makefile終極奧義

原創 deroy 編程學習基地 今天

點擊藍字 ☰ 關注我們

什麼是makefile？

或許很多 Winodws 的程序員都不知道這個東西，因為那些 Windows 的 IDE 都為你做了這個工作，但是一個好的和 professional 的程序員， makefile 還是要懂。這就好像現在有這麼多的 HTML 的編輯器，但如果你想成為一個專業人士，你還是要了解 HTML 的標識的含義。特別在 Unix 下的軟件編譯，你就不能不自己寫 makefile 了，「會不會寫 makefile，從一個側面說明了一個人是否具備完成大型工程的能力」。因為 makefile 關係到了整個工程的編譯規則。

上期「用GCC寫個庫給你玩」已經詳細介紹了GCC編譯鏈接的過程，那麼接下來就聊聊makefile藝術。

makefile介紹

make 命令執行時，需要一個 Makefile 文件，以告訴 make 命令需要怎麼樣的去編譯和鏈接程序。

makefile 的規則是：

1.如果這個工程沒有編譯過，那麼我們的所有C 文件都要編譯並被鏈接。

2.如果這個工程的某幾個C 文件被修改，那麼我們只編譯被修改的C 文件，並鏈接目標程序。

3.如果這個工程的頭文件被改變了，那麼我們需要編譯引用了這幾個頭文件的C文件，並鏈接目標程序。

只要我們的 `Makefile` 寫得夠好，所有的這一切，我們只用一個 `make` 命令就可以完成， `make` 命令會自動智能地根據當前的文件修改的情況來確定哪些文件需要重編譯，從而自己編譯所需要的文件和鏈接目標程序。

makefile的規則

在講述這個makefile之前，還是讓我們先來粗略地看一看makefile的規則。

```
target ... : prerequisites ...
    command
    ...
    ...
```

「`target`」

可以是一個 `object file` （目標文件），也可以是一個執行文件。

「`prerequisites`」

生成該 `target` 所依賴的文件和/或 `target`

「command」

該 target 要執行的命令（任意的 shell 命令）

一個示例

首先還是使用上期「編譯鏈接，你還不會用GCC生成標準庫」的測試代碼

```
div.c add.c div.c mult.c sub.c head.h

├── Calculator
├── add.c
├── div.c
├── head.h
├── main.c
├── mult.c
└── sub.c
```

那麼我們需要通過 makefile 將示例代碼編譯生成目標文件 app .

第一個版本

```
app:sub.o mult.o div.o add.o main.o
 gcc sub.o mult.o div.o add.o main.o -o app
sub.o:sub.c
 gcc -c sub.c
mult.o:mult.c
 gcc -c mult.c
div.o:div.c
 gcc -c div.c
add.o:add.c
```

```
 gcc -c add.c
main.o:main.c
 gcc -c main.c
#伪指令
.PHONY:clean
clean:
 rm -f sub.o mult.o div.o add.o main.o app
```

執行 **make** 命令看一下效果，很 **nice**

```
[root@Calculator]# make
gcc -c sub.c
gcc -c mult.c
gcc -c div.c
gcc -c add.c
gcc -c main.c
gcc sub.o mult.o div.o add.o main.o -o app
```

思考：為什麼寫這麼複雜，先生成 **.o** 再生成 **.c**

> 直接說答案：**「方便編譯鏈接」**

小實驗：修改 **add.c** 裡面的內容，隨便按一個空格，然後保存退出再執行 **make** 命令

```
[root@Calculator]# make
gcc -c add.c
gcc sub.o mult.o div.o add.o main.o -o app
```

結果很明顯：只對 `add.c` 進行了編譯，省略了沒有必要的編譯步驟

為什麼會這樣？那就要說說 `make` 是如何工作的

make是如何工作的

在默认的方式下，也就是我们只输入 `make` 命令。那么，

1. make会在当前目录下找名字叫 **“Makefile”或“makefile”** 的文件。
2. 如果找到，它会找文件中的第一个目标文件（target），在上面的例子中，他会找到“app”这个文件，并把这个文件作为最终的目标文件。
3. 如果app文件不存在，或是app所依赖的后面的 `.o` 文件的文件修改时间要比 `edit` 这个文件新，那么，他就会执行后面所定义的命令来生成 `edit` 这个文件，节省了没有必要的编译步骤。
4. 如果 `edit` 所依赖的 `.o` 文件也不存在，那么make会在当前文件中找目标为 `.o` 文件的依赖性，如果找到则再根据那一个规则生成 `.o` 文件。（这有点像一个堆栈的过程）
5. 当然，你的C文件和H文件是存在的啦，于是make会生成 `.o` 文件，然后再用 `.o` 文件生成make的终极任务，也就是执行文件 `edit` 了。

