

## 1、书接上回，Makefile讲解

### 1.1 Makefile文件中的基本语法构造

#### 1.2 规则

#### 1.3 Makefile文件中变量的使用

#### 1.4 Makefile文件中的函数

##### 1.4.1 函数1: wildcard

##### 1.4.2 函数2: patsubst

#### 1.5 条件执行

#### 1.6 Makefile执行过程

# 1、书接上回，Makefile讲解

## 1.1 Makefile文件中的基本语法构造

```
1 0》Makefile中的基本语法构成
2 规则 --》 Makefile的规则
3 变量 --》 变量声明
4 条件执行 --》 条件执行
5 文本、文件名处理函数 --》 函数
6 文件包含 --》 可以使用include包含其他的Makefile
7 注释 --》 注释
8 1》规则
9 构成Makefile的基本单元、构成依赖关系的核心部件，
10 其它内容可以看作规则服务
11 2》变量
12 定义变量，变量没有数据类型，都当成字符串处理，使用：
13 $(VAR)、${VAR}
14 可以让Makefile更加灵活
15 3》条件执行
16 根据某一变量的值来控制make执行或者忽略Makefile的某一部分
17
18 4》函数
19 文本处理函数：字符串替换、查找、过滤、排序、统计等
20 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
21 其它常用函数：if函数、shell函数、foreach函数
22
23 5》文件包含
24 类似于C语言的#include，使用include命令
25 6》注释
26 使用#开头，表示注释
```

## 1.2 规则

```
1 1. 语法格式
2 目标:依赖
3     shell命令
4
5 2. 目标:
6 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，
```

```

7      多个不同的目标具有相同的依赖及shell命令
8      比如:
9      clean distclean:
10         rm *.o stu_manager
11
12 3. 依赖:
13     一个规则中, 可以有多个不同的依赖, 只有依赖都存在时,
14     目标和依赖对应的关系才会成立。
15     比如:
16     stu_manager:main.o student.o
17         gcc main.o student.o -o stu_manager
18
19     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。
20     比如:
21     clean distclean:
22         rm *.o stu_manager
23     install:
24         cp stu_manager /mnt/hgfs/share
25
26 4. shell命令
27     shell命令必须使用tab键开头, 不可以使用4个空格代替tab键。
28     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。
29     比如:
30     all:stu_manager

```

## 1.3 Makefile文件中变量的使用

```

1 1. Makefile文件中定义的变量都是字符串
2
3 2. 定义变量的格式
4     变量名 = 变量的初始值
5     变量名 := 变量的初始值
6     变量名 += 变量的初始值
7     变量名 ?= 变量的初始值
8
9 3. Makefile文件中变量的引用
10     $(变量名)      ----> 注: ()不可以省略
11
12 4. = 赋值
13     =      :   延迟赋值, 当使用此变量时, 才会对变量进行赋值操作
14
15 5. := 赋值
16     :=     :   立即赋值
17
18 6. += 赋值
19     +=     :   追加赋值, 在变量原有的值的基础之上进行追加赋值,
20               追加赋值的字符串, 以空格隔开。
21
22     比如:
23     str1 = hello
24     str1 += world
25
26     all:
27         echo $(str1)

