# 1、书接上回，Makefile讲解

## 1.1 Makefile文件中的基本语法构造

```
0》Makefile中的基本语法构成
    规则    --》 Makefile的规则
    变量    --》 变量声明
    条件执行   --》 条件执行
    文本、文件名处理函数   --》 函数
    文件包含   --》 可以使用include包含其他的Makefile
    注释   --》 注释
1》规则
    构成Makefile的基本单元、构成依赖关系的核心部件，
    其它内容可以看作为规则服务
2》变量
    定义变量，变量没有数据类型，都当成字符串处理，使用：
    $(VAR)、${VAR}
    可以让Makefile更加灵活
3》条件执行
    根据某一变量的值来控制make执行或者忽略Makefile的某一部分

4》函数
    文本处理函数：字符串替换、查找、过滤、排序、统计等
    文件名处理函数：取目录/文件名、前后缀、加前缀/后缀、单词连接等函数
    其它常用函数：if函数、shell函数、foreach函数

5》文件包含
    类似于C语言的#include，使用include命令
6》注释
    使用#开头，表示注释
```

## 1.2 规则

```
1. 语法格式
    目标:依赖
        shell命令

2. 目标：
    在一个规则中目标不可以省略，一个规则中可以有多个不同的目标，
```

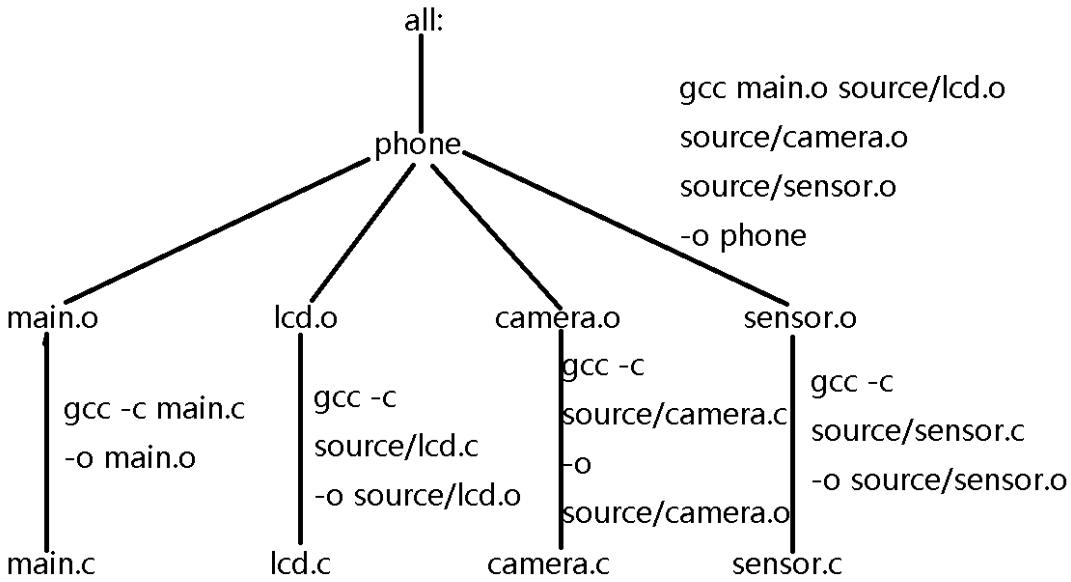
```
    多个不同的目标具有相同的依赖及shell命令
    比如：
    clean distclean:
        rm *.o  stu_manager

3. 依赖：
    一个规则中，可以有多个不同的依赖，只有依赖都存在时，
    目标和依赖对应的关系才会成立。
    比如：
    stu_manager:main.o student.o
        gcc main.o student.o -o stu_manager

    一个规则中，目标可以没有依赖，只是为了完成某种特定的操作。
    比如：
    clean distclean:
        rm *.o  stu_manager
    install:
        cp  stu_manager /mnt/hgfs/share

4. shell命令
    shell命令必须使用tab键开头，不可以使用4个空格代替tab键。
    一个规则中可以省略shell命令，只是为了表示某种依赖的关系。
    比如：
    all:stu_manager
```

