解剖 Makefile

一 工程管理器 make

当我们要编译成千上万个源程序文件的时候，光靠手工地使用 GCC 工具来达到目的也许就会很没有效率，我们亟需一款能够帮助我们自动检查文件的更新情况，自动进行编译的软件，GNU make（工程管理器 make 在不同环境有很多版本分支，比如 Qt 下的 qmake，windows 下的 nmake 等，下面提到的 make 指的是 LINUX 下的 GNU make）就是这样的一款软件。

而 Makefile，是 make 的配置文件，用来配置运行 make 的时候的一些相关细节，比如指定编译选项，指定编译环境等等。一般而言，一个工程项目不管是简单还是复杂，每一个源代码子目录都会有一个 Makefile 来管理，然后一般有个所谓的顶层 Makefile 来统一管理所有的子目录 Makefile。

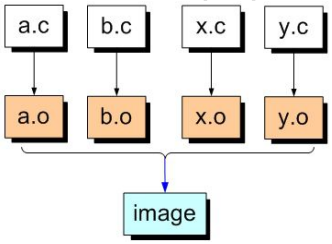
在捋起袖子准备大干一场之前，明确学习目的非常重要，因为 Makefile 的语法相对晦涩，尤其对于没有任何 LINUX 编程和 SHELL 编程经验的新手而言，第一次打开 Makefile 阅读常常有以为是乱码的幻觉！因此面对这样的东西初学者如果抱着对每一个细节“死追不放”的心态可能会死得很惨，信心将被大大挫败，而信心和兴趣的缺失是学习最大的敌人。

假如你是实用主义者，为的是在 LINUX 编程开发不被 Makefile 难倒，那我们学习 Makefile 的程度仅限于看得懂就行了，顶多有时会对某些大型项目的 Makefile 进行修改，但绝对不需要你像对 C 语言那样达到“精通到骨子里”的程度，而这一节的内容就是为这样的人准备的。另一方面，如果你是学院派，需要对工程管理做学术型研究，那可能出了阅读以下内容之外还需要阅读其他专门探讨该专题的文献，但不管你是哪一类人，以下内容作为学习 Makefile 的入门及提高的读物，应该算是这个地球上你能找得到的最贴心的资料了。

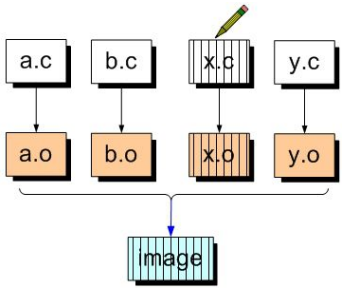
好了，下面通过一个经典例子，说明一下我们为什么需要 make 来管理工程项目：

二 概览性示例

假设我们有一个工程，这个工程总共有4个源文件，姑且叫做 a.c、b.c 以及 x.c 和 y.c 吧，他们最终将会链接生成可执行文件 image，请看：

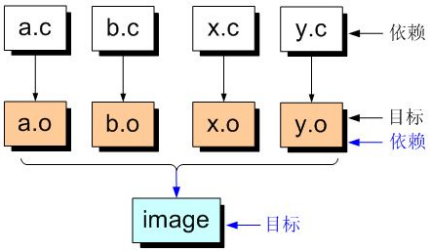


在开发的过程当中，假设我们对 x.c 这个源文件进行了修改，那么，为了在最终的 image 当中体现出来，我们必须重新编译生成 x.o，然后必须重新编译链接生成 image 文件，此过程中，其他未经修改的文件以及他们的目标文件都不需要改动：



由于文件比较少，我们用肉眼就可以简单地辨别，究竟哪些要编译哪些不需要重新再搞一遍，甚至所有文件重新编译一次也不是什么十恶不赦的事情。但是考虑一下一个由成千上万个源文件组成的庞大工程，比如 LINUX 源码，一旦我们对若干个地方进行了修改，重新编译的文件则需要精心地挑选，否则如果整体编译必将会浪费大量时间，这个“精心挑选”的任务，就留给 make 帮我们来实现。

现在，make 的工作目的就很清楚：编译那些需要编译的文件，那么究竟哪些文件需要重新编译呢？这个原理也非常简单：根据文件的时间戳来进行判断。每个文件都会记录其最近修改时间，我们只需要对比源文件及其生成的目标文件的时间戳，就可以判断他们的新旧关系，从而决定要不要编译。比方说我们刚刚修改了 x.c 这个文件，那么他的时间戳将会被更新为当前最新的系统时间，这样 make 通过对比就可以知道 x.c 比 x.o 要新，因此在需要使用 x.o 的时候就会自动重新编译 x.o，这样又会导致 x.o 的时间戳比 image 要新，于是 image 也会被自动重新编译，这种递推关系会在每一层目标-依赖之间传递。



在上面的例子中，image 是最终的目标，其依赖是四个可重定位文件，而对于每一个可重定位文件而言，他们自己本身也是目标，依赖于其相对应的.c 源程序文件。在 make 的眼中，所有的文件都有这么一层一层递推的目标-依赖关系，然后通过对比目标和依赖的时间戳来决定下一步动作，这就是 make 的最基本的工作原理。

下面从零开始，循序渐进，用几个例子将知识点一一攻破，最后看看 LINUX 内核源码中的顶层经典 Makefile，对细节查漏补缺。

三 书写格式

上面讲到，其实 make 的工作原理就是分析判断所谓的“目标-依赖”对，根据他们的存在性和时间戳，来决定下一步动作，这个最根本最原始的工作原理其实跟什么工程管理是没有关系的，比如我们可以写一个世界上最简单的 Makefile：

```
vincent@ubuntu:~$ cat Makefile -n
     1  funny:
     2      echo “just for fun”
vincent@ubuntu:~$ make
echo “just for fun”
just for fun
```

在这个最简单的 Makefile 只有两行，包含了其最核心的语法：第 1 行的 funny 被称之为目标，因为他后面有一个冒号，冒号后面是这个目标的依赖列表，这个例子中 funny 的依赖列表为空，紧跟着第 2 行的行首是一个制表符（即 Tab 键），这个制表符很重要，不能写成空格，更不能省略，其后紧跟着一个 SHELL 语句（事实上就因为有了那个制表符，make 才知道后面是一个 SHELL 命令）。这个目标，以及其后的依赖列表（可以没有），以及其下的 SHELL 命令（可以没有），统称为一套规则。

我们在该 Makefile 所在目录执行 make 命令，结果打印一句 “just for fun”。整个过程中发生的事情是这样的：
1，make 首先判断 funny 这个目标的依赖列表是否都存在，如果是则判断他们跟目标的时间戳关系，如果否则要确保依赖文件都存在。由于这个例子中 funny 没有依赖列表，因此也就不需要判断他们是不是存在了。
2，判断目标 funny 是否已经存在，如果是则退出，如果否则执行下面的 SHELL 命令。该例子中 funny 显然是不存在的，因此将会执行 echo 语句，而且每次执行 echo 语句之后也都不会产生 funny 这个文件，因此每次执行 make 都会打印一句 “just for fun”。

现在，我们来改一下，将这个 Makefile 改成一个更实用一点：用来帮我们 “自动” 执行编译的工作，比如上一节的 image，此时目标是 image，而其依赖则是四个.o 文件，而且，这四个.o 文件本身也是目标，他们依赖于其对应的.c 文件，这个 Makefile 应该长成这样：

```
vincent@ubuntu:~$ cat Makefile -n
     1  image:a.o b.o x.o y.o
     2      gcc a.o b.o x.o y.o -o image
     3
     4  a.o:a.c
     5      gcc a.c -o a.o -c
     6  b.o:b.c
     7      gcc b.c -o b.o -c
     8  x.o:x.c
     9      gcc x.c -o x.o -c
    10  y.o:y.c
    11      gcc y.c -o y.o -c
```

