跟我一起寫Makefile:書寫規則

書寫規則

規則包含兩個部分，一個是依賴關係，一個是生成目標的方法。

在Makefile中，規則的順序是很重要的，因為，Makefile中只應該有一個最終目標，其它的目標都是被這個目標所連帶出來的，所以一定要讓make知道你的最終目標是什麼。一般來說，定義在Makefile中的目標可能會有很多，但是第一條規則中的目標將被確立為最終的目標。如果第一條規則中的目標有很多個，那麼，第一個目標會成為最終的目標。make所完成的也就是這個目標。

好了，還是讓我們來看一看如何書寫規則。

規則舉例

```
foo.o : foo.c defs.h       # foo模块
        cc -c -g foo.c
```

看到這個例子，各位應該不是很陌生了，前面也已說過，foo.o是我們的目標，foo.c和defs.h是目標所依賴的源文件，而只有一個命令“cc -c -g foo.c”（以Tab鍵開頭）。這個規則告訴我們兩件事：

1. 文件的依賴關係，foo.o依賴於foo.c和defs.h的文件，如果foo.c和defs.h的文件日期要比foo.o文件日期要新，或是foo.o不存在，那麼依賴關係發生。
2. 生成或更新foo.o文件，就是那個cc命令。它說明了如何生成foo.o這個文件。（當然，foo.c文件include了defs.h文件）

規則的語法

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

或是這樣：

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

targets是文件名，以空格分開，可以使用通配符。一般來說，我們的目標基本上是一個文件，但也有可能是多個文件。

command是命令行，如果其不與“targets:prerequisites”在一行，那麼，必須以[Tab鍵]開頭，如果和prerequisites在一行，那麼可以用分號做為分隔。（見上）

prerequisites也就是目標所依賴的文件（或依賴目標）。如果其中的某個文件要比目標文件要新，那麼，目標就被認為是“過時的”，被認為是需要重生成的。這個在前面已經講過了。

如果命令太長，你可以使用反斜框（‘\’）作為換行符。make對一行上有多少個字符沒有限制。規則告訴make兩件事，文件的依賴關係和如何生成目標文件。

一般來說，make會以UNIX的標準Shell，也就是/bin/sh來執行命令。

在規則中使用通配符

如果我們想定義一系列比較類似的文件，我們很自然地就想起使用通配符。make支援三個通配符：“*”，“?”和“~”。這是和Unix的B-Shell是相同的。

波浪號（“~”）字符在文件名中也有比較特殊的用途。如果是“~/test”，這就表示當前用戶的$HOME目錄下的test目錄。而 “~hchen/test”則表示用戶hchen的宿主目錄下的test目錄。（這些都是Unix下的小知識了，make也支援）而在Windows或是 MS-DOS下，用戶沒有宿主目錄，那麼波浪號所指的目錄則根據環境變量“HOME”而定。

通配符代替了你一系列的文件，如“*.c”表示所有後綴為c的文件。一個需要我們注意的是，如果我們的文件名中有通配符，如：“*”，那麼可以用轉義字符“\”，如“*”來表示真實的“*”字符，而不是任意長度的字符串。

好吧，還是先來看幾個例子吧：

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

其實在這個clean:後面可以加上你想做的一些事情，如果你想看到在編譯完后執行cat看看main.c的源代碼，你可以在加上cat這個命令，例子如下：

```
clean :
     cat main.c
      rm -f *.o
```

其結果你試一下就知道的。 上面這個例子我不多說了，這是作業系統Shell所支援的通配符。這是在命令中的通配符。

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

上面這個例子說明了通配符也可以在我們的規則中，目標print依賴於所有的[.c]文件。其中的“$?”是一個自動化變量，我會在後面給你講述。

```
objects = *.o
```

上面這個例子，表示了通配符同樣可以用在變量中。並不是說[*.o]會展開， 確切的，objects的值就是“*.o”。Makefile中的變量其實就是C/C++中的宏。如果你要讓通配符在變量中展開，也就是讓objects的值是所有[.o]的文件名的集合，那麼，你可以這樣：

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

另給一個變量使用通配符的例子：

a. 列出一確定文件夾中的所有”.c"文件

```
objects := $(wildcard *.c)
```

b. 列出（a）中所有文件對應的".o"文件，在（c）中我們可以看到它是由make自動編譯出的。

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

c. 由（a）（b）兩步，可寫出編譯並鏈接所有“.c"和”.o"文件

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

這種用法由關鍵字“wildcard”，“patsubst”指出，關於Makefile的關鍵字，我們將在後面討論。

文件搜尋

在一些大的工程中，有大量的源文件，我們通常的做法是把這許多的源文件分類，並存放在不同的目錄中。所以，當make需要去找尋文件的依賴關係時，你可以在文件前加上路徑，但最好的方法是把一個路徑告訴make，讓make自動去找。

