跟我一起寫**Makefile:**隱含規則

出自**Ubuntu**中文

*導航 |概述 + MakeFile介紹 + 書寫規則 + 書寫命令 + 使用變量 + 使用條件判斷 |使用函數 + make運行 + 隱含規則 + 使用make更新函數庫文件 + 後序 |

目錄

隱含規則

在我們使用Makefile時，有一些我們會經常使用，而且使用頻率非常高的東西，比如，我們編譯C/C++的源程序為中間目標文件（Unix下是[.o]文件，Windows下是[.obj]文件）。本章講述的就是一些在Makefile中

的“隱含的”，早先約定了的，不需要我們再寫出來的規則。

“隱含規則”也就是一種慣例，make會按照這種“慣例”心照不喧地來運行，那怕我們的Makefile中沒有書寫這樣的規則。例如，把[.c]文件編譯成[.o]文件這一規則，你根本就不用寫出來，make會自動推導出這種規則，並生成我們需要的[.o]文件。

“隱含規則”會使用一些我們系統變量，我們可以改變這些系統變量的值來定製隱含規則的運行時的參數。如系統變量“CFLAGS”可以控制編譯時的編譯器參數。

我們還可以通過“模式規則”的方式寫下自己的隱含規則。用“後綴規則”來定義隱含規則會有許多的限制。使用“模式規則”會更回得智能和清楚，但“後綴規則”可以用來保證我們Makefile的兼容性。我們了解了“隱含規則”，可以讓其為我們更好的服務，也會讓我們知道一些“約定俗成”了的東西，而不至於使得我們在運行Makefile時出現一些我們覺得莫名其妙的東西。當然，任何事物都是矛盾的，水能載舟，亦可覆舟，所以，有時候“隱含規則”也會給我們造成不小的麻煩。只有了解了它，我們才能更好地使用它。

使用隱含規則

如果要使用隱含規則生成你需要的目標，你所需要做的就是不要寫出這個目標的規則。那麼，make會試圖去自動推導產生這個目標的規則和命令，如果 make可以自動推導生成這個目標的規則和命令，那麼這個行為就是隱含規則的自動推導。當然，隱含規則是make事先約定好的一些東西。例如，我們有下面的一個Makefile：

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

我們可以注意到，這個Makefile中並沒有寫下如何生成foo.o和bar.o這兩目標的規則和命令。因為make的“隱含規則”功能會自動為我們自動去推導這兩個目標的依賴目標和生成命令。

make會在自己的“隱含規則”庫中尋找可以用的規則，如果找到，那麼就會使用。如果找不到，那麼就會報錯。在上面的那個例子中，make調用的隱含規則是，把[.o]的目標的依賴文件置成[.c]，並使用C的編譯命令“cc -c $(CFLAGS) [.c]”來生成[.o]的目標。也就是說，我們完全沒有必要寫下下面的兩條規則：

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

因為，這已經是“約定”好了的事了，make和我們約定好了用C編譯器“cc”生成[.o]文件的規則，這就是隱含規則。

當然，如果我們為[.o]文件書寫了自己的規則，那麼make就不會自動推導並調用隱含規則，它會按照我們寫好的規則忠實地執行。

還有，在make的“隱含規則庫”中，每一條隱含規則都在庫中有其順序，越靠前的則是越被經常使用的，所以，這會導致我們有些時候即使我們顯示地指定了目標依賴，make也不會管。如下面這條規則（沒有命令）：

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

依賴文件“foo.p”（Pascal程序的源文件）有可能變得沒有意義。如果目錄下存在了“foo.c”文件，那麼我們的隱含規則一樣會生效，並會通過“foo.c”調用C的編譯器生成foo.o文件。因為，在隱含規則中，Pascal的規則出現在C的規則之後，所以，make找到可以生成 foo.o的C的規則就不再尋找下一條規則了。如果你確實不希望任何隱含規則推導，那麼，你就不要只寫出“依賴規則”，而不寫命令。

隱含規則一覽

這裏我們將講述所有預先設置（也就是make內建）的隱含規則，如果我們不明確地寫下規則，那麼，make就會在這些規則中尋找所需要規則和命令。當然，我們也可以使用make的參數“-r”或“--no-builtin-rules”選項來取消所有的預設置的隱含規則。