这个简单的 Makefile 文件总共有 11 行，5 套规则，其中第 1 行中的 image 是第 1 个目标，冒号后面是这个目标的依赖列表（四个.o 可重定位文件）。第 2 行行首是一个制表符，后面紧跟着一句 SHELL 命令。

下面从第 4 行到第 11 行，也都是这样的目标-依赖对，及其相关的 SHELL 命令。但是这里必须注意一点：虽然这个 Makefile 总共出现了 5 个目标，但是第一个规则的目标（即 image）被称之为终极目标，终极目标指的是当你执行 make 的时候，默认生成的那个文件。注意：如果第一个规则有多个目标，则只有第一个才是终极目标。另外，以圆点.开头的目标不在此讨论范围内。

这个 Makefile 的工作流程是：
1，找到由终极目标构成的一套规则。（第 1 行和第 2 行）

2，如果终极目标及其依赖列表都存在，则判断他们的时间戳关系，只要目标比任何一个依赖文件旧，就会执行其下面的 SHELL 命令。

3，如果有任何一个依赖文件不存在，或者该依赖文件比该依赖文件的依赖文件要旧，则需要执行以该依赖文件为目标的规则的 SHELL 命令。（比如 a.o 如果不存在或者比 a.c 要旧，则会找到第 4 行和第 5 行这一套规则，并执行第 5 行的 SHELL 命令）

4，如果依赖文件都存在并且都最新，但是目标不存在，则执行其下面的 SHELL 命令。

本例中，一开始所有的.o 文件都是不存在的，因此会执行第 5、第 7、第 9、第 11 行，分别生成 a.o、b.o、x.o 和 y.o，等这些文件都准备妥当了，将会执行第 2 行生成最终的目标文件 image。随后如果对任何一个源文件进行了修改（比如 x.c），执行 make 的时候将会发现其对应的.o 文件（a.o）比该源文件（a.c）要旧，因此就会自动地重新编译（第 9 行），然后根据一样的原理，终极目标文件 image 也被重新编译。

四 变量详解

通过上面的例子，应该对 make 的工作原理及其配置文件 Makefile 的语法结构有个粗浅的了解，但是感觉也没帮上什么忙，毕竟，写在 Makefile 里面的东西一点也没比直接在终端敲命令省事，而且更要命的是：加入现在工程当中再加一个文件 z.c，要放在一起编译，恐怕整个 Makefile 都需要重新修改一遍，另外，假设工程有 1000 个文件，貌似就要写 1000 套规则，这样的结论不免使我们沮丧。但事实上并不需要悲观，Makefile 提供了很多机制，比如变量、函数等来帮助我们更好更方便地组织工作。

下面先来说说变量。

跟SHELL脚本非常类似，在Makefile中也会使用“弱类型”变量（相对于C语言这种强类型语言而言），在Makefile中变量就是一个名字（像是C语言中的宏），代表一个文本字符串（变量的值）。在Makefile的目标、依赖、命令中引用一个变量的地方，变量会被它的值所取代（与C语言中宏引用的方式相同，因此其他版本的make也把变量称之为“宏”）。

在Makefile中变量的特征有以下几点：

1. 变量和函数的展开（除规则的命令行以外），是在make读取Makefile文件时进行的，这里的变量包括了使用“=”定义和使用指示符“define”定义的变量。

2. 变量可以用来代表一个文件名列表、编译选项列表、程序运行的选项参数列表、搜索源文件的目录列表、编译输出的目录列表和所有我们能够想到的事物。

3. 变量名不能包括“:”、“#”、“=”、前置空白和尾空白的任何字符串。需要注意的是，尽管在GNU make中没有对变量的命名有其它的限制，但定义一个包含除字母、数字和下划线以外的变量的做法也是不可取的，因为除字母、数字和下划线以外的其它字符可能会在以后的make版本中被赋予特殊含义，并且这样命名的变量对于一些SHELL来说不能作为环境变量使用。

4. 变量名是大小写敏感的。变量“foo”、“Foo”和“FOO”指的是三个不同的变量。Makefile传统做法是变量名是全采用大写的方式。推荐的做法是在对于内部定义的一般变量（例如：目标文件列表objects）使用小写方式，而对于一些参数列表（例如：编译选项CFLAGS）采用大写方式，这并不是要求的。但需要强调一点：对于一个工程，所有Makefile中的变量命名应保持一种风格，否则会显得你是一个蹩脚的开发者（就像代码的变量命名风格一样），随时有被鄙视的危险。

5. 另外有一些变量名只包含了一个或者很少的几个特殊的字符（符号）。称它们为自动化变量。像“<”、“@”、“?”、“*”、“@D”、“%F”、“^D”等等，后面会详述之。

6. 变量的引用跟SHELL脚本类似，使用美元符号和圆括号，比如有个变量叫A，那么对他的引用则是$(A)，有个自动化变量叫@，则对他的引用是$(@)，有个系统变量是CC则对其引用的格式是$(CC)。对于前面两个变量而言，他们都是单字符变量，因此对他们引用的括号可以省略，写成$A和$@。

Makefile 中有以下几种变量：

1，自定义变量，例如：

```
A = apple
B = I love China
C = $(A) tree
```

以上三个变量都是自定义变量，其中变量 A 包含了一个单词，变量 B 的值包含了三个单词，变量 C 的值引用了变量 A 的值，因此他的值是 “apple tree”。如果要将这三个变量的值打印出来，可以这么写：

```
vincent@ubuntu:~$ cat Makefile -n
     1  A = apple
     2  B = I love China
     3  C = $(A) tree
     4
     5  all:
     6      @echo $(A)   # echo 前面的@代表命令本身不打印出来
     7      @echo $(B)
     8      @echo $(C)
vincent@ubuntu:~$ make
apple
I love China
apple tree
```

使用自定义变量，可以将上述 Makefile 中的所有.o 文件用一个变量 OBJ 来代表：

```
vincent@ubuntu:~$ cat Makefile -n
     1  OBJ = a.o b.o x.o y.o
     2
     3  image:$(OBJ)
     4      gcc $(OBJ) -o image
     5
     6  a.o:a.c
     7      gcc a.c -o a.o -c
     8  b.o:b.c
     9      gcc b.c -o b.o -c
    10  x.o:x.c
    11      gcc x.c -o x.o -c
    12  y.o:y.c
    13      gcc y.c -o y.o -c
```

2，系统预定义变量，例如：

CFLAGS、CC、MAKE、SHELL 等等，这些变量已经有了系统预定义好的值，当然我们可以根据需要重新给他们赋值，例如 CC 的默认值是 gcc，当我们需要使用 c 编译器的时候可以直接使用他：

vincent@ubuntu:~$ **cat Makefile -n**

```
1  OBJ = a.o b.o x.o y.o
2
3  image:$(OBJ)
4      $(CC) $(OBJ) -o image
5
6  a.o:a.c
7      $(CC) a.c -o a.o -c
8  b.o:b.c
9      $(CC) b.c -o b.o -c
10 x.o:x.c
11     $(CC) x.c -o x.o -c
12 y.o:y.c
13     $(CC) y.c -o y.o -c
```

