# 概述

## 引言

什么是makefile？或许很多Windows的程序员都不知道这个东西，因为那些Windows的集成开发环境（integrated development environment，IDE）都为你做了这个工作，但我觉得要作一个好的和专业的程序员，makefile还是要懂。

因为，makefile关系到了整个工程的编译规则。一个工程中的源文件不计其数，并且按类型、功能、模块分别放在若干个目录中，makefile定义了一系列的规则来指定，哪些文件需要先编译，哪些文件需要后编译，哪些文件需要重新编译，甚至于进行更复杂的功能操作，因为makefile就像一个Shell脚本一样，其中也可以执行操作系统的命令。

makefile带来的好处就是——“自动化编译”，一旦写好，只需要一个make命令，整个工程完全自动编译，极大的提高了软件开发的效率。 make是一个命令工具，是一个解释makefile中指令的命令工具，一般来说，大多数的IDE都有这个命令，比如：Delphi的make，Visual C++的nmake，Linux下GNU的make。可见，makefile都成为了一种在工程方面的编译方法。

## 怎样生成可执行文件的

由C语言源代码文件生成可执行文件的过程要经历四个相互关联的步骤∶

1. 预处理(也称预编译，Preprocessing)

调用cpp进行预处理，生成以.i为后缀的预处理文件。在预处理过程中，对源代码文件中的文件包含(include)、预编译语句(如宏定义define等)进行分析；

1. 编译(Compilation)

调用cc1进行编译，生成以.s为后缀的汇编源文件。

1. 汇编(Assembly)

调用as进行汇编，生成以.o为后缀的目标文件。

1. 链接(Linking)

调用ld来生成可执行文件，在链接阶段，所有的目标文件被安排在可执行程序中的恰当的位置，同时，该程序所调用到的库函数也从各自所在的档案库中连到合适的地方。

# Makefile介绍

make命令执行时，需要一个makefile文件，以告诉make命令需要怎么样的去编译和链接程序。

## **makefile的规则**

<target > : <prerequisites> ...

<command>

...

...

**<target >**

可以是一个object file（目标文件），也可以是一个执行文件，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>**

生成该<target>所**依赖**的文件和/或<target>；

**<command>**

该<target>要执行的命令（任意的shell命令）；

**格式**

**冒号**前面是<**target>**，**冒号**后面是<**prerequisites>**， 另起一行以**Tab 键作为开头**是<**command>。**

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:



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。

## **Makefile里有什么？**

Makefile里主要包含了五个东西：显式规则、隐晦规则、变量定义、文件指示和注释。

**显式规则**

显式规则说明了如何生成一个或多个目标文件。这是由Makefile的书写者明显指出要生成的文件、文件的依赖文件和生成的命令。

**隐晦规则**

由于我们的make有自动推导的功能，所以隐晦的规则可以让我们比较简略地书写 Makefile，这是由make所支持的。

**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，这个有点像你C语言中的宏，当Makefile被执行时，其中的变量都会被扩展到相应的引用位置上。

**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。

**注释**

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，可以用反斜杠进行转义，如： \# 。

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

## **Makefile的文件名**

默认的情况下，make命令会在当前目录下按顺序找寻文件名为“GNUmakefile”、“makefile”、“Makefile”的文件，找到了解释这个文件。在这三个文件名中，最好使用“Makefile”这个文件名，因为，这个文件名第一个字符为大写，这样有一种显目的感觉。最好不要用“GNUmakefile”，这个文件是GNU的make识别的。有另外一些make只对全小写的“makefile”文件名敏感，但是基本上来说，大多数的make都支持“makefile”和“Makefile”这两种默认文件名。

## **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（可以包含路径和通配符）。

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。

## **make的工作方式**

GNU的make工作时的执行步骤如下：（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。

# 书写规则

规则包含两个部分，一个是依赖关系，一个是生成目标的方法。

在Makefile中，**规则的顺序**是很重要的，因为，Makefile中只应该有一个最终目标，其它的目标都是被这个目标所连带出来的，make中的**第一个目标是最终目标**。

## **规则举例**

foo.o: foo.c defs.h # foo模块

cc -c -g foo.c

其中：foo.o 是我们的目标， foo.c 和 defs.h 是目标**所依赖**的源文件，而只有一个命令 cc -c -g foo.c （以Tab键开头）。这个规则告诉我们两件事：

1. 文件的依赖关系， foo.o 依赖于 foo.c 和 defs.h 的文件，如果 foo.c 和 defs.h 的文件日期要比 foo.o 文件日期要新，或是 foo.o 不存在，那么依赖关系发生。
2. 生成或更新 foo.o 文件，就是那个cc命令。它说明了如何生成 foo.o 这个文件。（当然，foo.c文件include了defs.h文件）

## **规则的语法**

targets : prerequisites

command

...

或是这样：

targets : prerequisites ; command

command

...

**targets**是文件名，以空格分开，可以使用**通配符**。一般来说，我们的目标基本上是一个文件，但也有可能是多个文件；

**command**是命令行，如果其不与“target:prerequisites”在一行，那么，必须**以 Tab 键开头**，如果和prerequisites在一行，那么可以用**分号**做为分隔。

**prerequisites**也就是目标所依赖的文件（或依赖目标）。如果其中的某个文件要比目标文件要新，那么，目标就被认为是“过时的”，被认为是需要重生成的。

如果命令太长，你可以使用**反斜杠（ \ ）作为换行符**。make对一行上有多少个字符没有限制。规则告诉make两件事，文件的依赖关系和如何生成目标文件。

一般来说，make会以UNIX的标准Shell，也就是 /bin/sh 来执行命令。

## **在规则中使用通配符**

定义一系列比较类似的文件，可以使用通配符。make支持三个通配符： \*， ? 和 ~ 。

**波浪号（ ~ ）**字符在文件名中也有比较特殊的用途。如果是 ~/test ，这就表示环境变量“HOME”目录下的test目录。

通配符代替了你一系列的文件，如 **\*.c 表示所有后缀为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。

objects = \*.o

通配符同样在变量中时， \*.o不会展开，objects的值就是\*.o 。Makefile中的变量其实就是C/C++中的宏。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)

## **文件搜寻**

在一些大的工程中，有大量的源文件，我们通常的做法是把这许多的源文件分类，并存放在不同的目录中。所以，当make需要去找寻文件的依赖关系时，你可以在文件前加上路径，但最好的方法是把一个路径告诉make，让make在自动去找。

Makefile文件中的特殊变量 **VPATH** 就是完成这个功能的，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

上面的定义指定两个目录，“src”和“../headers”，make会按照这个顺序进行搜索。目录由**“冒号”**分隔。（当然，当前目录永远是最高优先搜索的地方）

## **伪目标**

我们生成了许多文件编译文件，我们也应该提供一个清除它们的“目标”（中间目标文件）以备完整地重编译而用，如下：

clean:

rm \*.o temp

“clean”就是一个“伪目标”。“伪目标”并不是一个文件，只是一个标签，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它是否要执行。我们只有通过显式地指明这个“目标”才能让其生效。（以“make clean”来使用该目标）

当然，“伪目标”的取名不能和文件名重名，不然其就失去了“伪目标”的意义了。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”来显式地指明一个目标是“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”。

.PHONY : clean

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

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。由于默认目标的特性是，总是被执行的，但由于“all”又是一个伪目标，伪目标只是一个标签不会生成文件，所以不会有“all”文件产生。于是，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”。（make all）（注：这里的显式“.PHONY : all” 不写的话一般情况也可以正确的执行，这样make可通过隐式规则推导出， “all” 是一个伪目标，执行make不会生成“all”文件，而执行后面的多个目标。建议：显式写出是一个好习惯。）

## **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。

gcc -M main.c

输出的是:

main.o: main.c defs.h /usr/include/stdio.h /usr/include/features.h \

/usr/include/sys/cdefs.h /usr/include/gnu/stubs.h \

/usr/lib/gcc-lib/i486-suse-linux/2.95.3/include/stddef.h \

/usr/include/bits/types.h /usr/include/bits/pthreadtypes.h \

/usr/include/bits/sched.h /usr/include/libio.h \

/usr/include/\_G\_config.h /usr/include/wchar.h \

/usr/include/bits/wchar.h /usr/include/gconv.h \

/usr/lib/gcc-lib/i486-suse-linux/2.95.3/include/stdarg.h \

/usr/include/bits/stdio\_lim.h

gcc -MM main.c

输出的则是:

main.o: main.c defs.h

那么，编译器的这个功能如何与我们的Makefile联系在一起呢。因为这样一来，我们的Makefile也要根据这些源文件重新生成，让 Makefile自已依赖于源文件？这个功能并不现实，不过我们可以有其它手段来迂回地实现这一功能。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自动更新或生成 .d 文件，并把其包含在我们的主Makefile中，这样，我们就可以自动化地生成每个文件的依赖关系了。

