

Makefile 使用

作者:向仔州

Makefile 变量使用.....	2
环境变量使用.....	3
通配符使用.....	4
Makefile 里面嵌套 shell 脚本语言.....	4
ifeq else 的用法, 类似(C 语言的 if …else..), ifneq, ifdef.....	5
上级目录 Makefile 编译下级多个目录的 Makefile.....	7
findstring 用法: 查找变量字符串, 返回一个状态.....	8
MAKEFLAGS 系统提前定义好的环境变量使用.....	9
wildcard 关键字使用.....	10
notdir 关键字使用.....	10
origin 使用: 告诉变量是从什么位置来.....	11
VPATH(大写)指定源文件目录来编译.....	11
vpath 关键字使用,指定编译源码的目录.....	13
include 使用.....	13
VPATH,include,wildcard,notdir 在 Makefile 项目工程当中的配合应用.....	15
subst 字符串替换.....	18
Makefile 编译下执行 shell 比如@\$(MKCONFIG) \$(@:_config=).....	18
2.我们试试 Makefile 变量值替换规则 : 冒号.....	18
addprefix 增加字符的前缀.....	19

Makefile 变量使用

```
var = "abcd"
all:
    echo $(var)
```

Makefile 变量定义和 shell 一样

```
root@ubuntu:/home/xiang/test# make
echo "abcd"
abcd
```

```
var = "abcd"
var ?= "efgh"
all:
    echo $(var)
```

?= 是同一个变量前面赋值了，后面就不能再赋值

```
root@ubuntu:/home/xiang/test# make
echo "abcd"
abcd
```

```
#var = "abcd"
var ?= "efgh"
all:
    echo $(var)
```

这就是前面没有给变量赋值

```
root@ubuntu:/home/xiang/test# make
echo "efgh"
efgh
```

```
var=abcd
var+=efgh
all:
    echo $(var)
```

+ = 就是变量 var 被赋值之后，我突然又想给 var 加点参数，但是前面给 var 赋值的参数不能被覆盖，我就用+ =

```
root@ubuntu:/home/xiang/test# make
echo abcd efgh
abcd efgh
```

+ = 这种就是我什么时候需要什么时候写。

```
objs := start.o led.o clock.o uart.o main.o
objs += lib/libc.a
```

```
objs := start.o led.o clock.o uart.o main.o lib/libc.a
#objs += lib/libc.a
```

这就是+ = 的好处

```
A=abc
B=$(A)def
all:
    echo $(B)
```

```
root@ubuntu:/home/xiang/test# make
echo abcdef
abcdef
```

```
A=abc
B=$(A)def
A=gh
all:
    echo $(B)
```

我们发现是最后一次给 A 变量赋值算作数

```
root@ubuntu:/home/xiang/test# make
echo ghdef
ghdef
```

```

1 A:=abc
2 B:=$(A)def
3 A:=gh
4 all:
5         echo $(B)

```

用`:=` A 变量就是第一次赋值作数，后面再给 A 赋值不作数

```

root@ubuntu:/home/xiang/test# make
echo abcdef
abcdef

```

所以如果变量是重复赋值的话，`= :=` 要注意使用

环境变量使用

Makefile 里定义的环境变量可以给工程中其它 Makefile 使用，用 `export` 来导出环境变量
局部变量是在自己的 Makefile 使用，所以环境变量最好用大写，而且要小心不能和工程中其他 Makefile 里面的环境变量重复。

```

1 CC=gcc
2
3 all:
4         @echo $(CC)

```

Makefile 可以从外部传入参数

```

root@ubuntu:/home/xiang/test# make
gcc

```

`@`这个符号是要求语句执行的时候不打印到屏幕上影响眼睛，但是语句还是在执行

```

root@ubuntu:/home/xiang/test# make CC=arm-linux
arm-linux
root@ubuntu:/home/xiang/test# make

```

这就是编译的时候 CC 接受参数，然后 cc 变量就是 `echo $(arm-linux)`，这也是环境变量的一种形式

`export` 导出的自定义环境变量只能在 Makefile 的指定的子 Makefile 中才生效。

```

root@ubuntu:/home/xiang/maketest# ls
Makefile1  Makefile2

```

Makefile1 为主 Makefile

Makefile2 为子 Makefile

```

root@ubuntu:/home/xiang/maketest/Makefile1# ls
Makefile

```

```
XIANGZIZHOU=xxxxzzzz
```

```
export XIANGZIZHOU
```

```
all:
```

```
    |   echo $(XIANGZIZHOU)
```

```
    |   make -f ../../Makefile2/Makefile
```

主 Makefile 导出的环境变量，必须由主 Makefile 指定的子 Makefile 才能使用

在执行完主 Makefile1 的程序后必须在最后一句用 `make -f` 指定子 Makefile2 执行 makefile

```

root@ubuntu:/home/xiang/maketest/Makefile2# ls
Makefile

```

```
all:
```

```
    |   echo "Makefile222"
```

```
    |   echo $(XIANGZIZHOU)
```

