MakeFile详解

什么是 Makefile 呢?

Makefile 可以简单的认为是一个工程文件的编译规则，描述了整个工程的编译和链接等规则。其中包含了那些文件需要编译，那些文件不需要编译，那些文件需要先编译，那些文件需要后编译，那些文件需要重建等等。编译整个工程需要涉及到的，在 Makefile 中都可以进行描述。换句话说，Makefile 可以使得我们的项目工程的编译变得自动化，不需要每次都手动输入一堆源文件和参数。

以 Linux 下的C语言开发为例来具体说明一下，多文件编译生成一个文件，编译的命令如下所示：

```
gcc -o outfile name1.c name2.c ...
```

outfile 要生成的可执行程序的名字，nameN.c 是源文件的名字。这是我们在 Linux 下使用 gcc 编译器编译 C 文件的例子。如果我们遇到的源文件的数量不是很多的话，可以选择这样的编译方式。如果源文件非常的多的话，就会遇到下面的这些问题。

1. 编译的时候需要链接库的的问题。拿C语言来说，编译的时候 gcc 只会默认链接一些基本的C语言标准库，很多源文件依赖的标准库都需要我们手动链接。

下面列举了一些需要我们手动链接的标准库：

- name1.c 用到了数学计算库 math 中的函数，我们得手动添加参数 -lm；
- name4.c 用到了小型数据库 SQLite 中的函数，我们得手动添加参数 -lsqlite3；
- name5.c 使用到了线程，我们需要去手动添加参数 -lpthread。

因为有很多的文件，还要去链接很多的第三方库。所以在编译的时候命令会很长，并且在编译的时候我们可能会涉及到文件链接的顺序问题，所以手动编译会很麻烦。

如果我们学会使用 Makefile 就不一样了，它会彻底简化编译的操作。把要链接的库文件放在 Makefile 中，制定相应的规则和对应的链接顺序。这样只需要执行 make 命令，工程就会自动编译。每次想要编译工程的时候就执行 make ，省略掉手动编译中的参数选项和命令，非常的方便。

2. 编译大的工程会花费很长的时间。

如果我们去做项目开发，免不了要去修改工程项目的源文件，每次修改后都要去重新编译。一个大的工程项目可不止有几个的源文件，里面的源文件个数可能有成百上千个。例如一个内核，或者是一个软件的源码包。这些都是我们做开发经常会遇到的。要完成这样的文件的编译，我们消耗的时间可不是一点点。如果文件特别大的话我们可能要花上半天的时间。

对于这样的问题我们 Makefile 可以解决吗？当然是可以的，Makefile 支持多线程并发操作，会极大的缩短我们的编译时间，并且当我们修改了源文件之后，编译整个工程的时候，make 命令只会编译我们修改过的文件，没有修改的文件不用重新编译，也极大的解决了我们耗费时间的问题。

这其实是我们遇到的比较常见的问题，当然可能遇到的问题还会有很多，比如：工程文件中的源文件的类型很多，编译的话需要选择的编译器；文件可能会分布在不同的目录中，使用时需要调价路径。这些问题都可以通过 Makefile 解决。并且文件中的 Makefile 只需要完成一次，一般我们只要不增加或者是删除工程中的文件，Makefile 基本上不用去修改，编译时只用一个 make 命令。为我们提供了极大的便利，很大程度上提高编译的效率。

Makefile文件中包含哪些规则?

想要书写一个完整的 Makefile文件，需要了解 Makefile 的相关的书写规则。我们已经知道了 Makefile 描述的是文件编译的相关规则，它的规则主要是两个部分组成，分别是依赖的关系和执行的命令，其结构如下所示：

```
targets : prerequisites
 command
```

或者是

```
targets : prerequisites; command
 command
```

相关说明如下：

- targets：规则的目标，可以是 Object File（一般称它为中间文件），也可以是可执行文件，还可以是一个标签；
- prerequisites：是我们的依赖文件，要生成 targets 需要的文件或者是目标。可以是多个，也可以是没有；
- command：make 需要执行的命令（任意的 shell 命令）。可以有多条命令，每一条命令占一行。

注意：我们的目标和依赖文件之间要使用冒号分隔开，命令的开始一定要使用`Tab`键。

通过下面的例子来具体使用一下 Makefile 的规则，Makefile文件中添代码如下：

```
test:test.c
   gcc -o test test.c
```

上述代码实现的功能就是编译 test.c 文件，通过这个实例可以详细的说明 Makefile 的具体的使用。其中 test 是的目标文件，也是我们的最终生成的可执行文件。依赖文件就是 test.c 源文件，重建目标文件需要执行的操作是`gcc -o test test.c`。这就是 Makefile 的基本的语法规则的使用。使用 Makefile 的方式：首先需要编写好 Makefile 文件，然后在 shell 中执行 make 命令，程序就会自动执行，得到最终的目标文件。

通过上面的例子我们可以了解到，Makefile 的规则很简单，但这并不是 Makefile 的全部，这个仅仅是它的冰山一角。仅仅靠一个规则满足不了我们对于大的工程项目的编译。甚至几个文件的编译都会出现问题，所以要学习的东西还有很多。

简单的概括一下Makefile 中的内容，它主要包含有五个部分，分别是：

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 命名有自动推导的功能，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 命令所支持的。

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：“#”。

Makefile的工作流程

Makefile 的具体工作流程可以通过例子来看一下：创建一个包含有多个源文件和 Makefile 的目录文件，源文件之间相互关联。在 Makefile 中添加下面的代码：

```
1. main:main.o test1.o test2.o
2. gcc main.o test1.o test2.o -o main
3. main.o:main.c test.h
4. gcc -c main.c -o main.o
5. test1.o:test1.c test.h
6. gcc -c test1.c -o test1.o
7. test2.o:test2.c test.h
8. gcc -c test2.c -o test2.o
```

在我们编译项目文件的时候，默认情况下，make 执行的是 Makefile 中的第一规则（Makefile 中出现的第一个依赖关系），此规则的第一目标称之为“最终目标”或者是“终极目标”。

在 shell 命令行执行的 make 命令，就可以得到可执行文件 main 和中间文件 main.o、test1.o 和 test2.o，main 就是我们要生成的最终文件。通过 Makefile 我们可以发现，目标 main"在 Makefile 中是第一个目标，因此它就是 make 的终极目标，当修改过任何 C 文件后，执行 make 将会重建终极目标 main。

它的具体工作顺序是：当在 shell 提示符下输入 make 命令以后。 make 读取当前目录下的 Makefile 文件，并将 Makefile 文件中的第一个目标作为其执行的“终极目标”，开始处理第一个规则（终极目标所在的规则）。在我们的例子中，第一个规则就是目标 "main" 所在的规则。规则描述了 "main" 的依赖关系，并定义了链接 ".o" 文件生成目标 "main" 的命令；make 在执行这个规则所定义的命令之前，首先处理目标 "main" 的所有的依赖文件（例子中的那些 ".o" 文件）的更新规则（以这些 ".o" 文件为目标的规则）。

对这些 ".o" 文件为目标的规则处理有下列三种情况：

- 目标 ".o" 文件不存在，使用其描述规则创建它；
- 目标 ".o" 文件存在，目标 ".o" 文件所依赖的 ".c" 源文件 ".h" 文件中的任何一个比目标 ".o" 文件“更新”（在上一次 make 之后被修改）。则根据规则重新编译生成它；
- 目标 ".o" 文件存在，目标 ".o" 文件比它的任何一个依赖文件（".c" 源文件、".h" 文件）“更新”（它的依赖文件在上一次 make 之后没有被修改），则什么也不做。

通过上面的更新规则我们可以了解到中间文件的作用，也就是编译时生成的 ".o" 文件。作用是检查某个源文件是不是进行过修改，最终目标文件是不是需要重建。我们执行 make 命令时，只有修改过的源文件或者是不存在的目标文件会进行重建，而那些没有改变的文件不用重新编译，这样在很大程度上节省时间，提高编程效率。小的工程项目可能体会不到，项目工程文件越大，效果才越明显。

当然 make 命令能否顺利的执行，还在于我们是否制定了正确的的依赖规则，当前目录下是不是存在需要的依赖文件，只要任意一点不满足，我们在执行 make 的时候就会出错。所以完成一个正确的 Makefile 不是一件简单的事情。

清除工作目录中的过程文件

我们在使用的时候会产生中间文件会让整个文件看起来很乱，所以在编写 Makefile 文件的时候会在末尾加上这样的规则语句：

.PHONY:clean

clean:

rm -rf *.o test

其中 "*.o" 是执行过程中产生的中间文件，"test" 是最终生成的执行文件。我们可以看到 clean 是独立的，它只是一个伪目标（在《Makefile伪目标》的章节中详细介绍），不是具体的文件。不会与第一个目标文件相关联，所以我们在执行 make 的时候也不会执行下面的命令。在shell 中执行 "make clean" 命令，编译时的中间文件和生成的最终目标文件都会被清除，方便我们下次的使用。

Makefile通配符的使用

Makefile 是可以使用 shell 命令的，所以 shell 支持的通配符在 Makefile 中也是同样适用的。 shell 中使用的通配符有："*"，"?"，"[...]"。具体看一下这些通配符的表示含义和具体的使用方法。

<style> td {white-space:pre-wrap;border:1px solid #dee0e3;}</style> <byte-sheet-html-origin data-id="1hcB1LqrnX-1623856826597" data-version="3" data-is-embed="true"><colgroup><col width="105"><col width="105"></colgroup>
关于自动化变量可以理解为由 Makefile 自动产生的变量。在模式规则中，规则的目标和依赖的文件名代表了一类的文件。规则的命令是对所有这一类文件的描述。我们在 Makefile 中描述规则时，依赖文件和目标文件是变动的，显然在命令中不能出现具体的文件名称，否则模式规则将失去意义。	使用说明
	匹配0个或者是任意个字符
那么模式规则命令中该如何表示文件呢? 就需要使用“自动化变量”，自动化变量的取值根据执行的规则来决定，取决于执行规则的目标文件和依赖文件。下面是对所有的自动化变量进行的说明：	匹配任意一个字符
[]	我们可以指定匹配的字符放在 "[]" 中 |</byte-sheet-html-origin>

通配符可以出现在模式的规则中，也可以出现在命令中，详细的使用情况如下。

实例 1：

```
.PHONY:clean
clean:
    rm -rf *.o test
```

这是在 Makefile 中经常使用的规则语句。这个实例可以说明通配符可以使用在规则的命令当中，表示的是任意的以 .o 结尾的文件。

实例 2：

```
test:*.c
    gcc -o $@ $^
```

这个实例可以说明我们的通配符不仅可以使用在规则的命令中，还可以使用在规则中。用来表示生所有的以 .c 结尾的文件。

但是如果我们的通配符使用在依赖的规则中的话一定要注意这个问题：不能通过引用变量的方式来使用，如下所示。

```
OBJ=*.c
test:$(OBJ)
    gcc -o $@ $^
```

我们去执行这个命令的时候会出现错误，提示我们没有 *".c" 文件，实例中我们想要表示的是当前目录下所有的 ".c" 文件，但是我们在使用的时候并没有展开，而是直接识别成了一个文件。文件名是 "*.c"。

如果我们就是相要通过引用变量的话，我们要使用一个函数 "wildcard"，这个函数在我们引用变量的时候，会帮我们展开。我们把上面的代码修改一下就可以使用了。

```
OBJ=$(wildcard *.c)
test:$(OBJ)
    gcc -o $@ $^
```

