Make 命令教程

代码变成可执行文件，叫做[编译](http://www.ruanyifeng.com/blog/2014/11/compiler.html)（compile）；先编译这个，还是先编译那个（即编译的安排），叫做[构建](http://en.wikipedia.org/wiki/Software_build" \t "_blank)（build）。

[Make](http://en.wikipedia.org/wiki/Make_%28software%29)是最常用的构建工具，诞生于1977年，主要用于C语言的项目。但是实际上 ，任何只要某个文件有变化，就要重新构建的项目，都可以用Make构建。

本文介绍Make命令的用法，从简单的讲起，不需要任何基础，只要会使用命令行，就能看懂。我的参考资料主要是Isaac Schlueter的[《Makefile文件教程》](https://gist.github.com/isaacs/62a2d1825d04437c6f08" \t "_blank)和[《GNU Make手册》](https://www.gnu.org/software/make/manual/make.html)。

一、Make的概念

Make这个词，英语的意思是"制作"。Make命令直接用了这个意思，就是要做出某个文件。比如，要做出文件a.txt，就可以执行下面的命令。

$ make a.txt

但是，如果你真的输入这条命令，它并不会起作用。因为Make命令本身并不知道，如何做出a.txt，需要有人告诉它，如何调用其他命令完成这个目标。

比如，假设文件 a.txt 依赖于 b.txt 和 c.txt ，是后面两个文件连接（cat命令）的产物。那么，make 需要知道下面的规则。

a.txt: b.txt c.txt

cat b.txt c.txt > a.txt

也就是说，make a.txt 这条命令的背后，实际上分成两步：第一步，确认 b.txt 和 c.txt 必须已经存在，第二步使用 cat 命令 将这个两个文件合并，输出为新文件。

像这样的规则，都写在一个叫做Makefile的文件中，Make命令依赖这个文件进行构建。Makefile文件也可以写为makefile， 或者用命令行参数指定为其他文件名。

$ make -f rules.txt

# 或者

$ make --file=rules.txt

上面代码指定make命令依据rules.txt文件中的规则，进行构建。

总之，make只是一个根据指定的Shell命令进行构建的工具。它的规则很简单，你规定要构建哪个文件、它依赖哪些源文件，当那些文件有变动时，如何重新构建它。

二、Makefile文件的格式

构建规则都写在Makefile文件里面，要学会如何Make命令，就必须学会如何编写Makefile文件。

2.1 概述

Makefile文件由一系列规则（rules）构成。每条规则的形式如下。

<target> : <prerequisites>

[tab] <commands>

上面第一行冒号前面的部分，叫做"目标"（target），冒号后面的部分叫做"前置条件"（prerequisites）；第二行必须由一个tab键起首，后面跟着"命令"（commands）。

"目标"是必需的，不可省略；"前置条件"和"命令"都是可选的，但是两者之中必须至少存在一个。

每条规则就明确两件事：构建目标的前置条件是什么，以及如何构建。下面就详细讲解，每条规则的这三个组成部分。

2.2 目标（target）

一个目标（target）就构成一条规则。目标通常是文件名，指明Make命令所要构建的对象，比如上文的 a.txt 。目标可以是一个文件名，也可以是多个文件名，之间用空格分隔。

除了文件名，目标还可以是某个操作的名字，这称为"伪目标"（phony target）。

clean:

rm \*.o

上面代码的目标是clean，它不是文件名，而是一个操作的名字，属于"伪目标 "，作用是删除对象文件。

$ make clean

但是，如果当前目录中，正好有一个文件叫做clean，那么这个命令不会执行。因为Make发现clean文件已经存在，就认为没有必要重新构建了，就不会执行指定的rm命令。

为了避免这种情况，可以明确声明clean是"伪目标"，写法如下。

.PHONY: clean

clean:

rm \*.o temp

声明clean是"伪目标"之后，make就不会去检查是否存在一个叫做clean的文件，而是每次运行都执行对应的命令。像.PHONY这样的内置目标名还有不少，可以查看[手册](http://www.gnu.org/software/make/manual/html_node/Special-Targets.html" \l "Special-Targets" \t "_blank)。

如果Make命令运行时没有指定目标，默认会执行Makefile文件的第一个目标。

$ make

上面代码执行Makefile文件的第一个目标。

2.3 前置条件（prerequisites）

前置条件通常是一组文件名，之间用空格分隔。它指定了"目标"是否重新构建的判断标准：只要有一个前置文件不存在，或者有过更新（前置文件的last-modification时间戳比目标的时间戳新），"目标"就需要重新构建。

result.txt: source.txt

cp source.txt result.txt

上面代码中，构建 result.txt 的前置条件是 source.txt 。如果当前目录中，source.txt 已经存在，那么make result.txt可以正常运行，否则必须再写一条规则，来生成 source.txt 。

source.txt:

echo "this is the source" > source.txt

上面代码中，source.txt后面没有前置条件，就意味着它跟其他文件都无关，只要这个文件还不存在，每次调用make source.txt，它都会生成。

$ make result.txt

$ make result.txt

上面命令连续执行两次make result.txt。第一次执行会先新建 source.txt，然后再新建 result.txt。第二次执行，Make发现 source.txt 没有变动（时间戳晚于 result.txt），就不会执行任何操作，result.txt 也不会重新生成。

