一、前言
二、示例代码说明
1. 功能描述
2. 文件结构
三、Linux 系统下操作
1. 通过 cmake 指令，生成 Makefile 文件
2. 编译 libA
3. 编译 libB
4. 编译可执行程序 appC
四、Windows 系统下操作
1. 生成 VS 解决方案
2. 编译 libA
3. 编译 libB
4. 编译 appC
五、总结

一、前言

在上一篇文章中(使用 cmake 来搭建跨平台的应用程序框架：C语言版本)，我们以源代码的形式，演示了利用利用 cmake 这个构建工具，来编译跨平台的动态库、静态库和应用程序。

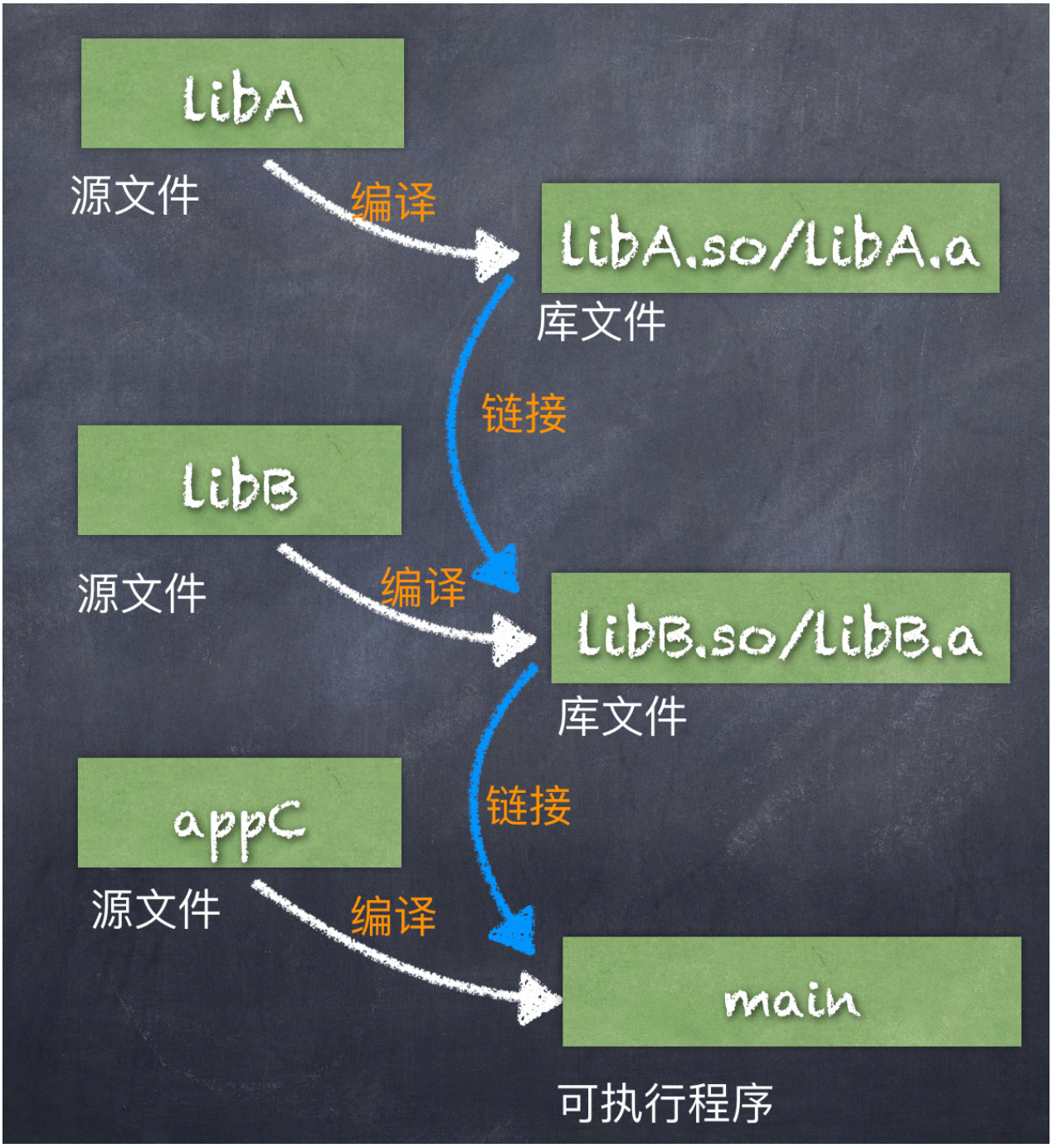
这篇文章描述的是同样的功能，只不过是用 C++ 来编码，另外，增加了一个小功能：如果在导出的库文件中，使用另一个第三方库。

