实验二 Makefile 实验

【实验目的】

1、了解 Makefile 的基本概念和基本结构

2、初步掌握编写简单 Makefile 的方法

3、了解递归 Make 的编译过程

4、初步掌握利用 GNU Make 编译应用程序的方法

【实验原理】

在 Linux 或 Unix 环境下，对于只含有几个源代码文件的小程序（如 hello.c）的编译，可以手工键入 gcc 命令对源代码文件逐个进行编译； 然而在大型的项目开发中， 可能涉及几十到几百个源文件， 采用手工键入的方式进行编译， 则非常不方便， 而且一旦修改了源代码，尤其头文件发生了的修改，采用手工方式进行编译和维护的工作量相当大，而且容易出错。

所以在 Linux 或 Unix 环境下，人们通常利用 GNU make 工具来自动完成应用程序的维护和编译工作。 实际上， GNU make 工具通过一个称为 Makefile 的文件来完成对应用程序的自动维护和编译工作。 Makefile 是按照某种脚本语法编写的文本文件，而 GNU make 能够对 Makefile 中指令进行解释并执行编译操作。 Makefile 文件定义了一系列的规则来指定哪些文件需要先编译， 哪些文件需要后编译， 哪些文件需要重新编译， 甚至于进行更复杂的功能操作。 GNU make 工作时的执行步骤如下：

1、读入所有的 Makefile 。

2、读入被 include 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。

1-5 步为第一个阶段， 6-7 为第二个阶段。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。但 make 并不会完全马上展开， make 使用的是拖延战术，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。下面对 makefile 的相关问题进行简单介绍：

1、 Makefile 的基本结构

Makefile 的一般结构：

target…… ： **dependency**……

command……

结构中各部分的含义：

（1）、target（目标）：一个目标文件，可以是 Object 文件，也可以是执行文件。还可以是一个标签（ Label）。

（2）、dependency（依赖）：要生成目标文件（ target）所依赖哪些文件

（3）、command（命令）：创建项目时需要运行的 shell 命令（ 注：命令（ **command**）部分的每行的缩进必须要使用 **Tab** 而不能使用多个空格 ）。

Makefile 实际上是一个文件的依赖关系， 也就是说， target 这一个或多个的目标文件依赖于 dependency 中的文件， 其生成规则定义在命令 command 中。如果依赖文件 （dependency）中有一个以上的文件比目标（ target）文件要新的话， shell 命令（ command）所定义的命令

就会被执行。这就是 Makefile 的规则。也就是 Makefile 中最核心的内容。

例如，假设有一个 C 源文件 test.c，该源文件包含有自定义的头文件 test.h，则目标文件 test.o 明确依赖于两个源文件： test.c 和 test.h。如果只希望利用 gcc 命令来生成 test.o 目标文件，这时，就可以利用如下的 makefile 来定义 test.o 的创建规则：

```
#This makefile just is a example.
test.o: test.c test.h
        gcc -c test.c
```

从上面的例子注意到，第一个字符为 #的行表示注释行。第一个非注释行指定 test.o 为目标，并且依赖于 test.c 和 test.h 文件。随后的行指定了如何从目标所依赖的文件建立目标。

当 test.c 或 test.h 文件在编译之后又被修改，则 make 工具可自动重新编译 test.o，如果在前后两次编译之间， test.c 和 test.h 均没有被修改，而且 test.o 还存在的话，就没有必要重新编译。这种依赖关系在多源文件的程序编译中尤其重要。 通过这种依赖关系的定义， make 工具可避免许多不必要的编译工作。

一个 makefile 文件中可定义多个目标，利用 make target 命令可指定要编译的目标，如果不指定目标，则使用第一个目标。通常， makefile 中定义有 clean 目标，可用来清除编译过程中的中间文件

```
# This makefile just is a example.
test.o: test.c test.h
        gcc -c  test.c
clean:
        rm -f *.o
```

