跟我一起寫**Makefile:**使用函數

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

使用函數

在Makefile中可以使用函數來處理變量，從而讓我們的命令或是規則更為的靈活和具有智能。make所支持的函數也不算很多，不過已經足夠我們的操作了。函數調用后，函數的返回值可以當做變量來使用。

函数的调用语法

函数调用，很像變量的使用，也是以“$”來標識的，其語法如下：

```
$(<function> <arguments>)
```

或是

```
${<function> <arguments>}
```

這裏，<function>就是函数名，make支持的函数不多。<arguments>為函数的參数，參数間以逗號“,”分隔，而函数名和參数之間以“空格”分隔。函数調用以“$”開頭，以圓括號或花括號把函数名和參数括起。感覺很像一個變量，是不是？函数中的參数可以使用變量，為了風格的統一，函数和變量的括號最好一樣，如使用“$(subst a,b,$(x))”這樣的形式，而不是“$(subst a,b, ${x})”的形式。因為統一會更清楚，也會減少一些不必要的麻煩。

還是來看一個示例：

```
comma:= ,
empty:=
space:= $(empty) $(empty)
foo:= a b c
bar:= $(subst $(space),$(comma),$(foo))
```

在這個示例中，$(comma)的值是一個逗號。$(space)使用了$(empty)定義了一個空格，$(foo)的值是“a b c”，$ (bar)的定義用，調用了函数“subst”，這是一個替換函数，這個函数有三個參数，第一個參数是被替換字串，第二個參数是替換字串，第三個參数是替換操作作用的字串。這個函数也就是把$(foo)中的空格替換成逗號，所以$(bar)的值是“a,b,c”。

字符串處理函数

```
$(subst <from>,<to>,<text>)
```

- 名稱：字符串替換函数——subst。
- 功能：把字串<text>中的<from>字符串替換成<to>。
- 返回：函数返回被替換過後的字符串。

- 示例：

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

把“feet on the street”中的“ee”替換成“EE”，返回結果是“fEEt on the strEEt”。

```
$(patsubst <pattern>,<replacement>,<text>)
```

- 名稱：模式字符串替換函数——patsubst。

- 功能：查找<text>中的單詞（單詞以“空格”、“Tab”或“回車”“換行”分隔）是否符合模式< pattern>，如果匹配的話，則以<replacement>替換。這裏，<pattern>可以包括通配符 “%”，表示任意長度的字串。如果<replacement>中也包含“%”，那麼，<replacement>中的這個 “%”將是<pattern>中的那個“%”所代表的字串。（可以用“\”來轉義，以“\%”來表示真實含義的“%”字符）
- 返回：函數返回被替換過後的字符串。

- 示例：

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

把字串“x.c.c bar.c”符合模式[%.c]的單詞替換成[%.o]，返回結果是“x.c.o bar.o”

備註：

這和我們前面“變量章節”說過的相關知識有點相似。如：

“$(var:<pattern>=<replacement>;)” 相當於 “$(patsubst <pattern>,<replacement>,$(var))”，

而“$(var: <suffix>=<replacement>)” 則相當於 “$(patsubst %<suffix>,%<replacement>,$(var))”。

例如有：objects = foo.o bar.o baz.o，那麼，“$(objects:.o=.c)”和“$(patsubst %.o,%.c,$(objects))”是一樣的。

```
$(strip <string>)
```

- 名稱：去空格函數——strip。
- 功能：去掉<string>;字串中開頭和結尾的空字符。
- 返回：返回被去掉空格的字符串值。
- 示例：

```
$(strip a b c )
```

把字串“a b c ”去到開頭和結尾的空格，結果是“a b c”。

```
$(findstring <find>,<in>)
```

- 名稱：查找字符串函數——findstring。
- 功能：在字串<in>中查找<find>字串。
- 返回：如果找到，那麼返回<find>，否則返回空字符串。
- 示例：

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

第一個函數返回“a”字符串，第二個返回“”字符串（空字符串）

```
$(filter <pattern...>,<text>)
```

- 名稱：過濾函數——filter。
- 功能：以<pattern>模式過濾<text>字符串中的單詞，保留符合模式<pattern>的單詞。可以有多個模式。
- 返回：返回符合模式<pattern>;的字串。
- 示例：

