GNU make中文手册

ver - 3.8

译者：徐海兵

2004-09-11

目 录

GNU make中文手册 1
ver - 3.8 1
第一章：概述 7
1.1 Make之前基本知识： 7
1.1.1 概述 7
1.1.2 准备知识 8
1.2 GNU make 介绍 8
1.2.1 Makefile简介 9
1.2.2 Makefile规则介绍 10
1.2.3 简单的示例 11
1.2.4 make如何工作 12
1.2.5 指定变量 13
1.2.6 自动推导规则 14
1.2.7 另类风格的makefile 15
1.2.8 清除工作目录过程文件 16
第二章：Makefile 总述 16
2 Makefile概述 16
2.1 Makefile的内容 16
2.2 makefile文件的命名 18
2.3 包含其它makefile文件 18
2.4 变量 MAKEFILES 20
2.5 变量 MAKEFILE_LIST 21
2.6 其他特殊变量 22
2.7 makefile文件的重建 22
2.8 重载另外一个makefile 23
2.9 make如何解析makefile文件 24
2.9.1 变量取值 25
2.9.2 条件语句 25
2.9.3 规则的定义 25
2.10 总结 25
第三章：Makefile的规则 26
3 Makefile规则 26
3.1 一个例子 27
3.2 规则语法 27
3.3 依赖的类型 28
3.4 文件名使用通配符 29
3.4.1 统配符使用举例 30
3.4.2 通配符存在的缺陷 30
3.4.3 函数wildcard 31
3.5 目录搜寻 32
3.5.1 一般搜索（变量VPATH） 32
3.5.2 选择性搜索（关键字vpath） 32
3.5.3 目录搜索的机制 33
3.5.4 命令行和搜索目录 35
3.5.5 隐含规则和搜索目录 36

3.5.6 库文件和搜索目录 36
3.6 Makefile伪目标 37
3.7 强制目标（没有命令或依赖的规则） 40
3.8 空目标文件 41
3.9 Makefile的特殊目标 41
3.10 多目标 44
3.11 多规则目标 44
3.12 静态模式 45
3.12.1 静态模式规则的语法 45
3.12.2 静态模式和隐含规则 47
3.13 双冒号规则 48
3.14 自动产生依赖 49
第四章：规则的命令 51
4 规则中书写命令 51
4.1 命令回显 51
4.2 命令的执行 52
4.3 并发执行命令 53
4.4 命令执行的错误 54
4.5 中断make的执行 56
4.6 make的递归执行 56
4.6.1 变量MAKE 57
4.6.2 变量和递归 58
4.6.3 命令行选项和递归 61
4.6.4 -w选项 63
4.7 定义命令包 63
4.8 空命令 65
第五章：Makefile中的变量 65
5 使用变量 65
5.1 变量的引用 66
5.2 两种变量定义（赋值） 68
5.2.1 递归展开式变量 68
5.2.2 直接展开式变量 69
5.2.3 如何定义一个空格 70
5.2.4 “?=”操作符 71
5.3 变量的高级用法 71
5.3.1 变量的替换引用 72
5.3.2 变量的套嵌引用 72
5.4 变量取值 76
5.5 如何设置变量 76
5.6 追加变量值 77
5.7 override 指示符 79
5.8 多行定义 80
5.9 系统环境变量 81
5.10 目标指定变量 83
5.11 模式指定变量 84
第六章：Makefile的条件执行 85
6 Makefile的条件判断 85
6.1 一个例子 85
6.2 条件判断的基本语法 86

6.3 标记测试的条件语句 89
第七章：make的内嵌函数 89
7 make的函数 89
7.1 函数的调用语法 90
7.2 文本处理函数 91
7.2.1 $(subst FROM,TO,TEXT) 91
7.2.2 $(patsubst PATTERN,REPLACEMENT,TEXT) 91
7.2.3 $(strip STRINT) 92
7.2.4 $(findstring FIND,IN) 92
7.2.5 $(filter PATTERN…,TEXT) 93
7.2.6 $(filter-out PATTERN...,TEXT) 93
7.2.7 $(sort LIST) 94
7.2.8 $(word N,TEXT) 94
7.2.9 $(wordlist S,E,TEXT) 94
7.2.10 $(words TEXT) 94
7.2.11 $(firstword NAMES…) 95
7.3 文件名处理函数 95
7.3.1 $(dir NAMES…) 95
7.3.2 $(notdir NAMES…) 96
7.3.3 $(suffix NAMES…) 96
7.3.4 $(basename NAMES…) 96
7.3.5 $(addsuffix SUFFIX,NAMES…) 97
7.3.6 $(addprefix PREFIX,NAMES…) 97
7.3.7 $(join LIST1,LIST2) 98
7.3.8 $(wildcard PATTERN) 98
7.4 foreach 函数 98
7.5 if 函数 100
7.6 call函数 100
7.7 value函数 102
7.8 eval函数 103
7.9 origin函数 104
7.10 shell函数 106
7.11 make的控制函数 106
7.11.1 $(error TEXT…) 107
7.11.2 $(warning TEXT…) 107
第八章：执行make 108
8 执行make 108
8.1 指定makefile文件 108
8.2 指定终极目标 109
8.3 替代命令的执行 111
8.4 防止特定文件重建 112
8.5 替换变量定义 113
8.6 使用make进行编译测试 114
8.7 make的命令行选项 115
第九章：make的隐含规则 119
9 使用隐含规则 119
9.1 隐含规则的使用 119
9.2 make的隐含规则一览 121
9.3 隐含变量 124
9.3.1 代表命令的变量 124
9.3.2 命令参数的变量 125
9.4 make隐含规则链 126

9.5 模式规则 128
9.5.1 模式规则介绍 128
9.5.2 模式规则示例 129
9.5.3 自动化变量 130
9.5.4 模式的匹配 133
9.5.5 万用规则 133
9.5.6 重建内嵌隐含规则 134
9.6 缺省规则 135
9.7 后缀规则 135
9.8 隐含规则搜索算法 137
第十章：使用make更新静态库文件 138
10 更新静态库文件 138
10.1 库成员作为目标 138
ARCHIVE(MEMBER) 138
10.2 静态库的更新 139
10.2.1 更新静态库的符号索引表 140
10.3 **make**静态库的注意事项 141
10.4 静态库的后缀规则 141
第十一章 ： GNU make的特点 142
11 GNU make的一些特点 142
11.1 源自**System v**的特点 142
11.2 源自其他版本的特点 143
11.3 GNU make自身的特点 143
第十二章 和其它版本的兼容 144
12 不兼容性 144
第十三章 Makefile的约定 146
13 书写约定 146
13.1 基本的约定 146
13.2 规则命令行的约定 147
13.3 代表命令变量 149
13.4 安装目录变量 150
13.5 **Makefile**的标准目标名 154
13.6 安装命令分类 159
第十四章 make的常见错误信息 161
14 make产生的错误信息 161
附录**1**：关键字索引 163
GNU make可识别的指示符： 163
GNU make函数： 164
GNU make的自动化变量 165
GNU make环境变量 166
后序 166

关于本书

本文瑾献给所有热爱 Linux 的程序员！本文档版权所有，禁止用于任何商业行为。

本文比较完整的讲述 GNU make 工具，涵盖 GNU make 的用法、语法。同时重点讨论如何为一个工程编写 Makefile。作为一个 Linux 程序员，make 工具的使用以及编写 Makefile 是必需的。系统、详细讲述 make 的中文资料比较少，出于对广大中文 Linuxer 的支持，本人在工作之余，花了 18 个多月时间完成对“info make”的翻译整理，完成这个中文版的书册。本书并不是一个语言的翻译版本，其中对 GNU make 的一些语法和用法根据我个人的工作经验进行了一些详细分析和说明，也加入了一些个人的观点和实践总结。本书的所有的例子都可以在支持 V3.8 版本的 GNU make 的系统中正确执行。

由于个人水平限制，本文在一些地方存在描述不准确之处。恳请大家在阅读过程中，提出您宝贵的意见，也是对我个人的帮助。我的个人电子邮箱地址：xhbdahai@126.com。非常愿意和大家交流！共同学习。

阅读本书之前，应该对 GNU 的工具链和 Linux 中的一些常用工具有所了解。诸如：gcc、as、ar、ld、yacc 等。在书写 Makefile 时，需要能够理解一些 shell 脚本程序。这些工具是维护一个工程的基础工具。如果对这些工具的使用不是很熟悉，可参考相关资料。

阅读本文的几点建议：

1. 如果之前你对 GNU make 没有了解、而当前也不想深入的学习 GNU make 的读者。你只需要阅读本文各章前半部分的基础知识。
2. 如果你已经对 GNU make 比较熟悉，可能你更需要关心此版本的一些特点、新增的功能和存在的不兼容。
3. 之前你对 GNU make 没有多少概念、或者刚开始接触，本身又想成为一个 Linux 下的专业程序员，那么建议：完整学习本文的各个章节，包括了基础知识和高级用法、技巧。它会为你以后在 Linux 下的工程开发、工程管理提供非常有用的帮助。

谢谢！

徐海兵　　2005-12-31

第一章：概述

1.1 Make之前基本知识：

1.1.1 概述

Linux 环境下的程序员如果不会使用 GNU make 来构建和管理自己的工程，应该不能算是一个合格的专业程序员，至少不能称得上是 Linux 程序员。在 Linux 环境下使用 GNU 的 make 工具能够比较容易的构建一个属于你自己的工程，整个工程的编译只需要一个命令就可以完成编译、连接以至于最后的执行。不过这需要我们投入一些时间去完成一个或者多个称之为 Makefile 文件的编写。此文件正是 make 正常工作的基础。

所要完成的 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得可执行文件。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（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。

make 是一个命令工具，它解释 Makefile 中的指令（应该说是规则），Makefile 中描述了工程中所有文件的编译顺序、规则。Makefile 有自己的书写格式、关键字、函数。像 C 语言有自己的格式、关键字和函数一样。而且在 Makefile 中可以使用 shell 所提供的任何命令来完成你想要的工作。Makefile（在其它的系统上可能是另外的文件名）在绝大多数的 IDE 开发环境中都在使用，已经成为一种工程的编译方法。

目前，系统完整的介绍 make 工具和如何编写 Makefile 的中文文档比较少。我整理这个文档就是希望能使众多的 Linux 环境下的程序员能够比较容易的掌握和学会使用它。本文所要介绍的是 GNU 的 make，采用 Red Hat FC3（包括最新发布的 GNU Linux 系统）所集成的 GUN make 工具。make 版本信息如下：

本文后续所用到的所有例子都采用C语言书写的源程序，因为它是目前最普遍使用的一种语言。当然 make 工具不仅仅是用来管理 C 语言工程的，那些编译器只要能够在 shell 下运行的语言所构建的工程都可以使用 make 工具来管理。而且也可以使用 make 工具来做一些其它的事。例如，有这样的需求：当我们修改了某个或者某些文件后，需要能够根据修改的文件来自动对相关文件进行重建或者更新。那么应该考虑使用 GNU make 工具。GNU make 工具为我们实现这个目的提供了非常有利的支持。工程中根据源文件的修改情况来进行代码的编译正是使用了 make 的这个特征。make 执行时，根据 Makefile 的规则检查文件的修改情况，决定是否执行定义的动作（那些修改过的文件将会被重新编译）。这是 GNU make 的执行依据。

1.1.2 准备知识

在开始我们关于 make 的讨论之前，首先需要明确一些基本概念：

编译：把高级语言书写的代码转换为机器可识别的机器指令。编译高级语言后生成的指令虽然可被机器识别，但是还不能被执行。编译时，编译器检查高级语言的语法、函数与变量的声明是否正确。只有所有的语法正确、相关变量定义正确编译器就可以编译出中间目标文件。通常，一个高级语言的源文件都可对应一个目标文件。目标文件在 Linux 中默认后缀为“.o”（如“foo.c”的目标文件为“foo.o”）。

为了和规则的目标文件相区别。本文将编译高级语言后生成的目标文件成为.o 文件。

链接：将多.o 文件，或者.o 文件和库文件链接成为可被操作系统执行的可执行程序。链接器不检查函数所在的源文件，只检查所有.o 文件中的定义的符号。将.o 文件中使用的函数和其它.o 或者库文件中的相关符号进行合并，最后生成一个可执行的程序。“ld”是 GNU 的链接器。

静态库：又称为文档文件（Archive File）。它是多个.o 文件的集合。Linux 中静态库文件的后缀为“.a”。静态库中的各个成员（.o 文件）没有特殊的存在格式，仅仅是一个.o 文件的集合。使用“ar”工具维护和管理静态库。

共享库：也是多个.o 文件的集合，但是这些.o 文件时有编译器按照一种特殊的方式生成。对象模块的各个成员的地址（变量引用和函数调用）都是相对地址。因此在程序运行时，可动态加载库文件和执行共享的模块（多个程序可以共享使用库中的某一个模块）。

参考 ***info ld*** 了解更加详细的关于 ld 的说明和用法。

1.2 GNU make 介绍

make 在执行时，需要一个命名为 Makefile 的文件。这个文件告诉 make 以何种方式编译源代码和链接程序。典型地，可执行文件可由一些.o 文件按照一定的顺序生成或者更新。如果在你的工程中已经存在一个或者多个正确的 Makefile。当对工程中的若干源文件修改以后，需要根据修改来更新可执行文件或者库文件，正如前面提到的你只需要在 shell 下执行“make”。make 会自动根据修改情况完成源文件的对应.o 文件的更新、库文件的更新、最终的可执行程序的更新。

make 通过比较对应文件（规则的目标和依赖，）的最后修改时间，来决定哪些文件需要更新、那些文件不需要更新。对需要更新的文件 make 就执行数据库中所记录的相应命令（在 make 读取 Makefile 以后会建立一个编译过程的描述数据库。此数据库中记录了所有各个文件之间的相互关系，以及它们的关系描述）来重建它，对于不需要重建的文件 make 什么也不做。

而且可以通过make的命令行选项来指定需要重新编译的文件。参考 8.2 指定终极目标 一节

Problems and Bugs
==========
If you have problems with GNU `make' or think you've found a bug, please report it to the developers;

we cannot promise to do anything but we might well want to fix it.

Before reporting a bug, make sure you've actually found a real bug.Carefully reread the documentation and see if it really says you can do what you're trying to do. If it's not clear whether you should be able to do something or not, report that too; it's a bug in the documentation!

Before reporting a bug or trying to fix it yourself, try to isolate it to the smallest possible makefile that reproduces the problem. Then send us the makefile and the exact results `make' gave you, including any error or warning messages. Please don't paraphrase these messages: it's best to cut and paste them into your report. When generating this small makefile, be sure to not use any non-free or unusual tools in your commands: you can almost always emulate what such a tool would do with simple shell commands. Finally, be sure to explain what you expected to occur; this will help us decide whether the problem was really in the documentation.

Once you have a precise problem you can report it in one of two ways.Either send electronic mail to:

bug-make@gnu.org

or use our Web-based project management tool, at:

http://savannah.gnu.org/projects/make/

In addition to the information above, please be careful to include the version number of `make' you are using. You can get this information with the command `make --version'. Be sure also to include the type of machine and operating system you are using. One way to obtain this information is by looking at the final lines of output from the command

`make --help'.

以上时 GNU make 的 bug 反馈方式。如果在你使用 GNU make 过程中。发现 bug 或者问题。可以通过以上的方式和渠道反馈。

好了。开始我们的神奇之旅吧！

1.2.1 Makefile简介

在执行 make 之前，需要一个命名为 Makefile 的特殊文件（本文的后续将使用 Makefile 作为这个特殊文件的文件名）来告诉 make 需要做什么（完成什么任务），该怎么做。通常，make 工具主要被用来进行工程编译和程序链接。

本节将分析一个简单的 Makefile，它对一个包含 8 个 C 的源代码和三个头文件的工程进行编译和链接。这个 Makefile 提供给了 make 必要的信息，make 程序根据 Makefile 中的规则描述执行相关的命令来完成指定的任务（如：编译、链接和清除编译过程文件等）。复杂的 Makefile 我们将会在本文后续进行讨论。

当使用 make 工具进行编译时，工程中以下几种文件在执行 make 时将会被编译（重新编译）：

1. **所有的源文件没有被编译过，则对各个 C 源文件进行编译并进行链接，生成最后的可执行程序；**
2. **每一个在上次执行 make 之后修改过的 C 源代码文件在本次执行 make 时将会被重新编译；**

3. **头文件在上一次执行 make 之后被修改。则所有包含此头文件的 C 源文件在本次执行 make 时将会被重新编译。**

后两种情况是 make 只将修改过的 C 源文件重新编译生成.o 文件，对于没有修改的文件不进行任何工作。重新编译过程中，任何一个源文件的修改将产生新的对应的.o 文件，新的.o 文件将和以前的已经存在、此次没有重新编译的.o 文件重新连接生成最后的可执行程序。

首先让我们先来看一些 Makefile 相关的基本知识。

1.2.2 Makefile规则介绍

一个简单的 Makefile 描述规则组成：

TARGET... : PREREQUISITES...
COMMAND
...
...

target：规则的目标。通常是程序中间或者最后需要生成的文件名。可以是.o文件、也可以是最后的可执行程序的文件名。另外，目标也可以是一个make执行的动作的名称，如目标“clean”，成这样的目标是“伪目标”。参考3.6 Makefile伪目标 一节

prerequisites：规则的依赖。生成规则目标所需要的文件名列表。通常一个目标依赖于一个或者多个文件。

command：规则的命令行。是 make 程序所有执行的动作（任意的 shell 命令或者可在 shell 下执行的程序）。

一个规则可以有多个命令行，每一条命令占一行。**注意：每一个命令行必须以[Tab]字符开始，[Tab]字符告诉 make 此行是一个命令行。make 按照命令完成相应的动作。这也是书写 Makefile 中容易产生，而且比较隐蔽的错误。**

命令就是在任何一个目标的依赖文件发生变化后重建目标的动作描述。一个目标可以没有依赖而只有动作（指定的命令）。比如 Makefile 中的目标“clean”，此目标没有依赖，只有命令。它所指定的命令用来删除 make 过程产生的中间文件（清理工作）。

在 Makefile 中“规则”就是描述在什么情况下、如何重建规则的目标文件，通常规则中包括了目标的依赖关系（目标的依赖文件）和重建目标的命令。make 执行重建目标的命令，来创建或者重建规则的目标（此目标文件也可以是触发这个规则的上一个规则中的依赖文件）。**规则包含了目标和依赖的关系以及更新目标所要求的命令。**

Makefile 中可以包含除规则以外的部分。一个最简单的 Makefile 可能只包含规则描述。规则在有些 Makefile 中可能看起来非常复杂，但是无论规则的书写是多么的复杂，它都符合规则的基本格式。

1.2.3 简单的示例

本小节开始我们在第一小节中提到的例子。此例子由3个头文件和8个C文件组成。我们讲述写一个简单的Makefile，来描述如何创建最终的可执行文件“edit”，此可执行文件依赖于8个C源文件和3个头文件。Makefile文件的内容如下：

```
#sample Makefile
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
```

在书写时，一个较长行可以使用反斜线（\）分解为多行，这样做可以使Makefile清晰、容易阅读。**注意：反斜线之后不能有空格（这也是大家最容易犯的错误，而且错误比较隐蔽）。**大家在书写Makefile时，推荐者中将较长行分解为使用反斜线连接得多个行的方式。当我们完成了这个Maekfile以后；创建可执行程序“edit”，你所要做的就是在包含此Makefile的目录（当然也在代码所在的目录）下输入命令“make”。删除已经本目录下生成的文件和所有的.o文件，只需要输入命令“make clean”就可以了。

在这个Makefile中，目标（target）包含：可执行文件“edit”和.o文件（main.o,kbd.o…），依赖（prerequisites）就是冒号后面的那些 .c 文件和 .h文件。所有的.o文件既是依赖（相对于可执行程序edit）又是目标（相对于.c和.h文件）。命令包括“cc –c maic.c”、“cc –c kbd.c”……

目标是一个文件时，当它的任何一个依赖文件被修改以后，这个目标文件将会被重新编译或者重新连接。当然，此目标的任何一个依赖文件如果有必要则首先会被重新编译。在这个例子中，“edit”的依赖为8个.o文件；而“main.o”的依赖文件为“main.c”和“defs.h”。当“main.c”或者“defs.h”被修改以后，再次执行“make”时“main.o”就会被更新（其它的.o文件不会被更新），同时“main.o”

的更新将会导致“edit”被更新。

在描述目标和依赖之下的shell命令行，它描述了如何更新目标文件。命令行必需以[Tab]键开始，以和Makefile其他行区别。**就是说所有的命令行必需以[Tab] 字符开始，但并不是所有的以[Tab]键出现行都是命令行。但make程序会把出现在第一条规则之后的所有的以[Tab]字符开始的行都作为命令行来处理**。（要记住：make程序不关心命令是如何工作的，对目标文件的更新需要你在规则的描述中提供正确的命令。“make”程序所做的就是当目标程序需要更新时执行规则所定义的命令）。

目标“clean”不是一个文件，它仅仅代表了执行一个动作的标识。通常情况下，不需要执行这个规则所定义的动作，因此目标“clean”没有出现在其它规则的依赖列表中。在执行make时，它所指定的动作不会被执行。除非执行make时明确地指定它作为重建目标。而且目标“clean”没有任何依赖文件，它只有一个目的，就是通过这个目标名来执行它所定义的命令。**Makefile中把那些没有任何依赖只有执行动作的目标称为“伪目标”（phony targets）**。参考3.6 Makefile伪目标 一节。执行“clean”目标所定义的命令，可在shell下输入：make clean。

1.2.4 make如何工作

默认的情况下，make执行Makefile中的第一个规则，此规则的第一个目标称之为“最终目的”或者“终极目标”（就是一个Makefile最终需要更新或者创建的目标， 参考8.2 指定终极目标 一节）。

上例的 Makefile，目标“edit”在 Makefile 中是第一个目标，因此它就是 make 的“终极目标”。当修改了任何 C 源文件或者头文件后，执行 make 将会重建终极目标“edit”。

当在 shell 提示符下输入“make”命令以后。make 读取当前目录下的 Makefile 文件，并将 Makefile 文件中的第一个目标作为其“终极目标”，开始处理第一个规则（终极目标所在的规则）。在我们的例子中，第一个规则就是目标“edit”所在的规则。规则描述了“edit”的依赖关系，并定义了链接.o 文件生成目标“edit”的命令； make 在处理这个规则之前，首先将处理目标“edit”的所有的依赖文件（例子中的那些.o 文件）的更新规则；对包含这些.o 文件的规则进行处理。对.o 文件所在的规则的处理有下列三种情况：

1. 目标.o 文件不存在，使用其描述规则创建它；
2. 目标.o 文件存在，目标.o 文件所依赖的.c 源文件、.h 文件中的任何一个比目标.o 文件“更新”（在上一次 make 之后被修改）。则根据规则重新编译生成它；
3. 目标.o 文件存在，目标.o 文件比它的任何一个依赖文件（的.c 源文件、.h 文件）“更新”（它的依赖文件在上一次 make 之后没有被修改），则什么也不做。

这些.o 文件所在的规则之所以会被执行，是因为这些.o 文件出现在“终极目标”的依赖列表中。如果在 Makefile 中一个规则所描述的目标不是“终极目标”所依赖的（或者“终极目标”的依赖

文件所依赖的），那么这个规则将不会被执行。除非明确指定这个规则（可以通过 make 的命令行指定重建目标，那么这个目标所在的规则就会被执行，例如 “make clean”）。在编译或者重新编译生成一个.o 文件时，make 同样会去寻找它的依赖文件的重建规则（是这样一个规则：这个依赖文件在规则中作为目标出现），就是.c 和.h 文件的重建规则。在上例的 Makefile 中没有哪个规则的目标是.c 或者.h 文件，所以没有重建.c 和.h 文件的规则。不过 C 言语的源程序文件可以使用工具 Bison 或者 Yacc 来生成（具体用法可参考相应的手册）。

完成了对.o 文件的创建（第一次编译）或者更新之后，make 程序将处理终极目标“edit”所在的规则，分为以下三种情况：

1. 目标文件“edit”不存在，则执行规则创建目标“edit”。
2. 目标文件“edit”存在，其依赖文件中有一个或者多个文件比它“更新”，则根据规则重新链接生成“edit”。
3. 目标文件“edit”存在，它比它的任何一个依赖文件都“更新”，则什么也不做。

上例中，如果更改了源文件“insert.c”后执行 make，“insert.o”将被更新，之后终极目标“edit”将会被重生成；如果我们修改了头文件“command.h”之后运行“make”，那么“kbd.o”、“command.o”和“files.o”将会被重新编译，之后同样终极目标“edit”也将被重新生成。

以上我们通过一个简单的例子，介绍了 Makefile 中目标和依赖的关系。对于 Makefile 中的目标。在执行“make”时首先执行终极目标所在的规则，接下来一层层地去寻找终极目标的依赖文件所在的规则并执行。当终极目标的规则被完全的展开以后，make 将从最后一个被展开的规则处开始执行，之后处理倒数第二个规则，……依次回退。最后一步执行的就是终极目标所在的规则。整个过程就类似于 C 语言中的递归实现一样。在更新（或者创建）终极目标的过程中，如果出现错误 make 就立即报错并退出。整个过程 make 只是负责执行规则，而对具体规则所描述的依赖关系的正确性、规则所定义的命令的正确性不做任何判断。就是说，一个规则的依赖关系是否正确、描述重建目标的规则命令行是否正确，make 不做任何错误检查。

因此，需要正确的编译一个工程。需要在提供给 make 程序的 Makefile 中来保证其依赖关系的正确性、和执行命令的正确性。

1.2.5 指定变量

同样是上边的例子，我们来看一下终极目标“edit”所在的规则：

```
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
```

在这个规则中.o 文件列表出现了两次；第一次：作为目标“edit”的依赖文件列表出现，第二

次：规则命令行中作为“cc”的参数列表。这样做所带来的问题是：如果我们需要为目标“edit”增加一个的依赖文件，我们就需要在两个地方添加（依赖文件列表和规则的命令中）。添加时可能在“edit”的依赖列表中加入了、但却忘记了给命令行中添加，或者相反。这样给后期的维护和修改带来了很多不方便，而且容易出现修改遗漏。

为了避免这个问题，在实际工作中大家都比较认同的方法是，使用一个变量“objects”、“OBJECTS”、“objs”、“OBJS”、“obj”或者“OBJ”来作为所有的.o 文件的列表的替代。在使用到这些文件列表的地方，使用此变量来代替。在上例的 Makefile 中可是添加这样一行：

```
objects = main.o kbd.o command.o display.o \
        insert.o search.o files.o utils.o
```

“objects”作为一个变量，它代表所有的.o文件的列表。在定义了此变量后，我们就可以在需要使用这些.o文件列表的地方使用“$(objects)”来表示它，而不需要罗列所有的.o文件列表（关于变量可参考 5 使用变量 一章）。因此上例的规则就可以这样写：

```
objects = main.o kbd.o command.o display.o \
            insert.o search.o files.o utils.o
edit : $(objects)
      cc -o edit $(objects)
      .......
      .......
clean :
      rm edit $(objects)
```

需要增加或者去掉一个.o 文件时。我们只需要改变“objects”的定义（加入或者去掉若干个.o 文件）。这样做不但减少维护的工作量，而且可以避免由于遗漏而产生错误的可能。

1.2.6 自动推导规则

在使用make编译.c源文件时，可以省略编译一个.c文件所使用的命令。这是因为make存在一个默认的规则，能够自动完成对.c文件的编译并生成对应的.o文件。它执行命令“cc -c”来编译.c源文件。对于上边的例子，此默认规则就使用命令“cc -c main.c -o main.o”来创建文件“main.o”。因此对一个目标文件是“N.o”，倚赖文件是“N.c”的规则。可以省略其规则的命令行，使用make的默认命令。此默认规则称为make的隐含规则（关于隐含规则可参考 9 使用隐含规则 一章）

我们书写 Makefile 时，对于一个.c 文件如果使用 make 的隐含规则，那么它会被自动作为对应.o 文件的一个依赖文件（对应是指：文件名除后缀外，其余都相同的两个文件）。因此我们也可以在规则中省略目标的倚赖.c 文件。

我们上边的例子就可以以更加简单的方式书写，使用了变量“objects”。简化版本的 Makefile

如下：

```
# sample Makefile
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)
```

这种格式的Makefile更接近于我们实际的应用。（关于目标“clean”的详细说明我们在后边。参考3.6 Makefile伪目标 一节 和 4.4 命令的错误 一节）

make 的隐含规则在实际工程的 make 中会经常使用，它使得编译过程变得方便。几乎在所有的 Makefile 中都用到了 make 的隐含规则，make 的隐含规则是非常重要的一个概念。后续我们会在第九章会专门讨论 make 的隐含规则。

1.2.7 另类风格的makefile

Makefile 中，目标使用隐含规则生成，我们就可以也可以书写另外一种风格 Makefile。在这个 Makefile 中，根据依赖而不是目标对规则进行分组。上例的 Makefile 就可以这样来实现：

```
#sample Makefile
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
```

例子中头文件“defs.h”作为所有.o 文件的依赖文件。其它两个头文件作为其对应规则的目标中所列举的所有.o 文件的依赖文件。

但是这种风格的 Makefile 并不值得我们借鉴。问题在于：同时把多个目标文件的依赖放在同

一个规则中进行描述（一个规则中含有多个目标文件），这样导致规则定义不明了，比较混乱。建议大家不要在 Makefile 中采用这种方式了书写。否则后期维护将会是一件非常痛苦的事情。

书写规则建议的方式是：单目标，多依赖。就是说尽量要做到一个规则中只存在一个目标文件，可有多个依赖文件。尽量避免多目标，单依赖的方式。这样后期维护也会非常方便，而且 Makefile 会更清晰、明了。

1.2.8 清除工作目录过程文件

在 Makefile 中的规则也可以完成除编译以外的任务。例如：前边提到的实现清除当前目录中在编译过程中生成的文件（edit 和哪些.o 文件）的规则：

```
clean :
    rm edit $(objects)
```

在实际应用时，我们会把这个规则写成如下稍微复杂一些的样子。以防止出现始料未及的情况。

```
.PHONY : clean
clean :
    -rm edit $(objects)
```

这两个实现有两点不同： 1. 通过“.PHONY”特殊目标将“clean”目标声明为伪目标。防止当磁盘上存在一个名为“clean”文件时，“clean”所在规则的命令无法执行（参考 3.6 Makefile 伪目标 一节）。2. 在命令行之前使用“-”，意思是忽略命令“rm”的执行错误（参考 4.4 命令的错误 一节）。

这样的一个目标在 Makefile 中，不能将其作为终极目标（Makefile 的第一个目标）。因为我们的初衷并不是当你在命令行上输入 make 以后执行删除动作。而是要创建或者更新程序。在我们上边的例子中。就是在输入 make 以后要需要对目标“edit”进行创建或者重建。

上例中因为目标“clean”没有出现在终极目标“edit”依赖关系中，所以我们执行“make”时，目标“clean”所在的规则将不会被处理。如果需要执行此规则，需要在make的命令行选项中明确指定这个目标（执行“make clean”）。关于make的执行可参考 8.2 指定终极目标 一节。

第二章：Makefile 总述

2 Makefile概述

2.1 Makefile的内容

在一个完整的 Makefile 中，包含了 5 个东西：显式规则、隐含规则、变量的定义、指示符和

注释。关于“规则”、“变量”和“Makefile 指示符”将在后续的章节进行详细的讨论。本章讨论的是一些基本概念。

- 显式规则：它描述了在何种情况下如何更新一个或者多个被称为目标的文件（Makefile 的目标文件）。在书写 Makefile 是需要明确地给出目标文件、目标的依赖文件列表以及更新目标文件所需要的命令。
- 隐含规则：它是make根据此类目标文件的命名（典型的是文件名的后缀）而自动推导出来的规则。make根据目标文件的名字，自动产生目标的依赖文件并使用默认的命令来对目标进行更新。关于隐含规则可参考 第九章 make的隐含规则
- 变量定义：就是使用一个字符串代表一段文本串，当定义了变量以后，Makefile后续在需要使用此文本串的地方，通过引用这个变量来实现对文本串的使用。第一章的的例子中，我们就定义了一个变量“objects”来表示一个.o文件列表。关于变量的详细讨论可参考 第五章 Makefile中的变量
- Makefile 指示符：指示符指明在 make 程序读取 makefile 文件过程中所要执行的一个动作。其中包括：
  - 读取一个文件，读取给定文件名的文件。参考 2.3 包含其它makefile文件一节
  - 决定（通常是根据一个变量的得值）处理或者忽略Makefile中的某一特定部分。参考 第六章Makefile的条件执行
  - 定义一个多行变量。参考 5.8 多行定义 一节
- 注释：Makefile 中“#”字符后的内容被作为是注释内容（和 shell 脚本一样）处理。如果此行的第一个非空字符为“#”，那么此行为注释行。注释行的结尾如果存在反斜线（\），那么下一行也被作为注释行。一般在书写 Makefile 时推荐将注释作为一个独立的行，而不要和 Makefile 的有效行放在一行中书写。当在 Makefile 中需要使用字符“#”时，可以使用反斜线加“#”（\#）来实现，其表示将“#”作为一字符而不是注释的开始标志。

需要注意的地方：

Makefile 中第一个规则之后的所有以[Tab]字符开始的的行，make 程序都会将其给系统的 shell 程序去解释执行。因此以[Tab]字符开始的注释行也会被交给 shell 来处理，此命令行是否需要被执行（shell 执行或者忽略）是由系统 shell 程序来判决的。

另外，在使用指示符“define”定义一个多行的变量或者命令包时，其定义体（“define”和“endef”之间的内容）会被完整的展开到 Makefile 中引用此变量的地方（包含定义体中的注释行）；make 在引用此变量的地方对所有的定义体进行处理，决定是注释还是有效内容。Makefile 中的变量可以和 C 语言中的宏（实质一样）一样来理解。对一个变量引用的地方 make 所做的就是将这个变量根据定义进行基于文本的展开，展开变量的过程不涉及到任何变量的具体含义和功能分析。

2.2 makefile文件的命名

默认的情况下，make 会在工作目录（执行 make 的目录）下按照文件名顺序寻找 makefile 文件读取并执行，查找的文件名顺序为：“GNUmakefile”、“makefile”、“Makefile”。

通常应该使用“makefile”或者“Makefile”作为一个 makefile 的文件名（我们推荐使用“Makefile”，首字母大写而比较显著，一般在一个目录中和当前目录的一些重要文件（README,Chagelist 等）靠近，在寻找时会比较容易的发现它）。而“GNUmakefile”是我们不推荐使用的文件名，因为以此命名的文件只有“GNU make”才可以识别，而其他版本的 make 程序只会在工作目录下“makefile”和“Makefile”这两个文件。

如果make程序在工作目录下无法找到以上三个文件中的任何一个，它将不读取任何其他文件作为解析对象。但是根据make隐含规则的特性，我们可以通过命令行指定一个目标，如果当前目录下存在符合此目标的依赖文件，那么这个命令行所指定的目标将会被创建或者更新，参见注释。（详细可参考 第九章 make的隐含规则）

当 makefile 文件的命名不是这三个任何一个时，需要通过 make 的“-f”或者“--file”选项来指定 make 读取的 makefile 文件。给 make 指定 makefile 文件的格式为：“-f NAME”或者“—file= NAME”，它指定文件“NAME”作为执行 make 时读取的 makefile 文件。也可以通过多个“-f”或者“--file”选项来指定多个需要读取的 makefile 文件，多个 makefile 文件将会被按照指定的顺序进行连接并被 make 解析执行。当通过“-f”或者“--file”指定 make 读取 makefile 的文件时，make 就不再自动查找这三个标准命名的 makefile 文件。

注释：通过命令指定目标使用make的隐含规则：

当前目录下不存在以“GNUmakefile”、“makefile”、“Makefile”命名的任何文件，

1. **当前目录下存在一个源文件foo.c的，我们可以使用“make foo.o”来使用make的隐含规则自动生成foo.o。当执行“make foo.o”时。我们可以看到其执行的命令为：**

   cc –c –o foo.o foo.c

   之后，foo.o将会被创建或者更新。

2. **如果当前目录下没有foo.c文件时，就是make对.o文件目标的隐含规则中依赖文件不存在。如果使用命令“make foo.o”时，将回到到如下提示：**

   make: * No rule to make target ‘foo.o’. Stop.**

3. **如果直接使用命令“make”时，得到的提示信息如下：**

   make: * No targets specified and no makefile found. Stop.**

2.3 包含其它makefile文件

本节我们讨论如何在一个 Makefile 中包含其它的 makefile 文件。Makefile 中包含其它文件的关键字是“include”，和 C 语言对头文件的包含方式一致。

“include”指示符告诉 make 暂停读取当前的 Makefile，而转去读取“include”指定的一

个或者多个文件，完成以后再继续当前 Makefile 的读取。Makefile 中指示符“include”书写在独立的一行，其形式如下：

include FILENAMES...

FILENAMES 是 shell 所支持的文件名（可以使用通配符）。

指示符“include”所在的行可以一个或者多个空格（make程序在处理时将忽略这些空格）开始，切忌不能以[Tab]字符开始（如果一行以[Tab]字符开始make程序将此行作为一个命令行来处理）。指示符“include”和文件名之间、多个文件之间使用空格或者[Tab]键隔开。行尾的空白字符在处理时被忽略。使用指示符包含进来的Makefile中，如果存在变量或者函数的引用。它们将会在包含它们的Makefile中被展开（详细可参考 第五章 Makefile中的变量）。

来看一个例子，存在三个.mk 文件，“$(bar)”被扩展为“bish bash”。则

include foo *.mk $(bar)

等价于

include foo a.mk b.mk c.mk bish bash

make 程序在处理指示符 include 时，将暂停对当前使用指示符“include”的 makefile 文件的读取，而转去依此读取由“include”指示符指定的文件列表。直到完成所有这些文件以后再回过头继续读取指示符“include”所在的 makefile 文件。

通常指示符“include”用在以下场合：

1. 有多个不同的程序，由不同目录下的几个独立的Makefile来描述其创建或者更新规则。它们需要使用一组通用的变量定义（可参考 5.5 如何设置变量 一节）或者模式规则（可参考 9.5 模式规则 一节）。通用的做法是将这些共同使用的变量或者模式规则定义在一个文件中（没有具体的文件命名限制），在需要使用的Makefile中使用指示符“include”来包含此文件。
2. 当根据源文件自动产生依赖文件时；我们可以将自动产生的依赖关系保存在另外一个文件中，主Makefile使用指示符“include”包含这些文件。这样的做法比直接在主Makefile中追加依赖文件的方法要明智的多。其它版本的make已经使用这种方式来处理。（参考 3.14 自动产生依赖 一节）

如果指示符“include”指定的文件不是以斜线开始（绝对路径，如/usr/src/Makefile...），而且当前目录下也不存在此文件；make将根据文件名试图在以下几个目录下查找：首先，查找使用命令行选项“-I”或者“--include-dir”（参考 8.7make的命令行选项 一节）指定的目录，如果找到指定的文件，则使用这个文件；否则依此搜索以下几个目录（如果其存在）：“/usr/gnu/include”、“/usr/local/include”和“/usr/include”。

当在这些目录下都没有找到“include”指定的文件时，make将会提示一个包含文件未找到的告警提示，但是不会立刻退出。而是继续处理Makefile的内容。当完成读取所有的makefile文件后，make将试图使用规则来创建通过指示符“include”指定的但未找到的文件（参考 2.7 makefile文件的重建 一节），当不能创建它时（没有创建这个文件的规则），make将提示致命错误并退出。会输出类似如下错误提示：

Makefile:错误的行数：未找到文件名：提示信息（No such file or directory）
Make： *** No rule to make target '<filename>'. Stop

我们可使用“-include”来代替“include”，忽略由于包含文件不存在或者无法创建时的错误提示（“-”的意思是告诉 make，忽略此操作的错误。make 继续执行）。像下边那样：

-include FILENAMES...

