Makefile基础

规则：

目标文件：依赖文件

TAB键　　命令

当"依赖"比"目标"新(比较时间)，或目标文件不存在，执行命令

> 假设有两个.c文件：a.c,b.c

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

执行make命令，默认生成第一个目标，每次执行时，都检查是否需要重新生成依赖

当文件很多时，不能逐一写出所有文件，而需要使用通配符%

```
test: a.o b.o
    gcc -o test $^
%.o: %.c
    gcc -c -o $@ $<
```

- $@ 目标文件
- $< 第一个依赖文件
- $^ 所有依赖文件

假想目标 .PHONY

```
test: a.o b.o
    gcc -o test $^
%.o: %.c
    gcc -c -o $@ $<
clean:
    rm *.o test
.PHONY: clean    #把clean定义为假象目标。既满足目标文件不存在
```

若没有 *.PHONY: clean* ，当前目录若存在名clean的文件，将无法执行rm *.o test语句。因为不满足命令执行的条件。

变量

- 即时变量：定义的时候就确定值
- 延时变量：使用的时候才确定值（根据表达式确定）

```
A := $(C)    #即时变量
B = $(C)     #延时变量
C = abc

all:
    @echo A = $(A)     #加上@，避免打印出命令本身
    @echo B = $(B)

#C = abc 放在后面也是一样的效果，执行make命令的时候会先从上到下读取全部makefile文件并解析
```

```
#输出：
A =
B = abc
```

- ：= 定义即时变量
- = 定义延时变量
- ？= 定义延时变量，如果是第一次定义才起效
- += 附加，它是即时变量还是延时变量取决于前面的定义

```
A := 5
A := $(A) + 5    #重新定义A为即时变量，且A等于'5 + 5'

B = 5
B = $(B) + 5 #报错，递归调用
```

可以理解为这几个运算符都是用来定义变量，每次给变量"赋值"，其实是重新定义

---

makefile里面可以包含很多函数，这些函数都是make本身实现的，下面我们来几个常用的函数。引用一个函数用"$"。

foreach函数（遍历）

foreach函数的用法流程大致如下：

```
$(foreach var,list,text)
```

1. 定义一个变量var，一个列表list，和一段文本text。
2. 对列表list中的每个元素，将其赋值给变量var，并执行文本text中的操作。
3. 将每次执行文本text中的操作得到的结果连接起来，用空格分隔，作为foreach函数的返回值。

例如：

```
$(foreach i,a b c,demo_$(i).o)
```

这个例子中，var是i，list是a b c，text是demo_$(i).o。

流程如下：

1. 对列表a b c中的第一个元素a，将其赋值给变量i，并执行文本demo_$(i).o中的操作。得到结果demo_a.o。

2. 对列表a b c中的第二个元素b，将其赋值给变量i，并执行文本demo_$(i).o中的操作。得到结果demo_b.o。
3. 对列表a b c中的第三个元素c，将其赋值给变量i，并执行文本demo_$(i).o中的操作。得到结果demo_c.o。
4. 将三次得到的结果连接起来，用空格分隔，作为foreach函数的返回值。即demo_a.o demo_b.o demo_c.o。

filter/filter-out函数（过滤）

filter函数可以用来从一个列表中筛选出符合某些模式的字符串。它的语法是：

```
$(filter pattern...,text)
```

其中pattern...是一个或多个用空格分隔的模式，text是一个空格分隔的列表。例如：

```
$(filter %.c %.s,foo.c bar.c baz.s ugh.h)
```

这个例子会从列表foo.c bar.c baz.s ugh.h中筛选出以.c或.s结尾的字符串，返回值是foo.c bar.c baz.s。

流程如下：

1. 定义一个或多个模式pattern...，和一个列表text。
2. 对列表text中的每个元素，检查它是否匹配任意一个模式pattern...。
3. 如果匹配，则保留该元素；如果不匹配，则丢弃该元素。
4. 将保留下来的元素连接起来，用空格分隔，作为filter函数的返回值。

filter-out函数与filter函数相反，它可以用来从一个列表中排除掉符合某些模式的字符串。它的语法是：

```
$(filter-out pattern...,text)
```

其中pattern...和text与filter函数相同。例如：

```
$(filter-out %.h,foo.c bar.c baz.s ugh.h)
```

