**参考文档：**<https://www.jianshu.com/p/7e49071f9e11>

### Makefile 的基本工作原理

makefile默认执行第一个目标

make XX的时候，Makefile会自动执行xx这个目标下面的命令语句。

当我们make xx的时候，是否执行命令是取决于依赖的。依赖如果成立就会执行命令，否则不执行。

### Makefile 的工作方式

读入主Makefile（主Makefile中可以引用其他Makefile）

读入被include的其他Makefile

初始化文件中的变量

推导隐晦规则, 并分析所有规则

为所有的目标文件创建依赖关系链

根据依赖关系, 决定哪些目标要重新生成

执行生成命令

### Makefile的基本语法知识

#### Makefile 中定义和使用变量

**定义变量（=or:=）**

其中 = 和 := 的区别在于, := 只能使用前面定义好的变量, = 可以使用后面定义的变量

#此行为注释行

#测试变量定义 =

OBJS2 = $(OBJS1) c.c

OBJS1 := $(OBJS0) b.c

OBJS0 = a.c

all:

    @echo $(OBJS2)

**输入 make test1**

**输出结果 b.c c.c**

**变量替换**

#变量替换

SRCS := a.c b.c c.c

OBJS := $(SRCS:%.c=%.o)

test2:

    @echo "SRCS: " $(SRCS)

    @echo "OBJS: " $(OBJS)

**输入 make test2**

**输出结果 SRCS: a.c b.c c.c**

**OBJS: a.o b.o c.o**

**变量追加**

SRCS1 := a.c b.c c.c

SRCS1 += d.c

test3:

    @echo "SRCS: " $(SRCS1)

**输入 make test3**

**输出结果 SRCS: a.c b.c c.c d.c**

**变量覆盖**

#变量覆盖

SRCS2 := d.c

override SRCS2 := a.c b.c c.c

test3:

    @echo "SRCS: " $(SRCS2)

**输入make test4**

**输出结果 SRCS: a.c b.c c.c**

**目标变量**

作用：使变量的作用域仅限于这个目标(target)

语法：<target ...> : <variable-assignment>

test5: TARGET1-SRCS := d.c

test5:

    @echo "SRCS: " $(TARGET1-SRCS)

test6:

    @echo "SRCS: " $(TARGET1-SRCS)

**输入：make test5**

**输出结果：SRCS: d.c**

**输入：make test6**

**输出结果：SRCS:**

#### Makefile 命令前缀

Makefile 中书写shell命令时可以加2种前缀 @ 和 -, 或者不用前缀

* 不用前缀 ：输出执行的命令以及命令执行的结果, 出错的话停止执行
* 前缀 @ ：只输出命令执行的结果, 出错的话停止执行
* 前缀 - ：命令执行有错的话, 忽略错误, 继续执行
* test7:
* echo "没有前缀"
* cat this\_file\_not\_exist
* echo "错误之后的命令"
* test8:
* @echo "前缀@"
* @cat this\_file\_not\_exist
* @echo "错误之后的命令"
* test9:
* -echo "前缀-"
* -cat this\_file\_not\_exist
* -echo "错误之后的命令"

**输入make test7:**

**输出结果echo "没有前缀"**

**没有前缀**

**cat this\_file\_not\_exist**

**cat: this\_file\_not\_exist: 没有那个文件或目录**

**Makefile:35: recipe for target 'test7' failed**

**make: \*\*\* [test7] Error 1**

**输入make test8:**

**输出结果：前缀@**

**cat: this\_file\_not\_exist: 没有那个文件或目录**

**Makefile:35: recipe for target 'test8' failed**

**make: \*\*\* [test8] Error 1**

**输入make test9:**

**输出结果：echo "前缀-"**

**前缀-**

**cat this\_file\_not\_exist**

**cat: this\_file\_not\_exist: 没有那个文件或目录**

**Makefile:43: recipe for target 'test9' failed**

**make: [test9] Error 1 (已忽略）**

**echo "错误之后的命令"**

**错误之后的命令**

#### Makefile 的伪目标（.PHONY）

(1)伪目标意思是这个目标本身不代表一个文件，执行这个目标不是为了得到某个文件或东西，而是单纯为了执行这个目标下面的命令。

(2)伪目标一般都没有依赖，因为执行伪目标就是为了执行目标下面的命令。既然一定要执行命令了那就不必加依赖，因为不加依赖意思就是无条件执行。

(3)伪目标可以直接写，不影响使用；但是有时候为了明确声明这个目标是伪目标会在伪目标的前面用.PHONY来明确声明它是伪目标。

典型的伪目标是 Makefile 中用来清理编译过程中中间文件的 clean 伪目标, 一般格式如下:

