在 Linux环境下使用 GNU 的 make工具能够比较容易的构建一个属于你自己的工程，整个工程的编译只需要一个命令就可以完成编译、连接以至于最后的执行。不过这需要我们投入一些时间去完成一个或者多个称之为 Makefile 文件的编写。此文件正是 make 正常工作的基础。
make 是一个命令工具，它解释 Makefile 中的指令（应该说是规则）。在 Makefile文件中描述了整个工程所有文件的编译顺序、编译规则。
准备知识：编译，链接，静态库，共享库
编译：把高级语言所书写的代码转换成机器可识别的指令，此时还不能够被执行，编译器通过检查高级语言的语法，函数和变量的声明是否正确！如果正确则产生中间目标文件（目标文件在Liunx中默认后缀为“.o”）
链接：将多.o 文件，或者.o 文件和库文件链接成为可被操作系统执行的可执行程序
静态库：又称为文档文件（Archive File）。它是多个.o文件的集合。Linux中静态库文件的后缀为“.a”
共享库：也是多个.o 文件的集合，但是这些.o 文件时有编译器按照一种特殊的方式生成（共享库已经具备了可执行条件）

在执行 make 之前，需要一个命名为 Makefile 的特殊文件（本文的后续将使用Makefile 作为这个特殊文件的文件名）来告诉 make 需要做什么（完成什么任务），该怎么做。

当使用make 工具进行编译时，工程中以下几种文件在执行make 时将会被编译（重新编译）：
1.所有的源文件没有被编译过，则对各个 C 源文件进行编译并进行链接，生成最后的可执行程序；
2.每一个在上次执行 make 之后修改过的 C 源代码文件在本次执行make 时将会被重新编译；
3.头文件在上一次执行make 之后被修改。则所有包含此头文件的 C 源文件在本次执make 时将会被重新编译。

Makefile规则介绍

一个简单的 Makefile 描述规则组成：

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

target：规则的目标。通常是最后需要生成的文件名或者为了实现这个目的而必需的中间过程文件名。可以是.o文件、也可以是最后的可执行程序的文件名等。另外，目标也可以是一个make执行的动作的名称，如目标“clean”（目标“clean”不是一个文件，它仅仅代表执行一个动作的标识。），我们称这样的目标是“伪目标”。

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

command：规则的命令行。是规则所要执行的动作（任意的shell 命令或者是可在shell 下执行的程序）。它限定了make 执行这条规则时所需要的动作。

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

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

在 Makefile 中“规则”就是描述在什么情况下、如何重建规则的目标文件，通常规则中包括了目标的依赖关系（目标的依赖文件）和重建目标的命令。make 执行重建目标的命令，来创建或者重建规则的目标（此目标文件也可以是触发这个规则的上一个规则中的依赖文件）。规则包含了文件之间的依赖关系和更新此规则目标所需要的命令。

一个 Makefile 文件中通常还包含了除规则以外的很多东西（后续我们会一步一步的展开）。一个最简单的Makefile 可能只包含规则。规则在有些 Makefile 中可能看起来非常复杂，但是无论规则的书写是多么的复杂，它都符合规则的基本格式。

make 程序根据规则的依赖关系，决定是否执行规则所定义的命令的过程我们称之为执行规则。

简单的示例

一个简单的Makefile，来描述如何创建最终的可执行文件“edit”，此可执行文件依赖于8个C源文件和3个头文件。Makefile文件的内容如下：

```
#sample Makefile
  edit : main.o kbd.o command.o display.o \
         insert.o search.o files.o utils.o
      cc -o edit main.o kbd.o command.o display.o \
          insert.o search.o files.o utils.o
main.o : main.c defs.h
      cc -c main.c
kbd.o : kbd.c defs.h command.h
      cc -c kbd.c
command.o : command.c defs.h command.h
      cc -c command.c
display.o : display.c defs.h buffer.h
      cc -c display.c
insert.o : insert.c defs.h buffer.h
      cc -c insert.c
search.o : search.c defs.h buffer.h
      cc -c search.c
files.o : files.c defs.h buffer.h command.h
      cc -c files.c
```

```
utils.o : utils.c defs.h
    cc -c utils.c
clean :
    rm edit main.o kbd.o command.o display.o \
        insert.o search.o files.o utils.o
```

首先书写时，可以将一个较长行使用反斜线（\ ）来分解为多行。但需要注意：反斜线之后不能有空格（这也是大 家最容易犯的错误，错误比较隐蔽）。
在完成了这个Maekfile以后；需要创建可执行程序“edit”，所要做的就是在包含此Makefile的目录（当然也在代码所在的目录）下输入命令“make”。删除已经此目录下之前使用“make”生成的文件（包括那些中间过程的.o文件），也只需要输入命令“make clean”就可以了。

make如何工作

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

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

当在shell 提示符下输入“make”命令以后。make 读取当前目录下的 Makefile 文件，并将 Makefile 文件中的第一个目标作为其执行的“终极目标”，开始处理第一个规则（终极目标所在的规则）。在上例中，第一个规则就是目标“edit”所在的规则。规则描述了“edit”的依赖关系，并定义了链接.o 文件生成目标“edit”的命令； make在执行这个规则所定义的命令之前，首先处理目标“edit”的所有的依赖文件（例子中的那些.o 文件）的更新规则（以这些.o 文件为目标的规则）。对这些.o 文件为目标的规则处理有下列三种情况：

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

这些.o 文件所在的规则之所以会被执行，是因为这些.o 文件出现在“终极目标”的依赖列表中。在Makefile 中一个规则的目标如果不是“终极目标”所依赖的（或者“终极目标”的依赖文件所依赖的），那么这个规则将不会被执行，除非明确指定执行这个规则（可以通过 make 的命令行指定重建目标，那么这个目标所在的规则就会被执行，例如 “make clean”）。在编译或者重新编译生成一个.o 文件时，make 同样会去寻找它的依赖文件的重建规则（是这样一个规则：这个依赖文件在规则中作为目标出现），在这里就是.c 和.h 文件的重建规则。在上例的 Makefile 中没有哪个规则的目标是.c或者.h 文件，所以没有重建.c 和.h 文件的规则

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

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

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

指定变量

“objects”作为一个变量，它代表所有的.o文件的列表。在定义了此变量后，我们就可以在需要使用这些.o文件列表的地方使用“$(objects)”来表示��，而不需要罗列所有的.o文件列表。

make如何解析makefile文件

GUN make 的执行过程分为两个阶段。

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

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

总结

make 的执行过程如下：

1.依次读取变量“MAKEFILES”定义的 makefile 文件列表

2.读取工作目录下的 makefile文件（根据命名的查找顺序“GNUmakefile”，“makefile”，“Makefile”，首先找到那个就读取那个）

3.依次读取工作目录 makefile 文件中使用指示符“include”包含的文件

4.查找重建所有已读取的 makefile 文件的规则（如果存在一个目标是当前读取的 某一个makefile 文件，则执行此规则重建此 makefile 文件，完成以后从第一步开始重新执行）

5.初始化变量值并展开那些需要立即展开的变量和函数并根据预设条件确定执行分支

6.根据“终极目标”以及其他目标的依赖关系建立依赖关系链表

7.执行除“终极目标”以外的所有的目标的规则（规则中如果依赖文件中任一个 文件的时间戳比目标文件新，则使用规则所定义的命令重建目标文件）

8.执行“终极目标”所在的规则

说明：

执行一个规则的过程是这样的：

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