# **[Linux下编写 makefile 详细教程](https://www.cnblogs.com/mfryf/p/3305778.html)**

近期在学习Linux下的C编程，买了一本叫《Linux环境下的C编程指南》读到makefile就越看越迷糊，可能是我的理解能不行。

于是google到了以下这篇文章。通俗易懂。然后把它贴出来，方便学习。

后记，看完发现这篇文章和《Linux环境下的C编程指南》的makefile一章所讲述的惊人的相似，仅仅是这篇文章从一个实例切入，在有些地方比較好理解。能让人看懂就是好文章。

跟我一起写 Makefile

陈皓 (CSDN)

概述  
——

什么是makefile？或许非常多Winodws的程序猿都不知道这个东西，由于那些Windows的IDE都为你做了这个工作，但我觉得要作一个好的和professional的程序猿，makefile还是要懂。这就好像如今有这么多的HTML的编辑器，但假设你想成为一个专业人士，你还是要了解HTML的标识的含义。特别在Unix下的软件编译，你就不能不自己写makefile了，会不会写makefile，从一个側面说明了一个人是否具备完成大型工程的能力。

由于，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数，其按类型、功能、模块分别放在若干个文件夹中，makefile定义了一系列的规则来指定，哪些文件须要先编译，哪些文件须要后编译，哪些文件须要又一次编译，甚至于进行更复杂的功能操作，由于makefile就像一个Shell脚本一样，当中也能够运行操作系统的命令。

makefile带来的长处就是——“自己主动化编译”，一旦写好，仅仅须要一个make命令，整个工程全然自己主动编译，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，比方：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。

如今讲述怎样写makefile的文章比較少，这是我想写这篇文章的原因。当然，不同产商的make各不相同，也有不同的语法，但其本质都是在“文件依赖性”上做文章，这里，我仅对GNU的make进行讲述，我的环境是RedHat Linux 8.0，make的版本号是3.80。必竟，这个make是应用最为广泛的，也是用得最多的。而且其还是最遵循于IEEE 1003.2-1992 标准的（POSIX.2）。

在这篇文档中，将以C/C++的源代码作为我们基础，所以必定涉及一些关于C/C++的编译的知识，相关于这方面的内容，还请各位查看相关的编译器的文档。这里所默认的编译器是UNIX下的GCC和CC。

关于程序的编译和链接  
——————————

在此，我想多说关于程序编译的一些规范和方法，一般来说，无论是C、C++、还是pas，首先要把源文件编译成中间代码文件，在Windows下也就是 .obj 文件，UNIX下是 .o 文件，即 Object File，这个动作叫做编译（compile）。然后再把大量的Object File合成运行文件，这个动作叫作链接（link）。

编译时，编译器须要的是语法的正确，函数与变量的声明的正确。对于后者，通常是你须要告诉编译器头文件的所在位置（头文件里应该仅仅是声明，而定义应该放在C/C++文件里），仅仅要全部的语法正确，编译器就能够编译出中间目标文件。一般来说，每一个源文件都应该对应于一个中间目标文件（O文件或是OBJ文件）。

链接时，主要是链接函数和全局变量，所以，我们能够使用这些中间目标文件（O文件或是OBJ文件）来链接我们的应用程序。链接器并无论函数所在的源文件，仅仅管函数的中间目标文件（Object File），在大多数时候，由于源文件太多，编译生成的中间目标文件太多，而在链接时须要明显地指出中间目标文件名称，这对于编译非常不方便，所以，我们要给中间目标文件打个包，在Windows下这样的包叫“库文件”（Library File)，也就是 .lib 文件，在UNIX下，是Archive File，也就是 .a 文件。

总结一下，源文件首先会生成中间目标文件，再由中间目标文件生成运行文件。在编译时，编译器仅仅检测程序语法，和函数、变量是否被声明。假设函数未被声明，编译器会给出一个警告，但能够生成Object File。而在链接程序时，链接器会在全部的Object File中找寻函数的实现，假设找不到，那到就会报链接错误码（Linker Error），在VC下，这样的错误通常是：Link 2001错误，意思说是说，链接器未能找到函数的实现。你须要指定函数的Object File.

好，言归正传，GNU的make有很多的内容，闲言少叙，还是让我们开始吧。

Makefile 介绍  
———————

make命令运行时，须要一个 Makefile 文件，以告诉make命令须要怎么样的去编译和链接程序。

首先，我们用一个演示例子来说明Makefile的书写规则。以便给大家一个感兴认识。这个演示例子来源于GNU的make使用手冊，在这个演示例子中，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命令怎样编译和链接这几个文件。我们的规则是：  
1）假设这个工程沒有编译过，那么我们的全部C文件都要编译并被链接。  
2）假设这个工程的某几个C文件被改动，那么我们仅仅编译被改动的C文件，并链接目标程序。  
3）假设这个工程的头文件被改变了，那么我们须要编译引用了这几个头文件的C文件，并链接目标程序。

仅仅要我们的Makefile写得够好，全部的这一切，我们仅仅用一个make命令就能够完成，make命令会自己主动智能地依据当前的文件改动的情况来确定哪些文件须要重编译，从而自己编译所须要的文件和链接目标程序。

一、Makefile的规则

在讲述这个Makefile之前，还是让我们先来粗略地看一看Makefile的规则。

target ... : prerequisites ...  
command  
...  
...

target也就是一个目标文件，能够是Object File，也能够是运行文件。还能够是一个标签（Label），对于标签这样的特性，在后续的“伪目标”章节中会有叙述。

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

command也就是make须要运行的命令。（随意的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中假设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运行。这就是Makefile的规则。也就是Makefile中最核心的内容。

说究竟，Makefile的东西就是这样一点，好像我的这篇文档也该结束了。呵呵。还不尽然，这是Makefile的主线和核心，但要写好一个Makefile还不够，我会以后面一点一点地结合我的工作经验给你慢慢到来。内容还多着呢。：）

二、一个演示例子

正如前面所说的，假设一个工程有3个头文件，和8个C文件，我们为了完成前面所述的那三个规则，我们的Makefile应该是以下的这个样子的。

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。这样比較便于Makefile的易读。我们能够把这个内容保存在文件为“Makefile”或“makefile”的文件里，然后在该文件夹下直接输入命令“make”就能够生成运行文件edit。假设要删除运行文件和全部的中间目标文件，那么，仅仅要简单地运行一下“make clean”就能够了。

在这个makefile中，目标文件（target）包括：运行文件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 文件又是运行文件 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。记住，make并无论命令是怎么工作的，他仅仅管运行所定义的命令。make会比較targets文件和prerequisites文件的改动日期，假设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就会运行后续定义的命令。

这里要说明一点的是，clean不是一个文件，它仅仅只是是一个动作名字，有点像C语言中的lable一样，其冒号后什么也沒有，那么，make就不会自己主动去找文件的依赖性，也就不会自己主动运行其后所定义的命令。要运行其后的命令，就要在make命令后明显得指出这个lable的名字。这样的方法非常实用，我们能够在一个makefile中定义不用的编译或是和编译无关的命令，比方程序的打包，程序的备份，等等。

三、make是怎样工作的

在默认的方式下，也就是我们仅仅输入make命令。那么，

1、make会在当前文件夹下找名字叫“Makefile”或“makefile”的文件。  
2、假设找到，它会找文件里的第一个目标文件（target），在上面的样例中，他会找到“edit”这个文件，并把这个文件作为终于的目标文件。  
3、假设edit文件不存在，或是edit所依赖的后面的 .o 文件的文件改动时间要比edit这个文件新，那么，他就会运行后面所定义的命令来生成edit这个文件。  
4、假设edit所依赖的.o文件也不存在，那么make会在当前文件里找目标为.o文件的依赖性，假设找到则再依据那一个规则生成.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，也就是运行文件edit了。

这就是整个make的依赖性，make会一层又一层地去找文件的依赖关系，直到终于编译出第一个目标文件。在找寻的过程中，假设出现错误，比方最后被依赖的文件找不到，那么make就会直接退出，并报错，而对于所定义的命令的错误，或是编译不成功，make根本不理。make仅仅管文件的依赖性，即，假设在我找了依赖关系之后，冒号后面的文件还是不在，那么对不起，我就不工作啦。

通过上述分析，我们知道，像clean这样的，沒有被第一个目标文件直接或间接关联，那么它后面所定义的命令将不会被自己主动运行，只是，我们能够显示要make运行。即命令——“make clean”，以此来清除全部的目标文件，以便重编译。

于是在我们编程中，假设这个工程已被编译过了，当我们改动了当中一个源文件，比方file.c，那么依据我们的依赖性，我们的目标file.o会被重编译（也就是在这个依性关系后面所定义的命令），于是file.o的文件也是最新的啦，于是file.o的文件改动时间要比edit要新，所以edit也会被又一次链接了（详见edit目标文件后定义的命令）。

而假设我们改变了“command.h”，那么，kdb.o、command.o和files.o都会被重编译，而且，edit会被重链接。

四、makefile中使用变量

在上面的样例中，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

我们能够看到[.o]文件的字符串被反复了两次，假设我们的工程须要添加一个新的[.o]文件，那么我们须要在两个地方加（应该是三个地方，另一个地方在clean中）。当然，我们的makefile并不复杂，所以在两个地方加也不累，但假设makefile变得复杂，那么我们就有可能会忘掉一个须要添加的地方，而导致编译失败。所以，为了makefile的易维护，在makefile中我们能够使用变量。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。

比方，我们声明一个变量，叫objects, OBJECTS, objs, OBJS, obj, 或是 OBJ，反正无论什么啦，仅仅要能够表示obj文件就可以了。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，我们就能够非常方便地在我们的makefile中以“$(objects)”的方式来使用这个变量了，于是我们的改良版makefile就变成以下这个样子：

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 : 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 $(objects)

于是假设有新的 .o 文件添加，我们仅仅需简单地改动一下 objects 变量就能够了。

关于变量许多其他的话题，我会在后续给你一一道来。

五、让make自己主动推导

GNU的make非常强大，它能够自己主动推导文件以及文件依赖关系后面的命令，于是我们就不是必需去在每一个[.o]文件后都写上相似的命令，由于，我们的make会自己主动识别，并自己推导命令。

仅仅要make看到一个[.o]文件，它就会自己主动的把[.c]文件加在依赖关系中，假设make找到一个whatever.o，那么whatever.c，就会是whatever.o的依赖文件。而且 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。我们的是新的makefile又出炉了。

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 : defs.h  
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)

这个方法，也就是make的“隐晦规则”。上面文件内容中，“.PHONY”表示，clean是个伪目标文件。

关于更为具体的“隐晦规则”和“伪目标文件”，我会在后续给你一一道来。

六、另类风格的makefile

即然我们的make能够自己主动推导命令，那么我看到那堆[.o]和[.h]的依赖就有点不爽，那么多的反复的[.h]，能不能把其收拢起来，好吧，沒有问题，这个对于make来说非常easy，谁叫它提供了自己主动推导命令和文件的功能呢？来看看最新风格的makefile吧。

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)

这样的风格，让我们的makefile变得非常easy，但我们的文件依赖关系就显得有点凌乱了。鱼和熊掌不可兼得。还看你的喜好了。我是不喜欢这样的风格的，一是文件的依赖关系看不清晰，二是假设文件一多，要添加几个新的.o文件，那就理不清晰了。

七、清空目标文件的规则

每一个Makefile中都应该写一个清空目标文件（.o和运行文件）的规则，这不仅便于重编译，也非常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，或许某些文件出现故障，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”。

上面就是一个makefile的概貌，也是makefile的基础，以下还有非常多makefile的相关细节，准备好了吗？准备好了就来。

Makefile 总述  
———————

一、Makefile里有什么？

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。有关这一部分的内容，我会在后续的部分中讲述。

5、凝视。Makefile中唯独行凝视，和UNIX的Shell脚本一样，其凝视是用“#”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。

二、Makefile的文件名称

默认的情况下，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，找到了解释这个文件。在这三个文件名称中，最好使用“Makefile”这个文件名称，由于，这个文件名称第一个字符为大写，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仅仅对全小写的“makefile”文件名称敏感，可是基本上来说，大多数的make都支持“makefile”和“Makefile”这两种默认文件名称。

当然，你能够使用别的文件名称来书写Makefile，比方：“Make.Linux”，“Make.Solaris”，“Make.AIX”等，假设要指定特定的Makefile，你能够使用make的“-f”和“--file”參数，如：make -f Make.Linux或make --file Make.AIX。

三、引用其他的Makefile

在Makefile使用includekeyword能够把别的Makefile包括进来，这非常像C语言的#include，被包括的文件会原模原样的放在当前文件的包括位置。include的语法是：

include <filename>

filename能够是当前操作系统Shell的文件模式（能够保含路径和通配符）

在include前面能够有一些空字符，可是绝不能是[Tab]键开始。include和<filename>能够用一个或多个空格隔开。举个样例，你有这样几个Makefile：a.mk、b.mk、c.mk，另一个文件叫foo.make，以及一个变量$(bar)，其包括了e.mk和f.mk，那么，以下的语句：

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

等效于：

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

make命令开始时，会把找寻include所指出的其他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。假设文件都沒有指定绝对路径或是相对路径的话，make会在当前文件夹下首先寻找，假设当前文件夹下沒有找到，那么，make还会在以下的几个文件夹下找：

1、假设make运行时，有“-I”或“--include-dir”參数，那么make就会在这个參数所指定的文件夹下去寻找。  
2、假设文件夹<prefix>/include（通常是：/usr/local/bin或/usr/include）存在的话，make也会去找。

