第十二章

makefile 的调试

makefile 的调试有点像魔法。可惜，并不存在 *makefile* 调试器之类的东西可用来查看特定规则是如何被求值的，或某个变量是如何被扩展的。相反，大部分的调试过程只是在执行输出的动作以及查看 *makefile*。事实上，GNU `make` 提供了若干可以协助调试的内置函数以及命令行选项。

用来调试 *makefile* 的一个最好方法就是加入调试挂钩以及使用具保护的编程技术，让你能够在事情出错时恢复原状。我将会介绍若干基本的调试技术以及我所发现的最有用的具保护能力的编码习惯。

make 的调试功能

`warning` 函数非常适合用来调试难以捉摸的 *makefile*。因为 `warning` 函数会被扩展成空字符串，所以它可以放在 *makefile* 中的任何地方：开始的位置、工作目标或必要条件列表中以及命令脚本中。这让你能够在最方便查看变量的地方输出变量的值。例如：

```
$(warning A top-level warning)

FOO := $(warning Right-hand side of a simple variable)bar
BAZ = $(warning Right-hand side of a recursive variable)boo

$(warning A target)target: $(warning In a prerequisite list)makefile
$(BAZ)
        $(warning In a command script)
        ls
$(BAZ):
```

这会产生如下的输出：

```
$ make
makefile:1: A top-level warning
```

```
makefile:2: Right-hand side of a simple variable
makefile:5: A target
makefile:5: In a prerequisite list
makefile:5: Right-hand side of a recursive variable
makefile:8: Right-hand side of a recursive variable
makefile:6: In a command script
ls
makefile
```

请注意，warning函数的求值方式是按照make标准的立即和延后求值算法。虽然对BAZ的赋值动作中包含了一个warning函数，但是直到BAZ在必要条件列表中被求值后，这个信息才会被输出来。

“可以在任何地方安插warning调用”的这个特性，让它能够成为一个基本的调试工具。

命令行选项

我找到了三个最适合用来调试的命令行选项：--just-print（-n）、--print-database（-p）以及--warn-undefined-variables。

--just-print

在一个新的*makefile*工作目标上，我所做的第一个测试就是以--just-print（-n）选项来调用make。这会使得make读进*makefile*并且输出它更新工作目标时将会执行的命令，但是不会真的执行它们。GNU make有一个方便的功能，就是允许你为将被输出的命令标上安静模式修饰符（@）。

这个选项被假设可以抑制所有命令的执行动作，然而这只在特定的状况下为真。实际上，你必须小心以对。尽管make不会运行命令脚本，但是在立即的语境之中，它会对shell函数调用进行求值动作。例如：

```
REQUIRED_DIRS = ...
_MKDIRS := $(shell for d in $(REQUIRED_DIRS); \
             do                               \
               [[ -d $$d ]] || mkdir -p $$d;  \
             done)

$(objects) : $(sources)
```

正如我们之前所见，_MKDIRS简单变量的目的是触发必要目录的创建动作。如果这个*makefile*是以--just-print选项的方式运行的，那么当make读进*makefile*时，shell命令将会一如往常般被执行。然后，make将会输出（但不会执行）更新$(objects)文件列表所需要进行的每个编译命令。

--print-data-base

`--print-data-base`（`-p`）是另一个你常会用到的选项。它会运行*makefile*，显示GNU版权信息以及`make`所运行的命令，然后输出它的内部数据库。数据库里的数据将会依种类划分成以下几个组：variables、directories、implicit rules、pattern-specific variables、files（explicit rules）以及 vpath search path。如下所示：

```
# GNU Make 3.80
# Copyright (C) 2002  Free Software Foundation, Inc.
# This is free software; see the source for copying conditions.
# There is NO warranty; not even for MERCHANTABILITY or FITNESS FOR A
# PARTICULAR PURPOSE.
正常的命令将会在此处执行

# Make data base, printed on Thu Apr 29 20:58:13 2004
# Variables
...
# Directories
...
# Implicit Rules
...
# Pattern-specific variable values
...
# Files
...
# VPATH Search Paths
```

