Makefile实战

零声学院 Darren 326873713
C/C++Linux服务器开发/高级架构师 https://ke.qq.com/course/420945?tuin=137bb271
带书签版本参考《源码和文档说明.txt》

课程重点讲解Cmake，makefile主要讲基础。

重点内容
- makefile原理
- 变量
- 函数
- 多目录
- 库生成

1 基础

为了更为高效的从事开发工作，以及做到在工作中对于处理与编译相关的错误更加的自信，驾驭 Makefile 是非常必要的。

Makefile 其实只是一个指示 make 程序（后面简称 make 或有时称之为 make 命令）如何为我们工作的命令文件，我们说 Makefile 其实是在说 make，这一点要有很清晰的认识。而对于我们的项目来说，Makefile 是指软件项目的编译环境。软件产品开发在编码阶段最常见的工作内容大致是：

1）开发人员根据概要设计进行编码。
2）开发人员编译所设计的源代码以生成可执行文件。
3）开发人员对软件产品进行测试来验证其功能的正确性。

上面的三个步骤是一个迭代过程，如果最终验证设计的正确性完全达到要求，那么就完成了编码阶段的开发，如果没有那还得重复这三个步骤，直到达到设计要求为止。

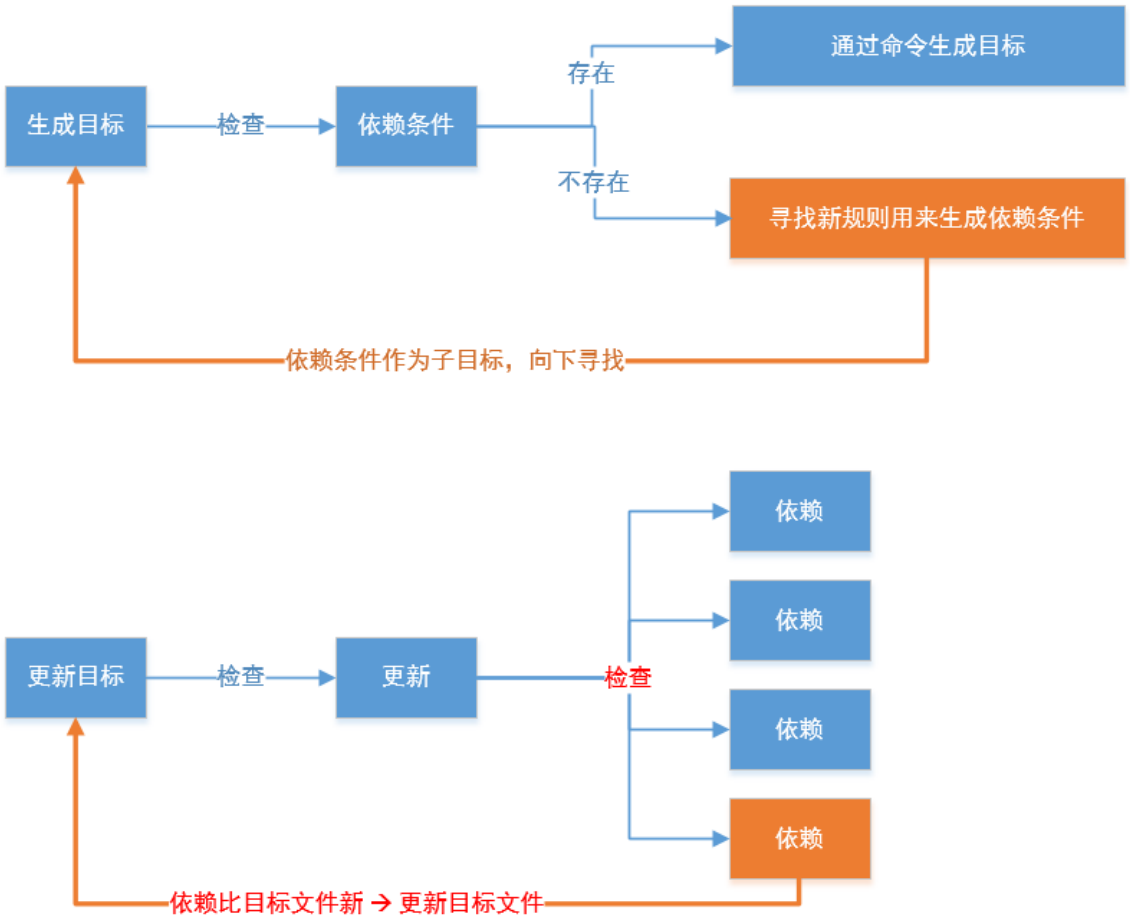
在上面的几步中，与 Makefile 关系最大的是第二步，那 Makefile 的好坏对于项目开发有些什么影响呢？设计得好的 Makefile，当我们重新编译时，只需编译那些上次编译成功后修改过的文件，也就是说编译的是一个 delta，而不是整个项目。反之，如果一个不好的 Makefile 环境，可能对于每一次的编译先要 clean，然后再重新编译整个项目。两种情况的差异是显然的，后者将耗费开发人员大量的时间用于编译，也就意味着低效率。对于小型项目，低效问题可能表现得并不明显，但对于规模相对大的项目，那就非常的明显了。开发人员可能一天做个十次编译（甚至更少）就没有时间用于编码和测试（调试）了。这就是为什么通常大型项目都会有一个专门维护 Makefile 的一个小团
队，来支撑产品的开发。

最为重要的是掌握二个概念，一个是目标（target），另一个就是依赖（dependency）。目标就是指要干什么，或说运行 make 后生成什么，而依赖是告诉 make 如何去做以实现目标。在 Makefile 中，目标和依赖是通过规则（rule）来表达的。我们最为熟悉的是采用make 来进行软件产品的代码编译，但它可以被用来做很多很多的事情，后面我们会给出一些不是用 make 来进行代码编译的例子。驾驭 Makefile，最为重要的是要学会**采用目标和依赖关系**来思考所需解决的问题。

Makefile的三要素

工作原理

工作原理



1.1 准备环境

我们需要一台 Linux 机器，或是在 Windows 上安装 Cgywin 来学习 Makefile。。为了验证 make 工具在你的环境中是否被正确的安装了，你可以运行“make –v”命令进行验证。图 1.1 是我在我的 Linux中运行“make –v”命令的输出结果，如果在你的环境中能看到相类似的 make 版本信息，那么说明make 在你的环境中是可用的，接下来就可以开始学习如何设计 Makefile 了。

```
$ make -v
GNU Make 4.2.1
Built for x86_64-pc-linux-gnu
Copyright (C) 1988-2016 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
```

```
7 There is NO WARRANTY, to the extent permitted by law.
```

图 1.1

1.2 规则

Hello World 几乎是所有编程语言进行语言讲解时采用的一个最为简单的例子，虽然 Makefile不是一个编程语言，但同样不防碍我们写一个在命令终端上输出“Hello World”的简单 Makefile。
采用一个文本编辑器编写一个如图 1.2 所示的 Makefile 文件，文件的存放目录可以是任意的。

```
1 all:
2     echo "Hello World"
```

图 1.2 Makefile 的“Hello World”实现

对于图 1.2 所示的 Makefile，需要提醒你注意的是 echo 前面必须只有 TAB（即你键盘上的 TAB键），且至少有一个 TAB，而不能用空格代替，这是我们需要学习的第一个 Makefile 语法。对于很多初学者，最为容易犯的就是这种“低级”错误。这种错误往往在对 Makefile 进行调试时，还不大容易发现，因为，从文本编辑器中看来，TAB 与空格有时没有太明显的区别。

我们说 Makefile 中第一个很重要的概念就是目标（target），图 1.2 所示的 Makefile 中的 all 就是我们的目标，目标放在‘：’的前面，其名字可以是由字母和下划线‘_’组成 。echo “Hello World”就是生成目标的命令，这些命令可以是任何你可以在你的环境中运行的命令以及 make 所定义的函数等等，后面我们会再细谈，现在只要知识 echo 是 BASH Shell 中的一个命令就行了，其功能是打印字符串到终端上。all 目标在这里就是代表我们希望在终端上打印出“Hello World”，有时目标会是一个比较抽象的概念。all 目标的定义，其实是定义了如何生成 all 目标，这我们也称之为规则，即我们说图 1.2 的 Makefile 中定义了一个生成 all 目标的规则。

我想你可能很急于看到这个 Makefile 的运行结果是什么，图 1.3 示例了三种不同的运行方式以及每种方式的运行结果。
第一种方式是只要在 Makefile 所在的目录下运行 make 命令，于是在终端上会输出二行，第一行实际上是我们在 Makefile 中所写的命令，而第二行则是运行命令的结果，你看到我们的 Makefile 确实在终端上打印了“Hello World”，真是太棒了！
第二种方式，则是运行“make all”命令，这告诉 make 工具，我要生成目标 all，其结果也不用多说了。
第三种方式则是运行 make test，指示 make 为我们生成 test 目标。由于我们根本没有定义 test 目标，所以运行结果是可想而知的，make 的确报告了不能找到 test 目标。
/Makefile/helloworld目录下

```
1 $make
```

```
2 echo "Hello World"
3 Hello World
4
5 $make all
6 echo "Hello World"
7 Hello World
8
9 $make test
10 make: *** No rule to make target `test'. Stop.
```

图 1.3 “Hello World”Makefile 的运行结果

现在，我们对图 1.2 的 Makefile 做一点小小的改动，如图 1.4 所示。其中的改动就是增加了 test规则用于构建 test 目标 —— 在终端上打印出“Just for test!”。

```
1 all:
2     echo "Hello World"
3 test:
4     echo "Just for test!"
```

图 1.4

图 1.5 是图 1.4 Makefile 的运行结果。从目前这两个 Makefile 的运行结果中我们学到了什么呢？我想有如下几点：

- 一个 Makefile 中可以定义多个目标。
- 调用 make 命令时，我们得告诉它我们的目标是什么，即要它干什么。当没有指明具体的目标是什么时，那么 make 以 Makefile 文件中定义的第一个目标作为这次运行的目标。这“第一个”目标也称之为默认目标（和是不是all没有关系）。
- 当 make 得到目标后，先找到定义目标的规则，然后运行规则中的命令来达到构建目标的目的。现在所示例的 Makefile 中，每一个规则中都只有一条命令，而实际的 Makefile，每一个规则可以包含很多条命令。

```
1 $make
2 echo "Hello World"
3 Hello World
4
5 $make test
6 echo "Just for test!"
```

```
Just for test!
```

图1.5

对于前面的示例我们看到当运行 make 时，在终端上还打印出了 Makefile 文件中的命令。有时，我们并不希望它这样，因为这样可能使得输出的信息看起来有些混乱。要使 make 不打印出命令，只要做一点小小的修改就行了，改过的 Makefile 如图 1.6 所示，就是在命令前加了一个‘@’。 这一符号告诉 make，在运行时不要将这一行命令显示出来。更改后相应的运行结果如图 1.7 所示。

```
all:
    @echo "Hello World"
test:
    @echo "Just for test!
```

图1.6

```
$make
Hello World
```

图 1.7

至此，再在图 1.4 Makefile 的基础上再做一点小小的改动，如图 1.8 所示。其中的改动之一是在各命令前增加了一个‘@’，之二则是在 all 目标之后的‘：’后加上了 test 目标。运行 make 和make test 命令的结果如图 1.9 所示。从输出结果中，你会发现当运行 make 时，test 目标好象也被构建了！
Makefile

```
all: test
    @echo "Hello World"
test:
    @echo "Just for test!"
```

图 1.8

```
$make
Just for test!
Hello World

```

```
$make test
Just for test!
```

图 1.9

这里需要引入 Makefile 中依赖关系的概念，图 1.8 中 all 目标后面的 test 是告诉 make，all 目标依赖 test 目标，这一依赖目标在 Makefile 中又被称之为先决条件。出现这种目标依赖关系时，make工具会按**从左到右的先后顺序先构建规则中所依赖的每一个目标。**如果希望构建 all 目标，那么make 会在构建它之前得先构建 test 目标，这就是为什么我们称之为**先决条件**的原因。图 1.10 采用UML 的类图表达了 all 目标的依赖关系。

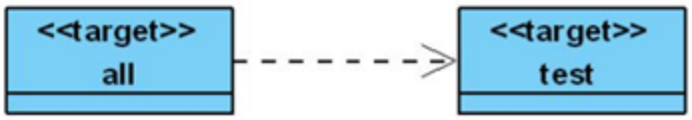


图 1.10 all 目标的依赖关系

至此，我们已经认识了 Makefile 中的细胞 —— 规则，图 1.13 是规则的 UML 语法表示，而图1.14 是其文字表示。一个规则是由**目标（targets）**、**先决条件（prerequisites）**以及**命令（commands）**所组成的。需要指出的是，目标和先决条件之间表达的就是依赖关系（dependency），这种依赖关系指明在构建目标之前，必须保证先决条件先满足（或构建）。而先决条件可以是其它的目标，当先决条件是目标时，其必须先被构建出来。还有就是一个规则中目标可以有多个，当存在多个目标，且这一规则是 Makefile 中的第一个规则时，如果我们运行 make 命令不带任何目标，那么规则中的第一个目标将被视为是缺省目标。图 1.11 是定义了两个目标的规则，而图 1.12 是其运行结果。

```
all test:
    @echo "Hello World"
```

图 1.11

```
$make
Hello World

$make test
Hello World
```

图 1.12

Makefile 说起来也很简单，因为其基本单元就是规则，不管多么复杂的 Makefile，都是用规则“码”出来的。当然，为了更高效的“码”出来，还得运用 Makefile 所提供的变量和函数等功能，这后面我们会慢慢

的讲到。

规则的功能就是指明 make 什么时候以及如何来为我们重新创建目标，在 Hello World 例子中，不论我们在什么时候运行 make 命令（带目标或是不带目标），其都会在终端上打印出信息来，和我们采用 make 进行代码编译时的表现好象有些不同。当采用 Makefile 来编译程序时，如果两次编译之间没有任何代码的改动，理论上说来，我们是不希望看到 make 会有什么动作的，只需说“目标是最新的”，而我们的最终目标也是希望构建出一个“聪明的”Makefile 的。与 Hello World 相比不同的是，采用 Makefile 来进行代码编译时，Makefile 中所存在的先决条件都是具体的程序文件，后面我们会看到。

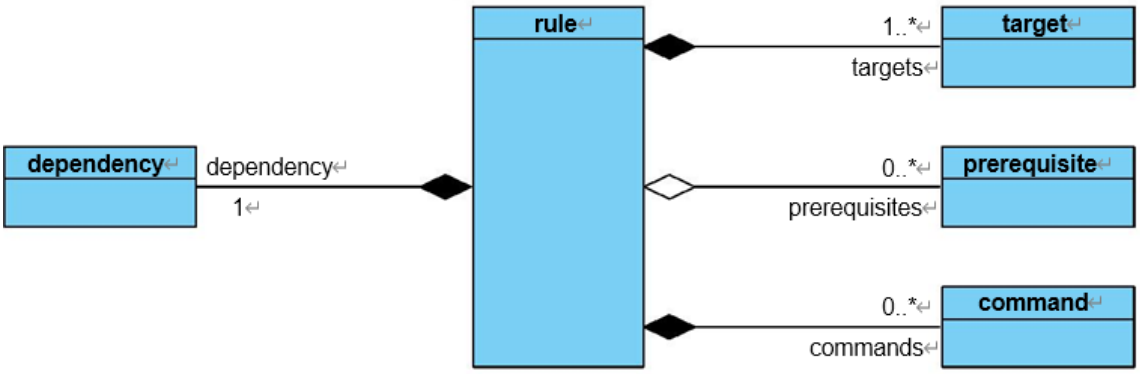


图 1.13 用 UML 表示的规则的语法

规则的语法

```
targets : prerequisites
    command
…
```

图 1.14 采用文字进行表达的规则的语法

对于图 1.8 中的 all 规则，我们说 all 是目标，test 则是 all 目标的依赖目标，而@echo “Hello World”则是用于生成 all 目标的命令。make 处理一个规则的活动图如图 1.15 所示，当中的构建依赖目标（build dependent target(s)）这一活动（注意是活动，而不是动作）就是重复图 1.15 所示的同样的活动，你可以看作是对图 1.15 活动图的递归调用。而运行命令构建目标（run command to build target）则是一个动作，是由命令所组成的动作。活动与动作的区别是，动作是只做一件事（但是可以有多个命令），而活动可以包括多个动作。

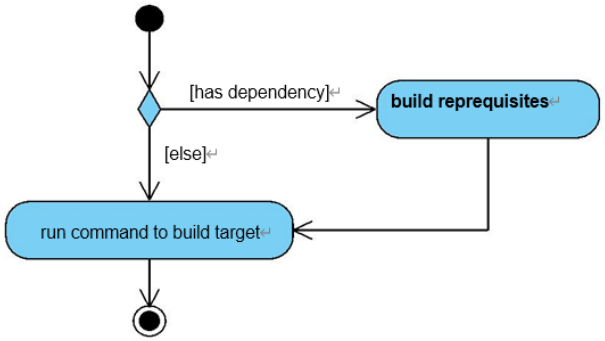


图 1.15 make 处理规则的活动图

为了更加深刻的理解图 1.15 的规则处理活动图，我们拿图 1.8 中的 Makefile 为例来说一说make 是如果处理 all 目标规则的。其处理活动图如图 1.16 所示，图中的左边是 all 规则的处理活动图，由于 all 规则有一个 test 依赖目标，所以其走的是[has dependency]分支，调用 build test taget活动，最后，运行 all 规则的 echo 命令。图的右边则是构建 test 目标的活动图，由于 test 目标没有依赖关系，所以走的是[else]分支。

