Warring：由于 PDF 文档不支持 Tab 字符，因此本文档内的 Makefile示例 不能直接使用，网页版可以直接使用：

```
https://github.com/1184893257/Tutorial_of_Make/blob/master/tutorial.md
```

深入学习Make命令和Makefile（上）

文章转载自网管之家： http://www.bitscn.com/os/linux/200806/143482.html

make是Linux下的一款程序自动维护工具，配合makefile的使用，就能够根据程序中模块的修改情况，自动判断应该对那些模块重新编译，从而保证软件是由最新的模块构成。本文分为上下两部分，我们将紧紧围绕make在软件开发中的应用展开详细的介绍。

一、都是源文件太多惹得祸

当我们在开发的程序中涉及众多源文件时，常常会引起一些问题。首先，如果程序只有两三个源文件，那么修改代码后直接重新编译全部源文件就行了，但是如果程序的源文件较多，这种简单的处理方式就有问题了。

设想一下，如果我们只修改了一个源文件，却要重新编译所有源文件，那么这显然是在浪费时间。其次，要是只重新编译那些受影响的文件的话，我们又该如何确定这些文件呢？比如我们使用了多个头文件，那么它们会被包含在各个源文件中，修改了某些头文件后，那些源文件受影响，哪些与此无关呢？如果采取拉网式大检查的话，可就费劲了。

由此可以看出，源文件多了可真是件让人头疼的事。幸运的是，实用程序make可以帮我们解决这两个问题——当程序的源文件改变后，它能保证所有受影响的文件都将重新编译，而不受影响的文件则不予编译，这真是太好了。

二、Make程序的命令行选项和参数

我们知道，make程序能够根据程序中各模块的修改情况，自动判断应对哪些模块重新编译，保证软件是由最新的模块构建的。至于检查哪些模块，以及如何构建软件由makefile文件来决定。

虽然make可以在makefile中进行配置，除此之外我们还可以利用make程序的命令行选项对它进行即时配置。 Make命令参数的典型序列如下所示：

make [-f makefile**文件名**][**选项**][**宏定义**][**目标**]

这里用[]括起来的表示是可选的。命令行选项由破折号“-”指明，后面跟选项，如

```
make -e
```

如果需要多个选项，可以只使用一个破折号，如

```
make -kr
```

也可以每个选项使用一个破折号，如

```
make -k -r
```

甚至混合使用也行，如

```
make -e -kr
```

Make命令本身的命令行选项较多，这里只介绍在开发程序时最为常用的三个，它们是：

-k：

如果使用该选项，即使make程序遇到错误也会继续向下运行；如果没有该选项，在遇到第一个错误时make程序马上就会停止，那么后面的错误情况就不得而知了。我们可以利用这个选项来查出所有有编译问题的源文件。

-n：

该选项使make程序进入非执行模式，也就是说将原来应该执行的命令输出，而不是执行。

-f：

指定作为makefile的文件的名称。 如果不用该选项，那么make程序首先在当前目录查找名为makefile的文件，如果没有找到，它就会转而查找名为Makefile的文件。如果您在Linux下使用GNU Make的话，它会首先查找GNUmakefile，之后再搜索makefile和Makefile。按照惯例，许多Linux程序员使用Makefile，因为这样能使Makefile出现在目录中所有以小写字母命名的文件的前面。所以，最好不要使用GNUmakefile这一名称，因为它只适用于make程序的GNU版本。

当我们想构建指定目标的时候，比如要生成某个可执行文件，那么就可以在make命令行中给出该目标的名称；如果命令行中没有给出目标的话， make命令会设法构建makefile中的第一个目标。我们可以利用这一特点，将all作为makefile中的第一个目标，然后将默认目标作为all所依赖的目标，这样，当命令行中没有给出目标时，也能确保它会被构建。

三、Makefile概述

上面提到，make命令对于构建具有多个源文件的程序有很大的帮助。事实上，只有make命令还是不够的，前面说过还必用须makefile告诉它要做什么以及怎么做才行，对于程序开发而言，就是告诉make命令应用程序的组织情况。

