Makefile Notes

关于编译的概念：

编译生成目标文件 .obj/.o 则是编译

生成可执行文件 .exe叫 链接

在大多数时候，由于源文件太多，编译生成的中间目标文件太多，而在链

接时需要明显地指出中间目标文件名，这对于编译很不方便，所以，我们要给中间目标文件打个包，在Windows下这种包叫“库文件”（Library File)，也就是 .lib 文件，在UNIX下，是Archive File，也就是 .a 文件。

Makefile格式：

target ... **:** prerequisites ...

[必须有tab] command

在找寻的过程中，如果出现错误，比如最后被依赖的文件找不到，那么make就会直接退出，并报错，而对于所定义的命令的错误，或是编译不成功，make根本不理

edit : main.o kbd.o command.o display.o insert.o search.o files.o utils.o

cc -o edit main.o kbd.o command.o display.o insert.o search.o files.o utils.o

最原始的规则

实际上可以将所有”.o” 用变量代替

obj= main.o kbd.o command.o display.o insert.o search.o files.o utils.o

然后就这样勒：

edit : $(obj)

cc -o edit $(obj)

main.o : defs.h

可以自动推导出man.c

.PHONY 伪目标

可以多个目标用一个依赖

kbd.o command.o files.o : command.h

clean的规则不要放在文件的开头，不然，这就会变成make的默认目标

注释用#开始

在Makefile中的命令，必须要以[Tab]键开始。

默认情况下make会按照顺序找三个文件：GNUmakefile,makefile,Makefile

一般使用Makefile 这样醒目

GNUmake是GNU的make识别。

同样也可以用其他makefile 但用 –f来指定文件

包含其他makefile

include <filename>

filename可以是当前操作系统Shell的文件模式（可以保含路径和通配符） 在include前面可以有一些空字符，但是**绝不能是[Tab]键开始**

如果make执行时，有“-I”或“--include-dir”参数，那么make就会在这个参数

所指定的目录下去寻找。

如果你想让make不理那些无法读取的文件，而继续执行，你**可以在include前加一个减号“-”**

GNU的make工作时的执行步骤入下：（想来其它的make也是类似）

1、读入所有的Makefile。

2、读入被include的其它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。

Makefile使用拖延的方式展开命令

make会以UNIX的标准Shell，也就是/bin/sh来执行命令

通配符：

make支持三各通配符：“\*”，“?”和“[...]”

Makefile中的变量其实就是C/C++中的宏

objects := $(wildcard \*.o) wildcard 是makefile的一个关键字

特殊变量“VPATH“：当make需要去找寻文件的依赖关系时，你可以在文件前加上路径，但最好的方法是把一个路径告诉make，让make在自动去找。如果没有指明这个变量，make只会在当前的目录中去找寻依赖文件和目标文件。如果定义了这个变量，那么，make就会在当当前目录找不到的情况下，到所指定的目录中去找寻文件了

VPATH = src:../headers 由冒号隔开

另一个设置文件搜索路径的方法是使用make的“vpath”关键字（注意，它是全小写的），这不是变量，这是一个make的关键字，这和上面提到的那个VPATH变量很类似，但是它更为灵活。它可以指定不同的文件在不同的搜索目录中。这是一个很灵活的功能。它的使用方法有三种：

1、vpath <pattern> <directories>

为符合模式<pattern>的文件指定搜索目录<directories>。

2、vpath <pattern>

清除符合模式<pattern>的文件的搜索目录。

3、vpath

清除所有已被设置好了的文件搜索目录

Eg:

vpath %.h ../headers 搜索 %.h(%表示0个或多个匹配) 在../headers目录中查找 （如果再当前目录没有找到的话）

vpath %.c foo:bar

vpath % blish

伪目标：

伪目标”并不是一个文件，只是一个标签，由于“伪目标”不是文件，所以make无法生成它的依赖关系和决定它是否要执行。我们只有通过显示地指明这个“目标”才能让其生效。当然，“伪目标”的取名不能和文件名重名，不然其就失去了“伪目标”的意义了

“.PHONY : all”声明了“all”这个目标为“伪目标”。

自动化变量： $@ 表示目前规则中所有的目标的集合

Makefile 函数：

-$(subst output,,$@)中的“$”表示执行一个Makefile的函数，函数名为subst，后面的为参数

**静态模式**

静态模式可以更加容易地定义多目标的规则，可以让我们的规则变得更加的有弹性和灵活

<targets ...>: <target-pattern>: <prereq-patterns ...>

<commands>

....

