# 使用变量

在Makefile中的定义的变量，就像是C/C++语言中的宏一样，他代表了一个文本字串，在Makefile中执行的时候其会自动原模原样地展开在所使用的地方。其与C/C++所不同的是，你可以在Makefile中改变其值。在Makefile中，变量可以使用在“目标”，“依赖目标”， “命令”或是Makefile的其它部分中。

变量的命名字可以包含字符、数字，下划线（可以是数字开头），但不应该含有“:”、“#”、“=”或是空字符（空格、回车等）。变量是區分英文字母大小写的，“foo”、“Foo”和“FOO”是三个不同的变量名。传统的Makefile的变量名是全大写的命名方式，但我推荐使用大小写搭配的变量名，如： MakeFlags。这样可以避免和系统的变量冲突，而发生意外的事情。

有一些变量是很奇怪字串，如“$<”、“$@”等，这些是自动化变量，我会在后面介绍。

## 变量的基础

变量在声明时需要给予初值，而在使用时，需要给在变量名前加上“$”符号，但最好用小括号“（）”或是大括号“{}”把变量给包括起来。如果你要使用真实的“$”字符，那么你需要用“$$”来表示。

变量可以使用在许多地方，如规则中的“目标”、“依赖”、“命令”以及新的变量中。先看一个例子：

objects = program.o foo.o utils.o

program : $(objects)

cc -o program $(objects)

$(objects) : defs.h

变量会在使用它的地方精确地展开，就像C/C++中的宏一样，例如：

foo = c

prog.o : prog.$(foo)

$(foo)$(foo) -$(foo) prog.$(foo)

展开后得到：

prog.o : prog.c

cc -c prog.c

当然，千万不要在你的Makefile中这样干，这里只是举个例子来表明Makefile中的变量在使用处展开的真实样子。可见其就是一个“替代”的原理。

另外，给变量加上括号完全是为了更加安全地使用这个变量，在上面的例子中，如果你不想给变量加上括号，那也可以，但我还是强烈建议你给变量加上括号。

## 变量中的变量

在定义变量的值时，我们可以使用其它变量来构造变量的值，在Makefile中有两种方式来在用变量定义变量的值。

先看第一种方式，也就是简单的使用“=”号，在“=”左侧是变量，右侧是变量的值，右侧变量的值可以定义在文件的任何一处，也就是说，右侧中的变量不一定非要是已定义好的值，其也可以使用后面定义的值。如：

foo = $(bar)

bar = $(ugh)

ugh = Huh?

all:

echo $(foo)

我们执行“make all”将会打出变量$(foo)的值是“Huh?”（ $(foo)的值是$(bar)，$(bar)的值是$(ugh)，$(ugh)的值是“Huh?”）可见，变量是可以使用后面的变量来定义的。

这个功能有好的地方，也有不好的地方，好的地方是，我们可以把变量的真实值推到后面来定义，如：

CFLAGS = $(include\_dirs) -O

include\_dirs = -Ifoo -Ibar

当“CFLAGS”在命令中被展开时，会是“-Ifoo -Ibar -O”。但这种形式也有不好的地方，那就是递归定义，如：

CFLAGS = $(CFLAGS) -O

或：

A = $(B)

B = $(A)

这会让make陷入无限的变量展开过程中去，当然，我们的make是有能力检测这样的定义，并会报错。还有就是如果在变量中使用函数，那么，这种方式会让我们的make运行时非常慢，更糟糕的是，他会使用到的两个make的函数“wildcard”和“shell”发生不可预知的错误。因为你不会知道这两个函数会被调用多少次。

为了避免上面的这种方法，我们可以使用make中的另一种用变量来定义变量的方法。这种方法使用的是“:=”操作符，如：

x := foo

y := $(x) bar

x := later

其等价于：

y := foo bar

x := later

值得一提的是，这种方法，前面的变量不能使用后面的变量，只能使用前面已定义好了的变量。如果是这样：

y := $(x) bar

x := foo

那么，y的值是“bar”，而不是“foo bar”。

上面都是一些比较简单的变量使用了，让我们来看一个复杂的例子，其中包括了make的函数、条件表达式和一个系统变量“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

关于条件表达式和函数，我们在后面再说，对于系统变量“MAKELEVEL”，其意思是，如果我们的make有一个嵌套执行的动作（参见前面的“嵌套使用make”），那么，这个变量会记录了我们的当前Makefile的调用层数。

下面再介绍两个定义变量时我们需要知道的，请先看一个例子，如果我们要定义一个变量，其值是一个空格，那么我们可以这样来：

nullstring :=

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

nullstring是一个Empty变量，其中什么也没有，而我们的space的值是一个空格。因为在操作符的右边是很难描述一个空格的，这里采用的技术很管用，先用一个Empty变量来标明变量的值开始了，而后面采用“#”注释符来表示变量定义的终止，这样，我们可以定义出其值是一个空格的变量。请注意这里关于“#”的使用，注释符“#”的这种特性值得我们注意，如果我们这样定义一个变量：

