子课程 - Linux C Programming

GCC文件

跟我一起写 Makefile文件

Make文件

创建一个真实的C项目作业

将过去的某个程序重构一个代码库。

借鉴类似gcc-3-real-project的代码结构，要求

•功能模块化为多个源代码 (\*.c/cpp) 文件

•每个独立的模块应该有独立的 \*.h file

•尝试使用 -c 参数编译项目

•写一个简单的Makefile 进行自动化编译

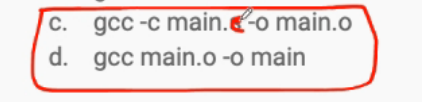
提交内容为项目打包的文件，接受文件类型：

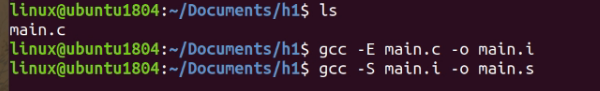
存档（ZIP） .zip

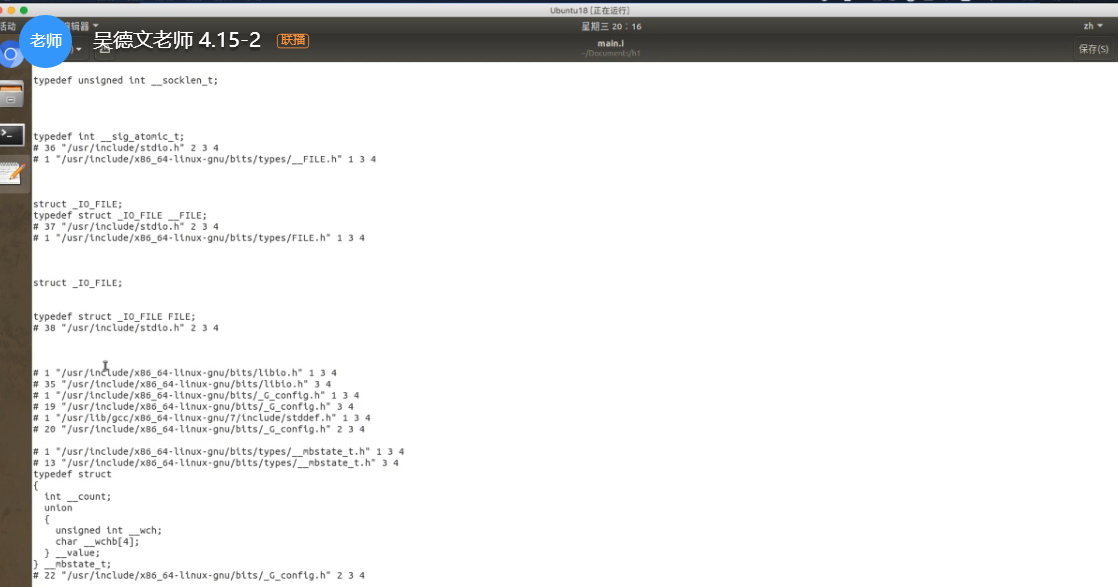
实验报告 22920182204307 王泽宇

理论知识部分：

编译一个程序，只需要如下2步：main.c是当前文件夹下面的文件



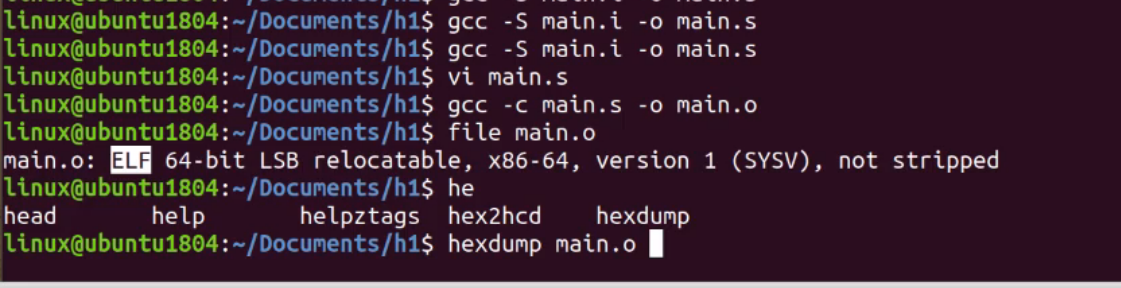


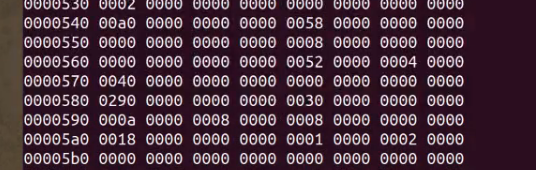


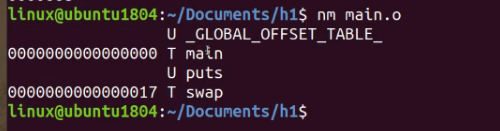
第一步是预处理：将main.c变成main.i中，导入头文件



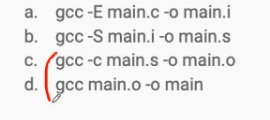
第二个步骤是将main.i变成main.s，原有代码转换成汇编代码，会去掉没有使用的变量；





