**4 规则书写**

Makefile中的规则描述了何时以及如何重建特定的文件,这类特定的文件就被称为规则的目标(通常一条规则只有一个目标).规则中除了目标之外,还包含了目标的依赖文件,以及如果创建或更新该目标的配方.

除了 makefile 的默认目标”所在的规则,其他规则的书写顺序并不重要.默认目标就是当没有使用 make命令行指定具体目标时， make 默认的更新的那个目标.默认目标是Makefile中书写的第一条规则 中的目标.如果第一条规则中包含了多个目标,那么只有其中的第一个目标将被作为默认目标.这里存在两种例外情况: 一种是目标名以点号“.” 开始的并且其后不存在斜线“/”；另一种是模式规则中的目标.满足这两种情况之一的目标都不会被作为默认目标.

因此,通常情况下,我们在Makefile中书写的第一条规则就应该是编译整个程序或多个程序的最终描述(通常将这条规则的目标名写为"all").

**4.1 规则语法**

通常,一条规则的语法格式如下:

目标列表 : 依赖列表

配方

...

或者:

目标列表 : 依赖列表；配方

配方

...

"目标列表"通常是空格分开的多个文件名 .文件名除了可以使用通配符之外,格式"a(m)"表示档案文件a中的成员m.通常一条规则只包含一个目标,但偶尔情况下也会包含多个目标.

独立一行书写"配方"时,必须以一个tab字符开始(或者是由.RECIPEPREFIX变量指定的字符).第一条"配方"可以写在"依赖列表"后面并用分号";"隔开,并且多条配方也可以写在同一行,同样使用分号";"隔开即可.这些书写格式的效果都是一样的.

由于Makefile中使用"$"符号来引用变量,所以如果你想要在目标或依赖中单纯的使用"$"符号,需要写两个"$$".如果是在已经使能了二次展开功能的情况下,需要使用四个"$$$$"来达到同样的目的.

你可以使用反斜线"\"将一个较长的行分割成多个独立行,尽管这不是必须的,因为make程序对Makefile中一行的长度没有做限制.

一条规则告诉了make程序两件事:一件是目标何时过期；另一件是当需要时如何去更新目标.

目标是否过期的标准是由那些使用空格分割的"依赖列表"所决定的(通配符和档案文件成员在这里也同样适用).如果目标不存在或者它比"依赖列表"中的一些成员老(通过比较最后修改时间)时,则判定该目标过期.目标文件的内容是基于"依赖列表"的信息生成的,所以如果"依赖列表"中有成员发生变化,那么已经存在的目标文件必然不再有效.

更新目标的方法是由"配方"指定的,"配方"也就是一行或多行shell命令以及一些附加功能.

**4.2 依赖类型**

实际上有两种不同类型的依赖可以被GNU make程序理解: 一种是之前章节中描述的常规依赖；另一种是"order-only"依赖.常规依赖主要陈述了两件事:首先,由于目标的配方在执行之前必须确保该目标的所有常规依赖都已经更新,所以这些常规依赖的书写顺序决定了各自的更新顺序；其次,它确定了一个依赖关系,就是如果有任意一个常规依赖比目标新,那么就认为目标已经过期而需要被重建.

通常有一点是很明确的,那就是如果一个目标的依赖被更新了,则该目标也需要被更新.

但是,偶尔你会遇到这样一种情况,你想在规则的依赖列表中设置一个特殊的分类,这个特殊分类中的依赖被**更新**后不会强制规则的目标也进行更新.这种情况下,你就需要定义"order-only"依赖."order-only"依赖可以通过在依赖列表中放置管道符号"|"来进行设置: 管道符号“ |” 左边的是常规依赖， 管道符号右边的就是“order-only” 依赖.这样的规则书写格式如下：

目标列表 : 常规依赖部分 | "order-only"依赖部分

"常规依赖部分"可以为空.同样,也可以对一个目标进行多次追加依赖.需要注意的是,如果你在规则的常规依赖和"order-only"依赖中都设置了同一个文件,那么该文件将优先被视为常规依赖(因为常规依赖的行为是"order-only"依赖行为的一个超集).

考虑下这么一个例子,你的目标文件位于一个独立的目录下,而这个目录在执行make之前可能并不存在.在这种情况下,你希望只是在任何一个目标文件被放置进去之前创建该目录,但是因为目录的时间戳会因为里面任何一个文件被添加,删除,或者重命名而修改,我们显然不希望因为目录时间戳一改变就去重建目标文件.处理这个问题的一种方法就是使用"order-only"依赖: 将该目录设置为所有目标的"order-only"依赖.

OBJDIR := objdir

OBJS := $(addprefix $(OBJDIR)/,foo.o bar.o baz.o)

$(OBJDIR)/%.o : %.c

$(COMPILE.c) $(OUTPUT\_OPTION) $<

all: $(OBJS)

$(OBJS): | $(OBJDIR)

$(OBJDIR):

mkdir $(OBJDIR)

上面这个例子中,规则会在"objdir/foo.o"文件被创建之前先创建objdir这个目录,但没有"objdir/\*.o"文件会因为objdir目录时间戳变化而被重建.

