跟我一起写Makefile:MakeFile介绍

目录

[跟我一起写Makefile:MakeFile介绍 1](#_Toc487917700)

[makefile 介绍 2](#_Toc487917701)

[makefile的规则 2](#_Toc487917702)

[一个示例 3](#_Toc487917703)

[make是如何工作的 4](#_Toc487917704)

[makefile中使用变量 5](#_Toc487917705)

[让make自动推导 6](#_Toc487917706)

[另类风格的makefile 7](#_Toc487917707)

[清空目标文件的规则 7](#_Toc487917708)

[Makefile里有什么？ 8](#_Toc487917709)

[Makefile的文件名 8](#_Toc487917710)

[引用其它的Makefile 9](#_Toc487917711)

[环境变量 MAKEFILES 10](#_Toc487917712)

[make的工作方式 10](#_Toc487917713)

|  |
| --- |
|  |

makefile 介绍

make命令执行时，需要一个 makefile 文件，以告诉make命令如何去编译和链接程序。

首先，我们用一个示例来说明makefile的书写规则。以便给大家一个感性认识。这个示例来源于gnu的make使用手册，在这个示例中，我们的工程有8个c文件，和3个头文件，我们要写一个makefile来告诉make命令如何编译和链接这几个文件。我们的规则是：

1）如果这个工程没有编译过，那么我们的所有c文件都要编译并被链接。

2）如果这个工程的某几个c文件被修改，那么我们只编译被修改的c文件，并链接目标程序。

3）如果这个工程的头文件被改变了，那么我们需要编译引用了这几个头文件的c文件，并链接目标程序。

只要我们的makefile写得够好，所有的这一切，我们只用一个make命令就可以完成，make命令会自动智能地根据当前的文件修改的情况来确定哪些文件需要重编译，从而自己编译所需要的文件和链接目标程序。

makefile的规则

在讲述这个makefile之前，还是让我们先来粗略地看一看makefile的规则。

target ... : prerequisites ...

command

...

...

target可以是一个object file(目标文件)，也可以是一个执行文件，还可以是一个标签（label）。对于标签这种特性，在后续的“伪目标”章节中会有叙述。

prerequisites就是，要生成那个target所需要的文件或是目标。

command也就是make需要执行的命令。（任意的shell命令）

这是一个文件的依赖关系，也就是说，target这一个或多个的目标文件依赖于prerequisites中的文件，其生成规则定义在 command中。说白一点就是说，prerequisites中如果有一个以上的文件比target文件要新的话，command所定义的命令就会被执行。这就是makefile的规则。也就是makefile中最核心的内容。

说到底，makefile的东西就是这样一点，好像我的这篇文档也该结束了。呵呵。还不尽然，这是makefile的主线和核心，但要写好一个makefile还不够，我会以后面一点一点地结合我的工作经验给你慢慢道来。内容还多着呢。：）

一个示例

正如前面所说的，如果一个工程有3个头文件，和8个c文件，我们为了完成前面所述的那三个规则，我们的makefile应该是下面这个样子的。

edit : main.o kbd.o command.o display.o \

insert.o search.o files.o utils.o /\*注释:如果后面这些.o文件比edit可执行文件新,那么才会去执行下面这句命令\*/

cc -o edit main.o kbd.o command.o display.o \

insert.o search.o files.o utils.o

main.o : main.c defs.h

cc -c main.c

kbd.o : kbd.c defs.h command.h

cc -c kbd.c

command.o : command.c defs.h command.h

cc -c command.c

display.o : display.c defs.h buffer.h

cc -c display.c

insert.o : insert.c defs.h buffer.h

cc -c insert.c

search.o : search.c defs.h buffer.h

cc -c search.c

files.o : files.c defs.h buffer.h command.h

cc -c files.c

utils.o : utils.c defs.h

cc -c utils.c

clean :

rm edit main.o kbd.o command.o display.o \

insert.o search.o files.o utils.o

反斜杠（\）是换行符的意思。这样比较便于makefile的易读。我们可以把这个内容保存在名字为“makefile”或“Makefile” 的文件中，然后在该目录下直接输入命令“make”就可以生成执行文件edit。如果要删除执行文件和所有的中间目标文件，那么，只要简单地执行一下 “make clean”就可以了。