假设有文件沒有找到的话，make会生成一条警告信息，但不会立即出现致命错误。它会继续加载其他的文件，一旦完成makefile的读取，make会再重试这些沒有找到，或是不能读取的文件，假设还是不行，make才会出现一条致命信息。假设你想让make不理那些无法读取的文件，而继续运行，你能够在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，都不要报错继续运行。和其他版本号make兼容的相关命令是sinclude，其作用和这一个是一样的。

四、环境变量 MAKEFILES

假设你的当前环境中定义了环境变量MAKEFILES，那么，make会把这个变量中的值做一个相似于include的动作。这个变量中的值是其他的Makefile，用空格分隔。仅仅是，它和include不同的是，从这个环境变中引入的Makefile的“目标”不会起作用，假设环境变量中定义的文件发现错误，make也会不理。

可是在这里我还是建议不要使用这个环境变量，由于仅仅要这个变量一被定义，那么当你使用make时，全部的Makefile都会受到它的影响，这绝不是你想看到的。在这里提这个事，仅仅是为了告诉大家，或许有时候你的Makefile出现了怪事，那么你能够看看当前环境中有未定义这个变量。

五、make的工作方式

GNU的make工作时的运行步骤入下：（想来其他的make也是相似）

1、读入全部的Makefile。  
2、读入被include的其他Makefile。  
3、初始化文件里的变量。  
4、推导隐晦规则，并分析全部规则。  
5、为全部的目标文件创建依赖关系链。  
6、依据依赖关系，决定哪些目标要又一次生成。  
7、运行生成命令。

1-5步为第一个阶段，6-7为第二个阶段。第一个阶段中，假设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并不会全然立即展开，make使用的是迟延战术，假设变量出如今依赖关系的规则中，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。

当然，这个工作方式你不一定要清晰，可是知道这个方式你也会对make更为熟悉。有了这个基础，后续部分也就easy看懂了。

书写规则  
————

规则包括两个部分，一个是依赖关系，一个是生成目标的方法。

在Makefile中，规则的顺序是非常重要的，由于，Makefile中仅仅应该有一个终于目标，其他的目标都是被这个目标所连带出来的，所以一定要让make知道你的终于目标是什么。一般来说，定义在Makefile中的目标可能会有非常多，可是第一条规则中的目标将被确立为终于的目标。假设第一条规则中的目标有非常多个，那么，第一个目标会成为终于的目标。make所完成的也就是这个目标。

好了，还是让我们来看一看怎样书写规则。

一、规则举例

foo.o : foo.c defs.h # foo模块  
cc -c -g foo.c

看到这个样例，各位应该不是非常陌生了，前面也已说过，foo.o是我们的目标，foo.c和defs.h是目标所依赖的源文件，而唯独一个命令“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，假设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、假设生成（或更新）foo.o文件。也就是那个cc命令，其说明了，怎样生成foo.o这个文件。（当然foo.c文件include了defs.h文件）

二、规则的语法

targets : prerequisites  
command  
...

或是这样：

targets : prerequisites ; command  
command  
...

targets是文件名称，以空格分开，能够使用通配符。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，假设其不与“target吐舌rerequisites”在一行，那么，必须以[Tab键]开头，假设和prerequisites在一行，那么能够用分号做为分隔。（见上）

prerequisites也就是目标所依赖的文件（或依赖目标）。假设当中的某个文件要比目标文件要新，那么，目标就被觉得是“过时的”，被觉得是须要重生成的。这个在前面已经讲过了。

假设命令太长，你能够使用反斜框（‘/’）作为换行符。make对一行上有多少个字符沒有限制。规则告诉make两件事，文件的依赖关系和怎样成成目标文件。

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

三、在规则中使用通配符

假设我们想定义一系列比較相似的文件，我们非常自然地就想起使用通配符。make支持三各通配符：“\*”，“?”和“[...]”。这是和Unix的B-Shell是相同的。

波浪号（“~”）字符在文件名称中也有比較特殊的用途。假设是“~/test”，这就表示当前用户的$HOME文件夹下的test文件夹。而“~hchen/test”则表示用户hchen的宿主文件夹下的test文件夹。（这些都是Unix下的小知识了，make也支持）而在Windows或是MS-DOS下，用户沒有宿主文件夹，那么波浪号所指的文件夹则依据环境变量“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。一个须要我们注意的是，假设我们的文件名称中有通配符，如：“\*”，那么能够用转义字符“/”，如“/\*”来表示真实的“\*”字符，而不是随意长度的字符串。

好吧，还是先来看几个样例吧：

clean:  
rm -f \*.o

上面这个样例我不不多说了，这是操作系统Shell所支持的通配符。这是在命令中的通配符。

print: \*.c  
lpr -p $?  
touch print

上面这个样例说明了通配符也能够在我们的规则中，目标print依赖于全部的[.c]文件。当中的“$?”是一个自己主动化变量，我会在后面给你讲述。

objects = \*.o

上面这个样例，表示了，通符相同能够用在变量中。并非说[\*.o]会展开，不！objects的值就是“\*.o”。Makefile中的变量事实上就是C/C++中的宏。假设你要让通配符在变量中展开，也就是让objects的值是全部[.o]的文件名称的集合，那么，你能够这样：

objects := $(wildcard \*.o)

这样的使用方法由keyword“wildcard”指出，关于Makefile的keyword，我们将在后面讨论。

四、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很多的源文件分类，并存放在不同的文件夹中。所以，当make须要去找寻文件的依赖关系时，你能够在文件前加上路径，但最好的方法是把一个路径告诉make，让make在自己主动去找。

Makefile文件里的特殊变量“VPATH”就是完成这个功能的，假设沒有指明这个变量，make仅仅会在当前的文件夹中去找寻依赖文件和目标文件。假设定义了这个变量，那么，make就会在当当前文件夹找不到的情况下，到所指定的文件夹中去找寻文件了。

VPATH = src:../headers

上面的的定义指定两个文件夹，“src”和“../headers”，make会依照这个顺序进行搜索。文件夹由“冒号”分隔。（当然，当前文件夹永远是最高优先搜索的地方）

另一个设置文件搜索路径的方法是使用make的“vpath”keyword（注意，它是全小写的），这不是变量，这是一个make的keyword，这和上面提到的那个VPATH变量非常相似，可是它更为灵活。它能够指定不同的文件在不同的搜索文件夹中。这是一个非常灵活的功能。它的使用方法有三种：

1、vpath <pattern> <directories>

为符合模式<pattern>的文件指定搜索文件夹<directories>。

2、vpath <pattern>

清除符合模式<pattern>的文件的搜索文件夹。

3、vpath

清除全部已被设置好了的文件搜索文件夹。

vapth使用方法中的<pattern>须要包括“%”字符。“%”的意思是匹配零或若干字符，比如，“%.h”表示全部以“.h”结尾的文件。<pattern>指定了要搜索的文件集，而<directories>则指定了<pattern>的文件集的搜索的文件夹。比如：

vpath %.h ../headers

该语句表示，要求make在“../headers”文件夹下搜索全部以“.h”结尾的文件。（假设某文件在当前文件夹沒有找到的话）

我们能够连续地使用vpath语句，以指定不同搜索策略。假设连续的vpath语句中出现了相同的<pattern>，或是被反复了的<pattern>，那么，make会依照vpath语句的先后顺序来运行搜索。如：

vpath %.c foo  
vpath % blish  
vpath %.c bar

其表示“.c”结尾的文件，先在“foo”文件夹，然后是“blish”，最后是“bar”文件夹。

vpath %.c foo:bar  
vpath % blish

而上面的语句则表示“.c”结尾的文件，先在“foo”文件夹，然后是“bar”文件夹，最后才是“blish”文件夹。

五、伪目标

最早先的一个样例中，我们提到过一个“clean”的目标，这是一个“伪目标”，

clean:  
rm \*.o temp

正像我们前面样例中的“clean”一样，即然我们生成了很多文件编译文件，我们也应该提供一个清除它们的“目标”以备完整地重编译而用。 （以“make clean”来使用该目标）

由于，我们并不生成“clean”这个文件。“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它是否要运行。我们唯独通过显示地指明这个“目标”才干让其生效。当然，“伪目标”的取名不能和文件名称重名，不然其就失去了“伪目标”的意义了。

当然，为了避免和文件重名的这样的情况，我们能够使用一个特殊的标记“.PHONY”来显示地指明一个目标是“伪目标”，向make说明，无论是否有这个文件，这个目标就是“伪目标”。

.PHONY : clean

仅仅要有这个声明，无论是否有“clean”文件，要运行“clean”这个目标，唯独“make clean”这样。于是整个过程能够这样写：

.PHONY: clean  
clean:  
rm \*.o temp

伪目标一般沒有依赖的文件。可是，我们也能够为伪目标指定所依赖的文件。伪目标相同能够作为“默认目标”，仅仅要将其放在第一个。一个演示例子就是，假设你的Makefile须要一口气生成若干个可运行文件，但你仅仅想简单地敲一个make完事，而且，全部的目标文件都写在一个Makefile中，那么你能够使用“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，其依赖于其他三个目标。由于伪目标的特性是，总是被运行的，所以其依赖的那三个目标就总是不如“all”这个目标新。所以，其他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”。

随便提一句，从上面的样例我们能够看出，目标也能够成为依赖。所以，伪目标相同也可成为依赖。看以下的样例：

.PHONY: cleanall cleanobj cleandiff

cleanall : cleanobj cleandiff  
rm program

cleanobj :  
rm \*.o

cleandiff :  
rm \*.diff

“make clean”将清除全部要被清除的文件。“cleanobj”和“cleandiff”这两个伪目标有点像“子程序”的意思。我们能够输入“make cleanall”和“make cleanobj”和“make cleandiff”命令来达到清除不同种类文件的目的。

六、多目标

Makefile的规则中的目标能够不止一个，其支持多目标，有可能我们的多个目标同一时候依赖于一个文件，而且其生成的命令大体相似。于是我们就能把其合并起来。当然，多个目标的生成规则的运行命令是同一个，这可能会可我们带来麻烦，只是好在我们的能够使用一个自己主动化变量“$@”（关于自己主动化变量，将在后面讲述），这个变量表示着眼下规则中全部的目标的集合，这样说可能非常抽象，还是看一个样例吧。

bigoutput littleoutput : text.g  
generate text.g -$(subst output,,$@) > $@

上述规则等效于：

bigoutput : text.g  
generate text.g -big > bigoutput  
littleoutput : text.g  
generate text.g -little > littleoutput

当中，-$(subst output,,$@)中的“$”表示运行一个Makefile的函数，函数名为subst，后面的为參数。关于函数，将在后面讲述。这里的这个函数是截取字符串的意思，“$@”表示目标的集合，就像一个数组，“$@”依次取出目标，并执于命令。

七、静态模式

静态模式能够更加easy地定义多目标的规则，能够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，能够有通配符。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。

这样描写叙述这三个东西，可能还是沒有说清晰，还是举个样例来说明一下吧。假设我们的<target-parrtern>定义成“%.o”，意思是我们的<target>集合中都是以“.o”结尾的，而假设我们的<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，假设你的文件名称中有“%”那么你能够使用反斜杠“/”进行转义，来标明真实的“%”字符。

看一个样例：

objects = foo.o bar.o

all: $(objects)

$(objects): %.o: %.c  
$(CC) -c $(CFLAGS) $< -o $@

上面的样例中，指明了我们的目标从$object中获取，“%.o”表明要全部以“.o”结尾的目标，也就是“foo.o bar.o”，也就是变量$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“$<”和“$@”则是自己主动化变量，“$<”表示全部的依赖目标集（也就是“foo.c bar.c”），“$@”表示目标集（也就是“foo.o bar.o”）。于是，上面的规则展开后等效于以下的规则：

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，假设我们的“%.o”有几百个，那种我们仅仅要用这样的非常easy的“静态模式规则”就能够写完一堆规则，实在是太有效率了。“静态模式规则”的使用方法非常灵活，假设用得好，那会一个非常强大的功能。再看一个样例：

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，仅仅要当中模式为“%.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。

八、自己主动生成依赖性

在Makefile中，我们的依赖关系可能会须要包括一系列的头文件，比方，假设我们的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

可是，假设是一个比較大型的工程，你必需清晰哪些C文件包括了哪些头文件，而且，你在添加或删除头文件时，也须要小心地改动Makefile，这是一个非常沒有维护性的工作。为了避免这样的繁重而又easy出错的事情，我们能够使用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，即自己主动找寻源文件里包括的头文件，并生成一个依赖关系。比如，假设我们运行以下的命令：

cc -M main.c

其输出是：

main.o : main.c defs.h

于是由编译器自己主动生成的依赖关系，这样一来，你就不必再手动书写若干文件的依赖关系，而由编译器自己主动生成了。须要提醒一句的是，假设你使用GNU的C/C++编译器，你得用“-MM”參数，不然，“-M”參数会把一些标准库的头文件也包括进来。

gcc -M main.c的输出是：

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的输出则是：

main.o: main.c defs.h

那么，编译器的这个功能怎样与我们的Makefile联系在一起呢。由于这样一来，我们的Makefile也要依据这些源文件又一次生成，让Makefile自已依赖于源文件？这个功能并不现实，只是我们能够有其他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己主动生成的依赖关系放到一个文件里，为每一个“name.c”的文件都生成一个“name.d”的Makefile文件，[.d]文件里就存放对应[.c]文件的依赖关系。