我们现在对makefile的位置和数量简单说一下。一般情况下，makefile会跟项目的源文件放在同一个目录中。另外，系统中可以有多个makefile，一般说来一个项目使用一个makefile就可以了；如果项目很大的话，我们就可以考虑将它分成较小的部分，然后用不同的makefile来管理项目的不同部分。

make命令和Makefile配合使用，能给我们的项目管理带来极大的便利，除了用于管理源代码的编译之外，还用于建立手册页，同时还能将应用程序安装到指定的目录。

因为Makefile用于描述系统中模块之间的相互依赖关系，以及产生目标文件所要执行的命令，所以，一个makefile由依赖关系和规则两部分内容组成。下面分别加以解释。

依赖关系由一个目标和一组该目标所依赖的源文件组成。这里所说的目标就是将要创建或更新的文件，最常见的是可执行文件。规则用来说明怎样使用所依赖得文件来建立目标文件。

当make命令运行时，会读取makefile来确定要建立的目标文件或其他文件，然后对源文件的日期和时间进行比较，从而决定使用那些规则来创建目标文件。一般情况下，在建立起最终的目标文件之前，肯定免不了要建立一些中间性质的目标文件。这时， Make命令也是使用makefile来确定这些目标文件的创建顺序，以及用于它们的规则序列。

四、makefile中的依赖关系

make程序自动生成和维护通常是可执行模块或应用程序的目标，目标的状态取决于它所依赖的那些模块的状态。 Make的思想是为每一块模块都设置一个时间标记，然后根据时间标记和依赖关系来决定哪一些文件需要更新。一旦依赖模块的状态改变了， make就会根据时间标记的新旧执行预先定义的一组命令来生成新的目标。

依赖关系规定了最终得到的应用程序跟生成它的各个源文件之间的关系。如下面的图1描述了可执行文件main对所有的源程序文件及其编译产生的目标文件之间的依赖关系，见下图：

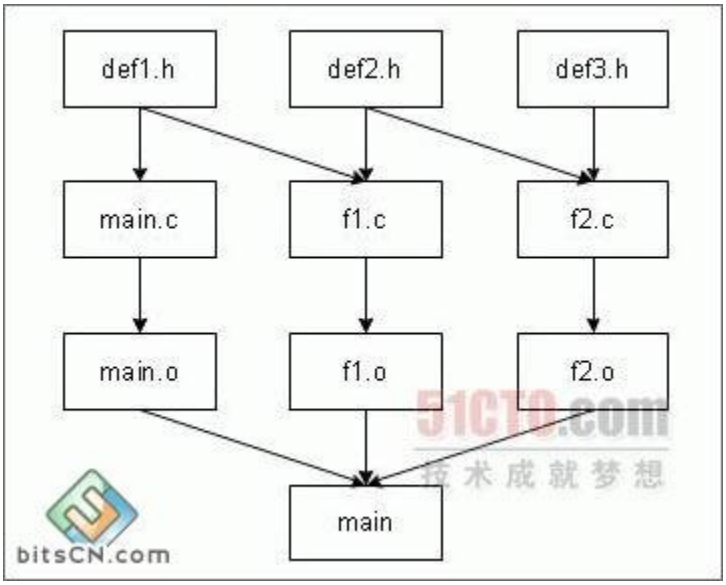

图1 模块间的依赖关系

就图1而言，我们可以说可执行程序main依赖于main.o、f1.o和ff1.o。与此同时，main.o依赖于main.c和def1.h； f1.o依赖于f1.c、def1.h和def2.h；而ff1.o则依赖于ff1.c、def2.h和def3. h。在makefile中，我们可以用目标名称，加冒号，后跟空格键或tab键，再加上由空格键或tab键分隔的一组用于生产目标模块的文件来描述模块之间的依赖关系。对于上例来说，可以作以下描述：

```
main: main.o f1.o f2.o
main.o: main.c def1.h
f1.o: f1.c def1.h def2.h
f2.o: f2.c def2.h def3.h
```

不难发现，上面的各个源文件跟各模块之间的关系具有一个明显的层次结构，如果def2.h发生了变化，那么就需要更新f1.o和f2.o，而f1.o和f2.o发生了变化的话，那么main也需要随之重新构建。

默认时，make程序只更新makefile中的第一个目标，如果希望更新多个目标文件的话，可以使用一个特殊的目标all，假如我们想在一个makefile中更新main和hello这两个程序文件的话，可以加入下列语句达到这个目的：

```
all: main hello
```

