Makefile 的编写指导

概述

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 IDE 都为你做了这个工作，但我觉得要作一个好的和 professional 的程序员，makefile 还是要懂。这就好像现在有这么多的 HTML 的编辑器，但如果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，从一个侧面说明了一个人是否具备完成大型工程的能力。

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数，其按类型、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。

现在讲述如何写 makefile 的文章比较少，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++的源码作为我们基础，所以必然涉及一些关于 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 合成执行文件，这个动作叫作链接（link）。

编译时，编译器需要的是语法的正确，函数与变量的声明的正确。对于后者，通常是你需要告诉编译器头文件的所在位置（头文件中应该只是声明，而定义应该放在 C/C++文件中），只要所有的语法正确，编译器就可以编译出中间目标文件。一般来说，每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，我们可以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。链接器并不管函数所在的源文件，只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，而在链接时需要明显地指出中间目标文件名，这对于编译很不方便，所以，我们要给中间目标文件打个包，在 Windows 下这种包叫"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执行文件。在编译时，编译器只检测程序语法，和函数、变量是否被声明。如果函数未被声明，编译器会给出一个警告，但可以生成 Object File。而在链接程序时，链接器会在所有的 Object File 中找寻函数的实现，如果找不到，那到就会报链接错误码（Linker Error），在 VC 下，这种错误一般是：Link 2001 错误，意思说是说，链接器未能找到函数的实现。你需要指定函数的 Object File.

好，言归正传，GNU 的 make 有许多的内容，闲言少叙，还是让我们开始吧。

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。

首先，我们用一个示例来说明 Makefile 的书写规则。以便给大家一个感兴认识。这个示例来源于 GNU 的 make 使用手册，在这个示例中，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 命令如何编译和链接这几个文件。我们的规则是：
1）如果这个工程没有编译过，那么我们的所有 C 文件都要编译并被链接。
2）如果这个工程的某几个 C 文件被修改，那么我们只编译被修改的 C 文件，并链接目标程序。
3）如果这个工程的头文件被改变了，那么我们需要编译引用了这几个头文件的 C 文件，并链接目标程序。

只要我们的 Makefile 写得够好，所有的这一切，我们只用一个 make 命令就可以完成，make 命令会自动智能地根据当前的文件修改的情况来确定哪些文件需要重编译，从而自己编译所需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，可以是 Object File，也可以是执行文件。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 就是，要生成那个 target 所需要的文件或是目标。

command 也就是 make 需要执行的命令。（任意的 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 中如果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执行。这就是 Makefile 的规则。也就是 Makefile 中最核心的内容。

说到底，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、一个示例

正如前面所说的，如果一个工程有 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 应该是下面的这个样子的。

```
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。这样比较便于 Makefile 的易读。我们可以把这个内容保存在文件为"Makefile"或"makefile"的文件中，然后在该目录下直接输入命令"make"就可以生成执行文件 edit。如果要删除执行文件和所有的中间目标文件，那么，只要简单地执行一下"make clean"就可以了。

在这个 makefile 中，目标文件（target）包含：执行文件 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 文件又是执行文件 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。

这里要说明一点的是，clean 不是一个文件，它只不过是一个动作名字，有点像 C 语言中的 lable 一样，其冒号后什么也没有，那么，make 就不会自动去找文件的依赖性，也就不会自动执行其后所定义的命令。要执行其后的命令，就要在 make 命令后明显得指出这个 lable 的名字。这样的方法非常有用，我们可以在一个 makefile 中定义不用的编译或是和编译无关的命令，比如程序的打包，程序的备份，等等。

三、make 是如何工作的

在默认的方式下，也就是我们只输入 make 命令。那么，

1、make 会在当前目录下找名字叫"Makefile"或"makefile"的文件。
2、如果找到，它会找文件中的第一个目标文件（target），在上面的例子中，他会找到"edit"这个文件，并把这个文件作为最终的目标文件。
3、如果 edit 文件不存在，或是 edit 所依赖的后面的 .o 文件的文件修改时间要比 edit 这个文件新，那么，他就会执行后面所定义的命令来生成 edit 这个文件。
4、如果 edit 所依赖的.o 文件也存在，那么 make 会在当前文件中找目标为.o 文件的依赖性，如果找到则再根据那一个规则生成.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，也就是执行文件 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，直到最终编译出第一个目标文件。在找寻的过程中，如果出现错误，比如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 只管文件的依赖性，即，如果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 这种，没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动执行，不过，我们可以显示要 make 执行。即命令——"make clean"，以此来清除所有的目标文件，以便重编译。

于是在我们编程中，如果这个工程已被编译过了，当我们修改了其中一个源文件，比如 file.c，那么根据我们的依赖性，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 的文件修改时间要比 edit 要新，所以 edit 也会被重新链接了（详见 edit 目标文件后定义的命令）。

而如果我们改变了"command.h"，那么，kdb.o、command.o 和 files.o 都会被重编译，并且，edit 会被重链接。

四、makefile 中使用变量

在上面的例子中，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
```

我们可以看到[.o]文件的字符串被重复了两次，如果我们的工程需要加入一个新的[.o]文件，那么我们需要在两个地方加（应该是三个地方，还有一个地方在 clean 中）。当然，我们的 makefile 并不复杂，所以在两个地方加也不累，但如果 makefile 变得复杂，那么我们就有可能会忘掉一个需要加入的地方，而导致编译失败。所以，为了 makefile 的易维护，在 makefile 中我们可以使用变量。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。

比如，我们声明一个变量，叫 objects, OBJECTS, objs, OBJS, obj, 或是 OBJ，反正不管什么啦，只要能够表示 obj 文件就行了。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，我们就可以很方便地在我们的 makefile 中以"$(objects)"的方式来使用这个变量了，于是我们的改良版 makefile 就变成下面这个样子：

```
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)
```