所以 `export` 必须是两个 Makefile 有父子关系才有用

这样 Makefile2 的 Makefile 程序才能获得 `export` 导出的环境变量值

通配符使用

```
root@ubuntu:/home/xiang/test# ls  
1.c 1.h 2.c 3.c Makefile
```

我创建了这多个文件

```
all: 1.c 2.c 3.c 1.h  
      @echo *.c
```

*表示当前目录下所有都是同一个后缀的文件放进来展开

```
root@ubuntu:/home/xiang/test# make  
1.c 2.c 3.c
```

你看 h 文件就不行。

```
all: 1.c 2.c 3.c 1.h  
      @echo [12].c
```

[]中括号,就是选择指定几个.c 文件的放进来展开,不像*号选择所有.c 文件

```
root@ubuntu:/home/xiang/test# make  
1.c 2.c
```

Makefile 里面嵌套 shell 脚本语言

```
var=$(shell ls)  
all:  
      echo $(var)
```

Makefile 执行 shell 语句,一定要将 shell 语句用\$()括起来

```
root@ubuntu:/home/xiang/test# ls  
1.c 1.h 1.o 2.c 3.c bin Makefile  
root@ubuntu:/home/xiang/test# make  
echo 1.c 1.h 1.o 2.c 3.c bin Makefile  
1.c 1.h 1.o 2.c 3.c bin Makefile  
root@ubuntu:/home/xiang/test#
```

```
HOST := $(shell uname -m)  
HOSTARCH := $(shell uname -m |  
           sed -e s/x86_64/i686/)  
all:  
      echo $(HOST)  
      echo $(HOSTARCH)
```

\这个是接续符号,如果你一行命令写不完,就用这个接续符号换行继续写命令

```
root@ubuntu:/home/xiang/test# make  
echo x86_64  
x86_64  
echo i686  
i686  
root@ubuntu:/home/xiang/test#
```

这就是 Makefile 的 shell 实际应用

Ifeq else 的用法，类似(C 语言的 if ...else..)， ifneq, ifdef

```
1 GCC:=ccg
2 all:
3 ifeq($(GCC),ccg)
4     echo "cccccg"
5 else
6     echo "no cccg"
7 endif
```

Ifeq(值 1, 值 2), 如果值 1 等于值 2 就执行 ifeq 下的语句, 否则执行 else 下的语句

```
root@ubuntu:/home/xiang/test# make
Makefile:3: *** missing separator. Stop.
root@ubuntu:/home/xiang/test#
```

为什么 make 出错?

```
1 GCC:=ccg
2 all:
3 ifeq $(GCC),ccg
4     echo "cccccg"
5 else
6     echo "no cccg"
7 endif
```

这里要加空格, ifeq 和括号之间用空格隔开, 这就是 makefile 奇葩之处

你看输出正确了

```
root@ubuntu:/home/xiang/test# make
echo "cccccg"
cccccg
root@ubuntu:/home/xiang/test#
```

```
1 all:
2 ifeq (ccg,ccg)
3     echo "cccccg"
4 else
5     echo "no cccg"
6 endif
```

把变量展开就是这样的

```
root@ubuntu:/home/xiang/test# make
echo "cccccg"
cccccg
```

```
1 all:
2 ifeq (gcc,ccg)
3     echo "cccccg"
4 else
5     echo "no cccg"
6 endif
```

把变量展开就是这样的

```
root@ubuntu:/home/xiang/test# make
echo "no cccg"
no cccg
```

```
1 ARCH =
2 all:
3 ifeq ($(ARCH),ccg)
4     echo "cccccg"
5 else
6     echo "no cccg"
7 endif
```

如果变量在 Makefile 里面不赋值, 那么在 make 执行后面要给变量赋值

```
root@ubuntu:/home/xiang/test# make
echo "no cccg"
no cccg
root@ubuntu:/home/xiang/test# make ARCH=ccg
echo "cccccg"
cccccg
root@ubuntu:/home/xiang/test#
```

```
var=xzz  
all:  
ifeq (,$(var))  
|    echo "xzz";  
else  
|    echo "null"  
endif
```

Ifeq , 前面没有变量就表示为空

```
var=xzz  
all:  
ifeq (,$(var))  
|    echo "xzz";  
else  
|    echo "null"  
endif
```

比如我给 var 变量赋值了, 那么 var 就不为空, 就应该执行 else

```
root@ubuntu:/home/xiang/maketest# make  
echo "null"  
null
```

```
var=  
all:  
ifeq (,$(var))  
|    echo "xzz";  
else  
|    echo "null"  
endif
```

这里 var 什么值都没有就表示为空
执行 ifeq 下面的语句

```
root@ubuntu:/home/xiang/maketest# make  
echo "xzz"  
xzz
```

Ifeq 的反作用是用 ifneq

```
2 all:  
3 ifneq (ccg,ccg)  
4     echo "cccccg"  
5 else  
6     echo "no cccg"  
7 endif
```

Ifneq 就是两边相等就返回假, 执行
else 下的代码, 两边不相等为真

```
root@ubuntu:/home/xiang/test# make  
echo "no cccg"  
no cccg
```

Ifdef, 测量一个变量是否有值

```
ARCH = xzz  
all:  
ifdef ARCH  
    echo "cccccg"  
else  
    echo "no cccg"  
endif
```

