GNU Make

GNU Make

GNU make 4.4版直接重新编译程序
2022年10月

Richard M. Stallman，Rol和McGrath，Paul D. Smith

本文件记录了GNU make实用程序，它自动确定大型程序的哪些部分需要重新编译，并发出重新编译它们的命令。

这是《GNU Make手册》的第0.76版，最后更新于2022年10月31日，适用于GNU Make版本4.4。

由自由软件基金会发布
51 Franklin St.–Fifth Floor Boston，MA02110-1301 USA ISBN1-882114-83-3

封面艺术由Etienne Suvasa创作。

简短内容

目录

1、制造概述

make工具自动确定大型程序中哪些部分需要重新编译，并发出命令重新编译它们。本手册描述的是GNU make，由Richard Stallman和Roland McGrath实现。自3.76版本以来的开发工作由Paul D. Smith负责。

GNU make符合IEEE标准1003.2-1992（POSIX. 2）第6.2节。

我们的例子展示了C程序，因为它们最为常见，但你可以用make来处理任何可以用shell命令运行其编译器的编程语言。事实上，make并不局限于程序。你可以用它来描述任何任务，在这些任务中，当其他文件发生变化时，某些文件必须自动从其他文件更新。

要准备使用make，您必须编写一个名为make的文件，该文件描述程序中各文件之间的关系，并提供更新每个文件的命令。通常情况下，在程序中，可执行文件是从对象文件更新的，而对象文件则是通过编译源文件生成的。

一旦存在合适的生成文件，每次更改某些源文件时，执行以下简单的Shell命令：

制造

足以执行所有必要的重新编译。make程序使用makefile数据库和文件s的最后修改时间来决定哪些文件需要更新。对于这些文件中的每一个，它都会发出数据库中recor ded中的配方。

您可以提供命令行参数来控制哪些文件应该重新编译，或者如何重新编译。请参见第9章[如何运行make]，第109页。

1.1如何阅读本手册

如果你是初次接触make，或者正在寻找一个总体介绍，请阅读每章的前几节，跳过后面的章节。在每一章中，前几节包含介绍性或一般性的信息，而后面的章节则包含专门或技术性的信息。例外情况是第2章[Makefile简介]，第3页。 ，
所有这些都是入门级的。

如果您熟悉其他编译程序，请参见第14章[GNUmake的功能]，第157页，其中列出了GNU make所具有的增强功能，以及第15章[不兼容性和缺失功能]，第161页，其中解释了GNU make缺少而其他编译程序所具有的少数功能。

快速总结请参见第9.8节[选项总结]，第114页，附录A[快速参考]，第179页，以及第4.8节[特殊目标]，第34页 .

1.2问题和缺陷

如果您在使用GNU make时遇到问题或认为自己发现了bug，请向开发人员报告；我们不能保证会做任何事情，但我们很可能会修复它。

在报告错误之前，请确保你确实发现了一个真正的错误。仔细重读文档，看看它是否真的说明了你可以做你尝试做的事情。如果不清楚你是否应该能够做某事，也要报告；这是文档中的错误！

在报告错误或尝试自行修复之前，请尽量将问题隔离到最小的可重现问题的构建文件中。然后发送给我们该构建文件以及你得到的确切结果，包括任何错误或警告消息。请不要改写这些消息：最好将其复制粘贴到你的报告中。在生成这个小型构建文件时，务必不要使用任何非自由或不常见的工具：几乎总是可以通过简单的shell命令来模拟这些工具的作用。最后，请务必说明你预期会发生什么；这将帮助我们判断问题是否真的出在文档中。

一旦您确定了确切的问题，您可以采用以下两种方式之一进行报告：

bug-make@gnu.org

或者使用我们的基于Web的项目管理工具，地址如下：

https://savannah.gnu.org/projects/make/

除了上述信息外，请务必包括您正在使用的版本号。您可以通过命令‘make--version’获取此信息。同时，务必注明您使用的机器类型和操作系统。一种获取这些信息的方法是查看命令‘make--help’的最终输出行。

如果您要提交代码更改，请参阅“提交补丁”一节中的README文件部分以获取相关信息。

2.Makefile简介

需要一个名为makefile的文件来告诉make该做什么。大多数情况下，makefile告诉make如何编译和链接程序。

在本章中，我们将讨论一个简单的生成文件，该文件描述了如何编译和链接由八个C源文件和三个头文件组成的文本编辑器。生成文件还可以告诉生成程序在明确请求时运行各种命令（例如，作为清理操作删除某些文件）。要查看更复杂的生成文件示例，请参见附录C[复杂生成文件]，第191页。

当重新编译编辑器时，每个更改的C源文件都必须重新编译。如果头文件发生变化，包含该头文件的每个C源文件都必须重新编译以确保安全。每次编译都会生成一个与源文件相对应的目标文件。最后，如果有任何源文件被重新编译，所有目标文件，无论是新生成的还是从之前的编译中保存的，都必须链接在一起，以生成新的可执行编辑器。

2.1规则的外观

一个简单的生成文件由具有以下形状的“规则”组成：

```
目标...：先决条件...
        食谱
        ...
        ...
```

目标通常是程序生成的文件的名称；目标的例子是可执行文件或对象文件。目标也可以是要执行的操作的名称，例如“clean”（见第4.5节[假目标]，第31页).

前提是使用一个文件作为创建目标的输入。目标通常依赖于多个文件。

食谱是一种行动指令。一个食谱可能包含多个命令，这些命令要么在同一行中，要么各自独立成行。请注意：每行食谱的开头都需要加一个制表符！这是一个容易被忽视的细节。如果您希望用除制表符以外的其他字符作为食谱的前缀，可以将. RECIPEPREFIX变量设置为另一个字符（参见第80页第6.14[特殊变量]节。).

通常情况下，一个配方包含前提条件和规则，如果任何前提条件发生变化，则用于创建目标文件。然而，指定目标配方的规则不必有前提条件。例如，与目标‘clean’关联的删除命令所在的规则就没有前提条件。

规则解释了如何以及何时重做某些文件，这些文件是特定规则的目标。make执行创建或更新目标的前提条件的配方。规则也可以解释如何以及何时执行一个操作。请参见第4章【写作规则】，第23页 .

生成文件可以包含除规则以外的其他文本，但是简单的生成文件只需要包含规则。规则可能看起来比这个模板中显示的更复杂，但是所有规则或多或少都符合这种模式。

2.2简单的Makefile

下面是一个简单的生成文件，它描述了可执行文件edit如何依赖于八个对象文件，而这些对象文件又如何依赖于八个C源文件和三个头文件。

在这个例子中，所有的C文件都包含defs .h，但是只有定义编辑命令的文件才包含command .h，只有改变编辑器缓冲区的低级文件才包含buffer .h。

```
编辑：main.o kbd.o command.odisplay.o \
        插入.o搜索.o文件.o utils .o
         cc-o编辑main.o kbd.o command.o display.o
                    插入.o搜索.o文件.o utils .o

main .o： main. c defs .h
         抄送-c主.c
kbd .o： kbd. c defs. h command .h
         cc - c kbd .c
命令.o：命令.c defs .h command.h
         cc-c命令.c
display.o：display .c defs .h buffer.h
         cc-c显示.c
insert.o：插入.c defs .h缓冲区.h
         cc-c插入.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文件.c
utils.o：utils .c defs .h
         cc-c-utils.c
干净的
         rm编辑main.o kbd.o command.odisplay.o \
             插入.o搜索.o文件.o utils .o
```

我们使用backslash/newline将每行长行拆分为两行；这就像使用一行长行一样，但更容易阅读。请参见第3.1.1节[拆分长行]，第12页 .

若要使用此生成文件创建名为edit的可执行文件，请键入：

```
制造
```

若要使用此生成文件删除目录中的可执行文件和所有目标文件，请键入：

```
清理
```

在示例生成文件中，目标包括可执行文件‘edit’和对象文件‘main.o’及‘kbd.o’。先决条件是诸如‘main.c’和‘defs.h’之类的文件。实际上，每个‘.o’文件既是目标也是先决条件。配方包括‘cc-c main.c’和‘cc-c kbd.c’。

当目标是文件时，如果其任何先决条件发生变化，则需要重新编译或重新链接。此外，任何自动生成的先决条件应首先更新。在这个例子中，edit依赖于八个对象文件；对象文件main .o依赖于源文件main .c和头文件defs .h。

每个包含目标和先决条件的行后面都可能有一个步骤。这些步骤说明如何更新目标文件。制表符（或由.RECIPEPREFIX变量指定的任何字符；请参见Section6.14[特殊变量]，第80页)必须出现在每个行的开头，以区分配方与其他行。（请注意，make不了解配方的工作原理。你需要提供能够正确更新目标文件的配方。make只是在目标文件需要更新时执行你指定的配方。）

目标“clean”不是文件，而只是一个操作的名称。由于你通常不希望在这个规则中执行这些操作，“clean”并不是其他任何规则的前提条件。因此，除非你特别指定，否则它不会做任何事情。请注意，这个规则不仅不是前提条件，也没有任何前提条件，所以该规则的唯一目的是运行指定的配方。不引用文件但仅作为操作的目标称为虚假目标。参见第4.5节[虚假目标]，第31页。，有关此类目标的信息，请参见第54页第5.5节[配方中的错误]，了解如何使make忽略rm或其他命令的错误。

2.3如何将过程制成一个Makefile

默认情况下，make从第一个目标开始（不包括那些名称以“.”开头的目标，除非它们也包含一个或多个“/”）。这被称为默认目标。（目标是make最终努力更新的目标。你可以通过命令行（见第109页第9.2节[指定目标的参数]）或使用. DEFAULT_GOAL特殊变量来覆盖此行为（见第80页第6.14节[其他特殊变量]）.

在上一节的简单示例中，默认目标是更新可执行程序edit；因此，我们将该规则放在第一位。

因此，当您执行以下命令时：

制造

`make`会读取当前目录中的`makefile`文件，并首先处理第一条规则。在这个例子中，这条规则是用于重新链接`ed it`；但在`make`完全处理这条规则之前，它必须先处理`ed it`所依赖的文件的规则，在这种情况下，这些文件就是目标文件。每个文件都会根据自己的规则进行处理。这些规则指示通过编译源文件来更新每个`o`文件。如果源文件或作为前提条件的任何头文件比目标文件更新得更晚，或者目标文件不存在，则必须重新编译。

其他规则被处理是因为它们的目标作为目标的先决条件出现。如果目标（或它所依赖的任何东西等）不依赖于其他规则，则不会处理该规则，除非您使用make命令告诉它这样做（例如：make clean)。

在重新编译一个对象文件之前，make会考虑更新其先决条件、源文件和头文件。这个makefile没有指定要为它们做些什么——‘.c’和‘.h’文件不是任何规则的目标——所以make对这些文件不做任何事情。但是，make会根据当前的规则自动更新由Bison或Yacc生成的C程序。

重新编译需要的任意对象文件后，make决定是否重新链接编辑。如果文件编辑不存在，或者任何对象文件都比它更新，则必须执行此操作。如果对象文件刚刚重新编译，那么它现在比编辑更新，因此编辑将重新编译。

因此，如果我们修改文件insert .c并运行make，make将编译该文件以更新insert.o，然后链接edit。如果我们修改文件command.h并运行make，make将重新编译object文件kbd. o、command .o和files .o，然后链接文件edit。

2.4变量使生成文件更简单

在我们的示例中，我们必须在编辑规则（此处重复）中两次列出所有对象文件：

```
编辑：main.o kbd.o command.odisplay.o \
        插入.o搜索.o文件.o utils .o
cc-o编辑main.o kbd.o command.o display.o
        插入.o搜索.o文件.o utils .o
```

这种复制容易出错；如果向系统中添加一个新的对象文件，我们可能会将其添加到一个列表中而忘记另一个。通过使用变量可以消除风险并简化生成文件。变量允许一次定义一个文本字符串，之后可以在多个地方替换（见第6章[如何使用变量]，第65页。).

每个生成文件的标准做法是，有一个名为objects、objects、objs、objs、obj或obj的变量，它是一个所有对象文件名的列表。我们可以在生成文件中用如下一行定义这样的变量objects：

```
对象=主控键盘command.odisplay.o \
        插入.o搜索.o文件.o utils .o
```

然后，我们想要在每个地方放置一个对象文件名列表，我们可以通过写入‘$（objects）’来替换变量的值（参见第6章[如何使用变量]，第65页).

使用变量作为对象文件时，完整的简单生成文件如下所示：

```
对象=主控键盘command.odisplay.o \
            插入.o搜索.o文件.o utils .o

编辑：$（对象）
        cc-o edit$(objects)
main .o：main. c defs .h
        抄送-c主.c
kbd .o： kbd. c defs. h command .h
        cc - c kbd .c
命令.o：命令.c defs .h command.h
        cc-c命令.c
display.o：display .c defs .h buffer.h
        cc-c显示.c
insert.o：插入.c defs .h缓冲区.h
        cc-c插入.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文件.c
utils.o：utils .c defs .h
        cc-c-utils.c
干净的
        rm edit$(objects)
```

2.5让让推理出配方

无需详细列出编译各个C源文件的步骤，因为make可以自行解决：它有一个隐式规则，可以通过相应的.c文件使用.cc-c命令来更新.a.o文件。例如，它会使用“cc- c main.c.o main .o”的配方来将main. c编译成main .o。因此，我们可以在目标文件的规则中省略这些配方。参见第10章[使用隐式规则]，第121页。

如果以这种方式自动使用.a.c文件，则它也会自动添加到必需项列表中。因此，只要省略配方，就可以省略必需项中的.c文件。

下面是完整的示例，包含了上述两种更改和变量对象：

```
对象=主控键盘command.odisplay.o \
          插入.o搜索.o文件.o utils .o

编辑：$（对象）
        cc-o edit$(objects)

main .o： defs .h
kbd .o： defs. h command .h
命令.o： defs .h command.h
display.o：defs .h buffer.h
insert.o： defs .h buffer.h
搜索.o： defs .h buffer.h
files.o： defs .h buffer.h command.h
utils .o： defs .h

.假：干净
干净的
        rm edit$(objects)
```

这就是我们在实际操作中编写Make文件的方式。（与“clean”相关的复杂性在其他地方有描述。请参见第4.5节[假目标]，第31页 ，以及第5.5节[配方中的错误]，第5页 .)

因为隐含规则非常方便，所以它们很重要。你会经常看到它们被使用。

2.6另一种Makefi文件样式

如果仅通过隐式规则创建了生成文件的对象，则可以采用另一种生成文件样式。在这种生成文件样式中，您将按先决条件而不是目标来分组条目。如下所示：

```
对象=主控键盘command.odisplay.o \
          插入.o搜索.o文件.o utils .o

编辑：$（对象）
        cc-o edit$(objects)

$(objects): defs .h
kbd. o命令.o文件.o：命令.h
显示.o插入.o搜索.o文件.o：缓冲区.h
```

这里，defs .h是所有对象文件的先决条件；command.h和buffer.h是它们所列特定对象文件的先决条件。

这是否更好是一个品味问题：它更紧凑，但有些人不喜欢它，因为他们发现把每个目标的所有信息放在一个地方更容易。

2.7目录清理规则

编写程序并不是你想要编写的规则的唯一内容。Makefile通常还告诉如何做其他一些事情，除了编写程序之外：例如，如何删除所有目标文件和可执行文件，使目录“干净”。

下面是如何为清理示例编辑器编写一个制作规则：

```
干净的
        rm edit$(objects)
```

在实际操作中，我们可能需要以一种更为复杂的方式编写规则来处理意外情况。我们这样做：

```
.假：干净干净：
        -rm edit$(objects)
```

这可以防止make被一个名为clean的实际文件混淆，并使其继续运行，尽管rm出错。（参见第4.5[假目标]节，第31页 ，以及第5.5节 [食谱中的错误]，第54页 .)

这样的规则不应该放在生成文件的开头，因为我们不希望它默认运行！因此，在示例生成文件中，我们希望编辑规则保持默认目标，该规则重新编译编辑器。

由于clean不是ed的先决条件，因此如果我们不带参数地给出命令‘make’，那么这条规则将根本不会运行。为了使这条规则运行，我们必须键入‘makeclean’。请参见第9章[如何运行make]，第109页。

3编写Makefile

告诉如何重新编译系统的信息来自于读取一个叫做makefile的数据库。

3.1生成文件包含的内容

Makefile包含五种内容：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描述。

•一个明确的规则规定了何时以及如何重做一个或多个文件，称为规则的目标。它列出了目标所依赖的其他文件，称为目标的先决条件，并且可能还给出了创建或更新目标的配方。请参见第4章 【写作规则】，第23页 .

•一个隐式规则规定了根据文件名何时以及如何重新创建文件类。它描述了目标如何依赖于名称与目标相似的文件，并给出了创建或更新此类目标的方法。请参见第10章[使用隐式规则]，第121页。

变量定义是一行指定变量的文本字符串值，该值可以稍后替换到文本中。简单的makefile示例显示了对象的变量定义，作为所有对象文件的列表（参见第2.4节[变量Make MakefilesSimpler]，第6页).

指令是在读取生成文件时发出的指示，用于执行特殊操作。这些包括：

•阅读另一个生成文件（参见第3.3节[包括其他生成文件]，第13页).

决定（根据变量值）是否使用或忽略生成文件的一部分（参见第7章[生成文件的条件部分]，第85页).

从包含多行的逐字字符串中定义变量（见第6.8节 [定义多行变量]，第76页).

•'#'在一个制文件行的开头开始一个注释。它和该行的其余部分都会被忽略，除非后跟的反斜杠未被另一个反斜杠转义，这会使注释跨多行继续。仅包含注释（可能前面有空格）的行实际上是空白的，并且会被忽略。如果你想使用字面意义上的#，可以用反斜杠转义（例如，\#）。注释可以出现在制文件的任何一行中，尽管在某些情况下它们会受到特殊对待。

不能在变量引用或函数调用中使用注释：在变量引用或函数调用中，任何instanceof #都将被当作字面量（而不是注释的开头）处理。

在配方中，注释传递给外壳，就像其他任何配方文本一样。外壳决定如何解释注释：这是否为注释由外壳决定。

在定义指令中，变量定义期间不会忽略注释，而是在变量值中保留注释。当变量被展开时，它们将被当作注释或回文处理，这取决于变量在评估时的上下文。

3.1.1拆分长行

Makefile使用一种“基于行”的语法，其中换行符是特殊的，它标记了语句的结束。GNU make对语句行的长度没有限制，直到你的计算机内存的大小为止。

然而，阅读过长而无法直接显示或滚动的行是困难的。因此，可以通过在语句中间添加换行符来格式化你的构建文件以提高可读性：具体做法是用反斜杠（\）字符转义内部换行符。我们需要区分的是，“物理行”是指以换行符结束的一行（无论是否已转义），而“逻辑行”则指包括所有已转义换行符在内的完整语句，直到第一个未转义的换行符。

backslash/newline组合的处理方式取决于该语句是配方行还是非配方行。关于配方行中backslash/newline的处理将在后面讨论（参见第45页第5.1.1[Splittin g RecipeLines]节).

在配方行之外，反斜杠/newline被转换成单个空格字符。完成此操作后，反斜杠/新线周围的空格将被压缩成单个空格：这包括反斜杠前的所有空格、反斜杠/新线后行首的所有空格以及任何连续的反斜杠/新线组合。

如果定义了. POSIX特殊目标，则会稍作修改以符合POSIX。2：首先，不会删除斜杠前的空格；其次，连续的斜杠/换行符不会被压缩。

不添加空格的拆分

如果需要拆分行，但又不想添加任何空白，则可以使用一个微妙的技巧：将反斜杠/换行符对替换为三个字符美元符号、反斜杠和换行符：

```
var:= one$\
      单词
```

在删除反斜杠/newline并把以下行缩进为一个空格之后，这相当于：

```
var：= one$个单词
```

然后，make将执行变量展开。变量引用“$”引用一个名为“”（空格）的单字符变量，该变量不存在，因此展开为空字符串，最终赋值相当于：

```
var:= oneword
```

3.2为您的制作文件命名

默认情况下，当make寻找makefile时，它会按以下顺序尝试以下名称：GNUmakefile、makefile和makefile。

通常你应该调用你的makefile，无论是makefile还是makefile。（我们推荐使用makefile，因为它在目录列表的开头附近显眼地出现在其他重要文件如readme旁边。）第一个名称检查到的GNUm akefile，不建议用于大多数makefile。如果你有一个makefile是

仅适用于GNU make，其他版本的make不会理解它。其他make程序会查找makef ile和makef ile，但不会查找GNUmakefile。

如果make没有找到这些名称，则它不会使用任何makefil e。那么，您必须指定一个目标并使用命令参数，然后make将尝试找出如何仅使用其内置隐式规则来重新创建它。请参见第10章[使用隐式规则]，第121页。

如果您想为您的生成文件使用非标准名称，可以使用‘-f’或‘--file’选项指定生成文件名称。参数‘-f name’或‘--file=name’告诉生成器将文件名作为生成文件读取。如果您使用多个‘-f’或‘--file’选项，可以指定多个生成文件。所有生成文件将按指定的顺序有效连接。如果您指定‘-f’或‘--file’，默认生成文件名称GNUmakefile、makefile和makefile不会自动检查。

3.3包含其他Makefile

include指令告诉make程序暂停当前的makefile，然后在继续之前读取一个或多个其他makefile。include指令是makefile中的一行代码，其形式如下：

包括文件名...

文件名可以包含shell文件名模式。如果文件名为空，则不包含任何内容，并且不会打印出错误信息。

行首允许和忽略额外空格，但第一个字符不能是制表符（或.recipeprefix的值）——如果行以制表符开始，则被视为配方行。include和文件名之间以及文件名之间需要空格；额外的空格在此处和指令末尾被忽略。行尾允许以‘#’开头的注释。如果文件名包含任何变量或函数引用，则会进行展开。参见第6章[如何使用变量]，第65页 .

