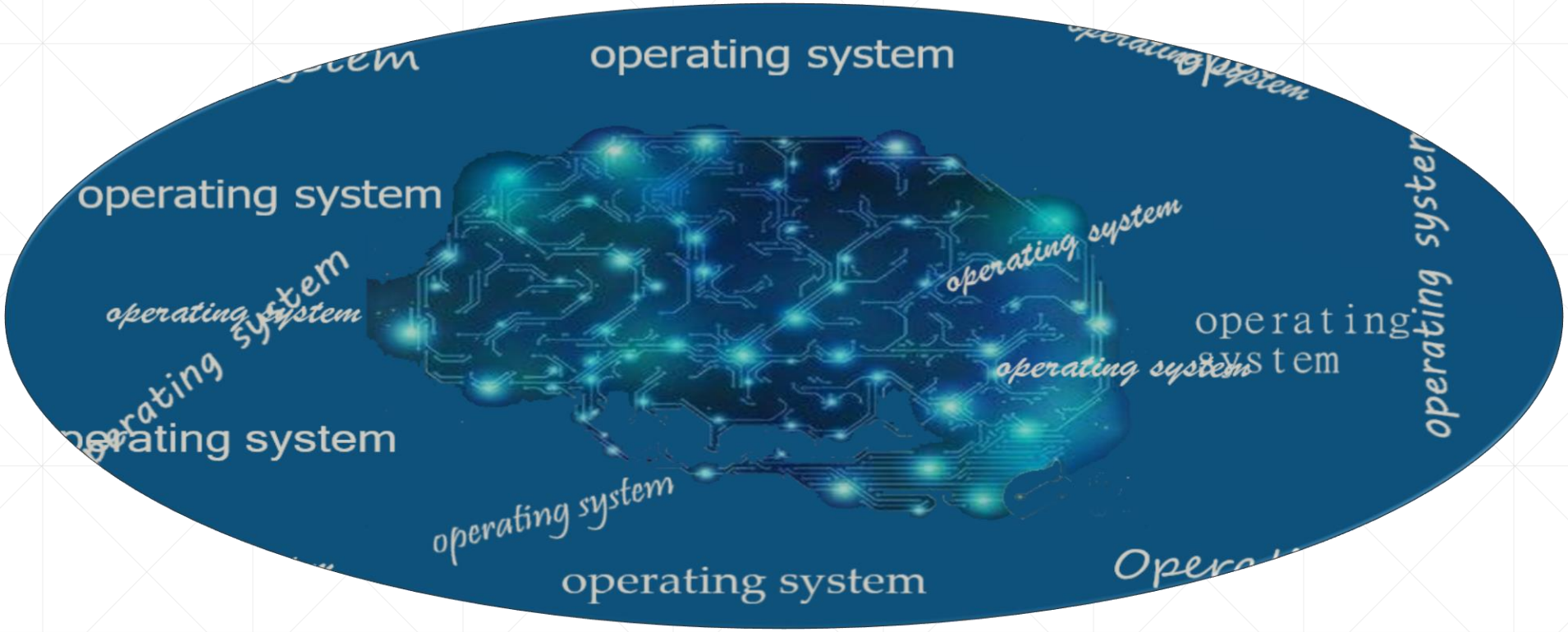


多模块编译链接工具make的使用

授课教师：

电子邮箱：

本章主要内容

- 1 多模块程序设计的概念
- 2 Make程序的基本功能和基本流程
- 3 Makefile规则
- 4 在makefile中执行命令
- 5 变量定义和使用

？问题

- 多模块程序的常见自动构造系统有哪些?
- Make是如何进行工作的?
- Make有哪些常用的命令?
- Makefile中有哪些规定的语法?
- 如何在Makefile中执行命令?

1.1 多模块程序设计

- 为了支持Linux平台上的程序编译和调试，通常采用GNU跨平台开发工具链，主要包括gcc，binutils，glibc和gdb。

- gcc是一个用于编程开发的自由编译器，它支持包括C、C++、Ada、ObjectC、Java及Go等语言。

- binutils提供了一系列用来创建、管理和维护二进制目标文件的工具程序，通常binutils与gcc是紧密相集成的，没有binutils的支持gcc不能正常工作。

1.1多模块程序设计

- glibc是GNU计划发布的libc库，也即C语言程序运行库。
- glibc将Linux系统所提供的系统调用以C语言函数的形式封装为应用程序开发接口（API）。
- 几乎其它任何的运行库都会倚赖于glibc。
- glibc除了封装Linux操作系统所提供的系统服务外，它本身也提供了许多其它一些必要功能服务的实现。

