## Makefile文件

源文件首先会生成中间目标文件，再由中间目标文件生成执行文件。在编译时，编译器只检测程序语法，和函数、变量是否被声明。如果函数未被声明，编译器会给出一个警告，但可以生成Object File。而在链接程序时，链接器会在所有的Object File中找寻函数的实现，如果找不到，那到就会报链接错误码（Linker Error）

### 规则

先来粗略地看一看**Makefile的规则**。

target ... : prerequisites ...

command

...

...

或是这样：

targets : prerequisites ; command

command

...

target也就是一个目标文件，可以是Object File，也可以是执行文件。还可以是一个标签（Label），对于标签这种特性，在后续的“伪目标”章节中会有叙述。

prerequisites就是，要生成那个target所需要的文件或是目标。

command也就是make需要执行的命令。（任意的Shell命令）

这是一个文件的依赖关系，也就是说，**target这一个或多个的目标文件依赖于prerequisites中的文件，其生成规则定义在command中**。说白一点就是说，prerequisites中如果有一个以上的文件比target文件要新的话，command所定义的命令就会被执行。这就是Makefile的规则。也就是Makefile中**最核心的内容**。

在这个示例中，我们的工程有8个C文件，和3个头文件，我们要写一个Makefile来告诉make命令如何编译和链接这几个文件。我们的规则是：

1）如果这个工程没有编译过，那么我们的所有C文件都要编译并被链接。

2）如果这个工程的某几个C文件被修改，那么我们只编译被修改的C文件，并链接目标程序。

3）如果这个工程的头文件被改变了，那么我们需要编译引用了这几个头文件的C文件，并链接目标程序。

只要我们的Makefile写得够好，所有的这一切，我们只用一个make命令就可以完成，make命令会**自动智能地根据当前的文件修改的情况来确定哪些文件需要重编译**，从而自己编译所需要的文件和链接目标程序。

|  |
| --- |
| 为了完成前面所述的那三个规则，我们的Makefile应该是下面的这个样子的。  edit : main.o kbd.o command.o display.o /  insert.o search.o files.o utils.o  cc -o edit main.o kbd.o command.o display.o /  insert.o search.o files.o utils.o  main.o : main.c defs.h  cc -c main.c  kbd.o : kbd.c defs.h command.h  cc -c kbd.c  command.o : command.c defs.h command.h  cc -c command.c  display.o : display.c defs.h buffer.h  cc -c display.c  insert.o : insert.c defs.h buffer.h  cc -c insert.c  search.o : search.c defs.h buffer.h  cc -c search.c  files.o : files.c defs.h buffer.h command.h  cc -c files.c  utils.o : utils.c defs.h  cc -c utils.c  **clean :**  rm edit main.o kbd.o command.o display.o /  insert.o search.o files.o utils.o |

反斜杠（/）是换行符的意思。这样比较便于Makefile的易读。我们可以把这个内容保存在文件为“Makefile”或“makefile”的文件中，然后在该目录下**直接输入命令“make”就可以生成执行文件edit**。如果要删除执行文件和所有的中间目标文件，那么，只要简单地执行一下“**make clean**”就可以了。

在这个makefile中，目标文件（target）包含：执行文件edit和中间目标文件（\*.o），依赖文件（prerequisites）就是冒号后面的那些 .c 文件和 .h文件。每一个 .o 文件都有一组依赖文件，而这些 .o 文件又是执行文件 edit 的依赖文件。依赖关系的实质上就是说明了目标文件是由哪些文件生成的，换言之，目标文件是哪些文件更新的。

在定义好依赖关系后，后续的那一行定义了如何生成目标文件的操作系统命令，**一定要以一个Tab键作为开头**。记住，make并不管命令是怎么工作的，他只管执行所定义的命令。make会比较targets文件和prerequisites文件的修改日期，如果prerequisites文件的日期要比targets文件的日期要新，或者target不存在的话，那么，make就会执行后续定义的命令。

这里要说明一点的是，**clean不是一个文件**，它只不过是一个动作名字，有点像C语言中的lable一样，**其冒号后什么也没有**，那么，make就**不会自动去找文件的依赖性**，也就不会自动执行其后所定义的命令。要执行其后的命令，就要在make命令后明显得指出这个lable的名字。这样的方法非常有用，我们**可以在一个makefile中定义不用的编译或是和编译无关的命令**，比如程序的打包，程序的备份，等等。

