

GNU make 指南*

Goerge[†]@Merton College.Oxford University.England 著

哈 少 翻译

尚书林@自由软件协会.CASIA 整理

译者按：本文是一篇介绍 GNU Make 的文章，读完后读者应该基本掌握了 make 的用法。而 make 是所有想在 Unix（当然也包括 Linux）系统上编程的用户必须掌握的工具。如果你写的程序中没有用到 make，则说明你写的程序只是个人的练习程序，不具有任何实用的价值。也许这么说有点儿偏激，但 make 实在是应该用在任何稍具规模的程序中的。希望本文可以为中国的 Unix 编程初学者提供一点儿有用的资料。中国的 Linux 用户除了学会安装红帽子以外，实在应该尝试写一些有用的程序。个人想法，大家参考。

拒绝承诺：作者对于任何因此而对任何事物造成的所有损害（你所拥有或不拥有的实际的，抽象的，或者虚拟的）。所有的损坏都是你自己的责任，而与我无关。

所有权：“多文件项目”部分属于作者的财产，版权归乔治富特 1997 年五月至七月。其它部分属 CScene 财产，版权 CScene 1997 年，保留所有版权。本 CScene 文章的分发，部分或全部，应依照所有其它 CScene 的文章的条件来处理。

1 介绍

本文将首先介绍为什么要将你的 C 源代码分离成几个合理的独立档案，什么时候需要分，怎么才能分的好。然后将会告诉你 GNU Make 怎样使你的编译和连接步骤自动化。对于其它 Make 工具的用户来说，虽然在用其它类似工具时要做适当的调整，本文的内容仍然是非常有用的。如果对你自己的编程工具有怀疑，可以实际的试一试，但请先阅读用户手册。

2 多文件项目

2.1 为什么使用它们？

首先，多文件项目的好处在那里呢？

它们看起来把事情弄的复杂无比。又要 header 文件，又要 extern 声明，而且如果需要查找一个文件，你要在更多的文件里搜索。

但其实我们很有力的理由支持我们把一个项目分解成小块。当你改动一行代码，编译器需要全部重新编译来生成一个新的可执行文件。但如果你的项目是分开在几个小文件里，当你改动其中一个文件的时候，别的源文件的目标文件(object files)已经存在，所以没有什么原因去重新编译它们。你所需要做的只是重现编译被改动过的那个文件，然后重新连接所有的目标文件罢了。在大型的项目中，这意味着从很长的（几分钟到几小时）重新编译缩短为十几，二十几秒的简单调整。

只要通过基本的规划，将一个项目分解成多个小文件可使你更加容易的找到一段代码。很简单，你根据代码的作用把你的代码分解到不同的文件里。当你要看一段代码时，你可以准确的知道在那个文件中去寻找它。

从很多目标文件生成一个程序包 (Library)比从一个单一的大目标文件生成要好的多。当然实际上这是否真是一个优势则是由你所用的系统来决定的。但是当使用 gcc/ld (一个 GNU C 编译 / 连接器) 把一个程序包连接到一个程序时，在连接的过程中，它会尝试不去连接没有使用到的部分。但它每次只能从程序包中把一个完整的目标文件排除在外。因此如果你参考一个程序包中某一个目标档中任何一个符号的话，那么这个目标文件整个都会被连接进来。要

*原名: 多文件项目和 GNU Make 工具

[†]Footgeorge.foot@merton.ox.ac.uk

是一个程序包被非常充分的分解了的话，那么经连接后，得到的可执行文件会比从一个大目标文件组成的程序包连接得到的文件小得多。

又因为你的程序是很模块化的，文件之间的共享部分被减到最少，那就有很多好处——可以很容易的追踪到臭虫，这些模块经常是可以用在其它的项目里的，同时别人也可以更容易的理解你的一段代码是干什么的。当然此外还有许多别的好处……

2.2 何时分解你的项目

很明显，把任何东西都分解是不合理的。象“世界，你们好”这样的简单程序根本就不能分，因为实在也没什么可分的。把用于测试用的小程序分解也是没什么意思的。但一般来说，当分解项目有助于布局、发展和易读性的时候，我都会采取它。在大多数的情况下，这都是适用的。（所谓“世界，你们好”，既 'hello world'，只是一个介绍一种编程语言时惯用的范例程序，它会在屏幕上显示一行 'hello world'。是最简单的程序。）