五、makefile中的规则

除了指明目标和模块之间的依赖关系之外， makefile还要规定相应的规则来描述如何生成目标，或者说使用哪些命令来根据依赖模块产生目标。就上例而言，当make程序发现需要重新构建f1.o的时候，该使用哪些命令来完成呢？很遗憾，到目前为止，虽然make知道哪些文件需要更新，但是却不知道如何进行更新，因为我们还没有告诉它相应的命令。

当然，我们可以使用命令gcc -c f1.c来完成，不过如果我们需要规定一个include目录，或者为将来的调试准备符号信息的话，该怎么办呢？所有这些，都需要在makefile中用相应规则显式地指出。

实际上，makefile是以相关行为基本单位的，相关行用来描述目标、模块及规则（即命令行）三者之间的关系。一个相关行格式通常为：冒号左边是目标（模块）名；冒号右边是目标所依赖的模块名；紧跟着的规则（即命令行）是由依赖模块产生目标所使用的命令。相关行的格式为：

```
目标：[依赖模块][;命令]
```

习惯上写成多行形式，如下所示：

```
目标：[依赖模块]
        命令
        命令
```

需要注意的是，如果相关行写成一行，“命令”之前用分号“；”隔开，如果分成多行书写的话，后续的行务必以tab字符为先导。 对于makefile而言，空格字符和tab字符是不同的。所有规则所在的行必须以tab键开头，而不是空格键。初学者一定对此保持警惕，因为这是新手最容易疏忽的地方，因为几个空格键跟一个tab键在肉眼是看不出区别的，但make命令却能明察秋毫。

此外，如果在makefile文件中的行尾加上空格键的话，也会导致make命令运行失败。所以，大家一定要小心了，免得耽误许多时间。

六、Makefile文件举例

根据图1的依赖关系，这里给出了一个完整的makefile文件，这个例子很简单，由四个相关行组成，我们将其命名为mymakefile1。文件内容如下所示：

```
main: main.o f1.o f2.o
        gcc -o main main.o f1.o f2.o
```

```
main.o: main.c def1.h
        gcc -c main.c
f1.o: f1.c def1.h def2.h
        gcc -c f1.c
f2.o: f2.c def2.h def3.h
        gcc -c f2.c
```

注意 ，由于我们这里没有使用缺省名makefile 或者Makefile，所以一定要在make命令行中加上-f选项。如果在没有任何源码的目录下执行命令“make -f Mymakefile1”的话，将收到下面的消息：

```
make: *** No rule to make target ‘main.c’, needed by ‘main.o’. Stop.
```

Make命令将makefile中的第一个目标即main作为要构建的文件，所以它会寻找构建该文件所需要的其他模块，并判断出必须使用一个称为main.c的文件。因为迄今尚未建立该文件，而makefile又不知道如何建立它，所以只好报告错误。好了，现在建立这个源文件，为简单起见，我们让头文件为空，创建头文件的具体命令如下：

```
$ touch def1.h
$ touch def2.h
$ touch def3.h
```

我们将main函数放在main.c文件中，让它调用function2和function3，但将这两个函数的定义放在另外两个源文件中。由于这些源文件含有#include命令，所以它们肯定依赖于所包含的头文件。如下所示：

```
/* main.c */
#include <stdio.h>
#include "def1.h"
extern void function2();
extern void function3();
int main()
{
    function2();
    function3();
    return 0;
}

/* f1.c */
#include "def1.h"
#include "def2.h"
void function2() {
}

/* f2.c */
#include "def2.h"
#include "def3.h"
void function3() {
```

```
}
```

建好源代码后，再次运行make程序，看看情况如何：

```
$ make -f Mymakefile1
gcc -c main.c
gcc -c f1.c
gcc -c f2.c
gcc -o main main.o f1.o f2.o
$
```

好了，这次顺利通过了。这说明Make命令已经正确处理了 makefile描述的依赖关系，并确定出了需要建立哪些文件，以及它们的建立顺序。虽然我们在makefile 中首先列出的是如何建立main，但是make还是能够正确的判断出这些文件的处理顺序，并按相应的顺序调用规则部分规定的相应命令来创建这些文件。当这些命令执行时，make程序会按照执行情况来显示这些命令。

如今，我们对def2.h加以变动，来看看makefile能否对此作出相应的回应：

```
$ touch def2.h
$ make -f Mymakefile1
gcc -c f1.c
gcc -c f2.c
gcc -o main main.o f1.o f2.o
$
```

