# 使用函数

在Makefile中可以使用函数来处理变量，从而让我们的命令或是规则更为的灵活和具有智能。make所支持的函数也不算很多，不过已经足够我们的操作了。函数调用后，函数的返回值可以当做变量来使用。

## 函数的调用语法

函数调用，很像变量的使用，也是以“$”来标识的，其語法如下：

$(<function> <arguments>)

或是

${<function> <arguments>}

这里，<function>就是函数名，make支持的函数不多。<arguments>为函数的参数，参数间以逗号“,”分隔，而函数名和参数之间以“空格”分隔。函数调用以“$”开头，以圆括号或花括号把函数名和参数括起。感觉很像一个变量，是不是？函数中的参数可以使用变量，为了风格的统一，函数和变量的括号最好一样，如使用“$(subst a,b,$(x))”这样的形式，而不是“$(subst a,b, ${x})”的形式。因为统一会更清楚，也会减少一些不必要的麻烦。

还是来看一个示例：

comma:= ,

empty:=

space:= $(empty) $(empty)

foo:= a b c

bar:= $(subst $(space),$(comma),$(foo))

在这个示例中，$(comma)的值是一个逗号。$(space)使用了$(empty)定义了一个空格，$(foo)的值是“a b c”，$ (bar)的定义用，调用了函数“subst”，这是一个替换函数，这个函数有三个参数，第一个参数是被替换字串，第二个参数是替换字串，第三个参数是替换操作作用的字串。这个函数也就是把$(foo)中的空格替换成逗号，所以$(bar)的值是“a,b,c”。

## 字符串处理函数

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

* 名称：字符串替换函数——subst。
* 功能：把字串<text>中的<from>字符串替换成<to>。
* 返回：函数返回被替换过后的字符串。
* 示例：

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

把“feet on the street”中的“ee”替换成“EE”，返回结果是“fEEt on the strEEt”。

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

* 名称：模式字符串替换函数——patsubst。
* 功能：查找<text>中的单词（单词以“空格”、“Tab”或“回车”“换行”分隔）是否符合模式< pattern>，如果匹配的话，则以<replacement>替换。这里，<pattern>可以包括通配符 “%”，表示任意长度的字串。如果<replacement>中也包含“%”，那么，<replacement>中的这个 “%”将是<pattern>中的那个“%”所代表的字串。（可以用“\”来转义，以“\%”来表示真实含义的“%”字符）
* 返回：函数返回被替换过后的字符串。
* 示例：

$(patsubst %.c,%.o,x.c.c bar.c)

把字串“x.c.c bar.c”符合模式[%.c]的单词替换成[%.o]，返回结果是“x.c.o bar.o”

备注：

这和我们前面“变量章节”说过的相关知识有点相似。如：

“$(var:<pattern>=<replacement>;)” 相当于 “$(patsubst <pattern>,<replacement>,$(var))”，

而“$(var: <suffix>=<replacement>)” 则相当于 “$(patsubst %<suffix>,%<replacement>,$(var))”。

例如有：objects = foo.o bar.o baz.o， 那么，“$(objects:.o=.c)”和“$(patsubst %.o,%.c,$(objects))”是一样的。

$(strip <string>)

* 名称：去空格函数——strip。
* 功能：去掉<string>;字串中开头和结尾的空字符。
* 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 ”去到开头和结尾的空格，结果是“a b c”。

$(findstring <find>,<in>)

* 名称：查找字符串函数——findstring。
* 功能：在字串<in>中查找<find>字串。
* 返回：如果找到，那么返回<find>，否则返回空字符串。
* 示例：

$(findstring a,a b c)

$(findstring a,b c)

第一个函数返回“a”字符串，第二个返回“”字符串（空字符串）

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

* 名称：过滤函数——filter。
* 功能：以<pattern>模式过滤<text>字符串中的单词，保留符合模式<pattern>的单词。可以有多个模式。
* 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>)

* 名称：反过滤函数——filter-out。
* 功能：以<pattern>模式过滤<text>字符串中的单词，去除符合模式<pattern>的单词。可以有多个模式。
* 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>)

* 名称：排序函数——sort。
* 功能：给字符串<list>中的单词排序（升序）。
* 返回：返回排序后的字符串。
* 示例：$(sort foo bar lose)返回“bar foo lose” 。
* 备注：sort函数会去掉<list>中相同的单词。

$(word <n>,<text>)

* 名称：取单词函数——word。
* 功能：取字符串<text>中第<n>个单词。（从一开始）
* 返回：返回字符串<text>中第<n>个单词。如果<n>比<text>中的单词数要大，那么返回空字符串。
* 示例：$(word 2, foo bar baz)返回值是“bar”。

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

* 名称：取单词串函数——wordlist。
* 功能：从字符串<text>中取从<ss>开始到<e>的单词串。<ss>和<e>是一个数字。
* 返回：返回字符串<text>中从<ss>到<e>的单词字串。如果<ss>比<text>中的单词数要大，那么返回空字符串。如果<e>大于<text>的单词数，那么返回从<ss>开始，到<text>结束的单词串。
* 示例： $(wordlist 2, 3, foo bar baz)返回值是“bar baz”。

