

编 制：北京万邦易嵌科技有限公司

版 本：V2.0

编制日期：2016 年 1 月 10 日

修改日期：2018 年 1 月 10 日

版权声明：该培训教程版权归北京万邦易嵌科技有限公司所有，未经公司授权禁止引用、发布、转载等，否则将追究其法律责任。

## Makefile 使用规则

### 1.1 makefile 简介

在 Linux (unix ) 环境下使用 GNU 的 make 工具能够比较容易的构建一个属于你自己的工程，整个工程的编译只需要一个命令就可以完成编译、连接以至于最后的执行。不过这需要我们投入一些时间去完成一个或者多个称之为 Makefile 文件的编写。

所要完成的 Makefile 文件描述了整个工程的编译、连接等规则。其中包括：工程中的哪些源文件需要编译以及如何编译、需要创建那些库文件以及如何创建这些库文件、如何最后产生我们想要的可执行文件。尽管看起来可能是很复杂的事情，但是为工程编写 Makefile 的好处是能够使用一行命令来完成“自动化编译”，一旦提供一个（通常对于一个工程来说会是多个）正确的 Makefile。编译整个工程你所要做的唯一的一件事就是在 shell 提示符下输入 **make** 命令。整个工程完全自动编译，极大提高了效率。

make 是一个命令工具，它解释 Makefile 中的指令（应该说是规则）。在 Makefile 文件中描述了整个工程所有文件的编译顺序、编译规则。Makefile 有自己的书写格式、关键字、函数。像 C 语言有自己的格式、关键字和函数一样。而且在 Makefile 中可以使用系统 shell 所提供的任何命令来完成想要的工作。Makefile（在其它的系统上可能是另外的文件名）在绝大多数的 IDE 开发环境中都在使用，已经成为一种工程的编译方法。

### 1.2 编写 makefile 格式说明

- ① Makefile 文件里使用 shell 命令时，命令的前面必须是 TAB 键。
- ② 输入 make 默认执行 Makefile 文件里的第一个命令。
- ③ 在 Makefile 文件里#号代表注释
- ④ Make 命令支持寻找解析：makefile”和“Makefile”这两种默认文件名。
- ⑤ 如果要指定特定的 Makefile，你可以使用 make 的“-f”和“--file”参数，如：make -f Make.Linux 或 make --file Make.AIX。
- ⑥ make -v 输出 make 版本和版权问题
- ⑦ makefile 里使用 echo 命令进行信息输出,类似于 C 语言的 printf。示例:echo “12345” 或者 echo \$(ABC)
- ⑧ 在 shell 命令前加上@符号，可以隐藏命令的执行过程！ 比如：@echo “12345”，只会输出 12345。

■ Make 命令的参数选项：

Make 命令本身可带有四种参数：标志、宏定义、描述文档名和目标文档名。其标准形式为：

Make [flags] [macro definitions] [targets]

Unix 系统下标志位 flags 选项及其含义为：

**-f file** 指定 file 文档为描述文档，假如 file 参数为 "-" 符，那么描述文档指向标准输入。假如没有 "-f" 参数，则系统将默认当前目录下名为 makefile 或名为 Makefile 的文档为描述文档。在 Linux 中，GNU make 工具在当前工作目录中按照 GNUmakefile、makefile、Makefile 的顺序搜索 makefile 文档。

- i** 忽略命令执行返回的出错信息。
- s** 沉默模式，在执行之前不输出相应的命令行信息。
- r** 禁止使用 build-in 规则。
- n** 非执行模式，输出任何执行命令，但并不执行。
- t** 更新目标文档。
- q** make 操作将根据目标文档是否已更新返回 "0" 或非 "0" 的状态信息。
- p** 输出任何宏定义和目标文档描述。
- d** Debug 模式，输出有关文档和检测时间的周详信息。