这样做的好处是：在不同平台中，c 编译器的名称也许会发生变化，如果我们的 Makefile 使用了 100 处 c 编译器的名字，那么换一个平台我们只需要重新给预定义变量 CC 赋值一次即可，而不需要修改 100 处不同的地方。比如我们换到 ARM 开发平台中，只需要重新给 CC 赋值为 arm-linux-gnu-gcc，请看：

vincent@ubuntu:~$ **cat Makefile -n**

```
1  OBJ = a.o b.o x.o y.o
2  CC = arm-linux-gnu-gcc
3
4  image:$(OBJ)
5      $(CC) $(OBJ) -o image
6
7  a.o:a.c
8      $(CC) a.c -o a.o -c
9  b.o:b.c
10     $(CC) b.c -o b.o -c
11 x.o:x.c
12     $(CC) x.c -o x.o -c
13 y.o:y.c
14     $(CC) y.c -o y.o -c
```

此时的 CC 就不是 gcc 而是交叉工具链 arm-linux-gnu-gcc 了，很方便。常用的系统预定义变量，请看下表：

变量名	含义	备注
AR	函数库打包程序，可创建静态库.a文档。默认是“ar”。	

AS	汇编程序。默认是"as"。	
CC	C编译程序。默认是"cc"。	
CXX	C++编译程序。默认是"g++"。	
CPP	C程序的预处理器。默认是"$(CC) –E"。	
RM	删除命令。默认是"rm –f"。	
ARFLAGS	执行AR命令的命令行参数。默认值是"rv"。	
ASFLAGS	汇编器AS的命令行参数（明确指定".s"或".S"文件时）。	
CFLAGS	执行CC编译器的命令行参数（编译.c源文件的选项）。	
CXXFLAGS	执行g++编译器的命令行参数（编译.cc源文件的选项）。	

3，自动化变量，例如：

<、@、?、#等等，这些特殊的变量之所以称为自动化变量，是因为他们的值会“自动地”发生变化，考虑普通的变量，只要你不给他重新赋值，那么他的值是永久不变的，比如上面的 CC，只要不对他重新赋值，CC 永远都等于 arm-linux-gnu-gcc。但是自动化变量的值是不固定的，你不能说@的值等于几，但是他的含义的固定的：@代表了其所在规则的目标的完整名称。

有关自动化变量的详细情况，见下表：

变量名	含义	备注
@	代表其所在规则的目标的完整名称	
%	代表其所在规则的静态库文件的一个成员名	
<	代表其所在规则的依赖列表的第一个文件的完整名称	
?	代表所有时间戳比目标文件新的依赖文件列表，用空格隔开	
^	代表其所在规则的依赖列表	同一文件不可重复
+	代表其所在规则的依赖列表	同一文件可重复，主要用在程序链接时，库的交叉引用场合。
*	在模式规则和静态模式规则中，代表茎	茎是目标模式中"%"所代表的部分（当文件名中存在目录时，茎也包含目录（斜杠之前）部分。

上述列出的自动量变量中。其中有四个在规则中代表一个文件名（$@、$<、$%、$*）。而其它三个的在规则中代表一个文件名的列表。

GUN make中，还可以通过这七个自动化变量来获取一个完整文件名中的目录部分或者具体文件名，需要在这些变量中加入"D"或者"F"字符。这样就形成了一系列变种的自动化变量：

变量名	含义	备注
@D	代表目标文件的目录部分(去掉目录部分的最后一个斜杠)	如果"$@"是"dir/foo.o"，那么"$(@D)"的值为"dir"。如果"$@"不存在斜杠，其值就是"."（当前目录)。注意它和函数"dir"

		的区别
@F	目标文件的完整文件名中除目录以外的部分（实际文件名）	如果“$@”为“dir/foo.o”，那么“$(@F)”只就是“foo.o”。“$(@F)”等价于函数“$(notdir $@)”
*D	代表目标茎中的目录部分	
*F	代表目标茎中的文件名部分	
%D	当以如“archive(member)”形式静态库为目标时，表示库文件成员“member”名中的目录部分	仅对“archive(member)”形式的规则目标有效
%F	当以如“archive(member)”形式静态库为目标时，表示库文件成员“member”名中的文件名部分	仅对“archive(member)”形式的规则目标有效
<D	代表规则中第一个依赖文件的目录部分	
<F	代表规则中第一个依赖文件的文件名部分	
^D	代表所有依赖文件的目录部分	同一文件不可重复
^F	代表所有依赖文件的文件名部分	同一文件不可重复
+D	代表所有依赖文件的目录部分	同一文件可重复
+F	代表所有依赖文件的文件名部分	同一文件可重复
?D	代表被更新的依赖文件的目录部分。	
?F	代表被更新的依赖文件的文件名部分。	

使用自动化变量，之前的 Makefile 变成：

```
vincent@ubuntu:~$ cat Makefile -n
     1  OBJ = a.o b.o x.o y.o
     2
     3  image:$(OBJ)
     4      $(CC) $^ -o $@
     5
     6  a.o:a.c
     7      $(CC) $^ -o $@ -c
     8  b.o:b.c
     9      $(CC) $^ -o $@ -c
    10  x.o:x.c
    11      $(CC) $^ -o $@ -c
    12  y.o:y.c
    13      $(CC) $^ -o $@ -c
```

但其实，自动化变量的用武之地是模式规则，在模式规则中才能体现自动化变量可以自动变化的特点，在上面的例子中仅仅是简化了单词的拼写而已。

Makefile 中定义的变量有以下几种不同的方式：

1，递归定义方式：

```
A = I love $(B)
```

B = China

此处，在变量 B 出现之前，变量 A 的定义包含了对变量 B 的引用，由于 A 的定义方式是所谓的“递归”定义方式，因此当出现$(B)时会对全文件进行搜索，找到 B 的值并代进 A 中，结果变量 A 的值是“I love China”

递归定义的变量有两个缺点：第一，使用此风格的变量定义，可能会由于出现变量的递归定义而导致make陷入到无限的变量展开过程中，最终使make执行失败。例如A=$(A)，这将导致无限嵌套迭代。第二，这种风格变量的定义中如果使引用了某一个函数，那么函数总会在其被引用的地方被执行。是因为这种风格变量的定义中，对函数引用的替换展开发生在展开它自身的时候，而不是在定义它的时候。这样所带来的问题是，可能可能会使make的执行效率降低，同时对某些变量和函数的引用出现问题。特别是当变量定义中引用了“shell”和“wildcard”函数的情况，可能出现不可控制或者难以预料的错误，因为我们无法确定它在何时会被展开。

2，直接定义方式：

B = China

A := I love $(B)

此处，定义 A 时用的是所谓的“直接”定义方式，说白了就是如果其定义里出现有对其他变量的引用的话，只会其前面的语句进行搜寻（不包含自己所在的那一行），而不是搜寻整个文件，因此，如果此处将变量 A 和变量 B 的定义交换一个位置：

A := I love $(B)

B = China

则 A 的值将不包含 China，因此在定义 A 时 B 的值为空。

3，条件定义方式：

有时我们需要先判断一个变量是否已经定义了，如果已经定义了则不作操作，如果没有定义再来定义它的值，这时最方便的方法就是采用所谓的条件定义方式：

A = apple

A ?= I love China

此处对 A 进行了两次定义，其中第二次是条件定义，其含义是：如果 A 在此之前没有定义，则定义为“I love China”，否则维持原有的值。

3，多行命令定义方式：

```
define commands
    echo “thank you!”
    echo “you are welcome.”
endef
```

此处定义了一个包含多行命令的变量commands，我们利用它的这个特点实现一个完整命令包的定义。注意其语法格式：以define开头，以endef结束，所要定义的变量名必须在指示符“define”的同一行之后，指示符define所在行的下一行开始一直到“end”所在行的上一行之间的若干行，是变量的值。这种方式定义的所谓命令包，可以理解为编程语言中的函数。

