

Makefile详解（超级好）



nginx_h 1

2009-6-10

陈皓

概述

——

什么是makefile？或许很多Windows的程序员都不知道这个东西，因为那些Windows的IDE都为你做了这个工作，但我觉得要作一个好的和 professional的程序员，makefile还是要懂。这就好像现在有这么多的HTML的编辑器，但如果你想成为一个专业人士，你还是要了解 HTML的标识的含义。特别在 Unix下的软件编译，你就不能不自己写makefile了，会不会写makefile，从一个侧面说明了一个人是否具备完成大型工程的能力。

因为，makefile关系到了整个工程的编译规则。一个工程中的源文件不计数，其按类型、功能、模块分别放在若干个目录中，makefile定义了一系列的规则来指定，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，甚至于进行更复杂的功能操作，因为makefile就像一个Shell 脚本一样，其中也可以执行操作系统的命令。

makefile带来的好处就是——“自动化编译”，一旦写好，只需要一个make命令，整个工程完全自动编译，极大的提高了软件开发的效率。make是一个命令工具，是一个解释makefile中指令的命令工具，一般来说，大多数的IDE都有这个命令，比如：Delphi的make，Visual C++的nmake，Linux下GNU的make。可见，makefile都成为了一种在工程方面的编译方法。

现在讲述如何写makefile的文章比较少，这是我想写这篇文章的原因。当然，不同厂商的make各不相同，也有不同的语法，但其本质都是在“文件依赖性”上做文章，这里，我仅对GNU的make进行讲述，我的环境是RedHat Linux 8.0，make的版本是3.80。毕竟，这个make是应用最为广泛的，也是用得最多的。而且其还是最遵循于IEEE 1003.2-1992 标准的（POSIX.2）。

在这篇文档中，将以C/C++的源码作为我们基础，所以必然涉及一些关于C/C++的编译的知识，相关于这方面的内容，还请各位查看相关的编译器的文档。这里所默认的编译器是UNIX下的GCC和CC。

关于程序的编译和链接

在此，我想多说关于程序编译的一些规范和方法，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

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Object File.



nginx_h 1
2009-6-10

好，言归正传，GNU的make有许多的内容，闲言少叙，还是让我们开始吧。

Makefile 介绍

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。

首先，我们用一个示例来说明Makefile的书写规则。以便给大家一个感兴认识。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。

一、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也就是一个目标文件，可以是Object File，也可以是执行文件。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给你慢慢到来。内容还多着呢。：)

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
```



乡村小说

2018-10-29 15:35

广告

看小说付费？别被套路，来这里免费读



乡村小说

立即下载



nginx_h ¹

2009-6-10

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

(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。

三、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。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，我们可以显示要 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。



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，我们声明一个变量，叫objects, OBJECTS, objs, OBJs, obj, 或是 OBJ，反正不管什么啦，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)
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)
```

于是如果有新的.o文件加入，我们只需简单地修改一下objects变量就可以了。

关于变量更多的话题，我会在后续给你一一道来。

五、让make自动推导

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。我们的是新的makefile又出炉了。

```
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
```



nginx_h 1

2009-6-10

```
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。

六、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。

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。

一、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。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，可以用反斜框进行转义，如：“\#”。



nginx_h 1

2009-6-10

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

二、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

三、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，会把找寻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。

四、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。

五、make的工作方式

GNU的make工作时的执行步骤入下：（想来其它的make也是类似）



nginx_h 1
2009-6-10

1、读入所有的Makefile。

- 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地方。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。

书写规则

规则包含两个部分，一个是依赖关系，一个是生成目标的方法。

在Makefile中，规则的顺序是很重要的，因为，Makefile中只应该有一个最终目标，其它的目标都是被这个目标所连带出来的，所以一定要让make知道你的最终目标是什么。一般来说，定义在Makefile中的目标可能会有很多，但是第一条规则中的目标将被确立为最终的目标。如果第一条规则中的目标有很多个，那么，第一个目标会成为最终的目标。make所完成的也就是这个目标。

好了，还是让我们来看一看如何书写规则。

一、规则举例

