# make 的运行

一般来说，最简单的就是直接在命令行下输入make命令，make命令会找当前目录的makefile来执行，一切都是自动的。但也有时你也许只想让make重编译某些文件，而不是整个工程，而又有的时候你有几套编译规则，你想在不同的时候使用不同的编译规则，等等。本章节就是讲述如何使用 make命令的。

## make的退出码

make命令执行后有三个退出码：

0 —— 表示成功执行。

1 —— 如果make运行时出现任何错误，其返回1。

2 —— 如果你使用了make的“-q”选项，并且make使得一些目标不需要更新，那么返回2。

Make的相关参数我们会在后续章节中讲述。

## 指定Makefile

前面我们说过，GNU make找寻默认的Makefile的规则是在当前目录下依次找三个文件——“GNUmakefile”、“makefile”和“Makefile”。其按顺序找这三个文件，一旦找到，就开始读取这个文件并执行。

当然，我们也可以给make命令指定一个特殊名字的Makefile。要达到这个功能，我们要使用make的“-f”或是“--file”参数（“--makefile”参数也行）。例如，我们有个makefile的名字是“hchen.mk”，那么，我们可以这样来让make来执行这个文件：

make –f hchen.mk

如果在make的命令行是，你不只一次地使用了“-f”参数，那么，所有指定的makefile将会被连在一起传递给make执行。

## 指定目标

一般来说，make的最终目标是makefile中的第一个目标，而其它目标一般是由这个目标连带出来的。这是make的默认行为。当然，一般来说，你的makefile中的第一个目标是由许多个目标组成，你可以指示make，让其完成你所指定的目标。要达到这一目的很简单，需在make命令后直接跟目标的名字就可以完成（如前面提到的“make clean”形式）

任何在makefile中的目标都可以被指定成终极目标，但是除了以“-”打头，或是包含了“=”的目标，因为有这些字符的目标，会被解析成命令行参数或是变量。甚至没有被我们明确写出来的目标也可以成为make的终极目标，也就是说，只要make可以找到其隐含规则推导规则，那么这个隐含目标同样可以被指定成终极目标。

有一个make的环境变量叫“MAKECMDGOALS”，这个变量中会存放你所指定的终极目标的列表，如果在命令行上，你没有指定目标，那么，这个变量是空值。这个变量可以让你使用在一些比较特殊的情形下。比如下面的例子：

sources = foo.c bar.c

ifneq ( $(MAKECMDGOALS),clean)

include $(sources:.c=.d)

endif

基于上面的这个例子，只要我们输入的命令不是“make clean”，那么makefile会自动包含“foo.d”和“bar.d”这两个makefile。

使用指定终极目标的方法可以很方便地让我们编译我们的程序，例如下面这个例子：

.PHONY: all

all: prog1 prog2 prog3 prog4

从这个例子中，我们可以看到，这个makefile中有四个需要编译的程序——“prog1”， “prog2”， “prog3”和 “prog4”，我们可以使用“make all”命令来编译所有的目标（如果把all置成第一个目标，那么只需执行“make”），我们也可以使用 “make prog2”来单独编译目标“prog2”。

即然make可以指定所有makefile中的目标，那么也包括“伪目标”，于是我们可以根据这种性质来让我们的makefile根据指定的不同的目标来完成不同的事。在Unix世界中，软件发布时，特别是GNU这种开源软件的发布时，其makefile都包含了编译、安装、打包等功能。我们可以参照这种规则来书写我们的makefile中的目标。

**“all”**

这个伪目标是所有目标的目标，其功能一般是编译所有的目标。

**clean”**

这个伪目标功能是删除所有被make创建的文件。

**“install”**

这个伪目标功能是安装已编译好的程序，其实就是把目标执行文件拷贝到指定的目标中去。

**print”**

这个伪目标的功能是例出改变过的源文件。

**“tar”**

这个伪目标功能是把源程序打包备份。也就是一个tar文件。

**“dist”**

这个伪目标功能是创建一个压缩文件，一般是把tar文件压成Z文件。或是gz文件。

**TAGS”**

这个伪目标功能是更新所有的目标，以备完整地重编译使用。

**“check”和“test”**

这两个伪目标一般用来测试makefile的流程。

当然一个项目的makefile中也不一定要书写这样的目标，这些东西都是GNU的东西，但是我想，GNU搞出这些东西一定有其可取之处（等你的 UNIX下的程序文件一多时你就会发现这些功能很有用了），这里只不过是说明了，如果你要书写这种功能，最好使用这种名字命名你的目标，这样规范一些，规范的好处就是——不用解释，大家都明白。而且如果你的makefile中有这些功能，一是很实用，二是可以显得你的makefile很专业（不是那种初学者的作品）。

## 检查规则

有时候，我们不想让我们的makefile中的规则执行起来，我们只想检查一下我们的命令，或是执行的序列。于是我们可以使用make命令的下述参数：

“-n” “--just-print” “--dry-run” “--recon” 不执行参数，这些参数只是打印命令，不管目标是否更新，把规则和连带规则下的命令打印出来，但不执行，这些参数对于我们调试makefile很有用处。

“-t” “--touch” 这个参数的意思就是把目标文件的时间更新，但不更改目标文件。也就是说，make假装编译目标，但不是真正的编译目标，只是把目标变成已编译过的状态。

