# 书写命令

每条规则中的命令和操作系统Shell的命令行是一致的。make会按顺序一条一条的执行命令，每条命令的开头必须以[Tab]键开头，除非，命令是紧跟在依赖规则后面的分号后的。在命令行之间中的空格或是空行会被忽略，但是如果该空格或空行是以Tab键开头的，那么make会认为其是一个空命令。

我们在UNIX下可能会使用不同的Shell，但是make的命令默认是被“/bin/sh”——UNIX的标准Shell解释执行的。除非你特别指定一个其它的Shell。Makefile中，“#”是注释符，很像C/C++中的“//”，其后的本行字符都被注释。

## 显示命令

通常，make会把其要执行的命令行在命令执行前输出到屏幕上。当我们用“@”字符在命令行前，那么，这个命令将不被make显示出来，最具代表性的例子是，我们用这个功能来像屏幕显示一些信息。如：

@echo 正在编译XXX模块......

当make执行时，会输出“正在编译XXX模块......”字串，但不会输出命令，如果没有“@”，那么，make将输出：

echo 正在编译XXX模块......

正在编译XXX模块......

如果make执行时，带入make参数“-n”或“--just-print”，那么其只是显示命令，但不会执行命令，这个功能很有利于我们调试我们的Makefile，看看我们书写的命令是执行起来是什么样子的或是什么顺序的。

而make参数“-s”或“--silent”则是全面禁止命令的显示。

## 命令执行

当依赖目标新于目标时，也就是当规则的目标需要被更新时，make会一条一条的执行其后的命令。需要注意的是，如果你要让上一条命令的结果应用在下一条命令时，你应该使用分号分隔这两条命令。比如你的第一条命令是cd命令，你希望第二条命令得在cd之后的基础上运行，那么你就不能把这两条命令写在两行上，而应该把这两条命令写在一行上，用分号分隔。如：

示例一：

exec:

cd /home/hchen

pwd

示例二：

exec:

cd /home/hchen; pwd

当我们执行“make exec”时，第一个例子中的cd没有作用，pwd会打印出当前的Makefile目录，而第二个例子中，cd就起作用了，pwd会打印出“/home/hchen”。

make一般是使用环境变量SHELL中所定义的系统Shell来执行命令，默认情况下使用UNIX的标准Shell——/bin/sh来执行命令。但在MS-DOS下有点特殊，因为MS-DOS下没有SHELL环境变量，当然你也可以指定。如果你指定了UNIX风格的目录形式，首先，make会在SHELL所指定的路径中找寻命令解释器，如果找不到，其会在当前盘符中的当前目录中寻找，如果再找不到，其会在PATH环境变量中所定义的所有路径中寻找。MS-DOS中，如果你定义的命令解释器没有找到，其会给你的命令解释器加上诸如“.exe”、“.com”、“.bat”、“.sh”等后缀。

## 命令出错

每当命令运行完后，make会检测每个命令的返回码，如果命令返回成功，那么make会执行下一条命令，当规则中所有的命令成功返回后，这个规则就算是成功完成了。如果一个规则中的某个命令出错了（命令退出码非零），那么make就会终止执行当前规则，这将有可能终止所有规则的执行。

有些时候，命令的出错并不表示就是错误的。例如mkdir命令，我们一定需要建立一个目录，如果目录不存在，那么mkdir就成功执行，万事大吉，如果目录存在，那么就出错了。我们之所以使用mkdir的意思就是一定要有这样的一个目录，于是我们就不希望mkdir出错而终止规则的运行。

为了做到这一点，忽略命令的出错，我们可以在Makefile的命令行前加一个减号“-”（在Tab键之后），标记为不管命令出不出错都认为是成功的。如：

clean:

-rm -f \*.o

还有一个全局的办法是，给make加上“-i”或是“--ignore-errors”参数，那么，Makefile中所有命令都会忽略错误。而如果一个规则是以“.IGNORE”作为目标的，那么这个规则中的所有命令将会忽略错误。这些是不同级别的防止命令出错的方法，你可以根据你的不同喜欢设置。

还有一个要提一下的make的参数的是“-k”或是“--keep-going”，这个参数的意思是，如果某规则中的命令出错了，那么就终止该规则的执行，但继续执行其它规则。