例如，如果您有三个.mk文件a.mk、b.mk和c.mk，并且$（bar）展开为bish bash，则以下表达式

包括foo*.mk$（bar）等

同于

包括foo a. mk b. mk c. mk bish bash

当make过程包含指令时，它会暂停对所包含的makefile的读取，并依次从每个列出的文件中读取。当此过程完成时，make将恢复对指令出现的makefile的读取。

使用include指令的一个场合是，当由不同目录中的各个独立makefile处理的多个程序需要使用一组共同的变量定义时（参见第72页第6.5节[SettingVariables]）或模式规则（参见第129页第10.5节[定义和重新定义模式规则]）。

另一个类似的情况是，当你希望自动从源文件生成前提条件时；这些前提条件可以放在一个被主构建文件包含的文件中。这种做法通常比传统上将前提条件附加到主构建文件末尾的方法更为干净。参见第4.13节[自动前提条件]，第42页。
.

如果指定的名称不以斜杠开头，且当前目录中未找到该文件，则会搜索其他几个目录。首先，搜索您使用'-I'或'--include-dir'选项指定的所有目录（参见第9.8节[选项概要]，第114页）。然后按以下顺序搜索以下目录（如果存在）：prefix/include（通常为/usr/local/include 1）、usr/gnu/include、usr/local/include、usr/include。

. INCLUDE_ DIRS变量将包含当前目录列表，其中将搜索所包含的文件。请参见第80页第6.14[其他特殊变量]节 .

通过添加命令行选项，可以避免在这些默认目录中进行搜索-我与命令行具有特殊值（例如，-I-）。这将导致make忽略任何已设置的包含目录，包括默认目录。

如果包含的生成文件在这些目录中找不到，则不会立即导致致命错误；包含该生成文件的处理会继续进行。一旦读取完生成文件后，make将尝试重新创建任何过时或不存在的生成文件。参见第3.5节[如何重写生成文件]，第15页。 只有在找不到重新生成生成文件的规则，或者找到规则但配方失败之后，生成程序才会将缺少生成文件诊断为致命错误。

如果要简单地忽略不存在或无法重新生成的生成文件，而不会出现错误消息，请使用-include指令代替includ e，例如：

-包括文件名...

这在所有方面都像包含一样，只是如果任何文件名（或任何文件名的任何先决条件）不存在或无法重新创建，则不会出现错误（甚至没有警告）。

为了与某些其他制造商的实现兼容，sinclude是另一个名称-包括。

3.4可变的生成文件

如果环境变量`env ironment`中定义了`MAKEFILES`，`make`将其值视为一个名称列表（由空格分隔），这些名称是其他`make`文件读取前需要读取的额外`make`文件。这与包含指令的工作方式类似：会搜索多个目录以查找这些文件（见第3.3节[包含其他`make`文件]，第13页。）。此外，defa的默认目标从未从这些生成文件（或它们包含的任何生成文件）中被删除，如果未找到生成文件中列出的文件，则不会出错。

makefile的主要用途是在递归调用make之间的通信（见第56页第5.7节[递归使用make]）。通常，在顶级调用`make`之前设置环境变量是不理想的，因为通常最好不要从外部干扰`make`文件。然而，如果你在没有特定`make`文件的情况下运行`make`，那么`MAKEFI LES`中的`make`文件可以做一些有用的事情，帮助内置的隐式规则更好地工作，例如定义搜索路径（参见第4.4节[目录搜索]，第27页).

有些用户在登录时会自动将makefile设置为环境变量，并且程序也会期望这样做。这是一个非常糟糕的主意，因为使用这样的

1为MS-DOS和MS-Windows编译的GNU Make表现得好像前缀已被定义为DJGPP树层次结构的根。

如果由其他人运行，makefiles将无法正常工作。最好在makefiles中写明包含指令。请参见第3.3节[包含其他makefiles]，第13页 .

3.5如何重制Makefile

有时，可以将其他文件，如RCS或SCCS文件，重新制作成make文件。如果可以将其他文件重新制作成make文件，则您可能需要使make获取最新的make文件版本来读取。

为此，在读取所有生成文件后，make将将每个文件视为目标目标，并按照处理的顺序尝试更新它。如果进行并行构建（请参见第51页第5.4节[并行执行] ）被启用后，也会并行重建makefile。

如果一个生成文件中有一条规则说明如何更新它（该规则可能出现在当前生成文件中或另一个生成文件中），或者如果存在一个隐式规则适用于它（参见第10章[使用隐式规则]，第121页），则会在必要时进行更新。所有生成文件检查完毕后，如果有任何实际被更改的生成文件，make将从头开始重新读取所有生成文件。（它还会尝试再次更新每个生成文件，但通常不会再次更改它们，因为这些生成文件已经是最新的。）每次重启都会导致特殊变量variableMAKE_RESTARTS 的更新（参见第6.14节[特殊变量]，第80页。).

如果你知道一个或多个生成文件无法重新创建，并且希望保持生成过程不执行隐式规则查找，可能出于效率考虑，你可以使用任何常规方法来防止隐式规则查找。例如，你可以编写一个显式规则，以生成文件为目标，并设置为空的配方（参见第5.9节[使用空配方]，第62页。).

如果生成文件指定了double-colon规则以用配方重制一个文件，但不指定prereq-site，则该文件将始终被重制（请参见第42页第4.12[Double-Colon]节).在生成文件的情况下，如果一个生成文件包含双冒号规则和配方但没有先决条件，那么每次运行生成时都会重新生成该文件，然后在生成重新开始并再次读取生成文件时也会重新生成。这会导致无限循环：生成会不断重新生成生成文件并重启，而不会执行其他任何操作。因此，为了避免这种情况，生成不会尝试重新生成那些被指定为双冒号规则的目标但没有先决条件的生成文件。

虚假目标（见第4.5节[PhonyTargets]，第31页 ）具有相同的效果：它们永远不会被认为是最新的，因此标记为假的包含文件会导致make程序不断重新启动。为了避免这种情况，make程序不会尝试重新创建标记为假的make文件。

您可以利用这一点来优化启动时间：如果您知道不需要重新生成Makefile，您可以通过添加以下内容来防止make尝试重新生成它：

```
。PHONY：
Makefile或：
  Makefile:: ;
```

如果您没有指定任何要通过“-f”或“--file”选项读取的make文件，那么make将尝试使用默认的make文件名；请参见第3.2节[给您的Makefile起什么名字]，第12页 与通过-f或--file选项显式请求的make文件不同，make不能确定

这些生成文件应该存在。但是，如果默认生成文件不存在，但可以通过运行生成规则创建，则您可能希望运行这些规则以便使用生成文件。

因此，如果不存在任何默认的make文件，make将尝试为每一个make文件创建一个，直到成功创建一个或者没有更多的名字可尝试为止。请注意，如果make找不到或无法创建任何make文件，这并不是一个错误；make文件并不总是必要的。

当你使用‘-t’或‘--touch’选项（见第9.3节[代替执行配方]，第111页），你不会希望用一个过时的生成文件来决定要触碰哪些目标。因此，即使指定了‘-t’选项，它也不会影响更新生成文件；实际上，即使没有指定‘-t’，生成文件也会被更新。同样，‘-q’（或‘--question’）和‘-n’（或‘--just-print’）也不会阻止更新生成文件，因为一个过时的生成文件会导致其他目标输出错误。因此，‘make-f mfile-nfoo’会更新mfi le，读取它，然后打印配方以更新foo及其先决条件，而无需运行它。为foo打印的配方将是mfi le中更新内容所指定的那个。

但是，有时您可能确实希望阻止更新甚至生成文件。您可以通过在命令行中将生成文件指定为goa ls以及将它们指定为生成文件来实现这一点。当生成文件名称被明确指定为目标时，选项‘-t’等等将适用于它们。

因此，“make-fmfile-n mfilefoo”将读取makefile mfile，打印更新它所需的配方，而实际上不运行它，然后打印更新foo所需的配方而不运行该。foo的配方将是现有指定的mfile的内容。

3.6超越其他文件夹

有时，拥有一个几乎与另一个makefile相同的文件是有用的。你可以经常使用‘include’指令在一个makefile中包含另一个，然后添加更多的目标或变量定义。然而，对于两个makefile来说，为同一个目标提供不同的配方是无效的。但还有另一种方法。

在包含的生成文件（即希望包含另一个生成文件的生成文件）中，可以使用任意匹配模式规则来说明，如果无法从包含的生成文件中的信息生成任何目标，则生成过程应查找另一个生成文件。有关模式规则的更多信息，请参见第10.5节[模式规则]，第129页。

例如，如果有一个名为makefile的生成文件说明了如何生成目标“foo”（以及其他目标），则可以编写一个名为GNUmakefile的生成文件，其中包含：

```
foo:
        frobnicate> foo

%：强制
        @$（MAKE）- f Makefile $@
强迫
```

如果你说“生成foo”，`make`会找到`GNUmakefile`，读取它，并发现要生成`foo`需要运行配方‘frobnicate>foo’。如果你说“生成bar”，`make`会在`GNUmakefile`中找不到生成`bar`的方法，因此它会使用模式规则中的配方：`make -f Makefile bar`。如果`Makefile`提供了更新`bar`的规则，`make`将应用该规则。同样地，对于`GNUmakefile`没有说明如何生成的任何其他tar文件也是如此。

这种方式的工作原理是，模式规则的模式仅为“%”，因此它可以匹配任何目标文件。规则指定了一个先决条件力，以确保即使目标文件已经存在，该配方也会被执行。我们给强制目标设置一个空的配方，以防止make搜索隐式规则来构建它——否则它会应用相同的任意匹配规则来强制自身，从而形成先决条件循环！

3.7如何将Reads转换为Makefile

GNU make的工作分为两个不同的阶段。在第一阶段，它读取所有.make文件，包括子.make文件等，并将所有变量及其值以及隐式和显式规则内化，构建所有目标及其先决条件的依赖图。在第二阶段，make使用这些内化数据来确定哪些目标需要更新，并运行必要的脚本来更新它们。

了解这种两阶段方法很重要，因为它直接影响变量和函数扩展的方式；这在编写Make文件时常常引起一些混淆。以下是Make文件中可以找到的不同构造及其扩展阶段的总结。

我们说，如果扩展发生在第一阶段，则称为即时扩展：在解析生成文件时，会扩展构造的那部分。如果扩展不是即时进行的，则称为延迟扩展。延迟扩展的构造部分会在使用时延迟扩展：要么是在即时上下文中引用时，要么是在第二阶段需要时。

您可能还不熟悉这些构造。您可以参考本节，以便在以后的章节中熟悉它们。

变量分配

变量定义解析如下：

```
立即=推迟
立即？=推迟
即时：=即时
即时：：=即时
即时：：：：=有逃生的即时
立即+=推迟或立即
立即！=立即

定义即时
  延期的
endef

定义即时=
  延期的
endef

定义即时=
  延期的
endef
```

```
定义即时：=
  即刻的
endef

定义immediate：：=
  即刻的
endef

定义immediate：：：=
  立即-带逃逸
endef

定义即时+=
  推迟或立即
endef

定义即时！=
  即刻的
endef
```

对于附加运算符‘+=’，如果变量以前被设置为简单变量（‘：=’或‘：：=’），则右侧被视为即时；否则，视为延迟。

对于立即转义运算符“：：：=”，右侧的值立即展开，然后转义（即，展开结果中的所有$实例都被替换为$$）。

对于shell任务运算符“！=”，立即对右侧进行求值并将其传递给shell。结果将存储在左侧命名的变量中，并且该变量被视为递归展开的变量（因此将在每次引用时重新求值）。

条件指令

条件指令会立即解析。这意味着，例如，自动变量不能在条件指令中使用，因为自动变量只有在调用该规则的配方时才会设置。如果您需要在条件指令中使用自动变量，必须将条件移到配方中，并改用shell条件语法。

规则定义

规则总是以同样的方式扩展，不考虑形式：

```
即时：即时；延迟
        延期的
```

也就是说，目标部分和先决条件部分会立即展开，并且用于构建目标的配方始终被推迟。这适用于显式规则、模式规则、后缀规则、静态模式规则以及简单的先决条件定义。

3.8如何解析Makefile

GNU make逐行解析makefile。解析过程使用以下步骤：

1.以完整的逻辑行读取，包括反斜杠转义的行（参见第3.1.1节[拆分LongLines]，第12页).

2.删除注释（参见第11页第3.1节[Makefile包含什么]).

3.如果行以配方前缀字符开头并且我们在规则上下文中，则添加将光标移到当前配方的行上并读取下一行（参见第5.1节[配方语法]，第45页).

4.展开在直接扩展上下文中出现的行元素（参见第3.7节[如何制作Reads aMakefile]，第17页).

5.扫描行，寻找分隔符字符，如“：”或“=”，以确定该行是宏赋值还是规则（请参见第5.1节[配方语法]，第45页).

6.将操作结果内化，并读取下一行。

这样做的一个重要后果是，如果宏只有一行，那么它可以扩展为整个规则。这样可以工作：

```
myrule=target：；ec ho建立

$(myrule)
```

但是，这行不通，因为make在扩展它们之后会创建我们的分隔线：

```
定义myrule
目标
        echo已构建
endef

$(myrule)
```

上述生成文件的结果是定义了一个目标“target”，其先决条件为“echo”和“built”，就像生成文件包含target：echo built一样，而不是包含一个规则及其配方。扩展完成后，行尾仍然存在的换行符仍被忽略为普通空白。

为了正确地扩展多行宏，必须使用eval函数：这将导致在扩展后的宏结果上运行make解析器（请参见第8.10[EvalFunction]节，第103页）。

3.9二次膨胀

我们之前已经了解到，GNU make的工作分为两个不同的阶段：读取输入阶段和目标更新阶段（参见第17页第3.7节[如何读取Makefile]). GNU make还具有启用某些或所有目标（仅限）的第二个扩展的前提条件的能力。为了使这个第二个扩展生效，必须在使用此功能的第一个前提条件列表之前定义特殊目标. SECONDEXPANSION。

如果定义了. SECONDEXPANSION，那么当GNU make需要检查目标的先决条件时，这些先决条件将第二次被展开。在大多数情况下，这

二次扩展将不会产生影响，因为所有变量和函数引用在生成文件的初始编译过程中已经进行了扩展。为了利用解析器的二次扩展阶段，需要在生成文件中转义变量或函数引用。在这种情况下，第一次扩展只是转义引用而不进行扩展，而扩展则留到二次扩展阶段。例如，考虑这个生成文件：

```
.SECONDEXPANSION:
ONEVAR = onefile
TWOVAR = twofile
myfile: $(ONEVAR) $$(T WOVAR)
```

第一次扩展阶段后，myfile目标的前置条件列表将变为onefile和$（TWOVAR）；第一个（未转义的）变量引用ONEVAR被扩展，而第二个（已转义的）变量引用则直接转义，但不被视为变量引用。现在，在第二次扩展过程中，第一个词再次被扩展，但由于它不包含任何变量或函数引用，因此保持为onefile的值，而第二个词现在是一个正常的变量TWOVAR引用，扩展后变为twofile的值。最终结果是有两个前置条件，分别是onefile和twofile。

显然，这不是一个非常有趣的案例，因为同样的结果可以通过让两个变量在先决条件列表中以未转义的形式出现而更容易地实现。如果变量被重置，就会出现一个明显的区别；考虑这个例子：

```
.SECONDEXPANSION:
AVAR=顶部
onefile:$(AVAR)
twofile:$$(AVAR)
AVAR=底部
```

这里，onefile的前置条件将立即展开，并解析为top值，而twofile的前置条件直到二级展开后才会完全展开，并产生bottom值。

这稍微令人兴奋一些，但这一功能的真正威力只有当你发现次级扩展总是发生在目标自动变量的范围内时才会显现出来。这意味着你可以在第二次扩展中使用诸如$@、$*等变量，它们将具有预期的值，就像在食谱中一样。你只需要通过转义$来推迟扩展。此外，次级扩展适用于显式和隐式（模式）规则。了解这一点后，这一功能的潜在用途将大大增加。例如：

```
.SECONDEXPANSION:
main_OBJS：=主机。o试验。o
lib_OBJS:= lib .o api .o

主库：$$（$$@_ OBJS）
```

在此，初始扩展之后，主目标和辅助目标的前提条件均为$（$@_OBJS）。在辅助扩展期间，$@变量被设置为主目标的名称，因此主目标的扩展将产生$（main_OBJS），或main.o try .o。

测试.o，而lib目标的二次扩展将产生$（lib_OBJS），或lib .o api .o。

也可以在此处混合函数，只要它们是正确的转义函数：

```
main_SRCS：=主.c试用.c测试.c
lib_SRCS:= lib .c api .c

.SECONDEXPANSION:
主库： $$（patsubst %.c，%.o，$$（$$@_ SRCS））
```

此版本允许用户指定源文件而不是对象文件，但给出的先决条件列表与前面的示例相同。

在二次膨胀阶段评估自动变量，特别是目标名称变量$$@，其行为类似于配方内的评估。然而，由于不同类型的规则定义存在一些细微差异和“边缘情况”，因此需要特别注意。下面将详细描述使用不同自动变量的细微差别。

明确规则的二次扩展

在显式规则的二次扩展过程中，$$@和$$%分别评估目标文件名和，当目标是存档成员时，目标成员名称。$$<变量评估为针对此目标的第一个规则中的第一个先决条件。^{和}+评估为已为同一目标出现的所有规则的先决条件列表（$$+包含重复项，而$$^不包含重复项）。以下示例将有助于说明这些行为：

```
.SECONDEXPANSION:

foo： foo .1 bar. 1 $$< $$^ $$+ # line #1

foo： foo .2 bar. 2 $$< $$^ $$+ #行#2

foo： foo .3 bar .3$$<$$^$$+ # line #3
```

在第一个前提列表中，所有三个变量（$$<、^{和}+）都扩展为空字符串。在第二个列表中，它们分别具有值foo .1、foo .1 bar .1和foo .1 bar .1。在第三个列表中，它们分别具有值foo .1、foo .1 bar .1、foo. 2 bar. 2和foo .1 bar .1、foo .1 bar .1、foo. 2 bar. 2和foo .1 bar .1。

规则按生成文件顺序进行二次扩展，但配方规则总是最后被评估。

变量$$？和$$*不可用，将展开为空字符串。

静态模式规则的二次扩展

静态模式规则的二次扩展规则与上述显式规则相同，只有一个例外：对于静态模式规则，$$*变量被设置为模式基。和显式规则一样，$$？不可用，并且扩展为空字符串。

隐式规则的二次扩展

在搜索隐式规则时，它会替换主干，然后对具有匹配目标模式的每个规则执行二次扩展。自动变量的值以与静态模式规则相同的方式得出。例如：

```
.SECONDEXPANSION:

foo： bar

foo foz: fo%: bo%

%oo: $$<$$^$$+$$*
```

当隐式规则用于目标foo时，$$<展开为bar，$$^展开为bar boo，$$+也展开为bar boo，$$*展开为f。

注意，如第10.8节[隐式规则搜索算法]第137页所述，目录前缀(D)将附加到（扩展后）先决条件列表中的所有模式。例如：

```
.SECONDEXPANSION:

/tmp/foo.o:

%.o：$$（添加后缀/%.c，fo bar）foo .h
        @echo$^
```

在二次扩展和目录前缀重建之后，打印出的先决条件列表将是/tmp/foo Foo.c/tmp/bar Foo.c foo .h。如果您对这种重建不感兴趣，则可以在先决条件列表中使用$$*代替%。

4写作规则

规则出现在生成文件中，它说明了何时以及如何重新创建某些文件，这些文件被称为规则的目标（通常每个规则只有一个目标）。它列出了作为目标的先决条件的其他文件，以及用于创建或更新目标的配方。

规则的顺序并不重要，除非用于确定默认目标：如果未另行指定，则默认目标是第一个make文件中第一条规则的第一个目标。有两个例外：以句点开头的目标除非还包含一个或多个斜杠'/'，否则不被视为默认目标；定义模式规则的目标对默认目标没有影响。（参见第10.5节[定义和重定义模式规则]，第129页。）

因此，我们通常编写生成文件，使第一个规则是编译整个程序或由生成文件描述的所有程序的规则（通常有一个名为"all"的目标）。请参见第9.2节[指定目标的参数]，第109页。

4.1规则语法

一般来说，规则如下：

```
目标：先决条件
        食谱
        ...
```

或者类似这样的：

```
目标：先决条件；配方
        食谱
        ...
```

目标是文件名，用空格分隔。可以使用通配符（参见第4.3节[在文件名中使用通配符]，第25页)和一个形式a的名称，a(m)表示档案文件a中的成员m（见第11.1节[作为目标的档案成员]，第139页）。通常每个规则只有一个目标，但有时需要有多个目标（见第4.9节[规则中的多个目标]，第37页).

配方行以制表符（或recipeprefix变量值中的第一个字符开始；请参阅第6.14节[特殊变量]，第80页).第一个食谱行可能出现在前置条件之后的行中，用制表符分隔，或者在同一行中，用分号分隔。无论哪种方式，效果都是一样的。食谱的语法还有其他差异。参见第5章[按照规则编写食谱]，第45页 .

因为美元符号用于开始可变引用，如果要在目标或先决条件中真正使用美元符号，必须写两个美元符号，'$$（参见第6章[如何使用 变量]，第65页 ）。如果您已启用辅助的次要扩展（请参见第3.9节[次要 扩展]，第19页)，并且您希望在先决条件列表中显示一个实际的美元符号，那么您必须实际写四个美元符号（'$$$$'）。

您可以通过插入一个反斜杠和一个换行符来拆分长行，但这不是必需的，因为make不对makefile中的行长度设置任何限制。

规则告诉我们两件事：目标何时过时，以及在必要时如何更新它们。

过时的标准是根据前提条件来规定的，这些前提条件由空格分隔的文件名组成。（这里也允许使用通配符和存档成员（见第11章[存档]，第139页）。）如果目标文件不存在或比任何前提条件更旧（通过比较最后修改时间），则该目标文件即被视为过时。其理念是，目标文件的内容是基于前提条件中的信息计算得出的，因此如果任何前提条件发生变化，现有目标文件的内容就不再必然有效。

如何更新由一个配方指定。这是一个或多个由shell（通常是sh）执行的行，但有一些额外的功能（见第5章[编写配方规则]，第45页).