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

$(filter %.c %.s,$(sources))返回的值是“foo.c bar.c baz.s”。

```
$(filter-out <pattern...>,<text>)
```

- 名稱：反過濾函數——filter-out。
- 功能：以<pattern>模式過濾<text>字符串中的單詞，去除符合模式<pattern>的單詞。可以有多個模式。
- 返回：返回不符合模式<pattern>的字串。
- 示例：

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

$(filter-out $(mains),$(objects)) 返回值是“foo.o bar.o”。

```
$(sort <list>)
```

- 名稱：排序函數——sort。
- 功能：給字符串<list>中的單詞排序（升序）。
- 返回：返回排序后的字符串。
- 示例：$(sort foo bar lose)返回“bar foo lose” 。
- 備註：sort函數會去掉<list>中相同的單詞。

```
$(word <n>,<text>)
```

- 名稱：取單詞函數——word。
- 功能：取字符串<text>中第<n>個單詞。（從一開始）
- 返回：返回字符串<text>中第<n>個單詞。如果<n>比<text>中的單詞數要大，那麼返回空字符串。
- 示例：$(word 2, foo bar baz)返回值是“bar”。

```
$(wordlist <ss>,<e>,<text>)
```

- 名稱：取單詞串函數——wordlist。
- 功能：從字符串<text>中取從<ss>開始到<e>的單詞串。<ss>和<e>是一個數字。

- 返回：返回字符串<text>中從<ss>到<e>的單詞字串。如果<ss>比<text>中的單詞數要大，那麼返回空字符串。如果<e>大於<text>的單詞數，那麼返回從<ss>開始，到<text>結束的單詞串。
- 示例： $(wordlist 2, 3, foo bar baz)返回值是“bar baz”。

```
$(words <text>)
```

- 名稱：單詞個數統計函數——words。
- 功能：統計<text>中字符串中的單詞個數。
- 返回：返回<text>中的單詞數。
- 示例：$(words, foo bar baz)返回值是“3”。
- 備註：如果我們要取<text>中最後的一個單詞，我們可以這樣：$(word $(words <text>),<text>)。

```
$(firstword <text>)
```

- 名稱：首單詞函數——firstword。
- 功能：取字符串<text>中的第一個單詞。
- 返回：返回字符串<text>的第一個單詞。
- 示例：$(firstword foo bar)返回值是“foo”。
- 備註：這個函數可以用word函數來實現：$(word 1,<text>)。

以上，是所有的字符串操作函數，如果搭配混合使用，可以完成比較複雜的功能。這裏，舉一個現實中應用的例子。我們知道，make使用“VPATH”變量來指定“依賴文件”的搜索路徑。於是，我們可以利用這個搜索路徑來指定編譯器對頭文件的搜索路徑參數CFLAGS，如：

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

如果我們的“$(VPATH)”值是“src:../headers”，那麼“$(patsubst %,-I%,$(subst :, ,$(VPATH)))”將返回“-Isrc -I../headers”，這正是cc或gcc搜索頭文件路徑的參數。

文件名操作函數

下面我們要介紹的函數主要是處理文件名的。每個函數的參數字符串都會被當做一個或是一系列的文件名來對待。

```
$(dir <names...>)
```

- 名稱：取目錄函數——dir。
- 功能：從文件名序列<names>中取出目錄部分。目錄部分是指最後一個反斜杠（“/”）之前的部分。如果沒有反斜杠，那麼返回“./”。
- 返回：返迴文件名序列<names>的目錄部分。
- 示例： $(dir src/foo.c hacks)返回值是“src/ ./”。

```
$(notdir <names...>)
```

- 名稱：取文件函數——notdir。

- 功能：從文件名序列<names>中取出非目錄部分。非目錄部分是指最後一個反斜杠（“/”）之後的部分。
- 返回：返迴文件名序列<names>的非目錄部分。
- 示例： $(notdir src/foo.c hacks)返回值是“foo.c hacks”。

```
$(suffix <names...>)
```

- 名稱：取後綴函數——suffix。
- 功能：從文件名序列<names>中取出各個文件名的後綴。
- 返回：返迴文件名序列<names>的後綴序列，如果文件沒有後綴，則返回空字串。
- 示例：$(suffix src/foo.c src-1.0/bar.c hacks)返回值是“.c .c”。

```
$(basename <names...>)
```