# 书写命令

每条规则中的命令和操作系统Shell的命令行是一致的。make会一按顺序一条一条的执行命令，每条命令的开头必须以 **Tab** 键开头，除非，命令是紧跟在依赖规则后面的分号后的。在命令行之间中的空格或是空行会被忽略，但是如果该空格或空行是以**Tab键开头**的，那么make会认为其是一个**空命令**。

## **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，我们用这个功能来向屏幕显示一些信息。如命令为:

@echo “正在编译XXX模块......”

输出:

“正在编译XXX模块……”

当命令为:

echo “正在编译XXX模块......”

**没有“@”**，那么，它会将命**令行一起输出**:

echo “正在编译XXX模块......”

“正在编译XXX模块......”

如果make执行时，带入make参数 -n 或 --just-print ，那么其只是显示命令，但不会执行命令，这个功能很有利于我们调试我们的Makefile，看看我们书写的命令是执行起来是什么样子的或是什么顺序的。

而make参数 -s 或 --silent 或 --quiet 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。如：

示例一：

exec:

cd /home/hchen

pwd

示例二：

exec:

cd /home/hchen; pwd

当我们执行make exec 时，第一个例子中的cd没有作用，pwd会打印出当前的Makefile目录，而第二个例子中，cd就起作用了，pwd会打印出“/home/hchen”。

make一般是使用环境变量SHELL中所定义的系统Shell来执行命令，默认情况下使用UNIX的标准Shell——/bin/sh来执行命令。但在MS-DOS下有点特殊，因为MS-DOS下没有SHELL环境变量，当然你也可以指定。如果你指定了UNIX风格的目录形式，首先，make会在SHELL所指定的路径中找寻命令解释器，如果找不到，其会在当前盘符中的当前目录中寻找，如果再找不到，其会在PATH环境变量中所定义的所有路径中寻找。MS-DOS中，如果你定义的命令解释器没有找到，其会给你的命令解释器加上诸如 .exe 、 .com 、 .bat 、 .sh 等后缀。

## 命令出错

为了**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。

还有一个要提一下的make的参数的是 **-k 或是 --keep-going** ，这个参数的意思是，如果某规则中的命令出错了，那么就终止该规则的执行，但继续执行其它规则。

## 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。

# 使用变量

在Makefile中的定义的变量，就像是C/C++语言中的宏一样，他代表了一个文本字串，在Makefile中执行的时候其会自动原模原样地展开在所使用的地方。其与C/C++所不同的是，你可以在Makefile中改变其值。在Makefile中，变量可以使用在“目标”，“依赖目标”， “命令”或是Makefile的其它部分中。

变量的命名字可以包含字符、数字，下划线（可以是数字开头），但不应该含有 : 、 # 、 = 或是空字符（空格、回车等）。变量是大小写敏感的，“foo”、“Foo”和“FOO”是三个不同的变量名。传统的Makefile的变量名是全大写的命名方式，但我推荐使用大小写搭配的变量名，如：MakeFlags。这样可以避免和系统的变量冲突，而发生意外的事情。

有一些变量是很奇怪字串，如 $< 、 $@ 等，这些是自动化变量，我会在后面介绍。

## 变量的基础

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。

变量可以使用在许多地方，如规则中的“目标”、“依赖”、“命令”以及新的变量中。先看一个例子：

objects = program.o foo.o utils.o

program : $(objects)

cc -o program $(objects)

$(objects) : defs.h

变量会在使用它的地方精确地展开，就像C/C++中的宏一样。

## 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。

1. 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? 。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

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，更糟糕的是，他会使用得两个make的函数“wildcard”和“shell”发生不可预知的错误。因为你不会知道这两个函数会被调用多少次。

