2014年09月08日

从简单实例开始，学会写Makefile（一）

作为一个刚刚从大学毕业的新人，进公司不久就遇到了一个不大不小的门槛——看不懂Makefile！而Makefile所干的事却关系到程序的编译和链接，一个好的Makefile文件可以极大地提升编译项目文件的效率，免去手动编译的烦恼。

不会写Makefile虽然还不至于影响到项目进度，从别的地方拷贝一份过来稍加修改就可以用了，但是，对于咱们“程序猿”来说这实在是一件让人感觉很不爽的事。于是，百度，谷歌（PS：吐槽一下，不XX的话Google已经完全不能用了，Bing的效果都要比百度好一些），各种看资料，看大牛的博客，或许是本人比较笨，也或许是网上的资料不太适合咱们这种新人，缺乏生动的实例讲解，所以决定自己动手研究一下，并把过程分享给大家，希望新人们看完这篇文章后就能够自己动手，为自己的项目编写合适的Makefile啦。

为什么要写Makefile

首先要确定我们的目标，Makefile是用来干嘛的？

曾经很长时间我都是在从事Windows环境下的开发，所以根本不知道Makefile是个什么东西。因为早已经习惯了使用VS、Eclipse等等优秀的IDE做开发，只要点一个按钮，程序就可以运行啦。但是进入公司以后，从事的是Unix环境下的开发工作，没有了IDE，要怎么才能让我写的代码编译后运行呢？

在这里，Makefile的作用就体现出来了，简单的四个字—— “自动编译”。一旦整个项目的Makefile都写好了以后，只需要一个简单的make命令，就可以实现自动编译了。当然，准确的说，是make这个命令工具帮助我们实现了我们想要做的事，而Makefile就相当于是一个规则文件，make程序会按照Makefile所指定的规则，去判断哪些文件需要先编译，哪些文件需要后编译，哪些文件需要重新编译。

俗话说，懒人创造了整个世界，程序员就是在不断偷懒的过程中获得进步，使用Makefile最根本的目的就是简化我们的工作。

下面我们就从头开始，一步一步的去学习如何写好一个Makefile文件吧！

从单个文件开始

1、单个文件的编译

为了便于大家学习，这篇文章是以常见的Linux平台为基础的，系统为Centos6.5，使用GNU make工具进行编译，项目文件为C++格式。这里假定看到这篇文章的都是已经对C++程序的编译等基础知识和相关命令有了一定的了解的，鉴于篇幅限制，如果还有不清楚的就请自行查阅相关资料啦。

假设我们在src目录下有一个test.cpp文件，我们是如何编译它的呢？

```
g++ -o test test.cpp
```

在shell界面执行这句命令，当前目录下会生成一个名为test的可执行程序，使用./test就可以执行该程序，看到输出结果。

现在我们尝试使用编写Makefile的方式来实现这一编译过程。 首先在当前目录下新建文件并命名为“Makefile”，这样编译的时候直接使用gmake命令即可，默认使用“Makefile”文件进行编译，也可以是其他名字，那样的话需要使用“gmake -f 文件名”的格式来指定Makefile文件。

Makefile文件内容如下：

```
test:test.cpp
    g++-o test test.cpp
```

在shell界面下执行gmake命令，敲下回车，OK。

可以发现，g++ -o test test.cpp 这条命令已经被自动执行了，生成了名为test的程序。

2、Makefile的描述规则

至此，我们已经完成了一个最简单的Makefile文件，向我们的最终目标迈出了一大步！

有的人会问，传说中的自动化编译呢？难道每一个文件都要自己去写文件名和命令？

不用急，我们先来分析一下这个Makefile文件。

```
TARGET... :PREREQUISITES...
    COMMAND
    ...
    ...
```

这是最简单的Makefile文件的描述规则，可以说，这也是Makefile中最精华的部分，其他部分都是围绕着这个最基本的描述规则的。先来解释一下：

- TARGET：规则生成的目标文件，通常是需要生成的程序名（例如前面出现的程序名test）或者过程文件（类似.o文件）。
- PREREQUISITES：规则的依赖项，比如前面的Makefile文件中我们生成test程序所依赖的就是test.cpp。

- COMMAND：规则所需执行的命令行，通常是编译命令。这里需要注意的是每一行命令都需要以[TAB]字符开头。

再来看我们之前写过的Makefile文件，这个规则，用通俗的自然语言翻译过来就是：

- 如果目标test文件不存在，根据规则创建它。
- 目标test文件存在，并且test文件的依赖项中存在任何一个比目标文件更新（比如修改了一个函数，文件被更新了），根据规则重新生成它。
- 目标test文件存在，并且它比所有的依赖项都更新，那么什么都不做。

