**POCO C++库学习和分析 --  跨平台库的生成**

     这一节主要讲一下，如何在window上生成一个类似于POCO动态库，以及POCO跨平台头文件的结构。

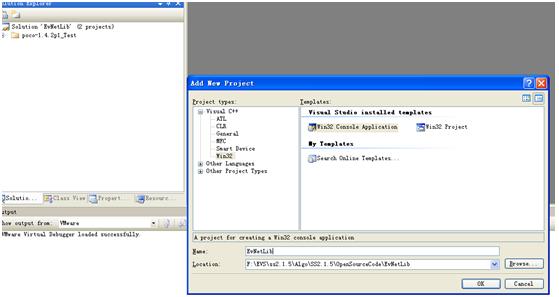
     POCO C++ Basic Edition版本，工程由四部分联合而成，分别是Foundation，Net，Util，Xml。

     其中每个部分都是跨平台结构，VC下编译采用没有预编译头文件的形式，为了做到跨平台，每个工程都有自己的输出定义文件，分别是“Foundation.h”,“Net.h”,“Util.h”和“Xml.h”，各工程的任何头文件基本上都会包括其对应的头文件，来控制输出。

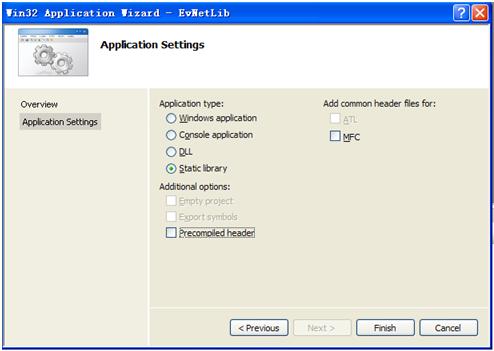
**1． 跨平台库的生成**

     在Poco库中所有的Dll都是没有资源、没有stdafx.h 、stdafx.cpp没有dllmain存在的工程。如何生成这样的工程和理解生成这样的工程Vs所做的工作都有一定意义。

     a)      建立win32 console Application



     b)      选取静态库，不勾选”预编译头文件”

****

     c)      在”工程”->”Properties”->”General”->”Configuration Type”中选择”DynamicLibrary（.dll）”。来控制生成dll还是lib

     d)      在添加文件后，自然会有”C/C++”选项;在修改成为”DLL”类型后，自然会有” Link”选项

     e)      如果是DLL,在”工程”->”Properties”->”C/C++”->”Preprocessor”->”PreprocessorDefinitions”中,把”\_LIB”定义换成” \_USRDLL”

**2． 跨平台的头文件定义**

     这个层次应该先了解一下编译之前系统或编译器已经为我们提供了什么,

     编译器的预编译宏。其中分类如下:

     ⑴   编译器制定平台类型, 如” \_WIN32”、” \_WIN64”、” \_\_FreeBSD\_\_”、” unix”。

     ⑵   编译器制定的平台硬件类型，如”\_\_x86\_64\_\_”、” \_\_IA64\_\_”、”\_\_arm\_\_”。

     ⑶   某个编译器下编译类型的定义，如”\_DLL” 、“\_DEBUG” 、 ” \_MSC\_VER” 、” \_WINDLL” 、” \_USRDLL” 、” \_AFXDLL”等。

     ⑷   通用的编译宏定义,如"\_\_DATE\_\_"、"\_\_FILE\_\_"、"\_\_LINE\_\_"

     因此我们的头文件定义应在了解编译器提供的预处理宏后，定义。一般会包括了以下几个部分：

     a)      编译定义宏，用来定义需要编译成什么样的版本。这些宏并不控制编译的平台，而只是指编译版本中一些特性，如” Poco/Config.h”中定义了线程栈大小，线程池最大线程数，是否支持UTF8等特性。

     b)      编译版本的输出类型，如编译的Dll的输出属性.如”Poco/foundation.h” 中”Foundation\_API”定义.