當然，即使是我們指定了“-r”參數，某些隱含規則還是會生效，因為有許多的隱含規則都是使用了“後綴規則”來定義的，所以，只要隱含規則中有 “後綴列表”（也就一系統定義在目標.SUFFIXES的依賴目標），那麼隱含規則就會生效。默認的後綴列表是：.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。具體的細節，我們會在後面講述。

還是先來看一看常用的隱含規則吧。

1、編譯C程序的隱含規則。

“<n>;.o”的目標的依賴目標會自動推導為“<n>;.c”，並且其生成命令是“$(CC) －c $(CPPFLAGS) $(CFLAGS)”

2、編譯C++程序的隱含規則。

“<n>;.o”的目標的依賴目標會自動推導為“<n>;.cc”或是“<n>;.C”，並且其生成命令是“$(CXX) －c $(CPPFLAGS) $(CFLAGS)”。（建議使用“.cc”作為C++源文件的後綴，而不是“.C”）

3、編譯Pascal程序的隱含規則。

“<n>;.o”的目標的依賴目標會自動推導為“<n>;.p”，並且其生成命令是“$(PC) －c $(PFLAGS)”。

4、編譯Fortran/Ratfor程序的隱含規則。

“<n>;.o”的目標的依賴目標會自動推導為“<n>;.r”或“<n>;.F”或“<n>;.f”，並且其生成命令是:

```
 “.f”  “$(FC) －c  $(FFLAGS)”
“.F”  “$(FC) －c  $(FFLAGS) $(CPPFLAGS)”
“.f”  “$(FC) －c  $(FFLAGS) $(RFLAGS)”
```

5、預處理Fortran/Ratfor程序的隱含規則。

“<n>;.f”的目標的依賴目標會自動推導為“<n>;.r”或“<n>;.F”。這個規則只是轉換Ratfor或有預處理的Fortran程序到一個標準的Fortran程序。其使用的命令是：

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

6、編譯Modula-2程序的隱含規則。

“<n>;.sym”的目標的依賴目標會自動推導為“<n>;.def”，並且其生成命令是：“$(M2C) $(M2FLAGS) $(DEFFLAGS)”。“<n.o>;” 的目標的依賴目標會自動推導為“<n>;.mod”，並且其生成命令是：“$(M2C) $(M2FLAGS) $(MODFLAGS)”。

7、彙編和彙編預處理的隱含規則。

“<n>;.o” 的目標的依賴目標會自動推導為“<n>;.s”，默認使用編譯品“as”，並且其生成命令是：“$ (AS) $(ASFLAGS)”。“<n>;.s” 的目標的依賴目標會自動推導為“<n>;.S”，默認使用C預編譯器 “cpp”，並且其生成命令是：“$(AS) $(ASFLAGS)”。

8、鏈接Object文件的隱含規則。

“<n>;”目標依賴於“<n>;.o”，通過運行C的編譯器來運行鏈接程序生成（一般是“ld”），其生成命令是： “$(CC) $(LDFLAGS) <n>;.o $(LOADLIBES) $(LDLIBS)”。這個規則對於只有一個源文件的工程有效，同時也對多個Object文件（由不同的源文件生成）的也有效。例如如下規則：

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

並且“x.c”、“y.c”和“z.c”都存在時，隱含規則將執行如下命令：

```
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
```

如果沒有一個源文件（如上例中的x.c）和你的目標名字（如上例中的x）相關聯，那麼，你最好寫出自己的生成規則，不然，隱含規則會報錯的。

9、Yacc C程序時的隱含規則。

“<n>;.c”的依賴文件被自動推導為“n.y”（Yacc生成的文件），其生成命令是：“$(YACC) $(YFALGS)”。（“Yacc”是一個語法分析器，關於其細節請查看相關資料）