targets定义了一系列的目标文件，可以有通配符。是目标的一个集合。

target-parrtern是指明了targets的模式，也就是的目标集模式

prereq-parrterns是目标的依赖模式，**它对target-parrtern形成的模式再进行一次依赖目标的定义**

**<target-parrtern>定义成“%.o”，意思是我们的<target>集合中都是以“.o”结尾的，而如果我们的<prereq-parrterns>定义成“%.c”，意思是对<target-parrtern>所形成的目标集进行二次定义，其计算方法是，取<target-parrtern>模式中的“%”（也就是去掉了[.o]这个结尾），并为其加上[.c]这个结尾，形成的新集合**

**Eg:**

*objects = foo.o bar.o*

*all: $(objects)*

*$(objects): %.o: %.c*

*$(CC) -c $(CFLAGS) $< -o $@*

自动化变量：，“$<” 表示所有的依赖目标集

自动化变量：，“$@” 表示所有的依赖目标集

$(filter %.o,$(files))表示调用Makefile的filter函数，过滤“$filter”集，只要其中模式为“%.o”的内容

可以通过-MM 显示依赖关系 例如：  
cc –MM main.c -M 会包含系统头文件

自动生成依赖规则

Eg:

%.d: %.c

@set -e; rm -f $@; \

$(CC) -M $(CPPFLAGS) $< > $@.$$$$; \

sed 's,\($\*\)\.o[ :]\*,\1.o $@ : ,g' < $@.$$$$ > $@; \

rm -f [$@.$$$$](mailto:$@.$$$$)

***表示没看明白***

$$$$ 表示一个随机编号

-n / --just-print 参数会执行打印，但是不会真的编译

-s / --slient 禁止所有的命令显示

**cd**

**pwd**

**和**

**cd;pwd**

是不是一样的

如果你要让上一条命令的结果应用在下一条命令时，你应该使用分号分隔这两条命令

**-i / --ignore—errors 忽略所有错误**

-C 指定编译的目录

总控Makefile的变量可以传递到下级的Makefile中（如果你显示的声明），但是不会覆盖下层的Makefile中所定义的变量，除非指定了“-e”参数。

export <variable ...> 传递变量到下一级目录

unexport <variable ...> 不传递某个变量

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，其总是要传递到下层Makefile中，特别是MAKEFILES变量，其中包含了make的参数信息，如果我们执行“总控Makefile”时有make参数或是在上层Makefile中定义了这个变量，那么MAKEFILES变量将会是这些参数，并会传递到下层Makefile中，这**是一个系统级的环境变量**。

“-C”,“-f”,“-h”“-o”和“-W” 这些参数是不会传递到下级目录的

如果不想传入可以这样写：

subsystem:

cd subdir && $(MAKE) **MAKEFLAGS=**

-w / --print-directory 可以看到make 的工作目录

当有 –C参数是 –w会被自动打开

如果 有 –s / --slient -w 会自动失效

命令包：

define **run-yacc**

yacc $(firstword $^)

mv y.tab.c $@

endef

run-yacc 是包名

**变量**

变量的命名字可以包含字符、数字，下划线（可以是数字开头），但不应该含有“:”、 “#”、“=”或是空字符（空格、回车等）

变量基础：

变量在声明时需要给予初值，而在使用时，需要给在变量名前加上“$”符号，但最好用小括号“（）”或是大括号“{}”把变量给包括起来。如果你要使用真实的“$”字符，那么你需要用“$$”来表示

变量中的变量

foo = $(bar)

bar = $(ugh)

ugh = Huh?

**变量可以使用那个后面的变量来定义！**

**不好的地方就是递归:**

A = $(B)

B = $(A)

Makefile是有能力检查这个错误的

还有就是如果**在变量中使用函数**，那么，这种方式会让我们的make运行时非常慢，更糟糕的是，他会使用得两个make的函数“wildcard”和“shell”发生不可预知的错误。因为你不会知道这两个函数会

**为了避免这种问题 可以用操作符 :=来定义 (**前面的变量不能使用后面的变量，只能使用前面已定义好了的变量**)**

Eg:

ifeq (0,${MAKELEVEL})

cur-dir := $(shell pwd)

whoami := $(shell whoami)

host-type := $(shell arch)

MAKE := ${MAKE} host-type=${host-type} whoami=${whoami}

Endif

MAKELEVEL: 记录我们的当前Makefile的调用层数

定义空格：

nullstring :=

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

#开始的地方就是空格结束的地方

**?=** 如果变量没有定义过就直接赋值，否则什么都不做

FOO ?= bar

和下面的等价

ifeq ($(origin FOO), undefined)

FOO = bar

endif

变量高级用法

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

bar := $(foo:.o=.c) 替换 将 .o 替换成.c

变量的值当成变量：

x = y

y = z

a := $($(x))