让我们更详细地查看以上这几个区段。

变量区段（variable）将会列出每个变量以及具描述性的注释：

```
# automatic
<D = $(patsubst %/,%,$(dir $<))
# environment
EMACS_DIR = C:/usr/emacs-21.3.50.7
# default
CWEAVE = cweave
# makefile (from `../mp3_player/makefile', line 35)
CPPFLAGS = $(addprefix -I ,$(include_dirs))
# makefile (from `../ch07-separate-binaries/makefile', line 44)
RM := rm -f
# makefile (from `../mp3_player/makefile', line 14)
define make-library
  libraries += $1
  sources   += $2

  $1: $(call source-to-object,$2)
        $(AR) $(ARFLAGS) $$@ $$^
endef
```

自动变量不会被显示出来，但是通过它们可以方便变量的获得，像$(<D)。注释所指出的是origin函数所返回的变量类型（参见“较不重要的杂项函数”一节）。如果变量被定义在一个文件中，则会在注释中指出其文件名以及该定义所在的行号。简单变量和递归变量的差别在于赋值运算符。简单变量的值将会被显示成右边部分被求值的形式。

下一个区段标示为Directories，它对make开发人员比对make用户有用。它列出了将会被make检查的目录，包括可能会存在的SCCS和RCS子目录，但它们通常不存在。对每个目录来说，make会显示实现细节，比如设备编号、inode以及文件名模式匹配的统计数据。

接着是Implicit Rules区段。这个区段包含了make数据库中所有的内置的和用户自定义的模式规则。此外，对于那些定义在文件中的规则，它们的注释将会指出文件名以及行号：

```
%.c %.h: %.y
# commands to execute (from `../mp3_player/makefile', line 73):
        $(YACC.y) --defines $<
        $(MV) y.tab.c $*.c
        $(MV) y.tab.h $*.h

%: %.c
#  commands to execute (built-in):
        $(LINK.c) $^ $(LOADLIBES) $(LDLIBS) -o $@

%.o: %.c
#  commands to execute (built-in):
        $(COMPILE.c) $(OUTPUT_OPTION) $<
```

查看这个区段，是让你能够熟悉make内置规则的变化和结构的最佳方法。当然，并非所有的内置规则都会被实现成模式规则。如果你没有找到你想要的规则，可以查看Files区段，旧式后缀规则就列在该处。

下一个区段被标示为Pattern-specific variables，此处所列出的是定义在*makefile*里的模式专属变量。所谓模式专属变量，就是变量定义的有效范围被限定在相关的模式规则执行的时候。例如，模式变量YYLEXFLAG被定义成：

```
%.c %.h: YYLEXFLAG := -d
%.c %.h: %.y
        $(YACC.y) --defines $<
        $(MV) y.tab.c $*.c
        $(MV) y.tab.h $*.h
```

将会被显示成：

```
# Pattern-specific variable values
```

```
%.c :
# makefile (from `Makefile', line 1)
# YYLEXFLAG := -d
# variable set hash-table stats:
# Load=1/16=6%, Rehash=0, Collisions=0/1=0%
%.h :
# makefile (from `Makefile', line 1)
# YYLEXFLAG := -d
# variable set hash-table stats:
# Load=1/16=6%, Rehash=0, Collisions=0/1=0%

# 2 pattern-specific variable values
```

接着是 Files 区段，此处所列出的都是与特定文件有关的自定义和后缀规则：

```
# Not a target:
.p.o:
#  Implicit rule search has not been done.
#  Modification time never checked.
#  File has not been updated.
#  commands to execute (built-in):
        $(COMPILE.p) $(OUTPUT_OPTION) $<

lib/ui/libui.a: lib/ui/ui.o
#  Implicit rule search has not been done.
#  Last modified 2004-04-01 22:04:09.515625
#  File has been updated.
#  Successfully updated.
#  commands to execute (from `../mp3_player/lib/ui/module.mk', line 3):
        ar rv $@ $^

lib/codec/codec.o: ../mp3_player/lib/codec/codec.c ../mp3_player/lib/codec/
    codec.c ../mp3_player/include/codec/codec.h
#  Implicit rule search has been done.
#  Implicit/static pattern stem: `lib/codec/codec'
#  Last modified 2004-04-01 22:04:08.40625
#  File has been updated.
#  Successfully updated.
#  commands to execute (built-in):
        $(COMPILE.c) $(OUTPUT_OPTION) $<
```

