ICS Lab4-Makelab 实验报告

Part 0

Task 0

- **运行make**，打印如下

```
echo hello world
hello world
```

- **运行make clean && make**, 打印如下

```
rm -rf /home/zou/makelab/makelab/build
mkdir /home/zou/makelab/makelab/build
echo hello world
hello world
```

- **更改.PHONY,运行make**,结果如下

```
mkdir /home/zou/makelab/makelab/build
mkdir: cannot create directory ‘/home/zou/makelab/makelab/build’: File
exists
Makefile:14: recipe for target '/home/zou/makelab/makelab/build' failed
make: *** [/home/zou/makelab/makelab/build] Error 1
```

- **.PHONY的效果和make的工作原理**

  .PHONY效果：如果make 命令后面跟的参数如果出现在.PHONY 定义的伪目标中，那么就直接会执行伪目标的命令，不在乎是否有同名文件等

  make工作原理：命令行执行make命令，会在该目录下寻找Makefile文件，并寻找该文件的第一个目标文件，接下来make会一层又一层地去找文件的依赖关系，直到最终编译出第一个目标文件。

- **在本项目中将all 和 clean 标记为.PHONY并不是必须的**

  但若目录下存在名为"all"或者"clean"的文件且没有依赖文件，将始终最新，如果没有.PHONY指定其为伪目标，在运行make all或者make clean，命令规则会失效，make clean 和make all永远不会执行

  因此还是建议将两者标记为.PHONY

- **输出两个hello world的原因**

  Makefile中**echo hello world**语句中，echo会使得make执行命令前将执行的命令进行输出(包括打印 echo)（又称为回显）,然后再次打印执行的命令内容

- **只输出一个hello world的方法**

  在echo前面加@,则不会回显

Task 1

- **在makedir前添加-运行make**,打印结果如下

```
mkdir /home/zou/makelab/makelab/build
mkdir: cannot create directory ‘/home/zou/makelab/makelab/build’: File
exists
Makefile:14: recipe for target '/home/zou/makelab/makelab/build' failed
make: [/home/zou/makelab/makelab/build] Error 1 (ignored)
echo hello world
hello world
```

- **将mkdir那一行更改为 mkdir $(OUTPUT_DIR) || true**，现象为

```
mkdir /home/zou/makelab/makelab/build || true
mkdir: cannot create directory ‘/home/zou/makelab/makelab/build’: File
exists
echo hello world
hello world
```

- **两种忽略略错误方法分析比较**

  第一种是指make在运行时遇到error自动忽略并继续执行**但是会抛出对应的error信息再退出**，第二种是与true取||代表不管是否遇到错误这条指令属于正常执行最后**以正常状态结束退出**，我觉得**第一种更好**，首先它既能像第二种方法保证错误忽略并继续执行，同时又能给用户抛出对应的错误信息，可以方便用户进行更正或者维护

- **忽略错误之后将$(OUTPUT_DIR)标记为.PHONY的Makefile与原先的比较**

  原先的Makefile在执行$(OUTPUT_DIR)语句时，需要进行逻辑判断——当前文件是否最新从而考虑是否执行命令mkdir；而更改后的Makefile由于将$(OUTPUT_DIR)标记为伪目标，每次其对应的命令无需判断都会执行，并且命令mkdir创建目录失败将忽略错误继续执行；两者得到的运行效果相似，但效率上有差异，比如我们使用Makefile时make clean之后执行make，那么后者的效率更高，因为无需判断便会执行mkdir，而前者需要判断后才会执行mkdir

Part 1

Task 2

- **运行make PART=1的现象如下**

```
mkdir /home/zou/makelab/makelab/build
mkdir: cannot create directory ‘/home/zou/makelab/makelab/build’: File
exists
Makefile:14: recipe for target '/home/zou/makelab/makelab/build' failed
make: [/home/zou/makelab/makelab/build] Error 1 (ignored)
make -j -C /home/zou/makelab/makelab/build -f
/home/zou/makelab/makelab/mk/part1.mk
make[1]: Entering directory '/home/zou/makelab/makelab/build'
make[1]: 'main' is up to date.
make[1]: Leaving directory '/home/zou/makelab/makelab/build'
```