如果 ARCH 变量得到 xzz 值, 执行 ifdef
下的语句

```
root@ubuntu:/home/xiang/test# make  
echo "cccccg"  
cccccg
```

```
1 ARCH =
2
3 all:
4 ifdef ARCH
5     echo "cccccg"
6 else
7     echo "no cccg"
8 endif
```

如果 ARCH 变量没有值，执行 else 下的语句

```
root@ubuntu:/home/xiang/test# make
echo "no cccg"
no cccg
```

上级目录 Makefile 编译下级多个目录的 Makefile

```
root@ubuntu:/home/xiang/S5PV210/noOS_driver/maketest#
make1 make2 Makefile
root@ubuntu:/home/xiang/S5PV210/noOS_driver/maketest#
```

顶层目录有两个子目录 make1, make2

```
root@ubuntu:/home/xiang/S5PV210/noOS_driver/maketest/make1# ls
hello1.c  Makefile
```

make1 目录有 c 文件和 Makefile 文件

```
#include<stdio.h>

int main()
{
    printf("make1 hello1...\n");
    return 0;
}
root@ubuntu:/home/xiang/S5PV210/noOS_driver/maketest/make1# cat Makefile
all:           gcc -o hello1 hello1.c
clean:         @echo "make1 clean"
               rm hello1
root@ubuntu:/home/xiang/S5PV210/noOS_driver/maketest/make2# ls
hello2.c  Makefile
```

make2 目录有 c 文件和 Makefile 文件

```
#include<stdio.h>

int main()
{
    printf("make2 hello2...\n");
    return 0;
}
root@ubuntu:/home/xiang/S5PV210/noOS_driver/maketest/make2# cat Makefile
all:           gcc -o hello2 hello2.c
clean:         @echo make2 clean
               rm hello2
```

顶层的 Makefile 就是控制 make1, make2 目录里面的 makefile 文件执行的

```
root@ubuntu:/home/xiang/  
make1  make2  Makefile  
  
1 all:  
2 |     make -C ./make1  
3 |     make -C ./make2  
4 clean:  
5 |     make clean -C ./make1  
6 |     make clean -C ./make2
```

make -C 指定目录，
就是执行指定目录里
面的 Makefile 文件

make clean-C 指定目录，就
是执行指定目录 Makefile 文
件里面的 clean 命令

这里指定的目录是 make1, make2 。 ./就是当前目录下
的 make1, make2 目录

findstring 用法：查找变量字符串，返回一个状态

```
str="xxxxxxxxzzzzzz"  
var=$(findstring xzz,$(str))  
  
all:  
|     echo $(var)
```

要查找的字符串 xzz

传入变量，看变量里面
有没有 xzz 字符串

如果 str 变量里面有 xzz
字符串，findstring 就返回
xzz，这个返回的 xzz 字符
就是 findstring 定义的

```
str="xxxxxx[xxx]zzzz"  
var=$(findstring xzz,$(str))  
  
all:  
|     echo $(var)
```

```
root@ubuntu:  
echo xzz  
xzz
```

你看返回字符串

```
str="xxxx[xxx]11zzzz"  
var=$(findstring xzz,$(str))  
  
all:  
|     echo $(var)
```

你看 str 没有和 findstring
定义的相同字符串 xzz，所
以返回空

```
root@ubuntu:/home/xiang/maketest# make  
echo
```

```
str=xzz  
var=$(findstring xzz,$(str))  
  
all:  
|     echo $(var)
```

不用""字符串，直接给字
符值也可以

```
root@ubuntu:  
echo xzz  
xzz
```

MAKEFLAGS 系统提前定义好的环境变量使用

MAKEFLAGS 和我们自定义的环境变量不一样，MAKEFLAGS 是系统定义好的，你只能使用，所以你在自己定义环境变量名字的时候不要和 MAKEFLAGS 这些名字冲突。

需要注意的是，有两个变量，一个是 SHELL，一个是 MAKEFLAGS，这两个变量不管你是否 export，其总是要传递到子 Makefile 中

```
all:  
|     echo $(MAKEFLAGS)  
  
root@ubuntu:/home/xiang/maketest# make -s  
s
```

我 make -s，那么 s 就会传递给 MAKEFLAGS 变量

还有 make -k 也可以将 k 传递进 MAKEFLAGS

Makefile1 Makefile2 比如我一个主 Makefile1 和一个子 Makefile2

```
all:  
|     echo "makefile111"  
|     echo $(MAKEFLAGS)  
|     make -f ../../Makefile2/Makefile  
  
主 Makefile1
```

```
Makefile  
1 |  
2 all:  
3 |     echo "Makefile222"  
4 |     echo $(MAKEFLAGS)  
  
子 Makefile2
```

```
root@ubuntu:/home/xiang/maketest/Makefile1# ls  
Makefile  
root@ubuntu:/home/xiang/maketest/Makefile1# make -s  
makefile111  
s  
Makefile222  
s  
root@ubuntu:/home/xiang/maketest/Makefile1#
```

我在主 makefile 执行 make -s，那么 s 就会传给主 makefile 和子 makefile

wildcard 关键字使用

```
root@ubuntu:/home/xiang/makefilepractice# ls  
Makefile makefiletest maketop1.c maketop1.o maketop2.c maketop2.o maketop3.c maketop3.o maketop.o projectmakefile xzz
```