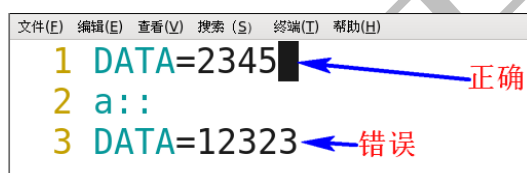
Linux 下 make 标志位的常用选项和 Unix 系统中稍有不同，下面只列出了不同部分：

- c dir** 在读取 makefile 之前改变到指定的目录 dir。
- I dir** 当包含其他 makefile 文档时，利用该选项指定搜索目录。
- h** help 文档，显示任何的 make 选项。
- w** 在处理 makefile 之前和之后，都显示工作目录。

### 1.3 makefile 变量的定义与使用

注意：变量只能在目标:符号范围之外进行定义赋值。

例如：



变量的定义格式： **ABC=**

变量的引用方式： **\$(ABC)**

变量的赋值方式：

- ① 直接赋值：ABC=1234
- ② 赋值多个值：ABC= 123 567 890
- ③ 在之前的变量基础上增加值：ABC+=789

示例图：

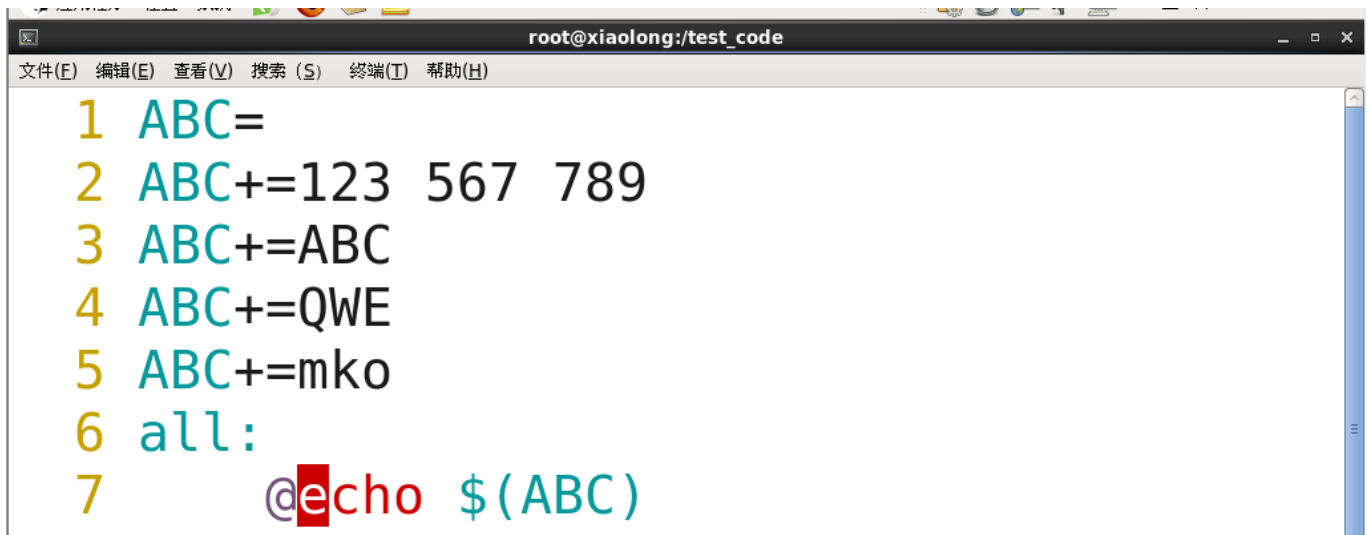


图 1-1 变量的定义与使用

## 1.4 echo 命令

通常，`make` 会把其要执行的命令行在命令执行前输出到屏幕上。当我们用 “@” 字符在命令行前，那么，这个命令将不被 `make` 显示出来，最具代表性的例子是，我们用这个功能来像屏幕显示一些信息。如：

@echo 正在编译 XXX 模块.....

