<https://www.cnblogs.com/wang_yb/p/3990952.html>

## ****1. Makefile 简介****

Makefile 是和 make 命令一起配合使用的.

很多大型项目的编译都是通过 Makefile 来组织的, 如果没有 Makefile, 那很多项目中各种库和代码之间的依赖关系不知会多复杂.

Makefile的组织流程的能力如此之强, 不仅可以用来编译项目, 还可以用来组织我们平时的一些日常操作. 这个需要大家发挥自己的想象力.

本篇博客是基于 [{精华} 跟我一起写 Makefile](http://bbs.chinaunix.net/forum.php?mod=viewthread&tid=408225) 而整理的, 有些删减, 追加了一些示例.

非常感谢 gunguymadman\_cu 提供如此详尽的Makefile介绍, 这正是我一直寻找的Makefile中文文档.

### ****1.1 Makefile 主要的 5个部分 (显示规则, 隐晦规则, 变量定义, 文件指示, 注释)****

Makefile基本格式如下:

target ... : prerequisites ...

command

...

...

其中,

* target        - 目标文件, 可以是 Object File, 也可以是可执行文件
* prerequisites - 生成 target 所需要的文件或者目标
* command       - make需要执行的命令 (任意的shell命令), Makefile中的命令必须以 [tab] 开头

1. **显示规则 :: 说明如何生成一个或多个目标文件(包括 生成的文件, 文件的依赖文件, 生成的命令)**
2. **隐晦规则 :: make的自动推导功能所执行的规则**
3. **变量定义 :: Makefile中定义的变量**
4. **文件指示 :: Makefile中引用其他Makefile; 指定Makefile中有效部分; 定义一个多行命令**
5. **注释     :: Makefile只有行注释 "#", 如果要使用或者输出"#"字符, 需要进行转义, "\#"**

### ****1.2 GNU make 的工作方式****

1. 读入主Makefile (主Makefile中可以引用其他Makefile)
2. 读入被include的其他Makefile
3. 初始化文件中的变量
4. 推导隐晦规则, 并分析所有规则
5. 为所有的目标文件创建依赖关系链
6. 根据依赖关系, 决定哪些目标要重新生成
7. 执行生成命令

## ****2. Makefile 初级语法****

### ****2.1 Makefile 规则****

#### ****2.1.1 规则语法****

规则主要有2部分: 依赖关系 和 生成目标的方法.

语法有以下2种:

target ... : prerequisites ...

command

...

或者

target ... : prerequisites ; command

command

...

**\*注\*** command太长, 可以用 "\" 作为换行符

#### ****2.1.2 规则中的通配符****

* \*     :: 表示任意一个或多个字符
* ?     :: 表示任意一个字符
* [...] :: ex. [abcd] 表示a,b,c,d中任意一个字符, [^abcd]表示除a,b,c,d以外的字符, [0-9]表示 0~9中任意一个数字
* ~     :: 表示用户的home目录

#### ****2.1.3 路径搜索****

当一个Makefile中涉及到大量源文件时(这些源文件和Makefile极有可能不在同一个目录中),

这时, 最好将源文件的路径明确在Makefile中, 便于编译时查找. Makefile中有个特殊的变量 **VPATH** 就是完成这个功能的.

指定了 **VPATH** 之后, 如果当前目录中没有找到相应文件或依赖的文件, Makefile 回到 **VPATH** 指定的路径中再去查找..

**VPATH** 使用方法:

* vpath <directories>            :: 当前目录中找不到文件时, 就从<directories>中搜索
* vpath <pattern> <directories>  :: 符合<pattern>格式的文件, 就从<directories>中搜索
* vpath <pattern>                :: 清除符合<pattern>格式的文件搜索路径
* vpath                          :: 清除所有已经设置好的文件路径