Makefile中的变量还有以下几种操作方式：

1，追加变量的值，例如：

```
A = apple
A += tree
```

这样，变量A的值就是apple tree。

2，修改变量的值，例如：

```
A = srt.c string.c tcl.c
B = $(A:%.c=%.o)
```

这样，变量B的值就变成了 srt.o string.o tcl.o。例子中$(A:%.c=%.o)的意思是：将变量A中所有以.c作为后缀的单词，替换为以.o作为后缀。其实这种变量的替换功能是内嵌函数patsubst的简单版本，使用patsubst也可以实现这个替换的功能：

```
A = srt.c string.c tcl.c
B = $(patsubst %.c, %.o, $(A))
```

3，override一个变量，例如：

```
override CFLAGS += -Wall
```

在执行make时，通常可以在命令行中携带一个变量的定义，如果这个变量跟Makefile中出现的某一变量重名，那么命令行变量的定义将会覆盖Makefile中的变量。就是说，对于一个在Makefile中使用常规方式（使用“=”、“:=”或者“define”）定义的变量，我们可以在执行make时通过命令行方式重新指定这个变量的值，命令行指定的值将替代出现在Makefile中此变量的值。比如：

```
vincent@ubuntu:~$ cat Makefile -n
     1  A = an apple tree
     2
     3  all:
     4      @echo $(A)
vincent@ubuntu:~$ make A="an elephant"
an elephant
```

可见，虽然Makefile定义了A的值为”an apple tree”，但被命令行定义的A的值覆盖了，变成了”an elephant”。如果不想被覆盖，则可以写成：

```
vincent@ubuntu:~$ cat Makefile -n
     1  override A = an apple tree
     2
     3  all:
     4      @echo $(A)
vincent@ubuntu:~$ make A="an elephant"
an apple tree
```

但是请注意，指示符“override”并不是用来防止Makefile的内部变量被命令行参数覆盖的，其存在的目的是为了使用户可以改变或者追加那些使用make的命令行指定的变量的定义。从另外一个角度来说，就是实现了在Makefile中增加或者修改命令行参数的一种机制。想象一下，我们可能会有这样的需求；通过命令行来指定一些附加的编译参数，对一些

通用的参数或者必需的编译参数我们可以在Makefile中指定，而在命令行中可以指定一些特殊的参数。对待这种需求，我们可以使用指示符“override”来实现。

例如无论命令行指定那些编译参数，必须打开所有的编译警告信息“-Wall”，我们的Makefile对“CFLAGS”应该这样写：

override CFLAGS += -Wall

这样，无论通过命令行指定那些编译选项，“-Wall”参数始终存在。比如：

```
vincent@ubuntu:~$ cat Makefile -n
     1  override CFLAGS += -Wall
     2
     3  test:test.c
vincent@ubuntu:~$ make CFLAGS="-g"
cc -g -O0 -Wall a.c -o a
```

4，导出变量，例如：

export CFLAGS = -Wall -g

在Makefile中导出一个变量的作用是：使得该变量可以传递给子Makefile。在缺省的情况下，除了两个特殊的变量”SHELL”、”MAKEFLAGS”、不为空的”MAKEFILES”以及在执行make之前就已经存在的环境变量之外，其他变量不会被传递给子Makefile。

例如：

```
vincent@ubuntu:~/test$ tree
.                          <= 用来测试的两个Makefile，其中子Makefile位于dir/下
├── dir/
│   └── Makefile
└── Makefile
vincent@ubuntu:~$ cat Makefile -n
     1  export A = apple     <= 在顶层Makefile中，将变量A导出
     2  B = banana
     3
     4  all:
     5      echo $(A)
     6      echo $(B)
     7      $(MAKE) -C dir/  <= 调用位于dir/中的子Makefile
vincent@ubuntu:~$ cat dir/Makefile -n
     1  all:
     2      echo $(A)
     3      echo $(B)
vincent@ubuntu:~$ make -sw    <= 在顶层执行Makefile，且显示目录
make: Entering directory `/home/vincent'
apple
banana                     <= 顶层Makefile将两个变量的值都打印了出来
make[1]: Entering directory `/home/vincent/dir'
apple
                           <= 子Makefile只打印了在上一级Makefile中通过export导出的A
make[1]: Leaving directory `/home/vincent/dir'
```

```
make: Leaving directory `/home/vincent'
```

对于默认就会被传递给子Makefile的变量，可以使用unexport来阻止他们的传递，比如：

```
unexport MAKEFLAGS
```

这样，上一级Makefile的命令行参数就不会传递给子Makefile了。

最后来看看几个重要的特殊变量：

1，VPATH

这个特殊的变量用以指定Makefile中文件的备用搜寻路径：当Makefile中的目标文件或者依赖文件不在当前路径时，make会在此变量所指定的目录中搜寻，如果VPATH包含多个备用路径，他们使用空格或者冒号隔开。例如：

```
vincent@ubuntu:~$ tree
.
├── Makefile
├── src1/
│   └── a.c
└── src2/
    └── b.c
vincent@ubuntu:~$ cat Makefile -n
     1  VPATH = src1/:src2/          ⇐ 指定文件搜寻除当前路径之外的备用路径
     2
     3  all: a b
     4  a:a.c                        ⇐ 若make发现当前路径下不存在a.c，则会到VPATH中去找
     5      gcc $^ -o $@
     6  b:b.c
     7      gcc $^ -o $@
vincent@ubuntu:~$ make
gcc src1/a.c -o a
gcc src2/b.c -o b
```

更进一步，可以使用小写的指示符vpath来更灵活地为各种不同的文件指定不同的路径，比如我们增加一个include/路径用来存放本工程的头文件，则Makefile改成：

```
vincent@ubuntu:~$ tree
.
├── include/
│   └── head.h
├── Makefile
├── src1/
│   └── a.c
└── src2/
    └── b.c
vincent@ubuntu:~$ cat Makefile -n
```

```
1  vpath %.c = src1/:src2/
2  vpath %.h = include/
3
4  all:a b
5
6  a:a.c head.h
7      $(CC) $< -o $@
8  b:b.c head.h
9      $(CC) $< -o $@
```


指定本Makefile中.c文件的可能路径


指定本Makefile中.h文件的可能路径

```
vincent@ubuntu:~$ make
cc src1/a.c -o a
cc src2/b.c -o b
```

注意，VPATH是一个变量，而vpath是一个指示符。

2，MAKE

当需要在一个Makefile中调用子Makefile时，用到的变量就是MAKE，实际上该变量代表了当前系统中make软件的全路径，比如：/usr/bin/make。其具体用法是：

```
$(MAKE) -C subdir/
```

其中 -C subdir/ 代表指定子Makefile所在目录，详细案例请参照前面“导出变量”小节。

3，MAKEFLAGS

此变量代表了在执行make时的命令行参数，这个变量是缺省会被传递给子Makefile的特殊变量之一。比如：

```
vincent@ubuntu:~$ cat Makefile -n
     1  all:
     2      echo $(MAKEFLAGS)
vincent@ubuntu:~$ make -s
s
```

此处，s就是make的命令行参数。

五 各种规则

第一，隐式规则。

上面用来管理四个源程序文件（a.c b.c x.c 和 y.c）的那个 Makefile 还是显得比较笨拙，需要对每一个文件编写一个规则，但其实 Makefile 是有一定的智能的，我们可以将编译语句省略掉，也可以将依赖文件都省略掉，甚至连目标都省略掉！比如可以写成这样：

```
vincent@ubuntu:~$ cat Makefile -n
     1  OBJ = a.o b.o x.o y.o
     2
     3  image:$(OBJ)
     4      $(CC) $(OBJ) -o image
vincent@ubuntu:~$ make
cc     -c -o a.o a.c
```

```
cc      -c -o b.o b.c
cc      -c -o x.o x.c
cc      -c -o y.o y.c
gcc a.o b.o x.o y.o -o image
```