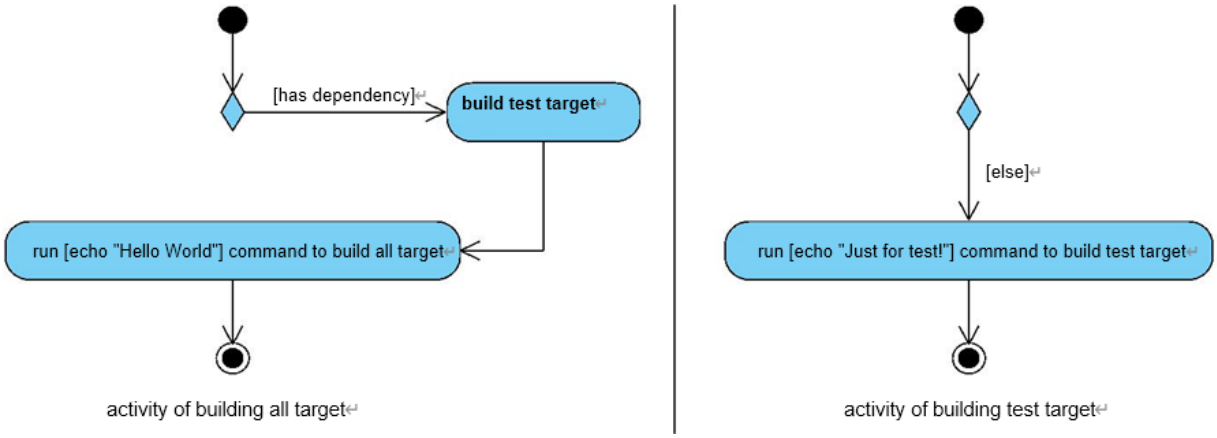


图 1.16 构建 all 目标的活动图

通过这一章节关于 Hello World 的几个例子，我们认识了 Makefile 中的规则。而规则中描述了目标是什么，先决条件是什么（即依赖关系），以及生成目标所需运行的命令是什么。对于规则需要特别注意的是，每一行命令之前必须用 TAB 键。当然，Hello World 的 Makefile 离我们现实工作中的 Makefile 还有很大的距离，其中的距离主要体现在功能性和可使用性上。为了让 Makefile 能更好的服务于我们的开发工作，我们还得学习 Makefile 中的其它的内容。无论如何，Hello World 是一个很好的开端！

1.3 原理

接下来我们试着将规则运用到程序编译当中去，下面我们假设有图 1.17 所示的用于创建 simple可执行文件的两个源程序文件，就假设我们是在做 simple 项目吧！现在，我们需要写一个用于创建simple 可执行程序的 Makefile 了，这个 Makefile 需要如何去写？还记得目标、依赖关系和命令吗？

此时的目录：Makefile/simple

foo.c

```c
#include <stdio.h>
void foo ()
{
    printf ("This is foo   ()!\n");
}
```

main.c

```c
extern void foo();
int main ()
{
    foo();
    return 0;
}
```

图 1.17 simple 项目的源程序文件

写一个 Makefile 文件的第一步不是一个猛子扎进去试着写一个规则，而是先用面向依赖关系的方法想清楚，所要写的 Makefile 需要表达什么样的依赖关系，这一点非常的重要。通过不断的练习，我们最终能达到很自然的运用依赖关系去思考问题。到那时，你在写 Makefile 时，头脑会非常的清楚自己在写什么，以及后面要写什么。现在抛开 Makefile，我们先看一看 simple 程序的依赖关系是什么。

我想是第一个跃入我们脑海中的依赖关系图，其中 simple 可执行文件显然是通过main.c 和 foo.c 最后编译并连接生成的。通过这个依赖图，其实我们就可以写出一个 Makefile 来了，这个任务交给读者你。这样的依赖关系所写出来的 Makefile，在现实中不是很可行，就是你得将所有的源程序都放在一行中让 GCC 为我们编译。如果是这样，那我们希望什么样的依赖关系呢？让我们想想看图 1.18 中缺了什么。目标文件？对了！

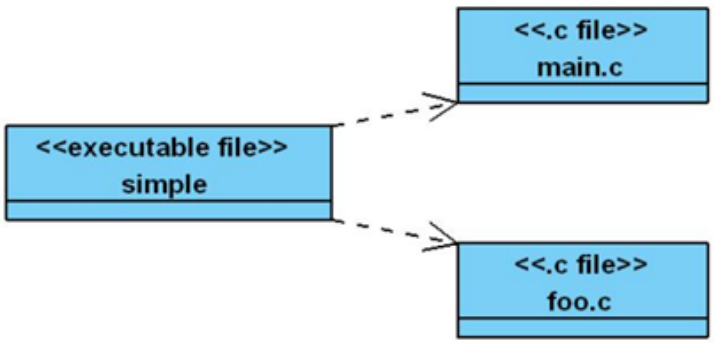


图 1.18

图 1.19 是 simple 程序的依赖关系更为精确的表达，其中我们加入了目标文件。对于 simple 可执行程序来说，图 1.19 表示的就是它的“依赖树”。接下来需要做的是将其中的每一个依赖关系，即其中的每一个带箭头的虚线，用 Makefile 中的规则来表示。

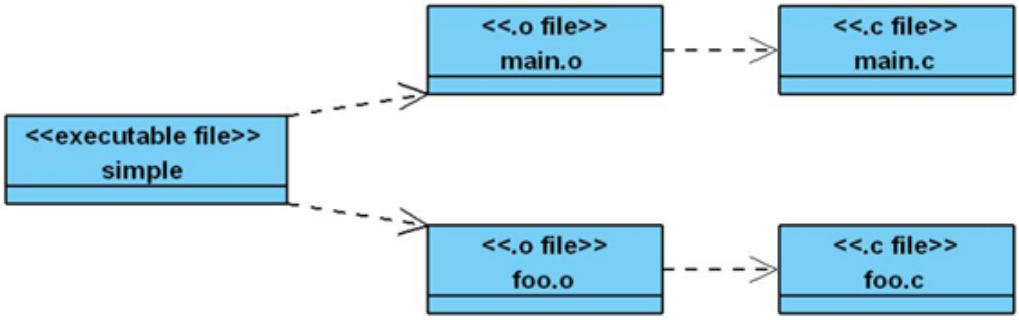


图 1.19 simple 程序的精细依赖关系

有了“依赖树”，写 Makefile 就会相对的轻松了。图 1.20 是图 1.19 所对应的 Makefile，而图 1.21则是依赖关系与规则的映射图。在这个 Makefile 中，我还增加了一个 clean 目标用于删除所生成的文件，包括目标文件和 simple 可执行程序，这在现实的项目中很是常见。

Makefile

```
all: main.o foo.o
    gcc –o simple main.o foo.o
main.o: main.c
    gcc –o main.o –c main.c
foo.o: foo.c
    gcc –o foo.o –c foo.c
clean:
    rm simple main.o foo.o
```

图 1.20

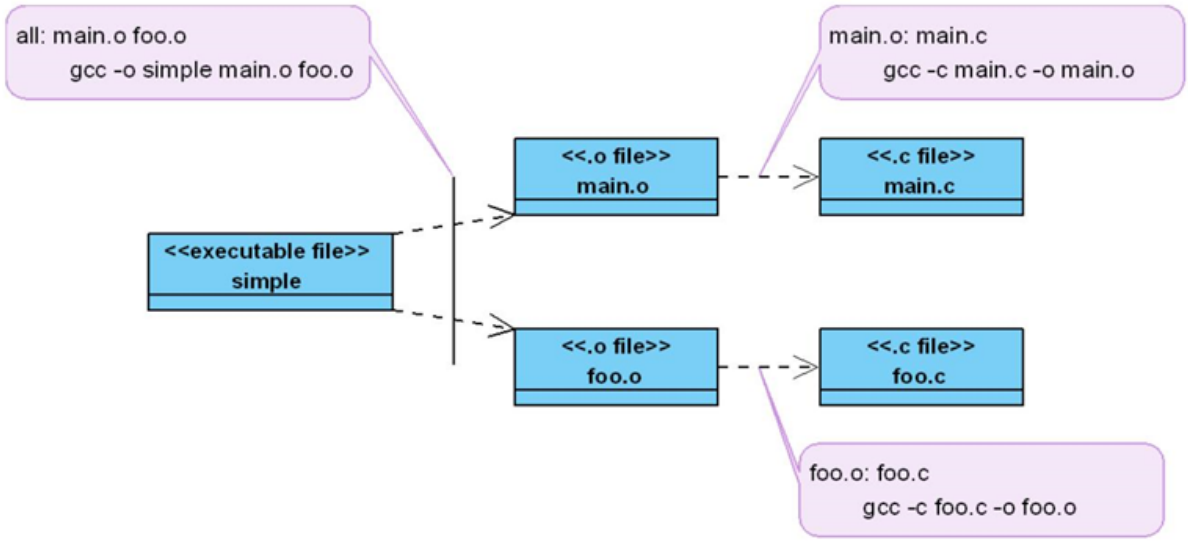


图 1.21

图 1.22 给出了 simple 程序的编译、执行以及清除的运行结果。就这么简单？当然！不过需要指出的是，这种方法与真正的大型项目所需要的还相差很远，慢慢来，我们已经迈出了很重要的一步！

```
$make
gcc -c main.c -o main.o
gcc -c foo.c -o foo.o
gcc -o simple main.o foo.o

$./simple
This is foo ()!

$make clean
rm simple main.o foo.o
```

图1.22

如果我们在不改变代码的清况下再编译会出现什么现象呢？图 1.23 给出了结果，注意到了第二次编译并没有构建目标文件的动作吗？但为什么有构建simple可执行程序的动作呢？为了明白为什么，我们需要了解 make 是如何决定哪些目标（这里是文件）是需要重新编译的。为什么 make会知道我们并没有改变 main.c 和 foo.c 呢？答案很简单，通过文件的时间戳！当 make 在运行一个规则时，我们前面已经提到了目标和先决条件之间的依赖关系，make 在检查一个规则时，采用的方法是：如果先决条件中相关的文件的时间戳大于目标的时间戳，即先决条件中的文件比目标更新，则知道有变化，那么需要运行规则当中的命令重新构建目标。这条规则会运用到所有与我们在 make时指定的目标的依赖树中的每一个规则。比如，对于 simple 项目，其依赖树中包括三个规则（如图1.21），make 会检查所有三个规则当中的目标（文件）与先决条件（文件）之间的时间先后关系，从而来决定是否要重新创建规则中的目标。

```
$make
gcc -c main.c -o main.o
gcc -c foo.c -o foo.o
gcc -o simple main.o foo.o

$make
gcc -o simple main.o foo.o
```

图1.23

知道了 make 是如何工作以后，我们不难想明白，为什么前面进行第二次 make 时，还会重新构建 simple 可执行文件，因为 simple 文件不存在。我们将 Makefile 做一点小小的改动，如图 1.24 所示。其最后的运行结果则如图 1.25所示。为什么还是和以前一样呢？哦，因为 Makefile 中的第一条规则中的

目标是 all，而 all 文件在我们的编译过程中并不生成，即 make 在第二次编译时找不到，所以又重新编译了一遍。

Makefile

```
all: main.o foo.o
    gcc –o simple main.o foo.o
main.o: main.c
    gcc –o main.o –c main.c
foo.o: foo.c
    gcc –o foo.o –c foo.c
clean:
    rm simple main.o foo.o
```

图 1.24

执行

```
$make
gcc –c main.c –o main.o
gcc –c foo.c –o foo.o
gcc –o simple main.o foo.o

$make
gcc –o simple main.o foo.o
```

图 1.25

再一次更改后的 Makefile 如图 1.26 所示，而图 1.27 是其最终的运行结果，它的确是发现了不需要进行第二次的编译。这正是我们所希望的！

Makefile

```
simple: main.o foo.o
    gcc –o simple main.o foo.o
main.o: main.c
    gcc –o main.o –c main.c
foo.o: foo.c
    gcc –o foo.o –c foo.c
clean:
    rm simple main.o foo.o
```

图 1.26

执行

```
$make
gcc –c main.c –o main.o
gcc –c foo.c –o foo.o
gcc –o simple main.o foo.o

$make
```

make: `simple' is up to date.

图 1.27

下面我们来验证一下如果对 foo.c 进行改动，是不是 make 能正确的发现并从新构建所需。对于make 工具，一个文件是否改动不是看文件大小，而是其时间戳。在我的环境中只需用 touch 命令来改变文件的时间戳就行了，这相当于模拟了对文件进行了一次编辑，而不需真正对其进行编辑。图1.28 列出了所有相关的命令操作，从最终的结果来看，make 发现了 foo.c 需要重新被编译，而这，最终也导致了 simple 需要重新被编译。

执行

```
$ls –l foo.c
–rw–r––r–– 1 Administrator None 70 Aug 14 07:38 foo.c

$touch foo.c

$ls –l foo.c
–rw–r––r–– 1 Administrator None 70 Aug 14 08:48 foo.c

$make
gcc –c foo.c –o foo.o
gcc –o simple main.o foo.o
```

图 1.28

至此，你完全明白了什么是目标的依赖关系以及 make 选择哪些目标需要重新编译的工作原理。掌握如果在头脑中勾画（当然初学时，可以用纸画一画）出我们想让 make 做的事的“依赖树”是编写 Makefile 最为重要和关键的一步。后面我们需要做的是让 Makefile 更加的简单但却更加的强大。

1.4 假目标

在前面的 sample 项目中，现在假设在程序所在的目录下面有一个 clean 文件，这个文件也可以通过 touch 命令来创建。创建以后，运行 make clean 命令，你会发现 make 总是提示 clean 文件是最新的，而不是按我们所期望的那样进行文件删除操作，如图 1.29 所示。从原理上我们还是可以理解的，这是因为 make 将 clean 当作文件，且在当前目录找到了这个文件，加上 clean 目标没有任何先决条件，所以，当我们要求 make 为我们构建 clean 目标时，它就会认为 clean 是最新的。

执行

```
$ls –l clean
ls: cannot access clean: No such file or directory

$touch clean

$ls –l clean
–rw–r––r–– 1 Administrator None 70 Aug 13:01 clean
```

```
$make clean
make: `clean' is up to date.
```

图 1.29

那对于这种情况，在现实中也难免存在所定义的目标与所存在的文件是同名的，采用 Makefile如何处理这种情况呢？Makefile 中的假目标（phony target）可以解决这个问题。假目标可以采用.PHONY 关键字来定义，需要注意的是其必须是大写字母。图 1.30 是将 clean 变为假目标后的Makefile，更改后运用 make clean 命令的结果如图 1.31 所示。

Makefile

```
.PHONY: clean
simple: main.o foo.o
    gcc –o simple main.o foo.o
main.o: main.c
    gcc –o main.o –c main.c
foo.o: foo.c
    gcc –o foo.o –c foo.c
clean:
    rm simple main.o foo.o
```

图 1.30

执行

```
$make clean
rm simple main.o foo.o
```

图 1.31

正如你所看到的，采用.PHONY 关键字声明一个目标后，make 并不会将其当作一个文件来处理，而只是当作一个概念上的目标。对于假目标，我们可以想像的是由于并不与文件关联，所以每一次 make 这个假目标时，其所在的规则中的命令都会被执行。

1.5 变量

只要是从事程序开发的，对于变量都很熟悉，因为每一个编程语言都有变量的概念。为了方便使用，Makefile 中也有变量的概念，我们可以在 Makefile 中通过使用变量来使得它更简洁、更具可维护性。下面，我们来看一看如何通过使用变量来提高 simple 项目 Makefile 的可维护性，图 1.32是运用变量的第一个 Makefile。

Makefile

```
.PHONY: clean
CC = gcc
RM = rm
EXE = simple
OBJS = main.o foo.o
$(EXE): $(OBJS)
    $(CC) –o $(EXE) $(OBJS)
main.o: main.c
```

```
	$(CC) -o main.o -c main.c
foo.o: foo.c
	$(CC) -o foo.o -c foo.c
clean:
	$(RM) $(EXE) $(OBJS)
```

图 1.32

从图 1.32 可以看出，一个变量的定义很简单，就是一个名字（变量名）后面跟上一个等号，然后在等号的后面放这个变量所期望的值。对于变量的引用，则需要采用$(变量名)或者${变量名}这种模式。在这个 Makefile 中，我们引入了 CC 和 RM 两个变量，一个用于保存编译器名，而另一个用于指示删除文件的命令是什么。还有就是引入了 EXE 和 OBJS 两个变量，一个用于存放可执行文件名，可另一个则用于放置所有的目标文件名。采用变量的话，当我们需要更改编译器时，只需更改变量赋值的地方，非常方便，如果不采用变量，那我们得更改每一个使用编译器的地方，很是麻烦。显然，变量的引入增加了 Makefile 的可维护性。你可能会问，既然定义了一个 CC 变量，那是不是要将-o 或是-c 命令参数也定义成为一个变量呢？好主意！的确，如果我们更改了一个编译器，那么很有可能其使用参数也得跟着改变。现在，我们不急着这么去做，为什么？因为后面我们还会对 Makefile 进行简化，到时再改变也来得及，现在我们只是将注意力放在变量的使用上。

1.5.1 自动变量

对于每一个规则，不知你是否觉得目标和先决条件的名字会在规则的命令中多次出，每一次出现都是一种麻烦，更为麻烦的是，如果改变了目标或是依赖的名，那得在命令中全部跟着改。有没有简化这种更改的方法呢？这我们需要用到 Makefile 中的自动变量，它们包括：

- $@用于表示一个规则中的目标。当我们的一个规则中有多个目标时，$@所指的是其中任**何造成命令被运行的目标。**
- $^则表示的是规则中的所有先择条件。
- $<表示的是规则中的第一个先决条件。

除了上面的两个自动变量，在 Makefile 中还有其它的动变量，我们在需要的时候再提及，就simple 项目的 Makefile 而言，为了简化它，采用这这些变量就足够了。图 1.33 是用于测试上面三个自动变量的值的 Makefile，其运行结果从图 1.34 中可以找到。需要注意的是，在 Makefile 中‘$’具有特殊的意思，因此，如果想采用 echo 输出‘$’，则必需用两个连着的‘$’。还有就是，$@对于 Shell 也有特殊的意思，我们需要在“$$@”之前再加一个脱字符‘\’。如果你还有困惑，你可以通过改一改 Makefile 来验证它。图 1.34 的最后一行是一个只有目标的规则，如果去除它会出现什么问题呢？试试看！

Makefile

```
.PHONY: all
all: first second third
	@echo "\$$@ = $@"
	@echo "$$^ = $^"
	@echo "$$< = $<"
first second third:
```

图 1.33

执行

```
$make
$@ = all
$^ = first second third
$< = first
```

图 1.34

采用自动变量后 simple 项目的 Makefile 可以被重写为如图 1.35 所示，用了自动变量以后这个Makefile 看起来有点怪怪的，有些什么‘^’、‘@’，等等。这就对了，你所看到的 Makefile 看起来不都很奇怪吗？我们要的就是这个“味”！

Makefile

```
.PHONY: clean
CC = gcc
RM = rm
EXE = simple
OBJS = main.o foo.o
$(EXE): $(OBJS)
	$(CC) -o $@ $^
main.o: main.c
	$(CC) -o $@ -c $^
foo.o: foo.c
	$(CC) -o $@ -c $^
clean:
	$(RM) $(EXE) $(OBJS)
```