这样我们再去使用的时候就可以了。调用函数的时候，会帮我们自动展开函数。

还有一个和通配符 "*" 相类似的字符，这个字符是 "%"，也是匹配任意个字符，使用在我们的的规则当中。

```
test:test.o test1.o
    gcc -o $@ $^
%.o:%.c
    gcc -o $@ $^
```

"%.o" 把我们需要的所有的 ".o" 文件组合成为一个列表，从列表中挨个取出的每一个文件，"%" 表示取出来文件的文件名（不包含后缀），然后找到文件中和 "%"名称相同的 ".c" 文件，然后执行下面的命令，直到列表中的文件全部被取出来为止。

这个属于 Makefile 中静态模规则：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。跟我们的多规则目标的意思相近，但是又不相同。

Makefile变量的定义和使用

变量的定义

Makefile 文件中定义变量的基本语法如下：变量的名称=值列表

Makefile 中的变量的使用其实非常的简单，因为它并没有像其它语言那样定义变量的时候需要使用数据类型。变量的名称可以由大小写字母、阿拉伯数字和下划线构成。等号左右的空白符没有明确的要求，因为在执行 make 的时候多余的空白符会被自动的删除。至于值列表，既可以是零项，又可以是一项或者是多项。如：VALUE_LIST = one two three

调用变量的时候可以用 "${VALUE_LIST}" 来替换，这就是变量的引用。实例：

```
OBJ=main.o test.o test1.o test2.o
test:$(OBJ)
    gcc -o test $(OBJ)
```

这就是引用变量后的 Makefile 的编写，比我们之前的编写方式要简单的多。当要添加或者是删除某个依赖文件的时候，我们只需要改变变量 "OBJ" 的值就可以了。

变量的基本赋值

知道了如何定义，下面我们来说一下 Makefile 的变量的四种基本赋值方式：

- 简单赋值（:=）编程语言中常规理解的赋值方式，只对当前语句的变量有效。
- 递归赋值（=）赋值语句可能影响多个变量，所有目标变量相关的其他变量都受影响。
- 条件赋值（?=）如果变量未定义，则使用符号中的值定义变量。如果该变量已经赋值，则该赋值语句无效。

- 追加赋值（ += ）原变量用空格隔开的方式追加一个新值。

简单赋值

```
x:=foo
y:=$(x)b
x:=new
test:
	@echo "y=>$(y)"
	@echo "x=>$(x)"
```

在 shell 命令行执行 `make test` 我们会看到:y=>foob

x=>new

递归赋值

```
x=foo
y=$(x)b
x=new
test:
	@echo "y=>$(y)"
	@echo "x=>$(x)"
```

在 shell 命令行执行 `make test` 我们会看到:y=>newb

x=>new

条件赋值

```
x:=foo
y:=$(x)b
x?=new
test:
	@echo "y=>$(y)"
	@echo "x=>$(x)"
```

在 shell 命令行执行 `make test` 我们会看到:y=>foob

x=>foo

追加赋值

```
x:=foo
y:=$(x)b
x+=$(y)
test:
	@echo "y=>$(y)"
	@echo "x=>$(x)"
```

在 shell 命令行执行 `make test` 我们会看到:y=>foob

x=>foo foob

不同的赋值方式会产生不同的结果，我们使用的时候应该根据具体的情况选择相应的赋值规则。

变量使用的范围很广，它可以出现在规则的模式中，也可以出现在规则的命令中或者是作为 Makefile 函数的参数来使用。总之，变量的使用在我们的 Makefile 编写中还是非常广泛的，可以说我们的 Makefile 中必不可少的东西。

其实变量在我们的 Makefile 中还是有很多种类的，它们的意义是不相同的。比如我们的环境变量，自动变量，模式指定变量等。其他的变量我们会在其他的文章里做介绍。

Makefile自动化变量

关于自动化变量可以理解为由 Makefile 自动产生的变量。在模式规则中，规则的目标和依赖的文件名代表了一类的文件。规则的命令是对所有这一类文件的描述。我们在 Makefile 中描述规则时，依赖文件和目标文件是变动的，显然在命令中不能出现具体的文件名称，否则模式规则将失去意义。

那么模式规则命令中该如何表示文件呢？就需要使用“自动化变量”，自动化变量的取值根据执行的规则来决定，取决于执行规则的目标文件和依赖文件。下面是对所有的自动化变量进行的说明:

自动化变量	说明
$@	表示规则的目标文件名。如果目标是一个文档文件（Linux 中，一般成 .a 文件为文档文件，也成为静态的库文件）， 那么它代表这个文档的文件名。在多目标模式规则中，它代表的是触发规则被执行的文件名。
$%	当目标文件是一个静态库文件时，代表静态库的一个成员名。
$<	规则的第一个依赖的文件名。如果是一个目标文件使用隐含的规则来重建，则它代表由隐含规则加入的第一个依赖文件。
$?	所有比目标文件更新的依赖文件列表，空格分隔。如果目标文件时静态库文件，代表的是库文件（.o 文件）。
$^	代表的是所有依赖文件列表，使用空格分隔。如果目标是静态库文件，它所代表的只能是所有的库成员（.o 文件）名。 一个文件可重复的出现在目标的依赖中，变量“$^”只记录它的第一次引用的情况。就是说变量“$^”会去掉重复的依赖文件。
$+	类似“$^”，但是它保留了依赖文件中重复出现的文件。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”也包含目录部分）。

image

下面我们就自动化变量的使用举几个例子。

实例1:

```
test:test.o test1.o test2.o
	gcc -o $@ $^
test.o:test.c test.h
	gcc -o $@ $<
test1.o:test1.c test1.h
	gcc -o $@ $<
test2.o:test2.c test2.h
	gcc -o $@ $<
```

这个规则模式中用到了 "<" 和 "@" 代表的是目标文件test，“<”代表的是依赖文件中的第一个。我们在执行 make 的时候，make 会自动识别命令中的自动化变量，并自动实现自动化变量中的值的替换，这个类似于编译C语言文件的时候的预处理的作用。

实例2:

```
lib:test.o test1.o test2.o
    ar r $?
```

假如我们要做一个库文件，库文件的制作依赖于这三个文件。当修改了其中的某个依赖文件，在命令行执行 make 命令，库文件 "lib" 就会自动更新。"$?" 表示修改的文件。

GNU make 中在这些变量中加入字符 "D" 或者 "F" 就形成了一系列变种的自动化变量，这些自动化变量可以对文件的名称进行操作。

下面是一些详细的描述：

变量名	功能
$(@D)	表示文件的目录部分（不包括斜杠）。如果 "$@" 表示的是 "dir/foo.o" 那么 "$(@D)" 表示的值就是 "dir"。如果 "$@" 不存在斜杠（文件在当前目录下），其值就是 "."。
$(@F)	表示的是文件除目录外的部分（实际的文件名）。如果 "$@" 表示的是 "dir/foo.o"，那么 "$@F" 表示的值为 "dir"。
$(*D) $(*F)	分别代表 "茎" 中的目录部分和文件名部分
$(%D) $(%F)	当以 "archive(member)" 形式静态库为目标时，分别表示库文件成员 "member" 名中的目录部分和文件名部分。踏进对这种新型时的目标有效。
$(<D) $(<F)	表示第一个依赖文件的目录部分和文件名部分。
$(^D) $(^F)	分别表示所有依赖文件的目录部分和文件部分。
$(+D) $(+F)	分别表示所有的依赖文件的目录部分和文件部分。
$(?D) $(?F)	分别表示更新的依赖文件的目录部分和文件名部分。

image

Makefile目标文件搜索（VPATH和vpath）

我们都知道一个工程文件中的源文件有很多，并且存放的位置可能不相同（工程中的文件会被放到不同的目录下），所以按照之前的方式去编写 Makefile 会有问题。

之前列举的例子，所有的源文件基本上都是存放在与 Makefile 相同的目录下。只要依赖的文件存在，并且依赖规则没有问题，执行 make命令整个工程就会按照对我们编写规则去编译，最终会重建目标文件。那如果需要的文件是存在于不同的路径下，在编译的时候要去怎么办呢（不改变工程的结构）？这就用到了 Makefile 中为我们提供的目录搜索文件的功能。

常见的搜索的方法的主要有两种：一般搜索 `VPATH` 和选择搜索 `vpath` 。乍一看只是大小写的区别，其实两者在本质上也是不同的。

VPATH 和 vpath 的区别：VPATH 是变量，更具体的说是环境变量，Makefile 中的一种特殊变量，使用时需要指定文件的路径；vpath 是关键字，按照模式搜索，也可以说成是选择搜索。搜索的时候不仅需要加上文件的路径，还需要加上相应限制的条件。

VPATH的使用

在 Makefile 中可以这样写：VPATH := src

我们可以这样理解，把 src 的值赋值给变量 VPATH，所以在执行 make 的时候会从 src 目录下找我们需要的文件。

当存在多个路径的时候我们可以这样写：VPATH := src car

或者是VPATH := src:car

多个路径之间要使用空格或者是冒号隔开，表示在多个路径下搜索文件。搜索的顺序为我们书写时的顺序，拿上面的例子来说，我们应该先搜索 src 目录下的文件，再搜索 car 目录下的文件。

注意：无论你定义了多少路径，make 执行的时候会先搜索当前路径下的文件，当前目录下没有我们要找的文件，才去 VPATH 的路径中去寻

找。如果当前目录下有我们要使用的文件，那么 make 就会使用我们当前目录下的文件。

实例：

```
VPATH=src car
test:test.o
    gcc -o $@ $^
```

假设 test.c 文件没有在当前的目录而在当前文件的子目录 "src" 或者是 "car" 下，程序执行是没有问题的，但是生成的 test 的文件没有在定义的子目录文件中而是在当前的目录下，当然生成文件路径可以指定。

vpath的使用

学习了 VPATH的使用，我们再来了解一下关键字搜索 vpath 的使用，这种搜索方式一般被称作选择性搜索。使用上的区别我们可以这样理解：VPATH 是搜索路径下所有的文件，而 vpath 更像是添加了限制条件，会过滤出一部分再去寻找。

具体用法：

1. vpath PATTERN DIRECTORIES

2. vpath PATTERN

3. vpath

(PATTERN：可以理解为要寻找的条件，DIRECTORIES：寻找的路径)

首先是用法一，命令格式如下：

```
vpath test.c src
```

可以这样理解，在 src 路径下搜索文件 test.c。多路径的书写规则如下：

```
vpath test.c src car
```

或者是

```
vpath test.c src : car
```

多路径的用法其实和 VPATH 差不多，都是使用空格或者是冒号分隔开，搜索路径的顺序是先 src 目录，然后是 car 目录。

其次是用法二，命令格式如下：

```
vpath test.c
```

用法二的意思是清除符合文件 test.c 的搜索目录。

最后是用法三，命令格式如下：vpath

vpath 单独使的意思是清除所有已被设置的文件搜索路径。

另外在使用 vpath 的时候，搜索的条件中可以包含模式字符“%”，这个符号的作用是匹配一个或者是多个字符，例如“%.c”表示搜索路径下所有的 .c 结尾的文件。如果搜索条件中没有包含“%"，那么搜索的文件就是具体的文件名称。

使用什么样的搜索方法，主要是基于编译器的执行效率。使用 VPATH 的情况是前路径下的文件较少，或者是搜索的文件不能使用通配符表示，这些情况下使用VPATH最好。如果存在某个路径的文件特别的多或者是可以使用通配符表示的时候，就不建议使用 VPATH 这种方法，为什么呢？因为 VPATH 在去搜索文件的时没有限制条件，所以它回去检索这个目录下的所有文件，每一个文件都会进行对比，搜索和我们目录名相同的文件，不仅速度会很慢，而且效率会很低。我们在这种情况下就可以使用 vpath 搜索，它包含搜索条件的限制，搜索的时候只会从我们规定的条件中搜索目标，过滤掉不符合条件的文件，当然查找的时候也会比较的快。

为了体验实例的效果的更加明显，我们按照源代码树的布局来放置文件。我们把源代码放置在src目录下，包含的文件文件是：list1.c、list2.c、main.c 文件，我们把头文件包含在 include 的目录下，包含文件 list1.h、list2.h 文件。Makefile 放在这两个目录文件的上一级目录。

我们按照之前的方式来编写 Makefile 文件：

```
main:main.o list1.o list2.o
    gcc -o $@ $<
main.o:main.c
    gcc -o $@ $^
list1.o:list1.c list1.h
    gcc -o $@ $<
list2.o:list2.c list2.h
    gcc -o $@ $<
```

我们编译执行的 make 时候会发现命令行提示我们：

make:*** No rule to make target 'main.c',need by 'main.o'. stop.