于是，我们能够写出[.c]文件和[.d]文件的依赖关系，并让make自己主动更新或自成[.d]文件，并把其包括在我们的主Makefile中，这样，我们就能够自己主动化地生成每一个文件的依赖关系了。

这里，我们给出了一个模式规则来产生[.d]文件：

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

这个规则的意思是，全部的[.d]文件依赖于[.c]文件，“rm -f $@”的意思是删除全部的目标，也就是[.d]文件，第二行的意思是，为每一个依赖文件“$<”，也就是[.c]文件生成依赖文件，“$@”表示模式“%.d”文件，假设有一个C文件是name.c，那么“%”就是“name”，“$$$$”意为一个随机编号，第二行生成的文件有可能是“name.d.12345”，第三行使用sed命令做了一个替换，关于sed命令的使用方法请參看相关的使用文档。第四行就是删除暂时文件。

总而言之，这个模式要做的事就是在编译器生成的依赖关系中添加[.d]文件的依赖，即把依赖关系：

main.o : main.c defs.h

转成：

main.o main.d : main.c defs.h

于是，我们的[.d]文件也会自己主动更新了，并会自己主动生成了，当然，你还能够在这个[.d]文件里添加的不仅仅是依赖关系，包括生成的命令也可一并添加，让每一个[.d]文件都包括一个完赖的规则。一旦我们完成这个工作，接下来，我们就要把这些自己主动生成的规则放进我们的主Makefile中。我们能够使用Makefile的“include”命令，来引入别的Makefile文件（前面讲过），比如：

sources = foo.c bar.c

include $(sources:.c=.d)

上述语句中的“$(sources:.c=.d)”中的“.c=.d”的意思是做一个替换，把变量$(sources)全部[.c]的字串都替换成[.d]，关于这个“替换”的内容，在后面我会有更为具体的讲述。当然，你得注意次序，由于include是按次来加载文件，最先加载的[.d]文件里的目标会成为默认目标。

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运行命令，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，可是假设该空格或空行是以Tab键开头的，那么make会觉得其是一个空命令。

我们在UNIX下可能会使用不同的Shell，可是make的命令默认是被“/bin/sh”——UNIX的标准Shell解释运行的。除非你特别指定一个其他的Shell。Makefile中，“#”是凝视符，非常像C/C++中的“//”，其后的本行字符都被凝视。

一、显示命令

通常，make会把其要运行的命令行在命令运行前输出到屏幕上。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，最具代表性的样例是，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运行时，会输出“正在编译XXX模块......”字串，但不会输出命令，假设沒有“@”，那么，make将输出：

echo 正在编译XXX模块......  
正在编译XXX模块......

假设make运行时，带入make參数“-n”或“--just-print”，那么其仅仅是显示命令，但不会运行命令，这个功能非常有利于我们调试我们的Makefile，看看我们书写的命令是运行起来是什么样子的或是什么顺序的。

而make參数“-s”或“--slient”则是全面禁止命令的显示。

二、命令运行

当依赖目标新于目标时，也就是当规则的目标须要被更新时，make会一条一条的运行其后的命令。须要注意的是，假设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。比方你的第一条命令是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：

演示例子一：  
exec:  
cd /home/hchen  
pwd

演示例子二：  
exec:  
cd /home/hchen; pwd

当我们运行“make exec”时，第一个样例中的cd沒有作用，pwd会打印出当前的Makefile文件夹，而第二个样例中，cd就起作用了，pwd会打印出“/home/hchen”。

make通常是使用环境变量SHELL中所定义的系统Shell来运行命令，默认情况下使用UNIX的标准Shell——/bin/sh来运行命令。但在MS-DOS下有点特殊，由于MS-DOS下沒有SHELL环境变量，当然你也能够指定。假设你指定了UNIX风格的文件夹形式，首先，make会在SHELL所指定的路径中找寻命令解释器，假设找不到，其会在当前盘符中的当前文件夹中寻找，假设再找不到，其会在PATH环境变量中所定义的全部路径中寻找。MS-DOS中，假设你定义的命令解释器沒有找到，其会给你的命令解释器加上诸如“.exe”、“.com”、“.bat”、“.sh”等后缀。

三、命令出错

每当命令运行完后，make会检测每一个命令的返回码，假设命令返回成功，那么make会运行下一条命令，当规则中全部的命令成功返回后，这个规则就算是成功完成了。假设一个规则中的某个命令出错了（命令退出码非零），那么make就会终止运行当前规则，这将有可能终止全部规则的运行。

有些时候，命令的出错并不表示就是错误的。比如mkdir命令，我们一定须要建立一个文件夹，假设文件夹不存在，那么mkdir就成功运行，万事大吉，假设文件夹存在，那么就出错了。我们之所以使用mkdir的意思就是一定要有这样的一个文件夹，于是我们就不希望mkdir出错而终止规则的运行。

为了做到这一点，忽略命令的出错，我们能够在Makefile的命令行前加一个减号“-”（在Tab键之后），标记为无论命令出不出错都觉得是成功的。如：

clean:  
-rm -f \*.o

另一个全局的办法是，给make加上“-i”或是“--ignore-errors”參数，那么，Makefile中全部命令都会忽略错误。而假设一个规则是以“.IGNORE”作为目标的，那么这个规则中的全部命令将会忽略错误。这些是不同级别的防止命令出错的方法，你能够依据你的不同喜欢设置。

另一个要提一下的make的參数的是“-k”或是“--keep-going”，这个參数的意思是，假设某规则中的命令出错了，那么就终目该规则的运行，但继续运行其他规则。

四、嵌套运行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文件夹中，我们能够在每一个文件夹中都书写一个该文件夹的Makefile，这有利于让我们的Makefile变得更加地简洁，而不至于把全部的东西全部写在一个Makefile中，这样会非常难维护我们的Makefile，这个技术对于我们模块编译和分段编译有着非常大的长处。

比如，我们有一个子文件夹叫subdir，这个文件夹下有个Makefile文件，来指明了这个文件夹下文件的编译规则。那么我们总控的Makefile能够这样书写：

subsystem:  
cd subdir && $(MAKE)

其等效于：

subsystem:  
$(MAKE) -C subdir

定义$(MAKE)宏变量的意思是，或许我们的make须要一些參数，所以定义成一个变量比較利于维护。这两个样例的意思都是先进入“subdir”文件夹，然后运行make命令。

我们把这个Makefile叫做“总控Makefile”，总控Makefile的变量能够传递到下级的Makefile中（假设你显示的声明），可是不会覆盖下层的Makefile中所定义的变量，除非指定了“-e”參数。

假设你要传递变量到下级Makefile中，那么你能够使用这样的声明：

export <variable ...>

假设你不想让某些变量传递到下级Makefile中，那么你能够这样声明：

unexport <variable ...>

如：

演示例子一：

export variable = value

其等效于：

variable = value  
export variable

其等效于：

export variable := value

其等效于：

variable := value  
export variable

演示例子二：

export variable += value

其等效于：

variable += value  
export variable

假设你要传递全部的变量，那么，仅仅要一个export就可以了。后面什么也不用跟，表示传递全部的变量。

须要注意的是，有两个变量，一个是SHELL，一个是MAKEFLAGS，这两个变量无论你是否export，其总是要传递到下层Makefile中，特别是MAKEFILES变量，当中包括了make的參数信息，假设我们运行“总控Makefile”时有make參数或是在上层Makefile中定义了这个变量，那么MAKEFILES变量将会是这些參数，并会传递到下层Makefile中，这是一个系统级的环境变量。

可是make命令中的有几个參数并不往下传递，它们是“-C”,“-f”,“-h”“-o”和“-W”（有关Makefile參数的细节将在后面说明），假设你不想往下层传递參数，那么，你能够这样来：

subsystem:  
cd subdir && $(MAKE) MAKEFLAGS=

假设你定义了环境变量MAKEFLAGS，那么你得确信当中的选项是大家都会用到的，假设当中有“-t”,“-n”,和“-q”參数，那么将会有让你意想不到的结果，或许会让你异常地恐慌。

另一个在“嵌套运行”中比較实用的參数，“-w”或是“--print-directory”会在make的过程中输出一些信息，让你看到眼下的工作文件夹。比方，假设我们的下级make文件夹是“/home/hchen/gnu/make”，假设我们使用“make -w”来运行，那么当进入该文件夹时，我们会看到：

make: Entering directory `/home/hchen/gnu/make'.

而在完成下层make后离开文件夹时，我们会看到：

make: Leaving directory `/home/hchen/gnu/make'

当你使用“-C”參数来指定make下层Makefile时，“-w”会被自己主动打开的。假设參数中有“-s”（“--slient”）或是“--no-print-directory”，那么，“-w”总是失效的。

五、定义命令包

假设Makefile中出现一些相同命令序列，那么我们能够为这些相同的命令序列定义一个变量。定义这样的命令序列的语法以“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，由于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演示例子中来看看吧。

foo.c : foo.y  
$(run-yacc)

我们能够看见，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”（有关这样的以“$”开头的特殊变量，我们会在后面介绍），make在运行命令包时，命令包中的每一个命令会被依次独立运行。

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中运行的时候其会自己主动原模原样地展开在所使用的地方。其与C/C++所不同的是，你能够在Makefile中改变其值。在Makefile中，变量能够使用在“目标”，“依赖目标”，“命令”或是Makefile的其他部分中。

变量的命名字能够包括字符、数字，下划线（能够是数字开头），但不应该含有“:”、“#”、“=”或是空字符（空格、回车等）。变量是大写和小写敏感的，“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，但我推荐使用大写和小写搭配的变量名，如：MakeFlags。这样能够避免和系统的变量冲突，而发生意外的事情。

有一些变量是非常奇怪字串，如“$<”、“$@”等，这些是自己主动化变量，我会在后面介绍。

一、变量的基础

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

变量能够使用在很多地方，如规则中的“目标”、“依赖”、“命令”以及新的变量中。先看一个样例：

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，比如：

foo = c  
prog.o : prog.$(foo)  
$(foo)$(foo) -$(foo) prog.$(foo)

展开后得到：

prog.o : prog.c  
cc -c prog.c

当然，千万不要在你的Makefile中这样干，这里仅仅是举个样例来表明Makefile中的变量在使用处展开的真实样子。可见其就是一个“替代”的原理。

另外，给变量加上括号全然是为了更加安全地使用这个变量，在上面的样例中，假设你不想给变量加上括号，那也能够，但我还是强烈建议你给变量加上括号。

二、变量中的变量

在定义变量的值时，我们能够使用其他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“=”左側是变量，右側是变量的值，右側变量的值能够定义在文件的不论什么一处，也就是说，右側中的变量不一定非要是已定义好的值，其也能够使用后面定义的值。如：

foo = $(bar)  
bar = $(ugh)  
ugh = Huh?

all:  
echo $(foo)

我们运行“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，变量是能够使用后面的变量来定义的。

这个功能有好的地方，也有不好的地方，好的地方是，我们能够把变量的真实值推到后面来定义，如：

CFLAGS = $(include\_dirs) -O  
include\_dirs = -Ifoo -Ibar

当“CFLAGS”在命令中被展开时，会是“-Ifoo -Ibar -O”。但这样的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。还有就是假设在变量中使用函数，那么，这样的方式会让我们的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。由于你不会知道这两个函数会被调用多少次。

为了避免上面的这个方法，我们能够使用make中的另一种用变量来定义变量的方法。这个方法使用的是“:=”操作符，如：

x := foo  
y := $(x) bar  
x := later

其等效于：

y := foo bar  
x := later

值得一提的是，这个方法，前面的变量不能使用后面的变量，仅仅能使用前面已定义好了的变量。假设是这样：

y := $(x) bar  
x := foo

那么，y的值是“bar”，而不是“foo bar”。

上面都是一些比較简单的变量使用了，让我们来看一个复杂的样例，当中包括了make的函数、条件表达式和一个系统变量“MAKELEVEL”的使用：

ifeq (0,${MAKELEVEL})  
cur-dir := $(shell pwd)  
whoami := $(shell whoami)  
host-type := $(shell arch)  
MAKE := ${MAKE} host-type=${host-type} whoami=${whoami}  
endif

关于条件表达式和函数，我们在后面再说，对于系统变量“MAKELEVEL”，其意思是，假设我们的make有一个嵌套运行的动作（參见前面的“嵌套使用make”），那么，这个变量会记录了我们的当前Makefile的调用层数。

以下再介绍两个定义变量时我们须要知道的，请先看一个样例，假设我们要定义一个变量，其值是一个空格，那么我们能够这样来：

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

nullstring是一个Empty变量，当中什么也沒有，而我们的space的值是一个空格。由于在操作符的右边是非常难描写叙述一个空格的，这里採用的技术非常管用，先用一个Empty变量来标明变量的值开始了，而后面採用“#”凝视符来表示变量定义的终止，这样，我们能够定义出其值是一个空格的变量。请注意这里关于“#”的使用，凝视符“#”的这样的特性值得我们注意，假设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，假设我们这样使用这样变量来指定别的文件夹——“$(dir)/file”那么就完蛋了。

另一个比較实用的操作符是“?=”，先看演示例子：

FOO ?= bar

其含义是，假设FOO沒有被定义过，那么变量FOO的值就是“bar”，假设FOO先前被定义过，那么这条语将什么也不做，其等效于：

ifeq ($(origin FOO), undefined)  
FOO = bar  
endif

三、变量高级使用方法

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。

我们能够替换变量中的共同拥有的部分，其格式是“$(var:a=b)”或是“${var:a=b}”，其意思是，把变量“var”中全部以“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”。

