

make

make and makefile

kevinluo

Contents

1	make	1
1.1	gnu make 安装	1
1.2	help	2
1.2.1	GNU 官方	2
1.2.2	gnu 其它	2
1.3	make misc	2
2	cmake	2
2.1	cmake install	2
2.2	cmake help	2
3	FAQ	2
3.1	Makefile 变量 \$@, \$^, \$< 代表的意义	2
3.2	怎么查到函数是哪个库的?	3
3.3	只知道函数的大概形式, 怎么找到头文件。用 man	3
4	引用文章全文	3
4.1	gcc 与 makefile	3
5	实践中的一些经验	7
5.1	eval 和 define 中变量展开的坑	7
5.2	输出文件的方法	8
5.3	一些工具	8
5.4	调试输出变量信息方式	9

目录

1 make

1.1 gnu make 安装

- make 官方下载地址

GNU ftp server: <http://ftp.gnu.org/gnu/make/> (via HTTP) and <ftp://ftp.gnu.org/gnu/make/> (via FTP)

- make4.2 (GNU make) 的安装步骤

1. 解压

```
tar -zxvf make4.2.tar.gz
```

2. 安装

window: 要用到 gcc of MinGW, 或者 visual studio.

```
E:\E-ProgramFiles\portable\codeblocks-mingw\MinGW\mingwvars.bat
```

```
cd make4.2
```

```
build_w32.bat gcc
```

在 \make-4.2\GccRel 下生成 gnumake.exe

Linux

```
cd make4.2
```

```
./configure
```

```
make && make install
```

3. 打开新的窗口，验证

```
make -v 或 gnumake -v
```

1.2 help

1.2.1 GNU 官方

[gnu make manual](#)

[gnu make Wildcard-Function](#)

[gnu make index](#)

1.2.2 gnu 其它

这篇文章写得短小全面。外链-> [gcc 与 makefile](#)

1.3 make misc

2 cmake

2.1 cmake install

[cmake download](#)

2.2 cmake help

[cmake documments](#)

[cmake tutorial](#)

[cmake help v3.15](#)

3 FAQ

3.1 Makefile 变量 \$@, \$^, \$< 代表的意义

[makefile 中 \\$@ \\$^ %< 使用](#)

\$@ 目标文件，\$^ 所有的依赖文件，\$< 第一个依赖文件。

这是再一次简化后的 Makefile

```
main: main.o mytool1.o mytool2.o
```

```
gcc -o $@ $^
```

```
.c.o:
```

```
gcc -c $<
```

3.2 怎么查到函数是哪个库的？

有时候我们使用了某个函数，但是我们不知道库的名字，这个时候怎么办呢？

比如我要找 `sin` 这个函数所在的库。就只好用命令

```
nm -o /lib/\*.so|grep sin>~/sin
```

然后看 `~/sin` 文件，会找到这样的一行

```
libm-2.1.2.so: 00009fa0 W sin
```

这样我就知道了 `sin` 在 `libm-2.1.2.so` 库里面，`-lm` 选项就可以了 (去掉前面的 `lib` 和后面的版本标志，就剩下 `m` 了所以是 `-lm`)。

```
gcc -o temp temp.c -lm
```

3.3 只知道函数的大概形式，怎么找到头文件。用 `man`

想知道 `fread` 这个函数的确切形式，我们只要执行 `man fread` 系统就会输出着函数的详细解释的。和这个函数所在的头文件说明了。

如果我们要 `write` 这个函数的说明，当我们执行 `man write` 时，输出的结果却不是我们所需要的。因为我们要的是 `write` 这个函数的说明，可是出来的却是 `write` 这个命令的说明。为了得到 `write` 的函数说明我们要用 `man 2 write` 表示我们用的 `write` 这个函数是系统调用函数，还有一个我们常用的是 `3` 表示函数是 C 的库函数。

4 引用文章全文

4.1 gcc 与 makefile

本文不会详细展开如何编写一个 `Makefile`。如想了解种种细节，请参考下面这个非常详细的教程，包含几乎 GNU `make` 的 `Makefile` 的所有细节：

[跟我一起写 Makefile](#)

而本文包含以下内容：

- `makefile` 小模板
- `gcc` 指令

`Makefile` 小模板

适用于纯 C 语言

```
# 指令编译器和选项
```

```
CC=gcc
```

```
CFLAGS=-Wall -std=gnu99
```

```
# 目标文件
```

```
TARGET=main
```

```
SRCS = main1.c \  
      main2.c \  
      main3.c
```

```
INC = -I./
```

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

```
$(TARGET):$(OBJS)
    $(CC) -o $@ $^
```

```
clean:
    rm -rf $(TARGET) $(OBJS)
```