当 make 执行时，会输出“正在编译 XXX 模块.....”字符串，但不会输出命令，如果没有“@”，那么，make 将输出：

```
echo 正在编译 XXX 模块.....
```

如果 make 执行时，带入 make 参数“-n”或“--just-print”，那么其只是显示命令，但不会执行命令，这个功能很有利于我们调试我们的 Makefile，看看我们书写的命令是执行起来是什么样子的或是什么顺序的。

而 make 参数 “-s” 或 “--slient” 则是全面禁止命令的显示。

-i 表示忽略 make 中出现的错误!

■ 同时输出多个值示例:

```
@echo "1234""5678""8900"
```

```
@echo "1234" "5678" "8900"
```

```
@echo "1234" $(aa) $(bb).....
```

```
@echo "1234 =$ (ppp)"
```

## ■ 输出””号

```
@echo "\"AAAAAA\""
```

输出结果:

"AAAAAA"

## 1.5 Makefile 的条件判断

### 1.5.1 ifeq 与 ifneq

使用条件判断，可以让 `make` 根据运行时的不同情况选择不同的执行分支。条件表达式可以是比较变量的值，或是比较变量和常量的值。

```
all:                #注意 ifeq 后边必须要有一个空格
ifeq (12,13)        #相等为真，执行下面代码。不相等就为假，执行 else 下边代码
    @echo "相等！"
else
    @echo "不相等！"
endif              #条件判断的结束语句
```

我们可以从上面的示例中看到三个关键字：`ifeq`、`else` 和 `endif`。`ifeq` 的意思表示条件语句的开始，并指定一个条件表达式，表达式包含两个参数，以逗号分隔，表达式以圆括号括起。`else` 表示条件表达式为假的情况。`endif` 表示一个条件语句的结束，任何一个条件表达式都应该以 `endif` 结束。**注意 ifeq 后边必须要有一个空格。其他 linux 命令需要以 TAB 键开头。**

#### 与 ifeq 相反

```
all:
ifneq (12,13)
    @echo "不相等！"
else
    @echo "相等！"
endif
```

### 1.5.2 ifdef 与 ifndef

关键字“`ifdef`”。语法是：`ifdef <variable-name>`  
如果变量`<variable-name>`的值非空，那到表达式为真。否则，表达式为假。

```
ABC=123
all:
ifdef ABC    #检测 ABC 是否定义
    @echo "已经定义！"
else
    @echo "没有定义！"
Endif
```

与 `ifdef` 刚好相反

```
ABC=123
all:
ifndef ABC   #检测 ABC 是否定义
    @echo "没有定义！"
else
```

```
@echo "已经定义！"  
  
endif
```

## 1.6 设置 Makefile 文件搜索路径

注意：vpath 和 VPATH 设置的路径只针对 Makefile 的依赖有效。对 linux 的命令无效。比如：GCC

gcc 编译依赖的.h 文件需要使用-I(大写 i) 进行指定！

### 1.6.1 VPATH

VPATH 特殊变量，可以设置 Makefile 中所有文件的搜索路径，包括依赖文件和目标文件。

变量“VPATH”的定义中，使用空格或者冒号(:)将多个目录分开。make 搜索的目录顺序，按照变量“VPATH”定义中顺序进行（当前目录永远是第一搜索目录）。

例如：

```
VPATH = src:../headers
```

它指定了两个搜索目录，“src”和“../headers”。

■ VPATH 示例：

```
VPATH=src    #设置搜索的路径  
all:123.o    #依赖条件  
    @gcc 123.o -o app
```

注意：gcc 编译时需要自己包含.h 头文件。否则会报错。比如：-I ./include -L 表示指定库路径。

### 1.6.2 vpath

vpath：关键字

它所实现的功能和上一小节提到的“VPATH”变量很类似，但是

它更为灵活。它可以为不同类型的文件（由文件名区分）指定不同的搜索目录。

```
vpath %.c ./FILE_1    #设置.c 文件的搜索路径
```

```
vpath %.h ./FILE_1    #设置.h 文件的搜索路径
```