也可以重组变量：

first\_second = Hello

a = first

b = second

all = $($a\_$b)

函数和条件：

ifdef do\_sort

func := sort

else

func := strip

endif

bar := a d b g q c

foo := $($(func) $(bar))

追加变量值: +=

如果变量之前没有定义过，那么，“+=”会自动变成“=”

override <variable> := <value>

对于命令包 define 同样适用

-e 系统环境变量覆盖Makefile环境变量

条件：

ifeq ($(CC),gcc)

libs=$(libs\_for\_gcc)

else

libs=$(normal\_libs)

endif

ifneq

ifeq (<arg1>, <arg2>)

ifeq '<arg1>' '<arg2>'

ifeq "<arg1>" "<arg2>"

ifeq "<arg1>" '<arg2>'

ifeq '<arg1>' "<arg2>"

Makefile 函数：

函数定义

$(<function> <arguments>)

或是

${<function> <arguments>}

<function>就是函数名，make支持的函数不多。<arguments>是函数的参数，参数间以逗号“,”分隔，而函数名和参数之间以“空格”分隔

comma:= ,

empty:=

space:= $(empty) $(empty)

foo:= a b c

bar:= $(subst $(space),$(comma),$(foo))

字符串替换函数：

**$(subst <from>,<to>,<text>)** 把字串<text>中的<from>字符串替换成<to> 返回替换后的字符

**$(patsubst <pattern>,<replacement>,<text>)** 查找<text>中的单词（单词以“空格”、“Tab”或“回车”“换行”分隔）是否符合模式<pattern>，如果匹配的话，则以<replacement>替换

Eg:

“$(var:<pattern>=<replacement>)”

相当于

“$(patsubst <pattern>,<replacement>,$(var))”，

而“$(var: <suffix>=<replacement>)” 则相当于

“$(patsubst %<suffix>,%<replacement>,$(var))”。

例如有：objects = foo.o bar.o baz.o，

那么，“$(objects:.o=.c)”和“$(patsubst %.o,%.c,$(objects))”是一样的

$(strip <string>)去掉<string>字串中开头和结尾的空字符

$(findstring <find>,<in>) 在字串<in>中查找<find>字串。

$(filter <pattern...>,<text>) 从<text> 中查找 <pattern>的内容并返回

$(filter-out <pattern...>,<text>) 反过滤函数，将模式匹配的内容去除并返回

$(sort <list>) 排序函数并返回

$(word <n>,<text>)取字符串<text>中第<n>个单词。如果<n>比<text>中的单词数要大，那么返

回空字符串。

$(wordlist <s>,<e>,<text>)如果<s>比<text>中的单词数要大，那么返回空字符串。如果<e>大于<text>的单词数，那么返回从<s>开始，到<text>结束的单词串

Eg:

$(wordlist 2, 3, foo bar baz)返回值是“bar baz”。

$(words <text>) 统计<text>列表中的字串个数

$(firstword <text>):取字符串<text>中的第一个单词

Eg: (这个表达式没怎么看懂) 哦看懂了，将VPATH中的“：”替换成空格 让后在将替换后的列表都加上前缀 –I 在赋值给CFLAGS，这个CFLAGS被重新赋值了（override）

override CFLAGS += $(patsubst %,-I%,$(subst :, ,$(VPATH)))

如果我们的“$(VPATH)”值是“src:../headers”，那么“$(patsubst %,-I%,$(subst :, ,$(VPATH)))”将返回“-Isrc -I../headers”，这正是cc或gcc搜索头文件路径的参数。

文件名操作

dir

$(dir <names...>) 从文件名序列<names>中取出目录部分。目录部分是指最后一个反斜杠（“/”）之前的部分。如果没有反斜杠，那么返回“./”。

Eg：

$(dir src/foo.c hacks)返回值是“src/ ./”。

$(notdir <names...>) 从文件名序列<names>中取出非目录部分。非目录部分是指最后一个反斜杠（“ /”）之后的部分。

$(suffix <names...>) 从文件名序列<names>中取出各个文件名的后缀

$(basename <names...>) 从文件名序列<names>中取出各个文件名的前缀部分

Eg:

$(basename src/foo.c src-1.0/bar.c hacks)返回值是“src/foo src-1.0/b

ar hacks”

$(addsuffix <suffix>,<names...>) 把后缀<suffix>加到<names>中的每个单词后面。

Eg:

$(addsuffix .c,foo bar)返回值是“foo.c bar.c”

和类似的addsuffix的一样

$(addprefix <prefix>,<names...>)