可以看到，虽然后四个规则的目标、依赖文件和编译语句都没写，但是执行 make 也照样可以运行，可见 make 会自动帮我们找到.o 文件所需要的源程序文件，也能自动帮我们生成对应的编译语句，这个情况称之为 Makefile 的隐式规则。

但是也看到，虽然我们可以省略后四个规则的依赖文件和编译语句，但是第一个规则的依赖文件和编译语句不能省略，因为隐式规则是有限制的，他只能自动找到跟目标同名的依赖文件，比如目标叫 a.o，那么他会自动查找到 a.c，换了个名字就找不到了，生成的编译语句也是缺省的单文件形式，像本例子中的第一个规则，隐式规则就无能为力了，因为 image 的依赖文件不止一个。

使用隐式规则虽然看起来方便，但是也有弊端：第一，有时一个目标可能并不是一个文件，而仅仅是一个动作，这时就不应该在其身上运用隐式规则。第二，使用隐式规则不能让我们更好地控制编译语句，比如我在编译的时候想要链接某个指定的库文件，或者添加某些指定的编译选项，此时隐式规则就显得笨拙。

针对第一点，有时我们需要明确地告诉 Makefile 不要对某个目标运用隐式规则，比如我们每次想要清理工程项目中所有的目标文件，可以将清理工作交给 Makefile 来完成：

vincent@ubuntu:~$ **cat Makefile -n**

```
1  OBJ = a.o b.o x.o y.o
2
3  image:$(OBJ)
4      $(CC) $(OBJ) -o image
5
6  clean:
7      $(RM) $(OBJ) image
8
9  .PHONY: clean
```

第 6、7 行声明了一个清理目标文件和 image 的规则，执行这条 SHELL 命令时需要指定 make 的参数 clean，clean 是一个动作的代号，而不是一个我们要生成的文件，但是根据隐式规则，假如当前目录恰巧有个文件叫做 clean.c，就可能会导致 Makefile 自动生成其对应的编译语句，从而引起混淆。在第 9 行中用指示符.PHONY 来明确地告诉 Makefile 不要对 clean 运用任何隐式规则，事实上，不能运用隐式规则的目标被称为伪目标。

第二，静态规则。

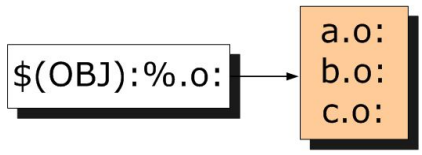
针对第二点，我们也许在编译.o 文件的时候需要一些特殊的编译选项，不能完全将他们弃之不管，但是又不想对每一个.o 文件写一个规则，那就可以使用静态规则：

vincent@ubuntu:~$ **cat Makefile -n**

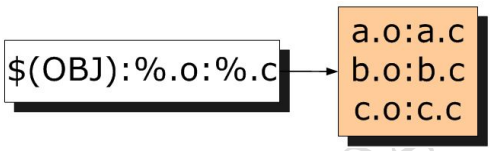
```
1  OBJ = a.o b.o x.o y.o
2
```

```
3  image:$(OBJ)
4      $(CC) $(OBJ) -o image
5
6  $(OBJ):%.o:%.c
7      $(CC) $^ -o $@ -Wall -c
8
9  clean:
10     $(RM) $(OBJ) image
11
12 .PHONY: clean
```

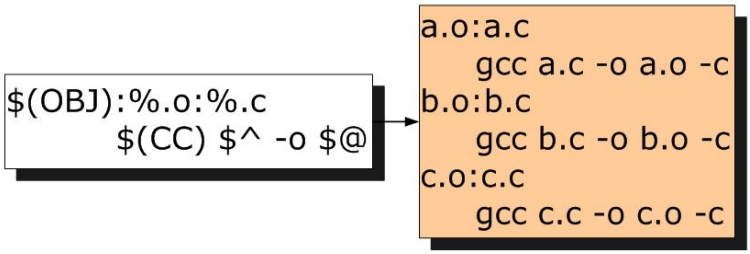
第 6、7 行运用了所谓的静态规则，其工作原理是：$(OBJ)被称为原始列表，即（a.o b.o x.o y.o），紧跟在其后的%.o 被称为匹配模式，含义是在原始列表中按照这种指定的模式挑选出能匹配得上的单词（在本例中要找出原始列表里所有以.o 为后缀的文件）作为规则的目标，这个过程用下图演示：



简单地讲，就是用一个规则来生成一系列的目标文件。接着，第二个冒号后面的内容就是目标对应的依赖，%可以理解为通配符，因此本例中%.o:%.c 的意思就是：每一个匹配出来的目标所对应的依赖文件是同名的.c 文件，这个过程也用图演示如下：



可见，静态规则的目的就是用一句话来自动生成很多目标及其依赖，接下来要针对每一对目标-依赖生成对应的编译语句：



此处可见自动化变量的用武之地了，因为每一对目标-依赖对的名字都不一样，因此在静态规则中不可能直接把名字写死，而要用自动化变量来自动调整为对应的名字。

总结一下，静态模式规则是这样一个规则：规则存在多个目标，并且不同的目标可以根据目标文件的名字来自动构造出依赖文件。静态模式规则比多目标规则更通用，它不需要多个目标具有相同的依赖。但是静态模式规则中的依赖文件必须是相类似的而不是完全相同的。

第三，多目标规则。

上面的 Makefile 除了可以使用静态规则，针对我们要生成的三个.o 文件，也可以使用所谓的多目标规则，具体（其中函数$(subst)的用法和功能请参阅下面有关“函数”的小节）如下：

vincent@ubuntu:~$ **cat Makefile -n**

```
 1  SRC = $(wildcard *.c)
 2  OBJ = $(SRC:%.c=%.o)
 3
 4  image:$(OBJ)
 5      $(CC) $(OBJ) -o image -lgcc
 6
 7  $(OBJ):$(SRC)
 8      $(CC) $(subst .o,.c,$@) -o $@ -c
 9
10  clean:
11      $(RM) $(OBJ) image
12
13  .PHONY:clean
```

着重看第 7、8 行，展开后是：

```
 7  a.o b.o x.o y.o:a.c b.c x.c y.c
 8      $(CC) $(subst .o,.c,$@) -o $@ -c
```

当中的四个.o文件都是这个规则的目标，规则所定义的命令对所有的目标有效。这个具有多目标的规则相当于多个规则，规则中命令对不同的目标的执行效果不同，因为在规则的命令中可能使用自动环变量”$@”，多目标规则意味着所有的目标具有相同的依赖文件。

比如，当目标是 a.o 时，多目标规则将自动构建如下针对 a.o 的规则：

```
a.o:a.c b.c x.c y.c
    $(CC) $(subst .o,.c,$@) -o $@ -c
```

即：

```
a.o:a.c b.c x.c y.c
    $(CC) a.c -o a.o -c
```

可以看到，在这个范例中使用多目标规则是比较笨拙的，因为他把所有的源文件都当成是 a.o 的依赖文件了，因为多目标规则不能根据目标来自动改变依赖文件，要做到这一点可以使用上面的静态规则。

一般而言，多目标规则应用在以下两种场合：

1，只需描述依赖关系，而不需要指定相关 SHELL 命令。比如当前的所有的目标文件都依赖于一个叫 head.h 的头文件，可以用多目标规则来表达：

```
a.o b.o x.o y.o:head.h
```