1.1多模块程序设计

- gdb是GNU计划发布的程序调试工具，gdb可以完成以下四个方面的功能：

（1）启动程序，按照自定义的要求运行程序

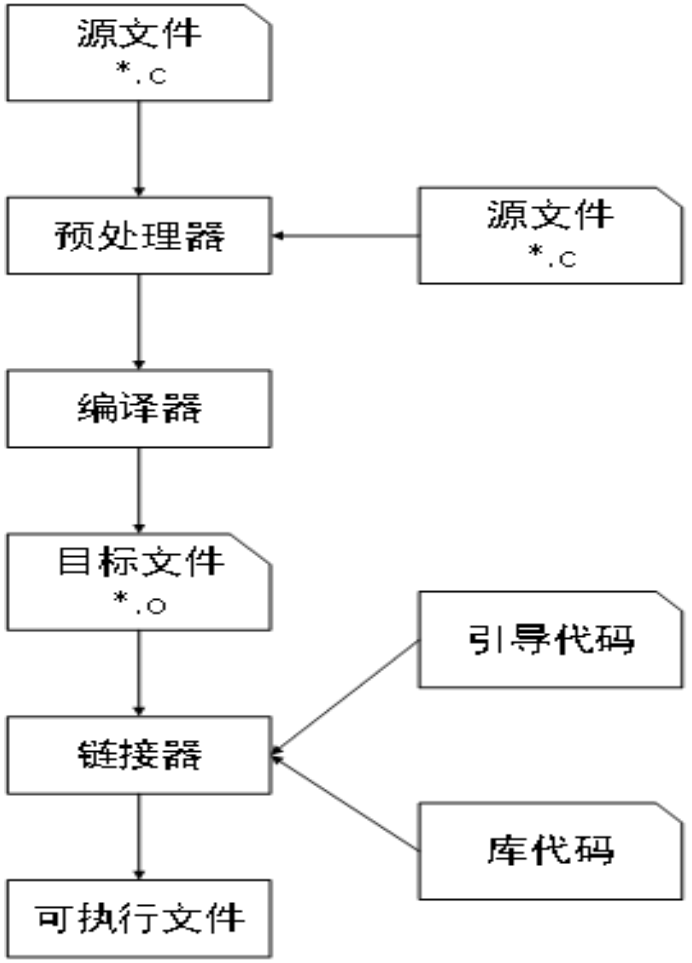
（2）可让被调试的程序在所指定的位置的断点处停住（断点可以是条件表达式）

（3）当程序被停住时，可以检查此时程序中所发生的情况

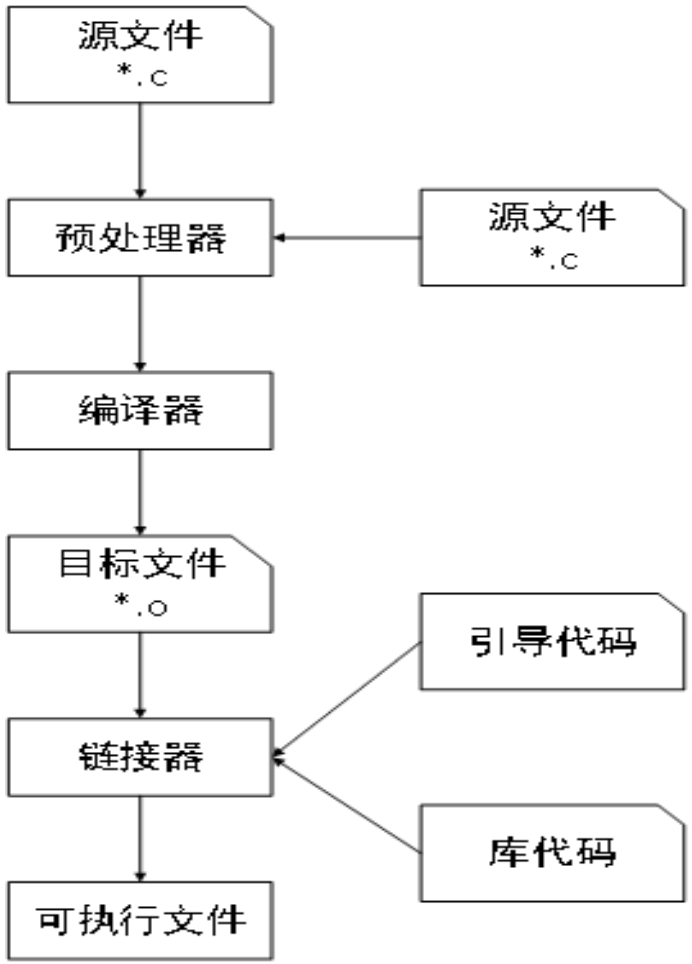
（4）动态的改变程序的执行环境。

1.1多模块程序设计

- 在Linux系统中，通过C语言编写的程序要转换为可执行文件需要经过预处理、编译、汇编以及链接的过程。



1.1多模块程序设计



- gcc首先调用cpp进行预处理，在预处理过程中，对源代码文件中的文件包含（例如通过include包含的头文件）、预编译语句（例如宏定义define等）进行分析。

