跟我一起寫**Makefile:**使用變量

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

使用變量

在Makefile中的定義的變量，就像是C/C++語言中的宏一樣，他代表了一個文本字串，在Makefile中執行的時候其會自動原模原樣地展開在所使用的地方。其與C/C++所不同的是，你可以在Makefile中改變其值。在Makefile中，變量可以使用在

“目標”，“依賴目標”， “命令”或是Makefile的其它部分中。

變量的命名字可以包含字符、數字，下劃線（可以是數字開頭），但不應該含有“:”、“#”、“=”或是空字符（空格、回車等）。變量是大小寫敏感的，“foo”、“Foo”和“FOO”是三個不同的變量名。傳統的Makefile的變量名是全大寫的命名方式，但我推薦使用大小寫搭配的變量名，如： MakeFlags。這樣可以避免和系統的變量衝突，而發生意外的事情。

有一些變量是很奇怪字串，如“$<”、“$@”等，這些是自動化變量，我會在後面介紹。

變量的基礎

變量在聲明時需要給予初值，而在使用時，需要給在變量名前加上“$”符號，但最好用小括號“（）”或是大括號“{}”把變量給包括起來。如果你要使用真實的“$”字符，那麼你需要用“$$”來表示。

變量可以使用在許多地方，如規則中的“目標”、“依賴”、“命令”以及新的變量中。先看一個例子：

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

$(objects)  : defs.h
```

變量會在使用它的地方精確地展開，就像C/C++中的宏一樣，例如：

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

展開后得到：

```
prog.o  : prog.c
        cc -c prog.c
```

當然，千萬不要在你的Makefile中這樣干，這裏只是舉個例子來表明Makefile中的變量在使用處展開的真實樣子。可見其就是一個“替代”的原理。

另外，給變量加上括號完全是為了更加安全地使用這個變量，在上面的例子中，如果你不想給變量加上括號，那也可以，但我還是強烈建議你給變量加上括號。

變量中的變量

在定義變量的值時，我們可以使用其它變量來構造變量的值，在Makefile中有兩種方式來在用變量定義變量的值。

先看第一種方式，也就是簡單的使用“=”號，在“=”左側是變量，右側是變量的值，右側變量的值可以定義在文件的任何一處，也就是說，右側中的變量不一定非要是已定義好的

值，其也可以使用後面定義的值。如：

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

all:
        echo $(foo)
```

我們執行“make all”將會打出變量$(foo)的值是“Huh?”（ $(foo)的值是$(bar)，$(bar)的值是$(ugh)，$(ugh)的值是“Huh?”）可見，變量是可以使用後面的變量來定義的。

這個功能有好的地方，也有不好的地方，好的地方是，我們可以把變量的真實值推到後面來定義，如：

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

當“CFLAGS”在命令中被展開時，會是“-Ifoo -Ibar -O”。但這種形式也有不好的地方，那就是遞歸定義，如：

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

或：

```
A = $(B)
B = $(A)
```

這會讓make陷入無限的變量展開過程中去，當然，我們的make是有能力檢測這樣的定義，並會報錯。還有就是如果在變量中使用函數，那麼，這種方式會讓我們的make運行時非常慢，更糟糕的是，他會使用得兩個make的函數“wildcard”和“shell”發生不可預知的錯誤。因為你不會知道這兩個函數會被調用多少次。

為了避免上面的這種方法，我們可以使用make中的另一種用變量來定義變量的方法。這種方法使用的是“:=”操作符，如：

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

其等價于：

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

值得一提的是，這種方法，前面的變量不能使用後面的變量，只能使用前面已定義好了的變量。如果是這樣：

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

那麼，y的值是“bar”，而不是“foo bar”。

上面都是一些比較簡單的變量使用了，讓我們來看一個複雜的例子，其中包括了make的函數、條件表達式和一個系統變量“MAKELEVEL”的使用：

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

關於條件表達式和函數，我們在後面再說，對於系統變量“MAKELEVEL”，其意思是，如果我們的make有一個嵌套執行的動作（參見前面的“嵌套使用make”），那麼，這個變量會記錄了我們的當前Makefile的調用層數。

下面再介紹兩個定義變量時我們需要知道的，請先看一個例子，如果我們要定義一個變量，其值是一個空格，那麼我們可以這樣來：

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

nullstring是一個Empty變量，其中什麼也沒有，而我們的space的值是一個空格。因為在操作符的右邊是很難描述一個空格的，這裏採用的技術很管用，先用一個Empty變量來標明變量的值開始了，而後面採用“#”註釋符來表示變量定義的終止，這樣，我們可以定義出其值是一個空格的變量。請注意這裏關於“#”的使用，註釋符“#”的這種特性值得我們注意，如果我們這樣定義一個變量：

