

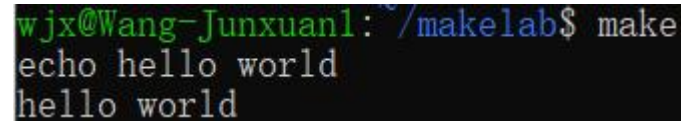
Part 0

Task 0

运行 make 之后，终端输出了

```
echo hello world
```

```
hello world
```



```
wjx@Wang-Junxuan1: ~/makelab$ make
echo hello world
hello world
```

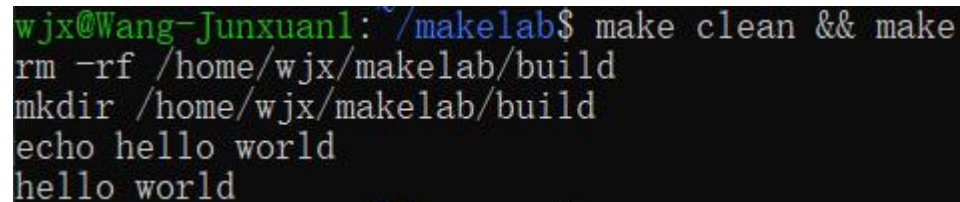
运行 make clean && make 之后，终端输出了

```
rm -rf /home/wjx/makelab/build
```

```
mkdir /home/wjx/makelab/build
```

```
echo hello world
```

```
hello world
```



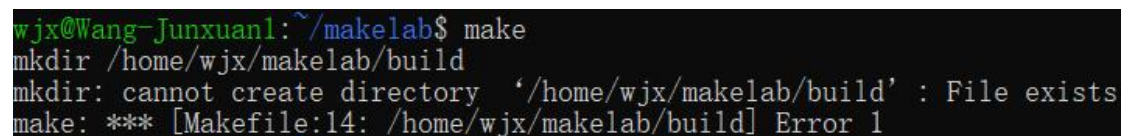
```
wjx@Wang-Junxuan1: ~/makelab$ make clean && make
rm -rf /home/wjx/makelab/build
mkdir /home/wjx/makelab/build
echo hello world
hello world
```

将 Makefile:4 .PHONY: clean all 改为 .PHONY: clean all \$(OUTPUT_DIR)之后运行 make，终端输出了

```
mkdir /home/wjx/makelab/build
```

```
mkdir: cannot create directory '/home/wjx/makelab/build': File exists
```

```
make: *** [Makefile:14: /home/wjx/makelab/build] Error 1
```



```
wjx@Wang-Junxuan1: ~/makelab$ make
mkdir /home/wjx/makelab/build
mkdir: cannot create directory '/home/wjx/makelab/build' : File exists
make: *** [Makefile:14: /home/wjx/makelab/build] Error 1
```

.PHONY 的效果：makefile 中将.PHONY 放在一个目标前就是指明这个目标是伪文件目标，即告诉 make 指定 target 列表不是文件名而是纯粹的任务，例如，make clean 命令的作用是删除一些文件，而如果目录下出现了文件名为 clean 的文件，在执行 make clean 时就不会 clean 目标下的命令了，对 all 的作用也同理。

make 的工作原理：执行 make 后面的 target 名称，如果只有一个单独的 make，就执行 Makefile 中的第一个 target，本例中是 all。

将 all 和 clean 标记为.PHONY 不是必须的，当目录下没有相应名称的文件名时可以省略

出现两个 hello world 的原因：正常情况下，make 会打印每条命令，然后再执行，这叫做回声 (echoing)，在命令的前面加上@即可关闭回声，使得屏幕上只有一个 hello world，如下图

```
wjx@Wang-Junxuan1:~/makelab$ make
hello world
all: $(OUTPUT_DIR)
ifeq ($(PART),)
    @echo hello world
```

Task 1

将 mkdir 改为 -mkdir 之后，当遇到错误时会输出错误并忽略它，然后继续执行
在语句后面加上 || true 则会直接认为该语句正确，不会打印错误，并继续执行后续的语句
我认为前者更好，显示错误并忽略它可以让用户看到哪里可能存在问题，直接认为语句正确可能会产生未知的风险，让错误更难被发现

我认为之后的 Makefile 更优，因为如果是原先的 Makefile 的话，当目录下存在待创建的文件时，\$(OUTPUT_DIR)会被识别为一个文件而非一个伪文件目标，从而系统会认为这就是最新的文件而不会有任何反应。当发生了文件名重复等问题时新的 Makefile 可以有效帮助我们发现问题。

Part 1

第二次运行 make PART=1 后，终端输出

```
make -j -C /home/wjx/makelab/build -f /home/wjx/makelab/mk/part1.mk
```

```
make[1]: Entering directory '/home/wjx/makelab/build'
```

```
make[1]: 'main' is up to date.
```

```
make[1]: Leaving directory '/home/wjx/makelab/build'
```

```
wjx@Wang-Junxuan1:~/makelab$ make PART=1
make -j -C /home/wjx/makelab/build -f /home/wjx/makelab/mk/part1.mk
make[1]: Entering directory '/home/wjx/makelab/build'
make[1]: 'main' is up to date.
make[1]: Leaving directory '/home/wjx/makelab/build'
```