于是如果有新的 .o 文件加入，我们只需简单地修改一下 objects 变量就可以了。

关于变量更多的话题，我会在后续给你一一道来。

五、让 make 自动推导

GNU 的 make 很强大，它可以自动推导文件以及文件依赖关系后面的命令，于是我们就没必要去在每一个[.o]文件后都写上类似的命令，因为，我们的 make 会自动识别，并自己推导命令。

只要 make 看到一个[.o]文件，它就会自动的把[.c]文件加在依赖关系中，如果 make 找到一个 whatever.o，那么 whatever.c，就会是 whatever.o 的依赖文件。并且 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)
```

这种方法，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。

关于更为详细的“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 可以自动推导命令，那么我看到那堆[.o]和[.h]的依赖就有点不爽，那么多的重复的[.h]，能不能把其收拢起来，好吧，没有问题，这个对于 make 来说很容易，谁叫它提供了自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)
```

这种风格，让我们的 makefile 变得很简单，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。我是不喜欢这种风格的，一是文件的依赖关系看不清楚，二是如果文件一多，要加入几个新的.o 文件，那就理不清楚了。

七、清空目标文件的规则

每个 Makefile 中都应该写一个清空目标文件（.o 和执行文件）的规则，这不仅便于重编译，也很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，也许某些文件出现问题，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，下面还有很多 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。

1、显式规则。显式规则说明了，如何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 有自动推导的功能，所以隐晦的规则可以让我们比较粗糙地简略地书写 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 被执行时，其中的变量都会被扩展到相应的引用位置上。

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。有关这一部分的内容，我会在后续的部分中讲述。

5、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

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。

当然，你可以使用别的文件名来书写 Makefile，比如：“Make.Linux”，“Make.Solaris”，“Make.AIX”等，如果要指定特定的 Makefile，你可以使用 make 的“-f”和“--file”参数，如：make -f Make.Linux 或 make --file Make.AIX。

三、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（可以保含路径和通配符）

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，会把找寻 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，如果当前目录下没有找到，那么，make 还会在下面的几个目录下找：

1、如果 make 执行时，有“-I”或“--include-dir”参数，那么 make 就会在这个参数所指定的目录下去寻找。
2、如果目录<prefix>/include（一般是：/usr/local/bin 或/usr/include）存在的话，make 也会去找。

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。

四、环境变量 MAKEFILES

如果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类似于 include 的动作。这个变量中的值是其它的 Makefile，用空格分隔。只是，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，如果环境变量中定义的文件发现错误，make 也会不理。

但是在这里我还是建议不要使用这个环境变量，因为只要这个变量一被定义，那么当你使用 make 时，所有的 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，只是为了告诉大家，也许有时候你的 Makefile 出现了怪事，那么你可以看看当前环境中有没有定义这个变量。

五、make 的工作方式

GNU 的 make 工作时的执行步骤入下：（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。

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

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，所以一定要让 make 知道你的最终目标是什么。一般来说，定义在 Makefile 中的目标可能会有很多，但是第一条规则中的目标将被确立为最终的目标。如果第一条规则中的目标有很多个，那么，第一个目标会成为最终的目标。make 所完成的也就是这个目标。

好了，还是让我们来看一看如何书写规则。

一、规则举例

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

看到这个例子，各位应该不是很陌生了，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、如果生成（或更新）foo.o 文件。也就是那个 cc 命令，其说明了，如何生成 foo.o 这个文件。（当然 foo.c 文件 include 了 defs.h 文件）

二、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。（见上）

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。这个在前面已经讲过了。

如果命令太长，你可以使用反斜框（'\'）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成成目标文件。

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

三、在规则中使用通配符

如果我们想定义一系列比较类似的文件，我们很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（"~"）字符在文件名中也有比较特殊的用途。如果是"~/test"，这就表示当前用户的$HOME 目录下的 test 目录。而"~hchen/test"则表示用户 hchen 的宿主目录下的 test 目录。（这些都是 Unix 下的小知识了，make 也支持）而在 Windows 或是 MS-DOS 下，用户没有宿主目录，那么波浪号所指的目录则根据环境变量"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。

好吧，还是先来看几个例子吧：

```
clean:
rm -f *.o
```

上面这个例子我不不多说了，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
```

上面这个例子说明了通配符也可以在我们的规则中，目标 print 依赖于所有的[.c]文件。其中的"$?"是一个自动化变量，我会在后面给你讲述。

```
objects = *.o
```

上面这个例子，表示了，通符同样可以用在变量中。并不是说[*.o]会展开，不！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)
```

这种用法由关键字"wildcard"指出，关于 Makefile 的关键字，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）

另一个设置文件搜索路径的方法是使用 make 的"vpath"关键字（注意，它是全小写的），这不是变量，这是一个 make 的关键字，这和上面提到的那个 VPATH 变量很类似，但是它更为灵活。它可以指定不同的文件在不同的搜索目录中。这是一个很灵活的功能。它的使用方法有三种：

1、vpath <pattern> <directories>

为符合模式<pattern>的文件指定搜索目录<directories>。

2、vpath <pattern>

清除符合模式<pattern>的文件的搜索目录。

3、vpath

清除所有已被设置好了的文件搜索目录。

vapth 使用方法中的<pattern>需要包含"%"字符。"%"的意思是匹配零或若干字符，例如，"%.h"表示所有以".h"结尾的文件。<pattern>指定了要搜索的文件集，而<directories>则指定了<pattern>的文件集的搜索的目录。例如：

vpath %.h ../headers

该语句表示，要求 make 在"../headers"目录下搜索所有以".h"结尾的文件。（如果某文件在当前目录没有找到的话）

我们可以连续地使用 vpath 语句，以指定不同搜索策略。如果连续的 vpath 语句中出现了相同的<pattern>，或是被重复了的<pattern>，那么，make 会按照 vpath 语句的先后顺序来执行搜索。如：