当前目录下有这些文件，我想把.o 为后缀的文件用变量接受进来

```
1 all:  
2 |     echo $(wildcard *.o)  
3 |  
4 |  
5 clean:
```

Makefile 用 wildcard 来搜索

wildcard 格式: wildcard *. 文件后缀

```
root@ubuntu:/home/xiang/makefilepractice# make  
echo maketop1.o maketop2.o maketop3.o maketop.o  
maketop1.o maketop2.o maketop3.o maketop.o
```

将我要的.o 文件放入变量了，其他目录文件和不是.o 的文件都将忽略掉。

wildcard 搜索指定目录下的.o 文件

```
root@ubuntu:/home/xiang/makefilepractice/xzz# ls  
xzz1.o xzz2.o
```

xzz 目录下有两个.o 文件，但是我的 Makefile 在上一级目录，看我如何用 wildcard 指定目录的

```
1 all:  
2 |     echo $(wildcard *.o ./xzz/*.o)  
3 |  
4 |  
5 clean:
```

表示 xzz 目录下所有.o 文件

wildcard 指定目录格式: wildcard *.文件后缀 指定搜索文件.o 所在的目录

```
root@ubuntu:/home/xiang/makefilepractice# make  
echo maketop1.o maketop2.o maketop3.o maketop.o ./xzz/xzz1.o ./xzz/xzz2.o  
maketop1.o maketop2.o maketop3.o maketop.o ./xzz/xzz1.o ./xzz/xzz2.o
```

我们发现除了把 xzz 目录下的.o 文件放入变量，当前 Makefile 所在目录下的.o 文件也放入了变量。所以 wildcard 不管你去搜索哪个目录的文件，当前目录下的文件也会搜索一遍。

```
root@ubuntu:/home/xiang/makefilepractice# make  
echo maketop1.o maketop2.o maketop3.o maketop.o ./xzz/xzz1.o ./xzz/xzz2.o  
maketop1.o maketop2.o maketop3.o maketop.o ./xzz/xzz1.o ./xzz/xzz2.o
```

我想把 xzz 目录路径去掉，只保留 xzz 目录下的.o 文件名放入变量

notdir 关键字使用

```
1 all:  
2 |     echo $(notdir $(wildcard *.o ./xzz/*.o))  
3 |  
4 |  
5 clean:
```

notdir 格式: notdir 变量名

```
root@ubuntu:/home/xiang/makefilepractice# make  
echo maketop1.o maketop2.o maketop3.o maketop.o xzz1.o xzz2.o  
maketop1.o maketop2.o maketop3.o maketop.o xzz1.o xzz2.o
```

将变量名里面包含的路径字符去掉，比如./xzz/xzz1.o，就是将./xzz 去掉，只保留 xzz1.o 字符串给变量

origin 使用：告诉变量是从什么位置来

```
var=$(origin V)
all: echo $(var)
```

Origin 判断 V 变量从哪里来，但是因为环境变量和 Makefile 局部都没有定义 V 变量，所以返回 undefined

```
root@ubuntu:/home/xiang/maketest# make
echo undefined
undefined
```



```
var=$(origin CC)
all: echo $(var)
```

CC 为系统自己提供的默认变量，所以返回 default

```
root@ubuntu:/home/xiang/maketest# make
echo default
default
```



```
2 var=$(origin PATH)
3
4 all: echo $(var)
```

PATH 是系统自带的环境变量，可以用在任何 Makefile，不像 export 这种导出的变量只能用在子 Makefile

```
root@ubuntu:/home/xiang/maketest# make
echo environment
environment
```

所以环境变量返回 environment


```
XIANG=xzz
var=$(origin XIANG)
all: echo $(var)
```

自己在 Makefile 定义的局部变量返回 file

```
root@ubuntu:/home/xiang/maketest# make
echo file
file
```



```
var=$(origin O)
all: echo $(var)
```

在 make 时候跟着的变量 O 传入进来，就返回 command line

```
root@ubuntu:/home/xiang/maketest# make O=xxxxxxx
echo command line
command line
```

如果 make 没有传入 O 变量，就返回 undefined

VPATH(大写)指定源文件目录来编译

多文件原始编译方法

```
root@ubuntu:/home/xiang/web_stdy_guide/makefile# ls
main.c Makefile test1 test2
```



```
test1
test1.c test1.h
```

test1 目录有 c 文件和 h 文件

```
test2
test2.c test2.h
```

test2 目录有 c 文件和 h 文件

然后 main.c 文件会去调用 test1.c 和 test2.c 文件里面的子函数

```

1 #include "./test1/test1.h"
2 #include "./test2/test2.h"
3
4 int main(void)
5 {
6     test1print();
7     test2print();
8     return 0;
9 }

```

在 main.c 文件里面还必须指定
你头文件的绝对路径

test1print();
test2print();

这是 main 文件要调用的函数

这时候 Makefile 怎么写呢？

```

1 1.编译出执行文件 main
2 main:main.o test1/test1.o test2/test2.o
3 |   gcc $^ -o $@ -I./test1 -I./test2
4
5 clean:
6 |   rm -rf *.o main