```

```

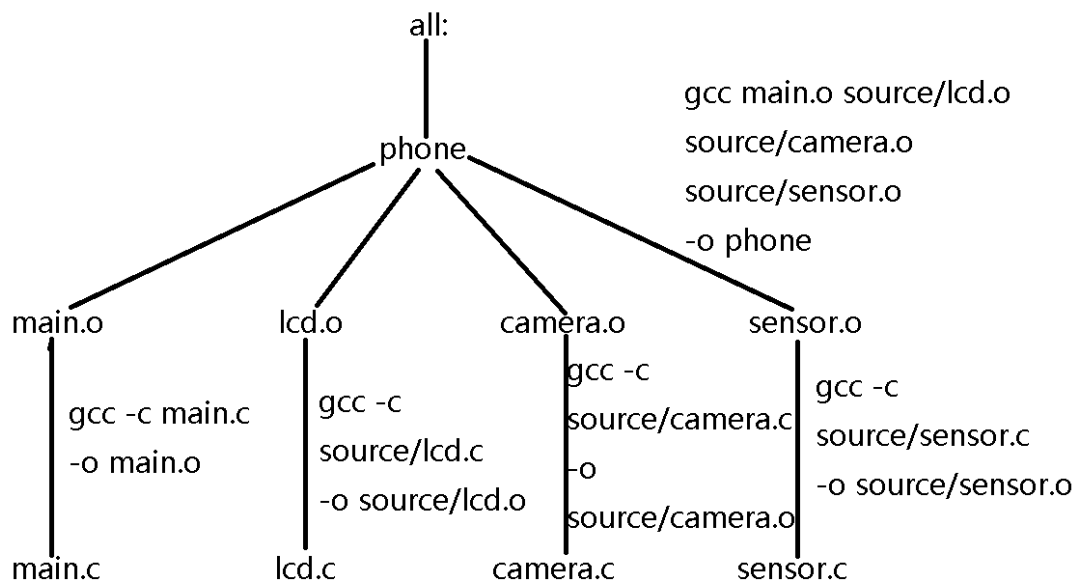
28     输出结果: hello world
29 7. ?= 赋值
30     ?=      : 条件赋值, 判断变量是否有初始值, 如果有值就不赋值,
31               如果没有初始值, 才会进行赋值的操作。
32     比如:
33     str1 := hello
34     str1 ?= world
35
36     all:
37         echo $(str1)
38
39     输出结果: hello
40
41     str1 ?= world
42
43     all:
44         echo $(str1)
45     输出结果: world
46
47 8. 局部变量
48     $@   : 目标对应的字符串
49     $^   : 所有的依赖对应的字符串
50     $<   : 第一个依赖对应的字符串
51
52 9. 通配符的使用
53     *     : 一般用在规则中的命令中, 比如rm *.o
54     %     : 模式匹配, 一般用在目标和依赖中, 比如%.o:%.c

```

```

1  phone文件夹中, 有以下文件:
2  └─ include
3  │   └─ camera.h
4  │   └─ lcd.h
5  │   └─ sensor.h
6  └─ main.c
7  └─ Makefile
8  └─ source
9      └─ camera.c
10     └─ lcd.c
11     └─ sensor.c

```



```

1 最基本，最容易理解的Makefile文件
2
3  all:phone
4
5  phone:main.o source/lcd.o source/camera.o source/sensor.o
6      gcc main.o source/lcd.o source/camera.o source/sensor.o -o phone
7  main.o:main.c
8      gcc -c main.c -o main.o
9  source/lcd.o:source/lcd.c
10     gcc -c source/lcd.c -o source/lcd.o
11 source/camera.o:source/camera.c
12     gcc -c source/camera.c -o source/camera.o
13 source/sensor.o:source/sensor.c
14     gcc -c source/sensor.c -o source/sensor.o
15
16 clean distclean:
17     rm phone *.o source/*.o

```

```

1 在makefile文件中，引入变量
2
3  OBJs += main.o
4  OBJs += source/lcd.o
5  OBJs += source/camera.o
6  OBJs += source/sensor.o
7  CC := gcc
8  CFLAGS := -c
9  EXE = phone
10
11 all:$(EXE)
12
13 $(EXE):$(OBJs)
14     $(CC) $(OBJs) -o $(EXE)
15 main.o:main.c
16     $(CC) $(CFLAGS) main.c -o main.o
17 source/lcd.o:source/lcd.c
18     $(CC) $(CFLAGS) source/lcd.c -o source/lcd.o
19 source/camera.o:source/camera.c

```

```

20     $(CC) $(CFLAGS) source/camera.c -o source/camera.o
21 source/sensor.o:source/sensor.c
22     $(CC) $(CFLAGS) source/sensor.c -o source/sensor.o
23
24 clean distclean:
25     rm $(EXE) *.o source/*.o