出现错误并且编译停止了，为什么会出现错误呢？我们来看一下出现错误的原因，再去重建最终目标文件 main 的时候我们需要 main.o 文件，但是我们再去重建目标main.o 文件的时候，发现没有找到指定的 main.c 文件，这是错误的根本原因。

这个时候我们就应该添加上路径搜索，我们知道路径搜索的方法有两个：VPATH 和 vpath。我们先来使用一下 VPATH，使用方式很简单，我们只需要在上述的文件开头加上这样一句话：

```
VPATH=src include
```

再去执行 make 就不会出现错误。所以 Makefile 中的最终写法是这样的：

```
VPATH=src include
main:main.o list1.o list2.o
    gcc -o $@ $<
main.o:main.c
    gcc -o $@ $^
list1.o:list1.c list1.h
    gcc -o $@ $<
list2.o:list2.c list2.h
    gcc -o $@ $<
```

我们使用 vpath 的话同样可以解决这样的问题，只需要把上述代码中的 VPATH 所在行的代码改写成：

```
vpath %.c src
vpath %.h include
```

这样我们就可以用 vpath 实现功能，代码的最终展示为：

```
vpath %.c src
vpath %.h include
main:main.o list1.o list2.o
    gcc -o $@ $<
main.o:main.c
    gcc -o $@ $^
list1.o:list1.c list1.h
    gcc -o $@ $<
list2.o:list2.c list2.h
    gcc -o $@ $<
```

Makefile隐含规则

隐含规则就是需要我们做出具体的操作，系统自动完成。编写 Makefile 的时候，可以使用隐含规则来简化Makefile 文件编写。

实例：

```
test:test.o
    gcc -o test test.o
test.o:test.c
```

我们可以在 Makefile 中这样写来编译 test.c 源文件，相比较之前少写了重建 test.o 的命令。但是执行 make，发现依然重建了 test 和 test.o 文件，运行结果却没有改变。这其实就是隐含规则的作用。在某些时候其实不需要给出重建目标文件的命令，有的甚至可以不需要给出规则。实例：

```
test:test.o
    gcc -o test test.o
```

运行的结果是相同的。注意：隐含条件只能省略中间目标文件重建的命令和规则，但是最终目标的命令和规则不能省略。

隐含规则的具体的工作流程：make 执行过程中找到的隐含规则，提供了此目标的基本依赖关系。确定目标的依赖文件和重建目标需要使用的命令行。隐含规则所提供的依赖文件只是一个基本的（在C语言中，通常他们之间的对应关系是：test.o 对应的是 test.c 文件）。当需要增加这个文件的依赖文件的时候要在 Makefile 中使用没有命令行的规则给出。实例：

```
test:test.o
    gcc -o test test.o
test:test1.h
```

其实在有些时候隐含规则的使用会出现问题。因为有一个 make 的“隐含规则库”。库中的每一条隐含规则都有相应的优先级顺序，优先级也就会越高，使用时也就会被优先使用。

例如在 Makefile 中添加这行代码：

```
foo.o:foo.p
```

我们都知道 .p 文件是 Pascal 程序的源文件，如果书写规则时不加入命令的话，那么 make 会按照隐含的规则来重建目标文件 foo.o。如果当前目录下恰好存在 foo.c 文件的时候，隐含规则会把 foo.c 当做是 foo.o 的依赖文件进行目标文件的重建。因为编译 .c 文件的隐含规则在编译 .p 文件之前，显然优先级也会越高。当 make 找到生成 foo.o 的文件之后，就不会再去寻找下一条规则。如果我们不想使用隐含规则，在使用的时候不仅要声明规则，也要添加上执行的命令。

这里讲的是预先设置的隐含规则。如果不明确的写下规则，那么make 就会自己寻找所需要的规则和命令。当然我们也可以使用 make 选项：`-r` 或 `-n-builtin-rules` 选项来取消所有的预设值的隐含规则。当然即使是指定了“-r”的参数，某些隐含规则还是会生效。因为有很多的隐含规则都是使用了后缀名的规则来定义的，所以只要隐含规则中含有“后缀列表”那么隐含规则就会生效。默认的列表是：

```
.out、.a、.in、.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。
```

下面是一些常用的隐含规则：

- 编译 C 程序
- 编译 C++ 程序
- 编译 Pascal 程序
- 编译 Fortran/Ratfor 程序
- 预处理 Fortran/Ratfor 程序
- 编译 Modula-2 程序
- 汇编和需要预处理的汇编程序
- 链接单一的 object 文件
- Yacc C 程序
- Lex C 程序时的隐含规则

上面的编译顺序都是一些常用的编程语言执行隐含规则的顺序，我们在 Makefile 中指定规则时，可以参考这样的列表。当需要编译源文件的时候，考虑是不是需要使用隐含规则。如果不需要，就要把相应的规则和命令全部书写上去。

内嵌隐含规则的命令中，所使用的变量都是预定义的。我们将这些变量称为“隐含变量”。这些变量允许修改：可以通过命令行参数传递或者是设置系统环境变量的方式都可以对它进行重新定义。无论使用哪种方式，只要 make 在运行的，这些变量的定义有效。Makefile 的隐含规则都会使用到这些变量。

比如我们编译 .c 文件在我们的 Makefile 中就是隐含的规则，默认使用到的编译命令时 `cc`，执行的命令时 `cc -c` 我们可以对用上面的任何一种方式将 `CC` 定义为 `ncc`。这样我们就编译 .c 文件的时候就可以用 `ncc` 进行编译。

隐含规则中使用的变量可以分成两类：

1.代表一个程序的名字。例如：“CC”代表了编译器的这个可执行程序。

2.代表执行这个程序使用的参数.例如：变量“CFLAGS”。多个参数之间使用空格隔开。

下面我们来列举一下代表命令的变量，默认都是小写。

- AR：函数库打包程序，科创价静态库 .a 文档。
- AS：应用于汇编程序。
- CC：C 编译程序。
- CXX：C++编译程序。
- CO：从 RCS 中提取文件的程序。
- CPP：C程序的预处理器。
- FC：编译器和与处理函数 Fortran 源文件的编译器。
- GET：从CSSC 中提取文件程序。
- LEX：将Lex语言转变为 C 或 Ratfo 的程序。
- PC：Pascal 语言编译器。
- YACC：Yacc 文法分析器（针对于C语言）
- YACCR：Yacc 文法分析器。

Makefile ifeq、ifneq、ifdef和ifndef（条件判断）

日常使用 Makefile 编译文件时，可能会遇到需要分条件执行的情况，比如在一个工程文件中，可编译的源文件很多，但是它们的类型是不相同的，所以编译文件使用的编译器也是不同的。手动编译去操作文件显然是不可行的（每个文件编译时需要注意的事项很多），所以 make 为我们提供了条件判断来解决这样的问题。

需要解决的问题：要根据判断，分条件执行语句。

条件语句的作用：条件语句可以根据一个变量的值来控制 make 执行或者时忽略 Makefile 的特定部分，条件语句可以是两个不同的变量或者是常量和变量之间的比较。

条件语句使用优点：Makefile 中使用条件控制可以做到处理的灵活性和高效性。注意：条件语句只能用于控制 make 实际执行的 Makefile 文件部分，不能控制规则的 shell 命令执行的过程。

下面是条件判断中使用到的一些关键字：

关键字	功能
ifeq	判断参数是否不相等，相等为 true，不相等为 false。
ifneq	判断参数是否不相等，不相等为 true，相等为 false。
ifdef	判断是否有值，有值为 true，没有值为 false。
ifndef	判断是否有值，没有值为 true，有值为 false。

image

ifeq 和 ifneq

条件判断的使用方式如下：

ifeq (ARG1, ARG2)

ifeq 'ARG1' 'ARG2'

ifeq "ARG1" "ARG2"

ifeq "ARG1" 'ARG2'

ifeq 'ARG1' "ARG2"

实例：

```
libs_for_gcc= -lgnu
normal_libs=
foo:$(objects)
ifeq($(CC),gcc)
    $(CC) -o foo $(objects) $(libs_for_gcc)
else
    $(CC) -o foo $(objects) $(noemal_libs)
endif
```

条件语句中使用到三个关键字“ifeq”、“else”、“endif”。其中：“ifeq”表示条件语句的开始，并指定一个比较条件（相等）。括号和关键字之间要使用空格分隔，两个参数之间要使用逗号分隔。参数中的变量引用在进行变量值比较的时候被展开。“ifeq”，后面的是条件满足的时候执行的，条件不满足忽略；“else”表示当条件不满足的时候执行的部分，不是所有的条件语句都要执行此部分；“endif”是判断语句结束标志，Makefile 中条件判断的结束都要有。

其实 "ifneq" 和 "ifeq" 的使用方法是完全相同的，只不过是满足条件后执行的语句正好相反。

上面的例子可以换一种更加简介的方式来写：

```
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)
```

ifdef 和 ifndef

使用方式如下：

```
ifdef VARIABLE-NAME
```

它的主要功能是判断变量的值是不是为空，实例：

实例 1：

```
bar =
foo = $(bar)
all:
ifdef foo
   @echo yes
else
   @echo  no
endif
```

实例 2：

```
foo=
all:
ifdef foo
   @echo yes
else
   @echo  no
endif
```

通过两个实例对比说明：通过打印 "yes" 或 "no" 来演示执行的结果。我们执行 make 可以看到实例 1打印的结果是 "yes"，实例 2打印的结果是 "no"。其原因就是在实例 1 中，变量“foo”的定义是“foo = $(bar)”。虽然变量“bar”的值为空，但是“ifdef”的判断结果为真，这种方式判断显然是有不行的，因此当我们需要判断一个变量的值是否为空的时候需要使用“ifeq" 而不是“ifdef”。

注意：在 make 读取 Makefile 文件时计算表达式的值，并根据表达式的值决定判断语句中的哪一个部分作为此 Makefile 所要执行的内容。因此在条件表达式中不能使用自动化变量，自动化变量在规则命令执行时才有效，更不能将一个完整的条件判断语句分卸在两个不同的 Makefile 的文件中。在一个 Makefile 中使用指示符 "include" 包含另一个 Makefile 文件。

Makefile伪目标

伪目标可以这样来理解，它并不会创建目标文件，只是想去执行这个目标下面的命令。伪目标的存在可以帮助我们找到命令并执行。

使用伪目标有两点原因：

- 避免我们的 Makefile 中定义的只执行的命令的目标和工作目录下的实际文件出现名字冲突。
- 提高执行 make 时的效率，特别是对于一个大型的工程来说，提高编译的效率也是我们所必需的。

