**Makefile 及其工作原理**

[**Linux中国**](https://www.zhihu.com/org/linuxzhong-guo)

[**​**](https://www.zhihu.com/question/48510028)

已认证帐号

76 人赞同了该文章

用这个方便的工具来更有效的运行和编译你的程序。

make 工具通过读取一个 Makefile（或 makefile）文件来决定如何运行，在该文件中定义了一系列需要执行的任务。你可以使用 make 来将源代码编译为可执行程序。大部分开源项目会使用 make 来实现最终的二进制文件的编译，然后使用 make install 命令来执行安装。

本文将通过一些基础和进阶的示例来展示 make 和 Makefile 的使用方法。在开始前，请确make保你的系统中安装了 make。

**基础示例**

依然从打印 “Hello World” 开始。首先创建一个名字为 myproject 的目录，目录下新建 Makefile文件，文件内容为：

say\_hello:

echo "Hello World"

echo 命令是 Linux 中最基本和最常用的命令之一。 传递给 echo 的参数被打印到标准输出中。

在 myproject 目录下执行 make，会有如下输出：

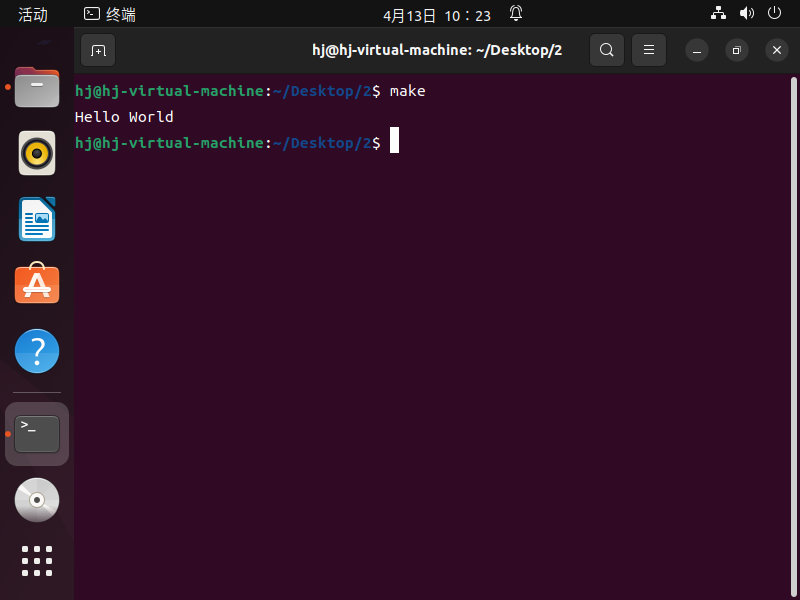
$ make

echo "Hello World"

Hello World

当 make 被执行时，整条指令 echo "Hello World" 都被显示出来，之后才是真正的执行结果。如果不希望指令本身被打印处理，需要在 echo 前添加 @。





在上面的例子中，“say\_hello” 类似于其他编程语言中的函数名。这被称之为 *目标(target)*。在该目标之后的是预置条件或依赖。为了简单起见，我们在这个示例中没有定义预置条件。echo ‘Hello World' 命令被称为 *步骤(recipe)*。这些步骤基于预置条件来实现目标。目标、预置条件和步骤共同构成一个规则。

总结一下，一个典型的规则的语法为：

目标: 预置条件

<TAB> 步骤

作为示例，目标可以是一个基于预置条件（源代码）的二进制文件。另一方面，预置条件也可以是依赖其他预置条件的目标。

final\_target: sub\_target final\_target.c

Recipe\_to\_create\_final\_target

sub\_target: sub\_target.c

Recipe\_to\_create\_sub\_target

目标并不要求是一个文件，也可以只是步骤的名字，就如我们的例子中一样。我们称之为“伪目标”。

===================

再回到上面的示例中，

接下来在 Makefile 中添加如下伪目标：generate 和 clean：

say\_hello:

@echo "Hello World"

generate:

@echo "Creating empty text files..."