4.2先决条件类型

GNU make识别两种不同的先决条件：正常先决条件和仅顺序先决条件。正常先决条件包含两个陈述：首先，它规定了配方调用的顺序：目标的所有先决条件的配方将在目标配方开始之前完成。其次，它规定了依赖关系：如果任何先决条件比目标更新，则认为目标已过期，必须重新构建。

通常情况下，这正是您想要的：如果目标的前提条件更新了，则目标也应更新。

有时您可能希望确保先决条件在目标之前构建，但不强制更新目标，即使先决条件已更新。仅订购的先决条件站点用于创建这种类型的关系。仅订购的先决条件可以通过在先决条件列表中放置管道符号（|）来指定：管道符号左侧的任何先决条件都是常规的；右侧的任何先决条件都是仅订购的：

```
目标：正常先决条件|仅订购先决条件
```

常规前提条件部分当然可以是空的。此外，你仍然可以为同一目标声明多行前提条件：它们会被适当附加（常规前提条件附加到常规前提条件列表中；仅顺序前提条件附加到仅顺序前提条件列表中）。请注意，如果你声明同一个文件既是常规前提条件又是仅顺序前提条件，那么常规前提条件优先（因为它们的行为严格超出了仅顺序前提条件的行为）。

在确定目标是否过期时，仅检查顺序先决条件；即使标记为虚假的顺序先决条件（参见第4.5节[虚假目标]，第31页 ）不会导致目标被重建。

考虑一个例子，你的目标文件需要放置在一个单独的目录中，而这个目录在运行`make`前可能并不存在。在这种情况下，你希望在任何目标文件放置之前先创建该目录，但由于目录的时间戳会随着文件的添加、删除或重命名而变化，我们当然不希望每当目录的时间戳改变时就重建所有目标文件。一种管理方法是使用仅顺序前提条件：将该目录作为所有目标文件的仅顺序前提条件：

```
OBJDIR:= objdir
OBJS := $(addprefix $(OBJDIR)/,foo.o b ar.o baz.o)
```

```
$(OBJDIR)/%.o: % .c
        $(COMPILE.c) $(OUTPUT_OPTION)$<

all:$(OBJS)

$(OBJS):| $(OBJDIR)

$( OBJDIR):
        mkdir $（OBJDIR）
```

现在，如果需要的话，在构建任何.o文件之前，将运行创建objdir目录的规则，但是不会构建.o文件，因为objdir目录的时间戳已更改。

4.3在文件名中使用通配符

单个文件名可以使用通配符指定多个文件。在make中，通配符为“*”、“？”和“[...]”，与Bourne shell中相同。例如，*.c指定所有以“.”结尾的文件（在工作目录中）的列表。

如果一个表达式匹配多个文件，则结果将被排序。1但是，多个表达式不会被全局排序。例如，*.c*.h将列出所有名称以‘.c’结尾的文件，并按此顺序排序，然后是所有名称以‘.h’结尾的文件，并按此顺序排序。

文件名开头的字符‘~’也有特殊意义。单独使用或后面跟一个斜杠时，它代表你的主目录。例如，~/ bin展开为/ home/ you/ bin。如果‘~’后面跟着一个单词，这个字符串就表示该单词所命名的用户的主目录。例如，~john/bin展开为/home/john/bin。在没有为每个用户设置主目录的系统（如MS-DOS或MS-Windows)上，可以通过设置环境变量home来模拟这一功能。

在target ts和prerequisites中，通过make自动执行Wildcard扩展。在配方中，shell负责扩展通配符。在其他上下文中，只有当您明确使用通配符函数请求扩展通配符时，才会发生扩展通配符。

通过在星号前加上反斜杠可以关闭通配符字符的特殊意义。因此，foo*bar将指代一个特定的文件，其名称由foo、星号和bar组成。

4.3.1通配符示例

规则的配方中可以使用通配符，它们由外壳进行扩展。例如，这里有一个删除所有对象文件的规则：

```
干净的
        rm-f* .o
```

在规则的前置条件中使用通配符也很有用。在生成文件中使用以下规则，将打印自上次打印以来所有更改的.c文件：“makeprint”将打印所有这些文件：

```
print:* .c
        lpr-p$?
```

1一些较旧版本的GNU make没有对通配符展开的结果进行排序。

指纹识别

此规则使用打印作为空目标文件；请参见第4.7节[用于记录的空目标文件 活动]，第34页 。（自动变量“$？”用于只打印已更改的文件；请参见第130页10.5.3[自动变量]一节。）

定义变量时，不会发生通配符扩展。因此，如果编写以下代码：objects=* .o

然后变量对象的值是实际字符串“*o*”。然而，如果您在目标或前提条件中使用对象的值，则会进行通配符扩展。如果在配方中使用对象的值，当配方运行时，shell可能会执行通配符扩展。要将对象设置为扩展，请改用：

对象：=$（通配符*.o）

见第4.3.3节[WildcardFunction]，第26页 .

4.3.2使用通配符的陷阱

现在，这里有一个使用通配符展开的简单方法的例子，它不能实现你的意图。假设你想说可执行文件foo是由目录中的所有对象文件组成的，你这样写：

```
objects=* .o

foo:$(objects)
        cc - o foo $（CFLAGS）$（对象）
```

对象的值是实际字符串“*o”。通配符展开发生在foo规则中，因此每个现有的.o文件都成为foo的前提条件，并且如果需要的话将重新编译。

但是，如果你删除了所有的.o文件呢？当通配符不匹配任何文件时，它会保持不变，因此foo将依赖于奇怪命名的文件.o。由于这样的文件很可能不存在，make会给出一个错误，说它无法确定如何生成.o。这可不是你想要的结果！

实际上，使用通配符展开可以得到所需的结果，但需要更复杂的技术，包括wildc ard函数和字符串替换。这些将在下面的章节中介绍。

Microsoft操作系统（MS-DOS和MS-Windows）使用反斜线来分隔路径名中的目录，例如：

```
c:\foo\bar\baz .c
```

这相当于Unix风格的c：/foo/bar/baz.c（其中c：部分是所谓的驱动器字母）。当make在这些系统上运行时，它支持反斜杠以及路径名中的Unix风格正斜杠。然而，这种支持不包括通配符扩展，在此情况下，反斜杠是一个引号字符。因此，在这些情况下，您必须使用Unix风格的斜杠。

4.3.3函数通配符

在规则中，通配符展开会自动进行。但是，当设置变量或函数参数时，通常不会进行通配符展开。如果要在这些位置执行通配符展开，则需要使用通配符函数，例如：

$（通配符模式...)

这段字符串在构建文件的任何位置使用时，会被替换为一个由空格分隔的现有文件名列表，这些文件名与给定的文件名模式之一匹配。如果没有现有文件名与该模式匹配，则该模式将从wildca rd函数的输出中省略。请注意，这与规则中未匹配的通配符行为不同，在规则中，通配符是按字面意思使用的而不是被忽略（参见第4.3.2节[通配符陷阱]，第26页).

与规则中的通配符展开一样，通配符函数的结果也是按顺序排列的。但是，每个单独的表达式都是单独排序的，因此“$（wildcar d*.c*.h）”将展开为所有匹配“c”的文件，按排序，然后是所有匹配“h”的文件，按排序。

使用通配符函数的一个用途是获取目录中所有C源文件的列表，例如：

$(wildcard*.c)

我们可以通过将结果中的“.c”后缀替换为“.o”来将C源文件列表转换为对象文件列表，如下所示：

$(patsubst%.c,%.o,$(wildcard*.c))

（这里我们使用了另一个函数，patsubst。请参见第8.2节[字符串替换和分析函数]，第92页。）

因此，可以编写如下代码来编译目录中的所有C源文件，然后将它们链接在一起：

对象：=$（patsubst%.c，%.o，$（通配符*.c））

```
foo:$(objects)
        cc-o foo$(objects)
```

（这利用了编译C程序的隐式规则，因此无需编写编译文件的显式规则。请参见第6.2节[变量的两种类型]，第66页，解释“：=”，它是“=”的一个变体。)

4.4搜索先决条件目录

对于大型系统，通常希望将源代码放在与二进制文件不同的目录中。make的目录搜索功能通过自动搜索多个目录来查找先决条件，从而实现这一点。当你将文件重新分配到不同的目录时，无需更改单独的规则，只需更改搜索路径即可。

4.4.1 VPATH：搜索所有先决条件站点的路径

make变量VPATH的值指定要搜索的目录列表。通常，这些目录中包含当前目录中不存在的必需文件；但是，make使用VPATH作为规则的必需项和目标的搜索列表。

因此，如果列出的目标或先决条件文件在当前目录中不存在，请搜索VPATH中列出的目录中是否有该名称的文件。如果在其中任何一个目录中找到文件，则该文件可能成为先决条件（见下文）。规则可以指定先决条件列表中的文件名，就像它们都存在于当前目录中一样。参见第4.4.4节[使用DirectorySearch编写配方]，第30页。.

在VPATH变量中，目录名用冒号或空格分隔。列出目录的顺序是make在其搜索过程中遵循的顺序。（在MS-DOS和MS-Windows上，VPATH使用分号作为目录名的分隔符，因为冒号可以在路径名本身中使用，即在驱动器字母之后。）

例如，

```
VPATH = src： ../头文件
```

指定一个路径，其中包含两个目录、src和../ headers，按此顺序进行搜索。

使用此VPATH值，可执行以下规则：

```
foo .o: foo .c
```

被解释为如果它是这样写的：

```
foo .o:src/foo .c
```

假设当前目录中不存在文件foo .c，但在目录src中找到。

4.4.2vpath指令

类似于VPATH变量，但更为选择性的VPATH指令（注意小写）允许你为特定类别的文件名指定搜索路径：即那些符合特定模式的文件。因此，你可以为某一类别文件名提供某些搜索目录，而为其他文件名则提供其他目录（或不提供任何目录）。

vpathdirective有三种形式：

vpath模式直接或方向
: 指定与模式匹配的文件名的搜索路径目录。

  搜索路径、目录，是用于搜索的目录列表，用冒号（MS-DOS和MS-Win操作系统中为分号）或空格分隔，就像在VPATH变量中使用的搜索路径一样。

vpath模式
: 清除与模式n相关的搜索路径。

vpath
: 清除以前使用vpath选项指定的所有搜索路径。

vpath模式是一个包含‘%’字符的字符串。该字符串必须与正在搜索的前提条件的文件名匹配，‘%’字符可以匹配任意零个或多个字符序列（如同模式规则；参见第10.5[定义和重定义模式规则]节，第129页）。例如，%.h匹配以.h结尾的文件。（如果没有‘%’，模式必须与前提条件完全匹配，这在实际应用中很少有用。）

‘%’路径指令模式中的字符可以用前导反斜杠（‘(’\\）引号。原本需要引号的反斜杠可以用更多的反斜杠来引号。用于引号的反斜杠或其他反斜杠在与文件名比较之前会被从模式中移除。不用于引号的反斜杠则不会受到影响。

如果当前目录中不存在某个先决条件，而vpath指令中的模式与该先决条件文件的名称相匹配，则将搜索该指令中的目录，就像搜索（并且在vpath变量中的目录之前）一样。

例如，

vpath%.h ../ headers

如果在当前目录中找不到文件，则告诉您查找任何名称以h结尾的.h文件夹中是否存在该前提条件。

如果多个vpath模式与预请求uisitefile的名称匹配，则逐个处理每个匹配的vpath指令，搜索每个指令中提到的所有目录。按它们在makefile中出现的顺序处理多个vpath指令；具有相同模式的多个指令是相互独立的。

因此

vpath % . c foo
vpath% blish
vpath %. c bar

将在foo、blish、n bar中查找以'.c'结尾的文件，而

vpath% .cfoo： bar
vpath% blish

将搜索foo、bar、blish中以'.c'结尾的文件。

4.4.3如何执行目录搜索

如果通过目录搜索找到一个先决条件，不管其类型（一般或选择性），所找到的路径名可能并不是实际提供给您的先决条件列表中的路径名。有时，通过目录搜索找到的路径名会被丢弃。

算法用于决定是否保留或放弃通过目录搜索找到的路径，如下所示：

1.如果在生成文件中指定的路径上不存在目标文件，则执行目录搜索。

2.如果目录搜索成功，则保留该路径，并将此文件暂存为目标文件。

3.使用相同的方法检查该目标的所有先决条件。

4.处理先决条件后，目标可能需要或不需要重建：

a.如果不需要重建目标，则使用在目录搜索期间找到的文件路径来查找包含该目标的所有先决条件列表。简而言之，如果make不需要重建目标，则使用通过目录搜索找到的路径。

b.如果目标确实需要重建（已过时），那么在目录搜索过程中找到的路径名将被丢弃，目标将使用m ake文件中指定的文件名重新构建。简而言之，如果m ake必须重建，则目标将在本地重建，而不是通过目录搜索找到的目录中重建。

这个算法可能看起来很复杂，但在实践中它往往正是你想要的。

其他版本使用简化算法：如果文件不存在，并且通过目录搜索找到，则无论目标是否需要构建，始终使用该路径名。因此，如果目标需要重建，则会在目录搜索中发现的路径名处创建目标。

如果实际上这是您希望某些或所有目录具有的行为，您可以使用GPATH变量来指示这一点。

GPATH的语法和格式与VPATH相同（即，用空格或冒号分隔的路径名列表）。如果在目录中通过目录搜索找到过时的目标，并且该目录也出现在GPATH中，则不会丢弃该路径名。目标将使用扩展后的路径重建。

4.4.4使用目录搜索编写配方

如果通过目录搜索在另一个目录中找到一个先决条件，这将不会改变规则的配方；它们将按所写的方式执行。因此，您必须仔细编写配方，以便它能在make找到它的目录中查找先决条件。

这是通过自动变量实现的，例如‘$^’（见第10.5.3[AutomaticVariables]节，第130页）。例如，‘$^’的值是一个包含规则所有前提条件的列表，包括它们所在目录的名称，而‘$@’的值是目标。因此：

```
foo .o: foo .c
        cc -c$(CFLAGS)$^-o$@
```

（变量CFLAGS存在是为了通过隐式规则指定C编译的标志；我们在这里使用它是为了保持一致性，因此它将均匀地影响所有C编译；参见第10.3节【隐式规则使用的变量】，第125页。）

通常，先决条件还包括头文件，您不希望在配方中提及这些头文件。自动变量“$<”只是第一个先决条件：

```
VPATH = src： ../ headers
foo .o： foo .c defs .h hack.h
        cc -c$(CFLAGS)$<-o$@
```

4.4.5目录搜索和隐式规则

在考虑隐式规则时（参见第10章[使用隐式规则]，第121页），也会搜索VPATH中指定的目录或VPATH。

例如，当文件foo .o没有显式规则时，make会考虑隐式规则，比如如果存在foo .c文件，则编译foo .c。如果当前目录中缺少这样的文件，则会在适当目录中搜索它。如果foo .c存在于任何目录中（或在make文件中提及），则应用C编译的隐式规则。

隐式规则的配方通常出于必要而使用自动变量；因此，它们将使用通过目录搜索找到的文件名，无需额外的努力。

4.4.6链接库目录搜索

目录搜索以一种特殊的方式适用于与链接器一起使用的库。当您编写一个名称形式为“-lname”的先决条件时，这一特殊功能就会发挥作用。（您可以判断这里有些奇怪的事情正在发生，因为先决条件通常是文件的名称，而库的文件名通常看起来像libname .a，而不是“-lname”。）

如果某个先决条件的名称具有形式“-lname”，则通过搜索文件libname .so，以及如果未找到，则在当前目录中搜索文件libname.a来特别处理它，

在通过匹配的vpath搜索路径和vpath搜索路径指定的目录中，然后在目录/lib、usr/lib和prefix/lib（通常为/usr/local/lib，但MS-DOS/MS-Windows版本的make会将prefix定义为DJGPP安装树的根目录）中。

例如，如果您的系统上存在usr/lib/libcurses.a库（而没有usr/lib/libcurses.so文件），那么

```
foo: foo .c-lcurses
        cc$^-o$@
```

会导致当foo比foo .c或usr/lib/libcurses.a更老时，执行命令ccfoo.c/usr/lib/libcurses.a-ofoo。

虽然默认搜索的文件集是libname .so和libname .a，但可以通过.LIBPATTER NS变量进行自定义。此变量值中的每个单词都是一个模式字符串。当看到像'-lname'这样的前置词时，make将用名称替换列表中每个模式中的百分号，并使用每个库文件名执行上述目录搜索。

默认情况下，. LIBPATTERNS的值为"lib%. so lib%.a"，因此提供了上述默认行为。

通过将此变量设置为空值，可以关闭链接库扩展完成。

4.5假目标

假目标不是文件的真正名称，而只是在您发出明确请求时执行的配方的名称。使用假目标有两个原因：避免与同名文件发生冲突，以及提高性能。

如果编写一个规则，其配方将不会创建目标文件，则每次目标出现需要重新制作时，都会执行该配方。下面是一个示例：

```
干净的
        rm* .o temp
```

因为rm命令不会创建名为clean的文件，所以很可能永远不会存在这样的文件。因此，每次您说"makeclean"时，rm命令都会执行。

在这个例子中，如果在此目录下创建了一个名为cl ean的文件，那么cl ean目标将无法正常工作。由于它没有前置条件，cl ean会一直被视为最新版本，其配方也不会被执行。为了避免这个问题，你可以显式声明该目标为虚假目标，使其成为特殊目标. PHONY的前置条件（参见第4.8节[特殊内置目标名称]，第34页) asfollows:

```
.假：干净
干净的
        rm* .o temp
```

完成此操作后，无论是否存在名为clean的文件，"makeclean"都将运行该程序。

.phy的先决条件总是被解释为字面目标名称，而不是模式（即使它们包含"%"字符）。要始终重建模式规则，请考虑使用"强制目标"（请参见第4.6节[没有配方或先决条件的规则]，第33页).

虚假目标也与递归调用make一起使用很有用（参见第56页第57节[递归使用make]）。在这种情况下，生成文件通常会包含一个变量，用于列出要构建的子目录。一种简单的处理方法是定义一个规则，使用一个循环子目录的配方，如下所示：

```
子目录= foo bar b az

subdirs:
        for dir in$（SUBDIRS）；do\
          $（制造）-C$$目录；
        已完成的
```

然而，这种方法存在一些问题。首先，该规则会忽略子目录中检测到的任何错误，因此即使有一个目录失败，它仍会继续构建其余目录。可以通过添加shell命令来记录错误并退出，但这样一来，即使使用-k选项调用make也会触发错误，这是不幸的。其次，也许更重要的是，你无法充分利用make并行构建目标的能力（参见第51页第5.4节[并行执行]。），因为只有一个规则。每个单独的生成文件的目标将并行构建，但一次只构建一个子目录。

通过将子目录声明为.fake target（您必须这样做，因为子目录显然总是性别歧视；否则它将不会被构建），您可以删除这些问题：

```
子目录= foo bar b az

. PHONY：子目录$（子目录）

subdirs: $( SUBDIRS )

$(SUBDIRS):
        $(MAKE)-C$@

foo: baz
```

我们还声明，在bazsub-directory完成之前不能构建foosub-directory；这种关系声明在尝试并行构建时特别重要。

对于PHON Y目标，将跳过隐式规则搜索（见第10章[隐式规则]，第121页）。这就是为什么将目标声明为. PHON Y对于性能有益的原因，即使您不担心实际文件是否存在。

虚假目标不应成为真实目标文件的前提条件；如果它是，每当make考虑该文件时，其脚本都会被运行。只要虚假目标从未成为真实目标的前提条件，虚假目标的脚本就只会当虚假目标被指定为目标时执行（参见第9.2[指定目标的参数]节，第10 9页）。

你不应该将包含的生成文件声明为虚假文件。虚假目标并不是用来表示真实文件的，而且由于目标总是被认为是过时的，所以生成文件总是会重建它然后重新执行自身（参见第3.5节[如何重建生成文件]，第15页 ）。为避免这种情况，如果包含的文件被标记为虚假文件，则make不会重新执行自身。

虚假目标可以有前置条件。当一个目录包含多个程序时，最方便的做法是在一个makefile中描述所有程序。由于默认情况下重新构建的目标将是makefile中的第一个目标，因此通常会将其命名为‘all’并作为前置条件提供所有单独的程序。例如：

```
全部：prog1 prog2 prog3
.假：全部

prog1： prog1 .o utils .o
        cc-o prog1 prog1 .o utils .o

程序2：程序2 .o
        cc - o prog 2 prog 2 .o

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

现在，您可以说“make”来重新创建所有三个程序，或者指定要重新创建的程序（如“make prog1prog3”）。虚假性不是继承的：虚假目标的前提条件本身不是虚假的，除非明确声明为如此。

当一个假目标是另一个假目标的前提条件时，它就作为另一个假目标的子程序。例如，这里“mak ecleanall”将删除对象文件、差异文件和文件程序：

```
. PHONY： cleanall cleanobj cleandiff

cleanall : cleanobj cleandiff
        rm程序

cleanobj:
        rm * .o

cleandiff :
        rm* .diff
```

4.6没有配方或先决条件的规则

如果规则没有先决条件或配方，而规则的目标是不存在的文件，则假定在运行规则时目标已更新。这意味着所有依赖于此目标的其他目标都将始终运行其配方。

下面举个例子说明：

```
清洁：强制
        rm$(objects)
力：
```

在这里，目标力满足特殊条件，因此依赖于它的目标清洁剂被迫运行其配方。名称力没有什么特别之处，但这是常用的一种名称。

如你所见，用这种方式使用“force”与使用“.phony：clean”有相同的结果。

使用.phony更加明确且更有效。然而，其他版本的make do不支持.phony；因此，在许多makefile中出现了.force。请参见第4.5节[Phon y 目标]，第31页 .

4.7清空目标文件以记录事件

空目标是假目标的变体；它用于保存您不时明确要求执行的操作的配方。与假目标不同，此目标文件可以真实存在；但是文件的内容并不重要，通常为空。

空目标文件的目的是记录规则的配方最后一次执行的时间，以及最后一次执行该配方的时间。它之所以这样做是因为配方中的一个命令是touch命令，用于更新目标文件。

空目标文件应该有一些先决条件（否则没有意义）。如果要求重新创建空目标，则如果任何先决条件比目标更新，则将执行该配方；换句话说，如果自上次重新创建目标以来先决条件已更改，则将执行该配方。下面是一个示例：