在这个makefile中，目标文件（target）包含：执行文件edit和中间目标文件（\*.o），依赖文件（prerequisites）就是冒号后面的那些 .c 文件和 .h文件。每一个 .o 文件都有一组依赖文件，而这些 .o 文件又是执行文件 edit 的依赖文件。依赖关系的实质上就是说明了目标文件是由哪些文件生成的，换言之，目标文件是哪些文件更新的。

在定义好依赖关系后，后续的那一行定义了如何生成目标文件的操作系统命令，一定要以一个tab键作为开头。记住，make并不管命令是怎么工作的，他只管执行所定义的命令。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。

这里要说明一点的是，clean不是一个文件，它只不过是一个动作名字，有点像c语言中的lable一样，其冒号后什么也没有，那么，make就不会自动去找它的依赖性，也就不会自动执行其后所定义的命令。要执行其后的命令（不仅用于clean，其他lable同样适用），就要在make命令后明显得指出这个lable的名字。这样的方法非常有用，我们可以在一个makefile中定义不用的编译或是和编译无关的命令，比如程序的打包，程序的备份，等等。

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“edit”这个文件，并把这个文件作为最终的目标文件。
3. 如果edit文件不存在，或是edit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。
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。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。

通过上述分析，我们知道，像clean这种，没有被第一个目标文件直接或间接关联，那么它后面所定义的命令将不会被自动执行，不过，我们可以显式要make执行。即命令——“make clean”，以此来清除所有的目标文件，以便重编译。

于是在我们编程中，如果这个工程已被编译过了，当我们修改了其中一个源文件，比如file.c，那么根据我们的依赖性，我们的目标file.o会被重编译（也就是在这个依性关系后面所定义的命令），于是file.o的文件也是最新的啦，于是file.o的文件修改时间要比edit要新，所以 edit也会被重新链接了（详见edit目标文件后定义的命令）。

而如果我们改变了“command.h”，那么，kdb.o、command.o和files.o都会被重编译，并且，edit会被重链接。

makefile中使用变量

在上面的例子中，先让我们看看edit的规则：

edit : main.o kbd.o command.o display.o \

insert.o search.o files.o utils.o

cc -o edit main.o kbd.o command.o display.o \

insert.o search.o files.o utils.o

我们可以看到[.o]文件的字符串被重复了两次，如果我们的工程需要加入一个新的[.o]文件，那么我们需要在两个地方加（应该是三个地方，还有一个地方在clean中）。当然，我们的makefile并不复杂，所以在两个地方加也不累，但如果makefile变得复杂，那么我们就有可能会忘掉一个需要加入的地方，而导致编译失败。所以，为了makefile的易维护，在makefile中我们可以使用变量。makefile的变量也就是一个字符串，理解成C语言中的宏可能会更好。

比如，我们声明任意一变量名，叫objects, OBJECTS, objs, OBJS, obj, 或OBJ，只要能够表示obj文件即可。我们在makefile起始处按如下定义此变量：

objects = main.o kbd.o command.o display.o \

insert.o search.o files.o utils.o

于是，我们就可以很方便地在我们的makefile中以“$(objects)”的方式来使用这个变量了，于是我们的改良版makefile变为如下：

objects = main.o kbd.o command.o display.o \

insert.o search.o files.o utils.o

edit : $(objects)

cc -o edit $(objects)

main.o : main.c defs.h

cc -c main.c

kbd.o : kbd.c defs.h command.h

cc -c kbd.c

command.o : command.c defs.h command.h

cc -c command.c

display.o : display.c defs.h buffer.h

cc -c display.c

insert.o : insert.c defs.h buffer.h

cc -c insert.c

search.o : search.c defs.h buffer.h

cc -c search.c

files.o : files.c defs.h buffer.h command.h

cc -c files.c

utils.o : utils.c defs.h

cc -c utils.c

clean :

rm edit $(objects)