```

2.加入编译出执行文件的主文件
main.o 也就是等用于 main.c

3.还要加入 main.c
调 用 子 函 数
test1.c 和 test2.c
的绝对路径

4.而且编译的时候还要加
入-I 头文件绝对路径

这就是 makefile 原始写法

这种情况下假如 test1,test2 下面还有很多很多目录，我是不是要在 test1/....test1.c 依赖里
面加很多路径呢？岂不是手都要敲打疼

```

1 VPATH = ./test1:./test2
2 main:main.o test1.o test2.o
3 |   gcc $^ -o $@ -I./test1 -I./test2
4
5 clean:
6 |   rm -rf *.o main

```

VPATH 关键字指定 C 文件的目
录路径，冒号是如果有多个目
录，用冒号分割

这里头文件-I 路径还是要加上，因为
VPATH 关键字是告诉 Makefile 源码路
径。而不是告诉 GCC 源码路径，所以 GCC
要加上-I

```

root@ubuntu:/home/xiang/web_stdy_guide/makefile# make
cc    -c -o main.o main.c
cc    -c -o test1.o ./test1/test1.c
cc    -c -o test2.o ./test2/test2.c
gcc main.o test1.o test2.o -o main -I./test1 -I./test2
root@ubuntu:/home/xiang/web_stdy_guide/makefile# ls
main  main.c  main.o  Makefile  test1  test1.o  test2  test2.o
root@ubuntu:/home/xiang/web_stdy_guide/makefile# ./main
test111111
test222222
root@ubuntu:/home/xiang/web_stdy_guide/makefile#

```

编译成功

```

1 VPATH = ./test1:./test2
2 main:main.o test1.o test2.o
3 |   gcc $^ -o $@ -I./test1 -I./test2
4
5 clean:
6 |   rm -rf *.o main

```

这些子文件.o 怎么会在顶层当前目
录呢？这是因为我的 Makefile 依赖
文件没有指定目录

没有指定.o 生成的目录路径。

感觉变化不大啊，只不过把目录路径给了 VPATH 变量了而已。如果有多级目录岂不是还要手动在 VPATH 后面加目录路径？
确实 VPATH 只能做到这个程度

vpath 关键字使用,指定编译源码的目录

```
root@ubuntu:/home/xiang/web_stdy_guide/makefile# ls  
Makefile src
```

现在我将自己写的多个 c 文件放在 src 目录下

```
root@ubuntu:/home/xiang/web_stdy_guide/makefile/src# ls  
test1.c test2.c
```

现在我想编译 src 目录下的每个 c 文件, 第一种方法是我在 src 目录下写个 Makefile 然后编译, 第二种方法是我在 src 上级目录写个 makefile 用 vpath 关键字来包含编译

```
1 vpath %.c src  
2  
3 all: test1.o  
4 |     @echo compile test1.c  
5 %.o:%.c  
6 |     gcc -c $< -o $@
```

用 vpath 关键字指定你要编译的源文件目录,%.c 意思是只编译 src 目录的.c 文件

我的 src 目录有两个文件, 你要编译哪一个文件, 你在 all 后面指定就行

```
root@ubuntu:/home/xiang/web_stdy_guide/makefile# make  
gcc -c src/test1.c -o test1.o  
compile test1.c  
root@ubuntu:/home/xiang/web_stdy_guide/makefile# ls  
Makefile src test1.o  
root@ubuntu:/home/xiang/web_stdy_guide/makefile#
```

test1 文件编译出来了, 编译 test2 就要去 Makefile 里面把 test1.o 改成 test2.o

但是这有什么用? 我直接去 src 目录里面去用 Makefile 编译不是一样的效率吗?

include 使用

include 主要用来包含 .mk 的 Makefile 文件

我们先看看.mk 怎么编译

```
1 include A.mk  
2 all:  
3 |     echo "AAAAAAAAAAAAAAA"
```

```
root@ubuntu:/home/xiang/maketest# make -f A.mk  
echo "AAAAAAAAAAAAAAA"  
AAAAAA
```

make 用-f 来指定.mk 文件才能把.mk 文件当作 Makefile 编译

下面我创建一个主 Makefile 和一个子 Makefile(子 Makefile 都是文件名.mk)

```
root@ubuntu:/home/xiang/web_stdy_guide/makefile/makefile2# ls  
Makefile sub.mk
```

主 makefile 文件

子 makefile 文件

```
1 include sub.mk  
2  
3 all:  
4 |     echo "top Makefile"
```

主 Makefile 文件的内容

```
1  
2 all:  
3 |     echo "submakefile"
```

子 sub.mk 文件的内容

```
root@ubuntu:/home/xiang/web_stdy_guide/makefile/makefile2# make  
Makefile:4: warning: overriding commands for target `all'  
sub.mk:3: warning: ignoring old commands for target `all'  
echo "top Makefile"  
top Makefile
```

编译的时候主 Makefile 确实按照要求
去先执行了 include 指定的 mk 文件

但是多个 Makefile 文件相互调用的情况下只能有一个 all 标志，所以 Makefile 程序发现 sub.mk 里面的 all 是不合法的，但是没有退出顶层 Makefile，而是忽略 include 的 mk 文件继续向下执行，所以主 Makefile 文件的后面内容成功执行