$(join <list1>,<list2>)把<list2>中的单词对应地加到<list1>的单词后面。如果<list1>的单词个数要比<list2>的多，那么，<list1>中的多出来的单词将保持原样。如果<list2>的单词个数要比<list1>多，那么，<list2>多出来的单词将被复制到<list2>中。

Eg:

$(join aaa bbb , 111 222 333)返回值是“aaa111 bbb222 333”

**Foreach函数**

$(foreach <var>,<list>,<text>) 把参数<list>中的单词逐一取出放到参数<var>所指定的变量中，然后再执行<text>所包含的表达式。每一次<text>会返回一个字符串，循环过程中，<text>的所返回的每个字符串会以空格分隔，最后当整个循环结束时，<text>所返回的每个字符串所组成的整个字符串（以空格分隔）将会是foreach函数的返回值

Eg:

names := a b c d

files := $(foreach n,$(names),$(n).o)

if 函数

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

或者

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

Call 函数：

函数是唯一一个可以用来创建新的参数化的函数

格式：

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

Eg:

reverse = $(1) $(2)

foo = $(call reverse,a,b)

origin 函数不像其它的函数，他并不操作变量的值，他只是告诉你你的这个变量是哪里来的

$(origin <variable>)

其返回值：

Undefined ：没有定义过就返回这个值

Default：默认定义例如CC这个变量

File: 变量被定义在Makefilezhong

“command line” 变量被定义在命令行中

“override” 指示提示符重新定义的

“automatic” 一个命令行的自动化变量

shell函数也不像其它的函数。顾名思义，它的参数应该就是操作系统Shell的命令。

eg:

contents := $(shell cat foo)

make控制的执行或者停止的函数：

$(error <text ...>)

$(warning <text ...>)

Make的退出码：

0 成功

1 出现任何错误返回1

2 使用 make –q 返回2

有一个make的环境变量叫“MAKECMDGOALS”，这个变量中会存放你所指定的终极目标的列表，如果在命令行上，你没有指定目标，那么，这个变量是空值

.PHONY

:all 这个伪目标是是所有目标的目标

:clean 删除所有被make创建的文件

: install 把目标可执行文件拷贝到指定目录中去

： print 列出所有改变过的文件

：tar 把源程序打包备份

： dist 创建一个压缩文件 把tar压缩成z文件或者gz文件

：TAGS 这个伪目标更新所有的目标，以备完整的重新编译

： check / test 这两个伪目标用来测试makefile流程

-t / --touch 这个参数的意思就是把目标文件的时间更新，但不更改目标文件

-q / --question 这个参数的行为是找目标的意思，也就是说，如果目标存在，那么其什么也不会输出，当然也不会执行编译，如果目标不存在，其会打印出一条出错信息

“-W <file>”

“--what-if=<file>”

“--assume-new=<file>”

“--new-file=<file>”

这个参数需要指定一个文件。一般是是源文件（或依赖文件），Make会根据规则推导来运行依赖于这个文件的命令，一般来说，可以和“-n”参数一同使用，来查看这个依赖文件所发生的规则命令

“-p”和“-v”来输出makefile被执行时的信息

make –pv 是可以打开很多有用的编译信息

-b /-m 这两个参数的作用是忽略和其它版本make的兼容性

Make 还有版本之分

-B / --always-make 认为所有目标需要更新（完全重编译）

“—debug[=<options>]”

输出make的调试信息。它有几种不同的级别可供选择，如果没有参数，那就是输出最简单的调试信息。下面是<options>的取值：

a —— 也就是all，输出所有的调试信息。（会非常的多）

b —— 也就是basic，只输出简单的调试信息。即输出不需要重编译的目标。

v —— 也就是verbose，在b选项的级别之上。输出的信息包括哪个makefile被解析，不需要被重编译的依赖文件（或是依赖目标）等。

i —— 也就是implicit，输出所以的隐含规则。

j —— 也就是jobs，输出执行规则中命令的详细信息，如命令的PID、返回码等。

m —— 也就是makefile，输出make读取makefile，更新makefile，执行makefile的信息。

“-l <load>”

“--load-average[=<load]”

“—max-load[=<load>]”

指定make运行命令的负载。

C的隐含规则：

“<n>.o”的目标的依赖目标会自动推导为“<n>.c”，并且其生成命令是“$(CC) –c $(CPPFLAGS) $(CFLAGS)”

C++的隐含规则：

“<n>.o”的目标的依赖目标会自动推导为“<n>.cc”或是“<n>.C”，并且其生成命令是“$(CXX) –c $(CPPFLAGS) $(CFLAGS)”。（建议使用“.cc”作为C++源文件的后缀，而不是“.C”）

AR

函数库打包程序。默认命令是“ar”。

AS

汇编语言编译程序。默认命令是“as”。

RM

删除文件命令。默认命令是“rm –f”。

$% 仅当目标是函数库文件中，表示规则中的目标成员名。例如，如果一个目标是"foo.a (bar.o)"，那么，"$%"就是"bar.o"，"$@"就是"foo.a"

$?

所有比目标新的依赖目标的集合。以空格分隔。

$^

所有的依赖目标的集合。以空格分隔。如果在依赖目标中有多个重复的，那个这个变量会去除重复的依赖目标，只保留一份。

$+

这个变量很像"$^"，也是所有依赖目标的集合。只是它不去除重复的依赖目标

$\* （GNUmake 特有的）

这个变量表示目标模式中"%"及其之前的部分。如果目标是"dir/a.foo.b"，并且目标的模式是"a.%.b"，那么，"$\*"的值就是"dir/a.foo"。这个变量对于构造有关联的文件名是比较有较。如果目标中没有模式的定义，那么"$\*"也就不能被推导出，但是，如果目标文件的后缀是make所识别的，那么"$\*"就是除了后缀的那一部分。例如：如果目标是"foo.c"，因为".c"是make所能识别的后缀名，所以，"$\*"的值就是"foo"

$(@D)

表示"$@"的目录部分（不以斜杠作为结尾），如果"$@"值是"dir/foo.o"，那么"$(@D)"就是"dir"，而如果"$@"中没有包含斜杠的话，其值就是"."（当前目录）。

$(@F)

表示"$@"的文件部分，如果"$@"值是"dir/foo.o"，那么"$(@F)"就是"foo.o"，"$(@F)"相当于函数"$(notdir $@)"。

"$(\*D)"

"$(\*F)"

和上面所述的同理，也是取文件的目录部分和文件部分。对于上面的那个例子，"$(\*D)"返回"dir"，而"$(\*F)"返回"foo"

"$(%D)"

"$(%F)"

分别表示了函数包文件成员的目录部分和文件部分。这对于形同"archive(member)"形式的目标中的"member"中包含了不同的目录很有用。

"$(<D)"

"$(<F)"

分别表示依赖文件的目录部分和文件部分。

"$(^D)"

"$(^F)"

分别表示所有依赖文件的目录部分和文件部分。（无相同的）

".SUFFIXES"来定义或是删除，如：

.SUFFIXES: .hack .win

archive(member) 这个不是一个命令，而一个目标和依赖的定义。一般来说，这种用法基本上就是为了"ar"命令来服务的

eg:

foolib(hack.o) : hack.o

ar cr foolib hack.o

很多规则不懂

eg：

.c.a:

$(CC) $(CFLAGS) $(CPPFLAGS) -c $< -o $\*.o

$(AR) r $@ $\*.o

$(RM) $\*.o

其等效于：

(%.o) : %.c

$(CC) $(CFLAGS) $(CPPFLAGS) -c $< -o $\*.o

$(AR) r $@ $\*.o

$(RM) $\*.o

eval”函数执行时会对它的参数进行两次展开。第一次展开过程发是由函数本身完成的，第二次是函数展开后的结果被作为Makefile内容时由make 解析时展开的。明确这一点对于使用“eval”函数非常重要。在理解了函数“eval”二次展开的过程后。实际使用时，当函数的展开结果中存在引用（格式 为：$(x)）时，那么在函数的参数中应该使用“$$”来代替“$”。因为这一点，所以通常它的参数中会使用函数“value”来取一个变量的文本值。

特殊目标

λ .IGNORE

make在执行命令行时，如果返回的是错误码，make的缺省动作是停止并退出。增加该目标后，make将忽略命令行返回的错误码，并继续执行后续的操作。

λ .SILENT

前面已经介绍过，make在执行命令行时会回显命令行内容，在命令行前增加“@”字符将抑制该命令行的回显。

如果增加该目标，所有的命令行不再回显，相当于在每个命令行前均增加了“@”字符。

λ .PRECIOUS

当收到一个信号或从shell命令返回非零的错误码时，make删除它所有已建立的文件。但有些文件即使出了错误，用户也不想让make删除，这些文件可以作为.PRECIOUS目标的参数。它可以在一个makefile中出现多次，每一次都累积文件列表。

λ .SUFFIXES

它为makefile指定新的后缀规则，新的后缀规则作为.SUFFIXES的依赖表给出。.SUFFIXES可以在一个makefile中多次使用，每一次都将新的后缀规则加入以前的后缀规则中，如果.SUFFIXES的依赖表为空，则设置后缀规则表为空。