## 嵌套执行make

在一些大的工程中，我们会把我们不同模块或是不同功能的源文件放在不同的目录中，我们可以在每个目录中都书写一个该目录的Makefile，这有利于让我们的Makefile变得更加地简洁，而不至于把所有的东西全部写在一个Makefile中，这样会很难维护我们的Makefile，这个技术对于我们模块编译和分段编译有着非常大的好处。

例如，我们有一个子目录叫subdir，这个目录下有个Makefile文件，来指明了这个目录下文件的编译规则。那么我们总控的Makefile可以这样书写：

subsystem:

cd subdir && $(MAKE)

其等价于：

subsystem:

$(MAKE) -C subdir

定义$(MAKE)宏变量的意思是，也许我们的make需要一些参数，所以定义成一个变量比较利于维护。这两个例子的意思都是先进入“subdir”目录，然后执行make命令。

我们把这个Makefile叫做“总控Makefile”，总控Makefile的变量可以传递到下级的Makefile中（如果你显示的声明），但是不会覆盖下层的Makefile中所定义的变量，除非指定了“-e”参数。

如果你要传递变量到下级Makefile中，那么你可以使用这样的声明：

export <variable ...>;

如果你不想让某些变量传递到下级Makefile中，那么你可以这样声明：

unexport <variable ...>;

如：

示例一：

export variable = value

其等价于：

variable = value

export variable

其等价于：

export variable := value

其等价于：

variable := value

export variable

示例二：

export variable += value

其等价于：

variable += value

export variable

如果你要传递所有的变量，那么，只要一个export就行了。后面什么也不用跟，表示传递所有的变量。

需要注意的是，有两个变量，一个是SHELL，一个是MAKEFLAGS，这两个变量不管你是否export，其总是要传递到下层 Makefile中，特别是MAKEFILES变量，其中包含了make的参数信息，如果我们执行“总控Makefile”时有make参数或是在上层 Makefile中定义了这个变量，那么MAKEFILES变量将会是这些参数，并会传递到下层Makefile中，这是一个系统级的环境变量。

但是make命令中的有几个参数并不往下传递，它们是“-C”,“-f”,“-h”“-o”和“-W”（有关Makefile参数的细节将在后面说明），如果你不想往下层传递参数，那么，你可以这样来：

subsystem:

cd subdir && $(MAKE) MAKEFLAGS=

如果你定义了环境变量MAKEFLAGS，那么你得确信其中的选项是大家都会用到的，如果其中有“-t”,“-n”,和“-q”参数，那么将会有让你意想不到的结果，或许会让你异常地恐慌。

还有一个在“嵌套执行”中比较有用的参数，“-w”或是“--print-directory”会在make的过程中输出一些信息，让你看到目前的工作目录。比如，如果我们的下级make目录是“/home/hchen/gnu/make”，如果我们使用“make -w”来执行，那么当进入该目录时，我们会看到：

make: Entering directory `/home/hchen/gnu/make'.

而在完成下层make后离开目录时，我们会看到：

make: Leaving directory `/home/hchen/gnu/make'

当你使用“-C”参数来指定make下层Makefile时，“-w”会被自动打开的。如果参数中有“-s”（“--slient”）或是“--no-print-directory”，那么，“-w”总是失效的。

## 定义命令包

如果Makefile中出现一些相同命令序列，那么我们可以为这些相同的命令序列定义成一个变量。定义这种命令序列的语法以“define”开始，以“endef”结束，如：

define run-yacc

yacc $(firstword $^)

mv y.tab.c $@

endef

这里，“run-yacc”是这个命令包的名字，其不要和Makefile中的变量重名。在“define”和“endef”中的两行就是命令序列。这个命令包中的第一个命令是运行Yacc程序，因为Yacc程序总是生成“y.tab.c”的文件，所以第二行的命令就是把这个文件改改名字。还是把这个命令包放到一个示例中来看看吧。

foo.c : foo.y

$(run-yacc)

我们可以看见，要使用这个命令包，我们就好像使用变量一样。在这个命令包的使用中，命令包“run-yacc”中的“$^”就是“foo.y”， “$@”就是“foo.c”（有关这种以“$”开头的特殊变量，我们会在后面介绍），make在执行命令包时，命令包中的每个命令会被依次独立执行。