还是看一个演示例子吧：

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

这个演示例子中，我们先定义了一个“$(foo)”变量，而第二行的意思是把“$(foo)”中全部以“.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”。

第二种变量替换的技术是以“静态模式”（參见前面章节）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，模式中必须包括一个“%”字符，这个样例相同让$(bar)变量的值为“a.c b.c c.c”。

第二种高级使用方法是——“把变量的值再当成变量”。先看一个样例：

x = y  
y = z  
a := $($(x))

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

我们还能够使用许多其他的层次：

x = y  
y = z  
z = u  
a := $($($(x)))

这里的$(a)的值是“u”，相关的推导留给读者自己去做吧。

让我们再复杂一点，使用上“在变量定义中使用变量”的第一个方式，来看一个样例：

x = $(y)  
y = z  
z = Hello  
a := $($(x))

这里的$($(x))被替换成了$($(y))，由于$(y)值是“z”，所以，终于结果是：a:=$(z)，也就是“Hello”。

再复杂一点，我们再加上函数：

x = variable1  
variable2 := Hello  
y = $(subst 1,2,$(x))  
z = y  
a := $($($(z)))

这个样例中，“$($($(z)))”扩展为“$($(y))”，而其再次被扩展为“$($(subst 1,2,$(x)))”。$(x)的值是“variable1”，subst函数把“variable1”中的全部“1”字串替换成“2”字串，于是，“variable1”变成“variable2”，再取其值，所以，终于，$(a)的值就是$(variable2)的值——“Hello”。（喔，好不easy）

在这样的方式中，或要能够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”。

再来看看结合第一种技术的样例：

a\_objects := a.o b.o c.o  
1\_objects := 1.o 2.o 3.o

sources := $($(a1)\_objects:.o=.c)

这个样例中，假设$(a1)的值是“a”的话，那么，$(sources)的值就是“a.c b.c c.c”；假设$(a1)的值是“1”，那么$(sources)的值是“1.c 2.c 3.c”。

再来看一个这样的技术和“函数”与“条件语句”一同使用的样例：

ifdef do\_sort  
func := sort  
else  
func := strip  
endif

bar := a d b g q c

foo := $($(func) $(bar))

这个演示例子中，假设定义了“do\_sort”，那么：foo := $(sort a d b g q c)，于是$(foo)的值就是“a b c d g q”，而假设未定义“do\_sort”，那么：foo := $(sort a d b g q c)，调用的就是strip函数。

当然，“把变量的值再当成变量”这样的技术，相同能够用在操作符的左边：

dir = foo  
$(dir)\_sources := $(wildcard $(dir)/\*.c)  
define $(dir)\_print  
lpr $($(dir)\_sources)  
endef

这个样例中定义了三个变量：“dir”，“foo\_sources”和“foo\_print”。

四、追加变量值

我们能够使用“+=”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，能够模拟为以下的这样的样例：

objects = main.o foo.o bar.o utils.o  
objects := $(objects) another.o

所不同的是，用“+=”更为简洁。

假设变量之前未定义过，那么，“+=”会自己主动变成“=”，假设前面有变量定义，那么“+=”会继承于前次操作的赋值符。假设前一次的是“:=”，那么“+=”会以“:=”作为其赋值符，如：

variable := value  
variable += more

等效于：

variable := value  
variable := $(variable) more

但假设是这样的情况：

variable = value  
variable += more

由于前次的赋值符是“=”，所以“+=”也会以“=”来做为赋值，那么岂不会发生变量的递补归定义，这是非常不好的，所以make会自己主动为我们解决问题，我们不必操心这个问题。

五、override 指示符

假设有变量是通常make的命令行參数设置的，那么Makefile中对这个变量的赋值会被忽略。假设你想在Makefile中设置这类參数的值，那么，你能够使用“override”指示符。其语法是：

override <variable> = <value>

override <variable> := <value>

当然，你还能够追加：

override <variable> += <more text>

对于多行的变量定义，我们用define指示符，在define指示符前，也相同能够使用ovveride指示符，如：

override define foo  
bar  
endef

六、多行变量

另一种设置变量值的方法是使用definekeyword。使用definekeyword设置变量的值能够有换行，这有利于定义一系列的命令（前面我们讲过“命令包”的技术就是利用这个keyword）。

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endefkeyword结束。其工作方式和“=”操作符一样。变量的值能够包括函数、命令、文字，或是其他变量。由于命令须要以[Tab]键开头，所以假设你用define定义的命令变量中沒有以[Tab]键开头，那么make就不会把其觉得是命令。

以下的这个演示例子展示了define的使用方法：

define two-lines  
echo foo  
echo $(bar)  
endef

七、环境变量

make运行时的系统环境变量能够在make开始运行时被加载到Makefile文件里，可是假设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。（假设make指定了“-e”參数，那么，系统环境变量将覆盖Makefile中定义的变量）

因此，假设我们在环境变量中设置了“CFLAGS”环境变量，那么我们就能够在全部的Makefile中使用这个变量了。这对于我们使用统一的编译參数有比較大的长处。假设Makefile中定义了CFLAGS，那么则会使用Makefile中的这个变量，假设未定义则使用系统环境变量的值，一个共性和个性的统一，非常像“全局变量”和“局部变量”的特性。

当make嵌套调用时（參见前面的“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，唯独通过命令行设置的变量会被传递。而定义在文件里的变量，假设要向下层Makefile传递，则须要使用exprotkeyword来声明。（參见前面章节）

当然，我并不推荐把很多的变量都定义在系统环境中，这样，在我们运行不用的Makefile时，拥有的是同一套系统变量，这可能会带来许多其他的麻烦。

八、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，我们都能够訪问这些变量。当然，“自己主动化变量”除外，如“$<”等这样的类量的自己主动化变量就属于“规则型变量”，这样的变量的值依赖于规则的目标和依赖目标的定义。

当然，我样相同能够为某个目标设置局部变量，这样的变量被称为“Target-specific Variable”，它能够和“全局变量”同名，由于它的作用范围仅仅在这条规则以及连带规则中，所以其值也仅仅在作用范围内有效。而不会影响规则链以外的全局变量的值。

其语法是：

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

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

<variable-assignment>能够是前面讲过的各种赋值表达式，如“=”、“:=”、“+=”或是“？=”。第二个语法是针对于make命令行带入的变量，或是系统环境变量。

这个特性非常的实用，当我们设置了这样一个变量，这个变量会作用到由这个目标所引发的全部的规则中去。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

在这个演示例子中，无论全局的$(CFLAGS)的值是什么，在prog目标，以及其所引发的全部规则中（prog.o foo.o bar.o的规则），$(CFLAGS)的值都是“-g”

九、模式变量

在GNU的make中，还支持模式变量（Pattern-specific Variable），通过上面的目标变量中，我们知道，变量能够定义在某个目标上。模式变量的长处就是，我们能够给定一种“模式”，能够把变量定义在符合这样的模式的全部目标上。

我们知道，make的“模式”通常是至少含有一个“%”的，所以，我们能够以例如以下方式给全部以[.o]结尾的目标定义目标变量：

%.o : CFLAGS = -O

相同，模式变量的语法和“目标变量”一样：

<pattern ...> : <variable-assignment>

<pattern ...> : override <variable-assignment>

override相同是针对于系统环境传入的变量，或是make命令行指定的变量。

使用条件推断  
——————

使用条件推断，能够让make依据运行时的不同情况选择不同的运行分支。条件表达式能够是比較变量的值，或是比較变量和常量的值。

一、演示例子

以下的样例，推断$(CC)变量是否“gcc”，假设是的话，则使用GNU函数编译目标。

libs\_for\_gcc = -lgnu  
normal\_libs =

foo: $(objects)  
ifeq ($(CC),gcc)  
$(CC) -o foo $(objects) $(libs\_for\_gcc)  
else  
$(CC) -o foo $(objects) $(normal\_libs)  
endif

可见，在上面演示例子的这个规则中，目标“foo”能够依据变量“$(CC)”值来选取不同的函数库来编译程序。

我们能够从上面的演示例子中看到三个keyword：ifeq、else和endif。ifeq的意思表示条件语句的开始，并指定一个条件表达式，表达式包括两个參数，以逗号分隔，表达式以圆括号括起。else表示条件表达式为假的情况。endif表示一个条件语句的结束，不论什么一个条件表达式都应该以endif结束。

当我们的变量$(CC)值是“gcc”时，目标foo的规则是：

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

而当我们的变量$(CC)值不是“gcc”时（比方“cc”），目标foo的规则是：

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

当然，我们还能够把上面的那个样例写得更简洁一些：

libs\_for\_gcc = -lgnu  
normal\_libs =

ifeq ($(CC),gcc)  
libs=$(libs\_for\_gcc)  
else  
libs=$(normal\_libs)  
endif

foo: $(objects)  
$(CC) -o foo $(objects) $(libs)

二、语法

条件表达式的语法为：

<conditional-directive>  
<text-if-true>  
endif

以及：

<conditional-directive>  
<text-if-true>  
else  
<text-if-false>  
endif

当中<conditional-directive>表示条件keyword，如“ifeq”。这个keyword有四个。

第一个是我们前面所见过的“ifeq”

ifeq (<arg1>, <arg2> )   
ifeq '<arg1>' '<arg2>'   
ifeq "<arg1>" "<arg2>"   
ifeq "<arg1>" '<arg2>'   
ifeq '<arg1>' "<arg2>"

比較參数“arg1”和“arg2”的值是否相同。当然，參数中我们还能够使用make的函数。如：

ifeq ($(strip $(foo)),)  
<text-if-empty>  
endif

这个演示例子中使用了“strip”函数，假设这个函数的返回值是空（Empty），那么<text-if-empty>就生效。

第二个条件keyword是“ifneq”。语法是：

ifneq (<arg1>, <arg2> )   
ifneq '<arg1>' '<arg2>'   
ifneq "<arg1>" "<arg2>"   
ifneq "<arg1>" '<arg2>'   
ifneq '<arg1>' "<arg2>"

其比較參数“arg1”和“arg2”的值是否相同，假设不同，则为真。和“ifeq”相似。

第三个条件keyword是“ifdef”。语法是：

ifdef <variable-name>

假设变量<variable-name>的值非空，那到表达式为真。否则，表达式为假。当然，<variable-name>相同能够是一个函数的返回值。注意，ifdef仅仅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。还是来看两个样例：

演示例子一：  
bar =  
foo = $(bar)  
ifdef foo  
frobozz = yes  
else  
frobozz = no  
endif

演示例子二：  
foo =  
ifdef foo  
frobozz = yes  
else  
frobozz = no  
endif

第一个样例中，“$(frobozz)”值是“yes”，第二个则是“no”。

第四个条件keyword是“ifndef”。其语法是：

ifndef <variable-name>

这个我就不多说了，和“ifdef”是相反的意思。

在<conditional-directive>这一行上，多余的空格是被同意的，可是不能以[Tab]键做为开始（不然就被觉得是命令）。而凝视符“#”相同也是安全的。“else”和“endif”也一样，仅仅要不是以[Tab]键开始就可以了。

特别注意的是，make是在读取Makefile时就计算条件表达式的值，并依据条件表达式的值来选择语句，所以，你最好不要把自己主动化变量（如“$@”等）放入条件表达式中，由于自己主动化变量是在运行时才有的。

而且，为了避免混乱，make不同意把整个条件语句分成两部分放在不同的文件里。

使用函数  
————

在Makefile中能够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非常多，只是已经足够我们的操作了。函数调用后，函数的返回值能够当做变量来使用。

一、函数的调用语法

函数调用，非常像变量的使用，也是以“$”来标识的，其语法例如以下：

$(<function> <arguments> )

或是

${<function> <arguments>}

这里，<function>就是函数名，make支持的函数不多。<arguments>是函数的參数，參数间以逗号“,”分隔，而函数名和參数之间以“空格”分隔。函数调用以“$”开头，以圆括号或花括号把函数名和參数括起。感觉非常像一个变量，是不是？函数中的參数能够使用变量，为了风格的统一，函数和变量的括号最好一样，如使用“$(subst a,b,$(x))”这样的形式，而不是“$(subst a,b,${x})”的形式。由于统一会更清晰，也会降低一些不必要的麻烦。

还是来看一个演示例子：

comma:= ,  
empty:=  
space:= $(empty) $(empty)  
foo:= a b c  
bar:= $(subst $(space),$(comma),$(foo))

在这个演示例子中，$(comma)的值是一个逗号。$(space)使用了$(empty)定义了一个空格，$(foo)的值是“a b c”，$(bar)的定义用，调用了函数“subst”，这是一个替换函数，这个函数有三个參数，第一个參数是被替换字串，第二个參数是替换字串，第三个參数是替换操作作用的字串。这个函数也就是把$(foo)中的空格替换成逗号，所以$(bar)的值是“a,b,c”。

二、字符串处理函数

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

名称：字符串替换函数——subst。  
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。

演示例子：

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

名称：模式字符串替换函数——patsubst。  
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>，假设匹配的话，则以<replacement>替换。这里，<pattern>能够包括通配符“%”，表示随意长度的字串。假设<replacement>中也包括“%”，那么，<replacement>中的这个“%”将是<pattern>中的那个“%”所代表的字串。（能够用“/”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。

演示例子：

$(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”