```
foo.o : foo.c defs.h # foo模块
```

```
cc -c -g foo.c
```

看到这个例子，各位应该不是很陌生了，前面也已说过，foo.o是我们的目标，foo.c和defs.h是目标所依赖的源文件，而只有一个命令“cc -c -g foo.c”（以Tab键开头）。这个规则告诉我们两件事：

- 1、文件的依赖关系，foo.o依赖于foo.c和defs.h的文件，如果foo.c和defs.h的文件日期要比foo.o文件日期要新，或是foo.o不存在，那么依赖关系发生。
- 2、如果生成（或更新）foo.o文件。也就是那个cc命令，其说明了，如何生成foo.o这个文件。（当然foo.c文件include了defs.h文件）

二、规则的语法

```
targets : prerequisites
```

```
command
```

```
...
```

或是这样：

```
targets : prerequisites ; command
```

```
command
```

```
...
```

targets是文件名，以空格分开，可以使用通配符。一般来说，我们的目标基本上是一个文件，但也有可能是多个文件。

command是命令行，如果其不与“target:prerequisites”在一行，那么，必须以[Tab键]开头，如果和prerequisites在一行，那么可以用分号做为分隔。（见上）

prerequisites也就是目标所依赖的文件（或依赖目标）。如果其中的某个文件要比目标文件要

规则告诉make两件事，文件的依赖关系和如何生成目标文件。

一般来说，make会以UNIX的标准Shell，也就是/bin/sh来执行命令。

二 在规则中指定通配符



打开贴吧APP，随时随地开启逗比模式

立即打开

12. 波浪号（“~”）字符在文件名中也有比较特殊的用途。如果是“~/test”，这就表示当前用户的

\$HOME目录下的test目录。而“~hchen/test”则表示用户hchen的宿主目录下的test目录。（这些都是Unix下的小知识了，make也支持）而在Windows或是MS-DOS下，用户没有宿主目录，那么波浪号所指的目录则根据环境变量“HOME”而定。

通配符代替了你一系列的文件，如“*.c”表示所以后缀为c的文件。一个需要我們注意的是，如果我们的文件名中有通配符，如：“*”，那么可以用转义字符“\”，如“*”来表示真实的“*”字符，而不是任意长度的字符串。



nginx_h 1
2009-6-10

好吧，还是先来看几个例子吧：

clean:

```
rm -f *.o
```

上面这个例子我不不多说了，这是操作系统Shell所支持的通配符。这是在命令中的通配符。

print: *.c

```
lpr -p $?
```

```
touch print
```

上面这个例子说明了通配符也可以在我们的规则中，目标print依赖于所有的[c]文件。其中的“\$?”是一个自动化变量，我会在后面给你讲述。

```
objects = *.o
```

上面这个例子，表示了，通符同样可以用在变量中。并不是说[*o]会展开，不！objects的值就是“*.o”。Makefile中的变量其实就是C/C++中的宏。如果你要让通配符在变量中展开，也就是让objects的值是所有[o]的文件名的集合，那么，你可以这样：

```
objects := $(wildcard *.o)
```

这种用法由关键字“wildcard”指出，关于Makefile的关键字，我们将在后面讨论。

四、文件搜寻

在一些大的工程中，有大量的源文件，我们通常的做法是把这许多的源文件分类，并存放在不同的目录中。所以，当make需要去找寻文件的依赖关系时，你可以在文件前加上路径，但最好的方法是把一个路径告诉make，让make在自动去找。

Makefile文件中的特殊变量“VPATH”就是完成这个功能的，如果没有指明这个变量，make只会在当前的目录中去找寻依赖文件和目标文件。如果定义了这个变量，那么，make就会在当当前目录找不到的情况下，到所指定的目录中去找寻文件了。

```
VPATH = src:../headers
```

上面的的定义指定两个目录，“src”和“../headers”，make会按照这个顺序进行搜索。目录由“冒号”分隔。（当然，当前目录永远是最高优先搜索的地方）

可以指定不同的文件在不同的搜索目录中。这是一个很灵活的功能。它的使用方法有三种：

1、`vpath <pattern>; <directories>;`

为符合模式<pattern>;的文件指定搜索目录<directories>;。

2、`vpath <pattern>;`

清除符合模式<pattern>;;的文件搜索目录。

3、`vpath`

清除所有已被设置好了的文件搜索目录。

`vpath`使用方法中的<pattern>;需要包含“%”字符。“%”的意思是匹配零或若干字符，例如，“%.h”表示所有以“.h”结尾的文件。<pattern>;指定了要搜索的文件集，而<directories>;则指定了<pattern>;的文件集的搜索的目录。例如：