如果是这样的话，那么我写子 mk 文件有什么意义呢？

子 mk 里面可以定义很多变量，然后主 Makefile 用 include 包含子.mk 文件的时候，会将子.mk 文件里面的内容在主 Makefile 里面展开，这样主 Makefile 就可以使用子.mk 文件里面的变量了。

```
root@ubuntu:/home/xiang/web_stdy_guide/makefile/makefile2# ls  
Makefile sub.mk test1.c
```

我建立了主 Makefile, 子 Makefile 文件 sub.mk, 和一个 c 文件

```
1 include sub.mk  
2  
3 all:  
4 |     gcc -c $(xVALUE)
```

主 Makefile 文件内容

```
1  
2 xVALUE = test1.c
```

sub.mk 文件内容

```
1 #include <stdio.h>  
2  
3 int main()  
4 {  
5     printf("test1111\n");  
6     return 0;  
7 }
```

test1.c 文件

现在我将 test1.c 文件编译成.o 文件：

- 1.首先在 sub.mk 文件里面声明了变量来代表 test1.c 字符
- 2.主 Makefile 用 include 关键字将 sub.mk 里面的所有变量获取出来。
- 3.然后主 Makefile 就可以使用 sub.mk 里面的变量了。

```
root@ubuntu:/home/xiang/web_stdy_guide/makefile/makefile2# make  
gcc -c test1.c  
root@ubuntu:/home/xiang/web_stdy_guide/makefile/makefile2# ls  
Makefile sub.mk test1.c test1.o
```

编译出了.o 文件

而且 include 不止可以包含.mk 格式的文件，还可以包含.conf 格式的文件

```
root@ubuntu:/home/xiang/  
Makefile subfile.conf  
root@ubuntu:/home/xiang/
```

```
1 include subfile.conf  
2  
3 all:  
4 |     echo $(CONFIG_XZZ)|
```

主 Makefile 文件内容

```
1  
2 CONFIG_XZZ=y
```

subfile.conf 文件内容

```
root@ubuntu:/home/xiang/web_stdy_guide/makefile/makefile2# make  
echo y  
y
```

Include 可以把.conf 文件的变量包含进主 Makefile，供其主 Makefile 使用

如果 subfile 是普通文件，include 还能包含吗？

```
root@ubuntu:/home/xiang/web_stdy_guide/makefile/makefile2# l  
Makefile subfile  
root@ubuntu:/home/xiang/web_stdy_guide/makefile/makefile2#  
把 subfile.conf 修改为普通文件 subfile  
root@ubuntu:/home/xiang/web_stdy_guide/makefile/makefile2# make  
echo y  
y
```

主 Makefile 的 include 还是可以包含，而且还是承认 CONFIG_XZZ 为一个变量。

以上就是 include 关键字的功能，让一些不经常变化或者固定的变量，写在配置文件中，然后主 Makefile 需要用这些变量的时候，就用 include 将这些配置文件包含进来。

VPATH,include,wildcard,notdir 在 Makefile 项目工程当中的配合应用

```
root@ubuntu:/home/xiang/web_stdy_guide/makefile# ls  
app debug Makefile test
```

debug 目录里面是 A 工程师设计的 c 程序，现在我要使用

test 目录里面是 B 工程师设计的 c 程序，现在我要使用

app 目录是我的程序文件，我要调用 A 工程师和 B 工程师里面的程序

```
root@ubuntu:/home/xiang/web_stdy_guide/makefile# ls  
debug.c debug.h Makefile
```

debug 目录里面的程序 Makefile
是 A 工程师写好了的

```
root@ubuntu:/home/xiang/web_stdy_guide/makefile/test# ls  
Makefile test.c test.h
```

test 目录里面的程序 Makefile
是 B 工程师写好了的

现在我们在顶层 Makefile 目录来建立两个文件，一个是 srco 文件用来存放 AB 工程师 Makefile 编译出来的.o 文件，因为我自己写的 app 主程序会去调用 debug.c 和 test.c 里面的函数，也就是间接依靠 debug.o 和 test.o 文件去调用的 debug.c 和 test.c，所以 debug.c 和 test.c 要比 app.c 文件提前编译好。

```
root@ubuntu:/home/xiang/web_stdy_guide/makefile# ls  
debug inc Makefile srco test
```

inc 目录用来存放 A,B 工程师 debug,test 程序的头文件，这样我写顶层 Makefile 的时候就去填写 inc 路径就可以了，不用去 A,B 工程师这么深的目录去找头文件路径，而且主程序 app.c 文件头文件包含也就只包含 inc 路径就可以了，不用去 A,B 工程师这么深的目录去包含头文件路径。

```

3 test.o:debug.c
4 |     gcc -c $^
5 |     echo $(xzzPATH)
6 |     cp *.o $(xzzPATH)
7 clean:
8 |     rm -rf *.o
9

```

A 工程师 debug 程序的 Makefile

A 工程师编译完成后，直接用 cp 把.o 文件复制到 xzzPATH 指定的顶层 src0 目录
xzzPATH 是顶层 Makefile 定义的路径变量

```

3 test.o:test.c
4 |     gcc -c $^
5 |     echo $(xzzPATH)
6 |     cp *.o $(xzzPATH)
7 clean:
8 |     rm -rf *.o
9

