make

make and makefile

kevinluo

Contents

目錄

1 make

1.1 gnu make 安裝

- make 官方下載地址

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

- make4.2（GNU make）的安裝步驟

  1. 解壓

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

  2. 安裝

     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. 打開新的窗口，驗證

```
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

這篇文章寫得短小全面。外鏈-> gcc 與 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 變量 $@，$^，$< 代表的意義

makefile 中 $@ $^ %< 使用

$@ 目標文件，$^ 所有的依賴文件，$< 第一個依賴文件。

這是再一次簡化後的 Makefile

```
main: main.o mytool1.o mytool2.o
gcc -o $@ $^
.c.o:
gcc -c $<
```

3.2 怎麼查到函數是哪個庫的?

有時候我們使用了某個函數，但是我們不知道庫的名字，這個時候怎麼辦呢?

比如我要找 sin 這個函數所在的庫。就祇好用命令

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

然後看 ~/sin 文件，會找到這樣的一行

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

這樣我就知道了 sin 在 libm-2.1.2.so 庫裏面，-lm 選項就可以了 (去掉前面的 lib 和後面的版本標志，就剩下 m 了所以是 -lm)。

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

3.3 祇知道函數的大概形式，怎麼找到頭文件。用 man

想知道 fread 這個函數的確切形式，我們祇要執行 man fread 系統就會輸出着函數的詳細解釋的。和這個函數所在的頭文件說明了。

如果我們要 write 這個函數的說明，當我們執行 man write 時，輸出的結果却不是我們所需要的。因爲我們要的是 write 這個函數的說明，可是出來的却是 write 這個命令的說明。爲了得到 write 的函數說明我們要用 man 2 write 2 表示我們用的 write 這個函數是系統調用函數，還有一個我們常用的是 3 表示函數是 C 的庫函數。

4 引用文章全文

4.1 gcc 與 makefile

本文不會詳細展開如何編寫一個 Makefile。如想了解種種細節，請參考下面這個非常詳細的教程，包含幾乎 GNU make 的 Makefile 的所有細節:

跟我一起寫 Makefile

而本文包含以下內容:

- makefile 小模板
- gcc 指令

Makefile 小模板

適用于純C語言

```
# 指令編譯器和選項
CC=gcc
CFLAGS=-Wall -std=gnu99

# 目標文件
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 有個規則就是命令行是以 tab 鍵開頭，4 個空格或其他則會報錯：
`Makefile:2: *** missing separator。stop`

- 相比于單個文件和多個文件的 makefile，通過變量 INC 制定了頭文件路徑。頭文件路徑之間通過空格隔開。
- 編譯規則%.o:%.c 中加入了頭文件參數 $(CC) $(CFLAGS) $(INC) -o $@ -c $<,
- 單個文件和多個文件的 makefile 相比增加了頭文件路徑參數。
- SRCS 變量中，文件較多時可通過“\”符號續行。
- $@ --代表目標文件
- $^ --代表所有的依賴文件
- $< --代表第一個依賴文件 (最左邊的那個)。

適用于 C/C++ 混合編譯

目錄結構如下：

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

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

Makefile 如下：

```
CC=gcc
CXX=g++

# 編譯器在編譯時的參數設置,包含頭文件路徑設置
CFLAGS:=-Wall -O2 -g
CFLAGS+=-I $(shell pwd)/inc
CXXFLAGS:=-Wall -O2 -g -std=c++11
CXXFLAGS+=-I $(shell pwd)/inc

# 庫文件添加
LDFLAGS:=
LDFLAGS+=

# 指定源程序存放位置
SRCDIRS:=.
SRCDIRS+=src

# 設置程序中使用文件類型
SRCEXTS:=.c .cpp
```

```
# 設置運行程序名
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
```

多個程序文件的編譯

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

預處理

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

或

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

gcc 的-E 選項，可以讓編譯器在預處理後停止，并輸出預處理結果。

編譯爲匯編代碼

預處理之後，可直接對生成的 test.i 文件編譯，生成匯編代碼:
```
gcc -S main.i -o main.s
```
gcc 的-S 選項，表示在程序編譯期間，在生成匯編代碼後，停止，-o 輸出匯編代碼文件。