# 示例1 - 当前目录中找不到文件时, 按顺序从 src目录 ../parent-dir目录中查找文件

VPATH src:../parent-dir

# 示例2 - .h结尾的文件都从 ./header 目录中查找

VPATH %.h ./header

# 示例3 - 清除示例2中设置的规则

VPATH %.h

# 示例4 - 清除所有VPATH的设置

VPATH

### ****2.2 Makefile 中的变量****

#### ****2.2.1 变量定义 ( = or := )****

OBJS = programA.o programB.o

OBJS-ADD = $(OBJS) programC.o

# 或者

OBJS := programA.o programB.o

OBJS-ADD := $(OBJS) programC.o

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

**测试 =**

# Makefile内容

OBJS2 = $(OBJS1) programC.o

OBJS1 = programA.o programB.o

all:

@echo $(OBJS2)

# bash中执行 make, 可以看出虽然 OBJS1 是在 OBJS2 之后定义的, 但在 OBJS2中可以提前使用

$ make

programA.o programB.o programC.o

**测试 :=**

# Makefile内容

OBJS2 := $(OBJS1) programC.o

OBJS1 := programA.o programB.o

all:

@echo $(OBJS2)

# bash中执行 make, 可以看出 OBJS2 中的 $(OBJS1) 为空

$ make

programC.o

#### ****2.2.2 变量替换****

# Makefile内容

SRCS := programA.c programB.c programC.c

OBJS := $(SRCS:%.c=%.o)

all:

@echo "SRCS: " $(SRCS)

@echo "OBJS: " $(OBJS)

# bash中运行make

$ make

SRCS: programA.c programB.c programC.c

OBJS: programA.o programB.o programC.o

#### ****2.2.3 变量追加值 +=****

# Makefile内容

SRCS := programA.c programB.c programC.c

SRCS += programD.c

all:

@echo "SRCS: " $(SRCS)

# bash中运行make

$ make

SRCS: programA.c programB.c programC.c programD.c

#### ****2.2.4 变量覆盖 override****

作用是使 Makefile中定义的变量能够覆盖 make 命令参数中指定的变量

语法:

* override <variable> = <value>
* override <variable> := <value>
* override <variable> += <value>

下面通过一个例子体会 override 的作用：

# Makefile内容 (没有用override)

SRCS := programA.c programB.c programC.c

all:

@echo "SRCS: " $(SRCS)

# bash中运行make

$ make SRCS=nothing

SRCS: nothing

#################################################

# Makefile内容 (用override)

override SRCS := programA.c programB.c programC.c

all:

@echo "SRCS: " $(SRCS)

# bash中运行make

$ make SRCS=nothing

SRCS: programA.c programB.c programC.c

#### ****2.2.5 目标变量****

作用是使变量的作用域仅限于这个目标(target), 而不像之前例子中定义的变量, 对整个Makefile都有效.

语法:

* <target ...> :: <variable-assignment>
* <target ...> :: override <variable-assignment> (override作用参见 变量覆盖的介绍)

示例:

# Makefile 内容

SRCS := programA.c programB.c programC.c

target1: TARGET1-SRCS := programD.c

target1:

@echo "SRCS: " $(SRCS)

@echo "SRCS: " $(TARGET1-SRCS)

target2:

@echo "SRCS: " $(SRCS)

@echo "SRCS: " $(TARGET1-SRCS)

# bash中执行make

$ make target1

SRCS: programA.c programB.c programC.c

SRCS: programD.c

$ make target2 <-- target2中显示不了 $(TARGET1-SRCS)

SRCS: programA.c programB.c programC.c

SRCS:

### ****2.3 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.

3种格式的shell命令区别如下:

* 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

示例:

# Makefile 内容 (不用前缀)

all:

echo "没有前缀"

cat this\_file\_not\_exist

echo "错误之后的命令" <-- 这条命令不会被执行

# bash中执行 make

$ make

echo "没有前缀" <-- 命令本身显示出来

没有前缀 <-- 命令执行结果显示出来

cat this\_file\_not\_exist

cat: this\_file\_not\_exist: No such file or directory

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

###########################################################

# Makefile 内容 (前缀 @)

all:

@echo "没有前缀"

