makefile的简单结构

1. 作为makefile其简要书写结构如下：

```
targets: prerequisites;command1
    command2
```

- targets：
    - 通常为需要生成的目标文件名
    - make 需要执行的命令名称
- prerequisites：
    - 当前目标所依赖的目标或者文件
- command：
    - 完成目标所需要执行的命令

```
hello.out all : func.o main.o
        gcc -o hello.out func.o main.o

func.o : func.c
        gcc -o func.o -c func.c

main.o : main.c
        gcc -o main.o -c main.c
clean :
        rm -f main.o
```

2. 这里要注意的是：
    1. 为了提高可读性，可以使用""符号将一行一分为二
    2. tab键这种问题就没必要说了吧
    3. "@"将命令去除回显功能

```
all : test
        echo "make all"
#@取消echo的回显
test :
        @echo "make test"
```

伪目标的引入

1. makefile中的伪目标：
    - 通过.PHONY关键字声明一个伪目标
    - 伪目标不对应任何的实际文件
    - 不管伪目标的依赖是否更新，命令总是被执行

    其实就本质而言：**伪目标就是对特殊目标.PHONY的依赖**
2. 对于伪目标可以作为函数调用来使用

    由于**一旦目标被伪目标所依赖，这个目标被定义的命令总是被执行的**

    所以可以实现如下：

```
.PHONY : rebuild clean all
rebuild : clean all
all : hello.out
clean :
        rm *.o hello.out
```

这里rebuild clean all无论文件是否为新的，make在解析的时候总是会执行的

3. 技巧:绕开.PHONY来定义伪目标（如果又想用make又想用cmake好像就可以这样玩）

```
clean : FORCE
        rm *.o hello.out
FORCE :
```

在自定义的目标FORCE的作用下，clean就会总是执行

变量和不同的复制方式

makefile中变量的命名规则：

- 变量名可以包含字符数字下划线
- 不能有":" "#" "=" " "
- 对大小写敏感
  其实makefile中的变量可以看作字符变量，作用多是字符替换，提升makefile的可扩展性:

```
CC := g++
TARGET := hello-world.out
$(TARGET) : func.o main.o
        $(CC) -o $(TARGET) func.o main.o
func.o : func.c
        $(CC) -o func.o -c func.c
main.o : main.c
        $(CC) -o main.o -c main.c
.PHONY : rebuild clean all
rebuild : clean all
all : $(TARGET)
clean :
        rm *.o $(TARGET)
```

makefile中变量的赋值方式

简单赋值(:=)

类似于C中的赋值，只在新的赋值成立后做出对应改变

```
x: =foo
y: =$(x)
x: =new
.PHONY: test
test:
    @echo "x=>$(x)"
    @echo "y=>$(y)"
```

输出：

```
x=>new
y=>foo
```

递归赋值(=)

一旦有一处改变，那么之前的赋值都要对应发生影响

```
x = foo
y = $(x)b
x = new
.PHONY: test
test:
    @echo "x=>$(x)"
    @echo "y=>$(y)"
```

输出:

```
x => new
y => newb
```

条件赋值(?=)

如果变量未定义，则使用该赋值语句定义变量
如果变量已经定义，则赋值语句无效---->多用于大型项目中

```
a = $(b)
b = $(c)
c = hello-makefile

x := foo
y := $(x)b
x ?= new#这里的赋值就被定义无效赋值

.PHONY : test

test :
        @echo "x => $(x)"
        @echo "y => $(y)"
        @echo "a => $(a)"
        @echo "b => $(b)"
        @echo "c => $(c)"
```

输出:

```
x => foo
y => foob
a => hello-makefile
b => hello-makefile
c => hello-makefile
```

追加赋值(+=)

类似于字符串的拼接

```
a = $(b)
b = $(c)
c = hello-makefile

x := foo
y := $(x)b
x += new

.PHONY : test

test :
        @echo "x => $(x)"
        @echo "y => $(y)"
        @echo "a => $(a)"
        @echo "b => $(b)"
        @echo "c => $(c)"
```

输出：

```
x => foo new
y => foob
a => hello-makefile
b => hello-makefile
c => hello-makefile-->递归赋值性质的体现
```

预定义变量的使用

makefile中的一些预定义的变量

- **自动变量**
  - $@ 当前规则中触发命令被执行的目标
  - $^ 当前规则中的所有依赖
  - $< 当前规则中的第一个依赖

```
.PHONY : all first second third

all : first second third
        @echo "\$$@ => $@"
        @echo "$$^ => $^"
        @echo "$$< => $<"
#对于makefile而言echo输出要加$作为转义
#此外$@对于shell也有特殊含义，所以要加\
#所以才会有这句：  @echo "\$$@ => $@"
firtst:
second:
third:
```

输出

```
$@ => all
$^ => first second third
$< => first
```

综合使用案例：

```
CC := g++
TARGET := hello-world.out
$(TARGET) : func.o main.o
        $(CC) -o $@ $^
func.o : func.c
        $(CC) -o $@ -c $^
main.o : main.c
        $(CC) -o $@ -c $^
.PHONY : rebuild clean all
rebuild : clean all
all : $(TARGET)
clean :
        $(RM) *.o $(TARGET)
```

- **特殊变量**
  - $(MAKE)当前make解释器的文件名
  - $(MAKECMDGOALS)命令行中指定的目标名（make的命令行参数）
  - $(MAKEFILE_LIST)
    - make所需要处理的makefile文件列表
    - 当前makefile的文件名总是位于列表的最后
    - 文件名之间以空格进行分隔

```
.PHONY : all out first second third test

all out :
        @echo "$(MAKE)"
        @echo "$(MAKECMDGOALS)"
        @echo "$(MAKEFILE_LIST)"
first :
        @echo "first"
second :
        @echo "second"
third :
        @echo "third"
#类似于函数调用
test :
        @$(MAKE) first
        @$(MAKE) second
        @$(MAKE) third
```