“-q” “--question” 这个参数的行为是找目标的意思，也就是说，如果目标存在，那么其什么也不会输出，当然也不会执行编译，如果目标不存在，其会打印出一条出错信息。

“-W <file>;” “--what-if=<file>;” “--assume-new=<file>;” “--new-file=<file>;” 这个参数需要指定一个文件。一般是是源文件（或依赖文件），Make会根据规则推导来运行依赖于这个文件的命令，一般来说，可以和“-n”参数一同使用，来查看这个依赖文件所发生的规则命令。

另外一个很有意思的用法是结合“-p”和“-v”来输出makefile被执行时的信息（这个将在后面讲述）。

## make的参数

下面列举了所有GNU make 3.80版的参数定义。其它版本和产商的make大同小异，不过其它产商的make的具体参数还是请参考各自的产品文档。

“-b” “-m” 这两个参数的作用是忽略和其它版本make的兼容性。

“-B” “--always-make” 认为所有的目标都需要更新（重编译）。

“-C <dir>” “--directory=<dir>” 指定读取makefile的目录。如果有多个“-C”参数，make的解释是后面的路径以前面的作为相对路径，并以最后的目录作为被指定目录。如：“make –C ~hchen/test –C prog”等价于“make –C ~hchen/test/prog”。

“—debug[=<options>]” 输出make的调试信息。它有几种不同的级别可供选择，如果没有参数，那就是输出最简单的调试信息。下面是<options>的取值：

a —— 也就是all，输出所有的调试信息。（会非常的多）

b —— 也就是basic，只输出简单的调试信息。即输出不需要重编译的目标。

v —— 也就是verbose，在b选项的级别之上。输出的信息包括哪个makefile被解析，不需要被重编译的依赖文件（或是依赖目标）等。

i —— 也就是implicit，输出所有的隐含规则。

j —— 也就是jobs，输出执行规则中命令的详细信息，如命令的PID、返回码等。

m —— 也就是makefile，输出make读取makefile，更新makefile，执行makefile的信息。

“-d” 相当于“--debug=a”。

“-e” “--environment-overrides” 指明环境变量的值覆盖makefile中定义的变量的值。

“-f=<file>” “--file=<file>” “--makefile=<file>” 指定需要执行的makefile。

“-h” “--help” 显示帮助信息。

“-i” “--ignore-errors” 在执行时忽略所有的错误。

“-I <dir>” “--include-dir=<dir>” 指定一个被包含makefile的搜索目标。可以使用多个“-I”参数来指定多个目录。

“-j [<jobsnum>]” “--jobs[=<jobsnum>]” 指同时运行命令的个数。如果没有这个参数，make运行命令时能运行多少就运行多少。如果有一个以上的“-j”参数，那么仅最后一个“-j”才是有效的。（注意这个参数在MS-DOS中是无用的）

“-k” “--keep-going” 出错也不停止运行。如果生成一个目标失败了，那么依赖于其上的目标就不会被执行了。

“-l <load>” “--load-average[=<load]” “—max-load[=<load>]” 指定make运行命令的负载。

“-n” “--just-print” “--dry-run” “--recon” 仅输出执行过程中的命令序列，但并不执行。

“-o <file>” “--old-file=<file>” “--assume-old=<file>” 不重新生成的指定的<file>，即使这个目标的依赖文件新于它。

“-p” “--print-data-base” 输出makefile中的所有数据，包括所有的规则和变量。这个参数会让一个简单的makefile都会输出一堆信息。如果你只是想输出信息而不想执行makefile，你可以使用“make -qp”命令。如果你想查看执行makefile前的预设变量和规则，你可以使用 “make –p –f /dev/null”。这个参数输出的信息会包含着你的makefile文件的文件名和行号，所以，用这个参数来调试你的 makefile会是很有用的，特别是当你的环境变量很复杂的时候。

“-q” “--question” 不运行命令，也不输出。仅仅是检查所指定的目标是否需要更新。如果是0则说明要更新，如果是2则说明有错误发生。

“-r” “--no-builtin-rules” 禁止make使用任何隐含规则。

“-R” “--no-builtin-variabes” 禁止make使用任何作用于变量上的隐含规则。

“-s” “--silent” “--quiet” 在命令运行时不输出命令的输出。

“-S” “--no-keep-going” “--stop” 取消“-k”选项的作用。因为有些时候，make的选项是从环境变量“MAKEFLAGS”中继承下来的。所以你可以在命令行中使用这个参数来让环境变量中的“-k”选项失效。

“-t” “--touch” 相当于UNIX的touch命令，只是把目标的修改日期变成最新的，也就是阻止生成目标的命令运行。

“-v” “--version” 输出make程序的版本、版权等关于make的信息。

“-w” “--print-directory” 输出运行makefile之前和之后的信息。这个参数对于跟踪嵌套式调用make时很有用。

“--no-print-directory” 禁止“-w”选项。

“-W <file>” “--what-if=<file>” “--new-file=<file>” “--assume-file=<file>” 假定目标<file>;需要更新，如果和“-n”选项使用，那么这个参数会输出该目标更新时的运行动作。如果没有“-n”那么就像运行UNIX的“touch”命令一样，使得<file>;的修改时间为当前时间。

“--warn-undefined-variables” 只要make发现有未定义的变量，那么就输出警告信息。