1. 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：

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**。如：

y := $(x) bar

x := foo

那么，y的值是“bar”，而不是“foo bar”。

下面再介绍两个定义变量时我们需要知道的，请先看一个例子，如果我们要定义一个变量，其值是一个**空格**，那么我们可以这样来：

nullstring :=

space := $(nullstring) # end of the line

nullstring是一个Empty变量，其中什么也没有，而我们的space的值是一个空格。因为在操作符的右边是很难描述一个空格的，这里采用的技术很管用，先用一个Empty变量来标明变量的值开始了，而后面采用“#”注释符来表示变量定义的终止，这样，我们可以定义出**其值是一个空格的变量**。

请注意这里关于“#”的使用（“#”表示变量定义的终止），注释符“#”的这种特性值得我们注意，如果我们这样定义一个变量：

dir := /foo/bar # directory to put the frobs in

dir这个变量的值是**“/foo/bar”，后面还跟了4个空格**，如果我们这样使用这个变量来指定别的目录——“$(dir)/file”那么就完蛋了。

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，其等价于：

ifeq ($(origin FOO), undefined)

FOO = bar

Endif

## 变量高级用法

这里介绍两种变量的高级使用方法，第一种是变量值的**替换**。

我们可以替换变量中的共有的部分，其格式是 $(var:a=b) 或是 ${var:a=b} ，其意思是，把变量“var”中所有以“a”字串“结尾”的“a”替换成“b”字串。这里的“结尾”意思是“空格”或是“结束符”。

Eg:

foo := a.o b.o c.o

bar := $(foo:.o=.c)

这个示例中，我们先定义了一个 $(foo) 变量，而第二行的意思是把 $(foo) 中所有以 .o 字串“结尾”全部替换成 .c ，所以我们的 $(bar) 的值就是“a.c b.c c.c”。

另外一种变量替换的技术是以“静态模式”（参见前面章节）定义的，如：

foo := a.o b.o c.o

bar := $(foo:%.o=%.c)

这**依赖于被替换字串中的有相同的模式，模式中必须包含一个 % 字符**，这个例子同样让 $(bar) 变量的值为“a.c b.c c.c”。

第二种高级用法是——“**把变量的值再当成变量**”。先看一个例子：

x = y

y = z

a := $($(x))

在这个例子中，$(x)的值是“y”，所以$($(x))就是$(y)，于是$(a)的值就是“z”。（注意，是“x=y”，而不是“x=$(y)”）

在这种方式中，或要可以使用多个变量来组成一个变量的名字，然后再取其值：

first\_second = Hello

a = first

b = second

all = $($a\_$b)

这里的 $a\_$b 组成了“first\_second”，于是， $(all) 的值就是“Hello”。

## 追加变量值

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）

使用 += 操作符，可以模拟为下面的这种例子：

objects = main.o foo.o bar.o utils.o

objects := $(objects) another.o

所不同的是，用 += 更为简洁。

如果变量之前没有定义过，那么， += 会自动变成 = ，如果前面有变量定义，那么 += 会继承于前次操作的赋值符。如果前一次的是 := ，那么 += 会以 := 作为其赋值符

## 多行变量

define指示符后面跟的是变量的名字，而**重起一行定义变量的值**，定义是以**endef** 关键字结束。其工作方式和“=”操作符一样。变量的值可以包含函数、命令、文字，或是其它变量。因为命令需要以[Tab]键开头，所以如果你用define定义的命令变量中没有以 Tab 键开头，那么make 就不会把其认为是命令。

下面的这个示例展示了define的用法:

define two-lines

echo foo

echo $(bar)

endef

## 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。如果Makefile中定义了CFLAGS，那么则会使用Makefile中的这个变量，如果没有定义则使用系统环境变量的值，一个共性和个性的统一，很像“全局变量”和“局部变量”的特性。

当make嵌套调用时（参见前面的“嵌套调用”章节），上层Makefile中定义的变量会以系统环境变量的方式传递到下层的Makefile 中。当然，默认情况下，只有通过命令行设置的变量会被传递。而定义在文件中的变量，如果要向下层Makefile传递，则需要使用export关键字来声明。（参见前面章节）

当然，我并不推荐把许多的变量都定义在系统环境中，这样，在我们执行不用的Makefile时，拥有的是同一套系统变量，这可能会带来更多的麻烦。

## 目标变量

前面我们所讲的在Makefile中定义的变量都是“全局变量”，在整个文件，我们都可以访问这些变量。当然，“自动化变量”除外，如 $< 等这种类量的自动化变量就属于“规则型变量”，这种变量的值依赖于规则的目标和依赖目标的定义。

