# 书写规则

规则包含两个部分，一个是依赖关系，一个是生成目标的方法。

在Makefile中，规则的顺序是很重要的，因为，Makefile中只应该有一个最终目标，其它的目标都是被这个目标所连带出来的，所以一定要让make知道你的最终目标是什么。一般来说，定义在Makefile中的目标可能会有很多，但是第一条规则中的目标将被确立为最终的目标。如果第一条规则中的目标有很多个，那么，第一个目标会成为最终的目标。make所完成的也就是这个目标。

好了，还是让我们来看一看如何书写规则。

## 规则举例

foo.o : foo.c defs.h # foo模块

cc -c -g foo.c

看到这个例子，各位应该不是很陌生了，前面也已说过，foo.o是我们的目标，foo.c和defs.h是目标所依赖的源文件，而只有一个命令“cc -c -g foo.c”（以Tab键开头）。这个规则告诉我们两件事：

1. 文件的依赖关系，foo.o依赖于foo.c和defs.h的文件，如果foo.c和defs.h的文件日期要比foo.o文件日期要新，或是foo.o不存在，那么依赖关系发生。
2. 生成或更新foo.o文件，就是那个cc命令。它说明了如何生成foo.o这个文件。（当然，foo.c文件include了defs.h文件）

## 规则的语法

targets : prerequisites

command

...

或是这样：

targets : prerequisites ; command

command

...

targets是文件名，以空格分开，可以使用通配符。一般来说，我们的目标基本上是一个文件，但也有可能是多个文件。

command是命令行，如果其不与“targets:prerequisites”在一行，那么，必须以[Tab键]开头，如果和prerequisites在一行，那么可以用分号做为分隔。（见上）

prerequisites也就是目标所依赖的文件（或依赖目标）。如果其中的某个文件要比目标文件要新，那么，目标就被认为是“过时的”，被认为是需要重生成的。这个在前面已经讲过了。

如果命令太长，你可以使用反斜框（‘\’）作为换行符。make对一行上有多少个字符没有限制。规则告诉make两件事，文件的依赖关系和如何生成目标文件。

一般来说，make会以UNIX的标准Shell，也就是/bin/sh来执行命令。

## 在规则中使用通配符

如果我们想定义一系列比较类似的文件，我们很自然地就想起使用通配符。make支持三个通配符：“\*”，“?”和“~”。这是和Unix的B-Shell是相同的。

波浪号（“~”）字符在文件名中也有比较特殊的用途。如果是“~/test”，这就表示当前用户的$HOME目录下的test目录。而 “~hchen/test”则表示用户hchen的宿主目录下的test目录。（这些都是Unix下的小知识了，make也支持）而在Windows或是 MS-DOS下，用户没有宿主目录，那么波浪号所指的目录则根据环境变量“HOME”而定。

通配符代替了你一系列的文件，如“\*.c”表示所有后缀为c的文件。一个需要我们注意的是，如果我们的文件名中有通配符，如：“\*”，那么可以用转义字符“\”，如“\\*”来表示真实的“\*”字符，而不是任意长度的字符串。

好吧，还是先来看几个例子吧：

clean:

rm -f \*.o

其实在这个clean:后面可以加上你想做的一些事情，如果你想看到在编译完后执行cat看看main.c的源代码，你可以在加上cat这个命令，例子如下：

clean :

cat main.c

rm -f \*.o

其结果你试一下就知道的。 上面这个例子我不多说了，这是操作系统Shell所支持的通配符。这是在命令中的通配符。

print: \*.c

lpr -p $?

touch print

上面这个例子说明了通配符也可以在我们的规则中，目标print依赖于所有的[.c]文件。其中的“$?”是一个自动化变量，我会在后面给你讲述。

objects = \*.o

上面这个例子，表示了通配符同样可以用在变量中。并不是说[\*.o]会展开， 确切的，objects的值就是“\*.o”。Makefile中的变量其实就是C/C++中的宏。如果你要让通配符在变量中展开，也就是让objects的值是所有[.o]的文件名的集合，那么，你可以这样：

objects := $(wildcard \*.o)

另给一个变量使用通配符的例子：

a. 列出一确定文件夹中的所有”.c"文件

objects := $(wildcard \*.c)

b. 列出（a）中所有文件对应的".o"文件，在（c）中我们可以看到它是由make自动编译出的。

$(patsubst %.c,%.o,$(wildcard \*.c))

c. 由（a）（b）两步，可写出编译并链接所有“.c"和”.o"文件

objects := $(patsubst %.c,%.o,$(wildcard \*.c))

foo : $(objects)

cc -o foo $(objects)