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

dir這個變量的值是“/foo/bar”，後面還跟了4個空格，如果我們這樣使用這樣變量來指定別的目錄——“$(dir)/file”那麼就完蛋了。

還有一個比較有用的操作符是“?=”，先看示例：

```
FOO  ?= bar
```

其含義是，如果FOO沒有被定義過，那麼變量FOO的值就是“bar”，如果FOO先前被定義過，那麼這條語將什麼也不做，其等價于：

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

變量高級用法

這裏介紹兩種變量的高級使用方法，第一種是變量值的替換。

我們可以替換變量中的共有的部分，其格式是“$(var:a=b)”或是“${var:a=b}”，其意思是，把變量“var”中所有以“a”字串“結尾”的“a”替換成“b”字串。這裏的“結尾”意思是“空格”或是“結束符”。

還是看一個示例吧：

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

這個示例中，我們先定義了一個“$(foo)”變量，而第二行的意思是把“$(foo)”中所有以“.o”字串“結尾”全部替換成“.c”，所以我們的“$(bar)”的值就是“a.c b.c c.c”。

另外一種變量替換的技術是以“靜態模式”（參見前面章節）定義的，如：

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

這依賴於被替換字串中的有相同的模式，模式中必須包含一個“%”字符，這個例子同樣讓$(bar)變量的值為“a.c b.c c.c”。

第二種高級用法是——“把變量的值再當成變量”。先看一個例子：

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

在這個例子中，$(x)的值是“y”，所以$($(x))就是$(y)，於是$(a)的值就是“z”。（注意，是“x=y”，而不是“x=$(y)”）

我們還可以使用更多的層次：

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

這裏的$(a)的值是“u”，相關的推導留給讀者自己去做吧。

讓我們再複雜一點，使用上“在變量定義中使用變量”的第一個方式，來看一個例子：

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

這裏的$($(x))被替換成了$($(y))，因為$(y)值是“z”，所以，最終結果是：a:=$(z)，也就是“Hello”。

再複雜一點，我們再加上函數：

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

這個例子中，“$($($(z)))”擴展為“$($(y))”，而其再次被擴展為“$($(subst 1,2,$(x)))”。$(x)的值是“variable1”，subst函數把“variable1”中的所有“1”字串替換成“2”字串，於是，“variable1”變成 “variable2”，再取其值，所以，最終，$(a)的值就是$(variable2)的值——“Hello”。（喔，好不容易）

在這種方式中，或要可以使用多個變量來組成一個變量的名字，然後再取其值：

```
first_second = Hello
a = first
b = second
all = $($a_$b)
```

這裏的“$a_$b”組成了“first_second”，於是，$(all)的值就是“Hello”。

再來看看結合第一種技術的例子：

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

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

這個例子中，如果$(a1)的值是“a”的話，那麼，$(sources)的值就是“a.c b.c c.c”；如果$(a1)的值是“1”，那麼$(sources)的值是“1.c 2.c 3.c”。

再來看一個這種技術和“函數”與“條件語句”一同使用的例子：

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

bar  := a d b g q c

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

這個示例中，如果定義了“do_sort”，那麼：foo := $(sort a d b g q c) 於是$(foo)的值就是“a b c d g q”，而如果沒有定義“do_sort”，那麼：foo := $(strip a d b g q c) 調用的就是strip函數。

當然，“把變量的值再當成變量”這種技術，同樣可以用在操作符的左邊：

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

這個例子中定義了三個變量：“dir”，“foo_sources”和“foo_print”。

追加變量值

我們可以使用“+=”操作符給變量追加值，如：

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

於是，我們的$(objects)值變成：“main.o foo.o bar.o utils.o another.o”（another.o被追加進去了）

使用“+=”操作符，可以模擬為下面的這種例子：

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

所不同的是，用“+=”更為簡潔。

如果變量之前沒有定義過，那麼，“+=”會自動變成“=”，如果前面有變量定義，那麼“+=”會繼承於前次操作的賦值符。如果前一次的是“:=”，那麼“+=”會以“:=”作為其賦值符，如：

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

等價于：

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

但如果是這種情況：

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

由於前次的賦值符是“=”，所以“+=”也會以“=”來做為賦值，那麼豈不會發生變量的遞補歸定義，這是很不好的，所以make會自動為我們解決這個問題，我們不必擔心這個問題。

override 指示符

如果有變量是通常make的命令行參數設置的，那麼Makefile中對這個變量的賦值會被忽略。如果你想在Makefile中設置這類參數的值，那麼，你可以使用“override”指示符。其語法是：

```
override <variable>; = <value>;

override <variable>;  := <value>;
```