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

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

```
vpath %.c foo:bar
vpath % blish
```

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

五、伪目标

最早先的一个例子中，我们提到过一个“clean”的目标，这是一个“伪目标”，

```
clean:
rm *.o temp
```

正像我们前面例子中的“clean”一样，即然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”以备完整地重编译而用。 （以“make clean”来使用该目标）

因为，我们并不生成“clean”这个文件。“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。我们只有通过显示地指明这个“目标”才能让其生效。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。

当然，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”来显示地指明一个目标是“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。由于伪目标的特性是，总是被执行的，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”。

随便提一句，从上面的例子我们可以看出，目标也可以成为依赖。所以，伪目标同样也可成为依赖。看下面的例子：

```
.PHONY: cleanall cleanobj cleandiff

cleanall : cleanobj cleandiff
rm program

cleanobj :
rm *.o

cleandiff :
rm *.diff
```

“make clean”将清除所有要被清除的文件。“cleanobj”和“cleandiff”这两个伪目标有点像“子程序”的意思。我们可以输入“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 的规则中的目标可以不止一个，其支持多目标，有可能我们的多个目标同时依赖于一个文件，并且其生成的命令大体类似。于是我们就能把其合并起来。当然，多个目标的生成规则的执行命令是同一个，这可能会可我们带来麻烦，不过好在我们的可以使用一个自动化变量“$@”（关于自动化变量，将在后面讲述），这个变量表示着目前规则中所有的目标的集合，这样说可能很抽象，还是看一个例子吧。

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

上述规则等价于：

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

其中，-$(subst output,,$@)中的“$”表示执行一个 Makefile 的函数，函数名为 subst，后面的为参数。关于函数，将在后面讲述。这里的这个函数是截取字符串的意思，“$@”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

静态模式可以更加容易地定义多目标的规则，可以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，可以有通配符。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。

这样描述这三个东西，可能还是没有说清楚，还是举个例子来说明一下吧。如果我们的<target-parrtern>定义成“%.o”，意思是我们的<target>集合中都是以“.o”结尾的，而如果我们的<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，如果你的文件名中有“%”那么你可以使用反斜杠“\”进行转义，来标明真实的“%”字符。

看一个例子：

```
objects = foo.o bar.o

all: $(objects)
```

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

上面的例子中，指明了我们的目标从$object 中获取，“%.o”表明要所有以“.o”结尾的目标，也就是“foo.o bar.o”，也就是变量$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“$<”和“$@”则是自动化变量，“$<”表示所有的依赖目标集（也就是“foo.c bar.c”），“$@”表示目标集（也就是“foo.o bar.o”）。于是，上面的规则展开后等价于下面的规则：

```
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，如果我们的“%.o”有几百个，那种我们只要用这种很简单的“静态模式规则”就可以写完一堆规则，实在是太有效率了。“静态模式规则”的用法很灵活，如果用得好，那会一个很强大的功能。再看一个例子：

```
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，只要其中模式为“%.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 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 自己依赖于源文件？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 自动更新或自成[.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。

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，所有的[.d]文件依赖于[.c]文件，“rm -f $@”的意思是删除所有的目标，也就是[.d]文件，第二行的意思是，为每个依赖文件“$<”，也就是[.c]文件生成依赖文件，“$@”表示模式“%.d”文件，如果有一个 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 命令的用法请参看相关的使用文档。第四行就是删除临时文件。

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加入[.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]文件也会自动更新了，并会自动生成了，当然，你还可以在这个[.d]文件中加入的不只是依赖关系，包括生成的命令也可一并加入，让每个[.d]文件都包含一个完赖的规则。一旦我们完成这个工作，接下来，我们就要把这些自动生成的规则放进我们的主 Makefile 中。我们可以使用 Makefile 的“include”命令，来引入别的 Makefile 文件（前面讲过），例如：

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)所有[.c]的字串都替换成[.d]，关于这个“替换”的内容，在后面我会有更为详细的讲述。当然，你得注意次序，因为 include 是按次来载入文件，最先载入的[.d]文件中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。

我们在 UNIX 下可能会使用不同的 Shell，但是 make 的命令默认是被“/bin/sh”——UNIX 的标准 Shell 解释执行的。除非你特别指定一个其它的 Shell。Makefile 中，“#”是注释符，很像 C/C++中的“//”，其后的本行字符都被注释。

一、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，我们用这个功能来像屏幕显示一些信息。如：

@echo 正在编译 XXX 模块.....

当 make 执行时，会输出“正在编译 XXX 模块.....”字串，但不会输出命令，如果没有“@”，那么，make 将输出：

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“--slient”则是全面禁止命令的显示。

二、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。

三、命令出错

每当命令运行完后，make 会检测每个命令的返回码，如果命令返回成功，那么 make 会执行下一条命令，当规则中所有的命令成功返回后，这个规则就算是成功完成了。如果一个规则中的某个命令出错了（命令退出码非零），那么 make 就会终止执行当前规则，这将有可能终止所有规则的执行。

有些时候，命令的出错并不表示就是错误的。例如 mkdir 命令，我们一定需要建立一个目录，如果目录不存在，那么 mkdir 就成功执行，万事大吉，如果目录存在，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目录，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，那么就终目该规则的执行，但继续执行其它规则。

四、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。

例如，我们有一个子目录叫 subdir，这个目录下有个 Makefile 文件，来指明了这个目录下文件的编译规则。那么我们总控的 Makefile 可以这样书写：

```
subsystem:
cd subdir && $(MAKE)
```