如果你需要开发一个相当大的项目，在开始前，应该考虑一下你将如何实现它，并且生成几个文件（用适当的名字）来放你的代码。当然，在你的项目开发的过程中，你可以建立新的文件，但如果你这么做的话，说明你可能改变了当初的想法，你应该想想是否需要对整个结构也进行相应的调整。

对于中型的项目，你当然也可以采用上述技巧，但你也可以就那么开始输入你的代码，当你的码多到难以管理的时候再把它们分解成不同的档案。但以我的经验来说，开始时在脑子里形成一个大概的方案，并且尽量遵从它，或在开发过程中，随着程序的需要而修改，会使开发变得更加容易。

2.3 怎样分解项目

先说明，这完全是我个人的意见，你可以（也许你真的会？）用别的方式来做。这会触动到有关编码风格的问题，而大家从来就没有停止过在这个问题上的争论。在这里我只是给出我自己喜欢的做法（同时也给出这么做的原因）：

- i) 不要用一个 header 文件指向多个源码文件（例外：程序包的 header 文件）。用一个 header 定义一个源码文件的方式会更有效，也更容易查寻。否则改变一个源文件的结构（并且它的 header 文件）就必须重新编译好几个文件。
- ii) 如果可以的话，完全可以用超过一个的 header 文件来指向同一个源码文件。有时将不可公开调用的函数原型，类型定义等等，从它们的 C 源码文件中分离出来是非常有用的。使用一个 header 文件装公开符号，用另一个装私人符号意味着如果你改变了这个源码文件的内部结构，你可以只是重新编译它而不需要重新编译那些使用它的公开 header 文件的其它的源文件。
- iii) 不要在多个 header 文件中重复定义信息。如果需要，在其中一个 header 文件里 `#include` 另一个，但是不要重复输入相同的 header 信息两次。原因是如果你以后改变了这个信息，你只需要把它改变一次，不用搜索并改变另外一个重复的信息。
- iv) 在每一个源码文件里，`#include` 那些声明了源码文件中的符号的所有 header 文件。这样一来，你在源码文件和 header 文件对某些函数做出的矛盾声明可以比较容易的被编译器发现。

2.4 对于常见错误的注释

- a) 定义符 (Identifier) 在源码文件中的矛盾：在 C 里，变量和函数的缺省状态是公用的。因此，任何 C 源码档案都可以引用存在于其它源码档中的通用 (global) 函数和通用变量，即使这个档案没有那个变量或函数的声明或原型。因此你必须保证在不同的两个档案里不能用同一个符号名称，否则会有连接错误或者在编译时会有警告。一种避免这种错误的方法是在公用的符号前加上跟其所在源文件有关的前缀。比如：所有在 `gfx.c` 里的函数都加上前缀“`gfx_`”。如果你很小心的分解你的程序，使用有意义的函数名称，并且不是过分使用通用变量，当然这根本就不是问题。

要防止一个符号在它被定义的源文件以外被看到，可在它的定义前加上关键字“`static`”。这对只在一个档案内部使用，其它档案都不会用到的简单函数是很有用的。

- b) 多次定义的符号：header 档会被逐字的替换到你源文件里 `#include` 的位置的。因此，如果 header 档被 `#include` 到一个以上的源文件 里，这个 header 档中所有的定义就会出现在每一个有关的源码文件里。这会使得它们里的符号被定义一次以上，从而出现连接错误（见 上）。

解决方法：不要在 header 档里定义变量。你只需要在 header 档里声明它们然后在适当的 C 源码文件（应该 `#include` 那个 header 档的那个）里定义它们（一次）。对于初学者来说，定义和声明是很容易混淆的。声明的作用是告诉编译器其所声明的符号应该存在，并且要有所指定的类型。但是，它并不会使编译器分配贮存空间。而定义的作用是要要求编译器分配贮存空间。当做一个声明而不是做定义的时候，在声明前放一个关键字“`extern`”。

例如，我们有一个叫“`counter`”的变量，如果想让它成为公用的，我们在一个源码程序（只在一个里面）的开始定义它：“`int counter;`”，再在相关的 header 档里声明它：“`extern int counter;`”。函数原型里隐含着 `extern` 的意思，所以不需顾虑这个问题。

