# GNU Make

**翻译**：loverszhaokai

最新版文档请参考github：

<https://github.com/loverszhaokai/GNUMakeManual_CN>

欢迎大家提出修改意见！谢谢！自由加油！

**原文**：<https://www.gnu.org/software/make/manual/>

**参考**：

1. 徐海兵 <http://www.yayu.org/book/gnu_make/>
2. 陈皓 <http://blog.csdn.net/haoel/article/details/2886>

**关于本手册的声明**：

本文瑾献给所有热爱Linux的程序员！本文档版权所有，禁止用于任何商业行为。（向徐海兵致敬！）

# make概览

在一个庞大的程序中，make命令自动决定了哪些文件需要重新编译，和重新编译它们的步骤。本手册介绍GNU make，它是由Richard Stallman和Roland McGrath实现的。开发工作从3.76版就由Paul D.Smith接手了。

GNU make遵循IEEE 标准1003.2-1992(POSIX.2)的第6.2章。

我们的示例采用的是C语言程序，因为程序语言基本都相似，凡是能够在shell执行编译的程序设计语言，都可以使用make。实际上，make不局限于程序。你可以使用make来描述任何任务，只要该任务满足一下规则，当文件A依赖的文件BCD…变化时，文件A就需要被更新。

在使用make之前，你需要编写一个文件名为makefile的文件，这个makefile文件负责描述程序中文件间的关系，并且提供更新文件所需要的命令。很明显的，在一个程序中，object文件的更新需要很多源文件，可执行文件的更新需要使用很多object文件。

一旦存在一个适当的makefile文件，每一次你改变源文件，只需要在shell输入make就可以重新编译整个程序：

# make

make通过使用makefile中文件的关系以及文件最近修改时间来决定是否需要更新。对于每一个需要更新的文件，都会执行makefile中对应的命令。

你也可以通过命令行参数指定重新编译哪些文件，或者如何编译。参考第9章[如何执行make],第99页。

## 如何阅读该手册

如果你是make新手，或者你正在寻找一个综合的介绍，那么你可以阅读每一章的前面几个小节，略过其余小节。在每一章，前几节包括简介和综合信息，后几节包括专用的信息。第2章[Makefile简介]，第3页是例外，该章全都是简介。

如果你熟悉其他make工具，请看第13章[GNU make特性]，第143页，这一章列举了GNU make的加强，还有第14章[不兼容点和不支持的特性]，第147页，

这一章说明了仅有的一些GNU make不支持但其它make工具支持的特性。

快速总结，请看第9.7节[选项总览]，第104页，附录A[快速索引]，第165页，第4.8节[特殊目标]，第32页。

## 问题和Bugs

对于GNU make，如何你有问题或你认为你发现一个bug，请把它提交给开发人员；我们不能保证解决你提出的问题或bug，但是我们会竭尽全力。

提交bug前，请确定你确实发现了一个bug。仔细地阅读文档，查看你是否能这样做。如果你还不确定是否可以这样做，请提交给我们，因为这是文档的bug。

在你提交bug或尝试自己修复之前，尽可能的分离出最小的makefile文件，该文件可以重现bug。然后将该makefile和make的结果发送给我们，包括任何错误信息和警告信息。请不要改变这些信息：最好是通过剪切和黏贴的方式。在分离出最小的makefile文件的过程中，请不要使用任务收费的或者不正常的工具：你总是可以使用shell命令来完成相同的功能的。最后，请解释清楚你预期的效果，这将是我们确定这个问题是否已经出现在文档中。

一旦你有一个明确的问题，你可以通过以下途径提交。发送电子邮件到：

[bug-make@gnu.org](mailto:bug-make@gnu.org)

或者使用基于浏览器的项目管理工具：

<http://savannah.gnu.org/projects/make/>

除了上面这些信息，请指明你所使用的make的版本号，你可以使用’make --version’命令来获得。还有，你使用的机器的型号以及操作系统的信息。通过’make --help’的最后一行，你可以获得操作系统的信息。