```
vpath %.h ../headers
```

该语句表示，要求make在“../headers”目录下搜索所有以“.h”结尾的文件。（如果某文件在当前目录没有找到的话）

我们可以连续地使用`vpath`语句，以指定不同搜索策略。如果连续的`vpath`语句中出现了相同的<pattern>;，或是被重复了的<pattern>;，那么，make会按照`vpath`语句的先后顺序来执行搜索。如：

```
vpath %.c foo
```

```
vpath % blish
```

```
vpath %.c bar
```

其表示“.c”结尾的文件，先在“foo”目录，然后是“blish”，最后是“bar”目录。

```
vpath %.c foo:bar
```

```
vpath % blish
```

而上面的语句则表示“.c”结尾的文件，先在“foo”目录，然后是“bar”目录，最后才是“blish”目录。

五、伪目标

最早先的一个例子中，我们提到过一个“clean”的目标，这是一个“伪目标”，

clean:

```
rm *.o temp
```

正像我们前面例子中的“clean”一样，即然我们生成了许多文件编译文件，我们也应该提供一个清除它们的“目标”以备完整地重编译而用。（以“make clean”来使用该目标）



nginx_h 1

2009-6-10

因为，我们并不生成“clean”这个文件。“伪目标”并不是一个文件，只是一个标签，由于“伪目标”不是文件，所以make无法生成它的依赖关系和决定它是否要执行。我们只有通过显示地指明这个“目标”才能让其生效。当然，“伪目标”的取名不能和文件名重名，不然其就失去了“伪目标”的意义了。

当然，为了避免和文件重名的这种情况，我们可以使用一个特殊的标记“.PHONY”来显示地指明一个目标是“伪目标”，向make说明，不管是否有这个文件，这个目标就是“伪目标”。

```
.PHONY : clean
```

只要有这个声明，不管是否有“clean”文件，要运行“clean”这个目标，只有“make clean”这样。

clean:

```
rm *.o temp
```

伪目标一般没有依赖的文件。但是，我们也可以为伪目标指定所依赖的文件。伪目标同样可以作为“默认目标”，只要将其放在第一个。一个示例就是，如果你的 Makefile需要一口气生成若干个可执行文件，但你只想简单地敲一个make完事，并且，所有的目标文件都写在一个Makefile中，那么你可以使用“伪目标”这个特性：

```
all : prog1 prog2 prog3
```

```
.PHONY : all
```

```
prog1 : prog1.o utils.o
```

```
cc -o prog1 prog1.o utils.o
```

```
prog2 : prog2.o
```

```
cc -o prog2 prog2.o
```

```
prog3 : prog3.o sort.o utils.o
```

```
cc -o prog3 prog3.o sort.o utils.o
```

我们知道，Makefile中的第一个目标会被作为其默认目标。我们声明了一个“all”的伪目标，其依赖于其它三个目标。由于伪目标的特性是，总是被执行的，所以其依赖的那三个目标就总是不如“all”这个目标新。所以，其它三个目标的规则总是会被决议。也就达到了我们一口气生成多个目标的目的。“`.PHONY : all`”声明了“all”这个目标为“伪目标”。

随便提一句，从上面的例子我们可以看出，目标也可以成为依赖。所以，伪目标同样也可成为依赖。看下面的例子：

```
.PHONY: cleanall cleanobj cleandiff
```

```
cleanall : cleanobj cleandiff
```

```
rm program
```

```
cleanobj :
```

```
rm *.o
```

```
cleandiff :
```

```
rm *.diff
```

“make clean”将清除所有要被清除的文件。“cleanobj”和“cleandiff”这两个伪目标有点像“子程序”的意思。我们可以输入“make cleanall”和“make cleanobj”和“make cleandiff”命令来达到清除不同种类文件的目的。

六、多目标

Makefile的规则中的目标可以不止一个，其支持多目标，有可能我们的多个目标同时依赖于一个文件，并且其生成的命令大体类似。于是我们就能把其合并起来。当然，多个目标的生成规则的执行命令是同一个，这可能会给我们带来麻烦，不过好在我们的可以使用一个自动化变量“\$@"（关于自动化变量，将在后面讲述），这个变量表示着目前规则中所有的目标的集合，这样说可能很抽象，还是看一个例子吧。

