Makefile文件组成：

规则（显式规则和隐含规则）、变量、指令（include 和 条件语句）

# 规则

目标、依赖、命令行

## 目标

### 假想目标

例如：

clean:

rm \*.o temp

因为clean没有具体的依赖，make会认为clean已经是最新，所以如果clean文件存在，则不会执行命令行，而如果clean文件不存在，则还是会执行命令行。

为了防止clean文件恰巧存在而导致命令行不执行的情况，可以使用

.PHONY: clean

clean:

rm \*.o temp

clean被声明为一个假想目标，则make不会去检测文件，而直接认为文件不存在，执行命令行

# 如何使用递归Make

使用$(MAKE)变量来调用执行子MAKE

subsystem:

$(MAKE) -C subdir

-C表示设置环境变量CURDIR的值为subdir

$(MAKE)是调用make的文件名。如文件名是‘/bin/make’，则执行命令`cd subdir && /bin/make'

默认使用上一级make相同版本

## 与子MAKE的通信

子MAKE自动获取当前命令行的定义的变量值或环境变量值来初始化自己shell的环境变量，

### 环境变量

自动将环境变量值传给子MAKE。子MAKE将传入的环境变量作为自己环境变量的初始值，但可以用自己makefile的变量值覆盖传入值

变量MAKELEVEL的值表示当前Make的深度

变量MAKEFILES表示一个makefile列表，子MAKE在执行自己的makefile之前要先执行MAKEFILES中的所有makefile

### 命令行选项

自动将命令行选项值传给子MAKE。实际上就是自动将命令行所有选项放入自动创建的变量MAKEFLAGS中向下传递

选项`-C', `-f', `-o', 和 ‘-W’不能放入变量MAKEFLAGS中向下传递，最好也不要使用‘-t', ‘-n', 和‘-q'

为了取消某些选项的向下传递，可以使用

subsystem:

cd subdir && $(MAKE) MAKEFLAGS=

### Make中自定义变量

通过显示地使用命令export、unexport将make中的自己定义的变量传递给子MAKE

## 命令序列

也是一个变量定义。类似于宏

define run-yacc

yacc $(firstword $^)

mv y.tab.c $@

endef

执行序列时，会在每一行前加Tab；

加前缀执行序列时，会在每一行前加前缀；

## 空命令

唯一的目的是为了阻止目标更新时使用隐含规则提供的命令

# 内部变量

变量可以用来代替任何文本字符串。包括目标、依赖、命令行等。其它版本make称变量为宏。变量不包括前后空格，而且最好是只包含字母数字下划线。

变量名：变量名中也可以包括变量引用和函数，这些引用与函数在读入时展开。

变量值：长度没有限制，但受限于计算机中的实际交换空间。大部分变量都自动设初值为空字串。

|  |  |  |
| --- | --- | --- |
| 变量作用域 变量类型 | 递归型变量 | 简单型变量 |
| 全局变量 |  |  |
| 自动变量 |  |  |
| 特定目标的变量 |  |  |

## 两种变量特色

两种变量区别在于的定义和**展开方式**：递归型是在执行时展开，所以无法确定实际的运行过程；而简单型是在读入时就完全展开了，到执行时所有的过程已经完全确定，所以可控。

**读入时展开：**除了以下几个部分外，其它各个部分的变量和函数都在读入时展开

**执行时展开：**shell命令、使用‘=’定义的‘=’右边的变量、以及使用define指令定义的变量体

### 递归展开型变量

1. 用“=” “define”来定义，并且在执行变量替换时才展开
2. 优点：变量可以在之后定义。因为执行时进行变量替换，而此时整个makefile都已经被读入。

Eg：

CFLAGS = $(include\_dirs) -O

include\_dirs = -Ifoo –Ibar

1. 缺点：

无法递归赋值：

如：CFLAGS = $(CFLAGS) –O会导致执行时死循环

另外，在变量定义中引用函数会导致每次引用此变量都会展开并执行一次函数，影响性能。如果是简单型，则只会在读入（定义）时执行一次函数，其它对此变量的引用就会直接用执行的结果