```
打印： foo .c bar .c
        lpr-p$?
        指纹识别
```

使用此规则，如果自上次执行“打印”以来任一源文件发生更改，则“打印”将执行lpr命令。自动变量“$？”用于仅打印已更改的文件（请参见第130页10.5.3[自动变量]一节）。

4.8特定内置目标名称

如果某些名称作为目标出现，则具有特殊含义。

.假的

特殊目标. PHON Y的先决条件被认为是虚假目标。当需要考虑此类目标时，make将无条件地运行其配方，无论是否存在该名称的文件或其最后修改时间是什么。请参见第4.5[虚假目标]节，第31页。.

.后缀

特殊目标. SUFFIXE S的先决条件是要列出的后缀列表 用于检查后缀规则。请参见第10.7节[过时的后缀规则]， 第136页。

默认

为任何没有规则的目标指定的. DEFAULT配方都将使用 发现（明确规则或隐含规则）。参见第10.6节[最后手段]， 第135页。如果指定了一个默认的配方，那么所有提到的文件都必须是aprerequi- 站点，而不是作为规则的目标，将代表它执行该配方。 请参见第137页第10.8节[隐式规则搜索算法]。

.价值连城

这些目标的珍贵属性取决于以下特殊治疗：如果在执行它们的配方时被杀死或中断，

目标未被删除。请参见第55页的第5.6节[中断或终止make] . 此外，如果目标是中间文件，则在删除后不会被删除 不再需要，通常这样做。请参见第10.4节[隐式规则链]， 第127页。在后一方面，它与次要特殊目标重叠。

您还可以将隐式规则的目标模式（例如“%.o”）作为特殊目标的先决文件，以便保留目标模式与该文件名称相匹配的规则创建的中间文件。

.中间

中间依赖的目标被视为中间文件。请参见第10.4[隐式规则链]节，第127页。没有先决条件的中间目标没有效果。

.NOTINTERMEDIATE

特殊目标的先决条件。没有先决条件的不是中间文件。请参见第10.4节[隐式规则链]，第127页。没有先决条件的不是中间文件，因此所有目标都被视为不是中间文件。

如果先决条件是目标模式，则使用该模式规则构建的目标不被视为中间件。

.二级

次级依赖的目标被视为中间文件，但它们永远不会被自动删除。请参见第10.4节[隐式规则链]，第127页。

。在某些特殊情况下，可以使用次级来避免冗余重建。例如：

你好，bin：你好。再见。
$(CC)-o$@$^

%.o:% .c
$(CC) -c-o$@$<

.二级：嗨，再见。

假设hello .bin的源文件是最新的，但对象文件hello. o缺失了。如果没有. SECONDARY make，即使源文件没有变化，也会重建hello. o，进而重建hello .bin。通过声明hello. o为. SECONDARY make，它将不需要重新构建，也不会需要重建hello .bin。当然，如果其中一个源文件更新了，则所有对象文件都需要重建，以确保hello .bin的生成成功。

.没有先决条件的次要目标会使所有目标都被视为次要（即，不会删除任何目标，因为它们被视为中间目标）。

.SECONDEXPANSION

如果在生成文件中任何地方提到. SECONDEXPANSION作为目标，则在它之后定义的所有先决条件列表将再次被扩展

毕竟已经读取了所有makefile。请参见第3.9节[SecondaryExpansion]，第19页。

.DELETE_ON_ERROR

如果在生成文件中任何地方提到.DELETE_ON_ERROR作为目标，那么生成程序将删除规则的目标，如果它已更改并且其配方退出状态非零，就像它接收到信号时一样。请参见第5.5节 [食谱中的错误]，第54页 .

忽略

如果为.IGNORE指定了先决条件，则将忽略对这些特定文件的配方执行过程中的错误。将忽略.IGNORE的配方（如果有的话）。

如果作为目标项提及而没有前置条件，则忽略执行所有文件的错误。Ignor E命令建议忽略所有文件中配方执行时的错误。这种使用‘.ignore’的方式仅出于历史兼容性而支持。由于这会影响生成文件中的每个配方，因此并不十分有用；我们建议您使用更选择性的方法来忽略特定配方中的错误。参见第5.5节[配方中的错误]，第54页。 .

低分辨率时间

如果为. LOW_RESOLUTION_TIME指定先决条件，则假设这些文件是由生成低分辨率时间戳的命令创建的。将忽略. LOW_RESOLUTION_TIME目标的配方。

许多现代文件系统的高分辨率文件时间戳减少了错误地认为文件已更新的可能性。不幸的是，一些主机不提供设置高分辨率文件时间戳的方法，因此像‘cp-p’这样的命令在显式设置文件时间戳时必须丢弃其子秒部分。如果文件是通过这样的命令创建的，您应该将其列为. LOW_RESOLUTION_TIME的先决条件，以防止编译器错误地认为该文件已过期。例如：

```
LOW_RESOLUTION_TIME：dst
dst: src
        cp-p src dst
```

由于cp-p丢弃了src的时间戳的子秒部分，即使dst的时间戳是当前时间，它通常也比src稍早。. LOW_RESOLUTION_TIME行使得make认为dst是当前时间，如果其时间戳正好位于与src时间戳同秒的开始。

由于存档格式的限制，存档成员的时间戳总是低分辨率的。您不需要将存档成员作为. LOW_RESOLUTION_TIME的先决条件，因为make会自动完成此操作。

沉默的

如果为. SILENT指定了先决条件，则在执行这些文件之前，make将不会打印用于重新创建这些特定文件的配方。将忽略. SILENT的配方。

如果被提及为一个没有先决条件的目标，则. SILENT将表示在执行这些目标之前不打印任何配方。您也可以使用更选择性的方式来使特定的配方命令行保持静默。请参见第5.2节[RecipeEchoing]，第47页 . I f

若要对特定运行的制作过程中的所有配方进行静音，请使用`-s‘或`--silent’选项（请参见第9.8节[选项摘要]，第114页）。

导出所有变量

仅仅因为被提及为目标，这就会告诉制造企业要输出所有变量 默认情况下，这是对使用export的一种替代方法。 没有争论。请参见第5.7.2节[向子制造过程传递变量] 第57页。

.NOTPARALLEL

如果. NOTPARALLEL被提及为一个没有先决条件的目标，则此调用中的所有目标都将串行运行，即使给出了`-j‘选项。任何递归调用的make命令仍然会并行运行配方（除非它的make le也包含此目标）。

如果. NOTPARALLEL的目标作为先决条件，则所有这些先决条件 目标将被串行运行。这隐式地在每个预- 所列目标的必要条件。请参见第5.4.1节[禁用并行执行]， 第51页。

.ONESHELL

如果将. ONESHEL L指定为目标，则在构建目标时，所有配方行都将被传递给shell的单次调用，而不是将每个行分别调用。请参阅第5.3[配方执行]节，第48页.

可移植性操作系统接口

如果提到.POSIX作为目标，则make le将在符合POSIX的模式下解析并运行。这并不意味着只有符合POSIX的make le被接受：所有高级的GNU make功能仍然可用。相反，这个目标使得make在那些默认行为不同的领域按照POSIX的要求运行。

特别是，如果提到此目标，则将调用食谱，就像传递了shell-eag：一样：食谱中的第一个失败命令将立即导致该食谱失败。

任何定义的隐式规则，如果作为目标出现，也可以被视为特殊目标，两个隐式规则的连接也是如此，例如`.c .o‘。这些目标是隐式规则的特殊形式，是一种过时的定义隐式规则的方法（但仍然广泛使用）。原则上，任何目标名称都可以通过将其分成两部分并将其全部添加到隐式规则列表中而成为特殊目标。实际上，隐式规则通常以`.’开头，因此这些特殊目标名称也以`.‘开头。参见第10.7节[传统隐式规则]，第136页。

4.9规则中的多个目标

当一个明确的规则有多个目标时，可以采用两种可能的方法之一：作为独立的目标或作为分组的目标。它们的处理方式由出现在目标列表之后的分隔符决定。

具有独立目标的规则

使用标准目标分隔符：，：的规则定义独立的目标。这等同于为每个目标编写一次相同的规则，但前提条件和配方重复。

通常，该配方将使用诸如“$@”之类的自动变量来指定正在构建的目标。

具有独立目标的规则在两种情况下很有用：

您需要的是先决条件，而不是方案。例如：

```
kbd. o命令.o files.o：命令.h
```

为提到的三个对象文件中的每一个提供了额外的前提条件。它等同于以下两行代码：

```
kbd.o： command.h
command.o： command.h
files.o： command.h
```

- 对于所有目标，类似的配方都适用。可以使用自动变量“$@”来替换要重新编译的特定目标到命令中（请参见第130页的10.5.3[自动变量]部分）。例如：

```
大输出小输出：文本.g
        生成文本.g-$（substoutput，，$@）>$@
```

等同于

```
bigoutput： text .g
        生成文本.g-big> bigou tput
小输出：文本.g
        生成文本.g-little>小输出
```

这里我们假设假设程序生成两种类型的输出，一种是给定的“大”输出，另一种是给定的“小”输出。关于子替换函数的解释，请参见第8.2节[字符串替换和分析函数]，第92页。

假设您希望根据目标来更改先决条件，就像变量‘$@’允许您更改配方一样。在普通规则中，您不能对多个目标这样做，但可以使用静态模式规则来实现。请参见第4.11节[静态模式规则]，第40页 .

具有分组目标的规则

如果使用的是生成多个文件的单个调用的配方，而不是独立的目标，则可以通过声明使用分组目标来表达这种关系。分组目标规则使用分隔符&：（这里的“&”用于表示“所有”）。

当构建组中的任何一个目标时，make认为组中的其他所有目标也会因调用配方而更新。此外，如果只有部分组目标过期或缺失，make会意识到运行配方将更新所有目标。最后，如果组中的任何目标过期，所有组目标都将被视为过期。

例如，此规则定义了组目标：

```
foo bar biz&： baz boz
        echo $^> foo
        echo $^>柱
        echo $^ >业务
```

在执行分组目标的配方时，自动变量“$@”将被设置为触发规则的组中的特定目标的名称。在分组目标规则的配方中依赖此变量时必须小心。

与独立目标不同，分组目标规则必须包含一个配方。但是，分组目标的成员也可以出现在没有配方的独立目标规则定义中。

每个目标可能只关联一个配方。如果一个分组的目标出现在独立目标规则中或另一个包含配方的分组目标规则中，您将收到警告，且后者配方将替换前者配方。此外，该目标将从之前的组中移除，并仅出现在新组中。

如果您希望某个目标出现在多个组中，则必须在声明包含该目标的所有组时使用双冒号分隔符&：：。分组的双冒号目标被视为独立的，且每个分组双冒号规则的配方最多只执行一次，前提是其多个目标中的至少一个需要更新。

4.10一个目标的多个规则

一个文件可以成为多个规则的目标。所有规则中提到的所有先决条件都合并到目标的先决条件列表中。如果目标比任何规则中的任何先决条件都要旧，则执行该配方。

对于一个文件，只能执行一个配方。如果多个规则都为同一个文件提供了配方，请使用最后一个给出的，并打印错误消息。（作为特殊情况，如果文件名以点开头，则不会打印错误消息。这种奇特的行为仅是为了与其他实现的make兼容……你应该避免使用它）。偶尔，让同一个目标调用多个在你的make文件不同部分定义的配方是有用的；你可以使用双冒号规则（参见第4.12 节 。
[双结肠]，第42页 为此

额外规则只需提供前提条件即可一次性给多个文件添加几个额外的前提条件。例如，生成文件通常包含一个变量，如`objects`，其中列出了正在构建的系统中所有编译输出文件。如果配置文件`config.h`发生更改，则需要重新编译所有这些文件，一种简便的方法是编写如下代码：

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

此选项可以插入或删除，而无需更改真正指定如何生成对象文件的规则，因此如果希望间歇性地添加额外的前提条件，则使用此选项非常方便。

另一个问题是，可以使用通过命令行参数设置的变量来指定附加的前提条件（请参见第9.5节[OverridingVariables]，第113页）。例如，

```
extradeps=
$(objects) : $(extradep s)
```

这意味着命令makeextradeps=foo.h将把foo .h视为每个目标文件的先决条件，但普通的make不会这样做。

如果目标的任何显式规则都没有配方，则搜索适用的隐式规则以找到一个，参见第10章[使用隐式规则]，第121页）。

4.11静态模式规则

静态模式规则是规定多个目标的规则，并且根据目标名称构建每个目标的前提条件名称。它们比具有多个目标的普通规则更为通用，因为这些目标不必具有相同的前提条件。它们的前提条件必须是相似的，但不一定是相同的。

4.11.1静态模式规则的语法

以下是静态模式规则的语法：

```
目标...：target-pattern：预先模式...
        食谱
        ...
```

目标列表指定了规则适用的目标。目标可以包含通配符字符，就像普通规则的目标一样(请参见第4.3节[使用通配符 “FileNames]”中的字符，第2页).

目标模式和前置条件模式说明了如何计算每个目标的前置条件。每个目标都与目标模式匹配，以提取目标名称的一部分，称为词干。这个词干被代入每个前置条件模式中，生成前置条件名称（每个前置条件模式一个）。

每个模式通常只包含一次字符“%”。当目标模式与目标匹配时，“%”可以匹配目标名称的任何部分；这部分称为词干。模式的其余部分必须完全匹配。例如，目标foo .o与模式%.o匹配，其中“foo”是词干。tar获取foo .c和foo .out则不匹配该模式。

每个目标的前提条件名称是通过将每个前提模式中的‘%’替换为stem生成的。例如，如果一个前提模式是%.c，则将stem‘foo’替换后得到的前提条件名称为foo .c。可以合法地编写不包含‘%’的前提条件模式；那么这个前提条件对所有目标都是相同的。

‘%’模式规则中的字符可以用前导反斜杠（‘(’\\）引出。原本需要使用反斜杠来引出‘%’字符的反斜杠可以用更多的反斜杠来引出。在模式与文件名比较或替换为茎之前，会移除引出‘%’字符或其他反斜杠的反斜杠。不会被引出‘%’字符的反斜杠则不受影响。例如，模式‘the%weird\\%pattern’中，在操作符‘%’前有‘the%weird’，在操作符‘pattern’后有‘pattern’。最后两个反斜杠保持不变，因为它们不会影响任何‘%’字符。

下面是一个示例，它将foo. o和bar. o从相应的.c文件中编译出来：

```
对象=foo.o bar .o

all:$(objects)

$( objects):%.o:% .c
        $(CC) -c$(CFLAGS)$<-o$@
```

这里，‘$<’是自动变量，它保存了先决条件的名称；‘$@’是自动变量，它保存了目标的名称；请参见Section10.5.3 [自动变量]，第130页。

每个指定的目标都必须与目标模式匹配；对于不匹配的目标，会发出警告。如果您有一份文件列表，其中只有部分文件与模式匹配，您可以使用过滤函数来移除不匹配的文件名（参见第8.2节[字符串替换和分析函数]，第92页）：

```
文件=foo.elc bar .o lose .o

$(filter%.o,$(files)):%.o:% .c
        $(CC) -c$(CFLAGS)$<-o$@
$(filter%.elc,$(files)):%.elc:% .el
        emacs-fbatch-byte-compile$<
```

在这个例子中，‘$（filter%.o，$（files））’的结果是bar .o lose .o，第一个静态模式规则导致每个对象文件通过编译相应的C源文件来更新。‘$（filter%.elc，$（files））’的结果是foo .elc，因此该文件由foo .el构成。

另一个示例说明如何在静态模式规则中使用$*：

```
大输出小输出：%输出：文本.g
        生成文本.g-$*>$@
```

运行生成命令时，$*将扩展为stem，即“big”或“little”。

4.11.2静态模式规则与隐式规则

静态模式规则与定义为模式规则的隐式规则有许多共同之处（见第10.5节[定义和重新定义模式规则]，第129页）。两者都包含目标模式和构建先决条件名称的模式。不同之处在于make如何决定规则何时适用。

隐式规则可以应用于任何与之模式相匹配的目标，但仅当目标没有其他指定的配方，并且前提是能够找到时才适用。如果出现多个适用的隐式规则，则只应用其中一个；选择取决于规则的顺序。

相比之下，静态模式规则适用于在规则中指定的精确目标列表。它不能应用于任何其他目标，并且它总是应用于指定的所有目标。如果两个冲突的规则都适用，并且都具有配方，则这是错误的。

静态模式规则比隐式规则更好，原因如下：

对于一些名称无法通过语法进行分类，但可以以显式列表形式给出的文件，您可能希望覆盖通常隐含的规则。

如果你无法确定正在使用的目录的确切内容，你可能不确定哪些无关文件会导致使用错误的隐式规则。选择可能取决于隐式规则搜索的顺序。对于静态模式规则，不存在不确定性：每条规则都精确地适用于指定的目标。

4.12双结规则

双冒号规则是用‘：：’而不是‘：’在目标名称后编写的明确规则。当同一个目标出现在多个规则中时，它们的处理方式与普通规则不同。具有双冒号的模式规则有完全不同的含义（见第10.5.5[匹配-任意规则]节，第134页）。

当目标出现在多个规则中时，所有规则必须是同一类型：全部普通规则或全部双冒号规则。如果它们是双冒号规则，则每个规则都独立于其他规则。每个双冒号规则的配方会在目标比该规则的任何先决条件更老时执行。如果没有该规则的先决条件，其配方总是会执行（即使目标已经存在）。这可能导致不执行、部分执行或全部执行双冒号规则。

具有相同目标的双冒号规则实际上完全独立于另一个规则。每个双冒号规则都单独处理，就像处理具有不同目标的规则一样。

目标的双冒号规则按照它们在生成文件中出现的顺序执行。然而，当执行食谱的顺序无关紧要时，双重结肠规则才是有意义的。

双冒号规则有些晦涩难懂，而且通常没有多大用处；它们提供了一种机制，用于处理目标t的更新方法取决于哪些先决文件导致更新的情况，而这种情况是r。

每个双冒号规则都应指定一个配方；如果未指定，则在应用时将使用隐式规则。请参见第10章[使用隐式规则]，第121页。

4.13自动生成预置站点

在程序的生成文件中，需要编写的许多规则通常只说某个目标文件依赖于某个头文件。例如，如果main .c通过include导入了defs .h，那么您将编写：

```
main.o：def.h
```

你需要这条规则，以便让make知道每当defs .h发生变化时必须重新生成main。你可以看到，对于一个大型程序，你必须在你的make文件中编写几十条这样的规则。而且，每次添加或删除一个#include时，你都必须非常小心地更新make文件。

为了避免这种麻烦，大多数现代C编译器可以通过查看源文件中的#include行来为你编写这些规则。通常这是通过编译器的‘-M’选项来完成的。例如，命令：

```
cc - M main .c
```

生成输出：

```
main .o：main. c defs .h
```

这样你就不用自己写所有这些规则了，编译器会为你完成。

请注意，这样的规则构成在make文件中提及main .o，因此它永远不能通过隐式规则搜索被视为中间文件。这意味着使用它之后make永远不会删除该文件；请参见第10.4节[隐式规则链]，第127页。

使用旧的make程序时，传统做法是利用编译器功能通过类似命令“makedepend”来按需生成前提条件。该命令会创建一个包含所有自动生成的前提条件的文件；然后make文件可以使用 include 来读取这些前提条件（参见第 3.3 节 [Include]，第 13 页。).

在GNU make中，重新生成make文件的功能使这种做法变得过时了——你永远不需要明确地告诉make重新生成必需项，因为它总是会重新生成任何过时的make文件。请参见第3.5节[重新生成make文件]，第15页 .

我们推荐的自动先决条件生成实践是为每个源文件创建一个对应的生成文件。对于每个源文件名.c，都有一个生成文件名.d，列出了目标文件名.o所依赖的文件。这样只需重新扫描那些已更改的源文件，即可生成新的先决条件。

下面是生成一个名为name.d的先决条件文件（即，一个make文件）的模式规则，该文件是从名为name.c的C源文件生成的：

```
%.d:% .c
        @ set -e；rm -f $@；\
          $(CC) -M $(CPPFLAGS)$<>$@.$$$$;\
          sed s,\($*\)\.o[:]*,\1.o $@ :,g < $@.$$$$ > $@;\
          rm -f$@.$$$$
```

有关定义模式规则的信息，请参见第129页的第10.5[PatternRules]节。shell的-e标志如果$（CC）命令（或任何其他命令）失败，则立即退出（退出时状态非零）。

使用GNU C编译器时，您可能希望使用“-MM”标志而不是“-M”。这将省略系统头文件的先决条件。有关详细信息，请参阅《使用GNU CC》中的“控制预处理器的选项”一节。

sed命令的目的是转换（例如）：

```
main .o： main. c defs
```

.hinto：

```
main .o main.d： main .c defs .h
```

这使得每个.d文件都依赖于相应的.o文件所依赖的所有源文件和头文件。这样一来，每当源文件或头文件中的任何一个发生变化时，make就会知道必须重新生成这些先决条件。

定义了重制.d文件的规则之后，就可以使用include指令来读取所有这些文件了。请参见第3.3节[Include]，第13页 . Forexample:

```
源文件=foo.c bar .c

include$(sources:.c=.d)
```

（此示例使用替换变量引用，将源文件列表“foo .c bar.c”转换为先决条件生成文件列表“foo.d bar.d”。请参见Section6.3.1 [替换 参考文献]，第69页 ，有关替换的完整信息，请参见参考资料。）由于“.d”文件和其他文件一样是生成文件，因此如果需要，make会重新生成它们，无需您进一步操作。请参见第3.5节[重新生成生成文件]，第15页 .

注意，.d文件包含目标定义；您应该确保在您的makefile中第一个默认目标之后放置include指令，否则可能会使随机对象文件成为默认目标。请参见第5页的第2.3节[Make的工作原理] .

5用规则编写配方

规则的配方由一个或多个要按它们出现的顺序依次执行的shell命令行组成。通常，执行这些命令的结果是使规则的目标更新。

用户使用许多不同的外壳程序，但是，除非makefile中指定了其他方式，否则makefile中的recipe总是由/bin/sh解释。请参见第5.3节[Recipe执行]，第48页 .

5.1配方语法

makefile具有一个不寻常的特性，即在一个文件中实际上有两种不同的语法。大多数makefile使用make语法（请参见第3章[编写Makefile]，第1 1页 ）。然而，配方是用shell语言编写的，因此它们的解释是由shell来完成的。make程序并不试图理解shell语言：它只是对配方的内容进行少量特定的转换，然后将其交给shell。

配方中的每一行都必须以制表符（或RECIPEPREFIX变量值的第一个字符开始；请参见第6.14节[特殊变量]，第80页 ），除了第一条配方行可以与目标和前提条件行用分号连接。makefile中任何以制表符开头且出现在“规则上下文”（即，在开始一个规则直到另一个规则或变量定义之后）的行都将被视为该规则的配方的一部分。空白行和仅包含注释的行可以在配方行之间出现；它们会被忽略。

这些规则的一些后果包括：

以制表符开头的空行不是空白：它是一个空配方（见第5.9节[EmptyRecipes]，第62页).

•配方中的注释不是注释；它将传递给shellas-is。无论外壳是否将其视为注释，这取决于您的外壳。

在“规则上下文”中，如果第一个字符是按制表符缩进的变量定义，则该变量定义将被视为配方的一部分，而不是生成变量定义，并传递给shell。

条件表达式(ifdef、ifeq等，请参见第7.2节[条件语句的语法]，第86页)在“规则上下文”中，该上下文以制表符缩进为行首字符，将被视为配方的一部分并传递给shell。

5.1.1分割配方行

在编写make文件时，解释食谱的一种方法是在换行符前加上反斜杠。就像常规的make文件语法一样，可以通过在每个换行符前放置反斜杠，将单个逻辑食谱行拆分为多个物理行。这样的一系列行被视为一条食谱行，系统会调用一个shell实例来运行它。

但是，与在生成文件的其他地方处理它们的方式相反（参见Section3.1.1 [Splitting LongLines]，第12页 ），反斜杠/换行符对不会从配方中移除。反斜杠和换行符都会被保留并传递给shell。反斜杠/换行符的解释方式取决于你的shell。如果第一个

backslash/newline之后的下一行的特性是配方前缀字符（默认为制表符；请参见Section6.14 [SpecialVariables]，第80页 ），然后删除该字符（且仅删除该字符）。配方中不会添加空白。

例如，此生成文件中所有目标的配方：

```
所有
        @echo no\
空间
        @echo no\
        空间
        @echo one\
        空间
        @echoone\
         空间
```

由四个独立的外壳命令组成，输出为：

```
无空间
无空间
一个空格
一个空格
```

作为一个更复杂的示例，此生成文件：

```
全部：；@echo‘hello\
          世界；echo“hello\
      世界
```

将用以下命令调用一个外壳：

```
echohello\
世界；echo“hello”
     世界
```

根据壳体引用规则s，将产生以下输出：

```
hello\
世界
你好世界
```

请注意，双引号（“...”）中引用的字符串内部的反斜杠/换行符pai r被移除了，而单引号（‘...’）中引用的字符串则没有。这是默认shell（/bin/sh）处理反斜杠/换行符对的方式。如果你在Make文件中指定了不同的shell，它可能会以不同的方式处理这些字符。

有时你希望在单引号内拆分一个长行，但又不希望引号内容中出现反斜杠/换行符。这种情况通常发生在将脚本传递给如Perl等语言时，因为脚本内部多余的反斜杠可能会改变其含义，甚至导致语法错误。一种简单的处理方法是将要引用的字符串，甚至是整个命令，放入一个变量中，然后在配方中使用该变量。在这种情况下，将使用make文件的换行符引用规则，反斜杠/换行符将被移除。如果我们用这种方法重写上面的例子：

```
你好=‘你好\
世界

所有：；@ echo $（HELLO）
```

我们将得到如下输出：

```
你好世界
```

如果您喜欢，也可以使用目标特定变量（参见Section6.11[目标特定VariableValues]，第78页)以获得变量和使用它的配方之间的更紧密的对应关系。

5.1.2在食谱中使用变量

通过扩展其中的任何变量引用来制作过程配方的另一种方法是（参见第65页第6.1节[Reference]）。这发生在ma ke读完所有Makefile后，目标文件被确定为过时之后；因此，未重新构建的目标的配方永远不会被展开。

变量和函数引用在配方中具有与其他地方相同的语法和语义。它们还遵循相同的引号规则：如果你想在配方中出现美元符号，必须将其双倍（‘$$’）。对于像默认shell这样的使用美元符号引入变量的shell，重要的是要清楚你想要引用的是一个构建变量（使用单个美元符号）还是一个shell变量（使用两个美元符号）。例如：

```
列表=一、二、三
所有
        对于i in$（LIST）；do\
            echo$$i;\
        已完成的
```

结果是将以下命令传递给shell：

```
对于i在one twothree；do\
    echo$i;\
已完成的
```

生成预期结果：

```
一
两个
三
```

5.2配方回声

通常在执行之前打印出每行配方。我们称此为回显，因为它看起来像是你自己在输入这些行。

当一行以‘@’开头时，该行的回显会被抑制。‘@’在行传递给shell之前会被丢弃。通常你会用这个方法来执行仅用于打印某物的命令，例如echo命令，以指示通过makefile：

```
@echo正在制作发行文件
```

当给出“标志”-n-或“仅打印”时，它只会回显大多数配方，而不会执行它们。参见第9.8节[选项总结]，第114页。在这种情况下，即使是以‘@’开头的配方行也会被打印出来。这个标志有助于了解配方认为哪些是必要的，而无需实际执行它们。

使用“-s”或“--silent”标志可以防止所有回声，就像所有的配方都以“@”开头一样。特殊目标无声的规则没有先决条件，具有相同的效果（参见第4.8节[特殊内置目标名称]，第34页）.

5.3配方执行

当需要执行更新目标的脚本时，它们通过为每个脚本行调用一个新的子shell来执行，除非生效了 . ONESHELL 特 殊 目 标 （ 参 见 第 5.3.1 节 [使 用 OneShell] ， 第 48 页)（在实践中，make可以使用不影响结果的快捷方式。）

请注意：这表明设置shell变量和调用如cd这样的shell命令来为每个进程设置局部上下文不会影响配方中的后续行。1如果您想使用cd影响下一条语句，请将两个语句放在一行中。然后make将调用一个shell来运行整个行，shell将按顺序执行这些语句。例如：

```
foo: bar/lose
        cd $（<D）&& gobble $（<F）> ../$@
```

在这里，我们使用了shell AND运算符（&&），这样如果cd命令失败，脚本将不会尝试调用错误目录中的gobble命令，从而避免了可能出现的问题（在这种情况下，至少会将../foo截断）。

5.3.1使用One Shell

有时你希望将配方中的所有行传递给一次shell调用。通常有两种情况这很有用：首先，它可以在包含许多命令行的make文件中提高性能，通过避免额外进程。其次，你可能希望在配方命令中包含换行符（例如，你可能使用的是与shell非常不同的解释器）。如果单shell特殊目标出现在make文件的任何位置，则每个目标的所有配方行都将被提供给一次shell调用。配方行之间的换行符将保留。例如：

```
.ONESHELL:
foo: bar/lose
        cd $(<D)
        gobble $（<F）> ../$@
```

现在可以按预期工作，即使命令位于不同的配方行上。

如果提供了.ONESHELL，则只会检查配方的第一行是否包含特殊前缀字符（‘@’、‘-’和‘+’）。当调用SHELL时，后续行中会包含配方行中的特殊字符。如果你想让你的配方以这些特殊字符之一开头，你需要安排它们不是第一行的第一个字符，可能需要添加注释或类似的内容。例如，在Perl中这将是一个语法错误，因为第一个‘@’被make移除了：

```
.ONESHELL:
SHELL=/usr/bin/perl
```

1在MS-DOS上，当前工作目录的值是全局值，因此更改它将影响这些系统上的以下配方行。

```
.SHELLFLAGS=-e
表明
        @f = qw（a b c）；
        print"@f\n";
```

然而，这两种选择都可以正常工作：

```
.ONESHELL:
SHELL=/usr/bin/perl
.SHELLFLAGS=-e
表明
        #确保“@”不是第一行的第一个字符@f=qw（a b c）；
        打印“@f”或
```

“\n”；或者

```
.ONESHELL:
SHELL=/usr/bin/perl
.SHELLFLAGS=-e
表明
        我的@f=qw（因为）；
        print"@f\n";
```

作为一项特殊功能，如果壳被确定为POSIX风格的壳，则在处理配方之前，“内部”注释行中的特殊前缀字符将被移除。此功能旨在允许现有的Makefile添加.O oneshell特殊目标，而无需进行大量修改即可正常运行。由于这些特殊前缀字符在POSIX SHELL脚本行首不合法，因此不会影响功能。例如，这可以按预期工作：

```
.ONESHELL:
foo: bar/lose
        @cd$(@D)
        @gobble $(@F)> ../$@
```

即使有这个特殊功能，使用.ONESHELL的构建文件也会表现出一些可能引起注意的不同行为。例如，通常情况下，如果配方中的任何一行失败，这会导致规则失败，不会再处理其他配方行。而在.ONESHELL下，除了最后一行配方外的任何一行失败都不会被make注意到。你可以修改.SHELLFLAGS以添加-e选项到shell中，这会使配方行中的任何失败都会导致shell失败，但这可能会使你的配方行为有所不同。最终，你可能需要加固你的配方行，使其能够与.ONESHELL兼容。

5.3.2选择外壳

用作外壳的程序取自变量shell。如果在你的makefile中未设置此变量，则程序/bin/sh将用作外壳。传递给shell的参数(s)取自变量.SHELLFLAGS。通常情况下，.SHELLFLAGS的默认值为-c，或者在POSIX兼容模式下为-ec。

与大多数变量不同，变量SHELL从环境设置。这是因为SHELL环境变量用于指定您的个人SHELL选择

交互式使用的程序。如果像这样个人选择影响到生成文件的功能，那将是非常糟糕的。请参见第6.10节[环境变量]，第77页 .

此外，当你在生成文件中设置SHELL时，该值不会导出到调用该生成文件的环境行中。相反，会导出从用户环境继承的值，如果有值的话。你可以通过显式导出SHELL来覆盖此行为（参见第5.7.2节[向子生成文件传递变量]，第57页），迫使它在环境中的配方行中通过。

然而，在MS-DOS和MS-Windows上使用环境中的SHELL值，因为这些系统上的大多数用户不会设置这个变量，因此它很可能被专门设置为由make使用。在MS-DOS上，如果SHELL的设置不适合make，则可以将变量MAKESHELL设置为make应该使用的SHELL；如果设置了这个变量，它将被用作SHELL而不是SHELL的值。

在DOS和Windows中选择外壳

在MS-DOS和MS-Windows中选择一个外壳比在其他系统上要复杂得多。

在MS-DOS上，如果未设置SHELL，则将使用变量COMSPEC的值（该值始终被设置）。

在MS-DOS上，设置变量S HELL的Makefile行的处理方式与在MS-DOS上不同。stock shell、command. com在功能上极其有限，许多使用make的用户倾向于安装一个替代shell。因此，在MS-DOS上，make检查shell的值，并根据它指向的是Unix风格的shell还是DOS风格的shell而改变其行为。这使得即使shell指向command. com也能实现合理的功能。

如果SHELL指向一个类Unix的SHELL，则在MS- DOS中，make还会检查是否确实可以找到该SHELL；如果没有找到，则忽略设置SHELL的那行。在MS- DOS中，GNU make在以下位置搜索SHELL：

1. 在由SHELL值所指向的精确位置。例如，如果生成文件指定“shell=/bin/sh”，则make将查找当前驱动器上的目录/bin。
2. 当前目录。
3. 按照顺序，在PATH变量的每个目录中。

在检查每个目录时，make首先会查找特定文件（例如上面的例子中的sh）。如果找不到该文件，它还会在该目录中查找具有已知扩展名的文件，这些扩展名标识可执行文件。例如. exe、. com、. bat、. btm、. sh等。

如果这些尝试中的任何一次成功，SHELL的值将被设置为找到的SHELL的完整路径名。然而，如果没有找到这些选项，SH ELL的值将不会改变，因此设置它的行将实际上被忽略。这样做是为了确保make只支持特定于Unix风格SHELL的功能，前提是该系统上确实安装了这样的SHELL。

请注意，对shell的扩展搜索仅限于从Makefile中设置shell的情况；如果在环境或命令行中设置了shell，则您应该将其设置为shell的完整路径名，就像在Unix上一样。

上述DOS特定处理的效果是，如果在PATH中的某个目录中安装了sh .exe，那么包含“shell=/bin/sh”（像许多Unix Makefile那样）的Makefile将在未修改的MS- DOS上工作。

5.4并行执行

GNU make知道如何同时执行多个任务。通常情况下，make一次只执行一个任务，在完成后再执行下一个。然而，通过使用‘-j’或‘--jobs’选项，make可以同时执行多个任务。你可以在make文件中抑制某些或所有目标的并行性（参见Section5.4.1 [D是禁用并行执行]，第51页).

在MS-DOS上，由于该系统不支持多处理，因此“-j”选项不起作用。

如果‘-j’选项后面跟着一个整数，这表示一次执行的作业数量；这被称为作业槽数。如果‘-j’选项后面没有看起来像整数的内容，则没有作业槽数的限制。默认的作业槽数为一，这意味着串行执行（一次只做一件事）。

处理递归的make调用会引发par allel执行的问题。有关此问题的更多信息，请参见第59页第5.7.3节[向子make发送通信选项] .

如果一个配方失败（被信号杀死或以非零状态退出），并且没有忽略该配方的错误（参见第54页第5.5节[配方中的错误] ），其余的重新生成相同目标的配方行将不会被执行。如果配方失败且未选择“-k”或“--keep-going”选项（见第9.8节[选项概要]，第114页），则会中止执行。如果由于任何原因（包括信号）导致make终止而子进程仍在运行，它会在子进程完成前等待它们结束，然后才会真正退出。

当系统负载较重时，你可能希望运行的作业数量少于负载较轻时的数量。你可以使用‘-l’选项告诉make根据负载平均值限制同时运行的作业数量。‘-l’或‘--max-load’选项后面跟着一个浮点数。例如，

```
-l 2 .5
```

如果负载平均值高于2.5，则不会允许启动超过一个作业。如果在先前的“-l”选项后面没有数字，则将删除负载限制。

更确切地说，当make开始运行一个作业时，如果它已经至少有一个作业在运行，它会检查当前的负载平均值；如果它低于用‘-l’给出的限制，make会等待直到负载平均值低于该限制，或者直到所有其他作业完成。

默认情况下，没有负载限制。

5.4.1禁用并行执行

如果一个生成文件完全准确地定义了其所有目标之间的依赖关系，则无论是否启用并行执行，生成器都将正确地构建目标。这是编写生成文件的理想方式。

但是，有时在makefile中的一些或全部目标无法并行执行，因此无法添加必要的先决条件来通知make。在这种情况下，makefile可以使用各种方法禁用并行执行。

如果在任何地方指定了没有先决条件的. NOTPARALLEL特殊目标，则无论并行设置如何，整个make实例都将串行运行。例如：

```
全部：一、二、三
一、二、三：@睡眠1；echo$@

.NOTPARALLEL:
```

不管调用的是哪个方法，目标一、二和三将按顺序运行。

如果. NOTPARALLEL特定目标有前置条件站点，则每个前置条件都将被视为一个目标，所有这些目标的前置条件将依次运行。请注意，只有在构建此目标时才会将前置条件串行运行：如果其他目标列出了相同的前置条件且不在. NOTPARALLEL中，则这些前置条件可以并行运行。例如：

```
全部：基底不平行

基数：一二三
不平行：一二三

一、二、三：；@睡眠1；echo$@

. NOTPARALLEL: notparallel
```

这里，make-jbase将并行运行目标一、二和三，而make-j不并行将串行运行它们。如果运行make-jall，则由于base列出了这些目标作为先决条件，并且没有序列化，因此它们将并行运行。

不并行tar get不应有命令。

最后，可以使用. WAIT特定目标以细粒度控制特定先决条件的序列化。当此目标出现在先决条件列表中且启用了并行执行时，make将不会构建位于. WAIT左侧的所有先决条件，直到. WAIT左侧的所有先决条件完成。例如：

```
全部：一、二、等三
一、二、三：；@睡眠1；echo$@
```

如果启用了并行执行，则make将尝试并行构建一和二，但不会尝试构建三，直到两个都完成。

与提供给.NOTPARALLEL的目标一样，.WAIT只在构建其先决条件列表的目标时生效。如果相同的先决条件出现在其他目标中，但没有.WAIT，则这些目标仍可以并行运行。因此，无论是.NOTPARALLEL还是.WAIT，在控制并行执行方面都不如定义先决条件关系可靠。然而，它们易于使用，在不太复杂的情况下可能已经足够。

规则的所有自动变量中都不会出现. WAIT先决条件。

您可以创建一个实际的目标。在生成文件中等待可移植性，但使用此功能时不需要这样做。如果创建了. WAIT目标，则它不应有先决条件或命令。

. WAIT特性也实现于其他版本的make中，并且在POSIX标准中对make进行了规定。

5.4.2并行执行期间的输出

当并行运行多个配方时，每个配方的输出在生成后立即显示，结果是来自不同配方的消息可能交错出现，有时甚至出现在同一行。这会使读取输出变得非常困难。

为了避免这种情况，可以使用‘--output-sync’（‘-O’）选项。此选项指示make保存其调用命令的输出，并在命令完成后一次性打印所有输出。此外，如果多个递归make调用并行运行，它们会进行通信，以确保每次只有一个生成输出。

如果启用工作目录打印（请参见Section5.7.4[The‘--print-directory’ 选项]，第61页 ），输入/离开消息会打印在每个输出分组周围。如果您不想看到这些消息，请添加“--no-print-directory”选项来使标志生效。

同步输出时有四个粒度级别，通过为选项指定参数（例如，“-Oline”或“--output-sync=recurse”）来指定。

否这是默认设置：所有输出直接发送，不进行同步-
然后执行标准化。

将配方中每一条单独的行输出分组并尽快打印出来
因为那行已经完成了。如果一个食谱由多行组成，它们可能与其他食谱的行交错在一起。

针对每个目标，将整个配方的输出进行分组并打印一次
目标已完成。如果在没有参数的情况下使用--output-sync或-O选项，则这是默认值。

每次递归调用make时，递归输出被分组并打印一次
递归调用已完成。

无论选择哪种模式，总构建时间都是相同的。唯一的区别在于输出的显示方式。

“目标”和“递归”模式都收集目标整个配方的输出，并在配方完成后不间断地显示它。它们之间的区别在于包含递归调用的make的配方是如何处理的（见第5.7节[递归使用make]，第56页 ）。对于所有没有递归行的脚本，‘target’和‘recurse’模式表现相同。

如果选择了“递归”模式，包含递归构建调用的配方将与其他目标相同处理：整个配方完成后，包括递归构建的输出在内的所有目标的输出都会被保存并打印出来。这确保了由给定递归构建实例生成的所有目标的输出可以组合在一起，从而可能使输出更容易理解。然而，这也导致在构建过程中会出现长时间看不到任何输出的情况，随后是一段大段的输出。如果你不是在构建过程中实时监控，而是事后查看构建日志，这可能是最适合你的选择。

如果你正在观看输出，构建过程中长时间的静默可能会令人沮丧。‘目标’输出同步模式检测到当递归调用时，使用标准方法，不会同步这些行的输出。递归构建会为其目标执行同步操作，每个目标完成后，其输出将立即显示。请注意，输出来自

配方的递归行未同步（例如，如果递归行在运行make前打印一条消息，则该消息不会同步）。

对于前端来说，如果要跟踪配方开始和完成时的输出，那么“线程”模式可以非常有用。

某些由make调用的程序，如果它们确定自己是在向终端而不是文件写输出（通常描述为“交互式”与“非交互式”模式），其行为可能会有所不同。例如，许多能够显示彩色输出的程序，在确定它们不是向终端写输出时，就不会这样做。如果你的make文件调用了这样的程序，那么使用输出同步选项会使该程序认为自己正在以“非交互式”模式运行，尽管最终输出会到达终端。

5.4.3并行执行期间的输入

两个进程不能同时从同一个设备接收输入。为了确保只有一个程序试图同时从终端接收输入，可以禁用除一个正在运行的程序外的所有标准输入流。如果另一个程序尝试从标准输入读取数据，通常会引发致命错误（即“管道断开”信号）。

无法预测哪个配方会有一个有效的标准输入流（这将来自终端，或者你重定向make的标准输入的任何地方）。第一个配方总是会得到它，然后在第一个配方结束后启动的第一个配方将得到它，依此类推。

如果找到更好的替代方案，我们将改变这一方面的制作方式。同时，如果您使用并行执行功能，不应依赖任何使用标准输入的配方；但如果您不使用此功能，则所有配方中标准输入均正常工作。

5.5配方错误

每次调用shell返回后，make都会查看其退出状态。如果shell成功完成（退出状态为零），则在新shell中执行配方中的下一行；最后一行结束后，规则结束。

如果出现错误（退出状态非零），make将放弃当前规则，也许还会放弃所有规则。

有时，某个特定的程序行失败并不表示有问题。例如，您可能使用mkdir命令来确保目录存在。如果目录已经存在，mkdir将报告错误，但您可能希望不管怎样都继续执行make。

若要忽略配方行中的错误，请在行文本的开头（初始制表符之后）写一个“-”。该“-”在行传递给shell执行之前被丢弃。

例如，