使用这种方式时，当所要包含的文件不存在时不会有错误提示、make 也不会退出；除此之外，和第一种方式效果相同。以下是**这两种方式的比较：**

使用“***include FILENAMES...***”，make 程序处理时，如果“***FILENAMES***”列表中的任何一个文件不能正常读取而且不存在一个创建此文件的规则时 make 程序将会提示错误并退出。

使用“***-include FILENAMES...***”的情况是，当所包含的文件不存在或者不存在一个规则去创建它，make 程序会继续执行，只有在因为 makefile 的目标的规则不存在时，才会提示致命错误并退出。

为了和其它的 make 程序进行兼容。也可以使用“sinclude”来代替“-include”（GNU 所支持的方式）。

2.4 变量 MAKEFILES

如果当前环境定义了一个“***MAKEFILES***”的环境变量，make执行时首先将此变量的值作为需要读入的Makefile文件，多个文件之间使用空格分开。类似使用指示符“include”包含其它Makefile文件一样，如果文件名非绝对路径而且当前目录也不存在此文件，make会在一些默认的目录去寻找（参考 2.3 包含其它makefile文件 一节）。此情况和使用“include”的区别：

1. 环境变量指定的 makefile 文件中的“目标”不会被作为 make 执行的“终极目标”。就是说，这些文件中所定义规则的目标，make 不会将其作为“终极目标”来看待。如果在 make 的工作目录下没有一个名为“Makefile”、“makefile”或者“GNUmakefile”的文件，make 同样会提示“***make: *** No targets specified and no makefile found. Stop.***”；而在 make 的工作目录下存在这样一个文件（“Makefile”、“makefile”或者“GNUmakefile”），那么 make 执行时的“终极目标”就是当前目录下这个文件中所定

义的“终极目标”。

2. 环境变量所定义的文件列表，在执行 make 时，如果不能找到其中某一个文件（不存在或者无法创建）。make 不会提示错误，也不退出。就是说环境变量“***MAKEFILES***”定义的包含文件是否存在不会导致 make 错误（这是比较隐蔽的地方）。
3. make 在执行时，首先读取的是环境变量“***MAKEFILES***”所指定的文件列表，之后才是工作目录下的 makefile 文件，“include”所指定的文件是在 make 发现此关键字的时、暂停正在读取的文件而转去读取“include”所指定的文件。

变量“MAKEFILES”主要用在“make”的递归调用过程中的的通信（参考4.6 make的递归执行 一节）。实际应用中很少设置此变量。一旦设置了此变量，在多层make调用时；由于每一级make都会读取“MAKEFILES”变量所指定的文件，这样可能导致执行的混乱（可能不是你想看到的执行结果）。不过，我们可以使用此环境变量来指定一个定义通用的“隐含规则”和用的变量的文件，比如设置默认搜索路径（可参考3.5 目录搜索 一节）；通过这种方式设置的“隐含规则”和定义的变量可以被任何make进程使用（有点象C语言中的全局变量）。

也有人想让 login 程序自动的在自己的工作环境中设置此环境变量，编写的 Makefile 建立在此环境变量的基础上。此想法可以肯定地说不是一个好主意。规劝大家千万不要这么干，否则你所编写的 Makefile 在其人的工作环境中肯定不能正常工作。因为别人的工作环境中可能没有设置相同的环境变量“MAKEFILES”。

推荐的做法实：在需要包含其它 makefile 文件时使用指示符“include”来实现。

2.5 变量 MAKEFILE_LIST

make 程序在读取多个 makefile 文件时，包括由环境变量“MAKEFILES”指定、命令行指、当前工作下的默认的以及使用指示符“include”指定包含的，在对这些文件进行解析执行之前make读取的文件名将会被自动的追加到变量“MAKEFILE_LIST”的定义域中。

这样我们就可以通过测试此变量的最后一个字来得知当前 make 程序正在处理的是具体的那个 makefile 文件。具体地说就是一个 makefile 文件中当使用指示符“include”包含另外一个文件之后，变量“MAKEFILE_LIST”的最后一个只可能是指示符“include”指定所要包含的那个文件的名字。如果一个 makefile 的内容如下：

```
name1 := $(word $(words $(MAKEFILE_LIST)),$(MAKEFILE_LIST))

include inc.mk

name2 := $(word $(words $(MAKEFILE_LIST)),$(MAKEFILE_LIST))

all:
    @echo name1 = $(name1)
    @echo name2 = $(name2)
```

执行 make，则看到的将是如下的结果：

name1 = Makefile
name2 = inc.mk

此例子中涉及到了 make 的函数的和变量定义的方式，这些将在后续的章节中有详细的讲述。

2.6 其他特殊变量

GNU make支持一个特殊的变量，此变量不能通过任何途经给它赋值。此变量展开以后是一个特定的值。第一个重要的特殊的变量是“.VARIABLES”。它被展开以后是此引用点之前、makefile文件中所定义的所有全局变量列表。包括：空变量（未赋值的变量）和make的内嵌变量（参考9.3 隐含变量 一节），但不包含目标指定的变量，目标指定变量值在特定目标的上下文有效。关于目标变量可参考 5.10 目标指定变量 一节

2.7 makefile文件的重建

有时，Makefile 可由其它文件生成，比如 RCS 或 SCCS 文件。如果 Makefile 由其它文件重建，那么在 make 开始解析 Makefile 时需要读取的是更新后的 Makefile、而不是那个没有更新的 Makefile。make 的处理过程是这样的：

make 在读入所有 makefile 文件之后，首先将所读取的每个 makefile 作为一个目标，试着去更新它。如果存在一个更新特定 makefile 文件明确规则或者隐含规则，则去更新这个 makefile 文件。在完成对所有的 makefile 文件的更新检查动作之后，如果之前所读取的 makefile 文件已经被更新，那么 make 就清除本次执行的状态重新读取一遍所有的 makefile 文件（此过程中，同样在读取完成以后也会去试图更新所有的已经读取的 makefile 文件，但是一般这些文件不会再次被重建，因为它们在时间戳上已经是最新的）。

实际应用中，我们会很明确的了解我们的那些makefile文件不需要重建。出于make效率的考虑，我们可以采用一些办法来避免make在执行过程时查找重建makefile的隐含规则。例如我们可以书写一个明确的规则，将makefile文件作为目标，命令为空。（参考 4.8 空命令 一节）

Makefile规则中，如果使用一个没有依赖只有命令行的双冒号规则去更新一个文件，那么每次执行make时，此规则的目标文件将会被无条件的更新。而假如此规则的目标文件是一个makefile文件，那么在执行make时，将会导致这个makefile文件被无条件更新，时make的执行陷入到一个死循环中（此makefile文件被不断的更新、重新读取、更新再重新读取的过程）。为了防止进入此循环，make在遇到一个目标是makefile文件的双冒号规则时，将忽略对这个规则的执行（其中包括了使用“MAKEFILES”指定、命令行选项指定、指示符“include”指定的需要make读取的所有makefile文件中定义的这一类双冒号规则）。

执行 make 时，如果没有使用“-f（--file）”选项指定一个文件，make 程序将读取缺省的文件。和使用“-f（--file）”选项不同，make 无法确定工作目录下是否存在缺省名称的 makefile 文件。如果缺省 makefile 文件不存在，但可以通过一个规则来创建它（此规则是隐含规则），则

会自动创建缺省 makefile 文件，之后重新读取它并开始执行。

因此，如果不存在缺省 makefile 文件，make 将按照搜索 makefile 文件的名称顺序去创建它，直到创建成功或者超越其缺省的命名顺序。需要明确的一点是：执行 make 时，如果不能成功地创建其缺省的 makefile 文件，并不一定会导致错误。运行 make 时一个 makefile 文件并不是必需的。（关于这一点大家会在后续的阅读过程中体会到）

当使用“-t（--touch）”选项来对 Makefile 目标文件进行时间戳更新时，对于哪些 makefile 文件的目标是无效的。就是说即使执行 make 时使用了选项“-t”，那些目标是 makefile 文件的规则同样也会被 make 执行（而其它的规则不会被执行，make 只是简单的更新规则目标文件的时间戳）；类似还有选项“-q（—question）”和“-n（—just-print）”，这主要是因为一个过时的 makefile 文件对其它目标的重建规则在当前看来可能是错误的。正因为如此，执行命令“make –f mfile –n foo”首先会试图重建“mfile 文件”、并重新读取它，之后会打印出更新目标“foo”规则中所定义的命令但不执行此命令。

在这种情况下，如果我们不希望重建 makefile 文件。那么我们就需要在执行 make 时，在命令行中将这个 makefile 文件中为一个最终目的，这样“–t”和其它的选项就对这个 makefile 文件的目标有效，防止执行这个 makefile 作为目标的规则。同样，命令“make –f mfile –n mfile foo”会读取文件“mfile”，打印出重建文件“mfile”的命令、重建“foo”的命令而实际不去执行此命令。并且所打印的用于更新“foo”目标的命令是选项“-f”指定的、没有被重建的“mfile”文件中所定义的命令。

2.8 重载另外一个makefile

有些情况下存在两个比较类似的 makefile 文件。其中一个（makefile-A）需要使用另外一个文件（makefile-B）中所定义的变量和规则。我们可以在“makefile-A”中使用指示符“include”来包含“mkaefile-B”来达到目的，这种情况下，如果两个 makefile 文件中存在相同目标，而其描述规则中使用不同的命令。相同目标有两个不同的规则命令，这是 makefile 所不允许的。遇到这种情况，使用指示符“include”显然是行不通的。GNU make 提供另外一种途径来达到此目的。具体的做法如下：

在需要包含的makefile文件（makefile-A）中，我们可以使用一个称之为“所有匹配模式”（参考 9.5 模式规则 一节）的规则来描述在“makefile-A”中没有明确定义的目标，make将会在给定的makefile文件中寻找没有在当前Makefile中给出的目标更新规则。

看一个例子，如果存在一个命名为“Makefile”的 makefile 文件，其中描述目标“foo”的规则和其他的一些规，我们也可以书写一个内容如下命名为“GNUmakefile”的文件。

```
#sample GNUmakefile
foo:
```

```
        frobnicate > foo

%: force
    @$(MAKE) -f Makefile $@
force: ;
```

执行命令“make foo”，make 将使用工作目录下命名为“GNUmakefile”的文件并执行目标“foo”所在的规则，创建它的命令是：“frobnicate > foo”。如果我们执行另外一个命令“make bar”， “GUNmakefile”中没有此目标的更新规则。那么，make 将会使用“所有匹配模式”规则，执行命令“$(MAKE) -f Makefile bar”。如果文件“Makefile”中存在此目标更新规则的定义，那么这个规则会被执行。此过程同样适用于其它 “GNUmakefile”中没有给出的目标更新规则。此方式的灵活之处在于：如果在“Makefile”文件中存在同样一一个目标“foo”的重建规则，由于 make 执行时首先读取文件“GUNmakefile”并在其中能够找到目标“foo”的重建规则，所以 make 就不会去执行这个“所有模式匹配规则”（上例中的目标是“%”的规则）。这样就避免了使用指示符“include”包含一个 makefile 文件时所带来的目标规则的重复定义问题。

此种方式，模式规则的模式只使用了单独的“%”（我们才称他为“所有模式匹配规则”），它可以匹配任何一个目标；它的依赖是“force”，保证了即使目标文件已经存在也会执行这个规则（文件已存在时，需要根据它的依赖文件的修改情况决定是否需要重建这个目标文件）；“force”规则中使用空命令是为了防止 make 程序试图寻找一个规则去创建目标“force”时，又使用了模式规则“%: force”而陷入无限循环。

2.9 make如何解析makefile文件

GUN make 的执行过程分为两个阶段。

第一阶段：读取所有的 makefile 文件（包括“MAKIFILES”变量指定的、指示符“include”指定的、以及命令行选项“-f(--file)”指定的 makefile 文件），内建所有的变量、明确规则和隐含规则，并建立所有目标和依赖之间的依赖关系结构链表。

在第二阶段：根据第一阶段已经建立的依赖关系结构链表决定哪些目标需要更新，并使用对应的规则来重建这些目标。

理解 make 执行过程的两个阶段是很重要的。它能帮助我们更深入的了解执行过程中变量以及函数是如何被展开的。变量和函数的展开问题是书写 Makefile 时容易犯错和引起大家迷惑的地方之一。本节将对这些不同的结构的展开阶段进行简单的总结（明确变量和函数的展开阶段，对正确的使用变量非常有帮助）。首先，明确以下基本的概念：在 make 执行的第一阶段中如果变量和函数被展开，那么称此展开是“立即”的，此时所有的变量和函数被展开在需要构建的结构链表的对应规则中（此规则在建立链表是需要使用）。其他的展开称之为“延后”的。这些变量和函数不会被“立即”展开，而是直到后续某些规则须要使用时或者在 make 处理的第二阶段它们才会被展开。

可能现在讲述的这些还不能完全理解。不过没有关系，通过后续章节内容的学习，我们会一步一步的熟悉 make 的执行过程。学习过程中可以回过头来参考本节的内容。相信在看完本书之后，会对 make 的整个过程有全面深入的理解。

2.9.1 变量取值

变量定义解析的规则如下：

```
IMMEDIATE = DEFERRED
IMMEDIATE ?= DEFERRED
IMMEDIATE := IMMEDIATE
IMMEDIATE += DEFERRED or IMMEDIATE
define IMMEDIATE
        DEFERRED
Endef
```

当变量使用追加符（+ = ）时，如果此前这个变量是一个简单变量（使用 := 定义的）则认为它是立即展开的，其它情况时都被认为是“延后”展开的变量。

2.9.2 条件语句

所有使用到条件语句在产生分支的地方，make 程序会根据预设条件将正确地分支展开。就是说条件分支的展开是“立即”的。其中包括：“ifdef”、“ifeq”、“ifndef”和“ifneq”所确定的所有分支命令。

2.9.3 规则的定义

所有的规则在 make 执行时，都按照如下的模式展开：

```
IMMEDIATE : IMMEDIATE ; DEFERRED
DEFERRED
```

其中，规则中目标和依赖如果引用其他的变量，则被立即展开。而规则的命令行中的变量引用会被延后展开。此模板适合所有的规则，包括明确规则、模式规则、后缀规则、静态模式规则。

2.10 总结

make 的执行过程如下：

1. 依次读取变量“MAKEFILES”定义的 makefile 文件列表
2. 读取工作目录下的 makefile 文件（根据命名的查找顺序“GNUmakefile”，“makefile”，“Makefile”，首先找到那个就读取那个）

3. 依次读取工作目录 makefile 文件中使用指示符“include”包含的文件
4. 查找重建所有已读取的 makefile 文件的规则（如果存在一个目标是当前读取的某一个 makefile 文件，则执行此规则重建此 makefile 文件，完成以后从第一步开始重新执行）
5. 初始化变量值并展开那些需要立即展开的变量和函数并根据预设条件确定执行分支
6. 根据“终极目标”以及其他目标的依赖关系建立依赖关系链表
7. 执行除“终极目标”以外的所有的目标的规则（规则中如果依赖文件中任一个文件的时间戳比目标文件新，则使用规则所定义的命令重建目标文件）
8. 执行“终极目标”所在的规则

说明：

执行一个规则的过程是这样的：

对于一个存在的规则（明确规则和隐含规则）首先，make程序将比较目标文件和所有的依赖文件的时间戳。如果目标的时间戳比所有依赖文件的时间戳更新（依赖文件在上一次执行make之后没有被修改），那么什么也不做。否则（依赖文件中的某一个或者全部在上一次执行make后已经被修改过），规则所定义的重建目标的命令将会被执行。这就是make工作的基础，也是其执行规制所定义命令的依据。（后续讨论规则时将会对此详细地说明）

第三章：Makefile的规则

3 Makefile规则

本章我们开始讨论 Makefile 的一个重要内容，Makefile 的规则。

Makefile 中，规则描述了何种情况下使用什么命令来重建一个特定的文件，此文件被称为规则“目标”（通常规则中的目标只有一个）。规则所罗列的其他文件称为“目标”的依赖，而规则中的命令是用来更新或者创建此规则的目标。

除了 makefile 的“终极目标”所在的规则以外，其他规则的顺序在 makefile 文件中没有意义。“终极目标”就是当没有使用 make 命令行指定具体目标时，make 默认的哪一个目标。它是 makefile 文件中第一个规则的目标。如果在 makefile 中第一个规则有多个目标的话，那么多个目标中的第一个将会被作为 make 的“终极目标”。有两种情况的例外：1. 目标名是以点号“.”开始的其后不存在斜线“/”（“./”被认为是当前目录；“../”被认为是上一级目录）；2. 作为模式规则的目标。此两种情况的 Makefile 的第一个目标都不会被作为“终极目标”来对待。

“终极目标”是执行 make 的唯一目的，其所在的规则作为第一个规则。而其他的规则是在完成重建“终极目标”的过程中被连带出来的。所以这些目标所在规则在 Makefile 中的顺序无关紧要。

因此，我们书写的 makefile 的第一个规则应该就是重建整个程序或者多个程序的依赖关系和执行命令的描述。

3.1 一个例子

我们来看一个规则的例子：

```
foo.o : foo.c defs.h          # module for twiddling the frobs
        cc -c -g foo.c
```

这是一个典型的规则。看到这个例子，大家也许能够说出这个规则的各个部分之间的关系。不过我们还是要把这个例子拿出来讨论。目的是让我们更加明确的理解 Makefile 的规则。本例第一行中，文件“foo.o”是规则需要重建的文件，而“foo.c”和“defs.h”是重建“foo.o”所要使用的文件。我们把规则所需要重建的文件称为规则的“目标”（foo.o），而把重新目标所需要的文件称为“目标”的“依赖”。规则中的第二行“cc -c -g foo.c”就是规则的“命令”。它描述了如何使用规则中的依赖文件重建目标。

而且，上面的规则告诉我们了两件事：

1. 如何确定目标文件是否过期（需要重建目标），过期是指目标文件不存在或者目标文件“foo.o”在时间戳上比依赖文件中的任何一个“foo.c”或者“defs.h”“老”。
2. 如何重建目标文件“foo.o”。这个规则中使用 cc 编译器。在命令中没有明确的使用到依赖文件“defs.h”。我们假设在源文件“foo.c”中已经包含了此头文件。这也是为什么它作为目标依赖出现的原因。

3.2 规则语法

通常规则的语法格式如下：

```
TARGETS : PREREQUISITES
    COMMAND
    ...
```

或者是这样：

```
TARGETS : PREREQUISITES ; COMMAND
    COMMAND
    ...
```

规则中“***TARGETS***”可以是空格分开的多个文件名，也可以是一个标签（执行清空的“clean”）。“***TARGETS***”的文件名可以使用通配符，格式“A(M)”表示档案文件（Linux下的静态库.a文件）的成员“M”（关于静态库的重建可参考 第十章 使用make更新静态库文件）。通常规则只有一个目标文件（建议这么做），偶尔会在一个规则中需要多个目标（可参考 3.10 多目标 一节）。

书写规则是我们需要注意的几点：

1. 规则的命令部分有两种书写方式：a. 命令可以和目标：依赖描述放在同一行。命令在依赖文件列表后并使用分号（；）和依赖文件列表分开。b. 命令在目标：依赖的描述的下一行，作为独立的命令行。当作为独立的命令行时此行必须以[Tab]字符开始。在 Makefile 中，在第一个规则之后出现的所有以[Tab]字符开始的行都会被当作命令来处理。
2. Makefile 中对“$”有特殊的含义（表示变量或者函数的引用），如果我们的规则如果需要“$”，需要书写两个连续的（“$$”）。
3. 在前边我们也提到过，Makefile 一个较长的行，可以使用反斜线“\”将其书写到几个独立的物理行上。虽然 make 对 Makefile 文本行的最大长度是没有限制的，但是还是建议这样做。不仅书写方便而且更有利于别人的阅读（这也是一个程序员修养的体现）。

一个规则告诉“make”两件事：1. 目标在什么情况下已经过期； 2. 在需要重建目标的时候，怎么样去重建这个目标。目标是否过期是由那些使用空格分开的规则的依赖文件所决定的。当目标文件不存在或者目标文件的最后修改时间比依赖文件中的任何一个都晚，则目标就会被创建或者重建。也就是说执行规则命令行的前提条件是：1。 目标文件不存在； 2. 存在一个依赖的最后修改时间比目标的最后修改时间晚。规则的中心思想就是：**目标文件的内容是由依赖文件文件决定，依赖文件的任何一处改动，将导致目前已经存在的目标文件的内容过期。**规则的命令为重建目标提供了方法。这些命令运行在系统 shell 之上。

3.3 依赖的类型

GNU make 的规则中可以使用两种不同类型的依赖：1. 在以前章节所提到的规则中使用的是常规依赖，这是我们书写的 Makefile 规则中最常用的一种。2. 另外一种在我们书写 Makefile 时不会经常使用，它比较特殊、称之为“order-only”依赖。一个规则的常规依赖（通常是多个依赖文件）表明了两件事：首先，它决定了重建规则目标所要执行命令的顺序；表明在更新这个规则的目标（执行此规则的命令行）之前必需要按照什么样的顺序、执行那些命令来重建这些依赖文件（对所有依赖文件的重建，使用明确或者隐含规则。就是说对于这样的规则：A:B C，那么在重建目标 A 之前，首先需要完成对它的依赖文件 B 和 C 的重建。重建 B 和 C 的过程就是执行 Makefile 中文件 B 和 C 所在的规则）。其次，它确定了一个依存关系；规则中如果依赖文件的任何一个比目标文件新，则被认为规则的目标已经过期同时需要重建目标。

通常，如果规则中依赖文件中的任何一个被更新，则规则的目标相应地也应该被更新。

有时，我们需要定义一个这样的规则，在更新目标（目标文件已经存在）时只需要根据依赖文件中的部分来决定目标是否需要被重建，而不是在依赖文件的任何一个被修改后都重建目标。为了实现这个目的，我们需要对依赖进行分类，一类是这些依赖文件的更新需要对应更新目标文件，另一类是这些依赖的更新不会导致目标被重建。第二类的依赖我们就称他为："order-only"依赖。在书写规则时，"order-only"依赖使用管道符号"|"开始，作为目标的一个依赖文件。规则的依赖列表中管道符号"|"左边的是常规依赖文件，所有出现在管道符号右边的就是"order-only"依赖。这样的规则书写格式如下：

TARGETS : NORMAL-PREREQUISITES | ORDER-ONLY-PREREQUISITES

规则中常规依赖文件可以是空。允许对一个目标声明多行按正确顺序依次追加的依赖。需要注意：规则依赖文件中如果一个文件被同时声明为常规依赖和"order-only"依赖，那么此文件被作为常规依赖处理（因为常规依赖所实现的动作是"order-only"依赖所实现的动作的一个超集）。

"order-only"依赖的使用举例：

```
LIBS = libtest.a
foo : foo.c | $(LIBS)
    $(CC) $(CFLAGS) $< -o $@ $(LIBS)
```

make在执行这个规则时，如果目标文件"foo"已经存在。当"foo.c"被修改以后，目标"foo"将会被重建，但是当"libtest.a"被修改以后。将不执行规则的命令来重建目标"foo"。

就是说，规则中依赖文件$(LIBS)只有在目标文件不存在的情况下，才会参与规则的执行。当目标文件存在时此依赖不会参与规则的执行过程。

3.4 文件名使用通配符

Maekfile 中表示一个单一的文件名时可使用通配符。可使用的通配符有："* "、"?"和"[..]"。在 Makefile 中通配符的用法和含义和 Linux（unix）的 Bourne shell 完全相同。例如，"* .c"代表了当前工作目录下所有的以".c"结尾的文件等。但是在 Makefile 中这些统配符并不是可以用在任何地方，Makefile 中统配符可以出现在以下两种场合：

1. 可以用在规则的目标、依赖中，此时 make 会自动将其展开；
2. 可出现在规则的命令中，其展开是在 shell 在执行此命令时完成。

除这两种情况之外的其它上下文中，不能直接使用通配符。二是需要通过函数"wildcard"（可参考 7.3 文件名处理函数 一节）来实现。

如果规则中的某一个文件的文件名包含作为统配符的字符（"* "、"."字符），在使用文件时需要对文件名中的统配字符进行转义处理，使用反斜线（\）来进行通配符的转义。例如"foo* bar"，在 Makefile 中它表示了文件"foo* bar"。Makefile 中对一些特殊字符的转移和 B-SHELL 以及 C

语言中的基本上相同。

另外需要注意：在 Linux（unix）中，以波浪线“~”开始的文件名有特殊含义。

单独使用它或者其后跟一个斜线（~/），代表了当前用户的宿主目录。（在 shell 下可以通过命令“echo ~(~\)”来查看）。例如“~/bin”代表“/home/username/bin/”（当前用户宿主目录下的 bin 目录）

波浪线之后跟一个单词（~word），其代表由这个“word”所指定的用户的宿主目录。例如“~john/bin”就是代表用户 john 的宿主目录下的 bin 目录。

在一些系统中（像 MS-DOS 和 MS-Windows），用户没有各自的宿主目录，此情况下可通过设置环境变量“HOME”来模拟。

3.4.1 统配符使用举例

本节开始已经提到过，通配符可被用在规则的命令中，它是在命令被执行时由 shell 进行处理的。例如 Makefile 的清空过程文件规则：

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

通配符也可以用在规则的依赖文件名中。看看下面这个例子。执行“make print”，执行的结果是打印当前工作目录下所有的在上一次打印以后被修改过的“.c”文件。

```
print: *.c
    lpr -p $?
    touch print
```

两点说明：1. 上述的规则中目标“print”时一个空目标文件。（不存在一个这样的文件，此目标不代表一个文件，它只是记录了一个所要执行的动作或者命令。参考 3.8 空目标文件 一节）。2. 自动环变量“$?”用在这里表示依赖文件列表中被改变过的所有文件。

变量定义中使用的通配符不会被展开（因此在定义变量不能按照这这种方式，下一小节将会详细讨论）。如果 Makefile 有这样一句：“objects = *.o”。那么变量“objects”的值就是“*.o”，而不是使用空格分开的所有.o 文件列表。如果需要变量“objects”代表所有的.o 文件，则需要是用函数“wildcard”来实现（objects = $(wildcar *.o)）。

3.4.2 通配符存在的缺陷

上一小节已经提到过在变量定义时使用通配符可能会导致意外的结果。本小节将此详细地分析和讨论。在书写 Makefile 时，可能存在这种不正确使用通配符的方法。这种看似正确的方式产生的结果可能并非你所期望得到的。假如在你的 makefile 中，期望能够根据所有的.o 文件生成可执

行文件“foo”。实现如下：

```
objects = *.o

foo : $(objects)
      cc -o foo $(CFLAGS) $(objects)
```

这里变量“objects”的值是一个字符串“*.o”。在重建“foo”的规则中对变量“objects”进行展开，目标“foo”的依赖就是“*.o”，即所有的.o 文件的列表。如果工作目录下已经存在必需的.o 文件，那么这些.o 文件将成为目标的依赖文件，目标“foo”将根据规则被重建。

但是如果我们将工作目录下所有的.o 文件删除，在执行规则时将会得到一个类似于“没有创建*.o 文件的规则” 的错误提示。这当然不是我们所期望的结果（可能在出现这个错误时会令你感到万分迷惑！）。为了实现们的初衷，在对变量进行定义的时需要使用一些高级的技巧，包括使用“wildcard”函数和实现字符串的置换。关于如何实现字符串的置换，我们将在后续进行详细地讨论。

3.4.3 函数wildcard

前边提到过，在规则中，通配符会被自动展开。但在变量的定义和使用函数是，通配符不会被自动的展开。这种情况下需要通配符有效，要用到函数“wildcard”，其用法：$(wildcard PATTERN...)；在 Makefile 中，它被展开未已经存在的、空格分割的、匹配此模式的所有文件列表。如果不存在符合此模式的文件，那么函数会忽略模式并返回空。需要注意的是：这种情况下规则中通配符的展开和上一小节匹配通配符的区别。

一般我们可以使用“$(wildcard *.c)”来获取工作目录下的所有的.c 文件列表。复杂一些用法：可以使用“$(patsubst %.c,%.o,$(wildcard *.c))”，首先使用“wildcard”函数获取工作目录下的.c 文件列表；之后将列表中所有文件名的后缀.c 替换为.o。这样我们就可以得到在当前目录可生成的.o 文件列表。因此在一个目录下可以使用如下内容的 Makefile 来将工作目录下的所有的.c 文件进行编译并最后连接成为一个可执行文件：

```
#sample Makefile
objects := $(patsubst %.c,%.o,$(wildcard *.c))

foo : $(objects)
      cc -o foo $(objects)
```

这里我们使用了make的隐含规则来编译.c的源文件。对变量的赋值也用到了一个特殊的符号（:=）。关于变量定义可参考 5.2 两种变量定义 一节。函数“patsubst”可参考 7.2 文本处理函数 一节

3.5 目录搜寻

在一个较大的工程中，一般会将源代码和二进制文件（.o 文件和可执行文件）安排在不同的目录来进行区分管理。这种情况下，我们需要使用 make 提供的目录自动搜索依赖文件功能（在指定的若干个目录下搜索依赖文件）。书写 makefile 时，指定依赖文件的搜索目录。当工程的目录结构发生变化时，我们就可以不更改 Makefile 的规则，而只更改依赖文件的搜索目录。

本节我们将详细讨论在书写 Makefile 时如何使用这一特性。在自己的工程中灵活运用这一特性，将会起到事半功倍的效果。

3.5.1 一般搜索（变量VPATH）

make 可识别一个特殊变量“VPATH”。通过变量“VPATH”可以指定依赖文件的搜索路径，在规则的依赖文件在当前目录不存在时，make 会在此变量所指定的目录下去寻找这些依赖文件。一般我们都是用此变量来说明规则中的依赖文件的搜索路径。其实“VPATH”变量所指定的是 Makefile 中所有文件的搜索路径，包括依赖文件和目标文件。

变量“VPATH”的定义中，使用空格或者冒号（:）将多个目录分开。make 搜索的目录顺序按照变量“VPATH”定义中顺序进行（当前目录永远是第一搜索目录）。例如对变量的定义如下：

VPATH = src:../headers

它指定了两个搜索目录，“src”和“../headers”。对于规则“foo:foo.c”如果“foo.c”在“src”目录下，此时此规则等价于“foo:src:/foo.c”。

通过“VPATH”变量指定的路径在 Makefile 中对所有文件有效。当需要为不类型的文件指定不同的搜索目录时，需要使用另外一种方式。下一小节我们将会讨论这种更高级的方式。

3.5.2 选择性搜索（关键字vpath）

另一个设置文件搜索路径的方法是使用 make 的“vpath”关键字（全小写的）。它不是一个变量，是一个 make 的关键字，它所实现的功能和上一小节提到的“VPATH”变量很类似，但是它更为灵活。它可以为不同类型的文件（由文件名区分）指定不同的搜索目录。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”。多个目录使用空格或者冒号（:）分开。类似上一小节的“VPATH”

2、***vpath PATTERN***

清除之前为符合模式“PATTERN”的文件设置的搜索路径。

3、***vpath***

清除所有已被设置的文件搜索路径。

vapth 使用方法中的“PATTERN”需要包含模式字符“%”。“%”意思是匹配一个或者多个字符，例如，“%.h”表示所有以“.h”结尾的文件。如果在“PATTERN”中没有包含模式字符“%”而是一个明确的文件名，就是指出了此文件所在的目录，我们很少使用这种方式来为单独的一个文件指定搜索路径。在“vpath”所指定的模式中我们可以使用反斜杠来对字符“%”的引用（和其他的特使字符的引用一样）。

“PATTERN”表示了具有相同特征的一类文件，而“DIRECTORIES”则指定了搜索此类文件目录。当规则的依赖文件列表中出现的文件不能在当前目录下找到时，make 程序将依次在“DIRECTORIES”所描述的目录下寻找此文件。例如：

vpath %.h ../headers

其含义是：Makefile 中出现的.h 文件；如果不能在当前目录下找到，则到目录“../headers”下寻找。注意：这里指定的路径仅限于在 Makefile 文件内容中出现的.h 文件。 并不能指定源文件中包含的头文件所在的路径（在.c 源文件中所包含的头文件需要使用 GCC 的命令行来说明）。

在 Makefile 中如果连续的多个 vpath 语句中使用了相同的“PATTERN”，make 就对这些 vpath 语句一个一个进行处理，搜索某种模式文件的目录将是所有的通过 vpath 指定的符合此模式的目录，其搜索目录的顺序由 vpath 语句在 Makefile 出现的先后次序来决定。多个具有相同“PATTERN”的 vpath 语句之间相互独立。下边是两种方式下，所有的.c 文件的查找目录的顺序（不包含工作目录，对工作目录的搜索永远处于最优先地位）比较：

vpath %.c foo
vpath % blish
vpath %.c bar

表示对所有的.c 文件，make 依次查找目录：“foo”、blish”、“bar”。

而：

vpath %.c foo：bar
vpath % blish

对于所有的.c 文件 make 将依次查找目录：“foo”、“bar”、“blish”

3.5.3 目录搜索的机制

规则中一个依赖文件可以通过目录搜寻找到（使用前边提到的一般搜索或者是选择性搜索任一种），但是有可能此文件的完整路径名（文件的相对路径或者绝对路径，如：/home/Stallman/foo.c）却并不是规则中列出的依赖（规则“foo : foo.c”，在执行搜索后可能

得到的依赖文件为："../src/foo.c"。目录"../src"是使用"VPATH"或"vpath"指定的)；因此使用目录搜索所到的完整的文件路径名可能需要废弃。make 在解析 Makefile 文件执行规则时对文件路径保存或废弃所依据的算法如下：

1. 首先，如果规则的目标文件在 Makefile 文件所在的目录（工作目录）下不存在，那么就执行目录搜寻。
2. 如果目录搜寻成功，在指定的目录下存在此规则的目标。那么搜索到的完整的路径名就被作为临时的目标文件被保存。
3. 对于规则中的所有依赖文件使用相同的方法处理。
4. 完成第三步的依赖处理后，make 程序就可以决定规则的目标是否需要重建，两种情况时后续处理如下：
   a) 规则的目标不需要重建：那么通过目录搜索得到的所有完整的依赖文件路径名有效，同样，规则的目标文件的完整的路径名同样有效。就是说，当规则的目标不需要被重建时，规则中的所有的文件完整的路径名有效。已经存在的目标文件所在的目录不会被改变。
   b) 规则的目标需要重建：那么通过目录搜索所得到的目标文件的完整的路径名无效，规则中的目标文件将会被在工作目录下重建。就是说，当规则的目标需要重建时，规则的目标文件会在工作目录下被重建，而不是在目录搜寻时所得到的目录。这里，必须明确：此种情况只有目标文件的完整路径名失效，依赖文件的完整路径名是不会失效的。否则将无法重建目标。

该算法看起来比较法杂，但它确实使 make 实现了我们所需要的东西。此算法使用纯粹的语言描述可能显得晦涩。本小节后续将使用一个例子来说明。使大家能够对此算法有明确的理解。对于其他版本的 make 则使用了一种比较简单的算法：如果规则的目标文件的完整路径名存在（通过目录搜索可以定位到目标文件），无论该目标是否需要重建，都使用搜索到的目标文件完整路径名。

实际上，GNU make 也可以实现这种功能。如果需要 make 在执行时，将目标文件在已存在的目录存下进行重建，我们可以使用"GPATH"变量来指定这些目标所在的目录。"GPATH"变量和"VPATH"变量具有相同的语法格式。make 在执行时，如果通过目录搜寻得到一个过时的完整的目标文件路径名，而目标存在的目录又出现在"GPATH"变量的定义列表中，则该目标的完整路径将不废弃，目标将在该路径下被重建。

为了更清楚地描述此算法，我们使用一个例子来说明。存在一个目录"prom"，"prom"的子目录"src"下存在"sum.c"和"memcp.c"两个源文件。在"prom"目录下的 Makefile 部分内容如下：

```
LIBS = libtest.a
VPATH = src
```

```
libtest.a : sum.o memcp.o
        $(AR) $(ARFLAGS) $@ $^
```

首先，如果在两个目录（“prom”和“src”）都不存在目标“libtest.a”，执行 make 时将会在当前目录下创建目标文件“libtest.a”。另外；如果“src”目录下已经存在“libtest.a”，以下两种不同的执行结果：

1) 当它的两个依赖文件“sum.c”和“memcp.c”没有被更新的情况下我们执行 make，首先 make 程序会搜索到目录“src”下的已经存在的目标“libtest.a”。由于目标“libtest.a”的依赖文件没有发生变化，所以不会重建目标。并且目标所在的目录不会发生变化。
2) 当我们修改了文件“sum.c”或者“memcp.c”以后执行 make。“libtest.a”和“sum.o”或者“memcp.o”文件将会被在当前目录下创建（目标完整路径名被废弃），而不是在“src”目录下更新这些已经存在的文件。此时在两个目录下（“prom”和“src”）同时存在文件“libtest.a”。但只有“prom/libtest.a”是最新的库文件。

当在上边的 Makefile 文件中使用“GPATH”指定目录时，情况就不一样了。首先看看怎么使用“GPATH”，改变后的 Makefile 内容如下：

```
LIBS = libtest.a
GPATH = src
VPATH = src
LDFLAGS += -L ./. –ltest
.......
......
```

同样；当两个目录都不存在目标文件“libtest.a”时，目标将会在当前目录（“prom”目录）下创建。如果“src”目录下已经存在目标文件“libtest.a”。当其依赖文件任何一个被改变以后执行 make，目标“libtest.a”将会被在“src”目录下被更新（目标完整路径名不会被废弃）。

3.5.4 命令行和搜索目录

make 在执行时，通过目录搜索得到的目标的依赖文件可能会在其它目录（此时依赖文件为文件的完整路径名），但是已经存在的规则命令却不能发生变化。因此，书写命令时我们必须保证当依赖文件在其它目录下被发现时规则的命令能够正确执行。

处理这种问题的方式就是使用“自动化变量”（可参考 9.5.3 自动化变量 一小节），诸如“$^”等。规则命令行中的自动化变量“$^”代表所有的是的通过目录搜索得到的依赖文件的完整路径名（目录+ 一般文件名）列表。“$@”代表规则的目标。所以对于一个规则我们可以进行如下的描述：

```
foo.o : foo.c
        cc -c $(CFLAGS) $^ -o $@
```

变量“CFLAGS”是编译.c 文件时 GCC 的命令行选项，可以在 Makefile 中给它指定明确的值、

也可以使用隐含的定义值。

规则的依赖文件列表中可以包含头文件，而在命令行不需要使用这些头文件（这些头文件的作用只有在 make 程序决定目标是否需要重建时才有意义）。我们可以使用另外一个变量来书代替“$^”，例如：

```
VPATH = src:../headers
foo.o : foo.c defs.h hack.h
        cc -c $(CFLAGS) $< -o $@
```

自动化变量“$<”代表规则中通过目录搜索得到的依赖文件列表的第一个依赖文件。关于自动化变量我们在后续有专门的讨论。

3.5.5 隐含规则和搜索目录

隐含规则同样会为依赖文件搜索通过变量“VPATH”、或者关键字“vpath”指定的搜索目录。例如：一个目标文件“foo.o”在 Makefile 中没有重建它的明确规则，make 会使用隐含规则来由已经存在的“foo.c”来重建它。当“foo.c”在当前目录下不存在时，make 将会进行目录搜索。如果能够在一个可以搜索的目录中找到此文件，同样 make 会使用隐含规则根据搜索到的文件完整的路径名去重建目标，编译这个.c 源文件。

隐含规则中的命令行中就是使用自动化变量来解决目录搜索可能带来的问题；相应的命令中的文件名都是使用目录搜索得到的完整的路径名。（可参考上一小节）

3.5.6 库文件和搜索目录

Makefile 中程序链接的静态库、共享库同样也可以有目录搜索得到。这一特性需要我们在书规则的依赖是指定一个类似“-lNNAM”的依赖文件名（一个奇怪的依赖文件名！一般依赖文件名应该是一个普通文件的名字。库文件的命名也应该是“libNAME.a”而不是所写的“-lNAME”。这是为什么，熟悉 GNU ld 的话我想这就不难理解了，“-lNAME”的表示方式和 ld 的对库的引用方式完全一样，只是我们在书写 Makefile 的规则时使用了这种书写方式。因此你不应该感到奇怪）。下边我们就来看看这种奇怪的依赖文件到底是什么。

当规则中依赖文件列表中存在一个“-lNAME”形式的文件时。make 将根据“NAME”首先搜索当前系统可提供的共享库，如果当前系统不能提供这个共享库，则搜索它的静态库（当然你可以在命令行中指定编译或者连接选项来指定动态连接还是静态连接，这里我们不讨论）。来看一下详细的过程。1. make 在执行规则时会在当前目录下搜索一个名字为“libNAME.so”的文件；2. 如果当前工作目录下不存在这样一个文件，则 make 程序会继续搜索使用“VPATH”或者“vpath”指定的搜索目录。3. 还是不存在，make 程序将搜索系统默认目录，顺序是：“/lib”、“/usr/lib”和“PREFIX/lib”（在 Linux 系统中为“/usr/local/lib”，其他的系统可能不同）。

如果“libNAME.so”通过以上的途径最后还是没有找到的话，那么 make 程序将会按照以上的搜索顺序查找名字为“libNAME.a”的文件。

假设你的系统中存在“/usr/lib/libcurses.a”（不存在“/usr/lib/libcurses.so”）这个库文件。看一个例子：

```
foo : foo.c -lcurses
      cc $^ -o $@
```

上例中，如果文件“foo.c”被修改或者“/usr/lib/libcurses.a”被更新，执行规则时将使用命令“cc foo.c /usr/lib/libcurses.a -o foo”来完成目标文件的重建。需要注意的是：如果“/usr/lib/libcurses.a”需要在执行 make 的时生成，那么就不能这样写，因为“-lNAME”只是告诉了链接器在生成目标时需要链接那个库文件。上例的中的“-lcurses”并没有告诉 make 程序其依赖的库文件应该如何重建。当搜索的所有目录中不存在库“libcurses”时。Make 将提示“没有规则可以创建目标“foo”需要的目标“-lcurses”。如果在执行 make 时，出现这样的提示信息，你应该明确发生了什么错误，而不要因为错误而不知所措。

当规则的依赖列表中出现了“-lNAME”格式的依赖，默认搜索的文件名为“libNAME.so”和“libNAME.a”，这是由变量“.LIBPATTERNS”来指定的。“.LIBPATTERNS”的值一般是多个包含模式字符（%）的字（一个不包含空格的字符串），多个字之间使用空格分开。在规则中出现“-lNAME”格式的的依赖时，首先使用这里的“NAME”代替变量“.LIBPATTERNS”的第一个字的模式字符（%）而得到第一个库文件名，根据这个文件名在搜索目录下查找，如果能够找到、就是用这个文件，否则使用“NAME”代替第二个字的模式字符，进行同样的查找。默认情况时，“.LIBPATTERNS”的值为：“lib%.so lib%.a”。这也是默认情况下在规则存在“-lNAME”格式的依赖时，链接生成目标时使用“libNAME.so”和“libNAME.a”的原因。

变量“.LIBPATTERNS”就是告诉链接器在执行链接过程中对于出现“-LNAME”的文件如何展开。当然我们也可以将此变量制空，取消链接器对“-lNAME”格式进行展开。

3.6 Makefile伪目标

本节我们讨论一个 Makefile 中的一个重要的特殊目标：伪目标。

伪目标是这样一个目标：它不代表一个真正的文件名，在执行 make 时可以指定这个目标来执行其所在规则定义的命令，有时我们也可以将一个伪目标称为标签。使用伪目标有两点原因：1. 避免在我们的 Makefile 中定义的只执行命令的的目标（此目标的目的为了执行执行一些列命令，而不需要创建这个目标）和工作目录下的实际文件出现名字冲突。2. 提高执行 make 时的效率，特别是对于一个大型的工程来说，编译的效率也许你同样关心。以下就这两个问题我们进行分析讨论：

1. 如果我们需要书写这样一个规则：规则所定义的命令不是去创建目标文件，而是使用 make

指定具体的目标来执一些特定的命令。像下边那样：

```
clean:
    rm *.o temp
```

规则中“rm”不是创建文件“clean”的命令，只是删除当前目录下的所有.o 文件和 temp 文件。在工作目录下不存在“clean”这个文件时，我们输入“make clean”后，“rm *.o temp”总会被执行。这是我们的初衷。

但当前工作目录下存在文件“clean”时情况就不一样了，在我们输入“make clean”时。规则没有依赖文件，所以目标被认为是最新的而不去执行规则作定义的命令，命令“rm”将不会被执行。这并不是我们的初衷。为了避免这个问题，我们可以将目标“clean”明确的声明为伪目标。将一个目标声明为伪目标需要将它作为特殊目标.PHONY”的依赖。如下：

```
.PHONY : clean
```

这样目标“clean”就是一个伪目标，无论当前目录下是否存在“clean”这个文件。我们输入“make clean”之后。“rm”命令都会被执行。而且，当一个目标被声明为伪目标后，make 在执行此规则时不会试图去查找隐含规则来创建这个目标。这样也提高了 make 的执行效率，同时我们也不用担心由于目标和文件名重名而使我们的期望失败。在书写伪目标规则时，首先需要声明目标是一个伪目标，之后才是伪目标的规则定义。目标“clean”书写格式应该如下：

```
.PHONY: clean
clean:
    rm *.o temp
```

2. 伪目标的另外一使用场合在 make 的并行和递归执行过程中。此情况下一般存在一个变量，其定义为所有需要 make 的子目录。对多个目录进行 make 的实现方式可以在一个规则中可以使用 shell 的循环来完成。如下：

```
SUBDIRS = foo bar baz

subdirs:
    for dir in $(SUBDIRS); do \
        $(MAKE) -C $$dir; \
    done
```

但这种实现方法存在以下几个问题。1. 当子目录执行 make 出现错误时，make 不会退出。就是说，在对某一个目录执行 make 失败以后，会继续对其他的目录进行 make。在最终执行失败的情况下，我们很难根据错误的提示定位出具体是是那个目录下的 Makefile 出现错误。这给问题定位造成了很大的困难。为了避免这样的问题，我们可以在命令行部分加入错误的监测，在命令执行错误后 make 退出。不幸的是，如果在执行 make 时使用了“-k”选项，此方式将失效。2. 另

外一个问题就是使用这种 shell 的循环方式时，没有用到 make 对目录的并行处理功能，因为规则的命令是一条完整的 shell 命令，不能被并行的执行。

我们可以通过伪目标方式来克服以上实现方式所存在的两个问题。

```
SUBDIRS = foo bar baz

.PHONY: subdirs $(SUBDIRS)

subdirs: $(SUBDIRS)
$(SUBDIRS):
    $(MAKE) -C $@
foo: baz
```

上边的实现中使用了一个没有命令行的规则“foo: baz”，用来限制子目录的 make 顺序。此规则的含义时在处理“foo”目录之前，需要等待“baz”目录处理完成。在书写一个并行执行 make 的 Makefile 时，目录的处理顺序是需要特别注意的。

一般情况下，一个伪目标不作为一个另外一个目标文件的依赖。这是因为当一个目标文件的依赖包含伪目标时，每一次在执行这个规则时伪目标所定义的命令都会被执行（因为它是规则的依赖，重建规则目标文件时需要首先重建它的依赖）。当伪目标没有作为任何目标（此目标是一个可被创建或者已存在的文件）的依赖时，我们只能通过 make 的命令行选项明确指定这个伪目标，来执行它所定义的命令。例如我们的“make clean”。

Makefile 中，伪目标可以有自己的依赖。在一个目录下如果需要创建多个可执行程序，我们可以将所有程序的重建规则在一个 Makefile 中描述。因为 Makefile 中第一个目标是“终极目标”，约定的做法是使用一个称为“all”的伪目标来作为终极目标，它的依赖文件就是那些需要创建的程序。下边就是一个例子：

```
#sample Makefile
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
```

执行 make 时，目标“all”被作为终极目标。为了完成对它的更新，make 会创建（不存在）或者重建（已存在）目标“all”的所有依赖文件（prog1、prog2 和 prog3）。当需要单独更新某一个程序时，我们可以通过 make 的命令行选项来明确指定需要重建的程序。(例如：“make prog1”)。

当一个伪目标作为另外一个伪目标依赖时，make 将其作为另外一个伪目标的子例程来处理（可以这样理解：其作为另外一个伪目标的必须执行的部分，就行 C 语言中的函数调用一样）。下边的例子就是这种用法：

```
.PHONY: cleanall cleanobj cleandiff
cleanall : cleanobj cleandiff
    rm program

cleanobj :
    rm *.o

cleandiff :
    rm *.diff
```

“cleanobj”和“cleandiff”这两个伪目标有点像“子程序”的意思（执行目标“clearall 时会触发它们所定义的命令被执行”）。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。例子首先通过特殊目标“.PHONY”声明了多个伪目标，它们之间使用空各分割，之后才是各个伪目标的规则定义。

说明：

通常在清除文件的伪目标所定义的命令中“rm”使用选项“–f”（--force）来防止在缺少删除文件时出错并退出，使“make clean”过程失败。也可以在“rm”之前加上“-”来防止“rm”错误退出，这种方式时 make 会提示错误信息但不会退出。为了不看到这些讨厌的信息，需要使用上述的第一种方式。

另外 make 存在一个内嵌隐含变量“RM”，它被定义为：“RM = rm –f”。因此在书写“clean”规则的命令行时可以使用变量“$(RM)”来代替“rm”，这样可以免出现一些不必要的麻烦！这是我们推荐的用法。

3.7 强制目标（没有命令或依赖的规则）

如果一个规则没有命令或者依赖，而且它的目标不是一个存在的文件名。在执行此规则时，目标总会被认为是最新的。就是说：这个规则一旦被执行，make 就认为它的目标已经被更新过。这样的目标在作为一个规则的依赖时，因为依赖总被认为被更新过，因此作为依赖所在的规则定义的命令总会被执行。看一个例子：

```
clean: FORCE
    rm $(objects)
FORCE:
```

这个例子中，目标“FORCE”符合上边的条件。它作为目标“clean”的依赖出现，在执行 make 时，它总被认为被更新过。所以“clean”所在的规则在被执行时规则所定义的命令总会被执行。

这样的一个目标通常我们将其命名为“FORCE”。

上边的例子中使用“FORCE”目标的效果和我们指定“clean”为伪目标效果相同。两种方式相比较，使用“.PHONY”方式更加直观高效。这种方式主要用在非 GNU 版本的 make 中。