```

```

1  在makefile中，引入$@ $^ $<变量，
2
3  OBJs += main.o
4  OBJs += source/lcd.o
5  OBJs += source/camera.o
6  OBJs += source/sensor.o
7  CC := gcc
8  CFLAGS := -c
9  EXE = phone
10
11 all:$(EXE)
12
13 $(EXE):$(OBJs)
14     $(CC) $^ -o $@
15 main.o:main.c
16     $(CC) $(CFLAGS) $< -o $@
17 source/lcd.o:source/lcd.c
18     $(CC) $(CFLAGS) $< -o $@
19 source/camera.o:source/camera.c
20     $(CC) $(CFLAGS) $< -o $@
21 source/sensor.o:source/sensor.c
22     $(CC) $(CFLAGS) $< -o $@
23
24 clean distclean:
25     rm $(EXE) *.o source/*.o

```

```

1  在makefile中，引入通配符* %,
2
3  OBJs += main.o
4  OBJs += source/lcd.o
5  OBJs += source/camera.o
6  OBJs += source/sensor.o
7  CC := gcc
8  CFLAGS := -c
9  EXE = phone
10
11 all:$(EXE)
12
13 $(EXE):$(OBJs)
14     $(CC) $^ -o $@
15 # 当第一个%表示main字符串时，后边的%也将表示main字符串
16 # 当第一个%表示source/lcd字符串时，后边的%也将表示source/lcd字符串
17 %.o:%.c
18     $(CC) $(CFLAGS) $< -o $@
19
20 clean distclean:
21     rm $(EXE) *.o source/*.o

```

```
1  练习题：
2  改造学生成绩管理系统的makefile文件，
3  1> 引入变量
4  CC := gcc
5  CFLAGS := -c
6  EXE = stu_manager
7  OBJs += main.o
8  OBJs += student.o
9  # 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
10 all:$(EXE)
11
12 $(EXE):$(OBJs)
13     $(CC) $(OBJs) -o $(EXE)
14
15 main.o:main.c
16     $(CC) $(CFLAGS) main.c -o main.o
17
18 student.o:student.c
19     $(CC) $(CFLAGS) student.c -o student.o
20
21 # 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
22 clean distclean:
23     rm *.o $(EXE)
24
25 2> 引入局部变量$@ $^ $<
26 CC := gcc
27 CFLAGS := -c
28 EXE = stu_manager
29 OBJs += main.o
30 OBJs += student.o
31 # 在规则中，规则可以没有命令，只是为了表示某种依赖的关系
32 all:$(EXE)
33
34 $(EXE):$(OBJs)
35     $(CC) $^ -o $@
36
37 main.o:main.c
38     $(CC) $(CFLAGS) $< -o $@
39
40 student.o:student.c
41     $(CC) $(CFLAGS) $< -o $@
42
43 # 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
44 clean distclean:
45     rm *.o $(EXE)
46
47 3> 引入通配符* %
48 CC := gcc
49 CFLAGS := -c
50 EXE = stu_manager
51 OBJs += main.o
52 OBJs += student.o
53 # 在规则中，规则可以没有命令，只是为了表示某种依赖的关系
54 all:$(EXE)
55
```

```

56 $(EXE):$(OBJS)
57     $(CC) $^ -o $@
58
59 %.o:%.c
60     $(CC) $(CFLAGS) $< -o $@
61
62 # 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
63 clean distclean:
64     rm *.o $(EXE)
65

```

## 1.4 Makefile文件中的函数

### 1.4.1 函数1: wildcard

```

1 $(wildcard PATTERN)
2
3 函数名称：获取匹配模式文件名函数-wildcard
4 函数功能：列出当前目录下所有符合模式“PATTERN”格式的文件名。
5 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。
6 函数说明：“PATTERN”使用shell可识别的通配符，包括“?”（单字符）、“*”（多字符）等。
7 示例：
8     $(wildcard *.c)
9     返回值为当前目录下所有.c 源文件列表。