```
干净的
        -rm-f* .o
```

即使rm无法删除文件，也会继续执行此操作。

在运行make时使用“-i”或“--忽略错误”标志，会忽略所有规则的所有配方中的错误。如果没有任何先决条件，那么在特殊目标. IGNORE的makefile中的规则也会产生同样的效果。这不太灵活，但有时很有用。

如果由于a‘-’或‘-i’标志而要忽略错误，则将treats作为错误返回，就像成功一样，只是它会打印出一条消息，告诉您shell退出时的状态代码，并说该错误已被忽略。

如果发生错误，而没有告诉它忽略，则意味着当前目标无法正确地重新制作，任何直接或间接依赖于它的目标也无法正确地重新制作。由于这些目标的先决条件未得到满足，因此不会为它们执行进一步的配方。

在这种情况下，Normal make立即放弃，返回非零状态。然而，如果指定了“-k”或“--keep-going”标志，则在放弃并返回非零状态之前，make会继续考虑待处理目标的其他先决条件，并在必要时重新生成它们。例如，在编译一个对象文件时出错后，“make-k”将继续编译其他对象文件，即使它已经知道链接这些文件是不可能的。参见第9.8[选项概要]节，第114页。

通常的行为假设你的目的是将指定的目标更新到最新状态；一旦发现这不可能实现，它可能会立即报告失败。‘-k’选项表示真正的目的是尽可能多地测试程序中所做的更改，也许是为了找出几个独立的问题，以便在下一次编译尝试前全部修正。这就是为什么Emacs的编译命令默认会传递‘-k’标志的原因。

通常当一个配方行失败时，如果它已经更改了目标文件，该文件就会被损坏，无法使用——或者至少没有完全更新。然而，文件的时间戳显示它现在是最新的，因此下次运行make时，它不会尝试更新该文件。这种情况与shell被信号杀死时的情况相同；参见第5.6节[中断]，第55页 通常情况下，如果开始修改文件后配方失败，正确的做法是删除目标文件。如果.DELETE_ON_ERROR出现在目标列表中，make会执行这一操作。这几乎总是你希望make执行的操作，但并非历史惯例；因此为了兼容性，你必须明确请求它。

5.6中断或终止

如果在执行时，make接收到致命信号，它可能会删除配方本应更新的目标文件。如果自make首次检查目标文件以来，目标文件的最后修改时间已更改，则会执行此操作。

删除目标文件的目的是确保在下次运行时重新生成。为什么这样做呢？假设你在编译器运行期间输入Ctrl- c，它已经开始写入对象文件foo .o。Ctrl- c会终止编译器，导致生成一个不完整的文件，其最后修改时间比源文件foo .c更晚。但make也会接收到Ctrl- c信号并删除这个不完整的文件。如果make没有这样做，下次调用make时会认为foo .o不需要更新——当链接器尝试链接缺少一半内容的对象文件时，就会产生奇怪的错误消息。

你可以通过让特殊的tar get. PRECIOUS依赖于它来防止目标文件被删除。在重新创建目标之前，请检查它是否出现在. PRECIOUS的必需项中，从而决定如果发生信号时目标是否应该被删除。你可能这样做的一些原因是目标已更新

以某种原子的方式存在，或者只存在于记录修改时间（内容无关紧要），或者必须始终存在以防止其他类型的麻烦。

尽管make尽最大努力清理，但在某些情况下仍无法完成清理。例如，make可能被一个无法捕捉的信号杀死。或者，make调用的一个程序可能被杀死或崩溃，留下一个更新但损坏的目标文件：make不会意识到这一失败需要清理目标文件。又或者，make本身可能会遇到错误并崩溃。

出于这些原因，最好编写防御性配方，即使失败也不会留下损坏的目标。最常见的做法是创建临时文件而不是直接更新目标，然后将临时文件重命名为最终的目标名称。有些编译器已经这样做了，因此你无需编写防御性配方。

5.7递归使用make

递归使用`make`意味着在`make`文件中将`make`作为命令使用。当您需要为组成较大系统的各个子系统编写独立的`make`文件时，这种技术非常有用。例如，假设您有一个子目录`subdir`，其中包含自己的`make`文件，并且希望该目录下的`make`文件能够对子目录运行`make`。您可以通过以下方式实现：

```
子系统
        cd subdir&&$(MAKE)
```

或者，等效地，本节（见第9.8[选项总结]节，第114页）：

```
子系统
        $（制造）-C subdir
```

你可以通过复制这个示例来编写递归的make命令，但关于它们的工作原理及其原因，以及子级make与顶级make的关系，还有很多需要了解的内容。你可能还会发现，将调用递归make命令的目标声明为“phony”（关于何时使用这一点的更多讨论，请参见第4.5节[Phony目标]，第31页).

为了方便起见，当GNU make启动时（在处理完any-C选项后），它会将变量CURDIR设置为当前工作目录的路径名。这个值永远不会被makeagain修改：特别是请注意，如果你从其他目录包含文件，CURDIR的值不会改变。该值具有与在make文件中设置时相同的优先级（默认情况下，环境变量CURDIR不会覆盖此值）。请注意，设置此变量不会影响make的操作（例如，它不会导致make更改其工作目录）。

5.7.1MAKE变量的工作原理

递归的make命令应始终使用变量make，而不是显式的命令名称“make”，如下所示：

```
子系统
        cd subdir&&$(MAKE)
```

此变量的值是调用make时所使用的文件名。如果此文件名是bin/ make，则执行的配方为“cd subdir&&/bin/make”。如果使用特殊

用于运行顶级生成文件的版本，递归调用时将执行相同的特殊版本。

作为一种特殊功能，使用规则配方中的变量MAKE可以改变‘-t’（--触摸）、‘-n’（--仅打印）或‘-q’（--询问）选项的效果。使用MAKE变量的效果与在配方行开头使用‘+’字符相同。参见第9.3节[不执行配方]，第111页。此特殊功能仅当MAKE变量直接出现在配方中时才启用；如果通过其他变量的扩展引用MAKE变量，则不适用。在这种情况下，必须使用‘+’标记才能获得这些特殊效果。

考虑上述示例中的命令‘make-t’。（‘-t’选项标记目标为最新，但不实际运行任何配方；参见第9.3节[执行替代方案]，第111页。）根据‘t’的常规定义，示例中的‘make-t’命令只会创建一个名为subsystem的文件，除此之外不会做其他事情。你真正希望它做的其实是运行‘cd subdir&& make-t’；但这需要执行配方，而‘t’指示不要执行配方。

特殊功能让这变得随心所欲：每当规则行中包含变量MAKE时，标志-t、-n和-q不会应用于该行。即使存在导致大多数规则不执行的标志，含有MAKE的规则行仍会正常执行。通常的MAKE标志机制会将这些标志传递给子-make（参见Section5.7.3[向子-make传达选项]，第59页），因此您对文件的“触摸”请求或对“打印”请求将传播到子系统。

5.7.2向子系统发送通信变量

顶级生成器的可变值可以通过环境通过显式请求传递给子生成器。这些变量在子生成器中定义为默认值，但除非使用‘-e’开关（见第9.8节[选项总结]，第114页），否则它们不会覆盖子生成器所使用的生成文件中定义的变量。

要传递或导出一个变量，make将该变量及其值添加到环境，以便在运行每行配方时使用。sub- make然后使用该环境来初始化其变量值表。请参见第6.10节[环境中的变量]，第77页 .

除非有明确要求，否则只有在环境初始中定义了出口，或者在命令行上设置的出口且其名称仅由字母、数字和下划线组成时，才将出口设为可变。

make变量SHELL的值不会导出。相反，调用环境变量中的S HELL变量的值会被传递给sub-make。你可以通过下面描述的export指令强制make导出其SHELL值。请参见第5.3.2节【选择外壳】，第49页 .

特殊变量MAKEFLAGS总是被导出（除非您将其取消导出）。如果将MAKEFILES设置为任何值，则会导出它。

通过将它们放入MAKEFLAGS变量中，使自动传递在命令行上定义的变量值。请参见下一节。

如果变量是通过默认方式由make创建的，则通常不会传递这些变量（请参见第10.3节[隐式规则使用的变量]，第125页）。子make将为这些变量定义它们。

如果要将特定变量导出到子-make，请使用导出指令，例如：export variable ...

如果要防止变量被导出，请使用unex port指令，例如：

取消导出变量...

在这两种形式中，扩展了导出和不导出的参数，因此也可以扩展为变量或函数，这些变量或函数扩展为要（不）导出的（变量名列表）。

为了方便起见，您可以同时定义一个变量并输出它，方法如下：

输出变量=值的结果与以下

相同：

变量=值
导出变量和

输出变量：=值的结果与下

列结果相同：

变量：=值
出口变量同样，

输出变量+=值如下所示：

变量+=值
导出变量

参见第6.6节[向变量添加更多文本]，第74页 .

您可能会注意到，导出和取消导出指令在shell中的运行方式与在shell中的运行方式相同。

如果要将所有变量都作为默认值导出，则可以单独使用export：

出口

这说明未在导出或取消导出指令中明确提到的变量应该被导出。在取消导出指令中给出的任何变量仍然不会被导出。

由导出指令本身引发的行为在较旧版本的GNU make中是默认设置。如果你的make文件依赖于这种行为，并且希望与旧版本的make兼容，可以将特殊目标.EXPORT_ALL_VARIABLES添加到你的make文件中，而不是使用导出指令。这将被旧版本的make忽略，而导出指令则会导致语法错误。

在使用export本身或.EXPORT_ALL_V ARIABLES导出变量时，默认情况下，只有名称完全由字母数字和下划线组成的变量才会被导出。要导出其他变量，必须在导出指令中明确提及它们。

向环境添加变量值需要将其展开。如果展开变量有副作用（如信息、评估或类似函数），那么每次调用命令时都会看到这些副作用。你可以通过确保这些变量的名称默认不可导出来避免这种情况。然而，更好的解决方案是

不使用这个“默认导出”功能，而是通过名称显式导出相关变量。

你可以单独使用`unexport`来告诉`make`不要默认导出变量。由于这是默认行为，只有当`export`之前（可能是在包含的`make`文件中）被单独使用时，你才需要这样做。你不能单独使用`export`和`unexport`来决定某些配方导出变量而其他配方不导出。最后一个单独出现的`export`或`unexport`指令决定了整个`make`运行的行为。

作为特殊功能，变量MAKELEVEL在逐级传递时会发生变化。该变量的值是一个字符串，表示当前层级的深度，以十进制数形式呈现。顶级make的值为‘0’；子make为‘1’；子子make为‘2’，依此类推。当make设置配方环境时，值会递增。

MAKELEVEL的主要用途是在条件指令中测试它（见第7章[Con- Makefile的其他部分，第85页 ）这样，如果递归运行，生成文件的行为方式与直接由您运行时的行为方式不同。

你可以使用变量`MAKEFILES`使所有子-make命令使用额外的.make文件。`MAKEFILES`的值是一个用空格分隔的文件名列表。如果在外部.make文件中定义了此变量，则该变量会通过环境传递下去；然后它作为子-make在读取常规或指定的.make文件之前需要读取的额外.make文件列表。请参见第3.4节[变量`MAKEFILES`]，第14页。.

5.7.3向子制造商传达信息

诸如“-s”和“-k”之类的标志自动传递给子程序make，通过变量MAKEFLAGS。此变量由make自动设置，以包含make接收到的标志字母。因此，如果y你执行“make-ks”，那么MAKEFLAGS将获得值“ks”。

因此，每个子类都会获得其环境中的MAKEFLAGS的值。作为响应，它从该值中获取这些标志，并像它们是作为参数给出的一样处理这些标志。请参见第114页的第9.8节[选项总结]。

MAKEFLAGS的值可能是一个空字符串，表示单字母选项，这些选项不带参数，后面跟着一个空格和任何带参数或具有长选项名称的选项。如果一个选项同时有单字母和长选项，则始终优先选择单字母选项。如果命令行中没有单字母选项，则MAKEFLAGS的值以空格开始。

同样地，通过MAKEFLAGS将命令行定义的变量传递给sub-make。MAKEFLAGS值中包含‘=’的单词会使变量被视为变量定义，就像它们出现在命令行一样。请参见第9.5节[覆盖变量]，第113页。

选项“-C”、“-f”、“-o”和“-W”不放入MAKEFLAGS中；这些选项不会传递下去。

“-j”选项是一个特殊情况（请参见第51页第5.4[并行执行]节).如果你将其设置为某个数值‘N’，并且你的操作系统支持它（大多数UNIX系统都会支持；其他系统通常不支持），父进程和所有子进程将进行通信以确保它们之间同时运行的任务数不超过‘N’个。请注意，任何标记为递归的任务（参见第9.3节[代替执行配方]，第111页）

不会影响到全部的工作（否则我们就会让n个子女人跑来跑去，而没有空位去做任何真正的工作！）

如果您的操作系统不支持上述通信，则不会向MAKEFLAGS添加“-j”，从而使子-make以非并行模式运行。如果将“-j”选项传递给子-make，您将获得比预期更多的并行作业。如果您不提供数字参数而仅使用“-j”，即希望尽可能多地并行运行作业，则此选项会被传递下去，因为多个无穷大值不超过一个。

如果您不想传递其他标志，您必须更改MAKEFLAGS的值，例如：

```
子系统
        cd subdir&&$(MAKE) MAKEFLAGS=
```

命令行变量定义确实出现在变量MAKEOVERRIDES中，而MAKEFLAGS包含对这个变量的引用。如果您确实要正常传递标志，但不想传递命令行变量定义，则可以将MAKEOVERRIDES重置为空，例如：

```
制作替代品=
```

这通常没有用处。然而，某些系统对环境大小有固定的限制，将大量信息放入MAKEFLAGS中可能会超出这个限制。如果你看到错误消息“参数列表过长”，这可能是问题所在。（为了严格遵守POSIX.2规范，如果makefile中出现了“special target”字样，则更改MAKEOVERRIDES不会影响MAKEFLAGS。你可能对此并不关心。）

为了历史兼容性，也存在一个类似的变量`MFLAGS`。它与`MAKEFLAGS`的值相同，但不包含命令行变量定义，并且除非为空，否则总是以连字符开头（`MAKEFLAGS`仅在以没有单字母版本的选项开头时才以连字符开头，例如`--warn-undefined-variables`）。传统上，在递归的`make`命令中显式使用了`MFLAGS`，如下所示：

```
子系统
        cd subdir &&$(MAKE)$(MFLAGS)
```

但是现在MAKEFLAGS使这种用法变得多余。如果您希望您的makefile与旧的make程序兼容，请使用此技术；它也适用于更现代的make版本。

`MAKEFLAGS`变量在你希望每次运行`make`时都设置某些选项时也非常有用，例如`‘-k`（见第9.8节[选项概要]，第114页）。你只需在环境中为`MAKEFLAGS`设置一个值。你也可以在`make`文件中设置`MAKEFLAGS`，以指定该文件生效时应使用的额外标志。（请注意，你不能用MFLAGS这样使用它。这个变量仅为了兼容性而设置；`make`不会以任何方式解释你为其设置的值。）

当`make`解释`MAKEFLAGS`的值（无论是从环境变量还是从`make`文件中获取）时，首先如果该值不以连字符开头，则会将其前缀加上连字符。然后，它将该值按空格分割成单词，并像处理命令行选项一样解析这些单词（但忽略`--`、`--`、`--`、`--`和`--`的长名称版本；并且对于无效选项没有错误）。

如果你确实要在环境中设置MAKEFLAGS，务必确保不包含任何会严重影响make操作并削弱make文件及make本身目的的选项。例如，如果在这些变量中设置‘-t’、‘-n’和‘-q’选项，可能会产生灾难性的后果，至少会带来令人惊讶甚至可能令人烦恼的影响。

如果您希望运行除GNU make之外的其他make实现，因此不想将GNU make特定的标志添加到MAKEFLAGS变量中，可以将它们添加到GNUMAKEFLAGS变量中。此变量在MAKEFLAGS之前解析，与MAKEFLAGS的方式相同。当make构造函数将MAKEFLAGS传递给递归make时，它会包含所有标志，即使是从GNUMAKEFLAGS中提取的标志也是如此。因此，在解析GNUMAKEFLAGS后，GNU make将此变量设置为空字符串，以避免在递归过程中重复标志。

最好只在不会实质性改变你的生成文件行为的情况下使用GNUMAKEFLAGS。如果生成文件无论如何都需要GNU make，则只需使用MAKEFLAGS。像‘--no-print-directory’或‘--output-sync’这样的标志可能适合用作GNUMAKEFLAGS。

5.7.4“打印直接或”选项

如果你使用多个递归调用级别，‘-w’或‘--print-directory’选项可以使输出更易于理解，通过显示每个目录在make开始处理和结束处理时的状态。例如，如果在目录/gnu/make下运行‘make-w’，make将打印一行类似以下的内容：

make：进入目录“/u/gnu/make”。

在做任何事情之前，以及一个形式的行：

make：离开目录‘/u/gnu/make’。

处理完成后。

通常情况下，您不需要指定此选项，因为“make”会为您完成这项工作：当使用“-C”选项时，“-w”会自动打开；在子进程中，如果使用“-s”，表示要保持静默，或者使用“--no-print-directory”来明确禁用它，那么“make”也不会自动打开“-w”。

5.8定义罐头配方

当同一序列的命令i在制作各种目标时很有用时，可以使用定义指令将其定义为一个库序列，并从这些目标的配方中引用库序列。库序列实际上是一个变量，因此其名称不能与其他变量名冲突。

下面是一个定义罐头食谱的例子：

```
definerun-yacc=
yacc$(firstword$^)
mv y . tab.c $@
endef
```

这里run- yacc是要定义的变量名；endef标记定义的结束；中间的行是命令。define指令不会在预设的序列中扩展变量引用和函数调用；‘$’字符，括号，

变量名等，都成为所定义变量值的一部分。请参见第76页第6.8[定义多行变量]节，以完整地解释定义。

此示例中的第一个命令在使用库序列的规则的第一个先决条件上运行Yacc。Yacc的输出文件总是名为y.tab .c。第二个命令将输出移动到规则的目标文件名。

要使用罐装序列，将变量代入规则的配方中。您可以像代入其他变量一样代入它（参见第6.1节[变量引用基础]，第65页 ）。因为由define定义的变量是递归展开的变量，所以你在define内部写的所有变量引用现在都会重新展开。例如：

```
foo .c: foo .y
        $(run-yacc)
```

当变量“$^”出现在run-yacc的值中时，将用“foo.y”替换它，而用“foo.c”替换“$@”。

这是一个现实的示例，但这个特定示例在实践中并不需要，因为make拥有一个隐式规则，可以根据所涉及的文件名来确定这些命令（请参阅第10章[使用隐式规则]，第121页）。

在配方执行中，每个罐装序列的每一行都像单独出现在规则中的那样被处理，前面有一个制表符。特别是，`ma ke`会为每行调用一个分离子壳。您可以在每个罐装序列的每一行上使用特殊前缀字符`th`来影响命令行（`@`、`-`和`+`）。参见第5章[在规则中编写配方]，第45页。例如，使用此库的序列：

```
定义frobnicate=
@echo“frobnicating target$@“
frob- step-1$<-o$@- step-1
frob- step-2$@- step-1-o$@
endef
```

make不会回显第一行，而是回显以下两行配方。

另一方面，配方行中的前缀字符指的是罐装序列，适用于序列中的每一行。因此，规则为：

```
frob.out：frob .in
        @$(frobnicate)
```

不回显任何配方行。（请参见第47页的第5.2节[配方回显]，以全面解释“@”。)

5.9使用空配方

有时定义一个什么都不做的配方是有用的，这可以通过给出一个仅由空白组成的配方来实现。例如：

```
目标
```

为目标定义一个空的配方。您也可以使用以配方前缀字符开头的一行来定义一个空的配方，但是这样做会很混乱，因为这样的行看起来是空的。

你可能会疑惑，为什么需要定义一个不依赖于隐式规则的配方。这样做有用的一个原因是防止目标对象获得隐式配方（来自隐式规则或.DEFAULT特殊目标；参见第10章[隐式规则]，第121页，以及第10.6节[定义最后的默认规则]，第135页）。

空食谱也可以用于避免目标错误，这些目标将作为另一个食谱的附带效果而创建：如果目标不存在，空食谱可以确保制作不会抱怨不知道如何构建目标，制作将假设目标过时。

你可能会倾向于为非实际文件的目标定义空的脚本，这些目标仅存在以便其先决条件可以被重新创建。然而，这并不是最佳方法，因为如果目标文件确实存在，先决条件可能无法正确地被重新创建。参见第4.5节[虚假目标]，第31页。，为了更好的方法来做这件事。

6如何使用变量

变量是在生成文件中定义的一个名称，用来表示一个文本字符串，称为变量的值。这些值可以通过显式的请求在目标、先决条件、配方和其他生成文件部分中替换。（在某些其他版本的生成程序中，变量被称为宏。）

变量和函数在读取时会被展开，但不包括配方、使用‘=’定义变量的右侧以及使用define直接指令定义变量的主体。变量展开后的值是其最近一次定义时的值。换句话说，变量是动态作用域的。

变量可以表示文件名列表、传递给编译器的选项、要运行的程序、要查找源文件的目录、要写入输出的直接目录，或者任何你能想到的东西。

变量名可以是任何不包含‘：’，‘#’、‘=’或空格的字符序列。然而，包含字母、数字和下划线以外字符的变量名应谨慎考虑，因为在某些外壳中，这些变量名无法传递到子-make（参见第5.7.2节[向子-make传递变量]，第57页）。以“.”开头的可变名和大写字母在以后版本的make中可能具有特殊含义。

变量名区分大小写。名称“foo”、“FOO”和“Foo”都指代不同的变量。

在变量名中使用大写字母是传统的做法，但我们建议在生成文件中用于内部目的的变量名中使用小写字母，并将大写字母保留用于控制隐式规则的参数或用户应通过命令选项覆盖的参数（参见第9.5节[覆盖变量]，第113页）。

一些变量的名称由单个点字符或几个字符组成。这些是自动变量，它们有特定的专用用途。请参见第130页的第10.5.3[AutomaticVariables]节。

6.1可变参考的基础

要替换变量的值，请在括号或方括号中写上美元符号后跟变量名：‘$（foo）’或‘${foo}’都是有效引用变量foo的方式。‘$’的特殊意义在于，你必须使用‘$$’才能在文件名或配方中产生单个美元符号的效果。

变量引用可以在任何上下文中使用：目标、先决条件、配方、大多数指令和新变量值。下面是一个常见案例的示例，其中变量保存程序中所有对象文件的名称：

```
对象=程序.o foo .o utils .o
program:$(objects)
        cc-o program$(objects)

$(objects): defs .h
```

变量引用通过严格的文本替换工作。因此，规则

```
foo = c
prog.o:prog.$(foo)
        $(foo)$(foo)-$(foo)prog.$(foo)
```

可以用来编译一个C程序prog .c。由于在变量赋值之前空格被忽略，foo的值正是“c”（不要这样编写你的make文件！）

美元符号后面跟着一个不是美元符号、方括号或大括号的字符，该字符被视为变量名。因此，你可以用‘$x’引用变量x。然而，这种做法可能会导致混淆（例如，‘$foo’指的是变量f后跟字符串oo），所以我们建议在所有变量周围使用括号或大括号，即使是单字母变量也是如此，除非省略它们能显著提高可读性。一个经常改善可读性的地方是自动变量（见第10.5.3[自动变量]节，第130页）。

6.2变量的两种类型

GNU make中的变量可以通过不同的方式获得值，我们称这些方式为变量的口味。这些口味的区别在于它们如何处理在makefile中分配给它们的值，以及当变量后来被使用和展开时，这些值是如何被管理的。

6.2.1递归扩展变量赋值

变量i的第一个变体是一个递归扩展的变量。这类变量通过使用‘=’定义（见第6.5节[设置变量]，第72页）或通过定义指令（参见第76页第6.8节[定义多行变量]）。您指定的值将逐字安装；如果它包含对其他变量的引用，则在替换此变量时（在扩展其他字符串的过程中），这些引用将被展开。当这种情况发生时，称为递归展开。

例如，

```
foo=$(bar)
bar= $( ugh )
ugh= Huh?