运行 make clean 时，执行 rm -f *.o 命令，删除编译过程中生成的所有中间文件。

2、 Makefile 的基本内容

Makefile 一般包括包含： 显式规则、 隐晦规则、 变量定义、 文件指示和注释等五个内容。

（1）、显式规则：显式规则说明如何生成一个或多个的目标文件。这是由 Makefile 的书写者明显指出，要生成的文件，文件的依赖文件，生成的命令。

（2）、变量定义。在 Makefile 中可以定义一系列的变量，变量一般都是字符串，当 Makefile 被执行时，变量的值会被扩展到相应的引用位置上。

（3）、隐含规则 ：由于 GNU make 具有自动推导功能，所以隐晦规则可以比较粗糙地简略地书写 Makefile ，然后由 GNU make 的自动推导功能完成隐晦规则的内容。

（4）、文件指示。其包括了三个部分，一个是在一个 Makefile 中引用另一个 Makefile ，就像 C 语言中的 include 一样；另一个是指根据某些情况指定 Makefile 中的有效部分，就像 C 语言中的预编译 #if 一样；还有就是定义一个多行的命令。

（5）、注释。 Makefile 中只有行注释，和 UNIX 的 Shell 脚本一样，其注释是用 #字符，如果你要在你的 Makefile 中使用 #字符，可以用反斜框进行转义，如： \#。

2.1 Makefile 中的变量

（1）、Makefile 中定义的变量，与 C/C++ 语言中的宏一样，代表一个文本字串，在 Makefile 被执行时候变量会自动地展开在所使用的地方。 Makefile 中的变量可以使用在 目标，依赖目标，命令或 Makefile 的其它部分中。

（2）、 Makefile 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） 。

（3）、 Makefile 中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

（4）、变量在声明时需要给予初值，而在使用时，需要在变量名前加上 $符号

```
# makefile test for hello program
#written by Emdoor
CC=gcc
CFLAGS=
OBJS=hello.o
all: hello
hello: $(OBJS)
	$(CC) $(CFLAGS) $(OBJS) -o hello
hello.o: hello.c
	$(CC) $(CFLAGS) -c hello.c -o $(OBJS)
clean:
	rm -rf hello *.o
```

上面自定义变量 OBJS 表示 hello.o，当 makefile 被执行时， 变量会在使用它的地方精确地展开，就像 C/C++ 中的宏一样。上述 makfile 变量展开后的形式为：

```
# makefile test for hello program
#written by Emdoor
CC=gcc
CFLAGS=
OBJS=hello.o
all: hello
hello: hello.o
	gcc hello.o -o hello
hello.o: hello.c
	gcc -c hello.c -o hello.o
clean:
	rm -rf hello *.o
```

GNU make 的主要预定义变量

GNU make 有许多预定义的变量，这些变量具有特殊的含义，可在规则中使用。以下给出了一些主要的预定义变量， 除这些变量外， GNU make 还将所有的环境变量作为自己的预定义变量。

**$@** ——表示规则中的目标文件集。在模式规则中，如果有多个目标，那么， "$@" 就是匹配于目标中模式定义的集合。

$% ——仅当目标是函数库文件中，表示规则中的目标成员名。例如，如果一个目标是 "foo.a(bar.o)" ，那么， "$%" 就是 "bar.o"， "$@" 就是 "foo.a" 。如果目标不是函数库文件（ Unix 下是 [.a]， Windows 下是 [.lib] ），那么，其值为空。

