* **规则：**

|  |
| --- |
| target ... : prerequisites ...  command  ...  或是  target…: prerequisites …; command  command |

target:是一个目标文件，可以是Object file，也可以是运行文件，还可能是一个标签，称为伪目标；

prerequisites:需要生成target所需要的文件或目标

command:生成目标所使用的命令

* **运行原理：**

1. make命令会在当前文件夹下按顺序找寻文件名称为“GNUmakefile”、“makefile”、“Makefile”的文件，如果需要指定时使用-f参数进行指定。
2. 假设找到，它会找文件里的第一个目标文件（target）作为最终的目标文件。
3. 假设目标文件不存在，或是目标文件所依赖的后面的 依赖文件的文件改动时间要比目标文件新，那么，他就会运行后面所定义的命令来重新生成目标文件。
4. 假设依赖文件也不存在，那么make会在当前文件里找目标为依赖文件的依赖性，假设找到则再依据其规则生成相应的目标文件。
5. 最终在所有依赖文件都存在且依赖文件的改动时间在源文件的时间之后时，生成最终的目标文件。
6. 读入全部的Makefile。
7. 读入被include的其他Makefile。
8. 初始化文件里的变量。
9. 推导隐晦规则，并分析全部规则。
10. 为全部的目标文件创建依赖关系链。
11. 依据依赖关系，决定哪些目标要又一次生成。
12. 运行生成命令。

* **Makefile有什么：**

Makefile里主要包括了五个东西：显式规则、隐晦规则、变量定义、文件指示和凝视。

1. 显式规则。显式规则说明了，怎样生成一个或多的的目标文件。这是由Makefile的书写者明显指出，要生成的文件，文件的依赖文件，生成的命令。
2. 隐晦规则。由于我们的make有自己主动推导的功能，所以隐晦的规则能够让我们比較粗糙地简略地书写Makefile，这是由make所支持的。
3. 变量的定义。在Makefile中我们要定义一系列的变量，变量一般都是字符串，这个有点你C语言中的宏，当Makefile被运行时，当中的变量都会被扩展到对应的引用位置上。
4. 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；另一个是指依据某些情况指定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。
5. 注释。Makefile中只有单行注释，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。假设你要在你的Makefile中使用“#”字符，能够用反斜框进行转义，如：“\#”。最后，还值得一提的是，在Makefile中的命令，必须要以[Tab]键开始。

* **语法：**

1. 变量定义和使用：varname = varvalue; $(varname)
2. 自主推导：make命令在看见一个[.o]文件时会自动将相应[.c]作为依赖文件，并自动推倒出cc -c [.c] [.o]的命令
3. .PHONY：表示伪目标文件
4. –命令：命令前添加一个小减号，表示某些文件出现故障时，不需要管，继续做后面的事情；
5. include：makefile中可以使用include包含其他makefile的内容，make执行时会把相应文件的内容安置在include的位置，文件的路径可以时相对路径或绝地路径，只有个文件时，会在当前目录找，然后在-I(--include-dir)参数值下查找；
6. VPATH变量可以指定文件搜索的目录，默认在makefile的当前目录下查找，如果没有找到，则会去VPATH路径找，多个路径使使用冒号分开；VPATH=src:../headers
7. make的关键之vpath也可以指定文件搜索的目录，vpath <pattern> <directories>:表示匹配该模式的的文件去相应的目录查找，pattern常用%用于匹配零个或任意多个字符；vpath <pattern>:表示清除匹配的文件的搜索文件夹；vpath:表示清除设置号的搜索文件夹；
8. $(funcname parameter1,parameter2):表示执行一个Makefile中的函数；$@用于表示目标中的集合，该符号称为“自己主动化变量“
9. 静态模式：静态模式下的语法变为：

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

<commands>

* targets: 代表目标集合
* target-pattern: 目标的匹配模式，用于过滤某些目标
* prereq-patterns: 依赖的匹配模式，用于确定相应的依赖

1. 在大型工程中一般每个模块会有一个单独的Makefile文件，在顶层的Makefile可以使用如下的方式来运行底层的Makefile

|  |
| --- |
| subsystem:  cd subdir && $(MAKE)  或  subsystem:  $(MAKE) -C subdir |

可以使用export在上下层Makefile中传递变量，如果不想让上层的Makefile传递到下层Makefile的话，可以使用unexport；

1. 如果一个Makefile中总是出现相同的命令序列，则可以将这个命令序列定义为一个变量，也成为命令包，以“define“开始，以“endef”结束；使用的时候像变量一样放在$()中；该种方式可以定义多行变量；

|  |
| --- |
| define run-func  echo hello-world  endef  all:  $(run-func) |

1. 条件选择

|  |
| --- |
| ifeq/ifneq (<arg1>, <arg2> )  command  else  command  endif  ifdef/ifndef <variable-name>  command  else  command  endif |