可以看到这里是嵌套make的实现，进入mk/part1.mk发现main.cpp是最新的（即未被修改），则退出执行

- **修改src/main.cpp,再次运行make PART=1的现象**

```
mkdir /home/zou/makelab/makelab/build
mkdir: cannot create directory ‘/home/zou/makelab/makelab/build’: File
exists
Makefile:14: recipe for target '/home/zou/makelab/makelab/build' failed
make: [/home/zou/makelab/makelab/build] Error 1 (ignored)
make -j -C /home/zou/makelab/makelab/build -f
/home/zou/makelab/makelab/mk/part1.mk
make[1]: Entering directory '/home/zou/makelab/makelab/build'
make[1]: 'main' is up to date.
make[1]: Leaving directory '/home/zou/makelab/makelab/build'
cp /home/zou/makelab/makelab/src/main.cpp main.cpp
g++  -I/home/zou/makelab/makelab/include  -c -o main.o main.cpp
main.cpp:4:16: warning: comma-separated list in using-declaration only
available with -std=c++1z or -std=gnu++1z
 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/zou/makelab/makelab/build'
```

相比刚才，由于对main.cpp修改，因此进入part1.mk后又重新编译main.cpp，然后才离开build目录

- **修改include/shared.h,再次运行make PART=1的现象**

结果与初次运行make PART=1一样，即未检测到shared.h的更改

- **增量编译的实现**

```
$(OUTPUT): A.a.o some.a.o B.b.o main.o
    $(CXX) -o $@ $^
```

上为mk/part1.mk的部分代码，可以看到其目标文件为$(OUTPUT)，make在执行时会自动比较其最终的依赖文件 .cpp（因为make会根据隐含规则把.cpp设置为.o的依赖文件）和目标文件的修改时间，如果前者修改时间大于目标文件，则执行下一条语句重新编译

- **如何处理设计头文件的增量编译**

由于.o自动依赖.cpp文件，因此我们对.cpp的修改make会识别出来，按照这种规则，我们只需要在config.mk文件中增加对头文件的依赖(如下)

```
%.cpp: $(SRC)/%.cpp $(HOME)/include/A.h $(HOME)/include/B.h
$(HOME)/include/shared.h
    cp $< $@

%.a.cpp: $(SRC)/A/%.cpp $(HOME)/include/A.h $(HOME)/include/shared.h
    cp $< $@

%.b.cpp: $(SRC)/B/%.cpp $(HOME)/include/B.h
    cp $< $@
```

增加.cpp对头文件的依赖，这样对.h的修改make才能发现

Task 3

- **注释掉shared.h中的#pragma once运行结果**

  注释掉之后运行将会报错， note: 'int LenOfMassSTR()' previously defined here，就是说**shared.h的函数被重复编译**，make执行失败

  #pragma once的作用：能够保证头文件只被编译一次

- **变量名前删去static的运行**

  结果显示**MassSTR**已经在A.a.o中被定义，因此在编译some.cpp main.cpp中由于包含了A.h头文件，而A.h包含了shared.h,变量MassSTR再次被定义，重定义导致make报错，执行失败

  static的效果：

  如下是删去static后objdump -Ct build/*输出的符号表信息中关于变量**MassSTR**的说明，可以看到其为GLOBAL即对外可见

  0000000000000000 g   O .bss   0000000000000020 MassSTR[abi:cxx11]

  因此可以看出定义全局变量使用static可以将该变量或者函数只限制于定义它的源文件中（LOCAL），其他源文件不能访问，而若该头文件被其他头文件包含，那么该变量或者函数将被重新定义

- **避免链接冲突的原因**

  - 变量a

    A.cpp中显式定义变量a存放在.common字段（弱符号），main.cpp中变量a定义为int a,为未初始化全局变量（弱符号），根据链接器处理多重定义的符号名的规则3——**当有多个弱符号同名，将从这些弱符号中任意选择一个**，从打印的符号表可以发现链接器选择将main.cpp中的a将其放在.bss段，这样的链接过程并不会发生冲突

  - 变量b

    A.cpp中显式定义**变量b为弱符号**, main.cpp中的b为未初始化全局变量，**也为弱符号**，同上，根据链接器处理多重定义的符号名的规则3，可以知道并不会发生重定义冲突

  - 变量c

    A.cpp中在变量c前加**extern**关键字，因此只是声明并未分配存储空间，真正的定义是在main.cpp中，因此不存在重定义，也就不会发生冲突

  - 变量d

    A.cpp中变量d用**const**修饰，放在常量区(.rodata段)，并且const修饰的全局变量默认是内部链接（效果同static相似），因此编译和链接中main.cpp和A.cpp中的变量d并不会识别为重定义

Task 4

- **函数名前删去static的运行结果**

  同前，执行make后会报错，因为函数LenofMassSTR重复定义，链接时发生冲突

- **inline避免链接发生冲突的原因**

  从打印的符号表中截取如下

  0000000000000000 w   F .text._Z12LenOfMassSTRv   0000000000000012 LenOfMassSTR()

  可以看到函数LenOfMassSTR是弱符号，因此同前分析链接时不发生冲突

用inline避免链接冲突做法的弊端：

1.事实上函数声明为inline也只是建议编译器将其视为内联函数，若函数体指令太多编译器也可将其视为非内联函数进行编译，这种情况下函数如果被多个头文件include则会出现重定义冲突