中间文件与后缀规则会被标示为 Not a target，其余是工作目标。每个文件将会包含注释，用以指出 `make` 是如何处理此规则的。被找到的文件在被显示的时候将会通过标准的 `vpath` 搜索来找出其路径。

最后一个区段被标示为 VPATH Search Paths，列出了 `VPATH` 的值以及所有的 `vpath` 模式。

对于大规模使用 `eval` 以及用户自定义函数来建立复杂的变量和规则的 *makefile* 来说，查看它们的输出结果通常是确认宏是否已被扩展成预期值的唯一方法。

--warn-undefined-variables

这个选项会使得make在未定义的变量被扩展时显示警告信息。因为未定义的变量会被扩展成空字符串，这常见于变量名称打错而且很长一段时间未被发现到。这个选项有个问题，这也是为什么我很少使用这个选项的原因，那就是许多内置规则都会包含未定义的变量以作为用户自定义值的挂钩。所以使用这个选项来运行make必然会产生许多不是错误的警告信息，而且对用户的*makefile*没有什么用处。例如：

```
$ make --warn-undefined-variables -n
makefile:35: warning: undefined variable MAKECMDGOALS
makefile:45: warning: undefined variable CFLAGS
makefile:45: warning: undefined variable TARGET_ARCH
...
makefile:35: warning: undefined variable MAKECMDGOALS
make: warning: undefined variable CFLAGS
make: warning: undefined variable TARGET_ARCH
make: warning: undefined variable CFLAGS
make: warning: undefined variable TARGET_ARCH
...
make: warning: undefined variable LDFLAGS
make: warning: undefined variable TARGET_ARCH
make: warning: undefined variable LOADLIBES
make: warning: undefined variable LDLIBS
```

不过，此命令在需要捕获此类错误的某些场合上可能非常有用。

--debug选项

当你需要知道make如何分析你的依存图时，可以使用--debug选项。除了运行调试器，这个选项是让你获得最详细信息的另一个方法。你有五个调试选项以及一个修饰符可用，分别是：basic、verbose、implicit、jobs、all以及makefile。

如果调试选项被指定成--debug，就是在进行basic调试；如果调试选项被指定成-d，就是在进行all调试；如果要使用选项的其他组合，则可以使用--debug=*option1,option2*这个以逗号为分隔符的列表，此处的选项可以是下面任何一个单词（实际上，make只会查看第一个字母）：

basic

这是所提供的信息最不详细的基本调试功能。启用时，make会输出被发现尚未更新的工作目标并更新动作的状态。它的输出会像下面这样：

```
File all does not exist.
  File app/player/play_mp3 does not exist.
    File app/player/play_mp3.o does not exist.
   Must remake target app/player/play_mp3.o.
```

```
gcc ... ../mp3_player/app/player/play_mp3.c
   Successfully remade target file app/player/play_mp3.o.
```

`verbose`

这个选项会设定 `basic` 选项，以及提供关于“哪些文件被分析、哪些必要条件不需要重建等”的额外信息：

```
File all does not exist.
 Considering target file app/player/play_mp3.
  File app/player/play_mp3 does not exist.
   Considering target file app/player/play_mp3.o.
    File app/player/play_mp3.o does not exist.
     Pruning file ../mp3_player/app/player/play_mp3.c.
     Pruning file ../mp3_player/app/player/play_mp3.c.
     Pruning file ../mp3_player/include/player/play_mp3.h.
    Finished prerequisites of target file app/player/play_mp3.o.
   Must remake target app/player/play_mp3.o.
gcc ... ../mp3_player/app/player/play_mp3.c
   Successfully remade target file app/player/play_mp3.o.
   Pruning file app/player/play_mp3.o.
```

