C++之makefile写法

原创 ZONG_XP 2018-07-20 16:18:49 17508 收藏 49 版权

参考：https://www.cnblogs.com/owlman/p/5514724.html

什么是makefile

Makefile 文件描述了整个工程的编译、连接等规则。其中包括：工程中的哪些源文件需要编译以及如何编译、需要创建那些库文件以及如何创建这些库文件、如何最后产生我们想要的可执行文件。尽管看起来可能是很复杂的事情，但是为工程编写Makefile 的好处是能够使用一行命令来完成“自动化编译”，一旦提供一个（通常对于一个工程来说会是多个）正确的 Makefile。编译整个工程你所要做的唯一的一件事就是在shell 提示符下输入make命令。整个工程完全自动编译，极大提高了效率。

编译与链接

一般来说，无论是C、C++、还是pas，首先要把源文件编译成中间代码文件，在Windows下也就是.obj文件，UNIX下是.o文件，即Object File，这个动作叫做编译（compile）。然后再把大量的Object File合成执行文件，这个动作叫作链接（link）。

编译时，编译器需要的是语法的正确，函数与变量的声明的正确。对于后者，通常是你需要告诉编译器头文件的所在位置（头文件中应该只是声明，而定义应该放在C/C++文件中），只要所有的语法正确，编译器就可以编译出中间目标文件。一般来说，每个源文件都应该对应于一个中间目标文件（O文件或是OBJ文件）。

链接时，主要是链接函数和全局变量，所以，我们可以使用这些中间目标文件（O文件或是OBJ文件）来链接我们的应用程序。链接器并不管函数所在的源文件，只管函数的中间目标文件（Object File），在大多数时候，由于源文件太多，编译生成的中间目标文件太多，而在链接时需要明显地指出中间目标文件名，这对于编译很不方便，所以，我们要给中间目标文件打个包，在Windows下这种包叫“库文件”（LibraryFile)，也就是 .lib文件，在UNIX下，是Archive File，也就是.a文件。

总结一下，源文件首先会生成中间目标文件，再由中间目标文件生成执行文件。在编译时，编译器只检测程序语法，和函数、变量是否被声明。如果函数未被声明，编译器会给出一个警告，但可以生成Object File。而在链接程序时，链接器会在所有的Object File中找寻函数的实现，如果找不到，那到就会报链接错误码（Linker Error），在VC下，这种错误一般是：Link 2001错误，意思说是说，链接器未能找到函数的实现。你需要指定函数的ObjectFile.

makefile基本格式

makefile基本格式如下：

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

其中：

- target - 目标文件, 可以是 Object File, 也可以是可执行文件

- prerequisites - 生成 target 所需要的文件或者目标
- command - make需要执行的命令 (任意的shell命令)，如果其不与“target:prerequisites”在一行，那么，必须以[Tab]开头，如果和prerequisites在一行，那么可以用分号做为分隔

make会比较targets文件和prerequisites文件的修改日期，如果prerequisites文件的日期要比targets文件的日期要新，或者target不存在的话，那么，make就会执行后续定义的命令。

make工作流程

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），并把这个文件作为最终的目标文件。
3. 如果目标文件不存在，或是目标文件所依赖的后面的 .o 文件的文件修改时间要比目标文件这个文件新，那么，他就会执行后面所定义的命令来生成edit这个文件。
4. 如果目标文件所依赖的.o文件也存在，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 文件声明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。在找寻的过程中，如果出现错误，比如最后被依赖的文件找不到，那么make就会直接退出，并报错，而对于所定义的命令的错误，或是编译不成功，make根本不理。make只管文件的依赖性，即，如果在我找了依赖关系之后，冒号后面的文件还是不在，那么对不起，我就不工作啦。

简单举例

我们用一个例子来做个说明。在这个例子中，我们有一个主程序代码(main.c)、三份函数代码(getop.c、stack.c、getch.c)以及一个头文件(calc.h)。通常情况下，我们需要这样编译它：

```
gcc -o calc main.c getch.c getop.c stack.c
```

如果没有makefile，在开发+调试程序的过程中，我们就需要不断地重复输入上面这条编译命令，要不就是通过终端的历史功能不停地按上下键来寻找最近执行过的命令。这样做两个缺陷：

1. 一旦终端历史记录被丢失，我们就不得不从头开始；
2. 任何时候只要我们修改了其中一个文件，上述编译命令就会重新编译所有的文件，当文件足够多时这样的编译会非常耗时。

那么Makefile又能做什么呢？我们先来看一个最简单的makefile文件：

```
calc: main.c getch.c getop.c stack.c
    gcc -o calc main.c getch.c getop.c stack.c
```

现在你看到的就是一个最基本的Makefile语句，它主要分成了三个部分，第一行冒号之前的calc，我们称之为目标（target），被认为是这条语句所要处理的对象，具体到这里就是我们所要编译的这个程序calc。冒号后面的部分（main.c getch.c getop.c stack.c），我们称之为依赖关系表，也就是编译calc所

需要的文件，这些文件只要有一个发生了变化，就会触发该语句的第三部分，我们称其为命令部分，相信你也看得出这就是一条编译命令。现在我们只要将上面这两行语句写入一个名为Makefile或者makefile的文件，然后在终端中输入make命令，就会看到它按照我们的设定去编译程序了。