这说明，当Make命令读取makefile 后，只对受def2.h的变化的影响的模块进行了必要的更新，注意它的更新顺序，它先编译了C程序，最后连接生产了可执行文件。现在，让我们来看看删除目标文件后会发生什么情况，先执行删除，命令如下：

```
$ rm f1.o
```

然后运行make命令，如下所示：

```
$ make -f Mymakefile1
gcc -c f1.c
gcc -o main main.o f1.o f2.o
$
```

很好，make的行为让我们非常满意。

七、makefile中的宏

在makefile中可以使用诸如XLIB、UIL等类似于Shell变量的标识符，这些标识符在makefile中称为“宏”，它可以代表一些文件名或选项。宏的作用类似于C语言中的define，利用它们来代表某些多处使用而又可能发生变化的内容，可以节省重复修改的工作，还可以避免遗漏。

Make的宏分为两类，一类是用户自己定义的宏，一类是系统内部定义的宏。用户定义的宏必须在makefile或命令行中明确定义，系统定义的宏不由用户定义。我们首先介绍第一种宏。

这里是一个包含宏的makefile文件，我们将其命名为mymakefile2，如下所示：

```
all: main
# 使用的编译器
CC = gcc
#包含文件所在目录
INCLUDE = .
# 在开发过程中使用的选项
CFLAGS = -g -Wall -ansi
# 在发行时使用的选项
# CFLAGS = -O -Wall -ansi
main: main.o f1.o f2.o
        $(CC) -o main main.o f1.o f2.o
main.o: main.c def1.h
        $(CC) -I$(INCLUDE) $(CFLAGS) -c main.c
f1.o: f1.c def1.h def2.h
        $(CC) -I$(INCLUDE) $(CFLAGS) -c f1.c
f2.o: f2.c def2.h def3.h
        $(CC) -I$(INCLUDE) $(CFLAGS) -c f2.c
```

我们看到，在这里有一些注释。在makefile中，注释以#为开头，至行尾结束。注释不仅可以帮助别人理解我们的makefile，如果时间久了，有些东西我们自己也会忘掉，它们对makefile的编写者来说也是很有必要的。

现在言归正传，先看一下宏的定义。我们既可以在make命令行中定义宏，也可以在makefile中定义宏。在makefile中定义宏的基本语法是：

```
宏标识符=值列表
```

其中，宏标识符即宏的名称通常全部大写，但它实际上可以由大、小写字母、阿拉伯数字和下划线构成。等号左右的空白符没有严格要求，因为它们最终将被make删除。至于值列表，既可以是零项，也可以是一项或者多项。如：

```
LIST_VALUE = one two three
```

当一个宏定义之后，我们就可以通过$(宏标识符)或者${宏标识符}来访问这个标识符所代表的值了。

在makefile中，宏经常用作编译器的选项。很多时候，处于开发阶段的应用程序在编译时是不用优化的，但是却需要调试信息；而正式版本的应用程序却正好相反，没有调试信息的代码不仅所占内存较小，进过优化的代码运行起来也更快。

对于Mymakefile1来说，它假定所用的编译器是gcc，不过在其他的UNIX系统上，更常用的编译器是cc或者c89，而非gcc。如果你想让自己的makefile适用于不同的UNIX操作系统，或者在一个系统上使用其他

种类的编译器，这时就不得不对这个makefile中的多处进行修改。

但对于mymakefile2来说则不存在这个问题，我们只需修改一处，即宏定义的值就行了。除了在makefile中定义宏的值之外，我们还可以在make命令行中加以定义，如：

```
$ make CC=c89
```

当命令行中的宏定义跟makefile中的定义有冲突时，以命令行中的定义为准。当在makefile文件之外使用时，宏定义必须作为单个参数进行传递，所以要避免使用空格，但是更妥当的方法是使用引号，如：

```
$ make "CC =   c89"
```

这样就不必担心空格所引起的问题了。现在让我们将前面的编译结果删掉，来测试一下mymakefile2的工作情况。命令如下所示：

```
$ rm *.o main
$ make -f Mymakefile2
gcc -I. -g -Wall -ansi -c main.c
gcc -I. -g -Wall -ansi -c f1.c
gcc -I. -g -Wall -ansi -c f2.c
gcc -o main main.o f1.o f2.o
$
```