在公众号后台留言【506】，可以收到示例代码。在 Linux/Windows 系统中可以直接编译、执行，拿来即用。

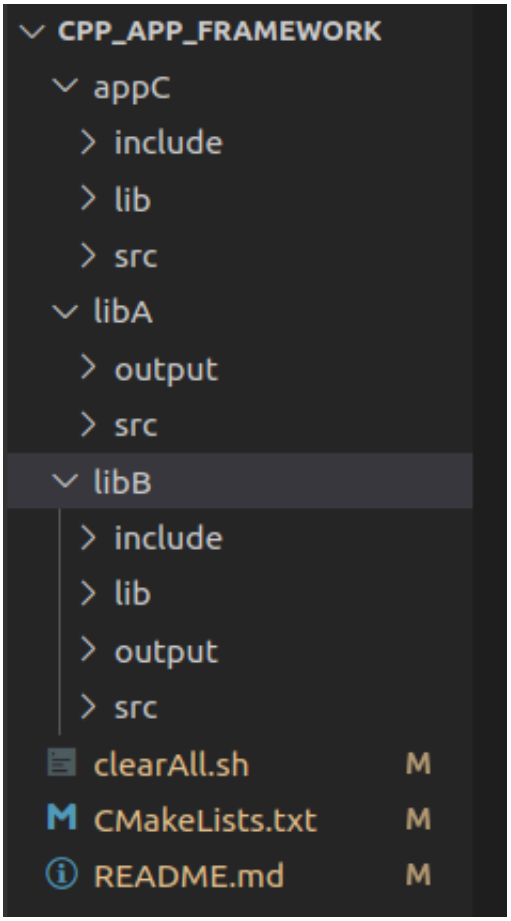
二、示例代码说明

1. 功能描述

示例代码的主要目的，是用来描述如何组织一个跨平台的应用程序结构。它的功能比较简单，如下图所示：


libA
源文件
编译
libA.so/libA.a
库文件
链接
libB
源文件
编译
libB.so/libB.a
库文件
appC
链接
源文件
编译
main
可执行程序

2. 文件结构



1. libA: 编译得到库文件 libA.so/libA.a;
2. libB: 编译得到库文件 libB.so/libB.a，它需要调用 libA 库中的函数;
3. appC：应用程序，它需要调用 libB 库中的函数；

三、Linux 系统下操作

1. 通过 cmake 指令，生成 Makefile 文件

为了不污染源代码，我们新建一个 build 目录，然后在其中编译：

```
$ make build
$ cd build
$ cmake ..
```

编译输出结果：

```
-- The C compiler identification is GNU 5.4.0
-- The CXX compiler identification is GNU 5.4.0
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Check for working C compiler: /usr/bin/cc - skipped
-- Detecting C compile features
-- Detecting C compile features - done
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Check for working CXX compiler: /usr/bin/c++ - skipped
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- Configuring done
-- Generating done
```

2. 编译 libA

```
$ cd libA/src
$ make
```

编译结果如下：

```
Scanning dependencies of target liba_shared
[ 25%] Building CXX object libA/src/CMakeFiles/liba_shared.dir/A.cpp.o
[ 50%] Linking CXX shared library libA.so
[ 50%] Built target liba_shared
Scanning dependencies of target liba_static
[ 75%] Building CXX object libA/src/CMakeFiles/liba_static.dir/A.cpp.o
[100%] Linking CXX static library libA.a
[100%] Built target liba_static
```

安装到源码下的 output 目录:

```
$ make install
```

```
/cpp_app_framework/libA/output/include/A.h
/cpp_app_framework/libA/output/include/ADll.h
/cpp_app_framework/libA/output/lib/linux/libA.so
/cpp_app_framework/libA/output/lib/linux/libA.a
```

此时，相关文件被安装到源码路径下 libA 的 output 目录下:

3. 编译 libB

由于 libB 调用了 libA 中的函数，因此需要手动把相关头文件和库文件复制到 libB 目录下，当然，这个步骤也可以写在 CMakeLists.txt 中。