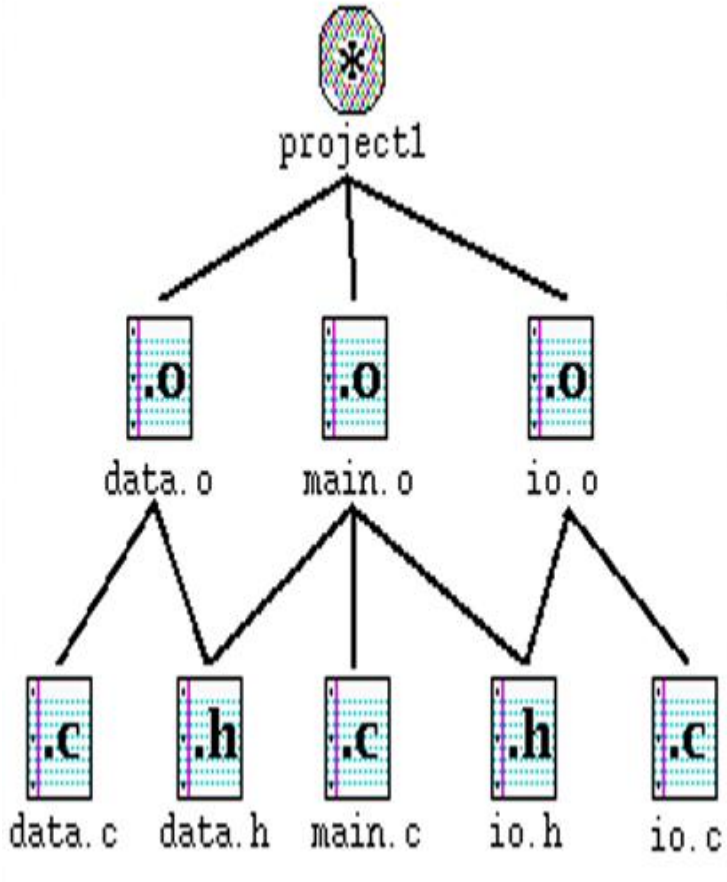
- 接着调用cc1进行编译，首先根据输入文件生成生成以.s为后缀的汇编语言源代码文件和汇编。再调用as生成以.o为后缀的目标文件。

- 当所有的目标文件都生成之后，gcc就调用ld来完成最后的关键性工作，这个阶段就是链接。在链接阶段，所有的目标文件被安排在可执行程序中的恰当的位置，同时，该程序所调用到的库函数也从各自所在的静态链接库中被链接到合适的地方。

1.1多模块程序设计

- 依赖关系图在构造系统中占有重要地位，它不仅列出了参与构造过程的文件，而且展示了这些文件之间的依赖关系。

- 诸如GNU Make等构造工具，使用依赖关系图来判断要对哪些文件进行编译，以及何时编译。



Sample Makefile

```
project1: data.o main.o io.o
    cc data.o main.o io.o -o project1
data.o: data.c data.h
    cc -c data.c
main.o: data.h io.h main.c
    cc -c main.c
io.o: io.h io.c
    cc -c io.c
```

(1) GNU make

- 文本格式的文件——Makefile可以使我们通过某种方法，以源代码的格式来表达依赖关系图，支持GNU make。
- 具体方式是：在文件中列出相关文件，描述这些文件之间的依赖关系，并说明将要使用哪些编译器命令。Makefile与源文件和目标文件保存在同一目录中。