Makefile文件中的特殊變量“VPATH”就是完成這個功能的，如果沒有指明這個變量，make只會在當前的目錄中去找尋依賴文件和目標文件。如果定義了這個變量，那麼，make就會在當前目錄找不到的情況下，到所指定的目錄中去找尋文件了。

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

上面的的定義指定兩個目錄，“src”和“../headers”，make會按照這個順序進行搜索。目錄由“冒號”分隔。（當然，當前目錄永遠是最高優先搜索的地方）

另一個設置文件搜索路徑的方法是使用make的“vpath”關鍵字（注意，它是全小寫的），這不是變量，這是一個make的關鍵字，這和上面提到的那個VPATH變量很類似，但是它更為靈活。它可以指定不同的文件在不同的搜索目錄中。這是一個很靈活的功能。它的使用方法有三種：

1、vpath <pattern> <directories>

為符合模式<pattern>的文件指定搜索目錄<directories>。

2、vpath <pattern>

清除符合模式<pattern>的文件的搜索目錄。

3、vpath

清除所有已被設置好了的文件搜索目錄。

vpath使用方法中的<pattern>需要包含“%”字符。“%”的意思是匹配零或若干字符，（需引用“%”，使用“\%"）例如，“%.h”表示所有以 “.h”結尾的文件。<pattern>指定了要搜索的文件集，而<directories>則指定了< pattern>的文件集的搜索的目錄。例如：

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

該語句表示，要求make在“../headers”目錄下搜索所有以“.h”結尾的文件。（如果某文件在當前目錄沒有找到的話）

我們可以連續地使用vpath語句，以指定不同搜索策略。如果連續的vpath語句中出現了相同的<pattern>，或是被重複了的<pattern>，那麼，make會按照vpath語句的先後順序來執行搜索。如：

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

其表示“.c”結尾的文件，先在“foo”目錄，然後是“blish”，最後是“bar”目錄。

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

而上面的語句則表示“.c”結尾的文件，先在“foo”目錄，然後是“bar”目錄，最後才是“blish”目錄。

偽目標

最早先的一個例子中，我們提到過一個“clean”的目標，這是一個“偽目標”，

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

正像我們前面例子中的“clean”一樣，既然我們生成了許多文件編譯文件，我們也應該提供一個清除它們的“目標”以備完整地重編譯而用。（以“make clean”來使用該目標）

因為，我們並不生成“clean”這個文件。“偽目標”並不是一個文件，只是一個標籤，由於“偽目標”不是文

件，所以make無法生成它的依賴關係和決定它是否要執行。我們只有通過顯式地指明這個“目標”才能讓其生效。當然，“偽目標”的取名不能和文件名重名，不然其就失去了“偽目標”的意義了。

當然，為了避免和文件重名的這種情況，我們可以使用一個特殊的標記“.PHONY”來顯式地指明一個目標是“偽目標”，向make說明，不管是否有這個文件，這個目標就是“偽目標”。

```
.PHONY : clean
```

只要有這個聲明，不管是否有“clean”文件，要運行“clean”這個目標，只有“make clean”這樣。於是整個過程可以這樣寫：

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

偽目標一般沒有依賴的文件。但是，我們也可以為偽目標指定所依賴的文件。偽目標同樣可以作為“默認目標”，只要將其放在第一個。一個示例就是，如果你的Makefile需要一口氣生成若干個可執行文件，但你只想簡單地敲一個make完事，並且，所有的目標文件都寫在一個Makefile中，那麼你可以使用“偽目標”這個特性：

```
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
```

我們知道，Makefile中的第一個目標會被作為其默認目標。我們聲明了一個“all”的偽目標，其依賴於其它三個目標。由於默認目標的特性是，總是被執行的，但由於“all”又是一個偽目標，偽目標只是一個標籤不會生成文件，所以不會有“all”文件產生。於是，其它三個目標的規則總是會被決議。也就達到了我們一口氣生成多個目標的目的。“.PHONY : all”聲明了“all”這個目標為“偽目標”。（注：這裏的顯式“.PHONY : all" 不寫的話一般情況也可以正確的執行，這樣 make 可通過隱式規則推導出，“all" 是一個偽目標，執行 make 不會生成 ”all" 文件，而執行後面的多個目標。建議：顯式寫出是一個好習慣。）