这种用法由关键字“wildcard”，“patsubst”指出，关于Makefile的关键字，我们将在后面讨论。

## 文件搜寻

在一些大的工程中，有大量的源文件，我们通常的做法是把这许多的源文件分类，并存放在不同的目录中。所以，当make需要去找寻文件的依赖关系时，你可以在文件前加上路径，但最好的方法是把一个路径告诉make，让make自动去找。

Makefile文件中的特殊变量“VPATH”就是完成这个功能的，如果没有指明这个变量，make只会在当前的目录中去找寻依赖文件和目标文件。如果定义了这个变量，那么，make就会在当前目录找不到的情况下，到所指定的目录中去找寻文件了。

VPATH = src:../headers

上面的的定义指定两个目录，“src”和“../headers”，make会按照这个顺序进行搜索。目录由“冒号”分隔。（当然，当前目录永远是最高优先搜索的地方）

另一个设置文件搜索路径的方法是使用make的“vpath”关键字（注意，它是全小写的），这不是变量，这是一个make的关键字，这和上面提到的那个VPATH变量很类似，但是它更为灵活。它可以指定不同的文件在不同的搜索目录中。这是一个很灵活的功能。它的使用方法有三种：

1、vpath <pattern> <directories>

为符合模式<pattern>的文件指定搜索目录<directories>。

2、vpath <pattern>

清除符合模式<pattern>的文件的搜索目录。

3、vpath

清除所有已被设置好了的文件搜索目录。

vpath使用方法中的<pattern>需要包含“%”字符。“%”的意思是匹配零或若干字符，（需引用“%”，使用“\%"）例如，“%.h”表示所有以 “.h”结尾的文件。<pattern>指定了要搜索的文件集，而<directories>则指定了< pattern>的文件集的搜索的目录。例如：

vpath %.h ../headers

该语句表示，要求make在“../headers”目录下搜索所有以“.h”结尾的文件。（如果某文件在当前目录没有找到的话）

我们可以连续地使用vpath语句，以指定不同搜索策略。如果连续的vpath语句中出现了相同的<pattern>，或是被重复了的<pattern>，那么，make会按照vpath语句的先后顺序来执行搜索。如：

vpath %.c foo

vpath %.c blish

vpath %.c bar

其表示“.c”结尾的文件，先在“foo”目录，然后是“blish”，最后是“bar”目录。

vpath %.c foo:bar

vpath %.c blish

而上面的语句则表示“.c”结尾的文件，先在“foo”目录，然后是“bar”目录，最后才是“blish”目录。

## 伪目标

最早先的一个例子中，我们提到过一个“clean”的目标，这是一个“伪目标”，

clean:

rm \*.o temp

正像我们前面例子中的“clean”一样，既然我们生成了许多文件编译文件，我们也应该提供一个清除它们的“目标”以备完整地重编译而用。 （以“make clean”来使用该目标）

因为，我们并不生成“clean”这个文件。“伪目标”并不是一个文件，只是一个标签，由于“伪目标”不是文件，所以make无法生成它的依赖关系和决定它是否要执行。我们只有通过显式地指明这个“目标”才能让其生效。当然，“伪目标”的取名不能和文件名重名，不然其就失去了“伪目标”的意义了。

当然，为了避免和文件重名的这种情况，我们可以使用一个特殊的标记“.PHONY”来显式地指明一个目标是“伪目标”，向make说明，不管是否有这个文件，这个目标就是“伪目标”。

.PHONY : clean

只要有这个声明，不管是否有“clean”文件，要运行“clean”这个目标，只有“make clean”这样。于是整个过程可以这样写：

.PHONY : clean

clean :

rm \*.o temp

伪目标一般没有依赖的文件。但是，我们也可以为伪目标指定所依赖的文件。伪目标同样可以作为“默认目标”，只要将其放在第一个。一个示例就是，如果你的Makefile需要一口气生成若干个可执行文件，但你只想简单地敲一个make完事，并且，所有的目标文件都写在一个Makefile中，那么你可以使用“伪目标”这个特性：

all : prog1 prog2 prog3

.PHONY : all

prog1 : prog1.o utils.o

cc -o prog1 prog1.o utils.o

prog2 : prog2.o

cc -o prog2 prog2.o

prog3 : prog3.o sort.o utils.o

cc -o prog3 prog3.o sort.o utils.o

我们知道，Makefile中的第一个目标会被作为其默认目标。我们声明了一个“all”的伪目标，其依赖于其它三个目标。由于默认目标的特性是，总是被执行的，但由于“all”又是一个伪目标，伪目标只是一个标签不会生成文件，所以不会有“all”文件产生。于是，其它三个目标的规则总是会被决议。也就达到了我们一口气生成多个目标的目的。“.PHONY : all”声明了“all”这个目标为“伪目标”。（注：这里的显式 “.PHONY : all" 不写的话一般情况也可以正确的执行，这样 make 可通过隐式规则推导出， “all" 是一个伪目标，执行 make 不会生成 ”all" 文件，而执行后面的多个目标。建议：显式写出是一个好习惯。）