2.同时inline函数可能被其他同名但功能不同的函数替代，从而影响代码的逻辑性，产生不可预期的效果

- **static inline联合使用**

  一方面可以解决inline的上述两个问题，另一方面当函数体较小static inline比static占用更少的内存空间

- **最优定义方式**

  static inline修饰最优，原因见上一条

Part 2

Task 5

- **执行notA.cpp中的void A()的原因**

  程序首先解析main.o时，A()和B()函数被放在集合U，然后解析静态链接库libB.a，将B.b.o放进集合E中以及将B()从集合U移到集合D，然后扫描libA.a, 按照顺序首先解析notA.o，此时发现notA.o中有A()则main.o中的A()有了定义，并将其从U移到集合D中，此时集合U为空，继续扫描直至集合U和集合D不发生变化，由于链接只注意集合U中是否有相关定义，不注意集合D中是否重复定义，因此在扫描A.a.o中并不会发生冲突，可以完成正常编译和链接，并且在运行main.cpp时A()执行的是notA.cpp中的A()函数

- **上述设计的利弊**

  利：由于静态链接只关心UNDEF的符号，因此一定程度上可以避免同名函数或者变量的带来的链接冲突

  弊：函数的运行结果与扫描顺序有关，并且链接对象顺序不当容易导致链接失败

Task 6

- **更改mk/part2.mk链接对象顺序运行结果**

  make执行失败，在解析main.o时A() B()是未定义的，即此时的集合U非空

- **更改mk/part1.mk链接对象顺序为B.b.o some.a.o A.a.o main.o运行结果**

  g++编译顺序变化，更改前为

```
g++  -I/home/zou/makelab/makelab/include  -c -o A.a.o A.a.cpp
g++  -I/home/zou/makelab/makelab/include  -c -o some.a.o some.a.cpp
g++  -I/home/zou/makelab/makelab/include  -c -o B.b.o B.b.cpp
g++  -I/home/zou/makelab/makelab/include  -c -o main.o main.cpp
```

  更改后为

```
g++  -I/home/zou/makelab/makelab/include  -c -o B.b.o B.b.cpp
g++  -I/home/zou/makelab/makelab/include  -c -o some.a.o some.a.cpp
g++  -I/home/zou/makelab/makelab/include  -c -o A.a.o A.a.cpp
g++  -I/home/zou/makelab/makelab/include  -c -o main.o main.cpp
```

但仍正常执行make

- **链接对象顺序对链接的影响及其原因**
  - 链接对象均为输入文件均为可重定位目标文件

    如果运行程序依赖的所有目标文件均在连接命令行，则顺序不会影响最终结果，make都会正常执行
  - 链接对象包括可重定位目标文件和存档文件

    若输入文件依赖存档文件某个目标文件，则链接时命令行中存档文件必须在该文件的后面，否则以来的目标文件将一直放在集合U中，导致最终集合U非空，编译失败

Part 3

Task 7

- **直接在根目录下运行build/main**

  现象：build/main: error while loading shared libraries: ./libB.so: cannot open shared object file: No such file or director

  程序报错，无法打开libB.so文件

  原因：g++进行动态链接时，动态库的代码不会被打包到可执行程序中，而是通过加载器通过文件地址查询到动态链接库加载到指定内存，直接在根目录下运行build/main无法找到libB.so,也就不能正常执行

- **如何运行程序**

  由于Linux中的动态库，系统默认会在**/lib**和**/usr/lib**中进行搜索，解决方法是运行sudo cp libA.so /usr/lib/和sudo cp libB.so /usr/lib/，将libA.so和libB.so文件复制到/usr/lib/中，从而就可以在运行时找到，之后进入build目录正常运行./main即可

Task 8

- **链接冲突仍能编译执行分析**

  与静态库中的避免链接冲突类似，在part3.mk中$(OUTPUT): main.o libB.so libA.so ，说明libB.so比libA.so更靠前，而notA.a.o是liB.so的依赖文件,因此main执行的是notA.cpp中的A()函数

- **改变链接对象运行现象**
  - 仅交换libB.so 和 libA.so，仍能运行，且此时main里面用的是A.cpp中的函数A()
  - 若main.o的顺序不在第一个，则编译报错，出现未定义函数

  说明Task6中讨论的规律对动态链接库也适用

- **注释CPPFLAGS += -fPIC的现象**

  此时.so文件的代码段需要重定位，无法实现动态链接，因此make执行报错

  -fPIC的作用：为动态链接发布恰当的、位置无关代码（position-independent code，即PIC），来避免GOT（Global Offset Table）的大小限制