就像我们看到的那样，Make程序会用相应的定义来替换宏引用$(CC)、 $(CFLAGS)和$(INCLUDE)，这跟C语言中的宏的用法比较相似。

上面介绍了用户定义的宏，现在介绍make的内部宏。常用的内部宏有：

`$?` ：比目标的修改时间更晚的那些依赖模块表。
`$@` ：当前目标的全路径名。可用于用户定义的目标名的相关行中。
`$<` ：比给定的目标文件时间标记更新的依赖文件名。
`$*` ：去掉后缀的当前目标名。例如，若当前目标是`pro.o`，则`$*`表示`pro`。

八、小结

我们在本文中分别介绍了make程序的使用方法， makefile中的依赖关系及规则等基础知识，同时还介绍了一些常用的宏。在下篇文章中，我们会对makefile的高级功能做进一步的介绍。

深入学习Make命令和Makefile（下）

文章转载自网管之家： http://www.bitscn.com/os/linux/200806/143483.html

make是Linux下的一款程序自动维护工具，配合makefile的使用，就能够根据程序中模块的修改情况，自动判断应该对那些模块重新编译，从而保证软件是由最新的模块构成。

本文分为上下两部分，我们在上一篇文章中分别介绍了make和makefile 的一些基本用法，在本文中，我们会对make和makefile的功能做进一步的介绍。

一、构建多个目标

有时候，我们想要在一个makefile中生成多个单独的目标文件，或者将多个命令放在一起，比如，在下面的示例mymakefile3中我们将添加一个clean 选项来清除不需要的目标文件，然后用install选项将生成的应用程序移动到另一个目录中去。这个makefile跟前面的mymakefile较为相似，不同之处笔者用黑体加以标识：

```
all: main
# 使用的编译器
CC = gcc
# 安装位置
INSTDIR = /usr/local/bin
# include文件所在位置
INCLUDE = .
# 开发过程中所用的选项
CFLAGS = -g -Wall -ansi
# 发行时用的选项
# CFLAGS = -O -Wall -ansi
main: main.o f1.o f2.o
         $(CC) -o main main.o f1.o f2.o
main.o: main.c def1.h
         $(CC) -I$(INCLUDE) $(CFLAGS) -c main.c
f1.o: f1.c def1.h def2.h
         $(CC) -I$(INCLUDE) $(CFLAGS) -c f1.c
f2.o: f2.c def2.h def3.h
         $(CC) -I$(INCLUDE) $(CFLAGS) -c f2.c
clean:
         rm main.o f1.o f2.o
install: main
         @if [ -d $(INSTDIR) ]; \
         then \
         cp main $(INSTDIR);\
         chmod a+x $(INSTDIR)/main;\
         chmod og-w $(INSTDIR)/main;\
         echo "Installed in $(INSTDIR)";\
         else \
         echo "Sorry, $(INSTDIR) does not exist";\
         fi
```

在这个makefile中需要注意的是，虽然这里有一个特殊的目标all，但是最终还是将main作为目标。因此，如果执行make命令时没有在命令行中给出一个特定目标的话，仍然会编译连接main程序。

其次要注意后面的两个目标：clean和install。目标clean没有依赖模块，因为没有时间标记可供比较，

所以它总被执行；它的实际意图是引出后面的rm命令来删除某些目标文件。我们看到rm命令以-开头，这时即使表示make将忽略命令结果，所以即使没有目标供rm命令删除而返回错误时，make clean依然继续向下执行。

接下来的目标install依赖于main，所以make知道必须在执行安装命令前先建立main。用于安装的指令由一些shell命令组成。

因为make调用shell来执行规则，并且为每条规则生成一个新的shell，所以要用一个shell来执行这些命令的话，必须添加反斜杠，以使所有命令位于同一个逻辑行上。这条命令用@开头，表示在执行规则前不会向标准输出打印命令。

为了安装应用程序，目标install会一条接一条地执行若干命令，并且执行下一个之前，不会检查上一条命令是否成功。若想只有当前面的命令取得成功时，随后的命令才得以执行的话，可以在命令中加入&&，如下所示：

```
@if [ -d $(INSTDIR) ]; \
then \
cp main $(INSTDIR) &&\
chmod a+x $(INSTDIR)/main && \
chmod og-w $(INSTDIR/main && \
echo "Installed in $(INSTDIR)" ;\
else \
echo "Sorry, $(INSTDIR) does not exist" ; false ; \
fi
```

