Makefile

- 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。
- 参考文档：https://seisman.github.io/how-to-write-makefile/index.html

    -
    - 介绍
        - 基本规则：
        - 示例：
        - make是如何工作的
        - makefile变量使用
        - 让make自动推导
        - makefile的另一种风格
        - 清空目录规则：
        - makefile里面有什么？
        - makefile文件名
        - 包含其他makefile
        - 环境变量makefiles
        - make的工作方式
    - makefile书写规则
        - 规则的语法
        - 使用通配符
        - 文件搜寻
        - 伪目标
        - 多目标
        - 静态模式：
        - 自动生成依赖性：
    - 书写命令
    - 显示命令
    - 执行命令
    - 命令出错
    - 嵌套执行make
    - 定义命令包
- 使用变量
    - 变量的基础
    - 变量中的变量
    - 变量高级用法
    - 追加变量值
    - override指令
    - 多行变量
    - 目标变量
- 使用条件判断
    - 示例
    - 语法
- 使用函数
    - 函数的调用语法
    - 字符串处理函数
        - subst
        - patsubst
        - strip
        - findstring
        - filter

        - filter-out
        - sort

冯恩来的主页 / 学习笔记

        - words
        - firstword
    - 文件名操作函数
        - dir
        - notdir
        - suffix
        - basename
        - addsuffix
        - addprefix
        - join
    - foreach 函数
    - if 函数
    - call函数
    - origin函数
    - shell函数
    - 控制make的函数
- make 的运行
    - make的退出码
    - 指定Makefile
    - 指定目标
    - 检查规则
- 隐含规则
    - 使用隐含规则
    - 隐含规则一览
    - 隐含规则使用的变量
        - 关于命令的变量。
    - 隐含规则链
- 使用make更新函数库文件
    - 函数库文件的成员
    - 函数库成员的隐含规则
    - 函数库文件的后缀规则
    - 注意事项

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，以便给大家一个感性认识。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，从而自动编译所需要的文件和链接目标程序。

基本规则：

```
target ... : prerequisites ...
    recipe
    ...
    ...
```

target

可以是一个object file（目标文件），也可以是一个可执行文件，还可以是一个标签（label）。对于标签这种特性，在后续的“伪目标”章节中会有叙述。

prerequisites

生成该target所依赖的文件和/或target。

该target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。即：

> prerequisites中如果有一个以上的文件比target文件要新的话，recipe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。

示例：

如果一个工程有3个头文件和8个C文件，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，没有实际意义。**这样比较便于makefile的阅读。我们可以把这个内容保存在名字为“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。如果要删除可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：可执行文件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` 文件又是可执行文件 `edit` 的依赖文件。依赖关系的实质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**recipe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label一样，其冒号后什么也没有，那么，make就不会自动去找它的依赖性，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label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。

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` 文件也不存在，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文件和头文件是存在的啦，于是make会生成 `.o` 文件，然后再用 `.o` 文件生成make的终极任务，也就是可执行文件 `edit` 了。

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。

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。

makefile变量使用

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。

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，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。我们的新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)
```

这种方法就是make的“隐式规则”。上面文件内容中， .PHONY 表示 clean 是个伪目标文件。

makefile的另一种风格

既然我们的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)
```

这里 defs.h 是所有目标文件的依赖文件， command.h 和 buffer.h 是对应目标文件的依赖文件。

这种风格能让我们的makefile变得很短，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。

清空目录规则：

每个Makefile中都应该写一个清空目标文件（ .o ）和可执行文件的规则，这不仅便于重编译，也很利于保持文件的清洁。一般的风格都是：

```
clean:
    rm edit $(objects)
```

更为稳健的做法是：

```
.PHONY : clean
```

前面说过， .PHONY 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**”。

makefile里面有什么?