如果需要生成多个文件，往往采用下面的写法。

source: file1 file2 file3

上面代码中，source 是一个伪目标，只有三个前置文件，没有任何对应的命令。

$ make source

执行make source命令后，就会一次性生成 file1，file2，file3 三个文件。这比下面的写法要方便很多。

$ make file1

$ make file2

$ make file3

2.4 命令（commands）

命令（commands）表示如何更新目标文件，由一行或多行的Shell命令组成。它是构建"目标"的具体指令，它的运行结果通常就是生成目标文件。

每行命令之前必须有一个tab键。如果想用其他键，可以用内置变量.RECIPEPREFIX声明。

.RECIPEPREFIX = >

all:

> echo Hello, world

上面代码用.RECIPEPREFIX指定，大于号（>）替代tab键。所以，每一行命令的起首变成了大于号，而不是tab键。

需要注意的是，每行命令在一个单独的shell中执行。这些Shell之间没有继承关系。

var-lost:

export foo=bar

echo "foo=[$$foo]"

上面代码执行后（make var-lost），取不到foo的值。因为两行命令在两个不同的进程执行。一个解决办法是将两行命令写在一行，中间用分号分隔。

var-kept:

export foo=bar; echo "foo=[$$foo]"

另一个解决办法是在换行符前加反斜杠转义。

var-kept:

export foo=bar; \

echo "foo=[$$foo]"

最后一个方法是加上.ONESHELL:命令。

.ONESHELL:

var-kept:

export foo=bar;

echo "foo=[$$foo]"

三、Makefile文件的语法

3.1 注释

井号（#）在Makefile中表示注释。

# 这是注释

result.txt: source.txt

# 这是注释

cp source.txt result.txt # 这也是注释

3.2 回声（echoing）

正常情况下，make会打印每条命令，然后再执行，这就叫做回声（echoing）。

test:

# 这是测试

执行上面的规则，会得到下面的结果。

$ make test

# 这是测试

在命令的前面加上@，就可以关闭回声。

test:

@# 这是测试

现在再执行make test，就不会有任何输出。

由于在构建过程中，需要了解当前在执行哪条命令，所以通常只在注释和纯显示的echo命令前面加上@。

test:

@# 这是测试

@echo TODO

3.3 通配符

通配符（wildcard）用来指定一组符合条件的文件名。Makefile 的通配符与 Bash 一致，主要有星号（\*）、问号（？）和 [...] 。比如， \*.o 表示所有后缀名为o的文件。

clean:

rm -f \*.o

3.4 模式匹配

Make命令允许对文件名，进行类似正则运算的匹配，主要用到的匹配符是%。比如，假定当前目录下有 f1.c 和 f2.c 两个源码文件，需要将它们编译为对应的对象文件。

%.o: %.c

等同于下面的写法。

f1.o: f1.c

f2.o: f2.c

使用匹配符%，可以将大量同类型的文件，只用一条规则就完成构建。

3.5 变量和赋值符

Makefile 允许使用等号自定义变量。

txt = Hello World

test:

@echo $(txt)

上面代码中，变量 txt 等于 Hello World。调用时，变量需要放在 $( ) 之中。

调用Shell变量，需要在美元符号前，再加一个美元符号，这是因为Make命令会对美元符号转义。

test:

@echo $$HOME

有时，变量的值可能指向另一个变量。

v1 = $(v2)

上面代码中，变量 v1 的值是另一个变量 v2。这时会产生一个问题，v1 的值到底在定义时扩展（静态扩展），还是在运行时扩展（动态扩展）？如果 v2 的值是动态的，这两种扩展方式的结果可能会差异很大。

为了解决类似问题，Makefile一共提供了四个赋值运算符 （=、:=、？=、+=），它们的区别请看[StackOverflow](http://stackoverflow.com/questions/448910/makefile-variable-assignment" \t "_blank)。

VARIABLE = value

# 在执行时扩展，允许递归扩展。

VARIABLE := value

# 在定义时扩展。

VARIABLE ?= value

# 只有在该变量为空时才设置值。

VARIABLE += value

# 将值追加到变量的尾端。

3.6 内置变量（Implicit Variables）

Make命令提供一系列内置变量，比如，$(CC) 指向当前使用的编译器，$(MAKE) 指向当前使用的Make工具。这主要是为了跨平台的兼容性，详细的内置变量清单见[手册](https://www.gnu.org/software/make/manual/html_node/Implicit-Variables.html" \t "_blank)。

output:

$(CC) -o output input.c

3.7 自动变量（Automatic Variables）

Make命令还提供一些自动变量，它们的值与当前规则有关。主要有以下几个。

**（1）$@**

$@指代当前目标，就是Make命令当前构建的那个目标。比如，make foo的 $@ 就指代foo。

a.txt b.txt:

touch $@

等同于下面的写法。

a.txt:

touch a.txt

b.txt:

touch b.txt

**（2）$<**

$< 指代第一个前置条件。比如，规则为 t: p1 p2，那么$< 就指代p1。

a.txt: b.txt c.txt

cp $< $@

等同于下面的写法。

a.txt: b.txt c.txt

cp b.txt a.txt

**（3）$?**

$? 指代比目标更新的所有前置条件，之间以空格分隔。比如，规则为 t: p1 p2，其中 p2 的时间戳比 t 新，$?就指代p2。

**（4）$^**

$^ 指代所有前置条件，之间以空格分隔。比如，规则为 t: p1 p2，那么 $^ 就指代 p1 p2 。

**（5）$\***

$\* 指代匹配符 % 匹配的部分， 比如% 匹配 f1.txt 中的f1 ，$\* 就表示 f1。

**（6）$(@D) 和 $(@F)**

$(@D) 和 $(@F) 分别指向 $@ 的目录名和文件名。比如，$@是 src/input.c，那么$(@D) 的值为 src ，$(@F) 的值为 input.c。

**（7）$(<D) 和 $(<F)**

$(<D) 和 $(<F) 分别指向 $< 的目录名和文件名。

所有的自动变量清单，请看[手册](https://www.gnu.org/software/make/manual/html_node/Automatic-Variables.html" \t "_blank)。下面是自动变量的一个例子。

dest/%.txt: src/%.txt

@[ -d dest ] || mkdir dest

cp $< $@

上面代码将 src 目录下的 txt 文件，拷贝到 dest 目录下。首先判断 dest 目录是否存在，如果不存在就新建，然后，$< 指代前置文件（src/%.txt）， $@ 指代目标文件（dest/%.txt）。

3.8 判断和循环

Makefile使用 Bash 语法，完成判断和循环。

ifeq ($(CC),gcc)

libs=$(libs\_for\_gcc)

else

libs=$(normal\_libs)

endif

上面代码判断当前编译器是否 gcc ，然后指定不同的库文件。

LIST = one two three

all:

for i in $(LIST); do \

echo $$i; \

done

# 等同于

all:

for i in one two three; do \

echo $i; \

done

上面代码的运行结果。

one

two

three

3.9 函数

Makefile 还可以使用函数，格式如下。

$(function arguments)

# 或者

${function arguments}

Makefile提供了许多[内置函数](http://www.gnu.org/software/make/manual/html_node/Functions.html" \t "_blank)，可供调用。下面是几个常用的内置函数。

**（1）shell 函数**

shell 函数用来执行 shell 命令

srcfiles := $(shell echo src/{00..99}.txt)

**（2）wildcard 函数**

wildcard 函数用来在 Makefile 中，替换 Bash 的通配符。

srcfiles := $(wildcard src/\*.txt)

**（3）subst 函数**

subst 函数用来文本替换，格式如下。

$(subst from,to,text)

下面的例子将字符串"feet on the street"替换成"fEEt on the strEEt"。

$(subst ee,EE,feet on the street)

下面是一个稍微复杂的例子。

comma:= ,

empty:=

# space变量用两个空变量作为标识符，当中是一个空格

space:= $(empty) $(empty)

foo:= a b c

bar:= $(subst $(space),$(comma),$(foo))

# bar is now `a,b,c'.

**（4）patsubst函数**

patsubst 函数用于模式匹配的替换，格式如下。

$(patsubst pattern,replacement,text)

下面的例子将文件名"x.c.c bar.c"，替换成"x.c.o bar.o"。

$(patsubst %.c,%.o,x.c.c bar.c)

**（5）替换后缀名**

替换后缀名函数的写法是：变量名 + 冒号 + 后缀名替换规则。它实际上patsubst函数的一种简写形式。

min: $(OUTPUT:.js=.min.js)

上面代码的意思是，将变量OUTPUT中的后缀名 .js 全部替换成 .min.js 。

四、Makefile 的实例

**（1）执行多个目标**

.PHONY: cleanall cleanobj cleandiff

cleanall : cleanobj cleandiff

rm program

cleanobj :

rm \*.o

cleandiff :

rm \*.diff

上面代码可以调用不同目标，删除不同后缀名的文件，也可以调用一个目标（cleanall），删除所有指定类型的文件。

**（2）编译C语言项目**

edit : main.o kbd.o command.o display.o

cc -o edit main.o kbd.o command.o display.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

cc -c display.c

clean :

rm edit main.o kbd.o command.o display.o

.PHONY: edit clean