- c) 重复定义，重复声明，矛盾类型：

请考虑如果在一个 C 源码文件中 `#include` 两个档 `a.h` 和 `b.h`，而 `a.h` 又 `#include` 了 `b.h` 档（原因是 `b.h` 档定义了一些 `a.h` 需要的类型），会发生什么事呢？这时该 C 源码文件 `#include` 了 `b.h` 两次。因此每一个在 `b.h` 中的 `#define` 都发生了两次，每一个声明发生了两次，等等。理论上，因为它们是完全一样的拷贝，所以应该不会有什问题，但在实际应用上，这是不符合 C 的语法的，可能在编译时出现错误，或至少是警告。

解决的方法是要确定每一个 header 档在任一个源码文件中只被包含了一次。我们一般是用预处理器来达到这个目的。当我们进入每一个 header 档时，我们为这个 header 档 `#define` 一个巨集指令。只有在这个巨集指令没有被定义的前提下，我们才真正使用该 header 档的主体。在实际应用上，我们只要简单的把下面一段码放在每一个 header 档的开始部分：

```
#ifndef FILENAME_H
#define FILENAME_H
```

然后把下面一行码放在最后：

```
#endif
```

用 header 档的档名（大写的）代替上面的 `FILENAME_H`，用底线代替档名中的点。有些人喜欢在 `#endif` 加上注释来提醒他们这个 `#endif` 指的是什么。例如：

```
#endif /* #ifndef FILENAME_H */
```

我个人没有这个习惯，因为这其实是很明显的。当然这只是各人的风格不同，无伤大雅。你只需要在那些有编译错误的 header 档中加入这个技巧，但在所有的 header 档中都加入也没什么损失，到底这是个好习惯。

2.5 重新编译一个多文件项目

清楚的区别编译和连接是很重要的。编译器使用源码文件来产生某种形式的目标文件(object files)。在这个过程中，外部的符号参考并没有被解释或替换。然后我们使用连接器来连接这些目标文件和一些标准的程序包再加你指定的程序包，最后连接生成一个可执行程序。在这个阶段，一个目标文件中对别的文件中的符号的参考被解释，并报告不能被解释的参考，一般是以错误信息的形式报告出来。

基本的步骤就应该是，把你的源码文件一个一个的编译成目标文件的格式，最后把所有的目标文件加上需要的程序包连接成一个可执行文件。具体怎么做是由你的编译器决定的。这里我只给出 `gcc`（GNU C 编译器）的有关命令，这些有可能对你的非 `gcc` 编译器也适用。

`gcc` 是一个多目标的工具。它在需要的时候呼叫其它的元件（预处理程序，编译器，组合程序，连接器）。具体的哪些元件被呼叫取决于输入文件的类型和你传递给它的开关。

一般来说，如果你只给它 C 源码文件，它将预处理，编译，组合所有的文件，然后把所得的目标文件连接成一个可执行文件（一般生成的文件被命名为 `a.out`）。你当然可以这么做，但这会破坏很多我们把一个项目分解成多个文件所得到的好处。

如果你给它一个 `-c` 开关，`gcc` 只把给它的文件编译成目标文件，用源码文件的文件名命名但把其后缀由“`.c`”或“`.cc`”变成“`.o`”。如果你给它的是一列目标文件，`gcc` 会把它们连接成可执行文件，缺省文件名是 `a.out`。你可以改变缺省名，用开关 `-o` 后跟你指定的文件名。

因此，当你改变了一个源码文件后，你需要重新编译它：‘gcc -c filename.c’ 然后重新连接你的项目：‘gcc -o exec_filename *.o’。如果你改变了一个 header 档，你需要重新编译所有 #include 过这个档的源码文件，你可以用 ‘gcc -c file1.c file2.c file3.c’ 然后象上边一样连接。

当然这么做是很繁琐的，幸亏我们有些工具使这个步骤变得简单。本文的第二部分就是介绍其中的一件工具：GNU Make 工具。

（好家伙，现在才开始见真章。您学到点儿东西没？）

3 GNU Make 工具

3.1 基本 makefile 结构

GNU Make 的主要工作是读进一个文本文件 makefile。这个文件里主要是有关哪些文件（‘target’ 目的文件）是从哪些别的文件（‘dependencies’ 依靠文件）中产生的，用什么命令来进行这个产生过程。有了这些信息，make 会检查磁碟上的文件，如果目的文件的时间戳（该文件生成或被改动时的时间）比至少它的一个依靠文件旧的话，make 就执行相应的命令，以便更新目的文件。（目的文件不一定是最后的可执行档，它可以是任何一个文件。）

makefile 一般被叫做 “makefile” 或 “Makefile”。当然你可以在 make 的命令行指定别的文件名。如果你不特别指定，它会寻找 “makefile” 或 “Makefile”，因此使用这两个名字是最简单的。

一个 makefile 主要含有一系列的规则，如下：

```
: ...
(tab) $<$command$>$
(tab) $<$command$>$
.
.
.
```

例如，考虑以下的 makefile：

=== makefile 开始 ===

```
myprog : foo.o bar.o
        gcc foo.o bar.o -o myprog

foo.o : foo.c foo.h bar.h
        gcc -c foo.c -o foo.o

bar.o : bar.c bar.h
        gcc -c bar.c -o bar.o
```

=== makefile 结束 ===

这是一个非常基本的 makefile——make 从最上面开始，把上面第一个目的，‘myprog’，做为它的主要目标（一个它需要保证其总是最新的最终目标）。给出的规则说明只要文件 ‘myprog’ 比文件 ‘foo.o’ 或 ‘bar.o’ 中的任何一个旧，下一行的命令将会被执行。

但是，在检查文件 foo.o 和 bar.o 的时间戳之前，它会往下查找那些把 foo.o 或 bar.o 做为目标文件的规则。它找到的关于 foo.o 的规则，该文件的依靠文件是 foo.c, foo.h 和 bar.h。它从下面再找不到生成这些依靠文件的规则，它就开始检查磁碟上这些依靠文件的时间戳。如果这些文件中任何一个的时间戳比 foo.o 的新，命令 ‘gcc -o foo.o foo.c’ 将会执行，从而更新文件 foo.o。

接下来对文件 bar.o 做类似的检查，依靠文件在这里是文件 bar.c 和 bar.h。

现在，make 回到 ‘myprog’ 的规则。如果刚才两个规则中的任何一个被执行，myprog 就需要重建（因为其中一个 .o 档就会比 ‘myprog’ 新），因此连接命令将被执行。

希望到此，你可以看出使用 make 工具来建立程序的好处——前一章中所有繁琐的检查步骤都由 make 替你做了：检查时间戳。你的源码文件里一个简单改变都会造成那个文件被重新编译（因为 .o 文件依靠 .c 文件），进而

可执行文件被重新连接（因为 .o 文件被改变了）。其实真正的得益是在当你改变一个 header 档的时候——你不再需要记住那个源码文件依靠它，因为所有的资料都在 makefile 里。make 会很轻松的替你重新编译所有那些因依靠这个 header 文件而改变了的源码文件，如有需要，再进行重新连接。

当然，你要确定你在 makefile 中所写的规则是正确无误的，只列出那些在源码文件中被 #include 的 header 档……

3.2 编写 make 规则 (Rules)

最明显的（也是最简单的）编写规则的方法是一个一个的查看源码文件，把它们的目标文件做为目的，而 C 源码文件和被它 #include 的 header 档做为依靠文件。但是你也要把其它被这些 header 档 #include 的 header 档也列为依靠文件，还有那些被包括的文件所包括的文件……然后你会发现要对越来越多的文件进行管理，然后你的头发开始脱落，你的脾气开始变坏，你的脸色变成菜色，你走在路上开始跟电线杆子碰撞，终于你捣毁你的电脑显示器，停止编程。到底有没有些容易点儿的方法呢？

当然有！向编译器要！在编译每一个源码文件的时候，它实在应该知道应该包括什么样的 header 档。使用 gcc 的时候，用 -M 开关，它会为每一个你给它的 C 文件输出一个规则，把目标文件做为目的，而这个 C 文件和所有应该被 #include 的 header 文件将做为依靠文件。注意这个规则会加入所有 header 文件，包括被角括号(<, >)和双引号(" ")所包围的文件。其实我们可以相当肯定系统 header 档（比如 stdio.h, stdlib.h 等等）不会被我们更改，如果你用 -MM 来代替 -M 传递给 gcc，那些用角括号包围的 header 档将不会被包括。（这会节省一些编译时间）

由 gcc 输出的规则不会含有命令部分；你可以自己写入你的命令或者什么也不写，而让 make 使用它的隐含的规则（参考下面的 2.4 节）。

3.3 Makefile 变量

上面提到 makefiles 里主要包含一些规则。它们包含的其它的东西是变量定义。

makefile 里的变量就像一个环境变量(environment variable)。事实上，环境变量在 make 过程中被解释成 make 的变量。这些变量是大小写敏感的，一般使用大写字母。它们可以从几乎任何地方被引用，也可以被用来做很多事情，比如：