```
bigoutput littleoutput : text.g
```

```
generate text.g -$(subst output,,$@) >; $@
```

上述规则等价于：

```
bigoutput : text.g
```

```
generate text.g -big >; bigoutput
```

其中，`$(subst output,$@)`中的“\$”表示执行一个Makefile的函数，函数名为subst，后面的为参数。关于函数，将在后面讲述。这里的这个函数是截取字符串的意思，“\$@”表示目标的集合，就像一个数组，“\$@”依次取出目标，并执于命令。



店宝宝

2018-10-29 15:35

广告

开网店卖什么赚钱？多种货源，自动发货，一键装修，推广服务这样做3天就能赚钱



店宝宝

[查看详情](#)nginx_h¹

2009-6-10

七、静态模式

静态模式可以更加容易地定义多目标的规则，可以让我们的规则变得更加的有弹性和灵活。我们还是先来看一下语法：

```
<targets ...>; <target-pattern>; <prereq-patterns ...>;
<commands>;
```

...

targets定义了一系列的目标文件，可以有通配符。是目标的一个集合。

target-parrtern是指明了targets的模式，也就是的目标集模式。

prereq-parrterns是目标的依赖模式，它对target-parrtern形成的模式再进行一次依赖目标的定义。

这样描述这三个东西，可能还是没有说清楚，还是举个例子来说明一下吧。如果我们的<target-parrtern>;定义成“%.o”，意思是我们的<target>;集合中都是以“.o”结尾的，而如果我们的<prereq-parrterns>;定义成“%.c”，意思是对<target-parrtern>;所形成的目标集进行二次定义，其计算方法是，取<target-parrtern>;模式中的“%”（也就是去掉了[.o]这个结尾），并为其加上[.c]这个结尾，形成的新集合。

所以，我们的“目标模式”或是“依赖模式”中都应该有“%”这个字符，如果你的文件名中有“%”那么你可以使用反斜杠“\”进行转义，来标明真实的“%”字符。

看一个例子：

```
objects = foo.o bar.o
```

```
all: $(objects)
```

```
$(objects): %.o: %.c
```

```
$(CC) -c $(CFLAGS) $< -o $@
```

上面的例子中，指明了我们的目标从\$object中获取，“%.o”表明要所有以“.o”结尾的目标，也就

的“\$<”和“\$@”则是自动化变量，“\$<”表示所有的依赖目标集（也就是“foo.c bar.c”），“\$@”表示目标集（也就是“foo.o bar.o”）。于是，上面的规则展开后等价于下面的规则：

```
foo.o : foo.c
$(CC) -c $(CFLAGS) foo.c -o foo.o
bar.o : bar.c
$(CC) -c $(CFLAGS) bar.c -o bar.o
```

试想，如果我们的“%.o”有几百个，那种我们只要用这种很简单的“静态模式规则”就可以写完一堆规则，实在是太有效率了。“静态模式规则”的用法很灵活，如果用得好，那会一个很强大的功能。再看一个例子：

```
files = foo.elc bar.o lose.o
$(filter %.o,$(files)): %.o: %.c
$(CC) -c $(CFLAGS) $< -o $@
$(filter %.elc,$(files)): %.elc: %.el
emacs -f batch-byte-compile $<
```

\$(filter %.o,\$(files))表示调用Makefile的filter函数，过滤“\$filter”集，只要其中模式为“%.o”的内容。其它的它内容，我就不用多说了吧。这个例子展示了Makefile中更大的弹性。

八、自动生成依赖性

在Makefile中，我们的依赖关系可能会需要包含一系列的头文件，比如，如果我们的main.c中有一句“#include "defs.h"”，那么我们的依赖关系应该是：

```
main.o : main.c defs.h
```

但是，如果是一个比较大型的工程，你必需清楚哪些C文件包含了哪些头文件，并且，你在加入或删除头文件时，也需要小心地修改Makefile，这是一个很没有维护性的工作。为了避免这种繁重而又容易出错的事情，我们可以使用C/C++编译的一个功能。大多数的C/C++编译器都支持一个“-M”的选项，即自动找寻源文件中包含的头文件，并生成一个依赖关系。例如，如果我们执行下面的命令：



nginx_h 1
2009-6-10

```
cc -M main.c
```

其输出是：

```
main.o : main.c defs.h
```

于是由编译器自动生成的依赖关系，这样一来，你就不必再手动书写若干文件的依赖关系，而由编译器自动生成了。需要提醒一句的是，如果你使用GNU的C/C++编译器，你得用“-MM”参数，不然，“-M”参数会把一些标准库的头文件也包含进来。

gcc -M main.c的输出是：