图 1.35 使用自动变量

自动变量在对它们还不熟悉时，看起来可能有那么一点吃力，但熟悉了你就会觉得其简捷（洁），那时也会觉得它们好用。这有点像我们用惯了 Windows 操作系统，刚开始用 Linux 时很不适应，比如在 Linux 中，ls 表示的是查看目录的文件，这名字就是有那么一点怪。但当我们对于 Linux 熟悉了，你会发现在 Linux 平台上工作，非常的自由和方便，远比 Windows 上灵活。从这个问题来看，设计是有一个平衡点的，Windows 容易上手，但它把用户当作“傻子”，对于高级用户来说却不方便；而 Linux 则更难上手，但一旦上手后，功能却更强大。

至此，你有没有觉得我们的 Makefile 更加的酷了呢？当然，我们写 Makefile 的目的不是为了让其酷得不能理解，相反是为了更简单和易于维护，这里的酷是指其看起来更专业。

1.5.2 特殊变量

在 Makefile 中有几个特殊变量，我们可能经常需要用到。第一个就是 MAKE 变量，它表示的是make 命令名是什么。当我们需要在 Makefile 中调用另一个 Makefile 时需要用到这个变量，采用这种方式，有利于写一个容易移植的 Makefile。图 1.36 是对 MAKE 变量进行测试的 Makefile，而图1.37 则是其测试结果。

Makefile

```
.PHONY: all
all:
```

```
    @echo "MAKE = $(MAKE)"
```

图 1.36

执行

```
$make
MAKE = make
```

图 1.37

第二个特殊变量则是 MAKECMDGOALS，它表示的是当前用户所输入的 make 目标是什么。图 1.38是用于对其进行测试的 Makefile

Makefile

```
.PHONY: all clean
all clean:
    @echo "\$$@ = $@"
    @echo "MAKECMDGOALS = $(MAKECMDGOALS)"
```

图 1.38

执行

```
$make all
$@ = all
MAKECMDGOALS = all

$make clean
$@ = clean
MAKECMDGOALS = clean

$make all clean
$@ = all
MAKECMDGOALS = all clean
$@ = clean
MAKECMDGOALS = all clean
```

图 1.39 显示了其测试结果。

执行

```
$make
$@ = all
MAKECMDGOALS =
```

```
$make all
$@ = all
MAKECMDGOALS = all

$make clean
$@ = clean
MAKECMDGOALS = clean

$make all clean
$@ = all
MAKECMDGOALS = all clean
$@ = clean
MAKECMDGOALS = all clean
```

图 1.39

从测试结果看来，MAKECMDGOALS 指的是用户输入的目标，当我们只运行 make 命令时，虽然根据 Makefile 的语法，第一个目标将成为缺省目标，即 all 目标，但 MAKECMDGOALS 仍然是空，而不是 all，这一点我们需要注意。

1.5.3 变量的类别

图 1.32 示例了使用等号进行变量定义和赋值，对于这种只用一个**“=”**符号定义的变量，我们称之为**递归扩展变量**（recursively expanded variable）。现在我们需要了解一些细节，先看一看图1.40 所示的 Makefile 及其运行结果（如图 1.41 所示）。

Makefile

```
.PHONY: all
foo = $(bar)
bar = $(ugh)
ugh = Huh?
all:
    @echo $(foo)
```

图 1.40

执行

```
$make
Huh?
```

图 1.41

从结果来看，递归扩展变量的引用是递归的。这种递归性有利也有弊。对于利，如图 1.42 所示的 Makefile，最后 CFLAGS 将会被展开为“-Ifoo -Ibar -O”。但这也存在弊，那就是我们不能对CFLAGS 变量再采用赋值操作。也就是说图 1.43 中的 CFLAGS 会出现一个死循环。

Makefile

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

```
include_dirs = –lfoo –lbar
```

图 1.42

Makefile

```
CFLAGS = $(CFLAGS) –O
```

图 1.43

除了递归扩展变量还有一种变量称之为简单扩展变量（simply expanded variables），是用“:=”操作符来定义的。对于这种变量，make 只对其进行一次扫描和替换，请看图 1.44 所示的 Makefile及图 1.45 所对应的运行结果。

Makefile

```
.PHONY: all
x = foo
y = $(x) b
x = later
xx := foo
yy := $(xx) b
xx := later
all:
    @echo "x = $(y), xx = $(yy)"
```

图 1.44

执行

```
$make
x = later b, xx = foo b
```

图 1.45

从图 1.45 可以明显的看出 make 是如何处理递归扩展变量和简单扩展变量的。最后，Makefile中还存在一种条件赋值符“?=”，图 1.46 和图 1.47 分别是运用条件赋值的 Makefile 和运行结果。

Makefile

```
.PHONY: all
foo = x
foo ?= y
bar ?= y
all:
    @echo "foo = $(foo), bar = $(bar)"
```

图 1.46

执行

```
$make
foo = x, bar = y
```

图 1.47

从运行结果来看，条件赋值的意思是当变量以前没有定义时，就定义它并且将左边的值赋值给它，如果已经定义了那么就不再改变其值。条件赋值类似于提供了给变量赋缺省值的功能。

对于前面所说的变量类别，是针对一个赋值操作而言的，图 1.48 示例了对于同一样变量采用不同的赋值操作的 Makefile，从图 1.49 的运行结果可以验证我们可以对同一个变量采用不同的赋值操作。

Makefile

```
.PHONY: all
objects = main.o foo.o bar.o utils.o
objects := $(objects) another.o
all:
	@echo $(objects)
```

图 1.48

执行

```
$make
main.o foo.o bar.o utils.o another.o
```

图 1.49

1.5.4 变量及其值的来源

变量可以从哪来呢？从前面的示例可以看出，在 Makefile 中我们可以对变量进行定义。此外，还有其它的地方让 Makefile 获得变量及其值。比如：

- 对于前面所说到的自动变量，其值是在每一个规则中根据规则的上下文自动获得变量值的。
- 可以在运行 make 时，在 make 命令行上定义一个或多个变量。对于图 1.46 所示的Makefile，如果采用 make bar=x 运行 Makefile，则得到的结果将完全不同，如图 1.50 所示。从结果可以看出，在 make 命令行中定义的变量及其值同样在 Makefile 中是可见的。其实，我们可以通过在 make 命令行中定义变量的方式从而覆盖 Makefile 中所定义的变量的值，图 1.51 示例了对图 1.46 中的 Makefile 中的变量值进行覆盖操作的结果。
- 变量还可以来自于 Shell 环境，图 1.52 示例了采用 Shell 中的 export 命令定义了一个 bar变量后 Makefile 的运行结果。

执行

```
$make bar=x
foo = x, bar = x
```

图 1.50

执行

```
$make foo=haha
foo = haha, bar = x
```

图 1.51

```
$make
foo = x, bar = y

$export bar=x
```

```
$make
foo = x, bar = x
```

图 1.52

接下来，我们看一看在 Makefile 还可以如何对变量赋值，图 1.53 示例了采用“+=”操作符对变量进行赋值的方法，而其输出结果与图 1.49是完全一样的。此外，图 1.54中的 Makefile与图 1.53中的是等价的。

Makefile

```
.PHONY: all
objects = main.o foo.o bar.o utils.o
objects += another.o
all:
    @echo $(objects)
```

图 1.53

Makefile

```
.PHONY: all
objects = main.o foo.o bar.o utils.o
objects := $(objects) another.o
all:
    @echo $(objects)
```

图 1.54

1.5.5 高级变量引用功能

图 1.55 的 Makefile 示例了变量引用的一种高级功能，即在赋值的同时完成后缀替换操作。从图 1.56 的结果来看，bar 变量中的文件名从.o 后缀都变成了.c。这种功能也可以采用后面我们将要说的 patsubst 函数来实现，与函数相比，这种功能更加的简洁。当然，patsubst 功能更强，而不只是用于替换后缀。

Makefile

```
.PHONY: all
foo = a.o b.o c.o
bar := $(foo:.o=.c)
all:
    @echo "bar = $(bar)"
```

图 1.55

执行

```
$make
bar = a.c b.c c.c
```

图 1.56

1.5.6 override 指令

前面我们说了，我们可以采用在 make 命令行上定义变量的方式，使得 Makefile 中定义的变量覆盖掉，从而不起作用。可能，在设计 Makefile 时，我们并不希望用户将我们在 Makefile 中定义的某个变量覆盖

掉，那就得用 override 指令了。图 1.57 和图 1.58 分别是使用了 override 指令的Makefile 及其运行结果。你可以对比前面的图 1.48 和图 1.51 来理解 override 指令的作用。

Makefile

```
.PHONY: all
override foo = x
all:
    @echo "foo = $(foo)"
```

图 1.57

执行

```
$make foo=haha
foo = x
```

图 1.58

1.6 模式

对于前面的 Makefile，其中存在多个规则用于构建目标文件。比如，main.o 和 foo.o 都是采用不同的规则进行描述的。我相信你也会觉得，如果对于每一个目标文件都得写一个不同的规则来描述，那会是一种“体力活”，太繁了！对于一个大型项目，就更不用说了。Makefile 中的模式就是用来解决我们的这种烦恼的，先看图 1.59 所示的运用了模式的 simple 项目的 Makefile。

Makefile

```
.PHONY: clean
CC = gcc
RM = rm
EXE = simple
OBJS = main.o foo.o
$(EXE): $(OBJS)
    $(CC) –o $@ $^
%.o: %.c
    $(CC) –o $@ –c $^
clean:
    $(RM) $(EXE) $(OBJS)
```

图 1.59

与 simple 项目前一版本的 Makefile 相比，最为直观的改变就是从二条构建目标文件的规则变成了一条。模式类似于我们在 Windows 操作系统中所使用的通配符，当然是**用“%”**而不是“*”。采用了模式以后，不论有多少个源文件要编译，我们都是应用同一个模式规则的，很显然，这大大的简化了我们的工作。使用了模式规则以后，你同样可以用这个 Makefile 来编译或是清除 simple 项目，这与前一版本在功能上是完全一样的。

1.7 函数

函数是 Makefile 中的另一个利器，现在我们看一看采用函数如何来简化 simple 项目的 Makefile。对于 simple 项目的 Makefile，尽管我们使用了模式规则，但还有一件比较恼人的事，我们得在这个Makefile 中指明每一个需要被编译的源程序。对于一个源程序文件比较多的项目，如果每增加或是删除一个文件都得更新 Makefile，其工作量也不可小视！

图 1.60 是采用了 **wildcard** 和 **patsubst** 两个函数后 simple 项目的 Makefile。你可以先用它来编译一下 simple 项目以验证其功能性。需要注意的是函数的语法形式很是特别，不过再特别也只是一种语法，对于我们来说只要记住其形式就行了。

Makefile

```
.PHONY: clean
CC = gcc
RM = rm
EXE = simple
SRCS = $(wildcard *.c)
OBJS = $(patsubst %.c,%.o,$(SRCS))   –》把.c 替换为.o
$(EXE): $(OBJS)
$(CC) –o $@ $^
%.o: %.c
    $(CC) –o $@ –c $^
clean:
    $(RM) $(EXE) $(OBJS)
```

图 1.60

现在，我们来模拟增加一个源文件的情形，看一看如果我们增加一个文件，在 Makefile 不做任何更改的情况下其是否仍能正常的工作。增加文件的方式仍然是采用 touch 命令，通过 touch 命令生成一个内容是空的 bar.c 源文件，然后再运行 make 和 make clean，其结果示于图 1.61。

执行

```
$touch bar.c

$make
gcc –o bar.o –c bar.c
gcc –o foo.o –c foo.c
gcc –o main.o –c main.c
gcc –o simple bar.o foo.o main.o

$make clean
rm simple bar.o foo.o main.o
```

图 1.61

从结果来看函数真的起作用了！这功能真的是酷！接下来，我们看一看几个函数的用法。需要提及的是，这里并不找算将 Makefile 中所有能用的函数都列出，而只是列出几个这篇文章中我们会用到的。当然，图 1.60 中的 wildcard 和 patsubst 函数一定列在其中。建议你看一看《GUN make》以了解 Makefile 中到底有些什么函数，这样的话，当我们在碰到具体的问题时就会想到它们。

1.7.1 addprefix 函数

addprefix 函数是用来在给字符串中的每个子串前加上一个前缀，其形式是：

$(addprefix prefix, names...)

图 1.62 示例了它的用法，addprefix 函数的最终行为可以从图 1.63 中看出。

Makefile

```
.PHONY: all
without_dir = foo.c bar.c main.o
with_dir := $( addprefix objs/, $(without_dir))
all:
@echo $(with_dir)
```

图 1.62

执行

```
$make
objs/foo.c objs/bar.c objs/main.o
```

图 1.63

1.7.2 filter

filter 函数用于从一个字符串中，根据模式得到满足模式的字符串，其形式是：

$(filter pattern..., text)

图 1.64 示例了它的用法，图 1.65 则是其运行结果。从结果来看，经过 filter 函数的调用以后，source变量中只存在.c 文件和.s 文件了，而.h 文件则则被过滤掉了。

Makefile

```
.PHONY: all
sources = foo.c bar.c baz.s ugh.h
sources := $(filter %.c %.s, $(sources))
all:
    @echo $(sources)
```

图 1.64

执行

```
$make
foo.c bar.c baz.s
```

图 1.65

1.7.3 filter-out

filter-out 函数用于从一个字符串中根据模式滤除一部分字符串，其形式是：

$(filter-out pattern..., text)

图 1.66 示例了它的用法，图 1.67 则是其运行结果。从结果来看，filter-out 函数将 main1.o 和 main2.o
从 objects 变量中给滤除了。filter 与 filter-out 是互补的。

Makefile

```
.PHONY: all
objects = main1.o foo.o main2.o bar.o
result = $(filter-out main%.o, $(objects))
all:
    @echo $(result)
```

图 1.66

执行

```
$make
foo.o bar.o
```

图 1.67

1.7.4 patsubst 函数

patsubst 函数是用来进行字符串替换的，其形式是：

$(patsubst pattern, replacement, text)

图 1.68 示例了它的用法，从图中可以看出 mixed 变量中包括了.c 文件也包括了.o 文件，采用patsubst 函数进行字符串替换时，我们希望将所有的.c 文件都替换成.o 文件。图 1.69 是最后的运行结果。这里示例的功能与我们在 1.5.5 节中所讲的变量的高级引用功能是一样的。当然，由于patsubst 函数可以使用模式，所以其也可以用于替换前缀等等，功能更加的强。

Makefile

```
.PHONY: all
mixed = foo.c bar.c main.o
objects := $(patsubst %.c, %.o, $(mixed))
all:
    @echo $(objects)
```

图 1.68

执行

```
$make
foo.o bar.o main.o
```

图 1.69

1.7.5 strip

strip 函数用于去除变量中的多余的空格，其形式是：

$(strip string)

图 1.70 示例了它的用法，图 1.71 则是其运行结果。从结果来看，strip 函数将 foo.c 和 bar.c 之间的多余的空格给去除了。

Makefile

```
.PHONY: all
original = foo.c   bar.c
stripped := $(strip $(original))
all:
    @echo "original = $(original)"
    @echo "stripped = $(stripped)"
```

图 1.70

执行

```
$make
original = foo.c   bar.c
stripped = foo.c bar.c
```

图 1.71

1.7.6 wildcard 函数

wildcard 是通配符函数，通过它可以得到我们所需的文件，这个函数如果我们在 Windows 或是Linux 命令行中的“*”。其形式是：

$(wildcard pattern)

图 1.72 示例了如何从当前 Makefile 所在的目录下通过 wildcard 函数得到所有的 C 程序源文件。图1.73 则显示了最后的运行结果。

Makefile

```
.PHONY: all
SRCS = $(wildcard *.c)
all:
    @echo $(SRCS)
```

图 1.72

执行

```
$make
bar.c foo.c main.c
```

图 1.73

1.8 小结

至此，我们已经掌握了 Makefile 中的不少概念，simple 项目的 Makefile 也随着我们的学习变得更加的简洁、易用和强大。但是正如 simple 这个程序的名字那样，毕竟这是一个简单的项目，现实情况中的项目却更加的复杂。比如通常会将目标文件放入一个 objs 的子目录中，而可执行文件放入一个 exes 的子目录，而不是直接将这些文件都放在与源程序相同的一个目录中。还有，我们的 simple项目只有两个源程序文件，即使加上后来生成的 bar.c 空文件，也就只有四个。为了简单，我们除了包含了 stdio.h 头文件外，其它的头文件一概没有用。实际上，如果将头文件等复杂的因素合在一起，你会发现 simple 项目的 Makefile 还有很多地方需要提高。接下来，我们将通过做一个复杂项目的Makefile 来学习更多的知识。让我们给这个将要做的复杂项目取一个与 simple 项目相对应的名字 —
— complicated。complicated 项目是我们下一个章节的内容。

2 提高

在进行 complicated 项目之前，我们需要了解对于它的 Makefile 的一些基本需求，这些基本需求是：

- 将所有的目标文件放入源程序所在目录的 objs 子目录中。
- 将所有最终生成的可执行程序放入源程序所在目录的 exes 子目录中。
- 将引入用户头文件来模拟复杂项目的情形。

现在，就让我们开始吧！

2.1 创建目录

毫无疑问，我们在编译项目之前希望用于存放文件的目录先准备好，当然，我们可以在编译之前通过手动来创建所需的目录，但这里我们希望采用自动的方式。虽然 complicated 项目现在还没有源文件，但这并不妨碍我们先写一个只有目录创建功能的 Makefile。下面我们想一想目录创建Makefile 的“依赖树”长得是什么样子。

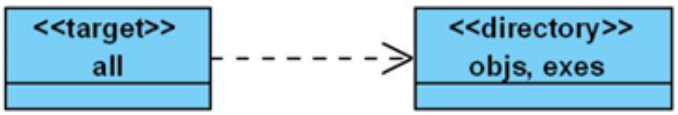


图 2.1

我所想到的目录依赖关系图如图 2.1 所示，你的也是这样吗？但是这个依赖图有一点问题，从概念上说来是对的，但从 Makefile 的实现上存在一些问题。我们说 all 是一个目标，如果 all 直接依赖 objs 和 exes 目录的话，那如何创建目录呢？不管如何先写一个 Makefile 吧，如图 2.2 所示，其运行结果如图 2.3 所示。在这个 Makefile 中我们定义了两个变量，一个是 MKDIR，另一个则是用于存放目录名的变量 DIRS。从结果来看，验证了我所提出的问题，即，目录如何创建呢？