@$(MAKE)输出

```
> make test
make[1]: Entering directory '/mnt/f/C学习/githubXXXXXX/C-basic/make_file/03_变量'
first
make[1]: Leaving directory '/mnt/f/C学习/githubXXXXXX/C-basic/make_file/03_变量'
make[1]: Entering directory '/mnt/f/C学习/githubXXXXXX/C-basic/make_file/03_变量'
second
make[1]: Leaving directory '/mnt/f/C学习/githubXXXXXX/C-basic/make_file/03_变量'
make[1]: Entering directory '/mnt/f/C学习/githubXXXXXX/C-basic/make_file/03_变量'
third
make[1]: Leaving directory '/mnt/f/C学习/githubXXXXXX/C-basic/make_file/03_变量'
```

$(MAKECMDGOALS)输出：

```
> make all out
all out
all out
```

$(MAKEFILE_LIST)输出：

```
> make all out
 makefile
 makefile
```

- $(MAKE_VERSION)
- $(CURDIR)
- $(.VARIABLES)

```
.PHONY : test1 test2

TDelphi := Delphi Tang
D.T.Software := D.T.

test1 :
        @echo "$(MAKE_VERSION)"
        @echo "$(CURDIR)"
        @echo "$(.VARIABLES)"

test2 :
        @echo "$(RM)"
```

输出

```
> make test1
4.1
/mnt/f/C学习/github同步的学习资料/C-basic/make_file/03_变量
<D ?F WSLENV .SHELLFLAGS CWEAVE ?D @D @F CURDIR SHELL RM CO _ PREPROCESS.F LINK.m LINK.
```

变量的替换

变量值的替换

- 定义：使用**指定字符（字符串）** 替换变量中的**后缀字符（字符串）**
- 语法： `$(var:a=b)` 或 `${var:a=b}`
  也就是将后缀为a的变量变为后缀为b的变量
  - 替换表达式不能有任何空格
  - make中支持使用${}对变量进行取值

```
src1 := a.cc b.cc c.cc
obj1 := $(src1:cc=o)

test1 :
        @echo "obj1 => $(obj1)"
```

输出

```
> make test1
obj1 => a.o b.o c.o
```

变量的模式替换

- 定义：**使用%保留变量值中的指定字符**，替换其他字符-->**除了%那段，%前后分别都要替换**
- 语法： `$(var:a%b=x%y)` 或 `${var:a%b=x%y}`
  也就是将后缀为a的变量变为后缀为b的变量
  - 替换表达式不能有任何空格
  - make中支持使用${}对变量进行取值

```
src2 := a11b.c a22b.c a33b.c
obj2 := $(src2:a%b.c=x%y)

test2 :
        @echo "obj2 => $(obj2)"
```

输出

```
> make test2
obj2 => x11y x22y x33y
```

规则中的模式替换

```
targets:target-pattern:prereq-pattern
    command1
    command2
```

- 意义：通过 `target-pattern` 从 `target` 中匹配子目标；再通过 `prereq-pattern` 从子目标中生成依赖；构成完成的规则

```
CC := gcc
TARGET := hello-makefile.out
OBJS := func.o main.o const.o

$(TARGET) : $(OBJS)
        $(CC) -o $@ $^
#这里就用到了模式替换---->可以理解成把.c替换成.o
$(OBJS) : %.o : %.c
        $(CC) -o $@ -c $^
#等价于
#func.o:func.c
#       gcc -o $@ -c $^
#main.o:main.c
#       gcc -o $@ -c $^
.PHONY : rebuild clean all

rebuild : clean all
all : $(TARGET)
clean :
        $(RM) *.o $(TARGET)
```

变量值的嵌套引用

- 一个变量名中可以包含对其他变量值的引用
  - 其实就本质而言：就是用一个变量表示另一变量

```
x: =y
y:=z
a:=$(($(y)))
#a:=z
```

命令行变量

- 运行make时，在命令行定义的变量
- 命令行默认覆盖makefile中定义的变量

```
hm:=hello makefile
test:
    @echo "hm => $(hm)"
```

输入

```
> make hm=wdnmd
hm => wdnmd
```

override关键字

- 用于指示makefie中定义的变量不能被覆盖
- **变量的定义和赋值都有需要override关键字**

```
override var := test
test:
    @echo "var => $(var)"
```

输入

```
> make hm=wdnmd
var => test
```

define关键字

- 用于在makefile中定义多行变量
- 多行变量的定义从变量名开始到 `endef` 结束
- 可使用 `override` 关键字防止被覆盖
- `define` 定义的变量**等价于**使用 `=` 定义变量

```
hm := hello makefile

override var := override-test

define foo
I'm fool!
endef

override define cmd
        @echo "run cmd ls ..."
        @ls
endef
test :
        @echo "hm => $(hm)"
        @echo "var => $(var)"
        @echo "foo => $(foo)"
        ${cmd}
```

输出

```
> make test
hm => hello makefile
var => override-test
foo => I'm fool!
run cmd ls ...
const.c  func.c  main.c  makefile  makefile.3  变量的高级技巧.md
```

makefile中的三种变量

环境变量（全局变量）

- makefile 中能够直接使用环境变量的值
  - 如果定义了**同名变量**，环境变量将被覆盖
  - 运行make时指定 `"-e"` 选项，优先使用环境变量

环境变量的优点：可以在所有的makefile中使用

- 变量在不同makefile之间的传递方式
  - 直接在外部定义环境变量进行传递--->移植性低
  - 使用 `export` 定义变量进行传递（定义临时环境变量）
  - 定义make命令行变量进行传递

    makefie:

```
JAVA_HOME := java home
export var := D.T.Software
new := TDelphi

test :
        @echo "JAVA_HOME => $(JAVA_HOME)"
        @echo "make another file ..."
        @$(MAKE) -f makefile.2
        @$(MAKE) -f makefile.2 new:=$(new)
```

makefile.2:

```
test:
        @echo "JAVA_HOME => $(JAVA_HOME)"
        @echo "var => $(var)"
        @echo "new => $(new)"
```