# Makefile简介

你需要一个名字是”makefile”的文件来指定make如何工作。通常makefile指定make如何编译和链接一个程序。

在这一章中，我们将会讨论一个简单的makefile文件如何编译和链接*edit*工程，该工程包含8个C语言源文件和3个头文件。makefile还可以指定make如何执行各种各样的命令（例如，make clean，执行删除指定文件的操作）。这里有makefile更复杂的示例，请看附录C[复杂的Makefile]，第177页。

当make重新编译*edit*，每一个改变过的C语言源文件都必须重新编译。为了安全，如果头文件发生改变，每一个包含该头文件的C语言源文件必须重新编译。每一次编译，都会产生与源文件关联的object文件。最后，只要任何一个object文件被重新编译了，所有的object文件不管是否是新的还是旧的，都必须被重新链接，以生成新的执行文件*edit*。

## Makefile规则

一个简单的makefile包含如下规则：

*target* … : *prerequisites* …

*recipe*

…

…

一个*target*通常是指程序生成的文件的名字；*targets*可以是执行文件的名字或者object文件的名字。*target*还可以是动作的名字，例如’clean’（请看第4.5节）[伪造的*target*]，第29页）。

*prerequisites*是指用来生成*target*的一些文件，一个*target*通常依赖一些文件。

*recipe*是指一个操作，该操作由make来执行。一个*recipe*可能含有多个命令，这些命令可以都在同一行，也可以各自独立一行。**请注意：***recipe*的每一行开头都要有tab键！这是为了防止有人粗心大意。如果你仍坚持在*recipes*前面加入除tab外的字符，你可以设置.RECIPEPREFIX变量为你想要的字符。（请看第6.14节[特殊变量]，第73页）。

通常，如果*prerequisites*发生变化，那么make就会执行对应的*recipe*以重新生成*target*。但是，*target*可以没有*prerequisites*，例如*target* ’clean’就没有*prerequisites*。

一个规则指明了如何以及何时重新编译确定的*target*。make通过对*prerequisites*执行*recipe*来生成或者更新*target*。一个规则还可以指明如何以及何时执行一个操作。请看第4章[编写规则]，第21页。

一个makefile可能包含除了规则的其他信息，但是一个简单的makefile仅仅只需要包含规则。规则可能看起来比这个模板的复杂，但是这些规则都多少遵循这种形式。

## 一个简单的Makefile

这是一个简单的makefile，它描述了执行文件*edit*依赖的8个object文件，转而，依赖于8个C语言源文件和3个头文件。

在这个示例中，所有的C语言源文件都包含’defs.h’，但是只有定义了编辑命令的文件包含’command.h’，只有改变编辑器的文件包含’buffer.h’。

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

我们使用反斜杠’\’将一行分割成两行，这就像是一行一样，但是更利于阅读。请看第3.11节[分割长行]第12页。

使用该makefile创建名为edit的执行文件，输入：

# make

使用该makefile删除执行文件和所有的object文件，输入：

# make clean

在makefile示例中，*targets*包括执行文件’edit’，和object文件’main.o’, ‘kbd.o’。*prerequisites*包括文件’main.c’, ‘defs.h’。事实上，每一个’.o’文件既是*target*又是*prerequisite*。*recipes*包括’cc –c main.c’和’cc –c kbd.c’。

当*target*是一个文件，如果对应的任何一个*prerequisites*发生变化，*target*都需要被重新编译或链接。另外，任何本身是自动生成的*prerequisites*都应该先被更新。在这个示例中，’edit’依赖这8个源文件中的每一个；’main.o’依赖于源文件’main.c’和头文件’defs.h’。

*recipe*可能在包含*target*和*prerequisites*的每一行之后出现。它描述了如何更新*target*。一个tab字符（或者其他由.RECIPEPREFIX定义的字符）必须出现在*recipe*每行的开头，这样做是为了区分*recipe*与makefile文件中的其它行。（铭记于心，make根本不知道*recipes*是如何工作的。它取决于你提供的能够更新*target*文件的*recipes*。make所做的只是在*target*文件需要被更新的时候，去执行你描述的*recipe*。）