`implicit`

这个选项会设定 `basic` 选项，以及提供关于“为每个工作目标搜索隐含规则”的额外信息：

```
File all does not exist.
  File app/player/play_mp3 does not exist.
  Looking for an implicit rule for app/player/play_mp3.
  Trying pattern rule with stem play_mp3.
  Trying implicit prerequisite app/player/play_mp3.o.
  Found an implicit rule for app/player/play_mp3.
    File app/player/play_mp3.o does not exist.
    Looking for an implicit rule for app/player/play_mp3.o.
    Trying pattern rule with stem play_mp3.
    Trying implicit prerequisite app/player/play_mp3.c.
    Found prerequisite app/player/play_mp3.c as VPATH ../mp3_player/app/
player/play_mp3.c
    Found an implicit rule for app/player/play_mp3.o.
   Must remake target app/player/play_mp3.o.
gcc ... ../mp3_player/app/player/play_mp3.c
   Successfully remade target file app/player/play_mp3.o.
```

`jobs`

这个选项会输出被 `make` 调用的子进程的细节，它不会启用 `basic` 选项的功能。

```
Got a SIGCHLD; 1 unreaped children.
gcc ... ../mp3_player/app/player/play_mp3.c
Putting child 0x10033800 (app/player/play_mp3.o) PID 576 on the chain.
Live child 0x10033800 (app/player/play_mp3.o) PID 576
Got a SIGCHLD; 1 unreaped children.
Reaping winning child 0x10033800 PID 576
Removing child 0x10033800 PID 576 from chain.
```

`all`

这会启用前面的所有选项，当你使用 `-d` 选项时，默认会启用此功能。

`makefile`

它不会启用调试信息，直到 *makefile* 被更新 —— 这包括更新任何的引入文件。如果使用此修饰符，`make` 会在重编译 *makefile* 以及引入文件的时候，输出被选择的信息。这个选项会启用 `basic` 选项，`all` 选项也会启用此选项。

编写用于调试的代码

如你所见，并没有太多的工具可用来调试 *makefile*，你只有几个方法可以输出若干可能有用的信息。当这些方法都不管用时，你就得将 *makefile* 编写成可以尽量减少错误发生的机会，或是可以为自己提供一个舞台来协助你进行调试。

这一节所提供的建议被我（有点随意地）分类成：良好的编码习惯、具保护功能的编码以及调试技术等部分。然而一些特殊的项目，像是检查命令的结束状态，可能会被放在良好的编码习惯中或是具保护功能的编码中，做这样的分类适当地反映出了趋势所在。将焦点好好地放在 *makefile* 上，尽量避免简单行事。采用具保护的编码以避免 *makefile* 被非预期的事件和环境状态所影响。最后，当缺陷出现时，使用你可以找到的用来压制它们的每个诀窍。

“ 简洁就是美 ”（Keep It Simple）的原则（*http://www.catb.org/~esr/jargon/html/K/KISS-Principle.html*）是所有良好设计的核心所在。正如你在前面几章所看到的，*makefile* 马上就会变得很复杂 —— 即使是一般的工作，比如依存关系的产生。要对抗 “ 在你的编译系统中加入越来越多的功能 ”的潮流，你将会失败，但如果你只是不经思索地加入你所发现的每个功能，失败并不会比你这么做的后果还糟。

良好的编码习惯

以我的经验来说，大部分的程序员都不会把 *makefile* 作为程序来写，因此，他们不会像编写 C++ 或 Java 时那样细心。事实上，`make` 语言是一个完整的非程序语言。如果可靠性和可维护性对你的编译系统来说很重要，那么请小心编写你的 *makefile*，并且尽量遵守良好的编码习惯。

编码健全的 *makefile* 的重点之一就是检查命令的返回状态。当然，`make` 将会自动检查简单的命令，但是 *makefile* 通常会使用可能不会处理失败状态的复合命令：

```
do:
        cd i-dont-exist; \
        echo *.c
```

运行时，此 *makefile* 并不会因为有错误发生而终止运行，尽管这是一个必然会发生的错误：

```
$ make
cd i-dont-exist; \
echo *.c
/bin/sh: line 1: cd: i-dont-exist: No such file or directory
*.c
```

此外，当文件名匹配表达式（globbing expression）找不到任何的 *.c* 文件时，它会不动声色地返回文件名匹配表达式。一个比较好的做法，就是在你编码此命令脚本时，使用 shell 的功能来检查以及防止错误：

```
SHELL = /bin/bash
do:
        cd i-dont-exist && \
        shopt -s nullglob &&
        echo *.c
```

