```
// 规则：
// 目标 : 目标依赖
//      命令
/*
0、命令一般由 shell 命令（echo、ls）和编译器的一些工具（gcc、ld、ar、objcopy 等）组成，使用 tab 键缩进

1、如果你不想在 make 编译的时候打印正在执行的执行，可以在每条命令的前面加一个@

2、我们可以在 Makefile 中定义一个变量 val，使用使用 $(val) 或 ${val} 的形式去引用它

3、条件赋值：?=
    条件赋值是指一个变量如果没有被定义过，就直接给它赋值；如果之前被定义过，那么这条赋值语句就什么都不做

4、追加赋值：+=
    追加赋值是指一个变量，以前已经被赋值，现在想给它增加新的值，此时可以使用+=追加赋值

    如：OBJS = hello.o
        OBJS = module.o
    等价于： OBJS = hello.o module.o

5、立即变量和延迟变量是按展开时间来划分的
    立即变量使用 := 操作符进行赋值，在解析阶段就直接展开了
    延迟变量则是使用 = 操作符进行赋值，等到实际使用这个变量时才展开，获得其真正的值

6、自动变量是局部变量，作用域范围在当前的规则内
常见：
    $@：目标
    $^：所有目标依赖
    $<：目标依赖列表中的第一个依赖
    $?：所有目标依赖中被修改过的文件
```

不常见：

$%:　　当规则的目标是一个静态库文件时，$%代表静态库的一个成员名

$+:　　类似$^，但是保留了依赖文件中重复出现的文件

$*:　　在模式匹配和静态模式规则中，代表目标模式中%的部分。比如 hello.c，当匹配模式为%.c 时，$*表示 hello

$(@D):　表示目标文件的目录部分

$(@F):　表示目标文件的文件名部分

$(*D):　在模式匹配中，表示目标模式中%的目录部分

$(*F):　在模式匹配中，表示目标模式中%的文件名部分

-: :　　告诉 make 在编译时忽略所有的错误

@: :　　告诉 make 在执行命令前不要显示命令

在 Linux 系统中，波浪线表示 HOME 目录，以波浪线表示的文件名也表示特殊的含义

以我当前使用的 Ubuntu 为例，登录用户名为 wit，则

~：表示/home/wit 这个目录

~/.bashrc：表示/home/wit/.bashrc 文件

~wit：表示用户 wit 的 HOME 目录，即/home/wit

~root：表示用户 root 的 HOME 目录，即/root

~wit/.bashrc：表示/home/wit/.bashrc 文件

~root/.bashrc：表示/root/.bashrc 文件

7、变量替换

```
.PHONY: all
SRC := main.c sub.c
OBJ := $(SRC:.c=.o)
all:
    @echo "SRC = $(SRC)"
    @echo "OBJ = $(OBJ)"
```

执行 make 命令，运行结果为：

```
SRC = main.c sub.c
OBJ = main.o sub.o
```

或者，使用匹配符%匹配变量

```
.PHONY: all
SRC := main.c sub.c
OBJ := $(SRC:%.c=%.o)
all:
    @echo "SRC = $(SRC)"
    @echo "OBJ = $(OBJ)"
```

执行 make 命令，运行结果为：

```
SRC = main.c sub.c
OBJ = main.o sub.o
```

8、环境变量

make 在解析 Makefile 中还会引入一些系统环境变量，如编译参数 CFLAGS、SHELL、MAKE 等

这些变量在 make 开始运行时被载入到 Makefile 文件中，因为是全局性的系统环境变量，所以这些变量对所有的 Makefile 都有效

9、通过 export 传递变量

在 shell 环境下使用 export 命令，就相当于将变量声明为系统环境变量（全局变量？）

10、递归执行，make 通过 -C subdir 参数，会分别到各个子目录下去执行，

解析各个子目录下的 Makefile 并运行，遍历完所有的子目录，make 最后才会退出

```
.PHONY:all
all:
    @echo "make start"
    make -C subdir1
    make -C subdir2
    make -C subdir3
    @echo "make done"
```