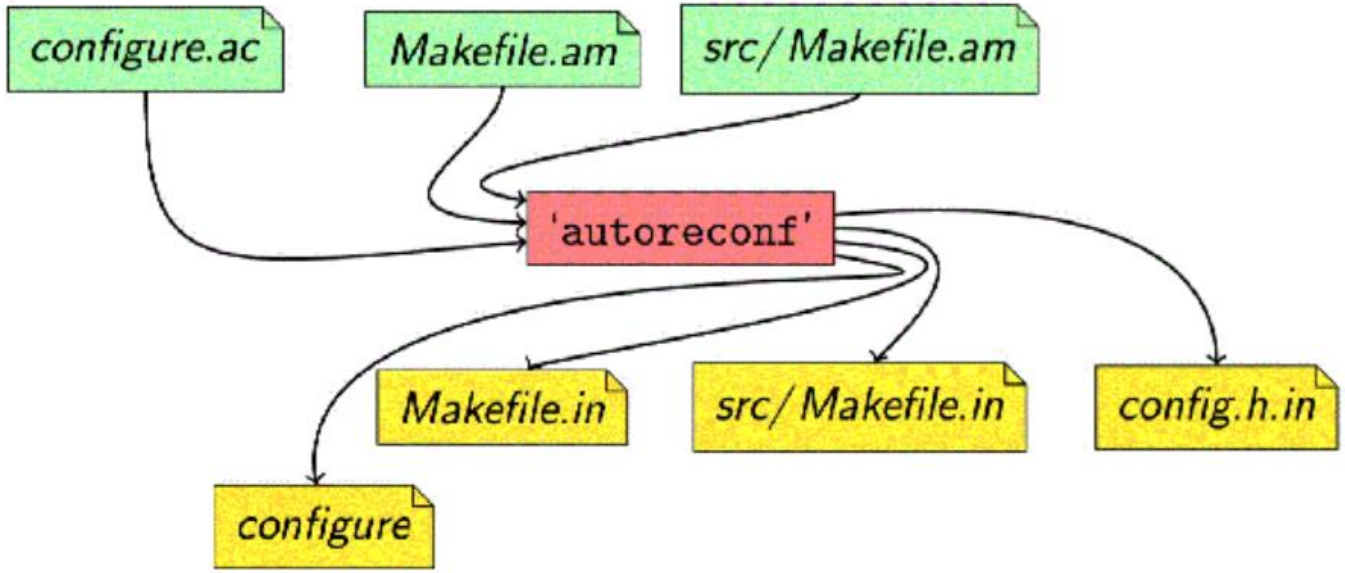
(1) GNU make

例： addressbook程序Makefile文件如下图所示

```
addressbook: main.o insert.o delete.o find.o
    gcc -g -o addressbook main.o insert.o delete.o find.o

main.o:  main.c  addressbook.h
    gcc -g -c main.c

insert.o: insert.c addressbook.h
    gcc -g -c insert.c

delete.o: delete.c addressbook.h
    gcc -g -c delete.c

find.o: find.c addressbook.h
    gcc -g -c find.c
```

(2) GNU build system

GNU构建系统，是利用脚本和make程序在特定的平台上构建软件的过程。
一般过程是configure，make，make install 三部曲。



- configure脚本是由软件开发者维护并发布的给用户使用的shell脚本。该脚本的作用是检测系统环境，最终目的是生成Make file和configure.h。
- make 通过读取Make file文件，开始构建软件。
- make install可以将软件安装到默认或者指定的系统路径

(2) GNU build system

开发者除了编写软件本身的代码外，还需要负责生成构建软件所需要的文件和工具。因此对于开发者而言，要么自己编写构建用的脚本，要么选择部分依赖工具。Auto tools就是这样的工具。

autotools使用流程

第一步：手工编写Makefile.am这个文件

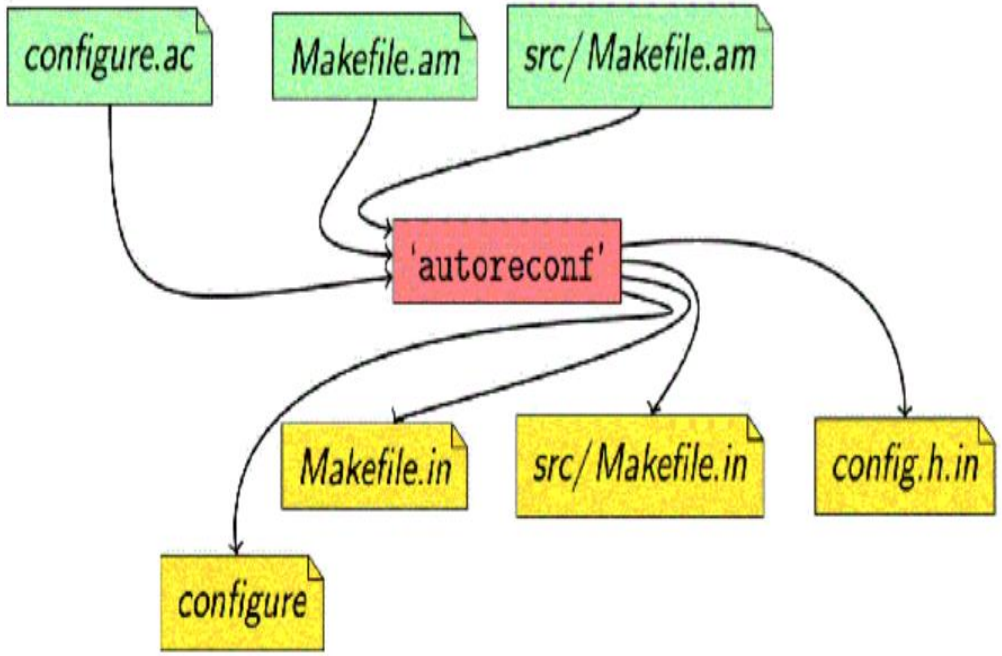
第二步：在源代码目录树的最高层运行autoscan。然后手动修改configure.scan文件，并改名为configure.ac/configure.in