**$<** ——依赖目标中的第一个目标名字。 如果依赖目标是以模式 （即 "%" ）定义的， 那么 "$<" 将是符合模式的一系列的文件集。注意，其是一个一个取出来的。

$? ——所有比目标新的依赖目标的集合。以空格分隔。

$^ ——所有的依赖目标的集合。以空格分隔。如果在依赖目标中有多个重复的，那个这个变量会去除重复的依赖目标，只保留一份。

$+ ——这个变量很像 "$^"，也是所有依赖目标的集合。只是它不去除重复的依赖目标。

命令的变量。

AR 函数库打包程序。默认命令是 —ar。

AS 汇编语言编译程序。默认命令是 —as。

CC C 语言编译程序。默认命令是 —cc。

CXX C++语言编译程序。默认命令是 —g++。

CO 从 RCS 文件中扩展文件程序。默认命令是 —co。

CPP C 程序的预处理器（输出是标准输出设备）。默认命令是 —$(CC) -E。

FC Fortran 和 Ratfor 的编译器和预处理程序。默认命令是 —f77。

GET 从 SCCS 文件中扩展文件的程序。默认命令是 —get。

LEX Lex 方法分析器程序（针对于 C 或 Ratfor）。默认命令是 —lex。

PC Pascal 语言编译程序。默认命令是 —pc。

YACC Yacc 文法分析器（针对于 C 程序）。默认命令是 —yacc。

YACCR Yacc 文法分析器（针对于 Ratfor 程序）。默认命令是 —yacc -r。

MAKEINFO 转换 Texinfo 源文件（.texi）到 Info 文件程序。默认命令是 —makeinfo。

TEX 从 TeX 源文件创建 TeX DVI 文件的程序。默认命令是 —tex。

TEXI2DVI 从 Texinfo 源文件创建军 TeX DVI 文件的程序。默认命令是 —texi2dvi。

WEAVE 转换 Web 到 TeX 的程序。默认命令是 —weave。

CWEAVE 转换 C Web 到 TeX 的程序。默认命令是 —cweave。

TANGLE 转换 Web 到 Pascal 语言的程序。默认命令是 —tangle。

CTANGLE 转换 C Web 到 C。默认命令是 —ctangle。

RM 删除文件命令。默认命令是 —rm -f

命令参数变量：

下面的这些变量都是相关上面的命令的参数。如果没有指明其默认值，那么其默认值都是空。

ARFLAGS 函数库打包程序 AR 命令的参数。默认值是 —rv。

ASFLAGS 汇编语言编译器参数。（当明显地调用 —.s 或—.S 文件时）。

CFLAGS C 语言编译器参数。

CXXFLAGS C++语言编译器参数。

COFLAGS RCS 命令参数。

CPPFLAGS C 预处理器参数。（C 和 Fortran 编译器也会用到）。

FFLAGS Fortran 语言编译器参数。

GFLAGS SCCS —get 程序参数。

LDFLAGS 链接器参数。（如：—ld）

LFLAGS Lex 文法分析器参数。

PFLAGS Pascal 语言编译器参数。

RFLAGS Ratfor 程序的 Fortran 编译器参数。

YFLAGS Yacc 文法分析器参数。

2.2 隐含规则

GNU make 包含有一些内置的或隐含的规则，这些规则定义了如何从不同的依赖文件建立特定类型的目标。

GNU make 支持两种类型的隐含规则：

（1）、后缀规则（ Suffix Rule ）。后缀规则是定义隐含规则的老风格方法。后缀规则定义了将一个具有某个后缀的文件（例如, .c 文件）转换为具有另外一种后缀的文件（例如, .o 文件）的方法。每个后缀规则以两个成对出现的后缀名定义，例如，将 .c 文件转换为 .o 文件的后缀规则可定义为：

```
.c.o:
    $(CC) $(CCFLAGS) $(CPPFLAGS) -c -o $@ $<
```

（2）、 模式规则（ pattern rules）。这种规则更加通用，因为可以利用模式规则定义更加复杂的依赖性规则。 模式规则看起来非常类似于正则规则, 但在目标名称的前面多了一个 % 号, 同时可用来定义目标和依赖文件之间的关系，例如下面的模式规则定义了如何将任意一个 X.c 文件转换为 X.o 文件：

```
%.c:%.o
    $(CC) $(CCFLAGS) $(CPPFLAGS) -c -o $@ $<
```

2.3 文件引用

在 Makefile 使用 include 关键字可以把别的 Makefile 包含进来，这很像 C 语言的#include ，被包含的文件会原模原样的放在当前文件的包含位置。

例如：有这样几个 Makefile ：a.mk、b.mk、c.mk，还有一个文件叫 foo.make，以及一个变量 $(bar)，其包含了 e.mk 和 f.mk，那么，下面的语句：

```
include foo.make *.mk $(bar)
```