其等价于：

```
subsystem:
$(MAKE) -C subdir
```

定义$(MAKE)宏变量的意思是，也许我们的 make 需要一些参数，所以定义成一个变量比较利于维护。这两个例子的意思都是先进入"subdir"目录，然后执行 make 命令。

我们把这个 Makefile 叫做"总控 Makefile"，总控 Makefile 的变量可以传递到下级的 Makefile 中（如果你显示的声明），但是不会覆盖下层的 Makefile 中所定义的变量，除非指定了"-e"参数。

如果你要传递变量到下级 Makefile 中，那么你可以使用这样的声明：

```
export <variable ...>
```

如果你不想让某些变量传递到下级 Makefile 中，那么你可以这样声明：

```
unexport <variable ...>
```

如：

示例一：

```
export variable = value
```

其等价于：

```
variable = value
export variable
```

其等价于：

```
export variable := value
```

其等价于：

```
variable := value
export variable
```

示例二：

```
export variable += value
```

其等价于：

variable += value
export variable

如果你要传递所有的变量，那么，只要一个 export 就行了。后面什么也不用跟，表示传递所有的变量。

需要注意的是，有两个变量，一个是 SHELL，一个是 MAKEFLAGS，这两个变量不管你是否 export，其总是要传递到下层 Makefile 中，特别是 MAKEFILES 变量，其中包含了 make 的参数信息，如果我们执行“总控 Makefile”时有 make 参数或是在上层 Makefile 中定义了这个变量，那么 MAKEFILES 变量将会是这些参数，并会传递到下层 Makefile 中，这是一个系统级的环境变量。

但是 make 命令中的有几个参数并不往下传递，它们是“-C”,“-f”,“-h”“-o”和“-W”（有关 Makefile 参数的细节将在后面说明），如果你不想往下层传递参数，那么，你可以这样来：

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

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

还有一个在“嵌套执行”中比较有用的参数，“-w”或是“--print-directory”会在 make 的过程中输出一些信息，让你看到目前的工作目录。比如，如果我们的下级 make 目录是“/home/hchen/gnu/make”，如果我们使用“make -w”来执行，那么当进入该目录时，我们会看到：

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

而在完成下层 make 后离开目录时，我们会看到：

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

当你使用“-C”参数来指定 make 下层 Makefile 时，“-w”会被自动打开的。如果参数中有“-s”（“--slient”）或是“--no-print-directory”，那么，“-w”总是失效的。

五、定义命令包

如果 Makefile 中出现一些相同命令序列，那么我们可以为这些相同的命令序列定义一个变量。定义这种命令序列的语法以“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，因为 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示例中来看看吧。

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

我们可以看见，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”（有关这种以“$”开头的特殊变量，我们会在后面介绍），make 在执行命令包时，命令包中的每个命令会被依次独立执行。

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。

一、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，例如：

```
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，这里只是举个例子来表明 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，给变量加上括号完全是为了更加安全地使用这个变量，在上面的例子中，如果你不想给变量加上括号，那也可以，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。

先看第一种方式，也就是简单的使用“=”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?”（ $(foo)的值是$(bar)，$(bar)的值是$(ugh)，$(ugh)的值是“Huh?”）可见，变量是可以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，我们可以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”。但这种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。

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：

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

其等价于：

```
y := foo bar
x := later
```

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。如果是这样：

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

那么，y 的值是“bar”，而不是“foo bar”。

上面都是一些比较简单的变量使用了，让我们来看一个复杂的例子，其中包括了 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，如果我们的 make 有一个嵌套执行的动作（参见前面的“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。

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，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样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
```

三、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”。

还是看一个示例吧：

```
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)”）

我们还可以使用更多的层次：

```
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，来看一个例子：

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

这里的$($(x))被替换成了$($(y))，因为$(y)值是“z”，所以，最终结果是：a:=$(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)))
```

这个例子中，“$($($(z)))”扩展为“$($(y))”，而其再次被扩展为“$($(subst 1,2,$(x)))”。$(x)的值是“variable1”，subst 函数把“variable1”中的所有“1”字串替换成“2”字串，于是，“variable1”变成“variable2”，再取其值，所以，最终，$(a)的值就是$(variable2)的值——“Hello”。（喔，好不容易）

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”。

再来看看结合第一种技术的例子：

```
a_objects := a.o b.o c.o
1_objects := 1.o 2.o 3.o

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

这个例子中，如果$(a1)的值是“a”的话，那么，$(sources)的值就是“a.c b.c c.c”；如果$(a1)的值是“1”，那么$(sources)的值是“1.c 2.c 3.c”。

再来看一个这种技术和“函数”与“条件语句”一同使用的例子：

```
ifdef do_sort
func := sort
else
func := strip
endif

bar := a d b g q c

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

这个示例中，如果定义了“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”，而如果没有定义“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”这种技术，同样可以用在操作符的左边：

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

这个例子中定义了三个变量：“dir”，“foo_sources”和“foo_print”。

四、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，如：

```
variable := value
variable += more
```

等价于：

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

但如果是这种情况：

```
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，这是很不好的，所以 make 会自动为我们解决这个问题，我们不必担心这个问题。

五、override 指示符

如果有变量是通常 make 的命令行参数设置的，那么 Makefile 中对这个变量的赋值会被忽略。如果你想在 Makefile 中设置这类参数的值，那么，你可以使用“override”指示符。其语法是：

```
override <variable> = <value>
```

override <variable> := <value>

当然，你还可以追加：

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，也同样可以使用 ovveride 指示符，如：

override define foo
bar
endef

六、多行变量

还有一种设置变量值的方法是使用 define 关键字。使用 define 关键字设置变量的值可以有换行，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关键字）。

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

七、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 exprot 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。

