# 隐含规则

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

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

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

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

## 使用隐含规则

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

foo : foo.o bar.o

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

我们可以注意到，这个Makefile中并没有写下如何生成foo.o和bar.o这两目标的规则和命令。因为make的“隐含规则”功能会自动为我们自动去推导这两个目标的依赖目标和生成命令。

make会在自己的“隐含规则”库中寻找可以用的规则，如果找到，那么就会使用。如果找不到，那么就会报错。在上面的那个例子中，make调用的隐含规则是，把[.o]的目标的依赖文件置成[.c]，并使用C的编译命令“cc –c $(CFLAGS) [.c]”来生成[.o]的目标。也就是说，我们完全没有必要写下下面的两条规则：

foo.o : foo.c

cc –c foo.c $(CFLAGS)

bar.o : bar.c

cc –c bar.c $(CFLAGS)

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

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

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

foo.o : foo.p

依赖文件“foo.p”（Pascal程序的源文件）有可能变得没有意义。如果目录下存在了“foo.c”文件，那么我们的隐含规则一样会生效，并会通过“foo.c”调用C的编译器生成foo.o文件。因为，在隐含规则中，Pascal的规则出现在C的规则之后，所以，make找到可以生成 foo.o的C的规则就不再寻找下一条规则了。如果你确实不希望任何隐含规则推导，那么，你就不要只写出“依赖规则”，而不写命令。

## 隐含规则一览

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

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

还是先来看一看常用的隐含规则吧。

1、编译C程序的隐含规则。

“<n>;.o”的目标的依赖目标会自动推导为“<n>;.c”，并且其生成命令是“$(CC) –c $(CPPFLAGS) $(CFLAGS)”

2、编译C++程序的隐含规则。

“<n>;.o”的目标的依赖目标会自动推导为“<n>;.cc”或是“<n>;.C”，并且其生成命令是 “$(CXX) –c $(CPPFLAGS) $(CFLAGS)”。（建议使用“.cc”作为C++源文件的后缀，而不是“.C”）

3、编译Pascal程序的隐含规则。

“<n>;.o”的目标的依赖目标会自动推导为“<n>;.p”，并且其生成命令是“$(PC) –c $(PFLAGS)”。

4、编译Fortran/Ratfor程序的隐含规则。

“<n>;.o”的目标的依赖目标会自动推导为“<n>;.r”或“<n>;.F”或“<n>;.f”，并且其生成命令是:

“.f” “$(FC) –c $(FFLAGS)”

“.F” “$(FC) –c $(FFLAGS) $(CPPFLAGS)”

“.f” “$(FC) –c $(FFLAGS) $(RFLAGS)”

5、预处理Fortran/Ratfor程序的隐含规则。

“<n>;.f”的目标的依赖目标会自动推导为“<n>;.r”或“<n>;.F”。这个规则只是转换Ratfor或有预处理的Fortran程序到一个标准的Fortran程序。其使用的命令是：

“.F” “$(FC) –F $(CPPFLAGS) $(FFLAGS)”

“.r” “$(FC) –F $(FFLAGS) $(RFLAGS)”

6、编译Modula-2程序的隐含规则。

“<n>;.sym”的目标的依赖目标会自动推导为“<n>;.def”，并且其生成命令是：“$(M2C) $ (M2FLAGS) $(DEFFLAGS)”。“<n.o>;” 的目标的依赖目标会自动推导为“<n>;.mod”，并且其生成命令是：“$(M2C) $(M2FLAGS) $(MODFLAGS)”。

7、汇编和汇编预处理的隐含规则。

“<n>;.o” 的目标的依赖目标会自动推导为“<n>;.s”，默认使用编译品“as”，并且其生成命令是：“$ (AS) $(ASFLAGS)”。“<n>;.s” 的目标的依赖目标会自动推导为“<n>;.S”，默认使用C预编译器 “cpp”，并且其生成命令是：“$(AS) $(ASFLAGS)”。

8、链接Object文件的隐含规则。

“<n>;”目标依赖于“<n>;.o”，通过运行C的编译器来运行链接程序生成（一般是“ld”），其生成命令是： “$(CC) $(LDFLAGS) <n>;.o $(LOADLIBES) $(LDLIBS)”。这个规则对于只有一个源文件的工程有效，同时也对多个Object文件（由不同的源文件生成）的也有效。例如如下规则：

x : y.o z.o

并且“x.c”、“y.c”和“z.c”都存在时，隐含规则将执行如下命令：

cc -c x.c -o x.o

cc -c y.c -o y.o

cc -c z.c -o z.o

cc x.o y.o z.o -o x

rm -f x.o

rm -f y.o

rm -f z.o

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

9、Yacc C程序时的隐含规则。

“<n>;.c”的依赖文件被自动推导为“n.y”（Yacc生成的文件），其生成命令是：“$(YACC) $(YFALGS)”。（“Yacc”是一个语法分析器，关于其细节请查看相关资料）