备注：

这和我们前面“变量章节”说过的相关知识有点相似。如：

“$(var:<pattern>=<replacement> )”  
相当于  
“$(patsubst <pattern>,<replacement>,$(var))”，

而“$(var: <suffix>=<replacement> )”  
则相当于  
“$(patsubst %<suffix>,%<replacement>,$(var))”。

比如有：objects = foo.o bar.o baz.o，  
那么，“$(objects:.o=.c)”和“$(patsubst %.o,%.c,$(objects))”是一样的。

$(strip <string> )

名称：去空格函数——strip。  
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。  
演示例子：

$(strip a b c )

把字串“a b c ”去到开头和结尾的空格，结果是“a b c”。

$(findstring <find>,<in> )

名称：查找字符串函数——findstring。  
功能：在字串<in>中查找<find>字串。  
返回：假设找到，那么返回<find>，否则返回空字符串。  
演示例子：

$(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“”字符串（空字符串）

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

名称：过滤函数——filter。  
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。能够有多个模式。  
返回：返回符合模式<pattern>的字串。  
演示例子：

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 <pattern...>,<text> )

名称：反过滤函数——filter-out。  
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。能够有多个模式。  
返回：返回不符合模式<pattern>的字串。  
演示例子：

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 <list> )

名称：排序函数——sort。  
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。  
演示例子：$(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 <n>,<text> )

名称：取单词函数——word。  
功能：取字符串<text>中第<n>个单词。（从一开始）  
返回：返回字符串<text>中第<n>个单词。假设<n>比<text>中的单词数要大，那么返回空字符串。  
演示例子：$(word 2, foo bar baz)返回值是“bar”。

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

名称：取单词串函数——wordlist。  
功能：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。  
返回：返回字符串<text>中从<s>到<e>的单词字串。假设<s>比<text>中的单词数要大，那么返回空字符串。假设<e>大于<text>的单词数，那么返回从<s>开始，到<text>结束的单词串。  
演示例子： $(wordlist 2, 3, foo bar baz)返回值是“bar baz”。

$(words <text> )

名称：单词个数统计函数——words。  
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。  
演示例子：$(words, foo bar baz)返回值是“3”。  
备注：假设我们要取<text>中最后的一个单词，我们能够这样：$(word $(words <text> ),<text> )。

$(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。  
演示例子：$(firstword foo bar)返回值是“foo”。  
备注：这个函数能够用word函数来实现：$(word 1,<text> )。

以上，是全部的字符串操作函数，假设搭配混合使用，能够完成比較复杂的功能。这里，举一个现实中应用的样例。我们知道，make使用“VPATH”变量来指定“依赖文件”的搜索路径。于是，我们能够利用这个搜索路径来指定编译器对头文件的搜索路径參数CFLAGS，如：

override CFLAGS += $(patsubst %,-I%,$(subst :, ,$(VPATH)))

假设我们的“$(VPATH)”值是“src:../headers”，那么“$(patsubst %,-I%,$(subst :, ,$(VPATH)))”将返回“-Isrc -I../headers”，这正是cc或gcc搜索头文件路径的參数。

三、文件名称操作函数

以下我们要介绍的函数主要是处理文件名称的。每一个函数的參数字符串都会被当做一个或是一系列的文件名称来对待。

$(dir <names...> )

名称：取文件夹函数——dir。  
功能：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“./”。  
返回：返回文件名称序列<names>的文件夹部分。  
演示例子： $(dir src/foo.c hacks)返回值是“src/ ./”。

$(notdir <names...> )

名称：取文件函数——notdir。  
功能：从文件名称序列<names>中取出非文件夹部分。非文件夹部分是指最后一个反斜杠（“/”）之后的部分。  
返回：返回文件名称序列<names>的非文件夹部分。  
演示例子： $(notdir src/foo.c hacks)返回值是“foo.c hacks”。

$(suffix <names...> )

名称：取后缀函数——suffix。  
功能：从文件名称序列<names>中取出各个文件名称的后缀。  
返回：返回文件名称序列<names>的后缀序列，假设文件沒有后缀，则返回空字串。  
演示例子：$(suffix src/foo.c src-1.0/bar.c hacks)返回值是“.c .c”。

$(basename <names...> )

名称：取前缀函数——basename。  
功能：从文件名称序列<names>中取出各个文件名称的前缀部分。  
返回：返回文件名称序列<names>的前缀序列，假设文件沒有前缀，则返回空字串。  
演示例子：$(basename src/foo.c src-1.0/bar.c hacks)返回值是“src/foo src-1.0/bar hacks”。

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

名称：加后缀函数——addsuffix。  
功能：把后缀<suffix>加到<names>中的每一个单词后面。  
返回：返回加过后缀的文件名称序列。  
演示例子：$(addsuffix .c,foo bar)返回值是“foo.c bar.c”。

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

名称：加前缀函数——addprefix。  
功能：把前缀<prefix>加到<names>中的每一个单词后面。  
返回：返回加过前缀的文件名称序列。  
演示例子：$(addprefix src/,foo bar)返回值是“src/foo src/bar”。

$(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。假设<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。假设<list2>的单词个数要比<list1>多，那么，<list2>多出来的单词将被拷贝到<list2>中。  
返回：返回连接过后的字符串。  
演示例子：$(join aaa bbb , 111 222 333)返回值是“aaa111 bbb222 333”。

四、foreach 函数

foreach函数和别的函数非常的不一样。由于这个函数是用来做循环用的，Makefile中的foreach函数差点儿是仿照于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。每一次<text>会返回一个字符串，循环过程中，<text>的所返回的每一个字符串会以空格分隔，最后当整个循环结束时，<text>所返回的每一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>能够是一个表达式，而<text>中通常会使用<var>这个參数来依次枚举<list>中的单词。举个样例：

names := a b c d

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

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

注意，foreach中的<var>參数是一个暂时的局部变量，foreach函数运行完后，參数<var>的变量将不在作用，其作用域仅仅在foreach函数当中。

五、if 函数

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

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

或是

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

可见，if函数能够包括“else”部分，或是不含。即if函数的參数能够是两个，也能够是三个。<condition>參数是if的表达式，假设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，假设<condition>为真（非空字符串），那个<then-part>会是整个函数的返回值，假设<condition>为假（空字符串），那么<else-part>会是整个函数的返回值，此时假设<else-part>沒有被定义，那么，整个函数返回空字串。

所以，<then-part>和<else-part>仅仅会有一个被计算。

六、call函数

call函数是唯一一个能够用来创建新的參数化的函数。你能够写一个非常复杂的表达式，这个表达式中，你能够定义很多參数，然后你能够用call函数来向这个表达式传递參数。其语法是：

$(call <expression>,<parm1>,<parm2>,<parm3>...)

当make运行这个函数时，<expression>參数中的变量，如$(1)，$(2)，$(3)等，会被參数<parm1>，<parm2>，<parm3>依次代替。而<expression>的返回值就是call函数的返回值。比如：

reverse = $(1) $(2)

foo = $(call reverse,a,b)

那么，foo的值就是“a b”。当然，參数的次序是能够自己定义的，不一定是顺序的，如：

reverse = $(2) $(1)

foo = $(call reverse,a,b)

此时的foo的值就是“b a”。

七、origin函数  
origin函数不像其他的函数，他并不操作变量的值，他仅仅是告诉你你的这个变量是哪里来的？其语法是：

$(origin <variable> )

注意，<variable>是变量的名字，不应该是引用。所以你最好不要在<variable>中使用“$”字符。Origin函数会以其返回值来告诉你这个变量的“出生情况”，以下，是origin函数的返回值:

“undefined”

假设<variable>从来未定义过，origin函数返回这个值“undefined”。

“default”

假设<variable>是一个默认的定义，比方“CC”这个变量，这样的变量我们将在后面讲述。

“environment”

假设<variable>是一个环境变量，而且当Makefile被运行时，“-e”參数沒有被打开。

“file”

假设<variable>这个变量被定义在Makefile中。

“command line”

假设<variable>这个变量是被命令行定义的。

“override”

假设<variable>是被override指示符又一次定义的。

“automatic”

假设<variable>是一个命令运行中的自己主动化变量。关于自己主动化变量将在后面讲述。

这些信息对于我们编写Makefile是非常实用的，比如，假设我们有一个Makefile其包了一个定义文件Make.def，在Make.def中定义了一个变量“bletch”，而我们的环境中也有一个环境变量“bletch”，此时，我们想推断一下，假设变量来源于环境，那么我们就把之重定义了，假设来源于Make.def或是命令行等非环境的，那么我们就不又一次定义它。于是，在我们的Makefile中，我们能够这样写：

ifdef bletch

ifeq "$(origin bletch)" "environment"

bletch = barf, gag, etc.

endif

endif

当然，你或许会说，使用overridekeyword不就能够又一次定义环境中的变量了吗？为什么须要使用这样的步骤？是的，我们用override是能够达到这样的效果，可是override过于粗暴，它同一时候会把从命令行定义的变量也覆盖了，而我们仅仅想又一次定义环境传来的，而不想又一次定义命令行传来的。

八、shell函数

shell函数也不像其他的函数。顾名思义，它的參数应该就是操作系统Shell的命令。它和反引號“`”是相同的功能。这就是说，shell函数把运行操作系统命令后的输出作为函数返回。于是，我们能够用操作系统命令以及字符串处理命令awk，sed等等命令来生成一个变量，如：

contents := $(shell cat foo)

files := $(shell echo \*.c)

注意，这个函数会新生成一个Shell程序来运行命令，所以你要注意其运行性能，假设你的Makefile中有一些比較复杂的规则，并大量使用了这个函数，那么对于你的系统性能是有害的。特别是Makefile的隐晦的规则可能会让你的shell函数运行的次数比你想像的多得多。

九、控制make的函数

make提供了一些函数来控制make的运行。通常，你须要检测一些运行Makefile时的运行时信息，而且依据这些信息来决定，你是让make继续运行，还是停止。

$(error <text ...> )

产生一个致命的错误，<text ...>是错误信息。注意，error函数不会在一被使用就会产生错误信息，所以假设你把其定义在某个变量中，并在后续的脚本中使用这个变量，那么也是能够的。比如：

演示例子一：

ifdef ERROR\_001

$(error error is $(ERROR\_001))

endif

演示例子二：

ERR = $(error found an error!)

.PHONY: err

err: ; $(ERR)

演示例子一会在变量ERROR\_001定义了后运行时产生error调用，而演示例子二则在文件夹err被运行时才发生error调用。

$(warning <text ...> )

这个函数非常像error函数，仅仅是它并不会让make退出，仅仅是输出一段警告信息，而make继续运行。

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命令会找当前文件夹的makefile来运行，一切都是自己主动的。但也有时你或许仅仅想让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。本章节就是讲述怎样使用make命令的。

一、make的退出码

make命令运行后有三个退出码：

0 —— 表示成功运行。  
1 —— 假设make运行时出现不论什么错误，其返回1。  
2 —— 假设你使用了make的“-q”选项，而且make使得一些目标不须要更新，那么返回2。

Make的相关參数我们会在后续章节中讲述。

二、指定Makefile

前面我们说过，GNU make找寻默认的Makefile的规则是在当前文件夹下依次找三个文件——“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运行。

当前，我们也能够给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”參数（“--makefile”參数也行）。比如，我们有个makefile的名字是“hchen.mk”，那么，我们能够这样来让make来运行这个文件：

make –f hchen.mk

假设在make的命令行是，你不仅仅一次地使用了“-f”參数，那么，全部指定的makefile将会被连在一起传递给make运行。

三、指定目标

一般来说，make的终于目标是makefile中的第一个目标，而其他目标通常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很多个目标组成，你能够指示make，让其完成你所指定的目标。要达到这一目的非常easy，需在make命令后直接跟目标的名字就能够完成（如前面提到的“make clean”形式）

不论什么在makefile中的目标都能够被指定成终极目标，可是除了以“-”打头，或是包括了“=”的目标，由于有这些字符的目标，会被解析成命令行參数或是变量。甚至沒有被我们明白写出来的目标也能够成为make的终极目标，也就是说，仅仅要make能够找到其隐含规则推导规则，那么这个隐含目标相同能够被指定成终极目标。

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，假设在命令行上，你沒有指定目标，那么，这个变量是空值。这个变量能够让你使用在一些比較特殊的情形下。比方以下的样例：

sources = foo.c bar.c  
ifneq ( $(MAKECMDGOALS),clean)  
include $(sources:.c=.d)  
endif

基于上面的这个样例，仅仅要我们输入的命令不是“make clean”，那么makefile会自己主动包括“foo.d”和“bar.d”这两个makefile。

使用指定终极目标的方法能够非常方便地让我们编译我们的程序，比如以下这个样例：

.PHONY: all  
all: prog1 prog2 prog3 prog4

从这个样例中，我们能够看到，这个makefile中有四个须要编译的程序——“prog1”， “prog2”， “prog3”和 “prog4”，我们能够使用“make all”命令来编译全部的目标（假设把all置成第一个目标，那么仅仅需运行“make”），我们也能够使用“make prog2”来单独编译目标“prog2”。

即然make能够指定全部makefile中的目标，那么也包括“伪目标”，于是我们能够依据这样的性质来让我们的makefile依据指定的不同的目标来完成不同的事。在Unix世界中，软件公布时，特别是GNU这样的开源软件的公布时，其makefile都包括了编译、安装、打包等功能。我们能够參照这样的规则来书写我们的makefile中的目标。