```

B 工程师 test 程序的 Makefile

B 工程师编译完成后，直接用 cp 把.o 文件复制到 xzzPATH 指定的顶层 src0 目录
xzzPATH 是顶层 Makefile 定义的路径变量

然后 A 工程师和 B 工程师把 debug.h 和 test.h 文件复制到顶层的 inc 目录

下面我用 app.c 主函数去调用 debug 和 test 程序

```

1 #include "../inc/debug.h"
2 #include "../inc/test.h"
3
4 int main(void)
5 {
6     debugprint();
7     testprint();
8     return 0;
9 }

```

主函数调用 debug 和 test 的程序，那么我们只需要包含顶层 inc 目录下的头文件就是了，因为这两个头文件是 AB 工程复刻到 inc 目录下的

调用 A,B 工程师实现的程序

下面是顶层 Makefile

```

1 xzzPATH=/home/xiang/web_stdy_guide/makefile/src0
2 export xzzPATH
3
4 INC=./inc
5 VPATH=./app:./src0
6
7 all:
8     make -C ./debug
9     make -C ./test
10    make app
11 app:app.o debug.o test.o
12     gcc $^ -o $@ -I$(INC)
13 clean:
14     rm -rf app *.o
15     make clean -C ./debug
16     make clean -C ./src0

```

root@ubuntu:/home/xiang/web_stdy_guide/makefile# make

编译的时候出现了 ld returned 1 exit status 错误

这个错误不是 Makefile 语法问题

```
root@ubuntu:/home/xiang/web_stdy_guide/makefile# ls  
app debug Makefile test
```

这个问题是因为你生成的二进制文件名是 app，但是在二进制 app 文件目录下有个名叫 app 的目录，所以冲突了。

```
root@ubuntu:/home/xiang/web_stdy_guide/makefile# ls  
debug inc Makefile srco test xzzapp
```

```
1 xzzPATH=/home/xiang/web_stdy_guide/makefile/srco  
2 export xzzPATH  
3  
4 INC=./inc  
5 VPATH=../xzzapp:./srco  
6  
7 all:  
8 |     make -C ./debug  
9 |     make -C ./test  
10 |    make app  
11 app:app.o debug.o test.o  
12 |     gcc $^ -o $@ -I$(INC)  
13 clean:  
14 |     rm -rf app *.o  
15 |     make clean -C ./debug  
16 |     make clean -C ./srco  
17 |     make clean -C ./test  
18 |
```

把 app.c 文件的目录名改成 xzzapp

Makefile 再修改下

```
make app  
make[1]: Entering directory '/home/xiang/web_stdy_guide/makefile'  
cc   -c -o app.o ./xzzapp/app.c  
gcc app.o ./srco/debug.o ./srco/test.o -o app -I./inc  
make[1]: Leaving directory '/home/xiang/web_stdy_guide/makefile'  
root@ubuntu:/home/xiang/web_stdy_guide/makefile# ls  
app app.o debug inc Makefile srco test xzzapp  
root@ubuntu:/home/xiang/web_stdy_guide/makefile# ./app  
debug print  
test print  
root@ubuntu:/home/xiang/web_stdy_guide/makefile#
```

再次 make，成功编译，成功执行。

我们下面来优化 Makefile 文件

```
1 xzzPATH=/home/xiang/web_stdy_guide/makefile/srco  
2 export xzzPATH  
3  
4 INC=./inc  
5 VPATH=../xzzapp:./srco  
6  
7 all:  
8 |     make -C ./debug  
9 |     make -C ./test  
10 |    make app  
11 app:app.o debug.o test.o  
12 |     gcc $^ -o $@ -I$(INC)  
13 clean:  
14 |     rm -rf app *.o  
15 |     make clean -C ./debug  
16 |     make clean -C ./srco  
17 |     make clean -C ./test  
18 |
```

我现在不想每次调用一个头文件，就要去 srco 目录里面去找对应的.o 文件写入依赖中，这样调用的文件多了，我这里要写一堆字符，好烦

```
1 xzzPATH=/home/xiang/web_stdy_guide/makefile/srco  
2 export xzzPATH  
3  
4  
5 INC=./inc  
6 VPATH=../xzzapp:./srco  
7 libo = $(notdir ${wildcard *.o ./srco/*.o})  
8  
9 all:  
10 |     make -C ./debug  
11 |     make -C ./test  
12 |     echo "xxxxxx"  
13 |     echo $(libo)  
14 |     make app  
15 app:app.o $(libo)  
16 |     gcc $^ -o $@ -I$(INC)  
17 clean:  
18 |     rm -rf app *.o  
19 |     make clean -C ./debug  
20 |     make clean -C ./srco  
21 |     make clean -C ./test  
22 |
```

用 wildcard 关键字去指定目录提取所有.o 文件的字符串给变量，然后用 notdir 将.o 文件路径去掉，这样 libo 变量就 = debug.o,test.o 了