当然，我也同样可以为某个目标设置局部变量，这种变量被称为“Target-specific Variable”，它可以和“全局变量”同名，因为它的作用范围只在这条规则以及连带规则中，所以其值也只在作用范围内有效。而不会影响规则链以外的全局变量的值。如：

prog : CFLAGS = -g

prog : prog.o foo.o bar.o

$(CC) $(CFLAGS) prog.o foo.o bar.o

prog.o : prog.c

$(CC) $(CFLAGS) prog.c

foo.o : foo.c

$(CC) $(CFLAGS) foo.c

bar.o : bar.c

$(CC) $(CFLAGS) bar.c

在这个示例中，不管全局的 $(CFLAGS) 的值是什么，在prog目标，以及其所引发的所有规则中（prog.o foo.o bar.o的规则）， $(CFLAGS) 的值都是 -g。

make的“模式”一般是至少含有一个 % 的，所以，我们可以以如下方式给所有以 .o 结尾的目标定义目标变量：

%.o : CFLAGS = -O

# 使用条件判断

ifeq 的意思表示条件语句的开始，并指定一个条件表达式，表达式包含两个参数，以逗号分隔。endif 表示一个条件语句的结束，任何一个条件表达式都应该以 endif 结束。

foo: $(objects)

ifeq ($(CC),gcc)

$(CC) -o foo $(objects) $(libs\_for\_gcc)

else

$(CC) -o foo $(objects) $(normal\_libs)

Endif

ifdef <variable-name>

如果变量 <variable-name> 的值非空，那到表达式为真。否则，表达式为假。当然， <variable-name> 同样可以是一个函数的返回值。注意， **ifdef 只是测试一个变量是否有值**，其并不会把变量扩展到当前位置。

# 使用函数

## 函数的调用语法

函数调用，很像变量的使用，也是以 **$** 来标识的，其语法如下：

$(<function> <arguments>)

这里， <function> 就是函数名，make支持的函数不多。 <arguments> 为函数的参数，参数间以**逗号 , 分隔**，而**函数名和参数之间以“空格”分隔**。函数调用以 **$** 开头，以**圆括号或花括号**把函数名和参数括起。使用 $(subst a,b,$(x)) 这样的形式。

## 字符串处理函数

### subst

$(subst <from>,<to>,<text>)

名称：字符串替换函数

功能：把字串 <text> 中的 <from> 字符串替换成 <to> 。

返回：函数返回被替换过后的字符串。

示例：

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

把 feet on the street 中的 ee 替换成 EE ，返回结果是 fEEt on the strEEt 。

### patsubst

$(patsubst <pattern>,<replacement>,<text>)

名称：模式字符串替换函数。

功能：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> ，如果匹配的话，则以 <replacement> 替换。这里，<pattern> 可以包括**通配符%** ，表示任意长度的字串。如果 <replacement> 中也包含% ，那么， <replacement> 中的这个% 将是 <pattern> 中的那个% 所代表的字串。（可以用 \ 来转义，以 \% 来表示真实含义的 % 字符）

返回：函数返回被替换过后的字符串。

示例：

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

把字串 x.c.c bar.c 符合模式 %.c 的单词替换成 %.o ，返回结果是 x.c.o bar.o

注：

objects = foo.o bar.o baz.o，

那么， $(objects:.o=.c) 和 $(patsubst %.o,%.c,$(objects)) 是一样的。

### strip

$(strip <string>)

名称：去空格函数。

功能：去掉 <string> 字串中**开头和结尾**的**空字符**。

返回：返回被去掉空格的字符串值。

示例：

$(strip a b c )

把字串“a b c ”去掉开头和结尾的空格，结果是“a b c”。

### findstring

$(findstring <find>,<in>)

名称：查找字符串函数

功能：在字串 <in> 中查找 <find> 字串。

返回：如果找到，那么返回 <find> ，否则返回空字符串。

示例：

$(findstring a,a b c)

$(findstring a,b c)

第一个函数返回 a 字符串，第二个返回空字符串

### filter

$(filter <pattern...>,<text>)

名称：过滤函数

功能：以 <pattern> 模式过滤 <text> 字符串中的单词，保留符合模式 <pattern> 的单词。可以有多个模式。

返回：返回符合模式 <pattern> 的字串。

示例：

sources := foo.c bar.c baz.s ugh.h

foo: $(sources)

cc $(filter %.c %.s,$(sources)) -o foo

$(filter %.c %.s,$(sources)) 返回的值是 foo.c bar.c baz.s 。

### filter-out

$(filter-out <pattern...>,<text>)