vpath 与 VPATH 的区别在于 VPATH 指定全局的搜索路径，而 vpath 可以针对特定的文件搜索路径。vpath 命令主要有三种形式：

vpath pattern path ：符合 pattern 的文件在 path 目录搜索。

vpath pattern ：清除 pattern 指定的文件搜索路径

vpath ：清除所有文件搜索路径。

■ vpath 示例：

```
vpath %.c src  
all:123.o  
    @gcc 123.o -o app
```

注意：如果使用 makefile 的自动推导功能就不能给 GCC 指定头文件路径，编译就会报错，可将.h 和.c 放入

同一个文件下即可。

## 1.7 makefile 之间嵌套调用与参数传递

### 1.7.1 Makefile 的嵌套调用

第一层的 makefile:

```
ADDR=./addr_1    #存放 makefile 文件的路径
all:
    make -C $(ADDR)
    @echo "调用成功!"
```

第二层的 Makefile:

```
all:
    @echo "下层 Makefile 调用成功!!"
```

### 1.7.2 Makefile 之间传参传递-1

使用 export 关键字声明!

第一层 Makefile:

```
ADDR=./addr_1
export ABC+=123 456 789
all:
    make -C $(ADDR)
    @echo "调用成功!"
```

第二层 makefile:

```
all:
    @echo "下层 Makefile 调用成功!!"
    @echo $(ABC)
```

### 1.7.3 Makefile 之间传参传递 2

直接传递参数。

第一层 Makefile:

```
ADDR=./addr_1
all:
    make -C $(ADDR) ABC="123456789"
    @echo "调用成功!"
```

第二层 makefile:

```
all:
    @echo "下层 Makefile 调用成功!!"
    @echo $(ABC)
```

### 1.7.4 指定调用下层 Makefile 命令 1

第一层 Makefile:

```
ADDR=./addr_1
```

```
all:
    make -C $(ADDR) ABC="123456789" clean
    @echo "调用成功!"
```

第二层 makefile:

```
all:

    @echo "下层 Makefile 调用成功!!"

    @echo $(ABC)

clean:

    @echo "下层 clean 调用成功!!"
```

### 1.7.5 指定调用下层 Makefile 命令 2

第一层 Makefile:

```
ADDR=./addr
all:
    make -C $(ADDR) obj1 obj2
```

第二层 makefile:

```
all:
    @echo "all 命令调用成功！"

obj1:
    @echo "obj1 命令调用成功！"

obj2:
    @echo "obj2 命令调用成功！"
```

执行结果:

```
obj1 命令调用成功！
obj2 命令调用成功！
```

### 1.8 Makefile 获取 shell 命令的输出

比如命令: ls pwd

两种形式调用: 1. `pwd` 2. \$(shell pwd)

■ 调用示例 1 :

```
A+= $(shell ls)
B+= `ls`
all:
    @echo $(A)
    @echo $(B)
```

## ■ 调用示例 2:

```
ADDR=`pwd`/addr_1
all:
    @echo $(ADDR)
    make -C $(ADDR) PWD=`pwd`
```

## 1.9 自动化编译

目的：将三个.c 文件，和两个.h 文件编译一个可执行文件。

```
#include "main_1.c"
#include "main_2.h"
#include "main_2.c"
#include "main_3.h"
#include "main_3.c"
```

### 1.9.1 Makefile 文件编写示例 1

NUM 就是将要生成的目标文件。

```
NUM:main_1.o main_2.o main_3.o #依赖项—Makefile 就是一层层的查找依赖
    gcc -o NUM main_1.o main_2.o main_3.o

main_1.o:main_1.c main_2.h main_3.h
    gcc -c main_1.c

main_2.o:main_2.c main_2.h
    gcc -c main_2.c

main_3.o:main_3.c main_3.h
    gcc -c main_3.c

clear:
    rm *.o -rf
```

