C++项目中替换内存分配实现方案

1. 需求和目标：

将C++项目中的内存分配和释放的接口调用链接到我们自己的内存池实现上，同时希望达到以下目标：

1. 对现有C++项目代码不用做任何修改；
2. 内存分配实现的替换只局限在指定项目的lib中，而不会影响到客户项目的lib。
3. 方案概述：

本方案通过替换默认的全局new/new[]和delete/delete[]的实现的方式，将C++项目中的内存分配和释放的接口调用链接到我们自己的内存池上实现上，方案实现大体分为三步：

1. 实现自定义的全局new/new[]和delete/delete[]，将源程序编译成静态库；
2. 将第1)步生成的静态库合并到现有C++项目代码生成的库（不同的平台库的形态不同）；
3. 使第2)步生成的合并库中的new/new[]和delete/delele[]符号对外不可见。
4. 方案详细描述：

方案概述里的第1)步属于C++标准中定义的，所以不区分平台，虽然各个平台生成静态库的方式各不相同，但是也是标准操作，这里也就不详细说明。方案概述里的第2)步和第3)步都要依赖链接器提供的功能，而且在各个平台上链接器提供的功能差异还是很大的，从而也导致了在不同平台上合并生成的库形态（静态库或动态库）并不相同，后续会分平台详细介绍。

* 1. 实现自定义的全局new/new[]和delete/delete[]：

假设我们的自定义实现放在了一个mynew.cpp里（当然也可以放在多个源文件中），mynew.cpp中大体要包含如下代码：

// ================================================

void \*operator new(std::size\_t size)

{

void \*ptr = my\_allocate\_impl(size);

if (ptr == NULL) {

throw std::bad\_alloc();

}

return ptr;

}

void \*operator new(std::size\_t size, const std::nothrow\_t& tag) throw()

{

void \*ptr = my\_allocate\_impl(size);

return ptr;

}

void \*operator new[](std::size\_t size)

{

…

}

void \*operator new[](std::size\_t size, const std::nothrow\_t& tag) throw()

{

…

}

void operator delete(void\* ptr)

{

my\_deallocate\_impl(ptr);

}

void operator delete(void\* ptr, const std::nothrow\_t& tag)

{

my\_deallocate\_impl(ptr);

}

void operator delete [](void\* ptr)

{

…

}

void operator delete [](void\* ptr, const std::nothrow\_t& tag)