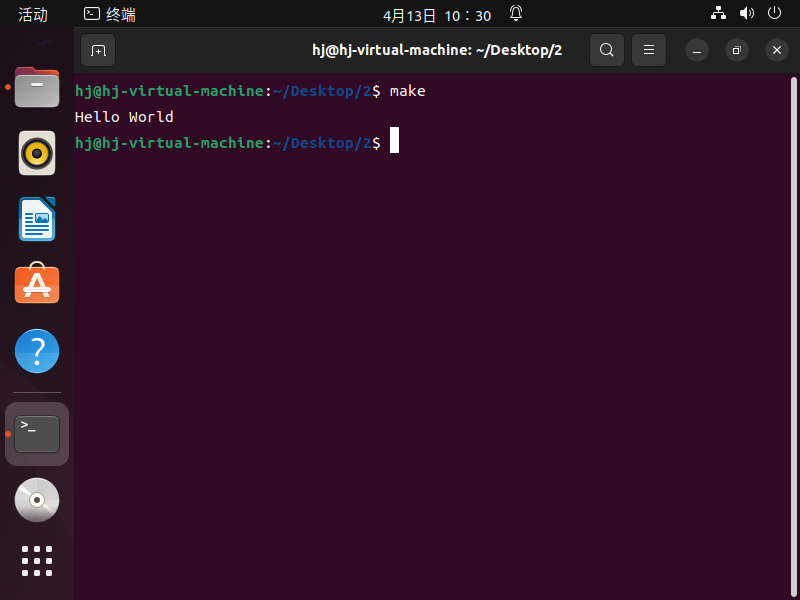
touch file-{1..10}.txt

clean:

@echo "Cleaning up..."

rm \*.txt

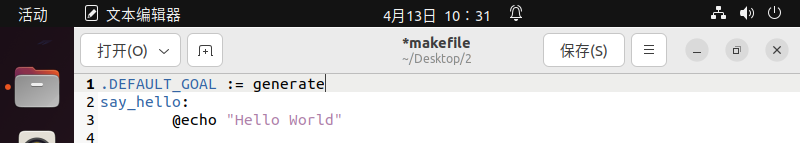
随后当我们运行 make 时，只有 say\_hello 这个目标被执行。

这是因为Makefile 中的第一个目标为默认目标。通常情况下会调用默认目标。我们可以通过 .DEFAULT\_GOAL 这个特殊的伪目标来覆盖掉默认的行为。

在 Makefile 文件开头增加 .DEFAULT\_GOAL：

.DEFAULT\_GOAL := generate

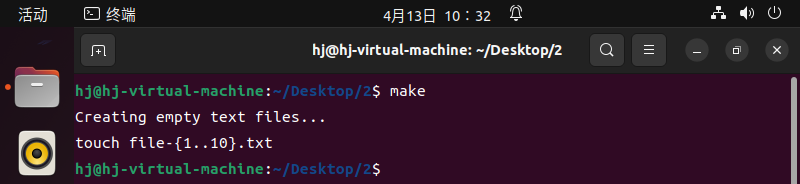
make 会将 generate 作为默认目标：



$ make

Creating empty text files...

touch file-{1..10}.txt



顾名思义，.DEFAULT\_GOAL 伪目标仅能定义一个目标。这就是为什么很多 Makefile 会包括 all 这个目标，这样可以调用多个目标。

下面删除掉 .DEFAULT\_GOAL，增加 all 目标：

all: say\_hello generate

say\_hello:

@echo "Hello World"

generate:

@echo "Creating empty text files..."

touch file-{1..10}.txt

clean:

@echo "Cleaning up..."

rm \*.txt

运行之前，我们再增加一些特殊的伪目标。.PHONY 用来定义这些不是文件的目标。make 会默认调用这些伪目标下的步骤，而不去检查文件名是否存在或最后修改日期(*也就是忽略一般错误*)。完整的 Makefile 如下：

.PHONY: all say\_hello generate clean

all: say\_hello generate

say\_hello:

@echo "Hello World"

generate:

@echo "Creating empty text files..."

touch file-{1..10}.txt

clean:

@echo "Cleaning up..."

rm \*.txt

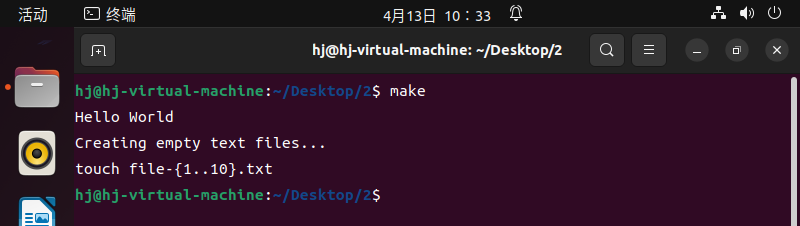
make 命令会调用 say\_hello 和 generate：

$ make

Hello World

Creating empty text files...

touch file-{1..10}.txt



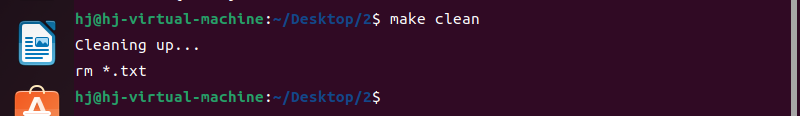
**另外：**

clean 不应该被放入 all 中，或者被放入第一个目标中。clean 应当在需要清理时手动调用，调用方法为 make clean。

$ make clean

Cleaning up...

