6.2 The Two Flavors of Variables

There are two ways that a variable in GNU make can have a value; we call them the two flavors of variables. The two flavors are distinguished in how they are defined and in what they do when expanded.

GNU make 中的变量有两种取值方式； 我们称它们为两种类型的变量。 这两种风味的区别在于它们的定义方式和扩展时的作用。

The first flavor of variable is a recursively expanded variable. Variables of this sort are defined by lines using '=' (see Setting Variables) or by the define directive (see Defining Multi-Line Variables). The value you specify is installed verbatim; if it contains references to other variables, these references are expanded whenever this variable is substituted (in the course of expanding some other string). When this happens, it is called recursive expansion.

第一种变量是递归扩展变量。 此类变量由使用 "=" 的行定义（请参阅设置变量）或定义指令（请参阅定义多行变量）。 您指定的值是逐字安装的； 如果它包含对其他变量的引用，则只要替换此变量（在扩展某些其他字符串的过程中），就会扩展这些引用。 发生这种情况时，称为递归扩展。

For example,

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

all:;echo $(foo)
```

will echo 'Huh?': '$(foo)' expands to '$(bar)' which expands to '$(ugh)' which finally expands to 'Huh?'.

将回显 "Huh？ "： "$(foo)" 扩展为 "$(bar)"，然后扩展为 "$(ugh)"，最终扩展为 "Huh？ "。

This flavor of variable is the only sort supported by most other versions of make. It has its advantages and its disadvantages. An advantage (most would say) is that:

这种类型的变量是大多数其他版本的 make 支持的唯一类型。 它有它的优点和缺点。 一个优势（大多数人会说）是：

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

will do what was intended: when ‘CFLAGS’ is expanded in a recipe, it will expand to ‘-Ifoo -Ibar -O’. A major disadvantage is that you cannot append something on the end of a variable, as in

---

将按预期进行：当 “CFLAGS” 在配方中扩展时，它将扩展为”-Ifoo -Ibar -O”。 一个主要的缺点是你不能在变量的末尾附加一些东西，如

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

because it will cause an infinite loop in the variable expansion. (Actually make detects the infinite loop and reports an error.)

因为它会导致变量扩展的无限循环。 （实际上make检测到死循环并报错。）

Another disadvantage is that any functions (see Functions for Transforming Text) referenced in the definition will be executed every time the variable is expanded. This makes make run slower; worse, it causes the wildcard and shell functions to give unpredictable results because you cannot easily control when they are called, or even how many times.

另一个缺点是每次扩展变量时都会执行定义中引用的任何函数（请参阅转换文本的函数）。 这使得 make 运行速度变慢； 更糟糕的是，它会导致通配符和 shell 函数给出不可预测的结果，因为您无法轻松控制它们何时被调用，甚至调用多少次。

To avoid all the problems and inconveniences of recursively expanded variables, there is another flavor: simply expanded variables.

为了避免递归扩展变量的所有问题和不便，还有另一种风格：简单扩展变量。

Simply expanded variables are defined by lines using ‘:=’ or ‘::=’ (see Setting Variables). Both forms are equivalent in GNU make; however only the ‘::=’ form is described by the POSIX standard (support for ‘::=’ was added to the POSIX standard in 2012, so older versions of make won’t accept this form either).

简单扩展的变量由使用 ‘:=’ 或 ‘::=’ 的行定义（参见设置变量）。 这两种形式在 GNU make 中是等价的； 然而，POSIX 标准只描述了 “::=” 形式（对 “::=” 的支持在 2012 年被添加到 POSIX 标准中，因此旧版本的 make 也不会接受这种形式）。

The value of a simply expanded variable is scanned once and for all, expanding any references to other variables and functions, when the variable is defined. The actual value of the simply expanded variable is the result of expanding the text that you write. It does not contain any references to other variables; it contains their values as of the time this variable was defined. Therefore,

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

is equivalent to

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

When a simply expanded variable is referenced, its value is substituted verbatim.

一个简单扩展变量的值被扫描一次，当变量被定义时，扩展对其他变量和函数的任何引用。 简单扩展变量的实际值是扩展您编写的文本的结果。 它不包含对其他变量的任何引用； 它包含它们在定义此变量时的值。 因此，当引用一个简单扩展的变量时，它的值被逐字替换。

Here is a somewhat more complicated example, illustrating the use of ‘:=’ in conjunction with the shell function. (See The shell Function.) This example also shows use of the variable MAKELEVEL, which is changed when it is passed down from level to level. (See Communicating Variables to a Sub-make, for information about MAKELEVEL.)

下面是一个稍微复杂一些的例子，说明 ‘:=’ 与 shell 函数的结合使用。 （请参阅 shell 函数。）此示例还显示了变量 MAKELEVEL 的使用，当它从一个级别向下传递到另一个级别时会更改。 （有关 MAKELEVEL 的信息，请参阅将变量传递给子制作。）

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

An advantage of this use of ‘:=’ is that a typical ‘descend into a directory’ recipe then looks like this:

使用 ‘:=’ 的一个优点是典型的‘下降到目录’配方看起来像这样：