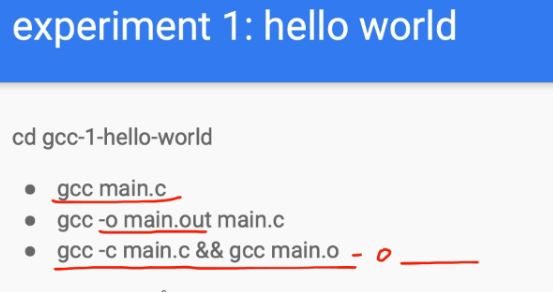


第三步将main.s文件转换成main.o二进制代码文件；得到ELF类型的BLN文件



完整步骤可以随时控制

第四步将汇编代码和其他的代码模块连接在一起

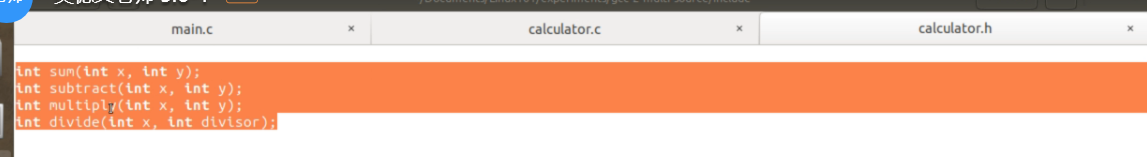


-E -S参数也可以使用；其实就是将上面提到的四个步骤执行一下

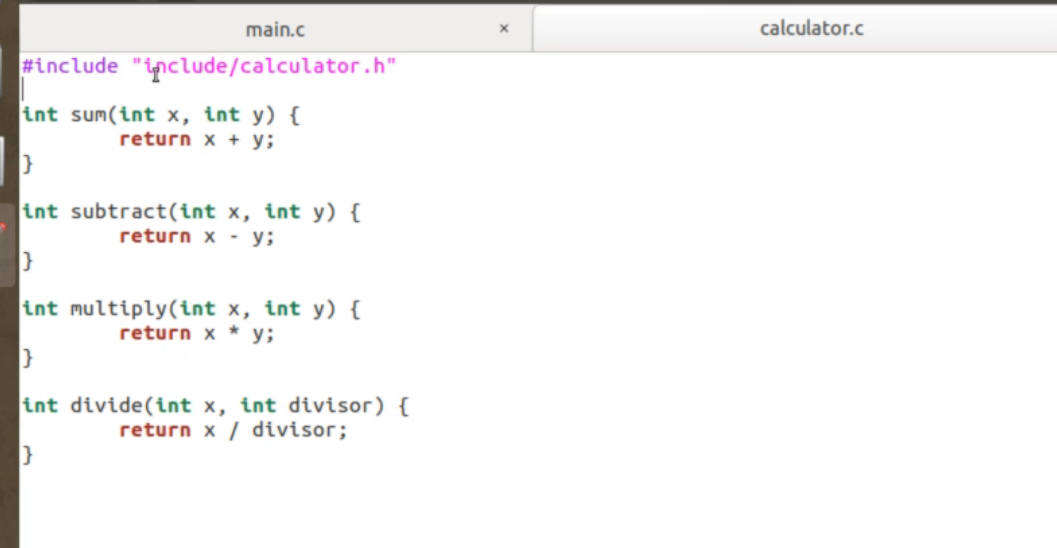
这四步写完之后直接输出程序的运行结果；

第二部分 头文件 5.6-1

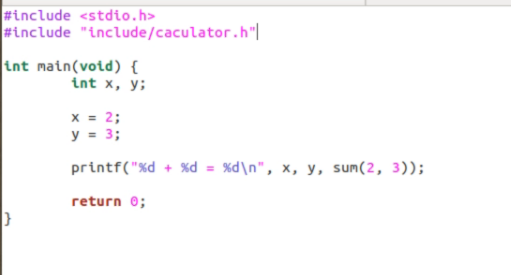
将声明和实现分开进行：



声明部分



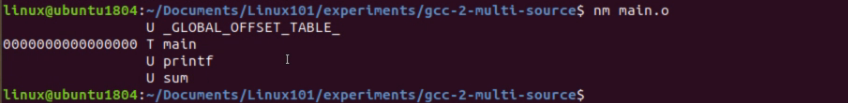
cal.c文件



实现部分 下面的main.c是直接实现的部分，include之后sum就有定义了；

所以其实是“little boring”的一个操作：

原来main.c程序中有主函数，有sum,mul等四个四则运算函数，同时有这四个函数的声明；现在将这四个函数放到另外一个.c程序中单独编译，最后拼接；同时将声明放到.h文件中，在四函数.c文件中使用include进行声明；