第三步：运行aclocal，它会根据configure.ac的内容生成aclocal.m4文件

第四步：运行autoconf，它根据configure.ac和aclocal.m4的内容生成configure这个配置脚本文件

第五步：运行automake －add-missing，它根据Makefile.am的内容生成Makefile.in

第六步：运行configure，它会根据Makefile.in的内容生成Makefile这个文件



(3) CMake

CMake 是一个跨平台的开源构建工具，使用 CMake 能够方便地管理依赖多个库的目录层次结构并生成 makefile 和使用 GNU make 来编译和连接程序。

cmake的特点主要有：

1. 开放源代码
2. 跨平台，并可生成native编译配置文件
   a) 在Linux/Unix平台，生成makefile
   b) 在苹果平台，可以生成xcode
   c) 在Windows平台，可以生成MSVC的工程文件
3. 能够管理大型项目
4. 简化编译构建过程和编译过程。
   Cmake的工具链非常简单：cmake + make。
5. 高效虑
6. 可扩展，可以为cmake编写特定功能的模块，扩充cmake功能。

1.3 GNU make构建系统

addressbook程序Makefile文件

```
addressbook: main.o insert.o delete.o find.o
    gcc -g -o addressbook main.o insert.o delete.o find.o

main.o:  main.c  addressbook.h
    gcc -g -c main.c

insert.o: insert.c addressbook.h
    gcc -g -c insert.c

delete.o: delete.c addressbook.h
    gcc -g -c delete.c

find.o: find.c addressbook.h
    gcc -g -c find.c
```

1.3 GNU make构建系统

Makefile中每一段都引入了一个新的规则。

- **第1行声明一个名为addressbook的文件，它依赖于其他所有文件：insert.o、delete.o、find.o和main.o。**
- **第2行提供了一个Linux命令，用来根据所有这些目标文件生成addressbook文件。**

```
addressbook: main.o insert.o delete.o find.o
    gcc -g -o addressbook main.o insert.o delete.o find.o
```

Target : dependencies (or prerequisites)
<Tab> system commands (or recipe)

- Target: The file we want to create
- Dependencies: Check before creating the target file
- Commands: Consider as shell script.

1.3 GNU make构建系统

- 第4行指出main.o同时依赖于main.c和addressbook.h两个文件，而第5行则提供了用来编译main.o的Linux命令。

后面的Makefile内容所提供的规则与上面类似，分别指定了其他源文件和目标文件。

以上的Makefile对依赖关系图的直截了当的转译容易让人理解，但效率较低。

```
main.o:  main.c  addressbook.h
    gcc -g -c main.c

insert.o: insert.c addressbook.h
    gcc -g -c insert.c

delete.o: delete.c addressbook.h
    gcc -g -c delete.c

find.o: find.c addressbook.h
    gcc -g -c find.c
```

1.3 GNU make构建系统

- 当完成上述Makefile文件编写时，可以通过在Linux shell中执行make命令，来进行addressbook程序的构造

- $make
- gcc -g –c main.c
- gcc –g –c insert.c
- gcc –g -c delete.c
- gcc –g -c find.c
- gcc -g -o addressbook main.o insert.o delete.o find.o

```
addressbook: main.o insert.o delete.o find.o
    gcc -g -o addressbook main.o insert.o delete.o find.o

main.o:  main.c  addressbook.h
    gcc -g -c main.c

insert.o: insert.c addressbook.h
    gcc -g -c insert.c

delete.o: delete.c addressbook.h
    gcc -g -c delete.c

find.o: find.c addressbook.h
    gcc -g -c find.c
```