在使用GNU make，尽量避免使用这种方式。在GNU make中我们推荐使用伪目标方式。关于伪目标可参考3.6 Makefile伪目标 一节

3.8 空目标文件

空目标是伪目标的一个变种；此目标所在规则执行的目的和伪目标相同——通过 make 命令行指定终极目标来执行规则所定义的命令。和伪目标不同的是：这个目标可以是一个存在的文件，一般文件的具体内容我们并不关心，通常此文件是一个空文件。

空目标文件只是用来记录上一次执行此规则定义命令的时间。一般在这样的规则中，命令部分都会使用“touch”在完成所有命令之后来更新目标文件的时间戳，记录此规则命令的最后执行时间。make 时通过命令行将此目标作为终极目标，当前目录下如果不存在这个文件，“touch”会在第一次执行时创建一个空的文件（命名为空目标文件名）。

通常，一个空目标文件应该存在一个或者多个依赖文件。将这个目标作为终极目标，在它所依赖的文件比它新时，此目标所在规则的命令行将被执行。就是说，如果空目标的依赖文件被改变之后，空目标所在规则中定义的命令会被执行。看一个例子：

```
print: foo.c bar.c
        lpr -p $?
        touch print
```

执行“make print”，当目标“print”的依赖文件任何一个被修改之后，命令“lpr –p $?”都会被执行，打印这个被修改的文件。关于自动化变量“$?”可参考 9.5.3 自动化变量 一小节。

3.9 Makefile的特殊目标

在 Makefile 中，有一些名字，当它们作为规则的目标出现时，具有特殊含义。它们是一些特殊的目标，GNU make 所支持的特殊的目标有：

.PHONY：

目标“**.PHONY**”的所有的依赖被作为伪目标。伪目标时这样一个目标：当使用make命令行指定此目标时，这个目标所在规则定义的命令、无论目标文件是否存在都会被无条件执行。参考 3.6 Makefile伪目标 一节

.SUFFIXES:

特殊目标“SUFFIXES”的所有依赖指出了一系列在后缀规则中需要检查的后缀名（就是当前make需要处理的后缀）。参考 9.7 后缀规则 一节

.DEFAULT

Makefile 中，目标“.DEFAULT”所在规则定义的命令，被用在重建那些没有具体规则的目标（明确规则和隐含规则）。就是说一个文件作为某个规则的依赖，但却不是另外一个规则的目标时。Make 程序无法找到重建此文件的规则，此种情况时就执行“.DEFAULT”所指定的命令。

.PRECIOUS

目标“.PRECIOUS”的所有依赖文件在make过程中会被特殊处理：当命令在执行过程中被中断时，make不会删除它们（可参考 4.5 中断make的执行 一节）。而且如果目标的依赖文件是中间过程文件，同样这些文件不会被删除。这一点目标“.PRECIOUS”和目标“.SECONDAY”实现的功能相同。参考 9.4 make隐含规则链 一节

另外，目标“.PRECIOUS”的依赖文件也可以是一个模式，例如“%.o”。这样可以保留有规则创建的中间过程文件。

.INTERMEDIATE

目标“.INTERMEDIATE”的依赖文件在make时被作为中间过程文件对待。没有任何依赖文件的目标“.INTERMEDIATE”没有意义。参考 9.4 make隐含规则链 一节

.SECONDARY

目标“.SECONDARY”的依赖文件被作为中间过程文件对待。但这些文件不会被自动删除（可参考 9.4 make隐含规则链 一节）

没有任何依赖文件的目标“.SECONDARY”的含义是：将所有的文件作为中间过程文件（不会自动删除任何文件）。

.DELETE_ON_ERROR

如果在Makefile中存在特殊目标“.DELETE_ON_ERROR”，make在执行过程中，如果规则的命令执行错误，将删除已经被修改的目标文件。参考 4.4 命令执行的错误 一节

.IGNORE

如果给目标“.IGNORE”指定依赖文件，则忽略创建这个文件所执行命令的错误。给此目标指定命令是没有意义的。当此目标没有依赖文件时，将忽略所有命令执行的错误。参考 4.4 命令执行的错误 一节

.LOW_RESOLUTION_TIME

目标“.LOW_RESOLUTION_TIME”的依赖文件被 make 认为是低分辨率时间戳文件。给目标“.LOW_RESOLUTION_TIME”指定命令是没有意义的。

通常文件的时间辍都是高分辨率的，make 在处理依赖关系时、对规则目标-依赖文件的高分辨率的时间戳进行比较，判断目标是否过期。但是在系统中并没有提供一个修改文件高分辨率时间辍的机制（方式），因此类似“cp -p”这样的命令在根据源文件创建目的文件时，所产生的目的文件的高分辨率时间辍的细粒度部分被丢弃（来源于源文件）。可能会造成目的文件的时间戳和源文

件的相等甚至不及源文件新。处理此类命令创建的文件时，需要将命令创建的文件作为目标“.LOW_RESOLUTION_TIME”的依赖，声明这个文件是一个低分辨率时间戳的文件。例如：

```
.LOW_RESOLUTION_TIME: dst
dst: src
        cp -p src dst
```

首先规则的命令“cp –p src dst”，所创建的文件“dst”在时间戳上稍稍比“src”晚（因为命令不能更新文件“dst”的细粒度时间）。因此 make 在判断文件依赖关系时会出现误判，将文件作为目标“.LOW_RESOLUTION_TIME”的依赖后，只要规则中目标和依赖文件的时间戳中的初始时间相等，就认为目标已经过期。这个特殊的目标主要作用是，弥补系统在没有提供修改文件高分辨率时间戳机制的情况下，某些命令在 make 中的一些缺陷。

对于静态库文件（文档文件）成员的更新也存在这个问题。make 在创建或者更新静态库时，会自动将静态库的所有成员作为目标“.LOW_RESOLUTION_TIME”的依赖。

.SILENT

出现在目标“.SILENT”的依赖列表中的文件，make 在创建这些文件时，不打印出重建此文件所执行的命令。同样，给目标“.SILENT”指定命令行是没有意义的。

没有任何依赖文件的目标“.SILENT”告诉make在执行过程中不打印任何执行的命令。现行版本make支持目标“.SILENT”的这种功能和用法是为了和旧版本的兼容。在当前版本中如果需要禁命令执行过程的打印，可以使用make的命令行参数“-s”或者“--silent”。参考 8.7 make的命令行选项 一节

.EXPORT_ALL_VARIABLES

此目标应该作为一个简单的没有依赖的目标，它的功能含义是将之后所有的变量传递给子make进程。参考 4.6 make的递归执行 一节

.NOTPARALLEL

Makefile 中，如果出现目标“.NOPARALLEL”，则所有命令按照串行方式执行，即使存在 make 的命令行参数“-j”。但在递归调用的字 make 进程中，命令可以并行执行。此目标不应该有依赖文件，所有出现的依赖文件将被忽略。

所有定义的隐含规则后缀作为目标出现时，都被视为一个特殊目标，两个后缀串联起来也是如此，例如“.c.o”。这样的目标被称为后缀规则的目标，这种定义方式是已经过时的定义隐含规则的方法（目前，这种方式还被用在很多地方）。原则上，如果将其分为两个部分、并将它们加到后缀列表中，任何目标都可采用这种方式来表示。实际中，后缀通常以“.”开始，因此，以上的这些特别目标同样是以“.”开始。可参考 9.7 后缀规则 一节

3.10 多目标

一个规则中可以有多个目标，规则所定义的命令对所有的目标有效。一个具有多目标的规则相当于多个规则。规则中命令对不同的目标的执行效果不同，因为在规则的命令中可能使用自动环变量“$@”。多目标规则意味着所有的目标具有相同的依赖文件。多目标通常用在以下两种情况：

- 仅需要一个描述依赖关系的规则，而不需要在规则中定义命令。例如

  kbd.o command.o files.o: command.h

  这个规则实现了给同时给三个目标文件指定一个依赖文件。

- 对于多个具有类似重建命令的目标。重建这些目标的命令并不需要是绝对相同，因为我们可以在命令行中使用make的自动环变量“$@”来引用具体一个目标，并完成对它的重建（可参考 9.5.3 自动化变量 一小节）。例如规则：

```
bigoutput littleoutput : text.g
        generate text.g -$(subst output,,$@) > $@
```

其等价于：

```
bigoutput : text.g
        generate text.g -big > bigoutput
littleoutput : text.g
        generate text.g -little > littleoutput
```

例子中的“generate”根据命令行参数来决定输出文件的类型。使用了make的字符串处理函数“subst”来根据目标产生对应的命令行选项（关于make的函数可参考 第七章 make的内嵌函数 一章）。

虽然在多目标的规则中，可以根据不同的目标使用不同的命令（在命令行中使用自动化变量“$@”）。但是，多目标的规则并不能做到根据目标文件自动改变依赖文件，就像我们在上边例子中使用自动化变量“$@”来改变规则的命令一样。需要实现这个目的是，要用到make的静态模式。（关于静态模式规则可参考 3.12 静态模式 一节）

3.11 多规则目标

Makefile 中，一个文件可以作为多个规则的目标出现。这种情况时，此目标文件的所有依赖文件将会被合并成此目标一个依赖文件列表，其中任何一个依赖文件比目标更新（比较目标文件和依赖文件的时间戳）时，make 将会执行特定的命令来重建这个目标。

对于一个多规则的目标，重建此目标的命令只能出现在一个规则中（可以是多条命令）。如果多个规则同时给出重建此目标的命令，make将使用最后一个规则所以的命令，同时提示错误信息（一个特殊的例外是：使用“.”开头的多规则目标文件，可以在多个规则中给出多个重建命令。这种方式只是为了和其他版本make进行兼容，一般在GNU make中应该避免使用这个功能）。某些

情况下，需要对相同的目标使用不同的规则中所定义的命令，这种情况我们可使用另外一种方式"双冒号"规则来实现。

一个仅仅描述依赖关系的描述规则可用来给出一个或做多个目标文件的依赖文件。例如，Makefile 中通常存在一个变量，就像以前我们提到的"objects"，它定义为所有的需要编译生成的.o 文件的列表。当这些.o 文件在其源文件所包含的头文件"config.h"发生变化之后能够自动的被重建，我们可以使用多目标像下边那样来书写我们的 Makefile：

```
objects = foo.o bar.o
foo.o : defs.h
bar.o : defs.h test.h
$(objects) : config.h
```

这样做的好处是：我们可以在源文件中增加或者删除了包含的头文件以后不用修改已经存在的Makefile的规则，只需要增加或者删除某一个.o文件依赖的头文件。这种方式很简单也很方便。对于一个大的工程来说，这样做的好处是显而易见的。在一个大的工程中，对于一个单独目录下的.o文件的依赖规则建议使用此方式。规则中头文件的依赖描述也可以使用GCC自动产生。可参考 3.14 自动产生依赖 一节

另外，我们也可以通过一个变量来增加目标的依赖文件，使用 make 的命令行来指定某一个目标的依赖头文件，例如：

```
extradeps=
$(objects) : $(extradeps)
```

它的意思是：如果我们执行"make extradeps=foo.h"那么"foo.h"将作为所有的.o 文件的依赖文件。当然我们只执行"make"的话，就没有指定任何文件作为.o 文件的依赖文件。

在多规则的目标中，如果目标的任何一个规则没有定义重建此目标的命令，make将会寻找一个合适的隐含规则来重建此目标。关于隐含规则可参考 第九章 make的隐含规则

3.12 静态模式

静态模式规则是这样一个规则：规则存在多个目标，并且不同的目标可以根据目标文件的名字来自动构造出依赖文件。静态模式规则比多目标规则更通用，它不需要多个目标具有相同的依赖。但是静态模式规则中的依赖文件必须是相类似的而不是完全相同的。

3.12.1 静态模式规则的语法

首先，我们来看一下静态模式规则的基本语法：

```
TARGETS ...: TARGET-PATTERN: PREREQ-PATTERNS ...
```

```
COMMANDS
...
```

“TAGETS”列出了此规则的一系列目标文件。像普通规则的目标一样可以包含通配符。关于通配符的使用可参考 3.4 文件名使用通配符 一节

“TAGET-PATTERN”和“PREREQ-PATTERNS”说明了如何为每一个目标文件生成依赖文件。从目标模式（TAGET-PATTERN）的目标名字中抽取一部分字符串（称为“茎”）。使用“茎”替代依赖模式（PREREQ-PATTERNS）中的相应部分来产生对应目标的依赖文件。下边详细说明这一替代的过程。

首先在目标模式和依赖模式中，一般需要包含模式字符“%”。在目标模式（TAGET-PATTERN）中“%”可以匹配目标文件的任何部分，模式字符“%”匹配的部分就是“茎”。目标文件和目标模式的其余部分必须精确的匹配。看一个例子：目标“foo.o”符合模式“%.o”，其“茎”为“foo”。而目标“foo.c”和“foo.out”就不符合此目标模式。

每一个目标的依赖文件是使用此目标的“茎”代替依赖模式（PREREQ-PATTERNS）中的模式字符“%”而得到。例如：上边的例子中依赖模式（PREREQ-PATTERNS）为“%.c”，那么使用“茎”“foo”替代依赖模式中的“%”得到的依赖文件就是“foo.c”。需要明确的一点是：在模式规则的依赖列表中使用不包含模式字符“%”也是合法的。代表这个文件是所有目标的依赖文件。

在模式规则中字符‘%’可以用前面加反斜杠“\”方法引用。引用“%”的反斜杠也可以由更多的反斜杠引用。引用“%”、“\”的反斜杠在和文件名比较或由“茎”代替它之前会从模式中被删除。反斜杠不会因为引用“%”而混乱。如，模式“the\%weird\\%pattern\\”是“the%weird\”+“%”+“pattern\\”构成。最后的两个反斜杠由于没有任何转义引用“%”所以保持不变。

我们来看一个例子，它根据相应的.c 文件来编译生成“foo.o”和“bar.o”文件：

```
objects = foo.o bar.o

all: $(objects)

$(objects): %.o: %.c
        $(CC) -c $(CFLAGS) $< -o $@
```

例子中，规则描述了所有的.o文件的依赖文件为对应的.c文件，对于目标“foo.o”，取其茎“foo”替代对应的依赖模式“%.c”中的模式字符“%”之后可得到目标的依赖文件“foo.c”。这就是目标“foo.o”的依赖关系“foo.o: foo.c”，规则的命令行描述了如何完成由“foo.c”编译生成目标“foo.o”。命令行中“$<”和“$@”是自动化变量，“$<”表示规则中的第一个依赖文件，“$@”表示规则中的目标文件（可参考 3.14 自动产生依赖 一节）。以上的规则就是描述了以下两个具体的规则：

```
foo.o : foo.c
```

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

在使用静态模式规则时，指定的目标必须和目标模式相匹配，否则在执行make时将会得到一个错误提示。如果存在一个文件列表，其中一部分符合某一种模式而另外一部分符合另外一种模式，这种情况下我们可以使用“filter”函数（可参考 第七章 make的内嵌函数）来对这个文件列表进行分类，在分类之后对确定的某一类使用模式规则。例如：

```
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 $<
```

其中；$(filter %.o,$(files))的结果为“bar.o lose.o”。“filter”函数过滤不符合“%.o”模式的文件名而至返回所有符合此模式的文件列表。第一条静态模式规则描述了这些目标文件是通过编译对应的.c 源文件来重建的。同样第二条规则也是使用这种方式。

我们通过另外一个例子来看一下自动环变量“$*”在静态模式规则中的使用方法：

```
bigoutput littleoutput : %output : text.g
        generate text.g -$* > $@
```

当执行此规则的命令时，自动环变量“$*”被展开为“茎”。在这里就是“big”和“little”。

静态模式规则在一个较大的工程中非常有用的。它可以对一个工程中的同类文件的重建规则进行一次定义，而实现对整个工程中此类文件指定相同的重建规则。比如，可以用来描述整个工程中所有的.o 文件的依赖规则和编译命令。通常的做法是将生成同一类目标的模式定义在一个make.rules 的文件中。在工程各个模块的 Makefile 中包含此文件。

3.12.2 静态模式和隐含规则

在 Makefile 中，静态模式规则和被定义为隐含规则的模式规则都是我们经常使用的两种方式。两者相同的地方都是用目标模式和依赖模式来构建目标的规则中的文件依赖关系，两者不同的地方是 make 在执行时使用它们的时机。

隐含规则可被用在任何和它相匹配的目标上，在 Makefile 中没有为这个目标指定具体的规则、存在规则但规则没有命令行或者这个目标的依赖文件可被搜寻到。当存在多个隐含规则和目标模式相匹配时，只执行其中的一个规则。具体执行哪一个规则取决于定义规则的顺序。

相反的，静态模式规则只能用在规则中明确指出的那些文件的重建过程中。不能用在除此之外的任何文件的重建过程中，并且它对指定的每一个目标来说是唯一的。如果一个目标存在于两个规

则，并且每一个规则中都定以了命令，make 执行时就会提示错误。

静态模式规则相比隐含模式规则由以下两个优点：

- ✧ 不能根据文件名通过词法分析进行分类的文件，我们可以明确列出这些文件，并使用静态模式规则来重建其隐含规则。
- ✧ 对于无法确定工作目录内容，而且不能确定是否此目录下的无关文件会使用错误的隐含规则而导致 make 失败的情况。当存在多个适合此文件的隐含规则时，使用哪一个隐含规则取决于其规则的定义顺序。这种情况下我们使用静态模式规则就可以避免这些不确定因素，因为静态模式中，指定的目标文件有特定的规则来描述其依赖关系和重建命令。

3.13 双冒号规则

双冒号规则就是使用“::”代替普通规则的“:”得到的规则。当同一个文件作为多个规则的目标时，双冒号规则的处理和普通规则的处理过程完全不同（双冒号规则允许在多个规则中为同一个目标指定不同的重建目标的命令）。

首先需要明确的是：Makefile **中，一个目标可以出现在多个规则中。但是这些规则必须是同一种规则，要么都是普通规则，要么都是双冒号规则。而不允许一个目标同时出现在两种不同的规则中**。双冒号规则和普通规则的处理的不同点表现在以下几个方面：

1. 双冒号规则中，当依赖文件比目标更新时。规则将会被执行。对于一个没有依赖而只有命令行的双冒号规则，当引用此目标时，规则的命令将会被无条件执行。而普通规则，当规则的目标文件存在时，此规则的命令永远不会被执行（目标文件永远是最新的）。
2. 当同一个文件作为多个双冒号规则的目标时。这些不同的规则会被独立的处理，而不是像普通规则那样合并所有的依赖到一个目标文件。这就意味着对这些规则的处理就像多个不同的普通规则一样。就是说多个双冒号规则中的每一个的依赖文件被改变之后，make 只执行此规则定义的命令，而其它的以这个文件作为目标的双冒号规则将不会被执行。

我们来看一个例子，在我们的 Makefile 中包含以下两个规则：

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

如果“foo.c”文件被修改，执行 make 以后将根据“foo.c”文件重建目标“Newprog”。而如果“bar.c”被修改那么“Newprog”将根据“bar.c”被重建。回想一下，如果以上两个规则为普通规时出现的情况是什么？（make 将会出错并提示错误信息）

当同一个目标出现在多个双冒号规则中时，规则的执行顺序和普通规则的执行顺序一样，按照其在 Makefile 中的书写顺序执行。

GNU make 的双冒号规则给我们提供一种根据依赖的更新情况而执行不同的命令来重建同一目标的机制。一般这种需要的情况很少，所以双冒号规则的使用比较罕见。一般双冒号规则都需要定义命令，如果一个双冒号规则没有定义命令，在执行规则时将为其目标自动查找隐含规则。

3.14 自动产生依赖

Makefile 中，可能需要书写一些规则来描述一个.o 目标文件和头文件的依赖关系。例如，如果在 main.c 中使用“#include defs.h”，那么我们可能需要如下那样的一个规则来描述当头文件“defs.h”被修改以后执行 make，目标“main.o”应该被重建。

main.o: defs.h

这样，在一个比较大型的工程中。就需要在 Makefile 中书写很多条类似于这样的规则。并且，当在源文件中加入或删除头文件后，也需要小心地去修改 Makefile。这是一件很费力、也很费时并且容易出错误的工作。为了避免这个令人讨厌的问题，现代的 c 编译器提供了通过查找源文件中的“#include”来自动产生这种依赖的功能。“GCC”支持一个“-M”的选项来实现此功能。“GCC”将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“main.c”只包含了头文件“defs.h”，那么在 Linxu 下执行下面的命令：

gcc -M main.c

其输出是：

main.o : main.c defs.h

既然编译器已经提供了自动产生依赖关系的功能，那么我们就不需要去动手写这些规则的依赖关系了。但是需要明确的是：在“main.c”中包含了其他的标准库的头文件，其输出的依赖关系中也包含了标准库的头文件。当不需要依赖关系中不考虑标准库头文件时，需要使用“-MM”参数。

需要注意的是，在使用“GCC”自动产生依赖关系时，所产生的规则中明确的指明了目标是“main.o”文件。一次在通过.c 文件直接产生可执行文件时，作为过程文件的“main.o”的中间过程文件在使用完之后将不会被删除。

在旧版本的 make 中，使用编译器此项功能通常的做法是：在 Makefile 中书写一个伪目标“depend”的规则来定义自动产生依赖关系文件的命令。输入“make depend”将生成一个称为“depend”的文件，其中包含了所有源文件的依赖规则描述。Makefile 使用“include”指示符包含这个文件。

在新版本的 make 中，推荐的方式是为每一个源文件产生一个描述其依赖关系的 makefile 文

件。对于一个源文件“NAME.c”，对应的这个 makefile 文件为“NAME.d”。“NAME.d”中描述了文件“NAME.o”所要依赖的所有头文件。采用这种方式，只有源文件在修改之后才会重新使用命令生成新的依赖关系描述文件“NAME.o”。

我们可以使用如下的模式规则来自动生成每一个.c 文件对应的.d 文件：

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

此规则的含义是：所有的.d 文件依赖于同名的.c 文件。

第一行：使用 c 编译器自自动生成依赖文件（$< ）的头文件的依赖关系，并输出成为一个临时文件，“$$$$”表示当前进程号。如果$(CC)为 GNU 的 C 编译工具，产生的依赖关系的规则中，依赖头文件包括了所有的使用的系统头文件和用户定义的头文件。如果需要生成的依赖描述文件不包含系统头文件，可使用“-MM”代替“-M”。

第二行：使用 sed 处理第二行已产生的那个临时文件并生成此规则的目标文件。这里 sed 完成了如下的转换过程。例如对已一个.c 源文件。将编译器产生的依赖关系：

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

转成：

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

这样就将.d 加入到了规则的目标中，其和对应的.o 文件文件一样依赖于对应的.c 源文件和源文件所包含的头文件。当.c 源文件或者头文件被改变之后规则将会被执行，相应的.d 文件同样会被更新。

第三行：删除临时文件。

使用上例的规则就可以建立一个描述目标文件依赖关系的.d文件。我们可以在Makefile中使用include指示符将描述将这个文件包含进来。在执行make时，Makefile所包含的所有.d文件就会被自动创建或者更新（具体过程可参考 2.7 makefile文件的重建 一节）。Makefile中对当前目录下.d文件处理可以参考如下：

```
sources = foo.c bar.c
sinclude $(sources:.c=.d)
```

例子中，变量“sources”定义了当前目录下的需要编译的源文件。变量引用变换“$(sources : .c= .d)”的功能是根据需要.c文件自动产生对应的.d文件（参考 5.3 变量的高级用法 一节），并在当前Makefile文件中包含这些.d文件。.d文件和其它的makefile文件一样，make在执行时读取并试图重建他们。其实这些.d文件也是一些可被make解析的makefile文件。

需要注意的是include指示符的书写顺序，因为在这些.d文件中已经存在规则。当一个Makefile

使用指示符include这些.d文件时，应该注意它应该出现在终极目标之后，以免.d文件中的规则被是Makefile的终极规则。关于这个前面我们已经有了比较详细的讨论。可参考 2.7 makefile文件的重建和 2.3 包含其他Makefile 两节的内容。

第四章：规则的命令

4 规则中书写命令

规则的命令由一些 shell 命令行组成，他们被一条一条的执行。规则中除了第一条紧跟在依赖列表之后使用分号隔开的命令以外，其它的每一行命令行必需一[Tab]字符开始。多个命令行之间可以有空行和注释行（所谓空行，就是没有包含任何字符的一行。如果以[Tab]键开始而其后没有命令的行，此行不是空行。是空命令行），在执行规则时他们将被忽略。

可能用户使用了多个不同的 shell。但是在 make 处理 Makefile 过程是，如果没有明确指定，那么对所有规则中命令行的解析使用“/bin/sh”来完成。

使用的 shell 决定了规则中的命令的语法和处理机制。当使用默认的“/bin/sh”时，命令中字符“#”到行末的内容被认为是注释。当然了“#”可以不在此行的行首，此时“#”之前的内容不会被作为注视处理。

另外在 make 解析 makefile 文件时，对待注释也是采用同样的处理方式。我们的 shell 脚本也一样。

4.1 命令回显

通常，make 在执行命令行之前会把要执行的命令行进行输出。我们称之为“回显”，就好像我们输入命令执行一样。

如果要执行的命令行以字符“@”开始，则 make 在执行时这个命令就不会被回显。典型的用法是我们在使用“echo”命令输出一些信息时。如：

@echo 开始编译XXX模块.....

当 make 执行时，将输出“开始编译 XXX 模块.....”这个信息。如果在命令行之前没有字符“@”，那么，make 的输出就是：

```
echo编译XXX模块.....
编译XXX模块.....
```

另外，如果使用make的命令行参数“-n”或“--just-print”，那么make执行时只显示所要执行的命令，但不会真正的去执行这些命令（可参考 8.7 make的命令行选项 一节）。只有在这种情况下make才会打印出所有make需要执行的命令，其中也包括了使用“@”字符开始的命令。这个选项对于我们调试Makefile非常有用，使用这个选项我们可以按执行顺序打印出Makefile中所有需要执行的命令。

而make参数“-s”或“--slient”则是禁止所有执行命令的显示，就好像所有的命令行均使用“@”开始一样。在Makefile中使用没有依赖的特殊目标“.SILENT”也可以禁止命令的回显（可参考 3.9 Makefile的特殊目标 一节），但是它的缺点是不如“@”灵活。因此我们在书写Makefile时，推荐使用“@”来控制命令的回显。

4.2 命令的执行

规则中，当目标需要被重建时。此规则所定义的命令将会被执行，如果是多行命令，那么 make 就为每一行命令使用一个独立的子 shell 去执行。因此，多行命令之间的执行是相互独立的，相互之间不存在依赖。

而在 Makefile 中书写在同一行中的多个命令属于一个完整的 shell 命令行，书写在独立行的一条命令是一个独立的 shell 命令行。**所以需要注意：在一个规则的命令中，命令行“cd”改变目录不会对其后的命令的执行产生影响。就是说其后的命令执行的工作目录不会是之前使用“cd”进入的那个目录。如果要实现这个目的，就不能把“cd”和其后的命令放在两行来书写。而应该把这两条命令写在一行上，用分号分隔。这样它们才是一个完整的 shell 命令行。如：**

```
foo : bar/lose
        cd bar; gobble lose > ../foo
```

如果希望把一个完整的 shell 命令行书写在多行上，需要使用反斜杠（\）来对处于多行的命令进行连接，表示他们是一个完整的 shell 命令行。例如上例我们以也可以这样书写：

```
foo : bar/lose
        cd bar;  \
        gobble lose > ../foo
```

make 对所有规则命令的解析使用环境变量“SHELL”所指定的那个程序，在 GNU make 中，默认的程序是“/bin/sh”。

不像其他绝大多数变量，它们的值可以直接从同名的系统环境变量那里获得。make 的环境变

量“SHELL”没有使用系统环境变量的定义。因为系统环境变量“SHELL”指定那个程序被用来作为用户和系统交互的接口程序，它对于不存在直接交互过程的 make 显然不合适。在 make 的环境变量中“SHELL”会被重新赋值；它作为一个变量我们也可以在 Makefile 中明确地给它赋值（指出解释程序的名字，当明确指定时需要使用完整的路径名。如“/bin/sh”），变量“SHELL”其值默认为“/bin/sh”。

（在 MS-DOS 下有些不同， MS-DOS 不存在 SHELL 环境变量。在 MS-DOS 下 make 的方法省略了，有兴趣地可以自行参考 info make）

4.3 并发执行命令

GUN make 可以同时执行多条命令。通常情况下，一个时刻只有一个命令在执行，下一个命令在当前命令执行完成之后才能够被执行。不过可以通过 make 的命令行选项“-j”或者“--job”来告诉 make 在同一时刻可以允许多条命令同时被执行（注意，在 MS-DOS 中此选项无效，因为它是单任务操作系统）。

如果选项“-j”之后存在一个整数，其含义是告诉 make 在同一时刻可允许同时执行命令的数目。这个数字被称为“job slots”。当“-j”选项之后没有出现一个数字时，那么同一时刻执行的命令数目没有要求。使用默认的“job slots”，其值为 1。表示 make 将串行的执行规则的命令（同一时刻只能有一条命令被执行）。

并行执行命令所带来的问题是显而易见：

1. 同一时刻的多个被执行的命令同时输出，造成输出到终端的信息的交替，显得凌乱。当出现问题是很难命令执行失败时很难根据命令的输出信息来定位错误。
2. 在同一时刻可能会存在多个命令执行进程读取标准输入，但是对于标注输入来设备来说，在同一时刻只能有一个进程访问输入设备。就是说在某个时间点，make 只能保证这个时刻正在执行的进程中的一个进程读取标准输入流，而其它的进程的标准输入流将变得无效。因此当同一时刻存在多个执行命令的进程需要读取标准输入流时其它的将会出输入流无效导致致命错误（通常此进程会得到操作系统的管道破裂信号而被终止）。

   这是因为：执行中的命令在什么时候会读取标准输入流（终端输入或重定向的标准输入）是不可预测的。而得到标准输入的顺序总是按照先来先获得的原则。那个命令首先被执行，那么它就可以首先得到标准输入设备。而其它后续需要获取标准输入设备的命令执行进程，由于不能得到标准输入而产生致命错误。再 Makefile 规则中如果存在很多命令需要读取标准输入设备，而它们又被允许并行执行时，就会出现这样的错误。

   对这个问题的解决。我们可以修改 Makefile 的规则命令使之在执行过程中不使用标准输

入设备。当然也可以实现在只存在一个命令在执行时会访问标准输入流的 Makefile。

3. 会导致make的递归调用出现问题。可参考 4.6 make的递归执行 一节。

Make在执行一个命令时，如果某一条命令执行失败（被一个信号中止，或非零退出），且该条命令产生的错误不可忽略（可参考 4.4 命令执行的错误 一节），那么其它的用于重建同一目标的命令执行将会被终止。此种情况下，如果make没有使用“-k”或“--keep-going”选项（可参考 8.7 make的命令行选项 一节），make将停止继续执行直接退出。另外：如果make在执行时，由某种原因（包括信号）被中止，此时它的子进程（那些执行规则命令行的shell子进程）正在运行，那么make将等到所有这些子进程结束之后才真正退出。

在执行 make 时，如果系统运行于重负荷状态下，我们需要控制（减轻）系统在执行 make 时的负荷。可以使用“-l”选项告诉 make 限制当前运行的任务的数量（make 所限制的只是它本身所需要占用的系统负载，而不能通过它去控制其它的任务所占用的系统负载）。“-l”或“--max-load”选项一般后边需要跟一个浮点数。例如：

-l 2.5

它的意思是告诉 make 当系统平均负荷高于 2.5 时，不再启动任何执行命令的子任务。不带浮点数的“-l”选项用于取消前面通“-l”给定的负荷限制。

更为准确一点就是：每一次，make 在启动一项任务之前，当前系统至少一项 make 的子任务正在运行。首先 make 会检查当前系统的负荷；如果当前系统的负荷高于通过“-l”选项指定的值，那么 make 就不会在其他任务完成之前启动任何任务。缺省情况下没有负荷限制。

4.4 命令执行的错误

通常；规则中的命令在运行结束后，make 会检测命令执行的返回状态，如果返回成功，那么就在另外一个子 shell 下执行下一条命令。规则中的所有命令执行完成之后，这个规则就执行完成了。如果一个规则中的某一个命令出错（返回状态非 0），make 就会放弃对当前规则的执行，也有可能会终止所有规则的执行。

在一些情况下，规则中的一个命令的执行失败并不代表规则执行的错误。例如我们使用“mkdir”命令来确保存在一个目录。当此目录不存在使我们就建立这个目录，当目录存在时那么“mkdir”就会执行失败。其实我们并不希望 mkdir 在执行失败后终止规则的执行。为了忽略一些无关命令执行失败的情况，我们可以在命令之前加一个减号“-”（在[Tab]字符之后），来告诉 make 忽略此命令的执行失败。命令中的“-”号会在在 shell 解析并执行此命令之前被去掉，shell 所解释的只是纯粹的命令，“-”字符是由 make 来处理的。例如对于“clean”目标我们就可以这么写：

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

其含义是：即使执行“rm”删除文件失败，make 也继续执行。

在执行 make 时，如果使用命令行选项“-i”或者“—ignore-errors”， make 将会忽略所有规则中命令执行执行的错误。没有依赖的特殊目标“.IGNORE”在 Makefile 中有同样的效果。但是“.IGNORE”的方式已经很少使用，因为它没有在命令行之前使用“-”字符方式灵活。

当使用 make 的“-i”选项或者使用“-”字符来忽略命令执行错误时，make 始终会把命令的执行结果作为成功来对待。但会提示错误信息，同时提示这个错误被忽略。

当没有使用这种方式来通知 make 忽略命令的执行错误时，而在错误发生时，就意味着定义这个命令的规则的目标不能被正确重建，同样，和此目标相关的其它目标也不会被正确重建。因此，由于先决条件不能建立，后续的命令将不会执行。

在发生这样情况时，一般 make 会立刻退出并返回一个非 0 状态，表示执行失败。像对待命令执行的错误一样，我们可以使用 make 的命令行选项“-k”或者“--keep-going”来通知 make，当出现错误时不立即退出，而是继续后续命令的执行。直到无法继续执行命令时才异常退出。例如：使用“-k”参数，在重建一个.o 文件目标时出现错误，make 不会立即退出。虽然 make 已经知道因为这个错误而无法完成终极目标的重建，但还是继续完成其它后续的依赖文件的重建。直到执行最后链接时才错误退出。

一般“-k”参数在实际中应用。它的用途主要在：当同时修改了工程中的多个文件后，“-k”参数可以帮助我们确认对那些文件的修改是正确的（可以被编译），那些文件的修改是不不正确的（不能正确编译）。例如我们修改了工程中的 20 个源文件，修改完成之后使用“-k”参数来进行 make，它可以一次性找出修改的 20 个文件中哪些是不能被编译的。

通常情况下，执行失败的命令一旦改变了它所在规则的目标文件，则这个改变了的目标可能不是一个被正确重建的文件。但是这个文件的时间戳已经被更新过了（这种情况也会发生在使用一个信号来强制中止命令执行的时候）。因此在下一次执行 make 时，由于时间戳更新它不会被再次重建。因此终极目标的重建很难保证是正确的。为了避免这种错误的出现，应该在一次 make 执行失败之后使用“make clean”来清除已经重建的所有目标，之后再执行 make。我们也可以让 make 自动完成这个动作，实现这个目的我们只需要 Makefile 中定义特殊目标“.DELETE_ON_ERROR”。但是这个做法存在不兼容。推荐的做法是：**在 make 执行失败时，修改错误之后执行 make 之前，使用“make clean”明确的删除第一次错误重建的所有目标。**

本节的最后，需要说明的是：虽然 make 提供了命令行选项来忽略命令执行的错误。建议对于此选项谨慎使用。因为在一个大型的工程中，可能需要对成千个源文件进行编译。编译过程中的任何一个文件的编译错误都不能被忽略。否则有可能最后完成的终极目标可能就是一个让你感到迷惑的东西，或者在运行时会产生一些莫名奇妙的现象。这需要程序员来保证其书写的 Makefile 的规则中的命令在执行时不会发生错误。特别需要注意哪些实现特殊目的规则的命令书写。当所有命令都可以被正确执行时，我们就没有必要为了避免一些讨厌的错误而使用“-i”选项，可以使用其它

的方式来实现。例如删除命令我们可以这样写： “$(RM)”或者“rm -f”，创建目录的命令可以这样写：“mkdir –p ”等等。

4.5 中断make的执行

make 在执行命令时如果收到一个致命信号（结束 make），make 将会删除命令重建的规则目标文件。其依据是此目标文件的当前时间戳是否和 make 开始时的时间戳相同。

删除这个目标文件的目的是为了确保下一次 make 时目标文件能够被正确重建。其原因我们上一节已经有所讨论。假设正在编译文件是键入“Ctrl-c”，在这时编译器已经开始写文件“foo.o”，但是“Ctrl-c”产生的信号关闭了编译器。这种情况时文件“foo.o”可能是不完整的，但这个内容不完整的“foo.o”文件的时间戳比源程序‘foo.c’的时间戳新。如果在 make 收到终止信号后不删除文件“foo.o”而直接退出，那么下次执行 make 时 make 将认为该文件已是最新而不会去重建文件它。最后在链接程序生成终极目标时可能由于某一个.o 文件的不完整，导致出现一些奇怪的令人难以理解的错误信息。

同时，我们可以在 Makefile 中将一个目标文件作为特殊目标“.PRECIOUS”的依赖，这样可以防止 make 在重建这个目标时异常终止时对这个目标文件的删除动作。每次 make 在重建一个目标之前，首先判断该目标文件是否出现在特殊目标“.PRECIOUS”的依赖列表中，决定在终止信号发生时需不需要删除这个目标文件。不删除这种目标文件的原因可能是：1. 目标的重建动作是一个原子的不可被中断的过程；2. 目标文件的存在仅仅为了记录其重建时间（不关心其内容无）；3. 这个目标文件必须一直存在来防止其它麻烦。

4.6 make的递归执行

make 的递归调用指的是：在 Makefile 中使用“make”作为一个命令来执行本身或者其它 makefile 文件。递归调用在一个村在有多级子目录的项目中非常有用。例如，当前目录下存在一个“subdir”子目录，这个子目录中有描述这个目录编译规则的 makefile 文件，在执行 make 时需要从上层目录（当前目录）开始并完成它所有子目录的编译。那么在当前目录下可以使用这样一个规则来实现对它的子目录的编译：

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

其等价于规则：

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

我们对这两个规则的命令进行简单说明，规则中“$(MAKE)”是对变量“MAKE”（下一小节

将详细讨论）的引用（关于变量可参考 第五章 Makefile中的变量 ）。第一个规则命令的意思是：进入子目录，然后在子目录下执行make。第二个规则时用了make的“-C”选项，同样是首先进入子目录而后再执行make。

书写这样的规则应该来说对于我们不是什么大问题，但是其中有一些需要我们深入了解的东西。首先需要了解它如何工作、上层make（在当前目录下运行的make进程）和下层make（subdir目录下运行的make进程）之间存在的联系。也许会发现这两个规则的实现，使用伪目标更能提高效率（可参考 3.6 Makefile伪目标 一节）。

在 make 的递归调用中，需要了解变量“CURDIR”，此变量代表了 make 当前的工作路径。如果使用“-C”选项进入一个子目录后，此变量将被重新赋值。总之，如果在 Makefile 中没有对此变量进行显式的赋值操作，那么它代表 make 的当前工作目录。我们也可以在 Makefile 为这个变量赋一个新的值。此时这变量将不再代表 make 的工作目录。

4.6.1 变量MAKE

在使用 make 的递归调用时，在 Makefile 中规则的命令行中应该使用变量“MAKE”来代替直接使用“make”。像上一小节的例子：

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

变量“MAKE”的值是“make”程序的文件名。如果其值为“/bin/make”那么上边规则的命令就为“cd subdir && /bin/make”。这样做的好处是：当我们使用一个其它版本的 make 程序时，可以保证最上层使用的 make 程序和其子目录下执行的 make 保持一致。

另外使用此变量的另外一个特点是：当规则命令行中变量MAKE是，它可以改变make的“-t”（“--touch”），“-n”（“--just-print”）和“-q”（“--question”）命令行选项的效果。它所实现的功能和在规则中命令行首使用字符“+”的效果相同（可参考 8.3 替代命令的执行 一节）。

在规则的命令行中使用“make”代替了“$(MAKE)”以后，上例子规则的命令行为：“cd subdir && make”。在我们执行“make -t”（“-t”选项用来更新所有目标的时间戳，而不执行任何规则的命令，参考 8.7 make的命令行选项 一节），结果是仅仅创建一个名为“subsystem”的文件，不会进入到目录“subdir”去更新此目录下文件的时间戳。我们使用“-t”命令行参数的初衷是对规则中的目标文件的时间戳进行更新。而如果使“cd subdir && $(MAKE)”作为规则的命令行，执行“make -t”就可以实现我们的初衷。

变量“MAKE”的这个特点是：在规则的命令行中如果使用变量“MAKE”，标志“-t”、“-n”和“-q”在这个命令的执行中不起作用。尽管这些选项是告诉 make 不执行规则的命令行，但包含变量“MAKE”的命令行除外，它们会被正常执行。同时，执行 make 时的命令行选项参数被通过一个变量“MAKEFLAGS”传递给子目录下的 make 程序。

例如，当使用 make 的命令行选项“-t”来更新目标的时间戳或者“-n”选项打印命令时，这些选项将会被赋值给变量“MAKEFLAGS”被传递到下一级的 make 程序中。在下一级子目录中执行的 make，这些选项会被附加作为 make 的命令行参数来执行，和在此目录下使用“make -t”或者“make -n”有相同的效果。

4.6.2 变量和递归

在 make 的递归执行过程中，上层 make 可以明确指定将一些变量的定义通过环境变量的方式传递给子 make 过程。没有明确指定需要传递的变量，上层 make 不会将其所执行的 Makefile 中定义的变量传递给子make过程。使用环境变量传递上层所定义的变量时，上层所传递给子make过程的变量定义不会覆盖子 make 过程所执行 makefile 文件中的同名变量定义。

如果子make过程所执行Makefile中存在同名变量定义，则上层传递的变量定义不会覆盖子Makefile中定义的值。就是说如果上层make传递的变量和子make所执行的Makefile中存在重复的变量定义，则以子Makefile中的变量定义为准。除非使用make的“-e”选项（参考 8.7 make 的命令行选项 一节）。

我们在本届的第一段中提到，当上层 make 过程要将所执行的 Makefile 中的变量传递给子 make 过程时，需要明确地指出。在 GNU make 中，实现此功能的指示符是“export”。当一个变量使用“export”进行声明后，变量和变量的值将被加入到当前工作的环境变量中，以后 make 所执行的所有规则的命令都可以使用这个变量。而当没有使用指示符“export”对任何变量进行声明的情况下，上层 make 只将那些已经初始化的环境变量（在执行 make 之前已经存在的环境变量）和使用命令行指定的变量（如命令“make CFLAGS +=-g”或者“make –e CFLAGS +=-g”）传递给子 make 程序，通常这些变量由字符、数字和下划线组成。需要注意的是：有些 shell 不能处理那些名字中包含（除字母、数字、下划线以外）其他字符的变量。

两个特殊的变量“SHELL”和“MAKEFLAGS”除非使用指示符“unexport”对它们进行声明，否则在整个 make 的执行过程中它们会始终被自动的传递给子 make。另外一个变量“MAKEFILES”，如果此变量有值（不为空）那么同样他会被自动的传递给子 make。在没有使用关键字“export”声明的变量，make 执行时不会被自动传递给子 make 过程，因此下层 Makefile 中可以定义和上层同名的变量，这样不会引起变量定义冲突。

上层 Makefile 中定义的某一个变量需要传递给子 make 时，应该在上层 Makefile 中使用指示符“export”对此变量进行声明。格式如下：

export VARIABLE ...

当不希望将一个变量传递给子 make 时，可以使用指示符“unexport”来声明这个变量。格式如下：

unexport VARIABLE ...