- i) 贮存一个文件名列表。在上面的例子里，生成可执行文件的规则包含一些目标文件名做为依靠。在这个规则的命令行里同样的那些文件被输送给 gcc 做为命令参数。如果在这里使用一个变数来贮存所有的目标文件名，加入新的目标文件会变的简单而且较不易出错。
- ii) 贮存可执行文件名。如果你的项目被用在一个非 gcc 的系统里，或者如果你想使用一个不同的编译器，你必须将所有使用编译器的地方改成用新的编译器名。但是如果使用一个变量来代替编译器名，那么你只需要改变一个地方，其它所有地方的命令名就都改变了。
- iii) 贮存编译器旗标。假设你想给你所有的编译命令传递一组相同的选项（例如 -Wall -O -g）；如果你把这组选项存入一个变量，那么你可以把这个变量放在所有呼叫编译器的地方。而当你改变选项的时候，你只需在一个地方改变这个变量的内容。

要设定一个变量，你只要在一行的开始写下这个变量的名字，后面跟一个 = 号，后面跟你要设定的这个变量的值。以后你要引用这个变量，写一个 \$ 符号，后面是围在括号里的变量名。比如在下面，我们把前面的 makefile 利用变量重写一遍：

===== makefile 开始 =====

```
OBJS = foo.o bar.o
CC = gcc
CFLAGS = -Wall -O -g

myprog : $(OBJS)
        $(CC) $(OBJS) -o myprog

foo.o : foo.c foo.h bar.h
```

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

```
bar.o : bar.c bar.h
```

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

===== makefile 结束 =====

还有一些设定好的内部变量，它们根据每一个规则内容定义。三个比较有用的变量是 `$$`, `$<` 和 `$$` (这些变量不需要括号括住)。`$$` 扩展成当前规则的目的文件名，`$<` 扩展成依靠列表中的第一个依靠文件，而 `$$` 扩展成整个依靠的列表 (除掉了里面所有重复的文件名)。利用这些变量，我们可以把上面的 makefile 写成：

===== makefile 开始 =====

```
OBJS = foo.o bar.o
```

```
CC = gcc
```

```
CFLAGS = -Wall -O -g
```

```
myprog : $(OBJS)
```

```
$(CC) $$ -o $$
```

```
foo.o : foo.c foo.h bar.h
```

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

```
bar.o : bar.c bar.h
```

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

===== makefile 结束 =====

你可以用变量做许多其它的事情，特别是当你把它们和函数混合使用的时候。如果需要更进一步的了解，请参考 GNU Make 手册。 ('man make', 'man makefile')

3.4 隐含规则 (Implicit Rules)

请注意，在上面的例子里，几个产生 .o 文件的命令都是一样的。都是从 .c 文件和相关文件里产生 .o 文件，这是一个标准的步骤。其实 make 已经知道怎么做——它有一些叫做隐含规则的内置的规则，这些规则告诉它当你没有给出某些命令的时候，应该怎么办。

如果你把生成 foo.o 和 bar.o 的命令从它们的规则中删除，make 将会查找它的隐含规则，然后会找到一个适当的命令。它的命令会使用一些变量，因此你可以按照你的想法来设定它：它使用变量 CC 做为编译器（象我们在前面的例子），并且传递变量 CFLAGS（给 C 编译器，C++ 编译器用 CXXFLAGS），CPPFLAGS（C 预处理器旗标），TARGET_ARCH（现在不用考虑这个），然后它加入旗标 '-c'，后面跟变量 \$<（第一个依靠名），然后是旗标 '-o' 跟变量 \$\$（目的文件名）。一个 C 编译的具体命令将会是：

```
$(CC) $(CFLAGS) $(CPPFLAGS) $(TARGET_ARCH) -c $$< -o $$
```

当然你可以按照你自己的需要来定义这些变量。这就是为什么用 gcc 的 -M 或 -MM 开关输出的码可以直接用在一个 makefile 里。

3.5 假象目的 (Phony Targets)

假设你的一个项目最后需要产生两个可执行文件。你的主要目标是产生两个可执行文件，但这两个文件是相互独立的——如果一个文件需要重建，并不影响另一个。你可以使用“假象目的”来达到这种效果。一个假象目的跟一个正常的目的几乎是一样的，只是这个目的文件是不存在的。因此，make 总是会假设它需要被生成，当把它的依赖文件更新后，就会执行它的规则里的命令行。