输出：

```
> make test
JAVA_HOME => java home
make another file ...
make[1]: Entering directory '/mnt/f/C学习/github同步的学习资料/C-basic/make_file/04_变量的高
JAVA_HOME =>
var => D.T.Software
new =>
make[1]: Leaving directory '/mnt/f/C学习/github同步的学习资料/C-basic/make_file/04_变量的高
make[1]: Entering directory '/mnt/f/C学习/github同步的学习资料/C-basic/make_file/04_变量的高
JAVA_HOME =>
var => D.T.Software
new => TDelphi
make[1]: Leaving directory '/mnt/f/C学习/github同步的学习资料/C-basic/make_file/04_变量的高
```

目标变量

- 作用域只在指定目标及连带规则中
  - `target: name <assignment> value`
  - `target: override name <assignment> value`

```
var := FUFU
test : var :=test-var

#只在test中起作用
test :
        @echo "test:"
        @echo "var => $(var)"
```

输出

```
> make test
test:
var => test-var
```

模式变量（可以看作目标变量plus）

- 模式变量是目标变量的扩展
- 作用域只在符合**模式**的目标及其连带规则中
    - `pattern: name <assignment> value`
    - `pattern: override name <assignment> value`

```
var := D.T.Software
new := TDelphi

test : var := test-var
%e : override new := test-new
#只要是以e结尾的都要执行override new := test-new
#一般模式变量都是要和override配合使用
test : another
        @echo "test :"
        @echo "var => $(var)"
        @echo "new => $(new)"

another :
        @echo "another :"
        @echo "var => $(var)"
        @echo "new => $(new)"

rule :
        @echo "rule :"
        @echo "var => $(var)"
        @echo "new => $(new)"
```

输出:

```
> make
another :
var => test-var
new => TDelphi
test :
var => test-var
new => TDelphi
```

条件判断语句

- 条件判断关键字

关键字	功能
ifeq	判断参数是否相等
ifneq	判断参数是否不相等
ifdef	判断变量是否有值（是否被赋值）
ifndef	判断变量是否没有有值（是否被赋值）

```
ifxxx (arg1,arg2)
#xxxxx
else
#xxxxx
endif
```

```
.PHONY : test
var1 := A
var2 := $(var1)
var3 :=
test:
    ifeq ($(var1),$(var2))
                @echo "var1 == var2"
    else
                @echo "var1 != var2"
    endif
    ifneq ($(var2),)
                @echo "var2 is NOT empty"
    else
                @echo "var2 is empty"
    endif
    ifdef var2
                @echo "var2 is NOT empty"
    else
                @echo "var2 is empty"
    endif
    ifndef var3
                @echo "var3 is empty"
    else
                @echo "var3 is NOT empty"
    endif
```

输出

```
> make test
var1 == var2
var2 is NOT empty
var2 is NOT empty
var3 is empty
```

其中格式注意：`(arg1,arg2)` 里面是不能有空格的
此外：条件判断语句只能用于控制make实际执行的语句，并不能控制命令的执行过程

工程经验：

- 条件判断语句之前可以有空格，但是不能有 `\t` 这类的tab字符
- 条件判断语句中不要使用自动变量(`$@ $< $^`)
- 一条完整的条件语句必须位于同一个makefile中
- 条件判断类似于条件编译：**预处理阶段有效，执行阶段无效**
- make 在加载makefile时
  - 首先计算表达式的值（赋值方式不同，计算方式不同）
  - 根据**判断语句的表达式**决定执行的内容

```
.PHONY : test

var1 :=
var2 := $(var1)
var3 =
var4 = $(var3)
var3 = 3
#先计算后判断
test:
    ifdef var1
                @echo "var1 is defined"
    else
                @echo "var1 is NOT defined"
    endif

    ifdef var2
                @echo "var2 is defined"
    else
                @echo "var2 is NOT defined"
    endif

    ifdef var3
                @echo "var3 is defined"
    else
                @echo "var3 is NOT defined"
    endif

    ifdef var4
                @echo "var4 is defined"
    else
                @echo "var4 is NOT defined"
    endif
```

输出

```
> make test
var1 is NOT defined
var2 is NOT defined
var3 is defined
var4 is defined
```

函数定义及调用

makefile支持函数的概念

- make解释器提供了一些已经定义好的函数供makefile调用
- makefile也支持自定义函数的实现，并调用执行
- 通过define关键字实现自定义函数

自定义函数

自定义函数的函数定义及其调用如下：

```
.PHONY : test
#函数定义：
define func1
        @echo "My name is $(0)"
endef

define func2
  @echo "My name is $(0)"
#函数的第0号参数-->也就是函数名
#后面的就是函数实参
  @echo "Param 1 => $(1)"
        @echo "Param 2 => $(2)"
endef
#函数调用与变量的区别：
func := $(call func1)
var := $(func1)

test :
        @echo "func => $(func)"
        @echo "var => $(var)"
#函数调用：
        $(call func1)  #@echo My name is func1
        $(call func2, WDNMD, fufu)
#call 其实是makefile已经定义好的一个函数
```

输出

```
> make
func =>         @echo My name is func1
var =>  @echo My name is
My name is func1
My name is func2
#函数的第0号参数-->也就是函数名
#后面的就是函数实参
Param 1 =>  WDNMD
Param 2 =>  fufu
```

- 自定义函数的深入理解
  - 本质上就是一个多行变量，无法直接调用
  - 是一种过程调用，所以没有任何返回值
  - 用于定义命令的集合吗，应用于规则中

本质上：

- makefile并不支持真正意义上的自定义函数，其本质就是多行变量

- 预定义的 `call` 函数在调用时将参数传给多行变量
- 自定义函数就是 `call` 函数的实参，在call中被执行

预定义函数

- 主要用于处理文件名，变量，命令（多为字符串处理函数）
- 可以在需要的地方调用预定义函数处理指定的参数
- 函数**在调用的地方被替换为处理结果**

```
.PHONY : test
define func1
        @echo "My name is $(0)"
endef

define func2
        @echo "My name is $(0)"
endef
var1 := $(call func1)
var2 := $(call func2)
var3 := $(abspath ./)
var4 := $(abspath test.cpp)
test :
        @echo "var1 => $(var1)"
        @echo "var2 => $(var2)"
        @echo "var3 => $(var3)"
        @echo "var4 => $(var4)"
```

输出

```
var1 =>         @echo My name is func1
var2 =>         @echo My name is func2
var3 => /mnt/f/C学习/github同步的学习资料/C-basic/make_file/05_ 条件判断语句和函数
var4 => /mnt/f/C学习/github同步的学习资料/C-basic/make_file/05_ 条件判断语句和函数/test.cpp
```

变量与函数的综合示例

常见的预定义函数：

`$(widcard <pattern>)`

作用：获取当前工作目录中满足 `pattern` 的文件或者目录表

`$(addprefix <prefix>,<names>)`

作用：给名字列表 `names` 中的每一个名字增加前缀 `prefix`

技巧

- 自动获取当前目录下的源文件列表（函数调用）

  ```
  SRCS := $(widcard *.c)
  ```

- 根据源文件列表生成目标文件列表（变量的值替换）

  ```
  OBJS := $(SRCS:.c=.o)
  ```

- 对每一个目标文件列表加上路径前缀（函数调用）

  ```
  OBJS := $(addprefix path/,$(OBJS))
  ```

综合示例

```
CC := gcc
MKDIR := mkdir
RM := rm -fr

DIR_OBJS := objs
DIR_TARGET := target
DIRS := $(DIR_OBJS) $(DIR_TARGET)

TARGET := $(DIR_TARGET)/hello-makefile.out
# main.c const.c func.c
SRCS := $(wildcard *.c)
# main.o const.o func.o
OBJS := $(SRCS:.c=.o)
# objs/main.o objs/const.o objs/func.o
OBJS := $(addprefix $(DIR_OBJS)/, $(OBJS))
#在规则中创建目录
.PHONY : rebuild clean all

$(TARGET) : $(DIRS) $(OBJS)
        $(CC) -o $@ $(OBJS)
        @echo "Target File ==> $@"

$(DIRS) :
        $(MKDIR) $@
#创建文件夹

$(DIR_OBJS)/%.o : %.c
    ifeq ($(DEBUG),true)
                $(CC) -o $@ -g -c $^
    else
                $(CC) -o $@ -c $^
    endif
#针对目录的模式替换
#加入路径前缀，

rebuild : clean all
all : $(TARGET)
clean :
        $(RM) $(DIRS)
```

总结：

- 目录可以成为目标的依赖，在规则中创建目录
- 规则中的模式匹配可以直接针对目录中的文件
- 可以使用命令行变量编译特殊的目标版本

自动生成依赖关系

提前需要的知识

问题：规则中以源文件为依赖（.h文件），命令可能无法执行

- 预处理器将头文件的代码直接插入到源文件
- 编译器只通过预处理后的源文件产生目标文件
  例如：

```
OBJS := func.o main.o

hello.out : $(OBJS)
        @gcc -o $@ $^
        @echo "Target File ==> $@"

$(OBJS) : %.o : %.c func.h
        @gcc -o $@ -c $<
#自动搜索
#$(OBJS) : %.o : %.c
#       @gcc -o $@ -c $^
#头文件没有写入的话，并不会更新对应内容
```

func.h

```
#ifndef FUNC_H
#define FUNC_H

#define HELLO "Hello D.T."
void foo();
#endif
```

func.c

```
#include "stdio.h"
#include "func.h"

void foo()
{
    printf("void foo() : %s\n", HELLO);
}
```

main.c

```
#include <stdio.h>
#include "func.h"

int main()
{
    foo();
    return 0;
}
```

但是当编译后再次修改func.h时，头文件将会再次作为依赖写入目标对应的规则中
如果有成百上千个头文件，头文件的改动将会导致任何文件都被重新编译，难以维护

解决方案：
通过命令自动生成头文件依赖
将生成的依赖自动包含生成makefile中
当头文件改动后，自动确认需要重新编译的文件

技术路线：

- Linux中的sed命令
  - sed 是一种流编辑器，用于流文本的增删改查
  - sed可用于流文本的中的字符串替换
  - sed的字符串替换方式为： `sed 's:src:des:g'`

```
echo "test=>abc+abc=abc"|sed 's:abc:xyz:g'
test=>xyz+xyz=xyz
```

关于正则表达式，可参考：
https://r2coding.com/#/?id=正则表达式

sed的正则表达式支持：

- 在sed中可以使用正则表达式匹配替换目标
- 并且可以使用匹配的目标生成替换结果

```
sed 's,\(.*\)\.o[ :]*,objs/\.o:,g'
```

```
echo "/a/s/d/main.o :main.c func.c" |sed 's,\(.*\)\.o[ :]*,objs/\.o:,g'
```