在以上的两种格式，**指示符“export”或者“unexport”的参数（变量部分），如果它是对一个变量或者函数的引用，这些变量或者函数将会被立即展开。并赋值给export或者unexport的变量**（关于变量展开的过程可参考 第 5 章 Makefile中的变量）。例如：

```
Y = Z
export X=$(Y)”
```

其实就是“export X=Z”。在这里进行展开是为了保证传递给子 make 的此变量的值有效。

“export”更方便的用法是在定义此变量时同时对它进行声明。如下的几个例子：

1.
```
export VARIABLE = value
```

等效于：

```
VARIABLE = value
export VARIABLE
```

2.

```
export VARIABLE := value
```

等效于：

```
VARIABLE := value
export VARIABLE
```

3.
```
export VARIABLE += value
```

等效于：

```
VARIABLE += value
export VARIABLE
```

其实在 Makefile 中指示符“export”和“unexport”的功能和在 shell 下功能相同。

另外一个不带任何参数的指示符“export”指示符：

export

含义是将此 Makefile 中定义的所有变量传递给子 make 过程。如果不需要传递其中一个变量，可以使用指示符“unexport”来声明这个变量，这个被声明的变量就不会被传递给子 make。使用“export”将所有定义的变量传递给子 Makefile 时，那些名字中包含其它字符（除字母、数字和下划线以外的字符）的变量可能不会被传递给子 make，对这类特殊命名的变量传递需要明确的使用“export”指示符对它进行声明。

需要说明的是：单独使用“export”来导出所有变量的行为是老版本 GNU make 所默认的。但是在新版本的 GNU make 中取消了这一默认的行为。因此在编写和老版本 GNU make 兼容的 Makefile 时，需要使用特殊目标“.EXPORT_ALL_VARIABLES”来代替“export”，此特殊目标的功和不带参数的“export”相同。它会被老版本的 make 忽略，只有新版本的 make 能够识别这个特殊目标。

因为，如果在老版本的 GNU make 中使用指示符“export”，将会出现语法错误。例如为了和老版本兼容可以这样来声明一些变量：

```
.EXPORT_ALL_VARIABLES:
VARIABLE1=var1
VARIABLE2=var2
```

这样对于不同版本的 make 来说都是兼容的，其含义是将特殊目标“.EXPORT_ALL_VARIABLES”的依赖中的所有变量全部传递给子 make。

和指示符“export”相似，也可以使用单独的“unexport”指示符来禁止一个变量的向下传递。这一动作也是现行版本 make 所默认的，因此我们就没有必要在上层的 Makefile 中使用它。在多级的 make 递归调用中，我么可以在中间的 Makefile 中使用它来限制上层传递来的变量再向下传递。需要明确的是，不能使用“export”或者“unexport”来实现命令中使用的变量控制功能。就是说，不能做到用这两个指示符来限制某个（些）变量在执行特定命令时有效，而对于其它的命令则无效。在 Makefile 中，最后一个出现的指示符“export”或者“unexport”决定整个 make 运行过程中变量是否进行传递。

在多级递归调用的 make 执行过程中。变量“MAKELEVEL”代表了调用的深度。在 make 一级级的执行过程中变量“MAKELEVEL”的值不断的发生变化，通过它的值我们可以了解当前 make 递归调用的深度。最上一级时“MAKELEVEL”的值为“0”、下一级时为“1”、再下一级为“2”.....例如：

Main 目录下的 Makefile 清单如下：

```
#maindir Makefile
.........
.........
.PHONY :test
test:
        @echo "main makelevel = $(MAKELEVEL)"
        @$(MAKE) –C subdir dislevel

#subdir Makefile
...........
...........
.PHONY : test
test :
        @echo "subdir makelevel = $(MAKELEVEL)"
```

在 maindir 目录下执行“make test”。将显式如下信息：

```
main makelevel = 0
make[1]: Entering directory `/...../ subdir '
subdir makelevel = 1
make[1]: Leaving directory `/...../ subdir '
```

在主控的 Makefile 中 MAKELEVEL 为“0”，而在 subdir 的 Makefile 中，MAKELEVEL 为“1”。

这个变量主要用在条件测试指令中。例如：我们可以通过测试此变量的值来决定是否执行递归方式的 make 调用或者其他方式的 make 调用。我们希望一个子目录必须被上层 make 进行调用才能可以执行此目录下的 Makefile，而不允许直接在其所在的目录下执行 make。我们可以这样实现：

```
.......
$(ifeq $(MAKELEVEL),0)
all : msg
else
all : other
endif

......
......

msg:
    @echo "Can not make in this directory!"
......
......
```

当在包含次条件判断的 Makefile 所在的目录下执行 make 时，将会得到提示“***Can not make in this directory!***”。

4.6.3 命令行选项和递归

在 make 的递归执行过程中。最上层（可以称之为主控）make 的命令行选项“-k”、“-s”等被自动的通过环境变量“MAKEFLAGS”传递给子 make 进程。变量“MAKEFLAGS”的值会被主控 make 自动的设置为包含执行 make 时的命令行选项的字符串。在主控执行 make 时使用“-k”和“-s”选项，那么“MAKEFLAGS”的值就为“ks”。子 make 进程处理时，会把此环境变量的值作为执行的命令行选项，因此子 make 进程就使用“-k”和“-s”这两个命令行选项。

同样，在执行 make 时命令行中给定了一个变量的定义（如“make CFLAGS+=-g”），此变量和它的值（CFLAGS+=-g）也会借助环境变量“MAKEFLAGS”传递给子 make 进程。可以借助 make 的环境变量“MAKEFLAGS” 传递我们在主控 make 所使用的命令行选项给子 make 进程。**需要注意的是有几个特殊的命令行选项例外，分别是：“-C”、“-f”、“-o”和“-W”。**这些

命令行选项不会被赋值给变量“MAKEFLAGS”。

Make 命令行选项中一个比较特殊的是“-j”选项。在支持这个选项的操作系统上，如果给它指定了一个数值“N”（多任务的系统 unix、Linux 支持，MS-DOS 不支持），那么主控 make 和子 make 进程会在执行过程使用通信机制来限制系统在同一时刻（包括所有的递归调用的 make 进程，否则，将会导致 make 任务的数目数目无法控制而使别的任务无法到的执行）的任务的执行数目不大于“N”。另外，当使用的操作系统不能支持 make 执行过程中的父子间通信，那么无论在执行主控 make 时指定的任务数目“N”是多少，变量“MAKEFLAGS”中选项“-j”的数目会都被设置为“1”，这样可以确保系统正常运转。

执行多级的 make 调用时，如果不希望“MAKEFLAGS”的值传递给子 make，就需要在执行子 make 时对它重新进行赋值。例如：

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

此规则取消了子 make 执行式的命令行选项（将变量的值赋为空）。

在执行make的同时可以通过命令行来定义一个变量，像上例中的那样；前边已经提到过，这种变量是借助环境“MAKEFLAGS”来传递给多级调用的子make进程的。其实真正的命令行中的变量定义 是通过另外一个变量“MAKEOVRRIDES”来记录的，变量“MAKEFLAGS”引用此变量，因而命令行中的变量定义就可以被记录在环境变量“MAKEFLAGS”中被传递下去。当不希望将上层make在命令行中定义的变量传递给子make时，就可以在上层Makefile中把“MAKEOVERRIDES”赋空来实现（MAKEOVERRIDES= ）。这种方式一般很少使用，建议非万不得已您还是最好不使用这种方式（为了和POSIX2.0 兼容，当Makefile中出现“.POSIX”这个特殊的目标时，在上层Makefile中修改变量“MAKEOVERRIDES”对子make不会产生任何影响）。另外一些系统中对环境变量的长度存在一个上限，因此当“MAKEFLAGS”的值超过一定的数目时，执行过程出现了类似“Arg list too long”的错误提示。

历史原因，在 make 中存在另外一个和“MAKEFLAGS”相类似的变量“MFLAGS”。现行版本中此变量保留的原因是和老版本的兼容需要。和“MAKEFLAGS”不同点是：1. 此变量在 make 的递归调用时不包含命令行选项中的变量定义部分（就是说此变量的定义没有包含对“MAKEOVERRIDES”的引用）；2. 此变量的值（除为空的情况）是以“-”开始的，而“MAKEFLAGS”的值只有在长命令选项格式（如：“--warn-undefined-variables”）时才以“-”开头。传统得此变量一般被明确的使用在 make 递归调用命令中。像下边那样：

```
subsystem:
        cd subdir && $(MAKE) $(MFLAGS)
```

在现行的 make 版本中，变量“MFLAGS”已经成为一个多余部分。在书写和老版本 make 兼容

的 Makefile 时可能需要这个变量。当然它在目前的版本上也能够正常的工作。

在某些特殊的场合，可能需要为所有的make进程指定一个统一的命令行选项。比如说需要给所有的运行的make指定“-k”选项（参考 8.7 make的命令行选项 一节）。实现这个目的，我们可以在执行make之前设置一个系统环境变量（存在于当前系统的环境中）“MAKEFLAGS=k”，或者在主控Makefile中将它的值赋为“k”。需要注意的是：不能通过变量“MFLAGS”来实现。

make 在执行时，首先将会对变量“MAKEFLAGS”的值（系统环境中或者在 Makefile 中设置的）进行分析。当变量的值不是以连字符（“-”）开始时，将变量的值按字分开，字之间使用空格分开。将这些字作为命令行的选项对待（除了选项“-C”、“-f”、“-h”、“-o”和“-W”以及他们的长格式，如果其中包含无效的选项也不会提示错误）。

最后需要强调的是：当把“MAKEFLAGS”设置到你的系统环境变量中时，要小心谨慎！将一些调试选项或者特殊选项设置为此变量值的一部分，在执行 make 时，会对 make 的正常执行产生影响，甚至是破坏性的影响。例如变量“MAKEFLAGS”中包含选项“t”、“n”、“q”这三个的任何一个，你在执行 make 时产生的结果可能并不是你所要达到的目的。建议大家最好不要随便更改“MAKEFLAGS”的值，更不要把它设置为系统的环境变量来使用。否则可能会产生一些奇怪甚至让你感到不解的现象。

4.6.4 -w选项

在多级 make 递归调用过程中，选项“-w”或者“--print-directory”可以让 make 在开始编译一个目录之前和完成此目录的编译之后给出相应的提示信息，方便开发人员能够跟踪 make 的执行过程。例如，在目录“/u/gnu/make”目录下执行“make -w”，将会看到如下的一些信息：在开始执行之前我们将看到：

make: Entering directory `/u/gnu/make'.

而在完成之后我们同样将会看到：

make: Leaving directory `/u/gnu/make'.

通常，选项“-w”会被自动打开。在主控Makefile中当如果使用“-C”参数来为make指定一个目录或者使用“cd”进入一个目录时，“-w”选项会被自动打开。主控make可以使用选项“-s”（“--slient”）来禁止此选项的自动打开。另外，make的命令行选项“--no-print-directory”，将禁止所有关于目录信息的打印。可参考 8.7 make的命令行选项 一节

4.7 定义命令包

在书写Makefile时，可能有多个规则会使用相同的一组命令。就像c语言程序中需要经常使用

到函数“printf”。这时我们就会想能不能将这样一组命令进行类似c语言函数一样的封装，以后在我们需要用到的地方可以通过它的名字（c语言中的函数名）来对这一组命令进行引用。这样就可减少重复工作，提高了效率。在GNU make中，可以使用指示符“define”来完成这个功能（关于指示符“define”可参考 5.8 多行定义 一节）。通过“define”来定义这样一组命令，同时用一个变量（作为一个变量，不能和Makefile中其它常规的变量命名出现冲突）来代表这一组命令。通常我们把使用“define”定义的一组命令称为一个命令包。定义一个命令包的语法以“define”开始，以“endef”结束，例如：

```
define run-yacc
yacc $(firstword $^)
mv y.tab.c $@
endef
```

这里，“run-yacc”是这个命令包的名字。在“define”和“endef”之间的命令就是命令包的主体。需要说明的是：使用“define”定义的命令包中，命令体中变量和函数的引用不会展开。命令体中所有的内容包括“$”、“（”、“)”等都是变量“run-yacc”的定义。它和c语言中宏的使用方式一样。关于变量可参考 第 5 章 Makefile中的变量

例子中，命令包中第一个命令是对引用它所在规则中的第一个依赖文件（函数“firstword”，可参考 7.2 文本处理函数 一节）运行yacc程序。yacc程序总是生成一个命名为“y.tab.c”的文件。第二行的命令就是把这个文件名改为规则目标的名字。

定义了这样一个命令包后，后续应该如何使用它？前面已经提到，命令包是使用一个变量来表示的。因此我们就可以按使用变量的方式来使用它。当在规则的命令行中使用这个变量时，命令包所定义的命令体就会对它进行替代。由于使用“define”定义的变量属于递归展开式变量（参考 5.2 两种变量定义（赋值） 一节），因此，命令包中所有命令中对其它变量的引用，在规则被执行时会被完全展开。例如这样一个规则：

```
foo.c : foo.y
        $(run-yacc)
```

此规则在执行时，我们来看一下命令包中的变量的替换过程：1. 命令包重中的“$^ ”会被“foo.y”替换；2. “$@”被“foo.c”替换。大家应该对“$< ”和“$@”不陌生吧，如果陌生可以 参考 9.5.1 自动化变量 一小节。

当在一个规则中引用一个已定义的命令包时，命令包中的命令体会被原封不动的展开在引用它的地方（和 c 语言中的宏一样）。这些命令就成为规则的命令。因此我们也可在定义命令包时使用前缀来控制单独的一个命令行（例如“@”，“-”和“+”）。例如：

```
define frobnicate
      @echo "frobnicating target $@"
```

```
        frob-step-1 $< -o $@-step-1
        frob-step-2 $@-step-1 -o $@
endef
```

此命令包的第一行命令执行前不会被回显，其它的命令在执行前都被回显。

另一方面，如果一个规则在引用此命令包之前使用了命令的前缀字符。那么这个前缀字符将会被添加到命令包定义的所有命令行之中。例如：

```
frob.out: frob.in
        @$(frobnicate)
```

这个规则执行时不会回显任何将要执行的命令。命令行回显可参考 4.1 命令回显 一节

4.8 空命令

有时可能存在这样的一个需求，需要定义一个什么也不做的规则（没有命令行）。前面我们已经看到过这样的用法（参考 2.8 重载另外一个Makefile）。这样的规则，只有目标文件（可以存在依赖文件）而没有命令行。可以像这样定义：

```
target: ;
```

这就是一个空命令的规则，为目标“target”定义了一个空命令。也可以命令行的格式来定义空命令，需要注意的是命令行必须以[Tab]字符开始。一般在定义空命令时，建议不使用命令行的方式，因为看起来空命令行和空行在感觉上没有区别。

大家会奇怪为什么要定义一个没有命令的规则。其唯一的原因是，使用空命令行可以防止make在执行时图重建这个目标而查找隐含命令（包括了使用隐含规则中的命令和“.DEFAULT”指定的命令。关于隐含规则可参考 第 9 章 使用隐含规则）。这一点和伪目标有相同之处（可参考3.6 Makefile伪目标一节）。在使用空命令的目标时，需要说明的是：实现一个没有实际文件的目标，这个目标只是作为一个标签，来完成它的依赖文件的重建动作。实现这个目的，首先应该想到伪目标而不是空命令目标。因为一个实际不存在的目标文件的依赖文件，可能不会被正确的重建。

所以，对于空命令规则，最好不要给它指定依赖文件。避免特殊情况下产生错误的情况。定义一个空命令规则，建议使用上例的格式。

第五章：Makefile中的变量

5 使用变量

在 Makefile 中，变量就是一个名字（像是 C 语言中的宏），代表一个文本字符串（变量的值）。在 Makefile 的目标、依赖、命令中引用一个变量的地方，变量会被它的值所取代（与 C 语言中宏

引用的方式相同，因此其他版本的 make 也把变量称之为“宏”)。在 Makefile 中变量的特征有以下几点：

1. Makefile 中变量和函数的展开（除规则的命令行以外），是在 make 读取 makefile 文件时进行的，这里的变量包括了使用“= ”定义和使用指示符“define”定义的。
2. 变量可以用来代表一个文件名列表、编译选项列表、程序运行的选项参数列表、搜索源文件的目录列表、编译输出的目录列表和所有我们能够想到的事物。
3. 变量名是不包括“:”、“# ”、“= ”、前置空白和尾空白的任何字符串。需要注意的是，尽管在GNU make中没有对变量的命名有其它的限制，但定义一个包含除字母、数字和下划线以外的变量的做法也是不可取的，因为除字母、数字和下划线以外的其它字符可能会在以后的make版本中被赋予特殊含义，并且这样命名的变量对于一些shell来说不能作为环境变量使用（前面已经在 4.6.2 变量和递归 一小节中提到)。
4. 变量名是大小写敏感的。变量“foo”、“Foo”和“FOO”指的是三个不同的变量。Makefile 传统做法是变量名是全采用大写的方式。推荐的做法是在对于内部定义定义的一般变量(例如：目标文件列表 objects)使用小写方式，而对于一些参数列表(例如：编译选项 CFLAGS)采用大写方式，这并不是要求的。但需要强调一点：对于一个工程，所有 Makefile 中的变量命名应保持一种风格，否则会显得你是一个蹩脚的程序员（就像代码的变量命名风格一样)。
5. 另外有一些变量名只包含了一个或者很少的几个特殊的字符(符号)。称它们为自动化变量。像“$< ”、“$@”、“$?”、“$* ”等。参考 9.5.3 自动化变量 一小节

5.1 变量的引用

当我们定义了一个变量之后，就可以在 Makefile 的很多地方使用这个变量。变量的引用方式是：使用“$（VARIABLE_NAME)”或者“${ VARIABLE_NAME } ”来引用一个变量的定义。例如：“$(foo) ”或者“${ foo} ”就是取变量“foo”的值。美元符号“$”在 Makefile 中有特殊的含义，所有在命令或者文件名中使用“$”时需要用两个美元符号“$$”来表示。对一个变量的引用可以在 Makefile 的任何上下文中，目标、依赖、命令、绝大多数指示符和新变量的赋值中。这里有一个例子，其中变量保存了所有.o 文件的列表：

```
objects = program.o foo.o utils.o
program : $(objects)
        cc -o program $(objects)

$(objects) : defs.h
```

变量引用的展开过程是严格的文本替换过程，就是说变量值的字符串被精确的展开在此变量被引用的地方。因此规则：

```
foo = c
prog.o : prog.$(foo)
        $(foo) $(foo) -$(foo) prog.$(foo)
```

被展开后就是：

```
prog.c : prog.c
        cc -c prog.c
```

通过这个例子我们可以看到变量的展开过程完全和 c 语言中的宏展开的过程一样，是一个严格的文本替换过程。上例中在变量“foo”被展开过程中，其值中的前导空格会被忽略。这里举这个例子的目的是为了让我们更清楚地了解变量的展开方式，而不是建议大家按照这样的方式来书写 Makefile。在实际书写时，千万不要使用这种方式。否则将会给你带来很多不必要的麻烦。

注意：Makefile 中在对一些简单变量的应用，我们也可以不使用“()”和“{}”来标记变量名，而直接使用“$x”的格式来实现，此种用法仅限于变量名为单字符的情况。另外自动化变量也使用这种格式。对于一般多字符变量的引用必须使用括号了标记，否则 make 将把变量名的首字母作为作为引用（“$PATH”在 Makefile 中实际上是“$(P)ATH”）。这一点和 shell 中变量的引用方式不同。shell 中变量的引用可以是“${xx}”或者“$xx”格式。但在 Makefile 中多字符变量名的引用只能是“$(xx)”或者“${xx}”格式。

一般在我们书写 Makefile 时，各部分变量引用的格式我们建议如下：

1. make 变量（Makefile 中定义的或者是 make 的环境变量）的引用使用“$(VAR)”格式，无论“VAR”是单字符变量名还是多字符变量名。
2. 出现在规则命令行中 shell 变量（一般为执行命令过程中的临时变量，它不属于 Makefile 变量，而是一个 shell 变量）引用使用 shell 的“$tmp”格式。
3. 对出现在命令行中的 make 变量我们同样使用“$(CMDVAR)” 格式来引用。

例如：

```
# sample Makefile
......
SUBDIRS := src foo

.PHONY : subdir
Subdir :
@for dir in $(SUBDIRS); do   \
        $(MAKE) –C $$dir || exit 1; \
      done

......
```

5.2 两种变量定义（赋值）

GNU make 中，一个变量的定义有两种方式（或者称为风格）。我们把这两种方式定义的变量可以看作变量的两种不同风格。变量的这两种不同的风格的区别在于：1. 定义方式；2. 展开时机。下边我们分别对这两种不同的风格进行详细地讨论。

5.2.1 递归展开式变量

第一种风格的变量就是递归方式扩展的变量。这一类型的变量的定义是通过“= ”（参考 5.5 如何设置变量 一节）或者使用指示符“define”（参考 5.8 多行定义 一节）定义的变量。对这种变量的引用，在引用的地方是严格的文本替换过程，此变量值的字符串原模原样的出现在引用它的地方。如果此变量定义中存在对其他变量的引用，这些被引用的变量会在它被展开的同时被展开。就是说在变量定义时，变量值中对其他变量的引用不会被替换展开。而是，变量在引用它的地方进行替换展开的同时，它所引用的其它变量才会被替换展开。语言的描述可能比较晦涩，让我们来看一个例子：

```
foo = $(bar)
bar = $(ugh)
ugh = Huh?

all:;echo $(foo)
```

执行“make”将会打印出“Huh?”。整个变量的替换过程时这样的：首先“$(foo)”被替换为“$(bar)”，接下来“$(bar)”被替换为“$(ugh)”，最后“$(ugh)”被替换为“Hug?”。整个替换的过程是在执行“echo $(foo)”是进行的。

这种类型的变量是其它版本的 make 所支持的类型。我们可以把这种类型的变量称为“递归展开”式变量。此类型的变量存有它的优点同时也存在其缺点。**其优点是：**

这种类型变量的定义时，可以引用其它的之前没有定义的变量（可能在后续部分定义，或者是通过 make 的命令行选项传递的变量）。看一个这样的例子：

```
CFLAGS = $(include_dirs) -O
include_dirs = -Ifoo -Ibar
```

“CFLAGS”会在命令中被展开为“-Ifoo -Ibar -O”。我们可以看到在“CFLAGS”定义中使用到了之后定义的变量“include_dirs”。

其缺点是：

1. **使用此风格的变量定义，可能会由于出现变量的递归定义而导致 make 陷入到无限的变量展开过程中，最终使 make 执行失败。**例如，接上边的例子，我们给这个变量追加值：

```
CFLAGS = $(CFLAGS) –O
```

它将会导致 make 进入对变量“CFLAGS”的无限展过程中去（这种定义就是变量的递归定义）。因为一旦后续同样存在对“CLFAGS”定义的追加，展开过程将是套嵌的、不能终止的（在发生这种情况时，make 会提示错误信息并结束）。一般在书写 Makefile 时，使用这种追加变量值的方法也很少使用（也不是我们推荐的方式）。我们可以来看另外一个例子：

```
x = $(y)
y = $(x) $(z)
```

这种情况下同样会导致 make 陷入到无限的变量展开过程中。因此我们在使用这种风格的变量，当一个变量定义需要引用其它的变量时需要特别的注意，防止变量的展开进入无限的循环。当出现这样的错误时，首先检查你的 Makefile 中变量是否出现了递归定义。这是它的一个缺点。

2. **第二个缺点：这种风格变量的定义中如果使引用了某一个函数，那么函数总会在其被引用的地方被执行。**

这是因为这种风格变量的定义中，对函数引用的替换展开发生在展开它自身的时候，而不是在定义它的时候。这样所带来的问题是，可能可能会使 make 的执行效率降低，同时对某些变量和函数的引用出现问题。特别是当变量定义中引用了“shell”和“wildcard”函数的情况，可能出现不可控制或者难以预料的错误，因为我们无法确定它在何时会被展开。

5.2.2 直接展开式变量

为了避免“递归展开式”变量存在的问题和不方便。在 GNU make 中可以使用另外一种风格的变量，我们将它称为“直接展开”式。这种风格的变量使用“:= ”来定义变量。在使用“:= ”定义变量时，变量值中对另外变量的引用或者函数的引用在定义时被展开（对变量进行替换）。所以在变量被定义以后就是一个实际所需要定义的文本串，其中不再包含任何对其它变量的引用。因此

```
x := foo
y := $(x) bar
x := later
```

就等价于：

```
y := foo bar
x := later
```

需要注意的是：此风格变量在定义时就完成了对所引用的变量的展开，因此它不能实现对其后定义变量的引用。例如：

```
CFLAGS := $(include_dirs) -O
include_dirs := -Ifoo -Ibar
```

由于在变量“include_dirs”的定义出现在“CFLAGS”定义之后。因此在“CFLAGS”的定义中，“include_dirs”的值为空。“CFLAGS”的值为“-O”而不是“-Ifoo -Ibar -O”。这一点也是直接展开式和递归展开式变量的不同点。注意这里的两个变量都是“直接展开”式的。大家不妨试试将其中某一个变量使用递归展开式定义后看一下又会出现什么样的结果。

下边让我们来看一个复杂一点的例子。分析一下直接展开式变量定义（:= ）的用法，这里也用到了make的shell函数和变量“MAKELEVEL”（此变量在make的递归调用时代表make的调用深度，参考 4.6.2 变量和递归 一小节）。

其中包括了 make 的函数、条件表达式和一个系统变量“MAKELEVEL”的使用：

```
ifeq (0,${MAKELEVEL})
cur-dir     := $(shell pwd)
whoami      := $(shell whoami)
host-type := $(shell arch)
MAKE := ${MAKE} host-type=${host-type} whoami=${whoami}
endif
```

第一行是一个条件判断，说明如果是顶层 Makefile，就定义下列变量。否则不定义任何变量。第二、三、四、五行分别定义了一个变量，在进行变量定义时对引用到的其它变量和函数展开。最后结束定义。由于直接展开式的这个优点我们就可以书写这样一个规则：

```
${subdirs}:
        ${MAKE} cur-dir=${cur-dir}/$@ -C $@ all
```

它实现了在不同子目录下变量“cur_dir”使用不同的值（为当前工作目录）。

在复杂的 Makefile 中，推荐使用直接展开式变量。因为这种风格变量的使用方式和大多数编程语言中的变量使用方式基本上相同。它可以使一个比较复杂的 Makefile 在一定程度上具有可预测性。而且这种变量允许我们利用之前所定义的值来重新定义它（比如使用某一个函数来对它以前的值进行处理并重新赋值），此方式在 Makefile 中经常用到。尽量避免和减少递归方式的变量的使用。

5.2.3 如何定义一个空格

使用直接扩展式变量定义我们可以实现将一个前导空格定义在变量值中。一般变量值中的前导空格字符在变量引用和函数调用时被丢弃。直接展开式变量在定义时对引用的其它变量或函数进行展开的特点，使我们可以实现在一个变量中包含前导空格并在引用此变量时对空格加以保护。像这样：

```
nullstring :=
space := $(nullstring) # end of the line
```

这里，变量“space”就正好表示一个一个空格。“space”定义的行中的注释在这里使得我们的目的更清晰（明确地描述一个空格字符比较困难），明确的指定我们需要的是一个空格。这是一个很好地实现方式。使用变量“nullstring”标明变量值的开始，采用“#”注释来结束，中间是一个空格字符。

make 对变量进行处理时变量值中尾空格是不被忽略的，因此定义一个包含一个或者多个空格的变量定义时，上边的实现就是一个简单并且非常直观的方式。但是需要注意的是对于不包含尾空格的变量的定义，就不能随便使用几个空格之后，在同行中放置它的注释内容。这是千万需要注意的。例如下边的做法就是不正确的：

```
dir := /foo/bar      # directory to put the frobs in
```

变量“dir”的值是“/foo/bar　　”（后面有 4 个空格），这可能并不是你想要实现的。假如一个特定的文件以它作为路径来表示“$(dir)/file”，那么大错特错了。

这里顺便提醒大家，在书写 Makefile 时。注释内容推荐书写在独立的一行或者多行，这样就可以防止出现这种意外情况的发生，而且注释行独立的行书写时也使得你的 Makefile 更加清晰，便于别人的预读。对于特殊的定义，比如定义包含一个或者多个空格空格的变量时进行详细地说明和注释。

5.2.4 “?=”操作符

GNU make 中，还有一个被称为条件赋值的赋值操作符“?=”。被称为条件赋值是因为：只有此变量在之前没有赋值的情况下才会对这个变量进行赋值。例如：

```
FOO ?= bar
```

其等价于：

```
ifeq ($(origin FOO), undefined)
    FOO = bar
endif
```

含义是：如果变量“FOO”在没有定义过，就给它赋值“bar”。否则不改变它的值。

5.3 变量的高级用法

本节讨论一些变量的高级用法，这些高级的用法使我们可以更灵活的使用变量。

5.3.1 变量的替换引用

对于一个已经定义的变量，可以使用“替换引用”将其值使用指定的字符（字符串）进行替换。格式为“$(VAR: A= B)”（或者“${ VAR: A= B} ”），意思是，替换变量“VAR”中所有“A”字符结尾的字为“B”结尾的字。“结尾”的含义是空格之前（变量值的多个字以空格分开）。而对于变量其它部分的“A”字符不进行替换。例如：

```
foo := a.o b.o c.o
bar := $(foo:.o=.c)
```

在这个定义中，变量“bar”的值就为“a.c b.c c.c”。使用变量的替换引用将变量“foo”以空格分开的值中的所有的字的尾字符“o”替换为“c”，其他部分不变。而且如果在变量“foo”中如果存在“o.o”时，那么变量“bar”的值为“a.c b.c c.c o.c”而不是“a.c b.c c.c c.c”。这一点需要明确。

变量的替换引用其实是函数“patsubst”（参考 7.2 文本处理函数 一节）的一个简化实现。在GNU make中同时提供了这两种方式来实现同样的目的，以兼容其它版本make。

另外一种引用替换的技术使用功能更强大的“patsubst”函数的所有功能。它的格式和上面“$(VAR: A= B)”的格式相类似，不过需要在这里的“A”和“B”中需要包含模式字符“%”。只是它就和“$(patsubst A,B $(VAR))”（可参考 7.2 make的文本处理函数 一小节）所实现功能相同。例如：

```
foo := a.o b.o c.o
bar := $(foo:%.o=%.c)
```

这个例子同样使变量“bar”的值为“a.c b.c c.c”。这种格式的替换引用方式比第一种方式更为通用。

5.3.2 变量的套嵌引用

计算的变量名是一个比较复杂的概念，仅用在那些复杂的 Makefile 中。通常我们不需要对它的计算过程有深入地了解，只要知道当一个被引用的变量名之中含有“$”时，可以得到另外一个值。你如果是一个比较喜欢追根问底的人，或者你想弄清楚 make 计算变量的过程。就可以参考本节的内容。

一个变量名（文本串）之中可以包含对其它变量的引用。这种情况我们称之为“变量的套嵌引用”或者“计算的变量名”。例如：

先看一个例子：

```
x = y
y = z
```

```
a := $($(x))
```

这个例子中，最终定义了“a”的值为“z”。来看一下变量的引用过程：首先最里边的变量引用“$(x)”被替换为变量名“y”（就是“$($(x))”被替换为了“$(y)”），之后“$(y)”被替换为“z”（就是 a := z）。这个例子中（a:=$($(x))）所引用的变量名不是明确声明的，而是由$(x)扩展得到。这里“$(x)”相对于外层的引用就是套嵌的变量引用。

上个例子我们可以看到是一个两层的套嵌引用，具有多层的套嵌引用在 Makefile 中也是允许的。下边我们在来看一个三层套嵌引用的例子：

```
x = y
y = z
z = u
a := $($($(x)))
```

这个例子的最终是定义了“a”的值为“u”。它的扩展过程和上边第一个例子的过程相同。首先“$(x)”被替换为“y”，则“$($(x))”就是“$(y)”，“$(y)”在北替换为“z”，所有就是“a:=$(z)”；“$(z)”最后被替换为“u”。

以上两个套嵌引用的例子中没有使用到递归展开式变量。递归展开式变量的变量名的计算过程，也是按照相同的方式被扩展的。我们看一个这样一个例子：

```
x = $(y)
y = z
z = Hello
a := $($(x))
```

此例最终实现了“a:=Hello”这么一个定义。其中“$($(x))”被替换成了“$($(y))”，“$($(y)) ”在被替换为“$(z)”，最终就是“a:=Hello”。

这里的$($(x))被替换成了$($(y))，因为$(y)值是“z”，所以，最终结果是：a:=$(z)，也就是“Hello”。

递归变量的套嵌引用过程，也可以包含变量的修改引用和函数调用。看下边的例子，使用了 make 的文本处理函数：

```
x = variable1
variable2 := Hello
y = $(subst 1,2,$(x))
z = y
a := $($($(z)))
```

此例同样的实现“a:=Hello”。“$($($(z)))”替换为“$($(y))”，之后再次被替换为“$($(subst 1,2,$(x)))”（“$(x)”的值是“variable1”，所以有“$($(subst 1,2,$(variable1)))”）。函数处理之后为“$(variable2)”

之后对它在进行替换展开。最终，变量“a”的值就是“Hello”。从上边的例子中我们看到，计算

的变量名的引用过程存在多层套嵌过程，也是用了文本处理函数。这个复杂的计算变量的过程，会使很多人感到混乱甚至迷惑。上例中所要实现的目的没有直接使用“a:=Hello”来的直观明了。我们在书写Makefile时，应尽量避免使用套嵌的变量引用方式。在一些必需的地方，也最好不要使用高于两级的套嵌引用。使用套嵌的变量引用时，如果涉及到递归展开式的变量引用时需要特别注意（参考 5.2.1 递归展开式变量 一小节）。如果处理不当可能会导致递归展开错误，或者出现一些那一预料的现象。

一个计算的变量名可以不是对一个完整、单一的其它变量的引用。其中可以包含多个变量的引用，也可以包含一些文本字符串。就是说，计算变量的名字可以由一个或者多个变量引用同时加上字符串混合组成。例如：

```
a_dirs := dira dirb
1_dirs := dir1 dir2

a_files := filea fileb
1_files := file1 file2

ifeq "$(use_a)" "yes"
a1 := a
else
a1 := 1
endif

ifeq "$(use_dirs)" "yes"
df := dirs
else
df := files
endif

dirs := $($(a1)_$(df))
```

这个例子实现对变量“dirs”的定义，它的可能取值为“a_dirs”、“1_dirs”、“a_files”或者“a_files”四个值其中之一，具体依赖于“use_a”和“use_dirs”的定义。

计算的变量名也可以使用上一小节我们讨论过的“变量的替换引用”。例如：

```
a_objects := a.o b.o c.o
1_objects := 1.o 2.o 3.o

sources := $($(a1)_objects:.o=.c)
```

这个例子实现对变量“sources”的定义，它的可能取值为“a.c b.c c.c”和“1.c 2.c 3.c”，具体依赖于“a1”的定义。大家自己分析一下计算变量名的过程。

使用嵌套的变量引用的唯一限制是，不能通过指定部分需要调用的函数名称（调用的函数包括了函数名本身和执行的参数）来实现对这个函数的调用。这是因为套嵌引用在展开之前已经完成了对函数名的识别测试。语言的描述可能比较难理解。我们来看一个例子，此例子试图将函数执行的

结果赋值给一个变量：

```
ifdef do_sort
func := sort
else
func := strip
endif

bar := a d b g q c

foo := $($(func) $(bar))
```

此例的本意是将“sort”或者“strip”（依赖于是否定义了变量“do_sort”）以“a d b g q c”的执行结果赋值变量“foo”。在这里使用了套嵌引用方式来实现，这个实现的结果是：变量“foo”的值为字符串“sort a d b g q c”或者“strip a d g q c”。这是目前版本的 make 在处理套嵌变量引用时的限制。

计算的变量名可以用在：1．一个使用赋值操作符定义的变量的左值部分；2．使用“define”定义的变量名中。例如：

```
dir = foo
$(dir)_sources := $(wildcard $(dir)/*.c)
define $(dir)_print
lpr $($(dir)_sources)
endef
```

在这个例子中我们定义了三个变量：“dir”，“foo_sources”和“foo_print”。

计算的变量名在进行替换时的顺序是：从最里层的变量引用开始，逐步向外进行替换。一层层展开直到最后计算出需要应用的具体的变量，之后进行替换展开得到实际的引用值。

变量的套嵌引用（计算的变量名）在我们的 Makefile 中应该尽量避免使用。在必需的场合使用时掌握的原则是：套嵌使用的层数越少越好，使用多个两层套嵌引用代替一个多层的套嵌引用。如果在你的 Makefile 中存在一个层次很深的套嵌引用。会给其他人阅读造成很大的困难。而且变量的多级套嵌引用在某些时候会使简单问题复杂化。

作为一个优秀的程序员，在面对一个复杂问题时，应该是寻求一种尽可能简单、直接并且高效的处理方式来解决，而不是将一个简单问题在实现上复杂化。如果想在简单问题上突出自己使用某种语言的熟练程度，是一种非常愚蠢、且不成熟的行为。

注意：

套嵌引用的变量和递归展开的变量在本质上存在区别。套嵌的引用就是使用一个变量表示另外一个变量，或者更多的层次；而递归展开的变量表示当一个变量存在对其它变量的引用时，对这变量替换的方式。递归展开在另外一个角度描述了这个变量在定义是赋予它的一个属性或者风格。并且我们可以在定义个一个递归展开式的变量时使用套嵌引用的方式，但是建议你的实际编写Makefile时要尽量避免这种复杂的用法。

5.4 变量取值

一个变量可以通过以下几种方式来获得值：

- ✧ 在运行make时通过命令行选项来取代一个已定义的变量值。参考 5.7 override指示符 一节
- ✧ 在makefile文件中通过赋值的方式（参考 5.5 如何设置变量 一节）或者使用“define”来为一个变量赋值（参考 5.8 多行定义 一节）。
- ✧ 将变量设置为系统环境变量。所有系统环境变量都可以被make使用。参考 5.9 系统环境变量 一节
- ✧ 自动化变量，在不同的规则中自动化变量会被赋予不同的值。它们每一个都有单一的习惯性用法。参考 9.5.3 自动化变量 一小节
- ✧ 一些变量具有固定的值。参考 9.3 隐含变量 一节

5.5 如何设置变量

Makefile 中变量的设置（也可以称之为定义）是通过“= ”（递归方式）或者“:= ”（静态方式）来实现的。“= ”和“:= ”左边的是变量名，右边是变量的值。下边就是一个变量的定义语句：

```
objects = main.o foo.o bar.o utils.o
```

这个语句定义了一个变量“objects”，其值为一个.o 文件的列表。变量名两边的空格和“= ”之后的空格在 make 处理时被忽略。

使用“= ”定义的变量称之为“递归展开”式变量；使用“:= ”定义的变量称为“直接展开”式变量，“直接展开”式的变量如果其值中存在对其变量或者函数的引用，在定义时这些引用将会被进行替换展开（详细可参考 5.2 两种变量定义（赋值）一节）。

定义一个变量时需要明确以下几点：

1. 变量名之中可以包含函数或者其它变量的引用，make在读入此行时根据已定义情况进行替换展开而产生实际的变量名。参考 5.3.2 变量的套嵌引用 一小节
2. 变量的定义值在长度上没有限制。不过还是需要考虑你的实际情况，保证你的机器上有足够的可用的交换空间来处理一个超常的变量值。变量定义较长时，一个好的做法就是将比较长的行分多个行来书写，除最后一行外行与行之间使用反斜杠（\）连接，表示一个完整的行。这样的书写方式对make的处理不会造成任何影响，便于后期修改维护而且使得你的Makefile更清晰。例如上边的例子就可以这样写：

   ```
   ojects = main.o foo.o \
            bar.o utils.o
   ```

3. 当引用一个没有定义的变量时，make默认它的值为空。

4. 一些特殊的变量在make中有内嵌固定的值（可参考 9.3 隐含变量 一节），不过这些变量允许我们在Makefile中显式得重新给它赋值。
5. 还存在一些由两个符号组成的特殊变量，称之为自动环变量。它们的值不能在Makefile中进行显式的修改。这些变量使用在规则中时，不同的规则中它们会被赋予不同的值。
6. 如果你希望实现这样一个操作，仅对一个之前没有定义过的变量进行赋值。那么可以使用速记符“?= ”（条件方式）来代替“= ”或者“:= ”来实现（可参考 5.2.4 “?=”操作符 一小节）。

5.6 追加变量值

通常对于一个通用变量在定义之后的其他一个地方，需要给它的值进行追加。这也是非常有用的。我们可以在开始给它定义一个基本的值，后续可以不断地根据需要给它增加一些必要值。在Makefile 中使用“+ = ”（追加方式）来实现对一个变量值的追加操作。像下边那样：

```
objects += another.o
```

这个操作把字符串“another.o”添加到变量“objects”原有值的末尾，使用空格将其分开。因此我们可以看到：

```
objects = main.o foo.o bar.o utils.o
objects += another.o
```

上边的两个操作之后变量“objects”的值成：“main.o foo.o bar.o utils.o another.o”。使用“+ = ”操作符，就相当于：

```
objects = main.o foo.o bar.o utils.o
objects := $(objects) another.o
```

但是，这两种方式可能在简单一些的 Makefile 有相同的效果，复杂的 Makefile 中它们之间的差异就会导致一些问题。为了方便我们调试，了解这两种实现的差异还是很有必要的。

1. 如果被追加值的变量之前没有定义，那么，“+ = ”会自动变成“= ”，此变量就被定义为一个递归展开式的变量。如果之前存在这个变量定义，那么“+ = ”就继承之前定义时的变量风格（可参考 5.2 两种变量定义 一节）。
2. 直接展开式变量的追加过程：变量使用“:= ”定义，之后“+ = ”操作将会首先替换展开之前此变量的值，尔后在末尾添加需要追加的值，并使用“:= ”重新给此变量赋值。实际的过程像下边那样：

```
variable := value
variable += more
```

就是：

```
variable := value
variable := $(variable) more
```

3. 递归展开式变量的追加过程：一个变量使用“= ”定义，之后“+ = ”操作时不对之前此变量值中的任何引用进行替换展开，而是按照文本的扩展方式（之前等号右边的文本未发生变化）替换，尔后在末尾添加需要追加的值，并使用“= ”给此变量重新赋值。实际的过程和上边的相类似：

```
variable = value
variable += more
```

相当于：

```
temp = value
variable = $(temp) more
```

当然了，上边的过程并不会存在中间变量：“temp”，这里只是使用它来描述得更形象。这种情况时如果“value”中存在某种引用，情况就有些不同了。看我们通常一个会用到的例子：

```
CFLAGS = $(includes) -O
...
CFLAGS += -pg # enable profiling
```

第一行定义了变量“CFLAGS”，它是一个递归展开式的变量。因此 make 在处理它的定义时不会对其值中的引用“$(includes) ”进行展开，它的替换展开是在变量“CFLAGS”被引用的规则中。因此，变量“include”可以在“CFLAGS”之前没有定义，只要它在实际引用“CFLAGS”之前定义就可以了。如果给“CFLAGS”追加值使用“:= ”操作符，我们按照下边那样实现：

```
CFLAGS := $(CFLAGS) -pg # enable profiling
```

这样似乎好像很正确，但是实际上它在有些情况时却不是你所要实现的。我们来看看，因为“:= ”操作符定义的是直接展开式变量，因此变量值中对其它变量或者函数的引用会在定义时进行展开。在这种情况下，如果变量“includes”在之前没有进行定义的话，变量“CFLAGS”的值为“-O -pg”（$(includes)被替换展开为空字符）。而其后出现的“includes”的定义对“CFLAGS”将不产生影响。相反的情况，如果我们在这里使用“+ = ”实现：

```
CFLAGS += -pg # enable profiling
```

那么变量“CFLAGS”的值就是文本串“$(includes) –O -pg”，因为之前“CFLAGS”定义为递归展开式，所以追加值时不会对其值的引用进行替换展开。因此变量“includes”只要出现在规则

对“CFLAGS”的引用之前定义，它都可以对“CFLAGS”的值起作用。对于递归展开式变量的追加，make 程序会同样会按照递归展开式的定义来实现对变量的重新赋值，不会发生递归展开式变量展开过程的无限循环。

5.7 override 指示符

通常在执行 make 时，如果通过命令行定义了一个变量，那么它将替代在 Makefile 中出现的同名变量的定义。就是说，对于一个在 Makefile 中使用常规方式（使用“=”、“:=”或者“define”）定义的变量，我们可以在执行 make 时通过命令行方式重新指定这个变量的值，命令行指定的值将替代出现在 Makefile 中此变量的值。为了防止命令行变量定义的值替代 Makefile 中变量定义的值。需要在 Makefile 中使用指示符“override”来声明这个变量，像下边那样：

override VARIABLE = VALUE

或者：

override VARIABLE := VALUE

也可以对变量使用追加方式：

override VARIABLE += MORE TEXT

对于追加方式需要说明的是：**变量在定义时使用了“override”，则后续对它值进行追加时，也需要使用带有“override”指示符的追加方式。否则对此变量值的追加不会生效。**

指示符“override”并不是用来调整 Makefile 和执行时命令参数的冲突，其存在的目的是为了使用户可以改变或者追加那些使用 make 的命令行指定的变量的定义。从另外一个角度来说，就是实现了在 Makefile 中增加或者修改命令行参数的一种机制。我们可能会有这样的需求：可以通过命令行来指定一些附加的编译参数，对一些通用的参数或者必需的编译参数我们可以在 Makefile 中指定，而在命令行中可以指定一些特殊的参数。对待这种需求，我们可以使用指示符“override”来实现。

例如无论命令行指定那些编译参数，必须打开调试开关“-g”，我们的 Makefile 对“CFLAGS”应该这样写：

override CFLAGS += -g

无论通过命令行指定那些编译选项，“-g”参数始终存在。

对于使用“define”定义的变量我们同样也可以使用“override”进行声明。例如：

override define foo
bar

```
endef
```

最后我们来看一个例子：

```
# sample Makefile

EXEF = foo

override CFLAGS += -Wall –g

.PHONY : all debug test
all : $(EXEF)

foo : foo.c
...........
...........

$(EXEF) : debug.h
    $(CC) $(CFLAGS) $(addsuffix .c,$@) –o $@

debug :
    @echo "CFLAGS = $(CFLAGS)"
```

执行：make CFLAGS=-O2 将显式结果为编译“foo”的过程是“cc –O2 –Wall –g foo.c –o foo”。执行“make CFLAGS=-O2 debug”可以查看到变量“CFLAGS”的值为“–O2 –Wall –g”。另外，这个例子中，如果把变量“CFLAGS”之前的指示符“override”去掉以后使用相同的命令将得到不同的结果。大家试试看！