这是shell的“与”指令，只有当在前的命令成功时随后的命令才被执行。这里不必关心前面命令是否取得成功，只需注意这种用法就可以了。

要想在/usr/local/bin目录安装新命令必须具有特权，所以调用make install命令之前，可以让Makefile使用一个不同的安装目录，或者修改该目录的权限，或切换到root用户。如下所示：

```
$ rm *.o main
$ make -f Mymakefile3
gcc -I. -g -Wall -ansi -c main.c
gcc -I. -g -Wall -ansi -c f1.c
gcc -I. -g -Wall -ansi -c f2.c
gcc -o main main.o f1.o f2.o
$ make -f Mymakefile3
make: Nothing to be done for 'all'.
$ rm main
$ make -f Mymakefile3 install
gcc -o main main.o f1.o f2.o
Installed in /usr/local/bin
$ make -f Mymakefile3 clean
rm main.o f1.o f2.o
$
```

让我们对此作一简单介绍，首先删除main和所有目标文件程序，由于将all作为目标，所以make命令会重新编译main。当我们再次执行make命令时，由于main是最新的，所以make什么也不做。之后，我们删除main程序文件，并执行make install，这会重新建立二进制文件main并将其复制到安装目录。最后，运行make clean命令，来删去所有目标程序。

二、内部规则

迄今为止，我们已经能够在makefile中给出相应的规则来指出具体的处理过程。实际上，除了我们显式给出的规则外，make还具有许多内部规则，这些规则是由预先规定的目标、依赖文件及其命令组成的相关行。在内部规则的帮助下，可以使makefile变得更加简洁，尤其是在具有许多源文件的时候。现在以实例加以说明，首先建立一个名为foo.c的C程序源文件，文件内容如下所示：

```
#include <stdio.h>
int main()
{
    printf("Hello World\n");
    return 0;
}
```

现在让我们用make命令来编译它：

```
$ make foo
cc foo.c -o foo
$
```

您会惊奇地发现，尽管我们没有指定makefile，但是make仍然能知道如何调用编译器，并且调用的是cc而不是gcc编译器。这在Linux上没有问题，因为cc常常会链接到gcc程序。这完全得益于make内建的内部规则，另外这些内部规则通常使用宏，所以只要为这些宏指定新的值，就可以改变内部规则的默认动作，如下所示：

```
$ rm foo
$ make CC=gcc CFLAGS="-Wall -g" foo
gcc -Wall -g foo.c -o foo
$
```

用make命令加-p选项后，可以打印出系统缺省定义的内部规则。它们包括系统预定义的宏、以及产生某些种类后缀的文件的内部相关行。内部规则涉及的文件种类很多，它不仅包括C源程序文件及其目标文件，还包括SCCS文件、yacc文件和lex文件，甚至还包括Shell文件。

当然，我们更关心的是如何利用内部规则来简化makefile，比如让内部规则来负责生成目标，而只指定依赖关系，这样makefile就简洁多了，如下所示：

```
main.o: main.c def1.h
f1.o: f1.c def1.h def2.h
f2.o: f2.c def2.h def3.h
```

三、后缀规则

前面我们已经看到，有些内部规则会根据文件的后缀（相当于Windows系统中的文件扩展名）来采取相应的处理。换句话说，这样当make见到带有一种后缀的文件时，就知道使用哪些规则来建立一个带有另外一种后缀的文件，最常见的是用以.c结尾的文件来建立以.o结尾的文件，即把源文件编译成目标程序，但是不连接。

现在举例说明后缀规则的应用。有时候，我们需要在不同的平台下编译源文件，例如Windows和Linux。假设我们的源代码是C++编写的，那么Windows下其后缀则为.cpp。不过Linux使用的make版本没有编译.cpp文件的内部规则，倒是有一个用于.cc的规则，因为在UNIX操作系统中c++文件扩展名通常为.cc。

这时候，要么为每个源文件单独指定一条规则，要么为make建立一条新规则，告诉它如何用.cpp 为扩展名的源文件来生成目标文件。如果项目中的源文件较多的话，后缀规则就可以派上用场了。要添加一条新后缀规则，首先在makefile文件中加入一行来告诉make新后缀是什么；然后就可以添加使用这个新后缀的规则了。这时，make要用到一条专用的语法：

```
.<旧后缀名>.<新后缀名>:
```