@cat this\_file\_not\_exist

@echo "错误之后的命令" <-- 这条命令不会被执行

# bash中执行 make

$ make

没有前缀 <-- 只有命令执行的结果, 不显示命令本身

cat: this\_file\_not\_exist: No such file or directory

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

###########################################################

# Makefile 内容 (前缀 -)

all:

-echo "没有前缀"

-cat this\_file\_not\_exist

-echo "错误之后的命令" <-- 这条命令会被执行

# bash中执行 make

$ make

echo "没有前缀" <-- 命令本身显示出来

没有前缀 <-- 命令执行结果显示出来

cat this\_file\_not\_exist

cat: this\_file\_not\_exist: No such file or directory

make: [all] Error 1 (ignored)

echo "错误之后的命令" <-- 出错之后的命令也会显示

错误之后的命令 <-- 出错之后的命令也会执行

### ****2.4 伪目标****

伪目标并不是一个"目标(target)", 不像真正的目标那样会生成一个目标文件.

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

### ****2.5 引用其他的 Makefile****

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

示例:

# Makefile 内容

all:

@echo "主 Makefile begin"

@make other-all

@echo "主 Makefile end"

include ./other/Makefile

# ./other/Makefile 内容

other-all:

@echo "other makefile begin"

@echo "other makefile end"

# bash中执行 make

$ ll

total 20K

-rw-r--r-- 1 wangyubin wangyubin 125 Sep 23 16:13 Makefile

-rw-r--r-- 1 wangyubin wangyubin 11K Sep 23 16:15 makefile.org <-- 这个文件不用管

drwxr-xr-x 2 wangyubin wangyubin 4.0K Sep 23 16:11 other

$ ll other/

total 4.0K

-rw-r--r-- 1 wangyubin wangyubin 71 Sep 23 16:11 Makefile

$ make

主 Makefile begin

make[1]: Entering directory `/path/to/test/makefile'

other makefile begin

other makefile end

make[1]: Leaving directory `/path/to/test/makefile'

主 Makefile end

### ****2.6 查看C文件的依赖关系****

写 Makefile 的时候, 需要确定每个目标的依赖关系.

GNU提供一个机制可以查看C代码文件依赖那些文件, 这样我们在写 Makefile 目标的时候就不用打开C源码来看其依赖那些文件了.

比如, 下面命令显示内核源码中 virt/kvm/kvm\_main.c 中的依赖关系

$ cd virt/kvm/

$ gcc -MM kvm\_main.c

kvm\_main.o: kvm\_main.c iodev.h coalesced\_mmio.h async\_pf.h <-- 这句就可以加到 Makefile 中作为编译 kvm\_main.o 的依赖关系

### ****2.7 make 退出码****

Makefile的退出码有以下3种：

* 0 :: 表示成功执行
* 1 :: 表示make命令出现了错误
* 2 :: 使用了 "-q" 选项, 并且make使得一些目标不需要更新

### ****2.8 指定 Makefile， 指定特定目标****

默认执行 make 命令时, GNU make在当前目录下依次搜索下面3个文件 "GNUmakefile", "makefile", "Makefile",

找到对应文件之后, 就开始执行此文件中的第一个目标(target). 如果找不到这3个文件就报错.

非默认情况下, 可以在 make 命令中指定特定的 Makefile 和特定的 目标.

示例：

# Makefile文件名改为 MyMake, 内容

target1:

@echo "target [1] begin"

@echo "target [1] end"

target2:

@echo "target [2] begin"

@echo "target [2] end"

# bash 中执行 make

$ ls

Makefile

$ mv Makefile MyMake

$ ls

MyMake

$ make <-- 找不到默认的 Makefile

make: \*\*\* No targets specified and no makefile found. Stop.

$ make -f MyMake <-- 指定特定的Makefile

target [1] begin

target [1] end

$ make -f MyMake target2 <-- 指定特定的目标(target)

target [2] begin

target [2] end

### ****2.9 make 参数介绍****

make 的参数有很多, 可以通过 make -h 去查看, 下面只介绍几个我认为比较有用的.

|  |  |
| --- | --- |
| **参数** | **含义** |
| --debug[=<options>] | 输出make的调试信息, options 可以是 a, b, v |
| -j --jobs | 同时运行的命令的个数, 也就是多线程执行 Makefile |
| -r --no-builtin-rules | 禁止使用任何隐含规则 |
| -R --no-builtin-variabes | 禁止使用任何作用于变量上的隐含规则 |
| -B --always-make | 假设所有目标都有更新, 即强制重编译 |

### ****2.10 Makefile 隐含规则****

这里只列一个和编译C相关的.

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

### ****2.11 隐含规则中的 命令变量 和 命令参数变量****

#### ****2.11.1 命令变量, 书写Makefile可以直接写 shell时用这些变量.****

下面只列出一些C相关的

|  |  |
| --- | --- |
| **变量名** | **含义** |
| RM | rm -f |
| AR | ar |
| CC | cc |
| CXX | g++ |

示例:

# Makefile 内容

all:

@echo $(RM)

@echo $(AR)

@echo $(CC)

@echo $(CXX)

# bash 中执行make, 显示各个变量的值

$ make

rm -f

ar

cc

g++

#### ****2.11.2 命令参数变量****

|  |  |
| --- | --- |
| **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 |

示例: 下面以 CFLAGS 为例演示

# test.c 内容

#include <stdio.h>

int main(int argc, char \*argv[])

{

printf ("Hello Makefile\n");

return 0;

}

# Makefile 内容

test: test.o

$(CC) -o test test.o

# bash 中用 make 来测试

$ ll

total 24K

-rw-r--r-- 1 wangyubin wangyubin 69 Sep 23 17:31 Makefile

-rw-r--r-- 1 wangyubin wangyubin 14K Sep 23 19:51 makefile.org <-- 请忽略这个文件

-rw-r--r-- 1 wangyubin wangyubin 392 Sep 23 17:31 test.c

$ make

cc -c -o test.o test.c

cc -o test test.o <-- 这个是自动推导的

$ rm -f test test.o

$ make CFLAGS=-Wall <-- 命令中加的编译器参数自动追加入下面的编译中了

cc -Wall -c -o test.o test.c

cc -o test test.o

### ****2.12 自动变量****

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 |

## ****3. Makefile 高级语法****

### ****3.1 嵌套Makefile****

在 Makefile 初级语法中已经提到过引用其它 Makefile的方法. 这里有另一种写法, 并且可以向引用的其它 Makefile 传递参数.

示例: (不传递参数, 只是调用子文件夹 other 中的Makefile)

# Makefile 内容

all:

@echo "主 Makefile begin"

@cd ./other && make

@echo "主 Makefile end"

# ./other/Makefile 内容

other-all:

@echo "other makefile begin"

@echo "other makefile end"

# bash中执行 make

$ ll

total 28K

-rw-r--r-- 1 wangyubin wangyubin 104 Sep 23 20:43 Makefile

-rw-r--r-- 1 wangyubin wangyubin 17K Sep 23 20:44 makefile.org <-- 这个文件不用管

drwxr-xr-x 2 wangyubin wangyubin 4.0K Sep 23 20:42 other

$ ll other/

total 4.0K

-rw-r--r-- 1 wangyubin wangyubin 71 Sep 23 16:11 Makefile

$ make

主 Makefile begin

make[1]: Entering directory `/path/to/test/makefile/other'

other makefile begin

other makefile end

make[1]: Leaving directory `/path/to/test/makefile/other'

主 Makefile end

示例: (用export传递参数)

# Makefile 内容

export VALUE1 := export.c <-- 用了 export, 此变量能够传递到 ./other/Makefile 中

VALUE2 := no-export.c <-- 此变量不能传递到 ./other/Makefile 中

all:

@echo "主 Makefile begin"

@cd ./other && make

@echo "主 Makefile end"

# ./other/Makefile 内容

other-all:

@echo "other makefile begin"

@echo "VALUE1: " $(VALUE1)

@echo "VALUE2: " $(VALUE2)

@echo "other makefile end"

# bash中执行 make

$ make

主 Makefile begin

make[1]: Entering directory `/path/to/test/makefile/other'

other makefile begin

VALUE1: export.c <-- VALUE1 传递成功

VALUE2: <-- VALUE2 传递失败

other makefile end

make[1]: Leaving directory `/path/to/test/makefile/other'

主 Makefile end

**\*补充\*** export 语法格式如下:

* export variable = value
* export variable := value
* export variable += value

### ****3.2 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.

**语法:**

define <command-name>

command

...

endef

**示例:**

# Makefile 内容

define run-hello-makefile

@echo -n "Hello"

@echo " Makefile!"

@echo "这里可以执行多条 Shell 命令!"

endef

all:

$(run-hello-makefile)

# bash 中运行make

$ make

Hello Makefile!

这里可以执行多条 Shell 命令!

### ****3.3 条件判断****

条件判断的关键字主要有 **ifeq ifneq ifdef ifndef**

**语法:**

<conditional-directive>

<text-if-true>

endif

# 或者

<conditional-directive>

<text-if-true>

else

<text-if-false>

endif

**示例:** ifeq的例子, ifneq和ifeq的使用方法类似, 就是取反

# Makefile 内容

all:

ifeq ("aa", "bb")

@echo "equal"

else

@echo "not equal"

endif

# bash 中执行 make

$ make

not equal

**示例:** ifdef的例子, ifndef和ifdef的使用方法类似, 就是取反

# Makefile 内容

SRCS := program.c

all:

ifdef SRCS

@echo $(SRCS)

else

@echo "no SRCS"

endif

# bash 中执行 make

$ make

program.c

### ****3.4 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.

函数调用语法如下:

$(<function> <arguments>)

# 或者

${<function> <arguments>}

* <function> 是函数名
* <arguments> 是函数参数

#### ****3.4.1 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

# Makefile 内容

all:

@echo $(subst t,e,maktfilt) <-- 将t替换为e

# bash 中执行 make

$ make

makefile

**模式字符串替换函数: $(patsubst <pattern>,<replacement>,<text>)**

功能: 查找<text>中的单词(单词以"空格", "tab", "换行"来分割) 是否符合 <pattern>, 符合的话, 用 <replacement> 替代.

返回: 替换过的字符串

# Makefile 内容

all:

@echo $(patsubst %.c,%.o,programA.c programB.c)

# bash 中执行 make

$ make

programA.o programB.o

**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

# Makefile 内容

VAL := " aa bb cc "

all:

@echo "去除空格前: " $(VAL)

@echo "去除空格后: " $(strip $(VAL))

# bash 中执行 make

$ make

去除空格前: aa bb cc

去除空格后: aa bb cc

**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

# Makefile 内容

VAL := " aa bb cc "

all:

@echo $(findstring aa,$(VAL))

@echo $(findstring ab,$(VAL))

# bash 中执行 make

$ make

aa

**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

# Makefile 内容

all:

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

# bash 中执行 make

$ make

program.o program.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

# Makefile 内容

all:

@echo $(filter-out %.o %.a,program.c program.o program.a)

# bash 中执行 make

$ make

program.c

**排序函数: $(sort <list>)**

功能: 给字符串 <list> 中的单词排序 (升序)

返回: 排序后的字符串

# Makefile 内容

all:

@echo $(sort bac abc acb cab)

# bash 中执行 make

$ make

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

# Makefile 内容

all:

@echo $(word 1,aa bb cc dd)

@echo $(word 5,aa bb cc dd)

@echo $(word 4,aa bb cc dd)

# bash 中执行 make

$ make

aa

dd

**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

# Makefile 内容

all:

@echo $(wordlist 1,3,aa bb cc dd)

@echo $(word 5,6,aa bb cc dd)

@echo $(word 2,5,aa bb cc dd)

# bash 中执行 make

$ make

aa bb cc

bb

**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

# Makefile 内容

all:

@echo $(words aa bb cc dd)

@echo $(words aabbccdd)

