Advanced Programming in the UNIX Environment — 多文件项目和 *GNU make* 入门

Goerge Foot

Contents

1 make 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。又要头文件，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。

- 但其实我们有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，所以没有什么必要去重新编译它们。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库 (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`(GNU C 编译/连接器) 把一个库连接到一个程序时，在连接的过程中，它会尝试不去连接没有使用到的部分。但它每次只能从库中把一个完整的目标文件排除在外。因此如果你引用一个库中某一个目标文件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。要是一个库被非常充分地分解了的话，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库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 bugs，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 “Hello World!” 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、扩展和易读性的时候，我都会采取它。在大多数的情况下，这都是适用的。
- 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体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）：

  1. 不要用一个头文件指向多个源文件（例外：库文件的头文件）。用一个头定义一个源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（以及它的头文件）就必须重新编译好几个文件。
  2. 如果可以的话，完全可以用超过一个的头文件来指向同一个源文件。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 源文件中分离出来是非常有用的。使用一个头文件装公开符号，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头文件的其它的源文件。
  3. 不要在多个头文件中重复定义信息。如果需要，在其中一个头文件里`#include`另一个，但是不要重复输入相同的头信息两次。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。
  4. 在每一个源文件里，`#include`那些声明了源文件中的符号的所有头文件。这样一来，你在源文件和头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

1. 标识符 (Identifier) 在源文件中的矛盾：

   - 在 C 里，变量和函数的缺省状态是公用的。因此，任何 C 源文件都可以引用存在于其它源文件中的全局 (global) 函数和全局变量，既使这个文件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文件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里的函数前面都加上前缀 “gfx_”。如果你很小心的分解你的程序，使用有意义的函数名称，并且不是过分使用全局变量，当然这根本就不是问题。
   - 要防止一个标识符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”。这对只在一个文件内部使用，其它文件都不会用到的简单函数是很有用的。

2. 多次定义的标识符：

   - 头文件会被逐字的读取到你源文件里#include的位置的。因此，如果头文件被#include到一个以上的源文件里，这个头文件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：**不要在头文件里定义变量。你只需要在头文件里声明它们然后在适当的 C 源文件（应该#include那个头文件的那个）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。而定义的做用是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，如果想让它成为全局的，我们在一个源程序（只在一个里面）的开始定义它：`int counter;`，再在相关的头文件里声明它：`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。

3. 重复定义，重复声明，矛盾类型：

   - 请考虑如果在一个 C 源文件中#include两个文件a.h和b.h，而a.h又#include了b.h文件（原因是b.h文件定义了一些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，所以应该不会有什么问题，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，至少是警告。
   - 解决的方法是要确定每一个头文件在任一个源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的。当我们进入每一个头文件时，我们为这个头文件#define一个宏指令。只有在这个宏指令没有被定义的前提下，我们才真正使用该头文件的主体。在实际应用上，我们只要简单的把下面一段码放在每一个头文件的开始部分：

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

   然后把下面一行码放在最后：

```
#endif
```

   - 用头文件的文件名（大写的）代替上面的FILENAME_H，用下划线代替文件名中的点。
   - 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头文件中加入这个技巧，但在所有的头文件中都加入也没什么损失，说到底这是个好习惯。

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，外部的符号引用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库再加你指定的库，最后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引用被解释，并报告不能被解释的引用，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库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。它在需要的时候调用其它的组件（预处理程序，编译器，组合程序，连接器）。具体的哪些组件被调用取决于输入文件的类型和你传递给它的选项。
- 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选项，gcc只把给它的文件编译成目标文件，用源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，用选项-o后跟你指定的文件名。
- 因此，当你改变了一个源文件后，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。如果你改变了一个头文件，你需要重新编译所有#include过这个文件的源文件，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。

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”依赖文件）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会检查磁盘上的文件，如果目的文件的时间戳（该文件的 LMT）比至少它的一个依赖文件旧的话，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：

```
1 ...
2 (tab)<command>
3 (tab)<command>
4 ...
```

例如，考虑以下的makefile：

```
1 === makefile beginning ===
2 myprog : foo.o bar.o
3     gcc foo.o bar.o -o myprog
4 foo.o : foo.c foo.h bar.h
```

```
5     gcc -c foo.c -o foo.o
6 bar.o : bar.c bar.h
7     gcc -c bar.c -o bar.o
8 === makefile ending ===
```

- 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，该文件的依赖文件是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，它就开始检查磁盘上这些依赖文件的时间戳。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，依赖文件在这里是文件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文件就会比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—前一节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文件依赖.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头文件的时候—你不再需要记住那个源码文件依赖它，因为所有的信息都在makefile里。make会很轻松地替你重新编译所有那些因依赖这个头文件而改变了的源文件，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，只列出那些在源文件中被#include的头文件。

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地查看源码文件，把它们的目标文件做为目的，而 C 源文件和被它#include的头文件做为依赖文件。但是你也要把其它被这些头文件#include的头文件也列为依赖文件，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。到低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头文件。使用gcc的时候，用-M选项，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的头文件将做为依赖文件。注意这个规则会加入所有头文件，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头文件（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头文件将不会被包括。（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3.4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：

1. 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赖。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。
2. 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。
3. 贮存编译器标志。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调用编译器的地方。而当你要改变选项的时候，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，后面跟一个等号，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：

```
1  === makefile beginning ===
2  OBJS = foo.o bar.o
3  CC = gcc
4  CFLAGS = -Wall -O -g
5
6  myprog : $(OBJS)
7    $(CC) $(OBJS) -o myprog
8  foo.o : foo.c foo.h bar.h
9    $(CC) $(CFLAGS) -c foo.c -o foo.o
10 bar.o : bar.c bar.h
11   $(CC) $(CFLAGS) -c bar.c -o bar.o
12 === makefile ending ===
```