等价于：

```
include foo.make a.mk b.mk c.mk e.mk f.mk
```

make 命令开始时，会把找寻 include 所指出的其它 Makefile ，并把其内容安置在当前的位置。如果文件都没有指定绝对路径或是相对路径的话, make 首先会在当前目录下寻找，如果当前目录下没有找到，那么, make 还会在下面的几个目录下找：

（1）如果 make 执行时，有 “- I ”或 “--include- dir ” 参数，那么 make 就会在这个参数所指定的目录下去寻找。

（2）、如果目录 <prefix>/include （一般是： /usr/local/bin 或/usr/include ）存在的话， make 也会去找。

如果有文件没有找到的话, make 会生成一条警告信息，但不会马上出现致命错误。它会继续载入其它的文件, 一旦完成 makefile 的读取， make 会再重试这些没有找到, 或是不能读取的文件，如果还是不行, make 才会出现一条致命信息。

2.4 Makefile 中的函数

在 Makefile 中可以使用函数来处理变量, 从而让命令或规则更为的灵活和具有智能, 函数调用，很像变量的使用，也是以 “$ ”来标识的，函数调用后，函数的返回值可以当做变量来使用。

例如：'wildcard' 的函 数，可以展开成一列所有符合由其参数描述的文件名。文件间以空格间隔。语法如下：

```
$(wildcard PATTERN...)
```

用'wildcard' 函数找出目录中所有的 ".c" 文件： SOURCES = $(wildcard *.c) 。实际上， GNU make 还是许多如 字符串处理函数、文件名操作函数等 其他函数

3、运行 **make**

3.1 Make 的执行

一般来说，最简单的就是直接在命令行下输入 make 命令， GNU make 找寻默认的 Makefile 的规则是在当前目录下依次找三个文件———GNUmakefile ‖、 —makefile ‖和 —Makefile ‖。其按顺序找这三个文件， 一旦找到， 就开始读取这个文件并执行， 也可以给 make 命令指定一个特殊名字的 Makefile 。要达到这个功能， 要求使用 make 的 —f ‖或是 —-file ‖参数,

例如： make －f Hello.makefile

3.2 嵌套执行 make

在一些大的工程中，不同模块或是不同功能的源文件放在不同的目录中，可以在每个目录中都书写一个该目录的 Makefile，这有利于 Makefile 变得更加地简洁，而不至于把所有的东西全部写在一个 Makefile 中，这个技术对于进行模块编译和分段编译有着非常大的好处。

例如，有一个子目录叫 subdir，这个目录下有个 Makefile 文件指明了这个目录下文件的编译规则。那么总控的 Makefile 可以书写：

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

如果要传递变量到下级 Makefile 中，那么可以使用 export <variable ...> 来声明。

3.3 GNU make 命令选项

GNU make 命令还有一些其他选项，下面给出了这些选项。

命令行选项	含义
-C DIR	在读取 makefile 之前改变到指定的目录 DIR。
-f FILE	以指定的 FILE 文件作为 makefile。
-h	显示所有的 make 选项。
-i	忽略所有的命令执行错误。
-I DIR	当包含其他 makefile 文件时，可利用该选项指定搜索目录。
-n	只打印要执行的命令，但不执行这些命令。
-p	显示 make 变量数据库和隐含规则。
-s	在执行命令时不显示命令。
-w	在处理 makefile 之前和之后，显示工作目录。
-W FILE	假定文件 FILE 已经被修改。

【实验仪器】

1、装有 Linux 操作系统的 PC 机一台；

2、 XSBase270 或 XSBase255 ARM 实验开发平台一套

【实验内容】

一、使用命令行的方式手动编译程序方法

1、利用文本编辑器创建 hello.c 文件

```
//hello.c
//written by Emdoor
#include <stdio.h>
int main()
{
        printf("Welcome Emdoor!\n");
        return 1;
}
```

2、手动编译 hello 应用程序

在 hello.c 的目录的终端下输入：

```
[root@local]$ gcc –c hello.c
[root@local]$gcc hello.o –o hello
```

