靠一個迴圈：在跟蹤目標 xxx 的依賴 yyy 時發現，依賴 yyy 的依賴中一個又以 xxx 為依賴。

`Recursive variable `*xxx*' references itself (eventually). Stop.'

這意味著您定義一個正常（遞迴調用性）make 變數 xxx，當它擴展時，它將引用它自身。無論對於簡單擴展型變數(:=)或追加定義（+=），這也都是不能允許的，參閱*使用變數*。

`Unterminated variable reference. Stop.'

這意味著您在變數引用或函數調用時忘記寫右括弧。

`insufficient arguments to function `*xxx*'. Stop.'

這意味著您在調用函數是您密友提供需要數目的參數。關於函數參數的詳細描述參閱*文本轉換函數*。

`missing target pattern. Stop.'

`multiple target patterns. Stop.'

`target pattern contains no `%'. Stop.'

這些錯誤資訊是畸形的靜態格式規則引起的。第一條意味著在規則的目標部分沒有規則，第二條意味著在目標部分有多個規則，第三條意味著沒有包含格式符%。參閱*靜態格式規則語法*。

`warning: -jN forced in submake: disabling jobserver mode.'

該條警告和下條警告是在 make 檢測到在能與子 make 通訊的系統中包含並行處理的錯誤（參閱*與子 **make** 通訊選項*）。該警告資訊是如果遞迴調用一個 make 過程，而且還使用了‘-jn’選項（這裏 n 大於 1）。這種情況很可能發生，例如，如果您設置環境變數 MAKE 為‘make –j2’。這種情況下，子 make 不能與其他 make 過程通訊， 而且還簡單假裝它由兩個任務。

`warning: jobserver unavailable: using -j1. Add `+' to parent make rule.'

為了保證 make 過程之間通訊，父層 make 將傳遞資訊給子 make。這可能導致問題，因為子過程有可能不是實際的一個 make，而父過程僅僅認為子過程是一個 make 而將所有資訊傳遞給子過程。父過程是採用正常的演算法決定這些的(參閱*變數 **MAKE** 的工作方式*)。如果 makefile 檔構建了這樣的父過程，它並不知道子過程是否為 make，那麼，子過程將拒收那些沒有用的資訊。這種情況下，子過程就會產生該警告資訊，然後按照它內建的次序方式進行處理。

17 複雜的 **makfile** 檔例子

這是一個用於 GNU tar 程式的 makefile 檔；這是一個中等複雜的 makefile 檔。

因為‘all’是第一個目標，所以它是缺省目標。該 makefile 檔一個有趣的地方是‘testpad.h'是由 testpad 程式創建的原始檔案，而且該程式自身由‘testpad.c'編譯得到的。

如果您鍵入‘make'或`make all'，則 make 創建名為‘tar'可執行檔, 提供遠端存取磁帶的進程‘rmt'，和名為‘tar.info'的 Info 文件。

如果您鍵入‘make install'，則 make 不但創建‘tar',‘rmt',和‘tar.info'，而且安裝它們。

如果您鍵入‘make clean', 則 make 刪除所有‘.o'檔，以及‘tar',‘rmt',‘testpad', ‘testpad.h',和‘core’文件。

如果您鍵入‘make distclean', 則 make 不僅刪除‘make clean'刪除的所有檔，而且包括檔‘TAGS', ‘Makefile', 和‘config.status' 文件。(雖然不明顯，但該 makefile (和‘config.status')是用戶用 configure 程式產生的，該程式是由發佈的 tar 檔提供，但這裏不進行說明。)

如果您鍵入‘make realclean', 則 make 刪除‘make distclean '刪除的所有檔，而且包括由‘tar.texinfo'產生的 Info 檔。

