2014年09月24日

从简单实例开始，学会写Makefile（二）

如果文件间存在着相互之间的引用关系该怎么办？如果把.h文件和.cpp文件放在了不同的目录下该怎么办？如果我想生成静态库，然后在其他地方引用静态库该怎么办？如果我想将程序迁移到Unix平台下，使用不同的编译器，难道要依次修改所有的Makefile？

.d文件，解决文件间的相互引用

自动生成依赖关系

在前文的项目基础上，考虑一下这种情况：如果我们在w1.h文件里包含了头文件w2.h以及w3.h并且用到其中定义的函数。

第一次编译没有遇到问题，但是如果后续的开发过程中修改了w2.h或者w3.h文件中的内容，再执行gmake命令的时候，就遇到问题了——w1.cpp文件不会被重新编译了！

显然，我们需要将生成目标文件w1.o的规则的依赖项加上w2.h和w3.h。可是如果手动的去检查每一个文件的引用关系，然后修改Makefile文件，这样做的效率就太低了。

万幸的是，编译器可以帮助我们自动生成依赖关系，只需要在编译命令中加上“-M”选项，就可以让编译器自动寻找源文件中包含的头文件，并生成一个依赖关系，例如，你可以在shell界面下敲下如下的命令：

```
g++-MM w1.cpp
```

可以看到，其输出为

```
w1.o:w1.cpp w2.h w3.h。
```

这里需要特别注意的是，我们使用“-MM”而不是“-M”，因为我们使用的是GUN的C/C++编译器，使用“-M”参数会将标准库的头文件也一并包含进来，但这并不是我们想要的，而使用“-MM”则不会。

现在的问题是，如何利用这个命令去写好我们的Makefile呢？

GUN组织建议把每一个源文件自动生成的依赖关系放到一个.d文件中，让每一个.cpp文件都对应一个.d文件，例如之前的w1.cpp，我们可以生成一个w1.d文件，内容为自动生成的依赖关系 w1.o:w1.cpp w2.h w3.h，然后在Makefile中包含所有的.d文件，我们只需要写出.cpp文件和.d文件的依赖关系，让make自动更新或生成.d文件即可。

生成.d文件

```
dep/%.d:%.cpp
    @if test ! -d "dep"; then\
        mkdir -p dep;\
    fi; \
    set -e; rm -f $@;
    g++ -MM $< > $@.$$$$; \
    sed 's/$*\.o[ :]*/obj\/$*\.o dep\/$*\.d: /g' < $@.$$$$ > $@; \
    rm -f $@.$$$$
```

在Makefile中加上如上的代码，就可以生成我们所需要的.d文件了。

又是一堆莫名其妙的符号，我们还是来逐句进行分析。

dep/%.d: %.cpp

使所有的.d文件依赖于对应的.cpp文件，也就是说只要.cpp更新了，我们就重新生成对应的.d文件。这里和.o文件类似的，我们也创建一个dep目录用来存放所有的.d文件，既能保持项目文件的整洁和统一，也方便管理。

@if test ! -d “dep”; then

```
@if test ! -d "dep"; then\
    mkdir -p dep;\
fi; \
```

检查当前目录下是否存在dep目录，如果不存在，就使用mkdir命令创建dep目录。

set -e; rm -f $@;

set–e 的作用是如果命令执行出错就直接退出。$@的含义之前已经说过，这里rm –f $@的意思就是删除所有的目标文件。

g++ -MM $< > $@.$$$$;

$< 的含义是第一个依赖项的名称，> 是重定向符号，将输出结果重定向到指定文件中。$@.$$$$ 就是这个文件的文件名，其中“$$$$”表示一个随机的编号，例如如果有目标文件是w1.d，那么“$@.$$$$”一个可能的结果就是w1.d.12345。那么，这句话的含义就是将g++ -MM w1.cpp的输出结果重定向到w1.d.12345这个文件中。

sed 's/$*.o[:]*/obj\/$*.o dep\/$*.d : /g' < $@.$$$$ > $@;

这里使用了sed这个工具对文本进行替换处理，单引号中的规则是's/old/new/g'，s表示替换，末尾的g代表全局的意思，对文本中所有符合要求的字符串进行替换，sed会将符合old模式的字符串替换为new，具体的使用方法可以查阅一下sed这个工具的帮助文档。

```
<$@$$$$，将这个文件的内容作为sed工具的输入。

>$@，将sed处理后的内容重定向输出到这个文件中。
```