### 简单展开型变量

1. 用“:=”来定义。并且变量在makefile被读入时就已经展开，所以所有简单型变量都不再包含对其它对象的引用，它们的值就是彻底展开后的结果。

Eg： x := foo

y := $(x) bar

x := later

y的值为foo bar；对比递归型变量，y的值为later bar

1. 优点：

对应于递归型的两个缺点，它既可以递归赋值，也只会对定义中的函数执行一次。

如：

variable := value

variable += more

等同于：

variable := value

variable := $(variable) more

1. 简单型变量能够保留后缀空格

如：

nullstring :=

space := $(nullstring) # end of the line

space的值为一个空格$(nullstring)与注释# end of the line之间的空格

又如：

dir := /foo/bar # directory to put the frobs in

dir的值为/foo/bar ，包括后面四个空格

* 特别的?=变量，表示左边的变量还没有被定义时，此赋值有效，否则无效

### 用+=追加文本

1. 简单型变量追加

variable := value

variable += more

完全等同于：

variable := value

variable := $(variable) more

1. 递归型变量追加

variable = value

variable += more

粗略等同于：

temp = value

variable = $(temp) more

1. 未定义变量追加

默认按照递归型变量追加，如：

objects += another.o

1. 带变量引用的递归型变量追加

CFLAGS = $(includes) -O

...

CFLAGS += -pg # enable profiling

此时$(includes)变量的延时定义和追加就显得很重要。

* 问题：6.6节中一开始为什么

objects = main.o foo.o bar.o utils.o

objects += another.o

等同于：

objects = main.o foo.o bar.o utils.o

objects := $(objects) another.o

？？？？？？

## 变量的高级用法

### 变量的替换引用

1. 语法：$(var:a=b) 含义是将变量var中所有以字符’a’结尾的字替换成以字符’b’结尾。

如：

foo := a.o b.o c.o

bar := $(foo:.o=.c)

bar的值为a.c b.c c.c

1. 第二种写法：

foo := a.o b.o c.o

bar := $(foo:%.o=%.c)

1. 第三种写法：

$(patsubst a,b,$(var))

### 嵌套变量

即变量名中含有$的变量，如$($(x))。不做细究

### 多行变量

define two-lines

echo foo

echo $(bar)

endef

定义的多行变量two-lines为递归型变量。注意：多行变量会在多个shell中运行。

## 特定目标的变量

*target* ... : *variable-assignment*

*或者：*

*target* ... : override *variable-assignment （target可有多个）*

这样定义的目标只在特定目标的命令脚本的上下文起作用。

# 环境变量

Makefile环境变量来至于运行make的环境。Makefile可以对这些环境变量进行重载（但也可以用-e选项让环境变量不能被重载）。

因为同一个makefile在不同环境变量的控制下，可能有不一样的行为，所以环境变量通常是不受欢迎的做法。（因为一个makefile对环境变量做了修改，会影响下一个makefile的执行，类似于全局变量多了易出问题一样）只有比较通用的、一般makefile都不会去改它的变量，如CFLAGS，才适合用环境变量定义。

# 指令

## 条件语句

条件语句是make在读入makefile文件时就已经展开条件变量，做出判断后只读入if或else分支的文本。所以条件变量不能使用自动变量，因为自动变量只在命令行中被定义。

例如：

foo: $(objects)

ifeq ($(CC),gcc)

$(CC) -o foo $(objects) $(libs\_for\_gcc)

else

$(CC) -o foo $(objects) $(normal\_libs)

endif

四个条件指令：

1. ifeq： 当测试是否有非空的值时，需要去掉无意义的空格再判断：ifeq ($(strip $(foo)),)
2. ifneq：
3. ifdef：测试此值是否被定义过，如果定义过，就算是全空格的空值，也是TRUE。

此时条件变量并不会展开做判断，如：

bar =

foo = $(bar)

ifdef bar 为FALSE

ifdef foo 为TRUE

1. ifndef：

# 函数调用

1. 函数调用可以理解为带参数的变量，形式为$(function arguments1, arguments2…)
2. 函数可以是make的内建函数，也可以是自定义的函数。
3. 函数参数变量按照参数列表的次序进行扩展。
4. “，”，“（）”，“{}”不能直接作为参数，而需要定义成变量再赋值。