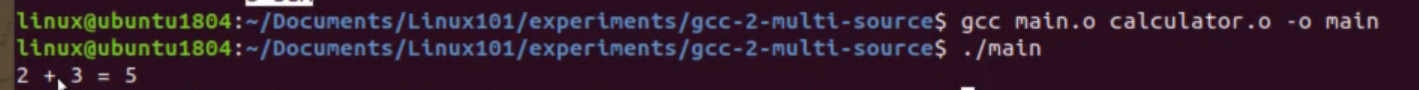


执行到第三步时，查看汇编文件；





cal.c中有sum，main中没有；



因此要将2部分都编译之后拼接起来才能够达到最终的效果；

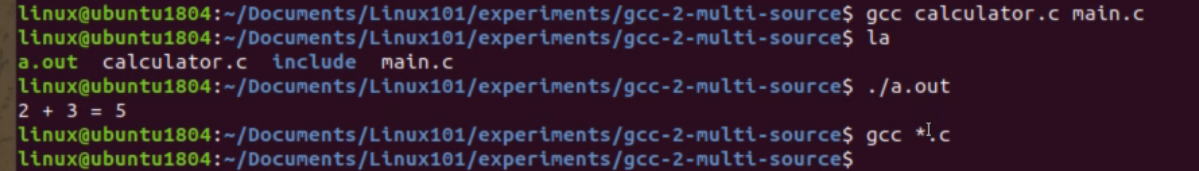
拼接的过程就是把2个main.o写在一起，然后 -o main进行编译；

所以前三步就是编译所有的c和h文件，最后一步再进行link，将它们连接在一起；

总之，就是将c和h文件配合起来使用



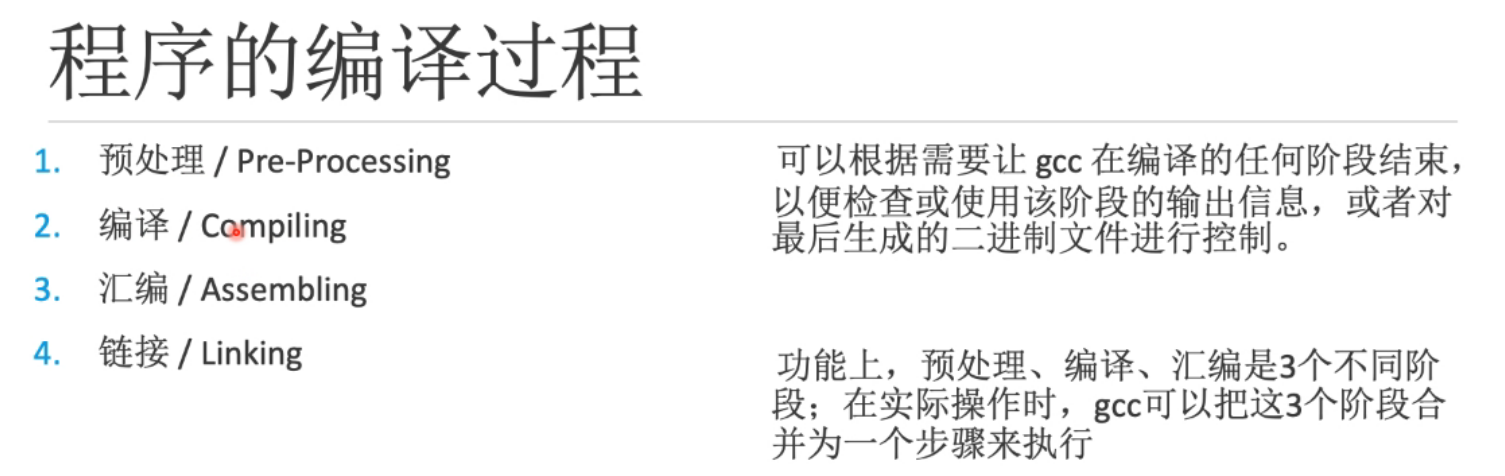
这就是planB



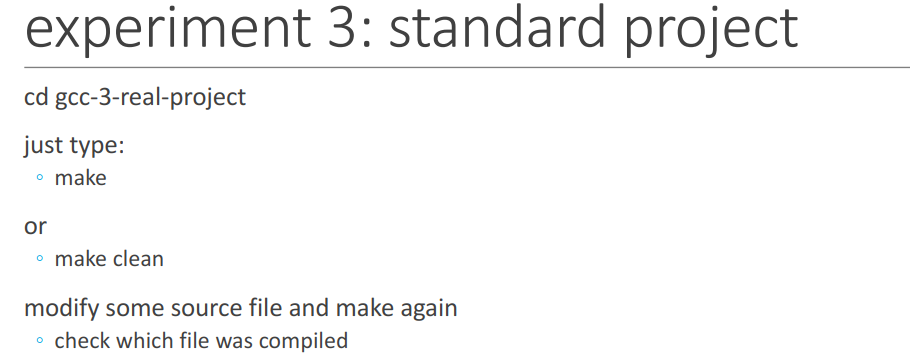
也可以直接放到一起编译；例如\*就是代表任意字符的意思

源代码有问题所以步骤都要重新编译一遍

总结

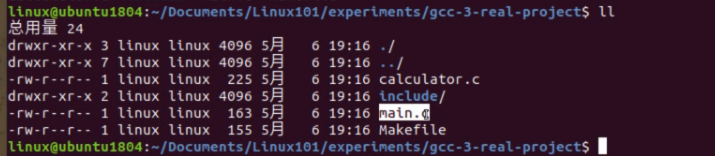


]