“all”  
这个伪目标是全部目标的目标，其功能通常是编译全部的目标。  
“clean”  
这个伪目标功能是删除全部被make创建的文件。  
“install”  
这个伪目标功能是安装已编译好的程序，事实上就是把目标运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，通常是把tar文件压成Z文件。或是gz文件。  
“TAGS”  
这个伪目标功能是更新全部的目标，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，可是我想，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非常实用了），这里仅仅只是是说明了，假设你要书写这样的功能，最好使用这样的名字命名你的目标，这样规范一些，规范的长处就是——不用解释，大家都明白。而且假设你的makefile中有这些功能，一是非常实用，二是能够显得你的makefile非常专业（不是那种刚开始学习的人的作品）。

四、检查规则

有时候，我们不想让我们的makefile中的规则运行起来，我们仅仅想检查一下我们的命令，或是运行的序列。于是我们能够使用make命令的下述參数：

“-n”  
“--just-print”  
“--dry-run”  
“--recon”  
不运行參数，这些參数仅仅是打印命令，无论目标是否更新，把规则和连带规则下的命令打印出来，但不运行，这些參数对于我们调试makefile非常实用处。

“-t”  
“--touch”  
这个參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，仅仅是把目标变成已编译过的状态。

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

“-W <file>”  
“--what-if=<file>”  
“--assume-new=<file>”  
“--new-file=<file>”  
这个參数须要指定一个文件。通常是是源文件（或依赖文件），Make会依据规则推导来运行依赖于这个文件的命令，一般来说，能够和“-n”參数一同使用，来查看这个依赖文件所发生的规则命令。

另外一个非常有意思的使用方法是结合“-p”和“-v”来输出makefile被运行时的信息（这个将在后面讲述）。

五、make的參数

以下列举了全部GNU make 3.80版的參数定义。其他版本号和产商的make大同小异，只是其他产商的make的具体參数还是请參考各自的产品文档。

“-b”  
“-m”  
这两个參数的作用是忽略和其他版本号make的兼容性。

“-B”  
“--always-make”  
觉得全部的目标都须要更新（重编译）。

“-C <dir>”  
“--directory=<dir>”  
指定读取makefile的文件夹。假设有多个“-C”參数，make的解释是后面的路径曾经面的作为相对路径，并以最后的文件夹作为被指定文件夹。如：“make –C ~hchen/test –C prog”等效于“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，假设沒有參数，那就是输出最简单的调试信息。以下是<options>的取值：  
a —— 也就是all，输出全部的调试信息。（会非常的多）  
b —— 也就是basic，仅仅输出简单的调试信息。即输出不须要重编译的目标。  
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，不须要被重编译的依赖文件（或是依赖目标）等。  
i —— 也就是implicit，输出所以的隐含规则。  
j —— 也就是jobs，输出运行规则中命令的具体信息，如命令的PID、返回码等。  
m —— 也就是makefile，输出make读取makefile，更新makefile，运行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>”  
指定须要运行的makefile。

“-h”  
“--help”  
显示帮助信息。

“-i”  
“--ignore-errors”  
在运行时忽略全部的错误。

“-I <dir>”  
“--include-dir=<dir>”  
指定一个被包括makefile的搜索目标。能够使用多个“-I”參数来指定多个文件夹。

“-j [<jobsnum>]”  
“--jobs[=<jobsnum>]”  
指同一时候运行命令的个数。假设沒有这个參数，make运行命令时能运行多少就运行多少。假设有一个以上的“-j”參数，那么仅最后一个“-j”才是有效的。（注意这个參数在MS-DOS中是没用的）

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

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

“-n”  
“--just-print”  
“--dry-run”  
“--recon”  
仅输出运行过程中的命令序列，但并不运行。

“-o <file>”  
“--old-file=<file>”  
“--assume-old=<file>”  
不又一次生成的指定的<file>，即使这个目标的依赖文件新于它。

“-p”  
“--print-data-base”  
输出makefile中的全部数据，包括全部的规则和变量。这个參数会让一个简单的makefile都会输出一堆信息。假设你仅仅是想输出信息而不想运行makefile，你能够使用“make -qp”命令。假设你想查看运行makefile前的预设变量和规则，你能够使用“make –p –f /dev/null”。这个參数输出的信息会包括着你的makefile文件的文件名称和行号，所以，用这个參数来调试你的makefile会是非常实用的，特别是当你的环境变量非常复杂的时候。

“-q”  
“--question”  
不运行命令，也不输出。仅仅是检查所指定的目标是否须要更新。假设是0则说明要更新，假设是2则说明有发生错误。

“-r”  
“--no-builtin-rules”  
禁止make使用不论什么隐含规则。

“-R”  
“--no-builtin-variabes”  
禁止make使用不论什么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。由于有些时候，make的选项是从环境变量“MAKEFLAGS”中继承下来的。所以你能够在命令行中使用这个參数来让环境变量中的“-k”选项失效。

“-t”  
“--touch”  
相当于UNIX的touch命令，仅仅是把目标的改动日期变成最新的，也就是阻止生成目标的命令运行。

“-v”  
“--version”  
输出make程序的版本号、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。这个參数对于跟踪嵌套式调用make时非常实用。

“--no-print-directory”  
禁止“-w”选项。

“-W <file>”  
“--what-if=<file>”  
“--new-file=<file>”  
“--assume-file=<file>”  
假定目标<file>须要更新，假设和“-n”选项使用，那么这个參数会输出该目标更新时的运行动作。假设沒有“-n”那么就像运行UNIX的“touch”命令一样，使得<file>的改动时间为当前时间。

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

隐含规则  
————

在我们使用Makefile时，有一些我们会常用，而且使用频率非常高的东西，比方，我们编译C/C++的源程序为中间目标文件（Unix下是[.o]文件，Windows下是[.obj]文件）。本章讲述的就是一些在Makefile中的“隐含的”，早先约定了的，不须要我们再写出来的规则。

“隐含规则”也就是一种惯例，make会依照这样的“惯例”心照不喧地来运行，那怕我们的Makefile中沒有书写这样的规则。比如，把[.c]文件编译成[.o]文件这一规则，你根本就不用写出来，make会自己主动推导出这样的规则，并生成我们须要的[.o]文件。

“隐含规则”会使用一些我们系统变量，我们能够改变这些系统变量的值来定制隐含规则的运行时的參数。如系统变量“CFLAGS”能够控制编译时的编译器參数。

我们还能够通过“模式规则”的方式写下自己的隐含规则。用“后缀规则”来定义隐含规则会有很多的限制。使用“模式规则”会更回得智能和清晰，但“后缀规则”能够用来保证我们Makefile的兼容性。  
我们了解了“隐含规则”，能够让其为我们更好的服务，也会让我们知道一些“约定俗成”了的东西，而不至于使得我们在运行Makefile时出现一些我们觉得莫名其妙的东西。当然，不论什么事物都是矛盾的，水能载舟，亦可覆舟，所以，有时候“隐含规则”也会给我们造成不小的麻烦。唯独了解了它，我们才干更好地使用它。

一、使用隐含规则

假设要使用隐含规则生成你须要的目标，你所须要做的就是不要写出这个目标的规则。那么，make会试图去自己主动推导产生这个目标的规则和命令，假设make能够自己主动推导生成这个目标的规则和命令，那么这个行为就是隐含规则的自己主动推导。当然，隐含规则是make事先约定好的一些东西。比如，我们有以下的一个Makefile：

foo : foo.o bar.o  
cc –o foo foo.o bar.o $(CFLAGS) $(LDFLAGS)

我们能够注意到，这个Makefile中并沒有写下怎样生成foo.o和bar.o这两目标的规则和命令。由于make的“隐含规则”功能会自己主动为我们自己主动去推导这两个目标的依赖目标和生成命令。

make会在自己的“隐含规则”库中寻找能够用的规则，假设找到，那么就会使用。假设找不到，那么就会报错。在上面的那个样例中，make调用的隐含规则是，把[.o]的目标的依赖文件置成[.c]，并使用C的编译命令“cc –c $(CFLAGS) [.c]”来生成[.o]的目标。也就是说，我们全然沒有必要写下以下的两条规则：

foo.o : foo.c  
cc –c foo.c $(CFLAGS)  
bar.o : bar.c  
cc –c bar.c $(CFLAGS)

由于，这已经是“约定”好了的事了，make和我们约定好了用C编译器“cc”生成[.o]文件的规则，这就是隐含规则。

当然，假设我们为[.o]文件书写了自己的规则，那么make就不会自己主动推导并调用隐含规则，它会依照我们写好的规则忠实地运行。

还有，在make的“隐含规则库”中，每一条隐含规则都在库中有其顺序，越靠前的则是越被常用的，所以，这会导致我们有些时候即使我们显示地指定了目标依赖，make也不会管。如以下这条规则（沒有命令）：

foo.o : foo.p

依赖文件“foo.p”（Pascal程序的源文件）有可能变得沒有意义。假设文件夹下存在了“foo.c”文件，那么我们的隐含规则一样会生效，并会通过“foo.c”调用C的编译器生成foo.o文件。由于，在隐含规则中，Pascal的规则出如今C的规则之后，所以，make找到能够生成foo.o的C的规则就不再寻找下一条规则了。假设你确实不希望不论什么隐含规则推导，那么，你就不要仅仅写出“依赖规则”，而不写命令。

二、隐含规则一览

这里我们将讲述全部预先设置（也就是make内建）的隐含规则，假设我们不明白地写下规则，那么，make就会在这些规则中寻找所须要规则和命令。当然，我们也能够使用make的參数“-r”或“--no-builtin-rules”选项来取消全部的预设置的隐含规则。

当然，即使是我们指定了“-r”參数，某些隐含规则还是会生效，由于有很多的隐含规则都是使用了“后缀规则”来定义的，所以，仅仅要隐含规则中有“后缀列表”（也就一系统定义在目标.SUFFIXES的依赖目标），那么隐含规则就会生效。默认的后缀列表是：.out, .a, .ln, .o, .c, .cc, .C, .p, .f, .F, .r, .y, .l, .s, .S, .mod, .sym, .def, .h, .info, .dvi, .tex, .texinfo, .texi, .txinfo, .w, .ch .web, .sh, .elc, .el。具体的细节，我们会在后面讲述。

还是先来看一看常用的隐含规则吧。

1、编译C程序的隐含规则。  
“<n>.o”的目标的依赖目标会自己主动推导为“<n>.c”，而且其生成命令是“$(CC) –c $(CPPFLAGS) $(CFLAGS)”

2、编译C++程序的隐含规则。  
“<n>.o”的目标的依赖目标会自己主动推导为“<n>.cc”或是“<n>.C”，而且其生成命令是“$(CXX) –c $(CPPFLAGS) $(CFLAGS)”。（建议使用“.cc”作为C++源文件的后缀，而不是“.C”）

3、编译Pascal程序的隐含规则。  
“<n>.o”的目标的依赖目标会自己主动推导为“<n>.p”，而且其生成命令是“$(PC) –c $(PFLAGS)”。

4、编译Fortran/Ratfor程序的隐含规则。  
“<n>.o”的目标的依赖目标会自己主动推导为“<n>.r”或“<n>.F”或“<n>.f”，而且其生成命令是:  
“.f” “$(FC) –c $(FFLAGS)”  
“.F” “$(FC) –c $(FFLAGS) $(CPPFLAGS)”  
“.f” “$(FC) –c $(FFLAGS) $(RFLAGS)”

5、预处理Fortran/Ratfor程序的隐含规则。  
“<n>.f”的目标的依赖目标会自己主动推导为“<n>.r”或“<n>.F”。这个规则仅仅是转换Ratfor或有预处理的Fortran程序到一个标准的Fortran程序。其使用的命令是：  
“.F” “$(FC) –F $(CPPFLAGS) $(FFLAGS)”  
“.r” “$(FC) –F $(FFLAGS) $(RFLAGS)”

6、编译Modula-2程序的隐含规则。  
“<n>.sym”的目标的依赖目标会自己主动推导为“<n>.def”，而且其生成命令是：“$(M2C) $(M2FLAGS) $(DEFFLAGS)”。“<n.o>” 的目标的依赖目标会自己主动推导为“<n>.mod”，而且其生成命令是：“$(M2C) $(M2FLAGS) $(MODFLAGS)”。

7、汇编和汇编预处理的隐含规则。  
“<n>.o” 的目标的依赖目标会自己主动推导为“<n>.s”，默认使用编译品“as”，而且其生成命令是：“$(AS) $(ASFLAGS)”。“<n>.s” 的目标的依赖目标会自己主动推导为“<n>.S”，默认使用C预编译器“cpp”，而且其生成命令是：“$(AS) $(ASFLAGS)”。

8、链接Object文件的隐含规则。  
“<n>”目标依赖于“<n>.o”，通过运行C的编译器来运行链接程序生成（通常是“ld”），其生成命令是：“$(CC) $(LDFLAGS) <n>.o $(LOADLIBES) $(LDLIBS)”。这个规则对于唯独一个源文件的工程有效，同一时候也对多个Object文件（由不同的源文件生成）的也有效。比如例如以下规则：

x : y.o z.o

而且“x.c”、“y.c”和“z.c”都存在时，隐含规则将运行例如以下命令：

cc -c x.c -o x.o  
cc -c y.c -o y.o  
cc -c z.c -o z.o  
cc x.o y.o z.o -o x  
rm -f x.o  
rm -f y.o  
rm -f z.o