### 1.9.2 Makefile 文件编写示例 2

使用 Makefile 自动推导的功能

```
OBJ=main_1.o main_2.o main_3.o #变量赋值，将依赖文件赋值给 OBJ 变量
NUM: $(OBJ)
    gcc -o NUM $(OBJ)

main_1.o:main_1.c main_2.h main_3.h #生成 main_1.o 需要依赖下面的文件
main_2.o:main_2.c main_2.h #使用 Makefile 自动推导功能，省去 gcc -c main_3.c
main_3.o:main_3.c main_3.h
```



```
clear:
    rm $(OBJ) -rf
```

### 1.9.3 Makefile 文件编写示例 3

Makefile 文件自动推导功能

```
OBJ=main_1.o main_2.o main_3.o #变量赋值，将依赖文件赋值给 OBJ 变量
NUM: $(OBJ)
    gcc -o NUM $(OBJ)
$(OBJ):main_2.h main_3.h #将依赖文件合在一起，自动推导的命令
clear:
    rm $(OBJ) -rf
```

## 1.10 Makefile 的特殊符号

- include 包含其他 Makefile。语法：include <路径>

### 1.10.1 赋值符号

=	最基本的赋值语句。
:=	覆盖变量之前的值。比如：ABC=123 ABC:=897 那么 ABC 最终的值等于 897
?=	如果变量定义过，则使用之前的值，本次赋值没有效。
+=	追加变量，每个变量之间自动使用空格隔开。
%	通配符，表示匹配所有文件。
-	忽略命令的错误。比如：-rm 123.c。如果 123.c 文件不存在，rm 删除就会报错，加上-可以忽略错误。
@	加在命令前面，隐藏命令的输出
:	依赖规则定义符。格式：规则:依赖文件

### 1.10.2 自动化编译

\$@	表示目标文件。
\$<	表示依赖文件集合中的第一个依赖文件。
\$?	表示更新的依赖文件。
^	表示所有的依赖文件，去除重复的依赖文件。
+	和^符号一样，没有去重的功能。

- 加入自动化编译变量的 makefile 写法

目的：编译 1 个.h 和 2 个.c

```
CC=gcc
OBJ=main.o print.o

app:$(OBJ)
    $(CC) -o $@ $(OBJ)
```

```
%.o:%.c
```

```
$(CC) -c $< -o $@
```

 等同于 

```
$(CC) -c $<
```

 不指定生成目标名称，默认使用默认的.c 文件命名。

```
clean:
```

```
@rm $(OBJ) -fv
```

## 1.11 常用的 makefile 函数

注意：函数只能在目标之外调用。

```
1 DATA=$(shell ls -l)
2 a::
3     echo $(DATA)
```

### 1.11.1 字符串替换函数

函数原型格式：\$(subst <A>,<C>,<D>)

函数功能：将 D 中的 A 全部替换为 C

函数返回值：替换后的完整值

示例：

```
ABC=1111188888
```

```
aa=$(subst 1,A,$(ABC))
```

 注意：subst 后面必须有一个空格

```
all:
```

```
@echo "变量="$(aa)
```

输出结果：

```
[root@xiaolong 2th]# make
变量=AAAAA88888
```

### 1.11.2 去掉字符串的前后空字符串

函数原型格式：\$(strip <str>)

函数功能：将 str 字符串的前后空字符去掉。

函数返回值：替换后的完整值

示例：

```
ABC=" 12345"
```

```
DATA=$(strip $(ABC))
```

```
all:
```

```
@echo "替换前"=$(ABC)
```

```
@echo "替换后"=$(DATA)
```