10、Lex C程序時的隱含規則。

“<n>;.c”的依賴文件被自動推導為“n.l”（Lex生成的文件），其生成命令是：“$(LEX) $(LFALGS)”。（關於“Lex”的細節請查看相關資料）

11、Lex Ratfor程序時的隱含規則。

“<n>;.r”的依賴文件被自動推導為“n.l”（Lex生成的文件），其生成命令是：“$(LEX) $(LFALGS)”。

12、從C程序、Yacc文件或Lex文件創建Lint庫的隱含規則。

“<n>;.ln” （lint生成的文件）的依賴文件被自動推導為“n.c”，其生成命令是：“$(LINT) $(LINTFALGS) $(CPPFLAGS) -i”。對於“<n>;.y”和“<n>;.l”也是同樣的規則。

隱含規則使用的變量

在隱含規則中的命令中，基本上都是使用了一些預先設置的變量。你可以在你的makefile中改變這些變量的值，或是在make的命令行中傳入這些值，或是在你的環境變量中設置這些值，無論怎麼樣，只要設置了這些特定的變量，那麼其就會對隱含規則起作用。當然，你也可以利用make的“-R”或 “--no－builtin-variables”參數來取消你所定義的變量對隱含規則的作用。

例如，第一條隱含規則——編譯C程序的隱含規則的命令是“$(CC) －c $(CFLAGS) $(CPPFLAGS)”。Make默認的編譯命令是“cc”，如果你把變量“$(CC)”重定義成“gcc”，把變量“$(CFLAGS)”重定義成“-g”，那麼，隱含規則中的命令全部會以 “gcc －c -g $(CPPFLAGS)”的樣子來執行了。

我們可以把隱含規則中使用的變量分成兩種：一種是命令相關的，如“CC”；一種是參數相的關，如“CFLAGS”。下面是所有隱含規則中會用到的變量：

1、關於命令的變量。

AR
　　函數庫打包程序。默認命令是“ar”。
AS
　　彙編語言編譯程序。默認命令是“as”。
CC
　　C語言編譯程序。默認命令是“cc”。
CXX
　　C++語言編譯程序。默認命令是“g++”。
CO
　　從 RCS文件中擴展文件程序。默認命令是“co”。
CPP
　　C程序的預處理器（輸出是標準輸出設備）。默認命令是“$(CC) －E”。
FC
　　Fortran 和 Ratfor 的編譯器和預處理程序。默認命令是“f77”。
GET
　　從SCCS文件中擴展文件的程序。默認命令是“get”。
LEX
　　Lex方法分析器程序（針對於C或Ratfor）。默認命令是“lex”。
PC
　　Pascal語言編譯程序。默認命令是“pc”。
YACC
　　Yacc文法分析器（針對於C程序）。默認命令是“yacc”。

YACCR
　　Yacc文法分析器（針對於Ratfor程序）。默認命令是“yacc －r”。
MAKEINFO
　　轉換Texinfo源文件（.texi）到Info文件程序。默認命令是“makeinfo”。
TEX
　　從TeX源文件創建TeX DVI文件的程序。默認命令是“tex”。
TEXI2DVI
　　從Texinfo源文件創建軍TeX DVI 文件的程序。默認命令是“texi2dvi”。
WEAVE
　　轉換Web到TeX的程序。默認命令是“weave”。
CWEAVE
　　轉換C Web 到 TeX的程序。默認命令是“cweave”。
TANGLE
　　轉換Web到Pascal語言的程序。默認命令是“tangle”。
CTANGLE
　　轉換C Web 到 C。默認命令是“ctangle”。
RM
　　刪除文件命令。默認命令是“rm －f”。

2、關於命令參數的變量

下面的這些變量都是相關上面的命令的參數。如果沒有指明其默認值，那麼其默認值都是空。

ARFLAGS
　　函數庫打包程序AR命令的參數。默認值是“rv”。
ASFLAGS
　　彙編語言編譯器參數。（當明顯地調用“.s”或“.S”文件時）。
CFLAGS
　　C語言編譯器參數。
CXXFLAGS
　　C++語言編譯器參數。
COFLAGS
　　RCS命令參數。
CPPFLAGS
　　C預處理器參數。（ C 和 Fortran 編譯器也會用到）。
FFLAGS
　　Fortran語言編譯器參數。
GFLAGS
　　SCCS “get”程序參數。
LDFLAGS
　　鏈接器參數。（如：“ld”）
LFLAGS
　　Lex文法分析器參數。
PFLAGS
　　Pascal語言編譯器參數。
RFLAGS
　　Ratfor 程序的Fortran 編譯器參數。
YFLAGS
　　Yacc文法分析器參數。