隨便提一句，從上面的例子我們可以看出，目標也可以成為依賴。所以，偽目標同樣也可成為依賴。看下面的例子：

舉一個簡單的Makefile如下：

```
kenny:
	@touch chardi
chardi:
	@echo Chardi
```

我期望每次執行make chardi的結果都會印出"Chardi"。
原則上沒什麼問題，但當我做了下面的動作：

```
$ make kenny$ make chardimake: `chardi' is up to date.
```

在make過程中產生了一個與target "chardi"相同名稱的檔案，造成make誤以為target "chardi"已經被執行過而略過她。此時，.PHONY就派上用場了：

```
.PHONY: chardi
kenny:
	@touch chardi
chardi:
	@echo Chardi
```

加了PHONY使得make略過了搜尋target的隱含規則，因此不管有沒有檔案chardi在該目錄中，都不會影響target "chardi"的執行。

```
.PHONY : cleanall cleanobj cleandiff

cleanall : cleanobj cleandiff
        rm program

cleanobj :
        rm *.o

cleandiff :
        rm *.diff
```

“make cleanall”將清除所有要被清除的文件。“cleanobj”和“cleandiff”這兩個偽目標有點像“子程式”的意思。我們可以輸入“make cleanall”和“make cleanobj”和“make cleandiff”命令來達到清除不同種類文件的目的。

多目標

Makefile的規則中的目標可以不止一個，其支援多目標，有可能我們的多個目標同時依賴於一個文件，並且其生成的命令大體類似。於是我們就能把其合併起來。當然，多個目標的生成規則的執行命令不是同一個，這可能會給我們帶來麻煩，不過好在我們可以使用一個自動化變量“$@”（關於自動化變量，將在後面講述），這個變量表示着目前規則中所有的目標的集合，這樣說可能很抽象，還是看一個例子吧。

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

上述規則等價于：

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

其中，-$(subst output,,$@)中的“$”表示執行一個Makefile的函數，函數名為subst，後面的為參數。關於函數，將在後面講述。這裏的這個函數是替換字符串的意思，“$@”表示目標的集合，就像一個數組，“$@”依次取出目標，並執于命令。

靜態模式

靜態模式可以更加容易地定義多目標的規則，可以讓我們的規則變得更加的有彈性和靈活。我們還是先來看一下語法：

```
<targets ...>: <target-pattern>: <prereq-patterns ...>
```

```
        <commands>
        ...
```

targets定義了一系列的目標文件，可以有通配符。是目標的一個集合。

target-pattern是指明了targets的模式，也就是的目標集模式。

prereq-patterns是目標的依賴模式，它對target-pattern形成的模式再進行一次依賴目標的定義。

這樣描述這三個東西，可能還是沒有說清楚，還是舉個例子來說明一下吧。如果我們的<target-pattern>定義成“%. o”，意思是我們的<target>集合中都是以“.o”結尾的，而如果我們的<prereq-patterns>定義成 “%.c”，意思是對<target-pattern>所形成的目標集進行二次定義，其計算方法是，取<target- pattern>模式中的“%”（也就是去掉了[.o]這個結尾），併為其加上[.c]這個結尾，形成的新集合。

所以，我們的“目標模式”或是“依賴模式”中都應該有“%”這個字符，如果你的文件名中有“%”那麼你可以使用反斜杠“\”進行轉義，來標明真實的“%”字符。

看一個例子：

```
objects = foo.o bar.o

all: $(objects)


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

上面的例子中，指明了我們的目標從$object中獲取，“%.o”表明要所有以“.o”結尾的目標，也就是“foo.o bar.o”，也就是變量$object集合的模式，而依賴模式“%.c”則取模式“%.o”的“%”，也就是“foo bar”，併為其加下“.c”的後綴，於是，我們的依賴目標就是“foo.c bar.c”。而命令中的“$<”和“$@”則是自動化變量，“$<”表示所有的依賴目標集（也就是 “foo.c bar.c”），“$@”表示目標集（也就是“foo.o bar.o”）。於是，上面的規則展開后等價于下面的規則：

```
foo.o : foo.c
        $(CC) -c $(CFLAGS) foo.c -o foo.o
bar.o : bar.c
        $(CC) -c $(CFLAGS) bar.c -o bar.o
```

試想，如果我們的“%.o”有幾百個，那種我們只要用這種很簡單的“靜態模式規則”就可以寫完一堆規則，實在是太有效率了。“靜態模式規則”的用法很靈活，如果用得好，那會一個很強大的功能。再看一個例子：

