

第四章 编译与打包的常见方法

在项目开发完成后，需要发布并安装部署到客户的主机上。因为不同的客户使用的 Mac/Linux 的版本不同，各个版本本身所安装的库又各不相同，所以导致了运行环境的千差万别，对客户环境的支持与排错的成本很高，并严重影响客户的体验。

常见的打包方法，一般主要是动态编译与静态编译。

一 动态库

不同的操作系统，动态库的文件格式是有所区别，在 Windows 系统中较为常见的是 dll 或者 ocx 等，在 Linux 系统中一般是 so 或者是 so.1，后面的 1 或者数字一般是该动态库的版本号，而在 Mac 中则是 dylib 后缀。

查看 Linux 系统动态库：

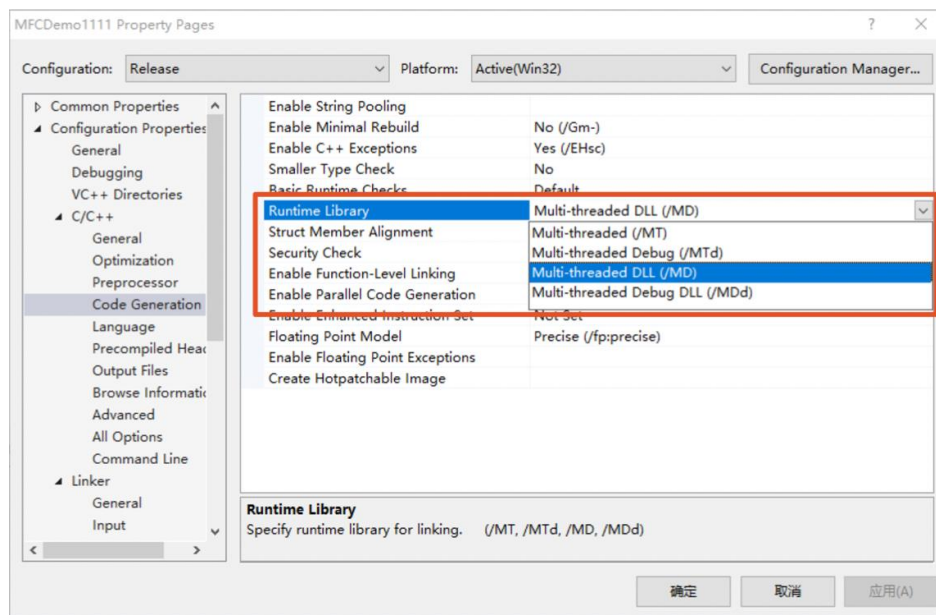
```
ibao:/opt/tools/cmake-3.21.3-linux-x86_64/bin$ ldd cmake
linux-vdso.so.1 => (0x00007ffffe858e000)
libdl.so.2 => /lib/x86_64-linux-gnu/libdl.so.2 (0x00007fd4dc92b000)
librt.so.1 => /lib/x86_64-linux-gnu/librt.so.1 (0x00007fd4dc723000)
libpthread.so.0 => /lib/x86_64-linux-gnu/libpthread.so.0 (0x00007fd4dc506000)
libm.so.6 => /lib/x86_64-linux-gnu/libm.so.6 (0x00007fd4dc1fd000)
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007fd4dbe33000)
/lib64/ld-linux-x86-64.so.2 (0x00007fd4dcb2f000)
```