接下来，让我们来解决一下效率方面的问题，先初步修改一下上面的代码：

```
cc = gcc
prom = calc
source = main.c getch.c getop.c stack.c

$(prom): $(source)
    $(cc) -o $(prom) $(source)
```

如你所见，我们在上述代码中定义了三个常量cc、prom以及source（请注意，很多教程将这里的cc、prom和source称之为变量，个人认为这是不妥当的，因为它们在整个文件的执行过程中并不是可更改的，作用也仅仅是字符串替换而已，非常类似于C语言中的宏定义。或者说，事实上它就是一个宏）。它们分别告诉了make我们要使用的编译器、要编译的目标以及源文件。这样一来，今后我们要修改这三者中的任何一项，只需要修改常量的定义即可，而不用再去管后面的代码部分了。

但我们现在依然还是没能解决当我们只修改一个文件时就要全部重新编译的问题。而且如果我们修改的是calc.h文件，make就无法察觉到变化了（所以有必要为头文件专门设置一个常量，并将其加入到依赖关系表中）。下面，我们来想一想如何解决这个问题。考虑到在标准的编译过程中，源文件往往是先被编译成目标文件，然后再由目标文件连接成可执行文件的。我们可以利用这一点来调整一下这些文件之间的依赖关系：

```
cc = gcc
prom = calc
deps = calc.h
obj = main.o getch.o getop.o stack.o

$(prom): $(obj)
    $(cc) -o $(prom) $(obj)

main.o: main.c $(deps)
    $(cc) -c main.c

getch.o: getch.c $(deps)
    $(cc) -c getch.c

getop.o: getop.c $(deps)
    $(cc) -c getop.c

stack.o: stack.c $(deps)
    $(cc) -c stack.c
```

这样一来，上面的问题显然是解决了，但同时我们又让代码变得非常啰嗦，啰嗦往往伴随着低效率，是不祥之兆。经过再度观察，我们发现所有.c都会被编译成相同名称的.o文件。我们可以根据该特点再对其做进一步的简化：

```
cc = gcc
prom = calc
```

```
deps = calc.h
obj = main.o getch.o getop.o stack.o

$(prom): $(obj)
    $(cc) -o $(prom) $(obj)

%.o: %.c $(deps)
    $(cc) -c $< -o $@
```

在这里，我们用到了几个特殊的宏。首先是%.o:%.c，这是一个模式规则，表示所有的.o目标都依赖于与它同名的.c文件（当然还有deps中列出的头文件）。再来就是命令部分的$<和$@，其中$<代表的是依赖关系表中的第一项（如果我们想引用的是整个关系表，那么就应该使用$^），具体到我们这里就是%.c。而$@代表的是当前语句的目标，即%.o。这样一来，make命令就会自动将所有的.c源文件编译成同名的.o文件。不用我们一项一项去指定了。整个代码自然简洁了许多。

Makefile 中很多时候通过自动变量来简化书写, 各个自动变量的含义如下:

自动变量	含义
$@	目标集合
$%	当目标是函数库文件时, 表示其中的目标文件名
$<	第一个依赖目标. 如果依赖目标是多个, 逐个表示依赖目标
$?	比目标新的依赖目标的集合
$^	所有依赖目标的集合, 会去除重复的依赖目标
$+	所有依赖目标的集合, 不会去除重复的依赖目标
$*	这个是GNU make特有的, 其它的make不一定支持

到目前为止，我们已经有了一个不错的makefile，至少用来维护这个小型工程是没有什么问题了。当然，如果要进一步增加上面这个项目的可扩展性，我们就会需要用到一些Makefile中的伪目标和函数规则了。例如，如果我们想增加自动清理编译结果的功能就可以为其定义一个带伪目标的规则；

```
cc = gcc
prom = calc
deps = calc.h
obj = main.o getch.o getop.o stack.o

$(prom): $(obj)
    $(cc) -o $(prom) $(obj)

%.o: %.c $(deps)
    $(cc) -c $< -o $@

clean:
    rm -rf $(obj) $(prom)
```

有了上面最后两行代码，当我们在终端中执行make clean命令时，它就会去删除该工程生成的所有编译文件。

另外，如果我们需要往工程中添加一个.c或.h，可能同时就要再手动为obj常量再添加第一个.o文件，如果这列表很长，代码会非常难看，为此，我们需要用到Makefile中的函数，这里我们演示两个：

```
cc = gcc
prom = calc
deps = $(shell find ./ -name "*.h")
src = $(shell find ./ -name "*.c")
obj = $(src:%.c=%.o)

$(prom): $(obj)
    $(cc) -o $(prom) $(obj)

%.o: %.c $(deps)
    $(cc) -c $< -o $@

clean:
    rm -rf $(obj) $(prom)
```

其中，shell函数主要用于执行shell命令，具体到这里就是找出当前目录下所有的.c和.h文件。而$(src:%.c=%.o)则是一个字符替换函数，它会将src所有的.c字串替换成.o，实际上就等于列出了所有.c文件要编译的结果。有了这两个设定，无论我们今后在该工程加入多少.c和.h文件，Makefile都能自动将其纳入到工程中来。