# 1.3 Makefile文件中变量的使用

```
1. Makefile文件中定义的变量都是字符串

2. 定义变量的格式
    变量名 = 变量的初始值
    变量名 := 变量的初始值
    变量名 += 变量的初始值
    变量名 ?= 变量的初始值

3. Makefile文件中变量的引用
    $(变量名)       ----> 注：()不可以省略

4. = 赋值
    =    :  延迟赋值，当使用此变量时，才会对变量进行赋值操作

5. := 赋值
    :=   :  立即赋值

6. += 赋值
    +=   :  追加赋值，在变量原有的值的基础之上进行追加赋值，
             追加赋值的字符串，以空格隔开。
    比如：
    str1 = hello
    str1 += world

    all:
        echo $(str1)

```

```
    输出结果：hello world
7. ?= 赋值
    ?=    : 条件赋值，判断变量是否有初始值，如果有值就不赋值，
            如果没有初始值，才会进行赋值的操作。
    比如：
    str1 := hello
    str1 ?= world

    all:
        echo $(str1)

    输出结果：hello

    str1 ?= world

    all:
        echo $(str1)
    输出结果：world

8. 局部变量
    $@  : 目标对应的字符串
    $^  : 所有的依赖对应的字符串
    $<  : 第一个依赖对应的字符串

9. 通配符的使用
    *   :  一般用在规则中的命令中， 比如rm  *.o
    %   :  模式匹配，一般用在目标和依赖中，比如%.o:%.c
```

```
phone文件夹中，有以下文件：
├── include
|   ├── camera.h
|   ├── lcd.h
|   └── sensor.h
├── main.c
├── Makefile
└── source
    ├── camera.c
    ├── lcd.c
    └── sensor.c
```




```
最基本，最容易理解的Makefile文件

all:phone

phone:main.o source/lcd.o source/camera.o source/sensor.o
    gcc main.o source/lcd.o source/camera.o source/sensor.o -o phone
main.o:main.c
    gcc -c main.c -o main.o
source/lcd.o:source/lcd.c
    gcc -c source/lcd.c -o source/lcd.o
source/camera.o:source/camera.c
    gcc -c source/camera.c -o source/camera.o
source/sensor.o:source/sensor.c
    gcc -c source/sensor.c -o source/sensor.o

clean distclean:
    rm phone  *.o source/*.o
```

```
在makefile文件中，引入变量

OBJs += main.o
OBJs += source/lcd.o
OBJs += source/camera.o
OBJs += source/sensor.o
CC := gcc
CFLAGS := -c
EXE = phone

all:$(EXE)

$(EXE):$(OBJs)
    $(CC) $(OBJs) -o $(EXE)
main.o:main.c
    $(CC) $(CFLAGS) main.c -o main.o
source/lcd.o:source/lcd.c
    $(CC) $(CFLAGS) source/lcd.c -o source/lcd.o
source/camera.o:source/camera.c
```

```
    $(CC) $(CFLAGS) source/camera.c -o source/camera.o
source/sensor.o:source/sensor.c
    $(CC) $(CFLAGS) source/sensor.c -o source/sensor.o

clean distclean:
    rm $(EXE)  *.o source/*.o
```

在makefile中，引入$@ $^ $<变量，

```
OBJs += main.o
OBJs += source/lcd.o
OBJs += source/camera.o
OBJs += source/sensor.o
CC := gcc
CFLAGS := -c
EXE = phone

all:$(EXE)

$(EXE):$(OBJs)
    $(CC) $^ -o $@
main.o:main.c
    $(CC) $(CFLAGS) $< -o $@
source/lcd.o:source/lcd.c
    $(CC) $(CFLAGS) $< -o $@
source/camera.o:source/camera.c
    $(CC) $(CFLAGS) $< -o $@
source/sensor.o:source/sensor.c
    $(CC) $(CFLAGS) $< -o $@

clean distclean:
    rm $(EXE)  *.o source/*.o
```

