

Makefile简介

一、Makefile入门

- make命令执行时，需要一个 Makefile 文件，以告诉make命令需要怎么样的去编译和链接程序。
- Makefile的简单用法：
 - 目标文件：依赖文件
 - [TAB]命令
- 说明：
 - [TAB]：是你键盘左边tab键；
 - 目标文件：要生成的文件；
 - 依赖文件：生成目标文件要依赖的文件；
 - 命令：执行的命令，生成目标文件。

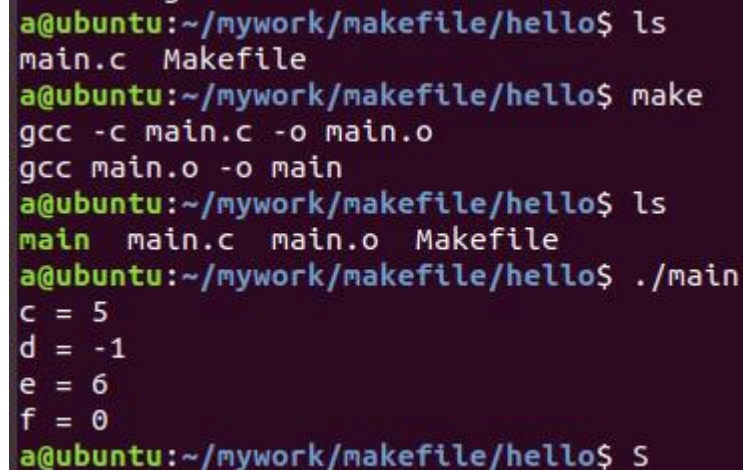
main.c源代码:

```
#include <stdio.h>
int add(int a, int b){
    return a+b;}
int sub(int a, int b){
    return a-b;}
int mul(int a, int b){
    return a*b;}
int div(int a, int b){
    return a/b;}
int main(int argc, char* argv[]){
    int m = 2;
    int n = 3;
    int c,d,e,f;
    c = add(m, n);
    printf("c = %d\n", c);
    d = sub(m, n);
    printf("d = %d\n", d);
    e = mul(m, n);
    printf("e = %d\n", e);
    f = div(m, n);
    printf("f = %d\n", f);
    return 0;
}
```

Makefile文件:

```
main: main.o
    gcc main.o -o main
main.o: main.c
    gcc -c main.c -o main.o
```

问题: 请描述下Makefile中文件的依赖关系



```
a@ubuntu:~/mywork/makefile/hello$ ls
main.c  Makefile
a@ubuntu:~/mywork/makefile/hello$ make
gcc -c main.c -o main.o
gcc main.o -o main
a@ubuntu:~/mywork/makefile/hello$ ls
main  main.c  main.o  Makefile
a@ubuntu:~/mywork/makefile/hello$ ./main
c = 5
d = -1
e = 6
f = 0
a@ubuntu:~/mywork/makefile/hello$ s
```

- Makefile执行顺序:

- 首先是main目标文件, 要生成目标文件得依赖main.o文件; 此时还没有main.o文件, Makefile会往下找, 找到了main.o目标文件, 其依赖main.c文件; main.c文件刚好有, 然后就会调用下面那条命令(`gcc -c main.c -o main.o`)去生成main.o文件; 有了main.o文件, 也就可以生成main目标文件了; Makefile紧接着会执行命令(`gcc main.o -o main`)去生成main目标文件!
- 也就是说, 是先执行`gcc -c main.c -o main.o`, 然后在执行`gcc main.o -o main`

```
main.c文件:
#include <stdio.h>
#include "add.h"
#include "sub.h"
#include "mul.h"
#include "div.h"
int main(int argc, char* argv[])
{
    int m = 2;
    int n = 3;
    int c,d,e,f;
    c = add(m, n);
    printf("c = %d\n", c);

    d = sub(m, n);
    printf("d = %d\n", d);

    e = mul(m, n);
    printf("e = %d\n", e);

    f = div(m, n);
    printf("f = %d\n", f);
    return 0;
}
```