```
${subdirs}:
        ${MAKE} -C $@ all
```

Simply expanded variables generally make complicated makefile programming more predictable because they work like variables in most programming languages. They allow you to redefine a variable using its own value (or its value processed in some way by one of the expansion functions) and to use the expansion functions much more efficiently (see Functions for Transforming Text).

简单扩展的变量通常会使复杂的 makefile 编程更容易预测，因为它们在大多数编程语言中就像变量一样工作。它们允许您使用变量自己的值（或由扩展函数以某种方式处理的值）重新定义变量，并更有效地使用扩展函数（请参阅转换文本的函数）。

You can also use them to introduce controlled leading whitespace into variable values. Leading whitespace characters are discarded from your input before substitution of variable references and function calls; this means you can include leading spaces in a variable value by protecting them with variable references, like this:

您还可以使用它们将受控的前导空格引入变量值。 在替换变量引用和函数调用之前，会从输入中丢弃前导空白字符； 这意味着您可以通过使用变量引用保护它们来在变量值中包含前导空格，如下所示：

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

Here the value of the variable space is precisely one space. The comment ‘# end of the line’ is included here just for clarity. Since trailing space characters are not stripped from variable values, just a space at the end of the line would have the same effect (but be rather hard to read). If you put whitespace at the end of a variable value, it is a good idea to put a comment like that at the end of the line to make your intent clear. Conversely, if you do not want any whitespace characters at the end of your variable value, you must remember not to put a random comment on the end of the line after some whitespace, such as this:

这里变量空间的值正好是一个空格。 为了清楚起见，此处包含注释"# end of the line"。 由于尾随空格字符不会从变量值中删除，因此行尾的空格将具有相同的效果（但很难阅读）。 如果您将空格放在变量值的末尾，最好在行尾添加这样的注释以明确您的意图。 相反，如果您不希望在变量值的末尾出现任何空白字符，则必须记住不要在行尾的一些空白之后放置随机注释，例如：

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

Here the value of the variable dir is '/foo/bar ' (with four trailing spaces), which was probably not the intention. (Imagine something like '$(dir)/file' with this definition!)

这里变量 dir 的值是 "/foo/bar"（有四个尾随空格），这可能不是本意。 （想象一下像 "$(dir)/file"这样的定义！）

There is another assignment operator for variables, '?='. This is called a conditional variable assignment operator, because it only has an effect if the variable is not yet defined. This statement:

变量还有另一个赋值运算符 '?=' 。 这称为条件变量赋值运算符，因为它仅在变量尚未定义时才有效。 这个说法：

```
FOO ?= bar
```

---

is exactly equivalent to this (see The origin Function):

完全等价于这个（参见原点函数）：

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

Note that a variable set to an empty value is still defined, so '?=' will not set that variable.

请注意，仍然定义了设置为空值的变量，因此 "?=" 不会设置该变量。