*target* ‘clean’ 并不是一个文件，它只是一个操作的名字。正常情况下，因为你不会去执行这个规则中的操作，所以’clean’不是任何其它规则的*prerequisite*。因此，除非你指定make去执行’clean’，否则’clean’永远不会被自动执行。注意：这个规则不仅不是其它*target*的*prerequisite*，而且它自己也没有任何*prerequisites*，所以这个规则的唯一目的就是执行*recipes*。类似这样的没有涉及*prerequisites*，并且只有*recipes*的*targets*称为*假的targets*。具体信息请看第4.5节[*假的targets*]，第29页。如何是make忽略rm或其它命令引起的错误，请看第5.5节[*recipes*中的错误]，第49页。

## make如何处理Makefile文件

默认情况下，make从第一个*target*（不会是以’.’开头的*targets*）开始执行。我们称第一个*target*为*默认目标*（目标是指那些make最终努力去更新的*targets*。你可以通过命令行参数（请看第9.2节[指定目标的参数]，第99页）或者设置.DEFAULT\_GOAL具体值（请看第6.14节[其他具体变量]，第73页），来重写这个行为）。

在上一节的简单示例中，默认的目标就是更新执行文件’edit’；因此，我们把这个规则放在第一行。

因此，当你输入命令：

# make

make读取当前目录下的makefile文件，开始处理第一个规则。在示例中，这个规则就是重新链接’edit’；但是在make完全处理这个规则之前，make必须先处理’edit’依赖的object文件的规则。这些object文件中的每一个都需要按照自己的规则进行处理。这些规则指明每一个’.o’文件都需要编译它的源文件。出现如下情况必须重新编译，存在任何一个*prerequisites*的文件修改时间比object文件新，或者object文件不存在。

其它规则会被处理因为它们的*targets*作为目标的*prerequisites*。如果一些规则没有被目标依赖（或者任何目标依赖的文件等），那么这些规则不会被处理，除非你指定make去这样做（例如，*make clean*）。

在重新编译object文件之前，make会考虑更新它的*prerequisites*，也就是源文件和头文件。这个makefile中没有指明为源文件和头文件做任何事情，因为’.c’和’.h’文件不是任何规则的*targets*，所以make不会为这些文件做任何事情。但是make可以通过规则更新由Bison或者Yacc自动生成的C语言程序。

在重新编译需要的object文件之后，make决定是否重新链接’edit’。如下情况下就需要重新链接，存在任何一个修改时间比’edit’新的object文件，或者’edit’不存在。如果一个object文件刚被重新编译，那么它就比’edit’新，所以’edit’就会被重新链接。

因此，如果我们改变’insert.c’，并且执行make，make则会编译’insert.c’以更新’insert.o’，然后链接生成’edit’。如果我们改变’command.h’并且执行make，make则会重新编译’kbd.o’, ’command.o’, ’files.o’，然后链接生成’edit’。

## 变量使Makefiles更简单

在我们的示例中，我们不得不在’eidt’的规则中两次列举所有的object文件：

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

这样的副本是易于出错的；如果一个新的文件被添加到系统中，我们就要把它添加到其中一个序列中，很可能会忘记添加到另一个序列中。我们可以通过使用*variables*去除这种风险，并且简化makefile。*variables*允许一个文本字符串被定义一次，并在之后的多个地方使用（请看第6章[如何使用*variables*]，第59页）。

这是一个惯例在每一个makefile文件中含有一个*variable*名为objects, OBJECTS, objs, OBJS, obj, 或者 OBJ，以上这些全都是object文件的名字。我们可以在makefile中定义一个这样的*variable* objects：

objects = main.o kbd.o command.o display.o \

insert.o search.o files.o utils.o

然后，每一个我们想使用object文件名字序列的地方，都可以用*variable*来替换，只要写’$(objects)’（ 请看第6章[如何使用*variables*]，第59页）。

下面是用变量替换后的makefile文件：

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)

## 让make来推断recipes

不一定要为单个的C语言源文件编写*recipes*来编译它，因为make可以进行如下推断：它有*隐式规则*指明通过相关联的’.c’文件，使用’cc -c’命令，来更新一个’.o’文件。例如，make使用’cc –c main.c –o main.o’ 命令来将’main.c’ 编译成’main.o’文件。因此，我们可以省略在object文件的规则中省略*recipes*。请看第10章[使用隐式规则]，第111页。