add.h文件:

```
#ifndef __ADD_H__
#define __ADD_H__

int add(int a, int b);

#endif
```

sub.h文件:

```
#ifndef __SUB_H__
#define __SUB_H__

int sub(int a, int b);

#endif
```

mul.h文件:

```
#ifndef __MUL_H__
#define __MUL_H__

int mul(int a, int b);

#endif
```

div.h文件:

```
#ifndef __DIV_H__
#define __DIV_H__

int div(int a, int b);

#endif
```

add.c文件:

```
#include "add.h"

int add(int a, int b)
{
    return a+b;
}
```

sub.c文件:

```
#include "sub.h"

int sub(int a, int b)
{
    return a-b;
}
```

mul.c文件:

```
#include "mul.h"

int mul(int a, int b)
{
    return a*b;
}
```

div.c文件:

```
#include "div.h"

int div(int a, int b)
{
    return a/b;
}
```

Makefile文件:

main: main.o add.o sub.o mul.o div.o

gcc main.o add.o sub.o mul.o div.o -o main

main.o: main.c

gcc -c main.c -o main.o

add.o: add.c

gcc -c add.c -o add.o

sub.o: sub.c

gcc -c sub.c -o sub.o

mul.o: mul.c

gcc -c mul.c -o mul.o

div.o: div.c

gcc -c div.c -o div.o

问题：请解释Makefile文件的执行顺序？

```
a@ubuntu:~/mywork/makefile/hello002$ ls
add.c add.h div.c div.h main.c Makefile mul.c mul.h sub.c sub.h
a@ubuntu:~/mywork/makefile/hello002$ make
gcc -c main.c -o main.o
gcc -c add.c -o add.o
gcc -c sub.c -o sub.o
gcc -c mul.c -o mul.o
gcc -c div.c -o div.o
gcc main.o add.o sub.o mul.o div.o -o main
a@ubuntu:~/mywork/makefile/hello002$ ls
add.c add.h add.o div.c div.h div.o main main.c main.o Makefile mul.c mul.h mul.o sub.c sub.h sub.o
a@ubuntu:~/mywork/makefile/hello002$ ./main
c = 5
d = -1
e = 6
f = 0
a@ubuntu:~/mywork/makefile/hello002$
```

二、Makefile书写规则

- 1、注释
 - 在makefile中只有行注释，没有那种/**/这样子的多行注释
 - 行注释以 # 开头
 - 当然，如果非要写多行，makefile中有一个反斜杠（\），意思是这样写不完了，写在下一行，但是本质上还是一行。

- 2、变量

- 2.1 内置环境变量

- 与命令相关的变量

- AR----函数库打包程序。默认命令是“ar”。
 - AS----汇编语言编译程序。默认命令是“as”。
 - CC----C语言编译程序。默认命令是“cc”。
 - CXX----C++语言编译程序。默认命令是“g++”。
 - CPP----C程序的预处理器（输出是标准输出设备）。默认命令是“\$(CC) -E”。
 - RM----删除文件命令。默认命令是“rm -f”。
 -

- 关于命令参数可以使用的变量
 - ARFLAGS----函数库打包程序AR命令的参数。默认值是“rv”。
 - ASFLAGS----汇编语言编译器参数。（当明显地调用“.s”或“.S”文件时）。
 - CFLAGS----C语言编译器参数。
 - CXXFLAGS----C++语言编译器参数。
 - CPPFLAGS----C预处理器参数。（C和Fortran编译器也会用到）。
 - LDFLAGS----链接器参数。（如：“ld”）
 -