隱含規則鏈

有些時候，一個目標可能被一系列的隱含規則所作用。例如，一個[.o]的文件生成，可能會是先被Yacc的[.y]文件先成[.c]，然後再被C的編譯器生成。我們把這一系列的隱含規則叫做“隱含規則鏈”。

在上面的例子中，如果文件[.c]存在，那麼就直接調用C的編譯器的隱含規則，如果沒有[.c]文件，但有一個[.y]文件，那麼Yacc的隱含規則會被調用，生成[.c]文件，然後，再調用C編譯的隱含規則最終由[.c]生成[.o]文件，達到目標。

我們把這種[.c]的文件（或是目標），叫做中間目標。不管怎麼樣，make會努力自動推導生成目標的一切方法，不管中間目標有多少，其都會執着地把所有的隱含規則和你書寫的規則全部合起來分析，努力達到目標，所以，有些時候，可能會讓你覺得奇怪，怎麼我的目標會這樣生成？怎麼我的 makefile發瘋了？

在默認情況下，對於中間目標，它和一般的目標有兩個地方所不同：第一個不同是除非中間的目標不存在，才會引發中間規則。第二個不同的是，只要目標成功產生，那麼，產生最終目標過程中，所產生的中間目標文件會被以“rm -f”刪除。

通常，一個被makefile指定成目標或是依賴目標的文件不能被當作中介。然而，你可以明顯地說明一個文件或是目標是中介目標，你可以使用偽目標“.INTERMEDIATE”來強制聲明。（如：.INTERMEDIATE： mid ）

你也可以阻止make自動刪除中間目標，要做到這一點，你可以使用偽目標“.SECONDARY”來強制聲明（如：.SECONDARY :sec）。你還可以把你的目標，以模式的方式來指定（如：%.o）成偽目標“.PRECIOUS”的依賴目標，以保存被隱含規則所生成的中間文件。

在“隱含規則鏈”中，禁止同一個目標出現兩次或兩次以上，這樣一來，就可防止在make自動推導時出現無限遞歸的情況。

Make會優化一些特殊的隱含規則，而不生成中間文件。如，從文件“foo.c”生成目標程序“foo”，按道理，make會編譯生成中間文件 “foo.o”，然後鏈接成“foo”，但在實際情況下，這一動作可以被一條“cc”的命令完成（cc －o foo foo.c），於是優化過的規則就不會生成中間文件。

定義模式規則

你可以使用模式規則來定義一個隱含規則。一個模式規則就好像一個一般的規則，只是在規則中，目標的定義需要有"%"字符。"%"的意思是表示一個或多個任意字符。在依賴目標中同樣可以使用"%"，只是依賴目標中的"%"的取值，取決於其目標。

有一點需要注意的是，"%"的展開發生在變量和函數的展開之後，變量和函數的展開發生在make載入Makefile時，而模式規則中的"%"則發生在運行時。

1、模式規則介紹

模式規則中，至少在規則的目標定義中要包含"%"，否則，就是一般的規則。目標中的"%"定

義表示對文件名的匹配，"%"表示長度任意的非空字符串。例如："%.c"表示以".c"結尾的文件名（文件名的長度至少為3），而"s.%.c"則表示以"s."開頭，".c"結尾的文件名（文件名的長度至少為5）。

如果"%"定義在目標中，那麼，目標中的"%"的值決定了依賴目標中的"%"的值，也就是說，目標中的模式的"%"決定了依賴目標中"%"的樣子。例如有一個模式規則如下：

```
%o  :  %.c  ; <command ......>;
```

其含義是，指出了怎麼從所有的[.c]文件生成相應的[.o]文件的規則。如果要生成的目標是"a.o b.o"，那麼"%c"就是"a.c b.c"。

一旦依賴目標中的"%"模式被確定，那麼，make會被要求去匹配當前目錄下所有的文件名，一旦找到，make就會規則下的命令，所以，在模式規則中，目標可能會是多個的，如果有模式匹配出多個目標，make就會產生所有的模式目標，此時，make關心的是依賴的文件名和生成目標的命令這兩件事。

2、模式規則示例

下面這個例子表示了,把所有的[.c]文件都編譯成[.o]文件.