# 书写命令

每条规则中的命令和操作系统Shell的命令行是一致的。make会按顺序一条一条的执行命令，每条命令的开头必须以[Tab]键开头，除非，命令是紧跟在依赖规则后面的分号后的。在命令行之间中的空格或是空行会被忽略，但是如果该空格或空行是以Tab键开头的，那么make会认为其是一个空命令。

我们在UNIX下可能会使用不同的Shell，但是make的命令默认是被“/bin/sh”——UNIX的标准Shell解释执行的。除非你特别指定一个其它的Shell。Makefile中，“#”是注释符，很像C/C++中的“//”，其后的本行字符都被注释。

## 显示命令

通常，make会把其要执行的命令行在命令执行前输出到屏幕上。当我们用“@”字符在命令行前，那么，这个命令将不被make显示出来，最具代表性的例子是，我们用这个功能来像屏幕显示一些信息。如：

@echo 正在编译XXX模块......

当make执行时，会输出“正在编译XXX模块......”字串，但不会输出命令，如果没有“@”，那么，make将输出：

echo 正在编译XXX模块......

正在编译XXX模块......

如果make执行时，带入make参数“-n”或“--just-print”，那么其只是显示命令，但不会执行命令，这个功能很有利于我们调试我们的Makefile，看看我们书写的命令是执行起来是什么样子的或是什么顺序的。

而make参数“-s”或“--silent”则是全面禁止命令的显示。

## 命令执行

当依赖目标新于目标时，也就是当规则的目标需要被更新时，make会一条一条的执行其后的命令。需要注意的是，如果你要让上一条命令的结果应用在下一条命令时，你应该使用分号分隔这两条命令。比如你的第一条命令是cd命令，你希望第二条命令得在cd之后的基础上运行，那么你就不能把这两条命令写在两行上，而应该把这两条命令写在一行上，用分号分隔。如：

示例一：

exec:

cd /home/hchen

pwd

示例二：

exec:

cd /home/hchen; pwd

当我们执行“make exec”时，第一个例子中的cd没有作用，pwd会打印出当前的Makefile目录，而第二个例子中，cd就起作用了，pwd会打印出“/home/hchen”。

make一般是使用环境变量SHELL中所定义的系统Shell来执行命令，默认情况下使用UNIX的标准Shell——/bin/sh来执行命令。但在MS-DOS下有点特殊，因为MS-DOS下没有SHELL环境变量，当然你也可以指定。如果你指定了UNIX风格的目录形式，首先，make会在SHELL所指定的路径中找寻命令解释器，如果找不到，其会在当前盘符中的当前目录中寻找，如果再找不到，其会在PATH环境变量中所定义的所有路径中寻找。MS-DOS中，如果你定义的命令解释器没有找到，其会给你的命令解释器加上诸如“.exe”、“.com”、“.bat”、“.sh”等后缀。

## 命令出错

每当命令运行完后，make会检测每个命令的返回码，如果命令返回成功，那么make会执行下一条命令，当规则中所有的命令成功返回后，这个规则就算是成功完成了。如果一个规则中的某个命令出错了（命令退出码非零），那么make就会终止执行当前规则，这将有可能终止所有规则的执行。

有些时候，命令的出错并不表示就是错误的。例如mkdir命令，我们一定需要建立一个目录，如果目录不存在，那么mkdir就成功执行，万事大吉，如果目录存在，那么就出错了。我们之所以使用mkdir的意思就是一定要有这样的一个目录，于是我们就不希望mkdir出错而终止规则的运行。

为了做到这一点，忽略命令的出错，我们可以在Makefile的命令行前加一个减号“-”（在Tab键之后），标记为不管命令出不出错都认为是成功的。如：

clean:

-rm -f \*.o

还有一个全局的办法是，给make加上“-i”或是“--ignore-errors”参数，那么，Makefile中所有命令都会忽略错误。而如果一个规则是以“.IGNORE”作为目标的，那么这个规则中的所有命令将会忽略错误。这些是不同级别的防止命令出错的方法，你可以根据你的不同喜欢设置。

还有一个要提一下的make的参数的是“-k”或是“--keep-going”，这个参数的意思是，如果某规则中的命令出错了，那么就终止该规则的执行，但继续执行其它规则。