这样只要 head.h 有改动，四个目标文件都将会被重新编译。

一个只描述依赖关系的规则，用来管理工程项目当中一些各自和公共文件会非常方便，

比如工程当中有许多.o 文件，用变量 OBJS 来表达，他们都依赖于 config.h 文件，而他们又可以各自依赖于其他的头文件：

```
OBJS = a.o b.o c.o
a.o:a.h
b.o:b.h B.h
c.o:c.h
$(OBJS):config.h
```

这样做的好处是：我们可以在源文件中增加或者删除了包含的头文件以后不用修改已经存在的 Makefile 的规则，只需要增加或者删除某一个.o 文件依赖的头文件。这种方式很简单也很方便。对于一个大的工程来说，这样做的好处是显而易见的。在一个大的工程中，对于一个单独目录下的.o 文件的依赖规则建议使用此方式。

2，有多个具有类似构建命令的目标，就是上面的例子那样，将之展开得到：

```
a.o:a.c b.c x.c y.c
    $(CC) a.c -o a.o -c
b.o:a.c b.c x.c y.c
    $(CC) b.c -o b.o -c
x.o:a.c b.c x.c y.c
    $(CC) x.c -o x.o -c
y.o:a.c b.c x.c y.c
    $(CC) y.c -o y.o -c
```

第四，双冒号规则。

双冒号规则就是使用“::”代替普通规则的“:”得到的规则。当同一个文件作为多个规则的目标时，双冒号规则的处理和普通规则的处理过程完全不同（双冒号规则允许在多个规则中为同一个目标指定不同的重建目标的命令）。

首先需要明确的是：Makefile中，一个目标可以出现在多个规则中。但是这些规则必须是同一种规则，要么都是普通规则，要么都是双冒号规则。而不允许一个目标同时出现在两种不同的规则中。双冒号规则和普通规则的处理的不同点表现在以下几个方面：

1， 双冒号规则中，当依赖文件比目标更新时。规则将会被执行。对于一个没有依赖而只有命令行的双冒号规则，当引用此目标时，规则的命令将会被无条件执行。而普通规则，当规则的目标文件存在时，此规则的命令永远不会被执行（目标文件永远是最新的）。

2，当同一个文件作为多个双冒号规则的目标时。这些不同的规则会被独立的处理，而不是像普通规则那样合并所有的依赖到一个目标文件。这就意味着对这些规则的处理就像多个不同的普通规则一样。就是说多个双冒号规则中的每一个的依赖文件被改变之后，make只执行此规则定义的命令，而其它的以这个文件作为目标的双冒号规则将不会被执行。

看一个例子：

```
vincent@ubuntu:~$ ls
a.c   b.c   libx.so   liby.so   Makefile
vincent@ubuntu:~$ cat Makefile -n
```

```
1  image::b.c
2      $(CC) a.c -o $@ -L. -lx
3
4  image::b.c
5      $(CC) b.c -o $@ -L. -ly
vincent@ubuntu:~$ make
cc a.c -o image
cc b.c -o image
```

范例中，不管a.c或者b.c都生成image文件，如果“a.c”文件被修改，执行make以后将根据“a.c”文件重建目标“image”。而如果“b.c”被修改那么“image”将根据“b.c”被重建。如果以上两个规则为普通规则，会出现什么情况？

当同一个目标出现在多个双冒号规则中时，规则的执行顺序和普通规则的执行顺序一样，按照其在 Makefile 中的书写顺序执行。GNU make 的双冒号规则给我们提供一种根据依赖的更新情况而执行不同的命令来重建同一目标的机制。

六 条件判断

之前提到，我们的工程文件可能在 PC 端编译，也可能在 ARM 平台运行，不同的编译环境需要使用不同的工具链，我们可以通过手工改动 Makefile 的方式来达到更改编译器的目的，也可以使用条件判断机制让 Makefile 自动处理：

```
vincent@ubuntu:~$ cat Makefile -n
1  OBJ = a.o b.o x.o y.o
2
3  ifdef TOOLCHAIN    # ifdef 语句用来判断变量 TOOLCHAIN 是否有定义
4      CC = $(TOOLCHAIN)
5  else
6      CC = gcc
7  endif
8
9  image:$(OBJ)
10     $(CC) $(OBJ) -o image
11
12 $(OBJ):%.o:%.c
13     $(CC) $^ -o $@ -c
14
15 clean:
16     $(RM) $(OBJ) image
17
18 .PHONY:clean
vincent@ubuntu:~$ make TOOLCHAIN=arm-linux-gnu-gcc
arm-linux-gnu-gcc a.c -o a.o -c
arm-linux-gnu-gcc b.c -o b.o -c
arm-linux-gnu-gcc x.c -o x.o -c
```

```
arm-linux-gnu-gcc y.c -o y.o -c
arm-linux-gnu-gcc a.o b.o x.o y.o -o image
```

在 Makefile 中增加了对变量 TOOLCHAIN 的判断，用来选择用户所指定的工具链，如果用户如上述代码所示，在执行 make 时指定了参数 TOOLCHAIN=arm-linux-gcc，那第 3 行的 ifdef 语句将成立，因此编译器 CC 被调整为用户指定的 TOOLCHAIN。在这个例子中我们同时也看到了如何在命令行中给 make 传递参数。

再进一步，假如在用户使用 gcc 编译时需要链接库文件 libgcc.so，而在使用交叉工具链 arm-linux-gnu-gcc 时不需要，那么我们的 Makefile 需要再改成：

```
vincent@ubuntu:~$ cat Makefile -n
     1  OBJ = a.o b.o x.o y.o
     2
     3  ifdef TOOLCHAIN
     4      CC = $(TOOLCHAIN)
     5  else
     6      CC = gcc
     7  endif
     8
     9  image:$(OBJ)
    10  ifeq ($(CC), gcc)    # ifeq ( )用来判断变量 CC 的值是否等于 gcc
    11      $(CC) $(OBJ) -o image -lgcc
    12  else
    13      $(CC) $(OBJ) -o image
    14  endif
    15
    16  $(OBJ):%.o:%.c
    17      $(CC) $^ -o $@ -c
    18
    19  clean:
    20      $(RM) $(OBJ) image
    21
    22  .PHONY:clean
```

在第 10 行中，使用 ifeq ( )来对 CC 进行了判断，注意：ifeq 跟后面的圆括号之间有一个空格！ifeq ( )也可以用来判断一个变量是否为空，例如：

```
ifeq ($(A),)
    echo "$(A) is empty"
endif
```

七 函数

上述 Makefile 乍看上去已经像模像样了，毕竟已经隐约出现了一点乱码的影子，显得非常高大上，但是其实还有一个大问题没有解决：假如某一天需要将 c.c 添加进工程中一起

编译，由于 Makefile 里面没有体现 c.c 文件，因此没办法对该文件进行处理，如果又要手工来添加的话，显然很麻烦，正确的做法是：使用 Makefile 提供的内嵌函数来帮我们自动地搜寻所需的文件，还可以利用这些函数帮我们对字符串进行各种处理。

怎样让 Makefile 知道我们的工程来了一个新的文件 c.c 呢？这需要一个叫 wildcard 的函数帮忙：

```
SRC = $(wildcard *.c)
```

注意到在 Makefile 中书写一个函数的格式：$(function arg1,arg2,arg3, ...) 其中 function 是函数的名字，后面跟一个空格，然后是参数列表，如果有多个参数则用逗号隔开（注意逗号后面最好不要有空格），整个函数用$( )包裹起来（跟变量一样）。