程序进入了/home/wjx/makelab/build 目录下更新了 main（并没有重新编译它）

修改 src/main.cpp，再运行 make PART=1 后，终端输出

```
make -j -C /home/wjx/makelab/build -f /home/wjx/makelab/mk/part1.mk
```

```
make[1]: Entering directory '/home/wjx/makelab/build'
```

```
cp /home/wjx/makelab/src/main.cpp main.cpp
```

```
g++ -I/home/wjx/makelab/include -c -o main.o main.cpp
```

```
main.cpp:4:16: warning: comma-separated list in using-declaration only available with
'-std=c++17' or '-std=gnu++17'
```

```
    4 | using std::cout, std::endl;
      |                   ^
```

```
g++ -o main A.a.o some.a.o B.b.o main.o
```

```
rm main.cpp
```

```
make[1]: Leaving directory '/home/wjx/makelab/build'
```

如下图

```
wjx@Wang-Junxuan1: ~/makelab$ make PART=1
make -j -C /home/wjx/makelab/build -f /home/wjx/makelab/mk/part1.mk
make[1]: Entering directory '/home/wjx/makelab/build'
cp /home/wjx/makelab/src/main.cpp main.cpp
g++ -I/home/wjx/makelab/include -c -o main.o main.cpp
main.cpp:4:16: warning: comma-separated list in using-declaration only available with
c++17' or '-std=gnu++17'
    4 | using std::cout, std::endl;
      |
g++ -o main A.a.o some.a.o B.b.o main.o
rm main.cpp
make[1]: Leaving directory '/home/wjx/makelab/build'
```

程序进入了/home/wjx/makelab/build 目录下重新编译了 main

修改 include/shared.h, 再运行 make PART=1 后, 终端输出

```
make -j -C /home/wjx/makelab/build -f /home/wjx/makelab/mk/part1.mk
make[1]: Entering directory '/home/wjx/makelab/build'
make[1]: 'main' is up to date.
make[1]: Leaving directory '/home/wjx/makelab/build'
```

如下图

```
wjx@Wang-Junxuan1: ~/makelab$ make PART=1
make -j -C /home/wjx/makelab/build -f /home/wjx/makelab/mk/part1.mk
make[1]: Entering directory '/home/wjx/makelab/build'
make[1]: 'main' is up to date.
make[1]: Leaving directory '/home/wjx/makelab/build'
```

增量编译的实现: 利用如果 target 所依赖的文件修改时间比该 target 早或该 target 不存在, 则执行后面的命令的特性实现增量编译, 否则再在当前文件中找目标为 target 所依赖的文件的依赖性, 重复上述步骤的特性来实现增量编译, 即如果发现某个.c 文件的修改时间晚于其对应的.o 文件, 则重新编译该.c 文件。

对于头文件的增量编译: 本部分代码中似乎并没有对头文件的增量编译有专门的处理, 应该在每个.o 文件的依赖项后面添加上其对应的.c 代码中包含了的头文件, 意思是, 如果某个头文件被修改了, 则要重新编译所有包含了该头文件的.c 文件。

Task 3

注释掉 include/shared.h:1 #pragma once , 运行 make clean && make PART=1 之后, 编译器报错, 显示全局变量 std::string MassSTR 和全局函数 static int LenOfMassSTR()被重复定义了。#pragma once 的用处是保证同一个文件不会被包含多次。可以避免此问题。

删去 include/shared.h:5 static std::string MassSTR 中的 static , 再运行 make clean && make PART=1, 编译报错, 显示 MassSTR[abi:cxx11]被多重定义, 查看符号表信息, 发现在 some.a.o, A.a.o, B.b.o, main.o 中的 MassSTR[abi:cxx11]的 BIND 列都是 g, 即 global 表示他们都是全局符号。而加上 static 之后他们被定义为静态局部变量, 查看符号表信息, 他们的 BIND 列都是 L, 即 local, 表示模块内部符号, 对外不可见, 就不会引起冲突。static 的作用是让变量尽在本文件内可见。

main.cpp 和 A.cpp 中的 a, b, d 不会出现冲突, 他们不会发生冲突的原因都是一个强符

号一个是弱符号。。当两者同时存在时，编译器会选择强符号，不出现多个同名的强符号时避免冲突的关键。其中，在 A.cpp 中，a 被设置了变量属性为常量，b 被设置为了弱符号，d 是一个常整形。而 c 是两个文件中都可见的一个变量，在 A.cpp 中，c 是外部变量的声明。

Task 4

将 include/shared.h:7 static int LenOfMassSTR() 中的 static 删去，运行 make clean && make PART=1，编译器报错，显示 LenOfMassSTR()被多重定义了。

添加 inline 时，函数 LenOfMassSTR 的 bind 都是 local，未被标记时则都是 global，这说明 inline 是一个弱符号，从而避免了冲突。这种做法存在，因为 inline 函数可能会被其他同名函数所替代，导致调用该函数时没有实现预期的功能。