名称：反过滤函数

功能：以 <pattern> 模式过滤 <text> 字符串中的单词，去除符合模式 <pattern> 的单词。可以有多个模式。

返回：返回**不符合模式** <pattern> 的字串。

示例：

objects=main1.o foo.o main2.o bar.o

mains=main1.o main2.o

$(filter-out $(mains),$(objects))

返回值是 foo.o bar.o 。

### sort

$(sort <list>)

名称：排序函数

功能：给字符串 <list> 中的**单词排序（升序）**。

返回：返回排序后的字符串。

示例：

$(sort foo bar lose)

返回 bar foo lose 。

备注： sort 函数会去掉 <list> 中相同的单词。

### word

$(word <n>,<text>)

名称：取单词函数

功能：取字符串 <text> 中第 <n> 个单词/字符串。（从一开始）

返回：返回字符串 <text> 中第 <n> 个单词/字符串。如果 <n> 比 <text> 中的单词数要大，那么返回空字符串。

示例：

$(word 2, foo bar baz)

返回值是 bar 。

### wordlist

$(wordlist <ss>,<e>,<text>)

名称：取单词串函数

功能：从字符串 <text> 中取从 <ss> 开始到 <e> 的单词串。 <ss> 和 <e> 是一个**数字**。

返回：返回字符串 <text> 中从 <ss> 到 <e> 的单词字串。如果 <ss> 比 <text> 中的单词数要大，那么返回空字符串。如果 <e> 大于 <text> 的单词数，那么返回从 <ss> 开始，到 <text> 结束的单词串。

示例：

$(wordlist 2, 3, foo bar baz)

返回值是 bar baz 。

### words

$(words <text>)

名称：单词个数统计函数

功能：统计 <text> 中字符串中的单词个数。

返回：返回 <text> 中的单词数。

示例：

$(words, foo bar baz)

返回值是 3 。

备注：如果我们要取 <text> 中最后的一个单词，我们可以这样：$(word $(words <text>),<text>) 。

### firstword

$(firstword <text>)

名称：首单词函数——firstword。

功能：取字符串 <text> 中的第一个单词。

返回：返回字符串 <text> 的第一个单词。

示例：

$(firstword foo bar)

返回值是 foo。

备注：这个函数可以用 word 函数来实现： $(word 1,<text>) 。

## 文件名操作函数

下面我们要介绍的函数主要是处理文件名的。每个函数的参数字符串都会被当做一个或是一系列的文件名来对待。

### dir

$(dir <names...>)

名称：取目录函数——dir。

功能：从文件名序列 <names> 中**取出目录部分**。目录部分是指最后一个反斜杠（ / ）之前的部分。如果没有反斜杠，那么返回 ./ 。

返回：返回文件名序列 <names> 的目录部分。

示例：

$(dir src/foo.c hacks)

返回值是 src/ ./ 。

### notdir

$(notdir <names...>)

名称：取文件函数——notdir。

功能：从文件名序列 <names> 中取出**非目录部分**。非目录部分是指最後一个反斜杠（ / ）之后的部分。

返回：返回文件名序列 <names> 的非目录部分。

示例:

$(notdir src/foo.c hacks)

返回值是 foo.c hacks 。

### suffix

$(suffix <names...>)

名称：取后缀函数——suffix。

功能：从文件名序列 <names> 中取出各个文件名的后缀。

返回：返回文件名序列 <names> 的后缀序列，**如果文件没有后缀，则返回空字串**。

示例：

$(suffix src/foo.c src-1.0/bar.c hacks)

返回值是 .c .c。

### basename

$(basename <names...>)

名称：取前缀函数——basename。

功能：从文件名序列 <names> 中取出各个文件名的前缀部分。

返回：返回文件名序列 <names> 的前缀序列，如果文件没有前缀，则返回空字串。

示例：

$(basename src/foo.c src-1.0/bar.c hacks)

返回值是 src/foo src-1.0/bar hacks 。

### addsuffix

$(addsuffix <suffix>,<names...>)