1.3 GNU make构建系统

- GNU Make的一个重要特点是增量式构造。
- GNU Make不会盲目执行命令，而是做一些事前分析，判断某些文件是否真的需要编译，或这些文件是否已存在。
- 在完成对程序的首次构造之后，如果没有任何修改就再次执行make命令，那么将会出现如下图所示的效果。

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

2 Make程序的基本功能和运行流程

- GNU Make工作时的执行步骤入下：

  （1）读入所有的Makefile文件。

  （2）读入通过include被包含的其它Makefile。

  （3）初始化文件中的变量。

  （4）推导隐晦规则，并分析所有规则。

  （5）为所有的目标文件创建依赖关系链。

  （6）根据依赖关系，决定哪些目标要重新生成。

  （7）执行生成命令。

- 其中1-5步为第一个阶段，6-7步为第二个阶段。在第一个阶段中，如果定义的变量被使用了，那么make会把其展开在使用的位置。如果变量出现在依赖关系的规则中，那么仅当这条依赖被决定要使用了，变量才会在其内部展开。

2 Make程序的基本功能和运行流程

在第一个阶段中，如果定义的变量被使用了，那么make会把其展开在使用的位置。

如果变量出现在依赖关系的规则中，那么仅当这条依赖被决定要使用了，变量才会在其内部展开。

```
1.  # I am a comment
2.  CC=g++
3.  CFLAGS=-c -Wall
4.
5.  all: hello
6.
7.  hello: main.o factorial.o hello.o
8.      $(CC) main.o factorial.o hello.o -o hello
9.
10. main.o: main.cpp
11.     $(CC) $(CFLAGS) main.cpp
12.
13. factorial.o: factorial.cpp
14.     $(CC) $(CFLAGS) factorial.cpp
15.
16. hello.o: hello.cpp
17.     $(CC) $(CFLAGS) hello.cpp
18.
19. clean:
20.     @echo "Clean..."
21.     -rm -rf *o hello
```

指定Makefile

- 当在Linuxshell中执行make命令时，如果没有附加任何的参数，那么make程序在当前目录下依次查找找名为“Makefile”或“makefile”或“GNUMakefile”的文件，一旦找到，就开始读取这个文件并执行。
- 也可以通过“-f ”或是“--file”参数或是“--Makefile”参数为make命令指定一个特殊名字的Makefile。如果Makefile的名字是“abc.mk”，那么，可以按照示例的方式来为make命令指定该文件：
- $make –f abc.mk
- 如果在执行make命令时多次使用了“-f ”参数，那么所指定的多个Makefile将会被连在一起传递给make命令。

控制make的函数

- 在Makefile中可以通过一些函数来控制make程序的运行，例如检测make的运行时信息，并且根据这些信息来决定是让make继续执行还是终止。

（1）error函数

- $(error<text...>)
- 上述语句将产生一个错误，并且终止make的执行，<text...>是错误提示信息

（2）warning函数

- $(warning<text...>)
- warning函数和error函数有类似之处，只是它并不会让make退出执行，而只是输出一段警告信息。

```
1  ERR = $(error found an error!)
2  .PHONY: err
3  err: ; $(ERR)
```

Make的命令行参数及退出码

make命令执行时可能出现“0”，“1”和“2”这三种不同情况的退出码。

- 0表示构建成功；
- 如果make运行时出现任何错误，则返回1；
- 如果在命令参数中使用了“-q”，并且make使得一些目标不需要更新，那么返回2。

Make的命令行参数及退出码

make命令支持很多的参数

- (1) “-b”，“-m”
  这两个参数的作用是忽略和其它版本make的兼容性。
- (2) “-B”，“--always-make”
  认为所有的目标都需要更新（重新编译）。
- (3) “-C<dir>”，“--directory=<dir>”

  指定读取Makefile文件的路径，如果有多个“-C”参数，那么表示后面参数指定的路径是相对路径，该路径基于前面参数所指定的路径，并且以最后的目录作为指定的读取Makefile文件的目录。例如，“make－C~test/test－Cprog”等价于“make－C~test/test/prog”。

Make的命令行参数及退出码

（4） “—debug[=<options>]”

输出make的调试信息，它有几种不同的级别可供选择，如果没有参数，那就是输出最简单的调试信息。<options>的可能取值包括：

- a，输出所有的调试信息；
- b，只输出简单的调试信息，即输出不需要重编译的目标；
- v，在b选项的级别之上，输出的信息包括哪个Makefile被解析，不需要被重编译的依赖文件（或是依赖目标）等；
- i，输出所有的隐含规则；
- j，输出执行规则中命令的详细信息，如命令的PID、返回码等；
- m，输出make读取Makefile，更新Makefile，执行Makefile的信息。