在makefile中，引入通配符* %，

```
OBJs += main.o
OBJs += source/lcd.o
OBJs += source/camera.o
OBJs += source/sensor.o
CC := gcc
CFLAGS := -c
EXE = phone

all:$(EXE)

$(EXE):$(OBJs)
    $(CC) $^ -o $@
# 当第一个%表示main字符串时，后边的%也将表示main字符串
# 当第一个%表示source/lcd字符串时，后边的%也将表示source/lcd字符串
%.o:%.c
    $(CC) $(CFLAGS) $< -o $@

clean distclean:
    rm $(EXE)  *.o source/*.o
```

```
练习题：
改造学生成绩管理系统的makefile文件，
1> 引入变量
CC := gcc
CFLAGS := -c
EXE = stu_manager
OBJs += main.o
OBJs += student.o
# 在规则中，规则可以没有命令，只是为了表示某种依赖的关系
all:$(EXE)

$(EXE):$(OBJs)
    $(CC) $(OBJs) -o $(EXE)

main.o:main.c
    $(CC) $(CFLAGS) main.c -o main.o

student.o:student.c
    $(CC) $(CFLAGS) student.c -o student.o

# 一个规则中可以有多个目标，多个目标具有共同的依赖及命令
clean distclean:
    rm *.o  $(EXE)

2> 引入局部变量$@ $^ $<
CC := gcc
CFLAGS := -c
EXE = stu_manager
OBJs += main.o
OBJs += student.o
# 在规则中，规则可以没有命令，只是为了表示某种依赖的关系
all:$(EXE)

$(EXE):$(OBJs)
    $(CC) $^ -o $@

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

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

# 一个规则中可以有多个目标，多个目标具有共同的依赖及命令
clean distclean:
    rm *.o  $(EXE)

3> 引入通配符* %
CC := gcc
CFLAGS := -c
EXE = stu_manager
OBJs += main.o
OBJs += student.o
# 在规则中，规则可以没有命令，只是为了表示某种依赖的关系
all:$(EXE)

```

```
$(EXE):$(OBJs)
    $(CC) $^ -o $@

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

# 一个规则中可以有多个目标，多个目标具有共同的依赖及命令
clean distclean:
    rm *.o  $(EXE)

```

# 1.4 Makefile文件中的函数

## 1.4.1 函数1：wildcard

```
$(wildcard PATTERN)

函数名称：获取匹配模式文件名函数–wildcard
函数功能：列出当前目录下所有符合模式“PATTERN”格式的文件名。
返回值：空格分割的、存在当前目录下的所有符合模式“PATTERN”的文件名。
函数说明：“PATTERN”使用shell可识别的通配符，包括“?”（单字符）、“*”（多字符）等。
示例：
    $(wildcard *.c)
    返回值为当前目录下所有.c 源文件列表。
```

## 1.4.2 函数2：patsubst

```
$(patsubst PATTERN,REPLACEMENT,TEXT)
函数名称：模式替换函数–patsubst。
函数功能：搜索“TEXT”中以空格分开的单词，将符合模式“PATTERN”替换为“REPLACEMENT”。
    参数“PATTERN”中可以使用模式通配符“%”来代表一个单词中的若干字符。
    如果参数“REPLACEMENT”中也包含一个“%”，那么“REPLACEMENT”中的“%”将是“PATTERN”中
    的那个“%”所代表的字符串。在“PATTERN”和“REPLACEMENT”中，
    只有第一个“%”被作为模式字符来处理，之后出现的不再作模式字符（作为一个字符）。
    在参数中如果需要将第一个出现的“%”作为字符本身而不作为模式字符时，
    可使用反斜杠“\”进行转义处理

返回值：替换后的新字符串。
函数说明：参数“TEXT”单词之间的多个空格在处理时被合并为一个空格，并忽略前导和结尾空格。

示例：
$(patsubst %.c, %.o, x.c.c bar.c)
把字串串“x.c.c bar.c”中以.c 结尾的单词替换成以.o 结尾的字符。
函数的返回结果是“x.c.o bar.o”
```

```
OBJSs = $(wildcard *.c)
OBJSs += $(wildcard source/*.c)
OBJs = $(patsubst %.c, %.o, $(OBJSs))

CC := gcc
CFLAGS := -c
```

```
EXE = phone

all:$(EXE)

$(EXE):$(OBJs)
    $(CC) $^ -o $@
# 当第一个%表示main字符串时，后边的%也将表示main字符串
# 当第一个%表示source/lcd字符串时，后边的%也将表示source/lcd字符串
%.o:%.c
    $(CC) $(CFLAGS) $< -o $@

clean distclean:
    rm $(EXE)  *.o source/*.o

var:
    echo $(OBJSs)
    echo $(OBJs)
```