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，我样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。

其语法是：

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

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

<variable-assignment>可以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。

这个特性非常的有用，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所有的规则中去。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”

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，变量可以定义在某个目标上。模式变量的好处就是，我们可以给定一种"模式"，可以把变量定义在符合这种模式的所有目标上。

我们知道，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
```

同样，模式变量的语法和"目标变量"一样：

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

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

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

使用条件判断

使用条件判断，可以让 make 根据运行时的不同情况选择不同的执行分支。条件表达式可以是比较变量的值，或是比较变量和常量的值。

一、示例

下面的例子，判断$(CC)变量是否"gcc"，如果是的话，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，在上面示例的这个规则中，目标"foo"可以根据变量"$(CC)"值来选取不同的函数库来编译程序。

我们可以从上面的示例中看到三个关键字：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，表达式包含两个参数，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，任何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（比如“cc”），目标 foo 的规则是：

```
foo: $(objects)
$(CC) -o foo $(objects) $(normal_libs)
```

当然，我们还可以把上面的那个例子写得更简洁一些：

```
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
```

其中<conditional-directive>表示条件关键字，如“ifeq”。这个关键字有四个。

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>"
```

比较参数“arg1”和“arg2”的值是否相同。当然，参数中我们还可以使用 make 的函数。如：

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

这个示例中使用了“strip”函数，如果这个函数的返回值是空（Empty），那么<text-if-empty>就生效。

第二个条件关键字是“ifneq”。语法是：

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

其比较参数“arg1”和“arg2”的值是否相同，如果不同，则为真。和“ifeq”类似。

第三个条件关键字是“ifdef”。语法是：

```
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。还是来看两个例子：

示例一：

```
bar =
foo = $(bar)
ifdef foo
frobozz = yes
else
frobozz = no
endif
```

示例二：

```
foo =
ifdef foo
frobozz = yes
else
frobozz = no
endif
```

第一个例子中，“$(frobozz)”值是“yes”，第二个则是“no”。

第四个条件关键字是“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，多余的空格是被允许的，但是不能以[Tab]键做为开始（不然就被认为是命令）。而注释符“#”同样也是安全的。“else”和“endif”也一样，只要不是以[Tab]键开始就行了。

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

而且，为了避免混乱，make 不允许把整个条件语句分成两部分放在不同的文件中。

使用函数

在 Makefile 中可以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很多，不过已经足够我们的操作了。函数调用后，函数的返回值可以当做变量来使用。

一、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>)
```

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。感觉很像一个变量，是不是？函数中的参数可以使用变量，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。因为统一会更清楚，也会减少一些不必要的麻烦。

还是来看一个示例：

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

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

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 <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>，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”

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))”。

例如有：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。
示例：

$(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：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“”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。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 <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。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 <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。
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 <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。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 <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。如果<s>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<s>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 <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。
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 <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>)。

以上，是所有的字符串操作函数，如果搭配混合使用，可以完成比较复杂的功能。这里，举一个现实中应用的例子。我们知道，make 使用“VPATH”变量来指定“依赖文件”的搜索路径。于是，我们可以利用这个搜索路径来指定编译器对头文件的搜索路径参数 CFLAGS，如：

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

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

三、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 <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 <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。非目录部分是指最后一个反斜杠（"/"）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 <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 <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 <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 <prefix>,<names...>)

名称：加前缀函数——addprefix。
功能：把前缀<prefix>加到<names>中的每个单词后面。
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 <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<list2>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"。

四、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。

五、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。

六、call 函数

call 函数是唯一一个可以用来创建新的参数化的函数。你可以写一个非常复杂的表达式，这个表达式中，你可以定义许多参数，然后你可以用 call 函数来向这个表达式传递参数。其语法是：

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

当 make 执行这个函数时，<expression>参数中的变量，如$(1)，$(2)，$(3)等，会被参数<parm1>，<parm2>，<parm3>依次取代。而<expression>的返回值就是 call 函数的返回值。例如：

reverse = $(1) $(2)

foo = $(call reverse,a,b)

那么，foo 的值就是“a b”。当然，参数的次序是可以自定义的，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 函数不像其它的函数，他并不操作变量的值，他只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”，下面，是 origin 函数的返回值:

“undefined”

如果<variable>从来没有定义过，origin 函数返回这个值“undefined”。

“default”

如果<variable>是一个默认的定义，比如“CC”这个变量，这种变量我们将在后面讲述。

“environment”

如果<variable>是一个环境变量，并且当 Makefile 被执行时，“-e”参数没有被打开。

“file”

如果<variable>这个变量被定义在 Makefile 中。

“command line”

如果<variable>这个变量是被命令行定义的。

“override”

如果<variable>是被 override 指示符重新定义的。

“automatic”

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

这些信息对于我们编写 Makefile 是非常有用的，例如，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，我们想判断一下，如果变量来源于环境，那么我们就把之重定义了，如果来源于 Make.def 或是命令行等非环境的，那么我们就不重新定义它。于是，在我们的 Makefile 中，我们可以这样写：

```
ifdef bletch

ifeq "$(origin bletch)" "environment"

bletch = barf, gag, etc.

endif

endif
```

当然，你也许会说，使用 override 关键字不就可以重新定义环境中的变量了吗？为什么需要使用这样的步骤？是的，我们用 override 是可以达到这样的效果，可是 override 过于粗暴，它同时会把从命令行定义的变量也覆盖了，而我们只想重新定义环境传来的，而不想重新定义命令行传来的。