```

### 1.4.2 函数2: patsubst

```

1 $(patsubst PATTERN,REPLACEMENT,TEXT)
2 函数名称：模式替换函数-patsubst。
3 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”。
4     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。
5     如果参数“REPLACEMENT”中也包含一个“%”，那么“REPLACEMENT”中的“%”将是“PATTERN”
6     中
7     的那个“%”所代表的字符串。在“PATTERN”和“REPLACEMENT”中，
8     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）。
9     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，
10    可使用反斜杠“\”进行转义处理
11
12 返回值：替换后的新字符串。
13
14 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。
15
16 示例：
17 $(patsubst %.c, %.o, x.c.c bar.c)
18 把字符串“x.c.c bar.c”中以.c 结尾的单词替换成以.o 结尾的字符。
19 函数的返回结果是“x.c.o bar.o”

```

```

1 OBJSs = $(wildcard *.c)
2 OBJSs += $(wildcard source/*.c)
3 OBJs = $(patsubst %.c, %.o, $(OBJSs))
4
5 CC := gcc
6 CFLAGS := -c

```

```

7  EXE = phone
8
9  all:$(EXE)
10
11 $(EXE):$(OBJS)
12     $(CC) $^ -o $@
13 # 当第一个%表示main字符串时，后边的%也将表示main字符串
14 # 当第一个%表示source/lcd字符串时，后边的%也将表示source/lcd字符串
15 %.o:%.c
16     $(CC) $(CFLAGS) $< -o $@
17
18 clean distclean:
19     rm $(EXE) *.o source/*.o
20
21 var:
22     echo $(OBJSS)
23     echo $(OBJS)

```

## 1.5 条件执行

```

1  1. 关键字
2     ifeq、else、endif
3     ifneq、else、endif
4
5  2. 使用
6     条件语句从ifeq开始，左括号与关键字(ifeq)用空格隔开
7     括号内的左右不可以有空格，逗号两边可以有空格
8     语法格式：
9     ifeq (变量/字符串,变量/字符串)
10         Makefile语句
11     else
12         Makefile语句
13     endif

```

```

1  OBJSS = $(wildcard *.c)
2  OBJSS += $(wildcard source/*.c)
3  OBJS = $(patsubst %.c, %.o, $(OBJSS))
4  # gcc : 编译的程序为x86-64的可执行程序
5  # arm-linux-gnueabi-hf-gcc : 交叉编译器
6  #     可以编译生成arm的可执行程序
7
8  # ARCH=x86-64:编译生成x86-64的可执行程序
9  # ARCH=arm:编译生成arm可执行程序
10 ifeq ($(ARCH),x86-64)
11     CC := gcc
12 else
13     CC := arm-linux-gnueabi-hf-gcc
14 endif
15 CFLAGS := -c
16 EXE = phone
17
18 all:$(EXE)
19
20 $(EXE):$(OBJS)

```



```

21 $(CC) $^ -o $@
22 # 当第一个%表示main字符串时, 后边的%也将表示main字符串
23 # 当第一个%表示source/lcd字符串时, 后边的%也将表示source/lcd字符串
24 %.o:%.c
25 $(CC) $(CFLAGS) $< -o $@
26
27 clean distclean:
28     rm $(EXE) *.o source/*.o
29
30 var:
31     echo $(OBJSS)
32     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[
shal]=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-gnueabi-gcc -c main.c -o main.o
arm-linux-gnueabi-gcc -c source/lcd.c -o source/lcd.o
arm-linux-gnueabi-gcc -c source/sensor.c -o source/sensor.o
arm-linux-gnueabi-gcc -c source/camera.c -o source/camera.o
arm-linux-gnueabi-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[shal]=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 1》执行过程
- 2 进入编译目录, 执行make命令
- 3 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
- 4 阶段2: 运行命令
- 5 1.1> 依赖关系解析阶段
- 6 解析Makefile, 建立依赖关系树
- 7 控制解析过程: 引入其他Makefile、变量展开、条件执行
- 8 生成依赖关系树
- 9
- 10 1.2> 命令执行阶段
- 11 把解析生成的依赖关系树加载到内存
- 12 按照依赖关系, 执行命令按顺序生成这些文件
- 13 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
- 14 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
- 15

16	2》 <b>Make</b> 执行结果
17	<b>Make</b> 的退出码
18	0: 表示成功执行
19	1: 运行错误， <b>make</b> 返回错误码