5.8 多行定义

定义变量的另外一种方式是使用“define”指示符。它定义一个包含多行字符串的变量，我们就是利用它的这个特点实现一个完整命令包的定义（可参考 4.7 定义命令包 一节）。使用“define”定义的命令包可以作为“eval”函数的参数来使用。参考 7.8 eval函数 一节

本文的前些章节已经不止一次的提到并使用了“define”。相信大家已经有所了解。本节就“define”定义变量从以下几个方面来讨论：

1. “define”定义变量的语法格式：以指示符“define”开始，“endif”结束，之间的所有内容就是所定义变量的值。所要定义的变量名字指示符“define”的同一行之后；指示符所在行的下一行开始一直到“end”所在行的上一行之间的若干行，是变量的值定义。

```
define two-lines
echo foo
echo $(bar)
endef
```

如果将变量“two-lines”作为命令包执行时，其相当于：

two-lines = echo foo; echo $(bar)

我想大家对这个命令的执行应该是比较熟悉的。它把变量“two-lines”的值作为一个完整的shell命令行来处理（是使用分号“;”分开的在同一行中的两个命令而不是作为两个命令行来处理），保证了变量完整。（关于完整命令行的执行可参考 4.2 命令的执行 一节）

2. 变量的风格：使用“define”定义的变量和使用“=”定义的变量一样，属于“递归展开”式的变量，两者只是在语法上不同。因此“define”所定义的变量值中，对其它变量或者函数引用不会在定义时替换展开，其展开是在“define”定义的变量被引用时进行的。
3. 可以套嵌引用。因为是递归展开式变量，所以在存在嵌套引用时“$(x)”将是变量的值的一部分。
4. 变量值中可以包含：换行符、空格等特殊符号（注意如果定义中某一行是以[Tab]字符开始时，当引用此变量时这一行会被作为命令行来处理）。
5. 可以使用“override”在定义时声明变量：这样可以防止变量的值被命令行指定的值替代。例如：

```
override define two-lines
foo
$(bar)
endef
```

5.9 系统环境变量

make 在运行时，系统的所有环境变量对它都是可见的。在 Makefile 中，可以引用任何已定义的系统环境变量。（这里我们区分系统环境变量和 make 的环境变量，系统环境变量是这个系统所有用户所拥有的，而 make 的环境变量只是对于 make 的一次执行过程有效，以下正文中出现没有限制的“环境变量”时默认指的是“系统环境变量”，在特殊的场合我们会区分两者）正因为如此，我们就可以设置一个命名为“CFLAGS”的环境变量，用它来指定一个默认的编译选项。我们就可以在所有的 Makefile 中直接使用这个变量来对 c 源代码就行编译。通常这种方式是比较安全的，但是它的前提是大家都明白这个变量所代表的含义，没有人在 Makefile 中把它作其他的用途。当然了，你也可以在你的 Makefile 中根据你的需要对它进行重新定义。

使用环境变量我们需要注意以下几点：

1. 在 Makefile 中对一个变量的定义或者以 make 命令行形式对一个变量的定义，都将覆盖同名的环境变量（注意：它并不改变系统环境变量定义，被修改的环境变量只在 make 执行过程有效）。而 make 使用“-e”参数时，Makefile 和命令行定义的变量不会覆盖同名的环境变量，make 将使用系统环境变量中这些变量的定义值。
2. make的递归调用中，所有的系统环境变量会被传递给下一级make。**默认情况下，只有环**

境变量和通过命令行方式定义的变量才会被传递给子make进程。在Makefile中定义的普通变量需要传递给子make时需要使用“export”指示符来对它声明。参考 4.6.2 变量和递归 一小节

3. 一个比较特殊的是环将变量“SHELL”。在系统中这个环境变量的用途是用来指定用户和系统的交互接口，显然对于make是不合适的。因此在make的执行环境变量“SHELL”的取值不使用同名的环境变量定义，而默认使用“/bin/sh”作为它的命令行解释程序（make执行之前设置它为“/bin/sh”）。（参考 4.2 命令的执行 一小节）

这里不推荐使用环境变量来完成普通变量的工作，特别是在 make 的递归调用中。任何一个环境变量的错误定义都对系统上的所有 make 产生影响，甚至是毁坏性的。因为环境变量具有全局的特征。所以尽量不要污染环境变量，造成环境变量名字污染。我想大多数系统管理员都明白环境变量对系统是多么的重要。

我们来看一个例子，结束本节。假如我们的机器名为“server-cc”；我们的 Makefile 内容如下：

```
# test makefile
HOSTNAME = server-http
............
............
.PHONY : debug
debug :
        @echo “hostname is : $( HOSTNAME)”
        @echo “shell is $(SHELL)”
```

1. 执行“make debug”将显示：

   hostname is : server-http
   shell is /bin/sh
2. 执行“make –e debug”；将显示：

   hostname is : server-cc
   shell is /bin/sh
3. 执行“make –e HOSTNAEM= server-ftp”；将显示：

   hostname is : server-ftp
   shell is /bin/sh

记住：除非必须，否则在你的 Makefile 中不要重置环境变量“SHELL”的值。因为一个不正确的命令行解释程序可能会导致规则定义的命令执行失败，甚至是无法执行！当需要重置它时，必须有充分的理由和配套的规则命令来适应这个新指定的命令行解释程序。

5.10 目标指定变量

在Makefile中定义一个变量，这个变量对此Makefile的所有规则都是有效的。它就像是一个“全局的”变量（仅限于定义它的那个Makefile中的所有规则，如果需要对其它的Makefile中的规则有效，就需要使用“export”对它进行声明）。当然“自动化变量”除外（参考 9.5.3 自动化变量 一节）。

另外一个特殊的变量就是所谓的“目标指定变量（Target-specific Variable）”。此特性允许对于相同变量根据目标指定不同的值，有点类似于自动化变量。目标指定的变量值只在指定它的目标的上下文中有效，对于其他的目标不产生影响。就是说目标指定的变量具有“局部性”。

设置一个目标指定变量的语法为：

TARGET ... : VARIABLE-ASSIGNMENT

或者：

TARGET ... : override VARIABLE-ASSIGNMENT

一个多目标指定的变量的作用域是所有这些目标的上下文，它包括了和这个目标相关的所有执行过程。

目标指定变量的一些特点：

1. “***VARIABLE-ASSIGNMENT***”可以使用任何一个有效的赋值方式，“=”（递归）、“:=”（静态）、“+=”（追加）或者“？=”（条件）。
2. 使用目标指定变量值时，目标指定的变量值不会影响同名的那个全局变量的值。就是说目标指定一个变量值时，如果在 Makefile 中之前已经存在此变量的定义（非目标指定的），那么对于其它目标全局变量的值没有变化。变量值的改变只对指定的这些目标可见。
3. 目标指定变量和普通变量具有相同的优先级。就是说，当我们使用 make 命令行的方式定义变量时，命令行中的定义将替代目标指定的同名变量定义（和普通的变量一样会被覆盖）。另外当使用 make 的“-e”选项时，同名的环境变量也将覆盖目标指定的变量定义。因此为了防止目标指定的变量定义被覆盖，可以使用第二种格式，使用指示符“override”对目标指定的变量进行声明。
4. 目标指定的变量和同名的全局变量属于两个不同的变量，它们在定义的风格（递归展开式和直接展开式）上可以不同。
5. 目标指定的变量变量会作用到由这个目标所引发的所有的规则中去。例如：

```
prog : CFLAGS = -g
prog : prog.o foo.o bar.o
```

这个例子中，无论 Makefile 中的全局变量“CFLAGS”的定义是什么。对于目标“prog”以

及其所引发的所有（包含目标为“prog.o”、“foo.o”和“bar.o”的所有规则）规则，变量“CFLAGS”值都是“-g”。

使用目标指定变量我们可以在 Makefile 实现，对于不同的目标文件使用不同的编译参数。看一个例子：

```
# sample Makefile

CUR_DIR = $(shell pwd)
INCS := $(CUR_DIR)/include
CFLAGS := -Wall –I$(INCS)

EXEF := foo bar

.PHONY : all clean
all : $(EXEF)

foo : foo.c
foo : CFLAGS+=-O2
bar : bar.c
bar : CFLAGS+=-g
……..
……..

$(EXEF) : debug.h
    $(CC) $(CFLAGS) $(addsuffix .c,$@) –o $@

clean :
    $(RM) *.o *.d $(EXES)
```

这个 Makefile 文件实现了在编译程序“foo”使用优化选项“-O2”但不使用调试选项“-g”，而在编译“bar”时采用了“-g”但没有“-O2”。这就是目标指定变量的灵活之处。目标指定变量的其它特性大家可以修改这个简单的 Makefile 来进行验证！

5.11 模式指定变量

GNU make中，除了支持上一节所讨论的模式指定变量之外（参考 5.10 目标指定变量 一节），还支持另外一种方式：模式指定变量（Pattern-specific Variable）。使用目标定变量定义时，此变量被定义在某个具体目标和由它所引发的规则的目标上。而模式指定变量定义是将一个变量值指定到所有符合特定模式的目标上去。对于同一个变量如果使用追加方式，通常一个目标的局部变量值的顺序是：（*为所有规则定义的全局值*）+（*引发它所在规则被执行的目标所指定值*）+（*它所符合的模式指定值*）+（*此目标所指定的值*）。这个大家也不需要深入了解。

设置一个模式指定变量的语法和设置目标变量的语法相似：

```
PATTERN ... : VARIABLE-ASSIGNMENT
```

或者：

PATTERN ... : override VARIABLE-ASSIGNMENT

和目标指定变量语法的唯一区别就是：这里的目标是一个或者多个“模式”目标（包含模式字符“%”）。例如我们可以为所有的.o 文件指定变量“CFLAGS”的值：

%.o : CFLAGS += -O

它指定了所有.o 文件的编译选项包含“-O”选项，不改变对其它类型文件的编译选项。

需要说明的是：在使用模式指定的变量定义时。目标文件一般除了模式字符（%）以外需要包含某种文件名的特征字符（例如：“a%”、“%.o”、“%.a”等）。当单独使用“%”作为目标时，指定的变量会对任何类型的目标文件都有效。

第六章：Makefile的条件执行

6 Makefile的条件判断

条件语句可以根据一个变量的值来控制对 Makefile 的执行，执行或者忽略 Makefile 的特定部分。条件语句可以是两个不同变量、或者变量和常量值得比较。需要注意的是：条件语句只能用于控制 make 实际执行的 makefile 文件部分，它不能控制规则的 shell 命令执行过程。Makefile 中使用条件控制可以做到处理的灵活性和高效性。

6.1 一个例子

首先我们先看一个使用条件语句的 Makefile 例子；对变量“CC”进行判断，值如果是“gcc”那么在进行程序连接时使用库“libgnu.so”或者“libgnu.a”，否则不链接任何库。我们 Makefile 中的条件判断部分如下：

```
......
libs_for_gcc = -lgnu
normal_libs =

......

foo: $(objects)

ifeq ($(CC),gcc)
      $(CC) -o foo $(objects) $(libs_for_gcc)
else
      $(CC) -o foo $(objects) $(normal_libs)
endif
      ......
```

在上例中，条件语句中使用到了三个关键字：“ifeq”、“else”和“endif”。其中：

1. “ifeq”表示条件语句的开始，并指定了一个比较条件（相等）。之后是用圆括号括包围的、使用逗号“,”分割的两个参数，和关键字“ifeq”用空格分开。参数中的变量引用在进行变量值比较时被展开。“ifeq”之后就是当条件满足 make 需要执行的，条件不满足时忽略。
2. “else”之后就是当条件不满足时的执行部分。不是所有的条件语句都需要此部分。
3. “endif”表示一个条件语句的结束，任何一个条件表达式都必须以“endif”结束。

通过上边的例子我们可以了解到。Makefile中，条件表达式是工作文本级别的，条件的解析是由make程序来完成的。make是在解析Makefile时根据条件表达式忽略条件表达式中的某一个文本行，解析完成后保留的只有表达式满足条件所需要执行的文本行（可参考 2.9 make如何读取Makefile 一节）。对于上例，make处理这个条件时：

当变量“CC”的值为“gcc”时，整个条件表达式等效于：

```
foo: $(objects)
        $(CC) -o foo $(objects) $(libs_for_gcc)
```

当变量“CC”值不等于“gcc”时等效于：

```
foo: $(objects)
        $(CC) -o foo $(objects) $(normal_libs)
```

上面的例子，一种更简洁实现方式：

```
libs_for_gcc = -lgnu
normal_libs =

ifeq ($(CC),gcc)
libs=$(libs_for_gcc)
else
libs=$(normal_libs)
endif

foo: $(objects)
     $(CC) -o foo $(objects) $(libs)
```

6.2 条件判断的基本语法

一个简单的不包含“else”分支的条件判断语句的语法格式为：

```
CONDITIONAL-DIRECTIVE
TEXT-IF-TRUE
endif
```

表达式中“TEXT-IF-TRUE”可以是若干任何文本行，当条件为真时它就将被 make 程序作为需要执行的一部分。当条件为假时，不作为需要执行的一部分。

另外包含“else”的复杂一点的语法格式为：

CONDITIONAL-DIRECTIVE
TEXT-IF-TRUE
else
TEXT-IF-FALSE
endif

表示了如果条件为真，则将“TEXT-IF-TRUE”作为执行 Makefile 的一部分，否则将“TEXT-IF-FALSE”作为执行的 Makefile 的一部分。和“TEXT-IF-TRUE”一样，“TEXT-IF-FALSE”可以是若干任何文本行。

条件判断语句中“CONDITIONAL-DIRECTIVE”对于上边的两种格式都是同样的。可以是以下四种用于测试不同条件的关键字。

6.2.1.1 关键字“ifeq”

此关键字用来判断参数是否相等，格式如下：

`ifeq (ARG1, ARG2)'
`ifeq 'ARG1' 'ARG2''
`ifeq "ARG1" "ARG2"'
`ifeq "ARG1" 'ARG2''
`ifeq 'ARG1' "ARG2"'

替换并展开“ARG1”和“ARG1”之后，对它们的值进行比较。如果相同则（条件为真）将“TEXT-IF-TRUE”作为 make 要执行的一部分，否则将“TEXT-IF-FALSE”作为 make 要执行的一部分（上边的第二种格式）。

通常我们会使用它来判断一个变量的值是否为空（不是任何字符）。参数值可能是通过引用变量或者函数得到的，因而在展开过程中可能造成参数值中包含空字符（空格等）。一般在处理这种情况时我们使用make的“strip”函数（参考 7.2 文本处理函数 一节）来对它进行处理，去掉变量值中的空字符。格式为：

ifeq ($(strip $(foo)),)
TEXT-IF-EMPTY
endif

这样，即就是在“$(foo)”中存在若干前导和结尾空格，make 在解析 Makefile 时“TEXT-IF-EMPTY”也会被作为执行的一部分。

6.2.1.2 关键字“ifneq”

次关键字是用来判断参数是否不相等，格式为：

`ifneq (ARG1, ARG2)'
`ifneq 'ARG1' 'ARG2''
`ifneq "ARG1" "ARG2"'
`ifneq "ARG1" 'ARG2''

`ifneq 'ARG1' "ARG2"'

关键字“ifneq”实现的条件判断语句和“ifeq”相反。首先替换并展开“ARG1”和“ARG1”，对它们的值进行比较。如果不相同（条件为真）则将“TEXT-IF-TRUE”作为 make 要执行的一部分，否则将“TEXT-IF-FALSE”作为 make 要执行的一部分。

6.2.1.3 关键字“ifdef”

关键字是“ifdef”用来判断一个变量是否定义。格式为：

`ifdef VARIABLE-NAME'

如果变量“VAEIABLE_NAME”的值非空，那么表达式为真，将“TEXT-IF-TRUE”作为 make 要执行的一部分。否则，表达式为假，如果存在“TEXT-IF-FALSE”，就将它作为 make 要执行一部分。当一个变量没有被定义时，它的值为空。“VARIABLE-NAME”可以是变量或者函数的引用。

对于“ifdef”需要说明的是：**ifdef 只是测试一个变量是否有值，其并不会对变量进行替换扩展来判断变量的值是否为空。对于变量“VARIABLE-NAME”，除了“VARIABLE-NAME=”这种情况以外，使用其它方式对它的定义都会使“ifdef”返回真。就是说，即使我们通过其它方式（比如，定义它的值引用了其它的变量）给它富了一个空值，“ifdef”也会返回真。我们来看一个例子：**

例1：

```
bar =
foo = $(bar)
ifdef foo
frobozz = yes
else
frobozz = no
endif
```

例 2：

```
foo =
ifdef foo
frobozz = yes
else
frobozz = no
endif
```

例 1 中的结果是：“frobozz = yes”；而例 2 的结果是：“frobozz = no”。其原因就是在例 1 中，变量“foo”的定义是“foo = $(bar)”。虽然变量“bar”的值为空，但是“ifdef”判断的结果是真。因此当我们需要判断一个变量的值是否为空的情况时，需要使用“ifeq”（或者“ifneq”）而不是“ifdef”。可参考前两个小节的内容。

6.2.1.4 关键字“ifndef”

关键字“ifndef”实现的功能和“ifdef”相反。格式为：

`ifdef VARIABLE-NAME'

这个就不详细讨论了，它的功能就是实现了和“ifdef”相反的条件判断。

在“***CONDITIONAL-DIRECTIVE***”这一行上，可以以若干个空格开始，make 处理时会被忽略这些空格。但不能以[Tab]字符做为开始（不然就被认为是命令）。条件判断语句中，在除关键字（包括“endif”）之前、和条件表达式的参数中之外，其他任何地方都可以使用多个空格或[Tab]字符，它不会影响条件判断语句的功能。同样行尾也可以使用注释（“#”开始直到一行的结束）。

“else”和“endif”也是条件判断语句的一部分。在书写时它们都是没有任何参数的，可以以多个空格开始（同样不能以[Tab]字符开始）多个空格或[Tab]字符结束。行尾同样可以有注释内容。

在 make 读取 makefile 文件时计算表达式的值，并根据表达式的值决定判断语句中那一部分被作为此 Makefile 所要执行的内容（选择符合条件的语句）。因此在条件表达式中不能使用自动化变量，自动化变量在规则命令执行时才有效。更不能将一个完整的条件判断语句分写在两个不同的 makefile 文件中，其中在某一个 makefile 文件使用指示符“include”包含另外一个。

6.3 标记测试的条件语句

我们可以使用条件判断语句、并使用使用变量“MAKEFLAGS”（参考 4.6.3 命令行选项和递归 一小节）和函数“findstring”（参考 7.2 文本处理函数 一节），实现对make命令行选项的测试。看一个例子：

```
archive.a: ...
ifneq (,$(findstring t,$(MAKEFLAGS)))
        +touch archive.a
        +ranlib -t archive.a
else
        ranlib archive.a
endif
```

这个条件语句判断 make 的命令行参数是否包含“-t”（用来更新目标文件的时间戳）。根据命令行参数情况完成对“archive.a”执行不同的操作。命令行前的“+”的意思是告诉 make，即使 make 使用了“-t”参数，“+”之后的命令都需要被执行。

第七章：make的内嵌函数

7 make的函数

GNU make 的函数为我们提供了处理文件名、变量、文本和命令的方法。是我们的 Makefile 更为灵活和健壮。我们可以在需要的地方地调用函数来处理指定的文本（参数），函数的在调用它

的地方被替换为它的处理结果。函数调用（引用）的展开和变量引用的展开方式相同。

7.1 函数的调用语法

GNU make 函数的调用格式类似于变量的引用格式，以“$”开头表示一个引用的过程。语法格式如下：

$(FUNCTION ARGUMENTS)

或者：

${FUNCTION ARGUMENTS}

对于函数调用的格式有以下几点说明：

1. 调用个语法格式中“FUNCTION”是需要调用的函数名，它应该是 make 内嵌的函数名。对于用户自己的函数需要通过 make 的“call”函数来间接调用。
2. “ARGUMENTS”是函数的参数，参数和函数名之间使用若干个空格或者[tab]字符分割（建议使用一个空格，这样不仅使在书写上比较直观，更重要的是当你不能确定哪些地方是允许用[Tab]时候，避免不必要的麻烦）；如果存在多个参数时，参数之间使用逗号“,”分割。
3. 以“$”开头，使用成对的圆括号或花括号把函数名和参数括起（在 Makefile 中，圆括号和花括号在任何地方必须成对出现）。参数中存在变量或者函数的引用时，对它们使用所使用的分界符（圆括号或者花括号）建议使用和引用函数使用的相同。最好不要使用两种不同的括号。建议在变量的引用和函数的引用中统一使用圆括号；这样在使用“vim”编辑器书写 Makefile 时，使用圆括它可以亮度显式 make 的内嵌函数名，避免函数名的拼写错误。一般在 Makefile 中应该这样来书写“$(sort $(x))”；而不是“$(sort ${x})”和其它几种。
4. 函数处理参数时，参数中如果存在对其它变量或者函数的引用，首先对这些引用进行展开得到参数的实际内容。而后才对它们进行处理。参数的展开顺序是按照参数的先后顺序来进行的。
5. 书写时，函数的参数中不能出现逗号“,”和空格。这是因为逗号被作为多个参数的分隔符，前导空格会被忽略。在实际书写 Makefile 时，当需要使用这两个作为函数的参数时，可以把它们赋值给变量，在函数的参数中使用相应的变量引用。我们来看一个这样的例子：

```
comma:= ,
empty:=
space:= $(empty) $(empty)
foo:= a b c
bar:= $(subst $(space),$(comma),$(foo))
```

这样我们就实现了“bar”的值是“a,b,c”。

7.2 文本处理函数

以下是 GNU make 内嵌的文本（字符串）处理函数。

7.2.1 $(subst FROM,TO,TEXT)

函数名称：字符串替换函数—subst。

函数功能：把字串“TEXT”中的“FROM”字符替换为“TO”。

返回值：替换后的新字符串。

示例：

$(subst ee,EE,feet on the street)

替换“feet on the street”中的“ee”为“EE”，结果是新的字符串“fEEt on the strEEt”。

7.2.2 $(patsubst PATTERN,REPLACEMENT,TEXT)

函数名称：模式替换函数—patsubst。

函数功能：搜索“TEXT”中以空格分开的单词，将否符合模式“TATTERN”替换为“REPLACEMENT”。参数“PATTERN”中可以使用模式通配符“%”来代表一个单词中的若干字符。如果参数“REPLACEMENT”中也包含一个“%”，那么“REPLACEMENT”中的“%”将是“TATTERN”中的那个“%”所代表的字符串。在“TATTERN”和“REPLACEMENT”中，只有第一个“%”被作为模式字符来处理，后续的作为字符本上来处理。在两个参数中当使用第一个“%”本是字符本身时，可使用反斜杠“\”对它进行转义处理（转义处理的机制和使用静态模式的转义一致，具体可参考 3.12.1 静态模式规则的语法 一小节）。

返回值：替换后的新字符串。

函数说明：参数“TEXT”单词之间的多个空格在处理时被合并为一个空格，但前导和结尾空格忽略。

示例：

$(patsubst %.c,%.o,x.c.c bar.c)

把字串“x.c.c bar.c”中以.c 结尾的单词替换成以.o 结尾的字符。函数的返回结果是“x.c.o bar.o”

本文的第五章在 变量的高级用法的第一小节 中我们曾经讨论了变量的替换引用，它是一个简化版的“patsubst”函数在变量引用过程的实现。变量替换引用中：

$(VAR:PATTERN=REPLACEMENT)

就等于：

$(patsubst PATTERN,REPLACEMENT,$(VAR))

而另外一种更为简单的替换字符后缀的实现：

$(VAR:SUFFIX=REPLACEMENT)

它等于：

$(patsubst %SUFFIX,%REPLACEMENT,$(VAR))

例如我们存在一个代表所有.o 文件的变量。定义为“objects = foo.o bar.o baz.o”。为了得到这些.o 文件所对应的.c 源文件。我们可以使用以下两种方式的任意一个：

$(objects:.o=.c)
$(patsubst %.o,%.c,$(objects))

7.2.3 $(strip STRINT)

函数名称：去空格函数—strip。

函数功能：去掉字串（若干单词，使用若干空字符分割）“STRINT”开头和结尾的空字符，并将其中多个连续空字符合并为一个空字符。

返回值：无前导和结尾空字符、使用单一空格分割的多单词字符串。

函数说明：空字符包括空格、[Tab]等不可显示字符。

示例：

STR = a b c
LOSTR = $(strip $(STR))

结果是“a b c”。

“strip”函数经常用在条件判断语句的表达式中，确保表达式比较的可靠和健壮！参考 6.2.1.2 关键字“ifeq” 一小节的例子。

7.2.4 $(findstring FIND,IN)

函数名称：查找字符串函数—findstring。

函数功能：搜索字串“IN”，查找“FIND”字串。

返回值：如果在“IN”之中存在“FIND”，则返回“FIND”，否则返回空。

函数说明：字串“IN”之中可以包含空格、[Tab]。搜索需要是严格的文本匹配。

示例：

```
$(findstring a,a b c)
$(findstring a,b c)
```

第一个函数结果是字“a”；第二个值为空字符。

7.2.5 $(filter PATTERN...,TEXT)

函数名称：过滤函数—filter。

函数功能：过滤掉字串“TEXT”中所有**不符合**模式“PATTERN”的单词，保留所有**符合**此模式的单词。可以使用多个模式。模式中一般需要包含模式字符“%”。存在多个模式时，模式表达式之间使用空格分割。

返回值：空格分割的“TEXT”字串中所有**符合**模式“PATTERN”的字串。

函数说明：“filter”函数可以用来去除一个变量中的某些字符串，我们下边的例子中就是用到了此函数。

示例：

```
sources := foo.c bar.c baz.s ugh.h
foo: $(sources)
     cc $(filter %.c %.s,$(sources)) -o foo
```

使用“$(filter %.c %.s,$(sources))”的返回值给 cc 来编译生成目标“foo”，函数返回值为“foo.c bar.c baz.s”。

7.2.6 $(filter-out PATTERN...,TEXT)

函数名称：反过滤函数—filter-out。

函数功能：和“filter”函数事项的功能相反。过滤掉字串“TEXT”中所有**符合**模式“PATTERN”的单词，保留所有**不符合**此模式的单词。可以有多个模式。存在多个模式时，模式表达式之间使用空格分割。。

返回值：空格分割的“TEXT”字串中所有**不符合**模式“PATTERN”的字串。

函数说明：“filter-out”函数也可以用来去除一个变量中的某些字符串，（实现和“filter”函数相反）。

示例：

```
objects=main1.o foo.o main2.o bar.o
mains=main1.o main2.o

$(filter-out $(mains),$(objects))
```

实现了去除变量“objects”中“mains”定义的字串（文件名）功能。它的返回值为“foo.o bar.o”。

7.2.7 $(sort LIST)

函数名称：排序函数—sort。

函数功能：给字串“LIST”中的单词以首字母为准进行排序（升序），并取掉重复的单词。

返回值：空格分割的没有重复单词的字串。

函数说明：两个功能，排序和去字串中的重复单词。可以单独使用其中一个功能。

示例：

$(sort foo bar lose foo)

返回值为：“bar foo lose”。

7.2.8 $(word N,TEXT)

函数名称：取单词函数—word。

函数功能：取字串“TEXT”中第“N”个单词（“N”的值从 1 开始）。

返回值：返回字串“TEXT”中第“N”个单词。

函数说明：如果“N”值大于字串“TEXT”中单词的数目，返回空字符串。如果“N”为 0，出错！

示例：

$(word 2, foo bar baz)

返回值为“bar”。

7.2.9 $(wordlist S,E,TEXT)

函数名称：取字串函数—wordlist。

函数功能：从字串“TEXT”中取出从“S”开始到“E”的单词串。“S”和“E”表示单词在字串中位置的数字。

返回值：字串“TEXT”中从第“S”到“E”（包括“E”）的单词字串。

函数说明：“S”和“E”都是从 1 开始的数字。

当“S”比“TEXT”中的字数大时，返回空。如果“E”大于“TEXT”字数，返回从“S”开始，到“TEXT”结束的单词串。如果“S”大于“E”，返回空。

示例：

$(wordlist 2, 3, foo bar baz)

返回值为：“bar baz”。

7.2.10 $(words TEXT)

函数名称：统计单词数目函数—words。

函数功能：字算字串“TEXT”中单词的数目。

返回值："TEXT"字串中的单词数。

示例：

$(words, foo bar)

返回值是"2"。所以字串"TEXT"的最后一个单词就是：$(word $(words TEXT),TEXT)。

7.2.11 $(firstword NAMES...)

函数名称：取首单词函数—firstword。

函数功能：取字串"NAMES..."中的第一个单词。

返回值：字串"NAMES..."的第一个单词。

函数说明："NAMES"被认为是使用空格分割的多个单词（名字）的序列。函数忽略"NAMES..."中除第一个单词以外的所有的单词。

示例：

$(firstword foo bar)

返回值为"foo"。函数"firstword"实现的功能等效于"$(word 1, NAMES..)"。

以上 11 个函数是make内嵌的的文本处理函数。在书写Makefile时可搭配使用，来实现复杂功能。最后我们使用这些函数来实现一个实际应用的需求。例子中我们使用函数"subst"和"patsbust"。Makefile中可以使用变量"VPATH"（参考 3.5.1 一般搜索（变量VPATH） 一小节）来指定搜索路径。我们的例子中需要指定所有头文件的搜索路径也在这些目录下。头文件的搜索路径需要使用GCC的"-I"参数通过命令行选项来实现，而通过"VPATH"罗列的目录是用冒号":"分割的。如下就是Makefile的片段：

```
.....
VPATH = src:../includes
override CFLAGS += $(patsubst %,-I%,$(subst :, ,$(VPATH)))
.......
```

那么第二条语句所实现的功能就是"CFLAGS + = -Isrc –I../includes"。

7.3 文件名处理函数

GNU make 除上一节所介绍的内嵌的文本处理函数之外，还存在一些针对于文件名的处理函数。这些函数主要用来对一系列空格分割的文件名进行转换，这些函数的参数被作为若干个文件名来对待。函数对作为参数的一组文件名按照一定方式进行处理并返回空格分割的多个文件名序列。

7.3.1 $(dir NAMES...)

函数名称：取目录函数—dir。

函数功能：从文件名序列"NAMES..."中取出各个文件名目录部分。文件名的目录部分就是包含在文件名中的最后一个斜线（"/"）（包括斜线）之前的部分。

返回值：空格分割的文件名序列“NAMES...”中每一个文件的目录部分。

函数说明：如果文件名中没有斜线，认为此文件为当前目录（“./”）下的文件。

示例：

$(dir src/foo.c hacks)

返回值为“src/ ./”。

7.3.2 $(notdir NAMES...)

函数名称：取文件名函数——notdir。

函数功能：从文件名序列“NAMES...”中取出非目录部分。目录部分是指最后一个斜线（“/”）（包括斜线）之前的部分。删除所有文件名中的目录部分，只保留非目录部分。

返回值：文件名序列“NAMES...”中每一个文件的非目录部分。

函数说明：如果“NAMES...”中存在不包含斜线的文件名，则不改变这个文件名。以反斜线结尾的文件名，是用空串代替，因此当“NAMES...”中存在多个这样的文件名时，返回结果中分割各个文件名的空格数目将不确定！这是**此函数的一个缺陷**。

示例：

$(notdir src/foo.c hacks)

返回值为：“foo.c hacks”。

7.3.3 $(suffix NAMES...)

函数名称：取后缀函数—suffix。

函数功能：从文件名序列“NAMES...”中取出各个文件名的后缀。后缀是文件名中最后一个以点“.”开始的（包含点号）部分，如果文件名中不包含一个点号，则为空。

返回值：以空格分割的文件名序列“NAMES...”中每一个文件的后缀序列。

函数说明：“NAMES...”是多个文件名时，返回值是多个以空格分割的单词序列。如果文件名没有后缀部分，则返回空。

示例：

$(suffix src/foo.c src-1.0/bar.c hacks)

返回值为“.c .c”。

7.3.4 $(basename NAMES...)

函数名称：取前缀函数—basename。

函数功能：从文件名序列“NAMES...”中取出各个文件名的前缀部分（点号之后的部分）。前

缀部分指的是文件名中最后一个点号之前的部分。

返回值：空格分割的文件名序列“NAMES...”中各个文件的前缀序列。如果文件没有前缀，则返回空字串。

函数说明：如果“NAMES...”中包含没有后缀的文件名，此文件名不改变。如果一个文件名中存在多个点号，则返回值为此文件名的最后一个点号之前的文件名部分。

示例：

$(basename src/foo.c src-1.0/bar.c /home/jack/.font.cache-1 hacks)

返回值为：“src/foo src-1.0/bar /home/jack/.font hacks”。

7.3.5 $(addsuffix SUFFIX,NAMES...)

函数名称：加后缀函数—addsuffix。

函数功能：为“NAMES...”中的每一个文件名添加后缀“SUFFIX”。参数“NAMES...”为空格分割的文件名序列，将“SUFFIX”追加到此序列的每一个文件名的末尾。

返回值：以单空格分割的添加了后缀“SUFFIX”的文件名序列。

函数说明：

示例：

$(addsuffix .c,foo bar)

返回值为“foo.c bar.c”。

7.3.6 $(addprefix PREFIX,NAMES...)

函数名称：加前缀函数—addprefix。

函数功能：为“NAMES...”中的每一个文件名添加前缀“PREFIX”。参数“NAMES...”是空格分割的文件名序列，将“SUFFIX”添加到此序列的每一个文件名之前。

返回值：以单空格分割的添加了前缀“PREFIX”的文件名序列。

函数说明：

示例：

$(addprefix src/,foo bar)

返回值为“src/foo src/bar”。

7.3.7 $(join LIST1,LIST2)

函数名称：单词连接函数——join。

函数功能：将字串“LIST1”和字串“LIST2”各单词进行对应连接。就是将“LIST2”中的第一个单词追加“LIST1”第一个单词字后合并为一个单词；将“LIST2”中的第二个单词追加到“LIST1”的第一个单词之后并合并为一个单词，……依次列推。

返回值：单空格分割的合并后的字（文件名）序列。

函数说明：如果“LIST1”和“LIST2”中的字数目不一致时，两者中多余部分将被作为返回序列的一部分。

示例 1：

$(join a b , .c .o)

返回值为：“a.c b.o”。

示例 2：

$(join a b c , .c .o)

返回值为：“a.c b.o c”。

7.3.8 $(wildcard PATTERN)

函数名称：获取匹配模式文件名函数—wildcard

函数功能：列出当前目录下所有符合模式“PATTERN”格式的文件名。

返回值：空格分割的、存在当前目录下的所有符合模式“PATTERN”的文件名。

函数说明：“PATTERN”使用shell可识别的通配符，包括“?”（单字符）、“* ”（多字符）等。可参考 3.4 文件名中使用通配符 一节。

示例：

$(wildcard *.c)

返回值为当前目录下所有.c 源文件列表。

7.4 foreach 函数

函数“foreach”不同于其它函数。它是一个循环函数。类似于 Linux 的 shell 中的循环（for 语句）。

- “foreach”函数的**语法**：

$(foreach VAR,LIST,TEXT)

- **函数功能：**这个函数的工作过程是这样的：如果必要（存在变量或者函数的引用），首先展开变量“VAR”和“LIST”；而表达式“TEXT”中的变量引用不被展开。执行时把“LIST”中使用空格分割的单词依次取出赋值给变量“VAR”，然后执行“TEXT”表达式。重复直到“LIST”的最后一个单词（为空时结束）。“TEXT”中的变量或者函数引用在执行时才被展开，因此如果在“TEXT”中存在对“VAR”的引用，那么“VAR”的值在每一次展开式将会到的不同的值。
- **返回值：**空格分割的多次表达式“TEXT”的计算的结果。

我们来看一个例子，实现了将变量“files”赋值为目录“dirs”下所有文件列表：

```
dirs := a b c d
files := $(foreach dir,$(dirs),$(wildcard $(dir)/*))
```

例子中，“TEXT”的表达式为“$(wildcard $(dir)/*)”。表达式第一次执行时将展开为“$(wildcard a/*)”；第二次执行时将展开为“$(wildcard b/*)”；第三次展开为“$(wildcard c/*)”；...；以此类推。所以此函数所实现的功能就和一下语句等价：

```
files := $(wildcard a/* b/* c/* d/*)
```

函数的“TEXT”表达式过于复杂时，我们可以通过定义一个中间的变量来表示表达式的一部分。对于上边的例子我们也可以对各个部分分解来实现，如下：

```
find_files = $(wildcard $(dir)/*)
dirs := a b c d
files := $(foreach dir,$(dirs),$(find_files))
```

在这里我们定义了一个变量（也可以称之为表达式），需要注意的是，在这里定义的是“递归展开”时的变量“find_files”。保证定义时变量值中的引用不被展开，而是在“find_files”才展开（如果这里使用直接展开式的定义将是无效的表达式）。可参考 5.2 两种变量定义 一节。

- **函数说明：**函数中参数“VAR”是一个局部的临时变量，它只在“foreach”函数的执行过程中有效，它在汉书中对它的定义不会影响其它部分定义的同名“VAR”变量的值。它在函数的执行过程中是一个“直接展开”式变量。

在使用函数“foreach”时，需要注意：**变量“VAR”的名字。我们建议使用一个单词、最好能够表达其含义的名字，不要使用一个奇怪的字符串作为变量名。虽然在可能在执行时没有错误，但是可能会让人很费解。**

没有人会喜欢这种方式，尽管可能它可以正常工作：

```
files := $(foreach Esta escrito en espanol!,b c ch,$(find_files))
```

7.5 if 函数

函数“if”提供了一个在函数上下文中实现条件判断的功能。就像make所支持的条件语句—ifeq（参考 6.2.1.1 关键字“ifeq” 一小节）一样。

- **函数语法：**

  $(if CONDITION,THEN-PART[,ELSE-PART])

- **函数功能：**第一个参数“CONDITION”，在函数执行时忽略其前导和结尾空字符并展开。“CONDITION”的展开结果非空，则条件为真，就将第二个参数“THEN_PATR”作为函数的计算表达式，函数的返回值就是第二表达式的计算结果；“CONDITION”的展开结果为空，将第三个参数“ELSE-PART”作为函数的表达式，返回结果为第三个表达式的计算结果。
- **返回值：**根据条件决定函数的返回值是第一个或者第二个参数表达式的计算结果。当不存在第三个参数“ELSE-PART”，并且“CONDITION”展开为空，函数返回空。
- **函数说明：**函数的条件表达式“CONDITION”决定了，函数的返回值只能是“THEN-PART”或者“ELSE-PART”两个之一的计算结果。
- **函数示例：**

  SUBDIR += $(if $(SRC_DIR) $(SRC_DIR),/home/src)

  函数的结果是：如果“SRC_DIR”变量值不为空，则将变量“SRC_DIR”指定的目录作为一个子目录；否则将目录“/home/src”作为一个子目录。

7.6 call函数

“call”函数是唯一一个可以创建定制参数化的函数的引用函数。我们可以将一个变量定义为一个复杂的表达式，用“call”函数根据不同的参数对它进行展开来获得不同的结果。

- **函数语法：**

  $(call VARIABLE,PARAM,PARAM,...)