如果一个’.c’文件使用*隐式规则*的时候，该’.c’文件还会自动地加入到*prerequisites*中，因此当我们不写*recipes*时，可以在*prerequisites*中省略’.c’文件。

下面是全部的示例，包括以上的所有改变：

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)

以上就是我们实践中编写的makefile。（关于’clean’的描述请看第4.5节[假的targets]，第29页和第5.5节[recipes中的错误]，第49页。）

因为隐式规则如此方便，所以这些规则很重要。你将会经常看到它们的应用。

## Makefile的另一种样式

当使用隐式规则创建object文件的时候，你就可以使用makefile的另一种样式。这种样式的makefile，你可以按照*prerequisites*进行分组而不是*targets*。下面是这种样式的makefile：

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)

上面的makefile中，’defs.h’是所有object文件的*prerequisite*；’command.h’和’buffer.h’是部分object文件的*prerequisite*。

这种风格的makefile是否更好取决于个人的偏好：它看起来更紧凑，但是一些人不喜欢它，因为他们觉得把每个*target*的信息单独放在一起看起来更清晰。

## 清除目录的规则

你编写规则可能不仅是想编译程序。Mafile通常可以完成除编译程序外的一些事情：例如，如何删除所有的object文件和执行文件，以使目录变得干净。

这里我们编写一个规则来清空我们示例中*edit*工程：

clean:

rm edit $(objects)

实际中，我们可能想要编写更复杂的规则来处理意料之外的情况。我们可以这样写：

.PHONY: clean

clean:

rm edit $(objects)

通过.PHONY可以避免当前目录存在名为’clean’的文件带来的混淆，并且忽略*rm*的错误继续执行。（请看第4.5节[假的targets]，第29页和第5.5节[recipes中的错误]，第49页。）

一个类似clean的规则不能放在makefile的开头，因为我们不想让这个规则默认执行。因此，就像示例中的makefile，我们想让重新编译editor的规则仍然为默认目标。

因为clean并不是edit的*prerequisite*，所以只要我们提供make命令行参数，这个规则永远都不会执行。为了执行这个规则，我们需要输入’make clean’。请看第9章[如何执行make]，第99页。

# 编写Makefiles

make读取makefile文件来确定如何重新编译。

## Makefiles包含什么

Makefiles包含五种类型的数据：*显式规则*，*隐式规则*，*变量定义*，*指令*和*注释*。规则，变量和指令将在后面的章节中详细地描述。

* 一个*显式规则*指出何时以及如何重新生成一个或多个文件，这些文件被称为targets。该规则列举出targets依赖的所有文件，这些文件被称为target的prerequisites。可能还会给出一个recipe用来创建或更新targets。请看第4章[编写规则]，第21页。
* 一个*隐式规则*指出何时以及如何根据文件名重新生成一系列文件。该规则描述了一个target是如何依赖于和该target文件名相近的文件，并且给出一个recipe用来创建或更新这个target。请看第10章[使用隐式规则]，第111页。
* 一个*变量*的定义是指一行用来描述一个变量的文本字符串，在定义之后可以使用该变量，并且使用该文本字符串替换该变量（功能与C语言的宏定义相同）。在[第2.4节](#_变量使Makefiles更简单)的简单示例中，描述了一个变量的定义，该变量指所有的object文件。
* 一个*指令*是指当读makefile文件的时候指示make去做一些特殊的事情的命令。

这些指令包括：

* + 读取另一个makefile（请看第3.3节[包含其它的Makefiles]，第13页）。
  + 根据变量的值决定是否使用或忽略makefile文件中的某一特定部分（请看第7章[Makefile的条件执行]，第77页）。
  + 定义一个包含多行的变量（请看第6.8节[定义多行变量]，第69页）。
* ‘#’在makefile中是一行注释的开头。该行中，’#’以后的字符将被忽略。如果行尾有 ’\’ 那么下一行还会是注释。如果想使用字符’#’（此时不是当做注释标记），那么就要使用’\#’来代替。注释可以出现在makefile的任意一行中。