这就是整个 `make` 的依赖性， `make` 会一层又一层地去找文件的依赖关系，直到最终编译出第一个目标文件。

上述还只是简单的 `makefile` ，属于**「显式规则」**，那么为了优化 `makefile` 我们介绍**「隐式规则」**

makefile中使用变量

在 Makefile 中我们要定义一系列的变量，变量一般都是字符串，这个有点「类似C语言中的宏」，当 Makefile 被执行时，其中的变量都会被扩展到相应的引用位置上，可以直接把变量当成C语言中的宏理解。

> Makefile中变量有四种定义(赋值)方式:
>
> 1,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（推荐使用）
>
> 2,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
>
> 3,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。
>
> 4,追加赋值(+=) 原变量用空格隔开的方式追加一个新值

使用变量非常简单，变量在声明时需要给予初值，而在使用时，需要给在变量名前加上 $ 符号，但最好用小括号 () 或是大括号 {} 把变量给包括起来。

```
OBJ:=main.o #定义变量
 #引用变量
${OBJ}
 #使用变量
$(OBJ)   #推荐使用
```

除了自己定义的变量之外makefile还提供了预定义的变量

> 在隐含规则中的命令中，基本上都是使用了一些预先设置的变量。你可以在你的makefile中改变这些变量的值，或是在make的命令行中传入这些值，或是在你的环境变量中设置这些值

命令的变量

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

命令参数的变量

命令	意义
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
LDFLAGS	链接器参数。（如：ld ）

隐晦规则

> 如果我们想定义一系列比较类似的文件，我们很自然地就想起使用通配符。

通配符

符号	含义
%	任意一个
?	匹配一个字符
*	所有

GNU的make很强大，它可以自动推导文件以及文件依赖关系后面的命令

例如：

只要make看到一个 `.o` 文件，它就会自动的把 `.c` 文件加在依赖关系中，如果make找到一个 `main.o` ，那么 main.c 就会是 main.o` 的依赖文件。

```
OBJ:=sub.o mult.o div.o add.o main.o
$(TARGET):$(OBJ)
 ${CC} $(OBJ) -o $(TARGET)
```

有.o文件没有.c文件，makefile会自动推导生成.o文件

除了通配符，makefile还提供了自动推导的自动变量

自动变量

符号	含义
$@	代表目标文件
$^	代表所有依赖文件

符号	含义
$<	代表第一个依赖文件

由此第二个版本出来了

第二个版本

```
CC:=gcc
TARGET:=app      #目标变量
OBJ:=sub.o mult.o div.o add.o main.o
$(TARGET):$(OBJ)
 ${CC} $(OBJ) -o $(TARGET)
.PHONY:clean
clean:
 $(RM) $(OBJ) $(TARGET)
```

已经很精简了是不是

伪指令

在第一第二个版本的makefile里面我都有写 `.PHONY:clean` 这个规则，并且在make的时候并没有执行这个规则。其实 `.PHONY` 表示 `clean` 是一个“伪目标”，并不在make的执行命令中，只有指定才会执行例如: `make clean`

比较健壮的伪指令写法是：

```
```

```
.PHONY:clean
clean:
 -rm -f $(OBJ) $(TARGET)
```

在 `rm` 命令前面加了一个小减号的意思就是，也许某些文件出现问题，但不要管，继续做后面的事.

函数

在 `Makefile` 中可以使用函数来处理变量，从而让我们的命令或是规则更为的灵活和具有智能。下面介绍三个最常用的函数

文本处理函数

「wildcard」

```
$(wildcard PATTERN...)
```

功能：该函数被展开为已经存在的、使用空格分开的、匹配此模式的所有文件列表。

「举例」

获取工作目录下的所有 `.c` 文件列表

```
SRC:=$(wildcard *.c)
```

字符串替换函数

「patsubst」

```
$(patsubst <pattern>,<replacement>,<text>)
```

功能：查找 `<text>` 中的单词（单词以“空格”、“Tab”或“回车”“换行”分隔）是否符合模式 `<pattern>` ，如果匹配的话，则以 `<replacement>` 替换。