@echo $(words )

# bash 中执行 make

$ make

4

1

0

**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

# Makefile 内容

all:

@echo $(firstword aa bb cc dd)

@echo $(firstword aabbccdd)

@echo $(firstword )

# bash 中执行 make

$ make

aa

aabbccdd

#### ****3.4.2 文件名函数****

**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

# Makefile 内容

all:

@echo $(dir /home/a.c ./bb.c ../c.c d.c)

# bash 中执行 make

$ make

/home/ ./ ../ ./

**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

# Makefile 内容

all:

@echo $(notdir /home/a.c ./bb.c ../c.c d.c)

# bash 中执行 make

$ make

a.c bb.c c.c d.c

**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

# Makefile 内容

all:

@echo $(suffix /home/a.c ./b.o ../c.a d)

# bash 中执行 make

$ make

.c .o .a

**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

# Makefile 内容

all:

@echo $(basename /home/a.c ./b.o ../c.a /home/.d .e)

# bash 中执行 make

$ make

/home/a ./b ../c /home/

**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

# Makefile 内容

all:

@echo $(addsuffix .c,/home/a b ./c.o ../d.c)

# bash 中执行 make

$ make

/home/a.c b.c ./c.o.c ../d.c.c

**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

# Makefile 内容

all:

@echo $(addprefix test\_,/home/a.c b.c ./d.c)

# bash 中执行 make

$ make

test\_/home/a.c test\_b.c test\_./d.c

**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

# Makefile 内容

all:

@echo $(join a b c d,1 2 3 4)

@echo $(join a b c d,1 2 3 4 5)

@echo $(join a b c d e,1 2 3 4)

# bash 中执行 make

$ make

a1 b2 c3 d4

a1 b2 c3 d4 5

a1 b2 c3 d4 e

#### ****3.4.3 foreach****

**语法:**

$(foreach <var>,<list>,<text>)

**示例:**

# Makefile 内容

targets := a b c d

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

all:

@echo $(targets)

@echo $(objects)

# bash 中执行 make

$ make

a b c d

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

#### ****3.4.4 if****

这里的if是个函数, 和前面的条件判断不一样, 前面的条件判断属于Makefile的关键字

**语法:**

$(if <condition>,<then-part>)

$(if <condition>,<then-part>,<else-part>)

**示例:**

# Makefile 内容

val := a

objects := $(if $(val),$(val).o,nothing)

no-objects := $(if $(no-val),$(val).o,nothing)

all:

@echo $(objects)

@echo $(no-objects)

# bash 中执行 make

$ make

a.o

nothing

#### ****3.4.5 call - 创建新的参数化函数****

**语法:**

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

**示例:**

# Makefile 内容

log = "====debug====" $(1) "====end===="

all:

@echo $(call log,"正在 Make")

# bash 中执行 make

$ make

====debug==== 正在 Make ====end====

#### ****3.4.6 origin - 判断变量的来源****

**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 |

**示例:**

# Makefile 内容

val-in-file := test-file

override val-override := test-override

all:

@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

# bash 中执行 make

$ make val-in-cmd=val-cmd

undefined

default

environment

file

command line

override

automatic

#### ****3.4.7 shell****

**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

#### ****3.4.8 make 控制函数****

**产生一个致命错误: $(error <text ...>)**

功能: 输出错误信息, 停止Makefile的运行

# Makefile 内容

all:

$(error there is an error!)

@echo "这里不会执行!"

# bash 中执行 make

$ make

Makefile:2: \*\*\* there is an error!. Stop.

**输出警告: $(warning <text ...>)**

功能: 输出警告信息, Makefile继续运行

# Makefile 内容

all:

$(warning there is an warning!)

@echo "这里会执行!"

# bash 中执行 make

$ make

Makefile:2: there is an warning!

这里会执行!

### ****3.5 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, 呵呵 :)

|  |  |
| --- | --- |
| **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 |

标签: [C学习](http://www.cnblogs.com/wang_yb/tag/C%E5%AD%A6%E4%B9%A0/)