它的作用是定义一条通用规则，用来将带有旧后缀名的文件变成带有新后缀名的文件，文件名保持不变，如要将.cpp文件编译成.o文件，可以使用一个新的通用规则：

```
.SUFFIXES: .cpp
.cpp.o:
        $(CC) -xc++ $(CFLAGS) -I$(INCLUDE) -c $<
```

上面的".cpp .o:"告诉make 这些规则用于把后缀为.cpp 的文件转换成后缀为.o的文件。其中的标志"-xc++"的作用是告诉gcc 这次要编译的源文件是c++源文件。这里，我们使用一个宏$<来通指需要编译的文件的名称，不管这些文件名具体是什么。我们只需知道，所有以.cpp为后缀的文件将被编译成以.o为后缀的文件，例如以是app.cpp的文件将变成app.o。

注意，我们只跟make说明如何把.cpp文件变成.o文件就行了，至于如何从目标程序文件变成二进制可执行文件，因为make早已知晓，所以就不用我们费心了。所以，当我们调用make程序时，它会使用新规则把类似app.cpp这样的程序变成app.o，然后使用内部规则将app.o文件连接成一个可执行文件app。

现在，make已经知道如何处理扩展名为.cpp的c++源文件，除此之外，我们还可以通过后缀规则将文件从一种类型转换为另一种类型。不过，较新版本的make包含一个语法可以达到同样的效果。例如，模式规则使用%作为匹配文件名的通配符，而不单独依赖于文件扩展名。以下模式规则相当于上面处理.cpp的规则，具体如下所示：

```
%.o: %.cpp
        $(CC) -xc++ $(CFLAGS) -I$(INCLUDE) -c $<
```

四、用make管理程序库

一般来说，程序库也是一种由一组目标程序构成的以.a为扩展名的文件，所以，Make命令也可以用来管理这些程序库。实际上，为了简化程序库的管理，make程序还专门设有一个语法： lib (file.o)，这意味着目标文件file.o以库文件lib.a的形式存放，这意味着lib.a库依赖于目标程序file.o。此外， make命令还具有一个内部规则用来管理程序库，该规则相当于如下内容：

```
.c.a:
        $(CC) -c $(CFLAGS) $<
        $(AR) $(ARFLAGS) $@ $*.o
```

其中，宏$(AR)和$(ARFLAGS)分别表示指令AR和选项rv。如上所见，要告诉make用.c文件生成.a库，必须用到两个规则：第一个规则是说把源文件编译成一个目标程序文件。第二个规则表示使用ar 指令向库中添加新的目标文件。

所以，如果我们有一个名为filed的库，其中含有bar.o文件，那么第一规则中的$<会被替换为bar.c；在第二个规则中的$@被库名filed.a所替代，而$*将被bar所替代。

下面举例说明如何用make来管理库。实际上，用make来管理程序库的规则是很简单的。比如，我们可以将前面示例加以修改，让f1.o和f2.o放在一个称为mylib.a的程序库中，这时的Makefile几乎无需改变，而新的mymakefile4看上去是这样的：

```
all: main
# 使用的编译器
CC = gcc
# 安装位置
INSTDIR = /usr/local/bin
# include文件所在位置
INCLUDE = .
# 开发过程中使用的选项
CFLAGS = -g -Wall -ansi
# 用于发行时的选项
# CFLAGS = -O -Wall -ansi
# 本地库
MYLIB = mylib.a
main: main.o $(MYLIB)
        $(CC) -o main main.o $(MYLIB)
$(MYLIB): $(MYLIB)(f1.o) $(MYLIB)(f2.o)
main.o: main.c def1.h
f1.o: f1.c def1.h def2.h
f2.o: f2.c def2.h def3.h
clean:
        rm main.o f1.o f2.o $(MYLIB)
install: main
        @if [ -d $(INSTDIR) ]; \
        then \
        cp main $(INSTDIR);\
        chmod a+x $(INSTDIR)/main;\
```

```
        chmod og-w $(INSTDIR)/main;\
        echo “Installed in $(INSTDIR)“;\
        else \
        echo “Sorry, $(INSTDIR) does not exist”;\
        fi
```