.PHONY: clean   <-- 这句没有也行, 但是最好加上

clean:

    -rm -f \*.o

#### 引用其他的 Makefile

语法: include <filename> (filename 可以包含通配符和路径)

./src/Makefile:

src-test1:

    @echo "other makefile begin"

    @echo "other makefile end"

./makefile

include ./src/Makefile

test10:

    @echo "主makefile begin"

    @make src-test1

    @echo "主makefile end"

输入：make test10

输出结果：

主makefile begin

make[1]: 进入目录“/home/liuxiang/WorkeSpace/Demo\_Makefile/Demo\_Makefile”

other makefile begin

other makefile end

make[1]: 离开目录“/home/liuxiang/WorkeSpace/Demo\_Makefile/Demo\_Makefile”

主makefile end

#### make 参数介绍

make 的参数有很多, 可以通过 make -h 去查看

#### 查看C文件的依赖关系

gcc – MM main.c

#### Makefile 隐含规则

编译C时，<n>.o 的目标会自动推导为 <n>.c

# Makefile 中  
main : main.o  
gcc -o main main.o

#会自动变为:

main : main.o

gcc -o main main.o

main.o: main.c <-- main.o 这个目标是隐含生成的

gcc -c main.c

**隐含规则中的 命令变量 和 命令**参数变量

变量 名含义

RM rm -f

AR ar

CC cc

CXX g++

##### 命令参变量

##### 变量名 含义

##### ARFLAGS AR命令的参数

##### CFLAGS C语言编译器的参数

##### CXXFLAGS C++语言编译器的参数数变量

#### 自动变量

在有些情况下文件集合中文件非常多，描述的时候很麻烦，所以我们Makefile就用一些特殊的符号来替代符合某种条件的文件集，这就形成了自动变量。

自动变量的含义：预定义的特殊意义的符号。就类似于C语言编译器中预制的那些宏**FILE**一样。

Makefile 中很多时候通过自动变量来简化书写, 各个自动变量的含义如下:

自动变量 含义

$@ 目标集合

$% 当目标是函数库文件时, 表示其中的目标文件名

$< 第一个依赖目标. 如果依赖目标是多个, 逐个表示依赖目标

$? 比目标新的依赖目标的集合

$^ 所有依赖目标的集合, 会去除重复的依赖目标

$+ 所有依赖目标的集合, 不会去除重复的依赖目标

$\* 这个是GNU make特有的, 其它的make不一定支持

#### Makefile 的文件名

Makefile的文件名合法的一般有2个：Makefile或者makefile，也可以是GNUMakefile

#### Makefile 中的注释用#

Makefile中注释使用#，和shell一样。

#### 命令前面的@用来静默执行

(1)在makefile的命令行中前面的@表示静默执行。  
(2)Makefile中默认情况下在执行一行命令前会先把这行命令给打印出来，然后再执行这行命令。  
(3)如果你不想看到命令本身，只想看到命令执行就静默执行即可。

#### Makefile 中几种变量赋值运算符

(1)= 最简单的赋值

(2):= 一般也是赋值  
以上这两个大部分情况下效果是一样的，但是有时候不一样。

用=赋值的变量，在被解析时他的值取决于最后一次赋值时的值，所以你看变量引用的值时不能只往前面看，还要往后面看。  
用:=来赋值的，则是就地直接解析，只用往前看即可。

(3)?= 如果变量前面并没有赋值过则执行这条赋值，如果前面已经赋值过了则本行被忽略。（实验可以看出：所谓的没有赋值过其实就是这个变量没有被定义过）

(4)+= 用来给一个已经赋值的变量接续赋值，意思就是把这次的值加到原来的值的后面，有点类似于strcat。（在shell makefile等文件中，可以认为所有变量都是字符串，+=就相当于给字符串stcat接续内容）（注意一个细节，+=续接的内容和原来的内容之间会自动加一个空格隔开）