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

dir这个变量的值是“/foo/bar”，后面还跟了4个空格，如果我们这样使用这样变量来指定别的目录——“$(dir)/file”那么就完蛋了。

还有一个比较有用的操作符是“?=”，先看示例：

FOO ?= bar

其含义是，如果FOO没有被定义过，那么变量FOO的值就是“bar”，如果FOO先前被定义过，那么这条语句将什么也不做，其等价于：

ifeq ($(origin FOO), undefined)

FOO = bar

endif

## 变量高级用法

这里介绍两种变量的高级使用方法，第一种是变量值的替换。

我们可以替换变量中的共有的部分，其格式是“$(var:a=b)”或是“${var:a=b}”，其意思是，把变量“var”中所有以“a”字串“结尾”的“a”替换成“b”字串。这里的“结尾”意思是“空格”或是“结束符”。

还是看一个示例吧：

foo := a.o b.o c.o

bar := $(foo:.o=.c)

这个示例中，我们先定义了一个“$(foo)”变量，而第二行的意思是把“$(foo)”中所有以“.o”字串“结尾”全部替换成“.c”，所以我们的“$(bar)”的值就是“a.c b.c c.c”。

另外一种变量替换的技术是以“静态模式”（参见前面章节）定义的，如：

foo := a.o b.o c.o

bar := $(foo:%.o=%.c)

这依赖于被替换字串中的有相同的模式，模式中必须包含一个“%”字符，这个例子同样让$(bar)变量的值为“a.c b.c c.c”。

第二种高级用法是——“把变量的值再当成变量”。先看一个例子：

x = y

y = z

a := $($(x))

在这个例子中，$(x)的值是“y”，所以$($(x))就是$(y)，于是$(a)的值就是“z”。（注意，是“x=y”，而不是“x=$(y)”）

我们还可以使用更多的层次：

x = y

y = z

z = u

a := $($($(x)))

这里的$(a)的值是“u”，相关的推导留给读者自己去做吧。

让我们再复杂一点，使用上“在变量定义中使用变量”的第一个方式，来看一个例子：

x = $(y)

y = z

z = Hello

a := $($(x))

这里的$($(x))被替换成了$($(y))，因为$(y)值是“z”，所以，最终结果是：a:=$(z)，也就是“Hello”。

再复杂一点，我们再加上函数：

x = variable1

variable2 := Hello

y = $(subst 1,2,$(x))

z = y

a := $($($(z)))

这个例子中，“$($($(z)))”扩展为“$($(y))”，而其再次被扩展为“$($(subst 1,2,$(x)))”。$(x)的值是“variable1”，subst函数把“variable1”中的所有“1”字串替换成“2”字串，于是，“variable1”变成 “variable2”，再取其值，所以，最终，$(a)的值就是$(variable2)的值——“Hello”。（喔，好不容易）

在这种方式中，或要可以使用多个变量来组成一个变量的名字，然后再取其值：

first\_second = Hello

a = first

b = second

all = $($a\_$b)

这里的“$a\_$b”组成了“first\_second”，于是，$(all)的值就是“Hello”。

再来看看结合第一种技术的例子：

a\_objects := a.o b.o c.o

1\_objects := 1.o 2.o 3.o

sources := $($(a1)\_objects:.o=.c)

这个例子中，如果$(a1)的值是“a”的话，那么，$(sources)的值就是“a.c b.c c.c”；如果$(a1)的值是“1”，那么$(sources)的值是“1.c 2.c 3.c”。

再来看一个这种技术和“函数”与“条件语句”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))

这个示例中，如果定义了“do\_sort”，那么：foo := $(sort a d b g q c)，于是$(foo)的值就是 “a b c d g q”，而如果没有定义“do\_sort”，那么：foo := $(strip a d b g q c)，调用的就是strip函数。

当然，“把变量的值再当成变量”这种技术，同样可以用在操作符的左边：

dir = foo

$(dir)\_sources := $(wildcard $(dir)/\*.c)

define $(dir)\_print

lpr $($(dir)\_sources)

endef

这个例子中定义了三个变量：“dir”，“foo\_sources”和“foo\_print”。

## 追加变量值

我们可以使用“+=”操作符给变量追加值，如：

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

objects += another.o

于是，我们的$(objects)值变成：“main.o foo.o bar.o utils.o another.o”（another.o被追加进去了）

使用“+=”操作符，可以模拟为下面的这种例子：

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

objects := $(objects) another.o

所不同的是，用“+=”更为简洁。

如果变量之前没有定义过，那么，“+=”会自动变成“=”，如果前面有变量定义，那么“+=”会继承于前次操作的赋值符。如果前一次的是“:=”，那么“+=”会以“:=”作为其赋值符，如：

variable := value

variable += more

等价于：

variable := value

variable := $(variable) more

但如果是这种情况：

variable = value

variable += more

由于前次的赋值符是“=”，所以“+=”也会以“=”来做为赋值，那么岂不会发生变量的递补归定义，这是很不好的，所以make会自动为我们解决这个问题，我们不必担心这个问题。