如果有新的 .o 文件加入，我们只需简单地修改变量objects即可。

更多关于变量的话题，我会在后续详细介绍。

让make自动推导

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，于是我们就没必要去在每一个[.o]文件后都写上类似的命令，因为，我们的make会自动识别，并自己推导命令。

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whatever.o，那么 whatever.c，就会是whatever.o的依赖文件。并且 cc -c whatever.c 也会被推导出来，于是，我们的makefile 再也不用写得这么复杂。我们的新makefile又出炉了。

objects = main.o kbd.o command.o display.o \

insert.o search.o files.o utils.o

cc = gcc

edit : $(objects)

cc -o edit $(objects)

main.o : defs.h

kbd.o : defs.h command.h

command.o : defs.h command.h

display.o : defs.h buffer.h

insert.o : defs.h buffer.h

search.o : defs.h buffer.h

files.o : defs.h buffer.h command.h

utils.o : defs.h

.PHONY : clean

clean :

rm edit $(objects)

这种方法，也就是make的“隐晦规则”。上面文件内容中，“.PHONY”表示，clean是个伪目标文件。

关于更为详细的“隐晦规则”和“伪目标文件”，我会在后续给你一一道来。

另类风格的makefile

既然我们的make可以自动推导命令，那么我看到那堆[.o]和[.h]的依赖就有点不爽，那么多的重复的[.h]，能不能把其收拢起来，好吧，没有问题，这个对于make来说很容易，谁叫它提供了自动推导命令和文件的功能呢？来看看最新风格的makefile吧。

objects = main.o kbd.o command.o display.o \

insert.o search.o files.o utils.o

edit : $(objects)

cc -o edit $(objects)

$(objects) : defs.h

kbd.o command.o files.o : command.h

display.o insert.o search.o files.o : buffer.h

.PHONY : clean

clean :

rm edit $(objects)

这种风格，让我们的makefile变得很简单，但我们的文件依赖关系就显得有点凌乱了。鱼和熊掌不可兼得。还看你的喜好了。我是不喜欢这种风格的，一是文件的依赖关系看不清楚，二是如果文件一多，要加入几个新的.o文件，那就理不清楚了。

清空目标文件的规则

每个Makefile中都应该写一个清空目标文件（.o和执行文件）的规则，这不仅便于重编译，也很利于保持文件的清洁。这是一个“修养”（呵呵，还记得我的《编程修养》吗）。一般的风格都是：

clean:

rm edit $(objects)

更为稳健的做法是：

.PHONY : clean

clean :

-rm edit $(objects)

前面说过，.PHONY意思表示clean是一个“伪目标”。而在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。当然，clean的规则不要放在文件的开头，不然，这就会变成make的默认目标，相信谁也不愿意这样。不成文的规矩是——“clean从来都是放在文件的最后”。

上面就是一个makefile的概貌，也是makefile的基础，下面还有很多makefile的相关细节，准备好了吗？准备好了就来。

Makefile里有什么？

Makefile里主要包含了五个东西：显式规则、隐晦规则、变量定义、文件指示和注释。

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，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. 文件指示。其包括了三个部分，一个是在一个Makefile中引用另一个Makefile，就像C语言中的include一样；另一个是指根据某些情况指定Makefile中的有效部分，就像C语言中的预编译#if一样；还有就是定义一个多行的命令。有关这一部分的内容，我会在后续的部分中讲述。
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：“\#”。

最后，还值得一提的是，在Makefile中的命令，必须要以[Tab]键开始。

Makefile的文件名

默认的情况下，make命令会在当前目录下按顺序找寻文件名为“GNUmakefile”、“makefile”、“Makefile”的文件，找到了解释这个文件。在这三个文件名中，最好使用“Makefile”这个文件名，因为，这个文件名第一个字符为大写，这样有一种显目的感觉。最好不要用 “GNUmakefile”，这个文件是GNU的make识别的。有另外一些make只对全小写的“makefile”文件名敏感，但是基本上来说，大多数的make都支持“makefile”和“Makefile”这两种默认文件名。