我们先来看一下第一种情况的使用。如果需要书写这样一个规则，规则所定义的命令不是去创建文件，而是通过 make 命令明确指定它来执行一些特定的命令。实例：

```
clean:
   rm -rf *.o test
```

规则中 rm 命令不是创建文件 clean 的命令，而是执行删除任务，删除当前目录下的所有的 .o 结尾和文件名为 test 的文件。当工作目录下不存在以 clean 命令的文件时，在 shell 中输入 make clean 命令，命令 rm -rf *.o test 总会被执行，这也是我们期望的结果。

如果当前目录下存在文件名为 clean 的文件时情况就会不一样了，当我们在 shell 中执行命令 make clean，由于这个规则没有依赖文件，所以目标被认为是最新的而不去执行规则所定义的命令。因此命令 rm 将不会被执行。为了解决这个问题，删除 clean 文件或者是在 Makefile 中将目标 clean 声明为伪目标。将一个目标声明称伪目标的方法是将它作为特殊的目标 `.PHONY` 的依赖，如下：

.PHONY:clean

这样 clean 就被声明成一个伪目标，无论当前目录下是否存在 clean 这个文件，当我们执行 make clean 后 rm 都会被执行。而且当一个目标被声明为伪目标之后，make 在执行此规则时不会去试图去查找隐含的关系去创建它。这样同样提高了 make 的执行效率，同时也不用担心目标和文件名重名而使我们的编译失败。

在书写伪目标的时候，需要声明目标是一个伪目标，之后才是伪目标的规则定义。目标 "clean" 的完整书写格式如下：

```
.PHONY:clean
clean:
    rm -rf *.o test
```

伪目标的另一种使用的场合是在 make 的并行和递归执行的过程中，此情况下一般会存在一个变量，定义为所有需要 make 的子目录。对多个目录进行 make 的实现，可以在一个规则的命令行中使用 shell 循环来完成。如下：

```
SUBDIRS=foo bar baz
subdirs:
    for dir in $(SUBDIRS);do $(MAKE) -C $$dir;done
```

代码表达的意思是当前目录下存在三个子文件目录，每个子目录文件都有相对应的 Makefile 文件，代码中实现的部分是用当前目录下的 Makefile 控制其它子模块中的 Makefile 的运行，但是这种实现方法存在以下几个问题：

- 当子目录执行 make 出现错误时，make 不会退出。就是说，在对某个目录执行 make 失败以后，会继续对其他的目录进行 make。在最终执行失败的情况下，我们很难根据错误提示定位出具体实在那个目录下执行 make 发生的错误。这样给问题定位造成很大的困难。为了解决问题可以在命令部分加入错误检测，在命令执行的错误后主动退出。不幸的是如果在执行 make 时使用了 "-k" 选项，此方式将失效。

- 另外一个问题就是使用这种 shell 循环方式时，没有用到 make 对目录的并行处理功能由于规则的命令时一条完整的 shell 命令，不能被并行处理。

有了伪目标之后，我们可以用它来克服以上方式所存在的两个问题，代码展示如下：

```
SUBDIRS=foo bar baz
.PHONY:subdirs $(SUBDIRS)
subdirs:$(SUBDIRS)
$(SUBDIRS):
    $(MAKE) -C $@
foo:baz
```

上面的实例中有一个没有命令行的规则“foo:baz”，这个规则是用来规定三个子目录的编译顺序。因为在规则中 "baz" 的子目录被当作成了 "foo" 的依赖文件，所以 "baz" 要比 "foo" 子目录更先执行，最后执行 "bar" 子目录的编译。

一般情况下，一个伪目标不作为另外一个目标的依赖。这是因为当一个目标文件的依赖包含伪目标时，每一次在执行这个规则伪目标所定义的命令都会被执行（因为它作为规则的依赖，重建规则目标时需要首先重建规则的所有依赖文件）。当一个伪目标没有任何目标（此目标是一个可被创建或者是已存在的文件）的依赖时，我们只能通过 make 的命令来明确的指定它的终极目标，执行它所在规则所定义的命令。例如 make clean。

伪目标实现多文件编辑

如果在一个文件里想要同时生成多个可执行文件，我们可以借助伪目标来实现。使用方式如下：

```
.PHONY:all
all:test1 test2 test3
test1:test1.o
    gcc -o $@ $^
test2:test2.o
    gcc -o $@ $^
test3:test3.o
    gcc -o $@ $^
```

我们在当前目录下创建了三个源文件，目的是把这三个源文件编译成为三个可执行文件。将重建的规则放到 Makefile 中，约定使用 "all" 的伪目标来作为最终目标，它的依赖文件就是要生成的可执行文件。这样的话只需要一个 make 命令，就会同时生成三个可执行文件。

之所以这样写，是因为伪目标的特性，它总会被执行，所以它依赖的三个文件的目标就不如 "all" 这个目标新，所以，其他的三个目标的规则总是被执行，这也就达到了我们一口气生成多个目标的目的。我们也可以实现单独的编译这三个中的任意一个源文件（我们想去重建 test1，我们可以执行命令 `make test1` 来实现 ）。

Makefile常用字符串处理函数

函数的调用和变量的调用很像。引用变量的格式为 `$(变量名)`，函数调用的格式如下：

```
$(<function> <arguments>) 或者是 ${<function> <arguments>}
```

其中，function 是函数名，arguments 是函数的参数，参数之间要用逗号分隔开。而参数和函数名之间使用空格分开。调用函数的时候要使用字符“$”，后面可以跟小括号也可以使用花括号。这个其实我们并不陌生，我们之前使用过许多的函数，比如说展开通配符的函数 wildcard，以及字符串替换的函数 patsubst，Makefile 中函数并不是很多。

今天主要讲的是字符串处理函数，这些都是我们经常使用到的函数，下面是对函数详细的介绍。

1. 模式字符串替换函数，函数使用格式如下：

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

函数说明：函数功能是查找 text 中的单词是否符合模式 pattern，如果匹配的话，则用 replacement 替换。返回值为替换后的新字符串。实例：

```
OBJ=$(patsubst %.c,%.o,1.c 2.c 3.c)
all:
    @echo $(OBJ)
```

执行 make 命令，我们可以得到的值是 "1.o 2.o 3.o"，这些都是替换后的值。

2. 字符串替换函数，函数使用格式如下：

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

函数说明：函数的功能是把字符串中的 form 替换成 to，返回值为替换后的新字符串。实例：

```
OBJ=$(subst ee,EE,feet on the street)
all:
    @echo $(OBJ)
```

执行 make 命令，我们得到的值是“fEEt on the strEEt”。

3. 去空格函数，函数使用格式如下：

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

函数说明：函数的功能是去掉字符串的开头和结尾的字符串，并且将其中的多个连续的空格合并成为一个空格。返回值为去掉空格后的字符串。实例：

```
OBJ=$(strip   a      b c)
all:
    @echo $(OBJ)
```

执行完 make 之后，结果是“a b c”。这个只是除去开头和结尾的空格字符，并且将字符串中的空格合并成为一个空格。

4. 查找字符串函数，函数使用格式如下：

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

函数说明：函数的功能是查找 in 中的 find ,如果我们查找的目标字符串存在。返回值为目标字符串，如果不存在就返回空。实例：

```
OBJ=$(findstring a,a b c)
all:
    @echo $(OBJ)
```

执行 make 命令，得到的返回的结果就是 "a"。

5. 过滤函数，函数使用格式如下：

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

函数说明：函数的功能是过滤出 text 中符合模式 pattern 的字符串，可以有多个 pattern 。返回值为过滤后的字符串。实例：

```
OBJ=$(filter %.c %.o,1.c 2.o 3.s)
all:
    @echo $(OBJ)
```

执行 make 命令，我们得到的值是“1.c 2.o”。

6. 反过滤函数，函数使用格式如下：

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

函数说明：函数的功能是功能和 filter 函数正好相反，但是用法相同。去除符合模式 pattern 的字符串，保留符合的字符串。返回值是保留的字符串。实例：

```
OBJ=$(filter-out 1.c 2.o ,1.o 2.c 3.s)
all:
   @echo $(OBJ)
```

执行 make 命令，打印的结果是“3.s”。

7. 排序函数，函数使用格式如下：

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

函数说明：函数的功能是将 `<list>` 中的单词排序（升序）。返回值为排列后的字符串。实例：

```
OBJ=$(sort foo bar foo lost)
all:
   @echo $(OBJ)
```

执行 make 命令，我们得到的值是“bar foo lost”。注意：sort会去除重复的字符串。

8. 取单词函数，函数使用格式如下：

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

函数说明：函数的功能是取出函数 `<text>` 中的第n个单词。返回值为我们取出的第 n 个单词。实例：

```
OBJ=$(word 2,1.c 2.c 3.c)
all:
   @echo $(OBJ)
```

执行 make 命令，我们得到的值是“2.c”。

Makefile常用文件名操作函数

我们在编写 Makefile 的时候，很多情况下需要对文件名进行操作。例如获取文件的路径，去除文件的路径，取出文件前缀或后缀等等。当遇到这样的问题的时手动修改是不太可能的，因为文件可能会很多，而且 Makefile 中操作文件名可能不止一次。所以 Makefile 给我们提供了相应的函数去实现文件名的操作。

注意：下面的每个函数的参数字符串都会被当作或是一个系列的文件名来看待。

1. 取目录函数，函数使用格式如下：

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

函数说明：函数的功能是从文件名序列 names 中取出目录部分，如果没有 names 中没有 "/" ，取出的值为 "./" 。返回值为目录部分，指的是最后一个反斜杠之前的部分。如果没有反斜杠将返回 “./” 。实例：

```
OBJ=$(dir src/foo.c hacks)
all:
    @echo $(OBJ)
```

执行 make 命令，我们可以得到的值是 “src/ ./” 。提取文件 foo.c 的路径是 "/src" 和文件 hacks 的路径 "./"。

2. 取文件函数，函数使用格式如下：

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

函数说明：函数的功能是从文件名序列 names 中取出非目录的部分。非目录的部分是最后一个反斜杠之后的部分。返回值为文件非目录的部分。实例：

```
OBJ=$(notdir src/foo.c hacks)
all:
    @echo $(OBJ)
```

执行 make 命令，我们可以得到的值是 “foo.c hacks” 。

3. 取后缀名函数，函数使用格式如下：

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

函数说明：函数的功能是从文件名序列中 names 中取出各个文件的后缀名。返回值为文件名序列 names 中的后缀序列，如果文件没有后缀名，则返回空字符串。实例：

```
OBJ=$(suffix src/foo.c hacks)
all:
    @echo $(OBJ)
```

执行 make 命令，我们得到的值是 “.c ” 。文件 "hacks" 没有后缀名，所以返回的是空值。

4. 取前缀函数，函数使用格式如下：

```
$(basename <names>)
```

函数说明：函数的功能是从文件名序列 names 中取出各个文件名的前缀部分。返回值为被取出来的文件的前缀名，如果文件没有前缀名则返回空的字符串。实例：

```
OBJ=$(notdir src/foo.c hacks)
all:
   @echo $(OBJ)
```

执行 make 命令，我们可以得到值是“src/foo hacks”。获取的是文件的前缀名，包含文件路径的部分。

5. 添加后缀名函数，函数使用格式如下：

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

函数说明：函数的功能是把后缀 suffix 加到 names 中的每个单词后面。返回值为添加上后缀的文件名序列。实例：

```
OBJ=$(addsuffix .c,src/foo.c hacks)
all:
   @echo $(OBJ)
```