- **函数功能：**在执行时，将它的参数“PARAM”依次赋值给临时变量“$(1)”、“$(2)”（这些临时变量定义在“VARIABLE”的值中，参考下边的例子）…… call 函数对参数的数目没有限制，也可以没有参数值，没有参数值的“call”没有任何实际存在的意义。执行时变量“VARIABLE”被展开为在函数上下文有效的临时变量，变量定义中的“$(1)”作为第一个参数，并将函数参数值中的第一个参数赋值给它；变量中的“$(2)”一样被赋值为函数的第二个参数值；依此类推（变量$(0)代表变量“VARIABLE”本身）。之后对变量

"VARIABLE" 表达式的计算值。

- **返回值：**参数值"PARAM"依次替换"$(1)"、"$(2)"…… 之后变量"VARIABLE"定义的表达式的计算值。
- **函数说明：**1. 函数中"VARIBLE"是一个变量名，而不是对变量的引用。因此，通常"call"函数中的"VARIABLE"中不包含"$"（当然，除了此变量名是一个计算的变量名）。2. 当变量"VARIBLE"是一个 make 内嵌的函数名时（如"if"、"foreach"、"strip"等），对"PARAM"参数的使用需要注意，因为不合适或者不正确的参数将会导致函数的返回值难以预料。3. 函数中多个"PARAM"之间使用逗号分割。4. 变量"VARIABLE"在定义时不能定义为直接展开式！只能定义为递归展开式。
- **函数示例：**

首先，来看一个简单的例子。

示例 1：

```
reverse =   $(2) $(1)
foo = $(call reverse,a,b)
```

变量"foo"的值为"ba"。这里变量"reverse"中的参数定义顺序可以根据需要来调整，并不是需要按照"$(1)"、"$(2)"、"$(3)"…… 这样的顺序来定义。

看一个稍微复杂一些的例子。我们定义了一个宏"pathsearch"来在"PATH"路径中搜索第一个指定的程序。

示例 2：

```
 pathsearch = $(firstword $(wildcard $(addsuffix /$(1),$(subst :, ,$(PATH)))))
LS := $(call pathsearch,ls)
```

变量"LS"的结果为"/bin/sh"。执行过程：函数"subst"将环境变量"PATH"转换为空格分割的搜索路径列表；"addsuffix"构造出可能的可执行程序"$(1)"（这里是"ls"）带路径的完整文件名（如："/bin/$(1)"），之后使用函数"wildcard"匹配，最后"firstword"函数取第一个文件名。

函数"call"以可以套嵌使用。每一层"call"函数的调用都为它自己的局部变量"$(1)"等赋值，覆盖上一层函数为它所赋的值。

示例 3：

```
map = $(foreach a,$(2),$(call $(1),$(a)))
o = $(call map,origin,o map MAKE)
```

那么变量“o”的值就为“file file default”。我们这里使用了“origin”函数。我们分析函数的执行过程：首先，“o= $(call map,origin, o map MAKE)”这个函数调用使用了变量“map”所定义的表达式；使用内嵌函数名“origin”作为它的第一个参数值，使用Makefile中的变量“o map MAKE”作为他的第二个参数值。当使用“call”函数每一的展开后等价于“$(foreach a,o map MAKE,$(origin $(a)))”。

- **注意：**和其它函数一样，“call”函数会保留出现在其参数值列表中的空字符。因此在使用参数值时对空格处理要格外小心。如果参数中存在多余的空格，函数可能会返回一个莫名奇妙的值。为了安全，在变量作为“call”函数参数值之前，应去掉其值中的多余空格（可以使用“strip”函数）。

7.7 value函数

函数“value”提供了一种在不对变量进行展开的情况下获取其值的方式。注意：并不是说函数会取消之前已经执行过的替换扩展。比如：我们定义了一个直接展开式的变量，此变量在定义过程中对其它变量的引用进行替换展开得到自身的值。我们在使用“value”函数取它的值时，得到的是不包含任何引用的实际值。而不是将定义过程中的替换展开动作取消后包含引用的定义值。就是说它不能取消此变量中已经发生了的替换展开动作。

- **函数语法：**

  $(value VARIABLE)

- **函数功能：**不对变量“VARIBLE”进行任何展开操作，直接返回变量“VARIBALE”代表的值。这里“VARIABLE”是一个变量名，一般不包含“$”（当然，除了计算的变量名），
- **返回值：**变量“VARIBALE”所定义文本值（不展开其中的变量或者函数应用）。
- **函数说明：**

示例：

```
# sample Makefile
FOO = $PATH

all:
    @echo $(FOO)
    @echo $(value FOO)
```

执行make时，我们可以看到的结果是：第一行为：“ATH”。这是因为变量“FOO”定义为“$PATH”，所以展开为“ATH”（“$P”为空，参考 5.1 变量的引用 一节）。

第二行才是我们需要显示的系统环境变量“PATH”的值（value函数得到变量“FOO”的值为“$PATH”）。

7.8 eval函数

- **函数功能：**函数“eval”是一个比较特殊的函数。使用它我们可以在我们的Makefile中构造一个可变的规则结构关系（依赖关系链），其中可以使用其它变量和函数。函数“eval”对它的参数进行展开，展开的结果作为Makefile的一部分，make可以对展开内容进行语法解析。展开的结果可以包含一个新变量、目标、隐含规则或者是明确规则等。也就是说此函数的功能主要是：根据其参数的关系、结构，对它们进行替换展开。
- **返回值：**函数“eval”的返回值时空，也可以说没有返回值。
- **函数说明：**“eval”函数执行时会对它的参数进行两次展开。第一次展开过程发是由函数本身完成的，第二次是函数展开后的结果被作为Makefile内容时由make解析时展开的。明确这一点对于使用“eval”函数非常重要。在理解了函数“eval”二次展开的过程后。实际使用时，当函数的展开结果中存在引用（格式为：$(x)）时，那么在函数的参数中应该使用“$$”来代替“$”（参考 5.1 变量的引用 一节）。因为这一点，所以通常它的参数中会使用函数“value”来取一个变量的文本值。

我们看一个例子。例子看起来似乎非常复杂，因为它综合了其它的一些概念和函数。不过我们可以考虑两点：其一，通常实际一个模板的定义可能比例子中的更为复杂；其二，我们可以实现一个复杂通用的模板，在我们的所有Makefile中包含它，以可作到一劳永逸。相信这一点可能是大多数程序员所推崇的。

示例：

```
# sample Makefile

PROGRAMS      = server client

server_OBJS = server.o server_priv.o server_access.o
server_LIBS = priv protocol

client_OBJS = client.o client_api.o client_mem.o
client_LIBS = protocol

# Everything after this is generic

.PHONY: all
all: $(PROGRAMS)

define PROGRAM_template
$(1): $$($(1)_OBJ) $$($(1)_LIBS:%=-l%)
ALL_OBJS     += $$($(1)_OBJS)
endef

$(foreach prog,$(PROGRAMS),$(eval $(call PROGRAM_template,$(prog))))

$(PROGRAMS):
```

```
        $(LINK.o) $^ $(LDLIBS) -o $@

clean:
        rm -f $(ALL_OBJS) $(PROGRAMS)
```

我们来看一下这个例子：它实现的功能是完成“PROGRAMS”的编译链接。例子中“$(LINK.o)”为“$(CC) $(LDFLAGS)”，意思是对所有的.o文件和指定的库文件进行链接。可参考 9.2 make隐含规则一览 一节

“***$(foreach prog,$(PROGRAM),$(eval $(call PROGRAM_template,$(prog))))***”展开为：

```
server : $(server_OBJS) –l$(server_LIBS)
client : $(client_OBJS) –l$(client_LIBS)
```

7.9 origin函数

函数“origin”和其他函数不同，函数“origin”的动作不是操作变量（它的参数）。它只是获取和此变量（参数）相关的信息，告诉我们这个变量的出处（定义方式）。

- **函数语法：**

  $(origin VARIABLE)

- **函数功能：**函数“origin”查询参数“VARIABLE”（通常是一个变量名）的出处。
- **函数说明：**“VARIABLE”是一个变量名而不是一个变量的引用。因此通常它不包含“$”（当然，计算的变量名例外）。
- **返回值：**返回“VARIABLE”的定义方式。用字符串表示。

  函数的返回情况有以下几种：

1. undefined

   变量“VARIABLE”没有被定义。

2. default

   变量“VARIABLE”是一个默认定义（内嵌变量）。如“CC”、“MAKE”、“RM”等变量（参考 9.3 隐含变量 一节）。如果在Makefile中重新定义这些变量，函数返回值将相应发生变化。

3. environment

   变量“VARIABLE”是一个系统环境变量，并且make没有使用命令行选项“-e”（Makefile中不存在同名的变量定义，此变量没有被替代）。参考 8.7 make的命令行选项 一节

4. environment override

   变量“VARIABLE”是一个系统环境变量，并且make使用了命令行选项“-e”。Makefile

中存在一个同名的变量定义，使用“make -e”时环境变量值替代了文件中的变量定义。参考 8.7 make的命令行选项 一节

5. file

变量“VARIABLE”在某一个 makefile 文件中定义。

6. command line

变量“VARIABLE”在命令行中定义。

7. override

变量“VARIABLE”在 makefile 文件中定义并使用“override”指示符声明。

8. automatic

变量“VARIABLE”是自动化变量。参考 9.5.3 自动化变量 一节

函数“origin”返回的关于变量的信息对我们书写 Makefile 是相当有用的，可以使我们在使用一个变量之前对它的值的合法性进行判断。假设在 Makefile 其包了另外一个名为 bar.mk 的 makefile 文件。我们需要在 bar.mk 中定义变量“bletch”（无论它是否是一个环境变量），保证“make –f bar.mk”能够正确执行。另外一种情况，当 Makefile 包含 bar.mk，在 Makefile 包含 bar.mk 之前有同样的变量定义，但是我们不希望覆盖 bar.mk 中的“bletch”的定义。一种方式是：我们在 bar.mk 中使用指示符“override”声明这个变量。但是它所存在的问题时，此变量不能被任何方式定义的同名变量覆盖，包括命令行定义。另外一种比较灵活的实现就是在 bar.mk 中使用“origin”函数，如下：

```
ifdef bletch
ifeq "$(origin bletch)" "environment"
bletch = barf, gag, etc.
endif
endif
```

这里，如果存在环境变量“bletch”，则对它进行重定义。

```
ifneq "$(findstring environment,$(origin bletch))" ""
bletch = barf, gag, etc.
endif
```

这个例子实现了：即使环境变量中已经存在变量“bletch”，无论是否使用“make -e”来执行 Makefile，变量“bletch”的值都是“barf,gag,etc”（在 Makefile 中所定义的）。环境变量不能替代文件中的定义。

如果“$(origin bletch)”返回“environment”或“environment override”，都将对变量“bletch”重新定义。关于函数“firststring”可参考 7.2 文本处理函数 一节

7.10 shell函数

shell函数不同于除“wildcard”函数之外的其它函数。make可以使用它来和外部通信。

- **函数功能：**函数“shell”所实现的功能和shell中的引用（`` ` ``）相同。实现了命令的扩展。意味着需要一个shell 命令作为它的参数，而返回的结果是此命令在shell中的执行结果。make仅仅对它的回返结果进行处理；make将函数的返回结果中的所有换行符（“\n”）或者一对“\n\r”替换为单空格；并去掉末尾的回车符号（“\n”）或者“\n\r”。函数展开式时，它所调用的命令（它的参数）得到执行。（可参考 2.9 make如何解析makefile 一节）。除了对它的引用出现在规则的命令行中和递归的变量定义引用以外，其它决大多数情况下，make在读取Makefile时函数shell就被扩展。
- **返回值：**函数“shell”的参数在 shell 中的执行结果。
- **函数说明：**函数本身的返回值是其参数的执行结果，没有进行任何处理。对结果的处理是由 make 进行的。当对函数的引用出现在规则的命令行中，命令行在执行时函数引用才被展开。展开过程函数参数的执行时在另外一个 shell 进程中完成的，因此对于出现在规则命令行的多级“shell”函数引用需要谨慎处理，否则会影响效率（每一级的“shell”函数的参数都会有各自的 shell 进程）。

示例 1：

contents := $(shell cat foo)

将变量“contents”赋值为文件“foo”的内容，文件中的行在变量中使用空格（而不是换行符）分割。

示例 2：

files := $(shell echo *.c)

将变量“files”赋值为当前目录下所有.c文件的列表（文件名之间使用空格分割）。在shell中之行的命令是“echo *.c”，它会返回当前目录下的所有.c文件列表。上例的执行结果和函数“$(wildcard *.c)”的结果相同，除非你使用的是一个奇怪的shell。

注意：通过上边的两个例子我们可以看到，在引用“shell”函数的变量定义使用直接展开式定义。这样就保证了函数的展开在make读入Makefile的过程中完成的。后续对此变量的引用就不会有展开过程。这样可以防止规则命令行中的变量引用在命令行执行时展开的情况发生（因为展开“shell”函数需要另外的shell进程完成，影响命令的执行效率）。这也是我们建议的方式。

7.11 make的控制函数

make 提供了两个控制 make 运行方式的函数。通常它们用在 Makefile 中，当 make 执行过

程中检测到某些错误是为用户提供消息，并且可以控制 make 过程是否继续。

7.11.1 $(error TEXT...)

- **函数功能：**产生致命错误，并提示“TEXT...”信息给用户，之后退出 make 的执行。需要说明的是：“error”函数是在函数展开式（函数被调用时）才提示信息并结束 make 进程。因此如果函数出现在命令中或者一个递归的变量定义中时，在读取 Makefile 时不会出现错误。而只有包含“error”函数引用的命令被执行，或者定义中引用此函数的递归变量被展开时，才会提示致命信息“TEXT...”同时 make 退出执行。
- **返回值：**空字符
- **函数说明：**“error”函数一般不出现在直接展开式的变量定义中，否则在make读取Makefile时将会提示致命错误。关于递归展开和直接展开可参考 5.2 两种变量定义 一节

假设我们的 Makefile 中包含以下两个片断；

示例 1：

```
ifdef ERROR1
$(error error is $(ERROR1))
endif
```

make读取Makefile时，此定义之前已经定义变量“EROOR1”，那么make将会提示致命错误信息“$(ERROR1)”并退出。关于“ifdef”可参考 6.2.1.3 关键字“ifdef” 一小节。

示例 2：

```
ERR = $(error found an error!)

.PHONY: err
err: ; $(ERR)
```

这个例子，在 make 读取 Makefile 时不会出现致命错误。只有目标“err”被作为一个目标被执行时才会出现。

7.11.2 $(warning TEXT...)

- **函数功能：**函数“warning”类似于函数“error”，区别在于它不会导致致命错误（make 不退出），而只是提示“TEXT...”，make 的执行过程继续。
- **返回值：**空字符
- **函数说明：**用法和“error”类似，展开过程相同。

第八章：执行make

8 执行make

一般描述整个工程编译规则的 Makefile 可以通过不止一种方式来执行。最简单直接的方法就是使用不带任何参数的“make”命令来重新编译所有过时的文件。通常我们的 Makefile 就书写为这种方式。

在某些情况下：

1. 可能需要使用 make 更新一部分过时文件而不是全部
2. 需要使用另外的编译器或者重新定义编译选项
3. 只需要察看那些文件被修改，而不需要重新编译

为了达到这些特殊的目的，需要使用 make 的命令行参数来实现。Make 的命令行参数能实现的功能不仅限于这些，通过 make 的命令行参数可以实现更多特殊功能。

另外，make 的退出状态有三种：

0 — 状态为 0 时，表示执行成功。

2 — 执行过程出现错误，同时会提示错误信息。

1 — 在执行make时使用了“-q”参数，而且在当前存在过时的目标文件。参考 8.3 替换命令的执行 一节

本章的内容主要讲述如何使用 make 的命令参数选项来实现一些特殊的目的。在本章最后会对 make 的命令行参数选项进行比较详细的讨论。

8.1 指定makefile文件

本文以前的部分对如何指定makefile文件已经有过介绍（参考 2.2 Makefile文件的命名 一节）。当需要将一个普通命名的文件作为makefile文件时，需要使用make的“-f”、“--file”或者“--makefile”选项。例如：“make –f altmake”，它的意思是告诉make将文件“altmake”作为makefile文件来解析执行。

当在 make 的命令行选项中出现多个“-f”参数时，所有通过“-f”参数指定的文件都被作为 make 解析执行的 makefile 文件。

默认情况，在没有使用“-f”（“--file”或者“--makefile”）指定文件时。make 会在工作目录（当前目录）依次搜索命名为“GNUmakefile”、“makefile”和“Makefile”的文件，最终解析执行的是这三个文件中首先搜索到的哪一个。

8.2 指定终极目标

关于“终极目标”的基本概念我们在前面已经提到过（参考 1.2.4 make如何工作 一节）。所谓终极目标就是make最终所要重建的、Makefile中某个规则的目标（也可以称之为“最终规则”）。为了完成对终极目标的重建，可能会触发它的依赖或者依赖的依赖文件被重建的过程。

默认情况下，终极目标就是出现在Makefile中，除以点号“.”（参考 3.9 Makefile的特殊目标 一节）开始的第一个规则中的第一个目标（如果第一个规则存在多个目标）。因此我们的Makefile就书写为：第一个目标的编译规则就描述了整个工程或者程序的编译过程和规则。如果在Makefile中的第一个规则有多个目标，那么默认的终极目标是多个目标中的第一个。我们在Makefile所在的目录下执行“make”时，将完成对默认终极目标的重建。

另外，也可以通过命令行将一个 Makefile 中的目标指定为此次 make 过程的终极目标，而不是默认的终极目标。使用 Makefile 中目标名作为参数来执行“make”（格式为 “make TARGET_NAME”，如：“make clean”），可以把这个目标指定为终极目标。使用这种方式，我们也可以同时指定多个多个终极目标。

任何出现在 Makefile 规则中的目标都可以被指定为终极目标（不包含以“-”开始的和包含“=”的赋值语句，一般它们也不会作为一个目标出现），而且也可以指定一个 Makefile 中不存在的目标作为终极目标，前提是存在一个对应的隐含规则能够实现对这个目标的 make。例如：目录“src”下存在一个.c 的源文件“foo.c”，在 Makefile 中不存在目标“foo”或者次目录下就没有 makefile 文件，为了编译“foo.c”生成可执行的“foo”。只需要将“foo”作为 make 的参数执行：“make foo”就可以实现编译“foo”的目的。

make 执行时设置一个特殊变量“MAKECMDGOALS”，此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。注意：此变量仅限于用在特殊的场合（比如判断），在 Makeifle 中最好不要对它进行重新定义！例如：

```
sources = foo.c bar.c

ifneq ($(MAKECMDGOALS),clean)
include $(sources:.c=.d)
endif
```

例子中我们使用了变量“MAKECMDGOALS”来判断命令行参数是否指定了终极目标为“clean”，如果不是才包含（关于指示符include， 可参考 2.3 包含其它makefile文件 一节）所有源文件对应的依赖关系描述文件（关于.d文件可参考 3.14 自动产生依赖 一节）。上例避免在“make clean”时make试图重建所有.d文件的过程。

这种方式主要用在以下几个方面：

1. 对程序的一部分进行编译，或者仅仅对某几个程序进行编译而不是完整编译这个工程（也

可以在命令行参数中明确给出原本的默认的终极目标，例如：make all，没有人会说它是错误的。但是大家都会认为多此一举）。例如以下一个 Makefile 的片段，其中各个文件都有自己的描述规则：

```
.PHONY: all
all: size nm ld ar as
```

仅需要重建“size”文件时，执行“make size”就可以了。其它的程序就不会被重建。

2. 指定编译或者创建那些正常编译过程不能生成的文件（例如重建一个调试输出文件、或者编译一个调试版本的程序等），这些文件在 Makefile 中存在重建规则，但是它们没有出现在默认终极目标目标的依赖中。
3. 指定执行一个由伪目标定义的若干条命令或者一个空目标文件。如绝大多数Makefile中都会包含一个“clean”伪目标，这个伪目标定义了删除make过程生成的所有文件的命令，需要删除这些文件时执行“make clean”就可以了。本节以下列出了一些典型的伪目标和空目标的名字（详细的GNU 标准的目标名可参考 13.5 Makefile的标准的目标名 一节）。

部分标准的伪目标和空目标命名：

- all

  作为 Makefile 的顶层目标，一般此目标作为默认的终极目标。

- clean

  这个伪目标定义了一组命令，这些命令的功能是删除所有由 make 创建的文件。

- mostlyclean

  和“clean”伪目标功能相似。区别在于它所定义的删除命令不会全部删除由 make 生成的文件。比如说不需要删除某些库文件。

- distclean
- realclean
- clobber

  同样类似于伪目标，只是它们所定义的删除命令所删除的文件更多。可以包含非 make 创建的文件。例如：编译之前系统的配置文件、链接文件等。

- install

  将 make 成功创建的可执行文件拷贝到 shell 环境变量“PATH”指定的某个目录。典型的，应用可执行文件被拷贝到目录“/usr/local/bin”，库文件拷贝到目录“/usr/local/lib”目录下。

- print

  打印出所有被更改的源文件列表。

- tar

  创建一个 tar 文件。

- ✧ shar

  创建一个源代码的 shell 文档（shar 文件）。

- ✧ dist

  为源文件创建发布的压缩包，可以使各种压缩方式的发布包。

- ✧ TAGS

  更新一个工程的“tags”列表。

- ✧ check
- ✧ test

  对 Makefile 最后生成的文件进行检查。

这些功能和目标的对照关系并不是 GNU make 规定的。你可以在 Makefile 中定义任何命名的伪目标。但是以上这些都被作为一个默认的规则，所有开源的工程中这些特殊的目标的命名都是按照这种约定来的。既然绝大多数程序员都遵循这种约定，自然我们也应该按照这种约定来做。否则在别人看来这样 Makefile 只能算一个样例，不能作为正式版本。

8.3 替代命令的执行

书写 Makefile 的目的就是为了告诉 make 一个目标是否过期，以及如果重建一个过期的目标。但是在某些时候，我们并不希望真正更新那些已经过期的目标文件（比如：只是检查更新目标的命令是否正确，或者察看那些目标需要更新）。要实现这样的目的，可以使用一些特定的参数来指定 make 所要执行的动作。通过指定的参数，就替代了 make 默认动作的执行。因此我们把这种方式称为：替代命令的执行。这些参数包括：

-n

--just-print

--dry-run

--recon

指定 make 执行空操作（不执行规则的命令），只打印出需要重建目标使用的命令（只打印过期的目标的重建命令），而不对目标进行重建。

-t

--touch

类似于 shell 下的“touch”命令的功能。更新所有目标文件的时间戳（对于过时的目标文件不进行内容更新，只更新时间戳）。

-q

--question

不执行任何命令并且不打印任何输出信息，只检查所指定的目标是否已经是最新更。如果是则返回 0，否则返回 1。使用“-q”（“--question”）的目的只是让 make 返回指定（没有指定

则时终极目标）的目标当前是否是最新的。我们可以根据它的返回值来判断是否须要真正的执行更新目标的动作（返回值时 1 的情况）。

-W FILE

--what-if= FILE

--assume-new= FILE

--new-file= FILE

这个参数需要指定一个文件名。通常是一个存在源文件。make 将当前系统时间作为这个文件的时间戳（假设这个文件被修改过，但不真正的更改文件本身的时间戳）。因此这个文件的时间戳被认为最新的，在执行时依赖于这个文件的目标将会被重建。通过这种方式并结合“-n”参数，我们可以查看那些目标依赖于指定的文件。

通常“-W”参数和“-n”参数一同使用，可以在修改一个文件后来检查修改会造成那些目标需要被更新，但并不执行更新的命令，只是打印命令。

“-W”和“-t”参数配合使用时，make 将忽略其它规则的命令。只对依赖于“-W”指定文件的目标执行“touch”命令，在没有使用“-s”时，可以看到那些文件执行了“touch”。需要说明的是，make 在对文件执行“touch”时不是调用 shell 的命令，而是由 make 直接操作。

“-W”和“-q”参数配合使用时。由于将当前时间作为指定文件的时间戳（目标文件相对于系统当前时间是过时的），所以 make 的返回状态在没有错误发生时为 1，存在错误时为 2。

注意：以上三个参数同时使用时可能会出现错误。

总结：

参数“-n”、“-t”和“-q”不影响之前带“+”号和包含“$(MAKE)”的命令行的执行。就是说如果在规则的命令行中命令之前使用了“+”或者此命令行是递归地make调用时，无论是否使用了这三个参数之一，这些命令都得到执行。

“-W”参数有两个特点：

1. 可以和“-n”或者“-q”参数配合使用来查看修改所带来的影响（导致那些目标会被重建）。
2. 在不指定“-n”和“-q”参数、只使用“-W”指定一个文件时，可以模拟这个文件被修改的状态。make 就会重建依赖于此文件的所有目标。

另外“-p”和“-v”参数可以允许我们输出Makefille被执行的过程信息，相信这一点在很多场合，特别是调试我们的Makefile是非常有用（本章的最后一节有详细讨论， 参考 8.7 make的命令行选项 一节）。

8.4 防止特定文件重建

有些时候当修改了工程中的某一个文件后，并不希望重建那些依赖于这个文件的目标。比如说

我们在给一个头文件中加入了一个宏定义、或者一个增加的函数声明，这些修改不会对已经编译完成的程序产生任何影响。但在执行 make 时，因为头文件的改变会导致所有包含它的源文件被重新编译，当然了终极目标肯定也会被重建（除非你的模块时独立的，或者说你的 Makefile 的规则链的定义本身就存在缺陷）。这种情况下，为了避免重新编译整个工程，我们可以按照下边的过程来处理：

第一种：

1. 使用“make”命令对所有需要更新的目标进行重建。保证修改某个文件之前所有的目标已经是最新的。
2. 编辑需要修改的那个源文件（**修改的头文件的内容不能对之前的编译的程序有影响，比如：更改了头文件中的宏定义。这样会造成已经存在的程序和实现不相符！这里所说的修改指：不改变已经存在的任何东西，除非你有特殊的要求**）。
3. 使用“make -t”命令来改变已存在的所有的目标文件的时间戳，将其最后修改时间修改到当前时间。

第一种方式的是现实基于这样的条件：有需要对未更改头文件之前的编译程序进行保存（有点像备份）。通常这种需求还是比较少见的。更多的情况是：修改这个头文件之前已经修改了很多源文件或者其它的头文件，并且也没有执行 make（缺少了上边的第一步）。这种方式，有点像有计划的修改。没有普遍性，修改通常是不可预知的。因此这种方式在实际应用中几乎没有参考的意义。对于不可预知的修改所带来的问题，我们需要另外一种方式。

第二种方式在很大程度上可以满足我们的需求：

第二种：

1. 执行编译，使用“make –o HEADERFILE”，“HEADERFILE”为需要忽略更改的头文件，防止那些依赖于这个头文件的目标被重建。忽略多个头文件的修改可使用多个“-o HEADERFILE”。这样，除了独立依赖于头文件“HEADERFILE”的目标不被重建以外，其它的目标会根据规则来决定是否会被重建。**需要注意的是：“-o”参数的这种使用方式仅限于头文件（.h 文件），不能使用“-o”来指定源文件。**
2. 执行“make -t”命令。

8.5 替换变量定义

执行make时，一个含有“=”的命令行参数“V=X”的含义是将变量“V”的值设置为“X”。通过这种方式定义的变量会替代在Makefile中同名变量定义（如果存在，并且在Makefile中没有使用指示符“override”对这个变量进行生命），称之为命令行参数定义覆盖普通变量定义。

通常用这种方式来传递一个公共变量给 make。例如：在 Makefile 中，使用变量“CFLAGS”来指定编译参数，在 Makefile 中规则的命令一般都是这么写的：

cc -c $(CFLAGS) foo.c

这样就可以通过改变“CFLAGS”的值来控制编译选项。我们可以在 Makefile 中为它指定值。例如：

CFLAGS=-g

当直接执行“make”时，编译命令是“cc –c –g foo.c”。如果需要改变“CFLAGS”的定义，我们可以在命令行中执行“make CFLAGS='-g –O2'”，此时所执行的编译命令将是“cc –c –g –O2 foo.c”（在参数中如果包含空格或者shell的特殊字符，则需要将参数放在引号中）。对变量“CFLAGS”定义追加的功能就是使用这种方式来实现的（可参考 5.7 “override”指示符 一节中的例子）。

变量“CFLAGS”可以通过这种方式来实现，它是 make 的隐含变量之一。对于普通变量的定义，也可以通过这种方式来对它进行重新定义（覆盖 Makefile 中的定义）、或者实现变量值的追加功能。

通过命令行参数定义变量时，也存在两种风格的变量定义：递归展式定义和直接展开式定义（参考 5.2 两种变量定义 一节）。上例中使用递归展开式的定义（使用“= ”），也可以是直接展开式的（使用“:= ”）。除非在命令行中指定变量的值中包含其它变量或者函数的引用，否则这两种方式在这种情况下是等价的。

为了防止命令行参数的变量定义覆盖 Makefile 中的同名变量定义，可以在 Makefile 中使用指示符“override”声明这个变量。这一点前边的章节不止一次的提到过！

8.6 使用make进行编译测试

正常情况 make 在执行 Makefile 过程中，如果出现命令执行错误，会立即放弃执行过程并返回一个非 0 的状态。就是说错误发生点之后的命令将不会被执行。一个错误的发生就表明了终极目标将不能被重建，make 一旦检查到错误就会立刻终止执行。

假如我们在修改了一些源文件之后重新编译工程，当然了，我们所希望的是在某一个文件编译出错以后能够继续进行后续文件的编译。直到最后出现链接错误时才退出。这样的目的是为了了解所修改的文件中那些文件没有修改正确。在下一次编译之前能够对出现错误的所有文件进行改正。而不是编译一次改正一个文件，或者改正一个文件再编译一次。

我了实现我们这个目的，需要使用 make 的“-k”或者“--keep-going”命令行选项。这个参数的功能是告诉 make 当出现错误时继续执行，直到最后出现致命错误（无法重建终极目标）才返回非 0 并退出。例如：当编译一个.o 目标文件时出现错误，如果使用“make -k”执行，make 将不会在检测到这个错误时退出（虽然已经知道终极目标是不可能会重建成功的）；只是给出一个

错误消息，make 将继续重建其它需要重建的目标文件；直到最后出现致命错误才退出。在没有使用“-k”或者“—keep-going”时，make 在检测到错误时会立刻退出。

总之：在通常情况下，make 的目的是重建终极目标。当它在执行过程中一旦发现无法重建终极目标，就立刻以非 0 状态退出。当使用“-k”或者“--keep-going”参数时，执行的目的是为了测试重建过程，需要发现存在的所有问题，以便在下一次 make 之前进行修正。这也是调试 Makefile 或者查找源文件错误的一种非常有效的手段。

8.7 make的命令行选项

本节罗列出了 make 所支持的所有命令行参数(这些参数可以通过 make 的 man 手册查看)：

-b

-m

忽略，提供其它版本 make 兼容性。

-B

--always-make

强制重建所有的规则中出现的目标文件。

-C DIR

--directory=DIR

在读取Makefile之前，进入目录“DIR”，就是切换工作目录到“DIR”之后执行make。存在多个“-C”选项时，make的最终工作目录是每一个目录将是前一个相对路径。例如：“make –C / -C etc”等价于“make –C /etc”。被用在递归地make调用中。

-d

make 执行时打印出所有的调试信息。包括：make 认为那些文件需要重建；那些文件需要比较它们的最后修改时间、比较的结果；重建目标所要执行的命令；使用的隐含规则等。使用“-d”选项可以看到 make 构造依赖关系链、重建目标过程的所有信息，它等效于“—debug=a”.

—debug[=OPTIONS]

make 执行时输出调试信息。可以使用“OPTIONS”控制调试信息级别。默认是“OPTIONS=b”，“OPTIONS”的可能值为下列这些，首字母有效（all 和 aw 等效）。

a（all）

输出所有类型的调试信息，等效于“-d”选项。

b（basic）

输出基本调试信息。包括：那些目标过期、是否重建成功过期目标文件。

v（verbose）

“basic”级别之上的输出信息。包括：解析的 makefile 文件名，不需要重建文件等。

此选项目默认打开“basic”级别的调试信息。

i（implicit）

输出所有使用到的隐含规则描述。此选项目默认打开“basic”级别的调试信息。

j（jobs）

输出所有执行命令的子进程，包括命令执行的 PID 等。

m（makefile）

也就是 makefile，输出 make 读取 makefile，更新 makefile，执行 makefile 的信息。

-e

--environment-overrides

使用系统环境变量的定义覆盖Makefile中的同名变量定义（参考 5.9 系统环境变量 一节）。

-f=FILE

--file= FILE

--makefile= FILE

指定“FILE”为make执行的makefile文件。（参考 2.2 makefile文件的命名 一节）

-h

--help

打印帮助信息。

-i

--ignore-errors

执行过程中忽略规则命令执行的错误。（参考 4.4 命令的错误 一节）

-I DIR

--include-dir=DIR

指定被包含makefile文件的搜索目录。在Makefile中出现“include”另外一个文件时，将在“DIR”目录下搜索（参考 2.3 包含其它makefile文件 一节）。多个“-I”指定目录时，搜索目录按照指定顺序进行。

-j [JOBS]

--jobs[=JOBS]

指定可同时执行的命令数目。在没有指定“-j”参数的情况下，执行的命令数目将是系统允许的最大可能数目。存在多个“-j”参数时，尽最后一个“-j”指定的数目（“JOBS”）有效（参考 4.3 并发执行命令 一节）。

-k

--keep-going

执行命令错误时不终止make的执行，make尽最大可能的执行所有的命令，直到出现致

命错误才终止。参考 8.6 测试Makeifle 一节。

-l LOAD

--load-average[=LOAD]

—max-load[=LOAD]

告诉make当存在其它任务在执行时，如果系统负荷超过“LOAD”（浮点数表示的，参考 4.3 并发执行命令 一节），不再启动新任务。没有指定“LOAD”的“-l”选项将取消之前“-l”指定的限制。

-n

--just-print

--dry-run

--recon

只打印出所要执行的命令，但不执行命令。

-o FILE

--old-file= FILE

--assume-old= FILE

指定文件“FILE”不需要重建，即使相对于它的依赖已经过期。因此依赖于文件“FILE”的目标也不会被重建。注意：此参数不会通过变量“MAKEFLAGS”传递给子make进程（参考 4.6.3 命令行选项和递归 一小节）。关于“-o”参数的用法可参考 8.4 防止特定文件的重建 一节

-p

--print-data-base

命令执行之前，打印出 make 读取的 Makefile 的所有数据（包括规则和变量的值），同时打印出 make 的版本信息。如果执行打印这些数据信息（不打印其他信息、也不执行命令）可以使用“make -qp”命令。查看 make 执行前的预设规则和变量，可使用命令“make －p －f /dev/null”。这个选项为我们在复杂环境中调试 Makefile 提供了手段。

-q

--question

称为“询问模式”；不运行任何命令，并且无输出。make只是返回一个查询状态。返回状态为 0 表示没有目标需要重建，1 表示存在需要重建的目标，2 表示有错误发生。参考 8.3 替代命令的执行 一节

-r

--no-builtin-rules

取消所有make内嵌的隐含规则，可以在Makefile中使用模式规则来定义一个规则。同时选项“-r”会取消所有支持后追规则的隐含后缀列表，同样我们也可以在Makefile中使用“.SUFFIXES”定义我们自己的后缀规则。“-r”选项不会取消make内嵌的隐含变量（参

考 9.3 隐含变量 一节)。

-R

--no-builtin-variabes

取消 make 内嵌的隐含变量,当然我们可以在 Makefile 中明确定义某些变量。注意,“-R”选项同时打开“-r”选项。因为没有了隐含变量,隐含规则将失去意义(隐含规则是以内嵌的隐含变量为基础的)。

-s

--silent

--quiet

取消命令执行过程的打印。参考 4.1 命令回显 一节

-S

--no-keep-going

--stop

取消“-k”选项。在递归的make过程中子make通过“MAKEFLAGS”变量继承了上层的命令行选项。我们可以在子make中使用“-S”选项取消上层传递的“-k”选项(参考 4.6 make的递归执行 一节),或者取消系统环境变量“MAKEFLAGS”中的“-k”选项。

-t

—touch

和Linux的touch命令实现功能相同,更新所有目标文件的时间戳到当前系统时间。防止make对所有过时目标文件的重建。可参考 8.3 替代命令的执行 一节

-v

--version

查看 make 版本信息。

-w

--print-directory

在 make 进入一个目录读取 Makefile 之前打印工作目录。这个选项可以帮助我们调试 Makefile,跟踪定位错误。使用“-C”选项时默认打开这个选项。参考本节前半部分“-C”选项的描述。

--no-print-directory

取消“-w”选项。可以是用在递归的 make 调用过程中,取消“-C”参数的默认打开“-w”功能。

-W FILE

--what-if= FILE

--new-file= FILE

--assume-file= FILE

设定文件“FILE”的时间戳为当前时间，但不改变文件实际的最后修改时间。此选项主要是为实现了对所有依赖于文件“FILE”的目标的强制重建。参考 8.3 替代命令的执行 一节

--warn-undefined-variables

在发现Makefile中存在对没有定义的变量进行引用时给出告警信息。此功能可以帮助我们调试一个存在多级套嵌变量引用的复杂Makefile。但是：我们建议在书写Makefile时尽量避免超过三级以上的变量套嵌引用。参考 5.3.2 变量的套嵌引用 一小节

第九章：make的隐含规则

9 使用隐含规则

在 Makefile 中重建一类目标的标准规则在很多场合需要用到。例如：根据.c 源文件创建对应的.o 文件，传统方式是使用 GNU 的 C 编译器。

“隐含规则”为 make 提供了重建一类目标文件通用方法，不需要在 Makefile 中明确地给出重建特定目标文件所需要的细节描述。例如：典型地；make 对 C 文件的编译过程是由.c 源文件编译生成.o 目标文件。当 Makefile 中出现一个.o 文件目标时，make 会使用这个通用的方式将后缀为.c 的文件编译称为目标的.o 文件。

另外，在 make 执行时根据需要也可能是用多个隐含规则。比如：make 将从一个.y 文件生成对应的.c 文件，最后再生成最终的.o 文件。就是说，只要目标文件名中除后缀以外其它部分相同，make 都能够使用若干个隐含规则来最终产生这个目标文件（当然最原始的那个文件必须存在）。例如；可以在 Makefile 中这样来实现一个规则：“foo : foo.h”，只要在当前目录下存在“foo.c”这个文件，就可以生成“foo”可执行文件。本文前边的很多例子中已经使用到了隐含规则。

内嵌的“隐含规则”在其所定义的命令行中，会使用到一些变量（通常也是内嵌变量）。我们可以通过改变这些变量的值来控制隐含规则命令的执行。例如：内嵌变量“CFLAGS”代表了 GCC 编译器编译源文件的编译选项，我们就可以在 Makefile 中重新定义它，来实现我们编译源文件所要使用的参数。

尽管我们不能改变 make 内嵌的隐含规则，但是我们可以使用模式规则重新定义自己的隐含规则。也可以使用后追规则来重新定义隐含规则，后缀规则存在某些限制（目前版本 make 保存它的原因是为了兼容以前版本）。通常模式规则更加清晰明了。

9.1 隐含规则的使用

使用 make 内嵌的隐含规则，我们的 Makefile 中就不需要明确给出重建某一个目标的命令，甚至可以不用写出明确的规则。make 会自动根据已存在（或者可以被创建）的源文件类型来启动

相应的隐含规则。例如：

```
foo : foo.o bar.o
        cc -o foo foo.o bar.o $(CFLAGS) $(LDFLAGS)
```

这里并没有给出重建文件“foo.o”的规则，make 执行这条规则时，无论文件“foo.o”存在与否，都会试图根据隐含规则来重建这个文件（就是试图重新编译文件“foo.c”或者其它类型的源文件）。

make 执行过程中找到的隐含规则，提供了此目标的基本依赖关系，确定了目标的依赖文件（通常是源文件，不包含对应的头文件依赖），和重建目标需要使用的命令行。隐含规则所提供的依赖文件只是一个最基本的（通常它们之间的对应关系为：“EXENAME.o”对应“EXENAME.c”、“EXENAME”对应于“EXENAME.o”）。当需要增加这个目标的依赖文件时，要在 Makefile 中使用没有命令行的规则来明确说明。

每一个内嵌的隐含规则中都存在一个目标模式和依赖模式关系，而且一个目标模式可以对应多个依赖模式。例如：一个.o 文件的目标可以存在多个源文件模式：C 编译器编译对应的.c 源文件、或者 Pascal 编译器编译.p 的源文件来实现。因此 make 会根据不同的源文件来使用不同的编译器。对于“foo.c”就是用 GCC 编译，对于“foo.p”就使用 Pascal 编译器编译。

上边提到，make 会自动根据已存在（或者可以被创建）的源文件类型来启动相应的隐含规则。这里说说的“可被创建”的文件是指：这个文件在 Makefile 中被作为目标或者依赖明确的提及，或者可以根据已存在的文件使用其它的隐含规则来创建。当一个隐含规则的目标是另外一个隐含规则的依赖时，我们称它们是一个隐含规则链。

通常，make会对那些没有命令行的规则、双冒号规则寻找一个隐含规则来执行。作为一个规则的依赖文件，在没有一个规则明确描述它的依赖关系的情况下；make会将其作为一个目标并为它搜索一个隐含规则，试图重建它（参考 9.8 隐含规则的搜索算法 一节）。

注意：给目标文件指定明确的依赖文件并不会影响隐含规则的搜索。我们来看一个例子：

```
foo.o: foo.p
```

这个规则指定了“foo”的依赖文件是“foo.p”。但是如果在工作目录下存在同名.c源文件“foo.c”。执行make的结果就不是用“pc”编译“foo.p”来生成“foo”，而是用“cc”编译“foo.c”来生成目标文件。这是因为在隐含规则列表中对.c文件的隐含规则处于.p文件隐含规则之前。可参考 9.2 make隐含规则一览 一节。

当需要给目标指定明确的重建规则时，规则描述中就不能省略命令行，这个规则必须提供明确的重建命令来说明目标需要重建所需要的动作。为了能够在存在“foo.c”的情况下编译“foo.p”。规则可以这样写：

```
foo.o: foo.p
        pc $< -o $@
```

这一点在多语言实现的工程编译中，需要特别注意！否则编译出来的可能就不是你想要得程序。

另外：当我们不想让make为一个没有命令行的规则中的目标搜索隐含规则时，我们需要使用空命令来实现。

最后让我们来看一个简单的例子，之前在 5.10 目标指定变量 一节的例子我们就可以简化为：

```
# sample Makefile

CUR_DIR = $(shell pwd)
INCS := $(CUR_DIR)/include
CFLAGS := -Wall –I$(INCS)

EXEF := foo bar

.PHONY : all clean
all : $(EXEF)

foo : CFLAGS+=-O2
bar : CFLAGS+=-g

clean :
    $(RM) *.o *.d $(EXES)
```

例子中没有出现任何关于源文件的描述。所有剩余工作全部交给了 make 去处理，它会自动寻找到相应规则并执行、最终完成目标文件的重建。

隐含规则为我们提供了一个编译整个工程非常高效的手段，一个大的工程中毫无例外的会用到隐含规则。实际工作中，灵活运用 GNU make 所提供的隐含规则功能，可以大大提供效率。

9.2 make的隐含规则一览

本节所罗列出了GUN make常见的一些内嵌隐含规则，除非在Makefile中对此规则有明确定义、或者使用命令行“-r”或者“-R”参数（参考 8.7 make的命令行选项 一节），否则这些隐含的规则将有效。

需要说明的是：即使我们没有使用命令行参数“-r”，在make中也并不是所有的这些隐含规则都被定义了。其实，很多的这些看似预定义的隐含规则在make执行时，实际是用后缀规则来实现的；因此，它们依赖于make中的“后缀列表”（也就是目标.SUFFIXES的一个后缀列表）。make的默认后缀列表为：“.out”、“.a”、“.ln”、“.o”、“.c”、“.cc”、“.C”、“.p”、“.f”、“.F”、“.r”、“.y”、“.l”、“.s”、“.S”、“.mod”、“.sym”、“.def”、“.h”、“.info”、“.dvi”、“.tex”、“.texinfo”、“.texi”、“txinfo”、“.w”、“.ch”、“.web”、“.sh”、“.elc”、“el”。所有我们下边将提到的隐含规则，如果其依赖文件中某一个满足列表中列出的后缀，则是后缀规则。我们可以通过“.SUFFIXES”来更改默认的后缀列表，这样做可能会是许多默认预定义的规则无效。参考 9.7 后缀规则 一节。以下是常用的一些隐含规则（对于不常见的隐含规则这里没有描述）：

1. **编译C程序**

"N.o"自动由"N.c" 生成，执行命令为"$(CC) -c $(CPPFLAGS) $(CFLAGS)"。

2. **编译C++程序**

"N.o"自动由"N.cc"或者"N.C" 生成，执行命令为"$(CXX) -c $(CPPFLAGS) $(CFLAGS)"。建议使用".cc"作为C++源文件的后缀，而不是".C"

3. **编译Pascal程序**

"N.o"自动由"N.p"创建，执行命令时"$(PC) -c $(PFLAGS)"。

4. **编译Fortran/Ratfor程序**

"N.o"自动由"N.r"、"N.F"或者"N.f" 生成，根据源文件后缀执行对应的命令：

.f — "$(FC) –c $(FFLAGS)"

.F — "$(FC) –c $(FFLAGS) $(CPPFLAGS)"

.r — "$(FC) –c $(FFLAGS) $(RFLAGS)"

5. **预处理Fortran/Ratfor程序**

"N.f"自动由"N.r"或者"N.F" 生成。此规则只是转换Ratfor或有预处理的Fortran程序到一个标准的Fortran程序。根据源文件后缀执行对应的命令：

.F — "$(FC) –F $(CPPFLAGS) $(FFLAGS)"

.r — "$(FC) –F $(FFLAGS) $(RFLAGS)"

6. **编译Modula-2程序**

"N.sym"自动由"N.def" 生成，执行的命令是："$(M2C) $(M2FLAGS) $(DEFFLAGS)"。"N.o"自动由"N.mod"生成，执行的命令是："$(M2C) $(M2FLAGS) $(MODFLAGS)"。

7. **汇编和需要预处理的汇编程序**

"N.s"是不需要预处理的汇编源文件，"N.S"是需要预处理的汇编源文件。汇编器为"as"。

"N.o" 可自动由"N.s"生成，执行命令是："$(AS) $(ASFLAGS)"。

"N.s" 可由"N.S"生成，C预编译器"cpp"，执行命令是："$(CPP) $(CPPFLAGS)"。

8. **链接单一的object文件**

"N"自动由"N.o"生成，通过C编译器使用链接器（GUN ld），执行命令是："$(CC) $(LDFLAGS) N.o $(LOADLIBES) $(LDLIBS)"。

此规则仅适用：由一个源文件可直接产生可执行文件的情况。当需要有多个源文件共同来创建一个可执行文件时，需要在Makefile中增加隐含规则的依赖文件。例如：

```
x : y.o z.o
```

当"x.c"、"y.c"和"z.c"都存在时，规则执行如下命令：

```
cc -c x.c -o x.o
cc -c y.c -o y.o
```

```
cc -c z.c -o z.o
cc x.o y.o z.o -o x
rm -f x.o
rm -f y.o
rm -f z.o
```

在复杂的场合，目标文件和源文件之间不存在向上边那样的名字对应关系时（“x”和“x.c”对应，因此隐含规则在进行链接时，自动将“x.c”作为其依赖文件）。需要明确给出描述目标依赖关系的规则。

通常，GCC在编译源文件时（根据源文件的后缀名来启动相应的编译器），如果没有指定“-c”选项，GCC会在编译完成之后调用“ld”连接成为可执行文件。

9. **Yacc C程序**

“N.c”自动由“N.y”，执行的命令：“$(YACC) $(YFALGS)”。（“Yacc”是一个语法分析工具）

10. **Lex C程序时的隐含规则。**

“N.c”自动由“N.l”，执行的命令是：“$(LEX) $(LFALGS)”。（关于“Lex”的细节请查看相关资料）

这里没有列出所有的隐含规则，仅列出我个人在实际工作中涉及到的。没有涉及的很难对英文文档进行深入地说明和理解。如果那些没有提到的各位有所使用，或者能够详细的描述可以添加到这个文档中！

在隐含规则中，命令行中的实际命令是使用一个变量计算得到，诸如：“COMPILE.c”、“LINK.o”（这个在前面也看到过）和“PREPROCESS.S”等。这些变量被展开之后就是对应的命令（包括了命令行选项），例如：变量“COMPILE.c”的定义为 “cc -c”（如果Makefile中存在“CFLAGS”的定义，它也会存在于这个变量中）。

make会根据默认的约定，使用“COMPILE.x”来编译一个“.x”的文件。类似地使用“LINK.x”来连接“.x”文件；使用“PREPROCESS.x”对“.x”文件进行预处理。

每一个隐含规则在创建一个文件时都是用了变量“OUTPUT_OPTION”。make执行命令时根据命令行参数来决定它的值，当命令行中没有包含“-o”选项时，它的值为：“-o $@”，否则为空。建议在规则的命令行中明确使用“-o”选项执行输出文件路径。这是因为在编译一个多目录的工程时，如果我们的Makefile中使用了“VPATH”指定搜索目录 时，编译后的.o文件或者其它文件会出现在和源文件不同的目录中。在有些系统的编译器不接受命令行的“-o”参数，而Makefile中包含“VPAT”的情况时，输出文件可能会出现在错误的目录下。解决这个问题的方式就是将“OUTPUT_OPTION”的值赋为“;mv $*.o $@”，其功能是将编译完成的.o文件改变为规则中

的目标文件。

9.3 隐含变量

内嵌隐含规则的命令中，所使用的变量都是预定义的变量。我们将这些变量称为“隐含变量”。这些变量我们可以对它进行修改：在Makefile中、通过命令行参数或者设置系统环境变量的的方式来对它进行重定义。无论是用那种方式，只要make在运行时它的定义有效，make的隐含规则都会使用这些变量。当然，也可以使用“-R”或“--no－builtin-variables”选项来取消所有的隐含变量（同时将取消了所有的隐含规则）。

例如，编译.c源文件的隐含规则为：“$(CC) －c $(CFLAGS) $(CPPFLAGS)”。默认的编译命令是“cc”，执行的命令是：“cc –c”。我们可以同上述的任何一种方式将变量“CC”定义为“ncc”，那么编译.c源文件所执行的命令将是“ncc -c”。同样我们可以对变量“CFLAGS”进行重定义。对这些变量重定义后如果需要整个工程的各个子目录有效，同样需要使用关键字“export”将他们导出；否则目录间编译命令可能出现不一致。关于对.c源文件的编译时，隐含规则使用“$(CC)”来引用编译器；“$(CFLAGS)”引用编译选项。

隐含规则中所使用的变量（隐含变量）分为两类：1．代表一个程序的名字（例如：“CC”代表了编译器这个可执行程序）。2．代表执行这个程序适用的参数（例如：变量“CFLAGS”），多个参数使用空格分开。当然也允许在程序的名字中包含参数。但是这种方式建议不要使用。**对于参数的描述我们应该将它们集中在一个变量中，即就是整个工程存在一个必需的参数，我们也可以把它放置一个特殊命名的变量定义中，通用的代表参数的那个变量的定义中对它引用；通过这种方式来实现，而不是将它和所要执行的程序名定义在一个变量中。我们的Makefile要尽量做到清晰、明了、干练的管理一个工程。而不要各个变量之间存在错综复杂的关系！模块化的编程思想也同样适用于我们编写Makefile。**