查看 Mac 系统的动态库：

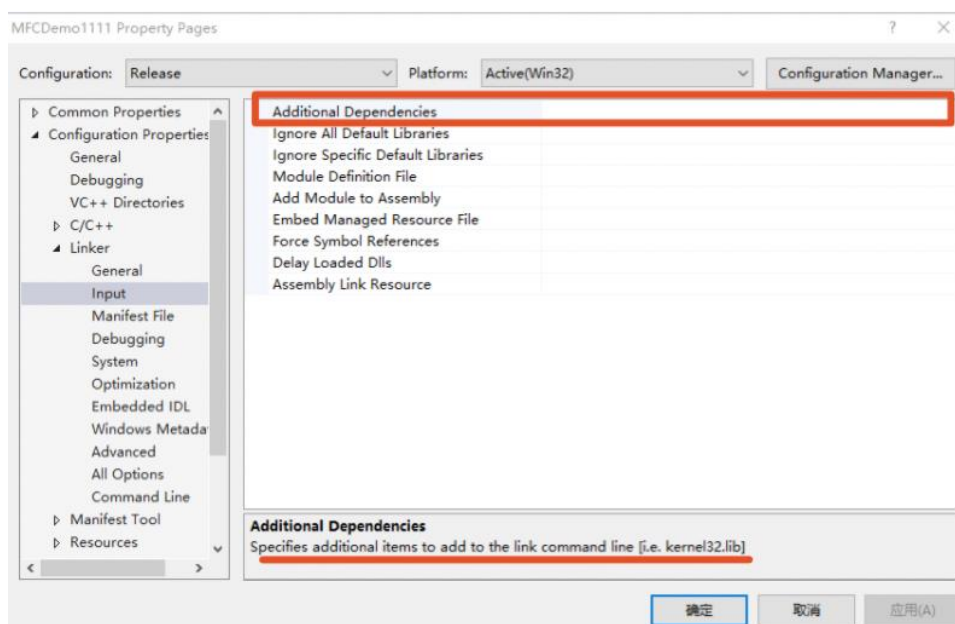
```
➤ MacOS otool -L /usr/local/bin/cmake
/usr/local/bin/cmake:
    /usr/lib/libz.1.dylib (compatibility version 1.0.0, current version 1.2.11)
    /usr/lib/libbz2.1.0.dylib (compatibility version 1.0.0, current version 1.0.5)
    /usr/lib/libcurl.4.dylib (compatibility version 7.0.0, current version 9.0.0)
    /System/Library/Frameworks/CoreFoundation.framework/Versions/A/CoreFoundation (compatibility versio
    /System/Library/Frameworks/CoreServices.framework/Versions/A/CoreServices (compatibility version 1.
    /usr/lib/libc++.1.dylib (compatibility version 1.0.0, current version 902.1.0)
    /usr/lib/libSystem.B.dylib (compatibility version 1.0.0, current version 1281.100.1)
```

二 静态库

在 Windows 平台编译的静态库一般是.lib 格式，在编译该静态库的时候需要选择 MT，如果是 Debug 模式，则需要选择 MTd。



在其他想中想引用该静态库时，需要设置在 Linker 的时候导入 lib，这样再引入静态库的头文件的，就能找到相应的函数。



在 Linux 或者 Mac 系统 (包括 Android、iOS)，静态库的文件后缀一般是.a 文件。

查看 Linux 系统的静态库：

```
dev@ubuntu: /usr/lib/x86_64-linux-gnu
dev@ubuntu: /usr/lib/x86_64-linux-gnu$ file libutil.a
libutil.a: current ar archive
```

查看 Mac 系统的静态库：

```
➔ openssl-OpenSSL_1_1_11 file libssl.a
libssl.a: current ar archive random library
```

在项目中可以增加静态文件.a、.lib 的链接，最终可以将静态文件链接到可执行文件中。但是这样会使可执行文件增加一定的大小，不过却可以在编译可执行文件后不再依赖库。