输出结果：

```
[root@xiaolong 2th]# make
```

```
替换前= 12345
替换后= 12345
```

### 1.11.3 字符串查找

函数原型格式: `$(findstring <A>,<C>)`

函数功能: 在 C 数据包中查找是否有 A 数据存在

函数返回值: 查找成功返回查找到的数据, 否则返回值空字符

#### ■ 示例 1:

```
ABC=" 12345"
all:
    @echo $(findstring 5,$(ABC))
```

输出结果:

```
[root@xiaolong 2th]# make
5
```

#### ■ 示例 2:

```
ABC=12345 哈哈
all:
    @echo $(findstring 哈,$(ABC))
```

输出结果:

```
[root@xiaolong 2th]# make
哈
```

#### ■ 示例 3:

```
ABC=12345
all:
ifeq ($(findstring 5,$(ABC)),5)
    @echo "查找成功"
else
    @echo "查找失败"
endif
```

输出结果:

```
[root@xiaolong 2th]# make
查找成功
```

### 1.11.4 执行 shell 命令

格式: `$(shell <执行的 shell 命令>)`

示例:

```
1 DATA=$(shell ls -l)
2 a::
3     echo $(DATA)
```

#### 1.11.5 产生错误信息

格式: \$(error <想要打印的错误提示>)

功能: 执行该函数会立即产生一个错误, 终止 Makefile 的执行。

示例:

```
ABC="123456789"
all:
    @echo $(error $(ABC))
    @echo "hello world"
```

输出结果:

```
[root@xiaolong 2th]# make
Makefile:3: *** "123456789"。 停止。
```

#### 1.11.6 产生警告信息

格式: \$(warning <输出的警告提示信息>)

功能: 执行该函数会产生警告信息, 但是不会终止 makefile 的执行。

示例:

```
ABC="123456789"
all:
    @echo $(warning $(ABC))
    @echo "hello world"
```

输出结果:

```
[root@xiaolong 2th]# make
Makefile:3: "123456789"

hello world
```

#### 1.11.7 判断变量是否是环境变量

语法: \$(origin <变量名称>)

如果是环境变量函数返回: environment , 不是环境变量返回 undefined。

示例:

```
1 a::
2 ifeq ($(origin PATH),environment)
3     echo "OK"
4 else
5     echo "NO"
6 endif
```

## 1.12 特殊变量

### 1.12.1 CC 变量

CC 变量定义了 makefile 默认编译程序使用的编译器。如果 makefile 中不定义 CC 变量，CC 默认表示 gcc  
示例：

```
OBJ=main.o print.o
CC=arm-linux-gcc
app:$(OBJ)
    $(CC) $(OBJ) -o app
```

输出结果：

```
[root@xiaolong 2th]# make
arm-linux-gcc -c -o main.o main.c
arm-linux-gcc -c -o print.o print.c
arm-linux-gcc main.o print.o -o app
```

### 1.12.2 模式指定变量 CFLAGS

CFLAGS 变量可以指定目标在编译时加载的参数。

#### ■ 示例 1:

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

在生成.o 的时候，都会加上-c 参数。

在生成.o 时就是这样：gcc -c xxx.c

#### ■ 当使用 “%” 作为目标时，指定的变量会对所有类型的目标文件有效。

例如：%:CFLAGS=-c

#### ■ 编译多个目录下的文件时 makefile 编写方式

源码目录结构：

```
├── include
│   └── print.h
└── Makefile
```

```
└── src
    ├── main.c
    └── print.c
```

Makefile 编写方式 1:

```
VPATH=include:src
OBJ=main.o print.o
CC=gcc
INCLUDE=-I include

app:$(OBJ)
    $(CC) $(OBJ) -o app

%.o:CFLAGS = $(INCLUDE)

clean:
    @rm $(OBJ) -f
```

编译结果:

```
[root@xiaolong 2th]# make
gcc -I include -c -o main.o src/main.c
gcc -I include -c -o print.o src/print.c
gcc main.o print.o -o app
```

Makefile 编写方式 2:

```
VPATH=include:src
OBJ=main.o print.o
CC=gcc
INCLUDE=-I include
CFLAGS = $(INCLUDE) #全局指定，后面的所有编译都会加上这个参数
app:$(OBJ)
    @$(CC) $(OBJ) -o app

clean:
    @rm $(OBJ) -f
```

### 1.13.3 环境变量

- SHELL : 环境变量，表示当前所用的 shell
- CURDIR : 环境变量，表示当前目录
- MAKEFLAGS : 环境变量，存储 make 的参数信息

比如: make pwd=123 abc=888 那么 MAKEFLAGS 就等于 pwd=123 abc=888