假设沒有一个源文件（如上例中的x.c）和你的目标名字（如上例中的x）相关联，那么，你最好写出自己的生成规则，不然，隐含规则会报错的。

9、Yacc C程序时的隐含规则。  
“<n>.c”的依赖文件被自己主动推导为“n.y”（Yacc生成的文件），其生成命令是：“$(YACC) $(YFALGS)”。（“Yacc”是一个语法分析器，关于其细节请查看相关资料）

10、Lex C程序时的隐含规则。  
“<n>.c”的依赖文件被自己主动推导为“n.l”（Lex生成的文件），其生成命令是：“$(LEX) $(LFALGS)”。（关于“Lex”的细节请查看相关资料）

11、Lex Ratfor程序时的隐含规则。  
“<n>.r”的依赖文件被自己主动推导为“n.l”（Lex生成的文件），其生成命令是：“$(LEX) $(LFALGS)”。

12、从C程序、Yacc文件或Lex文件创建Lint库的隐含规则。  
“<n>.ln” （lint生成的文件）的依赖文件被自己主动推导为“n.c”，其生成命令是：“$(LINT) $(LINTFALGS) $(CPPFLAGS) -i”。对于“<n>.y”和“<n>.l”也是相同的规则。

三、隐含规则使用的变量

在隐含规则中的命令中，基本上都是使用了一些预先设置的变量。你能够在你的makefile中改变这些变量的值，或是在make的命令行中传入这些值，或是在你的环境变量中设置这些值，无论怎么样，仅仅要设置了这些特定的变量，那么其就会对隐含规则起作用。当然，你也能够利用make的“-R”或“--no–builtin-variables”參数来取消你所定义的变量对隐含规则的作用。

比如，第一条隐含规则——编译C程序的隐含规则的命令是“$(CC) –c $(CFLAGS) $(CPPFLAGS)”。Make默认的编译命令是“cc”，假设你把变量“$(CC)”重定义成“gcc”，把变量“$(CFLAGS)”重定义成“-g”，那么，隐含规则中的命令全部会以“gcc –c -g $(CPPFLAGS)”的样子来运行了。

我们能够把隐含规则中使用的变量分成两种：一种是命令相关的，如“CC”；一种是參数相的关，如“CFLAGS”。以下是全部隐含规则中会用到的变量：

1、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文件里扩展文件程序。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文件里扩展文件的程序。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”。

2、关于命令參数的变量

以下的这些变量都是相关上面的命令的參数。假设沒有指明其默认值，那么其默认值都是空。

ARFLAGS   
函数库打包程序AR命令的參数。默认值是“rv”。  
ASFLAGS   
汇编语言编译器參数。（当明显地调用“.s”或“.S”文件时）。   
CFLAGS   
C语言编译器參数。  
CXXFLAGS   
C++语言编译器參数。  
COFLAGS   
RCS命令參数。   
CPPFLAGS   
C预处理器參数。（ C 和 Fortran 编译器也会用到）。  
FFLAGS   
Fortran语言编译器參数。  
GFLAGS   
SCCS “get”程序參数。  
LDFLAGS   
链接器參数。（如：“ld”）  
LFLAGS   
Lex文法分析器參数。  
PFLAGS   
Pascal语言编译器參数。  
RFLAGS   
Ratfor 程序的Fortran 编译器參数。  
YFLAGS   
Yacc文法分析器參数。

四、隐含规则链

有些时候，一个目标可能被一系列的隐含规则所作用。比如，一个[.o]的文件生成，可能会是先被Yacc的[.y]文件先成[.c]，然后再被C的编译器生成。我们把这一系列的隐含规则叫做“隐含规则链”。

在上面的样例中，假设文件[.c]存在，那么就直接调用C的编译器的隐含规则，假设沒有[.c]文件，但有一个[.y]文件，那么Yacc的隐含规则会被调用，生成[.c]文件，然后，再调用C编译的隐含规则终于由[.c]生成[.o]文件，达到目标。

我们把这样的[.c]的文件（或是目标），叫做中间目标。无论怎么样，make会努力自己主动推导生成目标的一切方法，无论中间目标有多少，其都会执着地把全部的隐含规则和你书写的规则全部合起来分析，努力达到目标，所以，有些时候，可能会让你觉得奇怪，怎么我的目标会这样生成？怎么我的makefile发疯了？

在默认情况下，对于中间目标，它和一般的目标有两个地方所不同：第一个不同是除非中间的目标不存在，才会引发中间规则。第二个不同的是，仅仅要目标成功产生，那么，产生终于目标过程中，所产生的中间目标文件会被以“rm -f”删除。

通常，一个被makefile指定成目标或是依赖目标的文件不能被当作中介。然而，你能够明显地说明一个文件或是目标是中介目标，你能够使用伪目标“.INTERMEDIATE”来强制声明。（如：.INTERMEDIATE ： mid ）

你也能够阻止make自己主动删除中间目标，要做到这一点，你能够使用伪目标“.SECONDARY”来强制声明（如：.SECONDARY : sec）。你还能够把你的目标，以模式的方式来指定（如：%.o）成伪目标“.PRECIOUS”的依赖目标，以保存被隐含规则所生成的中间文件。

在“隐含规则链”中，禁止同一个目标出现两次或两次以上，这样一来，就可防止在make自己主动推导时出现无限递归的情况。

Make会优化一些特殊的隐含规则，而不生成中间文件。如，从文件“foo.c”生成目标程序“foo”，按道理，make会编译生成中间文件“foo.o”，然后链接成“foo”，但在实际情况下，这一动作能够被一条“cc”的命令完成（cc –o foo foo.c），于是优化过的规则就不会生成中间文件。

五、定义模式规则

你能够使用模式规则来定义一个隐含规则。一个模式规则就好像一个一般的规则，仅仅是在规则中，目标的定义须要有"%"字符。"%"的意思是表示一个或多个随意字符。在依赖目标中相同能够使用"%"，仅仅是依赖目标中的"%"的取值，取决于其目标。

有一点须要注意的是，"%"的展开发生在变量和函数的展开之后，变量和函数的展开发生在make加载Makefile时，而模式规则中的"%"则发生在运行时。

1、模式规则介绍

模式规则中，至少在规则的目标定义中要包括"%"，否则，就是一般的规则。目标中的"%"定义表示对文件名称的匹配，"%"表示长度随意的非空字符串。比如："%.c"表示以".c"结尾的文件名称（文件名称的长度至少为3），而"s.%.c"则表示以"s."开头，".c"结尾的文件名称（文件名称的长度至少为5）。

假设"%"定义在目标中，那么，目标中的"%"的值决定了依赖目标中的"%"的值，也就是说，目标中的模式的"%"决定了依赖目标中"%"的样子。比如有一个模式规则例如以下：

%.o : %.c ; <command ......>

其含义是，指出了怎么从全部的[.c]文件生成对应的[.o]文件的规则。假设要生成的目标是"a.o b.o"，那么"%c"就是"a.c b.c"。

一旦依赖目标中的"%"模式被确定，那么，make会被要求去匹配当前文件夹下全部的文件名称，一旦找到，make就会规则下的命令，所以，在模式规则中，目标可能会是多个的，假设有模式匹配出多个目标，make就会产生全部的模式目标，此时，make关心的是依赖的文件名称和生成目标的命令这两件事。

2、模式规则演示例子

以下这个样例表示了,把全部的[.c]文件都编译成[.o]文件.

%.o : %.c  
$(CC) -c $(CFLAGS) $(CPPFLAGS) $< -o $@

当中，"$@"表示全部的目标的挨个值，"$<"表示了全部依赖目标的挨个值。这些奇怪的变量我们叫"自己主动化变量"，后面会具体讲述。

以下的这个样例中有两个目标是模式的：

%.tab.c %.tab.h: %.y  
bison -d $<

这条规则告诉make把全部的[.y]文件都以"bison -d <n>.y"运行，然后生成"<n>.tab.c"和"<n>.tab.h"文件。（当中，"<n>"表示一个随意字符串）。假设我们的运行程序"foo"依赖于文件"parse.tab.o"和"scan.o"，而且文件"scan.o"依赖于文件"parse.tab.h"，假设"parse.y"文件被更新了，那么依据上述的规则，"bison -d parse.y"就会被运行一次，于是，"parse.tab.o"和"scan.o"的依赖文件就齐了。（假设，"parse.tab.o"由"parse.tab.c"生成，和"scan.o"由"scan.c"生成，而"foo"由"parse.tab.o"和"scan.o"链接生成，而且foo和其[.o]文件的依赖关系也写好，那么，全部的目标都会得到满足）

3、自己主动化变量

在上述的模式规则中，目标和依赖文件都是一系例的文件，那么我们怎样书写一个命令来完成从不同的依赖文件生成对应的目标？由于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。

自己主动化变量就是完成这个功能的。在前面，我们已经对自己主动化变量有所提涉，相信你看到这里已对它有一个感性认识了。所谓自己主动化变量，就是这样的变量会把模式中所定义的一系列的文件自己主动地挨个取出，直至全部的符合模式的文件都取完了。这样的自己主动化变量仅仅应出如今规则的命令中。

以下是全部的自己主动化变量及其说明：

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

$%  
仅当目标是函数库文件里，表示规则中的目标成员名。比如，假设一个目标是"foo.a(bar.o)"，那么，"$%"就是"bar.o"，"$@"就是"foo.a"。假设目标不是函数库文件（Unix下是[.a]，Windows下是[.lib]），那么，其值为空。

$<  
依赖目标中的第一个目标名字。假设依赖目标是以模式（即"%"）定义的，那么"$<"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。

$?  
全部比目标新的依赖目标的集合。以空格分隔。

$^  
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。

$+  
这个变量非常像"$^"，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。

$\*   
这个变量表示目标模式中"%"及其之前的部分。假设目标是"dir/a.foo.b"，而且目标的模式是"a.%.b"，那么，"$\*"的值就是"dir/a.foo"。这个变量对于构造有关联的文件名称是比較有较。假设目标中沒有模式的定义，那么"$\*"也就不能被推导出，可是，假设目标文件的后缀是make所识别的，那么"$\*"就是除了后缀的那一部分。比如：假设目标是"foo.c"，由于".c"是make所能识别的后缀名，所以，"$\*"的值就是"foo"。这个特性是GNU make的，非常有可能不兼容于其他版本号的make，所以，你应该尽量避免使用"$\*"，除非是在隐含规则或是静态模式中。假设目标中的后缀是make所不能识别的，那么"$\*"就是空值。

当你希望仅仅对更新过的依赖文件进行操作时，"$?"在显式规则中非常实用，比如，假设有一个函数库文件叫"lib"，其由其他几个object文件更新。那么把object文件打包的比較有效率的Makefile规则是：

lib : foo.o bar.o lose.o win.o  
ar r lib $?

在上述所列出来的自己主动量变量中。四个变量（$@、$<、$%、$\*）在扩展时仅仅会有一个文件，而另三个的值是一个文件列表。这七个自己主动化变量还能够取得文件的文件夹名或是在当前文件夹下的符合模式的文件名称，仅仅须要搭配上"D"或"F"字样。这是GNU make中老版本号的特性，在新版本号中，我们使用函数"dir"或"notdir"就能够做到了。"D"的含义就是Directory，就是文件夹，"F"的含义就是File，就是文件。

以下是对于上面的七个变量分别加上"D"或是"F"的含义：

$(@D)  
表示"$@"的文件夹部分（不以斜杠作为结尾），假设"$@"值是"dir/foo.o"，那么"$(@D)"就是"dir"，而假设"$@"中沒有包括斜杠的话，其值就是"."（当前文件夹）。

$(@F)  
表示"$@"的文件部分，假设"$@"值是"dir/foo.o"，那么"$(@F)"就是"foo.o"，"$(@F)"相当于函数"$(notdir $@)"。

"$(\*D)"  
"$(\*F)"  
和上面所述的同理，也是取文件的文件夹部分和文件部分。对于上面的那个样例，"$(\*D)"返回"dir"，而"$(\*F)"返回"foo"

"$(%D)"  
"$(%F)"  
分别表示了函数包文件成员的文件夹部分和文件部分。这对于形同"archive(member)"形式的目标中的"member"中包括了不同的文件夹非常实用。

"$(<D)"  
"$(<F)"  
分别表示依赖文件的文件夹部分和文件部分。

"$(^D)"  
"$(^F)"  
分别表示全部依赖文件的文件夹部分和文件部分。（无相同的）

"$(+D)"  
"$(+F)"  
分别表示全部依赖文件的文件夹部分和文件部分。（能够有相同的）

"$(?D)"  
"$(?F)"  
分别表示被更新的依赖文件的文件夹部分和文件部分。

最后想提醒一下的是，对于"$<"，为了避免产生不必要的麻烦，我们最好给$后面的那个特定字符都加上圆括号，比方，"$(< )"就要比"$<"要好一些。

还得要注意的是，这些变量仅仅使用在规则的命令中，而且一般都是"显式规则"和"静态模式规则"（參见前面"书写规则"一章）。其在隐含规则中并沒有意义。

4、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"%"，或是沒有前后缀，直接就是一个"%"。由于"%"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"，比如"%.c"所匹配的文件"test.c"中"test"就是"茎"。由于在目标和依赖目标中同一时候有"%"时，依赖目标的"茎"会传给目标，当做目标中的"茎"。

当一个模式匹配包括有斜杠（实际也不常常包括）的文件时，那么在进行模式匹配时，文件夹部分会首先被移开，然后进行匹配，成功后，再把文件夹加回去。在进行"茎"的传递时，我们须要知道这个步骤。比如有一个模式"e%t"，文件"src/eat"匹配于该模式，于是"src/a"就是其"茎"，假设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）