當然，你還可以追加：

```
override <variable>; += <more text>;
```

對於多行的變量定義，我們用define指示符，在define指示符前，也同樣可以使用override指示符，如：

```
override define foo
bar
endef
```

多行變量

還有一種設置變量值的方法是使用define關鍵字。使用define關鍵字設置變量的值可以有換行，這有利於定義一系列的命令（前面我們講過“命令包”的技術就是利用這個關鍵字）。

define指示符後面跟的是變量的名字，而重起一行定義變量的值，定義是以endef 關鍵字結束。其工作方式和“=”操作符一樣。變量的值可以包含函數、命令、文字，或是其它變量。因為命令需要以[Tab]鍵開頭，所以如果你用define定義的命令變量中沒有以[Tab]鍵開頭，那麼make 就不會把其認為是命令。

下面的這個示例展示了define的用法：

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

環境變量

make運行時的系統環境變量可以在make開始運行時被載入到Makefile文件中，但是如果Makefile中已定義了這個變量，或是這個變量由make命令行帶入，那麼系統的環境變量的值將被覆蓋。（如果make指定了“-e”參數，那麼，系統環境變量將覆蓋Makefile中定義的變量）

因此，如果我們在環境變量中設置了“CFLAGS”環境變量，那麼我們就可以在所有的Makefile中使用這個變量了。這對於我們使用統一的編譯參數有比較大的好處。如果Makefile中定義了CFLAGS，那麼則會使用Makefile中的這個變量，如果沒有定義則使用系統環境變量的值，一個共性和個性的統一，很像“全局變量”和“局部變量”的特性。

當make嵌套調用時（參見前面的“嵌套調用”章節），上層Makefile中定義的變量會以系統環境變量的方式傳遞到下層的Makefile 中。當然，默認情況下，只有通過命令行設置的變量會被傳遞。而定義在文件中的變量，如果要向下層Makefile傳遞，則需要使用exprot關鍵字來聲明。（參見前面章節）

當然，我並不推薦把許多的變量都定義在系統環境中，這樣，在我們執行不用的Makefile時，擁有的是同一套系統變量，這可能會帶來更多的麻煩。

目標變量

前面我們所講的在Makefile中定義的變量都是“全局變量”，在整個文件，我們都可以訪問這

些變量。當然，“自動化變量”除外，如“$<”等這種類量的自動化變量就屬於“規則型變量”，這種變量的值依賴於規則的目標和依賴目標的定義。

當然，我也同樣可以為某個目標設置局部變量，這種變量被稱為“Target-specific Variable”，它可以和“全局變量”同名，因為它的作用範圍只在這條規則以及連帶規則中，所以其值也只在作用範圍內有效。而不會影響規則鏈以外的全局變量的值。

其語法是：

```
<target ...>  : <variableassignment>;

<target ...>  : overide <variableassignment>
```

<variable-assignment>;可以是前面講過的各種賦值表達式，如“=”、“:=”、“+=”或是“？=”。第二個語法是針對於make命令行帶入的變量，或是系統環境變量。

這個特性非常的有用，當我們設置了這樣一個變量，這個變量會作用到由這個目標所引發的所有的規則中去。如：

```
prog  : CFLAGS =-g
prog  : prog.o foo.o bar.o
        $(CC) $(CFLAGS) prog.o foo.o bar.o

prog.o  : prog.c
        $(CC) $(CFLAGS) prog.c

foo.o  : foo.c
        $(CC) $(CFLAGS) foo.c

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

在這個示例中，不管全局的$(CFLAGS)的值是什麼，在prog目標，以及其所引發的所有規則中（prog.o foo.o bar.o的規則），$(CFLAGS)的值都是“-g”

模式變量

在GNU的make中，還支持模式變量（Pattern-specific Variable），通過上面的目標變量中，我們知道，變量可以定義在某個目標上。模式變量的好處就是，我們可以給定一種“模式”，可以把變量定義在符合這種模式的所有目標上。

我們知道，make的“模式”一般是至少含有一個“%”的，所以，我們可以以如下方式給所有以[.o]結尾的目標定義目標變量：

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

同樣，模式變量的語法和“目標變量”一樣：

```
<pattern ...>;  : <variableassignment>;

<pattern ...>;  : override <variableassignment>;
```

override同樣是針對於系統環境傳入的變量，或是make命令行指定的變量。