$(words <text>)

* 名称：单词个数统计函数——words。
* 功能：统计<text>中字符串中的单词个数。
* 返回：返回<text>中的单词数。
* 示例：$(words foo bar baz)返回值是“3”。
* 备注：如果我们要取<text>中最后的一个单词，我们可以这样：$(word $(words <text>),<text>)。

$(firstword <text>)

* 名称：首单词函数——firstword。
* 功能：取字符串<text>中的第一个单词。
* 返回：返回字符串<text>的第一个单词。
* 示例：$(firstword foo bar)返回值是“foo”。
* 备注：这个函数可以用word函数来实现：$(word 1,<text>)。

以上，是所有的字符串操作函数，如果搭配混合使用，可以完成比较复杂的功能。这里，举一个现实中应用的例子。我们知道，make使用“VPATH”变量来指定“依赖文件”的搜索路径。于是，我们可以利用这个搜索路径来指定编译器对头文件的搜索路径参数CFLAGS，如：

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

如果我们的“$(VPATH)”值是“src:../headers”，那么“$(patsubst %,-I%,$(subst :, ,$(VPATH)))”将返回“-Isrc -I../headers”，这正是cc或gcc搜索头文件路径的参数

## 文件名操作函数

下面我们要介绍的函数主要是处理文件名的。每个函数的参数字符串都会被当做一个或是一系列的文件名来对待。

$(dir <names...>)

* 名称：取目录函数——dir。
* 功能：从文件名序列<names>中取出目录部分。目录部分是指最后一个反斜杠（“/”）之前的部分。如果没有反斜杠，那么返回“./”。
* 返回：返回文件名序列<names>的目录部分。
* 示例： $(dir src/foo.c hacks)返回值是“src/ ./”。

$(notdir <names...>)

* 名称：取文件函数——notdir。
* 功能：从文件名序列<names>中取出非目录部分。非目录部分是指最後一个反斜杠（“/”）之后的部分。
* 返回：返回文件名序列<names>的非目录部分。
* 示例： $(notdir src/foo.c hacks)返回值是“foo.c hacks”。

$(suffix <names...>)

* 名称：取後缀函数——suffix。
* 功能：从文件名序列<names>中取出各个文件名的后缀。
* 返回：返回文件名序列<names>的后缀序列，如果文件没有后缀，则返回空字串。
* 示例：$(suffix src/foo.c src-1.0/bar.c hacks)返回值是“.c .c”。

$(basename <names...>)

* 名称：取前缀函数——basename。
* 功能：从文件名序列<names>中取出各个文件名的前缀部分。
* 返回：返回文件名序列<names>的前缀序列，如果文件没有前缀，则返回空字串。
* 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...>)

* 名称：加后缀函数——addsuffix。
* 功能：把后缀<suffix>加到<names>中的每个单词后面。
* 返回：返回加过后缀的文件名序列。
* 示例：$(addsuffix .c,foo bar)返回值是“foo.c bar.c”。

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

* 名称：加前缀函数——addprefix。
* 功能：把前缀<prefix>加到<names>中的每个单词后面。
* 返回：返回加过前缀的文件名序列。
* 示例：$(addprefix src/,foo bar)返回值是“src/foo src/bar”。

$(join <list1>,<list2>)

* 名称：连接函数——join。
* 功能：把<list2>中的单词对应地加到<list1>的单词后面。如果<list1>的单词个数要比<list2>的多，那么，<list1>中的多出来的单词将保持原样。如果<list2>的单词个数要比<list1>多，那么，<list2>多出来的单词将被复制到<list1>中。
* 返回：返回连接过后的字符串。
* 示例：$(join aaa bbb , 111 222 333)返回值是“aaa111 bbb222 333”。

## foreach 函数

foreach函数和别的函数非常的不一样。因为这个函数是用来做循环用的，Makefile中的foreach函数几乎是仿照于Unix标准 Shell（/bin/sh）中的for语句，或是C-Shell（/bin/csh）中的foreach语句而构建的。它的语法是：

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

这个函数的意思是，把参数<list>中的单词逐一取出放到参数所指定的变量中，然后再执行< text>所包含的表达式。每一次<text>会返回一个字符串，循环过程中，<text>的所返回的每个字符串会以空格分隔，最后当整个循环结束时，<text>所返回的每个字符串所组成的整个字符串（以空格分隔）将会是foreach函数的返回值。

所以，最好是一个变量名，<list>可以是一个表达式，而<text>中一般会使用这个参数来依次枚举<list>中的单词。举个例子：

names := a b c d

files := $(foreach n,$(names),$(n).o)

上面的例子中，$(name)中的单词会被挨个取出，并存到变量“n”中，“$(n).o”每次根据“$(n)”计算出一个值，这些值以空格分隔，最后作为foreach函数的返回，所以，$(files)的值是“a.o b.o c.o d.o”。