#### Makefile 的环境变量

1. makefile中用export导出的就是环境变量。一般情况下要求环境变量名用大写，普通变量名用小写。
2. 环境变量和普通变量不同，可以这样理解：环境变量类似于整个工程中所有Makefile之间可以共享的全局变量，而普通变量只是当前本Makefile中使用的局部变量。所以要注意：定义了一个环境变量会影响到工程中别的Makefile文件，因此要小心
3. Makefile中可能有一些环境变量可能是makefile本身自己定义的内部的环境变量或者是当前的执行环境提供的环境变量（譬如我们在make执行时给makefile传参。make CC=arm-linux-gcc，其实就是给当前Makefile传了一个环境变量CC，值是arm-linux-gcc。我们在make时给makefile传的环境变量值优先级最高的，可以覆盖makefile中的赋值）。这就好像C语言中编译器预定义的宏\_*LINE\_* \_*FUNCTION\_*等一样。

#### Makefile 中使用通配符

\* 若干个任意字符

? 1个任意字符

[] 将[]中的字符依次去和外面的结合匹配

%， 表示任意多个字符，和\*很相似，但是%一般只用于规则描述中，又叫做规则通配符。

#### Makefile 高级语法

### 嵌套Makefile

使用export可以传递变量

主Makefile

include ./src/Makefile

export VALUE := a.c b.c c.c

test11:

    @echo "主makefile begin"

    @make src-test1

    @echo "主makefile end"

./src/Makefile

src-test1:

    @echo "other makefile begin"

    @echo $(VALUE)

    @echo "other makefile end"

输入make test11

输出结果：

主makefile begin

make[1]: 进入目录“/home/liuxiang/WorkeSpace/Demo\_Makefile/Demo\_Makefile”

other makefile begin

a.c b.c c.c

other makefile end

make[1]: 离开目录“/home/liuxiang/WorkeSpace/Demo\_Makefile/Demo\_Makefile”

主makefile end

#### 定义命令包

命令包有点像是个函数, 将连续的相同的命令合成一条, 减少 Makefile 中的代码量, 便于以后维护.

define run-hello-makefile

@echo -n "Hello"

@echo " Makefile!"

@echo "这里可以执行多条 Shell 命令!"

endef

test12:

    $(run-hello-makefile)

输入make test12

输出：

Hello Makefile!

这里可以执行多条 Shell 命令!

#### 条件判断

条件判断的关键字主要有 ifeq ifneq ifdef ifndef

test13:

ifeq ("aa", "bb")

    @echo "equal"

else

    @echo "not equal"

endif

输入：make test13

输出：not equal

#### Makefile 中的函数

Makefile 中自带了一些函数, 利用这些函数可以简化 Makefile 的编写.

函数调用

$(<function> <arguments>)

# 或者

${<function> <arguments>}

* <function> 是函数名
* <arguments> 是函数参数

##### 字符串函数

字符串替换函数: $(subst <from>,<to>,<text>)

功能: 把字符串<text> 中的 <from> 替换为 <to>

返回: 替换过的字符串

#字符串替换函数

str1 := string

str2 := $(subst s,S,$(str1))

test14:

    @echo $(str2)

输入：make test14

输出：String

**去空格函数: $(strip <string>)**

功能: 去掉 <string> 字符串中开头和结尾的空字符

返回: 被去掉空格的字符串值

**查找字符串函数: $(findstring <find>,<in>)**

功能: 在字符串 <in> 中查找 <find> 字符串

返回: 如果找到, 返回 <find> 字符串, 否则返回空字符串

**过滤函数: $(filter <pattern...>,<text>)**

功能: 以 <pattern> 模式过滤字符串 <text>, *保留* 符合模式 <pattern> 的单词, 可以有多个模式

返回: 符合模式 <pattern> 的字符串

test15:

    @echo $(filter %.o %.a,a.c b.o c.a)

输入：make test15

输出结果：b.o c.a

**反过滤函数: $(filter-out <pattern...>,<text>)**

功能: 以 <pattern> 模式过滤字符串 <text>, *去除* 符合模式 <pattern> 的单词, 可以有多个模式

返回: 不符合模式 <pattern> 的字符串

排序函数: $(sort <lis\t>)

功能: 给字符串 <list> 中的单词排序 (升序)

返回: 排序后的字符串

test16:

    @echo $(sort bac abc acb cab)

输入: make test16

输出结果：abc acb bac cab

**取单词函数: $(word <n>,<text>)**

功能: 取字符串 <text> 中的 第<n>个单词 (n从1开始)

返回: <text> 中的第<n>个单词, 如果<n> 比 <text> 中单词个数要大, 则返回空字符串

**取单词串函数: $(wordlist <s>,<e>,<text>)**

功能: 从字符串<text>中取从<s>开始到<e>的单词串. <s>和<e>是一个数字.