## 1.5 条件执行

```
1. 关键字
    ifeq、else、endif
    ifneq、else、endif

2. 使用
   条件语句从ifeq开始，左括号与关键字(ifeq)用空格隔开
   括号内的左右不可以有空格，逗号两边可以有空格
   语法格式:
    ifeq (变量/字符串,变量/字符串)
        Makefile语句
    else
        Makefile语句
    endif
```

```
OBJSs = $(wildcard *.c)
OBJSs += $(wildcard source/*.c)
OBJs = $(patsubst %.c, %.o, $(OBJSs))
# gcc :  编译的程序为x86-64的可执行程序
# arm-linux-gnueabihf-gcc : 交叉编译器
#        可以编译生成arm的可执行程序

# ARCH=x86-64:编译生成x86-64的可执行程序
# ARCH=arm:编译生成arm可执行程序
ifeq ($(ARCH),x86-64)
    CC := gcc
else
    CC := arm-linux-gnueabihf-gcc
endif
CFLAGS := -c
EXE = phone

all:$(EXE)

$(EXE):$(OBJs)
```

```
    $(CC) $^ -o $@
# 当第一个%表示main字符串时，后边的%也将表示main字符串
# 当第一个%表示source/lcd字符串时，后边的%也将表示source/lcd字符串
%.o:%.c
    $(CC) $(CFLAGS) $< -o $@

clean distclean:
    rm $(EXE)  *.o source/*.o

var:
    echo $(OBJSs)
    echo $(OBJs)
```

```
linux@ubuntu:phone$ make ARCH=x86-64
gcc -c main.c -o main.o
gcc -c source/lcd.c -o source/lcd.o
gcc -c source/sensor.c -o source/sensor.o
gcc -c source/camera.c -o source/camera.o
gcc main.o source/lcd.o source/sensor.o source/camera.o -o phone
linux@ubuntu:phone$ file phone
phone: ELF 64-bit LSB shared object, x86-64, version 1 (SYSV), dyn
amically linked, interpreter /lib64/ld-linux-x86-64.so.2, BuildID[
sha1]=e144df7a63fef0878d091b61eeee95822b5aa56e, for GNU/Linux 3.2.
0, not stripped
linux@ubuntu:phone$
```

执行make命令时，可以给变量赋值

```
linux@ubuntu:phone$ make ARCH=arm
arm-linux-gnueabihf-gcc -c main.c -o main.o
arm-linux-gnueabihf-gcc -c source/lcd.c -o source/lcd.o
arm-linux-gnueabihf-gcc -c source/sensor.c -o source/sensor.o
arm-linux-gnueabihf-gcc -c source/camera.c -o source/camera.o
arm-linux-gnueabihf-gcc main.o source/lcd.o source/sensor.o source
/camera.o -o phone
linux@ubuntu:phone$ file phone
phone: ELF 32-bit LSB executable, ARM, EABI5 version 1 (SYSV), dyn
amically linked, interpreter /lib/ld-linux-armhf.so.3, for GNU/Lin
ux 3.2.0, BuildID[sha1]=3c75f0c3261f59e7acfa028284a7dacc0b915662,
with debug_info, not stripped
linux@ubuntu:phone$
```

执行make时给变量赋值

# 1.6 Makefile执行过程

```
1》执行过程
    进入编译目录，执行make命令
    阶段1：make命令解析当前目录下的Makefile文件，进行依赖关系解析
    阶段2：运行命令
    1.1> 依赖关系解析阶段
        解析Makefile，建立依赖关系树
        控制解析过程：引入其他Makefile、变量展开、条件执行
        生成依赖关系树

    1.2> 命令执行阶段
        把解析生成的依赖关系树加载到内存
        按照依赖关系，执行命令按顺序生成这些文件
        再次编译Make会检查文件时间戳，判断是否过期,若无过期，不再编译
        若文件有更新，则依赖该文件的所有依赖关系上的目标重新更新、编译生成

```

```
2》Make执行结果
    Make的退出码
    0：表示成功执行
    1：运行错误，make返回错误码
```