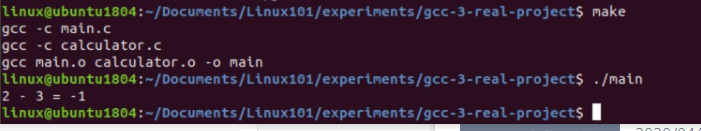


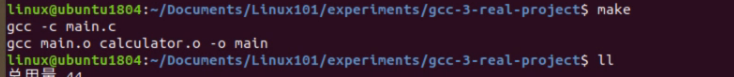
2、编译的时候还会进行优化，使用planB就会减少编译的时间

到那时这样执行的命令太多；

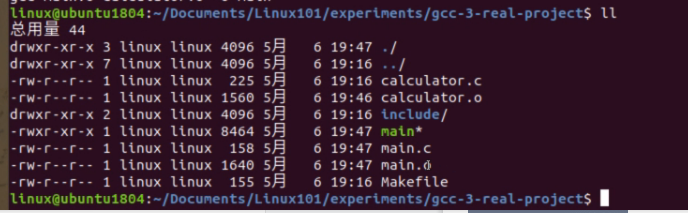


所以使用makefile





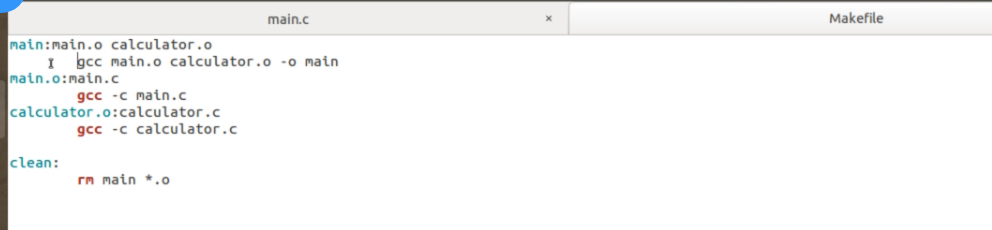
加一个make就全部都执行了



而且全部保持最新，



重新编译实现main.o main



makefile必须使用tab

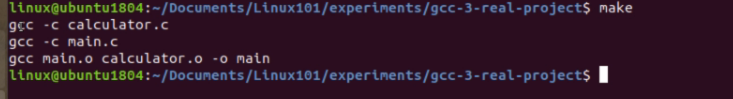
规则

main:规则名称 要依赖的第二条规则

二 要依赖的第三个文件

要依赖的下一个步骤…

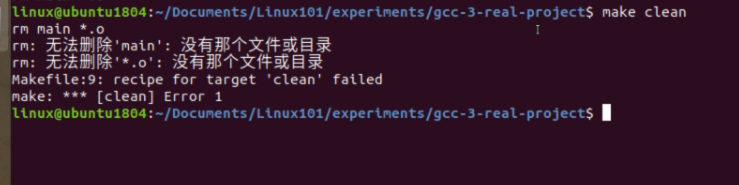
clean



这样就能够从上到下、从下到上地跑一遍，就把程序执行完了；

绿色黑色部分都是规则；

总之，makefile由规则、依赖和action组成