### 如何工作

在默认的方式下，也就是我们只输入make命令。那么，

1、make会在当前目录下找名字叫“Makefile”或“makefile”的文件。

2、如果找到，它会找文件中的**第一个目标文件**（target），在上面的例子中，他会找到“edit”这个文件，并把这个文件作为最终的目标文件。

3、如果edit文件不存在，或是edit所依赖的后面的 .o 文件的文件修改时间要比edit这个文件新，那么，他就会执行后面所定义的命令来生成edit这个文件。

4、如果edit所依赖的.o文件也不存在，那么make会在当前文件中找目标为.o文件的依赖性，如果找到则再根据那一个规则生成.o文件。（这有点像一个堆栈的过程）

5、当然，你的C文件和H文件是存在的啦，于是make会生成 .o 文件，然后再用 .o 文件生命make的终极任务，也就是执行文件edit了。

当然，你可以使用别的文件名来书写Makefile，比如：“Make.Linux”，“Make.Solaris”，“Make.AIX”等，如果要指定特定的Makefile，你可以使用make的“-f”和“--file”参数，如：make -f Make.Linux或make --file Make.AIX

GNU的make工作时的执行步骤入下：（想来其它的make也是类似）

1、读入所有的Makefile。

2、读入被include的其它Makefile。

3、初始化文件中的变量。

4、推导隐晦规则，并分析所有规则。

5、为所有的目标文件创建依赖关系链。

6、根据依赖关系，决定哪些目标要重新生成。

7、执行生成命令。

1-5步为第一个阶段，6-7为第二个阶段。第一个阶段中，如果定义的变量被使用了，那么，make会把其展开在使用的位置。但make并不会完全马上展开，make使用的是拖延战术，如果变量出现在依赖关系的规则中，那么仅当这条依赖被决定要使用了，变量才会在其内部展开。

### 使用变量

在上面的例子中，先让我们看看edit的规则：

|  |
| --- |
| edit : main.o kbd.o command.o display.o /  insert.o search.o files.o utils.o  cc -o edit main.o kbd.o command.o display.o /  insert.o search.o files.o utils.o |

我们可以看到[.o]文件的字符串被重复了两次，如果我们的工程需要加入一个新的[.o]文件，那么我们需要在两个地方加（应该是三个地方，还有一个地方在clean中）。当然，我们的makefile并不复杂，所以在两个地方加也不累，但如果makefile变得复杂，那么我们就有可能会忘掉一个需要加入的地方，而导致编译失败。所以，为了makefile的易维护，在makefile中我们可以使用变量。**makefile的变量也就是一个字符串**，**理解成C语言中的宏**可能会更好。

比如，我们声明一个变量，叫objects, OBJECTS, objs, OBJS, obj, 或是 OBJ，反正不管什么啦，只要能够表示obj文件就行了。我们在makefile一开始就这样定义：

|  |
| --- |
| objects = main.o kbd.o command.o display.o /  insert.o search.o files.o utils.o |

于是，我们就可以很方便地在我们的makefile中以“$(objects)”的方式来使用这个变量了，于是我们的改良版makefile就变成下面这个样子：

变量在声明时需要给予初值，而在使用时，需要给在变量名前加上“$”符号，但最好用小括号“（）”或是大括号“{}”把变量给包括起来。如果你要使用真实的“$”字符，那么你需要用“$$”来表示。

给变量加上括号完全是为了更加安全地使用这个变量，在上面的例子中，如果你不想给变量加上括号，那也可以

|  |
| --- |
| **objects** = main.o kbd.o command.o display.o /  insert.o search.o files.o utils.o  edit : **$(objects)**  cc -o edit **$(objects)**  main.o : main.c defs.h  cc -c main.c  ……  …… |