如果一个函数被标记为 static inline 那么就会成为一个只在本文件中可见的一个内联函数

对于小型工具函数，用 static inline 定义的方式最优，因为这样既不会在其他文件定义使用同名的函数时引起错误调用，也不会产生链接时的冲突。

Part 2

Task 5

通过输出的静态的链接库头信息和符号表信息，发现是链接了 notA.o 而不是 A.o，查阅资料发现这是因为编译器在寻找符号时是从前往后找符号的，在 part2 中，libA.a 的依赖项后面的顺序是 notA.a.o A.a.o some.a.o，所以函数 A () 就保留了 notA.a.o 中的版本，如果交换该处 notA.a.o 与 A.a.o 的顺序，会发现函数在 main 中的函数 A()调用的就是 A.cpp 中的版本

Task 6

更改 mk/part2.mk:3 \$(OUTPUT):后的 main.o libB.a libA.a 为 libA.a libB.a main.o，然后运行 make clean && make PART=2，编译器报错如下

```
/usr/bin/ld: main.o: in function `main':  
main.cpp:(.text+0x24): undefined reference to `A()'  
/usr/bin/ld: main.cpp:(.text+0x29): undefined reference to `B()'  
collect2: error: ld returned 1 exit status  
make[1]: *** [/home/wjx/makelab/mk/part2.mk:4: main] Error 1  
main 中的函数 A, B 均找不到定义。
```

更改 mk/part1.mk:3 \$(OUTPUT): 后的 A.a.o some.a.o B.b.o main.o 为 main.o some.a.o B.b.o A.a.o 然后运行 make clean && make PART=1，编译成功，且运行可执行文件能够得到正确的输出

上面两现象发生的原因是 gcc 在链接多个静态库时是有顺序的，即按照链接参数给定的顺序一次读入，如第一个 Task6 中的第一个实验现象中，如果首先读入的是 main.o，此时发现 A()函数还未定义，会加入未定义符号表，然后读入 libA.a 时发现 A()在这里面定义了，此时把 A()从未定义符号表中取出。链接完成后检查未定义表，发现没有未定义的符号，则链

接正常。反过来，修改顺序之后，发现 A()并没有在未定义符号表中，则编译器不会把 A()链接到最后的可执行文件中，然后再读取 main.o 就会发现 A()没有定义，编译结束之后就会报错。而链接.o 文件时不需要遵守顺序规范，因为链接器寻找符号时一定会在所有.o 文件中寻找，所以 Task6 的第二个实验现象中随意交换位置都能编译通过。

Linux 系统运行程序时查找动态链接库的顺序如下 1. gcc 编译时指定的运行时库路径 -Wl,-rpath 2. 环境变量 LD_LIBRARY_PATH 3. ldconfig 缓存 /etc/ld.so.cache 4 系统默认库位置 /lib 或 /usr/lib，由于不能重新编译，所以想要运行该程序需要把 libA.so 与 libB.so 复制到 usr/lib 目录下，之后不进入 build 直接使用 build/main 或进入 build 使用 ./main 命令都可以正确运行程序运行。

Task 8

可以编译执行的原因：是 libB.so 中的 notA 中的函数 A()参与了链接。程序是根据链接的顺序选择是哪一个函数的，在 part3.mk 中，\$(output)后的依赖项的顺序 himain.o libB.so libA.so 。

将顺序更改为依赖项顺序更改为 main.o libA.so libB.so 之后再运行 make clean && make PART=3，编译成功，进入 build 目录之后运行 main，输出如下

```
wjx@Wang-Junxuan1:~/makelab/build$ ./main
我是A哒 18216
我是B哒
测试成功!
main: 18216
```

此时 A()是 A.cpp 中的函数 A()，说明 Task 6 中的规律对于动态链接库也适用

注释掉 mk/part3.mk:3 CPPFLAGS += -fPIC 之后运行 make clean && make PART=3，终端报错如下

```
/usr/bin/ld: A.a.o: relocation R_X86_64_PC32 against symbol `__ZSt4cout@@GLIBCXX.3.4' can not be used when making a shared object; recompile with -fPIC
/usr/bin/ld: final link failed: bad value
collect2: error: ld returned 1 exit status
make[1]: *** [/home/wjx/makelab/mk/part3.mk:12: libB.so] Error 1
make[1]: *** Waiting for unfinished jobs....
make[1]: *** [/home/wjx/makelab/mk/part3.mk:9: libA.so] Error 1
rm some.a.cpp main.cpp notA.a.cpp A.a.cpp B.b.cpp
make[1]: Leaving directory '/home/wjx/makelab/build'
make: *** [Makefile:10: all] Error 2
```

要求加上 -fPIC

-fPIC 的作用是生成位置无关代码，即让编译器产生的代码只含有相对地址，这样在共享库被加载时他在内存的位置就不是固定的，可以把代码拷贝到需要的地方使用。CPPFLAGS 是预处理参数，加上 fPIC 之后能够多个进程共享 so 文件，一个库的代码在不同程序中地址不同，而操作系统会把他们映射到同一块物理内存上。