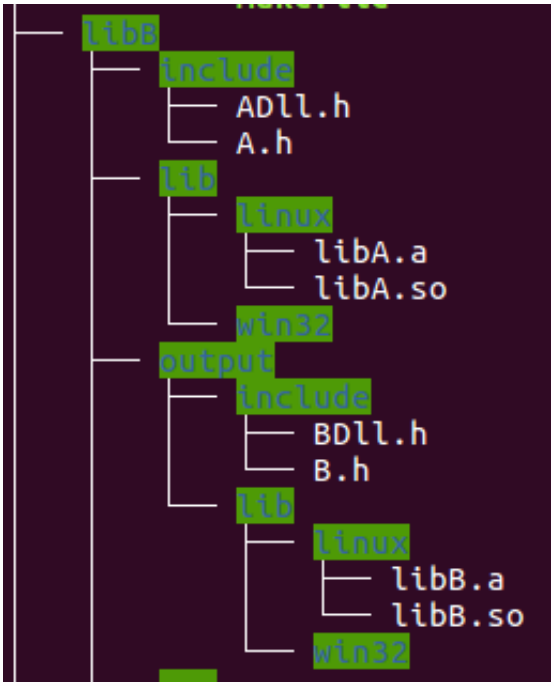
然后进入 build/libB/src 目录，执行编译指令：

```
$ cd build/libB/src
$ make
```

```
[ 25%] Linking CXX shared library libB.so
[ 50%] Built target libb_shared
Scanning dependencies of target libb_static
[ 75%] Building CXX object libB/src/CMakeFiles/libb_static.dir/B.cpp.o
[100%] Linking CXX static library libB.a
[100%] Built target libb_static
```

同样的，把 libB 生成的库文件和头文件，复制到源码中的 libB/output 目录下：

```
framework/libB/output/include/B.h
framework/libB/output/include/BDll.h
framework/libB/output/lib/linux/libB.so
/cpp_app_framework/libB/output/lib/linux/libB.so" to ""
framework/libB/output/lib/linux/libB.a
```



4. 编译可执行程序 appC

由于 appC 调用了 libB 中的函数，因此需要手动把相关头文件和库文件复制到 appC 目录下的 include 和 lib/linux 目录下。

此外，由于我一直使用动态库，所以还需要把 libA 的头文件和库文件也复制到 appC 目录下。

然后进入 build/appC/src 目录，执行编译指令：

```
$ cd build/appC/src
$ make
```

```
[ 50%] Linking CXX executable main
[100%] Built target main
```

执行输出结果：

```
A ctor
B ctor
result = 6

B dtor
A dtor
```

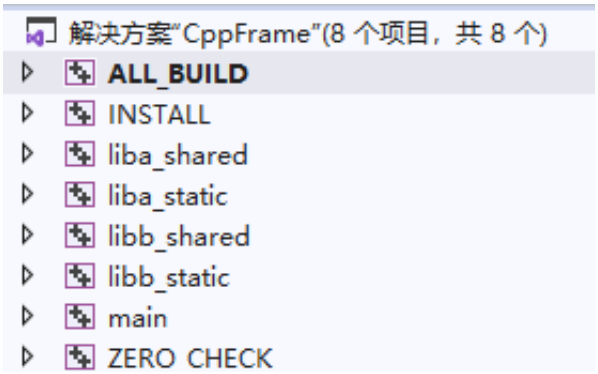
四、Windows 系统下操作

1. 生成 VS 解决方案

在 build 目录下执行 `cmake ..`，得到 VS 解决方案：

```
-- Building for: Visual Studio 16 2019
-- Selecting Windows SDK version 10.0.19041.0 to target Windows 10.0.18363.
-- The C compiler identification is MSVC 19.28.29337.0
-- The CXX compiler identification is MSVC 19.28.29337.0
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Check for working C compiler: C:/Program Files (x86)/Microsoft Visual Studio/
2019/Professional/VC/Tools/MSVC/14.28.29333/bin/Hostx64/x64/cl.exe - skipped
-- Detecting C compile features
-- Detecting C compile features - done
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Check for working CXX compiler: C:/Program Files (x86)/Microsoft Visual Studi
o/2019/Professional/VC/Tools/MSVC/14.28.29333/bin/Hostx64/x64/cl.exe - skipped
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- Configuring done
-- Generating done
-- Build files have been written to: I:/share/cpp_app_framework/build
```