返回: 从<s>到<e>的字符串

**单词个数统计函数: $(words <text>)**

功能: 统计字符串 <text> 中单词的个数

返回: 单词个数

**首单词函数: $(firstword <text>)**

功能: 取字符串 <text> 中的第一个单词

返回: 字符串 <text> 中的第一个单词

**取目录函数: $(dir <names...>)**

功能: 从文件名序列 <names> 中取出目录部分

返回: 文件名序列 <names> 中的目录部分

**取文件函数: $(notdir <names...>)**

功能: 从文件名序列 <names> 中取出非目录部分

返回: 文件名序列 <names> 中的非目录部分

**取后缀函数: $(suffix <names...>)**

功能: 从文件名序列 <names> 中取出各个文件名的后缀

返回: 文件名序列 <names> 中各个文件名的后缀, 没有后缀则返回空字符串

**取前缀函数: $(basename <names...>)**

功能: 从文件名序列 <names> 中取出各个文件名的前缀

返回: 文件名序列 <names> 中各个文件名的前缀, 没有前缀则返回空字符串

**加后缀函数: $(addsuffix <suffix>,<names...>)**

功能: 把后缀 <suffix> 加到 <names> 中的每个单词后面

返回: 加过后缀的文件名序列

加前缀函数: $(addprefix <prefix>,<names...>)

功能: 把前缀 <prefix> 加到 <names> 中的每个单词前面

返回: 加过前缀的文件名序列

**连接函数: $(join <list1>,<list2>)**

功能: <list2> 中对应的单词加到 <list1> 后面

返回: 连接后的字符串

##### foreach函数

targets := a b c d

objects := $(foreach i,$(targets),$(i).o)

test17:

    @echo $(targets)

    @echo $(objects)

输入：mak额test17

输出结果：

a b c d

a.o b.o c.o d.o

##### origin - 判断变量的来源函数

语法:

$(origin <variable>)

返回值有如下类型:

类型 含义

undefined <variable> 没有定义过

default <variable> 是个默认的定义, 比如 CC 变量

environment <variable> 是个环境变量, 并且 make时没有使用 -e 参数

file <variable> 定义在Makefile中

command line <variable> 定义在命令行中

override <variable> 被 override 重新定义过

automatic <variable> 是自动化变量

val-in-file := test-file

override val-override := test-override

test18:

    @echo $(origin not-define)    # not-define 没有定义

    @echo $(origin CC)            # CC 是Makefile默认定义的变量

    @echo $(origin PATH)         # PATH 是 bash 环境变量

    @echo $(origin val-in-file)    # 此Makefile中定义的变量

    @echo $(origin val-in-cmd)    # 这个变量会加在 make 的参数中

    @echo $(origin val-override) # 此Makefile中定义的override变量

    @echo $(origin @)             # 自动变量, 具体前面的介绍

输入：make test18 val-in-cmd=val-cmd

输出结果：

undefined

default

environment

file

command line

override

automatic

##### shell函数

语法:

$(shell <shell command>)

它的作用就是执行一个shell命令, 并将shell命令的结果作为函数的返回.

作用和 <shell command> 一样, ` 是反引号

##### make 控制函数

产生一个致命错误: $(error <text ...>)

error:

    $(error there is an error!)

    @echo "这里不会执行!"

输入：make error

输出结果：Makefile:102: \*\*\* there is an error!。 停止

功能: 输出错误信息, 停止Makefile的运行

输出警告: $(warning <text ...>)

功能: 输出警告信息, Makefile继续运行

warning:

    $(warning there is an warning!)

    @echo "这里会执行!"

输入：make warning

输出结果：

Makefile:106: there is an warning!

这里会执行!

## Makefile中一些GNU约定俗成的伪目标

如果有过在Linux上, 从源码安装软件的经历的话, 就会对 make clean, make install 比较熟悉.

像 clean, install 这些伪目标, 广为人知, 不用解释就大家知道是什么意思了.

下面列举一些常用的伪目标, 如果在自己项目的Makefile合理使用这些伪目标的话, 可以让我们自己的Makefile看起来更专业

伪目标 含义

all 所有目标的目标，其功能一般是编译所有的目标

clean 删除所有被make创建的文件

install 安装已编译好的程序，其实就是把目标可执行文件拷贝到指定的目录中去

print 列出改变过的源文件

tar 把源程序打包备份. 也就是一个tar文件

dist 创建一个压缩文件, 一般是把tar文件压成Z文件. 或是gz文件

TAGS 更新所有的目标, 以备完整地重编译使用

check 或 test 一般用来测试makefile的流程