**3.1 规则举例**

 foo.o: foo.c defs.h       # foo模块

           cc -c -g foo.c

**3.2 规则的语法**

 targets : prerequisites

       command

       ...

或是这样：

targets : prerequisites ; command

           ...

**3.3 在规则中使用通配符**

make支持三个通配符：“\*”，“?”和“[...]”

**"~"**

如果是“~/test”，这就表示当前用户的$HOME目录下的test目录（linux系统）。

**"\*"**

如“\*.c”表示所以后缀为c的文件。

 objects = \*.o

上面这个例子，objects的值就是“\*.o”。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)

**3.4 文件搜寻**

Makefile文件中的特殊变量“VPATH”，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。

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

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<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

例如，“%.h”表示所有以“.h”结尾的文件。

 vpath %.h ../headers

该语句表示在“../headers”目录下搜索所有以“.h”结尾的文件。

我们可以连续地使用vpath语句，以指定不同搜索策略。

如：

   vpath %.c foo

   vpath %   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。

**3.5 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

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”来显示地指明一个目标是“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”。

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

**3.6 多目标**

Makefile的规则中的目标可以不止一个，其支持多目标，当有多个目标时，我们可以把规则当做一个模式，

$@表示模式的目标，运行时会拆分成多个规则

   bigoutput littleoutput : text.g

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

   上述规则等价于：

   bigoutput : text.g

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

   littleoutput : text.g

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

**3.7 静态模式**

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

　　　<commands>

...

相当于

一堆文件：筛选出目标的模式：目标的依赖

命令

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-parrtern是指明了targets的模式，也就是的目标集模式。

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

我们的“目标模式”或是“依赖模式”中都应该有“%”这个字符，表示任意字符串

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，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”，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”。

命令中的“$<”和“$@”则是自动化变量，

$< 表示我们的模式依赖，“$@”表示我们的模式目标。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

**3.8 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，如：

   main.o : main.c defs.h

如果是一个比较大型的工程会有很多头文件

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

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。

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

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，

第一行，“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“$<”生成“$@”文件

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。

于是，我们的[.d]文件也会自动更新了，并会自动生成了。

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]