Makefile里主要包含了五个东西：**显式规则、隐式规则、变量定义、指令和注释。**

1. 显式规则。显式规则说明了如何生成一个或多个目标文件。这是由Makefile的书写者明显指出要生成的文件、文件的依赖文件和生成的命令。
2. 隐式规则。由于我们的make有自动推导的功能，所以隐式规则可以让我们比较简略地书写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，这个有点像你C语言中的宏，当Makefile被执行时，其中的变量都会被扩展到相应的引用位置上。
4. 指令。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，可以用反斜杠进行转义，如： \# 。

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

makefile文件名

默认的情况下，make命令会在当前目录下按顺序寻找文件名为 GNUmakefile 、 makefile 和 Makefile 的文件。在这三个文件名中，最好使用 Makefile 这个文件名，因为这个文件名在排序上靠近其它比较重要的文件，比如 README。最好不要用 GNUmakefile，因为这个文件名只能由GNU make ，其它版本的 make 无法识别，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.Solaris”，“Make.Linux”等，如果要指定特定的Makefile，你可以**使用make的 -f 或 --file 参数**，如： make -f Make.Solaris 或 make --file Make.Linux 。如果你使用多条 -f 或 --file 参数，你可以指定多个makefile。

包含其他makefile

在Makefile使用 include 指令可以把别的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 <filenames>...
```

<filenames> 可以是当前操作系统Shell的文件模式（可以包含路径和通配符）。

在 include 前面可以有一些空字符，但是绝不能是 Tab 键开始。 include 和 <filenames> 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 bish 和 bash ，那么，下面的语句：

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

make命令开始时，会找寻 include 所指出的其它Makefile，并把其内容安置在当前的位置。就好像C/C++的 #include 指令一样。如果文件都没有指定绝对路径或是相对路径的话，make会在当前目录下首先寻找，如果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. 接下来按顺序寻找目录 <prefix>/include （一般是 /usr/local/bin ）、 /usr/gnu/include 、 /usr/local/include 、 /usr/include 。

环境变量 .INCLUDE_DIRS 包含当前 make 会寻找的目录列表。你应当避免使用命令行参数 -I 来寻找以上这些默认目录，否则会使得 make “忘掉”所有已经设定的包含目录，包括默认目录。

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 <filenames>...
```

其表示，无论include过程中出现什么错误，都不要报错继续执行。如果要和其它版本 make 兼容，可以使用 sinclude 代替 -include 。

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

make的工作方式

GNU的make工作时的执行步骤如下：（想来其它的make也是类似）

1. 读入所有的Makefile。
2. 读入被include的其它Makefile。
3. 初始化文件中的变量。
4. 推导隐式规则，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。

makefile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。

```
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. 生成或更新 `foo.o` 文件，就是那个cc命令。它说明了如何生成 `foo.o` 这个文件。（当然，foo.c文件include了defs.h文件）

规则的语法

```
targets : prerequisites
    command
    ...
```

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。

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

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

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**三个通配符：`*`，`?` 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` 表示所有后缀为c的文件**。一个需要我们注意的是，如果我们的文件名中有通配符，如：`*`，那么可以用转义字符 `\`，如 `\*` 来表示真实的 `*` 字符，而不是任意长度的字符串。

几个例子：

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

其实在这个clean:后面可以加上你想做的一些事情，如果你想看到在编译完后看看main.c的源代码，你可以在加上cat这个命令，例子如下：

```
clean:
    cat main.c
    rm -f *.o
```

文件搜寻

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

Makefile文件中的特殊变量 `VPATH` 就是完成这个功能的，如果没有指明这个变量，make只会在当前的目录中去找寻依赖文件和目标文件。如果定义了这个变量，那么，make就会在当前目录找不到的情况下，到所指定的目录中去找寻文件了。

```
VPATH = src:../headers
```

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

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：

```
vpath <pattern> <directories>          #为符合模式<pattern>的文件指定搜索目录<directories>。
```

```
vpath <pattern>                    #清除符合模式<pattern>的文件的搜索目录。
```

```
vpath                                  #清除所有已被设置好了的文件搜索目录。
```

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，既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。我们只有通过显式地指明这个“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”来显式地指明一个目标是“伪目标”，向make说明，不管是否有这个文件，这个目标就是“伪目标”。

```
.PHONY : clean
```

只要有这个声明，不管是否有“clean”文件，要运行“clean”这个目标，只有“make clean”这样。于是整个过程可以这样写：

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

伪目标一般没有依赖的文件。但是，我们也可以为伪目标指定所依赖的文件。伪目标同样可以作为“默认目标”，只要将其放在第一个。一个示例就是，如果你的Makefile需要一口气生成若干个可执行文件，但你只想简单地敲一个make完事，并且，所有的目标文件都写在一个Makefile中，那么你可以使用“伪目标”这个特性：