- 2.2 自动化变量

- \$* 不包含扩展名的目标文件名称
- \$+ 所有的依赖文件，以空格分开，并以出现的先后为序，可能包含重复的依赖文件
- \$< 第一个依赖文件的名称
- \$? 所有时间戳比目标文件晚的的依赖文件，并以空格分开
- \$@ 目标文件的完整名称
- \$^ 所有不重复的目标依赖文件，以空格分开
- \$% 如果目标是归档成员，则该变量表示目标的归档成员名称

- 2.3 普通变量

- 普通变量的定义很简单，也不需要啥类型，因为就像一个宏定义，是扩展上去的，所以不需要类型，直接写就行。XYZ = "hello world"
- \$(XYY)就是普通变量的引用

- 2.4 命令包变量

- 这个变量是可以多行定义一个命令块的

```
root@ftpserver-VirtualBox:/home/ftp-server/ftpMakefile# more makefile
XYY = this is a normal var
define code_block
    echo $(CC)
    echo $(CFLAGS)
    echo $@
    echo $<
    echo $^
    echo $(XYY)
endef
main:main.o common.o
    #gcc main.o common.o -o ftp
    $(code_block)
main.o:main.c ftp.h
common.o:ftp.h common.c

clean:
    rm *.o
```

- 2.5 变量的赋值

- 主要有四种赋值操作：

- 简单赋值 (:=) 编程语言中常规理解的赋值方式，只对当前语句的变量有效。
 - 递归赋值 (=) 赋值语句可能影响多个变量，所有目标变量相关的其他变量都受影响。
 - 条件赋值 (?=) 如果变量未定义，则使用符号中的值定义变量，如果该变量已经赋值，则该赋值语句无效。
 - 追加赋值 (+=) 原变量用空格隔开的方式追加一个变量。

- 2.6 变量的引用

- 变量的引用是通过\$()来进行
- 在shell语句中, 如果想要使用shell中定义的变量, 要使用\$\$var来引用shell中的变量。
【注意, 在shell语句中, 如果一个shell语句位于一个情景下, 那么一定要写一行, 一行写不下用反斜杠 (\) 】

```
root@ftpserver-VirtualBox:/home/ftp-server/ftpMakefile# more makefile
list := a b c d
main:
    for var in $(list);do echo $$var;done;
clean:
    rm *.o
    rm ftp
```

```
root@ftpserver-VirtualBox:/home/ftp-server/ftpMakefile# make
for var in a b c d;do echo $var;done;
a
b
c
d
```

- 2.7 Makefile四个常用函数

- 1) 函数名称：模式替换函数—patsubst。
 - 函数格式：\$(patsubst PATTERN,REPLACEMENT,TEXT)
 - 函数功能：搜索“TEXT”中以空格分开的单词，将符合模式“PATTERN”替换“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。
 - 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。函数的返回结果是“x.c.o bar.o”

- 2) 函数名称：取文件名函数——notdir。
 - 函数格式：\$(notdir NAMES...)
 - 函数功能：从文件名序列“NAMES...”中取出非目录部分。目录部分是指最后一个斜线（“/”）（包括斜线）之前的部分。删除所有文件名中的目录部分，只保留非目录部分。
 - 返回值：文件名序列“NAMES...”中每一个文件的非目录部分。
 - 函数说明：如果“NAMES...”中存在不包含斜线的文件名，则不改变这个文件名。以反斜线结尾的文件名，是用空串代替，因此当“NAMES...”中存在多个这样的文件名时，返回结果中分割各个文件名的空格数目将不确定！这是此函数的一个缺陷。
 - 示例：\$(notdir src/foo.c hacks) 返回值为：“foo.c hacks”。

- 3) 函数名称：获取匹配模式文件名函数—wildcard
 - 函数格式：\$(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。解释：变量使用通配符需要加关键字 wildcard，表示匹配所有 .c 文件
 - 示例：\$(wildcard *.c) 返回值为当前目录下所有.c 源文件列表