在 Qt 中导入静态库方法 `LIBS += /opt/path/libssl.a`

三 动态库-静态加载

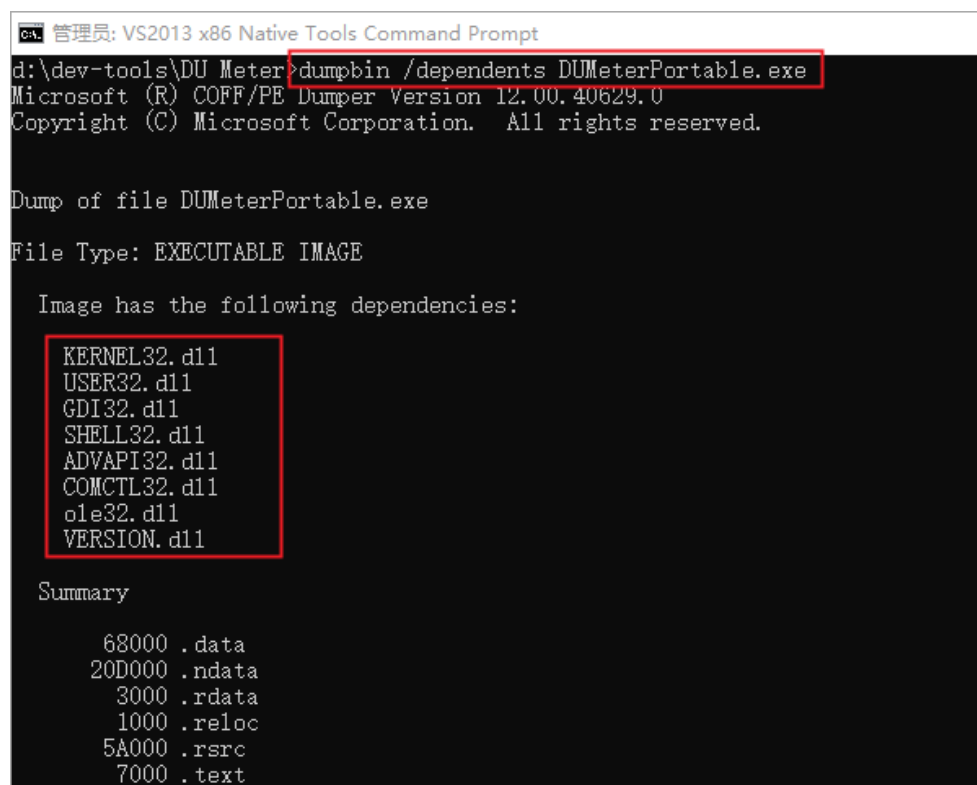
在 Windows 平台编译的动态库一般是有多个文件：.dll、.lib，在编译该静态库的时候需要选择 MD，如果是 Debug 模式，则需要选择 MDd。

在 Windows 平台如果要静态加载动态库则直接导入动态库的.lib 文件，同样需要将动态库的 dll 打包，并在运行的时候直接加载对应的 dll 文件，如果缺失，则会启动报错。在 windows 平台一般可以借助 dumpbin（VS 开发工具自带的）工具进行分析依赖。在 Linux 平台可以使用 ldd 命令

windows 系统：`dumpbin /dependents` 可执行文件名或者 dll

Linux 系统：`ldd` 可执行文件名或者 so

Mac 系统：`otool -L` 可执行文件名或者 dylib



```
管理员: VS2013 x86 Native Tools Command Prompt
d:\dev-tools\DU Meter>dumpbin /dependents DUMeterPortable.exe
Microsoft (R) COFF/PE Dumper Version 12.00.40629.0
Copyright (C) Microsoft Corporation. All rights reserved.

Dump of file DUMeterPortable.exe

File Type: EXECUTABLE IMAGE

Image has the following dependencies:

KERNEL32.dll
USER32.dll
GDI32.dll
SHELL32.dll
ADVAPI32.dll
COMCTL32.dll
ole32.dll
VERSION.dll

Summary

68000 .data
20D000 .ndata
3000 .rdata
1000 .reloc
5A000 .rsrc
7000 .text
```

在 Linux 平台，使用终端启动可执行文件如果缺失则同样会提示错误，无法启动，而另外一种情况是某些动态库不在系统目录找到同样也无法启动。

四 动态库-动态加载

如果是动态加载，则一般是在启动之后，调用系统函数 `LoadLibrary` 加载 `dll`，或者使用 `qt` 的 `QLibrary` 来进行加载，属于业务层或者新增的模块，加载失败则可以通过逻辑层判断，不会导致可执行程序启动失败，至少用户层代码是可以控制。

这时候使用 `dumpbin` 或者 `ldd` 之类的工具是无法直接看出问题的。但是对代码的编写稍微比静态加载要复杂。

五 最终目标

最终期望的是希望在 `qt` 项目编译的时候，不依赖第三方动态库，只保留对操作系统层的动态库依赖，换句话说：尽可能的对动态库的依赖，确保在大部分系统上都可以直接启动我们的可执行文件，选择使用静态编译，将第三方的 `xcb`、`font` 相关等的库设置为静态库，最终编译到可执行程序内部。