注意，foreach中的参数是一个临时的局部变量，foreach函数执行完后，参数的变量将不在作用，其作用域只在foreach函数当中。

## if 函数

if函数很像GNU的make所支持的条件语句——ifeq（参见前面所述的章节），if函数的语法是：

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

或是

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

可见，if函数可以包含“else”部分，或是不含。即if函数的参数可以是两个，也可以是三个。<condition>参数是 if的表达式，如果其返回的为非空字符串，那么这个表达式就相当于返回真，于是，<then-part>会被计算，否则<else- part>会被计算。

而if函数的返回值是，如果<condition>为真（非空字符串），那个<then-part>会是整个函数的返回值，如果<condition>为假（空字符串），那么<else-part>会是整个函数的返回值，此时如果< else-part>没有被定义，那么，整个函数返回空字串。

所以，<then-part>和<else-part>只会有一个被计算。

## call函数

call函数是唯一一个可以用来创建新的参数化的函数。你可以写一个非常复杂的表达式，这个表达式中，你可以定义许多参数，然后你可以用call函数来向这个表达式传递参数。其语法是：

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

当make执行这个函数时，<expression>参数中的变量，如$(1)，$(2)，$(3)等，会被参数< parm1>，<parm2>，<parm3>依次取代。而<expression>的返回值就是 call函数的返回值。例如：

reverse = $(1) $(2)

foo = $(call reverse,a,b)

那么，foo的值就是“a b”。当然，参数的次序是可以自定义的，不一定是顺序的，如：

reverse = $(2) $(1)

foo = $(call reverse,a,b)

此时的foo的值就是“b a”。

## origin函数

origin函数不像其它的函数，他并不操作变量的值，他只是告诉你你的这个变量是哪里来的？其语法是：

$(origin <variable>;)

注意，<variable>;是变量的名字，不应该是引用。所以你最好不要在<variable>;中使用“$”字符。Origin函数会以其返回值来告诉你这个变量的“出生情况”，下面，是origin函数的返回值:

“undefined”

如果<variable>;从来没有定义过，origin函数返回这个值“undefined”。

“default”

如果<variable>;是一个默认的定义，比如“CC”这个变量，这种变量我们将在后面讲述。

“environment”

如果<variable>;是一个环境变量，并且当Makefile被执行时，“-e”参数没有被打开。

“file”

如果<variable>;这个变量被定义在Makefile中。

“command line”

如果<variable>;这个变量是被命令行定义的。

“override”

如果<variable>;是被override指示符重新定义的。

“automatic”

如果<variable>;是一个命令运行中的自动化变量。关于自动化变量将在后面讲述。

这些信息对于我们编写Makefile是非常有用的，例如，假设我们有一个Makefile其包了一个定义文件Make.def，在 Make.def中定义了一个变量“bletch”，而我们的环境中也有一个环境变量“bletch”，此时，我们想判断一下，如果变量来源于环境，那么我们就把之重定义了，如果来源于Make.def或是命令行等非环境的，那么我们就不重新定义它。于是，在我们的Makefile中，我们可以这样写：

ifdef bletch

ifeq "$(origin bletch)" "environment"

bletch = barf, gag, etc.

endif

endif

当然，你也许会说，使用override关键字不就可以重新定义环境中的变量了吗？为什么需要使用这样的步骤？是的，我们用override是可以达到这样的效果，可是override过于粗暴，它同时会把从命令行定义的变量也覆盖了，而我们只想重新定义环境传来的，而不想重新定义命令行传来的。

## shell函数

shell函数也不像其它的函数。顾名思义，它的参数应该就是操作系统Shell的命令。它和反引号“`”是相同的功能。这就是说，shell函数把执行操作系统命令后的输出作为函数返回。于是，我们可以用操作系统命令以及字符串处理命令awk，sed等等命令来生成一个变量，如：

contents := $(shell cat foo)

files := $(shell echo \*.c)

注意，这个函数会新生成一个Shell程序来执行命令，所以你要注意其运行性能，如果你的Makefile中有一些比较复杂的规则，并大量使用了这个函数，那么对于你的系统性能是有害的。特别是Makefile的隐晦的规则可能会让你的shell函数执行的次数比你想像的多得多。

## 控制make的函数

make提供了一些函数来控制make的运行。通常，你需要检测一些运行Makefile时的运行时信息，并且根据这些信息来决定，你是让make继续执行，还是停止。

$(error <text ...>;)

产生一个致命的错误，<text ...>;是错误信息。注意，error函数不会在一被使用就会产生错误信息，所以如果你把其定义在某个变量中，并在后续的脚本中使用这个变量，那么也是可以的。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)

示例一会在变量ERROR\_001定义了后执行时产生error调用，而示例二则在目录err被执行时才发生error调用。

$(warning <text ...>;)

这个函数很像error函数，只是它并不会让make退出，只是输出一段警告信息，而make继续执行。