于是如果有新的 .o 文件加入，我们只需简单地修改一下 objects 变量就可以了。

**变量中的变量**

在定义变量的值时，我们可以使用其它变量来构造变量的值，在Makefile中有两种方式来在用变量定义变量的值。

先看第一种方式，也就是简单的**使用“=”号**，在“=”左侧是变量，右侧是变量的值，右侧变量的值可以定义在文件的任何一处，也就是说，右侧中的变量不一定非要是已定义好的值，其也可以使用后面定义的值。如：

|  |
| --- |
| foo = $(bar)  bar = $(ugh)  ugh = Huh?  all:  echo $(foo) |

我们执行“make all”将会打出变量$(foo)的值是“Huh?”（ $(foo)的值是$(bar)，$(bar)的值是$(ugh)，$(ugh)的值是“Huh?”）可见，变量是可以使用后面的变量来定义的。

但这种形式也有不好的地方，那就是递归定义，如：

|  |
| --- |
| CFLAGS = $(CFLAGS) -O  或：  A = $(B)  B = $(A) |

这会让make陷入无限的变量展开过程中去，当然，我们的make是有能力检测这样的定义，并会报错。还有就是如果在变量中使用函数，那么，这种方式会让我们的make运行时非常慢

为了避免上面的这种方法，我们可以使用make中的另一种用变量来定义变量的方法。这种方法使用的是**“:=”操作符**，如：

x := foo

y := $(x) bar

x := later

其等价于：

y := foo bar

x := later

值得一提的是，这种方法，**前面的变量不能使用后面的变量**，只能使用前面已定义好了的变量。

还有一个比较有用的操作符是“?=”，先看示例：

|  |
| --- |
| FOO ?= bar |

其含义是，如果FOO没有被定义过，那么变量FOO的值就是“bar”，如果FOO先前被定义过，那么这条语将什么也不做

我们可以使用“+=”操作符给变量追加值，如：

|  |
| --- |
| objects = main.o foo.o bar.o utils.o  objects += another.o |

于是，我们的$(objects)值变成：“main.o foo.o bar.o utils.o another.o”（another.o被追加进去了）

**环境变量**

make运行时的系统环境变量可以在make开始运行时被载入到Makefile文件中，但是如果Makefile中已定义了这个变量，或是这个变量由make命令行带入，那么系统的环境变量的值将被覆盖。（如果make指定了“-e”参数，那么，系统环境变量将覆盖Makefile中定义的变量）

因此，如果我们在环境变量中设置了“CFLAGS”环境变量，那么我们就可以在所有的Makefile中使用这个变量了。这对于我们使用统一的编译参数有比较大的好处。

### 自动推导

GNU的make很强大，它可以自动推导文件以及文件依赖关系后面的命令，于是我们就没必要去在每一个[.o]文件后都写上类似的命令，因为，我们的make会自动识别，并自己推导命令。

只要make看到一个[.o]文件，它就会**自动的把同名的[.c]文件加在依赖关系中**，如果make找到一个whatever.o，那么whatever.c，就会是whatever.o的依赖文件。并且 **cc -c** whatever.c **也会被推导出来**，于是，我们的makefile再也不用写得这么复杂。

|  |
| --- |
| objects = main.o kbd.o command.o display.o /  insert.o search.o files.o utils.o  edit : $(objects)  cc -o edit $(objects)  main.o : ~~main.c~~ defs.h  ~~cc -c main.c~~  kbd.o : defs.h command.h  command.o : defs.h command.h  display.o : defs.h buffer.h  insert.o : defs.h buffer.h  search.o : defs.h buffer.h  files.o : defs.h buffer.h command.h  utils.o : defs.h  .PHONY : clean  clean :  rm edit $(objects) |

上面文件内容中，“.PHONY”表示，clean是个伪目标文件。

### 另类风格的makefile

即然我们的make可以自动推导命令，那么我看到那堆[.o]和[.h]的依赖就有点不爽，那么多的重复的[.h]，能不能把其收拢起来，好吧，没有问题

|  |
| --- |
| objects = main.o kbd.o command.o display.o /  insert.o search.o files.o utils.o  edit : $(objects)  cc -o edit $(objects)  $(objects) : defs.h  kbd.o command.o files.o : command.h  display.o insert.o search.o files.o : buffer.h  .PHONY : clean  clean :  rm edit $(objects) |