```
main.o: main.c defs.h /usr/include/stdio.h /usr/include/features.h \
/usr/include/sys/cdefs.h /usr/include/gnu/stubs.h \
/usr/lib/gcc-lib/i486-suse-linux/2.95.3/include/stddef.h \
/usr/include/bits/types.h /usr/include/bits/pthreadtypes.h \
/usr/include/bits/sched.h /usr/include/libio.h \
```



```
/usr/lib/gcc-lib/i486-suse-linux/2.95.3/include/stdarg.h \
/usr/include/bits/stdio_lim.h
```

gcc -MM main.c的输出则是：

```
main.o: main.c defs.h
```

那么，编译器的这个功能如何与我们的Makefile联系在一起呢。因为这样一来，我们的Makefile也要根据这些源文件重新生成，让 Makefile自己依赖于源文件？这个功能并不现实，不过我们可以有其它手段来迂回地实现这一功能。GNU组织建议把编译器为每一个源文件的自动生成的依赖关系放到一个文件中，为每一个“name.c”的文件都生成一个“name.d”的Makefile文件，[.d]文件中就存放对应[.c]文件的依赖关系。

于是，我们可以写出[.c]文件和[.d]文件的依赖关系，并让make自动更新或自成[.d]文件，并把其包含在我们的主Makefile中，这样，我们就可以自动化地生成每个文件的依赖关系了。

这里，我们给出了一个模式规则来产生[.d]文件：

```
%.d: %.c
@set -e; rm -f $@; \
$(CC) -M $(CPPFLAGS) $< > $@.$$$$; \
sed 's,\($*\)\.o[ :]*,1.o $@ : ,g' < $@.$$$$ > $@; \
rm -f $@.$$$$
```

这个规则的意思是，所有的[.d]文件依赖于[.c]文件，“rm -f \$@"的意思是删除所有的目标，也就是[.d]文件，第二行的意思是，为每个依赖文件“\$<”，也就是[.c]文件生成依赖文件，“\$@"表示模式“%.d”文件，如果有一个C文件是name.c，那么“%”就是“name”，“\$\$\$\$”意为一个随机编号，第二行生成的文件有可能是“name.d.12345”，第三行使用sed命令做了一个替换，关于sed命令的用法请参看相关的使用文档。第四行就是删除临时文件。

总而言之，这个模式要做的事就是在编译器生成的依赖关系中加入[.d]文件的依赖，即把依赖关系：

```
main.o : main.c defs.h
```

转成：

```
main.o main.d : main.c defs.h
```

于是，我们的[.d]文件也会自动更新了，并会自动生成了，当然，你还可以在这个[.d]文件中加入的不只是依赖关系，包括生成的命令也可一并加入，让每个[.d]文件都包含一个完整的规则。一旦我们完成这个工作，接下来，我们就要把这些自动生成的规则放进我们的主Makefile中。我们可以使用 Makefile的“include”命令，来引入别的Makefile文件（前面讲过），例如：

```
sources = foo.c bar.c
```

```
include $(sources:.c=.d)
```

上述语句中的“\$(sources:.c=.d)”中的“.c=.d”的意思是做一个替换，把变量\$(sources)所有[.c]的字串都替换成[.d]，关于这个“替换”的内容，在后面我会有更为详细的讲述。当然，你得注意次序，因为include是按次来载入文件，最先载入的[.d]文件中的目标会成为默认目标。



nginx_h 1
2009-6-10

还有一个全局的办法是，给make加上“-i”或是“--ignore-errors”参数，那么，Makefile中所有命令

还有一个要提一下的make的参数的是“-k”或是“--keep-going”，这个参数的意思是，如果某规则中的命令出错了，那么就终止该规则的执行，但继续执行其它规则。

四、嵌套执行make

在一些大的工程中，我们会把我们不同模块或是不同功能的源文件放在不同的目录中，我们可以在每个目录中都书写一个该目录的Makefile，这有利于让我们的Makefile变得更加地简洁，而不至于把所有的东西全部写在一个Makefile中，这样会很难维护我们的Makefile，这个技术对于我们模块编译和分段编译有着非常大的好处。

例如，我们有一个子目录叫subdir，这个目录下有个Makefile文件，来指明了这个目录下文件的编译规则。那么我们总控的Makefile可以这样书写：