Make的命令行参数及退出码

- (5) “-e”，“--environment-overrides”
  指明环境变量的值覆盖Makefile中定义的变量的值。
- (6) “-f=<file>”，“--file=<file>”，“-Makefile=<file>”
  指定需要执行的Makefile文件。
- (7) “-h”，“—help”
  显示帮助信息。

- 除此之外，make还有很多的命令行参数，请同学们查阅课本或帮助文档来进一步了解。

3 Makefile规则

- 一般来说，定义在Makefile中的目标可能会有很多，但是第一条规则中的目标将被确立为最终的目标。如果第一条规则中的目标有很多个，那么第一条规则的第一个目标会成为最终的目标。

```
objects = main.o kbd.o command.o display.o \
          insert.o search.o files.o utils.o
```

```
edit : $(objects)
        cc -o edit $(objects)
:
:
clean :
        rm edit $(objects)
```

3 Makefile规则

- **insert.o : insert.c addressbook.h #insert模块**
  gcc -c -g insert.c

- insert.o是目标，insert.c和addressbook.h是目标所依赖的源文件，包含一个命令“gcc -c -g insert.c”，这个规则表达了两个方面的含义:

（1）文件的依赖关系。insert.o依赖于insert.c和addressbook.h的文件，如果insert.o和addressbook.h的文件日期要比insert.o的文件日期要新，或是insert.o文件不存在，那么依赖关系发生。

（2）如果需要生成（或更新）insert.o文件，那么后面的gcc命令说明了该如何生成insert.o这个文件。

3 Makefile规则

```
1.  main: main.o foo1.o
2.      gcc main.o foo1.o -o main
3.  main.o: main.c
4.      gcc main.c -c
5.  foo1.o: foo1.c
6.      gcc foo1.c -c
7.  clean:
8.      rm -rf main.o foo1.o
```

make
gcc main.c –c
gcc foo1.c –c
gcc main.o fool.o –o main

make
make: ‘main’ is up to date.

3 Makefile规则

```
1.  main: main.o fool.o
2.      gcc main.o fool.o -o main
3.  main.o: main.c
4.      gcc main.c -c
5.  fool.o: fool.c
6.      gcc fool.c -c
7.  clean:
8.      rm -rf main.o fool.o
```

make
gcc main.c –c
gcc foo1.c –c
gcc main.o fool.o –o main

touch fool.c
make
gcc foo1.c –c
gcc main.o fool.o –o main

4 在makefile中执行命令

- 在Makefile中每条规则中的命令和Linux操作系统的Shell命令行是一致的。make会按顺序一条一条的执行命令，每条命令的开头必须以Tab字符开头，除非命令是紧跟在依赖规则后面的分号后的。在命令行之间中的空格或是空行会被忽略，但是如果该空格或空行是以Tab字符开头的，那么make会认为其是一个空命令。

- 用户在Linux下可能会使用不同的Shell，但是除非特别指定，否则make的命令默认都是Linux的标准Shell解释执行的。

命令执行顺序

- 当依赖目标的产生时间比目标的产生时间更靠后时，也就是当规则的目标需要被更新时，make会一条一条的执行其后的命令。需要注意的是，如果需要让上一条命令的结果应用在下一条命令时，应该使用分号分隔这两条命令。例如，如果第一条命令是cd命令，并且希望第二条命令在cd执行之后的基础上执行，那么就不能把这两条命令写在两行上，而应该把这两条命令写在一行上，用分号分隔。

- 比如写成如下的指令格式是正确的：

```
exec:
    cd/home/test；pwd
```

命令出错

- 每当命令运行完后，make会检测每个命令的返回码，如果命令返回成功，那么make会执行下一条命令，当规则中所有的命令成功返回后，这个规则就算是成功完成了。如果一个规则中的某个命令出错了（命令退出码非零），那么make就会终止执行当前规则，这将有可能终止所有规则的执行。
- 有些时候，命令的出错并不表示就是错误的。例如执行mkdir命令，如果目录不存在，那么mkdir就成功执行；如果目录存在，那么就出错了。在这种情况下，之所以使用mkdir就是希望保证一定要有这样的一个目录，因此并不希望mkdir出错而终止规则的运行。

命令出错

- 为了做忽略命令的出错，可以在Makefile的命令行前加一个减号“-”（在Tab字符之后），标记为不管命令出不出错都认为是成功的
- 给make加上“-i”或是“--ignore-errors”参数，那么，Makefile中所有命令都会忽略错误。
- 如果一个规则是以“.IGNORE”作为目标的，那么这个规则中的所有命令将会忽略错误。
- make的参数的是“-k”或是“--keep-going”，这个参数的意思是，如果某规则中的命令出错了，那么就终目该规则的执行，但继续执行其它规则。