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

注意：Makefile 有个规则就是命令行是以 tab 键开头，4 个空格或其他则会报错：

Makefile:2: *** missing separator。 stop

- 相比于单个文件和多个文件的 makefile，通过变量 INC 制定了头文件路径。头文件路径之间通过空格隔开。
- 编译规则%.o:%.c 中加入了头文件参数 \$(CC) \$(CFLAGS) \$(INC) -o \$@ -c \$<，
- 单个文件和多个文件的 makefile 相比增加了头文件路径参数。
- SRCS 变量中，文件较多时可通过 “\” 符号续行。
- \$@ --代表目标文件
- \$^ --代表所有的依赖文件
- \$< --代表第一个依赖文件 (最左边的那个)。

适用于 C/C++ 混合编译

目录结构如下：

```
httpserver
  main.cpp
  Makefile
  inc
    mongoose.h
    http_server.h

  src
    http_server.cpp
    mongoose.c
    ...
```

Makefile 如下：

```
CC=gcc
CXX=g++
```

编译器在编译时的参数设置, 包含头文件路径设置

```
CFLAGS=-Wall -O2 -g
CFLAGS+=-I $(shell pwd)/inc
CXXFLAGS=-Wall -O2 -g -std=c++11
CXXFLAGS+=-I $(shell pwd)/inc
```

库文件添加

```
LDFLAGS:=
LDFLAGS+=
```

指定源程序存放位置

```
SRCDIRS:=.
SRCDIRS+=src
```

设置程序中使用文件类型

```
SRCEXTS:=.c .cpp
```

设置运行程序名

```
PROGRAM:=httpserver
```

```
SOURCES=$(foreach d,$(SRCDIRS),$(wildcard $(addprefix $(d)/*,$(SRCEXTS))))
```

```
OBJS=$(foreach x,$(SRCEXTS),$(patsubst %$(x),%.o,$(filter %$(x),$(SOURCES))))
```

```
.PHONY: all clean distclean install
```

```
%.o: %.c
```

```
$(CC) -c $(CFLAGS) -o $@ $<
```

```
%.o: %.cxx
```

```
$(CXX) -c $(CXXFLAGS) -o $@ $<
```

```
$(PROGRAM): $(OBJS)
```

```
ifeq ($(strip $(SRCEXTS)),.c)
```

```
$(CC) -o $(PROGRAM) $(OBJS) $(LDFLAGS)
```

```
else
```

```
$(CXX) -o $(PROGRAM) $(OBJS) $(LDFLAGS)
```

```
endif
```

```
install:
```

```
install -m 755 -D -p $(PROGRAM) ./bin
```

```
clean:
```

```
rm -f $(shell find -name "*.o")
```

```
rm -f $(PROGRAM)
```

```
distclean:
```

```
rm -f $(shell find -name "*.o")
```

```
rm -f $(shell find -name "*.d")
```

```
rm -f $(PROGRAM)
```

```
all:
```

```
@echo $(OBJS)
```

gcc 指令

一步到位

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

多个程序文件的编译

```
gcc main1.c main2.c -o main
```

预处理

```
gcc -E main.c -o main.i
```

或

```
gcc -E main.c
```

gcc 的-E 选项，可以让编译器在预处理后停止，并输出预处理结果。

编译为汇编代码

预处理之后，可直接对生成的 `test.i` 文件编译，生成汇编代码：

```
gcc -S main.i -o main.s
```

`gcc` 的 `-S` 选项，表示在程序编译期间，在生成汇编代码后，停止，`-o` 输出汇编代码文件。

汇编

对于上文中生成的汇编代码文件 `test.s`，`gas` 汇编器负责将其编译为目标文件，如下：

```
gcc -c main.s -o main.o
```

连接

`gcc` 连接器是 `gas` 提供的，负责将程序的目标文件与所需的所有附加的目标文件连接起来，最终生成可执行文件。附加的目标文件包括静态连接库和动态连接库。

对于上一小节中生成的 `main.o`，将其与 C 标准输入输出库进行连接，最终生成可执行程序 `main`。

检错

参数 `-Wall`，使用它能够使 `GCC` 产生尽可能多的警告信息。

```
gcc -Wall main.c -o main
```

在编译程序时带上 `-Werror` 选项，那么 `GCC` 会在所有产生警告的地方停止编译，迫使程序员对自己的代码进行修改，如下：

```
gcc -Werror main.c -o main
```

