0 Makefile 概述

什么是 makefile？或许很多 Winodws 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，不同产商的 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。必竟，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。

1 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。

1.1 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 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到来。内容还多着呢。：）

1.2 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
```

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“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。

1.3 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。

1.4 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。

1.5 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
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。

1.6 另类风格的 makefile

即然我们的 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。

1.7 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。

---

2 Makefile 总述

2.1 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：“\#”。

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

2.2Makefile 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。

2.3 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。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/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。

2.4 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。

2.5 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）

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。

3 Makefile 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。

3.1 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）

3.2 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新，那么，目标就被认为是“过时的”，被认为是需要重生成的。这个在前面已经讲过了。

如果命令太长，你可以使用反斜框（‘\’）作为换行符。make 对一行上有多少个字符没有限制。规则告诉 make 两件事，文件的依赖关系和如何成成目标文件。

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

3.3 在规则中使用通配符

如果我们想定义一系列比较类似的文件，我们很自然地就想起使用通配符。make 支持三各通配符：“*”，“?”和“[...]”。这是和 Unix 的 B-Shell 是相同的。

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。一个需要我们注意的是，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。

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。

3.4 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）

另一个设置文件搜索路径的方法是使用 make 的“vpath”关键字（注意，它是全小写的），这不是变量，这是一个 make 的关键字，这和上面提到的那个 VPATH 变量很类似，但是它更为灵活。它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>的文件的搜索目录。

3. vpath

   清除所有已被设置好了的文件搜索目录。

vapth 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。

3.5 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）

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。于是整个过程可以这样写：

```
 .PHONY: clean
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

3.6 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，这可能会可我们带来麻烦，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
littleoutput : text.g
        generate text.g -little > littleoutput
```

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。

3.7 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$object 中获取，“%.o”表明要所有以“.o”结尾的目标，也就是“foo.o bar.o”，也就是变量$object 集合的模式，而依赖模式“%.c”则取模式“%.o”的“%”，也就是“foo bar”，并为其加下“.c”的后缀，于是，我们的依赖目标就是“foo.c bar.c”。而命令中的“$<”和“$@”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（也褪恰癯 oo.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。其的它内容，我就不用多说了吧。这个例字展示了 Makefile 中更大的弹性。

3.8 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：

```
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/include/_G_config.h /usr/include/wchar.h \
    /usr/include/bits/wchar.h /usr/include/gconv.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]文件都包含一个完赖的规则。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

4 Makefile 书写命令

每条规则中的命令和操作系统 Shell 的命令行是一致的。make 会一按顺序一条一条的执行命令，每条命令的开头必须以[Tab]键开头，除非，命令是紧跟在依赖规则后面的分号后的。在命令行之间中的空格或是空行会被忽略，但是如果该空格或空行是以 Tab 键开头的，那么 make 会认为其是一个空令。

我们在 UNIX 下可能会使用不同的 Shell，但是 make 的命令默认是被“/bin/sh”——UNIX 的标准 Shell 解释执行的。除非你特别指定一个其它的 Shell。Makefile 中，“#”是注释符，很像 C/C++的“//”，其后的本行字符都被注释。

4.1 显示命令

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。

4.2 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。

4.3 命令出错

每当命令运行完后，make 会检测每个命令的返回码，如果命令返回成功，那么 make 会执行下一条命令，当规则中所有的命令成功返回后，这个规则就算是成功完成了。如果一个规则中的某个命令出错了（命令退出码非零），那么 make 就会终止执行当前规则，这将有可能终止所有规则的执行。

有些时候，命令的出错并不表示就是错误的。例如 mkdir 命令，我们一定需要建立一个目录，如果目录不存在，那么 mkdir 就成功执行，万事大吉，如果目录存在，那么就出错了。我们之所以使用 mkdir 的意思就是一定要有这样的一个目录，于是我们就不希望 mkdir 出错而终止规则的运行。

为了做到这一点，忽略命令的出错，我们可以在 Makefile 的命令行前加一个减号“-”（在 Tab 键之后），标记为不管命令出不出错都认为是成功的。如：

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

还有一个全局的办法是，给 make 加上“-i”或是“--ignore-errors”参数，那么，Makefile 中所有命令都会忽略错误。而如果一个规则是以“.IGNORE”作为目标的，那么这个规则中的所有命令将会忽略错误。这些是不同级别的防止命令出错的方法，你可以根据你的不同喜欢设置。

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，那么就终目该规则的执行，但继续执行其它规则。

4.4 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 export，其总是要
传递到下层 Makefile 中，特别是 MAKEFILES 变量，其中包含了 make 的参数信息，如果我们执行“总控
Makefile”时有 make 参数或是在上层 Makefile 中定义了这个变量，那么 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会用到的，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”（“--
slient”）或是“--no-print-directory”，那么，“-w”总是失效的。
---++++4.5 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。