「举例」

```
SRC:=$(wildcard *.c)
OBJ:=$(patsubst %.c,%.o,$(SRC)) #将SRC里面的.c文件替换成.o文件
```

shell函数

shell函数也不像其它的函数。顾名思义，它的参数应该就是操作系统Shell的命令。

```
$(shell <option>)
```

「举例」

```
SRC = $(shell find . -name "*.c")
```

将当前目录及其子目录下所有文件后缀为 「.c」 的文件以空格为限赋值给 SRC

最终版本

先总结一下前面都讲了些什么

Makefile里主要包含了五个东西：「显式规则」、「隐晦规则」、「变量定义」、「函数」、「注释」。

1. 显式规则。显式规则说明了如何生成一个或多个目标文件。这是由Makefile的书写者明显指出要生成的文件、文件的依赖文件和生成的命令。
2. 隐晦规则。由于我们的make有自动推导的功能，所以隐晦的规则可以让我们比较简略地书写 Makefile，这是由make所支持的。
3. 变量的定义。在Makefile中我们要定义一系列的变量，变量一般都是字符串，这个有点像你C语言中的宏，当Makefile被执行时，其中的变量都会被扩展到相应的引用位置上。
4. 函数。其包主要介绍了三个函数，一个是提取工作目录下的所有.c文件列表，另外一个就是将提取的.c列表转换成.o列表，最后就是shell函数，可以执行任何shell操作.
5. 注释。Makefile中只有行注释，和UNIX的Shell脚本一样，其注释是用 # 字符，这个就像C/C++中的 // 一样。如果你要在你的Makefile中使用 # 字符，可以用反斜杠进行转义，如：\# 。

然后「最终版本确定」，可以作为「模板」使用

```
TARGET=app
SRC = $(wildcard *.c)
OBJ = $(patsubst %.c,%.o,$(SRC))
DEFS =-DDEBUG
CFLAGS =-g
CC =gcc
LIBS =
$(TARGET):$(OBJ)
 $(CC) $(CFLAGS) $(DEFS) -o $@ $^ $(LIBS)
.PHONY:
clean:
 rm -rf *.o $(TARGET)
```

多目录makefile

作为一个健壮的makefile怎么能将所有代码放在一个文件夹下面呢？优秀的工程师都是分模块标准放置

先看一下目录树形结构

```
├── add
│   ├── add.c
│   └── Makefile
├── div
│   ├── div.c
│   └── Makefile
├── include
│   └── head.h
├── main.c
```

```
├── Makefile
├── Makefile.build
├── mult
│   ├── Makefile
│   └── mult.c
└── sub
    ├── Makefile
    └── sub.c
```

示例程序的Makefile分为3类:

1. 顶层目录的Makefile
2. 顶层目录的Makefile.build
3. 各级子目录的Makefile

「各级子目录的Makefile」

这个是最简单的，只需要 **obj-y+=** 将所有.o文件或者子级目录添加即可，例如

sub文件夹下的 makefile

```
obj-y += sub.o
```

「顶层目录的Makefile」

它除了定义obj-y来指定根目录下要编进程序去的文件、子目录外，主要是定义工具链、编译参数

```
CFLAGS = -g        #编译器参数
CFLAGS += -I $(shell pwd)/include #指定 include 包含文件的搜索目录
LDFLAGS := -lm       #链接器参数

export CFLAGS LDFLAGS

TOPDIR := $(shell pwd)
export TOPDIR

TARGET := app


obj-y += main.o
obj-y += sub/
obj-y += mult/
obj-y += div/
obj-y += add/

all :
 make -C ./ -f $(TOPDIR)/Makefile.build
 $(CC) $(LDFLAGS) -o $(TARGET) built-in.o


clean:
 rm -f $(shell find -name "*.o")
 rm -f $(TARGET)

distclean:
 rm -f $(shell find -name "*.o")
 rm -f $(shell find -name "*.d")
 rm -f $(TARGET)
```

「顶层目录的Makefile.build」

這是最複雜的部分，它的功能就是把某個目錄及它的所有子目錄中、需要編進程序去的文件都編譯出來，打包為.o文件.

這個也不好演示，但都是模板，改一些參數就可以直接用，後台發送關鍵字 **makefile** 獲取**「測試源碼」**和**「makefile文件」**

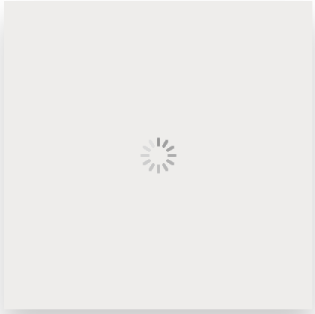
關鍵字【俄羅斯方塊】

End

作者：夢凡

夢想在，終不凡~

你們的在看就是對我最大的肯定，
點個在看好嗎~



編程學習基地

常回基地看看

喜歡此內容的人還喜歡

C++ 牛逼!

編程指北

【進階】同事用#include"xxx.c"把我給驚呆了！！

最後一個bug

嵌入式C高效編程(經驗、乾貨一網打盡)

最後一個bug