创建动态链接库

生成生成 o 文件

`gcc -c -fPIC add.c` //这里一定要加上 `-fPIC` 选项，目的使库不必关心文件内函数位置

再编译

```
gcc -shared -fPIC -o libadd.so add.o
```

库文件连接

开发软件时，完全不使用第三方函数库的情况是比较少见的，通常来讲都需要借助许多函数库的支持才能够完成相应的功能。从程序员的角度看，函数库实际上就是一些头文件（`.h`）和库文件（`so`、或 `lib`、`dll`）的集合。虽然 `Linux` 下的大多数函数都默认将头文件放到 `/usr/include/` 目录下，而库文件则放到 `/usr/lib/` 目录下；但也有的时候，我们要用的库不在这些目录下，所以 `GCC` 在编译时必须用自己的办法来查找所需要的头文件和库文件。

额外补充：`Linux` 需要连接 `so` 库文件（带软连接），可以完完整整的复制到 `/usr/include/` 或 `/usr/lib/` 目录下，使用 `cp -d * /usr/lib/` 命令，然后别忘记再运行 `ldconfig`。

其中 `include` 文件夹的路径是 `/home/test/include`，`lib` 文件夹是 `/home/test/lib`，`lib` 文件夹中里面包含二进制 `so` 文件 `libtest.so`

首先要进行编译 `main.c` 为目标文件，这个时候需要执行：

```
gcc -c -I /home/test/include main.c -o main.o
```

最后把所有目标文件链接成可执行文件：

```
gcc -L /home/test/lib -ltest main.o -o main
```

默认情况下，`GCC` 在链接时优先使用动态链接库，只有当动态链接库不存在时才考虑使用静态链接库，如果需要的话可以在编译时加上 `-static` 选项，强制使用静态链接库。

```
gcc -L /home/test/lib -static -ltest main.o -o main
```

静态库链接时搜索路径顺序：

1. `ld` 会去找 `GCC` 命令中的参数 `-L`
2. 再找 `gcc` 的环境变量 `LIBRARY_PATH`
3. 再找内定目录 `/lib`、`/usr/lib`、`/usr/local/lib` 这是当初 `compile gcc` 时写在程序内的

动态链接时、执行时搜索路径顺序：

1. 编译目标代码时指定的动态库搜索路径

2. 环境变量 `LD_LIBRARY_PATH` 指定的动态库搜索路径
3. 配置文件 `/etc/ld.so.conf` 中指定的动态库搜索路径
4. 默认的动态库搜索路径 `/lib`
5. 默认的动态库搜索路径 `/usr/lib`