全部：；echo$（foo）
```

将回声“嗯？”：“$（foo）”扩展为“$（bar）”，然后扩展为“$（ugh）”，最后扩展为“嗯？”。

这种变量类型是大多数其他版本的make支持的唯一类型。它有优点也有缺点。一个优点（大多数人会说）是：

```
CFLAGS= $（include_ dirs）-O
include_dirs=-伊福-伊巴尔
```

将执行预期的操作：当“CFLAGS”在配方中展开时，它将展开为“-Ifoo-Ibar-O”。一个主要缺点是，你不能在变量的末尾附加一些东西，如

```
CFLAGS =$(CFLAGS)-O
```

因为这会导致变量展开时出现无限循环。（实际上，make会检测到无限循环并报告错误。）

另一个缺点是，定义中引用的任何函数（见第8章[文本转换函数]，第91页）每次变量展开时都会执行。这会使运行速度变慢；更糟糕的是，它会导致通配符和shell函数产生不可预测的结果，因为你无法轻易控制它们何时被调用，甚至调用多少次。

6.2.2简单扩展变量赋值

为了避免递归扩展变量带来的问题和不便，还有一种方法：简单扩展变量。

简单扩展变量通过使用：=或：：=的行来定义（请参见第6.5节[设置　　变量]，第72页）。两种形式在GNU make中是等价的；但是只有“：：=”形式被POSIX标准描述（对“：：=”的支持被添加到POSIX标准和POSIX问题8中）。

变量的值在定义时仅扫描一次，扩展所有对其他变量和函数的引用。一旦扩展完成，该变量的值就不再被扩展：当使用该变量时，其值会被完全复制作为扩展。如果变量的值包含变量引用，那么扩展的结果将包含这些变量在定义时的值。因此，

```
x:= foo
y：=$(x) bar
x：= later
```

等于

```
y：= foo bar
x：=后
```

这里有一个稍微复杂一些的例子，展示了如何使用‘：=’结合shell函数。（参见第8.14节[shell函数]，第107页。）这个例子还展示了变量MAKELEVEL的使用，当它从I evel传递到I evel 时会发生变化。（参见Section5.7.2[向子-make传递变量]，第57页。，有关makelevel的信息。)

```
ifeq (0,${ MAKELEVEL})
whoami ：= $（shell whoami）
主机类型：= $（shell arch）
MAKE：= ${ MAKE}主机类型=${host-type}whoami=${whoami}
endif
```

使用“：=”的一个优点是，典型的“进入目录”配方看起来如下：

```
${subdirs}:
        ${MAKE} -C$@所有
```

简单扩展的变量通常使复杂的makefile编程更加可预测，因为它们像大多数编程语言中的变量一样工作。它们允许你使用变量自身的值（或通过某个扩展函数以某种方式处理的值）重新定义变量，并且可以更高效地使用扩展函数（参见第8章[文本转换函数]，第91页）。

您还可以使用它们将受控的引导空白引入到可变值中。在替换变量引用和函数调用之前，系统会从输入中丢弃前导空白字符；这意味着您可以通过使用变量引用保护变量值中的前导空格，例如：

```
nullstring :=
空格：=$（空字符串）#行尾
```

在这里，变量空间的值恰好是一个空格。注释“#行尾”仅为了清晰起见而包含在此处。由于变量值中不剥离尾随空格字符，因此行尾的一个空格也会产生相同的效果（但会更难阅读）。如果你在变量值的末尾放置空格，最好在该行的末尾加上类似的注释以明确你的意图。相反，如果你不想在变量值的末尾有任何空格字符，就必须记住不要在某些空格之后随意在行尾放置注释，例如：

```
dir ：=/foo/bar#用于存放frob文件的目录
```

此处变量dir的值为“/foo/bar”（后面跟着四个空格），这可能不是作者的本意。（想象一下使用这种定义的“$（dir）/file”！）

6.2.3立即扩展变量分配

另一种赋值方式允许即时扩展，但与简单赋值不同的是，生成的变量是递归的：每次使用都会再次展开。为了避免意外结果，在值立即展开后，它会自动被引号包围：展开后的值中所有$的实例都将转换为$$。这种赋值方式使用‘：：：=’运算符。例如，

```
var=最初
OUT:::=$( var )
变体=第二
```

结果是OUT变量包含文本“first”，而此处：

```
变量=一$$二
OUT:::=$( var )
var=three$$four
```

结果是OUT变量包含文本‘one$$two’。当变量被赋值时，该值会被展开，因此结果是变量‘one$two’的第一个值的展开；然后在赋值完成前重新转义该值，最终结果为‘one$$two’。

此后，将变数OUT视为递归变量，因此在使用时将重新展开。

这在功能上等同于：‘：=’/‘：：=’运算符，但有一些不同之处：

首先，在赋值后，变量是一个普通的递归变量；当你用‘+=’将其附加时，右侧的值不会立即展开。如果你希望‘+=’操作符能立即展开右侧的内容，应该改用‘：=’或‘：：=’赋值。

其次，这些变量的效率略低于直接扩展的变量，因为它们在使用时需要被扩展，而不仅仅是复制。然而，由于所有变量引用都被转义，这种扩展只是简单地转义了值，因此不会扩展任何变量或运行任何函数。

这里还有另一个例子：

```
变量= one$$ two
OUT:::=$( var )
OUT+=$（变量）
var=three$$four
```

在此之后，OUT的值是text‘one$$two$（var）。当使用此变量时，它将被展开并得到结果’one$twothree$four‘。

这种赋值方式等同于传统的BSD make‘：=’运算符；如您所见，它与GNU make‘：=’运算符的工作方式略有不同。：：=运算符是在Issue 8中添加到POSIX规范中的，以提供可移植性。

6.2.4条件变量赋值

还有另一个用于变量的赋值运算符，称为条件变量赋值运算符，因为它只有在变量尚未定义时才起作用。此语句：

```
FOO？=杆
```

与之完全等同（见第8.11节[起源函数]，第104页）：

```
ifeq ($（origin FOO），未定义）
  FOO=杆
endif
```

请注意，设置为空值的变量仍然被定义，因此“？=”不会设置该变量。

6.3变量参考的高级功能

本节介绍一些高级功能，您可以使用这些功能以更灵活的方式引用变量。

6.3.1替换参考文献

替换引用用你指定的更改来替代变量的值。它的形式为‘$（var：a=b）’（或‘${var：a=b}’），其含义是取变量var的值，将该值中每个单词末尾的a替换为b，并替换生成的字符串。

当我们说“在单词的末尾”时，我们指的是a必须出现在空白之后或值的末尾才能被替换；值中的其他a不会改变。例如：

```
foo：= a. o b. o l. a c .o
bar:=$(foo:.o=.c)
```

set‘bar’to‘a.c b .c l .a c.c’.见第6.5节[设置变量]，第72页 .

替换引用是patsubst扩展函数的简写（请参见第8.2节[字符串替换和分析函数]，第92页）：“$（var：a=b）”是

等同于“$（patsubst%a，%b，var）”。我们提供替换引用以及patsubst，以与make的其他实现兼容。

另一种子替换引用允许你使用patsubst函数的全部功能。它的形式与上述的‘$（var：a=b）’相同，只是现在a必须包含一个单个‘%’字符。这种情况下相当于‘$（patsubst a，b，$（var））’。有关patsubst函数的描述，请参见第92页第8.2节[字符串替换和分析函数]。例如：

```
foo：= a. o b. o l. a c .o
bar：=$（foo：%.o=%.c
```

）将‘bar’设定为a.c b. c l. a c.c‘

。

6.3.2计算变量名称

计算变量名是一个高级概念，在更复杂的文件编制程序中非常有用。在简单的情况下，您不必考虑它们，但它们可能非常有用。

变量可以在变量名中引用。这称为计算变量名或嵌套变量引用。例如，

```
x= y
y= z
a:=$($(x))
```

定义了a‘z’：‘$(x)’在‘$（$(x)）’中展开为‘y’，因此‘$（$(x)）’展开为‘$(y)’，再进一步展开为‘z’。这里引用变量的名称并未明确指出；它是通过‘$(x)’的展开计算得出的。这里的引用‘$(x)’存在于外部变量引用中。

前面的示例显示了两个嵌套级别，但可以有任意数量的级别。例如，这里有三个级别：

```
x= y
y= z
z= u
a:=$($($(x)))
```

这里最内层的$(x)展开为y，所以$（$(x)）展开为$(y)，再展开为z；现在我们有$(z)，它变成了u。

变量名中对递归扩展变量的引用按常规方式展开。例如：

```
x=$(y)
y= z
z=你好
a:=$($(x))
```

定义a为“Hello”：“$（$(x)）”变为“$（$(y)）”，然后变为“$(z)”，最后变为“Hello”。

嵌套变量引用也可以包含修改后的引用和函数调用（请参见第8章[用于转换文本的函数]，第91页），就像其他引用一样。

例如，使用子句函数（参见第8.2节[字符串替换和分析函数]，第92页）：

```
x= variable1
变量2：=你好
y=$( subst1,2,$(x))
z= y
a:=$($($(z)))
```

最终定义了a为“Hello”。很难想象有人会想要写出如此复杂的嵌套引用，但它确实有效：“$（$（$(z)））”展开为“$（$(y)）”，再进一步展开为“$（$（子式1,2，$(x)））”。这从x中获取了变量1的值，并通过替换将其变为变量2，从而使整个字符串变为“$（变量2）”，这是一个简单的变量引用，其值为“Hello”。

计算变量名不必完全由单个变量引用组成。它可以包含多个变量引用以及一些不变文本。例如，

```
a_dirs：= dira dirb
1_dirs:= dir1 dir2

a_files ：=文件a文件b
1_files ：= file1 file2

ifeq“$（use_a）”“yes”
a1:= a
其他的
a1:= 1
endif

ifeq“$（use_dirs）”“yes”
df:= dirs
其他的
df ：=文件
endif

dirs :=$($(a1)_$(df))
```

将根据use_a和use_dirs的设置，给dirs分配与sa_dirs、1_dirs、a_files or1_files相同的值。

也可以在替换引用中使用计算变量名：

```
a_对象：=a.o b. o c .o
1_ objects:= 1 .o 2 .o 3 .o

来源：= $（$（a1）_ objects：.o=.c）
```

根据a1的值，将源定义为“a.c b. c c.c”或“1.c 2.c3.c”。

对这种使用nest型变量引用的唯一限制是，它们不能指定要调用的函数名称的一部分。这是因为对已识别函数名称的测试是在展开嵌套引用之前进行的。例如，

```
ifdefdo_sort
函数：=sort
其他的
功能：=剪辑
endif

横杠：=a d b g q c

foo := $($(func)$(bar))
```

尝试将‘foo’的值赋给变量sort a d b g qc‘或strip a d b g q c’，而不是将‘a d b g qc’作为sort或strip函数的参数。如果这一改变被证明是好的，未来可以取消这一限制。

您也可以在变量赋值的左侧或定义指令中使用计算变量名，例如：

```
dir= foo
$（dir）_sources ：=$（wildcard$（dir）/*.c）
define$(dir)_print=
lpr$（$（dir）_sources）
endef
```

此示例定义变量“dir”、“foo_sources”和“foo_print”。

请注意，嵌套变量引用与递归展开变量有很大的不同（参见第66页第6.2节[变量的两种类型] ），在进行makefile程序ming时，两者都被以复杂的方式一起使用。

6.4变量如何获得其值

变量可以通过多种方式获取值：

运行make时，可以指定一个覆盖值。请参见第113页第9.5节[覆盖变量]。

可以在生成文件中指定一个值，可以使用赋值（参见第6.5节 [SettingVariables]，第72页 ）或逐字定义（见第6.8节[定义 多行变量]，第76页).

您可以使用let函数（见第98页第8.5[Let函数]节）或foreach函数（见第99页第8.6[Foreach函数]节）指定一个短生命周期值。

环境中的变量变为可变变量。请参见第6.10节[变量来自 《环境》，第77页 .

为每个规则赋予几个自动变量的新值。每个自动变量都有一个常规用途。请参见第130页的第10.5.3[AutomaticVariables]节。

几个变量具有常数初始值。请参见第10.3节[隐式规则使用的变量]，第125页。

6.5设置变量

要从生成文件中设置变量，请写一行开头为变量名，后面跟上赋值运算符=‘、’：=、：：=或‘：=’。运算符后面和行首的任何空格都将成为值。例如，

```
对象=main.ofoo.o bar .o utils .o
```

定义一个名为objects的变量，用于包含值“m main.ofoo.o bar.outils.o”。变量名周围的空白以及=之后的空白将被忽略。

用‘=’定义的变量是递归展开的变量。用‘：=’或‘：：=’定义的变量是简单展开的变量；这些定义可以包含在定义前会被展开的变量引用。用‘：：=’定义的变量是立即展开的变量。不同的赋值运算符在第6.2节[变量的两种类型]，第66页中有所描述。.

变量名可以包含函数和变量引用，当读取行以查找要使用的实际变量名时，这些引用会被展开。

变量值的长度没有限制，除了计算机上的内存量。为了便于阅读，可以将变量值拆分为多个物理行（参见第3.1.1节[拆分长行]，第12页。).

大多数变量名如果没有设置值，则默认为空字符串。一些变量具有内置的初始值，这些值并非空，但可以通过常规方式设置（见第10.3节[隐式规则使用的变量]，第125页）。几个特殊变量会根据每个规则自动设置为新值，这些称为自动变量（见第10.5.3节[自动变量]，第130页）。

如果希望只在变量尚未设置时才将其设置为某个值，那么可以使用简写运算符“？=”代替“=”。变量“FOO”的这两种设置是相同的（参见第8.11节[函数的起源]，第104页）：

```
FOO？=杆和
ifeq ($（origin FOO），未定义）
FOO=杆
endif
```

shell赋值运算符‘！=’可用于执行shell脚本并将变量设置为其输出。此运算符首先评估右侧表达式，然后将结果传递给shell进行执行。如果执行结果以一个换行符结尾，则该换行符将被移除；所有其他换行符将被替换为空格。生成的字符串随后会被放置到名为recursively-expanded的变量中。例如：

```
hash!=printf \043
文件列表！=查找.-name‘*.c’
```

如果执行结果可能产生一个$，并且你不希望后续的内容被解释为变量或函数引用，那么你必须在执行过程中将每个$替换为$$。或者，你可以使用shell函数调用将程序运行的结果设置为一个简单扩展的变量。参见第8.14节[shell函数]，第107页。例如：

```
哈希值：=$(shellprintf（‘\043’）
var ：= $（shell find. -name“*.c”）
```

与shell函数一样，刚刚调用的shell脚本的退出状态存储在. SHELLSTATUS变量中。

6.6向变量添加更多文本

通常，向已定义变量的值添加更多文本是有用的。您可以通过包含“+=”的一行代码来实现这一点，如下所示：

```
对象+=另一个.o
```

这将变量对象的值与文本“another.o”相加（如果它已有值，则在它前面添加一个空格）。因此：

```
对象=main.ofoo.o bar .o utils .o
对象+=另一个.o
```

将对象设置为“main.o foo .o bar .o utils .o another.o”。

使用“+=”与以下内容类似：

```
对象=main.ofoo.o bar .o utils .o
对象：=$（对象）另一个.o
```

但当使用更复杂的值时，它们在某些方面有所不同。

当变量未被定义时，‘+=’的行为与普通‘=’相同：它定义了一个递归扩展的变量。然而，如果有先前的定义，则‘+=’的具体行为取决于你最初定义的是哪种类型的变量。参见第6.2节[变量的两种类型]，第66页。 ，解释两种变量的类型。

当你用‘+=’来增加变量的值时，实际上就像你已经在变量的初始定义中包含了额外的文本一样。如果你最初是用‘：=’或‘：：=’定义它，使其成为一个简单扩展的变量，那么‘+=’会向这个简单扩展的定义中添加内容，并在将其附加到旧值之前扩展新的文本，就像‘：=’所做的那样（参见第6.5节 [设置变量]，第72页 ，为“：：=”或“：：=”的完整解释。事实上，

```
变量：=值
变量+=more与以
```

下内容完全等同：

```
变量：=值
变量：=$（变量）更多
```

另一方面，当您使用“+=”与您首先定义为使用plain‘=’或：：：=递归展开的变量一起使用时，将未展开的文本附加到现有值上，无论其是什么。这意味着

```
变量=值
变量+=more大致
```

相当于：

```
温度=值
变量=$（temp）更多
```

当然，它从未定义一个名为temp的变量。当变量的旧值包含变量引用时，这一点就显得很重要。请看这个常见的例子：

```
CFLAGS = $(包括）-O
...
CFLAGS+= - pg #启用配置文件
```

第一行定义了CFLAGS变量，它引用了另一个变量，包括。(CFLAGS在C编译规则中使用；请参见第10.2节[内置函数目录

规则]，第122页。）使用‘=’作为定义使得CFLAGS成为一个递归扩展的变量，这意味着当make处理CF LAGS的定义时，‘$（includes）-O’不会被扩展。因此，即使包含尚未定义，其值也会生效。它只需要在任何引用CFLAGS之前定义即可。如果我们尝试在不使用‘+=’的情况下追加CFLAGS的值，可以这样操作：

```
CFLAGS：= $（CFLAGS）- pg #启用配置文件
```

这已经很接近了，但还不完全符合我们的需求。使用‘：=’将CFLAGS重新定义为一个简单展开的变量；这意味着在设置变量时，make会将文本‘$（CFLAGS）-pg’扩展。如果includes尚未定义，我们会得到‘-O-pg’，后续对includes的定义将不起作用。相反，通过使用‘+=’我们将CFLAGS设置为未展开的值‘$（includes）-O-pg’。因此，我们保留了对includes的引用，所以如果该变量在稍后被定义，像‘$（CFLAGS）’这样的引用仍然会使用其值。

6.7超越指令

如果变量是通过命令参数设置的（见第9.5节[覆盖变量]，第113页），则生成文件中的普通赋值将被忽略。如果您希望即使变量已通过命令参数设置，仍能在生成文件中设置该变量，可以使用覆盖指令，其格式如下：

```
覆盖变量=值或
覆盖变量：=值
```

若要在命令行定义的变量中追加更多文本，请使用：

```
覆盖变量+=更多文本
```

参见第6.6节[向Variables添加更多文本]，第74页 .

带有覆盖标记的变量赋值的优先级高于所有其他赋值，但另一个覆盖除外。未带有覆盖标记的后续赋值或追加到此变量的赋值将被忽略。

覆盖指令不是为在生成文件和命令参数之间的战争中升级而发明的。它被发明是为了可以更改和添加用户通过命令参数指定的值。

例如，假设您总是希望在运行C编译器时使用“-g”开关，但您希望允许用户像往常一样通过命令参数指定其他开关。您可以使用此重写指令：

```
覆盖CFLAGS +=-g
```

您也可以使用覆盖指令与定义指令一起使用。这就像您可能期望的那样：

```
覆盖定义foo=
条
endef
```

有关定义的信息，请参见下一节。

6.8定义多行变量

另一种设置变量值的方法是使用define指令。该指令具有不寻常的语法，允许在值中包含换行字符，这对于定义预设的命令序列非常方便（参见第5.8节[定义CannedRecipes]，第61页 ），以及与eval一起使用的部分makefile语法（请参见第10页第8.10[EvalFunction]节）。

定义指令在同一行中跟随着要定义的变量名和一个（可选的）赋值运算符，之后没有任何内容。给变量赋予的值出现在以下行中。值的结束由仅包含单词endef的一行标记。

除了语法上的差异之外，定义工作就像定义其他变量一样。变量名可以包含函数和变量引用，当读取指令时，这些引用会被展开以查找要使用的实际变量名。

endef之前的最后一个换行符不包含在值中；如果要使值包含一个尾随换行符，则必须包含一个空行。例如，为了定义一个包含一个换行符的变量，必须使用两个空行，而不是一个：