嵌套执行make

- 在一些大的工程中往往会把不同模块或是不同功能的源文件放在不同的目录中，这种情况下可以在每个目录中都书写一个该目录的Makefile。这样就不需要把所有模块的依赖关系、执行命令等全部写在一个Makefile（总控Makefile）中，利于让总控Makefile变得更加地简洁，简化维护难度。

- 例如，有一个子目录叫subdir，这个目录下有个Makefile文件，来指明了这个目录下文件的编译规则。那么总控Makefile可以按照如下方式编写：

```
subsystem:
        cd subdir && $(MAKE)
```

嵌套执行make

- 也可以按照如下的方式编写。

```
subsystem:
    $(MAKE)  -C  subdir
```

- 这两个例子的意思都是先进入“subdir”目录，然后执行make命令。因为make命令可能需要一些参数，所以将其定义为$(MAKE)宏变量，这样比较利于维护。

嵌套执行make

- 总控Makefile的变量可以传递到下级的Makefile中，但是不会覆盖下层的Makefile中所定义的变量，除非指定了“-e”参数。如果需要传递变量到下级Makefile中，那么可以使用这样的声明：

  export <variable...>

- 如果不想让某些变量传递到下级Makefile中，那么可以这样声明：

  unexport <variable...>

  export variable=value

嵌套执行make

- 如果要传递所有的变量，那么只要一个export就行了。后面什么也不用写，表示传递所有的变量。

- 需要注意的是，有两个变量，一个是SHELL，一个是MAKEFLAGS，这两个变量不管是否export，其总是要传递到下层Makefile中，特别是MAKEFILES变量，其中包含了make的参数信息，如果执行“总控Makefile”时有make参数或是在上层Makefile中定义了这个变量，那么MAKEFILES变量将会是这些参数，并会传递到下层Makefile中，这是一个系统级的环境变量。

5 变量定义和使用

- 在Makefile中的定义的变量，就像是C/C++语言中的宏一样，代表了一个文本字串，在Makefile中执行的时候会将变量自动展开在所使用的地方。与C/C++所不同的是，可以在Makefile中改变其值。在Makefile中，变量可以使用在“目标”，“依赖目标”，“命令”或是Makefile的其他部分中。

- 变量的命名字可以包含字符、数字，下划线（可以是数字开头），但不应该含有“:”、“#”、“=”或是空字符（空格、回车等）。

- 变量是大小写敏感的，“foo”、“Foo”和“FOO”是三个不同的变量名。传统的Makefile的变量名是全大写的命名方式，但推荐使用大小写搭配的变量名，如：MakeFlags。这样可以避免和系统的变量冲突，而发生意外的事情。

5 变量定义和使用

- 有一些变量是很奇怪字串，如“$<”、“$@”等，这些是自动化变量。
- 变量使用的注意事项如下：

（1）变量在声明时需要给予初值，而在使用时，需要给在变量名前加上“$”符号，但最好用小括号“（）”或是大括号“{}”把变量给包括起来。如果需要使用“$”字符本身，那么需要用“$$”来表示。如：

```
objects=program.o  foo.o  utils.o
program:$(objects)
        cc-oprogram$(objects)
$(objects):defs.h
```

5 变量定义和使用

- 变量会在使用它的地方精确地展开，就像C/C++中的宏一样，例如：
- foo=c
- prog.o:prog.$(foo)

  (foo)(foo) -$(foo) prog.$(foo)
- 展开后得到：
- prog.o : prog.c

  cc -c prog.c
- 另外，给变量加上括号完全是为了更加安全地使用这个变量。

5 变量中的变量

- 在定义变量的值时，可以使用其他变量来构造变量的值，在Makefile中有两种方式来在用变量定义变量的值。
- 先看第一种方式，也就是简单的使用“=”号，在“=”左侧是变量，右侧是变量的值，右侧变量的值可以定义在文件的任何一处，也就是说，右侧中的变量不一定非要是已定义好的值，也可以使用后面定义的值。
- foo=$(bar)
- bar=$(ugh)
- ugh=Huh?
- all:

```
    echo   $(foo)
```

方便，但可能发生不可预知的错误

5 变量中的变量

为了避免上面的这种方法，我们可以使用make中的另一种用变量来定义变量的方法。这种方法使用的是“:=”操作符，如示例所示：

```
x:=foo
y:=$(x)bar
```

采用这种方法时前面的变量不能使用后面的变量，只能使用前面已定义好了的变量。

小测试

- 1.在Makefile中的命令必须要以什么键开始?