相关环境变量：

`LIBRARY_PATH` 环境变量：指定程序静态链接库文件搜索路径

`LD_LIBRARY_PATH` 环境变量：指定程序动态链接库文件搜索路径

5 实践中的一些经验

5.1 `eval` 和 `define` 中变量展开的坑

先上参考代码，下面代码中的错误，让我一阵好找，费几天时间。出现莫名其妙的错误，`DIR_STEM` 缺尾部的，`TBFILENAME` 引用不到，文件名中间被插入空格等等。原因都是行尾的引起。

```
define PROGRAM_template
#把文件分成4部分,基-干(DIR_STEM)-文件名.后缀名
DIR_STEM := $(subst $(DIR_BASE_OBJ),,$(dir $(1)))#XXX:这句语句执行完后展开后,行尾有\,会被视为连
TBFILENAME := $(subst .md,$(notdir $(1)))#XXX:此处因上面问题会连到上行
$(info $(TBFILENAME))#XXX:此处会显示不出东西来
#$(1): $(DIR_BASE_SRC)$$$(DIR_STEM)\$$$(TBFILENAME).rst
#$(1): $(DIR_BASE_SRC)$(subst $(DIR_BASE_OBJ),,$(dir $(1)))\$(subst .md,$(notdir $(1))).rst
#$(1): $(DIR_BASE_SRC)$$$(DIR_STEM)$$$(TBFILENAME).rst
#dep := $(DIR_BASE_SRC)$$$(DIR_STEM)\$$$(TBFILENAME).rst
#dep := $(patsubst %.md,%.rst,$(subst $(DIR_BASE_OBJ),$(DIR_BASE_SRC),$(1)))
dep := $(patsubst %.md,%.rst,$(subst $(DIR_BASE_OBJ),$(DIR_BASE_SRC),$(1)))
##不能直接写在[目标:依赖]里面,因为依赖里面带着模式匹配,有可能会使文件名乱套,未做实验再次证实,如果
#$(1): $(patsubst %.md,%.rst,$(subst $(DIR_BASE_OBJ),$(DIR_BASE_SRC),$(1)))
$(1): $$$(dep)
##必须要写成$$$(dep),$(dep)会使pandoc第一个参数为空。大概是因为命令集内部定义或组合生成的新变量要加
$(info $(1): $(dep))
pandoc $$< -o $$@
$$$(file >$(DIR_BASE_OBJ)-$$$(DIR_STEM)-$$$(TBFILENAME).tmp,$$(call def_hexo_md_head,$$TBFILENAME))
## 上面命令pandoc此处必须加$$,要不$<,$@会找不到,会出现pandoc -o 这样没有任何的参数带入的错误。花了
endif
## 写入文件的函数 $(file >xxx.xx,$(xxx)),这里要用$$$(file, $$$(call , 如果没有则在eval 的第一次展开

# 打散目标集合,一个一个送入命令集重组,同时用eval命令在makefile中使能。这样可以克服模式匹配依赖要一
$(foreach temp,$(OBJ_PATH_MDS),$(eval $(call PROGRAM_template,$(temp))))

改好好用的代码

$(OBJ_PATH_DIR):
#因为mkdir支持多目录同时写在一起,所以不用再用模式来拆开成一个一个了。
@echo "    MKDIR $@"
@mkdir $@

##定义一个命令包,来重新组合【目标:依赖】关系,配合$(eval)和foreach来使用。eval用来二次展开命令
##此处要注意的是,二次展开才用到的变量或函数要用$$,譬如自动变量$@等。
##define a function
#$(info $(TBFILENAME))
```



```

define PROGRAM_template
DIR_STEM := $(subst $(DIR_BASE_OBJ),,$(basename $(1)))
#TBFILENAME := $(subst .md,, $(notdir $(1)))
#$(1): $(DIR_BASE_SRC)$(DIR_STEM).rst
#dep := $(patsubst %.md,%.rst,$(subst $(DIR_BASE_OBJ),$(DIR_BASE_SRC),$(1)))
dep := $(basename $(subst $(DIR_BASE_OBJ),$(DIR_BASE_SRC),$(1))).rst
$(1): $(dep)
    @echo start hexo head output...
    $$ (file >$$@.tmp,$$(call def_hexo_md_head,$(subst .md,, $(notdir $(1)))))
#    @echo $$ (TBFILENAME)+2
#    @echo $(subst .md,, $(notdir $(1)))+1#直接函数填入才能取到。
    @echo convert to utf8
    iconv -f GBK -t UTF-8 $$@.tmp >$$@
    @echo start pandoc ...
    pandoc $$< -o - >>$$@
    @echo delete .tmp file...
    del $$@.tmp
    @echo copy .md file to hexo post...
    xcopy $$@ $(dir $(subst $(DIR_BASE_OBJ),$(DIR_BASE_HEXO_POST),$(1))) /y
endif

```

打散目标集合, 一个一个送入命令集重组, 同时用eval命令在makefile中使能。这样可以克服模式匹配依赖要一: \$(foreach temp,\$(OBJ_PATH_MDS),\$(eval \$(call PROGRAM_template,\$(temp))))

- 行尾有, 后一行的变量名被连上来了

```

define function
DIR_STEM := $(dir $(1))#这个不是出现在define中是没有关系的。但此处就有可能有问题
endif

```

或者

```
DIR_STEM := c:\tmp\
```

- eval 和 define

define 只是一堆文字, 在引用的地方展开, 但是并不作为 makefile 的一部分, 即展开的变量不会出现在 makefile 变量空间中, 1tab 缩进的命令会在展开时执行。

eval 则表示会有 2 次展开, 第一次展开和 define 一样。第二次展开是把展开的内容变为 makefile 变量等空间的一部分, 可以真正引用到。

eval 2 次展开才引用到的变量要用 \$\$, 自动变量也一样, 新生成变量也一样, define 中创建的变量也一样, eval 外面已经有的变量不用加双 \$, 案例参考上面代码。函数也一样, 如果是要在 2 次展开时, 才启动执行的话, 就需要加 \$\$ 延迟 defer

5.2 输出文件的方法

- \$(file >\$\$@.tmp,\$\$(call def_hexo_md_head,\$\$(TBFILENAME)))
- > 和 » 法

5.3 一些工具

- iconv 文件编码转换

因 `pandoc` 和 `Hexo` 都只支持 UTF-8 的编码形式，而中文版 windows 缺省输出的是 GBK 的中文编码，如果直接用 » 把 `pandoc` 的输出重定向到 GBK 编码的文件中时，会出现什么也没有输出的现象。这里就需要 `iconv` 来做一下转换了。

```
echo start hexo head output...
$$ (file >$$@.tmp,$$(call def_hexo_md_head,$$(TBFILENAME)))
echo convert to utf8
iconv -f GBK -t UTF-8 $$@.tmp >$$@
echo start pandoc ...
pandoc $$< -o - >>$$@
```

5.4 调试输出变量信息方式

- 输出信息方式为:

```
$(warning xxx)
$(error xxx)
$(info xxx)
```

- 输出变量方式为:

```
$(info $(dir $(1)))
$(warning $(XXX))
```