**4.3 文件名使用通配符**

单个文件名中可以使用通配符来指定许多文件.Makefile中的通配符和Bourne shell一样,包括"\*","?","[...]".例如,"\*.c"代表了当前工作目录下所有的以".c"结尾的文件.

以字符"~"开始的文件名有特殊的含义.当单独使用它或者其后跟一个斜线时,它都代表了你的home目录,例如"~/bin"代表"/home/用户名/bin".当"~"后面跟了一个单词时,代表用户名为这个单词的home目录,例如"~john/bin"代表"/home/jonh/bin".

make程序会自动对目标和依赖中的通配符进行展开,而配方中的通配符是由shell负责展开的.除此之外,其他上下文中不能直接使用通配符,必须要通过函数"wildcard"来实现通配展开的功能.

通过使用反斜线"\"转义处理之后,"\*"字符的通配特性可以被关闭.例如"foo\\*bar"指的是一个名为"foo\*bar"的文件.

**4.3.1 通配用法举例**

通配符可以被用于规则的配方中,它是由shell负责展开.例如下面这条删除所有".o"文件的规则:

clean:

rm -f \*.o

通配符也可以被用于规则的依赖中.例如一个Makefile包含了下面这条规则,执行"make print"将会打印当前目录下所有在上次打印之后修改过的".c"结尾的文件:

print: \*.c

lpr -p $?

touch print

这条规则中,"print"作为一个空目标文件使用,而自动变量"$?"表示依赖列表中被改变过的所有文件.

变量定义中使用的通配符不会被通配处理.因此如果你像下面这样写:

objects = \*.o

那么变量"objects"的值实际上就是字符串"\*.o".但是,如果你在目标或依赖中引用了这个变量,那么就会在引用的地方进行通配展开,而如果你在配方中引用了这个变量,那么shell会在执行该配方时负责进行展开.当确实需要变量"object"代表所有".o"文件列表时,应该这样写:

object := $(wildcard \*.o)

**4.3.2 通配符使用陷阱**

这里有一个使用通配符的简单例子,但这个例子并不会达到你设想的意图.你期望基于当前目录下的所有".o"文件生成可执行文件"foo",所以你这么写:

objects = \*.o

foo : $(objects)

cc -o foo $(CFLGAGS) $(objects)

变量"objects"的值实际上是字符串"\*.o".在当前目录中存在".o"文件的前提下,make程序会对依赖中的通配符进行展开,这样目标"foo"的依赖就是所有已经存在的".o"文件,只要有".o"文件改变就会重建目标"foo".

但是如果你事先删除了当前目录下的所有".o"文件,依赖中的通配符在匹配不到任何文件的情况下,就不会进行通配展开并且保留原样,这样目标"foo"的依赖就是"\*.o".因为没有"\*.o"这个文件存在,所以make程序就会返回一个类似"No rule to make target '\*.o'"的错误信息.这显然不是你想要的结果!

实际上我们是可以基于通配展开功能实现想要的结果的,只是需要使用一些高级技巧,比如使用函数wildcard或字符串替换.

**4.3.3 函数wildcard**

在规则中,通配符会被自动展开,但是在变量定义和函数参数传入过程中,通配符将失效.如果你想在这些地方实现通配展开功能,那么就需要使用wildcard函数,就像下面这样:

$(wildcard pattern...)

在Makefile中,它被展开为已经存在的, 使用空格分开的,匹配此模式的所有文件列表.如果不存在任何符合此模式的文件,函数会忽略模式字符并返回空.显然,这跟规则中的通配符在匹配失败时的行为不一样,后者采取的是保留原样而不是忽略.

wildcard函数的一个用法是获取目录下的所有c文件,就像下面这样:

$(wildcard \*.c)

通过替换c文件列表中的".c"后缀为".o"后缀,我们可以进一步得到".o"文件列表,就像下面这样:

$(patsubst %.c,%.o,$(wildcard \*.c))

因此,可以使用如下内容的Makefile来将目录下的所有的".c"文件进行编译并最后链接成为一个可执行文件:

objects := $(patsubst %.c,%.o,$(wildcard \*.c))

foo : $(objects)

cc -o foo $(objects)

这里因为利用了隐式规则来编译c程序,所以不需要书写显式规则来每个编译".o"文件.

**4.5 伪目标**

有这么一种目标,它不代表一个真正的文件名,执行make时可以指定这个目标来执行其所在规则定义的一系列配方,这种目标就叫伪目标.使用伪目标有两个原因:一个是避免跟同名文件出现冲突,另一个是提高执行 make 时的效率.

如果你要书写这么一条规则:它的配方不会创建目标文件,并且反复指定make该目标时,它的配方都会被执行.下面就是一个例子:

clean :

rm \*.o temp

由于rm命令并不会创建一个名为"clean"的文件,当前目录下通常也不会存在这个同名文件,所以每次你输入"make clean"时,规则中的rm命令总会被执行.