当然，你可以使用别的文件名来书写Makefile，比如：“Make.Linux”，“Make.Solaris”，“Make.AIX”等，如果要指定特定的Makefile，你可以使用make的“-f”和“--file”参数，如：make -f Make.Linux或make --file Make.AIX。

引用其它的Makefile

在Makefile使用include关键字可以把别的Makefile包含进来，这很像C语言的#include，被包含的文件会原模原样的放在当前文件的包含位置。include的语法是：

include <filename>;

filename可以是当前操作系统Shell的文件模式（可以包含路径和通配符）

在include前面可以有一些空字符，但是绝不能是[Tab]键开始。include和<filename>;可以用一个或多个空格隔开。举个例子，你有这样几个Makefile：a.mk、b.mk、c.mk，还有一个文件叫foo.make，以及一个变量$(bar)，其包含了 e.mk和f.mk，那么，下面的语句：

include foo.make \*.mk $(bar)

等价于：

include foo.make a.mk b.mk c.mk e.mk f.mk

make命令开始时，会找寻include所指出的其它Makefile，并把其内容安置在当前的位置。就好像C/C++的#include指令一样。如果文件都没有指定绝对路径或是相对路径的话，make会在当前目录下首先寻找，如果当前目录下没有找到，那么，make还会在下面的几个目录下找：

1. 如果make执行时，有“-I”或“--include-dir”参数，那么make就会在这个参数所指定的目录下去寻找。
2. 如果目录<prefix>;/include（一般是：/usr/local/bin或/usr/include）存在的话，make也会去找。

如果有文件没有找到的话，make会生成一条警告信息，但不会马上出现致命错误。它会继续载入其它的文件，一旦完成makefile的读取， make会再重试这些没有找到，或是不能读取的文件，如果还是不行，make才会出现一条致命信息。如果你想让make不理那些无法读取的文件，而继续执行，你可以在include前加一个减号“-”。如：

-include <filename>;

其表示，无论include过程中出现什么错误，都不要报错继续执行。和其它版本make兼容的相关命令是sinclude，其作用和这一个是一样的。

环境变量 MAKEFILES

如果你的当前环境中定义了环境变量MAKEFILES，那么，make会把这个变量中的值做一个类似于include的动作。这个变量中的值是其它的Makefile，用空格分隔。只是，它和include不同的是，从这个环境变量中引入的Makefile的“目标”不会起作用，如果环境变量中定义的文件发现错误，make也会不理。

但是在这里我还是建议不要使用这个环境变量，因为只要这个变量一被定义，那么当你使用make时，所有的Makefile都会受到它的影响，这绝不是你想看到的。在这里提这个事，只是为了告诉大家，也许有时候你的Makefile出现了怪事，那么你可以看看当前环境中有没有定义这个变量。

make的工作方式

GNU的make工作时的执行步骤如下：（想来其它的make也是类似）

1. 读入所有的Makefile。
2. 读入被include的其它Makefile。
3. 初始化文件中的变量。
4. 推导隐晦规则，并分析所有规则。
5. 为所有的目标文件创建依赖关系链。
6. 根据依赖关系，决定哪些目标要重新生成。
7. 执行生成命令。

1-5步为第一个阶段，6-7为第二个阶段。第一个阶段中，如果定义的变量被使用了，那么，make会把其展开在使用的位置。但make并不会完全马上展开，make使用的是拖延战术，如果变量出现在依赖关系的规则中，那么仅当这条依赖被决定要使用了，变量才会在其内部展开。

当然，这个工作方式你不一定要清楚，但是知道这个方式你也会对make更为熟悉。有了这个基础，后续部分也就容易看懂了。