执行 make 后我们可以得到“sec/foo.c.c hack.c”。我们可以看到如果文件名存在后缀名，依然会加上。

6. 添加前缀名函数，函数使用格式如下：

```
$(addperfix <prefix>,<names>)
```

函数说明：函数的功能是把前缀 prefix 加到 names 中的每个单词的前面。返回值为添加上前缀的文件名序列。实例：

```
OBJ=$(addprefix src/, foo.c hacks)
all:
   @echo $(OBJ)
```

执行 make 命令，我们可以得到值是 "src/foo.c src/hacks" 。我们可以使用这个函数给我们的文件添加路径。

7. 链接函数，函数使用格式如下：

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

函数说明：函数功能是把 list2 中的单词对应的拼接到 list1 的后面。如果 list1 的单词要比 list2的多，那么，list1 中多出来的单词将保持原样，如果 list1 中的单词要比 list2 中的单词少，那么 list2 中多出来的单词将保持原样。返回值为拼接好的字符串。实例：

```
OBJ=$(join src car,abc zxc qwe)
all:
   @echo $(OBJ)
```

执行 make 命令，我们可以得到的值是“srcabc carzxc qwe”。很显然 <list1> 中的文件名比 <list2> 的少，所以多出来的保持不变。

8. 获取匹配模式文件名函数，命令使用格式如下：

```
$(wildcard PATTERN)
```

函数说明：函数的功能是列出当前目录下所有符合模式的 PATTERN 格式的文件名。返回值为空格分隔并且存在当前目录下的所有符合模式 PATTERN 的文件名。实例：

```
OBJ=$(wildcard *.c  *.h)
all:
   @echo $(OBJ)
```

执行 make 命令，可以得到当前函数下所有的 ".c " 和 ".h" 结尾的文件。这个函数通常跟的通配符 "*" 连用，使用在依赖规则的描述的时候被展开（在这里我们的例子如果没有 wildcard 函数，我们的运行结果也是这样，"echo" 属于 shell 命令，在使用通配符的时通配符自动展开，我们这里只是相要说明一下这个函数在使用时，如果通过引用变量出现在规则中要被使用）。

Makefile中的其它常用函数

Makefile 中的其他的函数。以下是这些函数的详细说明。

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

函数的功能是：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> 的返所返回的每个字符串会以空格分割，最后当整个循环结束的时候， <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> 中一般会只用 <var> 这个参数来一次枚举 <list> 中的单词。

实例：

```
name:=a b c d
files:=$(foreach n,$(names),$(n).o)
all:
   @echo $(files)
```

执行 make 命令，我们得到的值是“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> 的变量将不再作用，其作用域只在 foreach 函数当中。

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

可见，if 函数可以包含 else 部分，或者是不包含，即if函数的参数可以是两个，也可以是三个。 condition 参数是 if 表达式，如果其返回的是非空的字符串，那么这个表达式就相当于返回真，于是， then-part 就会被计算，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），那么 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 将会是这个函数的返回值。此时如果 else-part 没有被定义，那么整个函数返回空字串符。所以， then-part 和 else-part 只会有一个被计算。

实例：

```
OBJ:=foo.c
OBJ:=$(if $(OBJ),$(OBJ),main.c)
all:
    @echo $(OBJ)
```

执行 make 命令我们可以得到函数的值是 foo.c，如果变量 OBJ 的值为空的话，我们得到的 OBJ 的值就是 main.c 。

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

call 函数是唯一一个可以用来创建新的参数化的函数。我们可以用来写一个非常复杂的表达式，这个表达式中，我们可以定义很多的参数，然后你可以用 call 函数来向这个表达式传递参数。

当 make 执行这个函数的时候， expression 参数中的变量 (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。

实例 1：

```
reverse = $(1) $(2)
foo = $(call reverse,a,b)
all:
    @echo $(foo)
```

那么，foo 的值就是“a b”。当然，参数的次序可以是自定义的，不一定是顺序的，

实例 2：

```
reverse = $(2) $(1)
foo = $(call reverse,a,b)
all:
    @echo $(foo)
```

此时的 foo 的值就是“b a”。

```
$(origin <variable>)
```

origin 函数不像其他的函数，它并不操作变量的值，它只是告诉你这个变量是哪里来的。注意： variable 是变量的名字，不应该是引用，所以最好不要在 variable 中使用“$”字符。origin 函数会员其返回值来告诉你这个变量的“出生情况”。

下面是origin函数返回值：

- “undefined”：如果<variable>从来没有定义过，函数将返回这个值。
- “default”：如果<variable>是一个默认的定义，比如说“CC”这个变量。
- “environment”：如果<variable>是一个环境变量并且当Makefile被执行的时候， “-e”参数没有被打开。

- “file”：如果<variable>这个变量被定义在Makefile中，将会返回这个值。
- “command line”：如果<variable>这个变量是被命令执行的，将会被返回。
- “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。

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 ，其包含了一个定义文件 `Make.def`，在 `Make.def` 中定义了一个变量 `bletch`，而我们的环境变量中也有一个环境变量 `bletch`，我们想去判断一下这个变量是不是环境变量，如果是我们就把它重定义了。如果是非环境变量，那么我们就不重新定义它。于是，我们在 Makefile 中，可以这样写：

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

当然，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的，可是 `override` 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，而不是重新定义命令行传来的。

Makefile命令的编写

令是由 shell 命令行组成，他们是一条一条执行的。多个命令之间要使用分号隔开，Makefile 中的任何命令都要以 `tab` 键开始。多个命令行之间可以有空行和注释行，在执行规则时空行会被自动忽略。

通常系统中可能存在不同的 shell 。但是 make 处理 Makefile 过程时，如果没有明确的指定，那么对所有规则中的命令行的解析使用 `bin/sh` 来完成。执行过程中使用的 shell 决定了规则中的命令的语法和处理机制。当使用默认的 `bin/sh` 时，命令中出现的字符“#”到行末的内容被认为是注释。当然了“#”可以不在此行的行首，此时“#”之前的内容不会被作为注释处理。

命令回显

通常 make 在执行命令行之前会把要是执行的命令行输出到标准输出设备。我们称之为 "回显"，就好像我们在 shell 环境下输入命令执行时一样。如果规则的命令行以字符“@”开始，则 make 在执行的时候就不会显示这个将要被执行的命令。典型的用法是在使用 `echo` 命令输出一些信息时。

实例 1：

```
OBJ=test main list
all:
   @echo $(OBJ)
```

执行时将会得到 `test main list` 这条输出信息，如果在执行命令之前没有字符“@”，那么make的输出将是 `echo test main list` 。

我们在执行 make 时添加上一些参数，可以控制命令行是否输出。当使用 make 的时候机加上参数 `-n` 或者是 `--just-print` ，执行时只显示所要执行的命令，但不会真正的执行这个命令。只有在这种情况下 make 才会打印出所有的 make 需要执行的命令，其中包括了使用的“@”字符开始的命令。这个选项对于我们调试 Makefile 非常的有用，使用这个选项就可以按执行顺序打印出 Makefile 中所需要执行的所有命令。而 make 参数 `-s` 或者是 `--slient` 则是禁止所有的执行命令的显示。就好像所有的命令行都使用“@”开始一样。

命令的执行

当规则中的目标需要被重建的时候，此规则所定义的命令将会被执行，如果是多行的命令，那么每一行命令将是在一个独立的子 shell 进程中被执行。因此，多命令行之间的执行命令时是相互独立的，相互之间不存在依赖。

在 Makefile 中书写在同一行中的多个命令属于一个完整的 shell 命令行，书写在独立行的一条命令是一个独立的 shell 命令行。因此：在一个规则的命令中命令行 “cd”改变目录不会对其后面的命令的执行产生影响。就是说之后的命令执行的工作目录不会是之前使用“cd”进入的那个目录。如果达到这个目的，就不能把“cd”和其后面的命令放在两行来书写。而应该把这两个命令放在一行上用分号隔开。这样才是一个完整的 shell 命令行。

实例 2:

```
foo:bar/lose
    cd bar;gobble lose >../foo
```

如果想把一个完整的shell命令行书写在多行上，需要使用反斜杠 ()来对处于多行的命令进行连接，表示他们是一个完整的shell命令行。例如上例我们也可以这样书写:

```
foo:bar.lose
    cd bar; \
    gobble lose > ../foo
```

make 对所有规则的命令的解析使用环境变量“SHELL”所指定的那个程序。在 GNU make 中，默认的程序时 “/bin/sh”。不像其他的绝大多数变量，他们的只可以直接从同名的系统环境变量那里获得。make 的环境变量 “SHELL”没有使用环境变量的定义。因为系统环境变量“SHELL”指定的那个程序被用来作为用户和系统交互的接口程序，他对于不存在直接交互过程的 make 显然不合适。在 make 环境变量中“SHELL”会被重新赋值；他作为一个变量我们也可以在 Makefile 中明确的给它赋值，变量“SHELL“的默认值时“/bin/sh”。

并发执行命令

GNU make 支持同时执行多条命令。通常情况下，同一时刻只有一个命令在执行，下一个命令只有在当前命令结束之后才能够开始执行。不过可以通过 make 命令行选项 "-j" 或者 "--jobs" 来告诉 make 在同一时刻可以允许多条命令同时执行。

如果选项 "-j" 之后存在一个整数，其含义是告诉 make 在同一时刻可以允许同时执行的命令行的数目。这个数字被称为 `job slots` 。当 "-j" 选项中没有出现数字的时候，那么同一时间执行的命令数目没有要求。使用默认的 `job solts` ，值为1，表示make将串行的执行规则的命令（同一时刻只能由一条命令被执行）。

并行执行命令所带来的问题是显而易见的:

- 多个同时执行的命令的输出信息将同时被输出到终端。当出现错误时很难根据一大堆凌乱的信息来区分那条命令执行错误。
- 在同一时刻可能会存在多个命令执行的进程同时读取到标准输入，但是对于白哦准输入设备来说，在同一时刻只能存在一个进程访问它。就是说在某个时间点，make只能保证此刻正在执行的进程中的一个进程读取标准输入流。而其他的进程键的标准输入流将设置为无效。因此在此一时刻多个执行命令的进程中只有一个进程获得标准输入，而其他的需要读取标准输入流的进程由于输入流无效而导致致命的错误。

Makefile include文件包含

包含其他文件使用的关键字是 "include"，和 C 语言包含头文件的方式相同。

当 make 读取到 "include" 关键字的时候，会暂停读取当前的 Makefile，而是去读 "include" 包含的文件，读取结束后再继读取当前的 Makefile 文件。"include" 使用的具体方式如下:

```
include <filenames>
```

filenames 是 shell 支持的文件名（可以使用通配符表示的文件）。注意："include" 关键字所在的行首可以包含一个或者是多个的空格（读取的时候空格会被自动的忽略），但是不能使用 Tab 开始，否则会把 "include" 当作式命令来处理。包含的多个文件之间要使用空格分隔开。使用 "include" 包含进来的 Makefile 文件中，如果存在函数或者是变量的引用，它们会在包含的 Makefile 中展开。

include 通常使用在以下的场合：

- 在一个工程文件中，每一个模块都有一个独立的 Makefile 来描述它的重建规则。它们需要定义一组通用的变量定义或者是模式规则。通用的做法是将这些共同使用的变量或者模式规则定义在一个文件中，需要的时候用 "include" 包含这个文件。
- 当根据源文件自动产生依赖文件时，我们可以将自动产生的依赖关系保存在另一个文件中。然后在 Makefile 中包含这个文件。

注意：如果使用 "include" 包含文件的时候，指定的文件不是文件的绝对路径或者是为当前文件下没有这个文件，make 会根据文件名会在以下几个路径中去找，首先我们在执行 make 命令的时候可以加入选项 "-I" 或 "--include-dir" 后面添加上指定的路径，如果文件存在就会被使用，如果文件不存在将会在其他的几个路径中搜索："usr/gnu/include"、"usr/local/include" 和 "usr/include"。

如果在上面的路径没有找到 "include" 指定的文件，make 将会提示一个文件没有找到的警示提示，但是不会退出，而是继续执行 Makefile 的后续的内容。当完成读取整个 Makefile 后，make 将试图使用规则来创建通过 "include" 指定但不存在的文件。当不能创建的时候，文件将会保存退出。

使用时，通常用 "-include" 来代替 "include" 来忽略文件不存在或者是无法创建的错误提示，使用格式如下：-include <filename>

使用方法和 "include" 的使用方法相同。

这两种方式之间的区别：

- 使用 "include <filenames>" ，make 在处理程序的时候，文件列表中的任意一个文件不存在的时候或者是没有规则去创建这个文件的时候，make 程序将会提示错误并保存退出。
- 使用 "-include <filenames>"，当包含的文件不存在或者是没有规则去创建它的时候，make 将会继续执行程序，只有真正由于不能完成终极目标重建的时候我们的程序才会提示错误保存退出。

Makefile嵌套执行make

我们都知道在一个大的工程文件中，不同的文件按照功能被划分到不同的模块中，也就说很多的源文件被放置在了不同的目录下。每个模块可能都会有自己的编译顺序和规则，如果在一个 Makefile 文件中描述所有模块的编译规则，就会很乱，执行时也会不方便，所以就需要在不同的模块中分别对它们的规则进行描述，也就是每一个模块都编写一个 Makefile 文件，这样不仅方便管理，而且可以迅速发现模块中的问题。这样我们只需要控制其他模块中的 Makefile 就可以实现总体的控制，这就是 make 的嵌套执行。

如何来使用呢？举例说明如下：

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

这个例子可以这样来理解，在当前目录下有一个目录文件 subdir 和一个 Makefile 文件，子目录 subdir 文件下还有一个 Makefile 文件，这个文件是用来描述这个子目录文件的编译规则。使用时只需要在最外层的目录中执行 make 命令，当命令执行到上述的规则时，程序会进入到子目录中执行 make。这就是嵌套执行 make，我们把最外层的 Makefile 称为是总控 Makefile。

上述的规则也可以换成另外一种写法：

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

在 make 的嵌套执行中，我们需要了解一个变量 "CURDIR"，此变量代表 make 的工作目录。当使用 make 的选项 "-C" 的时候，命令就会进入指定的目录中，然后此变量就会被重新赋值。总之，如果在 Makefile 中没有对此变量进行显式的赋值操作，那么它就表示 make 的工作目录。我们也可以在 Makefile 中为这个变量赋一个新的值，当然重新赋值后这个变量将不再代表 make 的工作目录。

export的使用

使用 make 嵌套执行的时候，变量是否传递也是我们需要注意的。如果需要变量的传递，那么可以这样来使用：export <variable>

如果不需要那么可以这样来写：unexport <variable>

<variable>是变量的名字，不需要使用 "$" 这个字符。如果所有的变量都需要传递，那么只需要使用 "export" 就可以，不需要添加变量的名字。

Makefile 中还有两个变量不管是不是使用关键字 "export" 声明，它们总会传递到下层的 Makefile 中。这两个变量分别是 SHELL 和 MAKEFLAGS，特别是 MAKEFLAGS 变量，包含了 make 的参数信息。如果执行总控 Makefile 时，make 命令带有参数或者在上层的 Makefile 中定义了这个变量，那么 MAKEFLAGS 变量的值将会是 make 命令传递的参数，并且会传递到下层的 Makefile 中，这是一个系统级别的环境变量。

make 命令中有几个参数选项并不传递，它们是:"-C"、"-f"、"-o"、"-h" 和 "-W"。如果我们不想传递 MAKEFLAGS 变量的值，在 Makefile 中可以这样来写：

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

案例：通过一个大的项目工程来详细的分析一下如何嵌套执行 make。

假设有一个 MP3 player 的应用程序，它可以被划分为若干个组件：用户界面（ui）、编解码器（codec）以及数据管理库（db）。它们分别可以用三个程序库来表示：libui.a、libcodec.a 和 libdb.a。将这些组件紧凑的放到一起就可以组成这个应用程序。具体的文件结构展示为（我们展示的只是目录文件，没有展示详细的源文件）：

```
├——Makefile //最外层的Makefile文件，不是目录文件。
```

```
├——include //编译的时候需要链接的库文件
```

```
|  ├——codec //libui.a 库文件所在的目录
```

```
|  ├——db //libdb.a 库文件所在的目录
```

```
|  ├——ui //libui.a库文件所在的目录
```

```
├——lib //源文件所在的目录，子目录文件中包含Makefile文件
```

```
|  ├——codec //编解码器所在的源文件的目录
```

```
|  ├——db //数据库源文件所在的目录
```

```
|  ├——ui //用户界面源文件所在目录
```

```
├——app
```

```
|  ├——player
```

```
└——doc //这个工程编译说明
```

我们可以看到最外层有一个 Makefile 文件，这就是我们的 "总控Makefile" 文件，我们使用这个 Makefile 调用项目中各个子目录的 Makefile 文件的运行。假设只有我们的 lib 目录下和 app 目录下的各个子目录含有 Makefile 文件。那我们总控的 Makefile 的文件可以这样来写：

```
lib_codec := lib/codec
lib_db    := lib/db
lib_ui    := lib/ui
libraries   := $(lib_codec) $(lib_db) $(lib_ui)
player    := app/player
.PHONY : all $(player) $(libraries)
all : $(player)
$(player) $(libraries) :
    $(MAKE) -C $@
```

我们可以看到在 "总控 Makefile" 中，一个规则在工作目标上列出了所有的子目录，它对每一个子目录的 Makefile 调用的代码是： (libraries) :

```
  $(MAKE) -C $@
```

在 Makefile 文件中，MAKE 变量应该总是用来调用 make 程序。make 程序一看到 MAKE 变量就会把它设成 make 的实际路径，所以递归调用中的每次调用都会使用同一个执行文件。此外，当命令 --touch(-t)、--just-print(-n) 和 --question(-q) 被使用时，包含 MAKE 变量的每一行都会受到特别的处理。

由于这些“工作目标目录”被设成 .PHONY 的依赖文件，所以即使工作目标已经更新，此规则仍旧会进行更新动作。使 --directory(-C) 选项的目的是要让 make 在读取 Makefile 之前先切换到相应的 "工作目录" 。

当 make 在建立依存图的时候找不到程序库与 app/player 工作目标之间的依存关系时，这意味着建立任何程序库之前，make 将会先执行

app/player 目录中的 Makefile。显然这将会导致失败的结果，因为应用程序的链接需要程序库。为解决这个问题，我们会提供额外的依存信息：

```
$(player) : $(libraries)
$(lib_ui) : $(lib_db) $(lib_codec)
```

我们在此处做了如下的描述：运行 app/player 目录中的 Makefile 之前必须先运行程序库子目录中的 Makefile。此外，编译 lib/ui 目录中的程序代码之前必须先编译 lib/db 和lib/codec 目录中的程序库。这么做可以确保任何自动产生的程序代码，在 lib/ui 目录中的程序代码被编译之前就已经产生出来了。

更新必要条件的时候，会引发微妙的次序问题。如同所有的依存关系，更新的次序取决于依存图的分析结果，但是当工作目标的必要条件（依赖文件）出现在同一行时，GNU make 将会从左至右的次序进行更新。例如：

```
all : a b c
all : d e f
```

如果不存在其他的依存关系，这6个必要条件的更新动作可以是任何次序，不过GNU make将会以从左向右的次序来更新出现在同一行的必要条件，这会产生如下的更新次序："a b c d e f" 或 "d e f a b c"。注意：不要因为之前这么做更新的次序是对的，就以为每次这么做都是对的，而忘了提供完整的依存信息。

最后，依存分析可能会产生不同的次序而引发一些问题。所以，如果有一组工作目标需要以特定的次序进行更新时，就必须提供适当的必要条件来实现正确的次序。

当我们在最外层执行 make 的时候我们会看到l输出的信息：

```
make -C lib/db
make[1]: Entering directory ‘/MP3_player/lib/db’
make[1]:Update db library...
make[1]: Leaving directory ‘/MP3_player/lib/db’
make -C lib/codec
make[1]: Entering directory ‘/MP3_player/lib/codec’
make[1]:Update codec library...
make[1]: Leaving directory ‘/MP3_player/lib/codec’
make -C lib/ui
make[1]: Entering directory ‘/MP3_player/lib/ui’
make[1]:Update ui library...
make[1]: Leaving directory ‘/MP3_player/lib/ui’
make -C app/player
make[1]: Entering directory ‘/MP3_player/app/player’
make[1]:Update player library...
make[1]: Leaving directory ‘/MP3_player/app/player’
```

当 make 发觉它正在递归调用另一个 make 时，他会启 用--print-directory(-w) 选项，这会使得 make 输出 Entering directory(进入目录) 和 Leaving directory(离开目录) 的信息。当 --directory(-C) 选项被使用时，也会启用这个选项。我们还可以看到每一行中，MAKELEVEL 这个 make 变量的值加上方括号之后被一起输出。在这个简单的例子里，每个组件的 Makefile 只会输出组件正在更新的信息，而不会真正的更新组件。

我们通过这个例子应该可以了解，在 make 的嵌套执行执行的时候的调用子目录的方式，还有子目录再去执行 make 时候的顺序。这是一个很典型的例子，我们的每一个工程文件都可以用上面的结构展示出来，我们只要懂得每一个子目录在被调用时候的顺序，我们就可以很轻松的编写 "总控Makefile" 。

make命令参数和选项大汇总

我们在在执行 make 命令时，有的时候需要加上一下参数选项来保证我们的程序的执行，其实之前已经遇到过 make 在执行命令的时候需要添加上参数选项，比如只打印命令但不执行使用的参数是 "-n" ，还有只执命令不打印命令的参数选项是 "-s"，包含其它文件的路径参数选项是 "-include"等等。