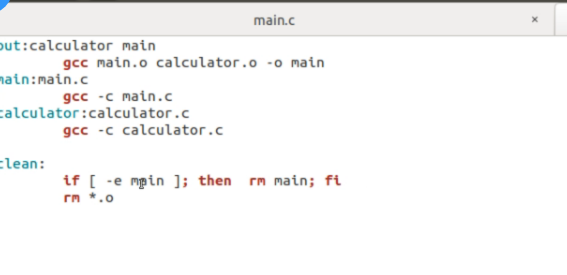
这里出错了，原因是上面已经删除完了

如何在没有文件的时候先判断一下，从而不进行删除文件的多余操作

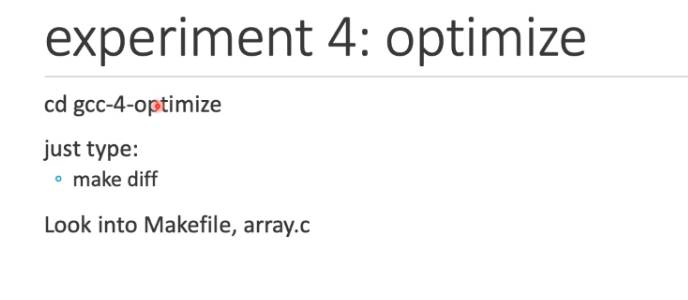
（之所以出现这种情况是因为makefile中本来就包含有这个命令）



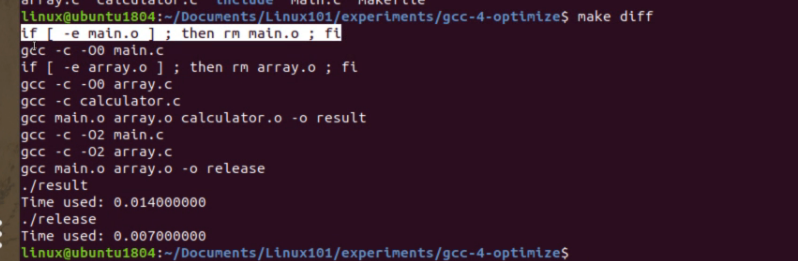
使用这个命令解决，使用了shell脚本



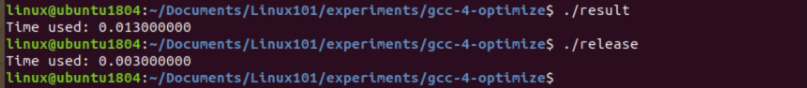
更新一下makefile



第四个步骤：优化：



删文件，再编译，优化、执行（得到release）说明耗时

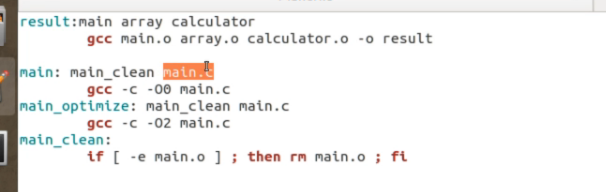


执行./result ./release都会得到相应地执行时间

make diff的构造方法

依据result和release

说明diff之后这2个文件就能够生成了；



这是 result的依赖路径；