随便提一句，从上面的例子我们可以看出，目标也可以成为依赖。所以，伪目标同样也可成为依赖。看下面的例子：

.PHONY : cleanall cleanobj cleandiff

cleanall : cleanobj cleandiff

rm program

cleanobj :

rm \*.o

cleandiff :

rm \*.diff

“make cleanall”将清除所有要被清除的文件。“cleanobj”和“cleandiff”这两个伪目标有点像“子程序”的意思。我们可以输入“make cleanall”和“make cleanobj”和“make cleandiff”命令来达到清除不同种类文件的目的。

## 多目标

Makefile的规则中的目标可以不止一个，其支持多目标，有可能我们的多个目标同时依赖于一个文件，并且其生成的命令大体类似。于是我们就能把其合并起来。当然，多个目标的生成规则的执行命令不是同一个，这可能会给我们带来麻烦，不过好在我们可以使用一个自动化变量“$@”（关于自动化变量，将在后面讲述），这个变量表示着目前规则中所有的目标的集合，这样说可能很抽象，还是看一个例子吧。

bigoutput littleoutput : text.g

generate text.g -$(subst output,,$@) > $@

上述规则等价于：

bigoutput : text.g

generate text.g -big > bigoutput

littleoutput : text.g

generate text.g -little > littleoutput

其中，-$(subst output,,$@)中的“$”表示执行一个Makefile的函数，函数名为subst，后面的为参数。关于函数，将在后面讲述。这里的这个函数是替换字符串的意思，“$@”表示目标的集合，就像一个数组，“$@”依次取出目标，并执于命令。

## 静态模式

静态模式可以更加容易地定义多目标的规则，可以让我们的规则变得更加的有弹性和灵活。我们还是先来看一下语法：

<targets ...>: <target-pattern>: <prereq-patterns ...>

<commands>

...

targets定义了一系列的目标文件，可以有通配符。是目标的一个集合。

target-pattern是指明了targets的模式，也就是的目标集模式。

prereq-patterns是目标的依赖模式，它对target-pattern形成的模式再进行一次依赖目标的定义。

这样描述这三个东西，可能还是没有说清楚，还是举个例子来说明一下吧。如果我们的<target-pattern>定义成“%. o”，意思是我们的<target>集合中都是以“.o”结尾的，而如果我们的<prereq-patterns>定义成 “%.c”，意思是对<target-pattern>所形成的目标集进行二次定义，其计算方法是，取<target- pattern>模式中的“%”（也就是去掉了[.o]这个结尾），并为其加上[.c]这个结尾，形成的新集合。

所以，我们的“目标模式”或是“依赖模式”中都应该有“%”这个字符，如果你的文件名中有“%”那么你可以使用反斜杠“\”进行转义，来标明真实的“%”字符。

看一个例子：

objects = foo.o bar.o

all: $(objects)

$(objects): %.o: %.c

$(CC) -c $(CFLAGS) $< -o $@

上面的例子中，指明了我们的目标从$object中获取，“%.o”表明要所有以“.o”结尾的目标，也就是“foo.o bar.o”，也就是变量$object集合的模式，而依赖模式“%.c”则取模式“%.o”的“%”，也就是“foo bar”，并为其加下“.c”的后缀，于是，我们的依赖目标就是“foo.c bar.c”。而命令中的“$<”和“$@”则是自动化变量，“$<”表示所有的依赖目标集（也就是 “foo.c bar.c”），“$@”表示目标集（也就是“foo.o bar.o”）。于是，上面的规则展开后等价于下面的规则：

foo.o : foo.c

$(CC) -c $(CFLAGS) foo.c -o foo.o

bar.o : bar.c

$(CC) -c $(CFLAGS) bar.c -o bar.o

试想，如果我们的“%.o”有几百个，那种我们只要用这种很简单的“静态模式规则”就可以写完一堆规则，实在是太有效率了。“静态模式规则”的用法很灵活，如果用得好，那会一个很强大的功能。再看一个例子：

files = foo.elc bar.o lose.o

$(filter %.o,$(files)): %.o: %.c

$(CC) -c $(CFLAGS) $< -o $@

$(filter %.elc,$(files)): %.elc: %.el

emacs -f batch-byte-compile $<

$(filter %.o,$(files))表示调用Makefile的filter函数，过滤“$files”集，只要其中模式为“%.o”的内容。其的它内容，我就不用多说了吧。这个例子展示了Makefile中更大的弹性。