注意**不能在变量引用和方法调用时使用注释**：在变量引用和方法调用的内部，都不会将字符’#’当成是注释的开头。

一个recipe中的注释会被传递给shell，就像任意其它的recipe文本一样。shell决定如何去解释它：这是否代表注释取决于shell。

在define（makefile中的关键字）指令中，不会忽略变量定义中的注释，而是把注释当成变量的整体。当这个变量被展开的时候，之前的注释可能会被认为是注释，也可能被认为是recipe中的一部分，这取决于变量的上下文。

### 分割长行

Makefiles使用基于行的语法分析，其中新一行的字符是特殊的，并且标记叙述的结尾。make没有限制行的长度，它取决于你的计算机的内存大小。

但是，阅读长度太长的一行文字太困难了。所以，为了更好地阅读，你可以使用转义的换行符格式化你的makefiles：通过使用转义的换行符’\’来另起一行。这里我们需要区分下： 实际行作为单独的一行，它的结尾是换行符；逻辑行是指一个语句的整体包括所有转义的换行符一直到非转义的换行符为止。

处理反斜杠/换行符的组合取决于是否为recipe。在recipe中处理反斜杠/换行符的组合将在稍后讨论（请看分割recipe行）。除了recipe中，反斜杠/换行符被转换成一个空格字符，一旦转换之后，周围所有的空白字符都会被认为是一个空格字符：这些包括反斜杠之前所有的空白字符，和所有连续的反斜杠/换行符组合。

如果.POSIX定义了特殊的target，那么反斜杠/换行符的处理将稍有不同：首先，不会删除反斜杠之前的空白字符；其次，反斜杠/换行符之后的也不会被压缩。

## Makefile文件的命名

默认的，make查找makefile文件会按照如下顺序依次查找：GNUmakefile，makefile，Makefile。

通常你应该命名为makefile或Makefile。（我们推荐Makefile，因为它出现在文件目录列表的开头附近，并且靠近重要文件例如README。）第一个检查的文件名GNUmakefile是不推荐的，你只有在确定使用GNU make的是否才能使用GNUmakefile这个名字，但是其他版本的make不识别这个名字，所有的make工具都查找makefile和Makefile，而没有GNUmakefile。

如果make没有找到这些文件，它就不会使用任何makefile文件。所以你就必须使用命令行参数指定目标，然后make会尝试猜测如何只使用内置的隐式规则重新生成目标。请看使用隐式规则。

如果你想使用非标准的名字命名你的makefile文件，你可以通过参数’-f’或者’--file’指定makefile文件的名字。参数’-f name’或者’—file=name’指定make去读取这些文件作为makefile。使用多余一个’-f’或者’--file’，你可以指定一些makefile文件。所有的这些makefile文件将会按顺序依次执行。如果使用参数’-f’或’--file’，make就不会查找默认的makefile文件。

## 包含其它Makefiles

include指令使make停止读取当前的makefile，在继续读取之前转而读取其它的一个或更多个makefiles。该指令用法如下所示：

include filenames …

filenames 可以包含shell中文件名的通配符。如果filenames是空的，将不会读取其他makefile文件，也不会输出错误。

行的开头可以有空格，但是make会忽略这些空格。如果以tab（或者.RECIPEPREFIX的值）开头，那么make认为该行为recipe。在include与filename之间需要有空白字符，filename之间也应该有。include的末尾可以有注释。如果文件名包含变量或函数引用，它们都将被展开。请看第6章[如何使用变量]，第59页。

例如，如果你有三个文件，’a.mk’，’b.mk’，’c.mk’，并且$（bar）表示bish bash，那么接下来的表达式

include foo \*.mk $(bar)

等同于

include foo a.mk b.mk c.mk bish bash

当make处理include指令时，它暂停当前makefile，依次读取列举出来的文件，当列举的文件处理完后，make会继续执行之前暂停的makefile。

### 通常include指令用于以下情形

1. 当多个程序，各自的makefile在不同的目录下，并且这些makefile使用共同的变量（请看第6.5节[设置变量]，第65页）或者规则（请看第10.5节[定义和重定义模式规则]，第118页），这时候可以使用include指令来完成，例如：

makefile\_1

**objects** = main.o kbd.o command.o display.o \

insert.o search.o files.o utils.o

makefile\_2

**include** makefile\_1

edit: $(**objects**)

cc -o edit $(**objects**)