输出：

```
objs//a/s/d/main.o:main.c func.c
```

以上正则表达式表示一种规则，对其进行替换

- 编译器依赖生成选项gcc -MM(gcc -M)
  - 生成依赖关系
    - 获取目标完成的依赖关系

      ```
      gcc -M test.c
      ```
    - 获取目标表部分的依赖关系（仅仅是自己编写的文件）

      ```
      gcc -MM test.c
      ```

综合：

```
gcc -MM -E main.c |sed 's,\(.*\)\.o[ :]*,objs/\.o:,g'
```

输出：

```
> gcc -MM -E main.c |sed 's,\(.*\)\.o[ :]*,objs/\.o:,g'
objs/main.o:main.c func.h


> gcc -M -E main.c |sed 's,\(.*\)\.o[ :]*,objs/\.o:,g'
objs/main.o:main.c /usr/include/stdc-predef.h /usr/include/stdio.h \
 /usr/include/x86_64-linux-gnu/bits/libc-header-start.h \
 /usr/include/features.h /usr/include/x86_64-linux-gnu/sys/cdefs.h \
 /usr/include/x86_64-linux-gnu/bits/wordsize.h \
 /usr/include/x86_64-linux-gnu/bits/long-double.h \
 /usr/include/x86_64-linux-gnu/gnu/stubs.h \
 /usr/include/x86_64-linux-gnu/gnu/stubs-64.h \
 /usr/lib/gcc/x86_64-linux-gnu/7/include/stddef.h \
 /usr/include/x86_64-linux-gnu/bits/types.h \
 /usr/include/x86_64-linux-gnu/bits/typesizes.h \
 /usr/include/x86_64-linux-gnu/bits/types/__FILE.h \
 /usr/include/x86_64-linux-gnu/bits/types/FILE.h \
 /usr/include/x86_64-linux-gnu/bits/libio.h \
 /usr/include/x86_64-linux-gnu/bits/_G_config.h \
 /usr/include/x86_64-linux-gnu/bits/types/__mbstate_t.h \
 /usr/lib/gcc/x86_64-linux-gnu/7/include/stdarg.h \
 /usr/include/x86_64-linux-gnu/bits/stdio_lim.h \
 /usr/include/x86_64-linux-gnu/bits/sys_errlist.h func.h
```

技巧：拆分目标依赖(有些依赖不必一次写完)

```
.PHONY : test a b c
test : a b
test : b c
test :
        @echo "$^"
```

等价于

```
.PHONY : test a b c
test : a b c
        @echo "$^"
```

include关键字

类似于C语言中 `include`
但是make对 `include` 关键字的处理方式有些许不同：如果搜索失败：
1.产生警告->2.对文件名作为目标查找并执行对应的规则->3.如果文件名都对应不上，最终产生错误

```
.PHONY : all
include test.txt
all :
        @echo "this is all"
test.txt :
        @echo "test.txt"
        @touch test.txt
```

输出

```
makefile:3: test.txt: No such file or directory #1.产生警告
test.txt #2.对文件名作为目标查找并执行对应的规则
this is all #3.如果文件名都对应不上，最终产生错误
```

makefile执行机制

- 规则中的每一个命令默认是在一个新的进程中执行（shell）
- 可以通过接续符（；）将多个命令组合成一个命令（改变上一个逻辑，让多个命令在同一进程中执行）
- 组合的命令依次在同一个进程中被执行
- set-e 指定发生错误后立即退出执行

```
.PHONY : all
all :
        set -e; \
        mkdir test; \
        cd test; \
        mkdir subtest
```

这样就可以在test里面创建一个子文件夹subtest

; 接续 \ 连接符

自动生成依赖关系的初步实现

基于以上知识点，我们就可以实现自动生成依赖关系的makefile

- 通过 `gcc -MM` 和 `sed` 得到 `.dep` 依赖文件（目标的部分依赖）
  - 技术点：规则中命令的连续执行
- 通过 `include` 指令包含所有的 `.dep` 依赖文件
  - 技术点：当 `.dep` 依赖文件不存在时，使用规则自动生成

```
.PHONY : all clean

MKDIR := mkdir
RM := rm -fr
CC := gcc

SRCS := $(wildcard *.c)
DEPS := $(SRCS:.c=.dep) #DEPS对应的规则

-include $(DEPS)

all :
        @echo "all"
#生成依赖文件
%.dep : %.c
        @echo "Creating $@ ..."
        @set -e; \
        $(CC) -MM -E $^ | sed 's,\(.*\)\.o[ :]*,objs/\.o : ,g' > $@

clean :
        $(RM) $(DEPS)
```

其中 `.dep` 中的内容为对应的依赖关系

```
objs/func.o : func.c func.h
```

但是还需要解决的问题是：如何组织依赖文件相关的规则与源码编译相关的规则，进而形成成功的完整的makefile程序？

面对海量的 .dep 文件又如何解决

解决思路：

1.通过规则和命令创建deps文件

2.将所有的.dep文件创建到deps文件夹下

3.dep文件记录目标文件的依赖关系

```
.PHONY : all clean

MKDIR := mkdir
RM := rm -fr
CC := gcc

DIR_DEPS := deps

SRCS := $(wildcard *.c)
DEPS := $(SRCS:.c=.dep)
DEPS := $(addprefix $(DIR_DEPS)/, $(DEPS))

include $(DEPS)

all :
        @echo "all"

$(DIR_DEPS) :
        $(MKDIR) $@

$(DIR_DEPS)/%.dep : $(DIR_DEPS) %.c
        @echo "Creating $@ ..."
        @set -e; \
        $(CC) -MM -E $(filter %.c, $^) | sed 's,\(.*\)\.o[ :]*,objs/\1.o : ,g' > $@

clean :
        $(RM) $(DIR_DEPS)
```

但是上面的代码有个问题，就是func.dep重复执行了一下

输出：

```
> make
makefile:13: deps/main.dep: No such file or directory
makefile:13: deps/func.dep: No such file or directory
mkdir deps
Creating deps/func.dep ...
Creating deps/main.dep ...
Creating deps/func.dep ...
make: Nothing to be done for 'objs/main.o'.
```

原因：make发现deps文件夹更新了

优化：

```
.PHONY : all clean

MKDIR := mkdir
RM := rm -fr
CC := gcc

DIR_DEPS := deps

SRCS := $(wildcard *.c)
DEPS := $(SRCS:.c=.dep)
DEPS := $(addprefix $(DIR_DEPS)/, $(DEPS))


all :
        @echo "all"

ifeq ("$(MAKECMDGOALS)", "all")
-include $(DEPS)
endif

ifeq ("$(MAKECMDGOALS)", "")
-include $(DEPS)
endif

$(DIR_DEPS) :
        $(MKDIR) $@

ifeq ("$(wildcard $(DIR_DEPS))", "")
$(DIR_DEPS)/%.dep : $(DIR_DEPS) %.c
else
$(DIR_DEPS)/%.dep : %.c
endif
        @echo "Creating $@ ..."
        @set -e; \
        $(CC) -MM -E $(filter %.c, $^) | sed 's,\(.*\)\.o[ :]*,objs/\1.o : ,g' > $@

clean :
        $(RM) $(DIR_DEPS)
```

include关键字的隐形操作

- 使用 - 号不但关闭了include发出的警告，同时关闭了错误，**当错误发生时make将忽略这些错误!**

```
.PHONY : all
-include test.txt
include test.txt
all :
        @echo "this is all"
#test.txt :
        #@echo "creating $@ ..."
        #@echo "other : ; @echo "this is other" " > test.txt
```

最好别用 - ，发布版本再用也不迟

- 创建的test.txt会被再次载入

```
.PHONY : all
-include test.txt
include test.txt
all :
        @echo "this is all"
test.txt :
        @echo "creating $@ ..."
        @echo "other : ; @echo "this is other" " > test.txt
```

- 如果包含的文件存在，如果改该文件作为依赖比目标更新，则会执行对应的目标

```
.PHONY : all

-include test.txt

all :
        @echo "this is all"

test.txt : b.txt
        @echo "creating $@ ..."
```

如果b.txt存在，但是make会依旧执行test.txt

- 如果需要包含的文件有修改，那么对原有文件进行修改

```
.PHONY : all

-include test.txt

all :
        @echo "$@ : $^"

test.txt : b.txt
        @echo "creating $@ ..."
        @echo "all : c.txt" > test.txt
```

对于include的总结:

- 当目标文件不存在时
  - 以文件名查找规则
- 当目标文件不存在，且查找到的规则中创建了目标文件
  - 将创建成功的目标文件包含进当前makefile
- 目标文件存在
  - 将目标文件包含进当前makefile
  - 以目标文件名查找是否有相应的规则
    - yes 比较规则的依赖关系，决定是否执行规则的命令
    - no 无操作

自动生成依赖关系的最后实现

注意：当 `.dep` 文件生成后，如果动态的改变头文件间的依赖关系，那么make可能无法检测到这个改变，进而做出错误的编译决策

解决方案:

- 将依赖文件名作为目标加入自动生成的依赖关系中（利用了include的隐性操作）
- 通过include加载依赖文件时判断是否执行规则
- 在规则执行时重新生成依赖关系文件

```
.PHONY : all clean rebuild

MKDIR := mkdir
RM := rm -fr
CC := gcc

DIR_DEPS := deps
DIR_EXES := exes
DIR_OBJS := objs

DIRS := $(DIR_DEPS) $(DIR_EXES) $(DIR_OBJS)

EXE := app.out
EXE := $(addprefix $(DIR_EXES)/, $(EXE))

SRCS := $(wildcard *.c)
OBJS := $(SRCS:.c=.o)
OBJS := $(addprefix $(DIR_OBJS)/, $(OBJS))
DEPS := $(SRCS:.c=.dep)
DEPS := $(addprefix $(DIR_DEPS)/, $(DEPS))


all : $(DIR_OBJS) $(DIR_EXES) $(EXE)

ifeq ("$(MAKECMDGOALS)", "all")
-include $(DEPS)
endif

ifeq ("$(MAKECMDGOALS)", "")
-include $(DEPS)
endif

$(EXE) : $(OBJS)
        $(CC) -o $@ $^
        @echo "Success! Target => $@"

$(DIR_OBJS)/%.o : %.c
        $(CC) -o $@ -c $(filter %.c, $^)

$(DIRS) :
        $(MKDIR) $@

ifeq ("$(wildcard $(DIR_DEPS))", "")
$(DIR_DEPS)/%.dep : $(DIR_DEPS) %.c
else
$(DIR_DEPS)/%.dep : %.c
endif
        @echo "Creating $@ ..."
        @set -e; \
        $(CC) -MM -E $(filter %.c, $^) | sed 's,\(.*\)\.o[ :]*,objs/\1.o $@ : ,g' > $@

clean :
        $(RM) $(DIRS)
```

```
rebuild :
        @$(MAKE) clean
        @$(MAKE) all
```

总结：

- makefile中可以将目标的依赖拆分写到不同的地方
- include关键字能够触发相应规则的执行
- 如果规则的执行导致依赖更新，可能导致再次执行相应的规则
- 依赖文件也需要依赖于源文件得到正确的编译决策
- 自动生成的文件间的依赖关系能够提高makefile的移植性

隐式规则

makefile有脚本属性，所以由上到下新的命令会把上面的旧的命令覆盖

- makefi在出现同名目标时
  - 依赖：
    - 所有依赖将合并在一起，成为目标的最终依赖
  - 命令：
    - 当多处出现同一目标命令时，make发出警告
    - 所有之间的命令被最后的命令取代

所有要注意：**include关键字需要确保被包含的文件的同名目标只有依赖，没有命令；否则同名目标的命令将被覆盖**

什么是隐式规则：make提供的一些常用的例行规则实现
当相应目标的规则未提供时，make会使用这些规则(可以理解成make自带的库函数)

```
SRCS := $(wildcard *.c)
OBJS := $(SRCS:.c=.o)

app.out : $(OBJS)
        $(CC) -o $@ $^
        $(RM) $^
        @echo "Target ==> $@"
```

```
> make
gcc    -c -o main.o main.c
gcc    -c -o func.o func.c
gcc -o app.out main.o func.o
rm -f main.o func.o
Target ==> app.out
```

所以我们可以看出make默认填写了CC的值

而

```
app.out : $(OBJS)
        $(CC) -o $@ $^
        $(RM) $^
        @echo "Target ==> $@"
```

被make理解成

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

隐式规则的执行逻辑：当发现目标依赖不存时

- 尝试通过依赖名逐一查找隐式规则
- 并且通过依赖名推导可能需要的源文件
  但是！！！要尽量避免！！！

隐式规则禁用

- 局部禁用
  - 在makefile中自定义规则
  - 在makefile中定义模式
- 全局禁用
  - make -r

后缀规则了解--一种旧的模式规则

```
app.out : main func.o
         $(CC) -lstdc++ -o $@ $^
.c.o :
        @echo "my suffix rule"
        $(CC) -o $@ -c $^
.c :
        @echo "my suffix rule"
        $(CC) -o $@ -c $^
```