这个例子会从列表foo.c bar.c baz.s ugh.h中排除掉以.h结尾的字符串，返回值是foo.c bar.c baz.s。

流程如下：

1. 定义一个或多个模式pattern...，和一个列表text。
2. 对列表text中的每个元素，检查它是否匹配任意一个模式pattern...。
3. 如果匹配，则丢弃该元素；如果不匹配，则保留该元素。
4. 将保留下来的元素连接起来，用空格分隔，作为filter-out函数的返回值。

Wildcard函数（通配符）

Wildcard函数可以用来在Makefile中进行通配符匹配，它的语法是：

```
$(wildcard pattern...)
```

其中pattern...是一个或多个用空格分隔的文件名模式，例如*.c或foo?.txt。例如：

```
$(wildcard *.c)
```

这个例子会匹配当前目录下所有以.c结尾的文件，并返回一个空格分隔的文件名列表。

流程如下：

1. 定义一个或多个文件名模式pattern...。
2. 对当前目录下的每个文件，检查它是否匹配任意一个模式pattern...。
3. 如果匹配，则保留该文件名；如果不匹配，则丢弃该文件名。
4. 将保留下来的文件名连接起来，用空格分隔，作为Wildcard函数的返回值。

注意：如果没有任何文件名匹配某个模式，则该模式本身会作为返回值的一部分。例如：

```
$(wildcard foo*.c bar*.c)
```

如果当前目录下有foo1.c和foo2.c，但没有以bar开头并以.c结尾的文件，则返回值是foo1.c foo2.c bar*.c。

patsubst函数（模式替换）

patsubst函数可以用来在Makefile中进行模式替换，它的语法是：

```
$(patsubst pattern,replacement,text)
```

其中pattern是一个文件名模式，replacement是一个替换字符串，text是一个空格分隔的列表。例如：

```
$(patsubst %.c,%.o,foo.c bar.c baz.c)
```

这个例子会将列表foo.c bar.c baz.c中所有以.c结尾的文件名替换成以.o结尾的文件名，并返回一个空格分隔的文件名列表，即foo.o bar.o baz.o。

流程如下：

1. 定义一个文件名模式pattern，和一个替换字符串replacement。
2. 对列表text中的每个元素，检查它是否匹配模式pattern。
3. 如果匹配，则将该元素按照replacement进行替换；如果不匹配，则保留该元素不变。
4. 将处理后的元素连接起来，用空格分隔，作为patsubst函数的返回值。

注意：如果replacement中包含%字符，则它表示匹配到pattern中%字符后面的部分。例如：

```
$(patsubst %.c,%_test.c,foo.c bar.c baz.c)
```

这个例子会将列表foo.c bar.c baz.c中所有以.c结尾的文件名替换成以_test.c结尾的文件名，并返回一个空格分隔的文件名列表，即foo_test.c bar_test.c baz_test.c。

更新头文件的问题

按照上面的写法，更新头文件并不能重新生成程序，因为头文件不在依赖中

```
test: a.o b.o
    gcc -o test $^

b.o : b.c b.h   #c.h修改后，因为后面没接命令，它会往下找到能生成目标c.o的命令(目标一致即可)

%.o : %.c
    gcc -c -o $@ $< #找到命令
clean:
    rm *.o test
.PHONY: clean
```

自动生成依赖

- gcc -M c.c 打印出依赖
- gcc -M -MF c.d c.c 把依赖文件写入c.d
- gcc -c -o c.o c.c -MD -MF c.d 编译c.o,把依赖写入c.d

```
objs = a.o b.o c.o

dep_files := $(patsubst %,.%.d, $(objs))
dep_files := $(wildcard $(dep_files))   #找到现有的依赖文件

test: $(objs)
    gcc -o test $^

ifneq ($(dep_files),)
include $(dep_files)    #如果dep_files为空，包含会报错
endif

%.o : %.c
    gcc -c -o $@ $< -MD -MF .$@.d

clean:
    rm *.o test

distclean:
    rm $(dep_files)

.PHONY: clean
```

编译参数 CFLAGS

```
#假设头文件放在当前目录下的include目录下，并且希望把所有警告当成错误处理

CFLAGS = -Werror -I include    #CFLAGS是一个约定俗称的变量，当然可以使用别的名字

.....

%.o : %.c
    gcc $(CFLAGS) -c -o $@ $< -MD -MF .$@.d
```