5. 在 Windows 上构建 C 和 C++ 扩展

这一章简要介绍了如何使用 Microsoft Visual C++ 创建 Python 的 Windows 扩展模块，然后再提供有关其工作机理的详细背景信息。 这些说明材料同时适用于 Windows 程序员学习构建 Python 扩展以及 Unix 程序员学习如何生成在 Unix 和 Windows 上均能成功构建的软件。

鼓励模块作者使用 distutils 方式来构建扩展模块，而不使用本节所描述的方式。 你仍将需要使用 C 编译器来构建 Python；通常为 Microsoft Visual C++。

备注： 这一章提及了多个包括已编码 Python 版本号的文件名。 这些文件名以显示为 `XY` 的版本号来代表；在实践中，`'X'` 将为你所使用的 Python 发布版的主版本号而 `'Y'` 将为次版本号。 例如，如果你所使用的是 Python 2.2.1，`XY` 将为 `22`。

5.1. 菜谱式说明

与在 Unix 上一样，在 Windows 上构造扩展模块也有两种方式：使用 `setuptools` 包来控制构建过程，或者全手动操作。 setuptools 方式适用于大多数扩展；使用 `setuptools` 构建和打包扩展模块的文档见 使用 setuptools 构建 C 和 C++ 扩展。 如果你发现你真的需要手动操作，那么研究一下 winsound 标准库模块的项目文件可能会很有帮助。standard library module.

5.2. Unix 和 Windows 之间的差异

Unix 和 Windows 对于代码的运行时加载使用了完全不同的范式。 在你尝试构建可动态加载的模块之前，要先了解你所用系统是如何工作的。

在 Unix 中，一个共享对象 (`.so`) 文件中包含将由程序来使用的代码，也包含在程序中可被找到的函数名称和数据。 当文件被合并到程序中时，对在文件代码中这些函数和数据的全部引用都会被改为指向程序中函数和数据在内存中所放置的实际位置。 这基本上是一个链接操作。

在 Windows 中，一个动态链接库 (`.dll`) 文件中没有悬挂的引用。 而是通过一个查找表执行对函数或数据的访问。 因此在运行时 DLL 代码不必在运行时进行修改；相反地，代码已经使用了 DLL 的查找表，并且在运行时查找表会被修改以指向特定的函数和数据。

在 Unix 中，只存在一种库文件 (`.a`)，它包含来自多个对象文件 (`.o`) 的代码。 在创建共享对象文件 (`.so`) 的链接阶段，链接器可能会发现它不知道某个标识符是在哪里定义的。 链接器将在各个库的对象文件中查找它；如果找到了它，链接器将会包括来自该对象文件的所有代码。

在 Windows 中，存在两种库类型，静态库和导入库 (扩展名都是 `.lib`)。 静态库类似于 Unix 的 `.a` 文件；它包含在必要时可被包括的代码。 导入库基本上仅用于让链接器能确保特定标识符是合法的，并且将在 DLL 被加载时出现于程序中。 这样链接器可使用来自导入库的信息构建查找表以便使用未包括在 DLL 中的标识符。 当一个应用程序或 DLL 被链接时，可能会生成一个导入库，它将需要被用于应用程序或 DLL 中未来所有依赖于这些符号的 DLL。

假设你正在编译两个动态加载模块 B 和 C，它们应当共享另一个代码块 A。 在 Unix 上，你 *不应* 将 `A.a` 传给链接器作为 `B.so` 和 `C.so`；那会导致它被包括两次，这样 B 和 C 将分别拥有它们自己的副本。 在 Windows 上，编译 `A.dll` 将同时编译 `A.lib`。 你 *应当* 将 `A.lib` 传给链接器用于 B 和 C。 `A.lib` 并不包含代码；它只包含将在运行时被用于访问 A 的代码的信息。

在 Windows 上，使用导入库有点像是使用 `import spam`；它让你可以访问 spam 中的名称，但并不会创建一个单独副本。 在 Unix 上，链接到一个库更像是 `from spam import *`；它会创建一个单独副本。

Py_NO_LINK_LIB

关闭在CPython头文件中执行的基于``#pragma``的与Python库的隐式链接。

> *Added in version 3.14.*

5.3. DLL 的实际使用

Windows Python 是在 Microsoft Visual C++ 中构建的；使用其他编译器可能会也可能不会工作。本节的其余部分是针对 MSVC++ 的。

在Windows中创建DLL时，你可以通过两种方式使用CPython库：

1. 默认情况下，直接包含 `PC/pyconfig.h` 或通过 `Python.h` 会触发与库的隐式、配置感知型的链接。 头文件选择 `pythonXY_d.lib` 用于调试，`pythonXY.lib` 用于发布，以及 `pythonX.lib` 用于启用了 受限 API 的发布。

   要构建两个 DLL，spam 和 ni (使用 spam 中找到的 C 函数)，你应当使用以下命令:

   ```
   cl /LD /I/python/include spam.c
   cl /LD /I/python/include ni.c spam.lib
   ```

   第一条命令创建了三个文件: `spam.obj`, `spam.dll` 和 `spam.lib`。 `Spam.dll` 不包含任何 Python 函数 (如 `PyArg_ParseTuple()`)，但因为有隐式链接的 `pythonXY.lib` 所以它知道如何找到 Python 代码。

   第二条命令创建了 `ni.dll` (以及 `.obj` 和 `.lib`)，它知道如何从 spam 以及 Python 可执行文件中找到所需的函数。

2. 在包含 `Python.h` 之前，手动定义 `Py_NO_LINK_LIB` 宏。 必须将 `pythonXY.lib` 传递给链接器。

   要构建两个 DLL，spam 和 ni (使用 spam 中找到的 C 函数)，你应当使用以下命令:

   ```
   cl /LD /DPy_NO_LINK_LIB /I/python/include spam.c ../libs/pythonXY.lib
   cl /LD /DPy_NO_LINK_LIB /I/python/include ni.c spam.lib ../libs/pythonXY.lib
   ```

   第一条命令创建了三个文件: `spam.obj`, `spam.dll` 和 `spam.lib`。 `Spam.dll` 不包含任何 Python 函数 (例如 `PyArg_ParseTuple()`)，但它通过 `pythonXY.lib` 可以知道如何找到所需的 Python 代码。

第二条命令创建了 `ni.dll` (以及 `.obj` 和 `.lib`)，它知道如何从 spam 以及 Python 可执行文件中找到所需的函数。

不是每个标识符都会被导出到查找表。 如果你想要任何其他模块（包括 Python）都能看到你的标识符，你必须写上 `_declspec(dllexport)`，就如在 `void _declspec(dllexport) initspam(void)` 或 `PyObject _declspec(dllexport) *NiGetSpamData(void)` 中一样。

Developer Studio 会添加很多你并不真正需要的导入库，命名你的可执行文件大小增加约 100K。 要摆脱它们，请使用项目设置对话框中的链接选项卡指定 *忽略默认库*。 将正确的 `msvcrtxx.lib` 添加到库列表中。