## override 指示符

<http://theory.uwinnipeg.ca/localfiles/infofiles/make/make_66.html>

如果有变量是通过make的命令行参数设置的，那么Makefile中对这个变量的赋值会被忽略。如果你想在Makefile中设置这类参数的值，那么，你可以使用“override”指示符。其语法是：

override <variable>; = <value>;

override <variable>; := <value>;

当然，你还可以追加：

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

对于多行的变量定义，我们用define指示符，在define指示符前，也同样可以使用override指示符，如：

override define foo

bar

endef

## 多行变量

还有一种设置变量值的方法是使用define关键字。使用define关键字设置变量的值可以有换行，这有利于定义一系列的命令（前面我们讲过“命令包”的技术就是利用这个关键字）。

define指示符后面跟的是变量的名字，而重起一行定义变量的值，定义是以endef 关键字结束。其工作方式和“=”操作符一样。变量的值可以包含函数、命令、文字，或是其它变量。因为命令需要以[Tab]键开头，所以如果你用define定义的命令变量中没有以[Tab]键开头，那么make 就不会把其认为是命令。

下面的这个示例展示了define的用法：

define two-lines

echo foo

echo $(bar)

endef

变量的值在通常的赋值语句中只能在一行中完成，但在define指令中在define指令行以后endef行之前中间所有的行都是变量值的一部分（最后一行除外，因为标示endef那一行不能认为是变量值的一部分）。前面的例子功能上等同于： two-lines = echo foo; echo $(bar) 因为两命令之间用分号隔开，其行为很接近于两个分离的shell命令。然而，注意使用两个分离的行，意味着make请求shell两次，每一行都在独立的子shell中运行。参阅执行命令。 如果您希望使用define指令的变量定义比使用命令行定义的变量优先，您可以把define指令和override指令一块使用： override define two-lines foo $(bar) endef

## 环境变量

make运行时的系统环境变量可以在make开始运行时被载入到Makefile文件中，但是如果Makefile中已定义了这个变量，或是这个变量由make命令行带入，那么系统的环境变量的值将被覆盖。（如果make指定了“-e”参数，那么，系统环境变量将覆盖Makefile中定义的变量）

因此，如果我们在环境变量中设置了“CFLAGS”环境变量，那么我们就可以在所有的Makefile中使用这个变量了。这对于我们使用统一的编译参数有比较大的好处。如果Makefile中定义了CFLAGS，那么则会使用Makefile中的这个变量，如果没有定义则使用系统环境变量的值，一个共性和个性的统一，很像“全局变量”和“局部变量”的特性。

当make嵌套调用时（参见前面的“嵌套调用”章节），上层Makefile中定义的变量会以系统环境变量的方式传递到下层的Makefile 中。当然，默认情况下，只有通过命令行设置的变量会被传递。而定义在文件中的变量，如果要向下层Makefile传递，则需要使用export关键字来声明。（参见前面章节）

当然，我并不推荐把许多的变量都定义在系统环境中，这样，在我们执行不用的Makefile时，拥有的是同一套系统变量，这可能会带来更多的麻烦。

## 目标变量

前面我们所讲的在Makefile中定义的变量都是“全局变量”，在整个文件，我们都可以访问这些变量。当然，“自动化变量”除外，如“$<”等这种类量的自动化变量就属于“规则型变量”，这种变量的值依赖于规则的目标和依赖目标的定义。

当然，我也同样可以为某个目标设置局部变量，这种变量被称为“Target-specific Variable”，它可以和“全局变量”同名，因为它的作用范围只在这条规则以及连带规则中，所以其值也只在作用范围内有效。而不会影响规则链以外的全局变量的值。

其语法是：

<target ...> : <variable-assignment>;

<target ...> : overide <variable-assignment>

<variable-assignment>;可以是前面讲过的各种赋值表达式，如“=”、“:=”、“+=”或是“？=”。第二个语法是针对于make命令行带入的变量，或是系统环境变量。

这个特性非常的有用，当我们设置了这样一个变量，这个变量会作用到由这个目标所引发的所有的规则中去。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

在这个示例中，不管全局的$(CFLAGS)的值是什么，在prog目标，以及其所引发的所有规则中（prog.o foo.o bar.o的规则），$(CFLAGS)的值都是“-g”

## 模式变量

在GNU的make中，还支持模式变量（Pattern-specific Variable），通过上面的目标变量中，我们知道，变量可以定义在某个目标上。模式变量的好处就是，我们可以给定一种“模式”，可以把变量定义在符合这种模式的所有目标上。

我们知道，make的“模式”一般是至少含有一个“%”的，所以，我们可以以如下方式给所有以[.o]结尾的目标定义目标变量：

%.o : CFLAGS = -O

同样，模式变量的语法和“目标变量”一样：

<pattern ...>; : <variable-assignment>;

<pattern ...>; : override <variable-assignment>;

override同样是针对于系统环境传入的变量，或是make命令行指定的变量。