当我们第一次执行gmake命令时，test文件还不存在，所以就会执行g++-o test test.cpp这条命令创建test文件。

而当我们再一次执行gmake时，会提示文件已经是最新的，什么都不做。

这时候，如果修改了test.cpp命令，再次执行gmake命令。

由于依赖项比目标文件更新，g++ -o test test.cpp这条命令就又会被再一次执行。

现在，我们已经学会如何写一个简单的Makefile文件了，每次修改过源文件以后，只要执行gmake命令就可以得到我们想要生成的程序，而不需要一遍遍地重复敲g++ -o test test.cpp这个命令。

多个文件的编译

1、使用命令行编译多个文件

一个项目不可能只有一个文件，学会了单个文件的编译，自然而然就要考虑如何去编译多个文件呢？

同样，假设当前目录下有如下7个文件，test.cpp、w1.h、w1.cpp、w2.h、w2.cpp、w3.h、w3.cpp。其中test.cpp包含main函数，并且引用了w1.h、w2.h以及w3.h。我们需要生成的程序名为test。

在shell界面下，为了正确编译我们的项目，我们需要敲下如下的命令：

```
g++ -c -o w1.ow1.cpp
g++ -c -o w2.o w2.cpp
g++ -c -o w3.o w3.cpp
```

这时当前目录下会生成w1.o、w2.o、w3.o三个.o文件。这里需要注意的是，“-c”命令是只编译，不链接，通常生成.o文件的时候使用。

```
g++ -o testtest.cpp w1.o w2.o w3.o
```

执行完这条命令后，编译成功，得到了我们想要的test文件。

2、使用Makefile编译多个文件

既然单个文件的Makefile会写了，相信多个文件举一反三也不是问题了。

Makefile具体内容如下：

```
test:test.cppw1.o w2.o w3.o
    g++ -o test test.cpp w1.o w2.o w3.o
w1.o:w1.cpp
    g++ -c -o w1.o w1.cpp
w2.o:w2.cpp
    g++ -c -o w2.o w2.cpp
w3.o:w3.cpp
    g++ -c -o w3.o w3.cpp
```

这里需要注意的是，我们写的第一个规则的目标，将会成为“终极目标”，也就是我们最终希望生成的程序，这里是“test”文件。根据我们的“终极目标”，make会进行自动推导，例如“终极目标”依赖于的.o文件，make就会寻找生成这些.o文件的规则，然后执行相应的命令去生成这些文件，这样一层一层递归地进行下去，直到最终生成了“终极目标”。

```
[wcl@localhost /home/wcl/fate/mktest/src]gmake
g++ -c -o w1.o w1.cpp
g++ -c -o w2.o w2.cpp
g++ -c -o w3.o w3.cpp
g++ -o test test.cpp w1.o w2.o w3.o
```

如上图所示，虽然生成test文件的规则写在最前面，但是由于依赖于w1.o、w2.o、w3.o，make会先执行生成w1.o、w2.o、w3.o所需的命令，然后才会执行g++ -o test test.cpp w1.o w2.o w3.o 来生成test文件。

3、使用伪目标来清除过程文件

我们现在已经可以自动编译多个文件的项目了，但是当我们需要全部重新编译的时候，难道还要手动地一个一个去删除那些生成的.o文件吗？

既然已经使用了Makefile，我们的目标就是实现自动化编译，那么这些清除过程文件这点小事必须得能够用一个命令搞定啦。

我们只需要在Makefile文件的最后加上如下几行：

```
clean:
    -rm-f test *.o
```

OK，轻松搞定，然后在shell界面下执行gmakeclean。仔细看看，是不是所有的.o文件和最后生成的程序文件已经被清除了？

这里说明一下，rm是Linux下删除文件或目录的命令，前面加上“-”符号意思是忽略执行rm产生的错误。“-f”参数是指强制删除，忽略不存在的文件。

这样的目标叫做“伪目标”，通过“gmake 目标名”来指定这个目标，然后执行这个目标规则下的命令。

使用变量简化Makefile

作为一个“懒惰”的程序员，现在问题又来了。如果按照上面的写法，在文件数量和名称不变的情况的下确实是没有问题，但是如果我们新增一个文件的话，岂不是又要去修改Makefile了，一个项目多的可能有成百上千的文件，这样管理起来得有多麻烦呀！

还记得我们在Linux下如果要查看当前目录下所有的cpp文件的时候，使用的命令吗？

```
ls *.cpp
```

通过这个命令，我们就可以将所有的cpp文件名称显示在界面上。而在Makefile中我们同样可以使用类似的规则来做简化，进一步减少后续开发过程中对Makefile文件的修改。