{

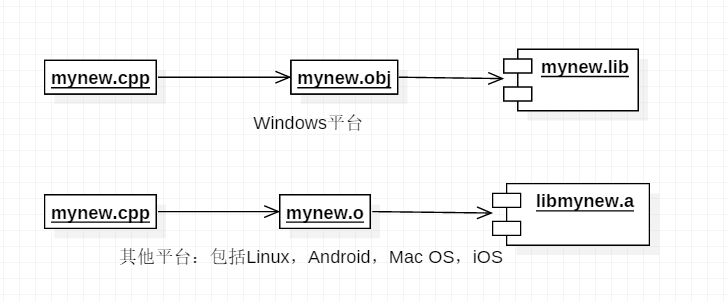
…

}

// ================================================

其中my\_allocate\_impl和my\_deallocate\_impl的定义就是我们自己的内存分配和释放实现。

在Windows平台我们将mynew.cpp编译成mynew.lib，在其他平台上我们将mynew.cpp编译成libmynew.a。

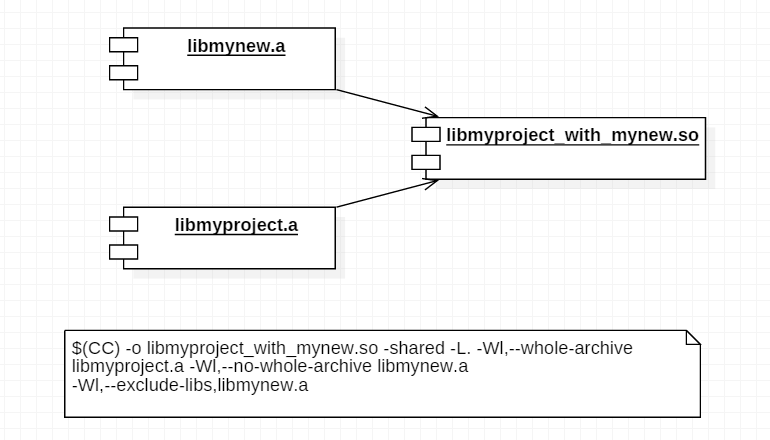


* 1. 编译链接自定义的new实现

假设我们现有的C++项目叫myproject，需要替换内存分配和释放的实现。

* + 1. Linux和Android平台

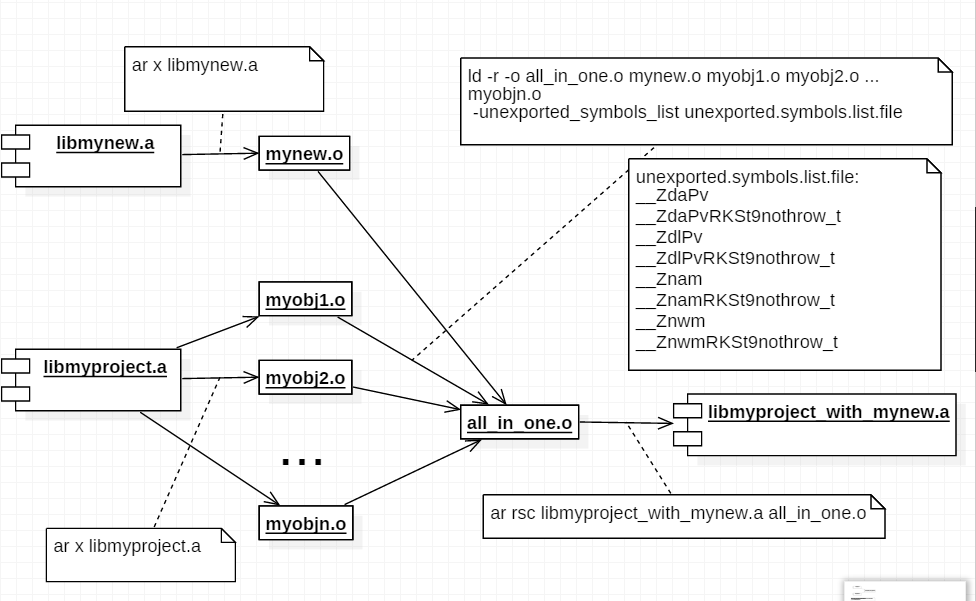
我们首先把myproject项目编译成静态库，我们叫它libmyproject.a，然后我们通过链接器工具，将libmyproject.a和libmynew.a一起打到一个动态库中，这里我们暂且叫libmyproject\_with\_mynew.so吧，那么我们希望libmyproject\_with\_mynew.so中的new和delete都会调用到mynew.cpp中的实现，并且libmyproject\_with\_mynew.so中的自定义的new和delete符号不会被导出，而libmyproject.a中需要导出的符号还是会在libmyproject\_with\_mynew.so中导出：这里需要用到如下的链接器选项--whole-archive、--no-whole-archive、--exclude-libs，如果是通过g++之类的编译器命令调用链接器工具，需要在flag前面加上-Wl。



如果myproject项目本身就是要生成动态库的，那么只需在生成动态库的命令后面加上-L. libmynew.a -Wl, --exclude-libs, libmynew.a就可以了。