```
定义换行符


endef
```

如果需要，可以省略变量赋值运算符。如果省略了，则假设其为“=”，并创建递归展开的变量（参见第6.2节[两种类型 变量]，第66页 ）。使用“+=”运算符时，值将附加到前一个值，就像其他附加运算一样：用空格分隔旧值和新值。

您可以嵌套定义指令：make将跟踪嵌套指令，并在它们没有全部正确地用endef结束时报告错误。请注意，以recipe前缀字符开头的行被视为配方的一部分，因此出现在此类行上的任何define或endef字符串都不会被视为make指令。

```
定义两行
echo foo
echo$(bar)
endef
```

在配方中使用时，前面的示例在功能上等同于以下内容：

```
两行=echofoo；echo$（bar）
```

由于两个由分号分隔的命令的行为与两个独立的shell命令非常相似。但是，注意使用两个独立的行意味着make将调用两次shell，为每行运行一个独立的子shell。请参见第5.3节[配方执行]，第48页 .

如果希望用define定义的变量优先于命令行变量定义，请使用override指令与define一起使用：

```
覆盖definetwo-lines=
foo
$(bar)
endef
```

见第6.7节【越权指令】第75页 .

6.9未定义变量

如果你想清除一个变量，将其值设为空通常就足够了。扩展这样的变量无论是否已设置，都会产生相同的结果（空字符串）。然而，如果你使用的是flavor（见Section8.12 [Flavor Function]，第105页）和origin（见第104页的Section 8.11[OriginFunction]）函数，从未设置过的变量和具有空值的变量之间存在差异。在这种情况下，你可能希望使用undefine指令来使变量看起来好像从未被设置过。例如：

```
foo:= foo
条形=条形

undefine foo
取消定义栏

$(info$(origin foo))
$（信息$（口味条））
```

此示例将为这两个变量打印“未定义”。

如果要取消定义命令行变量定义，可以使用override指令与undef ine一起使用，类似于为变量定义执行的操作：

```
覆盖未定义的CFLAGS
```

6.10环境变量

make中的变量可以来自运行make的环境。每当make启动时，它看到的每个环境变量都会被转换成一个同名且值相同的make变量。然而，make文件中的显式赋值或命令参数会覆盖环境变量。（如果指定了‘-e’标志，则环境中的值将覆盖make文件中的赋值。参见第114页第9.8节[选项概要]。但不建议这样做。）

因此，通过在环境中设置变量CFLAGS，您可以使用大多数生成文件中的所有编译器来选择您偏好的编译器开关。这适用于具有标准或常规含义的变量，因为您可以确信没有生成文件会将它们用于其他用途。（请注意，这并非完全可靠；某些生成文件会显式设置CFLAGS，因此不会受环境中的值影响。）

当构建一个配方时，make文件中定义的一些变量会被放置到每个命令调用的环境中。默认情况下，只有来自make环境或通过其命令行设置的变量才会被放置到命令的环境中。你可以使用 export 指令来传递其他变量。请参见第 5.7.2 节 。【向子制造传达变量】，第57页 详情请参阅。

不建议使用环境变量的其他用途。由于这会导致不同用户从同一个生成文件中得到不同的结果，因此，对于生成文件来说，依赖于其外部控制之外的环境变量是不明智的。这违背了大多数生成文件的初衷。

这类问题在可变外壳中尤其可能出现，通常环境会指定用户选择交互式外壳。如果这种选择影响到make，则非常不希望；因此，make以特殊方式处理外壳环境的可变性；参见第5.3.2节[选择外壳]，第49页 .

6.11目标特定变量值

变量值在`make`中通常是全局的；也就是说，无论它们在哪里被评估（当然，除非它们被重置），它们都是相同的。例外情况包括使用`let`函数定义的变量（见第8.5节[Let函数]，第98页）或`foreach`函数（见第8.6节[Foreach函数]，第99页），以及自动变量（见第10.5.3节[自动变量]，第130页）。

另一个例外是目标特定变量值。此功能允许您根据当前正在构建的目标为同一变量定义不同的值。与自动变量一样，这些值仅在目标的配方上下文中可用（以及其他目标特定的赋值）。

设置特定于目标的变量值，例如：

目标...：变量赋值

目标特定变量赋值可以以任何或全部特殊关键字export、unexport、override或private作为前缀；这些关键字仅对这个变量实例应用其正常行为。

多个目标值为目标列表中的每个成员单独创建目标特定的变量值。

变量赋值可以是任何有效的赋值形式；递归（‘=’）、简单（‘：=’或‘：：=’）、即时（‘：：=’）、追加（‘+=’）或条件（‘？=’）。变量赋值中出现的所有变量都在目标上下文中进行评估：因此，任何先前定义的目标特定变量值都将生效。请注意，这个变量实际上与任何“全局”值不同：这两个变量不必具有相同的类型（递归与简单）。

目标特定变量与任何其他生成文件变量具有相同的优先级。命令行（如果启用了th‘-e’选项，则在环境中）提供的变量将具有优先权。指定override指令将允许优先使用目标特定变量值。

还有一个关于目标特定变量的特殊功能：当你定义一个目标特定变量时，该变量的值对这个目标的所有前提条件以及它们的所有前提条件等都生效（除非这些前提条件用它们自己的目标特定变量值覆盖了该变量）。因此，例如，这样的语句：

程序：CFLAGS=-g
进程：进程.o foo .o bar .o

将在prog的配方中将CFLAGS设置为“-g”，但也会在创建prog .o、foo .o和bar .o的配方以及创建这些前提条件的任何配方中将CFLAGS设置为“-g”。

请注意，给定的前提条件最多只会被构建一次。如果同一个文件是多个目标的前提条件，而每个目标对同目标特定变量都有不同的值，则第一个要构建的目标是

构建该先决条件，然后该先决条件将从第一个目标继承目标特定值。它将忽略来自任何其他目标的目标特定值。

6.12模式特定变量值

除了目标特定变量值（见第6.11节[目标特定变量值 ues]，第78页），GNU make支持模式特定的变量值。以这种形式，变量被定义为与指定模式匹配的任何目标。

设置特定模式的变量值，如下所示：

模式...：变量分配

其中模式是a%-模式。与目标特定变量值一样，多个模式值为每个模式单独创建一个模式特定的变量值。变量赋值可以是任何有效的赋值形式。除非指定了覆盖，否则任何命令行变量设置都将优先。

例如：

```
%.o : CFLAGS=-O
```

将为所有与模式%.o匹配的目标分配CFLAGS值“-O”。

如果一个目标与多个模式匹配，则首先解释具有较长茎干的匹配模式特定变量。这导致更具体的变量优先于更通用的变量，例如：

```
%.o:% .c
        $(CC) -c $(CFLAGS)$(CPPFLAGS)$<-o$@

lib/%.o: CFLAGS:= - fPIC -g
%.o: CFLAGS:=-g

所有：foo.lib/bar.o
```

在本例中，将使用第一个CFLAGS变量定义来更新lib/ bar. o，尽管第二个变量定义也适用于此目标。模式特定变量产生的相同茎长按它们在生成文件中定义的顺序进行考虑。

在为该目标明确定义的任何目标特定变量之后，在为父目标定义的目标特定变量之前搜索模式特定变量。

6.13禁止继承

如前几节所述，变量由先决条件继承。此功能允许您根据导致先决条件重建的目标来修改其行为。例如，您可以在调试目标上设置一个特定于目标的变量，然后运行“make debug”会使该变量被调试的所有先决条件继承，而仅仅运行“make all”（例如）则不会进行这种赋值。

有时，你可能不希望某个变量被继承。对于这些情况，`make`提供了私有修饰符。虽然这个修饰符可以用于任何变量赋值，但它最适合于目标和模式特定的变量。任何标记为私有的变量都会对其本地目标可见，但不会被前置条件继承。

该目标。标记为私有的全局变量将在全局作用域中可见，但不会被任何目标继承，因此在任何配方中均不可见。

例如，考虑以下makefile：

```
extra_cflags=

程序：私有EXTRA_CFLAGS = -L/usr/local/lib
程序： a. o b .o
```

由于私有修饰符，a. o和b. o不会继承prog targ et的EXTRA_ CFLAGS变量赋值。

6.14其他特殊变量

GNU make支持具有特殊属性的一些变量。

makefile_list

包含每个由make解析的生成文件的名称，按解析顺序排列。名称在make开始解析生成文件之前附加。因此，如果生成文件首先检查此变量中的最后一个单词，那么它将是当前生成文件的名称。然而，一旦当前生成文件使用了include，最后一个单词将是刚刚包含的生成文件。

如果名为makefile的生成文件包含以下内容：

```
名称1：= $（lastword $（MAKEFILE_ LIST））

包括inc .mk

名称2：= $（lastword $（MAKEFILE_ LIST））

所有
        @echo name1=$(name1)
        @echo name2=$(name2)
```

然后，您将期望看到以下输出：

```
name1 = Makefile
name2 = inc .mk
```

默认目标

设置默认目标，如果命令行中未指定目标（见第9.2节[指定目标的参数]，第109页）。.DEFAULT_ GOAL变量允许您发现当前的默认目标，通过清除其值重新启动默认目标选择算法，或显式设置默认目标。以下示例说明了这些情况：

```
#查询默认目标。
ifeq（$（. DEFAULT_ GOAL），）
  $（警告：未设置默认目标）
endif

.PHONY： foo
foo:; @echo$@

$（警告，默认目标为$（DEFAULT_GOAL））

#重置默认目标。
默认目标：=

.假币：酒吧
条目：；@echo$@

$（警告，默认目标为$（DEFAULT_GOAL））

#设置我们自己的。
. DEFAULT_GOAL ：= foo
```

此生成文件打印：

```
未设置默认目标
默认目标为foo
默认目标是条形
foo
```

注意：为. DEFAULT_GOAL分配一个以上目标名称是无效的，并且会导致错误。

重新启动

只有当此就绪状态重新启动时才设置此变量（参见第3.5节 [如何重写Makefile]，第15页 ）：它将包含此实例重新启动的次数。请注意，这与递归（由MAKELEVEL变量计数）不同。您不应设置、修改或导出此变量。

make_termout
make_termerr

当make开始运行时，它会检查stdout和stderr是否会在终端显示输出。如果会，它将分别setMAKE_TERMOUT一个dMAKE_TERMERR到终端设备的名称（或在无法确定时为true）。如果设置这些变量，它们将被标记为导出。这些变量不会被make更改，即使已经设置也不会修改。

这些值可以使用（特别是与输出同步结合使用（参见第53页第5.4.2节[并行执行期间的输出])以确定自己是否在向一个终端写入；它们可以被测试以决定是否强制配方命令生成彩色输出，例如。

如果调用子make并重定向其stdout或stderr，则如果您的mak文件依赖于它们，则您有责任重置或取消导出这些变量。

. RECIPEPREFIX

此变量的第一个字符用作字符make假设引入一个配方行。如果变量为空（默认情况下是这样），则该字符为标准制表符。例如，这是有效的makefile：

```
. RECIPEPREFIX =>
所有
> @ echo Hello，world
```

可以多次更改.RECIPEPREFIX的值；一旦设置，它将一直对所有解析的规则生效，直到被修改。

变量

扩展为迄今为止定义的所有全局变量的名称列表。这包括具有空值的变量以及内置变量（参见第10.3节[隐式规则使用的变量]，第125页），但不包括仅在目标特定上下文中定义的任何变量。请注意，您分配给此变量的任何值都将被忽略；它将始终返回其特殊值。

.特性

扩展为由此版本的make支持的特殊功能列表。可能的值包括但不限于：

档案

使用特殊文件名语法支持ar（存档）文件。请参见第11章[使用make更新存档文件]，第1 39页。

‘检查-symlink’

支持-l（--check-symlink-times）标志。请参见第114页的第9.8[选项摘要]节。

else-if支持“else if”非嵌套条件语句。请参见第7.2节[语法]　　关于条件句]，第86页　.

‘额外先决条件’

支持. EXTRA_PREREQS特定目标。

‘分组目标’

支持分组目标语法以实现显式规则。请参见第4.9节　　[规则中的多个目标]，第37页　　.

“Guile”GNU Guile是否可用作嵌入式扩展语言。请参阅第12.1[GNU GuileIntegration]节，第143页。

jobserver

支持增强的“作业服务器”并行构建。请参见第5.4节　[并行执行]，第51页　.

jobserver-fifo

支持使用命名管道增强的“jobserver”并行构建。请参见第13章[集成GNUmake]，第153页。

“load”支持用于创建自定义扩展的动态可加载对象。请参见第12.2[加载动态对象]节，第145页。

notintermediate
支持. NOTINTERMEDIATE特定目标。请参阅第13章[Integrating GNUmake]，第153页。

oneshell
支持. ONESHELL特定目标。请参见第5.3.1节[使用 OneShell]，第48页。

仅限订购
支持仅订购的先决条件。请参见第4.2节[类型预 requisites]，第24页。

‘输出同步’
支持-- output- sync命令行选项。请参见第114页第9.8[选项总结]节。

“二次膨胀”
支持对先决条件列表的二次扩展。

“出口壳”
支持将变量导出到shell函数。

‘最短的茎’
使用“最短茎”方法选择多个适用选项中的哪个模式将被使用。请参见Section10.5.4 [How PatternsMatch]，第133页。

‘特定目标’
支持针对目标和模式的变量分配。请参见第6.11节[针对目标的变量值]，第78页 .

undefine
支持undefine指令。请参见第6.9节[undefine指令 tive]，第77页。

. include_dirs
扩展为一个目录列表，其中包含要搜索的mak文件（参见第13页第3.3节[包括其他mak文件] ）。请注意，修改此变量的值不会更改搜索的目录列表。

. extra_prereqs
此变量中的每个单词都是一个新的先决条件，会添加到设置的目标中。这些先决条件与普通先决条件不同，因为它们不会出现在任何自动变量中（见第10.5.3[AutomaticVariables]节，第130页）。这使得可以定义不会影响配方的先决条件。

考虑一个链接程序的规则：

```
myprog： myprog .o文件1.o文件2 .o
        $(CC) $(CFLAGS) $(LDFLAGS)-o$@$^$(LDLIBS)
```

现在假设您想增强这个生成文件，以确保对编译器的更新使程序可链接。您可以将编译器添加为先决条件，但必须确保它不作为链接命令的参数传递。您需要类似这样的东西：

```
myprog: myprog.o file1.o file2.o$(CC)
        $(CC) $(CFLAGS)$(LDFLAGS)- o$@\
            $(filter-out$(CC),$^)$(LDLIBS)
```

然后考虑使用多个额外的前提条件：它们都必须被过滤掉。使用.EXTRA_PREREQS和目标特定变量提供了一个更简单的解决方案：

```
myprog： myprog .o文件1.o文件2 .o
        $（CC）$（CFLAGS）$（LDFLAGS）-o$@$^$（LDLI
BS）myprog：.EXTRA_PREREQS =$（CC）
```

如果要向不易修改的生成文件添加先决条件，此功能也很有用：您可以创建一个新的文件，例如asextra.mk：

```
myprog：. EXTRA_PREREQS =$（CC）
```

然后调用make-f extra.mk-f Makefile。

设置. EXTRA_PREREQS全局化将导致这些先决条件被添加到所有目标（这些目标本身没有用特定于目标的值覆盖它）。注意，make已足够智能，不会将. EXTRA_PREREQS中列出的先决条件添加为其elf的先决条件。

7生成文件的条件部分

条件指令根据变量的值决定是否执行或忽略生成文件的一部分。条件语句可以比较两个变量的值，或者将一个变量的值与常量字符串进行比较。条件语句控制生成器实际看到的生成文件内容，因此不能用于在执行时控制配方。

7.1条件示例

以下示例展示了条件语句，如果变量`CC`的值为‘ gcc’，则使用一组库；否则使用另一组库。它通过控制两条指令中哪一条会被用于规则来实现这一点。结果是，`CC=gcc`作为参数不仅改变了所使用的编译器，还改变了链接的库。

```
libs_for_gcc=-林根
正常libs=

foo:$(objects)
ifeq ($(CC),gcc)
        $（CC）-ofoo$（对象）$（libs_for_gcc）
其他的
        $（CC）-o foo$（对象）$（normal_libs）
endif
```

此条件语句使用了三个指令：一个ifeq、一个else和一个endi f。

ifeq指令开始条件语句，并指定条件。它包含两个参数，用逗号分隔并用括号包围。对这两个参数进行变量替换，然后进行比较。如果两个参数匹配，则会执行ifeq之后的makefile行；否则这些行将被忽略。

else指令使如果前一个条件失败，则执行以下行。在上面的例子中，这意味着当第一个替代方案不使用时，将使用第二个链接替代方案。条件中可有或没有else指令。

endif指令结束条件。每个条件都必须以endif结束。无条件生成文件文本如下。

正如这个例子所示，条件句在文本层面起作用：条件句的行被视为构建文件的一部分，或者根据条件被忽略。这就是为什么构建文件中的较大语法单元，如规则，可能会跨越条件句的开头或结尾。

当变量CC的值为gcc时，上述示例h具有以下效果：

```
foo:$(objects)
        $（CC）-o foo$（objects）$（libs_for_g
```

cc）当变量CC具有其他值时，效果如下：

```
foo:$(objects)
        $（CC）-o foo$（对象）$（normal_libs）
```

也可以通过条件化变量赋值，然后无条件地使用变量来获得等效结果：

```
libs_for_gcc=-lgnu
正常libs=

ifeq ($(CC),gcc)
  库=$（libs_for_gcc）
其他的
  libs=$（正常libs）
endif

foo:$(objects)
        $(CC)-o foo$(objects)$(libs)
```

7.2条件句的语法

没有else的简单条件语句的语法如下：

```
条件指令
文本-如果为真
endif
```

如果条件为true，则Thetext-if-true可以是任何行文本，如果条件为true，则将它们视为生成文件的一部分。如果条件为false，则不使用任何文本，而是使用。

复杂条件句的语法如下：

```
条件指令
文本-如果是
其他的
文本-如果为假
endif
```

或

```
条件指令-一
如果一个为真，则显示文本
其他条件指令二
如果两个条件都为真，则显示文本
其他的
如果一和二为假，则文本为
endif
```

可以有任意数量的“其他条件指令”子句。一旦某个条件为真，就会使用text- if- true子句，而不会使用其他任何子句；如果没有条件为真，则会使用text- if- false子句。text- if- true和text- if- false子句可以是任意多行文本。

条件指令的语法，无论条件是简单还是复杂；在else或not后。有四种不同的指令来测试不同的条件。下面是它们的表格：

```
ifeq (arg1 , arg2)
ifeqarg1arg2
ifeq"arg1""arg2"
ifeq"arg1"arg2
ifeqarg1"arg2"
```

展开arg1和arg2中的所有变量引用并比较它们。如果它们相同，则text- if- true有效；否则，text- if- false有效，如果有任何变量引用的话。

通常你希望测试一个变量是否具有非空值。当值由变量和函数的复杂展开产生时，你认为为空的表达式实际上可能包含空白字符，因此不会被视为为空。然而，你可以使用strip函数（见第8.2节[文本函数]，第92页）来避免将空白字符解释为非空值。例如：

```
ifeq($(strip$(foo)),)
文本-如果为空
endif
```

将评估文本-如果-空，即使$（foo）的扩展包含空白字符。

```
ifneq (arg1 , arg2)
ifneqarg1arg2
ifneq"arg1""arg2"
ifneq"arg1"arg2
ifneqarg1"arg2"
```

展开arg1和arg2中的所有变量引用并比较它们。如果它们不同，则text- if- true有效；否则，如果有text- if- false，则if- true有效。

```
ifdefvariable-name
```

`ifdef`形式接受一个变量名作为其参数，而不是对变量的引用。如果该变量的值非空，则`text- if- true`有效；否则，`text-if-false`有效。从未定义过的变量具有空值。`textvariable-name`被展开，因此它可以是一个变量或函数，扩展后成为变量名。例如：

```
条形图=真
foo= bar
ifdef$(foo)
frobozz=是
endif
```

变量reference$（foo）被展开，生成bar，它被视为变量的名称。变量bar不被展开，但检查其值以确定它是否非空。

注意，ifdef仅测试变量是否有值。它不会将变量展开，以查看该值是否非空。因此，使用ifdef的测试

对于除foo=等定义之外的所有定义，返回true。要测试emp ty值，请使用ifeq（$（foo），）。例如，

```
bar=
foo=$(bar)
ifdef foo
frobozz=是
其他的
frobozz=否
endif
```

将“set’sfrobozz”设置为“yes”，同时：

```
foo=
ifdef foo
frobozz=是
其他的
frobozz=否
endif
```

setsfrobozz tono .

ifndefvariable-name

如果变量variable- name的值为空，则text- if- true有效；否则，text-if-false有效。变量名的展开和测试规则与ifdef指令相同。

条件指令行开头允许并忽略额外空格，但不允许使用制表符。（如果行以制表符开始，则会被视为规则配方的一部分。）除此之外，除非在指令名称或参数内，否则任何位置插入额外空格或制表符均无效。注释以‘#’开头，可以出现在行尾。

另外两个在条件语句中起作用的指令是else和endif。这两个指令各由一个单词组成，没有参数。行首允许并忽略空格，行尾可以有空格或制表符。以#开头的注释可能出现在行尾。

条件语句影响哪些行在构建文件中被使用。如果条件为真，`make`会读取`text- if- true`中的行作为构建文件的一部分；如果条件为假，`make`将完全忽略这些行。因此，构建文件中的语法单元，如规则，可以安全地拆分为条件语句的开头或结尾。

make在读取make文件时会评估条件语句。因此，不能在条件语句的测试中使用自动变量，因为这些变量是在运行配方后才定义的（请参见第130页的“10.5.3[自动变量]”）。

为防止出现无法忍受的混乱，不允许在一个生成文件中开始一个条件而在另一个生成文件中结束它。但是，您可以在条件中写入一个包含指令，只要您不试图在包含的文件中终止条件。

7.3测试标志的条件

您可以使用变量MAKEFLAGS和findstring函数编写一个条件来测试make命令标志，例如“-t”（请参见第8.2节[字符串函数