匯編

對于上文中生成的匯編代碼文件 test.s，gas 匯編器負責將其編譯爲目標文件，如下:
```
gcc -c main.s -o main.o
```

連接

gcc 連接器是 gas 提供的，負責將程序的目標文件與所需的所有附加的目標文件連接起來，最終生成可執行文件。附加的目標文件包括靜態連接庫和動態連接庫。
對于上一小節中生成的 main.o，將其與C標準輸入輸出庫進行連接，最終生成可執行程序 main。

檢錯

參數-Wall，使用它能够使 GCC 産生盡可能多的警告信息。
```
gcc -Wall main.c -o main
```
在編譯程序時帶上-Werror 選項，那麽 GCC 會在所有産生警告的地方停止編譯，迫使程序員對自己的代碼進行修改，如下:
```
gcc -Werrormain.c -o main
```

創建動態鏈接庫

生成生成 o 文件
```
gcc -c -fPIC add.c //這裏一定要加上-fPIC 選項，目的使庫不必關心文件內函數位置
```
再編譯
```
gcc -shared -fPIC -o libadd.so add.o
```

庫文件連接

開發軟件時，完全不使用第三方函數庫的情況是比較少見的，通常來講都需要借助許多函數庫的支持才能够完成相應的功能。從程序員的角度看，函數庫實際上就是一些頭文件（.h）和庫文件（so、或 lib、dll）的集合。雖然 Linux 下的大多數函數都默認將頭文件放到/usr/include/目録下，而庫文件則放到/usr/lib/目録下；但也有的時候，我們要用的庫不在這些目録下，所以 GCC 在編譯時必須用自己的辦法來查找所需要的頭文件和庫文件。
額外補充：Linux 需要連接 so 庫文件（帶軟連接），可以完完整整的復制到/usr/include/或/usr/lib/目録下，使用 cp -d * /usr/lib/ 命令，然後别忘記再運行 ldconfig。

其中 inclulde 文件夾的路徑是/home/test/include,lib 文件夾是/home/test/lib,lib 文件夾中裏面包含二進制 so 文件 libtest.so
首先要進行編譯 main.c 爲目標文件，這個時候需要執行:
```
gcc  c  I /home/test/include main.c  o main.o
```
最後把所有目標文件鏈接成可執行文件:
```
gcc  L /home/test/lib -ltest main.o  o main
```

默認情況下，GCC 在鏈接時優先使用動態鏈接庫，衹有當動態鏈接庫不存在時才考慮使用靜態鏈接庫，如果需要的話可以在編譯時加上-static 選項，强制使用靜態鏈接庫。
```
gcc  L /home/test/lib -static -ltest main.o  o main
```

靜態庫鏈接時搜索路徑順序：

1. ld 會去找 GCC 命令中的參數-L
2. 再找 gcc 的環境變量 LIBRARY_PATH
3. 再找内定目録 /lib、/usr/lib、/usr/local/lib 這是當初 compile gcc 時寫在程序内的

動態鏈接時、執行時搜索路徑順序:

1. 編譯目標代碼時指定的動態庫搜索路徑
2. 環境變量 LD_LIBRARY_PATH 指定的動態庫搜索路徑

3. 配置文件/etc/ld.so.conf 中指定的動態庫搜索路徑
4. 默認的動態庫搜索路徑/lib
5. 默認的動態庫搜索路徑/usr/lib

相關環境變量：
LIBRARY_PATH 環境變量：指定程序靜態鏈接庫文件搜索路徑
LD_LIBRARY_PATH 環境變量：指定程序動態鏈接庫文件搜索路徑

5 實踐中的一些經驗

5.1 eval 和 define 中變量展開的坑

先上參考代碼，下面代碼中的錯誤，讓我一陣好找，費幾天時間。出現莫名其妙的錯誤，DIR_STEM 缺尾部的，TBFILENAME 引用不到，文件名中間被插入空格等等。原因都是行尾的引起。

```
define PROGRAM_template
#把文件分成4部分,基-幹(DIR_STEM)-文件名.後綴名
DIR_STEM := $(subst $(DIR_BASE_OBJ),,$(dir $(1)))#XXX:這句語句執行完後展開後，行尾有\,會被視爲
TBFILENAME := $(subst .md,,$(notdir $(1)))#XXX:此處因上面問題會連到上行
$(info $(TBFILENAME))#XXX:此處會顯示不出東西來
#$(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),$(1)))
##不能直接寫在[目標:依賴]裏面,因爲依賴裏面帶着模式匹配,有可能會使文件名亂套,未做實驗再次證實
#$(1): $(patsubst %.md,%.rst,$(subst $(DIR_BASE_OBJ),$(DIR_BASE_SRC),$(1)))
$(1): $$(dep)
##必須要寫成$$(dep),$(dep)會使pandoc第一個參數爲空。大概是因爲命令集內部定義或組合生成的新變
 $(info $(1): $(dep))
 pandoc $$< -o $$@
 $$(file >$(DIR_BASE_OBJ)-$$(DIR_STEM)-$$(TBFILENAME).tmp,$$(call def_hexo_md_head,$$TBFILENA
## 上面命令pandoc此處必須加$$,要不$<,$@會找不到,會出現pandoc -o 這樣沒有任何的參數帶入的錯誤。
endef
## 寫入文件的函數 $(file >xxx.xx,$(xxx)),這裏要用$$(file, $$(call , 如果沒有則在eval 的第一次

# 打散目標集合,一個一個送入命令集重組,同時用eval命令在makefile中使能。這樣可以克服模式匹配依
$(foreach temp,$(OBJ_PATH_MDS),$(eval $(call PROGRAM_template,$(temp))))
```

改好好用的代碼

```
$(OBJ_PATH_DIR):
#因爲mkdir支持多目錄同時寫在一起,所以不用再用模式來拆開成一個一個了。
 @echo "   MKDIR $@..."
 @mkdir $@

##定義一個命令包，來重新組合【目標:依賴】關系，配合$(eval ) 和foreach 來使用。eval用來二次展開
##此處要注意的是，二次展開才用到的變量或函數要用$$,譬如自動變量$@等。
##define a function
#$(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#直接函數填入才能取到。
 @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
endef

# 打散目標集合,一個一個送入命令集重組,同時用eval命令在makefile中使能。這樣可以克服模式匹配依
$(foreach temp,$(OBJ_PATH_MDS),$(eval $(call PROGRAM_template,$(temp))))
```

- 行尾有,後一行的變量名被連上來了

  ```
  define function
  DIR_STEM := $(dir $(1))#這個不是出現在define中是沒有關系的。但此處就有可能有問題
  endef
  ```

  或者

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

- eval 和 define

  define 祇是一堆文字，在引用的地方展開，但是并不作爲 makefile 的一部分，即展開的變量不會出現在 makefile 變量空間中，1tab 縮進的命令會在展開時執行。

  eval 則表示會有 2 次展開，第一次展開和 define 一樣。第二次展開是把展開的內容變爲 makefile 變量等空間的一部分，可以真正引用到。

  eval 2 次展開才引用到的變量要用 $$, 自動變量也一樣，新生成變量也一樣，define 中創建的變量也一樣，eval 外面已經有的變量不用加雙 $，案例參考上面代碼。函數也一樣，如果是要在 2 次展開時，才啓動執行的話，就需要加 $$ 延遲 defer

5.2 輸出文件的方法

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

5.3 一些工具

- iconv 文件編碼轉換

因 pandoc 和 Hexo 都祇支持 UTF-8 的編碼形式，而中文版 windows 缺省輸出的是 GBK 的中文編碼，如果直接用 » 把 pandoc 的輸出重定向到 GBK 編碼的文件中時，會出現什麼也沒有輸出的現象。這裏就需要 iconv 來做一下轉換了。

```
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 調試輸出變量信息方式

- 輸出信息方式爲:

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

- 輸出變量方式爲:

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