10、Lex C程序时的隐含规则。

“<n>;.c”的依赖文件被自动推导为“n.l”（Lex生成的文件），其生成命令是：“$(LEX) $(LFALGS)”。（关于“Lex”的细节请查看相关资料）

11、Lex Ratfor程序时的隐含规则。

“<n>;.r”的依赖文件被自动推导为“n.l”（Lex生成的文件），其生成命令是：“$(LEX) $(LFALGS)”。

12、从C程序、Yacc文件或Lex文件创建Lint库的隐含规则。

“<n>;.ln” （lint生成的文件）的依赖文件被自动推导为“n.c”，其生成命令是：“$(LINT) $(LINTFALGS) $(CPPFLAGS) -i”。对于“<n>;.y”和“<n>;.l”也是同样的规则。

## 隐含规则使用的变量

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

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

我们可以把隐含规则中使用的变量分成两种：一种是命令相关的，如“CC”；一种是参数相的关，如“CFLAGS”。下面是所有隐含规则中会用到的变量：

1、关于命令的变量。

AR

函数库打包程序。默认命令是“ar”。

AS

汇编语言编译程序。默认命令是“as”。

CC

C语言编译程序。默认命令是“cc”。

CXX

C++语言编译程序。默认命令是“g++”。

CO

从 RCS文件中扩展文件程序。默认命令是“co”。

CPP

C程序的预处理器（输出是标准输出设备）。默认命令是“$(CC) –E”。

FC

Fortran 和 Ratfor 的编译器和预处理程序。默认命令是“f77”。

GET

从SCCS文件中扩展文件的程序。默认命令是“get”。

LEX

Lex方法分析器程序（针对于C或Ratfor）。默认命令是“lex”。

PC

Pascal语言编译程序。默认命令是“pc”。

YACC

Yacc文法分析器（针对于C程序）。默认命令是“yacc”。

YACCR

Yacc文法分析器（针对于Ratfor程序）。默认命令是“yacc –r”。

MAKEINFO

转换Texinfo源文件（.texi）到Info文件程序。默认命令是“makeinfo”。

TEX

从TeX源文件创建TeX DVI文件的程序。默认命令是“tex”。

TEXI2DVI

从Texinfo源文件创建军TeX DVI 文件的程序。默认命令是“texi2dvi”。

WEAVE

转换Web到TeX的程序。默认命令是“weave”。

CWEAVE

转换C Web 到 TeX的程序。默认命令是“cweave”。

TANGLE

转换Web到Pascal语言的程序。默认命令是“tangle”。

CTANGLE

转换C Web 到 C。默认命令是“ctangle”。

RM

删除文件命令。默认命令是“rm –f”。

2、关于命令参数的变量

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

ARFLAGS

函数库打包程序AR命令的参数。默认值是“rv”。

ASFLAGS

汇编语言编译器参数。（当明显地调用“.s”或“.S”文件时）。

CFLAGS

C语言编译器参数。

CXXFLAGS

C++语言编译器参数。

COFLAGS

RCS命令参数。

CPPFLAGS

C预处理器参数。（ C 和 Fortran 编译器也会用到）。

FFLAGS

Fortran语言编译器参数。

GFLAGS

SCCS “get”程序参数。

LDFLAGS

链接器参数。（如：“ld”）

LFLAGS

Lex文法分析器参数。

PFLAGS

Pascal语言编译器参数。

RFLAGS

Ratfor 程序的Fortran 编译器参数。

YFLAGS

Yacc文法分析器参数。

## 隐含规则链

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

在上面的例子中，如果文件[.c]存在，那么就直接调用C的编译器的隐含规则，如果没有[.c]文件，但有一个[.y]文件，那么Yacc的隐含规则会被调用，生成[.c]文件，然后，再调用C编译的隐含规则最终由[.c]生成[.o]文件，达到目标。

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

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

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

你也可以阻止make自动删除中间目标，要做到这一点，你可以使用伪目标“.SECONDARY”来强制声明（如：.SECONDARY : sec）。你还可以把你的目标，以模式的方式来指定（如：%.o）成伪目标“.PRECIOUS”的依赖目标，以保存被隐含规则所生成的中间文件。