* + 1. Mac OS和iOS平台

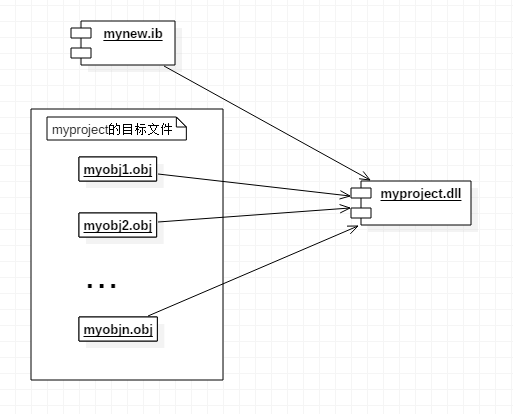
我们首先把myproject项目编译成静态库，我们叫它libmyproject.a，然后我们通过链接器工具，将libmyproject.a和libmynew.a一起打到另一个静态库中，这里我们暂且叫libmyproject\_with\_mynew.a吧，那么我们希望libmyproject\_with\_mynew.a中的new和delete都会调用到mynew.cpp中的实现，并且libmyproject\_with\_mynew.a中的自定义的new和delete符号对外不可见：具体步骤需要用到ar x把两个.a文件都反解成.o文件，然后通过ld -r将众多.o文件再打成一个巨大的.o文件，这里叫all\_in\_one.o吧，ld -r的命令行中要增加-unexported\_symbols\_list选项（或者unexported\_symbol）以指定new和delete符号对外不可见，最后再把all\_in\_one.o打成libmyproject\_with\_mynew.a（其实你也可以自己打成动态库）。



* + 1. Windows平台

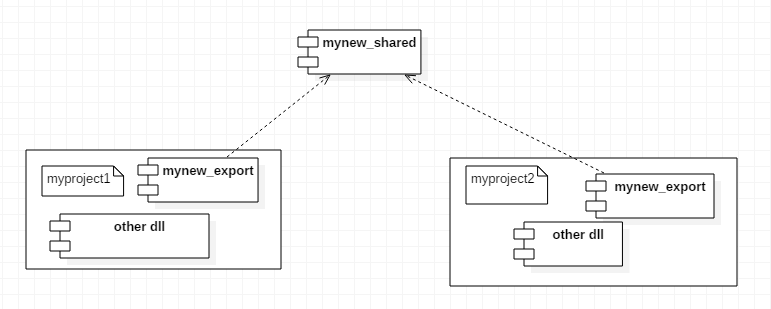
由于Windows平台上，Visual Studio中<new>里声明的new和delete本身就是动态库符号不导出的，所以Windows上的方案相对简单，也不需要什么额外处理。

我们首先把myproject项目编译成动态库，我们叫它myproject.dll，然后在Visual Studio中将mynew.lib设置为linker的input的additional dependencies ，基本就可以了。



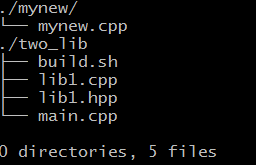
1. 多项目扩展

对于一个大项目（完整的app）中，如果依赖于多个子项目（库），而其中有至少2个子项目（库）希望采用相同的自定义内存分配方案，且不想在每个子项目中都包含各自独立的内存分配方案的实现实例（该方案默认的效果），可以考虑把mynew.cpp拆分成两部分：mynew\_export和mynew\_shared。其中mynew\_export里只是包含全局new/new[]和delete/delete[]的定义，而my\_allocate\_impl和my\_deallocate\_impl的定义挪到mynew\_shared里。