名称：加后缀函数——addsuffix。

功能：把后缀 <suffix> 加到 <names> 中的每个单词后面。

返回：返回加过后缀的文件名序列。

示例：

$(addsuffix .c,foo bar)

返回值是 foo.c bar.c 。

### addprefix

$(addprefix <prefix>,<names...>)

名称：加前缀函数——addprefix。

功能：把前缀 <prefix> 加到 <names> 中的每个单词前面。

返回：返回加过前缀的文件名序列。

示例：

$(addprefix src/,foo bar)

返回值是 src/foo src/bar 。

### join

$(join <list1>,<list2>)

名称：连接函数——join。

功能：把 <list2> 中的单词对应地加到 <list1> 的单词后面。如果 <list1> 的单词个数要比 <list2> 的多，那么， <list1> 中的多出来的单词将保持原样。如果 <list2> 的单词个数要比 <list1> 多，那么， <list2> 多出来的单词将被复制到 <list1> 中。

返回：返回连接过后的字符串。

示例：

$(join aaa bbb , 111 222 333)

返回值是 aaa111 bbb222 333 。

## foreach 函数

foreach函数和别的函数非常的不一样。因为这个函数是用来做循环用的，Makefile中的foreach函数几乎是仿照于Unix标准Shell（/bin/sh）中的for语句，或是C-Shell（/bin/csh）中的foreach语句而构建的。它的语法是：

$(foreach <var>,<list>,<text>)

这个函数的意思是，把参数 <list> 中的单词逐一取出放到参数 <var> 所指定的变量中，然后再执行 <text> 所包含的表达式。每一次 <text> 会返回一个字符串，循环过程中， <text> 的所返回的每个字符串会以空格分隔，最后当整个循环结束时， <text> 所返回的每个字符串所组成的整个字符串（以空格分隔）将会是foreach函数的返回值。

所以， <var> 最好是一个变量名， <list> 可以是一个表达式，而 <text> 中一般会使用 <var> 这个参数来依次枚举 <list> 中的单词。举个例子：

names := a b c d

files := $(foreach n,$(names),$(n).o)

上面的例子中， $(name) 中的单词会被挨个取出，并存到变量 n 中， $(n).o 每次根据 $(n) 计算出一个值，这些值以空格分隔，最后作为foreach函数的返回，所以， $(files) 的值是 a.o b.o c.o d.o 。

注意，foreach中的 <var> 参数是一个临时的局部变量，foreach函数执行完后，参数 <var> 的变量将不在作用，其作用域只在foreach函数当中。

## if 函数

if函数很像GNU的make所支持的条件语句——ifeq（参见前面所述的章节），if函数的语法是：

$(if <condition>,<then-part>)

或是

$(if <condition>,<then-part>,<else-part>)

可见，if函数可以包含“else”部分，或是不含。即if函数的参数可以是两个，也可以是三个。 <condition> 参数是if的表达式，如果其返回的为非空字符串，那么这个表达式就相当于返回真，于是， <then-part> 会被计算，否则 <else-part> 会被计算。

而if函数的返回值是，如果 <condition> 为真（非空字符串），那个 <then-part> 会是整个函数的返回值，如果 <condition> 为假（空字符串），那么 <else-part> 会是整个函数的返回值，此时如果 <else-part> 没有被定义，那么，整个函数返回空字串。

所以， <then-part> 和 <else-part> 只会有一个被计算。

# Make运行

## 指定目标

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。

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。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）。