我们现在列举一下 make 可以使用的参数选项，以及它们的功能是什么。

<style> td {white-space:pre-wrap;border:1px solid #dee0e3;}</style> <byte-sheet-html-origin data-id="EMJfPwGYuY-1623856826604" data-version="3" data-is-embed="true"><colgroup><col width="195"><col width="1268"></colgroup>
参数选项	功能
-b，-m	忽略，提供其他版本 make 的兼容性
-B，--always-make	强制重建所有的规则目标，不根据规则的依赖描述决定是否重建目标文件。
-C DIR，--directory=DIR	在读取 Makefile 之前，进入到目录 DIR，然后执行 make。当存在多个 "-C" 选项的时候，make 的最终工作目录是第一个目录的相对路径。
-d	make 在执行的过程中打印出所有的调试信息，包括 make 认为那些文件需要重建，那些文件需要比较最后的修改时间、比较的结果，重建目标是用的命令，遗憾规则等等。使用 "-d" 选项我们可以看到 make 构造依赖关系链、重建目标过程中的所有的信息。
--debug[=OPTIONS]	make 执行时输出调试信息，可以使用 "OPTIONS" 控制调试信息的级别。默认是 "OPTIONS=b" ，"OPTIONS" 的可值为以下这些，首字母有效：all、basic、verbose、implicit、jobs、makefile。
-e，--enverонment
-overrides	使用环境变量定义覆盖 Makefile 中的同名变量定义。
-f=FILE，--file=FILE，
--makefile=FILE	指定文件 "FILE" 为 make 执行的 Makefile 文件
-p，--help	打印帮助信息。
-i，--ignore-errors	执行过程中忽略规则命令执行的错误。
-I DIR，--include-dir=DIR	指定包含 Makefile 文件的搜索目录，在Makefile中出现另一个 "include" 文件时，将在 "DIR" 目录下搜索。多个 "-i" 指定目录时，搜索目录按照指定的顺序进行。
-j [JOBS]，--jobs[=JOBS]	可指定同时执行的命令数目，爱没有 "-j" 的情况下，执行的命令数目将是系统允许的最大可能数目，存在多个 "-j" 目标时，最后一个目标指定的 JOBS 数有效。
-k，--keep-going	执行命令错误时不终止 make 的执行，make 尽最大可能执行所有的命令，直至出现知名的错误才终止。
-l load，--load-average=[=LOAD]，--max-load[=LOAD]	告诉 make 在存在其他任务执行的时候，如果系统负荷超过 "LOAD"，不在启动新的任务。如果没有指定 "LOAD" 的参数 "-l" 选项将取消之前 "-l" 指定的限制。
-n，--just-print，--dry-run	只打印执行的命令，但是不执行命令。
-o FILE，--old-file=FILE，
--assume-old=FILE	指定 "FILE"文件不需要重建，即使是它的依赖已经过期；同时不重建此依赖文件的任何目标。注意：此参数不会通过变量 "MAKEFLAGS" 传递给子目录进程。
-p，--print-date-base	命令执行之前，打印出 make 读取的 Makefile 的所有数据，同时打印出 make 的版本信息。如果只需要打印这些数据信息，可以使用 "make -qp" 命令，查看 make 执行之前预设的规则和变量，可使用命令 "make -p -f /dev/null"
-q，-question	称为 "询问模式" ；不运行任何的命令，并且无输出。make 只返回一个查询状态。返回状态 0 表示没有目标表示重建，返回状态 1 表示存在需要重建的目标，返回状态 2 表示有错误发生。
-r，--no-builtin-rules	取消所有的内嵌函数的规则，不过你可以在 Makefile 中使用模式规则来定义规则。同时选项 "-r" 会取消所有后缀规则的隐含后缀列表，同样我们可以在 Makefile 中使用 ".SUFFIXES"，定义我们的后缀名的规则。"-r" 选项不会取消 make 内嵌的隐含变量。
-R，--no-builtin-variabes	取消 make 内嵌的隐含变量，不过我们可以在 Makefile 中明确定义某些变量。注意："-R" 和 "-r" 选项同时打开，因为没有了隐含变量，所以隐含规则将失去意义。
-s，--silent，--quiet	取消命令执行过程中的打印。
-S，--no-keep-going，
--stop	取消 "-k" 的选项在递归的 make 过程中子 make 通过 "MAKEFLAGS" 变量继承了上层的命令行选项那个。我们可以在子 make 中使用 “-S” 选项取消上层传递的 "-k" 选项，或者取消系统环境变量 "MAKEFLAGS" 中 "-k"选项。
-t，--touch	和 Linux 的 touch 命令实现功能相同，更新所有的目标文件的时间戳到当前系统时间。防止 make 对所有过时目标文件的重建。
-v，version	查看make的版本信息。
-w，--print-directory	在 make 进入一个子目录读取 Makefile 之前打印工作目录，这个选项可以帮助我们调试 Makefile，跟踪定位错误。使用 "-C" 选项时默认打开这个选项。
--no-print-directory	取消 "-w" 选项。可以是 用在递归的 make 调用的过程中 ， 取消 "-C" 参数的默认打开 "-w" 的功能。

| -W FILE，--what-if=FILE，
--new-file=FILE，
--assume-file=FILE | 设定文件 "FILE" 的时间戳为当前的时间，但不更改文件实际的最后修改时间。此选项主要是为了实现对所有依赖于文件 "FILE" 的目标的强制重建。 |
| --warn-undefined-variables | 在发现 Makefile 中存在没有定义的变量进行引用时给出告警信息。此功能可以帮助我们在调试一个存在多级嵌套变量引用的复杂 Makefile。但是建议在书写的时候尽量避免超过三级以上的变量嵌套引用。 |</byte-sheet-html-origin>

Makefile目标类型大汇总

模式规则中的目标。规则中的目标形式是多种多样的，它可以是一个或多个的文件、可以是一个伪目标，这是我们之前讲到过的，也是经常使用的。其实规则目标还可以是其他的类型，下面是对这些类型的详细的说明。

强制目标

如果一个目标中没有命令或者是依赖，并且它的目标不是一个存在的文件名，在执行此规则时，目标总会被认为是最新的。就是说：这个规则一旦被执行，make 就认为它的目标已经被更新过。这样的目标在作为一个规则的依赖时，因为依赖总被认为更新过，因此作为依赖在的规则中定义的命令总会被执行。看一个例子：

```
clean:FORCE
    rm $(OBJECTS)
FORCE:
```

这个例子中，目标 "FORCE" 符合上边的条件。它作为目标 "clean" 的依赖，在执行 make 的时候，总被认为更新过。因此 "clean" 所在的规则而在被执行其所定义的那个命令总会被执行。这样的一个目标通常我们将其命名为 "FORCE"。

例子中使用 "FORCE" 目标的效果和将 "clean" 声明为伪目标的效果相同。

空目标文件

空目标文件是伪目标的一个变种，此目标所在的规则执行的目的和伪目标相同——通过 make 命令行指定将其作为终极目标来执行此规则所定义的命令。和伪目标不同的是：这个目标可以是一个存在的文件，但文件的具体内容我们并不关心，通常此文件是一个空文件。

空目标文件只是用来记录上一次执行的此规则的命令的时间。在这样的规则中，命令部分都会使用 "touch" 在完成所有的命令之后来更新目标文件的时间戳，记录此规则命令的最后执行时间。make 时通过命令行将此目标作为终极目标，当前目标下如果不存在这个文件，"touch" 会在第一次执行时创建一个的文件。

通常，一个空目标文件应该存在一个或者多个依赖文件。将这个目标作为终极目标，在它所依赖的文件比它更新时，此目标所在的规则的命令行将被执行。就是说如果空目标文件的依赖文件被改变之后，空目标文件所在的规则中定义的命令会被执行。看一个例子：

```
print:foot.c bar.c
    lpr -p $?
    touch print
```

执行 "make print" ,当目标文件 "print" 的依赖文件被修改之后，命令 "lpr -p $?" 都会被执行，打印这个被修改的文件。

特殊的目标

<style> td {white-space:pre-wrap;border:1px solid #dee0e3;}</style> <byte-sheet-html-origin data-id="iPb9ly6NoW-1623856826609" data-version="3" data-is-embed="true"><colgroup><col width="209"><col width="1201"></colgroup>
| 名称 | 功能 |
| .PHONY: | 这个目标的所有依赖被作为伪目标。伪目标是这样一个目标：当使用 make 命令行指定此目标时，这个目标所在的规则定义的命

令、无论目标文件是否存在都会被无条件执行。 |
.SUFFIXES:	这个目标的所有依赖指出了一系列在后缀规则中需要检查的后缀名
.DEFAULT:	Makefile 中，这个特殊目标所在规则定义的命令，被用在重建那些没有具体规则的目标，就是说一个文件作为某个规则的依赖，却不是另外一个规则的目标时，make 程序无法找到重建此文件的规则，这种情况就执行 ".DEFAULT" 所指定的命令。
.PRECIOUS:	这个特殊目标所在的依赖文件在 make 的过程中会被特殊处理：当命令执行的过程中断时，make 不会删除它们。而且如果目标的依赖文件是中间过程文件，同样这些文件不会被删除。
.INTERMEDIATE:	这个特殊目标的依赖文件在 make 执行时被作为中间文件对待。没有任何依赖文件的这个目标没有意义。
.SECONDARY:	这个特殊目标的依赖文件被作为中过程的文件对待。但是这些文件不会被删除。这个目标没有任何依赖文件的含义是：将所有的文件视为中间文件。
.IGNORE	这个目标的依赖文件忽略创建这个文件所执行命令的错误，给此目标指定命令是没有意义的。当此目标没有依赖文件时，将忽略所有命令执行的错误。
.DELETE_ON_ERROR:	如果在 Makefile 中存在特殊的目标 ".DELETE_ON_ERROR" ，make 在执行过程中，荣国规则的命令执行错误，将删除已经被修改的目标文件。
.LOW_RESOLUTION_TIME:	这个目标的依赖文件被 make 认为是低分辨率时间戳文件，给这个目标指定命令是没有意义的。通常的目标都是高分辨率时间戳。
.SILENT:	出现在此目标 ".SILENT" 的依赖文件列表中的文件，make 在创建这些文件时，不打印出此文件所执行的命令。同样，给目标 "SILENT" 指定命令行是没有意义的。
.EXPORT_ALL_VARIABLES:	此目标应该作为一个简单的没有依赖的目标，它的功能是将之后的所有变量传递给子 make 进程。
.NOTPARALLEL:	Makefile 中如果出现这个特殊目标，则所有的命令按照串行的方式执行，即使是存在 make 的命令行参数 "-j" 。但在递归调用的子make进程中，命令行可以并行执行。此目标不应该有依赖文件，所有出现的依赖文件将会被忽略。 |</byte-sheet-html-origin>

多规则目标

Makefile 中，一个文件可以作为多个规则的目标。这种情况时，以这个文件为目标的规则的所有依赖文件将会被合并成此目标一个依赖文件列表，当其中的任何一个依赖文件比目标更新时，make 将会执行特定的命令来重建这个目标。

对于一个多规则的目标，重建这个目标的命令只能出现在一个规则中。如果多个规则同时给出重建此目标的命令，make 将使用最后一个规则中所定义的命令，同时提示错误信息。某些情况，需要对相同的目标使用不同的规则中所定义的命令，我们需要使用另一种方式——双冒号规则来实现。

一个仅仅描述依赖关系的描述规则可以用来给出一个或者时多个目标文件的依赖文件。例如，Makefile 中通常存在一个变量，就像我们以前提到的 "objects" ,它定义为所有的需要编译的生成 .o 文件的列表。这些 .o 文件在其源文件中包含的头文件 "config.h" 发生变化之后能够自动的被重建，我们可以使用多目标的方式来书写 Makefile:

```
objects=foo.o bar.o
foo.o:defs.h
bar.o:defs.h test.h
$(objects):config.h
```

这样做的好处是：我们可以在源文件增加或者删除了包含的头文件以后不用修改已存在的 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。

我们也可以通过一个变量来增加目标的依赖文件，使用 make 的命令行来指定某一个目标的依赖头文件，例如:

```
extradeps=
$(objects):$(exteradeps)
```

它的意思是：如果我们执 "make exteradeps=foo.h" 那么 "foo.h" 将作为所有的 .o 文件的依赖文件。当然如果只执行 "make" 的话，就没有指定任何文件作为 .o 文件的依赖文件。

Makefile变量的高级用法

变量的替换引用

我们定义变量的目的是为了简化我们的书写格式，代替我们在代码中频繁出现且冗杂的部分。它可以出现在我们规则的目标中，也可以是我们规则的依赖中。我们使用的时候会经常的对它的值（表示的字符串）进行操作。遇到这样的问题我们可能会想到我们的字符串操作函数，比如 "patsubst" 就是我们经常使用的。但是我们使用变量同样可以解决这样的问题，我们通过下面的例子来具体的分析一下。