11、递归传递变量

在顶层目录的主 Makefile 中，使用 export 声明需要传递给子目录下 makefile 的变量

12、override 指示符

如果不希望在命令行指定的变量值替代在 Makefile 中的原来定义，

那么我们可以在 Makefile 中使用指示符 override 对这个变量进行声明

13、条件判断

在 Makefile 中，可以使用 ifeq、ifneq、ifdef、ifndef 等关键字来进行条件判断

ifeq 关键字用来判断两个参数是够相等

ifdef 关键字用来判断一个变量是否已经定义

```
//判断变量 mode 的值是否为“debug”，if true 执行 command1
ifeq ($(mode),debug)
    command1
else
    command2
endif

//判断变量 mode 的值是否不为空字符，if true（如果 mode 有赋值） 执行 command1
ifneq ($(mode),)
    command1
else
    command2
endif

//判断 mode 是否已定义，if true 执行 command1
ifdef mode
    command1
else
    command2
endif

//判断 mode 是否尚未定义，if true 执行 command1
ifndef mode
```

```
        command1
    else
        command2
    endif
```

14、函数

GNU make 提供了大量的函数用来处理文件名、变量、文本和命令

函数主要分为两类：make 内嵌函数和用户自定义函数

对于 GNU make 内嵌的函数，直接引用就可以了

对于用户自定义的函数，要通过 make 的 call 函数来间接调用

函数和参数列表之间要用空格隔开，多个参数之间使用逗号隔开

如果在参数中引用了变量，变量的引用建议和函数引用使用统一格式：

要么是一对小括号，要么是一对大括号

自定义函数：

用户自定义函以 define 开头，endef 结束，给函数传递的参数在函数中使用$(0)、$(1)引用，

分别表示第 1 个参数、第 2 个参数…

```
define func
    @echo "pram1 = $(0)"
    @echo "pram2 = $(1)"
endef

all :
    $(call func, hello makefile)
```

call 函数是唯一一个可以用来创建新的参数化的函数

```
$(call <expression>,<parm1>,<parm2>,<parm3>...)
```

15、makefile 函数之文本处理函数

扩展通配符：wildcard

如：获取某个目录下所有的 C 文件列表赋值给 SRC

```
SRC  = $(wildcard *.c)
```

字符串替换：subst

如：将目录下的所有 c 文件的名称 xx.c 转换为 xx.o

```
SRC  = $(wildcard *.c)
OBJ  = $(subst .c,.o,$(SRC))
```

如：

```
@echo $(subst banana, apple, "banana is good, I like banana")
```

模式替换：patsubst

如：将目录下的所有 c 文件的名称 xx.c 转换为 xx.o

相比 subst，使用 patsubst 会更加方便

```
SRC  = $(wildcard *.c)
OBJ  = $(patsubst %.c, %.o, $(SRC))
```

去空格:strip

如：

```
.PHONY: all
STR =     hello a    b   c
STRIP_STR = $(strip $(STR))
all:
    @echo "STR = $(STR)"
    @echo "STRIP_STR = $(STRIP_STR)"
```

查找字符串：findstring

如：在 STR 中查找“hello”再赋给 FIND

```
.PHONY: all
STR =     hello a    b   c
FIND = $(findstring hello, $(STR))
all:
```

```
        @echo "STR = $(STR)"
        @echo "FIND = $(FIND)"
```

过滤字符串：filter

如：滤掉 FILE 中，除 .c 外的文件

```
    .PHONY: all
    FILE = a.c b.h c.s d.cpp
    SRC = $(filter %.c, $(FILE))
    all:
        @echo "FILE = $(FILE)"
        @echo "SRC = $(SRC)"
```

反过滤：filter-out

如：去掉 FILE 中，所有的 .c 文件

```
    .PHONY: all
    FILE = a.c b.h c.s d.cpp
    SRC = $(filter-out %.c, $(FILE))
    all:
        @echo "FILE = $(FILE)"
        @echo "SRC = $(SRC)"
```

单词排序：sort

如：对字符串 LIST 中的单词以首字母为准进行排序

```
    .PHONY: all
    LIST = banana pear apple peach orange
    SRC = $(sort $(FILE))
    all:
        @echo "FILE = $(FILE)"
        @echo "SRC = $(SRC)"
```

取单词：word

如：取出 LIST 中的某一项

```
    .PHONY: all
    LIST = banana pear apple peach orange
```

```
word1 = $(word 1, $(LIST))
word2 = $(word 2, $(LIST))
word3 = $(word 3, $(LIST))
word4 = $(word 4, $(LIST))
word5 = $(word 5, $(LIST))
word6 = $(word 6, $(LIST))
all:
    @echo "word1 = $(word1)"
    @echo "word2 = $(word2)"
    @echo "word3 = $(word3)"
    @echo "word4 = $(word4)"
    @echo "word5 = $(word5)"
    @echo "word6 = $(word6)"
```