八、shell 函数

shell 函数也不像其它的函数。顾名思义，它的参数应该就是操作系统 Shell 的命令。它和反引号“`”是相同的功能。这就是说，shell 函数把执行操作系统命令后的输出作为函数返回。于是，我们可以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 程序来执行命令，所以你要注意其运行性能，如果你的 Makefile 中有一些比较复杂的规则，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 函数执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，你需要检测一些运行 Makefile 时的运行时信息，并且根据这些信息来决定，你是让 make 继续执行，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，所以如果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，那么也是可以的。例如：

示例一：

```
ifdef ERROR_001

$(error error is $(ERROR_001))

endif
```

示例二：

```
ERR = $(error found an error!)

.PHONY: err

err: ; $(ERR)
```

示例一会在变量 ERROR_001 定义了后执行时产生 error 调用，而示例二则在目录 err 被执行时才发生 error 调用。

$(warning <text ...>)

这个函数很像 error 函数，只是它并不会让 make 退出，只是输出一段警告信息，而 make 继续执行。

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。

一、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。

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。

当前，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。

三、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）。

四、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）。

五、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，输出所以的隐含规则。
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。

隐含规则

在我们使用 Makefile 时，有一些我们会经常使用，而且使用频率非常高的东西，比如，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，不需要我们再写出来的规则。

“隐含规则”也就是一种惯例，make 会按照这种“惯例”心照不喧地来运行，那怕我们的 Makefile 中没有书写这样的规则。例如，把[.c]文件编译成[.o]文件这一规则，你根本就不用写出来，make 会自动推导出这种规则，并生成我们需要的[.o]文件。

“隐含规则”会使用一些我们系统变量，我们可以改变这些系统变量的值来定制隐含规则的运行时的参数。如系统变量“CFLAGS”可以控制编译时的编译器参数。

我们还可以通过“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许多的限制。使用“模式规则”会更回得智能和清楚，但“后缀规则”可以用来保证我们 Makefile 的兼容性。
我们了解了“隐含规则”，可以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，任何事物都是矛盾的，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。只有了解了它，我们才能更好地使用它。

一、使用隐含规则

如果要使用隐含规则生成你需要的目标，你所需要做的就是不要写出这个目标的规则。那么，make 会试图去自动推导产生这个目标的规则和命令，如果 make 可以自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动推导。当然，隐含规则是 make 事先约定好的一些东西。例如，我们有下面的一个 Makefile：

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

我们可以注意到，这个 Makefile 中并没有写下如何生成 foo.o 和 bar.o 这两目标的规则和命令。因为 make 的“隐含规则”功能会自动为我们自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”库中寻找可以用的规则，如果找到，那么就会使用。如果找不到，那么就会报错。在上面的那个例子中，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，我们完全没有必要写下下面的两条规则：

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

因为，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，如果我们为[.o]文件书写了自己的规则，那么 make 就不会自动推导并调用隐含规则，它会按照我们写好的规则忠实地执行。

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，越靠前的则是越被经常使用的，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。如下面这条规则（没有命令）：

```
foo.o：foo.p
```

依赖文件“foo.p”（Pascal 程序的源文件）有可能变得没有意义。如果目录下存在了“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。因为，在隐含规则中，Pascal 的规则出现在 C 的规则之后，所以，make 找到可以生成 foo.o 的 C 的规则就不再寻找下一条规则了。如果你确实不希望任何隐含规则推导，那么，你就不要只写出“依赖规则”，而不写命令。

二、隐含规则一览

这里我们将讲述所有预先设置（也就是 make 内建）的隐含规则，如果我们不明确地写下规则，那么，make 就会在这些规则中寻找所需要规则和命令。当然，我们也可以使用 make 的参数“-r”或“--no-builtin-rules”选项来取消所有的预设置的隐含规则。

当然，即使是我们指定了“-r”参数，某些隐含规则还是会生效，因为有许多的隐含规则都是使用了“后缀规则”来定义的，所以，只要隐含规则中有“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动推导为“<n>.c”，并且其生成命令是“$(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动推导为“<n>.cc”或是“<n>.C”，并且其生成命令是“$(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动推导为“<n>.p”，并且其生成命令是“$(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动推导为“<n>.r”或“<n>.F”或“<n>.f”，并且其生成命令是:
“.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动推导为“<n>.r”或“<n>.F”。这个规则只是转换 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动推导为“<n>.def”，并且其生成命令是：“$(M2C) $(M2FLAGS) $(DEFFLAGS)”。“<n.o>” 的目标的依赖目标会自动推导为“<n>.mod”，并且其生成命令是：“$(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动推导为“<n>.s”，默认使用编译品“as”，并且其生成命令是：

"$(AS) $(ASFLAGS)"。"<n>.s" 的目标的依赖目标会自动推导为"<n>.S"，默认使用 C 预编译器"cpp"，并且其生成命令是："$(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（一般是"ld"），其生成命令是："$(CC) $(LDFLAGS) <n>.o $(LOADLIBES) $(LDLIBS)"。这个规则对于只有一个源文件的工程有效，同时也对多个 Object 文件（由不同的源文件生成）的也有效。例如如下规则：

x : y.o z.o

并且"x.c"、"y.c"和"z.c"都存在时，隐含规则将执行如下命令：

```
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
```

如果没有一个源文件（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"也是同样的规则。

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。你可以在你的 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，只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，你也可以利用 make 的"-R"或"--no-builtin-variables"参数来取消你所定义的变量对隐含规则的作用。

例如，第一条隐含规则——编译 C 程序的隐含规则的命令是"$(CC) -c $(CFLAGS) $(CPPFLAGS)"。Make 默认的编译命令是"cc"，如果你把变量"$(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)"的样子来执行了。

我们可以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"；一种是参数相的关，如"CFLAGS"。下面是所有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”。

2、关于命令参数的变量

下面的这些变量都是相关上面的命令的参数。如果没有指明其默认值，那么其默认值都是空。

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

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。例如，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”。