Makefile

```
.PHONY: all
MKDIR = mkdir
DIRS = objs exes
all: $(DIRS)
```

图 2.2

执行

```
$make
make: *** No rule to make target `objs', needed by `all'. Stop.
```

图 2.3

改进后的依赖关系图如图 2.4 所示，从图中你可以看出 objs 和 exes 即是目标又是目录，而实现这一依赖关系图的 Makefile 如图 2.5 所示，同样我们将其运行结果也列了出来，如图 2.6 所示。

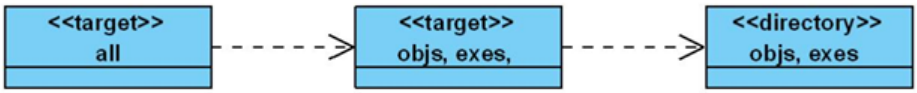


图 2.4

Makefile

```
.PHONY: all
MKDIR = mkdir
DIRS = objs exes
all: $(DIRS)
$(DIRS):
    $(MKDIR) $@
```

图 2.5

执行

```
$make
mkdir objs
mkdir exes

$ls
Makefile exes objs

$make
make: Nothing to be done for `all'.
```

图 2.6

在这个 Makefile 中，我们需要注意的是 OBJS 变量即是一个依赖目标也是一个目录，在不同的场合其意思是不同的。比如，第一次 make 时，由于 objs 和 exes 目录都不存在，所以 all 目标将它们视作是一个先决条件或者说是依赖目标，接着 Makefile 先根据目录构建规则构建 objs 和 exes 目标，即 Makefile 中的第二条规则就被派上了用场。构建目录时，第二条规则中的命令被执行，即真正的创建了 objs 和 exes 目录。当我们第二次进行 make 时，此时，make 仍以 objs 和 exes 为目标，但从目录构建规则中发现，这两个目标并没有依赖关系，而且能从当前目录中找到 objs 和 exes 目录，即认为 objs 和 exes 目标都是最新的，所以不用再运行目录构建规则中的命令来创建目录。图 2.7示例了 Makefile 与“依赖树”之间的映射关系。

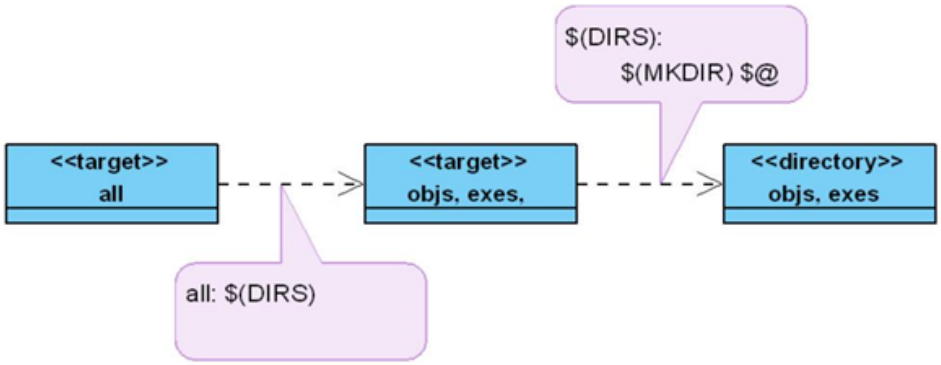


图 2.7

接下来我们也得为我们的 Makefile 创建一个 clean 目标，专门用来删除所生成的目标文件和可执行文件。加 clean 规则还是相当的直观的，如图 2.8 所示，其中我们又增加了两个变量，一个是RM，另一个则是 RMFLAGS，这与 simple 项目中所使用的方法是一样的。运行 make clean 命令的结果如图 2.9 所示。

Makefile

```
.PHONY: all clean
MKDIR = mkdir
RM = rm
RMFLAGS = –fr
DIRS = objs exes
all: $(DIRS)
$(DIRS):
    $(MKDIR) $@
clean:
    $(RM) $(RMFLAGS) $(DIRS)
```

图 2.8

执行

```
$make clean
rm –fr objs exes

$ls
Makefile
```

图 2.9

2.2 增加头文件

好了，目录的创建已经好了，接下来我们需要为我们的 complicated 项目在 simple 项目的基础之上增加头文件。图 2.10 是我们现在 complicated 项目的源程序文件。接下来要做的是，在complicated 项目的 Makefile 中加入对于源程序进行编译的部分，如图 2.11 所示。

foo.h

```
#ifndef __FOO_H
```

```
#define __FOO_H
void foo ();
#endif
```

foo.c

```
#include <stdio.h>
#include “foo.h”
void foo ()
{
    printf (“This is foo ()!\n”);
}
```

main.c

```
#include “foo.h”
int main ()
{
    foo ();
    return 0;
}
```

图 2.10

Makefile

```
.PHONY: all clean
MKDIR = mkdir
RM = rm
RMFLAGS = -fr
CC = gcc
EXE = complicated
DIRS = objs exes
SRCS = $(wildcard *.c)
OBJS = $(SRCS:.c=.o)
all: $(DIRS) $(EXE)
$(DIRS):
    $(MKDIR) $@
$(EXE): $(OBJS)
    $(CC) -o $@ $^
%.o: %.c
    $(CC) -o $@ -c $^
clean:
    $(RM) $(RMFLAGS) $(DIRS) $(EXE) $(OBJS)
```

图 2.11

这次的改动不少内容与 simple 项目是一样的，其中很重要的一个变化是，我们在 all 目标的后面再增加了对 EXE 目标的依赖。当一个规则中出来了多个先决条件时（这里的 all 规则就是），make会以从左到右

的顺序来一个一个的构建目标。make 相关的操作结果如图 2.12 所示，从图中你会发现，所有的目标文件以及可执行文件都放在当前目录下，而并没有如我们所希望那样放到 objs 和exes 目录中去。

图2.10和图2.11的内容合并到2.11目录。

执行

```
$make
mkdir objs
mkdir exes
cc –o foo.o –c foo.c
cc –o main.o –c main.c
cc –o complicated foo.o main.o

$ls
Makefile complicated exes foo.c foo.h foo.o main.c main.o objs

$./complicated
This is foo ()!

$make clean
rm –fr objs exes complicated foo.o main.o

$ls
Makefile foo.c foo.h main.c
```

图 2.12

2.3 将文件放入目录

为了将目标文件或是可执行程序分别放入所创建的 objs 和 exes 目录中，我们需要用到 Makefile中的一个函数 —— addprefix（参见 1.7.1 ）。现在，我们需要对 Makefile 进行一定的修改，以使目标文件都放入 objs 目录当中，更改后的 Makefile 如图 2.13 所示。

Makefile

```
.PHONY: all clean
MKDIR = mkdir
RM = rm
RMFLAGS = –fr
CC = gcc

DIR_OBJS = objs
DIR_EXES = exes
DIRS = $(DIR_OBJS) $(DIR_EXES)
```

```
EXE = complicated
SRCS = $(wildcard *.c)
OBJS = $(SRCS:.c=.o)
OBJS := $(addprefix $(DIR_OBJS)/, $(OBJS))

all: $(DIRS) $(EXE)

$(DIRS):
    $(MKDIR) $@
$(EXE): $(OBJS)
    $(CC) -o $@ $^
$(DIR_OBJS)/%.o: %.c
    $(CC) -o $@ -c $^
clean:
    $(RM) $(RMFLAGS) $(DIRS) $(EXE) ~~$(OBJS)~~
```

图 2.13

图 2.13 中最大的变化除了增加了对于 addprefix 函数的运用为每一个目标文件加上“objs/”前缀外，还有一个很大的变化是，我们需要在构建目标文件的模式规则中的目标前也加上“objs/”前缀，即增加“$(DIR_OBJS)/”前缀。之所以要加上，是因为规则的命令中的-o 选项需要以它作为目标文件的最终生成位置，还有就是因为 OBJS 也加上了前缀，而要使得 Makefile 中的目标创建规则被运用，也需要采用相类似的格式，即前面有“objs/”。此外，由于改动后的 Makefile 会将所有的目标文件放入 objs 目录当中，而我们的 clean 规则中的命令包含将 objs 目录删除的操作，所以我们可以去除命令中对 OBJS 中文件的删除。这导致的改动就是 Makefile 中的最后一行中删除了$(OBJS)。后面我们都采用在内容上加删除线的方式来表示这一内容需要被删除。更改以后的运行结果可以从图 2.14 看出，从其编译过程你可以看到，所生成的目标文件的确是放入了 objs 子目录。

执行

```
$make
mkdir objs
mkdir exes
cc -o objs/foo.o -c foo.c
cc -o objs/main.o -c main.c
cc -o complicated objs/foo.o objs/main.o

$ls
Makefile complicated exes foo.c foo.h main.c objs
```

图 2.14

采用同样的方法，**我们也可以将 complicated 放入到 exes 目录当中去**，且改动也是非常的小，更改后的 Makefile 如图 2.15 所示，图 2.16 则是这一改动后的运行结果。

Makefile

```
.PHONY: all clean
```

```
MKDIR = mkdir
RM = rm
RMFLAGS = -fr
CC = gcc
DIR_OBJS = objs
DIR_EXES = exes
DIRS = $(DIR_OBJS) $(DIR_EXES)
EXE = complicated
EXE := $(addprefix $(DIR_EXES)/, $(EXE))
SRCS = $(wildcard *.c)
OBJS = $(SRCS:.c=.o)
OBJS := $(addprefix $(DIR_OBJS)/, $(OBJS))

all: $(DIRS) $(EXE)

$(DIRS):
    $(MKDIR) $@
$(EXE): $(OBJS)
    $(CC) -o $@ $^
$(DIR_OBJS)/%.o: %.c
    $(CC) -o $@ -c $^
clean:
    $(RM) $(RMFLAGS) $(DIRS) $(EXE)
```

图 2.15

执行

```
$make
mkdir objs
mkdir exes
gcc -o objs/foo.o -c foo.c
gcc -o objs/main.o -c main.c
gcc -o exes/complicated objs/foo.o objs/main.o

$ls
Makefile exes foo.c foo.h main.c objs
```

图 2.16

2.4 更复杂的依赖关系

至此，我们的 Makefile 好象很具实用性了，但真的吗？现在我们需要看一看存在什么问题。假设我们对项目已经进行了一次编译（这一点非常重要，否则你看不到将要说的问题），接着对 foo.h文件进行了如

图 2.17 所示的更改，其改动就是对 foo ()函数增加了一个 int 类型的参数，而不对 foo.c进行相应的更改。这样一改的话，由于声名与定义不相同，所以理论上编译时应当出错。

foo.h

```
#ifndef __FOO_H
#define __FOO_H
void foo (int value);
#endif
```

图 2.17

图 2.18 示例了 make 的结果，是不是很吃惊？make 告诉我们没有什么事好做！那如果我们先make clean，然后再 make 又是什么结果呢？答案从图 2.19 可以看出，的确是出错了！那为什么在没有进行 make clean 之前，make 没有发现需要对项目的部分（或全部）进行重新构建呢？对于这样的 Makefile 如果运用到现实项目中，那对于开发效率还是有影响的，因为每一次 make 之前都得进行 clean，太费时！

执行

```
$make
make: Nothing to be done for `all'.
```

图 2.18

执行

```
$make clean
rm –fr objs exes
```

执行

```
$make
mkdir objs
mkdir exes
gcc –o objs/foo.o –c foo.c
gcc –o objs/main.o –c main.c
main.c: In function `main':
main.c:5: error: too few arguments to function `foo'
make: *** [objs/main.o] Error 1
```

图 2.19

现在，我们来分析一下此时的 Makefile 为什么会出现这一问题呢？图 2.20 是现有 Makefile 所表达的依赖关系树及与规则的映射关系图。前面的测试中，我们改动了 foo.h 文件，但从依赖关系图中你是否发现，**其中并没有出现对 foo.h 的依赖关系**，这就是为什么我们改动头文件时，make 无法发现的原因！

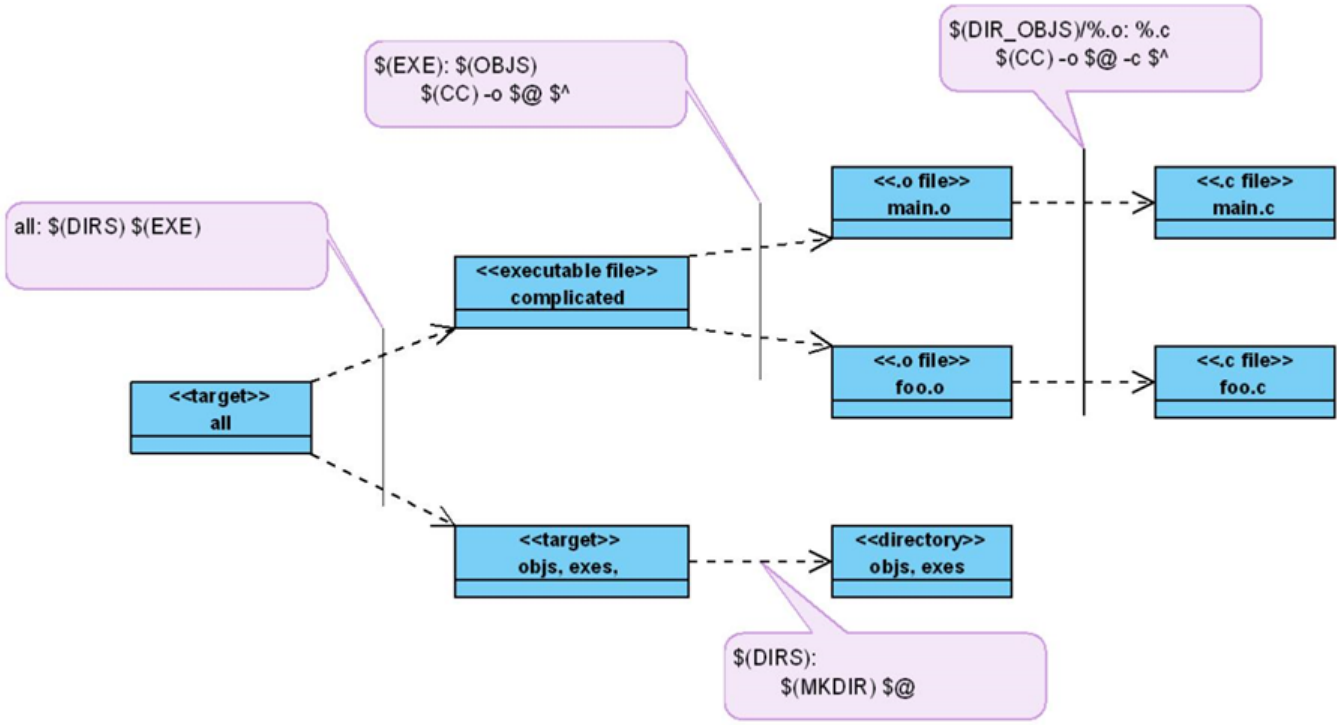


图 2.20

最为直接的改动是我们在构建目标文件的规则中，增加对于 foo.h 的依赖。改动后的 Makefile如图 2.21 所示。其改动也是非常的小的，需要指出的是，在这个 Makefile 中我们使用了**自动变量$<**（参见 1.5.1 节）。前面我们说子，这个变量与$^的区别是，其只表示所有的先决条件中的第一个，而$^则表示全部先决条件。之所以要用$<是因为，我们不希望将 foo.h 也作为一个文件让 gcc 去编译，这样的话会出错。

Makefile

```
.PHONY: all clean
MKDIR = mkdir
RM = rm
RMFLAGS = -fr
CC = gcc
DIR_OBJS = objs
DIR_EXES = exes
DIRS = $(DIR_OBJS) $(DIR_EXES)
EXE = complicated
EXE := $(addprefix $(DIR_EXES)/, $(EXE))
SRCS = $(wildcard *.c)
OBJS = $(SRCS:.c=.o)
OBJS := $(addprefix $(DIR_OBJS)/, $(OBJS))

all: $(DIRS) $(EXE)

$(DIRS):
```

```
    $(MKDIR) $@
$(EXE): $(OBJS)
    $(CC) -o $@ $^
$(DIR_OBJS)/%.o: %.c foo.h
    $(CC) -o $@ -c $<
clean:
    $(RM) $(RMFLAGS) $(DIRS)
```

图 2.21

现在，将 foo.h 改回以前的状态，即去除 foo ()函数中的 int 参数，然后编译，这次编译当然是成功的，接着再加入 int 参数，再编译。你发现这次真的能发现问题了！更改后的依赖关系图如图 2.22所示。

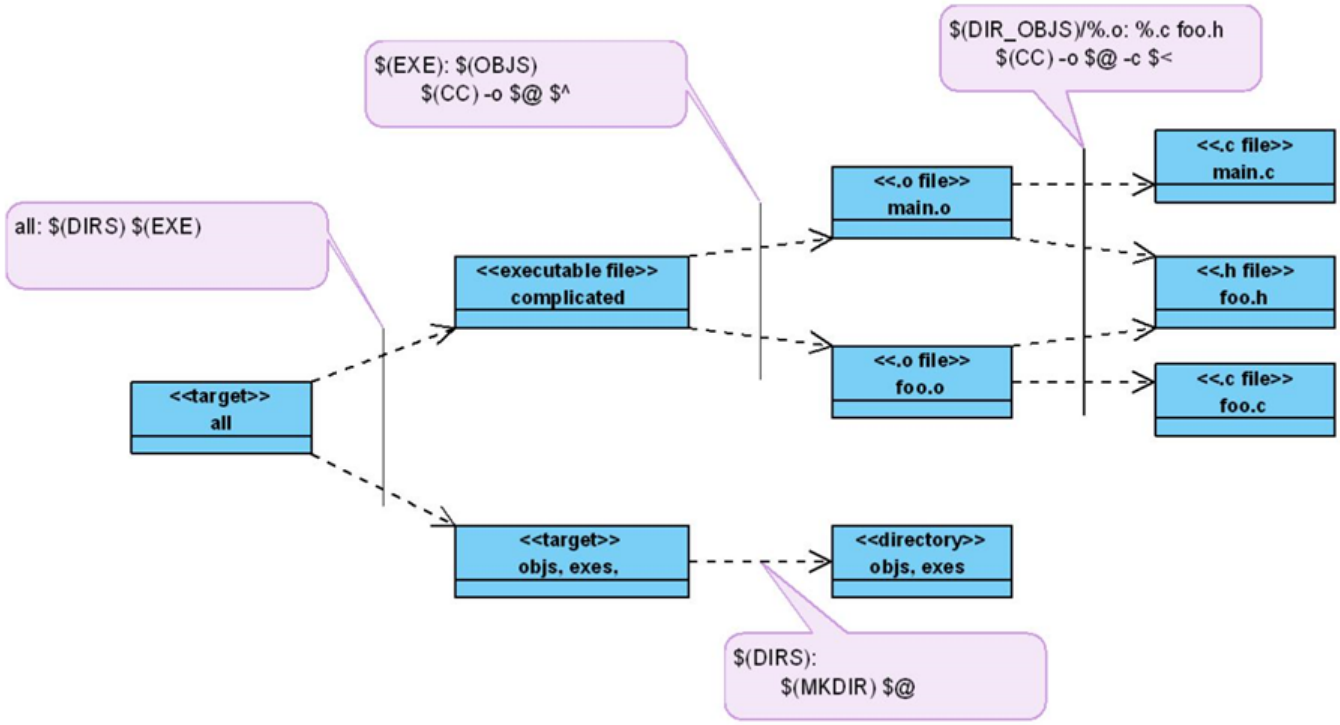


图 2.22

不错！我们又有了一些进展，但是现在需要想一想这种解决方法的可操作性。当项目复杂时，**如果我们要将每一个头文件都写入到 Makefile 相对应的规则中，这将会是一个恶梦！**看来我们还得找到另一种更好的方法。