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

其中，"$@"表示所有的目標的挨個值，"$<"表示了所有依賴目標的挨個值。這些奇怪的變量我們叫"自動化變量"，後面會詳細講述。

下面的這個例子中有兩個目標是模式的：

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

這條規則告訴make把所有的[.y]文件都以"bison -d <n>;.y"執行，然後生成"<n>;.tab.c"和"<n>;.tab.h"文件。（其中，"<n>;"表示一個任意字符串）。如果我們的執行程序"foo"依賴於文件"parse.tab.o"和"scan.o"，並且文件"scan.o"依賴於文件"parse.tab.h"，如果"parse.y"文件被更新了，那麼根據上述的規則，"bison -d parse.y"就會被執行一次，於是，"parse.tab.o"和"scan.o"的依賴文件就齊了。（假設，"parse.tab.o"由"parse.tab.c"生成，和"scan.o"由"scan.c"生成，而"foo"由 "parse.tab.o"和"scan.o"鏈接生成，而且foo和其[.o]文件的依賴關係也寫好，那麼，所有的目標都會得到滿足）

3、自動化變量

在上述的模式規則中，目標和依賴文件都是一系例的文件，那麼我們如何書寫一個命令來完成從不同的依賴文件生成相應的目標？因為在每一次的對模式規則的解析時，都會是不同的目標和依賴文件。

自動化變量就是完成這個功能的。在前面，我們已經對自動化變量有所提涉，相信你看到這

裏已對它有一個感性認識了。所謂自動化變量，就是這種變量會把模式中所定義的一系列的文件自動地挨個取出，直至所有的符合模式的文件都取完了。這種自動化變量只應出現在規則的命令中。

下面是所有的自動化變量及其說明：

$@
表示規則中的目標文件集。在模式規則中，如果有多個目標，那麼，"$@"就是匹配于目標中模式定義的集合。

$%
僅當目標是函數庫文件中，表示規則中的目標成員名。例如，如果一個目標是"foo.a(bar.o)"，那麼，"$%"就是 "bar.o"，"$@"就是"foo.a"。如果目標不是函數庫文件（Unix下是[.a]，Windows下是[.lib]），那麼，其值為空。

$<
依賴目標中的第一個目標名字。如果依賴目標是以模式（即"%"）定義的，那麼"$<"將是符合模式的一系列的文件集。注意，其是一個一個取出來的。

$?
所有比目標新的依賴目標的集合。以空格分隔。

$^
所有的依賴目標的集合。以空格分隔。如果在依賴目標中有多個重複的，那個這個變量會去除重複的依賴目標，只保留一份。

$+
這個變量很像"$^"，也是所有依賴目標的集合。只是它不去除重複的依賴目標。

$*
這個變量表示目標模式中"%"及其之前的部分。如果目標是"dir/a.foo.b"，並且目標的模式是"a.%.b"，那麼，"$*"的值就是"dir/a.foo"。這個變量對於構造有關聯的文件名是比較有較。如果目標中沒有模式的定義，那麼"$*"也就不能被推導出，但是，如果目標文件的後綴是make所識別的，那麼"$*"就是除了後綴的那一部分。例如：如果目標是"foo.c"，因為".c"是make所能識別的後綴名，所以，" $*"的值就是"foo"。這個特性是GNU make的，很有可能不兼容於其它版本的make，所以，你應該盡量避免使用"$*"，除非是在隱含規則或是靜態模式中。如果目標中的後綴是make所不能識別的，那麼"$*"就是空值。

當你希望只對更新過的依賴文件進行操作時，"$?"在顯式規則中很有用，例如，假設有一個函數庫文件叫"lib"，其由其它幾個object文件更新。那麼把object文件打包的比較有效率的Makefile規則是：