在上面的例子中，如果文件[.c]存在，那么就直接调用 C 的编译器的隐含规则，如果没有[.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 编译的隐含规则最终由[.c]生成[.o]文件，达到目标。

我们把这种[.c]的文件（或是目标），叫做中间目标。不管怎么样，make 会努力自动推导生成目标的一切方法，不管中间目标有多少，其都会执着地把所有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，只要目标成功产生，那么，产生最终目标过程中，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，你可以明显地说明一个文件或是目标是中介目标，你可以使用伪目标“.INTERMEDIATE”来强制声明。（如：.INTERMEDIATE ： mid ）

你也可以阻止 make 自动删除中间目标，要做到这一点，你可以使用伪目标“.SECONDARY”来强制声明（如：.SECONDARY : sec）。你还可以把你的目标，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，这一动作可以被一条“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

你可以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，只是在规则中，目标的定义需要有"%"字符。"%"的意思是表示一个或多个任意字符。在依赖目标中同样可以使用"%"，只是依赖目标中的"%"的取值，取决于其目标。

有一点需要注意的是，"%"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 载入 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含"%"，否则，就是一般的规则。目标中的"%"定义表示对文件名的匹配，"%"表示长度任意的非空字符串。例如："%.c"表示以".c"结尾的文件名（文件名的长度至少为 3），而"s.%.c"则表示以"s."开头，".c"结尾的文件名（文件名的长度至少为 5）。

如果"%"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。例如有一个模式规则如下：

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

其含义是，指出了怎么从所有的[.c]文件生成相应的[.o]文件的规则。如果要生成的目标是"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目录下所有的文件名，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，如果有模式匹配出多个目标，make 就会产生所有的模式目标，此时，make 关心的是依赖的文件名和生成目标的命令这两件事。

2、模式规则示例

下面这个例子表示了,把所有的[.c]文件都编译成[.o]文件.

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

其中，"$@"表示所有的目标的挨个值，"$<"表示了所有依赖目标的挨个值。这些奇怪的变量我们叫"自动化变量"，后面会详细讲述。

下面的这个例子中有两个目标是模式的：

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

这条规则告诉 make 把所有的[.y]文件都以"bison -d <n>.y"执行，然后生成"<n>.tab.c"和"<n>.tab.h"文件。（其中，"<n>"表示一个任意字符串）。如果我们的执行程序"foo"依赖于文件"parse.tab.o"和"scan.o"，并且文件"scan.o"依赖于文件"parse.tab.h"，如果"parse.y"文件被更新了，那么根据上述的规则，"bison -d parse.y"就会被执行一次，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，所有的目标都会得到满足）

3、自动化变量

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。在前面，我们已经对自动化变量有所提涉，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。

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

$?
所有比目标新的依赖目标的集合。以空格分隔。

$^
所有的依赖目标的集合。以空格分隔。如果在依赖目标中有多个重复的，那个这个变量会去除重复的依赖目标，只保留一份。

$+
这个变量很像"$^"，也是所有依赖目标的集合。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/a.foo"。这个变量对于构造有关联的文件名是比较有较。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。

当你希望只对更新过的依赖文件进行操作时，"$?"在显式规则中很有用，例如，假设有一个函数库文件叫"lib"，其由其它几个 object 文件更新。那么把 object 文件打包的比较有效率的 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动量变量中。四个变量（$@、$<、$%、$*）在扩展时只会有一个文件，而另三个的值是一个文件列表。这七个自动化变量还可以取得文件的目录名或是在当前目录下的符合模式的文件名，只需要搭配上"D"或"F"字样。这是 GNU make 中老版本的特性，在新版本中，我们使用函数"dir"或"notdir"就可以做到了。"D"的含义就是 Directory，就是目录，"F"的含义就是 File，就是文件。

下面是对于上面的七个变量分别加上"D"或是"F"的含义：

$(@D)
表示"$@"的目录部分（不以斜杠作为结尾），如果"$@"值是"dir/foo.o"，那么"$(@D)"就是"dir"，而如果"$@"中没有包含斜杠的话，其值就是"."（当前目录）。

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

"$(*D)"
"$(*F)"
和上面所述的同理，也是取文件的目录部分和文件部分。对于上面的那个例子，"$(*D)"返回"dir"，而"$(*F)"返回"foo"

"$(%D)"
"$(%F)"
分别表示了函数包文件成员的目录部分和文件部分。这对于形同"archive(member)"形式的目标中的"member"中包含了不同的目录很有用。

"$(<D)"
"$(<F)"
分别表示依赖文件的目录部分和文件部分。

"$(^D)"
"$(^F)"
分别表示所有依赖文件的目录部分和文件部分。（无相同的）

"$(+D)"
"$(+F)"
分别表示所有依赖文件的目录部分和文件部分。（可以有相同的）

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

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，比如，"$(<)"就要比"$<"要好一些。

还得要注意的是，这些变量只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"（参见前面"书写规则"一章）。其在隐含规则中并没有意义。

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）

5、重载内建隐含规则

你可以重载内建的隐含规则（或是定义一个全新的），例如你可以重新构造和内建隐含规则不同的命令，如：

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

你可以取消内建的隐含规则，只要不在后面写命令就行。如：

```
%.o : %.s
```

同样，你也可以重新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"

后缀规则是一个比较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取代。因为模式规则更强更清晰。为了和老版本的 Makefile 兼容，GNU make 同样兼容于这些东西。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"。单后缀规则只定义一个后缀，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，如果一个后缀是 make 所认识的，那么这个规则就是单后缀规则，而如果两个连在一起的后缀都被 make 所认识，那就是双后缀规则。例如：".c"和".o"都是 make 所知道。因而，如果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。如下示例：