如果有哪一个工具能帮助我们列出一个源程序所包含的头文件那就好了，这样的话我们或许可以在 make 时，动态的生成文件的依赖关系。还真是存在这么一个工具！就是我们的编译器 ——gcc。图 2.23 列出了采用 gcc 的-M 选项和-MM 选项列出 foo.c 对其它文件的依赖关系的结果，从结果你可以看出它们会列出 foo.c 中直接或是间接包含的头文件。-MM 选项与-M 选项的区别是，-MM选项并不列出对于系统头文件的依赖关系，比如 stdio.h 就属于系统头文件。其道理是，绝大多数情况我们并不会改变系统的头文件，而只会对自己项目的头文件进行更改。

执行

```
$gcc -M foo.c
```

```
foo.o: foo.c /usr/include/stdio.h /usr/include/_ansi.h \
/usr/include/newlib.h /usr/include/sys/config.h \
/usr/include/machine/ieeefp.h /usr/include/cygwin/config.h \
/usr/lib/gcc/i686-pc-cygwin/4.3.2/include/stddef.h \
/usr/lib/gcc/i686-pc-cygwin/4.3.2/include/stdarg.h \
/usr/include/sys/reent.h /usr/include/_ansi.h /usr/include/sys/_types.h \
/usr/include/sys/lock.h /usr/include/sys/types.h \
/usr/include/machine/_types.h /usr/include/machine/types.h \
/usr/include/sys/features.h /usr/include/cygwin/types.h \
/usr/include/sys/sysmacros.h /usr/include/stdint.h \
/usr/include/endian.h /usr/include/sys/stdio.h /usr/include/sys/cdefs.h \
foo.h
```

执行

```
$gcc -MM foo.c
foo.o: foo.c foo.h
```

图 2.23

对于采用 gcc 的-MM 的选项所生成的结果，还有一个问题，因为我们生成的目标文件是放在 objs目录当中的，因此，我们希望依赖关系中也包含这一目录信息，否则，在我们的 Makefile 中，跟本没有办法做到将生成的目标文件放到 objs 目录中去，这在前面的 Makefile 中我们就是这么做的。在使用新的方法时，我们仍然需要实现同样的功能。这时，我们需要用到 sed 工具了，这是 Linux 中非常常用的一个字符串处理工具。图 2.24 是采用 sed 进行查找和替换以后的输出结果，从结果中我们可以看到，就是在 foo.o 之前加上了“objs/”前缀。对于 sed 的用法说明可能超出了本文的范围，如果你不熟悉其功能，可以找一些资料看一看。

执行

```
$gcc -MM foo.c | sed 's,\(.*\)\.o[ :]*,objs/\1.o: ,g'
objs/foo.o: foo.c foo.h
```

图 2.24

gcc 还有一个非常有用的选项是-E，这个命令告诉 gcc 只做预处理，而不进行程序编译，在生成依赖关系时，其实我们并不需要 gcc 去编译，只要进行预处理就行了。这可以避免在生成依赖关系时出现没有必要的 warning，以及提高依赖关系的生成效率。

现在，我们已经有了自动生成依赖关系的方法了，那如何将其整合到我们的 Makefile 中呢？显然，自动生成的依赖信息，不可能直接出现在我们的 Makefile 中，因为我们不能动态的改变 Makefile中的内容，那采用什么方法呢？先别急，第一步我们能做的是，为每一个源文件通过采用 gcc 和 sed生成一个依赖关系文件，这些文件我们采用.dep 后缀结尾。从模块化的角度来说，我们不希望.dep文件与.o 文件或是可执行文件混放在一个目录中。为此，创建一个新的 deps 目录用于存放依赖文件似乎更为合理。图 2.25 中的 Makefile 增加了创建 **deps 目录**和为每一个源文件生成依赖关系文件
这两个功能。

Makefile

```
.PHONY: all clean
MKDIR = mkdir
```

```
RM = rm
RMFLAGS = -fr

CC = gcc

DIR_OBJS = objs
DIR_EXES = exes
DIR_DEPS = deps
DIRS = $(DIR_OBJS) $(DIR_EXES) $(DIR_DEPS)
EXE = complicated
EXE := $(addprefix $(DIR_EXES)/, $(EXE))
SRCS = $(wildcard *.c)
OBJS = $(SRCS:.c=.o)
OBJS := $(addprefix $(DIR_OBJS)/, $(OBJS))
DEPS = $(SRCS:.c=.dep)
DEPS := $(addprefix $(DIR_DEPS)/, $(DEPS))

all: $(DIRS) $(DEPS) $(EXE)

$(DIRS):
    $(MKDIR) $@
$(EXE): $(OBJS)
    $(CC) -o $@ $^
$(DIR_OBJS)/%.o: %.c foo.h
    $(CC) -o $@ -c $^
$(DIR_DEPS)/%.dep: %.c
    @echo "Making $@ ..."
    @set -e; \
    $(RM) $(RMFLAGS) $@.tmp ; \
    $(CC) -E -MM $^ > $@.tmp ; \
    sed 's,\(.*\)\.o[ :]*,objs/\1.o: ,g' < $@.tmp > $@ ; \
    $(RM) $(RMFLAGS) $@.tmp
clean:
    $(RM) $(RMFLAGS) $(DIRS)
```

图 2.25

图 2.25 中的 Makefile 存在以下更改：

- 增加了 DIR_DEPS 变量，用于保存需要创建的 deps 目录名，以及将这一变量的值加入到DIR 变量中。
- 删除了目标文件模式规则中对于 foo.h 文件的依赖，与此同时，我们将这个规则中的$<变成了$^。
- 增加了 DEPS 变量用于存放依赖文件。

- 为 all 目标增加了对于 DEPS 的依赖。
- 增加了一个用于创建依赖文件的模式规则。有这一规则的命令当中，我们使用了 gcc 的-E和-MM 选项来获取依赖关系，为了最终生成依赖文件，中间采用了一个临时文件。为了增加可读性，在生成一个依赖文件时，会在终端上打印类似“Making foo.dep ...”这样的提示信息。在这个规则中，set -e 的作用是告诉 BASH Shell 当生成依赖文件的过程中出现任何错误时，就直接退出。最强终的表现就是 make 会告诉我们出错了，从而停止后面的 make工作。如果不进行这一设置，当构建依赖文件出现错误时，make 还会继续后面的工作，这是我们所不希望的。同样，你可以试着将 set -e 去掉，然后故意在 foo.c 或是 main.c 中植入一个错误，观察一下 make 此时的行为是什么。

这里我们又有一个知识点需要注意，对于规则中的每一个命令，make 都是在一个新的 Shell 上运行它的，如果希望多个命令在同一个 Shell 中运行，则需要用‘;’将这些命令连起来。当命令很长时，为了方便阅读，我们需要将一行命令分成多行，**这需要用‘\’**。为了理解，我们可以做一个实验。**现在假设我们需要创建一个 test 目录**，然后，在这个 test 目录下面再创建一个 subtest 目录，如果你不知道 make 是如何执行命令的，你可能会写如所图 2.26 示的一个 Makefile。

Makefile

```
.PHONY: all
all:
    @mkdir test
    @cd test
    @mkdir subtest
```

图 2.26

图 2.27 是运行结果，从最后的 ls 结果来看，make 在同个目录中创建了 test 和 subtest 两个目录，而不是我们所希望的。现在，将 Makefile 做一下修改，运用前面提到的知识点，修改后的 Makefile如图 2.28 所示。

执行

```
$make

$ls
Makefile subtest test
```

图 2.27

```
Makefile
.PHONY: all
all:
    @mkdir test ; \
    cd test ; \
    mkdir subtest
```

图 2.28

现在先删除 test 和 subtest 目录，然后再运行一下 make 看一看结果，操作及结果如图 2.29 所示。这次的结果与我们所期望的完全相同！

执行

```
$rm -fr subtest/ test/
```

```
$make
目录 ~
$ls
Makefile test

$ ls test
subtest
```

图 2.29

现在，回到我们的 complicated 项目的 Makefile 上来，图 2.30 是图 2.25 的 Makefile 的运行结果，从图中你可以看出，Makefile 会为我们生成新的目录 deps、创建 foo.c 和 main.c 的依赖文件，分别是 deps/foo.dep 和 deps/main.dep。最后我们采用 cat 命令查看了一下 foo.dep 和 main.dep 中的内容，从内容上看来，的确是我们所需要的。

执行

```
$make
mkdir objs
mkdir exes
mkdir deps
Making deps/foo.dep ...
Making deps/main.dep ...
gcc –o objs/foo.o –c foo.c
gcc –o objs/main.o –c main.c
gcc –o exes/complicated objs/foo.o objs/main.o
```

执行

```
$cat deps/foo.dep
objs/foo.o: foo.c foo.h

$cat deps/main.dep
objs/main.o: main.c foo.h
```

图 2.30（和2.25一起使用）

2.5 包含文件

现在依赖文件已经有了，那如何为我们的 Makefile 所用呢？这需要用到 Makefile 中的 include关键字，它如同 C/C++中的#include 预处理指令。现在要做的就是在 Makefile 中加入对所有依赖文件的包含功能，更改后的 Makefile 如图 2.31 所示。

Makefile

```
.PHONY: all clean
MKDIR = mkdir
RM = rm
RMFLAGS = –fr
```

```
CC = gcc
DIR_OBJS = objs
DIR_EXES = exes
DIR_DEPS = deps
DIRS = $(DIR_OBJS) $(DIR_EXES) $(DIR_DEPS)
EXE = complicated
EXE := $(addprefix $(DIR_EXES)/, $(EXE))
SRCS = $(wildcard *.c)
OBJS = $(SRCS:.c=.o)
OBJS := $(addprefix $(DIR_OBJS)/, $(OBJS))
DEPS = $(SRCS:.c=.dep)
DEPS := $(addprefix $(DIR_DEPS)/, $(DEPS))

all: $(DIRS) $(DEPS) $(EXE)

include $(DEPS)

$(DIRS):
$(MKDIR) $@
$(EXE): $(OBJS)
$(CC) -o $@ $^
$(DIR_OBJS)/%.o: %.c
$(CC) -o $@ -c $<
$(DIR_DEPS)/%.dep: %.c
    @echo "Making $@ ..."
    @set -e; \
    $(RM) $(RMFLAGS) $@.tmp ; \
    $(CC) -E -MM $^ > $@.tmp ; \
    sed 's,\(.*\)\.o[ :]*,objs/\1.o: ,g' < $@.tmp > $@ ; \
    $(RM) $(RMFLAGS) $@.tmp
clean:
    $(RM) $(RMFLAGS) $(DIRS)
```

图 2.31

执行

```
$make
Makefile:25: deps/foo.dep: No such file or directory
Makefile:25: deps/main.dep: No such file or directory
Making deps/main.dep ...
/bin/sh: line 2: deps/main.dep.tmp: No such file or directory
make: *** [deps/main.dep] Error 1
```

图 2.32

现在需要运行一下看一看结果如何，如图 2.32 所示。从图中你可以看到，由于 make 在处理Makefile 的 include 命令时，发现找不到 deps/foo.dep 和 deps/main.dep，所以出错了。如何理解这一错误呢？从这一错误我们可知，make 对于 include 的处理是先于 all 目标的构建的，这样的话，由于依赖文件是在构建 all 目标时才创建的，所以很自然 make 在处理 include 指令时，是找不到依赖文件的。我们说第一次 make 的确没有依赖文件，所以 include 出错也是正常的，那能不能让 make忽略这一错误呢？可以的，**在 Makefile 中，如果在 include 前加上一个‘-’号，当 make 处理这一包含指示时，如果文件不存在就会忽略这一错误。**除此之外，需要对于 Makefile 中的 include 有更为深入的了解。当 make 看到 include 指令时，会先找一下有没有这个文件，如果有则读入。接着，

make 还会看一看对于包含进来的文件，在 Makefile 中是否存在规则来更新它。如果存在，则运行规则去更新需被包含进来的文件，当更新完了之后再将其包含进来。在我们的这个 Makefile 中，的确存在用于创建（或更新）依赖文件的规则。那为什么 make 没有帮助我们去创建依赖文件，而只是抱怨呢？因为 make 想创建依赖文件时，deps 目录还没有创建，所以无法成功的构建依赖文件。

有了这些信息之后，我们需要对 Makefile 的依赖关系进行调整，即将 deps 目录的创建放在构建依赖文件之前。其改动就是在依赖文件的创建规则当中增加对 deps 目录的信赖，且将其当作是第一个先决条件。采用同样的方法，我们将所有的目录创建都放到相应的规则中去。更改后的 Makefile如图 2.33 所示。

Makefile

```
.PHONY: all clean
MKDIR = mkdir
RM = rm
RMFLAGS = -fr
CC = gcc
DIR_OBJS = objs
DIR_EXES = exes
DIR_DEPS = deps
DIRS = $(DIR_OBJS) $(DIR_EXES) $(DIR_DEPS)
EXE = complicated
EXE := $(addprefix $(DIR_EXES)/, $(EXE))
SRCS = $(wildcard *.c)
OBJS = $(SRCS:.c=.o)
OBJS := $(addprefix $(DIR_OBJS)/, $(OBJS))
DEPS = $(SRCS:.c=.dep)
DEPS := $(addprefix $(DIR_DEPS)/, $(DEPS))

all: $(DIRS) $(DEPS) $(EXE)

-include $(DEPS)
```

```
$(DIRS):
    $(MKDIR) $@
$(EXE): $(DIR_EXES) $(OBJS)
    $(CC) -o $@ $(filter %.o, $^)
$(DIR_OBJS)/%.o: $(DIR_OBJS) %.c
    $(CC) -o $@ -c $(filter %.c, $^)
$(DIR_DEPS)/%.dep: $(DIR_DEPS) %.c
    @echo "Making $@ ..."
    @set -e; \
    $(RM) $(RMFLAGS) $@.tmp ; \
    $(CC) -E -MM $(filter %.c, $^) > $@.tmp ; \
    sed 's,\(.*\)\.o[ :]*,objs/\1.o: ,g' < $@.tmp > $@ ; \
    $(RM) $(RMFLAGS) $@.tmp
clean:
    $(RM) $(RMFLAGS) $(DIRS)
```

图 2.33

图 2.33 的 Makefile 中，我们使用了 filter 函数（参见 1.7.2 节）将所依赖的目录从先决条件中去除，否则的话会出现错误（你可以试试看，如果不改会出现什么错误）。正如前面所提及的，当make 看到 include 指令时，会试图去构建所需包含进来的依赖文件，这样一来，我们并不需要让 all目录依赖依赖文件，也就是从 all 规则中去除了对 DEPS 的依赖。图 2.34 示列了修订后的 Makefile的运行结果。

执行

```
$make
mkdir deps
Making deps/main.dep ...
Making deps/foo.dep ...
mkdir exes
mkdir objs
gcc -o objs/foo.o -c foo.c
gcc -o objs/main.o -c main.c
gcc -o exes/complicated objs/foo.o objs/main.o

$make clean
rm -fr objs exes deps
```

图 2.34

2.6 再复杂一点的依赖关系

现在，我们再对源程序文件进行一定的修改，如图 2.36 所示。其中的改动包括：

- 增加 define.h 文件并在其中定义一个 HELLO 宏。
- 在 foo.h 中包含 define.h 文件。

- 在 foo.c 中增加对 HELLO 宏的使用。

增加了这些改动以后，进行 make 操作，结果如图 2.37 所示。

define.h

```
#ifndef __DEFINE_H
#define __DEFINE_H
#define HELLO “Hello”
#endif
```

foo.h

```
#ifndef __FOO_H
#define __FOO_H
#include “define.h”
void foo ();
#endif
```

foo.c

```
#include <stdio.h>
#include “foo.h”
void foo ()
{
    printf (“%s, this is foo ()!\n”, HELLO);
}
```

main.c

```
#include “foo.h”
int main ()
{
    foo ();
    return 0;
}
```

图 2.36

So far so good！foo.dep 也如我们所想象的那样。现在我们得稍微改动一下代码，以便发现其它的问题。

执行

```
mkdir deps
Making deps/main.dep ...
Making deps/foo.dep ...
mkdir exes
mkdir objs
gcc –o objs/foo.o –c foo.c
gcc –o objs/main.o –c main.c
gcc –o exes/complicated objs/foo.o objs/main.o
```

执行

```
$cat deps/foo.dep
objs/foo.o: foo.c foo.h define.h
```

图 2.37

在上面成功编译过的基础之上，我们再做一些改动。注意一定要编译过，而不要 clean 后再做下面的改动，只有这样做我们才能发现问题。改动如图 2.32 所示，这次增加了一个 other.h 文件并将以前在 define.h 中定义的 HELLO 宏放到了这个文件当中，接着让 define.h 包含 other.h 文件。接下来，我们再进行一次 make 操作，结果如图 2.39 所示。从结果中你发现什么了吗？尽管 foo.c 和main.c 从新编译了，但依赖关系并没有重新构建！从运行 complicated 程序的结果来看，其打印的问候语也是 Hello，这正是我们程序所设计的。

define.h

```
#ifndef __DEFINE_H
#define __DEFINE_H
#include “other.h”
#endif
```

other.h

```
#ifndef __OTHER_H
#define __OTHER_H
#define HELLO “Hello”
#endif
```