打开工程文件 CppFrame.sln，右侧的解决方案如图：

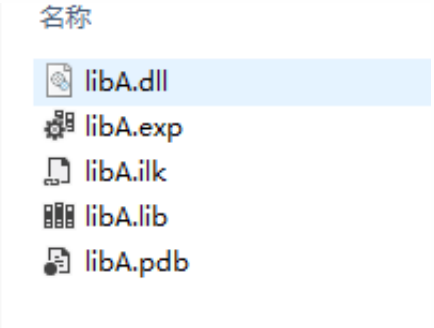


2. 编译 libA

在 libA_shared 上单击右键，选择【生成】：

```
已启动重新生成…
1>——— 已启动全部重新生成: 项目: ZERO_CHECK, 配置: Debug x64 ———
1>Checking Build System
1>CMake is re-running because I:/share/cpp_app_framework/build/libA/src/CMakeFiles/generate.stamp is out-of-date.
1>  the file 'I:/share/cpp_app_framework/libA/src/CMakeLists.txt'
1>  is newer than 'I:/share/cpp_app_framework/build/libA/src/CMakeFiles/generate.stamp.depend'
1>  result='-1'
1>— Selecting Windows SDK version 10.0.19041.0 to target Windows 10.0.18363.
1>— Configuring done
1>— Generating done
1>— Build files have been written to: I:/share/cpp_app_framework/build
2>——— 已启动全部重新生成: 项目: liba_shared, 配置: Debug x64 ———
2>Building Custom Rule I:/share/cpp_app_framework/libA/src/CMakeLists.txt
2>cl : 命令行 warning D9002: 忽略未知选项“-std=c++11”
2>A.cpp
2>  正在创建库 I:/share/cpp_app_framework/build/libA/src/Debug/libA.lib 和对象 I:/share/cpp_app_framework/build/libA/src/Debug/libA.exp
2>liba_shared.vcxproj -> I:\share\cpp_app_framework\build\libA\src\Debug\libA.dll
2>已完成生成项目“liba_shared.vcxproj”的操作。
========== 全部重新生成: 成功 2 个，失败 0 个，跳过 0 个 ==========
```

可以看到，在 build\libA\src\Debug 目录下看到编译生成的文件：



这里有一个问题需要注意一下：在 libA/src/CMakeLists.txt 中，如果编译动态库，请如下设置：

```
4    # 编译动态库
5    add_definitions("-DLIBA_API_EXPORTS")
6
7    # 编译静态库
8    # add_definitions("-DLIBA_STATIC")
```

如果编译静态库，请如下设置：

```
4    # 编译动态库
5    # add_definitions("-DLIBA_API_EXPORTS")
6
7    # 编译静态库
8    add_definitions("-DLIBA_STATIC")
```