1. 使用函数

|  |
| --- |
| $(<function> <arguments> )  或  ${<function> <arguments>}  # function是函数名，<arguments>是函数的参数，参数使用逗号隔开，函数名和参数之间使用逗号隔开；  支持的函数：  $(subst <from>,<to>,<text> )：把字串<text>中的<from>字符串替换成<to>。  $(patsubst <pattern>,<replacement>,<text> )：查找<text>中的单词（单词以“空格”、“Tab”或“回车”“换行”分隔）是否符合模式<pattern>，假设匹配的话，则以<replacement>替换。  $(strip <string> )：去掉<string>字串中开头和结尾的空字符。  $(findstring <find>,<in> )：在字串<in>中查找<find>子串。  $(filter <pattern...>,<text> )：以<pattern>模式过滤<text>字符串中的单词，保留符合模式<pattern>的单词。  $(filter-out <pattern...>,<text> )：以<pattern>模式过滤<text>字符串中的单词，去除符合模式<pattern>的单词。能够有多个模式。  $(sort <list> )：给字符串<list>中的单词排序（升序）。  $(word <n>,<text> )：取字符串<text>中第<n>个单词。（从1开始）。  $(wordlist <s>,<e>,<text> )：从字符串<text>中取从<s>开始到<e>的单词串。<s>和<e>是一个数字。  $(words <text> )：统计<text>中字符串中的单词个数。  $(firstword <text> )：取字符串<text>中的第一个单词。  $(dir <names...> )：从文件名称序列<names>中取出文件夹部分。文件夹部分是指最后一个反斜杠（“/”）之前的部分。假设沒有反斜杠，那么返回“./”。  $(suffix <names...> )：从文件名称序列<names>中取出各个文件名称的后缀。  $(basename <names...> )：从文件名称序列<names>中取出各个文件名称的前缀部分。  $(addsuffix <suffix>,<names...> )：把后缀<suffix>加到<names>中的每一个单词后面。  $(addprefix <prefix>,<names...> )：把前缀<prefix>加到<names>中的每一个单词后面。  $(join <list1>,<list2> )：把<list2>中的单词对应地加到<list1>的单词后面。  $(foreach <var>,<list>,<text> )：把參数<list>中的单词逐一取出放到參数<var>所指定的变量中，然后再运行<text>所包括的表达式。  $(if <condition>,<then-part> )或$(if <condition>,<then-part>,<else-part> )：条件  $(call <expression>,<parm1>,<parm2>,<parm3>...)：call函数是唯一一个能够用来创建新的參数化的函数。  $(origin <variable> )：其返回值来告诉你这个变量的“出生情况”，<variable>是变量的名字，不应该是引用。  contents := $(shell command parameter)：shell函数把运行操作系统命令后的输出作为函数返回。  $(error <text ...> )：产生致命错误。  $(warning <text ...> )：产生警告信息 |

1. 使用-f指定Makefile文件

* **通配符：**

1. make支持三种通配符:”\*”, “?”和“[…]”

* **书写命令**

1. 每条命令必须以TAB键开头，和依赖文件同行则需要添加分号；
2. make会把执行的命令输出到屏幕上，在命令前加@时在运行期间则不会输出；
3. make -n或make -just-print会输出make执行期间的命令，很方便Makefile的调试；
4. 命令的运行是逐行从上至下运行的，当两条命令之间由前后驱关系式，必须写在一行，并且使用分号隔开；
5. 如果命令中的某行运行出错，则会使make终止，如果是一些可以忽略的错误，则可以在命令前添加一个小减号”-”，如果要忽略Makefile中的所有错误，则可以在make命令后面添加-i或-ignore-errors，另外-k和--keep-going参数可以跳过命令出错的规则而继续执行其他的规则；

* **自动生成依赖**

cc -M source.c和gcc -MM source.c 两个命令可以生成源文件的依赖文件，使用该命令来编写Makefile,使用方法是使用该命令生成一个中间文件，然后在Makefile中使用include关键字把中间文件包含进来，生成.d文件的的命令如下：

|  |
| --- |
| %.d: %.c  @set -e; rm -f $@; /  $(CC) -M $(CPPFLAGS) $< > $@.$$$$; /  sed 's,/($\*/)/.o[ :]\*,/1.o $@ : ,g' < $@.$$$$ > $@; /  rm -f $@.$$$$  # $$$$是一个随机编号 |