经过这一步的处理后，就把自动生成的依赖关系：

```
w1.o:w1.cpp w2.h w3.h
```

转成：

```
w1.o w1.d:w1.cpp w2.h w3.h
```

这样，我们的.d文件也会自动更新啦。

rm -f $@.$$$$

删除掉这个临时文件。

使用include包含其他文件

在Makefile中我们也可以像在C++文件中那样包含其他文件。

现在在我们的Makefile中加上这样一句：

```
include w1.d
```

使用这个语句就可以将之前我们生成的.d文件中的内容包含到当前的Makefile中。

当然，也可以用这个命令来包含其他的Makefile文件。具体的用法后面再进行介绍。

我们希望把所有的.d文件都包含在当前的Makefile中。

先定义一个变量，存放所有的.d文件名：

```
DEPS = $(addsuffix .d,$(addprefix dep/,$(BASE)))
```

然后使用include$(DEPS) 包含所有的.d文件。

-I，引用其他目录下的.h文件

考虑这种情况：现在有两个目录，一个inc目录用来存放.h文件，一个src目录，用来存放.cpp文件。怎么让编译器找到引用的.h文件在哪个目录下呢？

我们可以使用“-I”选项。 格式为“-I目录名”，这样在编译的时候，编译器就会依次到我们指定的目录中寻找.h文件。

同样，先定义一个变量，存放所有头文件的目录名：

```
INCLUDEDIR = -I../inc
```

然后将

```
g++ -c -o $@ $<
```

这样的编译命令中写成

g++ -c -o $@ $(INCLUDEDIR) $<

OK，再来尝试用gmake命令编译一下吧，已经可以成功编译了。

如果需要包含多个目录下的.h文件，可以重复使用-I选项，中间需要用空格隔开。

使用静态库

修改生成静态库的Makefile

有的时候我们不需要生成一个可执行的程序，而是生成一个静态库文件，之后在其他的地方引用这个静态库文件。

假设我们的项目目录结构是这样的，src是项目根目录，src下面有common和app以及lib两个目录，common和app下面都有inc和src两个目录。common存放公共库的源文件，app存放程序源文件，lib存放生成的静态库。

修改我们在common目录下的Makefile文件：

```
top_srcdir = ../..
#生成静态库后所存放的位置
libdir = $(top_srcdir)/lib
#静态库文件名
LIBNAME = libfa_common.a
#路径+静态库文件名
TARGET = $(libdir)/$(LIBNAME)

$(TARGET): $(OBJS)
    -rm -f $@
    ar cr $(TARGET) $(OBJS)
```

- top_srcdir是项目根目录的路径，使用相对路径，方便我们在后面引用其他目录。
- libdir是生成的静态库所存放的路径。
- LIBNAME是静态库名称，注意，静态库的命名必须以“lib”开头，以“.a”结尾。
- TARGET是目标文件名称，包含路径。
- 在生成静态库文件的规则中，使用ar这个命令。

修改引用静态库的Makefile

在app/src目录下的源文件中，编译的时候需要引用libfa_common.a这个静态库，这就需要我们再修改app目录下的Makefile文件。

这里使用了两个新的参数，“-l”和“-L”。

“-l”参数指定要引用的库的名称。例如我们要引用libfa_common.a这个静态库，那么需要在编译命令里加上“-lfa_common”，可以看出，-l后面的库名称需要去除前面的“lib”和后面的“.a”。

“-L”参数指定了要引用的库的目录，用法和之前的“-I”一样。这里需要注意的是，我们需要修改一下VPATH这个变量，指明要引用的静态库的目录。类似这样：

```
VPATH:= -L $(top_srcdir)/lib
```

完整的Makefile

其实在每一个目录下的Makefile中有很多部分是重复的，我们可以考虑将重复的部分提取出来，单独放在一个公共的Makefile中，然后在其他Makefile中用include包含这个公共的Makefile即可。

我写了三套Makefile，分别是Makefile（app）、Makefile（lib）、Make.rules。

其中，Make.rules是公共部分，Makefile（app）是用来生成可执行程序的，Makefile（lib）是用来生成静态库的，为了以后迁移方便，考虑到Linux和Unix平台的差异，以及各个编译器之间的差异，可以将各种命令也定义成变量，之后使用宏定义进行条件编译。

贴一下完整的Makefile代码。

Make.rules

```
#公用Make规则配置

#设置编译器类型
CXX := g++
CC := gcc

#设置编译.d文件相关内容
DEPFLAGS := -MM
DEPFILE = $@.$$$$

#设置所有静态库文件所在位置，会根据每个Makefile文件的top_srcdir设置相对位置
LIBDIR := $(top_srcdir)/lib

#设置编译程序时需要在哪些目录查找静态库文件
LDFLAGS := -L.\
           -L$(top_srcdir)/lib

#设置VPATH，在检查依赖关系时，如果查找-lxxxx时，在哪些目录查找静态库文件
VPATH := $(LIBDIR)

#设置编译程序时查找头文件的目录位置
INCLUDEDIR := -I.\
              -I../inc\

#声明要生成的目标文件，具体规则在具体的Makefile中定义
$(TARGET):

#生成.o文件所依赖的.cpp和.c文件
obj/%.o:%.cpp
@if test ! -d "obj"; then\
    mkdir-p obj;\
fi;
$(CXX)-c -o $@ $(INCLUDEDIR) $<

obj/%.o:%.c
    @iftest ! -d "obj"; then\
            mkdir-p obj;\
    fi;
    $(CC)-c -o $@ $(INCLUDEDIR) $<

#生成.d文件,存放.cpp文件的所有依赖规则
dep/%.d: %.cpp
    @iftest ! -d "dep"; then\
            mkdir-p dep;\
    fi;\
    set-e; rm -f $@;
    $(CXX)$(DEPFLAGS) $(INCLUDEDIR) $< >$(DEPFILE); \
    sed's/$*\.o[ :]*/obj\/$*\.o dep\/$*\.d : /g' < $@.$$$$ > $@;\
    rm-f $@.$$$$

#生成.d文件,存放.c文件的所有依赖规则
dep/%.d: %.c
    @iftest ! -d "dep"; then\
            mkdir-p dep;\
    fi;\
    set-e; rm -f $@;
    $(CC)$(DEPFLAGS) $(INCLUDEDIR) $< > $(DEPFILE); \
    sed's/$*\.o[ :]*/obj\/$*\.o dep\/$*\.d : /g' < $@.$$$$ > $@; \
    rm-f $@.$$$$

include $(DEPS)

#检测是否有文件被修改，只要有就全部编译
all: $(SRCS) $(TARGETS)

#清除编译文件
.PHONY:clean
clean:
    -rm-f $(TARGET)
    -rm-f obj/*.o
    -rm-f dep/*.d
    -rm-f core
```

