**Makefile分析学习**

**一、Makefile规则**

**1.1基础知识：**

Makefile 是和 make 命令一起配合使用的。make命令执行时，需要一个 Makefile 文件，以告诉make命令需要怎么样的去编译和链接程序。很多大型项目的编译都是通过 Makefile 来组织的, 如果没有 Makefile, 那很多项目中各种库和代码之间的依赖关系不知会多复杂。

Makefile基本格式如下:

target ... : prerequisites ...

command

...

...

* 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), Makefile中的命令必须以 [tab] 开头

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。

**1.2 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规则，我们的Makefile应该是下面的这个样子的。  
  
    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  
    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

我们可以把这个内容保存在文件为“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，其冒号后什么也没有，那么make就不会自动去找文件的依赖性，也就不会自动执行其后所定义的命令。要执行其后的命令，就要在make命令后明显得指出clean名字，make clean。

**1.3 make工作机制**

在默认的方式下，也就是我们只输入make命令。那么，  
    a.make会在当前目录下找名字叫“Makefile”或“makefile”的文件。  
    b.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。  
    c.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。  
    d.如果edit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）  
    e.当然，你的C文件和H文件是存在的make才会生成 .o 文件，然后再用.o 文件生命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。  
 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。

**1.4 变量**

我们声明一个变量，叫objects, OBJECTS, objs, OBJS, obj, 或是 OBJ，反正不管什么，只要能够表示obj文件就行了。我们在makefile一开始就这样定义：  
     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就变成下面这个样子：  
    objects = main.o kbd.o command.o display.o insert.o search.o files.o utils.o

edit : $(objects)  
          cc -o edit $(objects)

**1.5 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。  
    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 : 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。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。

1. **写Makefile**

**1.makefile内容**

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，如何生成一个或多的的目标文件。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粗糙地简略地书写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。  
 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，可以用反斜框进行转义，如：“\#”。  
 最后，还值得一提的是，在Makefile中的命令，必须要以[Tab]键开始。

**2.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。

**3.引用其他Makefile**

在Makefile使用include关键字可以把别的Makefile包含进来，，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的文件路径模式（可以保含路径和通配符）。在include前面可以有一些空字符，但是绝不能是[Tab]键开始，include和可以用一个或多个空格隔开。

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。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：

a.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。

b.如果目录/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>

## **4.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。

**5.命令执行**

当依赖目标新于目标时，也就是当规则的目标需要被更新时，make会一条一条的执行其后的命令。需要注意的是，如果你要让上一条命令的结果应用在下一条命令时，你应该使用分号分隔这两条命令。比如你的第一条命令是cd命令，你希望第二条命令得在cd之后的基础上运行，那么你就不能把这两条命令写在两行上，而应该把这两条命令写在一行上，用分号分隔。如：

   示例一：

       exec:

               cd /home/hchen

               pwd

   示例二：

       exec:

               cd /home/hchen; pwd

当我们执行“make exec”时，第一个例子中的cd没有作用，pwd会打印出当前的Makefile目录，而第二个例子中，cd就起作用了，pwd会打印出“/home/hchen”。

make一般是使用环境变量SHELL中所定义的系统Shell来执行命令，默认情况下使用UNIX的标准Shell——/bin/sh来执行命令。但在MS-DOS下有点特殊，因为MS-DOS下没有SHELL环境变量，当然你也可以指定。如果你指定了UNIX风格的目录形式，首先，make会在SHELL所指定的路径中找寻命令解释器，如果找不到，其会在当前盘符中的当前目录中寻找，如果再找不到，其会在PATH环境变量中所定义的所有路径中寻找。MS-DOS中，如果你定义的命令解释器没有找到，其会给你的命令解释器加上诸如“.exe”、“.com”、“.bat”、“.sh”等后缀。

**6.常用功能**

**a.@**

通常，make会把其要执行的命令行在命令执行前输出到屏幕上。当我们用“@”字符在命令行前，那么，这个命令将不被make显示出来，最具代表性的例子是，我们用这个功能来像屏幕显示一些信息。如：

   @echo 正在编译XXX模块......

当make执行时，会输出“正在编译XXX模块......”字串，但不会输出命令，如果没有“@”，那么，make将输出：

   echo 正在编译XXX模块......

   正在编译XXX模块......

如果make执行时，带入make参数“-n”或“--just-print”，那么其只是显示命令，但不会执行命令，这个功能很有利于我们调试我们的Makefile，看看我们书写的命令是执行起来是什么样子的或是什么顺序的。

而make参数“-s”或“--slient”则是全面禁止命令的显示。

**b.定义命令包 define**

define run-yacc

yacc $(firstword $^)

mv y.tab.c $@

endef

这里，“run-yacc”是这个命令包的名字，其不要和Makefile中的变量重名。在“define”和“endef”中的两行就是命令序列。这个命令包中的第一个命令是运行Yacc程序，因为Yacc程序总是生成“y.tab.c”的文件，所以第二行的命令就是把这个文件改改名字。还是把这个命令包放到一个示例中来看看吧。

   foo.c : foo.y

           $(run-yacc)

[我们可以看见，要使用这个命令包，我们就好像使用变量一样。在这个命令包的使用中，命令包“run-yacc”中的“$^”就是“foo.y”，“$@”就是“foo.c”，make在执行命令包时，命令包中的每个命令会被依次独立执行。](mailto:我们可以看见，要使用这个命令包，我们就好像使用变量一样。在这个命令包的使用中，命令包\“run-yacc\”中的\“$^\”就是\“foo.y\”，\“$@\”就是\“foo.c\”，make在执行命令包时，命令包中的每个命令会被依次独立执行。)

**c.MAKECMDGOALS**

该变量记录了命令行参数指定的终极目标列表，没有通过参数指定终极目标时此变量为空。该变量仅限于用在特殊场合(比如判断)，在 Makefile 中最好不要对它进行重新定义。

**d.MAKEFLAGS**

make递归执行时，上层make称为主控make，它的命令行选项会通过环境变量MAKEFLAGS传递给子make进程。MAKEFLAGS会被主控make自动设置为包含所执行的make时的命令行选项的字符串。比如主控执行make时使用-k和-s选项，那么MAKEFLAGS的值就为ks，子make进程处理时，会把此环境变量作为执行的命令行选项。

**e. .SUFFIXES**

声明该部分为隐式规则

<http://wiki.ubuntu.org.cn/%E8%B7%9F%E6%88%91%E4%B8%80%E8%B5%B7%E5%86%99Makefile:%E9%9A%90%E5%90%AB%E8%A7%84%E5%88%99>

**f. .EXPORT\_ALL\_VARIABLES**

只是作为一个目标存在，指示make将所有变量输出到子进程中。

**7.参数列表**

-k:

-s:

--no-builtin-rules: