从四个问题透析 Linux 下 C++ 编译&链接

高效程序员 2021-06-21 08:53

收录于合集

#C++

103个

链接 | https://my.oschina.net/u/4526289/blog/4651990

【导读】：编译与链接对C&C++程序员既熟悉又陌生，熟悉在于每份代码都要经历编译与链接过程，陌生在于大部分人并不会刻意关注编译与链接的原理。本文通过开发过程中碰到的四个典型问题来探索64位linux下C++编译&链接的那些事。

以下是正文

编译原理

将如下最简单的C++程序（main.cpp）编译成可执行目标程序，实际上可以分为四个步骤:预处理、编译、汇编、链接，可以通过

g++ main.cpp –v看到详细的过程，不过现在编译器已经把预处理和编译过程合并。

预处理：g++ -E main.cpp -o main.ii，-E表示只进行预处理。预处理主要是处理各种宏展开；添加行号和文件标识符，为编译器产生调试信息提供便利；删除注释；保留编译器用到的编译器指令等。

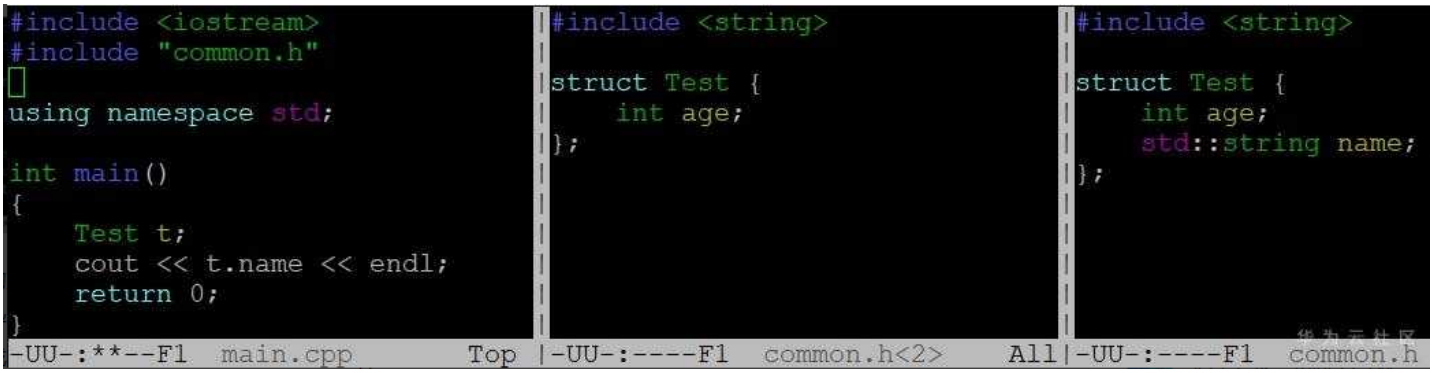
编译：g++ -S main.ii –o main.s，-S表示只编译。编译是在预处理文件基础上经过一系列词法分析、语法分析及优化后生成汇编代码。

汇编：g++ -c main.s –o main.o。汇编是将汇编代码转化为机器可以执行的指令。

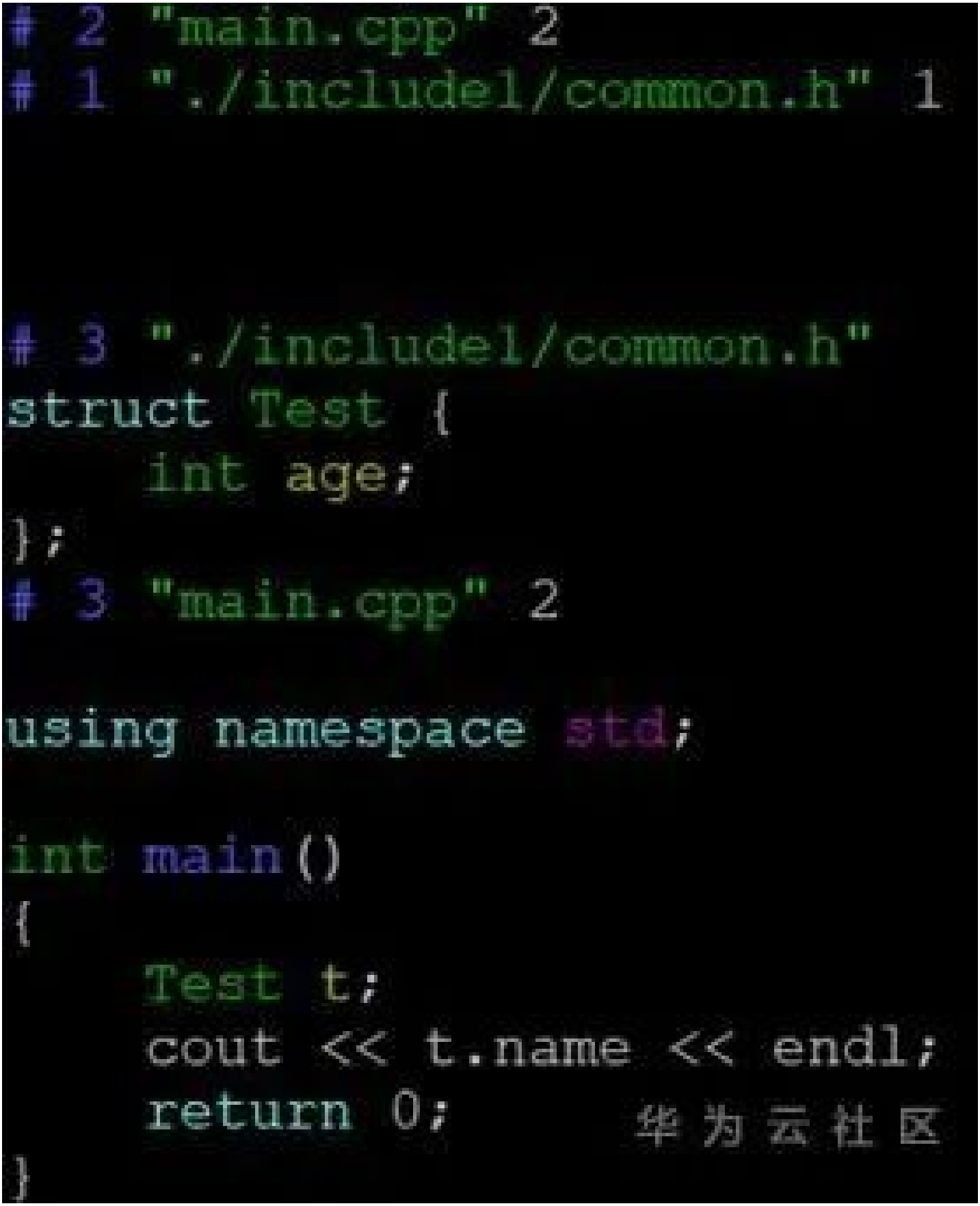
链接：g++ main.o。链接生成可执行程序，之所以需要链接是因为我们代码不可能像main.cpp这么简单，现代软件动则成百上千万行，如果写在一个main.cpp既不利于分工合作，也无法维护，因此通常是由一堆cpp文件组成，编译器分别编译每个cpp，这些cpp里会引用别的模块中的函数或全局变量，在编译单个cpp的时候是没法知道它们的准确地址，因此在编译结束后，需要链接器将各种还没有准确地址的符号（函数、变量等）设置为正确的值，这样组装在一起就可以形成一个完整的可执行程序。

问题一：头文件遮挡

在编译过程中最诡异的问题莫过于头文件遮挡，如下代码中main.cpp包含头文件common.h，真正想用的头文件是图中最右边那个包含name



成员的文件（所在目录为./include），但在编译过程中中间的common.h（所在目录为./include1）抢先被发现，导致编译器报错：Test结构没有name成员，对程序员来讲，自己明明定义了name成员，居然说没有name这个成员，如果第一次碰到这种情况可能会怀疑人生。应对这种诡异的问题，我们可以用-E参数看下编译器预处理后的输出，如下图。



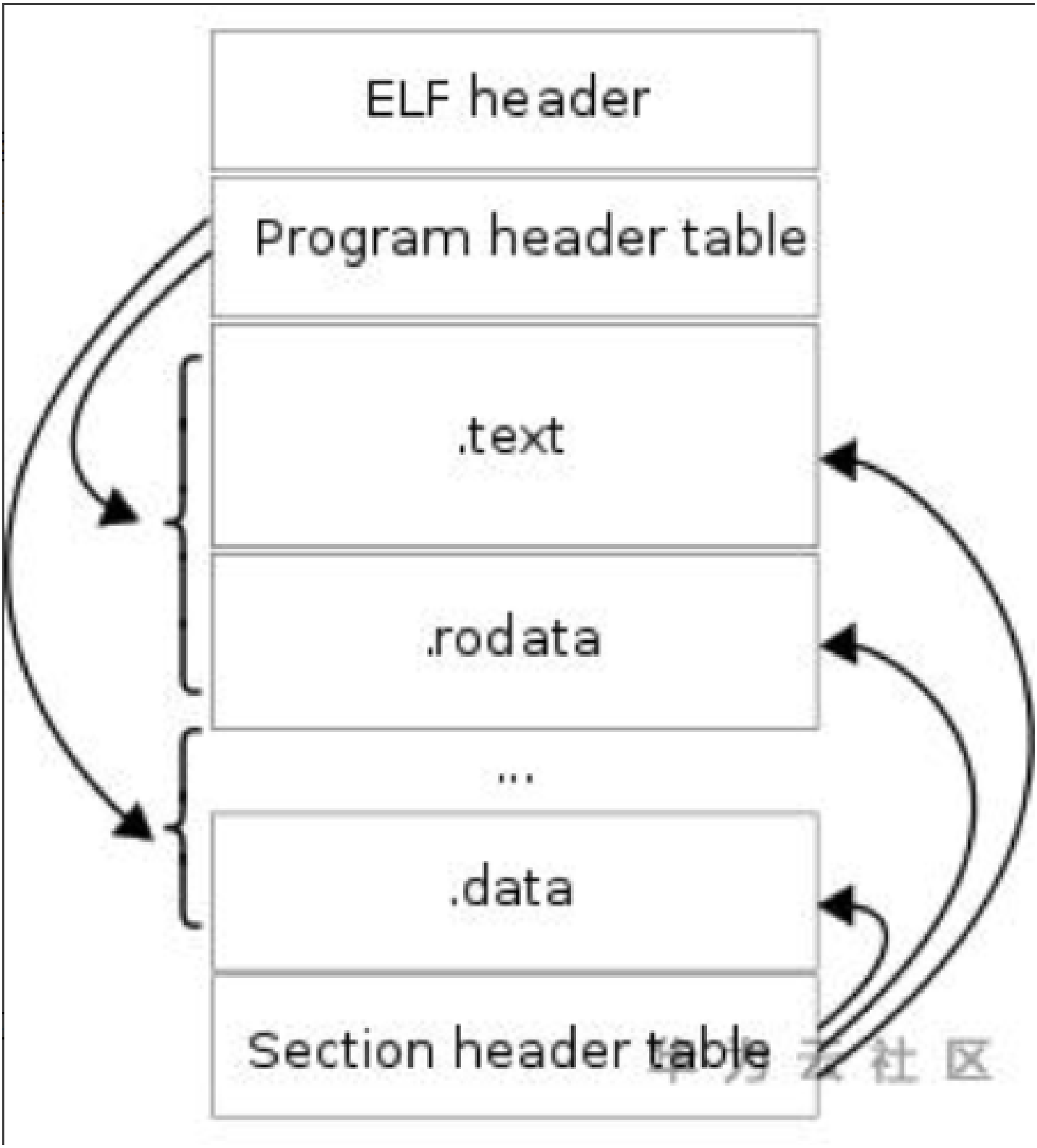
预处理文件格式如下：# linenum filename flag，表示之后的内容是从文件名为filaname的文件中第linenum行展开的，flag的取值可以是1,2,3,4，可以是用空格分开的多值，1表示接下来要展开一个新文件；2表示一个文件展开完毕；3表示接下来内容来自一个系统头文件；4表示接下来的内容应该看做是extern C形式引入的。

从展开后的输出我们可以清楚地看到Test结构确实没有定义name这个成员，并且Test这个结构是在./include1中的common.h中定义的，到此真相大白，编译器压根就没用我们定义的Test结构，而是被别的同名头文件截胡了。我们可以通过调整-I或者在头文件中带上部分路径更详细制定头文件位置来解决。

目标文件：

编译链接最终会生成各种目标文件，Linux下目标文件格式为ELF（Executable Linkable Format），详细定义见/usr/include/elf.h头文件，常见的目标文件有：可重定位目标文件，也即.o结尾的目标文件，当然静态库也归为此类；可执行文件，比如默认编译出的a.out文件；共享目标文件.so；核心转储文件，也就是core dump后产出的文件。Linux文件格式可以通过file命令查看。

一个典型的ELF文件格式如下图所示，文件有两种视角：编译视角，以section头部表为核心组织程序；运行视角，程序头部表以segment为核心组织程序。这么做主要是为了节约存储，很多细碎的section在运行时由于对齐要求会导致很大的内存浪费，运行时通常会将权限类似的section组织成segment一起加载。



通过命令objdump和readelf可以查看ELF文件的内容。

对可重定位目标文件常见的section有:

section名称	说明
.text	程序代码
.rodata	只读数据，比如字符串常量
.data	已初始化的全局和静态变量
.bss	未初始化的静态变量，包括初始化为0的全局和静态变量。未初始化的全局变量被编译器分配到COMMON块，但最终会放入bss中
.symtab	符号表，保存了程序中定义和引用的函数、全局变量信息。链接器就是通过符号的定义和引用关系来把相关模块粘连到一起
.rel.text	.text中位置列表，代码中对外部函数的引用需要链接器修改这些位置
.rel.data	类似text，只不过是针对全局变量

符号解析：

链接器会为对外部符号的引用修改为正确的被引用符号的地址，当无法为引用的外部符号找到对应的定义时，链接器会报undefined reference to XXXX的错误。另外一种情况是，找到了多个符号的定义，这种情况链接器有一套规则。在描述规则前需要了解强符号和弱符号的概念，简单讲函数和已初始化的全局变量是强符号，未初始化的全局变量是弱符号。

针对符号的多重定义链接器处理规则如下（作者在gcc 7.3.0上貌似规则2,3都按1处理）：

1. 不允许多个强符号定义，链接器会报告重复定义貌似的错误

2. 如果一个强符号和多个弱符号同名，则选择强符号

3. 如果符号在所有目标文件中都为弱符号，那么选择占用空间最大的一个

有了这些基础，我们先来看一下静态链接过程：

1. 链接器从左到右按照命令行出现顺序扫描目标文件和静态库

2. 链接器维护一个目标文件的集合E，一个未解析符号集合U，以及E中已定义的符号集合D，初始状态E、U、D都为空

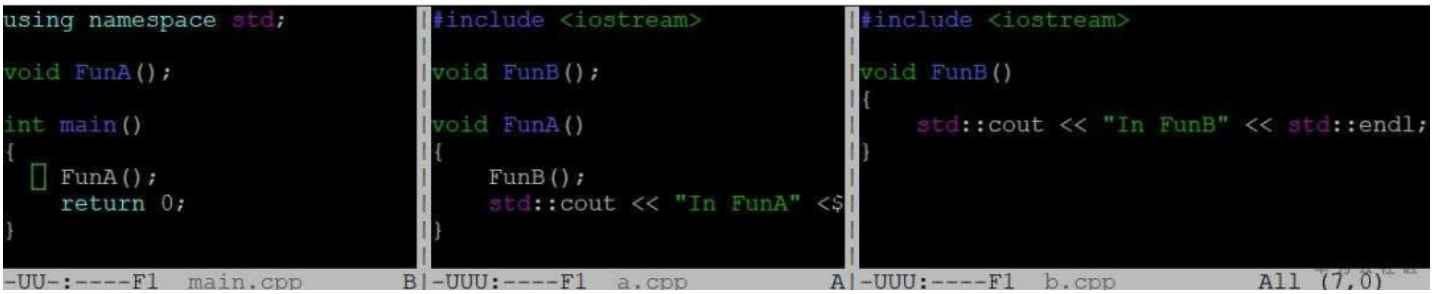
3. 对命令行上每个文件f，链接器会判断f是否是一个目标文件还是静态库，如果是目标文件，则f加入到E，f中未定义的符号加入到U中，已定义符号加入到D中，继续下一文件

4. 如果是静态库，链接器尝试到静态库目标文件中匹配U中未定义的符号，如果m中匹配U中的一个符号，那么m就和上步中文件f一样处理，对每个成员文件都依次处理，直到U、D无变化，不包含在E中的成员文件简单丢弃

5. 所有输入文件处理完后，如果U中还有符号，则出错，否则链接正常，输出可执行文件

问题二：静态库顺序

如下图所示，main.cpp依赖liba.a，liba.a又依赖libb.a，根据静态链接算法，如果用g++ main.cpp liba.a libb.a的顺序能正常链接，因为解析liba.a时未定义符号FunB会加入到上述算法的U中，然后在libb.a中找到定义，如果用g++ main.cpp libb.a liba.a的顺序编译，则无法找到FunB的定义，因为根据静态链接算法，在解析libb.a的时候U为空，所以不需要做任何解析，简单抛弃libb.a，但在解析liba.a的时候又发现FunB没有定义，导致U最终不为空，链接错误，因此在做静态链接时，需要特别注意库的顺序安排，引用别的库的静态库需要放在前面，碰到链接很多库的时候，可能需要做一些库的调整，从而使依赖关系更清晰。



动态链接：

之前大部分内容都是静态链接相关，但静态链接有很多不足：不利于更新，只要有一个库有变动，都需要重新编译；不利于共享，每个可执行程序都单独保留一份，对内存和磁盘是极大的浪费。

要生成动态链接库需要用到参数"-shared -fPIC"表示要生成位置无关PIC（Position Independent Code）的共享目标文件。对静态链接，在生成可执行目标文件时整个链接过程就完成了，但要想实现动态链接的效果，就需要把程序按照模块拆分成相对独立的部分，在程序运行时将他们链接成一个完整的程序，同时为了实现代码在不同程序间共享要保证代码是和位置无关的（因为共享目标文件在每个程序中被加载的虚拟地址都不一样，要保证它不管被加载在哪都能工作），而为了实现位置无关又依赖一个前提：数据段和代码段的距离总是保持不变。

由于不管在内存中如何加载一个目标模块，数据段和代码段间的距离是不变的，编译器在数据段前面引入了一个全局偏移表GOT（Global Offset Table），被引用的全局变量或者函数在GOT中都有一条记录，同时编译器为GOT中每个条目生成一个重定位记录，因为数据段是可以修改的，动态链接器在加载时会重定位GOT中的每个条目，这样就实现了PIC。

大体原理基本就这样，但具体实现时，对函数的处理和全局变量有所不同。由于大型程序函数成千上万，而程序很可能只会用到其中的一小部分，因此没必要加载的时候把所有的函数都做重定位，只有在用到的时候才对地址做修订，为此编译器引入了过程链接表PLT（Procedure Linkage Table）来实现延时绑定。PLT在代码段中，它指向了GOT中函数对应的地址，第一次调用时候，GOT存放的不是函数的实际地址，而是PLT跳转到GOT代码的后一条指令地址，这样第一次通过PLT跳转到GOT，然后通过GOT又调回到PLT的下一条指令，相当于什么也没做，紧接着PLT后面的代码会将动态链接需要的参数入栈，然后调用动态链接器修正GOT中的地址，从这以后，PLT中代码跳转到GOT的地址就是函数真正的地址，从而实现了所谓的延时绑定。

对共享目标文件而言，有几个需要关注的section：

section名称	说明
.interp	指明了动态链接器位置（objdump –s a.out|readelf –l a.out）
.dynamic	类似静态链接中section头部（readelf –d XX.so）
.dynsym	动态符号的导入导出关系（readelf –sD XX.so）
.rel.dyn、.rel.plt	全局变量和函数的重定位表（readelf –r XX.so）

有了以上基础后，我们看一下动态链接的过程：

1. 装载过程中程序执行会跳转到动态链接器

2. 动态链接器自举通过GOT、.dynamic信息完成自身的重定位工作

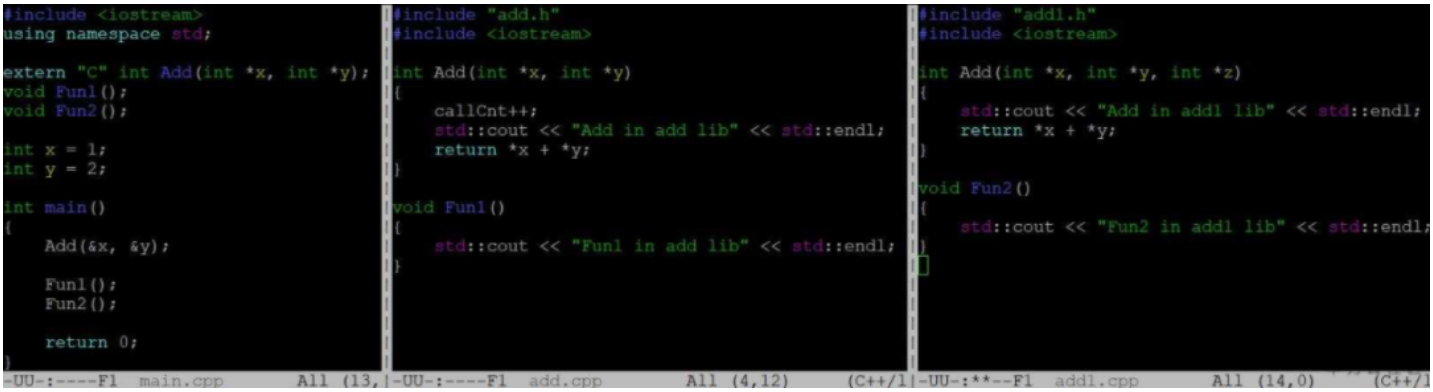
3. 装载共享目标文件：将可执行文件和链接器本身符号合并入全局符号表，依次广度优先遍历共享目标文件，它们的符号表会不断合并到全局符号表中，如果多个共享对象有相同的符号，则优先载入的共享目标文件会屏蔽掉后面的符号

4. 重定位和初始化

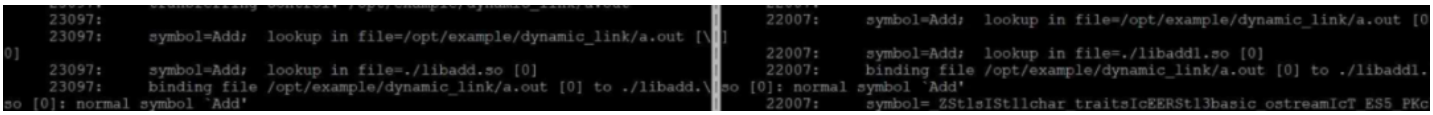
问题三：全局符号介入

动态链接过程中最关键的第3步可以看到，当多个共享目标文件中包含一个相同的符号，那么会导致先被加载的符号占住全局符号表，后续共享目标文件中相同符号被忽略。当我们代码中没有很好的处理命名的话，会导致非常奇怪的错误，幸运的话立刻core dump，不幸的话直到程序运行很久以后才莫名其妙的core dump，甚至永远不会core dump但是结果不正确。

如下图所示，main.cpp中会用到两个动态库libadd.so，libadd1.so的符号，我们把重点



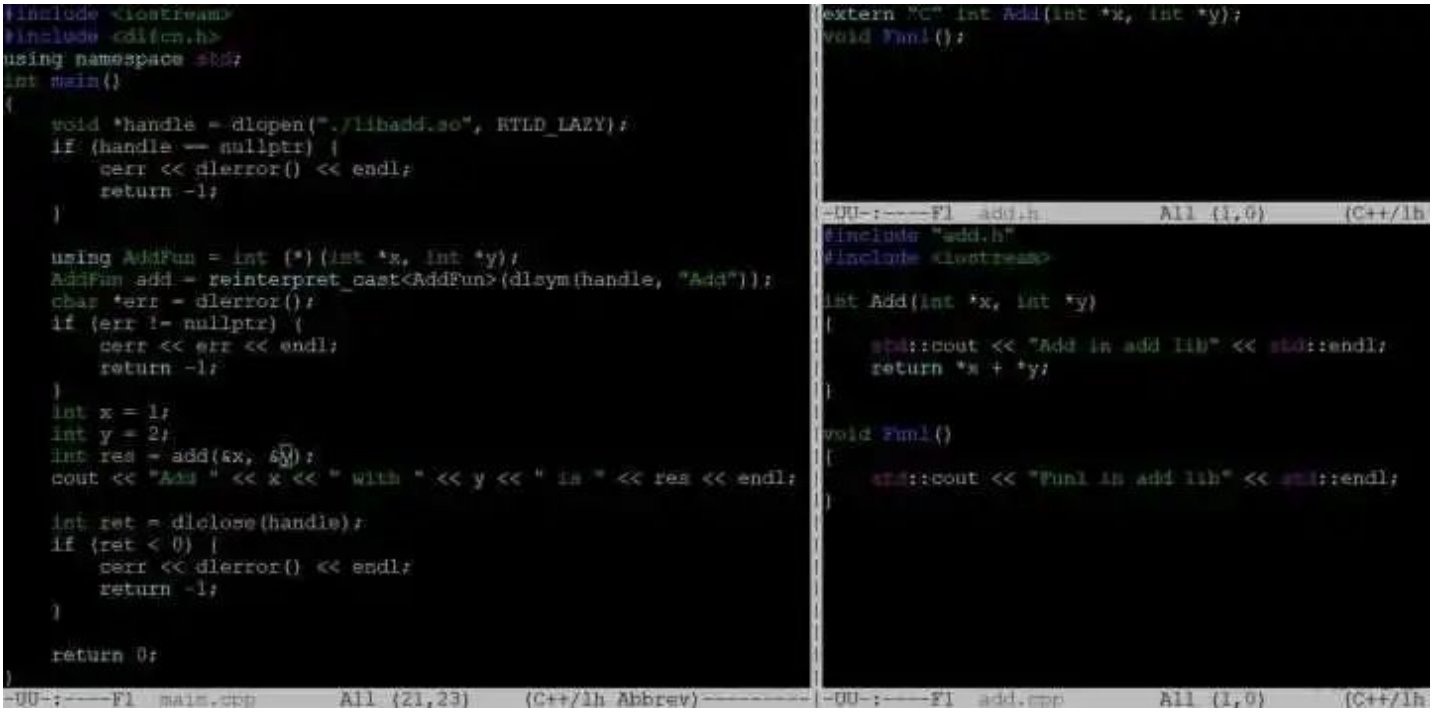
放在Add函数的处理上，当我们以g++ main.cpp libadd.so libadd1.so编译时，程序输出“Add in add lib”说明Add是用的libadd.so中的符号（add.cpp），当我们以g++ main.cpp libadd1.so libadd.so编译时，程序输出“Add in add1 lib”说明Add是用的libadd1.so中的符号，这时候问题就大了，调用方main.cpp中认为Add只有两个参数，而add1.cpp中认为Add有三个参数，程序中如果有这样的代码，可以预见很可能造成巨大的混乱。具体符号解析我们可以通过LD_DEBUG=all ./a.out来观察Add的解析过程，如下图所示：左边是对应libadd.so在编译时放在前面的情况，Add绑定在libadd.so中，右边对应libadd1.so放前面的情况，Add绑定在libadd1.so中。



运行时加载动态库：

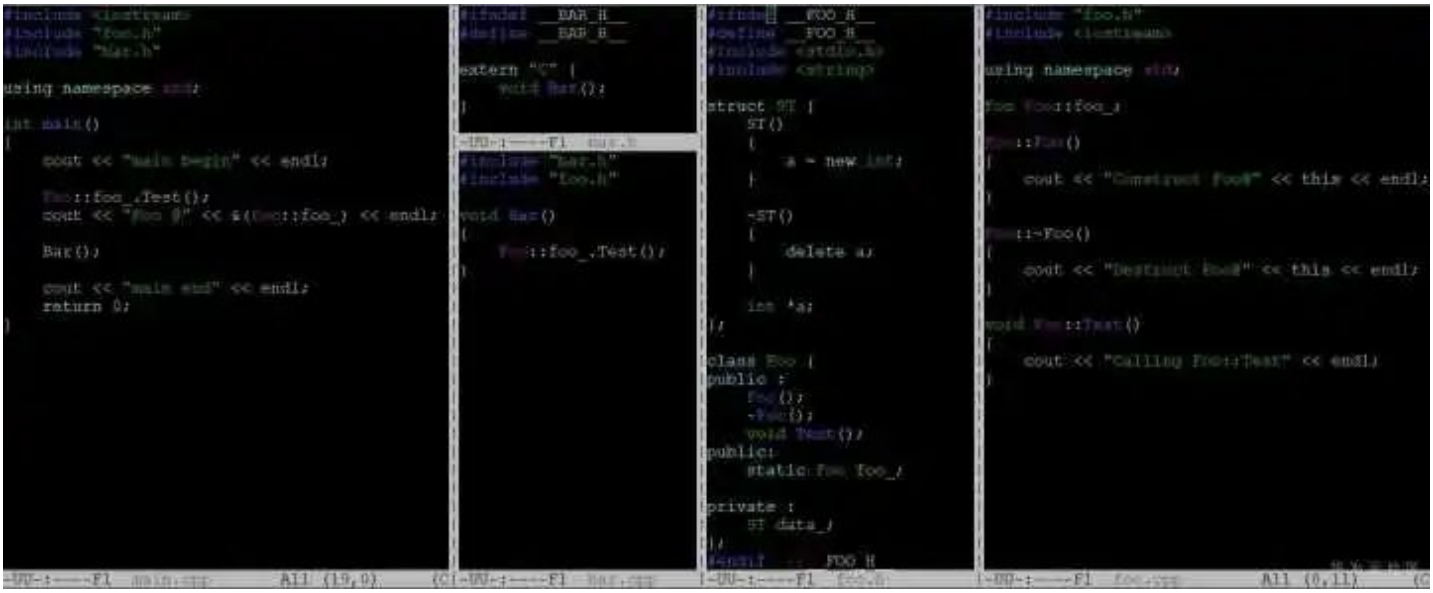
有了动态链接和共享目标文件的加持，Linux提供了一种更加灵活的模块加载方式：通过提供dlopen，dlsym，dlclose，dlerror几个API，可以实现在运行的时候动态加载模块，从而实现插件的功能。

如下代码演示了动态加载Add函数的过程，add.cpp按照正常编译“g++ -fPIC –shared –o libadd.so add.cpp”成libadd.so，main.cpp通过“g++ main.cpp -ldl”编译为a.out。main.cpp中首先通过dlopen接口取得一个句柄void *handle，然后通过dlsym从句柄中查找符号Add，找到后将其转化为Add函数，然后就可以按照正常的函数使用，最后dlclose关闭句柄，期间有任何错误可以通过dlerror来获取。



问题四：静态全局变量与动态库导致double free

在全面了解了动态链接相关知识后，我们来看一个静态全局变量和动态库纠结在一起引发的问题，代码如下，foo.cpp中有一个静态全局对象foo_，foo.cpp会编译成一个libfoo.a，bar.cpp依赖libfoo.a库，它本身会编译成libbar.so，main.cpp既依赖于libfoo.a又依赖libbar.so。



编译的makefile如下：

```
a.out: main.cpp libfoo.a libbar.so
	g++ -g -o a.out main.cpp -lfoo -lbar -L.
libfoo.a: foo.cpp
	g++ -c -fPIC -o foo.o foo.cpp
	ar crv libfoo.a foo.o
libbar.so: bar.cpp libfoo.a
	g++ -g -shared -fPIC -o libbar.so bar.cpp -lfoo -L.
clean:
	rm -f libfoo.a libbar.so a.out
	rm *.o
```

运行a.out会导致double free的错误。这是由于在一个位置上调用了两次析构函数造成的。之所以会这样是因为链接的时候先链接的静态库，将foo_的符号解析为静态库中的全局变量，当动态链接libbar.so时，由于全局已经有符号foo_，因此根据全局符号介入，动态库中对foo_的引用会指向静态库中版本，导致最后在同一个对象上析构了两次。

```
Construct Foo@0x6021b8
Construct Foo@0x6021b8
main begin
Calling Foo::Test
Foo @0x6021b8
Calling Foo::Test
main end
Destruct Foo@0x6021b8
Destruct Foo@0x6021b8
*** Error in `/opt/example/dynamic_link/double_free/a.out': double free or corruption (fasttop): 0x0000000001586c40 ***
======= Backtrace: =========
/lib64/libc.so.6(+0x81679)[0x7f3540a50679]
/opt/example/dynamic_link/double_free/a.out(_ZN2STD2Ev+0x20)[0x400f64]
/opt/example/dynamic_link/double_free/a.out(_ZN3FooD1Ev+0x55)[0x400e55]
/lib64/libc.so.6(__cxa_finalize+0x9a)[0x7f3540a0900a]
./libbar.so(+0xcd3)[0x7f3541637cd3]
```

解决办法如下：

1. 不使用全局对象

2. 编译时候调换库的顺序，动态库放在前面，这样全局只会有一个foo_对象

3. 全部使用动态库

4. 通过编译器参数来控制符号的可见性。

总结：