## 嵌套执行make

在一些大的工程中，我们会把我们不同模块或是不同功能的源文件放在不同的目录中，我们可以在每个目录中都书写一个该目录的Makefile，这有利于让我们的Makefile变得更加地简洁，而不至于把所有的东西全部写在一个Makefile中，这样会很难维护我们的Makefile，这个技术对于我们模块编译和分段编译有着非常大的好处。

例如，我们有一个子目录叫subdir，这个目录下有个Makefile文件，来指明了这个目录下文件的编译规则。那么我们总控的Makefile可以这样书写：

subsystem:

cd subdir && $(MAKE)

其等价于：

subsystem:

$(MAKE) -C subdir

定义$(MAKE)宏变量的意思是，也许我们的make需要一些参数，所以定义成一个变量比较利于维护。这两个例子的意思都是先进入“subdir”目录，然后执行make命令。

我们把这个Makefile叫做“总控Makefile”，总控Makefile的变量可以传递到下级的Makefile中（如果你显示的声明），但是不会覆盖下层的Makefile中所定义的变量，除非指定了“-e”参数。

如果你要传递变量到下级Makefile中，那么你可以使用这样的声明：

export <variable ...>;

如果你不想让某些变量传递到下级Makefile中，那么你可以这样声明：

unexport <variable ...>;

如：

示例一：

export variable = value

其等价于：

variable = value

export variable

其等价于：

export variable := value

其等价于：

variable := value

export variable

示例二：

export variable += value

其等价于：

variable += value

export variable

如果你要传递所有的变量，那么，只要一个export就行了。后面什么也不用跟，表示传递所有的变量。

需要注意的是，有两个变量，一个是SHELL，一个是MAKEFLAGS，这两个变量不管你是否export，其总是要传递到下层 Makefile中，特别是MAKEFILES变量，其中包含了make的参数信息，如果我们执行“总控Makefile”时有make参数或是在上层 Makefile中定义了这个变量，那么MAKEFILES变量将会是这些参数，并会传递到下层Makefile中，这是一个系统级的环境变量。

但是make命令中的有几个参数并不往下传递，它们是“-C”,“-f”,“-h”“-o”和“-W”（有关Makefile参数的细节将在后面说明），如果你不想往下层传递参数，那么，你可以这样来：

subsystem:

cd subdir && $(MAKE) MAKEFLAGS=

如果你定义了环境变量MAKEFLAGS，那么你得确信其中的选项是大家都会用到的，如果其中有“-t”,“-n”,和“-q”参数，那么将会有让你意想不到的结果，或许会让你异常地恐慌。

还有一个在“嵌套执行”中比较有用的参数，“-w”或是“--print-directory”会在make的过程中输出一些信息，让你看到目前的工作目录。比如，如果我们的下级make目录是“/home/hchen/gnu/make”，如果我们使用“make -w”来执行，那么当进入该目录时，我们会看到：

make: Entering directory `/home/hchen/gnu/make'.

而在完成下层make后离开目录时，我们会看到：

make: Leaving directory `/home/hchen/gnu/make'

当你使用“-C”参数来指定make下层Makefile时，“-w”会被自动打开的。如果参数中有“-s”（“--slient”）或是“--no-print-directory”，那么，“-w”总是失效的。

## 定义命令包

如果Makefile中出现一些相同命令序列，那么我们可以为这些相同的命令序列定义成一个变量。定义这种命令序列的语法以“define”开始，以“endef”结束，如：

define run-yacc

yacc $(firstword $^)

mv y.tab.c $@

endef

这里，“run-yacc”是这个命令包的名字，其不要和Makefile中的变量重名。在“define”和“endef”中的两行就是命令序列。这个命令包中的第一个命令是运行Yacc程序，因为Yacc程序总是生成“y.tab.c”的文件，所以第二行的命令就是把这个文件改改名字。还是把这个命令包放到一个示例中来看看吧。

foo.c : foo.y

$(run-yacc)

我们可以看见，要使用这个命令包，我们就好像使用变量一样。在这个命令包的使用中，命令包“run-yacc”中的“$^”就是“foo.y”， “$@”就是“foo.c”（有关这种以“$”开头的特殊变量，我们会在后面介绍），make在执行命令包时，命令包中的每个命令会被依次独立执行。