以下是一些作为程序名的隐含变量定义：

9.3.1 代表命令的变量

AR

函数库打包程序，可创建静态库.a文档。默认是“ar”。

AS

汇编程序。默认是“as”。

CC

C编译程序。默认是“cc”。

CXX

C++编译程序。默认是“g++”。

CO

从 RCS中提取文件的程序。默认是“co”。

CPP

C程序的预处理器（输出是标准输出设备）。默认是“$(CC) -E”。

FC

编译器和预处理Fortran 和 Ratfor 源文件的编译器。默认是“f77”。

GET

从SCCS中提取文件程序。默认是“get”。

LEX

将 Lex 语言转变为 C 或 Ratfo 的程序。默认是“lex”。

PC

Pascal语言编译器。默认是“pc”。

YACC

Yacc文法分析器（针对于C程序）。默认命令是“yacc”。

YACCR

Yacc文法分析器（针对于Ratfor程序）。默认是“yacc -r”。

MAKEINFO

转换Texinfo源文件（.texi）到Info文件程序。默认是“makeinfo”。

TEX

从TeX源文件创建TeX DVI文件的程序。默认是“tex”。

TEXI2DVI

从Texinfo源文件创建TeX DVI 文件的程序。默认是“texi2dvi”。

WEAVE

转换Web到TeX的程序。默认是“weave”。

CWEAVE

转换C Web 到 TeX的程序。默认是“cweave”。

TANGLE

转换Web到Pascal语言的程序。默认是“tangle”。

CTANGLE

转换C Web 到 C。默认是“ctangle”。

RM

删除命令。默认是“rm -f”。

9.3.2 命令参数的变量

下边的是代表命令执行参数的变量。如果给出默认值则默认值为空。

ARFLAGS

执行“AR”命令的命令行参数。默认值是“rv”。

ASFLAGS

执行汇编语器“AS”的命令行参数（明确指定“.s”或“.S”文件时）。

CFLAGS

执行“CC”编译器的命令行参数（编译.c源文件的选项）。

CXXFLAGS

执行“g++”编译器的命令行参数（编译.cc源文件的选项）。

COFLAGS

执行“co”的命令行参数（在RCS中提取文件的选项）。

CPPFLAGS

执行C预处理器“cc -E”的命令行参数（C 和 Fortran 编译器会用到）。

FFLAGS

Fortran语言编译器“f77”执行的命令行参数（编译Fortran源文件的选项）。

GFLAGS

SCCS “get”程序参数。

LDFLAGS

链接器参数。（如：“ld”）

LFLAGS

Lex文法分析器参数。

PFLAGS

Pascal语言编译器参数。

RFLAGS

Ratfor 程序的Fortran 编译器参数。

YFLAGS

Yacc文法分析器参数。

9.4 make隐含规则链

有时候，一个文件可以由一系列隐含规则进行创建。例如：文件“N.o”的创建过程可以是这样的：首先执行“yacc”由“N.y”生成文件“N.c”，之后执行“cc”将“N.c”编译成为“N.o”。我们把这样的一个系列称为一个“链”。

我们来看一下上边例子的执行过程。有两种情况：

1. 如果文件“N.c”存在或者它在Makefile中被提及，那就不需要进行其它搜索，make处理的过程是：首先，make可以确定出“N.o”可由“N.c”创建；之后，make试图使用隐

含规则来重建“N.c”。它会寻找“N.y”这个文件，如果“N.y”存在，则执行隐含规则来重建“N.c”这个文件。之后再由“N.c”重建“N.o”；当不存在“N.y”文件时，直接编译“N.c”生成“N.o”。

2. 文件“N.c”不存在也没有在Makefile中提及的情况，只要存在“N.y”这个文件，那么make也会经过这两个步骤来完成重建“N.o”（N.y → N.c → N.o）的动作。这种情况下，文件“N.c”作为一个中间的过程文件。make过程中如果需要一个中间文件才能完成目标的重建，make将会自动将这个中间文件加入到依赖关系链中（和Makefile中明确提及的目标作相同处理），并根据隐含规则来重建它。

make的中间过程文件和那些明确指定的文件在规则的使用上完全相同。但make在对待中间过程文件和普通文件存在下列两点不同：

第一：中间文件不存在时，make处理两者的方式不同。对于一个普通文件来说，因为Makefile中有明确的提及，此文件可能是作为一个目标的依赖，make在执行它所在的规则前会试图重建它。但是对于中间文件，因为没有明确提及，make不会去试图重建它。除非这个中间文件的依赖文件（上例第二种情况中的文件“N.y”；N.c是中间过程文件）被更新。

第二：如果make执行时需要用到一个中间过程文件，默认的动作是：这个过程文件在make执行结束之后会被删除（make会在删除中间过程文件时打印出执行的命令以显示那些文件被删除了）。因此一个中间过程文件在执行完make之后就不再存在了。

在Makefile中明确提及的所有文件都不会被作为中间过程文件来处理，这是缺省动作。不过可以在Makefile中使用特殊目标“.INTERMEDIATE”来声明那些文件需要被作为中间过程文件来处理（这些文件作为目标“.INTERMEDIATE”的依赖文件罗列），即使它们在Makefile中有明确的提及。这些作为特殊目标“.INTERMEDIATE”依赖的文件在make执行结束之后会被自动删除。

而另一方面，如果我们希望保留某些中间过程文件（它没有在Makefile中被提及），不希望make结束时自动删除。可以在Makefile中使用特使目标“.SECONDARY”来声明这些文件（这些文件将被作为“secondary”文件；需要保留的文件作为特殊目标“.SECONDARY”的依赖文件罗列）。“secondary”文件也同时被作为中间过程文件来对待。

需要保留中间过程文件还存在另外一种实现方式。例如需要保留所有.o的中间过程文件，我们可以将.o文件的模式（%.o）作为特殊目标“.PRECIOUS”的依赖，这样就可以实现保留所有的.o中间过程文件。可参考 4.5 中断make的执行 一节

一个“链”可以包含两个以上的隐含规则的调用。一个隐含规则在一个“链”只能出现一次。否则会出现像“foo”依赖“foo.o.o”甚至“foo.o.o.o.o...”的不合逻辑的情况发生。因为如果允许同一个“链”多次调用同一隐含规则（N : N.o; $(LINK.o) $(LDFLAGS) N.o $(LOADLIBES) $(LDLIBS) ），会导致make进入到无限的循环中。

隐含规则链中的某些隐含规则，在某些情况会被优化处理。例如：从文件“foo.c”创建可执行文件“foo”，这个过程可以是：经隐含规则将“foo.c”编译生成“foo.o”文件，之后再使用另一个隐含规则来完成对“foo.o”的链接，最后生成执行文件“foo”。这个过程中编译和链接使用隐含规则链中的两个独立的规则。但是实际情况是，完成编译和链接是在一个规则中完成的，它使用“cc foo.c foo”命令直接来完成。make的隐含规则表中，所有可用的优化规则处于首选地位。

9.5 模式规则

模式规则类似于普通规则，只是在模式规则中，目标的定义中需要包含“%”字符（确切地说是一个），包含“%”的目标被用来匹配一个文件名，“%”可以匹配任何非空字符串。规则的依赖文件中同样可以使用“%”，依赖中的“%”的取值情况由目标中的“%”来决定。例如：模式规则“%.o : %.c”表示了所有的.o文件是由.c文件来生成的。我们可以使用模式规则来定义一个隐含规则。

需要注意的是：“%”的替换是在规则的其他变量和函数引用扩展完成之后进行的，变量和函数的展开一般发生在make读取Makefile时（变量和函数的展开可参考 第 5 章 使用变量 和 第 7 章 make的函数 ），而模式规则中的“%”则发生在make的执行过程。

9.5.1 模式规则介绍

关于模式规则的概念本节的开始部分已经有了描述。隐含规则中目标文件，“%.c”匹配所有的以“.c”结尾的所有文件名（匹配的文件名长度最少为3个字母），“s%.c”匹配了所有第一个字母为“s”并且以“.c”结尾的文件名，它的文件名长度最小为5个字母。文件名中“%”匹配的部分称为“茎”（前面我们已经提到过，可参考 3.12 静态模式 一节）。

模式规则中，如果依赖文件同样存在模式字符“%”，那么它也代表规则中目标文件匹配之后所得到的“茎”。使用模式规则时，目标文件匹配的同时，根据目标的匹配所得的“茎”产生对应的依赖文件，此文件必须是存在的或者可被创建的。

因此，一个模式规则的格式为：

%.o : %.c ; COMMAND...

这个模式规则指定了如何由文件“N.c”来创建文件“N.o”，文件“N.c”应该是已存在的或者可被创建的。

模式规则中依赖文件也可以不包含模式字符“%”。当模式规则的依赖文件名中不包含模式字符“%”时，它的含义是所有符合目标模式的目标文件都依赖于一个指定的文件（例如：%.o : debug.h，表示所有的.o目标文件都依赖于头文件“debug.h”）。这样的模式规则在很多场合是非常有用的。

同样一个模式规则可以由多个目标文件。但是多个目标的模式规则和普通的多目标的规则有些不同，普通多目标的规则的处理是将每一个目标作为一个独立的规则来处理，所以多个目标就就对应多个独立的规则（这些规则各自有自己的命令行，各个规则的命令行可能相同）。但对于多目标的模式规则，所有规则的目标共同拥有依赖文件和规则的命令行，当文件符合多个目标模式中的任何一个时，规则定义的重建命令可能将会执行；因为多个目标共同用于这个规则的命令行，因此一次命令执行之后，规则不会再去检查是否需要重建符合其它模式的目标。据个例子会描述得更清楚一些：

```
#sample Makefile

Objects = foo.o bar.o
CFLAGS := -Wall

%x : CFLAGS += -g
%.o : CFLAGS += -O2

%.o %.x : %.c
    $(CC) $(CFLAGS) $< -o $@
```

当我们在命令行中执行“make foo.o foo.x”时，会看到只有一个文件“foo.o”被创建了，同时make会提示“foo.x”文件是最新的（其实“foo.x”并没有被创建）。它表明了多目标的模式规则在make处理时是作为一个整体来处理的。这是多目标模式规则和多目标的普通规则的不同之处。大家可以把这个例子改写为普通的多目标规则试试看。

最后需要说明的是：

1. 模式规则在Makefile中的顺序需要注意，当一个目标文件符合多个模式规则的目标时，make将会按照第一个找到的作为重建它的规则。
2. 在Makefile中指定的模式规则会覆盖隐含的模式规则。就是说在Makefile中明确指定的会替代隐含的模式规则。
3. 另外，依赖文件存在或者被提及的规则，优先于那些需要使用隐含规则来创建其依赖文件的规则。

9.5.2 模式规则示例

本小节来看一些关于模式规则的例子，这些模式规则是make实际预定义的一些模式规则。首先我们来看编译.c文件到.o文件的隐含模式规则：

```
%.o : %.c
    $(CC) -c $(CFLAGS) $(CPPFLAGS) $< -o $@
```

这个规则描述了一个.o目标文件如何由对应的.c文件创建。规则的命令行中使用了自动化变量“$< ”和“$@”，其中自动化变量“$< ”代表规则的依赖，“$@”代表规则的目标文件。此规则在执行

时，命令行中的自动化变量将根据实际的目标文件区对应值。关于自动化变量可参考 9.5.3 自动化变量 一节

make中第二个内嵌模式规则是：

```
% :: RCS/%,v
        $(CO) $(COFLAGS) $<
```

这个规则的含义是：任何一个文件“X”都可以由目录“RCS”下的相应文件“x.v”生成。规则的目标为“%”，因此只要存在相对应的依赖文件（N.v），此目标可以创建文件（N）。双冒号表示该规则是最终规则，意味着规则的依赖文件不是中间过程文件。参考 9.6 万用规则 一节

另外，一个具有多目标的隐含规则是：

```
%.tab.c %.tab.h: %.y
        bison -d $<
```

这个规则是一个多目标的模式规则，关于多目标的特征可参考 9.5.1 模式规则介绍 一小节最后一个例子。

9.5.3 自动化变量

在模式规则中，规则的目标和依赖文件名代表了一类文件名。命令是对所有这一类文件重建过程的描述，显然，在命令中不能指定特定的文件名，否则模式规则将没有了意义。那么在模式规则的命令行中该如何表示文件，将成我们这一小节的讨论重点。make中使用了“自动环变量”来实现这个目的，自动化变量的取值是根据具体的规则决定的，就是说对不同的规则其所代表的文件名不同。

前边我们也看到了很多例子中使用到了自动化变量。下面对所有的自动化变量进行说明：

$@

代表规则中的目标文件名。如果目标是一个文档（Linux中，一般称.a文件为文档），那么它代表这个文档的文件名。在多目标的模式规则中，它代表的是哪个触发规则被执行的目标文件名。

$%

规则的目标文件是一个静态库文件时，代表静态库的一个成员名。例如，规则的目标是“foo.a(bar.o)”，那么，“$%”的值就为“bar.o”，“$@”的值为“foo.a”。如果目标不是函数库文件，其值为空。

$<

规则的第一个依赖文件名。如果是隐含规则，则它代表通过目标指定的第一个依赖文件。

$?

所有比目标文件更新的依赖文件列表，空格分割。如果目标是静态库文件名，代表的是库成

员（.o文件）的更新情况。

$^

规则的所有依赖文件列表，使用空格分隔。如果目标是静态库文件名，它所代表的只能是所有库成员（.o文件）名。一个文件可重复的出现在目标的依赖中，变量“$^ ”只记录它的一次引用情况。就是说变量“$^ ”会去掉重复的依赖文件。

$+

类似“$^ ”，但是它保留了依赖文件中重复出现的文件。主要用在程序链接时，库的交叉引用场合。

$*

在模式规则和静态模式规则中，代表“茎”。“茎”是目标模式中“% ”所代表的部分（当文件名中存在目录时，“茎”也包含目录（斜杠之前）部分，可参考 9.5.4 模式的匹配 一小节）。例如：文件“dir/a.foo.b”，当目标的模式为“a.%.b”时，“$* ”的值为“dir/a.foo”。“茎”对于构造相关文件名非常有用。

自动化变量“$* ”需要两点说明：

- 对于一个明确指定的规则来说不存在“茎”，这种情况下“$* ”所代表的值发生变化。此时，如果目标文件名带有一个可识别的后缀（参考 9.7 后缀规则 一节），那么“$* ”表示文件中除后缀以外的部分。例如：“foo.c”则“$* ”的值为：“foo”，因为.c是一个可识别的文件后缀名。GUN make对明确规则的这种奇怪的处理行为是为了和其它版本的make兼容。通常，在除静态规则和模式规则以外，明确指定目标文件的规则中避免使用这个变量。
- 当明确指定文件名的规则中目标文件名包含不可识别的后缀时，此变量为空。

自动化变量“$?”在显式规则中也是非常有用的，规则中可以使用它来指定只对更新的依赖文件进行操作。例如，函数库文件“libN.a”，它由一些.o文件组成。如下的规则实现了根据变化的.o文件更新库文件：

```
lib: foo.o bar.o lose.o win.o
     ar r lib $?
```

上述列出的自动量变量中。其中有四个在规则中代表一个文件名（$@、$< 、$%、$* ）。而其它三个的在规则中代表一个文件名的列表。GUN make中，还可以通过这七个自动化变量来获取一个完整文件名中的目录部分或者具体文件名，需要在这些变量中加入“D”或者“F”字符。这样就形成了一系列变种的自动环变量。这些变量在以前版本的make中使用，在当前版本的make中，可以使用“dir”或者“notdir”函数来实现同样的功能（可参考 7.3 文件名处理函数 一节）。

$(@D)

代表目标文件的目录部分（去掉目录部分的最后一个斜杠）。如果“$@”是“dir/foo.o”，

那么“$(@D)”的值为“dir”。如果“$@”不存在斜杠，其值就是“.”（当前目录）。注意它和函数“dir”的区别！

$(@F)

目标文件的完整文件名中除目录以外的部分（实际文件名）。如果“$@”为“dir/foo.o”，那么“$(@F)”只就是“foo.o”。“$(@F)”等价于函数“$(notdir $@)”。

$(*D)

$(*F)

分别代表目标“茎”中的目录部分和文件名部分。

$(%D)

$(%F)

当以如“archive(member)”形式静态库为目标时，分别表示库文件成员“member”名中的目录部分和文件名部分。它仅对这种形式的规则目标有效。

$(<D)

$(<F)

分别表示规则中第一个依赖文件的目录部分和文件名部分。

$(^D)

$(^F)

分别表示所有依赖文件的目录部分和文件部分（不存在同一文件）。

$(+D)

$(+F)

分别表示所有依赖文件的目录部分和文件部分（可存在重复文件）。

$(?D)

$(?F)

分别表示被更新的依赖文件的目录部分和文件部分。

需要说明的是：在讨论自动化变量时，为了和普通变量（如：“CFLAGS”）的区别，我们直接使用了“$<”的形式。这只是仅仅为了和普通变量区别，没有别的目的。其实对于自动环变量和普通变量一样，代表规则第一个依赖文件名的变量名实际上是“<”，我们完全可以使用“$(<)”来替代“$<”。我们在引用自动化变量时通常的做法是形如“$<”，因为自动化变量是一个字符。

GUN make提供了支持“Sysv”的特性，允许在规则的依赖列表中使用特殊的变量引用（一般的自动化变量只能在规则的命令行中被引用）“$$@”、“$$(@D)”和“$$(@F)”（注意：要使用“$$”），它们分别代表了“目标完整的文件名”、“目标文件名中的目录部分”和“目标的实际文件名部分”。

这三个特殊的变量只能用在明确指定目标文件名的规则中或者是静态模式规则中，不能使用在

于隐含规则中。另外Sysv make和GNU make对规则依赖的处理也不尽相同。Sysv make会对规则的依赖进行两次替换展开，而GUN make对依赖列表的处理只有一次，对其中的引用直接进行展开。

自动化变量存在这个有点古怪的特性完全是为了兼容Sysv 版本的makefile文件。我们在使用GNU make时可以不考虑这个特性，也可以在Makefile中使用伪目标“.POSIX”来禁止这一特性。

9.5.4 模式的匹配

通常，模式规则中目标模式由前缀、后缀、模式字符“%”组成，这三个部分中可以同时两部分为空。实际的文件名应该是以模式指定的前缀开始、后缀结束的任何文件名。文件名中除前缀和后缀以外的所有部分我们称之为“茎”（模式字符“%”可以代表若干字符。因此：模式“%.o”所匹配的文件“test.c”中“test”就是“茎”）。模式规则中依赖文件名的确定过程是：在处理模式规则时。首先根据规则定义的目标模式匹配实际的目标文件，确定“茎”，之后使用 “茎”替代规则依赖文件名中的模式字符“%”，生成依赖文件名。这样就形成了一个明确指定了目标和依赖文件的规则。例如模式规则：“%.o：%.c”，当“test.o”需要重建时将形成规则“test.o：test.c”。

当目标模式中包含斜杠（目录部分）。在进行目标文件匹配时，文件名中包含的目录字符串在匹配之前被移除，只进行基本文件名的匹配；匹配成功后，再目录加入到匹配之后的字符串之前形成“茎”。文字的描述看起来比较复杂，来看一个例子：例如目标模式为“e%t”，文件“src/eat”匹配于这个目标模式，那么“茎”就是“src/a”；模式规则中依赖文件的产生是：首先使用“茎”非目录部分（“a”）替代依赖文件中的模式字符“%”，之后再将目录部分（“src/”）加入到形成的依赖文件名之前构成依赖文件的全路径名。这里如果模式规则的依赖模式为“c%r”，则那么目标“src/eat”对应的依赖文件就为“src/car”。

9.5.5 万用规则

当一个模式规则的目标为“%”（它可以匹配任何文件名）时，我们称这个规则为万用规则。它非常有用，但是使用它可能会影响make的效率，因为make执行时将会为所有的规则的目标和依赖文件考虑万用规则。

假如在一个存在万用规则的Makefile中提及了文件“foo.c”。为了创建这个目标文件，make可能需要查找以下规则来创建这个目标：1. 通过连接一个.o文件“foo.c.o”来实现；2. 通过使用C编译和连接程序由文件“foo.c.c”创建来创建；3. 编译连接Pascal程序“foo.c.p”来创建……，make会试图使用所有可能的隐含规则去创建这个目标。

当然，我们都清楚这样的一个过程是没有必要的，我们很清楚“foo.c”是一个.c原文件，而不是一个可执行程序。make试图根据这些隐含规则来创建这个文件，由于目标的依赖文件（“foo.c.o”、“foo.c.c”等）不存在，最终这些隐含的规则无法被执行。但是存在万用规则，make

所要考虑的情况就会很多，导致了make的执行效率会很低。

为了避免由于万用规则所带来的效率问题，可以对万用规则的使用加以限制。通常有两种方式来实现，在定义万用规则时可以使用两者之一来对其进行限制。

1. 将万用规则设置为最终规则，定义时使用双冒号规则。作为最终规则，此规则只有在它的依赖文件存在时才能被应用，即使它的依赖可以由隐含规则创建也不行。就是说，最终规则中没有进一步的“链”。

例如，从RCS和SCCS文件中提取源文件的内建的隐含规则是最终规则。因此如果文件“foo.c,v”不存在，make就不会试图从一个中间文件“foo.c,v.o”或“RCS/SCCS/s.foo.c,v”去创建它。 RCS 和 SCCS 文件一般都是最终源文件，它不能由其它任何文件重建；make可以记录它的时间戳，但不寻找重建它们的方式。

如果万用规则每一定义为最终规则，它就是非最终规则。非最终的万用规则不会被用在那些有明确模式指定的目标和依赖文件中。就是说Makefile中如果存在和此文件匹配的模式规则（非万用规则），那么对于这个目标来说其重建的规则就不是万用规则而是那个模式匹配的非万用规则。

例如，对于文件“foo.c”，如果在Makefile中同时存在一个万用规则和模式规则 “%.c：%.y”（该规则运行Yacc）。无论该规则是否实际使用（如果碰巧存在文件“foo.y”，该规则将运行）。那么make试图重建“foo.c”的规则都是“%.c：%.y”所定义的规则，而不是万用规则。这样避免了make试图使用非最终的万用规则来重建文件“foo.c”的情况发生。

2. 特殊的内嵌伪模式规则用来指定一些特定的文件，避免在处理这些这种文件名的文件时使用非最终万用规则。这些伪模式规则没有依赖和命令行，用在其它的场合时被忽略。例如，内嵌的隐含规则：“%.p ：”保证Pascal源程序如“foo.p”匹配特定的目标格式，从而避免make寻找“foo.p.o”或“foo.p.c”。 在后缀规则中，为后缀列表中的每一个有效后缀创建了一个“%.p”形式的伪模式规则。可参考 9.7 后缀规则 一节。

9.5.6 重建内嵌隐含规则

对隐含规则，我们可以对它进行重建。重建一个隐含规则时，需要使用相同的目标和依赖文件，但是命令可以不同。这样就替代了有相同目标和依赖的那个make内嵌的规则，替代之后的隐含规则的顺序由它在Makefile中的位置决定。例如通常我们Makefile中可能会包含这样一个规则：

```
%.o : %.c
        $(CC) $(CFLAGS) –D__DEBUG__   $< -o $@
```

这个模式规则替代了编译.c文件的内嵌隐含规则。

也可以取消一个内嵌的隐含规则。同样需要定义一个和隐含规则具有相同的目标和依赖，没有命令行的模式规则。例如：如下的规则取消了make编译.s文件的内嵌规则。

```
%.o : %.s
```

9.6 缺省规则

有时需要定义一个缺省的规则，当在Makefile中没有重建的规则时（没有重建它的明确规则和可用的隐含规则）。就是用这个规则来重建它。就是说，当所需要的重建的目标文件没有可用的命令时、就执行这个缺省规则的命令来创建它。

定义这样一个规则，我们可以使用最终万用规则来实现。例如：在调试Makefile时（可能一些源文件还没有完成），我们所关心的是Makefile中规则的正确执行，而不关心源文件的内容。这样我们就可以使用没有内容的源文件来代替。可以在Makefile中定义这样一个规则：

```
%::
    touch $@
```

对没有找到的所有的.c文件使用“touch”命令创建一个空的文件。

实现一个缺省规则的方式也可以不使用万用规则来实现，可以使用伪目标“.DEFAULT”（可参考 3.9 Makefile的特殊目标 一节）实现。上边的例子也可以这样来书写：

```
.DEFAULT :
        touch $@
```

对于没有命令行的伪目标“.DEFAULT”，其含义是取消前边所有使用“.DEFAULT”指定的缺省执行命令。

同样，也可以让这个缺省的规则不执行任何命令（给它定义个一个空命令）。

另外缺省规则也可用来实现在一个Makefile中重载另一个makefile文件（参考 2.8 重载另外一个makefile 一节）。

9.7 后缀规则

后缀规则是一种老风格定义隐含规则的方式。新版本的make中使用模式规则取代了这种实现方式，模式规则和它相比更加清晰明了。现在版本保留它的原因只是为了能够兼容旧的makefile文件。后缀规则有两种类型：“双后缀”和“单后缀”。

双后缀规则定义一对后缀：目标文件的后缀和依赖目标的后缀。它匹配所有后缀为指定目标后缀的文件名。对于一个匹配的目标文件，它的依赖文件是这样形成：将目标文件中的后缀替换为依赖文件的后缀之后得到。如：一个描述目标和依赖后缀的“.o”和“.c”的规则就等价于模式规则“%o：%c”。

单后缀规则只定义一个后缀：此后缀是源文件的后缀。它可以匹配任何文件，其依赖文件名是这样形成：将源文件后缀追加到目标文件名之后得到。例如：单后缀“.c”就等价于模式规则“% :

%.c”。

后缀规则的识别过程：比较make本身所定义的后缀和规则中出现的后缀，如果规则的目标中只有一个可识别的后缀，则这个规则是一个“单后缀”规则；当规则的目标中有两个可识别的后缀时，这个规则就是“双后缀”规则。

例如：“.c”和“.o”都是make可识别的后缀。因此当定义了一个目标是“.c.o”的规则时。make会将它作为一个双后缀规则来处理，它的含义是目标为“.o”文件、依赖为“.c”文件。下边是使用后追规则定义的、编译.c源文件的规则：

```
.c.o:
        $(CC) -c $(CFLAGS) $(CPPFLAGS) -o $@ $<
```

一个后缀规则依赖关系描述中不能包含依赖文件。否则，此规则将被作为一个普通规则对待。因此规则：

```
.c.o: foo.h
        $(CC) -c $(CFLAGS) $(CPPFLAGS) -o $@ $<
```

它不是一个后缀规则。它是目标文件为“.c.o”、依赖文件是“foo.h”的普通规则。它不等价于规则：

```
%.o: %.c foo.h
        $(CC) -c $(CFLAGS) $(CPPFLAGS) -o $@ $<
```

需要注意的是：没有命令行的后缀规则是没有任何意义的。它和没有命令行的模式规则不同（参考 9.5.6 重建内嵌隐含规则 一节），它并不能取消之前使用后追规则定义的规则。它所实现的仅仅是将这个后缀规则作为目标加入到make的数据库中。

可识别的后缀指的是特殊目标“.SUFFIXES”的所有依赖的名字。通过给特殊目标“SUFFIXES”添加依赖来增加一个可被识别的后缀：

```
.SUFFIXES: .hack .win
```

它所实现的功能是把后缀“.hack”和“.win”加入到可识别后缀列表的末尾。

如果需要重设默认所有的可识别后缀，可以这样来实现：

```
.SUFFIXES:              #删除所有已定义的可识别后缀
.SUFFIXES: .c .o .h     #重新定义
```

其过程是首先使用没有依赖的特殊目标“.SUFFIXES”来删除所有已定义的可识别后缀；之后再重新定义。

注意：make的“-r”或“-no-builtin-rules”可以清空所有已定义的可识别后缀。

在make读取所有的makefile文件之前，变量“SUFFIXE”被定义为默认的可识别后缀列表。

虽然存在这样一个变量，但是请不要通过修改这个变量值的方式来改变可识别的后缀列表，应该使用特殊目标“.SUFFIXES”来实现。

9.8 隐含规则搜索算法

对于目标“T”，make为它搜索隐含规则的算法如下。此算法适合于：1．任何没有命令行的双冒号规则；2．任何没有命令行的普通规则；3．那些不是任何规则的目标、但它是另外某些规则的依赖文件；4．在递归搜索过程中，隐含规则链中前一个规则的依赖文件。

在搜索过程中没有提到后缀规则，因为所有的后缀规则在make读取Makefile时，都被转换为对应的模式规则。

对于形式为“ARCHIVE(MEMBER)”的目标，下边的算法会执行两次，第一次的目标是整个目标名“T”（“ARCHIVE(MEMBER)”），如果搜索失败，进行第二次搜索，第二次以“member”作为目标来搜索。

搜索过程如下：

1. 将目标“T”的目录部分分离，分离后目录部分称为“D”，其它部分称“N”。例如：“T”为“src/foo.o”时，D就是“src/”，“N”就为“foo.o”。
2. 列出所有和“T”或者“N”匹配的模式规则。如果模式规则的目标中包含斜杠，则认为和“T”相匹配，否则认为此模式规则和“N”相匹配。
3. 只要这个模式规则列表中包含一个非万用规则的规则，那么将列表中所有的非最终万用规则删除。
4. 删除这个模式规则列表中的所有没有命令行的规则。
5. 对于这个模式规则列表中的所有规则：
   a) 计算出模式规则的“茎”S，S应该是“T”或“N”中匹配“%”的非空的部分。
   b) 计算依赖文件。把依赖中的“%”用“S”替换。如果目标模式中不包含斜杠，则把“D”加在替换之后的每一个依赖文件开头，构成完整的依赖文件名。可参考 9.5.4 模式的匹配 一小节
   c) 测试规则的所有依赖文件是否存在或是应该存在（一个文件，如果在Makefile中它作为一个明确规则的目标，或者依赖文件被提及，我们就说这个文件是一个“应该存在”的文件）。如果所有的依赖文件都存在、应该存在或是这个规则没有依赖。退出查找，使用该规则。
6. 截止到第5步，合适的规则还是没有被找到，进一步搜索。对于这个模式规则列表中的每一规则：
   a) 如果规则是一个终止规则，则忽略它，继续下一条模式规则。
   b) 计算依赖文件（同第5步）。

c) 测试此规则的依赖文件是否存在或是应该存在。

d) 对于此规则中不存在的依赖文件，递归的调用这个算法查找它是否可由隐含规则来创建。

e) 如果所有的依赖文件存在、应该存在、或是它可以由一个隐含规则来创建。退出查找，使用该规则。

7. 如果没有隐含规则可以创建这个规则的依赖文件，则执行特殊目标“.DEFAULT”所指定的命令（可以创建这个文件，或者给出一个错误提示）。如果在Makefile中没有定义特殊目标“DEFAULT”，就没有可用的命令来完成“T”的创建。make退出。

一旦为一类目标查找到合适的模式规则。除匹配“T”或者“N”的模式以外，对其它模式规则中的目标模式进行配置，使用“茎”S替换其中的模式字符“%”，将得到的文件名保存直到执行命令更新这个目标文件（“T”）。在命令执行以后，把每一个储存的文件名放入数据库，并且标志为已更新，其时间戳和文件“T”相同。

在执行更新文件“T”的命令时，使用自动化变量表示规则中的依赖文件和目标文件。

第十章：使用make更新静态库文件

10 更新静态库文件

静态库文件也称为“文档文件”，它是一些.o 文件的集合。在 Linux（Unix）中使用工具“ar”对它进行维护管理。它所包含的成员（member）就是若干.o 文件。

10.1 库成员作为目标

一个静态库通常由多个成员（.o 文件）组成。静态库文件的成员可独立被作为一个规则的目标，将库成员作为目标时需要按照如下的格式来书写：

ARCHIVE(MEMBER)

注意，这种书写方式只能出现在规则的目标和依赖文件中，不能出现在规则的命令行中！因为，绝大多数命令不能支持这种表示的语法，它们不能直接对库的成员进行操作。这种表达式在规则的目标或者依赖中，它表示库“ARCHIVE”的成员“MEMBER”。含有这种表达方式规则的命令行只能是“ar”命令或者其它可以对库成员进行操作的命令。例如：下边规则创建了库“foolib”，并将“hack.o”成员纳入到库：

```
foolib(hack.o) : hack.o
        ar cr foolib hack.o
```

实际上，这个规则实现对库的所有成员的更新，其过程使用了隐含规则来创建.o 文件。另外需要注意工具“ar”的使用。

如果规则中需要同时指定一个库中的多个成员，可以将多个成员罗列在括号内，下边那样：

foolib(hack.o kludge.o)

它就等价于：

foolib(hack.o) foolib(kludge.o)

在描述的多个成员时也可以使用shell的通配符（参考 3.4 文件名使用通配符 一节）。例如：“foolib(*.o)”，它代表库文件“foolib”的所有.o成员。

10.2 静态库的更新

上一节已经讲述了规则中形如“A(M)”目标的含义，它表示规则的目标是静态库“A”的成员“M”。在 make 为这样一个目标搜索隐含规则时，对于满足“(M)”的隐含规则，同样也被认为是满足“A(M)”这个目标的隐含规则。

这样就出现了一个特殊的模式规则，它的目标模式是“(%)”。这个特殊的规则用来更新目标“A(M)”，规则将文件“M”拷贝到库“A”中，如果静态库文件“A”不存在，则会创建这个库文件。例如：如果一个目标为“foo.a(bar.o)”，那么规则将完成：首先使用隐含规则生成“bar.o”文件，之后将“bar.o”加入到库“foo.a”中。那么“bar.o”就作为库文件“foo.a”的一个成员（当然如果“foo.a”不存在，就会被创建件。此特殊规则的命令行一般都是“ar”命令）。

这个特殊的规则可以和其它的隐含规则一起构成一个隐含规则链。因此我们可以直接在命令行中执行“make ‘foo.a(bar.o)’”（注意“foo.a(bar.o)”作为命令行选项，需要使用引号，否则 shell 会将“(”作为特殊字符对待），只要当前目录下存在“bar.c”这个文件，就会执行如下命令：

```
cc -c bar.c -o bar.o
ar r foo.a bar.o
rm -f bar.o
```

我们注意到了“bar.o”时被作为中间过程文件处理的。参考 9.4 make的隐含规则链 一节。需要说明的是，包含上述命令的规则，在规则的命令行中使用自动化变量“$%”来代替“bar.o”。参考 9.5.3 自动化变量 一小节

一个静态库（文档文件）中，其所有的成员名是不包含目录描述的。就是说对于静态库，当使用“nm”命令查看其成员时，能够获得的信息只是静态库所包含的成员（一系列.o 文件，文件名中并没有包含目录）。不能得知静态库在创建时它的每一个成员所在的目录情况。但我们在 Makefile 中，采用“A(M)”格式的目标，书写建立（重建或者更新）静态库的规则时，可以指定它的成员来

源于那个目录。就是说我们可以对它的成员使用完整路径的文件名进行说明。Make 在处理这样一个目标“foo.a(dir/file.o)”时，可能会执行如下的命令：

```
ar r foo.a dir/file.o
```

和上边的例子类似，它是将指定目录下的.o 文件纳入到库文件中，作为静态库的一个成员。Makefile 中出现类似的目标时，隐含规则的命令行中可能就需要使用自动化变量“%D”和“%F”。

10.2.1 更新静态库的符号索引表

本小节的内容相对简单。前边提到过，静态库文件可以使用工具“ar”来创建、维护。当需要给静态库增建一个成员时（加入一个.o 文件到静态库中），“ar”可以直接需要增加的.o 文件拷贝到静态库中。这样直接拷贝可能存在一个问题。在使用这个库进行连接生成可执行文件时。“ld”却提示了错误，可能是：主程序使用了之前加入到库中的.o 文件中定义的一个函数（全局变量），但连接程序却告诉你无法找到这个函数（或者变量）。

这个问题的原因就是：之前我们将编译完成的.o 文件加入到库中，但是并没有更新库的符号表。连接程序进行连接时，在静态库的符号索引表中无法定位刚才加入的.o 文件中定义的函数或者变量。如果需要让加入的所有.o 文件中定义的函数（变量）有效，需要使用另外一个工具“ranlib”来对静态库的符号索引表进行更新。

我们所使用到的静态库（文档文件）中，存在这样一个特殊的成员，它的名字是“__.SYMDEF”。它包含了静态库中所有成员所定义的有效符号（函数名、变量名）。因此，当为库增加了一个成员时，相应的就需要更新成员“__.SYMDEF”，否则所增加的成员中定义的所有的符号将无法被连接程序定位。完成更新的命令是：

```
ranlib ARCHIVEFILE
```

通常在 Makefile 中我们可以这样来实现：

```
libfoo.a: libfoo.a(x.o) libfoo.a(y.o) ...
        ranlib libfoo.a
```

它所实现的是：更新静态库成员“x.o”和“y.o”之后，同时对静态库的成员“__.SYMDEF”进行更新（更新库的符号索引表）。

如果我们使用 GNU ar 工具来维护、管理静态库，我们就不需要考虑这一步。GNU ar 本身已经提供了在创建库时同时更新符号索引表的功能（这是默认行为，我们也可以通过某些命令行选项来控制。大家可以参考 GNU ar 工具的 man 手册）。

10.3 make静态库的注意事项

在make静态库（文档文件）时，特别需要注意的一点是：make的并行执行（执行make时使用“-j”选项）可能会给更新静态库所有影响。因为，在同一时刻，当多个“ar”命令同时维护相同的静态库时，会造成静态库的损坏，甚至导致此静态库不可用。

可能在后期的 make 版本中，会提供一种在并行执行时防止同时多个“ar”命令对同一静态库的操作的机制。但是就目前的版本来说，这个问题是存在的。因此要求我们在书写 Makefile 时，加入控制策略，来避免 make 并发执行时多个“ar”命令同时操作同一个静态库文件。或者不使用 make 并发执行功能。

10.4 静态库的后缀规则

静态库的后缀规则属于后缀规则的特殊应用，后缀规则在目前版本的 GNU make 中已经使用模式规则替代。但目前版本 make 同样支持旧风格的后缀规则，主要是为了兼容老版本的 make。对于静态库也可以使用它的后缀规则。目标后缀需要是“.a”（在 Linux（Unix）中、静态库的后缀为.a）。例如这样一个后缀规则：

```
.c.a:
        $(CC) $(CFLAGS) $(CPPFLAGS) -c $< -o $*.o
        $(AR) r $@ $*.o
        $(RM) $*.o
```

它相当于模式规则：

```
(%.o): %.c
        $(CC) $(CFLAGS) $(CPPFLAGS) -c $< -o $*.o
        $(AR) r $@ $*.o
        $(RM) $*.o
```

对于老风格的后缀规则，它的目标是“.a.c”，当转换为模式规则时。后缀规则中的“.a”被作为模式规则的目标文件的后缀（目标文件就是 N.a）；“.c”被作为了模式规则依赖文件的后缀（依赖文件就是 N.c）。

存在这样一种情况：可能一个不是静态库的文件使用了后缀.a。那么make也会按照正常的方式静态库的后缀规则转换成为一个模式规则。因此一个双后缀规则的目标“.*x*.a”，make转换时将会产生两个模式规则：“(%.o): %.*x*”和“%.a: %.*x*”。具体的转换过程可参考 9.7 后缀规则 一节

第十一章 ： GNU make的特点

11 GNU make的一些特点

截至本章为止，所有关于 GNU make 使用的讨论全部结束。相信大家也能够独立、熟练的书写 Makefile，并能够使用 GNU 的 make 来管理自己的工程。

本章是 GNU make 特点的总结。主要是和其它版本 make 得比较。这些特征都是以 4.2 BSD 中的 make 的为基准。当需要书写可移植到不同的类 UNIX 系统上的 Makefile 时，应避免使用 GNU 版本 make 自身的一些特征。

11.1 源自System v的特点

下面所罗列的这些是来自 System V 版本 make 的一些特点：

- 变量“VPATH”及它的含义（参考 3.5 目录搜索 一节）。System V版本的make支持，但没有得到验证。4.3 BSD 的make支持（据说是对System V的make这一功能的仿照）。
- 可包含其它makefile文件（参考 2.3 包含其它makefile文件 一节）。使用指示符“include”可同时包含多个文件是GNU的扩展。
- 可使用系统环境变量。参考 5.9 系统环境变量 一节
- 变量“MAKEFLAGS”。用于make递归调用时传递命令行选项。参考 4.6 make的递归调用 一节
- make静态库（文档文件）是“$%”可代表静态库成员名。参考 9.5.3 自动化变量 一节
- 对自动变量“$@”、“$*”、“$<”、“$%”和“$?”进行了扩展，支持“$(@F)”和“$(@D)”等形式的自动环变量。并以此对自动变量“$^”进行了扩展。参考 9.5.3 自动化变量 一节
- 变量引用。参考 5.1 变量的引用 一节
- 支持使用命令行选项“-b”和“-m”来兼容其它版本的 make。System V 的 make 中，这些选项有实际含义。
- 使用变量“MAKE”执行的递归调用。可支持“-n”、“-q”和“-t”向子make进程的传递。参考 4.6 make的递归调用 一节
- 支持后缀“.a”（参考 10.4 静态库的后缀规则 一节）。不过这个特点在GNU make的新版本中已经被模式规则所取代。
- 保持 Makefile 规则命令行的书写格式，只是去掉了初始空字符。执行命令的回显保持 Makefile 中的书写格式不变。

11.2 源自其他版本的特点

下面的特点来自于其它版本的 make，但每一个特征来自由哪个版本的 make 不太清楚：

- 模式规则使用模式字符“%”。目前在多个不同版本的make中都有使用“%”。但具体是那个版本的make提出它不甚清楚。关于模式规则参考 9.5 模式规则 一节
- 规则链以及隐含的中间过程文件。这个特点首次是在Stu Feldman 的make版本中实现，并用于AT&T第八版的Unix研究中。后来AT&T贝尔实验室的Andrew Hume 在它的mk程序中应用（这里称为“传递闭合”）。但不清楚这个特征是对它们的继承还是GNU自己的重新实现。参考 9.4 隐含规则链 一节
- 包含规则所有依赖文件列表的自动化变量“$^ ”。可以确定这不是GNU创造的，但具体是哪个版本的make创造也不清楚。参考 9.5.3 自动化变量 一节
- 命令行的“what if”选项（GNU make的“-W”选项）据说是Andrew Hume 在mk中首次提出的的。参考 8.7 make的命令行选项 一节
- 并发执行的观点，在其它多种版本的make中都有支持。但System V 和BSD 没有实现此功能。参考 4.3 并发执行命令 一节
- 变量的模式替换引用来自SunOS 4。参考 5.3 变量的高级用法 一节。GNU make中，这个功能是在SunOS 4 实现之前由函数“patsubst”提供。同一功能的两种实现，在两个版本的make中，难以确定是哪一个最早提出这个概念。
- 命令行之前使用“+”字符，它有特殊的含义（参考 8.3 替代命令的执行 一节）。这种做法是由IEEE Standard 1003.2-1992 (POSIX.2)定义的。
- 变量值追加“+=”的语法来自于SunOS 4 版本的make。参考 5.6 追加变量值 一节
- 静态库成员列表作为目标的语法“ARCHIVE(MEM1 MEM2...)”源自SunOS 4 make。参考 10.1 库成员作为目标 一节
- 使用“-include”包括多个其它的 makefile 文件，当所包含的文件不存在时不出错。这个特征源自 SunOS 4 版本的 make。（但 SunOS 4 版本的 make 不能使用单指示符同时包含多个文件）。GNU make 的这个特征和 SGI make 的指示符“sinclude”相同。GNU make 也支持“sinclude”。

11.3 GNU make自身的特点

以下特点是由 GNU make 本身的：

- 命令行选项“-v”或“--version”打印 make 的版本和版权信息。
- 使用“-h”或“--help”列出 make 支持的所有命令行选项。
- 直接展开式变量。参考 5.2 两种变量定义 一节
- 变量“MAKE”支持make递归调用时命令行选项的传递。参考 4.6 make的递归调用 一节