```
subsystem:
```

```
cd subdir && $(MAKE)
```

其等价于：

```
subsystem:
```

```
$(MAKE) -C subdir
```

定义\$(MAKE)宏变量的意思是，也许我们的make需要一些参数，所以定义成一个变量比较利于维护。这两个例子的意思都是先进入“subdir”目录，然后执行make命令。

我们把这个Makefile叫做“总控Makefile”，总控Makefile的变量可以传递到下级的Makefile中（如果你显示的声明），但是不会覆盖下层的Makefile中所定义的变量，除非指定了“-e”参数。

如果你要传递变量到下级Makefile中，那么你可以使用这样的声明：

```
export <variable ...>;
```

如果你不想让某些变量传递到下级Makefile中，那么你可以这样声明：

```
unexport <variable ...>;
```

如：

示例一：

```
export variable = value
```

其等价于：

```
variable = value
```

```
export variable
```

其等价于：

```
export variable := value
```

其等价于：

```
variable := value
```

```
export variable
```

示例二：

```
export variable += value
```

其等价于：

```
variable += value
```

```
export variable
```

如果你要传递所有的变量，那么，只要一个export就行了。后面什么也不用跟，表示传递所有的变量。

需要注意的是，有两个变量，一个是SHELL，一个是MAKEFLAGS，这两个变量不管你是否

么MAKEFILES变量将会是这些参数，并会传递到下层Makefile中，这是一个系统级的环境变量。

但是make命令中的有几个参数并不往下传递，它们是“-C”、“-f”、“-h”、“-o”和“-W”（有关Makefile参数的细节将在后面说明），如果你不想往下层传递参数，那么，你可以这样来：

subsystem:

```
cd subdir && $(MAKE) MAKEFLAGS=
```

如果你定义了环境变量MAKEFLAGS，那么你得确信其中的选项是大家都用得到的，如果其中有“-t”、“-n”，和“-q”参数，那么将会有让你意想不到的结果，或许会让你异常地恐慌。

还有一个在“嵌套执行”中比较有用的参数，“-w”或是“--print-directory”会在make的过程中输出一些信息，让你看到目前的工作目录。比如，如果我们的下级make目录是“/home/hchen/gnu/make”，如果我们使用“make -w”来执行，那么当进入该目录时，我们会看到：

```
make: Entering directory `/home/hchen/gnu/make'.
```

而在完成下层make后离开目录时，我们会看到：

```
make: Leaving directory `/home/hchen/gnu/make'
```

当你使用“-C”参数来指定make下层Makefile时，“-w”会被自动打开的。如果参数中有“-s”（“--silent”）或是“--no-print-directory”，那么，“-w”总是失效的。

五、定义命令包

如果Makefile中出现一些相同命令序列，那么我们可以为这些相同的命令序列定义一个变量。定义这种命令序列的语法以“define”开始，以“endef”结束，如：

```
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”（有关这种以“\$”开头的特殊变量，我们会在后面介绍），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。

使用变量

在Makefile中的定义的变量，就像是C/C++语言中的宏一样，他代表了一个文本字符串，在Makefile中执行的时候其会自动原模原样地展开在所使用的地方。其与C/C++所不同的是，你可以在Makefile中改变其值。在Makefile中，变量可以使用在“目标”，“依赖目标”，“命令”或是Makefile的其它部分中。

变量的命名字可以包含字符、数字，下划线（可以是数字开头）

好贴



iamgong ¹

2009-10-4

好 好



超姐的弟弟 ¹

2009-10-12

赞！！



sqddsunjian ¹

2010-3-19

你咋这么强的呢



122.205.94.*

2010-3-24

真的十分感谢!



sqxing1989 ¹

2010-4-8

太强大了



精灵夜路 ¹

2010-4-14

这个不错，正好对我有帮助了，谢谢了



58.198.94.*

2010-4-27

👉 太棒了~~内牛满面啊，谢楼主呀



58.40.16.*

2010-4-27

非常感谢！

我还要好好学学！



119.164.14.*

2010-4-30



正在学习 linux 呢 终于找到了呵呵



百君 1
2010-5-11

好。。



tree_herber 1
2010-6-1

很有用，多谢了



61.178.21.*
2010-6-1

摘录走了。真是牛人！



fuxiao511 1
2010-6-11

占位为了自己查看方便
谢谢楼主了，很好很强大



dyingfair 1
2010-6-15

好厉害

1/4



年轻人的潮流文化社区

客户端

登录

注册

反馈

©2018 Baidu 使用百度前必读