注意使用了o0 o2进行优化；o1 o2 o3优化效果越来越好 90 95 99

o2的是经常使用的

main的编译工作不优化

main\_op是优化的

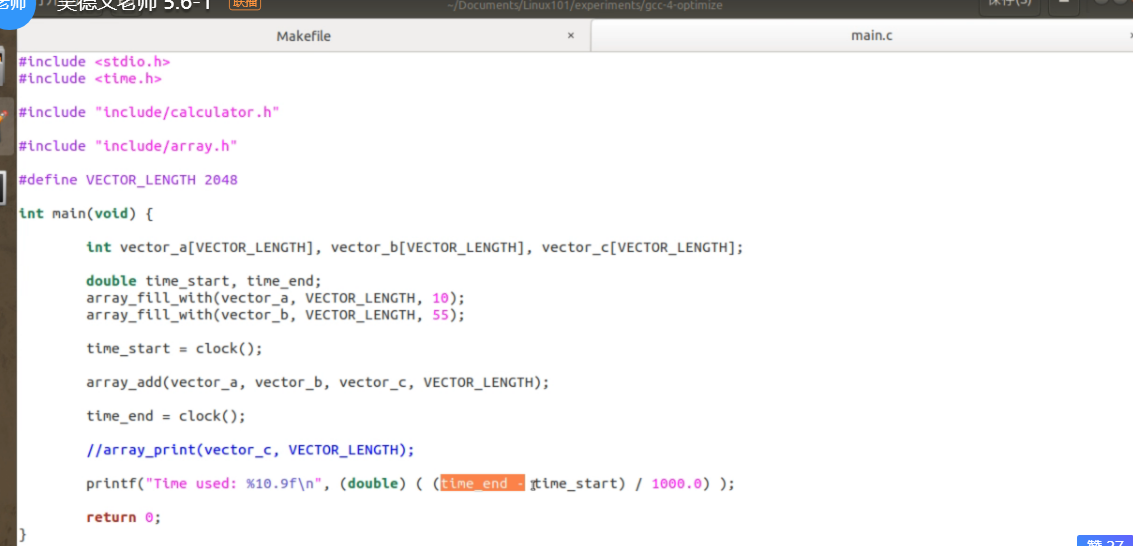
上面的代码中：release有优化，result没有优化

cal没分优化

array有做优化

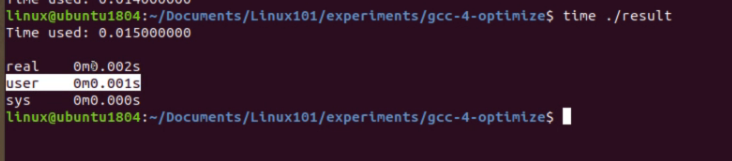
计时系统：

程序执行的时候，有一个时间戳



clock可以计算出走了多长时间，2个时间相减可以除以1000得到s；

因为每s有1000下

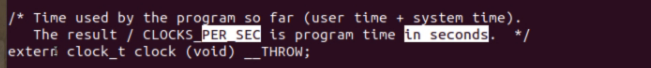


里面的分块时间也可以显示；



但是有时候时间显示的会不准

这是因为除以1000的时候发生了问题，在Linux下面是除以1000,000

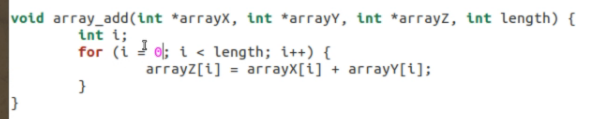


可以直接用CLOCK\_PER\_SEC来进行表示





得到正确时间



优化的具体原理：

批量地读入数组，自动把for中的加法程序编程并行，16位同时去加