## 文本处理函数

1. $(subst from,to,text)：用to替换text中的from；
2. $(patsubst pattern,replacement,text)，用*replacement替换text中符合pattern样式的字符串*
3. $(strip string)：去掉前导和结尾空格，并将中间的多个空格压缩为单个空格
4. $(findstring find,in)：在字符串‘in’中搜‘find’，如果找到，返回值是‘find’，否则返回空
5. $(filter pattern...,text)：返回在‘text’中由空格隔开且匹配格式‘pattern...’的字，对于不符合格式‘pattern...’的字移出。如：

sources := foo.c bar.c baz.s ugh.h

foo: $(sources)

cc $(filter %.c %.s,$(sources)) -o foo

1. $(filter-out *pattern*...,*text*) ：函数filter的反函数，返回在‘text’中由空格隔开且不匹配格式‘pattern...’的字。
2. $(sort *list*) ：将‘list’中的字按字母顺序排序，并取掉重复的字。输出是由单个空格隔开的字的列表。

## 文件名函数

1. $(dir *names*...) ：抽取‘names’中每一个文件名的路径部分。

如： $(dir src/foo.c hacks)

产生的结果为 ‘src/ ./’。

1. $(notdir *names*...) ：抽取‘names’中每一个文件名中除路径部分外一切字符（真正的文件名）。如：

如： $(notdir src/foo.c hacks)

产生的结果为‘foo.c hacks’。

1. $(suffix *names*...) ：抽取‘names’中每一个文件名的后缀。如：

如： $(suffix src/foo.c src-1.0/bar.c hacks)

产生的结果是‘.c .c’。

1. $(basename names...): 抽取‘names’中每一个文件名中除后缀外一切字符。如：

如： $(basename src/foo.c src-1.0/bar hacks)

产生的结果为‘src/foo src-1.0/bar hacks’

1. $(addsuffix *suffix*,*names*...) ：参数‘names’作为一系列的文件名，文件名之间用空格隔开；

如： $(addsuffix .c,foo bar)

结果为‘foo.c bar.c’。

1. $(addprefix *prefix*,*names*...) ：参数‘names’作为一系列的文件名，文件名之间用空格隔开；

如： $(addprefix src/,foo bar)

结果为‘src/foo src/bar’。

1. $(join list1,list2)：

如：‘$(join a b,.c .o)'产生‘a.c b.o'

可用于将函数dir、notdir的结果合并，产生原始给定的文件列表

1. $(word n,text) 返回‘text’中的第n个字。N的合法值从1开始。如果n比‘text’中的字的数目大，则返回空值。

如： $(word 2, foo bar baz)

返回 ‘bar’。

1. $(wordlist s,e,text) 返回‘text’中的从第s个字开始到第e个字结束的一列字。S、e的合法值从1开始。如果s比‘text’中的字的数目大，则返回空值；

如： $(wordlist 2, 3, foo bar baz)

返回`bar baz'

1. $(words *text*) ：返回‘text’中字的数目。

如：‘text’中的最后一个字是‘$(word $(words *text*),*text*)’。

1. $(firstword *names*...) ：参数‘names’作为一系列的文件名，文件名之间用空格隔开；返回第一个文件名，其余的忽略。

相当于$(word 1,text)

1. $(wildcard *pattern*) ：函数结果是一列和格式匹配的且文件存在的文件名，文件名之间用一个空格隔开

## 特殊函数

1. $(foreach var,list,text)：先扩展var和list，对于list中的每一个字，都赋值给展开后的var，然后‘text’每次都引用不同的var进行计算。

如： dirs := a b c d

files := $(foreach dir,$(dirs),$(wildcard $(dir)/\*))

相当于

files := $(wildcard a/\* b/\* c/\* d/\*)

条件函数：

1. $(if condition,then-part[,else-part])：
2. 如果扩展为非空字符串，则条件‘condition’为‘真’；如果扩展为空字符串，则条件‘condition’为‘假’；
3. condition为真，计算then-part参数变量，否则计算else-part，没有else-part就返回空值。
4. $(call variable,param,param,...)：可用于调用自定义函数的函数
5. $(origin variable)：结果是一个字符串，表示*variable*的一些来源类型等信息

典型的结果有：

‘undefined'

如果变量‘variable’从没有定义。

‘default'

变量‘variable’是缺省定义，

environment'

变量‘variable’作为环境变量定义，选项‘-e’没有打开

‘environment override'

变量‘variable’作为环境变量定义，选项‘-e’已打开

等等。。。

1. Shell函数：执行shell命令，在大多数情况下，make读入时，shell函数就已经执行了（跟shell命令不一样）。

如： files := $(shell echo \*.c)

1. make的控制函数

$(error text...)