现在 `cd` 的错误会被正确传送到 `make`，所以 `echo` 命令不会被执行，而且 `make` 会因为有错误发生而终止运行。此外，设定 `bash` 的 `nullglob` 选项，将会使得文件名匹配模式在找不到文件时返回空字符串。（当然，你的应用程序可能比较喜欢文件名匹配模式。）

```
$ make
cd i-dont-exist && \
echo *.c
/bin/sh: line 1: cd: i-dont-exist: No such file or directory
make: *** [do] Error 1
```

另一个良好的编码习惯，就是将你的代码编排成最具可读性的形式。我所看过的 *makefile*，多半编排得很差，这必然会造成难以阅读的情况。下面这两段代码哪一个比较容易阅读？

```
_MKDIRS := $(shell for d in $(REQUIRED_DIRS); do [[ -d $$d \
]] || mkdir -p $$d; done)
```

或：

```
_MKDIRS := $(shell                              \
             for d in $(REQUIRED_DIRS);         \
             do                                 \
               [[ -d $$d ]] || mkdir -p $$d;    \
             done)
```

如果你像大部分人那样，你将会觉得第一段代码比较难分析，不容易找到分号，很难计算有几句语句。这些都是必须注意到的地方。在命令脚本中，你会遇到的语法错误，多半是由于漏掉了分号、反斜线或是其他的分隔符，比如管道（pipe）和逻辑运算符。

此外请注意，并非任何分隔符被漏掉都会产生错误。例如，下面的错误都不会产生shell的语法错误：

```
TAGS:
        cd src \
        ctags --recurse

disk_free:
        echo "Checking free disk space..." \
        df . | awk '{ print $$4 }'
```

把命令编排得具有可读性，将会让以上所提到的错误很容易被发现。编排用户自定义函数的时候可以采用内缩的做法。有时候，宏扩展后的结果中，额外的空格将会造成问题。如果是这样，你可以将它的编排结果封装在`strip`函数的调用中。编排一长串值时，你可以让每个值自成一行。在每个工作目标的前面加上注释，可以提供简介以及说明参数列表。

下一个良好的编码习惯就是大量使用变量来保存常用的值。如同在程序中一样，过度使用文字值将会造成重复的程序代码，以及导致维护困难与缺陷。变量的另一个优点是在执行期间，你可以基于调试的目的，让`make`把它们给显示出来。稍后你将会在“调试技术”一节中看到一个不错的命令行界面。

具保护功能的编码

具保护功能的代码，就是如果你的假设或预计有一个是错误的（`if`测试结果永远为假、`assert`函数决不会失败或追踪代码）才会执行的代码，这让你能够查看`make`内部工作的状态。

事实上，你已经在本书其他地方看到过此类代码，不过为了方便起见，此处会重复加以描述。

确认检查就是具保护功能代码的最佳范例。如下的代码范例可用来确认当前所运行的`make`版本是否为3.80：

```
NEED_VERSION := 3.80
$(if $(filter $(NEED_VERSION),$(MAKE_VERSION)),,              \
  $(error You must be running make version $(NEED_VERSION).))
```

对Java应用程序来说，它可用来检查`CLASSPATH`中的文件。

进行确认的代码还可以用来确认某个东西是否为真，比如前一节用来创建目录的代码就是这样。

另一个重要的具保护功能的编码技术，就是使用“ 流程控制 ”一节所定义的assert函数。下面是其中的若干版本：

```
# $(call assert,condition,message)
define assert
  $(if $1,,$(error Assertion failed: $2))
endef

# $(call assert-file-exists,wildcard-pattern)
define assert-file-exists
  $(call assert,$(wildcard $1),$1 does not exist)
endef

# $(call assert-not-null,make-variable)
define assert-not-null
  $(call assert,$($1),The variable "$1" is null)
endef
```