图 2.38

执行

```
$make
gcc –o objs/foo.o –c foo.c
gcc –o objs/main.o –c main.c
gcc –o exes/complicated objs/foo.o objs/main.o

$./exes/complicated
Hello, this is foo ()!
```

图 2.39

现在对 other.h 进行更改，更改后内容如图 2.40 所示。改动就是将问候语 Hello 变成了 Hi，更改后又一次进行 make 操作，结果如图 2.41 所示。

other.h

```
#ifndef __OTHER_H
#define __OTHER_H
#define HELLO "Hi"
#endif
```

图 2.40

执行

```
$make
make: Nothing to be done for `all'.
```

```
$cat deps/foo.dep
objs/foo.o: foo.c foo.h define.h

$cat deps/main.dep
objs/main.o: main.c foo.h define.h
```

图 2.41

问题出来了，程序并没有因为我们更改了 other.h 而重新编译，问题出在哪呢？从 foo.dep 和main.dep 的内容来看，其中并没有指出 foo.o 和 main.o 依赖于 other.h 文件，所以当我们进行 make时，make 程序没有发现 foo.o 和 main.o 需要重新编译。那如何解决呢？我们说，当我们进行 make时，如果此时 make 能发现 foo.dep 和 main.dep 需要重新生成的话，此时会发现 foo.o 和 main.o都依赖 other.h 文件，那自然就会发现 foo.o 和 main.o 也需要重新编译。

也就是说我们也需要对依赖文件采用 foo.o 和 main.o 相类似的依赖规则，为此，我们希望在Makefile 中存在如图 2.42 所示的依赖关系。如果存在这样的依赖关系，当我们对 define.h 进行更改以增加对 other.h 文件的包含时，通过这个依赖关系 make 就能发现需要重新生成新的依赖文件，一旦重新生成依赖文件，other.h 也就自然会成为 foo.o 和 main.o 的一个先决条件。如果这样的话，就不会出现前面所看到的依赖关系并不重新构建的问题了。

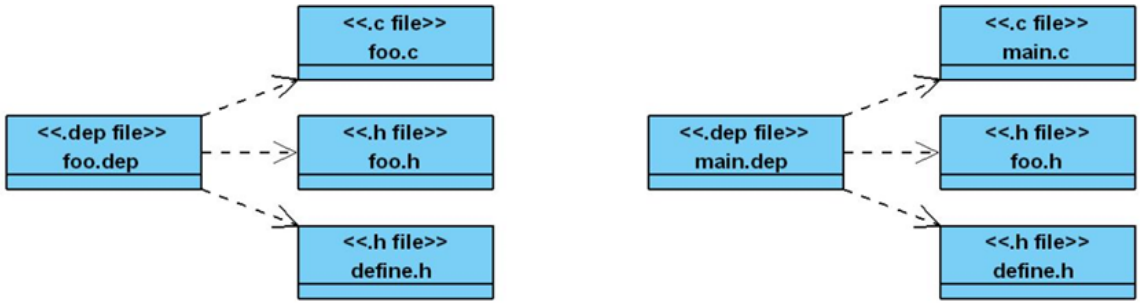


图 2.42

要增加这个依赖关系，在现有的 Makefile 上还是很简单的一件事。我们只需对 Makefile 进行小改就能做到。图 2.43 示例了改动的原理，其实，只要在依赖文件的构建规则中多增加依赖文件自身这个目标就行了。

执行

```
$gcc –MM foo.c | sed 's,\(.*\)\.o[ :]*,objs/\1.o deps/foo.dep: ,g'
objs/foo.o deps/foo.dep: foo.c foo.h define.h
```

图 2.43

现在看来，我们在生成依赖关系时，也需要将依赖文件作为一个目标。更改后的 Makefile 如图2.44 所示。

Makefile

```
.PHONY: all clean
MKDIR = mkdir
RM = rm
```

```
RMFLAGS = -fr
CC = gcc
DIR_OBJS = objs
DIR_EXES = exes
DIR_DEPS = deps
DIRS = $(DIR_OBJS) $(DIR_EXES) $(DIR_DEPS)
EXE = complicated
EXE := $(addprefix $(DIR_EXES)/, $(EXE))
SRCS = $(wildcard *.c)
OBJS = $(SRCS:.c=.o)
OBJS := $(addprefix $(DIR_OBJS)/, $(OBJS))
DEPS = $(SRCS:.c=.dep)
DEPS := $(addprefix $(DIR_DEPS)/, $(DEPS))

all: $(EXE)

-include $(DEPS)

$(DIRS):
    $(MKDIR) $@
$(EXE): $(DIR_EXES) $(OBJS)
    $(CC) -o $@ $(filter %.o, $^)
$(DIR_OBJS)/%.o: $(DIR_OBJS) %.c
    $(CC) -o $@ -c $(filter %.c, $^)
$(DIR_DEPS)/%.dep: $(DIR_DEPS) %.c
    @echo "Making $@ ..."
    @set -e; \
    $(RM) $(RMFLAGS) $@.tmp ; \
    $(CC) -E -MM $(filter %.c, $^) > $@.tmp ; \
    sed 's,\(.*\)\.o[ :]*,objs/\1.o $@: ,g' < $@.tmp > $@ ; \
    $(RM) $(RMFLAGS) $@.tmp
clean:
    $(RM) $(RMFLAGS) $(DIRS)
```

图 2.44

从这个 Makefile 中你可以看出，我们只需在相应的规则命令中增加一个$@就行了，因为这个表示的是目标，即在创建 deps/foo.dep 时，其代表的就是 deps/foo.dep。有了这样的改动后，你不能直接 make 来观察其效果，而是必需先 make clean 然后依此重新做图 2.38 和图 2.40 中所列出的变化。有了这样的变化之后，你会发现不论你如何更改源程序，make 都能发现并且做出正确的构建操作，当然这一切还得规功于我们，因为 Makefile 是我们写的呀！自已试试看吧！

2.7 条件语法

当 make 看到条件语法时将立即对其进行分析，这包括 ifdef、ifeq、ifndef 和 ifneq 四种语句形式。这也说明自动变量（参见 1.5.1 节）在这些语句块中不能使用，因为自动变量的值是在命令处理阶段才被赋值的。如果非得用条件语法，那得使用 Shell 所提供的条件语法而不是 Makefile 的。

Makefile 中的条件语法有三种形式，如图 2.45 所示。其中的 conditional-directive 可以是 ifdef、ifeq、ifndef 和 ifneq 中的任意一个。

```
conditional-directive
    text-if-true
endif
或
conditional-directive
    text-if-true
else
    text-if-false
endif
或
conditional-directive
    text-if-one-is-true
else conditional-directive
    text-if-true
else
    text-if-false
endif
```

图 2.45

对于 ifeq 和 ifneq，可以采用如图 2.46 所示的格式。图 2.47 是使用 ifeq 和 ifneq 条件语法的一个例子，图 2.48 则显示了其运行结果。

```
ifeq-or-ifneq (arg1, arg2)
ifeq-or-ifneq 'arg1' 'arg2'
ifeq-or-ifneq "arg1" "arg2"
ifeq-or-ifneq "arg1" 'arg2'
ifeq-or-ifneq 'arg1' "arg2"
```

图 2.46

Makefile

```
.PHONY: all

sharp = square
desk = square
table = circle
```

```
ifeq ($(sharp), $(desk))
    result1 = "desk == sharp"
endif

ifneq "$(table)" 'square'
    result2 = "table != square"
endif

all:
    @echo $(result1)
    @echo $(result2)
```

图 2.47

执行

```
$make
desk == sharp
table != square
```

图 2.48

ifdef和ifndef的格式如图 2.49所示。图 2.50是使用ifdef和ifndef的一个示例Makefile，图 2.51则是其运行结果。

```
ifdef-or-ifndef variable-name
```

图 2.49

Makefile

```
.PHONY: all
foo = defined

ifdef foo
    result1 = "foo is defined"
endif

ifndef bar
    result2 = "bar is not defined"
endif

all:
    @echo $(result1)
    @echo $(result2)
```

图 2.50

执行

```
$make
foo is defined
```

```
bar is not defined
```

图 2.51

现在回到我们的 complicated 项目（图2.44，对应源码目录2.44），你可能发现了当我们进行一次 make，然后接着进行两次
make clean 操作时，第二次的 make clean 与第一次有所不同，如图 2.52 所示。

执行

```
$make
mkdir deps
Making deps/main.dep ...
Making deps/foo.dep ...
mkdir exes
mkdir objs
gcc –o objs/foo.o –c foo.c
gcc –o objs/main.o –c main.c
gcc –o exes/complicated objs/foo.o objs/main.o

$make clean
rm –fr objs exes deps

$make clean
mkdir deps
Making deps/main.dep ...
Making deps/foo.dep ...
rm –fr objs exes deps
```

图 2.52

看到了吗？当进行第二次 make clean 时，make 还会先构建依赖文件，接着再删除，这是因为我们进行 make clean 也需要包含依赖文件的缘故。显然，其中构建依赖文件的动作有点多余，因为后面马上又被删除了。为了去除在 make clean 时不必要的依赖文件构建，我们可以用条件语法来解决这一问题。方法就是，当进行 make clean 时，我们不希望 complicated 项目的 Makefile 包含依赖文件进来，更改以后的 Makefile 如图 2.53 所示。这里的更改，我们用到了 MAKECMDGOALS 变量（参见 1.5.2 节）。有了这一改动后，即使是进行连续的多次 make clean，make 也不会先构建依赖文件。

Makefile

```
.PHONY: all clean

MKDIR = mkdir
RM = rm
RMFLAGS = –fr

CC = gcc
```

```
DIR_OBJS = objs
DIR_EXES = exes
DIR_DEPS = deps
DIRS = $(DIR_OBJS) $(DIR_EXES) $(DIR_DEPS)

EXE = complicated
EXE := $(addprefix $(DIR_EXES)/, $(EXE))

SRCS = $(wildcard *.c)
OBJS = $(SRCS:.c=.o)
OBJS := $(addprefix $(DIR_OBJS)/, $(OBJS))
DEPS = $(SRCS:.c=.dep)
DEPS := $(addprefix $(DIR_DEPS)/, $(DEPS))

all: $(EXE)

ifneq ($(MAKECMDGOALS), clean)
-include $(DEPS)
endif

$(DIRS):
    $(MKDIR) $@
$(EXE): $(DIR_EXES) $(OBJS)
    $(CC) -o $@ $(filter %.o, $^)
$(DIR_OBJS)/%.o: $(DIR_OBJS) %.c
    $(CC) -o $@ -c $(filter %.c, $^)
$(DIR_DEPS)/%.dep: $(DIR_DEPS) %.c
    @echo "Making $@ ..."
    @set -e; \
    $(RM) $(RMFLAGS) $@.tmp ; \
    $(CC) -E -MM $(filter %.c, $^) > $@.tmp ; \
    sed 's,\(.*\)\.o[ :]*,objs/\1.o $@: ,g' < $@.tmp > $@ ; \
    $(RM) $(RMFLAGS) $@.tmp
clean:
    $(RM) $(RMFLAGS) $(DIRS)
```

图 2.53

2.8 小结

complicated 项目到目前为止算是胜利的完成了，在这个项目中我们采用了如图 2.54 所示的目录结构。尽管这种单一的结构在现实项目中比较少见，但是，这个项目的 Makefile 已经是非常的完整了。毕竟，它能准确无误的发现哪些文件需要从新构建，而不是需要我们先 make clean，然后再make。当然，更不用担心我们对代码进行了更改，但 make 却没有对其进行编译，而我们认为所有的更改都被编译了。如果存在这种情况，那调程序时，就是抓破了头也想不明白！Worry free!

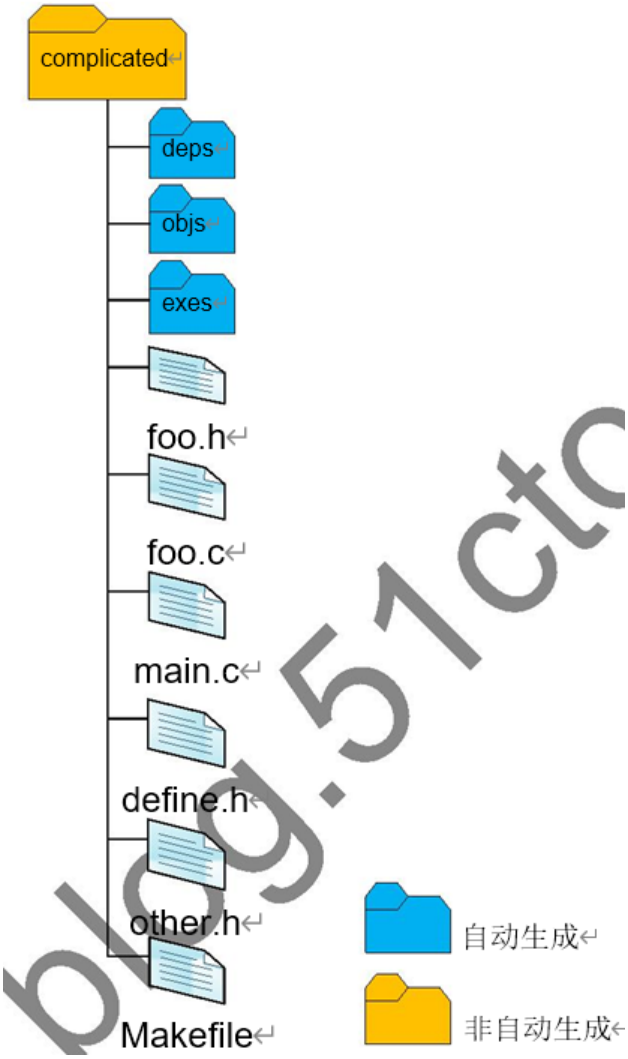


图 2.54

3 进阶

为了取得进一步的提高，我们需要做一个更为大型的项目，就称其为 huge 项目吧！与前面的

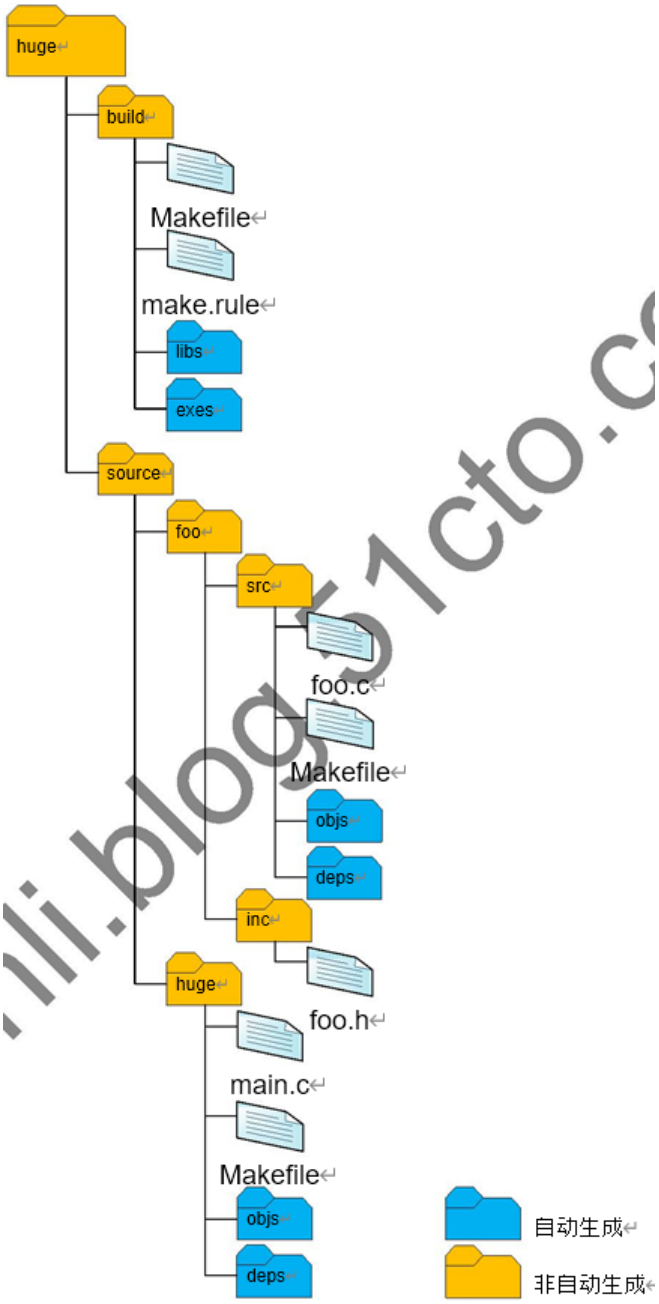


图 3.1

complicated 项目相比，huge 项目将会采用更加复杂但却合理的目录结构。图 3.1 示例了 huge 项目将要采用的目录结构，从图中可以看出，huge 项目是上层有两个目录，一个是 build 目录，而另一个则是 source 目录。前者用于存放共公的 make.rule，以及编译整个项目的的 Makefile。此外在 build 目录中还会在编译时自动生成 libs 和 exes 两个子目录。libs 目录用于存放编译出来的库文件，而 exes 用于存放编译出来的可执行文件。source 目录则用于存放项目的源程序，其下将按各个软件模块分成不同的子目录。在现在的 huge 项目中，包括 foo 库和 huge 主程序，所以在 source目录下分别创建了 foo 和 huge 子目录。对于每一个软件模块子目录，又分为用于存放源程序的 src子目录和用于存放头文件的 inc 子目录。当进行编译时，我们希望 Makefile 在 src 目录下面创建一个 deps 目录，用于存放依赖文件，这一点与 compliated 项目中的 deps 目录是一样的；另一个目录objs，也同 complicated 项目一样，

用于存放编译过程中生成的目标文件。另外，在每一个 src 目录中都会有一个 Makefile，这一 Makefile 的作用是只用于构建所在目录中的源程序，可以想像的是，在 build 目录下面的 Makefile 将调用每一个软件模块 src 目录下的 Makefile，从而实现整个项目的构建。为了实现各软件模块 src 目录中 Makefile 之间最大程度的复用，我们可以让它们共用的部分放在一个独立的文件中 —— 这就是 build 目录下的 make.rule 的作用。

3.1 创建目录

毫无例外的是，我们的 huge 项目也是以创建目录作为第一步的，在此刻，我们并不关心项目的源程序，源程序可以在需要时再放入。请先根据图 3.1 创建好那些需要手动创建的目录，采用图3.2 所示的命令能完成这些手动目录的创建工作。

