# 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* ‘mean’ 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*隐含规则*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[使用隐含规则]，第111页。

如果一个’.c’文件使用*隐含规则*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。）

因为隐含规则如此方便，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

当使用隐含规则创建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。