我发现在*makefile*中到处声明`assert`的调用，是找出漏掉和打错的参数以及违反其他假定的既便宜又有效的方法。

我曾在第四章中编写了一对可用来追踪用户自定义函数扩展过程的函数：

```
# $(debug-enter)
debug-enter = $(if $(debug_trace),\
                $(warning Entering $0($(echo-args))))

# $(debug-leave)
debug-leave = $(if $(debug_trace),$(warning Leaving $0))

comma := ,
echo-args   = $(subst ' ','$(comma) ',\
                $(foreach a,1 2 3 4 5 6 7 8 9,'$($a)'))
```

你可以把这些宏调用到自己的函数里，并让它们处在停用状态，直到你需要进行调试。要启用它们时，请将`debug_trace`设定成任何非空值：

```
$ make debug_trace=1
```

正如第四章所说，这些追踪宏本身存在一些问题，不过仍然可用来追踪缺陷。

最后要介绍的具保护功能的编码技术，就是通过`make`变量让`@`命令修饰符的禁用更容易进行：

```
QUIET := @
...
target:
        $(QUIET) some command
```

使用此技术时，如果想看到安静模式命令的执行，你可以在命令行上以如下的方式重新定义 QUIET：

```
$ make QUIET=
```

调试技术

这一节将会探讨一般的调试技术与相关主题。最后你会觉得，调试就好像是一个装了各种你需要的东西的幸运袋。这些技术对我来说都很实用，即使是最简单的*makefile*问题，我也是靠着它们来进行调试的，或许它们也能协助你。

3.80版中一个非常恼人的缺陷是，当make汇报*makefile*中的问题时还会包含一个行号，我发现那个行号通常是错的。我并未调查出是否此问题是由于引入文件、多行变量赋值或用户自定义宏的关系，但是它的确是存在的。make 所汇报的行号通常会比实际的行号还大，在复杂的 *makefile* 中，我发现行号差了 20 行之多。

通常，查看 make 变量值的最简单方法，就是在工作目标的执行期间输出它。尽管使用*warning*加入输出语句很简单，而为了在长期运行中节省时间你会想要加入通用的debug工作目标，但是必须多费一番工夫。下面是一个简单的 debug 工作目标：

```
debug:
        $(for v,$(V), \
          $(warning $v = $($v)))
```

要使用此功能，只需要在命令行上将一份需要输出的变量的列表赋值给变量V以及指定 debug 工作目标：

```
$ make V="USERNAME SHELL" debug
makefile:2: USERNAME = Owner
makefile:2: SHELL = /bin/sh.exe
make: debug is up to date.
```

如果你觉得这样很麻烦，只要使用MAKECMDGOALS就可以避免对变量V进行赋值的动作：

```
debug:
        $(for v,$(V) $(MAKECMDGOALS), \
          $(if $(filter debug,$v),,$(warning $v = $($v))))
```

现在，你只需要在命令行上直接指定需要输出的变量即可。但是我并不建议使用这个技术，因为当 make 的警告信息指出它不知道如何更新变量时（因为它们是以工作目标的形式出现在命令行上的），你可能会产生混淆：

```
$ make debug PATH SHELL
makefile:2: USERNAME = Owner
```

```
makefile:2: SHELL = /bin/sh.exe
make: debug is up to date.
make: *** No rule to make target USERNAME.  Stop.
```

我在第十章曾简单提到过，使用开启调试功能的shell可协助我们了解make在后台所进行的活动。尽管 make 在执行命令之前会输出命令脚本中的命令，但是它并不会输出shell函数中所执行的命令。通常这些命令是既微妙且复杂的，尤其是因为它们可能会被立即执行或是延后执行（如果它们出现在递归变量中）。查看这些命令如何执行的一个方法，就是要求 subshell 启用调试的功能：

```
DATE := $(shell date +%F)
OUTPUT_DIR = out-$(DATE)

make-directories := $(shell [ -d $(OUTPUT_DIR) ] || mkdir -p
$(OUTPUT_DIR))

all: ;
```

如果运行时指定了 sh 的调试选项，我们将会看到：

```
$ make SHELL="sh -x"
+ date +%F
+ '[' -d out-2004-05-11 ']'
+ mkdir -p out-2004-05-11
```