由于 wildcard 函数的作用就是找到参数匹配的文件名，因此该语句的作用就相当于：

```
SRC = a.c b.c c.c x.c y.c
```

有了源程序文件名字列表，通过变量的替换操作，很容易就可以得到.o 文件列表：

```
OBJ = $(SRC: %.c=%.o)
```

于是，我们的 Makefile 又进化成了：

```
vincent@ubuntu:~$ cat Makefile -n
     1  SRC = $(wildcard *.c)
     2  OBJ = $(SRC:%.c=%.o)
     3
     4  ifdef TOOLCHAIN
     5      CC = $(TOOLCHAIN)
     6  else
     7      CC = gcc
     8  endif
     9
    10  image:$(OBJ)
    11  ifeq ($(CC),gcc)
    12      $(CC) $(OBJ) -o image -lgcc
    13  else
    14      $(CC) $(OBJ) -o image
    15  endif
    16
    17  $(OBJ):%.o:%.c
    18      $(CC) $^ -o $@ -c
    19
    20  clean:
    21      $(RM) $(OBJ) image
    22
    23  .PHONY:clean
```

下面列出 Makefile 中常用到的内嵌函数的详细信息，以供查阅。

第一类：文本处理函数。此类函数专门用于处理文本（字符串）：

1，$(subst FROM,TO,TEXT)

功能：

将字符串 TEXT 中的字符 FROM 替换为 TO。

返回：

替换之后的新字符串。

范例：

A = $(subst pp,PP,apple tree)

替换之后变量 A 的值是”aPPle tree”

2，$(patsubst PATTERN,REPLACEMENT,TEXT)

功能：

按照 PATTERN 搜索 TEXT 中所有以空格隔开的单词，并将它们替换为 REPLACEMENT。注意：参数 PATTERN 可以使用模式通配符%来代表一个单词中的若干字符，如果此时 REPLACEMENT 中也出现%，那么 REPLACEMENT 中的%跟 PATTERN 中的%是一样的。

返回：

替换之后的新字符串。

范例：

A = $(patsubst %.c,%.o,a.c b.c)

替换之后变量 A 的值是”a.o b.o”

3，$(strip STRING)

功能：

去掉字符串中开头和结尾的多余的空白符（掐头去尾），并将其中连续的多个空白符合并为一个。注意：所谓的空白符指的是空格、制表符。

返回：

去掉多余空白符之后的新字符串。

范例：

A = $(strip apple tree)

处理之后，变量 A 的值是”apple tree”

4，$(findstring FIND, STRING)

功能：

在给定的字符串 STRING 中查找 FIND 子串。

返回：

找到则返回 FIND，否额返回空。

范例：

A = $(findstring pp, apple tree)

B = $(findstring xx, apple tree)

变量 A 的值是”pp”，变量 B 的值是空。

5，$(filter PATTERN,TEXT)

功能：

过滤掉 TEXT 中所有不符合给定模式 PATTERN 的单词。其中 PATTERN 可以是多个模式的组合。

返回：

TEXT 中所有符合模式组合 PATTERN 的单词组成的子串。

范例：

A = a.c b.o c.s d.txt

B = $(filter %.c %.o,$(A))

过滤后变量 B 的值是”a.c b.o”。

6，$(filter-out PATTERN,TEXT)

功能：

过滤掉 TEXT 中所有符合给定模式 PATTERN 的单词，与函数 filter 功能相反。

返回：

TEXT 中所有不符合模式组合 PATTERN 的单词组成的子串。

范例：

A = a.c b.o c.s d.txt

B = $(filter %.c %.o,$(A))

过滤后变量 B 的值是”c.s d.txt”。

7，$(sort LIST)

功能：

将字符串 LIST 中的单词按字母升序的顺序排序，并且去掉重复的单词。

返回：

排完序且没有重复单词的新字符串。

范例：

A = foo bar lose foo ugh

B = $(sort $(A))

处理后变量 B 的值是”bar foo lose ugh”。

8，$(word N,TEXT)

功能：

取字符串 TEXT 中的第 N 个单词。注意，N 必须为正整数。

返回：

第 N 个单词（如果 N 大于 TEXT 中单词的总数则返回空）。

范例：

A = an apple tree

B = $(word 2 $(A))

处理后变量 B 的值是”apple”。

9，$(wordlist START,END,TEXT)

功能：

取字符串 TEXT 中介于 START 和 END 之间的子串。

返回：

介于 START 和 END 之间的子串（如果 START 大于 TEXT 中单词的总数或者 START 大于 END 时返回空，否则如果 END 大于 TEXT 中单词的总数则返回从 START 开始到 TEXT 的最后一个单词的子串）。

范例：

A = the apple tree is over 5 meters tall

B = $(wordlist 4,100,$(A))

处理后变量 B 的值是”is over 5 meters tall”。

10，$(words TEXT)

功能：

计算字符串 TEXT 的单词数。

返回：

字符串 TEXT 的单词数。

范例：

A = the apple tree is over 5 meters tall

B = $(words $(A))

处理后变量 B 的值是”8”。

11，$(firstword TEXT)

功能：

取字符串 TEXT 中的第一个单词。相当于$(word 1 TEXT)

返回：

字符串 TEXT 的第一个单词。

范例：

A = the apple tree is over 5 meters tall

B = $(firstword $(A))

处理后变量 B 的值是”the”。

以上11个函数是make内嵌的的文本处理函数。在书写Makefile时可搭配使用，来实现复杂功能。GNU make除上一节所介绍的内嵌的文本处理函数之外，还存在一些针对于文件名的处理函数。这些函数主要用来对一系列空格分割的文件名进行转换，这些函数的参数被作为若干个文件名来对待，函数对这样的一组文件名按照一定方式进行处理，并返回以空格分隔的多个文件名序列。

他们是：

1，$(dir NAMES)

功能：

取文件列表 NAMES 中每一个路径的目录部分。

返回：

每一个路径的目录部分组成的新的字符串。

范例：

A = /etc/init.d /home/vincent/.bashrc /usr/bin/man

B = $(dir $(A))

处理后变量 B 的值是”/etc/ /home/vincent/ /usr/bin/”。

2，$(notdir NAMES)

功能：

取文件列表 NAMES 中每一个路径的文件名部分。

返回：

每一个路径的文件名部分组成的新的字符串。注意：如果NAMES中存在不包含斜线的文件名，则不改变这个文件名，而以反斜线结尾的文件名，用空串代替。

范例：

A = /etc/init.d /home/vincent/.bashrc /usr/bin/man

B = $(dir $(A))

处理后变量 B 的值是”init.d .bashrc man”。

3，$(suffix NAMES)

功能：

取文件列表 NAMES 中每一个路径的文件的后缀部分。后缀指的是最后一个.后面的子串。

返回：

每一个路径的文件名的后缀部分组成的新的字符串。

范例：

A = /etc/init.d /home/vincent/.bashrc /usr/bin/man

B = $(suffix $(A))

处理后变量 B 的值是”.d .bashrc”。

4，$(basename NAMES)

功能：

取文件列表 NAMES 中每一个路径的文件的前缀部分。前缀指的是最后一个.后面除了后缀的子串。

返回：

每一个路径的文件名的前缀部分组成的新的字符串。

范例：

A = /etc/init.d /home/vincent/.bashrc /usr/bin/man

B = $(basename $(A))

处理后变量 B 的值是”/etc/init /home/vincent/ /usr/bin/man”。

5，$(addsuffix SUFFIX,NAMES)

功能：

为文件列表 NAMES 中每一个路径的文件名添加后缀 SUFFIX。

返回：

添加了后缀 SUFFIX 的字符串。

范例：

A = /etc/init.d /home/vincent/.bashrc /usr/bin/man

B = $(addsuffix .bk,$(A))