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

在上述所列出來的自動量變量中。四個變量（$@、$<、$%、$*）在擴展時只會有一個文件，而另三個的值是一個文件列表。這七個自動化變量還可以取得文件的目錄名或是在當前目錄下的符合模式的文件名，只需要搭配上"D"或"F"字樣。這是GNU make中老版本的特性，在新版本中，我們使用函數"dir"或"notdir"就可以做到了。"D"的含義就是Directory，就是目錄，"F"的含義就是File，就是文件。

下面是對於上面的七個變量分別加上"D"或是"F"的含義：

```
$(@D)
```

表示"$@"的目錄部分（不以斜杠作為結尾），如果"$@"值是"dir/foo.o"，那麼"$(@D)"就是"dir"，而如果"$@"中沒有包含斜杠的話，其值就是"."（當前目錄）。

```
$(@F)
```

表示"$@"的文件部分，如果"$@"值是"dir/foo.o"，那麼"$(@F)"就是"foo.o"，"$(@F)"相當於函數"$(notdir $@)"。

```
"$(*D)"
"$(*F)"
```

和上面所述的同理，也是取文件的目錄部分和文件部分。對於上面的那個例子，"$(*D)"返回"dir"，而"$(*F)"返回"foo"

```
"$(%D)"
"$(%F)"
```

分別表示了函數包文件成員的目錄部分和文件部分。這對於形同"archive(member)"形式的目標中的"member"中包含了不同的目錄很有用。

```
"$(<D)"
"$(<F)"
```

分別表示依賴文件的目錄部分和文件部分。

```
"$(^D)"
"$(^F)"
```

分別表示所有依賴文件的目錄部分和文件部分。（無相同的）

```
"$(+D)"
"$(+F)"
```

分別表示所有依賴文件的目錄部分和文件部分。（可以有相同的）

```
"$(?D)"
"$(?F)"
```

分別表示被更新的依賴文件的目錄部分和文件部分。

最後想提醒一下的是，對於"$<"，為了避免產生不必要的麻煩，我們最好給$後面的那個特定字符都加上圓括號，比如，"$(<)"就要比"$<"要好一些。

還得要注意的是，這些變量只使用在規則的命令中，而且一般都是"顯式規則"和"靜態模式規則"（參見前面"書寫規則"一章）。其在隱含規則中並沒有意義。

4、模式的匹配

一般來說，一個目標的模式有一個有前綴或是後綴的"%"，或是沒有前後綴，直接就是一個"%"。因為"%"代表一個或多個字符，所以在定義好了的模式中，我們把"%"所匹配的內容叫做"莖"，例如"%.c"所匹配的文件"test.c"中"test"就是"莖"。因為在目標和依賴目標中同時有"% "時，依賴目標的"莖"會傳給目標，當做目標中的"莖"。

當一個模式匹配包含有斜杠（實際也不經常包含）的文件時，那麼在進行模式匹配時，目錄部分會首先被移開，然後進行匹配，成功后，再把目錄加回去。在進行"莖"的傳遞時，我們需要知道這個步驟。例如有一個模式"e%t"，文件"src/eat"匹配於該模式，於是"src/a"就是其"莖"，如果這個模式定義在依賴目標中，而被依賴於這個模式的目標中又有個模式"c%r"，那麼，目標就是"src/car"。（"莖"被傳遞）

5、重載內建隱含規則

你可以重載內建的隱含規則（或是定義一個全新的），例如你可以重新構造和內建隱含規則不同的命令，如：

```
%o  :  %.c
       $(CC) -c $(CPPFLAGS) $(CFLAGS) -D$(date)
```

你可以取消內建的隱含規則，只要不在後面寫命令就行。如：

```
%o  :  %.s
```

同樣，你也可以重新定義一個全新的隱含規則，其在隱含規則中的位置取決於你在哪裡寫下這個規則。朝前的位置就靠前。

老式風格的"後綴規則"

後綴規則是一個比較老式的定義隱含規則的方法。後綴規則會被模式規則逐步地取代。因為模式規則更強更清晰。為了和老版本的Makefile兼容，GNU make同樣兼容於這些東西。後綴規則有兩種方式："雙後綴"和"單後綴"。