这么做，你不仅可以看到 make 的警告信息，也可以看到额外的调试信息，因为开启调试功能的 shell 还会显示变量和表达式的值。

本书所举过的许多范例都用到了嵌套层极深的表达式，比如下面这个用来在 Windows/Cygwin 系统上检查 PATH 变量的表达式：

```
$(if $(findstring /bin/,                                  \
       $(firstword                                        \
         $(wildcard                                       \
           $(addsuffix /sort$(if $(COMSPEC),.exe),        \
             $(subst :, ,$(PATH))))),,                    \
  $(error Your PATH is wrong, c:/usr/cygwin/bin should    \
    precede c:/WINDOWS/system32))
```

要对这些表达式进行调试并没有什么好办法。一个可行的办法就是将它们拆开，输出每个子表达式（subexpression）：

```
$(warning $(subst :, ,$(PATH)))
$(warning /sort$(if $(COMSPEC),.exe))
$(warning $(addsuffix /sort$(if $(COMSPEC),.exe),      \
             $(subst :, ,$(PATH))))
$(warning $(wildcard                                   \
            $(addsuffix /sort$(if $(COMSPEC),.exe),    \
              $(subst :, ,$(PATH)))))
```

尽管这有点烦人，但是在没有调试器可用的状况下，这或许是确定各个子表达式值的最好办法（有时是唯一的办法）。

常见的错误信息

3.81版的GNU make在线使用手册列有make的错误信息以及它们产生的原因。我们在此只会介绍若干最常见的错误。此处所提到的问题中的部分并非完全是make的错误，比如命令脚本中的语法错误，但是它们仍然是开发人员常会遇到的问题。至于完整的make错误列表，请参考make在线使用手册。

make所输出的错误信息具有如下的标准格式：

```
makefile:n: *** message. Stop.
```

或：

```
make:n: *** message. Stop.
```

*makefile*部分是发生错误的*makefile*或引入文件的名称，下一个部分是发生错误的行号，接着是三个星号，最后是错误信息。

请注意，make的工作就是运行其他的程序，如果发生错误，即使问题出在你的*makefile*上，也非常可能会让人觉得错误是来自其他程序。例如，shell发生错误有可能是命令脚本形式不正确的结果，编译器发生错误有可能是因为命令行参数不正确。找出错误信息产生自哪个程序，是你解决此问题时所必须进行的第一项工作。幸好，make的错误信息相当具有说明性。

语法错误

这些通常是打字上的错误：漏掉圆括号、以空格代替跳格等。

make的新用户最常会遇到的一个错误，就是漏掉变量名称的圆括号：

```
foo:
        for f in $SOURCES; \
        do                 \
           ...             \
        done
```

这可能会使得make把$S扩展成空无一物，而且shell只会以值为OURCES的f执行循环一次。你可能会看到如下适当的shell错误信息：

```
OURCES: No such file or directory
```

不过也可能看不到任何信息，这取决于你处理 f 的方式。所以，别忘了为你的 make 变量加上圆括号。

missing separator

如下的错误信息：

```
makefile:2:missing separator. Stop.
```

或：

```
makefile:2:missing separator (did you mean TAB instead of 8 spaces?).  Stop.
```

通常代表你的命令脚本以空格代替了跳格。

以文字来解释的话，就是 make 想要查找一个 make 分隔符，比如 :、= 或一个跳格符，但是找不到。它所找到的是它不了解的东西。

commands commence before first target

跳格符的问题又出现了！

此信息首次出现在“ 分析命令 ”一节中。当命令脚本之外的文本行以一个跳格符开头时，此错误似乎通常会出现在*makefile*的中间。make 将会尽可能消除此模糊不清的状态，但如果该文本行无法被确定为变量赋值、条件表达式或多行宏定义，make 就会认为这代表命令放错地方了。

unterminated variable reference

这是一个简单但常见的错误，代表你没有为变量引用或函数调用加上适当数目的右圆括号。当函数调用和变量引用嵌套很多层时，make 文件看起来很像 Lisp！使用能够检查圆括号是否完整的编辑器，比如 Emacs，是避免此类错误最可靠的方法。

