Makefile文件使用简要说明

hanhj

February 14, 2020

前言

Makefile是一种脚本文件，被make命令所使用，用来自动执行一些命令，从而简化人们的工作。最常见的用途是用来编译程序，比如编译c或cpp文件。当然也可以用来做其他一些自动化工作，比如编译tex文件，生成pdf文档。编译doxygen文件，生成帮助文档等等。

所以可以理解makefile文件就是一个用来实现自动化工作的脚本文件，与其他比如sh脚本文件一样的，只不过sh文件被bash所解释执行，而makefile文件被make所解释执行，二者有相同的地方，也有一些不同的地方，比如关于变量定义，使用都很相似，而在流程控制方面有些不同。此外，makefile也有一些自己的处理函数。

查看make和makefile的在线帮助可以用man make和info make来获得。

makefile中的注释是用#开头。

1 基本命令

1.1 第一个makefile

```
test:test.c
    gcc test.c -o test
```

假如我们有test.c文件，想把它编译成test程序。在命令行中，我们输入如下命令：gcc test.c，就可以了。但是，如果每次修改完test.c我们都要敲如上的字符，觉得太麻烦了，这时makefile就可以登场了，敲入如上代码，然后保存到makefile（或Makefile）文件中，这时，我们每次修改完test.c文件，直接敲make就可以得到test程序了。

注意第二行，前面需要用Tab键缩进。

解释一下，第一行test:test.c,前面的test表示要生成的目标，后面的test.c表示为了生成这个目标所要依赖的文件，就是说为了生成test这个程序，需要test.c这个文件，这句话表示的就是这种依赖关系。

第二行gcc test.c -o test表示的是为了生成这个目标所要执行的动作，其实就是我们在命令行中所敲入的命令。不过需要注意的是在makefile文件中，描述这种动作的语句需要在前面用Tab键缩进。

```
目标:依赖的文件
        为生成目标所要执行的命令
```

Makefile的基本语法就是这样。

1.2 变量

在makefile中使用变量可以替换字符，提高效率与灵活性。

1.2.1 变量定义

```
var=xxx
```

将xxx赋值给变量var。使用的时候，用$(var)或${var}，当make命令遇到该符号时，将其用定义时的字符串展开。即用xxx来代替$(var)。

var=xxx 是一种**延后展开方式**。即如果xxx中还包括变量，则make会在扫描完所有makefile文件之后,明确该变量的值后再赋值给var。此外其他几种赋值方式:

var:=xxx	立即展开方式。即如果xxx中还包括变量，则在此处立即展开该变量。
var+=xxx	加方式。在var后面添加xxx变量
var?=xxx	预定义方式。即如果var这个变量如果预先定义过了就用预先定义的值，否则用xxx

Table 1: 变量赋值

举例:

```
var=$(name)
name=john
all:
    @echo $(var)
```

make之后结果为john。因为name在后面赋值了。虽然name的赋值语句在var赋值语句之后，但是make依然会得到正确结果。

```
var:=$(name)
name=john
all:
    @echo $(var)
```

make之后的结果为空。因为make在看到:=的赋值语句后，立即对name变量展开，而此时name并没有赋值，所以var的值为空。

```
CC?=gcc
all:
    @echo $(CC)
```

make之后的结果是cc，而不是gcc。因为在linux中CC预定义为cc。

```
var=merry
var+=john
all:
    @echo $(var)
```
make之后的结果是merry john

1.2.2 变量的用途

变量可以用作表示目标文件，依赖文件，或者一些参数等等。

举例：
用作依赖文件：
```
objs=main.o test.o
test:$(objs) #test依赖于objs
    gcc $(objs) -o test
```

用作目标文件：
```
target=test
$(target):test.o    #target 是目标文件
    gcc test.o -o $(target)
```

用作参数：
cflags=-c -g #cflags是编译c文件的参数，-c表示仅编译，不生成目标，-g 表示在目标文件中加入调试信息。
```
test.o:test.c
    gcc $(cflags) test.c -o test.o
test:test.o
    gcc test.o -o test
```