路径搜索

VPATH

考虑到实际的项目的源文件不会和头文件放在一起。**所以项目中的makefile必须正确定位源文件和依赖文件。**

特殊的预定义变量**VPATH**

- 用于指示make如何查找文件
- 不同文件夹可作为VPATH的值同时出现
- 文件夹名字之间需要使用**分隔符**进行区分
  例如：
  ```
  VPATH := $(INC) $(SRC)
  VPATH := $(INC);$(SRC)
  VPATH := $(INC):$(SRC)
  ```
  皆可
  注意：
- VPATH只能决定make的搜索路径，无法决定命名的搜索路径
- 对于特定的编译命令，需要独立指定编译搜索路径
  ```
  gcc -I include-path \
  ```

```
OBJS := func.o main.o
INC := inc
SRC := src
VPATH := $(INC) $(SRC)
CFLAGS := -I $(INC)

hello.out : $(OBJS)
        @gcc -o $@ $^
        @echo "Target File ==> $@"

$(OBJS) : %.o : %.c func.h
        @gcc $(CFLAGS) -o $@ -c $<
```

vpath

但是VPATH路径中的文件夹中如果有多个同名文件，VPATH会选择第一次搜索到的文件
用**vpath**关键字为不同类型文件指定不同的搜索路径

```
vpath %.h inc
vpath %.c src
```

所以最终的makefile可以为：

```
OBJS := func.o main.o
INC := inc
SRC := src
CFLAGS := -I $(INC)

vpath %.h $(INC)
vpath %.c $(SRC)

hello.out : $(OBJS)
        @gcc -o $@ $^
        @echo "Target File ==> $@"

# vpath %.h

$(OBJS) : %.o : %.c func.h
        @gcc $(CFLAGS) -o $@ -c $<
```

此外**vpath**还可以取消搜索
取消.h文件搜索： `vpath %.h`
设置 `$(INC)` 下.h文件搜索： `vpath %.h $(INC)`

VPATH与vpath

如果文件路径上文件是这样：

```
PRJ
|
|--->inc
        |--->func.h
|--->src1
        |--->func.c
|--->src2
        |--->func.c
|--->main.c
|
```

有如下的makefile:

```
VPATH := src1
CFLAGS := -I inc

vpath %.c src2
vpath %.h inc

app.out : func.o main.o
        @gcc -o $@ $^
        @echo "Target File ==> $@"

%.o : %.c func.h
        @gcc $(CFLAGS) -o $@ -c $<
```

编译出的app.out输出打印的为src2的func.c

```
> make
Target File ==> app.out
> ./app.out
void foo() : This file is from src2 ...
```

vpath优先于VPATH，如果找不到才会选VPATH
如果还是找不到就要启动隐式规则

vpath的执行顺序

```
CFLAGS := -I inc

vpath %.c src1
vpath %.c src2

vpath %.h inc

app.out : func.o main.o
        @gcc -o $@ $^
        @echo "Target File ==> $@"

%.o : %.c func.h
        @gcc $(CFLAGS) -o $@ -c $<
```

最终选的是src1
是**自上而下的搜索，如果找不到就找下一个vpath,还是找不到就会跑到隐式规则（避免使用VPATH触发隐式规则）**

make指定目标文件的最终位置

默认的生成app.out的位置的逻辑:

- 当app.out完全不存在
  - make会在当前目录下生成app.out
- 当app.out存在于根文件以外的文件目录下
  - 如果所有的依赖不变，make不会重新创建app.out
  - 如果依赖更新，make会在当前文件夹下创建app.out

解决方案：GPATH这个特殊变量指定目标文件夹

```
GPATH := src
VPATH := src
CFLAGS := -I inc

app.out : func.o main.o
        @gcc -o $@ $^
        @echo "Target File ==> $@"

%.o : %.c inc/func.h
        @gcc $(CFLAGS) -o $@ -c $<
```

这个时候生成app.out的逻辑:

- 当app.out完全不存在
  - make会在当前目录下生成app.out
- 当app.out存在于根文件以外的文件目录下
  - make会在app.out所在位置的文件夹下的创建及其更新app.out

工程中注意
为编译得到的结果创建独立的文件夹
不要在源码的文件夹中生成目标文件
尽量使用vpath为不同文件指定搜索路径
避免使用VPATH与GVATH（误触隐式规则，这俩最好成对出现）

路径搜索实战

需求：

- 支持调试版本的编译选项
- 易于拓展，可以复用于同类型项目
- build目录存放编译结果
- 不希望源码中文件夹在编译时被改动

用到的知识点

- `$(wildcard $(DIR)/*type)`
  - 获取 `$(DIR)` 下的type类型文件
- `$(notdir _names)`
  - 去除_names中每一个文件的路径前缀
- `$(patsubst _pattern, replacement,_text)`
  - 将 _text中符合_pattern部分替换为replacement

关键技巧

- 自动获取源文件列表（函数调用实现）
  - `SRCS := $(wildcard src/*c)`
- 根据源文件列表生成目标文件列表（变量的值替换）
  - `OBJS := $(SRC:.c=.o)`
- 替换每一个目标文件的路径前缀（函数调用实现）
  - `OBJS := $(patsubst src/%, build/%,$(OBJS))`

```
.PHONY : all clean

DIR_BUILD := build
DIR_SRC := src
DIR_INC := inc

TYPE_INC := .h
TYPE_SRC := .c
TYPE_OBJ := .o

CC := gcc
LFLAGS :=
CFLAGS := -I $(DIR_INC)
ifeq ($(DEBUG),true)
CFLAGS += -g
endif

MKDIR := mkdir
RM := rm -fr

APP := $(DIR_BUILD)/app.out
HDRS := $(wildcard $(DIR_INC)/*$(TYPE_INC))
HDRS := $(notdir $(HDRS))
OBJS := $(wildcard $(DIR_SRC)/*$(TYPE_SRC))
OBJS := $(OBJS:$(TYPE_SRC)=$(TYPE_OBJ))
OBJS := $(patsubst $(DIR_SRC)/%, $(DIR_BUILD)/%, $(OBJS))

vpath %$(TYPE_INC) $(DIR_INC)
vpath %$(TYPE_SRC) $(DIR_SRC)

all : $(DIR_BUILD) $(APP)
        @echo "Target File ==> $(APP)"

$(DIR_BUILD) :
        $(MKDIR) $@

$(APP) : $(OBJS)
        $(CC) $(LFLAGS) -o $@ $^

$(DIR_BUILD)/%$(TYPE_OBJ) : %$(TYPE_SRC) $(HDRS)
        $(CC) $(CFLAGS) -o $@ -c $<

clean :
        $(RM) $(DIR_BUILD)
```