注意：我们是如何让省缺规则来替我们完成大部分工作的。如今，我们可以试一下新版的makefile的工作情况：

```
$ rm -f main *.o mylib.a
$ make -f Mymakefile4
gcc -g -Wall -ansi -c -o main.o main.c
gcc -g -Wall -ansi -c -o f1.o f1.c
ar rv mylib.a f1.o
a - f1.o
gcc -g -Wall -ansi -c -o f2.o f2.c
ar rv mylib.a f2.o
a - f2.o
gcc -o main main.o mylib.a
$ touch def3.h
$ make -f Mymakefile4
gcc -g -Wall -ansi -c -o f2.o f2.c
ar rv mylib.a f2.o
r - f2.o
gcc -o main main.o mylib.a
$
```

现在对上面的例子做必要的说明。首先删除全部目标程序文件和程序库，然后让make 重新构建main，因为当连接main.o时需要用到库，所以要先编译和创建库。此后，我们还测试f2.o的依赖关系，我们知道如果def3.h发生了改变，那么必须重新编译f2.c，事实表明make在重新构建main可执行文件之前，正确地编译了f2.c并更新了库。

五、Makefile和子目录

如果你的项目比较大的话，可以考虑将某些文件组成一个库，然后单独存放到一个子目录内。这时，对于makefile有两种处理方法，下面分别介绍。

第一种方法：在子目录中放置一个辅助makefile，然后把这个子目录中的源文件编译成一个程序库，最后将这个库复制到主目录中。上级目录中的主要makefile可以放上一个规则，通过调用辅助makefile来建立该库：

```
mylib.a:
        (cd mylibdirectory;$(MAKE))
```

这样的话，我们就会总是构建mylib.a，因为冒号右边为空。当make调用该规则构建该库时，它会切换到子目录mylibdirectory中，然后调用一个新的make命令来管理该库。因为调用了一个新的shell来完成此任

务，所以使用makefile 的程序不必进行目录切换。不过，被调用的shell是在一个不同的目录中利用该规则构建该库的，所以括弧能确保所有处理都是由一个shell完成的。

第二种方法：在单个makefile中使用更多的宏，不过这些附加的宏需要在目录名上加D并且为文件名加上F。例如，可以用下面的命令来覆盖内建的.c.o后缀规则：

```
.c.o:
        $(CC) $(CFLAGS) -c $(@D)/$<
```

为在子目录编译文件，并将目标放在子目录中，可以用像下面这样的依赖关系和规则来更新当前目录中的库：

```
mylib.a: mydir/f1.o mydir/f2.o
        ar -rv mylib.a $?
```

上述两种方法都是可行的，至于使用哪一种，需要根据您的项目的具体情况来决定。

六、GNU make和gcc的有关选项

如果您当前正在使用GNU make 和GNU gcc编译器的话，那么它们还分别有一个额外的选项可以使用，下面分别加以说明。

我们首先介绍用于make程序的-jN 选项。这个选项允许make同时执行N条命令。这样的话，就可以将该项目的多个部分单独进行编译，make将同时调用多个规则。如果具有许多源文件的话，这样做能够节约大量编译时间。

其次，gcc还有一个-MM选项可用，该选项会为make生成一个依赖关系表。在一个含有大量源文件的项目中，很可能每个源文件都包含一组头文件，而头文件有时又会包含其它头文件，这时正确区分依赖关系就比较难了。这时为了防止遗漏，最笨的方法就是让每个源文件依赖于所有头文件，但这显然没有必要；另一方面，如果你遗漏一些依赖关系的话，就根本就无法编译通过。这时，我们就可以用gcc的-MM选项来生成一张依赖关系表，例如：

```
$ gcc -MM main.c f1.c f2.c
main.o: main.c def1.h
f1.o: f1.c def1.h def2.h
f2.o: f2.c def2.h def3.h
$
```

这时，Gcc编译器会扫描所有源文件，并生产一张满足makefile格式要求的依赖关系表，我们只须将它保存到一个临时文件内，然后将其插入makefile即可。

七、小结

继上一篇文章之后，本文又对make和makefile的一些高级应用作了相应的介绍，至此，我们已经对make和makefile在程序开发中的应用有了一个较为全面的认识，希望本文能对读者的学习和工作有所帮助。