1.2.3 常用的makefile中的内部变量

为了方便，在makefile中定义了一些内部变量，可以方便的使用以节省敲字符。
自动化变量：

$@	代表目标文件
$<	代表依赖关系中的首个依赖文件
$^	代表依赖关系中的所有依赖文件
$?	代表依赖关系中依赖文件中更新文件

Table 2: 自动化变量

用上面的变量改写第一个makefile：

```
test:fun1.c test.c #假设依赖两个文件
    gcc $^ -o $@
```

通配符：
makefile中的通配符*,?,[]一般用来匹配文件，与shell中的用法基本一致。

*	匹配任意字符任意次
?	匹配任意一个字符1次
[..]	匹配[...]中出现的字符1次
除此以外，还有	
%	匹配任意1到多个字符，常用于依赖规则书写

Table 3: 通配符

举例：

```
srcs=*.c #匹配当前目录下的所有.c文件
makefile=[Mm]akefile #匹配Makefile或makefile文件
main_c=main?.c #匹配mainx.c，x是任意字符。
```

默认的编译相关命令

变量名	命令	默认文件后缀名
$(CC)	cc， c程序编译器	c
$(CXX)	g++， c++程序编译器	cpp,cc,C
$(LD)	ld， 连接器	o
$(AS)	as，汇编程序编译器	s,S
...	...	...

Table 4: 默认的编译相关命令

1.3 语法

makefile中的语法包括依赖关系，命令等。
最基本的语法就是开始所说的：

```
目标:依赖的文件
        为生成目标所要执行的命令
```

1.3.1 显式依赖规则

就像开头的第一个makefile一样，我们在makefile中明显的指明目标与依赖文件之间的关系，并指明由依赖文件生成目标文件的命令。这种方式称为显式依赖。

采用这种方式的好处是依赖关系非常直观，但是也存在输入字符太多，如果项目中的文件很多，则这种人工的方式显得比较低效。而且一旦文件名发生变动，需要手动的修改makefile文件，也比较费事。

为此，我们可以创建一种规则，通过文件后缀名建立一种依赖关系。这样我们修改的就是项目中所包括的文件名，而不用修改依赖规则。下面是一个改进的makefile:

```
target=test
objs=fun.o main.o
%.o:%.c
    $(CC) -c -g $< -o $@
target:$(objs)
gcc $^ -o $@
```

这里，我们建立了一个从.c文件到.o文件的显式依赖。

1.3.2 隐式依赖规则

除了用显式规则建立依赖关系，对于makefile来说还有一种称为隐式依赖规则，即如果没有在makefile中明显的根据文件后缀名依赖关系，make命令会自动的根据一般的理解建立这种依赖关系。比如如下代码:

```
target=test
objs=fun.o main.o
target:$(objs)
gcc $^ -o $@
```

与上面一段代码所不同的是，这里没有建立.c到.o的依赖关系，但是依然能够正确执行，原因就是make命令自动建立了依赖关系。其实就是建立了如下的代码:

```
%.o:%.c
    $(CC) -c $(CFLAGS) -o $@ $<
```

CFLAGS是一个内部变量，用于在编译c程序的时候使用，需要在make命令的时候在命令行中给出，或者在makefile文件中给出。默认是空。这一点，我们在执行make命令的时候，可以通过其执行过程可以看出。

1.3.3 包含其他文件

include xxx

makefile文件可以包括其他文件，包括makefile文件。在makefile文件中包括其他文件，有利于将各部分内容分开，有利于模块化。将经常变动的部分与不变部分分开，也有利于对项目的维护。

常见的做法是将项目中全局的定义，全局的依赖规则定义在一个主文件中，其他makefile文件仅负责其本地有变化的内容。

1.3.4 搜索路径

VPATH=path
vapth patten path

设置搜索路径的目的是为makefile文件中的目标文件或依赖文件在当前目录找不到的情况下，提供一个搜索的路径。

注意，该路径并不提供在源文件中所包括的头文件的搜索路径，仅仅是为makefile中出现的文件服务的。

举例：

```
VPATH=src:include #两个以上目录用:分开
vpath %.c src #模式搜索，为.c文件提供搜索路径。
```

1.3.5 伪目标

```
.PHONY:xxx
xxx:
    command
```

伪目标不生成文件，但是往往是需要执行的动作。比如清除编译过程中的临时文件。
举例：

```
target=test
objs=fun.o main.o
$(target):$(objs)
    gcc $^ -o $@
.PHONY:clean
clean:
    rm -f $(target) $(objs)
```