- 名稱：取前綴函數——basename。
- 功能：從文件名序列<names>中取出各個文件名的前綴部分。
- 返回：返迴文件名序列<names>的前綴序列，如果文件沒有前綴，則返回空字串。
- 示例：$(basename src/foo.c src-1.0/bar.c hacks)返回值是“src/foo src-1.0/bar hacks”。

```
$(addsuffix <suffix>,<names...>)
```

- 名稱：加後綴函數——addsuffix。
- 功能：把後綴<suffix>加到<names>中的每個單詞後面。
- 返回：返回加過後綴的文件名序列。
- 示例：$(addsuffix .c,foo bar)返回值是“foo.c bar.c”。

```
$(addprefix <prefix>,<names...>)
```

- 名稱：加前綴函數——addprefix。
- 功能：把前綴<prefix>加到<names>中的每個單詞後面。
- 返回：返回加過前綴的文件名序列。
- 示例：$(addprefix src/,foo bar)返回值是“src/foo src/bar”。

```
$(join <list1>,<list2>)
```

- 名稱：連接函數——join。
- 功能：把<list2>中的單詞對應地加到<list1>的單詞後面。如果<list1>的單詞個數要比<list2>的多，那麼，<list1>中的多出來的單詞將保持原樣。如果<list2>的單詞個數要比<list1>多，那麼，<list2>多出來的單詞將被複制到<list2>中。
- 返回：返回連接過後的字符串。
- 示例：$(join aaa bbb , 111 222 333)返回值是“aaa111 bbb222 333”。

==foreach 函數==

foreach函數和別的函數非常的不一樣。因為這個函數是用來做循環用的，Makefile中的foreach函數幾乎是仿照于Unix標準 Shell（/bin/sh）中的for語句，或是C-Shell（/bin/csh）中的foreach語句而構建的。它的語法是：

```
$(foreach ,<list>,<text>)
```

這個函數的意思是，把參數<list>中的單詞逐一取出放到參數<var>所指定的變量中，然後再執行< text>所包含的表達式。每一次<text>會返回一個字符串，循環過程中，<text>的所返回的每個字符串會以空格分隔，最後當整個循環結束時，<text>所返回的每個字符串所組成的整個字符串（以空格分隔）將會是foreach函數的返回值。

所以，<var>最好是一個變量名，<list>可以是一個表達式，而<text>中一般會使用<var>這個參數來依次枚舉<list>中的單詞。舉個例子：

```
names  := a b c d
files  := $(foreach n,$(names),$(n).o)
```

上面的例子中，$(name)中的單詞會被挨個取出，並存到變量“n”中，“$(n).o”每次根據“$(n)”計算出一個值，這些值以空格分隔，最後作為foreach函數的返回，所以，$(files)的值是“a.o b.o c.o d.o”。

注意，foreach中的<var>參數是一個臨時的局部變量，foreach函數執行完后，參數<var>的變量將不在作用，其作用域只在foreach函數當中。

'==if 函数==

if函數很像GNU的make所支持的條件語句——ifeq（參見前面所述的章節），if函數的語法是：

```
$(if <condition>,<then-part>)
```

或是

```
$(if <condition>,<then-part>,<else-part>)
```

可見，if函數可以包含“else”部分，或是不含。即if函數的參數可以是兩個，也可以是三個。<condition>參數是 if的表達式，如果其返回的為非空字符串，那麼這個表達式就相當於返回真，於是，<then-part>會被計算，否則<else- part>會被計算。

而if函數的返回值是，如果<condition>為真（非空字符串），那個<then-part>會是整個函數的返回值，如果<condition>為假（空字符串），那麼<else-part>會是整個函數的返回值，此時如果< else-part>沒有被定義，那麼，整個函數返回空字串。

所以，<then-part>和<else-part>只會有一個被計算。

==call函數==

call函數是唯一一個可以用來創建新的參數化的函數。你可以寫一個非常複雜的表達式，這個表達式中，你可以定義許多參數，然後你可以用call函數來向這個表達式傳遞參數。其語法是：

```
$(call <expression>,<parm1>,<parm2>,<parm3>,...)
```