将 libo 放入依赖，展开就是 app.o debug.o test.o 这样就可以编译出执行程序了

```
make[1]: Leaving directory '/home/xiang/web_stdy_guide/makefile/test'  
echo "xxxxxx"  
xxxxxx  
echo  
  
make app  
make[1]: Entering directory '/home/xiang/web_stdy_guide/makefile'  
cc   -c -o app.o ./xzzapp/app.c  
gcc app.o ./srco/debug.o ./srco/test.o -o app -I./inc  
make[1]: Leaving directory '/home/xiang/web_stdy_guide/makefile'  
root@ubuntu:/home/xiang/web_stdy_guide/makefile# ls  
app app.o debug inc Makefile srco test xzzapp  
root@ubuntu:/home/xiang/web_stdy_guide/makefile# ./app  
debug print  
test print  
root@ubuntu:/home/xiang/web_stdy_guide/makefile#
```

这样 AB 工程师想 srco 目录放库文件，我主程序调用库文件就是了。

subst 字符串替换

```
1 var="aaaabbccc"
2 OBJ=$(subst bb,xx,$(var))
3 all:
4 |     echo $(OBJ)
```

像 subst 这种命令是不放在 all 目标下执行的，因为不属于编译命令，所以放在任何地方都可以执行，就是不能放在 TAB 下

```
root@ubuntu:/home/
echo "aaaaxxxccc"
aaaaxxxccc
```

Makefile 编译下执行 shell 比如@\$(MKCONFIG) \$(@:_config=)

1. 我们先试试 Makefile 的变量调用 shell

```
root@ubuntu:/home/xiang/maketest/test# 
Makefile mkconfig.sh
root@ubuntu:/home/xiang/maketest/test# 目录下有两个文件
```

```
1 MKCONFIG=./mkconfig.sh
2 x210_config:
3 |     $(MKCONFIG)
```

这是 Makefile 执行 shell 的方法

```
root@ubuntu:/home/xiang/maketest/test# make x210_config
./mkconfig.sh
0
```

2. 我们试试 Makefile 变量值替换规则：冒号

```
var=10
all:
|     echo $(var:10=20)
|
root@ubuntu
echo 20
20
```

变量后边的 : 符号就是说将变量的 10 替换成 20

输出正确

3. 我们来试试@\$(MKCONFIG) \$(@:_config=) 替换目标字符，传参进 shell

```
1#!/bin/sh
2
3 echo $#
4 echo $1
5 echo $2
6 echo $3
```

我们的 mkconfig.sh 只输出传入的参数变量

```

root@ubuntu:~/home/xiang/maketest/test#
MKCONFIG=./mkconfig.sh
x210_config:
| $(MKCONFIG) ${@:_config=} xzz 50 arm

```

\$(@获取目标：左边的字)
 这里的_config 将 x210_config 的_config 字符串替换为空
 为什么是空，因为按照变量替换规则 \$(var:10=20) 我这个=后面没有东西

替换完成之后就是\$(MKCONFIG) x210 xzz 50 arm，这样了。将参数传入进 shell

```

root@ubuntu:/home/xiang/maketest/test# make
./mkconfig.sh x210 xzz 50 arm
4
x210
xzz
50

```

然后提取\$1 是 x210
 \$2 是 xzz，以此类推
 Shell 要求\$#就是传入 shell 字符个数，这里是 4 个参

addprefix 增加字符的前缀

```

bar=xxx
OBJS=$(bar)
all:
| echo $(OBJS)

```

我输出最简单的字符

```

root@ubuntu:/home/xiang/maketest/test# make
echo xxx
xxx

```

我现在想给字符前面在加点字符，

```

bar=zzzxxxx
OBJS=$(bar)
all:
| echo $(OBJS)

```

这样修改很普遍

```

root@ubuntu:/home/xiang/maketest/test# make
echo zzzxxxx
zzzxxxx

```

```
bar=xxx  
OBJS=$(addprefix zzz,$(bar))  
  
all:  
|     echo $(OBJS)
```

但是我想用 addprefix 来修改

addprefix 第一项写的 zzz 字符，就是加给逗号后面变量的字符串前缀
这样做有什么意义呢？

如果我们所有文件只有一个目录(比如 src 目录)，但是 src 目录里面的文件在不断的修改和增加，但是编译的时候每个文件都需要指定 src 目录，我是不是每个文件参数前面都去写 src 呢？这样好累

```
bar1=xxx  
bar2=uuu  
OBJS=$(addprefix src/,$(bar1) $(bar2))  
  
all:  
|     echo $(OBJS)
```

这就是每个文件字符串前面都加 src。
每个文件(字符串)
用空格隔离开

```
1 OBJS=$(addprefix src/,foo bar)  
2  
3 all:  
4 |     echo $(OBJS)
```

我们也可以不用变
量，直接写文件名

```
1 OBJS=$(addprefix src/,foo bar xxx zzzz uuuu)  
2  
3 all:  
4 |     echo $(OBJS)
```

可以写多个文件名，每
个文件名都加上 src 前
缀字符

```
root@ubuntu:~/home/xctang/maketest/tester$ make  
echo src/foo src/bar src/xxx src/zzzz src/uuuu  
src/foo src/bar src/xxx src/zzzz src/uuuu
```