另外，目標 shar 和 dist 創造了發佈檔的核心。

```
# 自動從 makefile.in 產生
# 用於 GNU tar 程式的 Unix Makefile
```

```
# Copyright (C) 1991 Free Software Foundation, Inc.

# 本程式是自由軟體；在遵照 GNU 條款的情況下
# 您可以重新發佈它或更改它
# 普通公眾許可證 ...
...
...

SHELL = /bin/sh

#### 啟動系統配置部分 ####

srcdir = .

# 如果您使用 gcc, 您應該在運行
# 和它一起創建的固定包含的腳本程式以及
# 使用-traditional 選項運行 gcc 中間選擇其一。
# 另外的 ioctl 調用在一些系統上不能正確編譯
CC = gcc -O
YACC = bison -y
INSTALL = /usr/local/bin/install -c
INSTALLDATA = /usr/local/bin/install -c -m 644

# 您應該在 DEFS 中添加的內容：
# -DSTDC_HEADERS            如果您有標準 C 的頭檔和
#                           庫文件。
# -DPOSIX                   如果您有 POSIX.1 的頭文件和
#                           庫文件。
# -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
#                           字串和記憶體控制函數
#                           和頭文件， 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             如果您的信號控制器
#                           返回 int,非 void.
# -DNO_MTIO                 如果您缺少 sys/mtio.h
#                           (magtape ioctls).
# -DNO_REMOTE               如果您沒有一個遠端 shell
#                           或 rexec.
# -DUSE_REXEC               對遠端磁帶使用 rexec
#                           操作代替
#                           分支 rsh 或 remsh.
# -DVPRINTF_MISSING         如果您缺少函數 vprintf
#                           (但是有_doprnt).
# -DDOPRNT_MISSING          如果您缺少函數 _doprnt.
#                           同樣需要定義
#                           -DVPRINTF_MISSING.
```

```
# -DFTIME_MISSING          如果您缺少系統調用 ftime
# -DSTRSTR_MISSING         如果您缺少函數 strstr。
# -DVALLOC_MISSING          如果您缺少函數 valloc。
# -DMKDIR_MISSING          如果您缺少系統調用 mkdir 和
#                         rmdir。
# -DRENAME_MISSING         如果您缺少系統調用 rename。
# -DFTRUNCATE_MISSING     如果您缺少系統調用 ftruncate。
#
# -DV7                     在 Unix 版本 7 (沒有
#                         進行長期測試).
# -DEMUL_OPEN3              如果您缺少 3-參數版本
#                         的 open, 並想通過您有的系統調用
#                         仿真它。
# -DNO_OPEN3                如果您缺少 3-參數版本的 open
#                         並要禁止 tar –k 選項
#                         代替仿真 open.
# -DXENIX                   如果您有 sys/inode.h
#                         並需要它包含 94

DEFS =    -DSIGTYPE=int -DDIRENT -DSTRSTR_MISSING \
          -DVPRINTF_MISSING -DBSD42
# 設置為 rtapelib.o ，除非使它為空時
# 您定義了 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
# 每一個安裝程式的首碼。
#正常為空或`g'。
binprefix =

# 安裝 tar 的路徑
bindir = $(prefix)/bin

# 安裝 info 檔的路徑.
infodir = $(prefix)/info

#### 系統配置結束部分 ####

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 個變換/減少衝突。

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
```

註腳

(1)

為 MS-DOS 和 MS-Windows 編譯的 GNU Make 和將首碼定義為 DJGPP 樹體系的根的行為相同。

(2)

在 MS-DOS 上，當前工作目錄的值是 **global**，因此改變它將影響這些系統中隨後的命令行。

(3)

texi2dvi 使用 TeX 進行真正的格式化工作. TeX 沒有和 Texinfo 一塊發佈。

本文檔使用版本 1.54 的 texi2html 翻譯器于 2000 年 7 月 19 日產生。

名詞翻譯對照表

archive	檔案
archive member targets	檔案成員目標
arguments of functions	函數參數
automatic variables	自動變數
backslash (\)	反斜杠

basename	基本檔案名
binary packages	二進位包
compatibility	相容性
data base	資料庫
default directries	缺省目錄
default goal	缺省最終目標
defining variables verbatim	定義多行變數
directive	指令
dummy pattern rule	偽格式規則
echoing of commands	命令回顯
editor	編輯器
empty commands	空命令
empty targets	空目標
environment	環境
explicit rule	具體規則
file name functions	檔案名函數
file name suffix	檔案名尾碼
flags	標誌
flavors of variables	變數的特色
functions	函數
goal	最終目標
implicit rule	隱含規則
incompatibilities	不相容性
intermediate files	中間檔
match-anything rule	萬用規則
options	選項
parallel execution	並行執行
pattern rule	格式規則
phony targets	假想目標
prefix	首碼
prerequisite	依賴
recompilation	重新編譯
rule	規則
shell command	shell 命令
static pattern rule	靜態格式規則
stem	徑
sub-make	子 make
subdirectories	子目錄
suffix	尾碼
suffix rule	尾碼規則
switches	開關
target	目標
value	值
variable	變數
wildcard	通配符
word	字