如果在我们的 makefile 开始处输入：

```
all : exec1 exec2
```

其中 `exec1` 和 `exec2` 是我们做为目的的两个可执行文件。 `make` 把这个 `'all'` 做为它的主要目的，每次执行时都会尝试把 `'all'` 更新。但既然这行规则里没有哪个命令来作用在一个叫 `'all'` 的实际文件（事实上 `all` 并不会在磁碟上实际产生），所以这个规则并不真的改变 `'all'` 的状态。可既然这个文件并不存在，所以 `make` 会尝试更新 `all` 规则，因此就检查它的依靠 `exec1`, `exec2` 是否需要更新，如果需要，就把它们更新，从而达到我们的目的。

假象目的也可以用来描述一组非预设的动作。例如，你想把所有由 `make` 产生的文件删除，你可以在 `makefile` 里设立这样一个规则：

```
veryclean :
    rm *.o
    rm myprog
```

前提是没有其它的规则依靠这个 `'veryclean'` 目的，它将永远不会被执行。但是，如果你明确的使用命令 `'make veryclean'`， `make` 会把这个目的做为它的主要目标，执行那些 `rm` 命令。

如果你的磁碟上存在一个叫 `veryclean` 文件，会发生什么事？这时因为在这个规则里没有任何依靠文件，所以这个目的文件一定是最新了（所有的依靠文件都已经是最新了），所以即使用户明确命令 `make` 重新产生它，也不会有任何事情发生。解决方法是标明所有的假象目的（用 `.PHONY`），这就告诉 `make` 不用检查它们是否存在于磁碟上，也不用查找任何隐含规则，直接假设指定的目的需要被更新。在 `makefile` 里加入下面这行包含上面规则的规则：

```
.PHONY : veryclean
```

就可以了。注意，这是一个特殊的 `make` 规则，`make` 知道 `.PHONY` 是一个特殊目的，当然你可以在它的依靠里加入你想用的任何假象目的，而 `make` 知道它们都是假象目的。

3.6 函数 (Functions)

`makefile` 里的函数跟它的变量很相似——使用的时候，你用一个 `$` 符号跟开括号，函数名，空格后跟一系列由逗号分隔的参数，最后用关括号结束。例如，在 GNU Make 里有一个叫 `'wildcard'` 的函数，它有一个参数，功能是展开成一系列所有符合由其参数描述的文件名，文件间以空格间隔。你可以像下面所示使用这个命令：

```
SOURCES = $(wildcard *.c)
```

这行会产生一个所有以 `'.c'` 结尾的文件的列表，然后存入变量 `SOURCES` 里。当然你不需要一定要把结果存入一个变量。

另一个有用的函数是 `patsubst`（`patten substitute`, 匹配替换的缩写）函数。它需要 3 个参数——第一个是一个需要匹配的式样，第二个表示用什么来替换它，第三个是一个需要被处理的由空格分隔的字列。例如，处理那个经过上面定义后的变量，

```
OBJS = $(patsubst %.c,%.o,$(SOURCES))
```

这行将处理所有在 `SOURCES` 字列中的字（一系列文件名），如果它的结尾是 `'.c'`，就用 `'.o'` 把 `'.c'` 取代。注意这里的 `%` 符号将匹配一个或多个字符，而它每次所匹配的字串叫做一个‘柄’（`stem`）。在第二个参数里，`%` 被解读成用第一参数所匹配的那个柄。

3.7 一个比较有效的 `makefile`

利用我们现在所学的，我们可以建立一个相当有效的 `makefile`。这个 `makefile` 可以完成大部分我们需要的依靠检查，不用做太大的改变就可直接用在大多数的项目里。

首先我们需要一个基本的 `makefile` 来建我们的程序。我们可以让它搜索当前目录，找到源码文件，并且假设它们都是属于我们的项目的，放进一个叫 `SOURCES` 的变量。这里如果也包含所有的 `*.cc` 文件，也许会更保险，因为源码文件可能是 C++ 码的。

```
SOURCES = $(wildcard *.c *.cc)
```

利用 `patsubst`，我们可以由源码文件名产生目标文件名，我们需要编译出这些目标文件。如果我们的源码文件既有 `.c` 文件，也有 `.cc` 文件，我们需要使用相嵌的 `patsubst` 函数呼叫：

```
OBJS = $(patsubst %.c,%.o,$(patsubst %.cc,%.o,$(SOURCES)))
```