在这个例子中,如果当前目录下已经存在一个名为"clean"的文件时,则目标"clean"所在的规则就不会被执行.因为没有依赖,目标"clean",也就是文件"clean",总被认为是最新的,所以它的配方将不会被执行.为了避免这个问题,你需要将目标"clean"声明为伪目标,方法是将它作为特殊目标".PHONY"的依赖,如下:

.PHONY : clean

clean :

rm \*.o temp

目标"clean"声明为伪目标之后,无论在当前目录下是否存在"clean"这个文件,输入"make clean"后都会执行目标"clean"对应的配方.

伪目标在make程序执行递归的过程中同样有用.在这种情况中,Makefile通常包含了这么一个变量,该变量定义为所有需要make的子目录.处理这类场景的一种简单的方法是定义一条规则,在规则的配方中使用shell循环遍历每个子目录进行make,就像下面这样:

SUBDIRS = foo bar baz

subdirs :

for dir in $(SUBDIRS); do \

$(MAKE) -C $$dir;\

done

但这种方法存在几个问题.第一,子目录中执行make时出现的任何错误都会被忽略掉,就是说,在对某一个目录执行 make 失败以后,会继续对其他的目录进行 make.尽管可以在配方中添加出错检测并退出的shell命令,不幸的是,如果在执行 make 时使用了" -k"选项,此方法将失效.第二,更重要的一点是,这种方法使你无法利用make并行处理规则目标的功能,因为只有一条规则.

将子目录声明为伪目标(必须这样做，因为子目录显然总是存在的),就可以解决这些问题:

SUBDIRS = foo bar baz

.PHONY : subdirs $(SUBDIRS)

subidrs : $(SUBDIRS)

$(MAKE) -C $@

foo : baz

这里我们还声明了一条规则"foo : baz",这条规则用来确保只有在子目录"baz" make完毕后才会对子目录"foo"执行make.在make并行处理规则目标时,这种顺序关系声明特别重要.

make程序不会对伪目标尝试搜索隐含规则,也不关心是否存在和伪目标同名的文件,这也就是为什么将目标声明为伪目标可以提高make执行效率的原因.

伪目标通常不会作为一个真正的目标文件的依赖,这是因为如果伪目标成为一个目标文件的依赖时,每次make程序尝试刷新该目标文件时,作为依赖的伪目标,其配方必然会被执行.只要伪目标不作为一个真正目标的依赖,那么该伪目标的配方只会在make程序显式指定执行该目标时被执行.

伪目标可以有自己的依赖.当一个目录下需要创建多个可执行程序时,一种最方便的方法就是将所有程序的重建规则在一个Makefile中进行描述.因为 Makefile 中第一个目标就是默认目标,约定的做法是使用一个称为"all"的伪目标来作为默认目标,它的依赖文件就是那些需要构建的独立程序.下边就是一个例子:

all : prog1 prog2 prog3

.PHONY : all

prog1 : prog1.o utils.o

cc -o prog1 prog1.o utils.o

prog2 : prog2.o

cc -o prog2 prog2.o

prog3 : prog3.o sort.o utils.o

cc -o prog3 prog3.o sort.o utils.o

现在我们只需要简单的输入"make"就可以构建所有的程序,或者在make的命令行参数中指定要构建的程序(比如"make prog1 prog3").伪目标的特性是不会被继承的: 除非特定声明某个目标为伪目标,否则伪目标的依赖不会也变成伪目标.

当一个伪目标作为另外一个伪目标的依赖时,make程序会将该伪目标作为另外一个伪目标的子程序来处理.在下面这个例子中,输入"make clean"就会删除所有".o"文件,".diff"文件,以及"program"文件:

.PHONY : cleanall cleanobj cleandiff

cleanall : cleanobj cleandiff

rm program

cleanobj :

rm \*.o

cleanobj :

rm \*.diff

(这里似乎缺少关于上面这个例子的讲解内容)

4.6 没有配方或依赖的规则

如果一条规则没有配方或依赖,并且规则的目标不是一个存在的文件名,则执行此规则时,make 就认为它的目标已经被更新过.这样的目标在作为一个规则的依赖时,因为依赖总被认为被更新过,因此作为依赖所在的规则中定义的命令总会被执行.看一个例子:

clean : FORCE

rm $(objects)

FORCE:

这个例子中,目标"FORCE"就符合了上边的条件,所以当make执行目标"clean"所在规则时:,它的配方总会被执行.这种没有配方或依赖的特殊目标并没有一个固定的名字,只是习惯上通常命名为"FORCE".

正如你所看到的,使用"FORCE"目标的效果和将"clean"声明为伪目标的效果相同.

使用" .PHONY"方式显然更加直观高效,只不过因为其他版本的make程序不一定支持".PHONY"方式,所以许多Makefile中还是使用了"FORCE"目标.

4.7 记录事件用的空目标文件

空目标是伪目标的一种变体,它也是一个用来标识一系列配方的集合,并且只会在make命令行指定该目标时执行.和伪目标不同的是,这个目标可以是一个存在的文件,但文件的具体内容我们并不关心,通常就是一个空文件.

空目标文件只是用来记录上一次执行此规则命令的时间戳.实现的方式是在配方中执行touch命令来更新目标文件.