```
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 $<
```

$(filter %.o,$(files))表示調用Makefile的filter函數，過濾“$files”集，只要其中模式為“%.o”的內容。其的它內容，我就不用多說了吧。這個例子展示了Makefile中更大的彈性。

自動生成依賴性

在Makefile中，我們的依賴關係可能會需要包含一系列的頭文件，比如，如果我們的main.c中有一句“#include "defs.h"”，那麼我們的依賴關係應該是：

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

但是，如果是一個比較大型的工程，你必需清楚哪些C文件包含了哪些頭文件，並且，你在加入或刪除頭文件時，也需要小心地修改Makefile，這是一個很沒有維護性的工作。為了避免這種繁重而又容易出錯的事情，我們可以使用C/C++編譯的一個功能。大多數的C/C++編譯器都支援一個“-M”的選項，即自動找尋源文件中包含的頭文件，並生成一個依賴關係。例如，如果我們執行下面的命令：

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

其輸出是：

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

於是由編譯器自動生成的依賴關係，這樣一來，你就不必再手動書寫若干文件的依賴關係，而由編譯器自動生成了。需要提醒一句的是，如果你使用GNU的C/C++編譯器，你得用“-MM”參數，不然，“-M”參數會把一些標準庫的頭文件也包含進來。

gcc -M main.c的輸出是：

```
main.o: main.c defs.h /usr/include/stdio.h /usr/include/features.h \
    /usr/include/sys/cdefs.h /usr/include/gnu/stubs.h \
    /usr/lib/gcc-lib/i486-suse-linux/2.95.3/include/stddef.h \
    /usr/include/bits/types.h /usr/include/bits/pthreadtypes.h \
    /usr/include/bits/sched.h /usr/include/libio.h \
    /usr/include/_G_config.h /usr/include/wchar.h \
    /usr/include/bits/wchar.h /usr/include/gconv.h \
    /usr/lib/gcc-lib/i486-suse-linux/2.95.3/include/stdarg.h \
    /usr/include/bits/stdio_lim.h
```

gcc -MM main.c的輸出則是：

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

那麼，編譯器的這個功能如何與我們的Makefile聯繫在一起呢。因為這樣一來，我們的Makefile也要根據這些源文件重新生成，讓 Makefile自己依賴於源文件？這個功能並不現實，不過我們可以有其它手段來迂迴地實現這一功能。GNU組織建議把編譯器為每一個源文件的自動生成的依賴關係放到一個文件中，為每一個“name.c”的文件都生成一個“name.d”的Makefile文件，[.d]文件中就存放對應[.c]文件的依賴關係。

於是，我們可以寫出[.c]文件和[.d]文件的依賴關係，並讓make自動更新或自成[.d]文件，並把其包含在我們的主Makefile中，這樣，我們就可以自動化地生成每個文件的依賴關係了。

這裏，我們給出了一個模式規則來產生[.d]文件：

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

這個規則的意思是，所有的[.d]文件依賴於[.c]文件，“rm -f $@”的意思是刪除所有的目標，也就是[.d]文件，第二行的意思是，為每個依賴文件“$<”，也就是[.c]文件生成依賴文件，“$@”表示模式“%.d”文件，如果有一個C文件是name.c，那麼“%”就是 “name”，“$$$$”意為一個隨機編號，第二行生成的文件有可能是“name.d.12345”，第三行使用sed命令做了一個替換，關於sed命令的用法請參看相關的使用文件。第四行就是刪除臨時文件。

總而言之，這個模式要做的事就是在編譯器生成的依賴關係中加入[.d]文件的依賴，即把依賴關係：

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

轉成：

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

於是，我們的[.d]文件也會自動更新了，並會自動生成了，當然，你還可以在這個[.d]文件中加入的不只是依賴關係，包括生成的命令也可一併加入，讓每個[.d]文件都包含一個完賴的規則。一旦我們完成這個工作，接下來，我們就要把這些自動生成的規則放進我們的主Makefile中。我們可以使用Makefile的“include”命令，來引入別的Makefile文件（前面講過），例如：

```
sources = foo.c bar.c

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

上述語句中的“$(sources:.c=.d)”中的“.c=.d”的意思是做一個替換，把變量$(sources)所有[.c]的字串都替換成[.d]，關於這個“替換”的內容，在後面我會有更為詳細的講述。當然，你得注意次序，因為include是按次來載入文件，最先載入的[.d]文件中的目標會成為默認目標。