Makefile中inflcude时使用该命令：

|  |
| --- |
| sources = foo.c bar.c  include $(sources:.c=.d) # 将相应的sources文件中的.c替换为.d |

* **伪目标：**

伪目标并非一个文件，仅仅是一个标签，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他是否要运行，唯独通过显示地指明这个目标才让其生效。注意，伪目标的取名不能和文件重名，为了避免出现这种情况，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。另外，伪目标一般没有依赖文件，但可以作为依赖文件；

* **使用变量**

1. 变量可以包含字母，数字和下划线，并且可以数字开头，但不能包含=,#,;或是空格
2. 变量对大小写敏感
3. 变量在声明时必须赋值，在使用时在变量名前加$或使用$( )或${ }来还原变量，如果Makefile中需要使用真是的$,则使用$$;
4. 变量可以使用在目标，依赖或命令以及新的变量中；
5. 使用 = 操作符给变量赋值时，它不在乎变量定义的顺序，值中的变量不一定非要是已定义好的顺序，
6. 使用 := 操作符给变量赋值时，值中的变量必须是已经定义好的变量，并且是在该变量之前，不能使用之后申请的变量；
7. $(MAKELEVEL)表示在嵌套make中时该Makefile所处的层次，从0开始
8. ?= 表示如果变量已经定义了，那就什么都不做，如果变量没有定义，则把后面的值赋予变量；
9. 变量的替换：$(var:a=b)表示把变量var中以a结尾的部分都替换为b，结尾表示的是空格或结束符；
10. 变量的值可以再当成一个变量；
11. +=操作符可以给变量追加值；如果变量之前未顶一个过，那么+=会自动变为=，如果变量定义过，那么+=会继承前次赋值的操作符；
12. 假设变量通常通过make的命令行参数设置，那么Makefile中的变量的赋值会被忽略，该种类别的变量再Makefile中定义时可以在其变量前添加override关键字；
13. 环境变量可以在系统中所有的Makefile中使用，本地Makefile的局部值会覆盖全局的值，如果需要再不同层级Makefile之间传递变量，默认只传递make参数，此时需要使用export关键字导出变量未全局变量；
14. 目标变量只对该条规则以及连带规则中有效

|  |
| --- |
| <target ...> : <variable-assignment>  <target ...> : overide <variable-assignment> |

1. 模式变量，将变量绑定再模式上，模式中至少含有一个百分号%，只要符合匹配模式的目标都有该变量

|  |
| --- |
| <pattern ...> : <variable-assignment>  <pattern ...> : override <variable-assignment>  %.o : CFLAGS = -O |

1. MAKECMDGOALS变量会包含make运行时指定的目标，使用默认目标不填写时该值为空；

* **自己主动化变量：**

1. $@: 目标中的集合
2. $<: 表示依赖目标集合, 依赖目标中的第一个目标名字。假设依赖目标是以模式（即"%"）定义的，那么"$<"将是符合模式的一系列的文件集。注意，其是一个一个取出来的。
3. $%:表示规则中的目标成员名。比如，假设一个目标是"foo.a(bar.o)"，那么，"$%"就是"bar.o"，"$@"就是"foo.a";
4. $?:全部比目标新的依赖目标的集合。
5. $^:全部的依赖目标的集合。以空格分隔。假设在依赖目标中有多个反复的，那个这个变量会去除反复的依赖目标，仅仅保留一份。
6. $+:这个变量非常像"$^"，也是全部依赖目标的集合。仅仅是它不去除反复的依赖目标。

* **注意点：**

1. 在定义好依赖关系后，后续的那一行定义了怎样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。
2. make管理的只是文件间的依赖性，对于命令执行出错时不会导致make终止的，只有当依赖存在问题是才会终止make。
3. clean从来都是放在文件的最后，放在文件开头会被作为默认目标。
4. 在include前面能够有一些空字符，可是绝不能是[Tab]键开始。
5. 如果一些无关紧要的命令有可能出现错误而又不想其使编译过程中断，可以在该命令前加上一个减号”-”
6. 伪目标的取名不能和文件重名；
7. 总控Makefiel的变量会传递到下级的Makefile中，但是不会覆盖下层Makefile中定义的变量，除非指定参数e；
8. 注意变量后的注释符#表示变量定义终止的地方，如：

|  |
| --- |
| dir := /foo/bar # directory |

以上定义的变量值为/foo/bar加四个空格，所以用到该变量时会有问题；

1. 自己主动化变量只用再运行时才有值；

https://www.cnblogs.com/mfryf/p/3305778.html