rm \*.txt



现在你应该已经对 Makefile 有了基础的了解，接下来我们看一些进阶的示例。

**进阶示例**

**变量**

在之前的实例中，大部分目标和预置条件是已经固定了的，但在实际项目中，它们通常用变量和模式来代替。

定义变量最简单的方式是使用 = 操作符。例如，将命令 gcc 赋值给变量 CC：

CC = gcc

这被称为递归扩展变量，用于如下所示的规则中：

hello: hello.c

${CC} hello.c -o hello

你可能已经想到了，这些步骤将会在传递给终端时展开为：

gcc hello.c -o hello

${CC} 和 $(CC) 都能对 gcc 进行引用。但如果一个变量尝试将它本身赋值给自己，将会造成死循环。让我们验证一下：

CC = gcc

CC = ${CC}

all:

@echo ${CC}

此时运行 make 会导致：

$ make

Makefile:8: \*\*\* Recursive variable 'CC' references itself (eventually). Stop. 递归变量“CC”引用自身（最终）

为了避免这种情况发生，可以使用 := 操作符（这被称为简单扩展变量）。以下代码不会造成上述问题：

CC := gcc

CC := ${CC}

all:

@echo ${CC}

**Makefile 隐式规则**

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

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

一、使用隐含规则

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

**模式和函数**

下面的 Makefile 使用了变量、模式和函数来实现所有 C 代码的编译。我们来逐行分析下：

# Usage:

# make # compile all binary

# make clean # remove ALL binaries and objects

.PHONY = all clean

CC = gcc # compiler to use

LINKERFLAG = -lm

SRCS := $(wildcard \*.c)

BINS := $(SRCS:%.c=%)

all: ${BINS}

%: %.o

@echo "Checking.."

${CC} ${LINKERFLAG} $< -o $@

%.o: %.c

@echo "Creating object.."

${CC} -c $<

clean:

@echo "Cleaning up..."

rm -rvf \*.o ${BINS}

* 以 # 开头的行是注释。
* .PHONY = all clean 行定义了 all 和 clean 两个伪目标。
* 变量 LINKERFLAG 定义了在步骤中 gcc 命令需要用到的参数。
* SRCS := $(wildcard \*.c)：$(wildcard pattern) 是与文件名相关的一个函数。在本示例中，所有 “.c”后缀的文件会被存入 SRCS 变量。
* BINS := $(SRCS:%.c=%)：这被称为替代引用。本例中，如果 SRCS 的值为 'foo.c bar.c'，则 BINS的值为 'foo bar'。
* all: ${BINS} 行：伪目标 all 调用 ${BINS} 变量中的所有值作为子目标。
* 规则：  
  %: %.o @echo "Checking.." ${CC} ${LINKERFLAG} $&lt; -o $@  
  下面通过一个示例来理解这条规则。假定 foo 是变量 ${BINS} 中的一个值。% 会匹配到 foo（%匹配任意一个目标）。下面是规则展开后的内容：  
  foo: foo.o @echo "Checking.." gcc -lm foo.o -o foo  
  如上所示，% 被 foo 替换掉了。$< 被 foo.o 替换掉。$<用于匹配预置条件，$@ 匹配目标。对 ${BINS} 中的每个值，这条规则都会被调用一遍。
* 规则：  
  %.o: %.c @echo "Creating object.." ${CC} -c $&lt;  
  之前规则中的每个预置条件在这条规则中都会都被作为一个目标。下面是展开后的内容：  
  foo.o: foo.c @echo "Creating object.." gcc -c foo.c
* 最后，在 clean 目标中，所有的二进制文件和编译文件将被删除。

下面是重写后的 Makefile，该文件应该被放置在一个有 foo.c 文件的目录下：

# Usage:

# make # compile all binary

# make clean # remove ALL binaries and objects

.PHONY = all clean

CC = gcc # compiler to use

LINKERFLAG = -lm

SRCS := foo.c

BINS := foo

all: foo

foo: foo.o

@echo "Checking.."

gcc -lm foo.o -o foo

foo.o: foo.c

@echo "Creating object.."

gcc -c foo.c

clean:

@echo "Cleaning up..."

rm -rvf foo.o foo

关于 Makefile 的更多信息，[GNU Make 手册](https://link.zhihu.com/?target=https%3A//www.gnu.org/software/make/manual/make.pdf)提供了更完整的说明和实例。