     c)      支持平台定义，操作系统与硬件结构关系定义。如”Poco/ Platform.h”，定义了支持的操作系统和平台的关系。

     d)      特定平台的下平台特性定义。如” Poco/ Platform\_WIN32.h”，定义了windows版本号等。这层的定义等同于使用MS Visial Studio默认生成的stdAfx.h中的定义.

     e)      一些辅助信息，如” Poco/Bugcheck.h”中的一些调试帮助

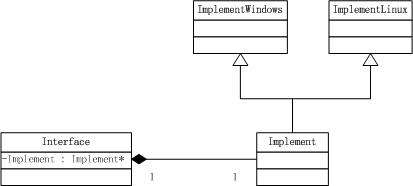
     f)       基本数据结构的统一，包括不同平台int、long等数据结构的定义。如”Poco/types.h”

     头文件的引入，在定义了所有的上述信息后，打包成一个文件，在Lib或DLL中的.h和.cpp文件全部引用到上述申明。如在Poco库中所有的文件都包含了”Poco/foundation.h”，而”Poco/foundation.h”包含了” Poco/Config.h”、 ”Poco/ Platform.h”等一系列头文件。

**3． 跨平台的类的实现**

     不同平台和操作系统的差异，主要体现在数据结构长度和底层操作系统API的使用上。其中第一个问题”数据结构长度“的差异可以通过定义统一数据结构来解决，POCO库也一样。在对待第二个问题，底层API的差异上，POCO库主要使用了三种方法。

**3.1 对于复杂的结构，通常抽象一个接口类和一个实现类，接口类提供统一接口，实现类按照操作系统分别定义实现，在程序中通过宏定义控制**。如下图。

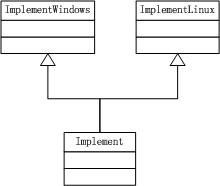


    代码结构上则为:

头文件: "interface.h"、”Implement.h“、"ImplementWindow.h"、”ImplementLinux.h“

CPP文件: "interface.cpp"、”Impement.cpp“、"ImplementWindow.cpp"、”ImplementLinux.cpp“

**3.2 对于简单一点的结构，则直接从接口继承。如下图：**



    代码结构上则为:

     头文件: ”Impement.h“、"ImplementWindow.h"、”ImplementLinux.h“

     CPP文件: ”Impement.cpp“、"ImplementWindow.cpp"、”ImplementLinux.cpp“

    对于3.1和3.2来说，可以在”Implement.h“文件中可以看到如下定义：

**[cpp]** [view plaincopy](http://blog.csdn.net/arau_sh/article/details/8568674)

1. **if** defined(POCO\_OS\_FAMILY\_WINDOWS)
2. #if defined(\_WIN32\_WCE)
3. #include "Poco/XXX\_WINCE.h"
4. #else
5. #include "Poco/XXX\_WIN32.h"
6. #endif
7. #elif defined(POCO\_VXWORKS)
8. #include "Poco/XXX\_VX.h"
9. #else
10. #include "Poco/XXX\_POSIX.h"
11. #endif

     对于3.1和3.2的写法，我挺赞成的，这样写代码可读性很好，深合”代码即文档“的原则。

**3.3 再简单一点，操作系统区别之接在同一个文件中区分。**

    代码结构上则为:

     头文件: ”Impement.h“

     CPP文件: ”Impement.cpp“

**[cpp]** [view plaincopy](http://blog.csdn.net/arau_sh/article/details/8568674)

1. **if** defined(POCO\_OS\_FAMILY\_WINDOWS)
2. #**if** defined(\_WIN32\_WCE)
3. **int** nRst = xxx\_wce\_api(...);
4. #**else**
5. **int** nRst = xxx\_windows\_api(...);
6. #endif #elif defined(POCO\_VXWORKS)
7. **int** nRst = xxx\_vxworks\_api(...);
8. #**else**
9. **int** nRst = xxx\_unix\_api(...);
10. #endif

（版权所有，转载时请注明作者和出处 <http://blog.csdn.net/arau_sh/article/details/8568674>）