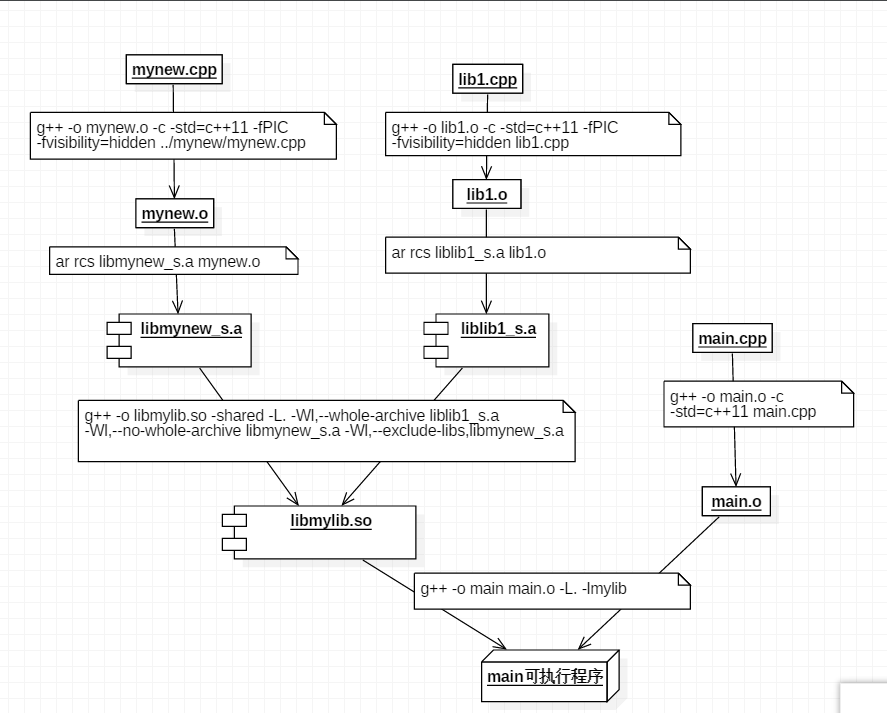
1. 实际范例
   1. Linux平台(Android平台同样适用)：

解压linux\_demo.tar.gz，项目结构如下：



其中mynew.cpp重定义了global的new和delete运算符，lib1.cpp中定义了一个导出符号的函数foo，函数foo里面会有内存申请和释放的操作；另外main.cpp包含main函数，并且会调用lib1::foo，而且main函数本身也会有内存申请和释放的操作。

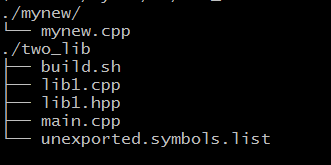
我们下面将mynew.cpp和lib1.cpp打包到一个动态库libmylib.so中，将main.cpp编译成main.o，并且将main.o和libmylib.so链接成叫main的可执行程序。具体命令如下：



运行main可执行程序，可以观察到结果，lib1::foo的内存申请和释放使用的是mynew.cpp中的实现，而main.cpp中本身的内存申请和释放使用的是libstdc++的。

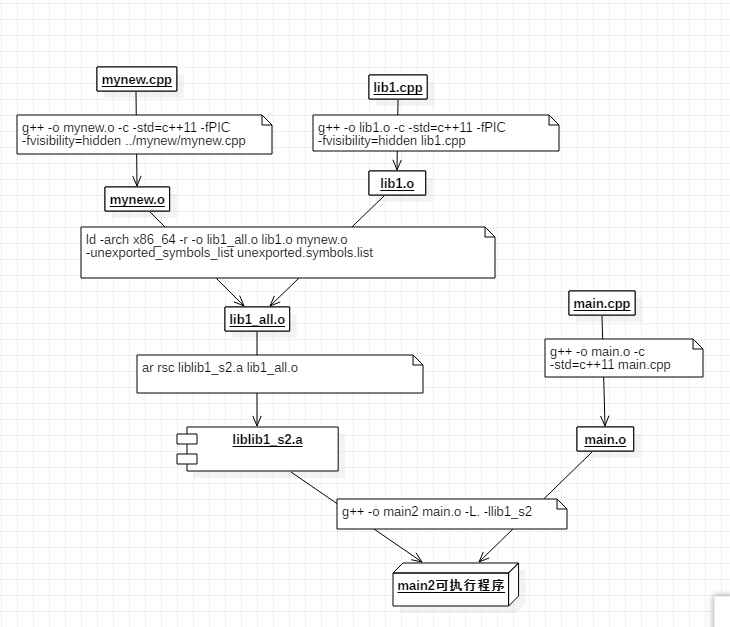
* 1. Mac OS平台（iOS平台同样适用）

解压macos\_demo.tar.gz，项目结构如下：



其中mynew.cpp重定义了global的new和delete运算符，lib1.cpp中定义了一个导出符号的函数foo，函数foo里面会有内存申请和释放的操作；另外main.cpp包含main函数，并且会调用lib1::foo，而且main函数本身也会有内存申请和释放的操作。

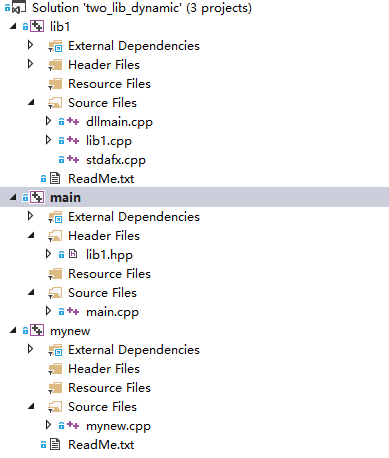
我们下面将mynew.cpp和lib1.cpp打包到一个静态库liblib1\_s2.a中，将main.cpp编译成main.o，并且将main.o和liblib1\_s2.a链接成叫main2的可执行程序。具体命令如下：



对比于linux上的方案，最大的区别是，我们首先把mynew.o和lib1.o合并成一个大的.o文件，然后再把.o转成.a，而且是在合并过程中通过ld的-unexported\_symbols\_list选项将new和delete的符号设置为不导出的。

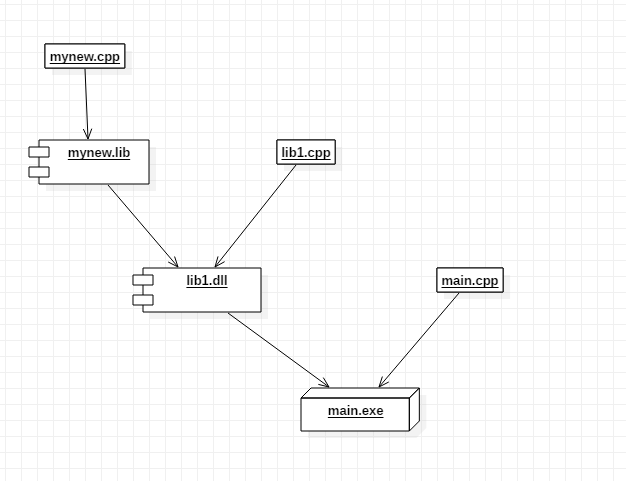
* 1. Windows平台

解压windows\_demo1.tar.gz，用visual studio打开solution，项目结构如下：



其中mynew.cpp重定义了global的new和delete运算符，lib1.cpp中定义了一个导出符号的函数foo，函数foo里面会有内存申请和释放的操作；另外main.cpp包含main函数，并且会调用lib1::foo，而且main函数本身也会有内存申请和释放的操作。

我们下面将mynew.cpp放到一个叫mynew的工程中，生成一个静态库mynew.lib；把lib1.cpp放到一个叫lib1的工程中，生成一个动态库lib1.dll和lib1.lib，并且指定lib1的linker->Input属性为链接mynew.lib；将main.cpp放到一个叫main的工程中，生成可执行程序main.exe，并且指定main的inker->Input属性为链接lib1.lib。示意图如下：

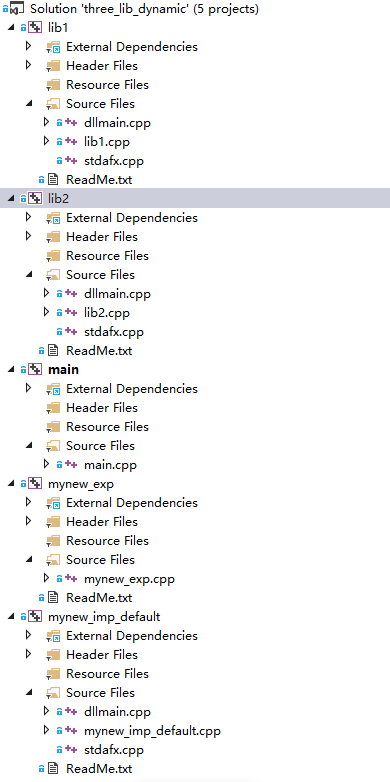


对比于Linux和Mac OS平台，由于VS编译器中new和delete的符号类型本身就是不导出的，所以几乎不需要什么特别需要指定的，就是按照正常项目构建方式就可以，不过后面在多项目扩展的例子中，我们会发现，在VS编译器上还是需要做些额外工作的。

1. 多项目扩展范例

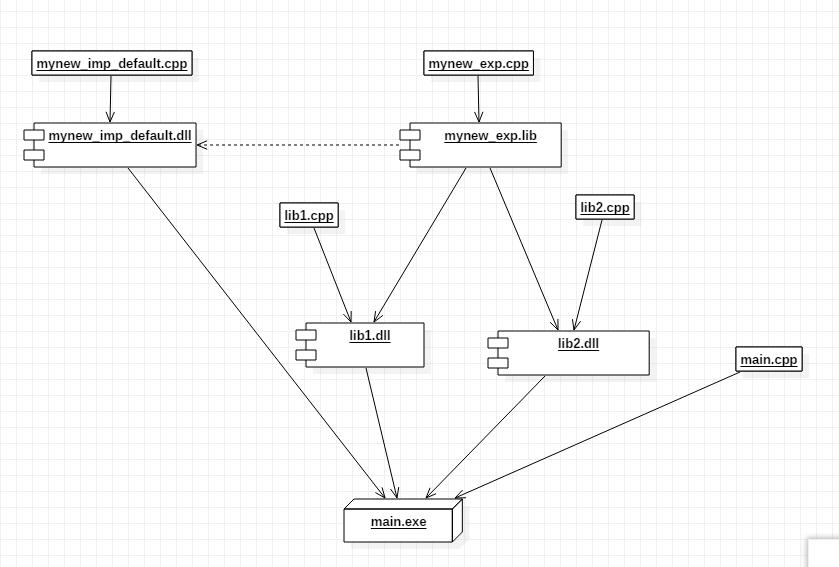
多项目扩展我们仅举一个windows上的例子。

解压windows\_demo2.tar.gz，用visual studio打开solution，项目结构如下：



其中mynew\_exp.cpp重定义了global的new和delete运算符，但不提供内存申请和释放的具体实现，而是转发调用my\_operator\_new\_implement和my\_operator\_delete\_implement；mynew\_imp\_default.cpp中提供了my\_operator\_new\_implement和my\_operator\_delete\_implement的实现；lib1.cpp中定义了一个导出符号的函数foo，函数foo里面会有内存申请和释放的操作；lib2.cpp中定义了一个导出符号的函数bar，函数bar里面会有内存申请和释放的操作；另外main.cpp包含main函数，并且会调用lib1::foo和lib2::bar，而且main函数本身也会有内存申请和释放的操作。

我们下面将mynew\_imp\_default.cpp放到一个叫mynew\_imp\_default的工程中，生成一个动态库mynew\_imp\_default.dll和mynew\_imp\_default.lib；将mynew\_exp.cpp放到一个叫mynew\_exp的工程中，生成一个静态库mynew\_exp.lib；把lib1.cpp放到一个叫lib1的工程中，生成一个动态库lib1.dll和lib1.lib，并且指定lib1的linker->Input属性为链接mynew\_exp.lib和mynew\_imp\_default.lib；把lib2.cpp放到一个叫lib2的工程中，生成一个动态库lib2.dll和lib2.lib，并且指定lib2的linker->Input属性为链接mynew\_exp.lib和mynew\_imp\_default.lib；将main.cpp放到一个叫main的工程中，生成可执行程序main.exe，并且指定main的inker->Input属性为链接lib1.lib和lib2.lib。示意图如下：



这样，lib1.dll和lib2.dll就公用了一份mynew\_imp\_default的实现。