Makefile (lib)

```
#需要生成静态库的Makefile

#程序根目录
top_srcdir         =../../..

#生成静态库后所存放的位置
libdir = $(top_srcdir)/lib
#静态库文件名
LIBNAME          =libfa_common.a
#路径+静态库文件名
TARGET           =$(libdir)/$(LIBNAME)

CPP_FILES = $(shell ls *.cpp)
C_FILES = $(-shell ls *.c)
SRCS = $(CPP_FILES) $(C_FILES)
BASE = $(basename $(SRCS))
OBJS = $(addsuffix .o, $(addprefixobj/,$(BASE)))
DEPS = $(addsuffix .d, $(addprefixdep/,$(BASE)))

#包含公共Make规则
include$(top_srcdir)/makeinclude/Make.rules

#设置头文件及库文件的位置
INCLUDEDIR := $(INCLUDEDIR)

$(TARGET): $(OBJS)
    -rm-f $@
    ar cr $(TARGET) $(OBJS)
```

Makefile（app）

```
#需要生成可执行程序的Makefile

#程序根目录
top_srcdir         =../../..

#目标程序名
TARGET = test

CPP_FILES = $(shell ls *.cpp)
C_FILES = $(-shell ls *.c)
SRCS = $(CPP_FILES) $(C_FILES)
BASE = $(basename $(SRCS))
OBJS = $(addsuffix .o, $(addprefixobj/,$(BASE)))
DEPS = $(addsuffix .d, $(addprefixdep/,$(BASE)))

#包含公共Make规则
include $(top_srcdir)/makeinclude/Make.rules

#额外需要包含的头文件的目录位置
INCLUDEDIR := $(INCLUDEDIR)\
              -I$(top_srcdir)/src/common/inc\

#所有要包含的静态库的名称
LIBS := -lfa_common

#设置目标程序依赖的.o文件
$(TARGET):$(OBJS) $(LIBS)
    -rm-f $@
    $(CXX)-o $(TARGET) $(INCLUDEDIR) $(LDFLAGS) $(OBJS) $(LIBS)
```

作者：fatedier (http://blog.fatedier.com/)
本文出处：http://blog.fatedier.com/2014/09/24/learn-to-write-makefile-02/ (http://blog.fatedier.com/2014/09/24/learn-to-write-makefile-02/)
文章版权归本人所有，欢迎转载，但未经作者同意必须保留此段声明，且在文章页面明显位置给出原文链接，否则保留追究法律责任的权利。

c/cpp (/tags/c/cpp/) Linux (/tags/linux/)

相关文章

- 如何修改进程的名称 (/2015/08/24/how-to-modify-process-name/) (2015年08月24日)
- 在C++中利用反射和简单工厂模式实现业务模块解耦 (/2015/03/04/decoupling-by-using-reflect-and-simple-factory-pattern-in-cpp/) (2015年03月04日)
- epoll使用说明 (/2015/01/25/introduction-of-using-epoll/) (2015年01月25日)
- 如何处理僵尸进程 (/2014/12/16/how-to-deal-with-zombie-process/) (2014年12月16日)
- 能否被8整除 (/2014/11/13/can-be-divisible-by-eight/) (2014年11月13日)

- C/C++获取精确到微秒级的系统时间 (/2014/09/30/get-systime-accurate-to-microseconds-in-c-or-cpp/) (2014年09月30日)
- size() == 0和empty()的比较 (/2014/09/26/function-size-equal-zero-compare-with-empty/) (2014年09月26日)
- 从简单实例开始，学会写Makefile（一）(/2014/09/08/learn-to-write-makefile-01/) (2014年09月08日)

← Prev (/2014/09/26/function-size-equal-zero-compare-with-empty/) All Posts (/post/) Next → (/2014/09/08/learn-to-write-makefile-01/)

© 2017 fatedier blog

Powered by Hugo (https://gohugo.io)