编译的四个阶段

1.预处理：

将文件中带#的语句进行解析，把具体展开的内容展开在当前文件中，宏定义的替换、头文件的展开，不会进行语法的错误检查

gcc -E 1.c -o 1.i

2.编译：

进行语法错误的检查，编译生成汇编代码

gcc -S 1.i -o 1.s

3.汇编

编译汇编文件生成二进制文件

gcc -c 1.s -o 1.o

4.链接

将库函数对应的函数库或者其他.o文件一起链接进来生成可执行二进制文件

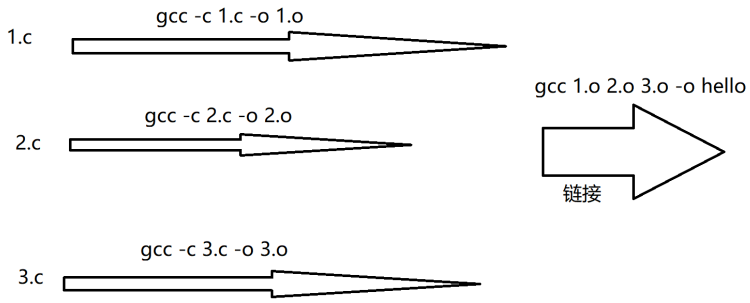
gcc 1.o -o hello

注意：一个项目中存在多个.c文件时，在链接阶段一起链接生成可执行文件

Makefile的编写

1.什么是Makefile

是一个工程管理工具，本质上是一个文件，这个文件存放的是项目代码的编译规则，Makefile可以根据“时间戳”来决定工程内的文件是否需要参与本次的编译，也会检查文件的依赖关系



2.为甚要使用Makefile

a.避免多文件编译时在命令行输入比较长的编译命令，可以把编译规则写在Makefile文件里，减少工作量 直接终端输入make

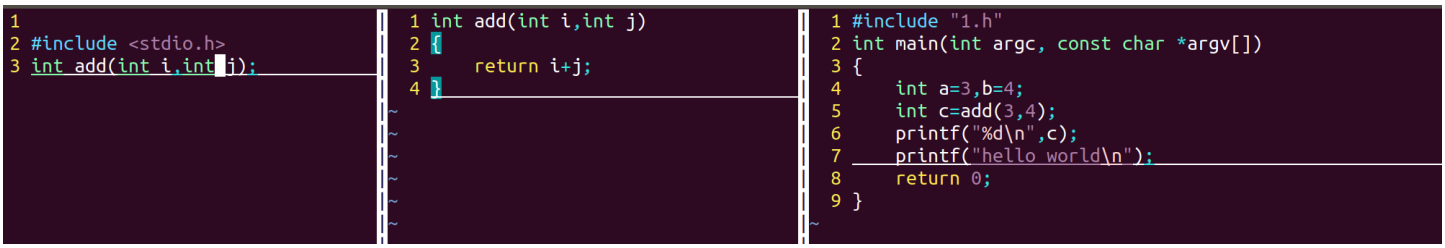
b.当修改工程代码时，有一些文件没被修改，没必要重新去编译，Makefile可以帮助我们完成这个时间戳的检查

c.在开发阶段，有些工程没有办法单纯通过gcc或者其他编译工具完成编译，需要借助makefile来完成编译工作

3.什么是make

make 是一个可执行文件，在/usr/bin,如果查看不带，说明没安装make,sudo apt-get install make 安装，make的作用是解析Makefile文件中的编译规则，根据编译规则编译工程

4.Makefile编写实例



```
1
2 #include <stdio.h>
3 int add(int i,int j);
```

```
1 int add(int i,int j)
2 {
3     return i+j;
4 }
```

```
1 #include "1.h"
2 int main(int argc, const char *argv[])
3 {
4     int a=3,b=4;
5     int c=add(3,4);
6     printf("%d\n",c);
7     printf("hello world\n");
8     return 0;
9 }
```

```
#注释用#
#Makefile里面编译是目标：依赖，后面指定编译规则
#最终生成的目标叫做aaa,一定放在最上面
#编译命令和左边栏之间有一个tab，不是空格
#目标文件名要顶格写
aaa:1.o 2.o
        gcc 1.o 2.o -o aaa

1.o:1.c
        gcc 1.c -c -o 1.o

2.o:2.c
        gcc 2.c -c -o 2.o
```

练习

把编译的四个步骤放在makefile里实现

```
aaa:1.o
        gcc 1.o -o aaa
1.o:1.s
        gcc 1.s -c -o 1.o
1.s:1.i
```

```
        gcc 1.i -S -o 1.s
1.i:1.c
        gcc 1.c -E -o 1.i

  写法2（不建议使用）：
  all:
        gcc 1.c -E -o 1.i
        gcc 1.i -S -o 1.s
        gcc 1.s -c -o 1.o
        gcc 1.o -o aaa
```