修改后的Makefile文件如下：

```
TARGET = test

CPP_FILES = $(shell ls *.cpp)
BASE = $(basename $(CPP_FILES))
OBJS = $(addsuffix .o, $(addprefix obj/,$(BASE)))

$(TARGET):$(OBJS)
    -rm -f $@
    g++ -o $(TARGET)$(OBJS)

obj/%.o:%.cpp
    @if test ! -d"obj"; then\
    mkdir -pobj;\
    fi;
    g++ -c -o $@ $<

clean:
    -rm -f test
    -rm -f obj/*.o
```

是不是瞬间有种摸不着头脑的感觉？别急，这是因为我们用到了一些新的语法和命令，其实，本质上和我们之前所写的Makefile文件是一个意思，下面我们就逐条来进行分析。

TARGET = test

定义一个变量，保存目标文件名，这里我们需要生成的程序名就叫test。

CPP_FILES = $(shell ls *.cpp)

定义一个变量，内容为所有的以.cpp为后缀的文件的文件名，以空格隔开。

这里&(shell 命令)的格式，说明这里将会用shell命令执行后输出的内容进行替换，就和在命令行下输入ls *.cpp得到的结果一样。

BASE = $(basename $(CPP_FILES))

定义一个变量，内容为所有的以.cpp为后缀的文件的文件名去除掉后缀部分。

$(CPP_FILES)是引用CPP_FIFES这个变量的内容，相信学过如何写shell命令的同学肯定不会陌生。basename 是一个函数，其作用就是去除掉文件名的后缀部分，例如“test.cpp”，经过这一步后就变成了“test”。

OBJS = $(addsuffix .o, $(addprefix obj/,$(BASE)))

定义一个变量，内容为所有的以.cpp为后缀的文件去除调后缀部分后加上“.o”。

和basename一样，addsuffix和addprefix同样也是调用函数。addprefix的作用是给每个文件名加上前缀，这里是加上“obj/”，而addsuffix的作用是给每个文件名加上后缀，这里是在文件名后加上“.o”。例如“test”，经过变换后变成了“obj/test.o”。

为什么要在文件名前加上“obj/”？

这个不是必须的，只是我自己觉得将所有的.o文件放在一个obj目录下统一管理会让目录结构显得更加清晰，包括以后的.d文件会统一放在dep目录下一样。当然，你也可以选择不这样做，而是全部放在当前目录下。

$(TARGET):$(OBJS)

```
$(TARGET):$(OBJS)
    -rm -f $@
    g++ -o $(TARGET) $(OBJS)
```

这个描述规则和我们之前写过的很像，只不过，使用了变量进行替换。其中需要注意的是$@这个奇怪的符号，它的含义是这个规则的目标文件的名称，在这里就相当于是$(TARGET)。

把这里的变量替换成我们之前项目中的实际值，就相当于：

```
test:test.ow1.o w2.o w3.o
    -rm-f test
    g++ -o test test.o w1.o w2.o w3.o
```

如果按照这种写法，当我们新增了一个w4.cpp文件的时候，就需要对Makefile进行修改，而如果我们使用了变量进行替换，那么我们就什么都不用做，直接再执行一遍gmake命令即可。

obj/%.o:%.cpp

```
obj/%.o:%.cpp
    @if test ! -d"obj"; then\
        mkdir -p obj;\
    fi;
    g++ -c -o $@ $<
```

这是依次生成所有cpp文件所对应的.o文件的规则。

%.o和%.c表示以.o和.c结尾的文件名。因为我们准备把所有的.o文件放在obj目录下，所以这里在“%.o”前面加上前缀“obj/”。

下面命令行的前三行，具体的作用是检查当前目录下是否有名为“obj”的目录，如果没有，则使用mkdir命令创建这个目录。如果不了解的同学不如先去看一下shell编程的相关知识吧。

最后一句中的$@前面已经解释过了，是代表规则的目标文件名称，而$<与之对应的，则是代表规则的依赖项中第一个依赖文件的名称。

例如obj/test.o:test.cpp

那么$@的值为“test.o”，$<的值为“test.cpp”

clean:

```
clean:
    -rm -f test
    -rm -f obj/*.o
```

这个就没什么好说的啦，这里只是修改了一下.o文件的路径。

到这里，相信你对如何使用Makefile来编译一个小的项目已经颇有些眉目了吧。使用这个Makefile文件，不管你往这个目录下加多少文件，轻轻松松一个gmake命令搞定，不需要再因为加了一个新的文件而去修改Makefile了。

但是，你难道没有觉得仍然存在着很多问题吗？

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？

大家可以先尝试着自己解决以上的问题，在之后的篇幅中我们会就以上几点继续通过举例的方式来加以解决。

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（二） (/2014/09/24/learn-to-write-makefile-02/) (2014年09月24日)

← Prev (/2014/09/24/learn-to-write-makefile-02/) All Posts (/post/) Next →