具体参考 Makefile/huge源码

执行

$mkdir –p build source/foo/src source/foo/inc source/huge/src

图 3.2

接下来假设我们先创建位于 source/foo/src 下面的 Makefile，这个 Makefile 我们可以基于 complicated 项目最终版本的 Makefile 进行一定的更改而得到，如图 3.3 所示。

source/foo/src/Makefile

```
.PHONY: all clean

MKDIR = mkdir
RM = rm
RMFLAGS = –fr

CC = gcc

AR = ar
ARFLAGS = crs
DIR_OBJS = objs
DIR_EXES = ../../../build/exes
DIR_DEPS = deps
DIR_LIBS = ../../../build/libs
DIRS = $(DIR_DEPS) $(DIR_OBJS) $(DIR_EXES) $(DIR_LIBS)
RMS = $(DIR_OBJS) $(DIR_DEPS)

EXE = complicated
ifneq ($(EXE), "")
EXE := $(addprefix $(DIR_EXES)/, $(EXE))
RMS += $(EXE)
endif
```

```
LIB = libfoo.a
ifneq ($(LIB), "")
LIB := $(addprefix $(DIR_LIBS)/, $(LIB))
RMS += $(LIB)
endif

SRCS = $(wildcard *.c)
OBJS = $(SRCS:.c=.o)
OBJS := $(addprefix $(DIR_OBJS)/, $(OBJS))
DEPS = $(SRCS:.c=.dep)
DEPS := $(addprefix $(DIR_DEPS)/, $(DEPS))

ifneq ($(EXE), "")
all: $(EXE)
endif

ifneq ($(LIB), "")
all: $(LIB)
endif

ifneq ($(MAKECMDGOALS), clean)
-include $(DEPS)
endif

$(DIRS):
$(MKDIR) $@
$(EXE): $(DIR_EXES) $(OBJS)
    $(CC) -o $@ $(filter %.o, $^)
$(LIB): $(DIR_LIBS) $(OBJS)
    $(AR) $(ARFLAGS) $@ $(filter %.o, $^)
$(DIR_OBJS)/%.o: $(DIR_OBJS) %.c
    $(CC) -o $@ -c $(filter %.c, $^)
$(DIR_DEPS)/%.dep: $(DIR_DEPS) %.c
    @set -e ; \
    echo "Making $@ ..." ; \
    $(RM) $(RMFLAGS) $@.tmp ; \
    $(CC) -E -MM $(filter %.c, $^) > $@.tmp ; \
sed 's,\(.*\)\.o[ :]*,objs/\1.o $@: ,g' < $@.tmp > $@ ; \
        $(RM) $(RMFLAGS) $@.tmp
```

```
clean:
    $(RM) $(RMFLAGS) $(DIRS) $(RMS)
```

图 3.3

图 3.3 中的 Makefile 与 complicated 项目的最终版本相比，包含如下的更改：

- 增加了 AR 和 ARFLAGS 两个变量，用途是用以生成静态库。对于静态库及 ar 工具的使用你可能需要参照《熟悉 binutils 工具集》一文。
- 给 DIR_EXES 变量赋了正确的值，用于表示 exes 目录的实际位置。现在，还是采用相对路径。
- 增加了 DIR_LIBS 变量，同样采用相对路径的方式。
- 在 DIRS 变量中增加了 DIR_LIBS 变量中的内容，以便在生成库文件之前创建 build/libs 目录。
- 新增了 RMS 变量，用于表示需要删除的目录和（或）文件。由于这个 Makefile 只是针对构建 libfoo.a 库的，所以当我们运行 make clean 时，我们不能将位于 build 目录下的 exes和 libs 目录全部删除，这与前面的 complicated 项目是完全不同的。
- 删除了将 compliated 赋值给 EXE 变量，同时增加了 ifneq 条件语句用于判断 EXE 变是否被定义。之所以要用条件语句，是因为后面在设置 all 目标的依赖关系时，需要判断EXE 变量是否有值，如果没有值，我们并不需要让 all 目标依赖$(EXE)。这里增加条件语句的作用就是，如果 EXE 没有值，我们也不需要为其调用 addprefix 函数增加前缀，否则这会打破我们后面判断 EXE 变量是否有值，从而决定是否让 all 目录依赖于它这一方法。
- 此外，如果 EXE 有值那么我们应当将其值加入到 RMS 变量中，以便在我们调用 make clean时清除它。
- 新增了 LIB 变量，用于存放库名，在这里库的名称就是 libfoo.a。同样采用处理 EXE 变量的方法增加了条件语句。
- 对 all 目标增加条件语句以决定是否将$(EXE)或是$(LIB)作为它的先决条件。
- 增加了一条用于构建库的规则，方法就是采用 crs 参数调用 ar 命令以生成库。
- 在 clean 目标命令中，采用删除 RMS 变量中的内容，而不是 DIRS 变量中的内容。这一点前面我们说过了，因为我们不希望在 foo模块中 make clean时将 build目录下的 libs 和 exes目录也删除。

现在需要试一试这个 Makefile 是否能工作，在试之前，我们需要先在 src 目录中增加一个源程序。这可以采用 touch 命令来创建一个空的 foo.c 文件，操作结果如图 3.4 所示。

执行

```
$touch foo.c
目录/Makefile/huge/source/foo/src
$make
mkdir deps
Making deps/foo.dep ...
mkdir objs
gcc –o objs/foo.o –c foo.c
ar crs ../../../build/libs/libfoo.a objs/foo.o
目录 /Makefile/huge/source/foo/src
$ls
Makefile deps foo.c objs
```

```
目录 /Makefile/huge/source/foo/src
$ls ../../../build/libs/
libfoo.a
目录 /Makefile/huge/source/foo/src
$make clean
rm –fr objs deps ../../../build/libs/libfoo.a
目录 /Makefile/huge/source/foo/src
$ls ../../../build/libs/
```

图 3.4

从运行的结果来看，的确是在 build/libs 目录的下面生成了一个 libfoo.a 库文件。运行 make clean以后，也并没有将 build/libs 这个目录给删除，而只是删除了 libfoo.a 文件，这正是我们所设计的。接下来，我们需要考虑的是，将这个文件运用到 source/huge/src 目录下面去。最为直接的想法是，将 foo 模块的 Makefile 拷贝到 huge/src 目录下面去，然后做一些小的改动。在这样做之前，请等一下！Makefile 的设计与我们采用编程语言进行设计是一样的，我们需要考虑尽可能的复用，以提高可维护性。要做好这一点，这需要我们站在更高的层次去思考如何设计 Makefile。

3.2 提高复用性

在一开始，我们提到在 build 的目录下面将会放一个 make.rule 文件，这个文件的目的就是为了让所有位于各软件模块的 src 目录下面的 Makefile 能使用它以提高复用性。也就是说，我们需要考虑将 foo 模块的 Makefile 中的一部分内容放入到 make.rule 中。显然，只能将那些我们认为是公共的部分放入其中。那反过来，foo 模块的 Makefile 中，哪些是不能公用的呢？我想有以下几点：

- 变量 EXE 和 LIB 的定义对于每一个软件模块是不同的。比如在我们的 huge 项目中，我们需要在 source/foo/src 目录中的 Makefile 里将 LIB 变量的值设置为 libfoo.a，且 EXE 变量应当为空。但是，在 source/huge/src 目录中的 Makefile 里面，我们却要反过来，只定义EXE 变量的值为 huge。
- DIR_EXES 变量和 DIR_LIBS 变量由于运用了相对路径，所以也是每个模块特有的。但是可以采用绝对路径的方式解决这个问题。比如，我们可以定义一个 ROOT 环境变量，其值设置为 huge 项目的根目录，这样的话 DIR_EXES 和 DIR_LIBS 就可以通过以 ROOT 为相对路径，从而使得其值对于所有的模块都相同。

考虑到复用性，现在 foo 模块的 Makefile 由两部分组成了，build 目录中的 make.rule 和 source/foo/src 目录中的 Makefile，其内容如图 3.5 所示。

build/make.rule

```
.PHONY: all clean
MKDIR = mkdir
RM = rm
RMFLAGS = –fr

CC = gcc
AR = ar
```

```
ARFLAGS = crs

DIR_OBJS = objs
DIR_EXES = $(ROOT)/build/exes
DIR_DEPS = deps
DIR_LIBS = $(ROOT)/build/libs
DIRS = $(DIR_DEPS) $(DIR_OBJS) $(DIR_EXES) $(DIR_LIBS)
RMS = $(DIR_OBJS) $(DIR_DEPS)

~~EXE =~~
ifneq ($(EXE), "")
EXE := $(addprefix $(DIR_EXES)/, $(EXE))
RMS += $(EXE)
endif

~~LIB = libfoo.a~~
ifneq ($(LIB), "")
LIB := $(addprefix $(DIR_LIBS)/, $(LIB))
RMS += $(LIB)
endif

SRCS = $(wildcard *.c)
OBJS = $(SRCS:.c=.o)
OBJS := $(addprefix $(DIR_OBJS)/, $(OBJS))
DEPS = $(SRCS:.c=.dep)
DEPS := $(addprefix $(DIR_DEPS)/, $(DEPS))

ifneq ($(EXE), "")
all: $(EXE)
endif

ifneq ($(LIB), "")
all: $(LIB)
endif

ifneq ($(MAKECMDGOALS), clean)
-include $(DEPS)
endif

$(DIRS):
```

```
    $(MKDIR) $@
$(EXE): $(DIR_EXES) $(OBJS)
    $(CC) -o $@ $(filter %.o, $^)
$(LIB): $(DIR_LIBS) $(OBJS)
    $(AR) $(ARFLAGS) $@ $(filter %.o, $^)
$(DIR_OBJS)/%.o: $(DIR_OBJS) %.c
    $(CC) -o $@ -c $(filter %.c, $^)
$(DIR_DEPS)/%.dep: $(DIR_DEPS) %.c
    @echo "Making $@ ..."
        @set -e; \
    $(RM) $(RMFLAGS) $@.tmp ; \
    $(CC) -E -MM $(filter %.c, $^) > $@.tmp ; \
    sed 's,\(.*\)\.o[ :]*,objs/\1.o $@: ,g' < $@.tmp > $@ ; \
    $(RM) $(RMFLAGS) $@.tmp
clean:
    $(RM) $(RMFLAGS) $(RMS)
```

source/foo/src/Makefile

```
EXE =
LIB = libfoo.a
include $(ROOT)/build/make.rule
```

图 3.5

有了这一改动以后，你会发现 foo 模块的 Makfile 中 的内容很是简单，其中原来的大部分内容都被放到了 make.rule 文件中。如果要运行 make，那么必须先在 Shell 上 export 所需的 ROOT 变量，图 3.6 示例了操作步骤。

执行

```
目录 /Makefile/huge
$export ROOT=`pwd`
目录 /Makefile/huge
$cd source/foo/src
目录 /Makefile/huge/source/foo/src
$make
mkdir deps
Making deps/foo.dep ...
mkdir objs
gcc -o objs/foo.o -c foo.c
ar crs /Makefile/huge/build/libs/libfoo.a objs/foo.o
目录 /Makefile/huge/source/foo/src
$make clean
rm -fr objs deps /Makefile/huge/build/libs/libfoo.a
```

图 3.6

如果你是一个 Linux 新手，需要注意的是在 export 所需的 ROOT 变量时，除了先要进入 huge项目的根目录外，pwd 命令前后的字符是“`”而不是“'”，这个字符是键盘上“！”键左边的那一个。从运行结果来看，我们为了提高复用性的努力有一点进展！

接下来需要考虑 sourc/huge/src 目录中的 Makefile 了，我们希望在这个目录中放的程序能生成一个可执行文件，在测试 Makefile 时，需要在目录中放置一个 main.c 文件，其中的内容如图 3.7所示，而 Makefile 的内容如图 3.8 所示。

source/huge/src/main.c

```
int main ()
{
    return 0;
}
```

图 3.7

source/huge/src/Makefile

```
EXE = huge
LIB =
include $(ROOT)/build/make.rule
```

图 3.8

紧接着，我们需要进入 source/huge/src 目录进行 make，以检验其下的 Makefile 是否能正常工作，结果如图 3.9 所示。你一定会对这一运行的结果感到满意！

执行

```
目录 /Makefile/huge/source/huge/src
$make clean
mkdir deps
Making deps/main.dep ...
mkdir objs
gcc –o objs/main.o –c main.c
gcc –o /Makefile/huge/build/exes/huge objs/main.o

目录 /Makefile/huge/source/huge/src
$ls $ROOT/build/exes
huge
目录 /Makefile/huge/source/huge/src
$make clean
rm –fr objs deps /Makefile/huge/build/exes/huge
```

图 3.9

3.3 加入源程序文件

现在我们的 Makefile 系统已经有了初步的成果了，下面要做的是将源程序加入到目录中。所需的程序内容可以从 complicated 项目中得来，如图 2.10 所示。现在让我们试一试进入 source/foo/src目录中进

行编译，运行结果如图 3.10 所示。

执行

```
目录 /Makefile/huge/source/foo/src
$make
mkdir deps
Making deps/foo.dep ...
foo.c:2:17: error: foo.h: No such file or directory
mkdir objs
gcc –o objs/foo.o –c foo.c
foo.c:2:17: error: foo.h: No such file or directory
make: *** [objs/foo.o] Error 1
```

图 3.10

正如你所看到的，在构建 foo.dep 依赖文件时，make 就发现了错误，但是还是继续进行编译，直到编译 foo.c 时才因为错误而终止。这种行为并不是我们所期望的，理想的情况是，make 一旦发现错误就应当立即退出，这有利于我们在第一时间知道第一个出错点，从而不容易隐藏错误。那这是为什么呢？前面我们说了我们在构建依赖文件的规则中运用了 set –e 命令，以告诉 Shell 一发现错误就退出，可是为什么没有退出呢？想一想我们是如何包含依赖文件的？我们在 include 指令的前面放了一个‘–’，告诉 make 在 include 出现错误时，不要报错。正是这一原因导致了即使 make 发现了错误，仍进行后续的构建操作。改动很简单，将 make.rule 中 include 语句前的‘–’去掉，去
掉之后的 make 结果如图 3.11 所示。

执行

```
目录 /Makefile/huge/source/foo/src
$make
/Makefile/huge/build/make.rule:43: deps/foo.dep: No such file or directory
mkdir deps
Making deps/foo.dep ...
foo.c:2:17: error: foo.h: No such file or directory
make: *** [deps/foo.dep] Error 1
```

图 3.11

这一次虽然 make 会报告找不到 foo.dep，但继续进行 foo.dep 的生成操作，这是我们所希望的行为。接着，在生成 foo.dep 时，由于找不到 foo.h 头文件所以出错了，这一次 make 直接就告诉了我们这一错误。

下面要做的是解决 foo.h 头文件找不到的问题。根据图 3.1 所示的 huge 项目的目录结构，你会发现，现在的 foo.c 源文件和 foo.h 头文件是分别被存放在不同的目录中的，当进行依赖文件构建时，我们需要采用一种形式告诉编译器到指定的目录中查找头文件。看来，我们得对 Makefile 进行一定的改造了，改造的基本原理是，利用 gcc 的–I（‘i’的大写）选项。更改后的 Makefile 如图 3.12 所示。

build/make.rule

```
.PHONY: all clean
```

```
MKDIR = mkdir
RM = rm
RMFLAGS = -fr

CC = gcc
AR = ar
ARFLAGS = crs

DIR_OBJS = objs
DIR_EXES = $(ROOT)/build/exes
DIR_DEPS = deps
DIR_LIBS = $(ROOT)/build/libs
DIRS = $(DIR_DEPS) $(DIR_OBJS) $(DIR_EXES) $(DIR_LIBS)
RMS = $(DIR_OBJS) $(DIR_DEPS)

ifneq ($(EXE), "")
EXE := $(addprefix $(DIR_EXES)/, $(EXE))
RMS += $(EXE)
endif

ifneq ($(LIB), "")
LIB := $(addprefix $(DIR_LIBS)/, $(LIB))
RMS += $(LIB)
endif

SRCS = $(wildcard *.c)
OBJS = $(SRCS:.c=.o)
OBJS := $(addprefix $(DIR_OBJS)/, $(OBJS))
DEPS = $(SRCS:.c=.dep)
DEPS := $(addprefix $(DIR_DEPS)/, $(DEPS))
ifneq ($(EXE), "")
all: $(EXE)
endif
ifneq ($(LIB), "")
all: $(LIB)
endif
ifneq ($(MAKECMDGOALS), clean)
include $(DEPS)
endif
```

```
ifneq ($(INC_DIRS), "")
INC_DIRS := $(strip $(INC_DIRS))
INC_DIRS := $(addprefix -I, $(INC_DIRS))
endif

$(DIRS):
    $(MKDIR) $@
$(EXE): $(DIR_EXES) $(OBJS)
    $(CC) -o $@ $(filter %.o, $^)
$(LIB): $(DIR_LIBS) $(OBJS)
    $(AR) $(ARFLAGS) $@ $(filter %.o, $^)
$(DIR_OBJS)/%.o: $(DIR_OBJS) %.c
    $(CC) $(INC_DIRS) -o $@ -c $(filter %.c, $^)
$(DIR_DEPS)/%.dep: $(DIR_DEPS) %.c
    @echo "Making $@ ..."
    set -e ; \
    $(RM) $(RMFLAGS) $@.tmp ; \
    $(CC) $(INC_DIRS) -E -MM $(filter %.c, $^) > $@.tmp ; \
    sed 's,\(.*\)\.o[ :]*,objs/\1.o $@: ,g' < $@.tmp > $@ ; \
    $(RM) $(RMFLAGS) $@.tmp
clean:
    $(RM) $(RMFLAGS) $(RMS)
```

source/foo/src/Makefile

```
EXE =
LIB = libfoo.a
INC_DIRS = $(ROOT)/source/foo/inc
include $(ROOT)/build/make.rule
```

source/huge/src/Makefile

```
EXE = huge
LIB =
INC_DIRS = $(ROOT)/source/foo/inc
include $(ROOT)/build/make.rule
```

图 3.12

从改动来看就是增加了一个用于存放各模块所需用到的全部头文件的目录的变量 ——INC_DIRS，这一变量可以存放多个目录，这完全是根据软件模块的需要的。此外，在 make.rule 中增加了一个条件语句块，即当 INC_DIRS 中的值不为空时，先采用 strip 函数去除多余的空格，然后再用 addprefix 函数为所有的目录的前面加上“-I”前缀。最后的改动就是，让目标文件生成规则和依赖文件生成规则中增加对