- 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，$<扩展成依赖列表中的第一个依赖文件，而$^扩展成整个依赖的列表（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：

```
1  === makefile beginning ===
2  OBJS = foo.o bar.o
3  CC = gcc
4  CFLAGS = -Wall -O -g
5
6  myprog : $(OBJS)
7    $(CC) $^ -o $@
8  foo.o : foo.c foo.h bar.h
9    $(CC) $(CFLAGS) -c $< -o $@
10 bar.o : bar.c bar.h
11   $(CC) $(CFLAGS) -c $< -o $@
12 === makefile ending ===
```

- 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 预处理器标志），TARGET_ARCH（现在不用考虑这个），然后它加入选项‘-c’，后面跟变量$<（第一个依赖名），然后是选项-o跟变量$@（目的文件名）。一个 C 编译的具体命令将会是：

```
1 $(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选项输出的码可以直接用在一个makefile里的原因。

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：

```
1 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’并不会在磁盘上实际产生），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，因此就检查它的依赖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：

```
1 veryclean :
2   rm *.o
3   rm myprog
```

- 前提是没有其它的规则依赖这个‘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。

- 如果你的磁盘上存在一个叫veryclean文件，会发生什么事？这时因为在这个规则里没有任何依赖文件，所以这个目的文件一定是最新的了（所有的依赖文件都已经是最新的了），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盘上，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：

```
1 .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：

```
1 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。当然你不一定非得把结果存入一个变量。

- 另一个有用的函数是patsubst（patten substitude, 模式替换的缩写）函数。它需要 3 个参数—第一个是一个需要匹配的模式，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符串序列。例如，处理那个经过上面定义后的变量，

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

这行将处理所有在SOURCES字列中的字（一列文件名），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赖检查，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++ 的。

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

- 利用patsubst，我们可以由源文件名产生目标文件名，我们需要编译出这些目标文件。如果我们的源文件既有.c文件，也有.cc文件，我们需要使用嵌套的patsubst函数调用：

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

最里面一层patsubst的调用会对.cc文件进行后缀替代，产生的结果被外层的patsubst调用处理，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：

```
1 myprog : $(OBJS)
2   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：

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

- 在这里如果一个叫 'depends' 的文件不存在，或任何一个源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文件的一部分。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文件。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，因为每当一个源文件被改动，所有的源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，这是因为当一个头文件被改动时，依赖信息并不会被更新。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地用在大部分项目里。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' 库、和 RM -F 变量都反映了这一点。

```
=== makefile beginning ===
######################################################################
#
# Generic makefile
#
# by George Foot
# email: george.foot@merton.ox.ac.uk
#
# Copyright (c) 1997 George Foot
# All rights reserved.
# 保留所有版权
#
# 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
```

```
32 CXXFLAGS := $(CFLAGS)
33
34 # The next bit checks to see whether rm is in your djgpp bin
35 # directory; if not it uses del instead, but this can cause
36 # (harmless) `File not found' error messages. If you are not
37 # using DOS at all, set the variable to something which will
38 # unquestioningly remove files.
39 ifneq ($(wildcard $(DJDIR)/bin/rm.exe),)
40 RM-F := rm -f
41 else
42 RM-F := del
43 endif
44
45 # You shouldn't need to change anything below this point.
46 SOURCE := $(wildcard *.c) $(wildcard *.cc)
47 OBJS := $(patsubst %.c,%.o,$(patsubst %.cc,%.o,$(SOURCE)))
48 DEPS := $(patsubst %.o,%.d,$(OBJS))
49 MISSING_DEPS := $(filter-out $(wildcard $(DEPS)),$(DEPS))
50 MISSING_DEPS_SOURCES := $(wildcard $(patsubst %.d,%.c,\
51   $(MISSING_DEPS)) $(patsubst %.d,%.cc,$(MISSING_DEPS)))
52 CPPFLAGS += -MD
53
54 .PHONY : everything deps objs clean veryclean rebuild
55
56 everything : $(EXECUTABLE)
57
58 deps : $(DEPS)
59
60 objs : $(OBJS)
61
62 clean :
63   @$(RM-F) *.o
64   @$(RM-F) *.d
65
66 veryclean: clean
67   @$(RM-F) $(EXECUTABLE)
68
69 rebuild: veryclean everything
70
71 ifneq ($(MISSING_DEPS),)
72 $(MISSING_DEPS) :
73   @$(RM-F) $(patsubst %.d,%.o,$@)
74 endif
75
76 -include $(DEPS)
77
78 $(EXECUTABLE) : $(OBJS)
79   gcc -o $(EXECUTABLE) $(OBJS) $(addprefix -l,$(LIBS))
80 === makefile ending ===
```

- 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间的一段代码加入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标志。而-MD选项类似-M选项，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。
- 如果任何依赖文件不存在，makefile会把相应的.o文件从磁盘上删除，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/依赖文件（*.d和*.o）。

  veryclean 做 'clean' 和删除可执行文件。

  rebuild 先做 'veryclean' 然后 'everything'；即完全重建。
- 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文件的目录，这个makefile会失败的情况，除非依赖文件被弄乱。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赖文件会被删除，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，这些对于你来说应该没什么困难。

- 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.)。

The End of `make` Introduction.