实例：

```
foo:=a.c b.c d.c
obj:=$(foo:.c=.o)
All:
    @echo $(obj)
```

这段代码实现的功能是字符串的后缀名的替换，把变量 foo 中所有的以 .c 结尾的字符串全部替换成 .o 结尾的字符串。我们在 Makefile 中这样写，然后再 shell 命令行执行 make 命令，就可以看到打印出来的是 "a.o b.o d.o" ，实现了文件名后缀的替换。注意：括号中的变量使用的是变量名而不是变量名的引用，变量名的后面要使用冒号和参数选项分开，表达式中间不能使用空格。第二个变量 obj 是对整体的引用。

上面的例子我们可以换一种更加通用的方式来写，代码展示如下：

```
foo:=a.c b.c d.c
obj:=$(foo:%.c=%.o)
All:
    @echo $(obj)
```

我们在 shell 中执行 make 命令，发现结果是相同的。

对比上面的实例我们可以看到，表达式中使用了 "%" 这个字符，这个字符的含义就是自动匹配一个或多个字符。在开发的过程中，我们通常会使用这种方式来进行变量替换引用的操作。

为什么这种方式比第一种方式更加实用呢？我们在实际使用的过程中，我们对变量值的操作不只是修改其中的一个部分，甚至是改变其中的多个，那么第一种方式就不能实现了。我们来看一下这种情况：

```
foo:=a123c a1234c a12345c
obj:=$(foo:a%c=x%y)
All:
    @echo $(obj)
```

我们可以看到这个例子中我们操作的是两个不连续的部分，我们执行 make 后打印的值是 "x123y x1234y x12345y"，这种情况下我们使用第一种情况就不能实现，所以第二种的使用更全面。

变量的嵌套使用

变量的嵌套引用的具体含义是这样的，我们可以在一个变量的赋值中引用其他的变量，并且引用变量的数量和和次数是不限制的。下面我们通过几个实例来说明一下。

实例 1：

```
foo:=test
var:=$(foo)
All:
    @echo $(var)
```

这种用法是最常见的使用方法，打印出 var 的值就是 test。我们可以认为是一层的嵌套引用。

实例 2:

```
foo=bar
var=test
var:=$($(foo))
All:
    @echo $(var)
```

我们再去执行 make 命令的时候得到的结果也是 test，我们可以来分析一下这段代码执行的过程：$(foo) 代表的字符串是 bar，我们也定义了变量 bar，所以我们可以对 bar 进行引用，变量 bar 表示的值是 test，所以对 bar 的引用就是 test，所以最终 var 的值就是 test。这是变量的二层嵌套执行，当然我们还可以使用三层的嵌套执行，写法跟上面的方式是一样的。嵌套的层数也可以更多，但是不提倡使用。

我们再去使用变量的时候，我们并不是只能引用一个变量，可以有多个变量的引用，还可以包含很多的变量还可以是一些文本字符。我们可以通过一些例子来说明一下。

实例 4:

```
first_pass=hello
bar=first
var:=$(bar)_pass
all:
    @echo $(var)
```

在命令行执行 make 我们可以得到 var 的值是 hello。这是变量嵌套引用的时候可以包含其它字符的使用情况。

实例 5:

```
first_pass=hello
bar=first
foo=pass
var:=$(bar)_$(foo)
all:
    @echo $(var)
```

这个实例跟上面实例的运行结果是一样的。我们可以看到这个实例中使用了两个变量的引用还有其它的字符。

变量的嵌套引用和我们的变量的递归赋值的区别：嵌套引用的使用方法就是用一个变量表示另外一个变量，然后进行多层的引用。而递归展开的变量表示当一个变量存在对其它变量的引用时，对这变量替换的方式。递归展开在另外一个角度描述了这个变量在定义是赋予它的一个属性或者风格。并且我们可以在定义个一个递归展开式的变量时使用套嵌引用的方式，但是建议你的实际编写 Makefile 时要尽量避免这种复杂的用法。

在实际使用的过程中变量的第一种用法经常使用的，第二种用法我们很少使用，应该说是尽量避免使用变量的嵌套引用。在必须要使用的时候我们应该做到嵌套的层数是越少越好的。因为使用这种方法表达会比较的复杂，如果条理不清楚的话我们就会出错。并且在给其他人看的时候也会不容易理解。

Makefile控制函数error和warning

Makefile 中提供了两个控制 make 运行方式的函数。其作用是当 make 执行过程中检测到某些错误时为用户提供消息，并且可以控制 make 执行过程是否继续。这两个函数是 "error" 和 "warning"，我们来详细的介绍一下这两个函数。

```
$(error TEXT...)
```

函数说明如下：

- 函数功能：产生致命错误，并提示 "TEXT..." 信息给用户，并退出 make 的执行。需要说明的是："error" 函数是在函数展开时（函数被调用时）才提示信息并结束 make 进程。因此如果函数出现在命令中或者一个递归的变量定义时，读取 Makefile 时不会出现错误。而只有包含 "error" 函数引用的命令被执行，或者定义中引用此函数的递归变量被展开时，才会提示知名信息 "TEXT..." 同时退出 make。
- 返回值：空
- 函数说明："error" 函数一般不出现在直接展开式的变量定义中，否则在 make 读取 Makefile 时将会提示致命错误。

我们通过两个例子来说明一下；

实例 1：

```
ERROR1=1234
all:
   ifdef ERROR1
   $(error error is $(ERROR1))
   endif
```

make 读取解析 Makefile 时，如果所起的变量名是已经定义好的"ERROR1"，make 将会提示致命错误信息 "error is 1234" 并保存退出。

实例 2：

```
ERR=$(error found an error!)
.PHONY:err
err:;$(ERR)
```

这个例子，在 make 读取 Makefile 时不会出现致命错误。只有目标 "err" 被作为是一个目标被执行时才会出现。

```
$(warning TEXT...)
```

函数说明如下：

- 函数功能：函数 "warning" 类似于函数 "error"，区别在于它不会导致致命错误（make不退出），而只是提示 "TEXT..."，make 的执行过程继续。
- 返回值：空
- 函数说明：用法和 "error" 类似，展开过程相同。

Makefile中常见的错误信息

make 执行过程的致命错误都带有前缀字符串 "***"。错误信息都有前缀，一种是执行程序名作为错误前缀（通常是 "make"）；另外一种是当 Makefile 本身存在语法错误无法被 make 解析并执行时，前缀包含了 Makefile 文件名和出现错误的行号。

在下述的错误列表中，省略了普通前缀：

```
[FOO] Error NN
[FOO] signal description
```

这类错误并不是 make 的真正错误。它表示 make 检测到 make 所调用的作为执行命令的程序返回一个非零状态（Error NN），或者此命令程序以非正常方式退出（携带某种信号）。

如果错误信息中没有附加 "***" 字符串，则是子过程的调用失败，如果 Makefile 中此命令有前缀 "-"，make 会忽略这个错误。

```
missing separator. Stop.
missing separator (did you mean TAB instead of 8 spaces?). Stop.
```

错误的原因：不可识别的命令行，make 在读取 Makefile 过程中不能解析其中包含的内容。GNU make在读取 Makefile 时根据各种分隔符（:, =, [TAB]字符等）来识别 Makefile 的每一行内容。这些错误意味着 make 不能发现一个合法的分隔符。

出现这些错误信息的可能的原因是（或许是编辑器，绝大部分是ms- windows的编辑器）在 Makefile 中的命令之前使用了4个（或者8个）空格代替了 [Tab] 字符。这种情况，将产生上述的第二种形式产生错误信息。且记，所有的命令行都应该是以 [Tab] 字符开始的。

```
commands commence before first target. Stop.
missing rule before commands. Stop.
```

Makefile 可能是以命令行开始：以 [Tab] 字符开始，但不是一个合法的命令行（例如，一个变量的赋值）。命令行必须和规则一一对应。

产生第二种的错误的原因可能是一行的第一个非空字符为分号，make 会认为此处遗漏了规则的 "target: prerequisite" 部分。

```
No rule to make target 'XXX'.
No rule to make target 'XXX ', needed by 'yyy'.
```

无法为重建目标“XXX”找到合适的规则，包括明确规则和隐含规则。

修正这个错误的方法是：在 Makefile 中添加一个重建目标的规则。其它可能导致这些错误的原因是 Makefile 中文件名拼写错误，或者破坏了源文件树（一个文件不能被重建，可能是由于依赖文件的问题）。

```
No targets specified and no makefile found. Stop.
No targets. Stop.
```

第一个错误表示在命令行中没有指定需要重建的目标，并且 make 不能读入任何 Makefile 文件。第二个错误表示能够找到 Makefile 文件，但没有终极目标或者没有在命令行中指出需要重建的目标。这种情况下，make 什么也不做。

```
Makefile 'XXX' was not found.
Included makefile 'XXX' was not found.
```

没有使用 "-f" 指定 Makefile 文件，make 不能在当前目录下找到默认 Makefile（makefile 或者 GNUmakefile）。使用 "-f" 指定文件，但不能读取这个指定的 Makefile 文件。

```
warning: overriding commands for target 'XXX'
warning: ignoring old commands for target 'XXX'
```

对同一目标 "XXX" 存在一个以上的重建命令。GNU make 规定：当同一个文件作为多个规则的目标时，只能有一个规则定义重建它的命令（双冒号规则除外）。如果为一个目标多次指定了相同或者不同的命令，就会产生第一个告警；第二个告警信息说新指定的命令覆盖了上一次指定的命令。

```
Circular XXX <- YYY dependency dropped.
```

规则的依赖关系产生了循环：目标 "XXX" 的依赖文件为 "YYY"，而依赖 "YYY" 的依赖列表中又包含 "XXX"。Recursive variable 'XXX' references itself (eventually). Stop.

make 的变量 "XXX"（递归展开式）在替换展开时，引用它自身。无论对于直接展开式变量（通过:=定义的）或追加定义（+=），这都是不允许的。

```
Unterminated variable reference. Stop.
```

变量或者函数引用语法不正确，没有使用完整的的括号（缺少左括号或者右括号）。

```
insufficient arguments to function 'XXX'. Stop.
```

函数 "XXX" 引用时参数数目不正确。函数缺少参数。

```
missing target pattern. Stop.
multiple target patterns. Stop.
target pattern contains no '%'. Stop.
mixed implicit and static pattern rules. Stop.
```

不正确的静态模式规则。

第一条错误的原因是：静态模式规则的目标段中没有模式目标;

第二条错误的原因是：静态模式规则的目标段中存在多个模式目标;

第三条错误的原因是：静态模式规则的目标段目标模式中没有包含模式字符“%”；

第四条错误的原因是：静态模式规则的三部分都包含了模式字符“%”。正确的应该是只有后两个才可以包含模式字符“%”。

```
warning: -jN forced in submake: disabling jobserver mode.
```

这一条告警和下条告警信息发生在：make 检测到递归的 make 调用时，可通信的子 make 进程出现并行处理的错误。递归执行的 make 的命令行参数中存在 "-jN" 参数（N的值大于1），在有些情况下可能导致此错误，例如：Makefile 中变量 "MAKE" 被赋值为 "make -j2"，并且递归调用的命令行中使用变量 "MAKE"。在这种情况下，被调用 make 进程不能和其它 make 进程进行通信，其只能简单的独立的并行处理两个任务”。warning: jobserver unavailable: using -j1. Add '+' to parent make rule.

为了现实 make 进程之间的通信，上层 make 进程将传递信息给子 make 进程。在传递信息过程中可能存在这种情况，子 make 进程不是一个实际的 make 进程，而上层make却不能确定子进程是否是真实的 make 进程。它只是将所有信息传递下去。上层 make 采用正常的算法来决定这些。当出现这种情况，子进程只会接受父进程传递的部分有用的信息。子进程会产生该警告信息，之后按照其内建的顺序方式进行处理。

扩展

EXTRA_CFLAGS详解