5、重载内建隐含规则

你能够重载内建的隐含规则（或是定义一个全新的），比如你能够又一次构造和内建隐含规则不同的命令，如：

%.o : %.c  
$(CC) -c $(CPPFLAGS) $(CFLAGS) -D$(date)

你能够取消内建的隐含规则，仅仅要不在后面写命令就可以。如：

%.o : %.s

相同，你也能够又一次定义一个全新的隐含规则，其在隐含规则中的位置取决于你在哪里写下这个规则。朝前的位置就靠前。

六、老式风格的"后缀规则"

后缀规则是一个比較老式的定义隐含规则的方法。后缀规则会被模式规则逐步地代替。由于模式规则更强更清晰。为了和老版本号的Makefile兼容，GNU make相同兼容于这些东西。后缀规则有两种方式："双后缀"和"单后缀"。

双后缀规则定义了一对后缀：目标文件的后缀和依赖目标（源文件）的后缀。如".c.o"相当于"%o : %c"。单后缀规则仅仅定义一个后缀，也就是源文件的后缀。如".c"相当于"% : %.c"。

后缀规则中所定义的后缀应该是make所认识的，假设一个后缀是make所认识的，那么这个规则就是单后缀规则，而假设两个连在一起的后缀都被make所认识，那就是双后缀规则。比如：".c"和".o"都是make所知道。因而，假设你定义了一个规则是".c.o"那么其就是双后缀规则，意义就是".c"是源文件的后缀，".o"是目标文件的后缀。例如以下演示例子：

.c.o:  
$(CC) -c $(CFLAGS) $(CPPFLAGS) -o $@ $<

后缀规则不同意不论什么的依赖文件，假设有依赖文件的话，那就不是后缀规则，那些后缀统统被觉得是文件名称，如：

.c.o: foo.h  
$(CC) -c $(CFLAGS) $(CPPFLAGS) -o $@ $<

这个样例，就是说，文件".c.o"依赖于文件"foo.h"，而不是我们想要的这样：

%.o: %.c foo.h  
$(CC) -c $(CFLAGS) $(CPPFLAGS) -o $@ $<

后缀规则中，假设沒有命令，那是毫无意义的。由于他也不会移去内建的隐含规则。

而要让make知道一些特定的后缀，我们能够使用伪目标".SUFFIXES"来定义或是删除，如：

.SUFFIXES: .hack .win

把后缀.hack和.win添加后缀列表中的末尾。

.SUFFIXES: # 删除默认的后缀  
.SUFFIXES: .c .o .h # 定义自己的后缀

先清晰默认后缀，后定义自己的后缀列表。

make的參数"-r"或"-no-builtin-rules"也会使用得默认的后缀列表为空。而变量"SUFFIXE"被用来定义默认的后缀列表，你能够用".SUFFIXES"来改变后缀列表，但请不要改变变量"SUFFIXE"的值。

七、隐含规则搜索算法

比方我们有一个目标叫 T。以下是搜索目标T的规则的算法。请注意，在以下，我们沒有提到后缀规则，原因是，全部的后缀规则在Makefile被加载内存时，会被转换成模式规则。假设目标是"archive(member)"的函数库文件模式，那么这个算法会被运行两次，第一次是找目标T，假设沒有找到的话，那么进入第二次，第二次会把"member"当作T来搜索。

1、把T的文件夹部分分离出来。叫D，而剩余部分叫N。（如：假设T是"src/foo.o"，那么，D就是"src/"，N就是"foo.o"）

2、创建全部匹配于T或是N的模式规则列表。

3、假设在模式规则列表中有匹配全部文件的模式，如"%"，那么从列表中移除其他的模式。

4、移除列表中沒有命令的规则。

5、对于第一个在列表中的模式规则：  
1）推导其"茎"S，S应该是T或是N匹配于模式中"%"非空的部分。  
2）计算依赖文件。把依赖文件里的"%"都替换成"茎"S。假设目标模式中沒有包括斜框字符，而把D加在第一个依赖文件的开头。  
3）测试是否全部的依赖文件都存在或是理当存在。（假设有一个文件被定义成另外一个规则的目标文件，或者是一个显式规则的依赖文件，那么这个文件就叫"理当存在"）  
4）假设全部的依赖文件存在或是理当存在，或是就沒有依赖文件。那么这条规则将被採用，退出该算法。

6、假设经过第5步，沒有模式规则被找到，那么就做更进一步的搜索。对于存在于列表中的第一个模式规则：  
1）假设规则是终止规则，那就忽略它，继续下一条模式规则。  
2）计算依赖文件。（同第5步）  
3）测试全部的依赖文件是否存在或是理当存在。  
4）对于不存在的依赖文件，递归调用这个算法查找他能否够被隐含规则找到。  
5）假设全部的依赖文件存在或是理当存在，或是就根本沒有依赖文件。那么这条规则被採用，退出该算法。

7、假设沒有隐含规则能够使用，查看".DEFAULT"规则，假设有，採用，把".DEFAULT"的命令给T使用。

一旦规则被找到，就会运行其相当的命令，而此时，我们的自己主动化变量的值才会生成。

使用make更新函数库文件  
———————————

函数库文件也就是对Object文件（程序编译的中间文件）的打包文件。在Unix下，通常是由命令"ar"来完成打包工作。

一、函数库文件的成员

一个函数库文件由多个文件组成。你能够以例如以下格式指定函数库文件及其组成：

archive(member)

这个不是一个命令，而一个目标和依赖的定义。一般来说，这样的使用方法基本上就是为了"ar"命令来服务的。如：

foolib(hack.o) : hack.o  
ar cr foolib hack.o

假设要指定多个member，那就以空格分开，如：

foolib(hack.o kludge.o)

其等效于：

foolib(hack.o) foolib(kludge.o)

你还能够使用Shell的文件通配符来定义，如：

foolib(\*.o)

二、函数库成员的隐含规则

当make搜索一个目标的隐含规则时，一个特殊的特性是，假设这个目标是"a(m)"形式的，其会把目标变成"(m)"。于是，假设我们的成员是"%.o"的模式定义，而且假设我们使用"make foo.a(bar.o)"的形式调用Makefile时，隐含规则会去找"bar.o"的规则，假设未定义bar.o的规则，那么内建隐含规则生效，make会去找bar.c文件来生成bar.o，假设找得到的话，make运行的命令大致例如以下：

cc -c bar.c -o bar.o  
ar r foo.a bar.o  
rm -f bar.o

另一个变量要注意的是"$%"，这是专属函数库文件的自己主动化变量，有关其说明请參见"自己主动化变量"一节。

三、函数库文件的后缀规则

你能够使用"后缀规则"和"隐含规则"来生成函数库打包文件，如：

.c.a:  
$(CC) $(CFLAGS) $(CPPFLAGS) -c $< -o $\*.o  
$(AR) r $@ $\*.o  
$(RM) $\*.o

其等效于：

(%.o) : %.c  
$(CC) $(CFLAGS) $(CPPFLAGS) -c $< -o $\*.o  
$(AR) r $@ $\*.o  
$(RM) $\*.o

四、注意事项

在进行函数库打包文件生成时，请小心使用make的并行机制（"-j"參数）。假设多个ar命令在同一时间运行在同一个函数库打包文件上，就非常有能够损坏这个函数库文件。所以，在make未来的版本号中，应该提供一种机制来避免并行操作发生在函数打包文件上。

但就眼下而言，你还是应该不要尽量不要使用"-j"參数。

后序  
——

终于到写结束语的时候了，以上基本上就是GNU make的Makefile的全部细节了。其他的产商的make基本上也就是这样的，无论什么样的make，都是以文件的依赖性为基础的，其基本是都是遵循一个标准的。这篇文档中80%的技术细节都适用于不论什么的make，我推测"函数"那一章的内容可能不是其他make所支持的，而隐含规则方面，我想不同的make会有不同的实现，我沒有精力来查看GNU的make和VC的nmake、BCB的make，或是别的UNIX下的make有些什么样的区别，一是时间精力不够，二是由于我基本上都是在Unix下使用make，曾经在SCO Unix和IBM的AIX，如今在Linux、Solaris、HP-UX、AIX和Alpha下使用，Linux和Solaris下许多其他一点。只是，我能够肯定的是，在Unix下的make，无论是哪种平台，差点儿都使用了Richard Stallman开发的make和cc/gcc的编译器，而且，基本上都是GNU的make（公司里全部的UNIX机器上都被装上了GNU的东西，所以，使用GNU的程序也就多了一些）。GNU的东西还是非常不错的，特别是使用得深了以后，越来越觉得GNU的软件的强大，也越来越觉得GNU的在操作系统中（主要是Unix，甚至Windows）"杀伤力"。

对于上述全部的make的细节，我们不但能够利用make这个工具来编译我们的程序，还能够利用make来完成其他的工作，由于规则中的命令能够是不论什么Shell之下的命令，所以，在Unix下，你不一定仅仅是使用程序语言的编译器，你还能够在Makefile中书写其他的命令，如：tar、awk、mail、sed、cvs、compress、ls、rm、yacc、rpm、ftp……等等，等等，来完成诸如"程序打包"、"程序备份"、"制作程序安装包"、"提交代码"、"使用程序模板"、"合并文件"等等五花八门的功能，文件操作，文件管理，编程开发设计，或是其他一些异想天开的东西。比方，曾经在书写银行交易程序时，由于银行的交易程序基本一样，就见到有人书写了一些交易的通用程序模板，在该模板中把一些网络通讯、数据库操作的、业务操作共性的东西写在一个文件里，在这些文件里用些诸如"@@@N、###N"奇怪字串标注一些位置，然后书写交易时，仅仅需依照一种特定的规则书写特定的处理，最后在make时，使用awk和sed，把模板中的"@@@N、###N"等字串替代成特定的程序，形成C文件，然后再编译。这个动作非常像数据库的"扩展C"语言（即在C语言中用"EXEC　SQL"的样子运行SQL语句，在用cc/gcc编译之前，须要使用"扩展C"的翻译程序，如cpre，把其翻译成标准C）。假设你在使用make时有一些更为绝妙的方法，请记得告诉我啊。

回头看看整篇文档，不觉记起几年前刚刚开始在Unix下做开发的时候，有人问我会不会写Makefile时，我两眼发直，根本不知道在说什么。一开始看到别人在vi中写完程序后输入"!make"时，还以为是vi的功能，后来才知道有一个Makefile在作怪，于是上网查啊查，那时又不愿意看英文，发现就根本沒有中文的文档介绍Makefile，仅仅得看别人写的Makefile，自己瞎碰瞎搞才积累了一点知识，但在非常多地方全然是知其然不知所以然。后来开始从事UNIX下产品软件的开发，看到一个400人年，近200万行代码的大工程，发现要编译这样一个庞然大物，假设沒有Makefile，那会是多么恐怖的一样事啊。于是横下心来，狠命地读了一堆英文文档，才觉得对其掌握了。但发现眼下网上对Makefile介绍的文章还是少得那么的可怜，所以想写这样一篇文章，共享给大家，希望能对各位有所帮助。

如今我终于写完了，看了看文件的创建时间，这篇技术文档也写了两个多月了。发现，自己知道是一回事，要写下来，跟别人讲述又是另外一回事，而且，如今越来越沒有时间专研技术细节，所以在写作时，发如今阐述一些细节问题时非常难做到严谨和精练，而且对先讲什么后讲什么不是非常清晰，所以，还是參考了一些国外网站上的资料和题纲，以及一些技术书籍的语言风格，才得以完成。整篇文档的提纲是基于GNU的Makefile技术手冊的提纲来书写的，并结合了自己的工作经验，以及自己的学习历程。由于从来沒有写过这么长，这么细的文档，所以一定会有非常多地方存在表达问题，语言歧义或是错误。因些，我迫切地得等待各位给我指证和建议，以及不论什么的反馈。

最后，还是利用这个后序，介绍一下自己。我眼下从事于全部Unix平台下的软件研发，主要是做分布式计算/网格计算方面的系统产品软件，而且我对于下一代的计算机革命——网格计算非常地感兴趣，对于分布式计算、P2P、Web Service、J2EE技术方向也非常感兴趣，同一时候，对于项目实施、团队管理、项目管理也小有心得，希望相同和我战斗在“技术和管理并重”的阵线上的年轻一代，能够和我多多地交流。我的MSN是：haoel@hotmail.com（常用），QQ是：753640（不常用）。（注：请勿给我MSN的邮箱发信，由于hotmail的垃圾邮件导致我拒收这个邮箱的全部来信）

我欢迎不论什么形式的交流，无论是讨论技术还是管理，或是其他海阔天空的东西。除了政治和娱乐新闻我不关心，其他仅仅要积极向上的东西我都欢迎！

最最后，我还想介绍一下make程序的设计开发人员。

首当其冲的是： Richard Stallman

开源软件的领袖和先驱，从来沒有领过一天工资，从来沒有使用过Windows操作系统。对于他的事迹和他的软件以及他的思想，我无需说过多的话，相信大家对这个人并不比我陌生，这是他的主页：http://www.stallman.org/ 。

第二位是：Roland McGrath

个人主页是：http://www.frob.com/~roland/ ，以下是他的一些事迹：

1） 合作编写了并维护GNU make。

2） 和Thomas Bushnell一同编写了GNU Hurd。

3） 编写并维护着GNU C library。

4） 合作编写并维护着部分的GNU Emacs。

在此，向这两位开源项目的斗士致以最真切的敬意。