```
  all : prog1 prog2
prog3 .PHONY : all
```

```
    cc -o prog1 prog1.o utils.o

prog2 : prog2.o
    cc -o prog2 prog2.o

prog3 : prog3.o sort.o utils.o
    cc -o prog3 prog3.o sort.o utils.o
```

我们知道，Makefile中的第一个目标会被作为其默认目标。我们声明了一个“all”的伪目标，其依赖于其它三个目标。由于默认目标的特性是，总是被执行的，但由于“all”又是一个伪目标，伪目标只是一个标签不会生成文件，所以不会有“all”文件产生。于是，其它三个目标的规则总是会被决议。也就达到了我们一口气生成多个目标的目的。 `.PHONY : all` 声明了“all”这个目标为“伪目标”。（注：这里的显式“.PHONY : all” 不写的话一般情况也可以正确的执行，这样make可通过隐式规则推导出， “all” 是一个伪目标，执行make不会生成“all”文件，而执行后面的多个目标。建议：显式写出是一个好习惯。）

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 cleanall”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。

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不是同一个，这可能会给我们带来麻烦，不过好在我们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 **`$` 表示执行一个Makefile的函数**，**函数名为subst**，后面的为参数。关于函数，将在后面讲述。这里的这个函数是替换字符串的意思， **`$@` 表示目标的集合，就像一个数组**， `$@` 依次取出目标，并执于命令。

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-pattern是指明了targets的模式，也就是的目标集模式。

如果我们的 target-pattern 是 %.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-pattern 是 %.c，意思是对<target-pattern>所形成的目标集进行二次定义，其计算方法是，取<target-pattern>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。

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。

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。

显示命令

通常，make会把其要执行的命令行在命令执行前输出到屏幕上。当我们用 `@` 字符在命令行前，那么，这个命令将不被make显示出来，最具代表性的例子是，我们用这个功能来向屏幕显示一些信息。如:

```
@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 `--silent` 或 `--quiet` 则是全面禁止命令的显示。

执行命令

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
```

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

命令出错

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。

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 ...>;
```

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                 #命令包名称
yacc $(firstword $^)            #命令
mv y.tab.c $@                   #命令
```

```
endef
```

这里，“run-yacc”是这包的名字，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。

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   //变量
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
```

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
```

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

还有一个比较有用的操作符是 `?=`，先看示例：

```
FOO ?= bar
```

其含义是，**如果FOO没有被定义过，那么变量FOO的值就是“bar”，如果FOO先前被定义过**，那么这条语将什么也不做，其等价于：

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

变量高级用法

这里介绍两种变量的高级使用方法，第一种是**变量值的替换。**

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

还是看一个示例吧：

```
foo := a.o b.o c.o
bar := $(foo:.o=.c)       #执行变量替换
```

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”
```

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

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”。**

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 := $(strip 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”。

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 := $(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。

override指令

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”指令。其语法是:

```
override <variable>; = <value>;
override <variable>; := <value>;
```

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指令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。

define的用法:

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

目标变量

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

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

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。

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。

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)
```

语法

条件表达式的语法为:

```
<conditional-directive>
```

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

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` 键作为开始（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。

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。

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>)        #函数名+参数
```

或是:

```
${<function> <arguments>}
```

这里，`<function>` 就是函数名，make支持的函数不多。`<arguments>` 为函数的参数，参数间以逗号 `,` 分隔，而函数名和参数之间以“空格”分隔。函数调用以 `$` 开头，以圆括号或花括号把函数名和参数括起。感觉很像一个变量，是不是？函数中的参数可以使用变量，为了风格的统一，函数和变量的括号最好一样，如使用 `$(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。

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`。

字符串处理函数

subst

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

- 名称：字符串替换函数
- 功能：把字串 `<text>` 中的 `<from>` 字符串替换成 `<to>`。
- 返回：函数返回被替换过后的字符串。
- 示例：