INC_DIRS 变量的引用，以告诉 gcc 到哪去找头文件。增加这些改动之后，让我们再看看编译 libfoo.a 的结果是怎样的，如图 3.13 所示。

执行

```
目录 /Makefile/huge/source/foo/src
$make
/Makefile/huge/build/make.rule:43: deps/foo.dep: No such file or directory
mkdir deps
Making deps/foo.dep ...
mkdir objs
gcc –I../inc –o objs/foo.o –c foo.c
ar crs /Makefile/huge/build/libs/libfoo.a objs/foo.o

目录 /Makefile/huge/source/huge/src
$ls $ROOT/build/libs
libfoo.a
```

图 3.13

这次 libfoo.a 能被成功构建出来了，那 huge 可执行程序呢？进入 source/huge/src 目录，运行 make 的结果如图 3.14 所示。出错了！

```
目录 /Makefile/huge/source/huge/src
$make
/Makefile/huge/build/make.rule:43: deps/main.dep: No such file or directory
Making deps/main.dep ...
mkdir objs
gcc –I/Makefile/huge/source/foo/inc –o objs/main.o –c main.c
gcc –o /Makefile/huge/build/exes/huge objs/main.o
objs/main.o:main.c:(.text+0x17): undefined reference to `_foo'
collect2: ld returned 1 exit status
make: *** [/Makefile/huge/build/exes/huge] Error 1
```

图 3.14

可以看出 main.o 目标文件被正确的生成了，但在连接时找不到_foo 的引用，即找不到 foo ()函数。由于 foo ()函数的实现是放在 libfoo.a 库中的，而我们又没有告诉编译器，在连接时应当到 libfoo.a中找所需的函数。其实，现在要做的与前面头文件目录的指定工作是相类似，只是这一次要用到 gcc的–l（小写的 L）和–L 选项，更改后的 Makefile 如图 3.15 所示。

build/make.rule

```
.PHONY: all clean
MKDIR = mkdir
RM = rm
RMFLAGS = –fr
CC = gcc
AR = ar
```

```
ARFLAGS = crs
DIR_OBJS = objs
DIR_EXES = $(ROOT)/build/exes
DIR_DEPS = deps
DIR_LIBS = $(ROOT)/build/libs
DIRS = $(DIR_DEPS) $(DIR_OBJS) $(DIR_EXES) $(DIR_LIBS)
RMS = $(DIR_OBJS) $(DIR_DEPS)
ifneq ($(EXE), "")
EXE := $(addprefix $(DIR_EXES)/, $(EXE))
RMS += $(EXE)
endif

ifneq ($(LIB), "")
LIB := $(addprefix $(DIR_LIBS)/, $(LIB))
RMS += $(LIB)
endif
SRCS = $(wildcard *.c)
OBJS = $(SRCS:.c=.o)
OBJS := $(addprefix $(DIR_OBJS)/, $(OBJS))
DEPS = $(SRCS:.c=.dep)
DEPS := $(addprefix $(DIR_DEPS)/, $(DEPS))
ifneq ($(EXE), "")
all: $(EXE)
endif
ifneq ($(LIB), "")
all: $(LIB)
endif
ifneq ($(MAKECMDGOALS), clean)
include $(DEPS)
endif
ifneq ($(INC_DIRS), "")
INC_DIRS := $(strip $(INC_DIRS))
INC_DIRS := $(addprefix -I, $(INC_DIRS))
endif
ifneq ($(LINK_LIBS), "")
LINK_LIBS := $(strip $(LINK_LIBS))
LINK_LIBS := $(addprefix -l, $(LINK_LIBS))
endif
$(DIRS):
$(MKDIR) $@
```

```
$(EXE): $(DIR_EXES) $(OBJS)
$(CC) -L$(DIR_LIBS) -o $@ $(filter %.o, $^) $(LINK_LIBS)
$(LIB): $(DIR_LIBS) $(OBJS)
$(AR) $(ARFLAGS) $@ $(filter %.o, $^)
$(DIR_OBJS)/%.o: $(DIR_OBJS) %.c
$(CC) $(INC_DIRS) -o $@ -c $(filter %.c, $^)
$(DIR_DEPS)/%.dep: $(DIR_DEPS) %.c
@echo "Making $@ ..."
set -e ; \
$(RM) $(RMFLAGS) $@.tmp ; \
$(CC) $(INC_DIRS) -E -MM $(filter %.c, $^) > $@.tmp ; \
    sed 's,\(.*\)\.o[ :]*,objs/\1.o $@: ,g' < $@.tmp > $@ ; \
$(RM) $(RMFLAGS) $@.tmp
clean:
$(RM) $(RMFLAGS) $(RMS)
```

source/foo/src/Makefile

```
EXE =
LIB = libfoo.a
INC_DIRS = $(ROOT)/source/foo/inc
LINK_LIBS =
include $(ROOT)/build/make.rule
```

source/huge/src/Makefile

```
EXE = huge
LIB =
INC_DIRS = $(ROOT)/source/foo/inc
LINK_LIBS = foo
include $(ROOT)/build/make.rule
```

图 3.15

这次的改动包含：

- 在 make.rule 文件中增加了 LINK_LIBS 变量（的引用），这一变量用来存放所有需要在连接时用到的库。
- 同样在 make.rule 中将 DIR_LIBS 变量加入到连接器的搜索目录中去，这通过使用 gcc 的-L 选项做到。由于我们采用将所有的库文件都放入$(DIR_LIBS)目录中，现在看来这种方式能简化 Makefile 的设计，因为我们不需要指定多个目录。
- 在各模块的 src 目录中的 Makefile 中，增加了 LINK_LIBS 变量（的定义），且在 source/huge/src/Makefile 中对 LINK_LIBS 赋值为 foo。在 Linux 中，一个库名的格式为 libxxxx.a（或.so），其中的 xxxx 就是我们采用 gcc 的-l 选项时所需给的名。在这个 Makefile中，LINK_LIBS 变量的 foo 值就是表示 libfoo.a 库。

图 3.16 是更改后的编译结果，现在 huge 可执行程序能正确的生成了。在结果的最后，你也可以看到运行 huge 可执行程序的输出结果。

执行

```
目录 /Makefile/huge/source/huge/src
$make
/Makefile/huge/build/make.rule:43: deps/main.dep: No such file or directory
mkdir deps
Making deps/main.dep ...
mkdir objs
gcc –I/Makefile/huge/source/foo/inc –o objs/main.o –c main.c
gcc –L/Makefile/huge/build/libs –o /Makefile/huge/build/exes/huge objs/main.o –lfoo
目录 /Makefile/huge/source/huge/src
$$ROOT/build/exes/huge
This is foo ()!
```

图 3.16

至此，我们已经完成了各模块 src 目录下面的 Makefile 文件的创建工作。此时，假设我们想往huge 项目中增加一个 libbar.a 库，或说 bar 模块。让我们看一看对于新增模块的 Makfile 需要做些什么工作，以此，也可以检验我们的编译系统设计得如何。假设我们在图 3.1 的目录结构的基础上增加如图 3.17 所示的目录结构，用于存放 bar 模块的源程序。bar 模块的源程序及 Makefile 则如图3.18 所示。

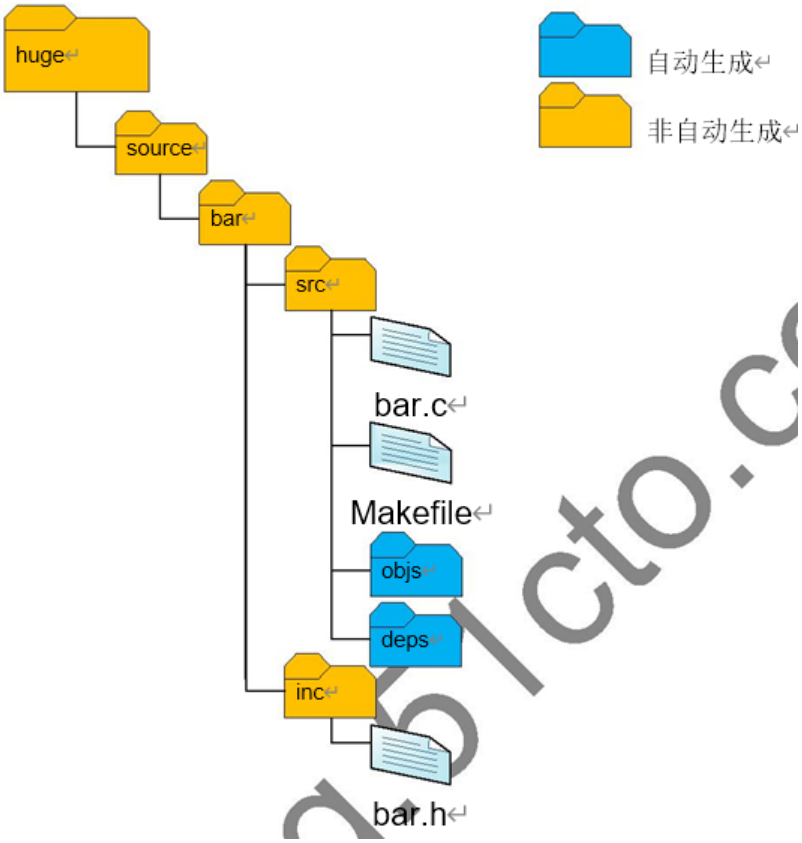


图 3.17

为了构建 libbar.a，我们需要进入 source/bar/src 目录运行 make 命令，结果如图 3.19 所示。

不可思义吧！当需要增加一个软件模块时，我们在 Makefile 方面需要做的工作非常的少。只需将已存在的一个 Makefile 拷贝过来，然后小改一下就好了，这是一个好的程序编译环境应当具备的一个特征，这一点 huge 项目做到了！

source/bar/inc/bar.h

```
#ifndef __BAR_H
#define __BAR_H
void bar ();
#endif
```

source/bar/src/bar.c

```
#include <stdio.h>
#include “bar.h”
void bar ()
{
    printf (“This is bar ()!\n”);
}
```

source/bar/src/Makefile

```
EXE =
LIB = libbar.a
```

```
INC_DIRS = $(ROOT)/source/bar/inc
LINK_LIBS =
include $(ROOT)/build/make.rule
```

图 3.18

执行

```
目录 /Makefile/huge/source/bar/src
$make
/Makefile/huge/build/make.rule:43: deps/bar.dep: No such file or directory
mkdir deps
Making deps/bar.dep ...
mkdir objs
gcc –I/Makefile/huge/source/bar/inc –o objs/bar.o –c bar.c
ar crs /Makefile/huge/build/libs/libbar.a objs/bar.o

目录 /Makefile/huge/source/bar/src
$ls $ROOT/build/libs
libbar.a libfoo.a
```

图 3.19

是时候看一看我们的可执行程序部分了，在进入 source/huge/src 目录之后进行编译之前，我们需要对main.c和其 Makefile进行一点改动，以使其使用新增加的 bar库。更改后的 main.c和 Makefile如图 3.20 所示。

source/huge/src/main.c

```
#include “foo.h”
#include “bar.h”
int main ()
{
    foo ();
    bar ();
    return 0;
}
```

source/huge/src/Makefile

```
EXE = huge
LIB =
INC_DIRS = $(ROOT)/source/foo/inc \
    $(ROOT)/source/foo/inc
LINK_LIBS = foo bar
include $(ROOT)/build/make.rule
```

图 3.20

图 3.21 示例了 huge 可执行程序的编译和运行结果。回过头来看一看图 3.20 中 Makefile 的改动，你会发现为了给 huge 可执行程序增加一个库，改动非常的小。这又一次证明了 huge 项目的编译环境具有很好的可维护性。

执行

```
目录 /Makefile/huge/source/huge/src
$make
Making deps/main.dep ...
gcc –I/Makefile/huge/source/foo/inc –I/Makefile/huge/source/bar/inc –o objs/main.o –c main.c
gcc –L/Makefile/huge/build/libs –o /Makefile/huge/build/exes/huge objs/main.o –lfoo –lbar
目录 /Makefile/huge/source/huge/src
$$ROOT/build/exes/huge
This is foo ()!
This is bar ()!
```

图 3.21

3.4 简化操作

至此，所有源程序目录下的 Makefile 都准备好了。为了编译 huge 项目，我们需要进入不同的目录进行 make，这并不是一件容易事。下面，我们得考虑如何简化 huge 项目的编译，还记得图 3.1中我们所规划的 build 目录下面的 Makefile 吗？我们现在就是要完成这个 Makefile 的内容，从而简化 huge 项目的编译工作。

这一次我不打算一步一步的介绍这个 Makefile 是如何设计出来的的，而是直接列出在图 3.22。以我们现在的水平，我想你可以明白这个 Makefile 在干什么。其中有一点需要指出的是，在这个 Makefile 中，使用了 Shell 中的 for 语句，用于遍历变量 DIRS 中的每一个目录，并进入目录运行make 命令。在 1.5.2 节中我们提到了 MAKE 特殊变量，在这个 Makefile 中我们就用到了它。之所以用它，而不直接使用 make 是为了更好的移植性。另外，还有一点需要注意的是，由于库必须比可执行程序先构建出来，所以在 DIRS 变量中，我们必需将库目录放在可执行程序的目录之前，因为 Makefile 是根据目录的先后顺序来进行构建工作的。

source/build/Makefile

```
.PHONY: all clean

DIRS = $(ROOT)/source/foo/src \
    $(ROOT)/source/bar/src \
    $(ROOT)/source/huge/src
RM = rm
RMFLAGS = –fr
RMS = $(ROOT)/build/exes $(ROOT)/build/libs

all:
    @set –e; \
```

```
    for dir in $(DIRS); \
        do \
        cd $$dir && $(MAKE) ; \
    done
    @echo ""
    @echo ":-) Completed"
    @echo ""
clean:
    @set -e; \
    for dir in $(DIRS); \
        do \
        cd $$dir && $(MAKE) clean;\
    done
    $(RM) $(RMFLAGS) $(RMS)
    @echo ""
    @echo ":-) Completed"
    @echo ""
```

图 3.22

这个 Makefile 的结果如何？图 3.23 给出了答案。结果证明，这个 Makefile 的确是简化了我们的工作，我们只要以在 build 目录下运行 make 命令就可以构建整个项目。此外，也可以在各个软件模块的 src 目录下单独构建各个模块。你可能会问，我们这里所有的库都是静态库，每一个库的重新构建都需要重新生成 huge 可执行文件，因此，单独构建某一个模块似乎没有意义。是的，但是如果我们的库采用的是动态库（这个工作留给你了□），那么重新构建某个单独的模块就很有意义了，尤其是大型的项目。因为，在很多情况下，动态库是可以单独构建的，对其的重新构建并不需要对主程序进行重构。

执行

```
目录 /Makefile/build
$make
make[1]: Entering directory `/Makefile/huge/source/foo/src'
/Makefile/huge/build/make.rule:43: deps/foo.dep: No such file or directory
mkdir deps
Making deps/foo.dep ...
make[1]: Leaving directory `/Makefile/huge/source/foo/src'
make[1]: Entering directory `/Makefile/huge/source/foo/src'
mkdir /Makefile/huge/build/libs
mkdir objs
gcc -I/Makefile/huge/source/foo/inc -o objs/foo.o -c foo.c
ar crs /Makefile/huge/build/libs/libfoo.a objs/foo.o
make[1]: Leaving directory `/Makefile/huge/source/foo/src'
make[1]: Entering directory `/Makefile/huge/source/bar/src'
/Makefile/huge/build/make.rule:43: deps/bar.dep: No such file or directory
```

```
mkdir deps
Making deps/bar.dep ...
make[1]: Leaving directory `/Makefile/huge/source/bar/src'
make[1]: Entering directory `/Makefile/huge/source/bar/src'
mkdir objs
gcc -I/Makefile/huge/source/bar/inc -o objs/bar.o -c bar.c
ar crs /Makefile/huge/build/libs/libbar.a objs/bar.o
make[1]: Leaving directory `/Makefile/huge/source/bar/src'
make[1]: Entering directory `/Makefile/huge/source/huge/src'
/Makefile/huge/build/make.rule:43: deps/main.dep: No such file or directory
mkdir deps
Making deps/main.dep ...
make[1]: Leaving directory `/Makefile/huge/source/huge/src'
make[1]: Entering directory `/Makefile/huge/source/huge/src'
mkdir /Makefile/huge/build/exes
mkdir objs
gcc -I/Makefile/huge/source/foo/inc -I/Makefile/huge/source/bar/inc -o objs/main.o -c
main.c
gcc -L/Makefile/huge/build/libs -o /Makefile/huge/build/exes/huge objs/main.o -lfoo -lbar
make[1]: Leaving directory `/Makefile/huge/source/huge/src'
:-) Completed
目录 /Makefile/build
$make clean
make[1]: Entering directory `/Makefile/huge/source/foo/src'
rm -fr objs deps /Makefile/huge/build/libs/libfoo.a
make[1]: Leaving directory `/Makefile/huge/source/foo/src'
make[1]: Entering directory `/Makefile/huge/source/bar/src'
rm -fr objs deps /Makefile/huge/build/libs/libbar.a
make[1]: Leaving directory `/Makefile/huge/source/bar/src'
make[1]: Entering directory `/Makefile/huge/source/huge/src'
rm -fr objs deps /Makefile/huge/build/exes/huge
make[1]: Leaving directory `/Makefile/huge/source/huge/src'
rm -fr /Makefile/huge/build/exes /Makefile/huge/build/libs
:-) Completed
```

图 3.23

在 build 目录中的运行 make，只要我们看到一张笑脸出现在终端上，我们的项目就是一定构建成功了，否则一定是出了错，而我们的 Makefile 告诉了 make“请在第一时间停止在第一个出错的地方”，以等待我们去解决它！

完整编译之前：

```
cd Makefile/huge
```

```
export ROOT=`pwd`
cd build
make
```

3.5 小结

huge 项目中的 Makefile 是一个真正实用的实现，其完全可以被运用到大型项目中去。到此，你已经学会了如何一步一步的构建一个真正实用的软件编译环境了！

该文根据 李云的 《驾驭Makefile》整理和修改。

4 生成动态库静态库和调用

参考 src\Makefile\template项目的代码：
shared：生成静态库
static：生成静态库
callshared：调用动态库
callstatic：调用静态库

参考

该文根据 李云的 《驾驭Makefile》整理和修改。
makefile中PHONY的重要性 https://www.cnblogs.com/fengliu-/p/10217198.html
Makefile 编译动态库文件及链接动态库
https://blog.csdn.net/wanglf1986/article/details/78725553