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

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

## 定义模式规则

你可以使用模式规则来定义一个隐含规则。一个模式规则就好像一个一般的规则，只是在规则中，目标的定义需要有"%"字符。"%"的意思是表示一个或多个任意字符。在依赖目标中同样可以使用"%"，只是依赖目标中的"%"的取值，取决于其目标。

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

1、模式规则介绍

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

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

 %.o : %.c ; <command ......>;

其含义是，指出了怎么从所有的[.c]文件生成相应的[.o]文件的规则。如果要生成的目标是"a.o b.o"，那么"%c"就是"a.c b.c"。

一旦依赖目标中的"%"模式被确定，那么，make会被要求去匹配当前目录下所有的文件名，一旦找到，make就会规则下的命令，所以，在模式规则中，目标可能会是多个的，如果有模式匹配出多个目标，make就会产生所有的模式目标，此时，make关心的是依赖的文件名和生成目标的命令这两件事。

2、模式规则示例

下面这个例子表示了,把所有的[.c]文件都编译成[.o]文件.

 %.o : %.c

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

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

下面的这个例子中有两个目标是模式的：

 %.tab.c %.tab.h: %.y

bison -d $<

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

3、自动化变量

在上述的模式规则中，目标和依赖文件都是一系例的文件，那么我们如何书写一个命令来完成从不同的依赖文件生成相应的目标？因为在每一次的对模式规则的解析时，都会是不同的目标和依赖文件。

自动化变量就是完成这个功能的。在前面，我们已经对自动化变量有所提涉，相信你看到这里已对它有一个感性认识了。所谓自动化变量，就是这种变量会把模式中所定义的一系列的文件自动地挨个取出，直至所有的符合模式的文件都取完了。这种自动化变量只应出现在规则的命令中。

下面是所有的自动化变量及其说明：

$@

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

$%

仅当目标是函数库文件中，表示规则中的目标成员名。例如，如果一个目标是"foo.a(bar.o)"，那么，"$%"就是 "bar.o"，"$@"就是"foo.a"。如果目标不是函数库文件（Unix下是[.a]，Windows下是[.lib]），那么，其值为空。

$<

依赖目标中的第一个目标名字。如果依赖目标是以模式（即"%"）定义的，那么"$<"将是符合模式的一系列的文件集。注意，其是一个一个取出来的。

$?

所有比目标新的依赖目标的集合。以空格分隔。

$^

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

$+

这个变量很像"$^"，也是所有依赖目标的集合。只是它不去除重复的依赖目标。

$\*

这个变量表示目标模式中"%"及其之前的部分。如果目标是"dir/a.foo.b"，并且目标的模式是"a.%.b"，那么，"$\*"的值就是"dir/a.foo"。这个变量对于构造有关联的文件名是比较有用。如果目标中没有模式的定义，那么"$\*"也就不能被推导出，但是，如果目标文件的后缀是make所识别的，那么"$\*"就是除了后缀的那一部分。例如：如果目标是"foo.c"，因为".c"是make所能识别的后缀名，所以，" $\*"的值就是"foo"。这个特性是GNU make的，很有可能不兼容于其它版本的make，所以，你应该尽量避免使用"$\*"，除非是在隐含规则或是静态模式中。如果目标中的后缀是make所不能识别的，那么"$\*"就是空值。

当你希望只对更新过的依赖文件进行操作时，"$?"在显式规则中很有用，例如，假设有一个函数库文件叫"lib"，其由其它几个object文件更新。那么把object文件打包的比较有效率的Makefile规则是：

lib : foo.o bar.o lose.o win.o

ar r lib $?

在上述所列出来的自动量变量中。四个变量（$@、$<、$%、$\*）在扩展时只会有一个文件，而另三个的值是一个文件列表。这七个自动化变量还可以取得文件的目录名或是在当前目录下的符合模式的文件名，只需要搭配上"D"或"F"字样。这是GNU make中老版本的特性，在新版本中，我们使用函数"dir"或"notdir"就可以做到了。"D"的含义就是Directory，就是目录，"F"的含义就是File，就是文件。

下面是对于上面的七个变量分别加上"D"或是"F"的含义：

$(@D)

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

$(@F)

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

"$(\*D)"

"$(\*F)"

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

"$(%D)"

"$(%F)"

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

"$(<D)"

"$(<F)"

分别表示依赖文件的目录部分和文件部分。

"$(^D)"

"$(^F)"

分别表示所有依赖文件的目录部分和文件部分。（无相同的）

"$(+D)"

"$(+F)"