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

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

---

patsubst

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

- 名称：模式字符串替换函数。
- 功能：查找 <text> 中的单词（单词以“空格”、“Tab”或“回车”“换行”分隔）是否符合模式 <pattern>，如果匹配的话，则以 <replacement> 替换。这里， <pattern> 可以包括通配符 %，表示任意长度的字串。如果 <replacement> 中也包含 %，那么， <replacement> 中的这个 % 将是 <pattern> 中的那个 % 所代表的字串。（可以用 \ 来转义，以 \% 来表示真实含义的 % 字符）
- 返回：函数返回被替换过后的字符串。
- 示例：

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

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

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

```
$(strip <string>)
```

- 名称：去空格函数。
- 功能：**去掉 <string> 字串中开头和结尾的空字符。**
- 返回：返回被去掉空格的字符串值。
- 示例：

```
$(strip a b c )
```

把字串 a b c **去掉开头和结尾的空格**，结果是 a b c。

findstring

```
$(findstring <find>,<in>)
```

- 名称：查找字符串函数
- 功能：在字串 <in> 中查找 <find> 字串。
- 返回：如果找到，那么返回 <find>，否则返回空字符串。
- 示例：

```
$(findstring a,a b c)
$(findstring a,b c)
```

第一个函数返回 a 字符串，第二个返回空字符串

filter

```
$(filter pattern
```


- 名称：过滤函数
- 功能：以 <pattern> 模式过滤 <text> 字符串中的单词，保留符合模式 <pattern> 的单词。可以有多个模式。
- 返回：返回符合模式 <pattern> 的字串。
- 示例：

```
sources := foo.c bar.c baz.s ugh.h
foo: $(sources)
    cc $(filter %.c %.s,$(sources)) -o foo
```

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

filter-out

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

- 名称：反过滤函数
- 功能：以 <pattern> 模式过滤 <text> 字符串中的单词，去除符合模式 <pattern> 的单词。可以有多个模式。
- 返回：返回不符合模式 <pattern> 的字串。
- 示例：

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

`$(filter-out $(mains),$(objects))` 返回值是 `foo.o bar.o` 。

sort

```
$(sort <list>)
```

- ：排序函数
- 功能：给字符串 <list> 中的单词排序（升序）。
- 返回：返回排序后的字符串。
- 示例： `$(sort foo bar lose)` 返回 `bar foo lose` 。
- 备注： `sort` 函数会去掉 <list> 中相同的单词。

word

```
$(word <n>,<text>)
```

- 取单词函数
- 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

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

- 名称：取单词串函数
- 功能：从字符串 <text> 中取从 <ss> 开始到 <e> 的单词串。 <ss> 和 <e> 是一个数字。
- 返回：返回字符串 <text> 中从 <ss> 到 <e> 的单词字串。如果 <ss> 比 <text> 中的单词数要大，那么返回空字符串。如果 <e> 大于 <text> 的单词数，那么返回从 <ss> 开始，到 <text> 结束的单词串。
- 示例： `$(wordlist 2, 3, foo bar baz)` 返回值是 `bar baz` 。

words

- 名称：单词个数统计函数
- 功能：统计 `<text>` 中字符串中的单词个数。
- 返回：返回 `<text>` 中的单词数。
- 示例：`$(words, foo bar baz)` 返回值是 `3`。
- 备注：如果我们要取 `<text>` 中最后的一个单词，我们可以这样：`$(word $(words <text>),<text>)`。

firstword

```
$(firstword <text>)
```

- 名称：首单词函数——firstword。
- 功能：取字符串 `<text>` 中的第一个单词。
- 返回：返回字符串 `<text>` 的第一个单词。
- 示例：`$(firstword foo bar)` 返回值是 `foo`。
- 备注：这个函数可以用 `word` 函数来实现：`$(word 1,<text>)`。