### 清空目标文件的规则

每个Makefile中都应该写一个清空目标文件（.o和执行文件）的规则，这不仅便于重编译，也很利于保持文件的清洁。这是一个“修养”（呵呵，还记得我的《编程修养》吗）。一般的风格都是：

|  |
| --- |
| clean:  rm edit $(objects) |

更为稳健的做法是：

|  |
| --- |
| .PHONY : clean  clean :  -rm edit $(objects) |

前面说过，.PHONY意思表示clean是一个“伪目标”，。而在rm命令前面加了**一个小减号**的意思就是，也许某些文件出现问题，但不要管，继续做后面的事。当然，clean的规则不要放在文件的开头，不然，这就会变成make的默认目标，相信谁也不愿意这样。不成文的规矩是——“**clean从来都是放在文件的最后**”。

Makefile中只有**行注释**，和UNIX的Shell脚本一样，其注释是用“#”字符，这个就像C/C++中的“//”一样。如果你要在你的Makefile中使用“#”字符，可以用反斜框进行转义，如：“/#”。

最后，还值得一提的是，在Makefile中的命令，必须要以[Tab]键开始。

### 文件指示

文件指示。其包括了三个部分，一个是在一个Makefile中引用另一个Makefile，就像C语言中的include一样；另一个是指根据某些情况指定Makefile中的有效部分，就像C语言中的预编译#if一样；还有就是定义一个多行的命令。

**引用其它的Makefile**

在Makefile使用include关键字可以把别的Makefile包含进来，这很像C语言的#include，被包含的文件会原模原样的放在当前文件的包含位置。include的语法是：

|  |
| --- |
| include <filename> |

filename可以是当前操作系统Shell的文件模式（可以保含路径和通配符）

在include前面可以有一些空字符，但是**绝不能是[Tab]键开始**。include和<filename>可以用一个或多个空格隔开。举个例子，你有这样几个Makefile：a.mk、b.mk、c.mk，还有一个文件叫foo.make，以及一个变量$(bar)，其包含了e.mk和f.mk，那么，下面的语句：

|  |
| --- |
| include foo.make \*.mk $(bar) |

等价于：

|  |
| --- |
| include foo.make a.mk b.mk c.mk e.mk f.mk |

make命令开始时，会把找寻include所指出的其它Makefile，并把其内容安置在当前的位置。就好像C/C++的#include指令一样。如果文件都没有指定绝对路径或是相对路径的话，make会在当前目录下首先寻找，如果当前目录下没有找到，那么，make还会在下面的几个目录下找：

1、如果make执行时，有“-I”或“--include-dir”参数，那么make就会在这个参数所指定的目录下去寻找。

2、如果目录<prefix>/include（一般是：/usr/local/bin或/usr/include）存在的话，make也会去找。

如果有文件没有找到的话，make会生成一条警告信息，但不会马上出现致命错误。它会继续载入其它的文件，一旦完成makefile的读取，make会再重试这些没有找到，或是不能读取的文件，如果还是不行，make才会出现一条致命信息。如果你想让make不理那些无法读取的文件，而继续执行，你可以在include前**加一个减号“-”**。如：

|  |
| --- |
| -include <filename> |

其表示，无论include过程中出现什么错误，都不要报错继续执行。和其它版本make兼容的相关命令是sinclude，其作用和这一个是一样的。

### 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”表示所以后缀为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。

|  |
| --- |
| **命令**中的通配符  clean:  rm -f \*.o  **规则**中的通配符  print: \*.c  lpr -p $?  touch print  目标print依赖于所有的[.c]文件。其中的“$?”是一个自动化变量，我会在后面给你讲述。  **变量**中的通配符  objects = \*.o  并不是说[\*.o]会展开，不！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)  这种用法由关键字“wildcard”指出 |

伪目标

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。由于伪目标的特性是，总是被执行的，所以其依赖的那三个目标就总是不如“all”这个目标新。所以，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”。

### 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-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。

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-parrtern>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-parrterns>定义成“%.c”，意思是对<target-parrtern>所形成的目标集进行二次定义，其计算方法是，取<target-parrtern>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。

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“$filter”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。这个例字展示了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。

更多

**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......

make参数“-s”或“--slient”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。

**命令出错**

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。

**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 ...> |

**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。

**使用函数**

在Makefile中可以使用函数来处理变量，从而让我们的命令或是规则更为的灵活和具有智能。make所支持的函数也不算很多，不过已经足够我们的操作了。函数调用后，函数的返回值可以当做变量来使用。

<http://blog.csdn.net/liang13664759/article/details/1771246>