小结

- 小规模项目没必要使用自动生成依赖关系，可以直接让源文件依赖于头文件便于维护
- 变量的灵活运用可以提升扩展性
- 模式规则的使用可以提高复用性

针对的项目目录

```
.
├── 11_打造专业的编译环境.md
├── build
│   ├── app.out
│   ├── common
│   │   ├── common.dep
│   │   ├── common.o
│   │   ├── func.dep
│   │   └── func.o
│   ├── common.a
│   ├── main
│   │   ├── main.dep
│   │   └── main.o
│   ├── main.a
│   ├── module
│   │   ├── module.dep
│   │   └── module.o
│   └── module.a
├── cmd-cfg.mk
├── common
│   ├── inc
│   │   ├── common.h
│   │   └── func.h
│   ├── makefile
│   └── src
│   ├── common.c
│   └── func.c
├── main
│   ├── inc
│   │   └── define.h
│   ├── makefile
│   └── src
│   └── main.c
├── module
│   ├── inc
│   │   └── module.h
│   ├── makefile
│   └── src
```

```
│   └── module.c
├── makefile
├── mod-cfg.mk
├── mod-rule.mk
├── pro-cfg.mk
└── pro-rule.mk
```

要关注的点

- 自动生成依赖关系（gcc -MM）
- 自动搜索需要的文件（vpath）
- 将目标文件打包为静态库文件（ar crs）

设计的makefile结构

```
├── makefile
```

```
include cmd-cfg.mk #定义命令相关的变量
include pro-cfg.mk  #定义项目的变量及其编译路径
include pro-rule.mk #定义其他变量和规则
```

```
├── mod-cfg.mk
├── mod-rule.mk
```

```
├── pro-cfg.mk
├── mod-cfg.mk
├── mod-rule.mk
└── pro-rule.mk
├── mod-cfg.mk
├── mod-rule.mk
```

小的注意事项

- 在makefile中shell的变量记得加 `$$x`
- `-Xlinker` 可以自动确定依赖关系
    - `$(CC) -o $(APP) -Xlinker "-(" $^ -Xlinker "-)" $(LFLAGS)`

独立模块编译

好处：
方便单元测试
可以单独编译项目的某一个部分

获取make命令中指定编译的模块名

- 预定义变量： `$(MAKECMDGOALS)`

解决脚本复用问题

```
define x1
    @echo "xxxxxxxxx"
endef

$(call x1 ,p1)
```

对于pro_rule.mk：

```
.PHONY : all compile link clean rebuild $(MODULES)

DIR_PROJECT := $(realpath .)
DIR_BUILD_SUB := $(addprefix $(DIR_BUILD)/, $(MODULES))
MODULE_LIB := $(addsuffix .a, $(MODULES))
MODULE_LIB := $(addprefix $(DIR_BUILD)/, $(MODULE_LIB))


APP := $(addprefix $(DIR_BUILD)/, $(APP))

define makemodule
        cd $(1) && \
        $(MAKE) all \
                        DEBUG:=$(DEBUG) \
                        DIR_BUILD:=$(addprefix $(DIR_PROJECT)/, $(DIR_BUILD)) \
                        DIR_COMMON_INC:=$(addprefix $(DIR_PROJECT)/, $(DIR_COMMON_INC))
                        CMD_CFG:=$(addprefix $(DIR_PROJECT)/, $(CMD_CFG)) \
                        MOD_CFG:=$(addprefix $(DIR_PROJECT)/, $(MOD_CFG)) \
                        MOD_RULE:=$(addprefix $(DIR_PROJECT)/, $(MOD_RULE)) && \
        cd .. ;
endef

all : compile $(APP)
        @echo "Success! Target ==> $(APP)"

compile : $(DIR_BUILD) $(DIR_BUILD_SUB)
        @echo "Begin to compile ..."
        @set -e; \
        for dir in $(MODULES); \
        do \
                $(call makemodule, $$dir) \
        done
        @echo "Compile Success!"

link $(APP) : $(MODULE_LIB)
        @echo "Begin to link ..."
        $(CC) -o $(APP) -Xlinker "-(" $^ -Xlinker "-)" $(LFLAGS)
        @echo "Link Success!"

$(DIR_BUILD) $(DIR_BUILD_SUB) :
        $(MKDIR) $@

clean :
        @echo "Begin to clean ..."
        $(RM) $(DIR_BUILD)
        @echo "Clean Success!"

rebuild : clean all

$(MODULES) : $(DIR_BUILD) $(DIR_BUILD)/$(MAKECMDGOALS)
        @echo "Begin to compile $@"
        @set -e; \
        $(call makemodule, $@)
```

库有问题(并不支持我现在用的WLS2平台)，学学makefile就是了。。。。

第三方函数库

libs
├── inc
│ ├── dlib.h
│ └── slib.h
└── lib
├── dlib.so
└── slib.a

第三方库在编译阶段的支持

- 定义变量DIR_LIBS_INC用于指示头文件的存储设置
  - `DIR_LIBS_INC := $(DIR_PROJECT)/libs/inc`
- 使用DIR_LIBS_INC提示make头文件的存储位置
  - `vpath %$(TYPE_INC) $(DIR_LIBS_INC)`
- 使用DIR_LIB_INC提示编译器头文件的存储位置
  - `CFLAGS += -I$(DIR_LIBS_INC)`