这几个宏定义，需要结合 ADll.h 中的定义来理解，主要是解决 Windows 平台下的动态库的导出与导入问题。

在下面编译 libB 库的时候，也需要同样的操作。

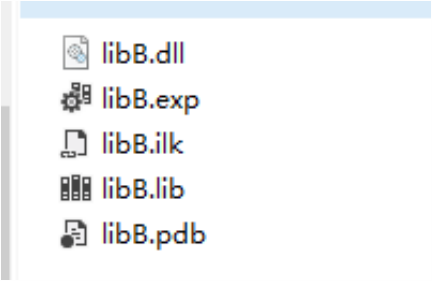
3. 编译 libB

由于 libB 调用了 libA 中的函数，因此，需要手动把 libA 库相关的头文件和库文件复制到 libB 目录下。

在 libB_shared 目标上，单击【生成】，编译输出如下：

```
1>——— 已启动生成: 项目: ZERO_CHECK, 配置: Debug x64 ———
1>Checking Build System
1>CMake is re-running because I:/share/cpp_app_framework/build/libA/src/CMakeFiles/generate.stamp is out-of-date.
1>  the file 'I:/share/cpp_app_framework/libA/src/CMakeLists.txt'
1>  is newer than 'I:/share/cpp_app_framework/build/libA/src/CMakeFiles/generate.stamp.depend'
1>  result='-1'
1>— Selecting Windows SDK version 10.0.19041.0 to target Windows 10.0.18363.
1>— Configuring done
1>— Generating done
1>— Build files have been written to: I:/share/cpp_app_framework/build
2>——— 已启动生成: 项目: libb_shared, 配置: Debug x64 ———
2>Building Custom Rule I:/share/cpp_app_framework/libB/src/CMakeLists.txt
2>cl : 命令行 warning D9002: 忽略未知选项“-std=c++11”
2>B.cpp
2>I:\share\cpp_app_framework\libB\src\B.cpp(1,1): warning C4819: 该文件包含不能在当前代码页(936)中表示的字符。请将该文件保存为 Unicode 格式以防止数据丢失
2>I:\share\cpp_app_framework\libB\src\B.h(1,1): warning C4819: 该文件包含不能在当前代码页(936)中表示的字符。请将该文件保存为 Unicode 格式以防止数据丢失
2>  正在创建库 I:/share/cpp_app_framework/build/libB/src/Debug/libB.lib 和对象 I:/share/cpp_app_framework/build/libB/src/Debug/libB.exp
2>libb_shared.vcxproj -> I:\share\cpp_app_framework\build\libB\src\Debug\libB.dll
2>已完成生成项目“libb_shared.vcxproj”的操作。
========== 生成: 成功 2 个，失败 0 个，最新 0 个，跳过 0 个 ==========
```

此时，在 build/libB/src/Debug 目录下，看到生成的库文件：



我们把 libB 库文件和头文件，手动复制到 appC 目录下备用。

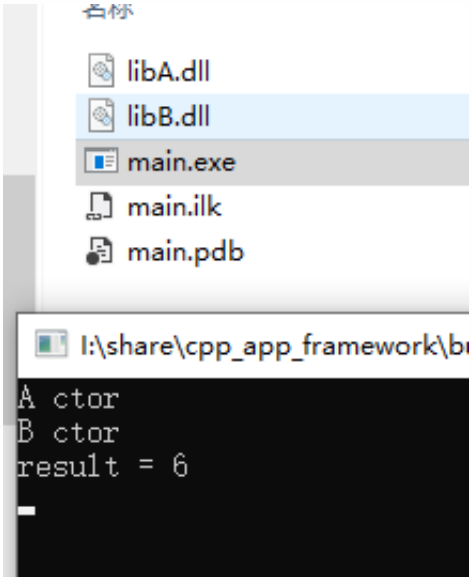
4. 编译 appC

在 VS 的 main 目标上，单击【生成】，编译输出如下：

```
已启动生成…
1>——— 已启动生成: 项目: main, 配置: Debug x64 ———
1>Building Custom Rule I:/share/cpp_app_framework/appC/src/CMakeLists.txt
1>cl : 命令行 warning D9002: 忽略未知选项“-std=c++11”
1>main.cpp
1>I:\share\cpp_app_framework\appC\include\B.h(1,1): warning C4819: 该文件包含不能在当前代
1>main.vcxproj -> I:\share\cpp_app_framework\build\appC\src\Debug\main.exe
1>已完成生成项目“main.vcxproj”的操作。
========== 生成: 成功 1 个，失败 0 个，最新 1 个，跳过 0 个 ==========
```

此时，在 build\appC\src\Debug 目录下即可看到可执行程序 main.exe。

为了执行这个程序，还需要把 libA.dll, libB.dll 复制到当前目录下才可以，如下所示：



五、总结

这篇文章的操作过程主要以动态库为主，如果编译、使用静态库，执行过程是一样一样的。

如果操作过程有什么问题，欢迎留言、讨论，谢谢！

在公众号后台留言【506】，可以收到示例代码。在 Linux/Windows 系统中可以直接编译、执行，拿来即用。

祝您好运！

---------- End ----------

让知识流动起来，越分享，越幸运！

星标公众号，能更快找到我！

Hi~你好，我是道哥，一枚嵌入式开发老兵。

推荐阅读

【1】C语言指针-从底层原理到花式技巧，用图文和代码帮你讲解透彻
【2】C指针的这些使用技巧，掌握后立刻提升一个Level
【3】提高代码逼格的利器：宏定义-从入门到放弃
【4】原来gdb的底层调试原理这么简单
【5】一步步分析-如何用C实现面向对象编程
【6】我最喜欢的进程之间通信方式-消息总线
【7】如何利用Google的protobuf，来思考、设计、实现自己的RPC框架
【8】都说软件架构要分层、分模块，具体应该怎么做(一)

【9】都说软件架构要分层、分模块，具体应该怎么做(二)
【10】内联汇编很可怕吗？看完这篇文章，终结它！