处理后 B 为”/etc/init.d.bk /home/vincent/.bashrc.bk /usr/bin/man.bk”。

6，$(addprefix PREFIX,NAMES)

功能：

为文件列表 NAMES 中每一个路径的文件名添加前缀 PREFIX。

返回：

添加了前缀 PREFIX 的字符串。

范例：

A = /etc/init.d /home/vincent/.bashrc /usr/bin/man

B = $(addprefix host:,$(A))

处理后 B 的值为：

”host:/etc/init.d host:/home/vincent/.bashrc host:/usr/bin/man”。

7，$(wildcard PATTERN)

功能：

获取匹配模式为 PATTERN 的文件名。

返回：

匹配模式为 PATTERN 的文件名。

范例：

A = $(wildcard *.c)

假设当前路径下有两个.c 文件 a.c 和 b.c，则处理后 A 的值为：”a.c b.c”。

8，$(foreach VAR,LIST,TEXT)

功能：

首先展开变量“VAR”和“LIST”，而表达式“TEXT”中的变量引用不被展开。执行时把“LIST”中使用空格分割的单词依次取出赋值给变量“VAR”，然后执行“TEXT”表达式，重复直到“LIST”的最后一个单词（为空时结束）。

它是一个循环函数，类似于Linux的shell中的循环。注意：由于“TEXT”中的变量或者函数引用在执行时才被展开，因此如果在“TEXT”中存在对“VAR”的引用，那么“VAR”的值在每一次展开式将会到的不同的值。

返回：

以空格分隔的多次表达式“TEXT”的计算的结果。

范例：

假设当前目录下有两个子目录dir1/和dir2/，先要将他们里面的所有文件赋值给变量FILES，可以这么写：

```
vincent@ubuntu:~$ tree
.
├── dir1
│   ├── file1
│   └── file2
├── dir2
│   ├── a.c
│   └── b.c
└── Makefile
vincent@ubuntu:~$ cat Makefile -n
```

```
1  DIR = dir1 dir2
2  FILES = $(foreach dir,$(DIR),$(wildcard $(dir)/*))
3
4  all:
5      echo $(FILES)
```

```
vincent@ubuntu:~$ make -s
dir1/file1 dir1/file2 dir2/a.c dir2/b.c
```

9，$(if CONDITION,THEN-PART[,ELSE-PART])

功能：

判断 CONDITION 是否为空，如果非空则执行 THEN-PART 且将结果作为函数的返回值，否则如果为空则执行 ELSE-PART 且将结果作为函数的返回值，如果此时没有 ELSE-PART 则函数返回空。

返回：

根据 CONDITION 返回 THEN-PART 或者 ELSE-PART 的执行结果。

范例：

install-dir := $(if $(INSTALL_DIR),$(INSTALL_DIR),extra)

先判断 INSTALL_DIR 是否为空，如果为空则将 extra 赋值给 install-dir，否则如果不为空，则 install-dir 的值等于$(INSTALL_DIR)（该范例摘选自 LINUX-3.9.8 源码顶层 Makefile）。

10，$(call VAR,ARGS,...)

功能：

执行 VAR，并将 ARGS 一一对应地替换 VAR 里面的$(1)、$(2)……。因此函数$(call)被称为是 Makefile 中唯一一个创建定制参数的函数。

返回：

将 ARGS 替换 VAR 中的$(1)、$(2)……之后 VAR 的执行结果。

范例 1：

A = my name is $(1) $(2)

B = $(call A Michael,Jackson)

将 Michael,Jackson 分别替换变量 A 里面的$(1)和$(2)，于是 B 的值就是 my name is Michael Jackson。

范例2：

使用Makefile的命令，找出指定系统SHELL指令的完整路径（类似which的功能）。

1，使用subst将环境变量PATH中每一个路径的分隔符冒号替换成空格：

A = $(subst :, ,$(PATH))

2，将指定的SHELL指令添加到每一个可能的路径后面：

B = $(addsuffix /$(1),$(A))

3，使用wildcard匹配所有正确的路径：

C = $(wildcard $(B))

将上述命令组合起来就能完成类似命令which的功能，暂且叫他为WHICH：

WHICH = $(wildcard $(addsuffix /$(1),$(subst :, ,$(PATH))))

注意：此处WHICH的定义只能是这样递归定义方式，而不能是直接定义方式。

最后，使用call来给这个复杂的变量传递一个定制化的参数$(1)，比如要获得系统命令ps的完整路径：

```
$(call WHICH,ps)
```

11，$(origin VAR)

功能：

顾名思义，该函数用来查看参数 VAR 的出处。

返回：

参数 VAR 的出处，有如下几种情况：

1，undefined

表示变量 VAR 尚未被定义。

2，default

表示变量 VAR 是一个默认的内嵌变量，比如 CC、MAKEFLAGS 等。

3，environment

表示变量 VAR 是一个系统环境变量，比如 PATH。

4，file

表示变量 VAR 在另一个 Makefile 中被定义。

5，command line

表示变量 VAR 是一个在命令行定义的变量。

6，override

表示变量 VAR 在本 Makefile 定义并使用了 override 指示符。

7，automatic

表示变量 VAR 是一个自动化变量，比如@、^等等。

范例：

```
ifeq (“$(origin V)”, “command line”)
    KBUILD_VERBOSE = $(V)
endif
```

判断变量 V 的出处，如果该变量来自命令行，则将 KBUILD_VERBOSE 赋为 V 的值（该范例摘选自 LINUX-3.9.8 源码顶层 Makefile）。

12，$(shell COMMANDS)

功能：

在 Makefile 中执行 COMMANDS，此处的 COMMANDS 是一个或几个 SHELL 命令，功能与在 SHELL 脚本中使用`COMMANDS`的效果相同。该函数返回这些命令的最终结果。

返回：

返回 COMMANDS 的执行结果，并把其中的回车符替换成空格符。

范例：

```
contents := $(shell cat file.txt)
```

使用cat命令显示file.txt的内容，并将其中的回车符替换为空格之后赋给contents。注意到此处用的是直接定义方式而不是递归定义方式，这是为了防止后续再有对此变量的引用就不会有展开过程。这样可以防止规则命令行中的变量引用在命令行执行时展开的情况发生（因为展开“shell”函数需要另外的shell进程完成，影响命令的执行效率）。

八 实用 make 选项集锦

1，指定要执行的 Makefile 文件：

make -f altmake

make --file altmake

make --makefile altmake

以上三种方式都可以用来执行一个普通命令的文件作为 Makefile 文件。在缺省的情况下不指定任何 Makefile 文件，则 make 会在当前目录下依次查找命名为 GNUmakefile 和 Makefile 以及 makefile 的文件。

2，指定终极目标：

make TARGET

所谓的终极目标指的是 Makefile 中第一个出现的规则中的第一个目标（详细解释请参见 1.5.3 小节），是缺省的整个工程或者程序编译过程的总的规则和目的。如果想要执行除该目标之外的其他普通目标位编译的最终目的，则可以在执行 make 的同时指定。

在我们需要对程序的一部分进行编译，或者仅仅对某几个程序进行编译而不是完整地编译整个工程的时候，指定终极目标就很有用。

3，强制重建所有规则中目标：

make -B

make -always-make

4，指定 Makefile 的所在路径：

make -C dir/

make --directory=dir/

假如要执行的 Makefile 文件不在当前目录，可以使用该选项指定。这个选项一般用在一个 Makefile 内部调用另一个子 Makefile 的场景（详见 1.5.4 小节中关于特殊变量的部分）。