- 命令选项“-C”或“--directory”改变make执行的工作目录。参考 8.7 make的命令行选项 一节
- 支持多行变量的定义。参考 5.8 定义多行变量 一节
- 使用特殊目标“.PHONY”声明伪目标。AT&T 贝尔实验室Andrew Hume 使用不同的语法在它的mk程序中也实现了该功能。两者几乎在同时支持。参考 3.6 Makefile伪目标 一节
- 支持过个文本处理函数。参考 7.2 文本处理函数 一节
- 支持使用“-o”或者“--old-file”选项指定一个文件是未修改文件（告诉make不需要考虑这个文件的时间戳）。参考 8.4 防止特定文件重建 一节
- 条件执行。众多其它版本的make也支持；它似乎是对c语言预处理程序和宏语言的自然扩展，不能算是一个全新的概念。参考 6 Makefile的条件判断 一章
- 指定包含makefile文件的搜寻路径。参考 2.3 包含其它makefile文件 一节
- 环境变量“MAKEFILES”指定需要默认读取的makefile文件。参考 2.4 变量 MAKEFILES 一节
- 去除文件名中的“./”，因此“./file”和“file”等价。
- 支持在规则依赖中使用“-lNAME”来指定连接库。参考 3.5 目录搜索 一节
- 允许后缀规则中后缀名可以是任何字符串（参考 9.7 后缀规则 一节）。其它版本make中后缀必须以“.”开始，并且后缀中不能包含斜杠“/”。
- 内嵌变量“MAKELEVEL”，这个变量被用来跟踪make递归调用过程调用深度。 参考 4.6 make的递归执行 一节
- 内嵌变量“MAKECMDGOALS”，这个变量代表了make执行的终极目标。8.2 指定终极目标 一节
- 指定静态模式规则。参考 3.12 静态模式 一节
- 支持选择性搜索关键字“vpath”。参考 3.5 目录搜索 一节
- 支持计算的变量引用。参考 5.3 变量的高级用法 一节
- 支持自动重建makefile文件（参考 2.7 makefile文件的重建 一节）。
- 多个新的内嵌隐含规则。参考 第 9 章 使用隐含规则
- 内嵌变量“MAKE_VERSION”代表当前的 make 版本。

第十二章 和其它版本的兼容

12 不兼容性

GNU make 存在一些和其它版本 make 不兼容的功能，对于其它版本 make 的部分功能，GNU

make 也没有实现。POSIX.2 标准 （IEEE Standard 1003.2-1992）没有规定以下的这些特点需要在 make 中实现。

- 形如“FILE((ENTRY))”的目标代表静态库文件“FILE”的一个成员。它的成员不是用文件名，而是一个定义了连接符号“ENTRY”的.o文件。GNU make没有支持它的原因是。这样的规则需要对静态库内部符号索引表需要熟悉。GNU make对静态库成员的更新可参考 10 更新静态库文件 一章
- 以字符“~” 结尾的后缀在System V make的后缀规则中有特别的含义；它指的是对应目标文件的依赖文件是没有“~”的SCCS 文件。例如，System V的后缀规则“.c~.o”含义是：“N.o”是从SCCS文件“s.N.c”中提取的。为了完全覆盖，可能需要一系列的这样的规则。GNU make对它的处理是：使用两个模式规则从SCCS文件抽取一个文件，这两个模式规则形成一个隐含规则链。可参考 9.4 隐含规则链 一节
- 在System V 和 4.3 BSD make中，通过“VPATH”指定的目录搜寻（参考 3.5 目录搜索 一节）的文件，对应的文件名需要经过make改变后才加入对应规则到命令行中。GNU make使用自动环变量来实现这一功能。
- 在一些Unix的make中，自动化变量“$*”（参考 9.5.3 自动化变量 一节）作为规则的依赖时，具有奇怪的特征：它会被扩展为该规则的目标全名。它和GNU make中“$*”的含义完全不同。
- 在一些Unix的make中。会为所有目标搜索隐含规则（参考 9 使用隐含规则 一章），不仅仅是指对那些没有命令的目标。就是说如果我们的规则时这样的话：

  foo.o:
  cc -c foo.c

  这种版本的 make 就会认为目标“foo.o”的依赖文件是“foo.c”。我们认为 make 的这种处理方法容易导致混乱。因为 Makefile 已经对目标有明确的定义，而在为目标搜索隐含规则是不合逻辑的。因此 GNU make 没有支持这种方式的规则处理。
- GNU make 不包含任何编译以及预处理 EFL 程序的内嵌隐含规则。如果其它那种版本的 make 已经实现了这个特性，我们会很乐意地把它加入到 GNU make 支持的特性中。
- 在 SVR4 版本的make中，一个没有命令的后缀规则被作为一个空命令规则来处理（参考 4.8 空命令 一节）。例如：

  .c.a:

  这个规则将覆盖内嵌的目标为“.c.a”的规则。

  GNU make 对这个规则的行为是为目标.a 添加一个依赖类型.c。上述的行为在 GNU make 中的实现是：

  .c.a: ;
- 一些版本的make 在调用shell时使用“-e”参数告诉shell，在除执行“make -k”（参考 8.6 使用make进行编译测试 一节）时，一旦命令行执行失败（返回状态非 0）就立即退出。我们

认为对于不同的命令的执行结果要视情况来处理。因此GNU make的没有支持这种方式。

第十三章 Makefile的约定

13 书写约定

本章主要讲述，在为 GNU make 书写 Makefile 时需要遵循的约定。使用工具“Automake”将会帮助我们创建一个遵循这些约定的 Makefile。所有 GNU 发布的软件包中的 Makefile 都是按照这些标准的约定来书写的。因此理解本章的内容，可帮助我们能够很快的熟悉那些开源代码的结构。对于我们，在为一个工程书写 Makefile 时，也尽量遵循这些约定。虽然并没有强求你这么做，但是建议你还是按照已约定的规则来书写自己的 Makefile。

13.1 基本的约定

本节讨论的是书写 Makefile 中需要的一些基本约定，由于不同版本 make 之间的一些差异。可能在 GNU make 环境中正常工作的 Makefile，换成其它版本的 make 执行时会发生错误。为了最大可能的兼容不同版本的 make，这里给出了一些基本的约定。

1. 所有的 Makefile 中应该包含这样一行：

   SHELL = /bin/sh

   这样做的目的是为了避免变量“SHELL”在有些系统上可能继承同名的系统环境变量而导致错误。虽然在 GNU 版本的 make 中不会出现这种问题（GNU make 中变量“SHELL”的默认值是“/bin/sh”，它不同于系统环境变量“SHELL”）。

2. 小心处理后缀和隐含规则。不同 make 的可识别后缀和隐含规则存在不兼容，它可能会导致混乱或者错误。因此在特定 Makefile 中明确限定可识别的后缀是一个不错的注意。Makefile 中应该这样做：

   .SUFFIXES:
   .SUFFIXES: .c .o

   第一行首先取消掉 make 默认的可识别后缀列表，第二行通过特殊目标“.SUFFIXES”重新指定可识别的

   后缀规则。

3. 小心处理规则中的路径。当需要处理指定目录的的文件时，需要明确指定路径。如“./”代表当前目录，“$(srcdir)”代表源代码目录。当没有指定明确路径时，意味着是当前目录。

目录“./”（当前目录，GNU 的发布软件包中的“build”目录）和“$(srcdir)”的区别和重要，我们可以通过“configure”脚本的选项“--srcdir”指定源代码所在的目录（可参考 GNU 发布的软件包中的 configure 脚本）。当源代码目录和 build 目录不同时，规则：

```
foo.1 : foo.man sedscript
        sed –e sedscript foo.man > $@
```

将执行失败，是因为“foo.man”和“sedscript”并不在当前目录（当然，处理这种错误的手段可能有很多，诸如使用变量“VPATH”等）。当前目录只是 build 目录，并不是软件包目录。

4. 使用GNU make的变量“VPATH”指定搜索目录。当规则只有一个依赖文件时。我们应该使用自动化变量“$< ”和“$@”代替出现在命令的依赖文件和目标文件（其它版本的make，只在隐含规则中设置自动化变量“$< ”）。对于一个这样的目标：

```
foo.o : bar.c
        $(CC) –I. –I$(srcdir) $(CFLAGS) –c bar.o –o foo.o
```

我们在 Makefile 中应该是用这种方式来书写：

```
foo.o : bar.c
        $(CC) –I. –I$(srcdir) $(CFLAGS) –c $< –o $@
```

另外，对于有多个依赖的规则，为了规则能被正确执行。应该在规则的命令行中明确的指定文件的完整路径名。例如第一个例子就可以这样写（需要在规则之前使用“VPATH”指定搜索目录）：

```
foo.1 : foo.man sedscript
        sed –e $(srcdir)/sedscript $(srcdir)/foo.man > $@
```

在 GNU 的发布软件包中，包括了很多非源代码的文件。诸如：“info”文件、“Autoconf”的输出文件、“Automake”、“Bison”或者“Flex”等文件。这些文件在发布时就在源代码的目录中。因此 Makefile 中对它们的重建也应该是在源代码目录，而不应该在 build 目录。

相反的，对于那些本来就不存在于源代码目录下的文件，也不应该将它们创建在源代码的目录下。**要记住，make 的过程不应该以任何方式修改源代码，或者改变源代码目录的结构。**

13.2 规则命令行的约定

本节将讨论书写规则命令的一些约定，在书写多系统兼容的 Makefile 时，特别需要注意不同系统之间的命令的不兼容。这里对规则命令行做出了一些基本约定：

1. 书写 Makefile 时，规则的命令（包括其他的脚本文件，如：configure）应该是“sh”而不因该是“csh”所支持的。
2. 用于创建和安装的“configure”脚本和 Makefile 中的命令除下面所列出的意外，避免使用其它命令：

   cat cmp cp diff echo egrep expr false grep install-info
   ln ls mkdir mv pwd rm rmdir sed sleep sort tar test touch true

3. 在目标“dist”的命令行中可以使用压缩工具“gzip”。
4. 对于可使用的这些工具命令，尽量使用它的通用选项。不要使用那些只在特定系统上有效的选项。如：“mkdir -p”这个命令在 Linux 系统上能够很好的工作，但是其它很多系统却并不支持“mkdir”的“-p”选项。
5. 尽量不要在规则的命令行重创建符号连接文件（使用“ln”命令）。因为有些系统不支持（对于类 Unix 的系统我们基本上没有问题，可能这里所说的是 MS-DOS 系统的系统。我想大家也没有兴趣或者说没有必要在MS-DOS下写Makefile，所以这个限制基本可以不考虑）。
6. 重建或者安装目标（一般是伪目标）的命令行可使用编译器或者相关工具程序，应该使用一个变量来表示所要执行的命令。这样会比较方便，需要修改一个命令时，只需要更改变量的值就可以了。对于以下的这些命令程序：

   ar bison cc flex install ld ldconfig lex
   make makeinfo ranlib texi2dvi yacc

   在 Makefile 规则中的命令行中，使用以下这些变量来代替它们：

   $(AR) $(BISON) $(CC) $(FLEX) $(INSTALL) $(LD) $(LDCONFIG) $(LEX)
   $(MAKE) $(MAKEINFO) $(RANLIB) $(TEXI2DVI) $(YACC)

如果在规则的命令行需要使用“ranlib”或者“ldconfig”这些工具时，需要考虑它们是否在其它系统上被支持。当在不支持它的系统中执行包含此命令的规则时，要能够给出提示信息（提示原因是告诉用户系统不支持此命令，但不应该出现错误而退出执行）。

对另外一些命令组件的使用，应该都是通过变量的形式来实现。例如如下的这些命令：

chgrp chmod chown mknod

我们可以在 Makefile 中为这些命令组件定义一个代表它的变量（如：CHRP、CHMOD 等，在命令行中就可以使用$(CHMOD)来引用）。

书写多系统兼容 Makefile 时需要遵循以上的约定。如果只考虑一种系统时，以上的这些约定中有一部分可以灵活处理，比如：在命令组件的使用上，我们就可以使用这个系统独具的那些命令

组件；系统支持符号链接时，我们就可以在命令行重创建一个符号链接文件。**对于上边的第 6 个约定，强烈建议所有的 Makefile 都应该遵循。**

13.3 代表命令变量

Makefile 应该为重设所有的命令、选项等提供变量。就是说用户可以通过修改一个变量值来重新指定所要执行的命令，或者来控制命令执行的选项、参数等。

Makefile 中，所有的命令都应该使用变量定义。在规则中需要使用此命令时，通过对相应的变量的引用来实现命令的调用。例如：定义变量“CC = gcc”，规则中可使用“$(CC)”来引用“gcc”。

对于一些件管理器工具如“ln”，“rm”“mv”等，可以不为它们定义变量，而直接使用。

为所有命令执行的选项参数也应该定义一个变量（可称为命令的选项变量）。在命令变量（代表一个命令的变量）后添加“FLAGS”来命名这个选项变量。例如：变量“CFLAGS”是 c 编译器（命令变量为“CC”）的命令行选项变量；变量 YFLAGS 时命令“yacc”（命令变量为“YACC”）选项变量；变量“LDFLAGS”是命令“ld”（命令变量为“LD”）的选项变量等。规则中 c 预处理的命令使用变量“CCFLAGS”来替代；同样任何需要执行链接的命令行中使用“LDFLAGS”作为命令行选项。

c编译器的编译选项变量“CFLAGS”在Makefile中通常是为编译所有的源文件提供选项的变量。对一个特定文件增加的选项，不应包含在变量“CFLAGS”中。编译特定的文件（或者一类特定文件）时，如果需要使用特定的选项参数，可以将这些选项写在编译它所执行的规则的命令行中（也可以使用目标指定变量或者模式指定变量）。例如：

```
CFLAGS = -g
ALL_CFLAGS = -I $(CFLAGS)
.c.o:
        $(CC) -c $(CPPFLAGS) $(ALL_CFLAGS) $<
```

例子中，变量“CFLAGS”指定了编译选项为“-g”，本例中它作为缺省的选项。对于其它所有源文件的编译使用“-g”选项。双后缀规则的命令行中为编译生成“.o”文件指定了另外的选项“-I. -g ”

在所有编译命令行中，变量“CFLAGS”应该放在选项的最后。这样可以保证命令行选项参数在重复时，“CFLAGS”是有效的。在任何调用 c 编译器的命令行中应该使用选项变量“CFLAGS”，无论是进行编译还是连接。

如果需要使用 make 实现文件的安装，则在 Makefile 中需要定义变量“INSTALL”。此变量代表安装命令（install）。同时在 Makefile 中需要定义变量“INSTALL_PROGRAM”和“INSTALL_DATA”（“INSTALL_PROGRAM”的缺省值都是“$(INSTALL)”；“INSTALL_DATA”的缺省值是“${INSTALL} –m 644”）。可以是用这些变量，来安装可执行

程序或者非可执行程序到指定位置。例如：

```
$(INSTALL_PROGRAM) foo $(bindir)/foo
$(INSTALL_DATA) libfoo.a $(libdir)/libfoo.a
```

另外，也可以使用变量“DESTDIR”来指定目标需要安装的目录。通常在 Makefile 中不需要定义变量“DESTDIR”，可以通过命令行或者参数的形式指定。例如：“make DESTDIR= exec/install”。因此上边的命令就可以这样实现：

```
$(INSTALL_PROGRAM) foo $(DESTDIR)$(bindir)/foo
$(INSTALL_DATA) libfoo.a $(DESTDIR)$(libdir)/libfoo.a
```

安装命令中文件名作为作为第二个参数。每一个需要安装的文件使用单独的命令（包括安装一个目录）。

13.4 安装目录变量

在 Makefile 中，安装目录同样需要使用变量指定，这样就可以很方便的修改文件的安装路径。安装目录的标准命名下边将一一介绍。这些变量基于标准的文件系统结构，这些变量的变种在 SVR4、4.4BSD、Linux、Ultrix v4 以及其它现代操作系统中都有使用。

- 以下所罗列的两个变量是指定安装文件的根目录。所有其它安装目录都使它们的子目录。文件不能直接安装在这两个目录下。

  prefix

  这个前缀用于构造下列（除这两个安装根目录以外的其它目录变量）变量的缺省值。变量“prefix”缺省值是“/usr/local”。创建完整的GNU系统时，变量prefix的缺省值是空值，“/usr”是“/”的符号连接符文件。（如果使用“Autoconf”工具，它应该写成“@prefix@”）。注意：当更改了变量“PREFIX”以后执行“make install”，不会重建可执行程序（终极目标）。

  exec_prefix

  这个前缀用于构造下列变量的缺省值。变量“exec_prefix”缺省值是“$(prefix)”（如果使用“Autoconf”工具，它应该写为“@exec_prefix@”）。通常，“$(exec_prefix)”目录中的子目录中存放和机器相关文件（例如可执行文件和例程库）。“$(prefix)”目录的子目录存放通用的一般文件。同样：改变“exec_prefix”的值之后执行“make install”，不会重建可执行程序（终极目标）。

- 文件（包括可执行程序、说明文档等）的安装目录：

bindir

用于安装一般用户可运行的可执行程序。通常它的值为："/usr/local/bin"，使用时应写为："$(exec_prefix)/bin"。（使用"Autoconf"工具时，应该为"@bindir@"）

sbindir

安装可在shell中直接调用执行的程序。这些命令仅对系统管理员有用（系统管理工具）。通常它的值为："/usr/local/sbin"，要求在使用时应写为："$(exec_prefix)/sbin"。（使用"Autoconf"工具时，应该为"@sbindir@"）

libexecdir

用于安装那些通常不是由用户直接使用，而是由其它程序调用的可执行程序。通常它的值为："/usr/local/libexec"，要求在使用时应写为："$(exec_prefix)/libexec"。（使用"Autoconf"工具时，应该为"@libexecdir@"）

- 程序执行时使用的数据文件可从以下两个方面来分类：
  1. 是否可由程序更改。程序可更改或者不能更改的文件（虽然用户可编辑其中某些文件）。
  2. 是否和体系结构相关。和体系结构无关的文件，可被所有类型的机器共享；体系相关文件，仅可被相同类型机器、操作系统共享；其它的就是那些不能被任何两个机器共享的文件。

这样就存在六种不同的可能。除编译生成的目标文件（.o 文件）和库文件以外，不推荐使用那些和特定机器体系结构相关的文件，使用和体系无关的数据文件更加简洁，而且，它的实现也并不非常困难。

在 Makefile 中应该使用以下变量为不同类型的文件指定对应的安装目录：

datadir

用于安装和机器体系结构无关的只读数据文件。通常它的值为："/usr/local/share"，使用时应写为："$(prefix)/share"。（使用"Autoconf"工具时，应该为"@datadir@"）。"$(infodir)"和"$(includedir)"作为例外情况，参考后续对它们的详细描述。

sysconfdir

用于安装从属于特定机器的只读数据文件，包括：主机配置文件、邮件服务、网络配置文件、"/etc/passwd"文件等。所有该目录下的文件都应该是普通文本文件（可识别的"ASCII"码文本文件）。通常它的值为："/usr/local/etc"，在使用时应写为："$(prefix)/etc"。（使用"Autoconf"工具时，应该为"@sysconfdir@"）。

不要将可执行文件安装在这个目录下（可执行文件的安装目录应该是"$(libexecdir)"或者"$(sbindir)"）。也不要在这个目录下安装那些需要更改的文件（系统的配置文件等）。这些文件应该安装在目录"$(localstatedir)"下。

sharedstatedir

用于安装那些可由程序运行时修改的文件，这些文件与体系结构无关。通常它的值为："/usr/local/com"，要求在使用时应写为："$(prefix)/com"。（使用"Autoconf"工具时，应该为"@sharedstatedir@"）

localstatedir

用于安装那些可由程序运行时修改的文件，但这些文件和体系结构相关。用户没有必要通过直接修改这些文件来配置软件包，对于不同的配置文件，将它们放在"$(datadir)"或者"$(sysconfdir)"目录中。"$(localstatedir)"值通常为："/usr/local/var"，在使用时应写为："$(prefix)/var"。（使用"Autoconf"工具时，应该为"@localstatedir@"）

libdir

用于存放编译后的目标文件（.o）文件库文件（文档文件或者执行的共享库文件）。不要在此目录下安装可执行文件（可执行文件应该安装在目录"$(libexecdir)"下）。变量libdir值通常为："/usr/local/lib"，使用时应写为："$(exec_prefix)/lib"。（使用"Autoconf"工具时，应该为"@libdir@"）

infodir

用于安装软件包的 Info 文件。它的缺省值为："/usr/local/info"，使用时应写为："$(prefix)/info"。（使用"Autoconf"工具时，应该为"@infodir@"）

lispdir

用于安装软件包的 Emacs Lisp 文件的目录。它的缺省值为："/usr/local/share/emacs/site-lisp"，使用时应写为："$(prefix)/share/emacs/site-lisp"。当使用Autoconf工具时，应将写为"@lispdir@"。为了保证"@lispdir@"能够正常工作，需要在"configure.in"文件中包含如下部分：

```
lispdir='${datadir}/emacs/site-lisp'
AC_SUBST(lispdir)
```

includedir

用于安装用户程序源代码使用"#include"包含的头文件。它的缺省值为："/usr/local/include"，使用时应写为："$(prefix)/include"。（使用"Autoconf"工具时，应该为"@includedir@"）。

除GCC外的大多数编译器不会在目录"/usr/local/include"中搜寻头文件，因此这种方式只适用GCC编译器。这一点应该不是一个问题，因为很多情况下一些库需要GCC才能工作。对那些依靠其它编译器的库文件，需要将头文件安装在两个地方，一个由变量"includedir"指定，另一个由变量"oldincludedir"指定。

oldincludedir

它所指定的目录也同样用于安装头文件，这些头文件用于非GCC的编译器。它的缺省值为："/usr/include"。（使用"Autoconf"工具时，应该为"@oldincludedir@"）。

Makefile在安装头文件时，需要判断变量"oldincludedir"的值是否为空。如果是空值，就不使用它进行头文件的安装（一般是安装完成"/usr/local/include"下的头文件之后才安装此目录下的头文件）。

一个软件包的安装不能替换该目录下已经存在的头文件，除非是同一个软件包（重新使用相同的软件包在此目录下安装头文件）。例如，软件包"Foo"需要在"oldincludedir"指定的目录下安装一个头文件"foo.h"时，可安装的条件为：1. 目录"$(oldincludedir)"目录下不存在头文件"foo.h"；2. 已经存在头文件"foo.h"，存在的头文件"foo.h"是之前软件包"Foo"安装的。

检查头文件"foo.h"是否来自于软件包Foo，需要在头文件的注释中包含一个"magic"字符串，使用命令"grep"来在该文件中查找这个magic。

- Unix 风格的帮助文件需要安装在以下目录中：

mandir

安装该软件包的帮助文档（如果有）的顶层目录。它的缺省值为："/usr/local/man"，要求在使用时应写为："$(prefix)/man"。（使用"Autoconf"工具时，应该为"@@mandir@@"）

man1dir

用于安装帮助文档的第一节（man 1）。它的缺省值为："$(mandir)/man1"。

man2dir

用于安装帮助文档的第二节（man 2）。它的缺省值为："$(mandir)/man2"。

...

不要将 GNU 软件的原始文档作为帮助页。应该编写使用手册。帮助页仅仅是为了帮助用户在 Unix 上方便运行 GNU 软件，它是附属的运行程序。

manext

文件名扩展字，它是对安装手册的扩展。以点号（.）开始的十进制数。缺省值为："."。

man1ext

帮助文档的第一节（man 1）的文件名扩展字。

man2ext

帮助文档的第二节（man 2）的文件名扩展字。

...

当一个软件包的帮助手册有多个章节时，使用这些变量代替"manext"。（第一节

"man1ext"，第二节"man2ext"，第三节"man3ext"……）

➢ 而且下列这些变量也应该在 Makefile 中定义：

srcdir

此变量指定的目录是需要编译的源文件所在的目录。该变量的值在使用"configure"脚本对软件包进行配置时产生的。（使用"Autoconf"工具，应该书写为"srcdir = @srcdir@"）

例如：

```
# 安装的普通目录路径前缀。
# 注意：该目录在开始安装前必须存在
prefix = /usr/local
exec_prefix = $(prefix)
# 放置"gcc"命令使用的可执行程序
bindir = $(exec_prefix)/bin
# 编译器需要的目录
libexecdir = $(exec_prefix)/libexec
# 软件包的Info 文件所在目录
infodir = $(prefix)/info
```

在用户标准指定的目录下安装大量文件时，可以将这些文件分类安装在指定目录的多个子目录下。可以在 Makefile 中实现一个"install"伪目标来描述安装这些文件的命令（包括创建子目录，安装文件到对应的子目录中）。

在发布的软件包中，不能强制要求用户必须指定这些安装目录的变量。使用一套标准的安装目录变量来指定安装目录，当用户需要指定安装目录时，通过修改变量定义来指定具体的目录，在用户没有指定的情况下，使用默认的目录。

13.5 Makefile的标准目标名

所有 GNU 发布的软件包的 Makefile 中，必须包含以下这些目标：

all

此目标的动作是编译整个软件包。"all"应该为Makefile的终极目标。该目标的动作不重建任何文档文件（只编译所有的源代码，生成可执行程序）；Info文件应该作为发布文件的一部分，DVI文件只在明确指定的时候才应该被重建。

缺省情况下，对所有源程序的编译和连接应该使用选项"-g"，是最终的可执行程序中包含调试信息。当最终的可执行程序不需要包含调试信息时，可使用"strip"去掉可执行程序中的调试符号已缩减最终的程序大小。

install

此目标的动作是完成程序的编译并将最终的可执行程序、库文件等拷贝到安装的目录。如果只是验证这些程序是否可被正确安装，它的动作应该是一个测试安装动作。

安装时一般不要对可执行程序进行strip（去掉可执行程序内部的调试信息）。存在另外一个目标“install-strip”，它实现安装时strip。

尽可能保证目标“install”的动作不更改程序创建目录（builid目录）下的任何文件，对这个目录下文件的修改（重建或者更新）是目标“all”所要定义的动作。

“install”目标定义的动作在安装目录不存在时，能够创建这些不存在的安装目录。这些目录包括：变量“prefix”和“exec_prefix”指定的目录和所有必要的子目录。完成此任务的方式可以使用下边介绍的“installdirs”目标。

在安装man文档的命令前使用“-”忽略这安装命令的错误，这样可以避免在没有Unix man文档的系统上执行安装时出现错误。

安装Info文档的方法是使用变量“INSTALL_DATA”将Info文档拷贝到“$(infodir)”目录下去（参考 13.4安装目录的变量 一节），如果存在“install-info”命令则执行它。“install-info”是一个编辑Info“dir”文件的程序，更新或者修改“info”文档的入口和目录；它是Texinfo软件包的一部分。这里有一个安装Info文档的例子：

```
$(DESTDIR)$(infodir)/foo.info: foo.info
        $(POST_INSTALL)
  #  可能在“.”（当前目录）存在一个新的文档，而不是“srcdir”。
        -if test -f foo.info; then d=.; \
        else d=$(srcdir); fi; \
        $(INSTALL_DATA) $$d/foo.info $(DESTDIR)$@; \
  # 如果install-info命令存在则运行它
  #  使用“if”代替在命令行前的“-”
  #  这样，就可以看到运行install-info产生的真正错误
  #  我们使用“$(SHELL) -c”是因为在一些shell中
  #  遇到未知的命令不会运行失败
        if $(SHELL) -c 'install-info --version' \
                >/dev/null 2>&1; then \
```

```
install-info --dir-file=$(DESTDIR)$(infodir)/dir \
$(DESTDIR)$(infodir)/foo.info; \
else true; fi
```

目标install的命令需要分为三类：正常命令、预安装命令和安装后命令。参考 13.6 安装命令分类 一节

uninstall

删除所有已安装文件——由install创建的文件拷贝。规则所定义的命令不能修改编译目录下的文件，仅仅是删除安装目录下的文件。像install目标的命令一样，uninstall目标的命令也分为三类。参考 13.6 安装命令分类 一节

install-strip

和目标install的动作类似，但是install-strip指定的命令在安装时对可执行文件进行strip（去掉程序内部的调试信息）。它的定义如下：

```
install-strip:
    $(MAKE) INSTALL_PROGRAM='$(INSTALL_PROGRAM) -s' \
        install
```

如果软件包的存在安装脚本时，目标install-strip所定义的命令就不能是对目标“install”的引用，它所完成的仅仅是对可执行文件strip。

“install-strip”不应该直接在build目录下对可执行文件进行strip，应该是对安装目录下的可执行文件进行strip。就是说“install-strip”所的命令不能build目录下的文件产生影响。

一般不建议安装时对可执行文件进行strip，因为去掉可执行文件的调试信息后，在出现bug时，就不能对可执行程序进行跟踪调试。

clean

清除当前目录下编译生成的所有文件，这些文件由make程序创建。记住，不要删除软件包的配置文件，同时需要保留build时创建的那些文件（诸如：创建的目录、build生成的信息记录文件等）。因为这些文件都是发布版本的一部分。

对于.dvi文件，当它不作为发布版本的一部分时，可以删除。

distclean

删除当前目录下的的配置过程产生的文件、build过程产生的文件。目标“distclean”指定的删除命令应该删除软件包中所有非发布文件。

mostlyclean

类似于目标“clean”，但是可保留一些编译生成的文件，避免在下次编译时对这些文件重建。例如，用于GCC的目标“mostlyclean”不删除文件“libgcc.a”，因为在绝大多数情况下它都不需要重新编译。

maintainer-clean

此目标所定义的命令几乎会删除所有当前目录下能够由Makefile重建的文件。典型的，包括目标“distclean”删除的文件、由Bison生成的.c源文件、tags记录文件、Ifon文件等。但是有一个例外，就是执行“make maintainer-clean”不能删除“configure”这个配置脚本文件，即使“configure”可以由Makefile生成。因为“configure”是软件包的配置脚本。

目标“maintainer-clean”应该只能由维护软件包的用户使用，而不能被普通用户使用。因为它会删除一些软件包的发布文件，而重建这些文件可能需要专门的工具。因此我们在使用此目标是需要小心。

为了让用户能够在执行前得到提示，通常目标“maintainer-clean”的命令以下两行为开始：

@echo“该命令用于维护此软件包的用户使用”；

@echo“它删除的文件可能需要使用特殊的工具来重建。”

TAGS

此目标所定义的命令完成对该程序的tags记录文件的更新。

info

产生必要的Info文档。此目标应该按照如下书写：

```
info: foo.info

foo.info: foo.texi chap1.texi chap2.texi
$(MAKEINFO) $(srcdir)/foo.texi
```

必须在Makefile定义变量“MAKEINFO”，代表命令工具makeinfo，该工具是发布软件Texinfo的一部分。

通常，GNU的发布程序会和Info文档会被一同创建，这意味着Info文档是在源文件的目录下。用户在创建发布软件时，一般情况下，make不更新Info文档，因为它们已经更新到最新了。

dvi

为所有的Texinfo文件创建对应的DVI文件。例如：

```
dvi: foo.dvi

foo.dvi: foo.texi chap1.texi chap2.texi
        $(TEXI2DVI) $(srcdir)/foo.texi
```

必须在Makefile中定义变量“TEXI2DVI”。它代表命令工具texi2dvi，该工具是发布软件Texinfo一部分。

规则中也可以没有命令行，这样make程序会自动为它推导对应的命令。

dist

此目标指定的命令创建发布程序的tar文件。创建的tar文件应该是这个软件包的目录，文件名中也可以包含版本号（就是说创建的tar文件在解包之后应该是一个目录）。例如，发布的GCC 1.40版的tar文件解包的目录为“gcc-1.40”。

通常的做法是是创建一个空目录，如使用ln或cp将所需要的文件加入到这个目录中，之后对这个目录使用tar进行打包。打包之后的tar文件使用gzip压缩。例如，实际的GCC 1.40版的发布文件叫“gcc-1.40.tar.gz”。

目标“dist”的依赖文件为软件包中所有的非源代码的文件，因此在使用目标进行发布软件打包压缩之前必须保证这些文件是最新的。

check

此目标指定的命令完成所有的自检功能。在执行检查之前，应确保所有程序已经被创建，可以不安装。为了对它们进行测试，需要实现在程序没有安装的情况下被执行的规则命令。

的目标，对于各种程序它们很有用：

installcheck

执行安装检查。在执行安装检查之前，确保所有程序已经被创建并且被安装。需要注意的是：安装目录“$(bindir)”是否在搜索路径中。

installdirs

使用目标“installdirs”创建安装目录以及它的子目录在很多场合是非常有用的。脚本“mkinstalldirs”就是为了实现这个目的而编写的；发布的Texinfo软件包中就包含了这个脚本文件。Makefile中的规则可以这样书写：

```
# 确保所有安装目录（例如 $(bindir)）存在，如有必要则创建这些目录
installdirs: mkinstalldirs
            $(srcdir)/mkinstalldirs $(bindir) $(datadir) \
                                    $(libdir) $(infodir) \
```

```
                                        $(mandir)
```

或者可以使用变量“DESTDIR”：

```
# 确保所有安装目录（例如 $(bindir)）存在，如有必要则创建这些目录
installdirs: mkinstalldirs
            $(srcdir)/mkinstalldirs \
$(DESTDIR)$(bindir)   $(DESTDIR)$(datadir) \
                $(DESTDIR)$(libdir)   $(DESTDIR)$(infodir) \
                $(DESTDIR)$(mandir)
```

该规则不能更改软件的编译目录，仅仅是创建程序的安装目录。

13.6 安装命令分类

在为 Makefile 书写“install”目标时，需要将其命令分为三类：正常命令、安装前命令和安装后命令。

正常命令是把文件移动到合适的地方，并设置它们的模式。这个过程不修改任何文件，仅仅是把需要安装的文件从软件包中拷贝到安装目录。

安装前命令和安装后命令可能修改某些文件；通常，修改一些配置文件和系统的数据库文件。典型地，安装前命令在正常命令之前执行，安装后命令在正常命令执行后执行。

大多数情况是，安装后命令是运行“install-info”程序。它所完成的工作不能由正常命令完成，因为它更新了一个文件（Info 的目录），该文件不能单独的或者完整的从软件包来进行安装，因为它只能在正常安装命令完成安装软件包的 info 文档之后才可正确执行。

大多数程序不需要安装前命令，但应该在 Makefile 中提供。

将“install”规则的命令分为这三类时，应该在命令之间插入分类行（category lines）。分类行说明了后续需要执行的命令的类别。

分类行是由一个[Tab]字符开始的 make 的特殊变量的引用，行尾是可选的注释内容。可以使用三个特殊的变量，每一个代表一种类别。分类行不能出现在普通的执行文件中，因为这三个特殊的 make 变量没有定义（也不应该在 Makefile 中定义它们）。

以下是三种可能的分类行，并对它们进行了注释：

```
$(PRE_INSTALL)       # 以下是安装前命令
$(POST_INSTALL)      # 以下是安装后命令
$(NORMAL_INSTALL)    # 以下是正常命令
```

如果安装规则（install 所在的规则）的命令行没有使用分类行，那么在第一个出现的分类行之前的所有的命令行都被认为是正常命令。如果命令行中没有分类行，那么规则的所有命令行都都被认为是正常命令行。

对应的，一下这三个是“uninstall”命令的分类行：

```
$(PRE_UNINSTALL)        #以下是卸载前命令
$(POST_UNINSTALL)       #以下是卸载后命令
$(NORMAL_UNINSTALL)     #以下是正常命令
```

卸载前命令应该是取消软件包 Info 文档的入口。

如果目标 install 或 uninstall 存在依赖，其作为安装的子例程，那么就应该在每一个以来目标的命令行开始使用分类行，同时目标 insall 和 uninstall 的命令行开始也需要使用分类行。这样就可以确保任何调用方式时每一条命令都被正确的分类。

除下列命令外，安装前命令和安装后命令不应该使用其它命令：

```
[ basename bash cat chgrp chmod chown cmp cp dd diff echo
egrep expand expr false fgrep find getopt grep gunzip gzip
hostname install install-info kill ldconfig ln ls md5sum mkdir
mkfifo mknod mv printenv pwd rm rmdir sed sort tee test
touch true uname xargs yes
```

按照这种方式分类命令的原因是为了创建二进制的软件包。典型的二进制软件包包括可执行文件、必须安装的其它文件以及它自己的安装文件，因此二进制软件包就不需要任何正常命令、只需要安装前命令和安装后命令。

创建二进制软件包的程序通过提取安装前命令和安装后命令工作。这里有一个抽取安装前命令的方法：

```
make -n install -o all \
    PRE_INSTALL=pre-install \
    POST_INSTALL=post-install \
    NORMAL_INSTALL=normal-install \
    | gawk -f pre-install.awk
```

文件“pre-install.awk”可能包括以下内容：

```
$0 ~ /^\t[ \t]*(normal_install|post_install)[ \t]*$/ {on = 0}
on {print $0}
$0 ~ /^\t[ \t]*pre_install[ \t]*$/ {on = 1}
```

安装前命令的执行结果和软件包的安装 shell 脚本的结果一样。

第十四章 make的常见错误信息

14 make产生的错误信息

本章是对 make 执行过程中可能出现常见错误进行汇总、分析，并给出修正的可能方法。

make 执行过程中产生的错误并不是所有都是致命的；特别在命令行之前使用“-”、或者使用 make 的“-k”选项。make 执行过程的致命错误都带有前缀字符串“* * *”。

错误的信息都有前缀，一种是执行程序名作为错误的前缀（通常是“make”）；另外一种是当 Makefile 本身存在错误时，前缀包含了文件名和出现错误的行号。

在下述的错误列表中，省略了普通的前缀：

[FOO] Error NN

[FOO] signal description

这类错误并不是make的真正的错误。它表示make所调用的程序返回非零的错误码（Error NN），这种情况make会将其作为失败处理、或者以非正常方式退出（携带某种信号），参考 4.4 命令的错误 一节。

如果错误信息中没有附加“* * *”字符串，则是子过程的调用失败，如果Makefile中此命令有前缀“-”，make会忽略这个错误。

missing separator. Stop.

missing separator (did you mean TAB instead of 8 spaces?). Stop.

不可识别的命令行，make在读取Makefile过程中不能解析其中包含的内容。GNU make 在读取Makefile时根据各种分隔符（:，=，[TAB]字符等）识别Makefile的每一行内容。这些错误意味着make不能发现一个合法的分隔符。

出现这些错误信息的可能的原因是（或许是编辑器，绝大部分是ms-windows的编辑器）在Makefile中的命令之前使用了4个（或者8个）空格代替了[Tab]字符。这种情况，将产生上述的第二种形式产生错误信息。且记，所有的命令行都应该是以[Tab]字符开始的。参考 4 规则中的命令书写 一章

commands commence before first target. Stop.

missing rule before commands. Stop.

Makefile可能是以命令行开始：以[Tab]字符开始，但不是一个合法的命令行（例如，一个变量的赋值）。命令行必须和规则一一对应。

产生第二种的错误的原因可能是一行的第一个非空字符为分号，make会认为此处遗漏了规

则的“target: prerequisite”部分。参考 3.2 规则的语法 一节

No rule to make target `XXX'.

No rule to make target ` XXX ', needed by `yyy'.

无法为重建目标“XXX”找到合适的规则，包括明确规则和隐含规则。

修正这个错误的方法是：在Makefile中添加一个重建目标的规则。其它可能导致这些错误的原因是Makefile中文件名拼写错误，或者破坏了源文件树（一个文件不能被重建，可能是由于依赖文件的问题）。

No targets specified and no makefile found. Stop.

No targets. Stop.

第一个错误表示在命令行中没有指定需要重建的目标，并且make不能读入任何makefile文件。第二个错误表示能够找到makefile文件，但没有终极目标或者没有在命令行中指出需要重建的目标。这种情况下，make什么也不做。参考 第 8 章 执行make

Makefile `XXX' was not found.

Included makefile `XXX' was not found.

没有使用“-f”指定makefile文件，make不能在当前目录下找到默认Makefile（makefile或者GNUmakefile）。使用“-f”指定文件，但不能读取这个指定的makefile文件。

warning: overriding commands for target `XXX'

warning: ignoring old commands for target `XXX'

对同一目标“XXX”存在一个以上的重建命令。GNU make规定：当同一个文件作为多个规则的目标时，只能有一个规则定义重建它的命令（双冒号规则除外）。如果为一个目标多次指定了相同或者不同的命令，就会产生第一个告警；第二个告警信息说新指定的命令覆盖了上一次指定的命令。

Circular XXX <- YYY dependency dropped.

规则的依赖关系产生了循环：目标“XXX”的依赖文件为“YYY”，而依赖“YYY”的依赖列表中又包含“XXX”。

Recursive variable `XXX' references itself (eventually). Stop.

make的变量“XXX”（递归展开式）在替换展开时，引用它自身。无论对于直接展开式变量（通过:=定义的）或追加定义（+=），这都是不允许的。参考 第 5 章 使用变量

Unterminated variable reference. Stop.

变量或者函数引用语法不正确，没有使用完整的的括号（缺少一左括号或者右括号）。

insufficient arguments to function `XXX'. Stop.

函数“XXX”引用时参数数目不正确。函数缺少参数。

missing target pattern. Stop.

multiple target patterns. Stop.

target pattern contains no `%'. Stop.

mixed implicit and static pattern rules. Stop.

不正确的静态模式规则。

第一条错误的原因是：静态模式规则的目标段中没有模式目标；

第二条错误的原因是：静态模式规则的目标段中存在多个模式目标；

第三条错误的原因是：静态模式规则的目标段目标模式中没有包含模式字符“%”；

第四条错误的原因是：静态模式规则的三部分都包含了模式字符“%”。正确的应该是只有后两个才可以包含模式字符“%”。

关于静态模式规则可参考 3.12 静态模式 一节

warning: -jN forced in submake: disabling jobserver mode.

这一条告警和下条告警信息发生在：make检测到递归的make调用时，可通信的子make进程出现并行处理的错误（参考 4.6 make的递归执行 一节）。递归执行的make的命令行参数中存在“-jN”参数（N的值大于1），在有些情况下可能导致此错误，例如：Makefile中变量“MAKE”被赋值为“make –j2”，并且递归调用的命令行中使用变量“MAKE”。在这种情况下，被调用make进程不能和其它make进程进行通信，其只能简单的独立的并行处理两个任务”。

warning: jobserver unavailable: using -j1. Add `+' to parent make rule.

为了现实make进程之间的通信，上层make进程将传递信息给子make进程。在传递信息过程中可能存在这种情况，子make进程不是一个实际的make进程，而上层make却不能确定子进程是否是真实的make进程。它只是将所有信息传递下去。上层make采用正常的算法来决定这些（参考 4.6.1 变量MAKE 一小节）。当出现这种情况，子进程只会接受父进程传递的部分有用的信息。子进程会产生该警告信息，之后按照其内建的顺序方式进行处理。

附录1：关键字索引

GNU make可识别的指示符：

define VARIABLE

endef　　参考 5.8 多行定义 一节

---

ifdef VARIABLE

ifndef VARIABLE

ifeq (A,B)

ifeq "A" "B"
ifeq 'A' 'B'
ifneq (A,B)
ifneq "A" "B"
ifneq 'A' 'B'
else
endif
	参考 6.2 条件判断的基本语法 一节

include FILE -include FILE sinclude FILE	参考 2.3 包含其他makefile文件 一节
override VARIABLE = VALUE override VARIABLE := VALUE override VARIABLE += VALUE override VARIABLE ?= VALUE override define VARIABLE endef	参考 5.7 override指示符 一节
export unexport VARIABLE	参考 4.6 make的递归执行 一节
vpath PATTERN PATH vpath PATTERN vpath	参考 3.12 目录搜索 一节

GNU make函数：

subst
patsubst
strip
findstring
filter
filter-out
sort
word
words

wordlist	
firstword	参考 7.2 文本处理函数 一节
dir	
notdir	
suffix	
basename	
addsuffix	
addprefix	
join	
wildcard	参考 7.3 文件名处理函数 一节
error	参考 7.11 make的控制函数 一节
warning	参考 7.11 make的控制函数 一节
shell	参考 7.10 shell函数 一节
origin	参考 7.9 origin函数 一节
foreach	参考 7.4 foreach函数 一节
call	参考 7.6 call函数 一节
if	参考 7.5 if函数 一节
eval	参考 7.8 eval函数 一节
value	参考 7.7 value函数 一节

GNU make的自动化变量

$@

$%

$<

$?

$^

$+

$*

$(@D)

$(@F)

$(*D)

$(*F)

$(%D)

$(%F)
$(<D)
$(<F)
$(^D)
$(^F)
$(+D)
$(+F)
$(?D)
$(?F) 参考 9.5.3 自动化变量 一小节

GNU make环境变量

MAKEFILES	参考 2.4 变量 MAKEFILES 一节
MAKEFILES_LIST	参考 2.5 变量 MAKEFILE_LIST 一节
VPATH	参考 3.5.1 一般搜索 一小节
SHELL	参考 4.2 命令的执行 一节
MAKESHELL	参考 4.2 命令的执行 一节
MAKE	参考 4.6 make的递归执行 一节
MAKELEVEL	参考 4.6 make的递归执行 一节
MAKEFLAGS	参考 4.6 make的递归执行 一节
MAKECMDGOALS	参考 8.2 指定终极目标 一节
CURDIR	参考 4.6 make的递归执行 一节
SUFFIXES	参考 9.8 隐含规则的搜索算法 一节
.LIBPATTERNS	参考 3.5.6 库文件和搜索目录 一小节

后序

刚开始在 Linux 环境下开发时，对于 make 这个东西没有多少了解。工作中网络、Linux 社区论坛给了我很大的帮助。这些对我就像启蒙的教育一样重要、一样有影响力。特别感谢那些前辈们的经验总结。特别感谢 make 程序的设计开发者们。也特别感谢“info make”原文的作者。

介绍一下 make 程序的设计开发者

首当其冲的是： Richard Stallman

开源软件的领袖和先驱，从来没有领过一天工资，从来没有使用过 Windows 操作系统。他的事迹和他的软件以及他的思想，虽然以前听说过他，但是对他的思想不是很了解。最近通过网络和一些资料对他的事迹有更多地了解。对他的思想和个人情操致上我的敬意。

他的他的主页：http://www.stallman.org/ 。

徐海兵 2006 – 01 -20