最里面一层 `patsubst` 的呼叫会对 `.cc` 文件进行后缀替代，产生的结果被外层的 `patsubst` 呼叫处理，进行对 `.c` 文件后缀的替代。

现在我们可以设立一个规则来建可执行文件：

```
myprog : $(OBJS)
        gcc -o myprog $(OBJS)
```

进一步的规则不一定需要，`gcc` 已经知道怎么去生成目标文件(object files)。下面我们可以设定产生依靠信息的规则：

```
depends : $(SOURCES)
        gcc -M $(SOURCES) > depends
```

在这里如果有一个叫 `'depends'` 的文件不存在，或任何一个源码文件比一个已存在的 `depends` 文件新，那么一个 `depends` 文件会被生成。`depends` 文件将会含有由 `gcc` 产生的关于源码文件的规则（注意 `-M` 开关）。现在我们要让 `make` 把这些规则当做 `makefile` 档的一部分。这里使用的技巧很像 C 语言中的 `#include` 系统——我们要求 `make` 把这个文件 `include` 到 `makefile` 里，如下：

```
include depends
```

GNU Make 看到这个，检查 `'depends'` 目的是否更新了，如果没有，它用我们给它的命令重新产生 `depends` 档。然后它会把这组（新）规则包含进来，继续处理最终目标 `'myprog'`。当看到有关 `myprog` 的规则，它会检查所有的目标文件是否更新——利用 `depends` 文件里的规则，当然这些规则现在已经是更新过的了。

这个系统其实效率很低，因为每当一个源码文件被改动，所有的源码文件都要被预处理以产生一个新的 `'depends'` 文件。而且它也不是 100% 的安全，这是因为当一个 `header` 档被改动，依靠信息并不会被更新。但就基本工作来说，它也算相当有用的了。

3.8 一个更好的 `makefile`

这是一个我为我大多数项目设计的 `makefile`。它应该可以不需要修改的用在大部分项目里。我主要把它用在 `djgpp` 上，那是一个 DOS 版的 `gcc` 编译器。因此你可以看到执行的命令名、`'alleg'` 程序包、和 `RM -F` 变量都反映了这一点。

===== `makefile` 开始 =====

```
#####
#
# Generic makefile
#
# by George Foot
# email: george.foot@merton.ox.ac.uk
#
# Copyright (c) 1997 George Foot
# All rights reserved.
#
# No warranty, no liability;
# you use this at your own risk.
#
# You are free to modify and
# distribute this without giving
# credit to the original author.
#
#####

### Customising
```



```

#
# Adjust the following if necessary; EXECUTABLE is the target
# executable's filename, and LIBS is a list of libraries to link in
# (e.g. alleg, stdcx, iostr, etc). You can override these on make's
# command line of course, if you prefer to do it that way.

EXECUTABLE := mushroom.exe
LIBS := alleg

# Now alter any implicit rules' variables if you like, e.g.:

CFLAGS := -g -Wall -O3 -m486
CXXFLAGS := $(CFLAGS)

# The next bit checks to see whether rm is in your djgpp bin
# directory; if not it uses del instead, but this can cause (harmless)
# 'File not found' error messages. If you are not using DOS at all,
# set the variable to something which will unquestioningly remove
# files.

ifneq ($(wildcard $(DJDIR)/bin/rm.exe),)
RM-F := rm -f
else
RM-F := del
endif

# You shouldn't need to change anything below this point.

SOURCE := $(wildcard *.c) $(wildcard *.cc)
OBJS := $(patsubst %.c,%.o,$(patsubst %.cc,%.o,$(SOURCE)))
DEPS := $(patsubst %.o,%.d,$(OBJS))
MISSING_DEPS := $(filter -out $(wildcard $(DEPS)),$(DEPS))
MISSING_DEPS_SOURCES := $(wildcard $(patsubst %.d,%.c,$(MISSING_DEPS)) \
$(patsubst %.d,%.cc,$(MISSING_DEPS)))
CPPFLAGS += -MD

.PHONY : everything deps objs clean veryclean rebuild

everything : $(EXECUTABLE)

deps : $(DEPS)

objs : $(OBJS)

clean :
    @$(RM-F) *.o
    @$(RM-F) *.d

veryclean: clean