取字串:wordlist

如：将字符串 LIST 中的前三个单词赋值给 sub_list

```
.PHONY: all
LIST = banana pear apple peach orange
sub_list = $(wordlist 1, 3, $(LIST))
all:
    @echo "LIST = $(LIST)"
    @echo "sub_list = $(sub_list)"
```

统计单词数目：words

如：LIST 中单词的个数

```
.PHONY: all
LIST = banana pear apple peach orange
all:
    @echo "LIST = $(LIST)"
    @echo "LIST len = $(words $(LIST))
```

取首个单词：firstword

如：

```
.PHONY: all
LIST = banana pear apple peach orange
all:
```

```
        @echo "LIST = $(LIST)"
        @echo "first word = $(firstword $(LIST))"
```

16、文件名处理函数

取路径名的目录：dir

$(dir NAMES…)

取出各个文件路径名中的目录部分并返回

取文件名：notdir

$(notdir NAMES…)

从一个文件路径名中去取文件名，而不是目录

取文件名后缀：suffix

$(suffix NAMES…)

文件名的后缀是文件名中以点号 . 开始（包括点号）的部分

若文件名没有后缀， suffix 函数则返回空

取文件名前缀：basename

$(basename NAMES…)

basename 函数返回最后一个点号之前的文件名（包括文件目录）部分；

如果一个文件名没有前缀，函数返回空字符串

给文件名加后缀：addsuffix

$(addsuffix SUFFIX, NAMES…)

给文件列表中的每个文件名添加后缀 SUFFIX

给文件名加前缀：addprefix

$(addprefix PREFIX, NAMES…)

给文件列表中的每个文件名添加前缀 PREFIX

单词连接：join

$(join LIST1,LIST2)

将字符串 LIST1 和字符串 LIST2 的各个对应单词依次连接

如：

```
    .PHONY: all
    LIST = /as/df/apple.c /we/er/aim.c /admin/usr/lnn.h
    P = echo
    all:
            @$(P) "LIST = $(LIST)"
```

```
	@$(P) "basename = $(basename $(LIST))"
	@$(P) "suffix = $(suffix $(LIST))"
	@$(P) "dir = $(dir $(LIST))"
	@$(P) "notdir = $(notdir $(LIST))"
	@$(P) "addsuffix = $(addsuffix .m, $(basename $(LIST)))"
	@$(P) "addprefix = $(addprefix _t,$(notdir $(LIST)))"
	@$(P) "join = $(join $(basename $(LIST)),".a .b .c")"
```

17、其他常用函数

foreach 函数

在 makefile 中做一些循环或者遍历操作

$(foreach VAR,LIST,TEXT)

把 LIST 中使用空格分割的单词依次取出并赋值给变量 VAR，然后执行 TEXT 表达式

重复这个过程，直到遍历完 LIST 中的最后一个单词。函数的返回值是 TEXT 多次计算的结果

if 函数

实现条件判断的功能，类似于 ifeq 关键字

$(if CONDITION,THEN-PART)

$(if CONDITION,THEN-PART[,ELSE-PART])

if condition 为真（非空），执行 then-part ,否则执行 else-part

```
.PHONY: all
install_path = $(if $(install_path), $(install_path), /usr/local)
all:
    @echo "install_path = $(install_path)"
```

origin 函数

$(origin <variable>)

origin 函数的作用就是告诉你，你所关注的一个变量是从哪里来的

常见的返回值：

default：变量是一个默认的定义，比如 CC 这个变量

file：这个变量被定义在 Makefile 中

command line：这个变量是被命令行定义的

override：这个变量是被 override 指示符重新定义过的

automatic：一个命令运行中的自动化变量

```
        shell 函数
        在 Makefile 中运行 shell 命令
        shell 函数的参数是 shell 命令，
        shell 命令的运行结果即为 shell 函数的返回值
        error 和 warning
        都用来给用户提示信息
        $(error TEXT…)
        只有包含 error 函数引用的命令执行时，
        或者包含这个函数的定义变量被展开时，才会提示错误信息 TEXT 并终止 make 的运行
        $(warning TEXT…)
        warning 函数不会终止 make 的运行，make 会继续运行下去
*/
```