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。

文件名操作函数

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

dir

```
$(dir <names...>)
```

- 名称：取目录函数——dir。
- 功能：从文件名序列 `<names>` 中取出目录部分。目录部分是指最后一个反斜杠（`/`）之前的部分。如果没有反斜杠，那么返回 `./`。
- 返回：返回文件名序列 `<names>` 的目录部分。
- 示例：`$(dir src/foo.c hacks)` 返回值是 `src/ ./`。

notdir

```
$(notdir <names...>)
```

- 名称：取文件函数——notdir。
- 功能：从文件名序列 `<names>` 中取出非目录部分。非目录部分是指最後一个反斜杠（`/`）之后的部分。
- 返回：返回文件名序列 `<names>` 的非目录部分。
- 示例：`$(notdir src/foo.c hacks)` 返回值是 `foo.c hacks`。

suffix

```
$(suffix <names...>)
```

- 名称：取后缀函数——suffix。
- 功能：从文件名序列 `<names>` 中取出各个文件名的后缀。
- 返回：返回文件名序列 `<names>` 的后缀序列，如果文件没有后缀，则返回空字串。
- 示例：`$(suffix src/foo.c src-1.0/bar.c hacks)` 返回值是 `.c .c`。

basename

```
```

- 名称：取前缀函数——basename。
- 功能：从文件名序列 `<names>` 中取出各个文件名的前缀部分。
- 返回：返回文件名序列 `<names>` 的前缀序列，如果文件没有前缀，则返回空字串。
- 示例：`$(basename src/foo.c src-1.0/bar.c hacks)` 返回值是 `src/foo src-1.0/bar hacks`。

addsuffix

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

- 名称：加后缀函数——addsuffix。
- 功能：把后缀 `<suffix>` 加到 `<names>` 中的每个单词后面。
- 返回：返回加过后缀的文件名序列。
- 示例：`$(addsuffix .c,foo bar)` 返回值是 `foo.c bar.c`。

addprefix

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

- 名称：加前缀函数——addprefix。
- 功能：把前缀 `<prefix>` 加到 `<names>` 中的每个单词前面。
- 返回：返回加过前缀的文件名序列。
- 示例：`$(addprefix src/,foo bar)` 返回值是 `src/foo src/bar`。

join

```
$(join <list1>,<list2>)
```

- 名称：连接函数——join。
- 功能：把 `<list2>` 中的单词对应地加到 `<list1>` 的单词后面。如果 `<list1>` 的单词个数要比 `<list2>` 的多，那么，`<list1>` 中的多出来的单词将保持原样。如果 `<list2>` 的单词个数要比 `<list1>` 多，那么，`<list2>` 多出来的单词将被复制到 `<list1>` 中。
- 返回：返回连接过后的字符串。
- 示例：`$(join aaa bbb , 111 222 333)` 返回值是 `aaa111 bbb222 333`。

foreach 函数

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

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

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

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

```
names := a b c d

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

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

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

if 函数

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

```
$(if <condition>,<then-part>)
或者是
$(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。

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call函数来向这个表达式传递参数。其语法是：

```
$(call <expression>,<parm1>,<parm2>,...,<parmn>)
```

当make执行这个函数时， `<expression>` 参数中的变量，如 `$(1)` 、 `$(2)` 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` 。

需要注意：在向 call 函数传递参数时要尤其注意空格的使用。call 函数在处理参数时，第2个及其之后的参数中的空格会被保留，因而可能造成一些奇怪的效果。因而在向call函数提供参数时，最安全的做法是去除所有多余的空格。

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：

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。

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：

```
ifdef bletch
    ifeq "$(origin bletch)" "environment"
        bletch = barf, gag, etc.
    endif
endif
```

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的隐式规则可能会让你的shell函数执行的次数比你想像的多得多。

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。

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

make的退出码

make命令执行后有三个退出码：

0

表示成功执行。

1

如果make运行时出现任何错误，其返回1。

2

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

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。

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。
-
-
-

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

检查规则

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

- `-n`, `--just-print`, `--dry-run`, `--recon`

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

- `-t`, `--touch`

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

- `-q`, `--question`

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

- `-W <file>`, `--what-if=<file>`, `--assume-new=<file>`, `--new-file=<file>`

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

隐含规则

“隐含规则”也就是一种惯例，make会按照这种“惯例”心照不宣地来运行，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”会显得更智能和清楚，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。

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)  foo.c` 来生成 `foo.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。

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。

常用的隐含规则：

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) $(CXXFLAGS) 。（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 ，默认使用编译器 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。

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：

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

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。

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。

函数库文件的成员

一个函数库文件由多个文件组成。你可以用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)
```

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。

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
   ) $(CPPFLAGS) -c $< -o $*.o
    $(AR) r $@ $*.o
    $(RM) $*.o
```

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。

无标签