$(warning *text*...)

# Make的执行参数

通常执行make编译所有过期文件时，不需要参数。当有一些特殊需求时，比如仅仅找出而不更新过期文件，使用不同的编译器或编译选项等，就需要使用make参数了。

## 指定最终目标

1. Makefile中有一个缺省目标（也可以有多个并列，第一个为缺省）；
2. 如果执行make时要将一个非缺省目标当作最终目标，就要在执行make时指定参数：

如：

make clean，make all，make mostlyclean make install，make print，make tar，make dist等等；

## 指定命令选项

更新目标是makefile的默认工作，但有时不需要更新目标。可以使用一些选项来改变make的行为，如-n、-t等等

make –C：用于在将makefile读入之前，切换路径

make –d：用于打印make执行的调试信息

make –e：用于设置环境变量的值，优先级高于makefile中的设置。可用于禁止对此环境变量的重载

make –f：指定makefile文件。可以指定多个文件，这样会连在一起作为make的makefile。

make –h：显示make程序的所有命令选项

make –i：忽略所有重建文件过程中产生的错误？？？

make –I：知道makefile的搜索目录

make –j：make最多可执行的任务数

make –k：测试型运行make。通常在执行shell命令时一旦有错误发生，make立即退出返回非零状态。但如果使用使用‘-k’或‘--keep-going’选项，则make会在遇到致命错误之后，也会强行运行下去，将之后的所有错误也都暴露出来，以便一次性解决所有的问题。

make –n、

make –t：直接更新目标的时间标志，却不更新目标本身。即假装更新了目标。使用者更改某文件后，保守做法所有依赖于它的文件都必须被更新，但当使用者有把握依赖于它的其它文件不需要编译更新时，就可以使用-t选项，节省时间。

make –q等等

## 指定环境变量值

例如：make CFLAGS='-g -O' 是不是类似于在makefile中重载变量：override CFLAGS='-g -O'？？

# 隐含规则

隐含规则就是对一些目标文件没有定义更新规则，而采用的默认规则。通过隐含规则，可以使用一些默认规则来执行make，减少makefile的篇幅。Makefile通过一些变量来改变隐含规则的工作方式：如CFLAGS可以控制编译器的隐含规则等。

格式规则：可用于自定义自己的隐含规则；后缀规则：

## 使用隐含规则：

1. 根据makefile文件中提到的各种文件，使用隐含规则
2. 如果文件的依赖需要额外添加，就需要定义显式的规则
3. 如果一条隐含规则的依赖是另一条隐含规则的结果，我们说产生了‘链’，隐含规则链
4. 对同一个目标，如果有多个可选依赖存在，则使用一个规则列表按优先级顺序放置依赖。如foo.o依赖于foo.c或foo.p，如果没有显式定义，则优先使用foo.c

## 定义隐含规则

格式规则

后缀规则

# Make的其它用途

使用make更新档案文件（如用于链接的子例程库）

# Makefile编码规范

1. makefile文件都应该包含这行：

SHELL = /bin/sh

这样可以避免不同环境中SHELL变量值不同的问题

1. 不同的make程序有不同的后缀列表和隐含规则，所以最好显式地定义，避免混乱

.SUFFIXES:

.SUFFIXES: .c .o

第一行清除了后缀列表，第二行定义了在该makefile中可能被隐含规则使用的后缀。

1. 路径变量的使用：

**Build目录一般用‘./’**

源码目录一般用‘$(srcdir)’

## Makefile的可移植性

用于创建和安装的‘configure’脚本和Makefile 规则不要使用任何ksh或bash的特殊特点。除下面所列出工具外不应该直接使用其它的任何工具：

cat cmp cp diff echo egrep expr false grep install-info

ln ls mkdir mv pwd rm rmdir sed sleep sort tar test touch true

# 附录：

变量、函数、指令汇总

错误列表