## 自动生成依赖性

在Makefile中，我们的依赖关系可能会需要包含一系列的头文件，比如，如果我们的main.c中有一句“#include "defs.h"”，那么我们的依赖关系应该是：

main.o : main.c defs.h

但是，如果是一个比较大型的工程，你必需清楚哪些C文件包含了哪些头文件，并且，你在加入或删除头文件时，也需要小心地修改Makefile，这是一个很没有维护性的工作。为了避免这种繁重而又容易出错的事情，我们可以使用C/C++编译的一个功能。大多数的C/C++编译器都支持一个“-M”的选项，即自动找寻源文件中包含的头文件，并生成一个依赖关系。例如，如果我们执行下面的命令：

cc -M main.c

其输出是：

main.o : main.c defs.h

于是由编译器自动生成的依赖关系，这样一来，你就不必再手动书写若干文件的依赖关系，而由编译器自动生成了。需要提醒一句的是，如果你使用GNU的C/C++编译器，你得用“-MM”参数，不然，“-M”参数会把一些标准库的头文件也包含进来。

gcc -M main.c的输出是：

main.o: main.c defs.h /usr/include/stdio.h /usr/include/features.h \

/usr/include/sys/cdefs.h /usr/include/gnu/stubs.h \

/usr/lib/gcc-lib/i486-suse-linux/2.95.3/include/stddef.h \

/usr/include/bits/types.h /usr/include/bits/pthreadtypes.h \

/usr/include/bits/sched.h /usr/include/libio.h \

/usr/include/\_G\_config.h /usr/include/wchar.h \

/usr/include/bits/wchar.h /usr/include/gconv.h \

/usr/lib/gcc-lib/i486-suse-linux/2.95.3/include/stdarg.h \

/usr/include/bits/stdio\_lim.h

gcc -MM main.c的输出则是：

main.o: main.c defs.h

那么，编译器的这个功能如何与我们的Makefile联系在一起呢。因为这样一来，我们的Makefile也要根据这些源文件重新生成，让 Makefile自已依赖于源文件？这个功能并不现实，不过我们可以有其它手段来迂回地实现这一功能。GNU组织建议把编译器为每一个源文件的自动生成的依赖关系放到一个文件中，为每一个“name.c”的文件都生成一个“name.d”的Makefile文件，[.d]文件中就存放对应[.c]文件的依赖关系。

于是，我们可以写出[.c]文件和[.d]文件的依赖关系，并让make自动更新或自成[.d]文件，并把其包含在我们的主Makefile中，这样，我们就可以自动化地生成每个文件的依赖关系了。

这里，我们给出了一个模式规则来产生[.d]文件：

%.d: %.c

@set -e; rm -f $@; \

$(CC) -M $(CPPFLAGS) $< > $@.$$$$; \

sed 's,\($\*\)\.o[ :]\*,\1.o $@ : ,g' < $@.$$$$ > $@; \

rm -f $@.$$$$

这个规则的意思是，所有的[.d]文件依赖于[.c]文件，“rm -f $@”的意思是删除所有的目标，也就是[.d]文件，第二行的意思是，为每个依赖文件“$<”，也就是[.c]文件生成依赖文件，“$@”表示模式“%.d”文件，如果有一个C文件是name.c，那么“%”就是 “name”，“$$$$”意为一个随机编号，第二行生成的文件有可能是“name.d.12345”，第三行使用sed命令做了一个替换，关于sed命令的用法请参看相关的使用文档。第四行就是删除临时文件。

总而言之，这个模式要做的事就是在编译器生成的依赖关系中加入[.d]文件的依赖，即把依赖关系：

main.o : main.c defs.h

转成：

main.o main.d : main.c defs.h

于是，我们的[.d]文件也会自动更新了，并会自动生成了，当然，你还可以在这个[.d]文件中加入的不只是依赖关系，包括生成的命令也可一并加入，让每个[.d]文件都包含一个完赖的规则。一旦我们完成这个工作，接下来，我们就要把这些自动生成的规则放进我们的主Makefile中。我们可以使用Makefile的“include”命令，来引入别的Makefile文件（前面讲过），例如：

sources = foo.c bar.c

include $(sources:.c=.d)

上述语句中的“$(sources:.c=.d)”中的“.c=.d”的意思是做一个替换，把变量$(sources)所有[.c]的字串都替换成[.d]，关于这个“替换”的内容，在后面我会有更为详细的讲述。当然，你得注意次序，因为include是按次来载入文件，最先载入的[.d]文件中的目标会成为默认目标。