- 4) foreach 函数, 函数 “foreach” 不同于其它函数。它是一个循环函数。类似于 Linux 的 shell 中的 for 语句。
 - 函数格式: `$(foreach VAR,LIST,TEXT)`
 - 函数功能: 这个函数的工作过程是这样的: 如果需要 (存在变量或者函数的引用), 首先展开变量 “VAR” 和 “LIST” 的引用; 而表达式 “TEXT” 中的变量引用不展开。执行时把 “LIST” 中使用空格分割的单词依次取出赋值给变量 “VAR”, 然后执行 “TEXT” 表达式。重复直到 “LIST” 的最后一个单词 (为空时结束)。“TEXT” 中的变量或者函数引用在执行时才被展开, 因此如果在 “TEXT” 中存在 “VAR” 的引用, 那么 “VAR” 的值在每一次展开式将会到的不同的值。
 - 返回值: 空格分割的多次表达式 “TEXT” 的计算的结果。
 - 示例: 定义变量 “files”, 它的值为四个目录 (变量 “dirs” 代表的 a、b、c、d 四个目录) 下的文件列表: `dirs := a b c d`
`files := $(foreach dir,$(dirs),$(wildcard $(dir)/*))`
例子中, “TEXT” 的表达式为 “`$(wildcard $(dir)/*)`”。表达式第一次执行时将展开为 “`$(wildcard a/*)`”; 第二次执行时将展开为 “`$(wildcard b/*)`”; 第三次展开为 “`$(wildcard c/*)`”;; 以此类推。所以此函数所实现的功能就和一下语句等价:
`files := $(wildcard a/* b/* c/* d/*)`

• 3、语句

- 格式:

- ifeq - else - endif
- ifneq - else - endif
- ifdef - else - endif
- ifndef - else - endif

关键字	功能
ifeq	判断参数是否不相等, 相等为 true, 不相等为 false。
ifneq	判断参数是否不相等, 不相等为 true, 相等为 false。
ifdef	判断是否有值, 有值为 true, 没有值为 false。
ifndef	判断是否有值, 没有值为 true, 有值为 false。

```
root@ftpserver-VirtualBox:/home/ftp-server/ftpMakefile# more makefile
X = shuangshuang
main:
ifeq ($(X),shuangshuang)
    echo beauty
else
    echo happy
endif

ifneq ($(X),yueyang)
    echo not yueyang
else
    echo yueyang
endif

ifdef (X)
    echo "define X"
else
    echo "not define"
endif

ifndef X
    echo "not define"
else
    echo "define X"
endif
```

```
root@ftpserver-VirtualBox:/home/ftp-server/ftpMakefile# make
echo beauty
beauty
echo not yueyang
not yueyang
echo "define X"
define X
echo "define X"
define X
```

- 4、略微完善一些的Makefile

```
OBJS = main.o add.o sub.o mul.o div.o
TARGET = main
CC = gcc
CFLAGS = -Wall -O -g
main: $(OBJS)
    $(CC) $(OBJS) -o $(TARGET)
main.o: main.c
    $(CC) $(CFLAGS) -c $< -o $@
add.o: add.c
    $(CC) $(CFLAGS) -c $< -o $@
sub.o: sub.c
    $(CC) $(CFLAGS) -c $< -o $@
mul.o: mul.c
    $(CC) $(CFLAGS) -c $< -o $@
div.o: div.c
    $(CC) $(CFLAGS) -c $< -o $@
clean:
    rm $(OBJS) main
```

三、Make使用

- 直接运行make
- 选项
 - -C dir读入指定目录下的Makefile
 - -f file读入当前目录下的file文件作为Makefile
 - -i忽略所有的命令执行错误
 - -I dir指定被包含的Makefile所在目录
 - -n只打印要执行的命令，但不执行这些命令
 - -p显示make变量数据库和隐含规则
 - -s在执行命令时不显示命令
 - -w如果make在执行过程中改变目录，打印当前目录名

- 谢谢!