命令脚本中的错误

脚本中有三种常见的错误：在多行命令中漏掉一个分号，一个不完整或不正确的路径变量，或是一个“ 执行时会遇到问题的 ”命令。

我们已经在“ 良好的编码习惯 ”一节中探讨过漏掉分号的问题，所以此处不再做进一步的说明。

当 shell 无法找到 foo 命令时，将会显示如下的典型错误信息：

```
bash: foo: command not found
```

这表示shell已经搜索过PATH变量中的每个变量，但是找不到相符的可执行文件。要修正此错误，你必须更新你的PATH变量，它通常被放在你的*.profile*文件（Bourne shell）、*.bashrc*文件（bash）或*.cshrc*文件（C shell）中。当然，它也有可能设定在*makefile*文件中的PATH变量里，并且从make导出PATH变量。

最后，当shell命令执行失败的时候，它会以非零的结束状态终止执行。在此状况下，make将会以如下的信息汇报此错误：

```
$ make
touch /foo/bar
touch: creating /foo/bar: No such file or directory
make: *** [all] Error 1
```

此处执行失败的命令是touch，它会输出自己的错误信息以说明此状态。下一行是make的错误摘要。执行失败的*makefile*工作目标会被显示在中括号里，后面还会跟着运行失败的程序的结束值。如果程序结束运行是因为信号的缘故，make将会输出比较详细的信息，而不会只显示非零的结束状态。

并请注意，因为@修饰符而安静执行的命令也会执行失败。在此状况下，所显示的错误信息好像到处都是。

不管是以上哪种状况，错误信息皆来自make所运行的程序，而不是make本身。

No Rule to Make Target

此信息有两种形式：

```
make: *** No rule to make target XXX. Stop.
```

以及：

```
make: *** No rule to make target XXX, needed by YYY. Stop.
```

这代表make判断文件*XXX*需要更新，但是make找不到执行此工作的任何规则。在放弃和输出此信息之前，make将会在它的数据库中搜索所有的隐含和具体规则。

此项错误的理由可能有三个：

- 你的*makefile*漏掉了更新此文件所需要的一个规则。在此状况下，你必须加入描述如何建立此工作目标的规则。
- 在*makefile*中打错了字。不是make找错了文件，就是更新此文件的规则指定了错

误的文件。因为 make 变量的使用，你很难在 *makefile* 中发现打错字的问题。有时候，要确定复杂文件名的值是否正确唯有将它输出：你可以直接输出变量，或是查看 make 的内部数据库。

- 这个文件应该存在，但是 make 就是找不到它，可能是因为把它漏掉了，或是因为 make 不知道要到哪里找它。当然，有时 make 是绝对正确的，文件缺失的原因或许是你忘了将它从 CVS 调出。较常见的状况是，make 找不到源文件只是因为文件放错地方了。有时是因为源文件放在独立的源文件树中，或是文件产生自另一个程序且所产生的文件放在二进制文件树中。

Overriding Commands for Target

make 只允许一个工作目标拥有一个命令脚本（双冒号规则除外，但是很少使用）。如果一个工作目标被指定了一个以上的命令脚本，make 将会输出如下的警告信息：

```
makefile:5: warning: overriding commands for target foo
```

它也可能会显示如下的警告信息：

```
makefile:2: warning: ignoring old commands for target foo
```

第一个警告信息指出，make 在第 5 行找到了第二个命令脚本；第二个警告信息指出，位于第 2 行的最初命令脚本被覆盖掉了。

在复杂的 *makefile* 中，一个工作目标通常会被定义许多次，每一次都会加入它自己的必要条件。这些工作目标中通常会有一个被指定命令脚本，但是在开发或调试期间，你很容易会加入另一个命令脚本而忘记这么做会覆盖掉现有的命令脚本。

例如，我们可能会在一个引入文件中定义一个通用的工作目标：

```
# 建立一个 jar 文件。
$(jar_file):
        $(JAR) $(JARFLAGS) -f $@ $^
```

这使得其他的 *makefile* 可以加入自己的必要条件。然后我们可能会在某个 *makefile* 文件中这么做：

```
# 为 jar 的建立设定工作目标并且加入必要条件
jar_file = parser.jar
$(jar_file): $(class_files)
```

如果我们不小心将一个命令脚本加入此 *makefile*，make 可能会产生 overriding 的警告信息。