1.3.6 命令的前缀

在执行的命令前可以有-,@,+前缀表示不同的执行方式。

- - :表示即使命令执行出错，也继续执行下去。
  比如

  ```
  target=test
  objs=fun.o main.o
  $(target):$(objs)
      gcc $^ -o $@
  .PHONY:clean
  clean:
     -rm -f $(target) $(objs) #这里用-rm
  ```

- @ :表示不显示执行的命令本身，而只显示执行结果。

```
target=test
objs=fun.o main.o
$(target):$(objs)
    gcc $^ -o $@
.PHONY:clean
clean:
    rm -f $(target) $(objs)
    @echo rm  $(target) $(objs) #
```

- + :不常用，表示即使make命令行中包括-n，-t，-p选项也要执行。

1.4 makefile中常用函数

makefile中的函数调用方法是:

```
$(fun,arg1,arg2...)
```

1.4.1 模式变量替换

`$(list_str:patten=replace)`
将`list_str`中匹配到`patten`的部分替换成`replace`。严格的说这个不算函数。
比如:

```
objs=func.o main.o
depend_file=$(objs,%.o=%.d)
```

结果`depend_file`为`func.d main.d`

1.4.2 subst变量替换函数

`$(subset,from,to,txt)`
将`txt`中的`from`的字符替换成`to`
举例:

```
objs=func.o main.o
depend_file=$(subst,o,d,$(objs))
```

结果`depend_file`为`func.d main.d`

1.4.3 patsubst模式变量替换函数

`$(patsubset,patten,replace,txt)`
将`txt`中匹配到`patten`的字符替换成`replace`。可以使用通配符。
举例:

```
objs=func.o main.o
depend_file=$(patsubst,%.o,%.d,$(objs))
```

结果`depend_file`为`func.d main.d`

1.5 make命令

1.5.1 并发执行

make -j[n]

j（jobs）表示多条任务同时进行,n是一个数字，表示同时并发数，比如make -j3。如果不带n将不对并发数进行限制。如果命令中有多个-j选项，则以最后一个为准。

1.5.2 进入子目录

make -C dir

make进入子目录dir执行。该命令对于组织大型项目很有效，称为递归make。例如项目主目录下有sub1,sub2两个目录。

```
subdir=sub1 sub2
all:
    for i in $(subdir) do;make -C $$(i) ;done ;
```

$$(i)第意思是由于在make命令中出现了变量$i,所以需要在该变量前面再加一个$。

2 扩展用法

2.1 自动化建立依赖关系

对于c程序来说，目标依赖源程序，.c源程序中包括头文件。当改变.c源程序的时候，make时会重构程序，但是改变.h程序往往不能自动重构，这时候就需要建立.o文件与h头文件的依赖关系，从而当h文件改变的时候也能够重构程序。

gcc编译器有一个-M选项，可以预编译c程序，建立.o文件与所依赖的.c文件以及.h文件之间的依赖关系。-MM选项可以建立.o文件与c文件，标准头文件以及用户h文件之间的依赖关系。

```
target=test
objs=fun.o main.o
depend_file=$(objs:%.o=%.d)
-include $(depend_file) #包括进来依赖关系文件
%.o:%.c
    gcc $< -o $@
%.d:%.c
    gcc -M $< -o $@ #建立.d依赖关系文件，该文件中包括.o文件与.c,.h文件的依赖关系
$(target):$(objs)
    gcc $^ -o $@
.PHONY:clean
clean:
    rm -f $(depend_file) $(objs) $(target)
```

2.2 条件判断

```
ifeq(exp1,exp2)
    command1
else
    command2
endif
```

举例:

```
CC?=gcc
ifeq($(CC),cc)
    @echo CC is cc
else
    @echo CC is gcc
endif
```

2.3 自动建立makefile

待续

3 另外一种make工具cmake

待续

References

[1] man make,info make

[2] 徐海兵.GNU make中文手册. 2004,9

List of Tables