# 隐含规则

## 关于命令的变量

AR : 函数库打包程序。默认命令是 ar

AS : 汇编语言编译程序。默认命令是 as

CC : C语言编译程序。默认命令是 cc

CXX : C++语言编译程序。默认命令是 g++

CO : 从 RCS文件中扩展文件程序。默认命令是 co

CPP : C程序的预处理器（输出是标准输出设备）。默认命令是 $(CC) –E

FC : Fortran 和 Ratfor 的编译器和预处理程序。默认命令是 f77

GET : 从SCCS文件中扩展文件的程序。默认命令是 get

LEX : Lex方法分析器程序（针对于C或Ratfor）。默认命令是 lex

PC : Pascal语言编译程序。默认命令是 pc

YACC : Yacc文法分析器（针对于C程序）。默认命令是 yacc

YACCR : Yacc文法分析器（针对于Ratfor程序）。默认命令是 yacc –r

MAKEINFO : 转换Texinfo源文件（.texi）到Info文件程序。默认命令是 makeinfo

TEX : 从TeX源文件创建TeX DVI文件的程序。默认命令是 tex

TEXI2DVI : 从Texinfo源文件创建军TeX DVI 文件的程序。默认命令是 texi2dvi

WEAVE : 转换Web到TeX的程序。默认命令是 weave

CWEAVE : 转换C Web 到 TeX的程序。默认命令是 cweave

TANGLE : 转换Web到Pascal语言的程序。默认命令是 tangle

CTANGLE : 转换C Web 到 C。默认命令是 ctangle

RM : 删除文件命令。默认命令是 rm –f

## 自动化变量

如何书写一个命令来完成从不同的依赖文件生成相应的目标？因为在每一次的对模式规则的解析时，都会是不同的目标和依赖文件。

**自动化变量**就是完成这个功能的。所谓自动化变量，就是这种变量会把模式中所定义的一系列的文件自动地挨个取出，直至所有的符合模式的文件都取完了。这种**自动化变量只应出现在规则的命令**中。

下面是所有的自动化变量及其说明：

**$@ :** 表示规则中的**目标文件集**。在模式规则中，如果有多个目标，那么， $@ 就是匹配于目标中模式定义的集合。

**$% :** 仅当目标是**函数库文件**中，表示规则中的**目标成员名**。例如，如果一个目标是 foo.a(bar.o) ，那么， $% 就是 bar.o ， $@ 就是 foo.a 。如果目标不是函数库文件（Unix下是 .a ，Windows下是 .lib ），那么，其值为空。

**$< : 第一个依赖文件**。如果依赖文件是以模式（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。

**$? :** 所有**比目标文件新的依赖文件**的集合。以空格分隔。

**$^ :** **所有的依赖文件**的集合。以空格分隔。如果在依赖文件中有多个重复的，那么这个变量会**去除重复的依赖目标**，只保留一份。

**$+ :** 这个变量很像 $^ ，也是所有依赖文件的集合。只是它**不去除重复**的依赖目标。

**$\* :** 这个变量表示目标模式中 % 及其之前的部分。如果目标是 dir/a.foo.b ，并且目标的模式是 a.%.b ，那么， $\* 的值就是 dir/foo 。这个变量对于构造有关联的文件名是比较有效。如果目标中没有模式的定义，那么 $\* 也就不能被推导出，但是，如果目标文件的后缀是make所识别的，那么 $\* 就是除了后缀的那一部分。例如：如果目标是 foo.c ，因为 .c 是make所能识别的后缀名，所以， $\* 的值就是 foo 。这个特性是GNU make的，很有可能不兼容于其它版本的make，所以，你应该尽量避免使用 $\* ，除非是在隐含规则或是静态模式中。如果目标中的后缀是make所不能识别的，那么 $\* 就是空值。

## 模式的匹配

一般来说，一个目标的模式有一个有前缀或是后缀的 % ，或是没有前后缀，直接就是一个 % 。因为 % 代表一个或多个字符，所以在定义好了的模式中，我们把 % 所匹配的内容叫做“茎”，例如 %.c 所匹配的文件“test.c”中“test”就是“茎”。因为在目标和依赖目标中同时有 % 时，依赖目标的“茎”会传给目标，当做目标中的“茎”。

当一个模式匹配包含有斜杠（实际也不经常包含）的文件时，那么在进行模式匹配时，目录部分会首先被移开，然后进行匹配，成功后，再把目录加回去。在进行“茎”的传递时，我们需要知道这个步骤。例如有一个模式 e%t ，文件 src/eat 匹配于该模式，于是 src/a 就是其“茎”，如果这个模式定义在依赖目标中，而被依赖于这个模式的目标中又有个模式 c%r ，那么，目标就是 src/car 。（“茎”被传递）

-c

编译产生对象文件(\*.obj)而不链接成可执行文件,当编译几个独立的模块,而待以后由链接程序把它们链接在一起时,就可以使用这个选项

-o

允许用户指定输出文件名

-g

指明编译程序在编译的输出中应产生调试信息. 这个调试信息使源代码和变量名引用在调试程序中或者当程序异常退出后在分析core文件时可被使用.

单行注释：#

多行注释：<!--

-->

Mkdir -p //创建文件夹

Cp //copy

批处理.bat

Rem //注释