5.Makefile中的变量

变量的性质

1.Makefile中变量的使用方式和shell脚本比较类似，不需要定义变量，直接拿来用

```
var1=hahha

all:
        echo $(var1)
```

2.变量的引用：$(变量名)/${变量名}

3.在使用echo输出变量数值时也会把命令打印出来，如果不想打印，在echo前面+@

```
var1=hahha

all:
        @echo $(var1)
```

变量的赋值方式

1. ‘=’：这种赋值方式被赋值的变量数值始终保持跟赋值的变量一致

```
var1=hahha
var2=$(var1)
var1=nihao

all:
        @echo $(var1)
        @echo $(var2)
```

```
输出结果：
        nihao
        nihao
```

2. ':= ' ： 这种赋值方式和shell脚本里的赋值方式一样

```
  var1=hahha
var2:=$(var1)
var1=nihao

all:
        @echo $(var1)
        @echo $(var2)
 输出结果：
     nnihao
     hahha
```

3. '+=' :附加赋值：把在原来有数值的基础上再赋值

```
var1=hahha
var2=aaa
var2+=$(var1)
var1=nihao

all:
        @echo $(var1)
        @echo $(var2)
 输出结果：
     nihao
     aaa nihao
```

4.'?=':询问赋值

如果被复制的变量在本次赋值之前已经被赋值了，那么本次赋值不成立

```
var2=aaa
var2?=bbb
all:
        @echo $(var2)
 输出结果：aaa
```

6 Makefile里的命令行输入

make 变量=数值

```
ubuntu@ubuntu:~/3$ make var2=hhh
hhh
```

7.通过定义变量的形式对多文件编译的Makefile进行优化

```
#定义变量
#目标
TARGET:=aaa
#依赖
OBJS:=1.o 2.o
#指定编译器
CC:=gcc
#编译时添加的参数
CFLAGS:=-o
CFLAGSs:=-c -o

#开始写编译规则
$(TARGET):$(OBJS)
	$(CC) $(OBJS) $(CFLAGS) $(TARGET)

1.o:1.c
	$(CC) 1.c $(CFLAGSs) 1.o

2.o:2.c
	$(CC) 2.c $(CFLAGSs) 2.o
```

7.Makefile里的通配符

*：用于匹配任意数量的字符

%：用于编译阶段匹配特定字符

```
#定义变量
#目标
TARGET:=aaa
#依赖
OBJS:=1.o 2.o
#指定编译器
CC:=gcc
#编译时添加的参数
```

```
CFLAGS:=-o
CFLAGSs:=-c -o

#开始写编译规则
$(TARGET):$(OBJS)
	$(CC) $^ $(CFLAGS) $@

%.o:%.c
	$(CC) $*.c $(CFLAGSs)  $*.o
.PHONY:clean
clean:
	rm *.o $(TARGET)

```

8.Makefile里的伪目标

在编译命令下面指定伪目标，伪目标可以按照指令做一些事情

执行伪目标命令：make 伪目标名

```
#定义变量
#目标
TARGET:=aaa
#依赖
OBJS:=1.o 2.o
#指定编译器
CC:=gcc
#编译时添加的参数
CFLAGS:=-o
CFLAGSs:=-c -o

#开始写编译规则
$(TARGET):$(OBJS)
	$(CC) $(OBJS) $(CFLAGS) $(TARGET)

1.o:1.c
	$(CC) 1.c $(CFLAGSs) 1.o
2.o:2.c
	$(CC) 2.c $(CFLAGSs) 2.o

clean:
```

```
22          rm *.o $(TARGET)
23
24
```

注意：如果当前Makefile路径下存在和伪目标同名的文件存在，伪目标这时无法执行性成功

解决：使用.PHONY声明伪目标

```
1  .PHONY:clean
2  clean:
3          rm *.o $(TARGET)
```

9.Makefile里的特殊字符

$@:目标文件

$^:所有的依赖文件

$<:第一个依赖文件

$*:目标去除后缀后的所有字段

```
1  #定义变量
2  #目标
3  TARGET:=aaa
4  #依赖
5  OBJS:=1.o 2.o
6  #指定编译器
7  CC:=gcc
8  #编译时添加的参数
9  CFLAGS:=-o
10 CFLAGSs:=-c -o
11
12 #开始写编译规则
13 $(TARGET):$(OBJS)
14         $(CC) $^ $(CFLAGS) $@
15
16 %.o:%.c
17         $(CC) $< $(CFLAGSs)  $@
18 .PHONY:clean
19 clean:
20         rm *.o $(TARGET)
21
22
```