分别表示所有依赖文件的目录部分和文件部分。（可以有相同的）

"$(?D)"

"$(?F)"

分别表示被更新的依赖文件的目录部分和文件部分。

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

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

4、模式的匹配

一般来说，一个目标的模式有一个有前缀或是后缀的"%"，或是没有前后缀，直接就是一个"%"。因为"%"代表一个或多个字符，所以在定义好了的模式中，我们把"%"所匹配的内容叫做"茎"，例如"%.c"所匹配的文件"test.c"中"test"就是"茎"。因为在目标和依赖目标中同时有"% "时，依赖目标的"茎"会传给目标，当做目标中的"茎"。

当一个模式匹配包含有斜杠（实际也不经常包含）的文件时，那么在进行模式匹配时，目录部分会首先被移开，然后进行匹配，成功后，再把目录加回去。在进行"茎"的传递时，我们需要知道这个步骤。例如有一个模式"e%t"，文件"src/eat"匹配于该模式，于是"src/a"就是其"茎"，如果这个模式定义在依赖目标中，而被依赖于这个模式的目标中又有个模式"c%r"，那么，目标就是"src/car"。（"茎"被传递）

5、重载内建隐含规则

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

 %.o : %.c

$(CC) -c $(CPPFLAGS) $(CFLAGS) -D$(date)

你可以取消内建的隐含规则，只要不在后面写命令就行。如：

 %.o : %.s

同样，你也可以重新定义一个全新的隐含规则，其在隐含规则中的位置取决于你在哪里写下这个规则。朝前的位置就靠前。

## 老式风格的"后缀规则"

后缀规则是一个比较老式的定义隐含规则的方法。后缀规则会被模式规则逐步地取代。因为模式规则更强更清晰。为了和老版本的Makefile兼容，GNU make同样兼容于这些东西。后缀规则有两种方式："双后缀"和"单后缀"。

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

后缀规则中所定义的后缀应该是make所认识的，如果一个后缀是make所认识的，那么这个规则就是单后缀规则，而如果两个连在一起的后缀都被 make所认识，那就是双后缀规则。例如：".c"和".o"都是make所知道。因而，如果你定义了一个规则是".c.o"那么其就是双后缀规则，意义就是".c"是源文件的后缀，".o"是目标文件的后缀。如下示例：

.c.o:

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

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

.c.o: foo.h

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

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

 %.o: %.c foo.h

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

后缀规则中，如果没有命令，那是毫无意义的。因为他也不会移去内建的隐含规则。

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

.SUFFIXES: .hack .win

把后缀.hack和.win加入后缀列表中的末尾。

.SUFFIXES: # 删除默认的后缀

.SUFFIXES: .c .o .h # 定义自己的后缀

先清除默认后缀，后定义自己的后缀列表。

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

## 隐含规则搜索算法

比如我们有一个目标叫 T。下面是搜索目标T的规则的算法。请注意，在下面，我们没有提到后缀规则，原因是，所有的后缀规则在Makefile被载入内存时，会被转换成模式规则。如果目标是"archive(member)"的函数库文件模式，那么这个算法会被运行两次，第一次是找目标T，如果没有找到的话，那么进入第二次，第二次会把"member"当作T来搜索。

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

2、创建所有匹配于T或是N的模式规则列表。

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

4、移除列表中没有命令的规则。

5、对于第一个在列表中的模式规则：

1）推导其"茎"S，S应该是T或是N匹配于模式中"%"非空的部分。

2）计算依赖文件。把依赖文件中的"%"都替换成"茎"S。如果目标模式中没有包含斜框字符，而把D加在第一个依赖文件的开头。

3）测试是否所有的依赖文件都存在或是理当存在。（如果有一个文件被定义成另外一个规则的目标文件，或者是一个显式规则的依赖文件，那么这个文件就叫"理当存在"）

4）如果所有的依赖文件存在或是理当存在，或是就没有依赖文件。那么这条规则将被采用，退出该算法。

6、如果经过第5步，没有模式规则被找到，那么就做更进一步的搜索。对于存在于列表中的第一个模式规则：

1）如果规则是终止规则，那就忽略它，继续下一条模式规则。

2）计算依赖文件。（同第5步）

3）测试所有的依赖文件是否存在或是理当存在。

4）对于不存在的依赖文件，递归调用这个算法查找他是否可以被隐含规则找到。

5）如果所有的依赖文件存在或是理当存在，或是就根本没有依赖文件。那么这条规则被采用，退出该算法。

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

一旦规则被找到，就会执行其相当的命令，而此时，我们的自动化变量的值才会生成。