當make執行這個函數時，<expression>;參數中的變量，如$(1)，$(2)，$(3)等，會被參數<parm1>;，<parm2>;，<parm3>;依次取代。而<expression>;的返回值就是 call函數的返回值。例如：

```
reverse =  $(1) $(2)
foo = $(call reverse,a,b)
```

那麼，foo的值就是“a b”。當然，參數的次序是可以自定義的，不一定是順序的，如：

```
reverse =  $(2) $(1)
foo = $(call reverse,a,b)
```

此時的foo的值就是“b a”。

origin *函數*

origin函數不像其它的函數，他並不操作變量的值，他只是告訴你你的這個變量是哪裡來的？其語法是：

```
$(origin <variable>;)
```

注意，<variable>;是變量的名字，不應該是引用。所以你最好不要在<variable>;中使用“$”字符。Origin函數會以其返回值來告訴你這個變量的“出生情況”，下面，是origin函數的返回值:

“undefined”

如果<variable>;從來沒有定義過，origin函數返回這個值“undefined”。

“default”

如果<variable>;是一個默認的定義，比如“CC”這個變量，這種變量我們將在後面講述。

“environment”

如果<variable>;是一個環境變量，並且當Makefile被執行時，“-e”參數沒有被打開。

“file”

如果<variable>;這個變量被定義在Makefile中。

“command line”

如果<variable>;這個變量是被命令行定義的。

“override”

如果<variable>;是被override指示符重新定義的。

“automatic ”

如果<variable>;是一個命令運行中的自動化變量。關於自動化變量將在後面講述。

這些信息對於我們編寫Makefile是非常有用的，例如，假設我們有一個Makefile其包了一個定義文件Make.def，在 Make.def中定義了一個變量“bletch ”，而我們的環境中也有一個環境變量“bletch ”，此時，我們想判斷一下，如果變量來源於環境，那麼我們就把之重定義了，如果來源於Make.def或是命令行等非環境的，那麼我們就不重新定義它。於是，在我們的Makefile中，我們可以這樣寫：

```
ifdef bletch

ifeq "$(origin bletch)" "environment"

bletch = barf, gag, etc.

endif

endif
```

當然，你也許會說，使用override關鍵字不就可以重新定義環境中的變量了嗎？為什麼需要使用這樣的步驟？是的，我們用override是可以達到這樣的效果，可是override過於粗暴，它同時會把從命令行定義的變量也覆蓋了，而我們只想重新定義環境傳來的，而不想重新定義命令行傳來的。

shell *函數*

shell函數也不像其它的函數。顧名思義，它的參數應該就是操作系統Shell的命令。它和反引號“` ”是相同的功能。這就是說，shell函數把執行操作系統命令后的輸出作為函數返回。於是，我們可以用操作系統命令以及字符串處理命令awk，sed等等命令來生成一個變量，如：

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

注意，這個函數會新生成一個Shell程序來執行命令，所以你要注意其運行性能，如果你的Makefile中有一些比較複雜的規則，並大量使用了這個函數，那麼對於你的系統性能是有害的。特別是Makefile的隱晦的規則可能會讓你的shell函數執行的次數比你想像的多得多。

控制 ***make*** *的函數*

make提供了一些函數來控制make的運行。通常，你需要檢測一些運行Makefile時的運行時信息，並且根據這些信息來決定，你是讓make繼續執行，還是停止。

```
$(error <text ...>;)
```

產生一個致命的錯誤，<text ...>;是錯誤信息。注意，error函數不會在一被使用就會產生錯誤信息，所以如果你把其定義在某個變量中，並在後續的腳本中使用這個變量，那麼也是可以的。

例如：

示例一：

```
ifdef ERROR_001
$(error error is $(ERROR_001))
endif
```

示例二：

```
ERR = $(error found an error!)
.PHONY: err
err:  ; $(ERR)
```

示例一會在變量ERROR_001定義了后執行時產生error調用，而示例二則在目錄err被執行時才發生error調用。

```
$(warning <text ...>;)
```

這個函數很像error函數，只是它並不會讓make退出，只是輸出一段警告信息，而make繼續執行。

取自"http://wiki.ubuntu.org.cn/index.php?title=%E8%B7%9F%E6%88%91%E4%B8%80%E8%B5%B7%E5%86%99Makefile:%E4%BD%BF%E7%94%A8%E5%87%BD%E6%95%B0&variant=zh-hant"