通过 ls 命令查看当前目录下是否生成源代码　　hello.c 的 object 文件 hello.o 和可执行文件　hello，运行可执行文件　hello。查看一下运行结果。

```
[root@local]$./hello
```

3、修改　hello.c 文件，重新手动编译应用程序。

4、删除　hello.o 和 hello 文件

```
[root@local]$rm　-f hello.o
[root@local]$rm　-f hello
```

二、利用　GNU make　自动编译应用程序方法

1、 利用文本编辑器创建一个　　makefile 文件，并将其保存到与　　hello.c 相同的目录下。

```
# makefile test for hello program
#written by Emdoor
CC=gcc
CFLAGS=
all: hello
hello: hello.o
	$(CC) $(CFLAGS) hello.o　-o hello
hello.o: hello.c
	$(CC) $(CFLAGS)　-c hello.c　-o hello.o
clean:
	rm　-rf hello *.o
```

2、先后执行如下命令

```
[root@local]$make
[root@local]$ls
[root@local]$./hello
```

查看并记录所生成的文件和运行的结果。

3、执行　make clean 命令：

```
[root@local]$make clean
```

4、修改　hello.c 文件，重复第　2、3 步操作，查看并记录所生成的文件和运行结果，并与手动编译进行比较，写出你的结论。

5、重新编辑　makefile 文件（斜黑体表示修改部分）

```
# makefile test for hello program
#written by Emdoor
CC=gcc
CFLAGS=
OBJS=hello.o
all: hello
hello:  $(OBJS)
	$(CC) $(CFLAGS)　$^ -o $@
hello.o: hello.c
	$(CC) $(CFLAGS)　-c $< -o $@
clean:
	rm　-rf hello *.o
```

6、重复第 2，3 步操作，查看并记录所生成的文件和运行的结果。比较这两种操作，写出你的结论。同时指出 $^ 、$@、$<在上述 Makefile 中的含义。

三、多个 .c 文件的编译

1、创建文件 hello1.c、hello2.c、hello.h 和 makefile

```
//hello1.c
//written by Emdoor
#include <stdio.h>
int main()
{
        printf("Welcome Emdoor!\n");
        test2();
        return 1;
}
```

```
//hello2.c
//written by Emdoor
 # include "hello2.h"
#include <stdio.h>
void test2(void)
{
        printf("Welcome Emdoor!     hello2\n");
}
```

```
//hello2.h
//written by Emdoor
void test2(void);
```

```
# makefile test for multi files program
#written by Emdoor
CC=gcc
CFLAGS=
OBJS=hello1.o hello2.o
all: hello
hello: $(OBJS)
        $(CC) $(CFLAGS) $^ -o $@
hello1.o: hello1.c
        $(CC) $(CFLAGS)    -c $< -o $@
hello2.o: hello2.c
        $(CC) $(CFLAGS)    -c $< -o $@
clean:
        rm  -rf hello *.o
```

2、先后执行如下命令

[root@local]$make

[root@local]$ls

[root@local]$./hello

查看并记录所生成的文件和运行的结果　　，写出你的结论。

3、修改　makefile 文件（斜黑体表示修改部分）

```
# makefile test for multi files program
#written by Emdoor
CC=gcc
CFLAGS=
CFILES=$(wildcard *.c)
OBJS=$(CFILES:%c=%.o)
all: hello
hello: $(OBJS)
	$(CC) $(CFLAGS)    -o hello $(OBJS)
.c.o:
	$(CC)   -c $<
clean:
	rm  -rf hello *.o
```

4、重复第　2　步操作，查看并记录所生成的文件和运行的结果　　，写出你的结论。并指出 wildcard 、.c.o 的含义和变量　CFILES 代表的内容。

【思考题】

1、 根据提供的　Linux 操作系统源码中得　Makefile 结构，分析在工程中多级目录中存在着多个 makefile 时，编译的顺序如何？

2、 根据 Makefile 中变量定义规则，　如果实验中的　hello.c 文件编译到目标平台中运行，　应该怎样修改　Makefile 中变量参数？