……

在makefile\_1中定义了变量objects，在makefile\_2中使用该变量，需要先inlcudemakefile\_1，然后就可以使用了。

1. 当你想要通过源文件自动生成prerequisites时，这些prerequisites可以单独放在一份文件中，然后makefile包含这个文件。这样makefile会更清晰。请看第4.13节[自动的prerequisites]，第38页。

### include指令的搜索路径

如果文件名开头不是’/’，并且当前目录没有该文件，那么make将会在指定目录下查找。

首先，查找通过参数’-i’或者--include-dir’指定的目录（请看第9.7节[选项总结]，第104页）。

然后，依次查找接下来的目录：’prefix/include’（通常是’/usr/local/include’[[1]](#footnote-1)），’/usr/gnu/include’，’/usr/local/include’，’/usr/include’。

如果在上述路径中都没有找到被包含的makefile文件，make会产生一条警告信息，但是这不是立即的严重错误；makefile的处理还会继续进行。一旦make完成对makefiles的处理，它会尝试重新生成任何过期的或者不存在的文件。请看第3.5节[Makefile如何控制重新生成]，第14页。当make尝试重新生成一个makefile文件并且失败时，make才会确认缺失makefile是一个严重错误。

使用-include指令可以让make简单地忽略不存在的或者不能重新生成的makefile，并且不产生错误信息，如下：

-include filenames …

这样使用include永远不会产生错误信息甚至警告，即使这些文件不存在或者不能被重新生成。

建议使用sinclude替换-include，因为sinclude是为了与其他make工具的兼容，而给-include起的别名。

## 变量MAKEFILES

如果已经定义环境变量MAKEFILES，make会认为它的值是一系列makefile的文件名，make先解析这些文件再解析’makefile’或’Makefile’。它的作用跟include指令相似：在指定路径下查找这些文件（[请看第3.3节[包含其它Makefiles]，第15页](#_包含其它Makefiles)）。另外，默认的目标不会因为要包含的这些makefile文件而改变，并且找不到MAKEFILES中列举的文件也不是错误。

变量MAKEFILES主要应用于递归调用中的通信。它通常不设置在最开始调用之前，因为最好不要将它与外部的makefile混淆。但是，如果没有具体的makefile文件，此时执行make，变量MAKEFILES中指定的makefile文件可以帮助内置的规则更好的使用，例如定义搜素路径（请看第4.4节[搜素目录]，第25页）。

一些用户尝试在开机时自动设置变量MAKEFILES，并且makefile文件依赖于该变量。这是一个非常糟糕的想法，因为这样的makefiles在任何其他计算机上执行都会失败。更好的方法是使用显式的include指令。[请看第3.3节[包含其它Makefiles]，第13页](#_包含其它Makefiles)。

## Makefiles如何控制重新生成

有些时候makefiles可以由其它文件更新，例如RCS或者SCCS（这两个都是版本控制系统）。如果makefile可以由其它文件更新，那么你在make之前应该更新makefile。

首先，make读入所有的makefile文件，把每个文件当成是一个目的target并且尝试更新该文件。如果一个makefile有一个规则，该规则指明如何更新该makefile（这个规则可能在该makefile中或者在其它的makefile中），或者使用隐式规则（请看第10章[使用隐式规则]，第111页）。当检查完所有的makefile时，如果有makefile文件被改变了，make将重新开始读入所有的makefile文件。（make还会尝试更新这些makefile，但是通常不会再改变了，因为它们已经是最新的了。）

如果你知道一个或多个makefile文件不能被重新生成，并且你不希望make使用隐式规则来搜索这些文件，可能是为了效率，你可以使用任何正常的方法去阻止查找隐式规则。例如，你可以为makefile编写明确的规则，并且该规则的target是这个makefile文件，recipe是空的（请看第5.9节[使用空的recipes]，第57页）。

1. 在MS-DOS和MS-Windows平台下编译的GNU Make，prefix被定义为DJGPP的根节点。 [↑](#footnote-ref-1)