```

```
@$(RM-F) $(EXECUTABLE)
```

```
rebuild: veryclean everything
```

```
ifneq ($(MISSING_DEPS),)
```

```
$(MISSING_DEPS) :
```

```
@$(RM-F) $(patsubst %.d,%.o,$@)
```

```
endif
```

```
-include $(DEPS)
```

```
$(EXECUTABLE) : $(OBJS)
```

```
gcc -o $(EXECUTABLE) $(OBJS) $(addprefix -l,$(LIBS))
```

==== makefile 结束 ====

有几个地方值得解释一下的。首先，我在定义大部分变量的时候使用的是 `:=` 而不是 `=` 符号。它的作用是立即把定义中参考到的函数和变量都展开了。如果使用 `=` 的话，函数和变量参考会留在那儿，就是说改变一个变量的值会导致其它变量的值也被改变。例如：

```
A = foo
```

```
B = $(A)
```

```
# 现在 B 是 $(A)，而 $(A) 是 'foo'。
```

```
A = bar
```

```
# 现在 B 仍然是 $(A)，但它的值已随着变成 'bar' 了。
```

```
B := $(A)
```

```
# 现在 B 的值是 'bar'。
```

```
A = foo
```

```
# B 的值仍然是 'bar'。
```

make 会忽略在 `#` 符号后面直到那一行结束的所有文字。

`ifneg...else...endif` 系统是 makefile 里让某一部分码有条件的失效 / 有效的工具。`ifeq` 使用两个参数，如果它们相同，它把直到 `else`（或者 `endif`，如果没有 `else` 的话）的一段码加进 makefile 里；如果不同，把 `else` 到 `endif` 间的一段码加入 makefile（如果有 `else`）。`ifneq` 的用法刚好相反。

`'filter-out'` 函数使用两个用空格分开的列表，它把第二列表中所有的存在于第一列表中的项目删除。我用它来处理 DEPS 列表，把所有已经存在的项目都删除，而只保留缺少的那些。

我前面说过，`CPPFLAGS` 存有用于隐含规则中传给预处理器的一些旗标。而 `-MD` 开关类似 `-M` 开关，但是从源码文件 `.c` 或 `.cc` 中形成的文件名是使用后缀 `.d` 的（这就解释了我形成 DEPS 变量的步骤）。DEPS 里提到的文件后来用 `'include'` 加进了 makefile 里，它隐藏了所有因文件不存在而产生的错误信息。

如果任何依靠文件不存在，makefile 会把相应的 `.o` 文件从磁碟上删除，从而使得 make 重建它。因为 `CPPFLAGS` 指定了 `-MD`，它的 `.d` 文件也被重新产生。

最后，`'addprefix'` 函数把第二个参数列表的每一项前缀上第一个参数值。

这个 makefile 的那些目的是（这些目的可以传给 make 的命令行来直接选用）：

- `everything`:（预设）更新主要的可执行程序，并且为每一个源码文件生成或更新一个 `'d'` 文件和一个 `'o'` 文件。
- `deps`: 只是为每一个源码程序产生或更新一个 `'d'` 文件。
- `objs`: 为每一个源码程序生成或更新 `'d'` 文件和目标文件。
- `clean`: 删除所有中介 / 依靠文件（`*.d` 和 `*.o`）。
- `veryclean`: 做 `'clean'` 和删除可执行文件。
- `rebuild`: 先做 `'veryclean'` 然后 `'everything'`；既完全重建。

除了预设的 `everything` 以外，这里头只有 `clean`，`veryclean`，和 `rebuild` 对用户是有意义的。

我还没有发现当给出一个源码文件的目录，这个 `makefile` 会失败的情况，除非依靠文件被弄乱。如果这种弄乱的情况发生了，只要输入 `'make clean'`，所有的目标文件和依靠文件会被删除，问题就应该被解决了。当然，最好不要把它们弄乱。如果你发现在某种情况下这个 `makefile` 文件不能完成它的工作，请告诉我，我会把它整好的。

4 总结

我希望这篇文章足够详细的解释了多文件项目是怎么运作的，也说明了怎样安全而合理的使用它。到此，你应该可以轻松的利用 GNU Make 工具来管理小型的项目，如果你完全理解了后面几个部分的话，这些对于你应该没什么困难。

GNU Make 是一件强大的工具，虽然它主要是用来建立程序，它还有很多别的用处。如果想要知道更多有关这个工具的知识，它的句法，函数和许多别的特点，你应该参看它的参考文件 (info pages, 别的 GNU 工具也一样，看它们的 info pages.)。