雙後綴規則定義了一對後綴：目標文件的後綴和依賴目標（源文件）的後綴。如".c.o"相當於"%o : %c"。單後綴規則只定義一個後綴，也就是源文件的後綴。如".c"相當於"% : %c"。

後綴規則中所定義的後綴應該是make所認識的，如果一個後綴是make所認識的，那麼這個規則就是單後綴規則，而如果兩個連在一起的後綴都被 make所認識，那就是雙後綴規則。例如：".c"和".o"都是make所知道。因而，如果你定義了一個規則是".c.o"那麼其就是雙後綴規則，意義就是".c"是源文件的後綴，".o"是目標文件的後綴。如下示例：

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

後綴規則不允許任何的依賴文件，如果有依賴文件的話，那就不是後綴規則，那些後綴統統被認為是文件名，如：

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

這個例子，就是說，文件".c.o"依賴於文件"foo.h"，而不是我們想要的這樣：

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

後綴規則中，如果沒有命令，那是毫無意義的。因為他也不會移去內建的隱含規則。

而要讓make知道一些特定的後綴，我們可以使用偽目標".SUFFIXES"來定義或是刪除，如：

```
.SUFFIXES: .hack .win
```

把後綴.hack和.win加入後綴列表中的末尾。

```
.SUFFIXES:              # 删除默认的后缀
.SUFFIXES: .c .o .h   # 定义自己的后缀
```

先清楚默認後綴，后定義自己的後綴列表。

make的參數"-r"或"-no-builtin-rules"也會使用得默認的後綴列表為空。而變量"SUFFIXE"被用來定義默認的後綴列表，你可以用".SUFFIXES"來改變後綴列表，但請不要改變變量"SUFFIXE"的值。

隱含規則搜索算法

比如我們有一個目標叫 T。下面是搜索目標T的規則的算法。請注意，在下面，我們沒有提到後綴規則，原因是，所有的後綴規則在Makefile被載入內存時，會被轉換成模式規則。如果目標是"archive(member)"的函數庫文件模式，那麼這個算法會被運行兩次，第一次是找目標T，如果沒有找到的話，那麼進入第二次，第二次會把"member"當作T來搜索。

1、把T的目錄部分分離出來。叫D，而剩餘部分叫N。（如：如果T是"src/foo.o"，那麼，D就是"src/"，N就是"foo.o"）

2、創建所有匹配于T或是N的模式規則列表。

3、如果在模式規則列表中有匹配所有文件的模式，如"%"，那麼從列表中移除其它的模式。

4、移除列表中沒有命令的規則。

5、對於第一個在列表中的模式規則：

1）推導其"莖"S，S應該是T或是N匹配于模式中"%"非空的部分。

2）計算依賴文件。把依賴文件中的"%"都替換成"莖"S。如果目標模式中沒有包含斜框字符，而把D加在第一個依賴文件的開頭。

3）測試是否所有的依賴文件都存在或是理當存在。（如果有一個文件被定義成另外一個規則的目標文件，或者是一個顯式規則的依賴文件，那麼這個文件就叫"理當存在"）

4）如果所有的依賴文件存在或是理當存在，或是就沒有依賴文件。那麼這條規則將被採用，退出該算法。

6、如果經過第5步，沒有模式規則被找到，那麼就做更進一步的搜索。對於存在於列表中的第一個模式規則：

1）如果規則是終止規則，那就忽略它，繼續下一條模式規則。

2）計算依賴文件。（同第5步）

3）測試所有的依賴文件是否存在或是理當存在。

4）對於不存在的依賴文件，遞歸調用這個算法查找他是否可以被隱含規則找到。

5）如果所有的依賴文件存在或是理當存在，或是就根本沒有依賴文件。那麼這條規則被採用，退出該算法。

7、如果沒有隱含規則可以使用，查看".DEFAULT"規則，如果有，採用，把".DEFAULT"的命令給T使用。

一旦規則被找到，就會執行其相當的命令，而此時，我們的自動化變量的值才會生成。