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

后缀规则不允许任何的依赖文件，如果有依赖文件的话，那就不是后缀规则，那些后缀统统被认为是文件名，如：

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

这个例子，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，如果没有命令，那是毫无意义的。因为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，我们可以使用伪目标".SUFFIXES"来定义或是删除，如：

```
.SUFFIXES: .hack .win
```

把后缀.hack 和.win 加入后缀列表中的末尾。

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

先清楚默认后缀，后定义自己的后缀列表。

make 的参数"-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，你可以用".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

比如我们有一个目标叫 T。下面是搜索目标 T 的规则的算法。请注意，在下面，我们没有提到后缀规则，原因是，所有的后缀规则在 Makefile 被载入内存时，会被转换成模式规则。

如果目标是"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，如果没有找到的话，那么进入第二次，第二次会把"member"当作 T 来搜索。

1、把 T 的目录部分分离出来。叫 D，而剩余部分叫 N。（如：如果 T 是"src/foo.o"，那么，D 就是"src/"，N 就是"foo.o"）

2、创建所有匹配于 T 或是 N 的模式规则列表。

3、如果在模式规则列表中有匹配所有文件的模式，如"%"，那么从列表中移除其它的模式。

4、移除列表中没有命令的规则。

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。把依赖文件中的"%"都替换成"茎"S。如果目标模式中没有包含斜框字符，而把 D 加在第一个依赖文件的开头。
3）测试是否所有的依赖文件都存在或是理当存在。（如果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）如果所有的依赖文件存在或是理当存在，或是就没有依赖文件。那么这条规则将被采用，退出该算法。

6、如果经过第 5 步，没有模式规则被找到，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）如果规则是终止规则，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）测试所有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是否可以被隐含规则找到。
5）如果所有的依赖文件存在或是理当存在，或是就根本没有依赖文件。那么这条规则被采用，退出该算法。

7、如果没有隐含规则可以使用，查看".DEFAULT"规则，如果有，采用，把".DEFAULT"的命令给 T 使用。

一旦规则被找到，就会执行其相当的命令，而此时，我们的自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，一般是由命令"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。你可以以如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，这种用法基本上就是为了"ar"命令来服务的。如：

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

如果要指定多个 member，那就以空格分开，如：

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

其等价于：

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

你还可以使用 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，如果这个目标是"a(m)"形式的，其会把目标变成"(m)"。于是，如果我们的成员是"%.o"的模式定义，并且如果我们使用"make foo.a(bar.o)"的形式调用 Makefile 时，隐含规则会去找"bar.o"的规则，如果没有定义 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，如果找得到的话，make 执行的命令大致如下：

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

还有一个变量要注意的是"$%"，这是专属函数库文件的自动化变量，有关其说明请参见"自动化变量"一节。

三、函数库文件的后缀规则

你可以使用"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"参数）。如果多个 ar 命令在同一时间运行在同一个函数库打包文件上，就很有可以损坏这个函数库文件。所以，在 make 未来的版本中，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。

但就目前而言，你还是应该不要尽量不要使用"-j"参数。

后序

终于到写结束语的时候了，以上基本上就是 GNU make 的 Makefile 的所有细节了。其它的产商的 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任何的 make，我猜测"函数"那一章的内容可能不是其它 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，我没有精力来查看 GNU 的 make 和 VC 的 nmake、BCB 的 make，或是别的 UNIX 下的 make 有些什么样的差别，一是时间精力不够，二是因为我基本上都是在 Unix 下使用 make，以前在 SCO Unix 和 IBM 的 AIX，现在在 Linux、Solaris、HP-UX、AIX 和 Alpha 下使用，Linux 和 Solaris 下更多一点。不过，我可以肯定的是，在 Unix 下的 make，无论是哪种平台，几乎都使用了 Richard Stallman 开发的 make 和 cc/gcc 的编译器，而且，基本上都是 GNU 的 make（公司里所有的 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 的东西还是很不错的，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"。

对于上述所有的 make 的细节，我们不但可以利用 make 这个工具来编译我们的程序，还可以利用 make 来完成其它的工作，因为规则中的命令可以是任何 Shell 之下的命令，所以，在 Unix 下，你不一定只是使用程序语言的编译器，你还可以在 Makefile 中书写其它的命令，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，或是其它一些异想天开的东西。比如，以前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中，在这些文件中用些诸如"@@@N、###N"奇怪字串标注一些位置，然后书写交易时，只需按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。这个动作很像数据库的"扩展 C"语言（即在 C 语言中用"EXEC　SQL"的样子

执行 SQL 语句，在用 cc/gcc 编译之前，需要使用"扩展 C"的翻译程序，如 cpre，把其翻译成标准 C）。如果你在使用 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，发现就根本没有中文的文档介绍 Makefile，只得看别人写的 Makefile，自己瞎碰瞎搞才积累了一点知识，但在很多地方完全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，如果没有 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。但发现目前网上对 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。

现在我终于写完了，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，现在越来越没有时间专研技术细节，所以在写作时，发现在阐述一些细节问题时很难做到严谨和精练，而且对先讲什么后讲什么不是很清楚，所以，还是参考了一些国外站点上的资料和题纲，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 技术手册的提纲来书写的，并结合了自己的工作经验，以及自己的学习历程。因为从来没有写过这么长，这么细的文档，所以一定会有很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，以及任何的反馈。

最后，还是利用这个后序，介绍一下自己。我目前从事于所有 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，并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 技术方向也很感兴趣，同时，对于项目实施、团队管理、项目管理也小有心得，希望同样和我战斗在"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所有来信）

我欢迎任何形式的交流，无论是讨论技术还是管理，或是其它海阔天空的东西。除了政治和娱乐新闻我不关心，其它只要积极向上的东西我都欢迎！

最最后，我还想介绍一下 make 程序的设计开发者。