1.概论

先来阐述一下 DLL(Dynamic Linkable Library) 的概念，你可以简单的把 DLL 看成一种仓库，它提供给你一些可以直接拿来用的变量、函数或类。在仓库的发展史上经历了 ¡°无库－静态链接库－动态链接库¡±的时代。

静态链接库与动态链接库都是共享代码的方式， 如果采用静态链接库， 则无论你愿不愿意， lib 中的指令都被直接包含在最终生成的 EXE 文件中了。但是若使用 DLL，该 DLL 不必被包含在最终 EXE 文件中，EXE 文件执行时可以 ¡°动态¡±地引用和卸载这个与 EXE 独立的 DLL 文件。静态链接库和动态链接库的另外一个区别在于静态链接库中不能再包含其他的动态链接库或者静态库，而在动态链接库中还可以再包含其他的动态或静态链接库。

对动态链接库，我们还需建立如下概念：

（1）DLL 的编制与具体的编程语言及编译器无关

只要遵循约定的 DLL 接口规范和调用方式，用各种语言编写的 DLL 都可以相互调用。譬如 Windows 提供的系统 DLL（其中包括了 Windows 的 API），在任何开发环境中都能被调用，不在乎其是 Visual Basic、Visual C++还是 Delphi。

（2）动态链接库随处可见

我们在 Windows 目录下的 system32文件夹中会看到 kernel32.dll、user32.dll和 gdi32.dll，windows 的大多数 API 都包含在这些 DLL 中。kernel32.dll 中的函数主要处理内存管理和进程调度； user32.dll中的函数主要控制用户界面； gdi32.dll 中的函数则负责图形方面的操作。

一般的程序员都用过类似 MessageBox的函数，其实它就包含在 user32.dll 这个动态链接库中。由此可见 DLL 对我们来说其实并不陌生。

(3)VC 动态链接库的分类

Visual C++支持三种 DLL，它们分别是 Non-MFC DLL（非 MFC 动态库）、MFC Regular DLL（MFC 规则 DLL）、MFC Extension DLL（MFC 扩展 DLL）。

非 MFC 动态库不采用 MFC 类库结构，其导出函数为标准的 C 接口，能被非 MFC 或 MFC 编写的应用程序所调用； MFC 规则 DLL 包含一个继承自 CWinApp 的类，但其无消息循环；MFC 扩展 DLL 采用 MFC 的动态链接版本创建， 它只能被用 MFC 类库所编写的应用程序所调用。

由于本文篇幅较长， 内容较多， 势必需要先对阅读本文的有关事项进行说明， 下面以问答形式给出。

问：本文主要讲解什么内容？

答：本文详细介绍了 DLL 编程的方方面面， 努力学完本文应可以对 DLL 有较全面的掌握，并能编写大多数 DLL 程序。

问：如何看本文？

答：本文每一个主题的讲解都附带了源代码例程，可以随文下载（每个工程都经 WINRAR 压缩）。所有这些例程都由笔者编写并在 VC++6.0 中调试通过。

1.概论

先来阐述一下 DLL(Dynamic Linkable Library) 的概念，你可以简单的把 DLL 看成一种仓库，它提供给你一些可以直接拿来用的变量、函数或类。在仓库的发展史上经历了 ¡°无库－静态链接库－动态链接库¡±的时代。

静态链接库与动态链接库都是共享代码的方式， 如果采用静态链接库， 则无论你愿不愿意， lib 中的指令都被直接包含在最终生成的 EXE 文件中了。但是若使用 DLL，该 DLL 不必被包含在最终 EXE 文件中，EXE 文件执行时可以 ¡°动态¡±地引用和卸载这个与 EXE 独立的 DLL 文件。静态链接库和动态链接库的另外一个区别在于静态链接库中不能再包含其他的动态链接库或者静态库，而在动态链接库中还可以再包含其他的动态或静态链接库。

对动态链接库，我们还需建立如下概念：

（1）DLL 的编制与具体的编程语言及编译器无关

只要遵循约定的 DLL 接口规范和调用方式，用各种语言编写的 DLL 都可以相互调用。譬如 Windows 提供的系统 DLL（其中包括了 Windows 的 API），在任何开发环境中都能被调用，不在乎其是 Visual Basic、Visual C++还是 Delphi。

（2）动态链接库随处可见

我们在 Windows 目录下的 system32文件夹中会看到 kernel32.dll、user32.dll和 gdi32.dll，windows 的大多数 API 都包含在这些 DLL 中。kernel32.dll 中的函数主要处理内存管理和进程调度； user32.dll中的函数主要控制用户界面； gdi32.dll 中的函数则负责图形方面的操作。

一般的程序员都用过类似 MessageBox的函数，其实它就包含在 user32.dll 这个动态链接库中。由此可见 DLL 对我们来说其实并不陌生。

(3)VC 动态链接库的分类

Visual C++支持三种 DLL，它们分别是 Non-MFC DLL（非 MFC 动态库）、MFC Regular DLL（MFC 规则 DLL）、MFC Extension DLL（MFC 扩展 DLL）。

非 MFC 动态库不采用 MFC 类库结构，其导出函数为标准的 C 接口，能被非 MFC 或 MFC 编写的应用程序所调用； MFC 规则 DLL 包含一个继承自 CWinApp 的类，但其无消息循环；MFC 扩展 DLL 采用 MFC 的动态链接版本创建， 它只能被用 MFC 类库所编写的应用程序所调用。

由于本文篇幅较长， 内容较多， 势必需要先对阅读本文的有关事项进行说明， 下面以问答形式给出。

问：本文主要讲解什么内容？

答：本文详细介绍了 DLL 编程的方方面面， 努力学完本文应可以对 DLL 有较全面的掌握，并能编写大多数 DLL 程序。

问：如何看本文？

答：本文每一个主题的讲解都附带了源代码例程，可以随文下载（每个工程都经 WINRAR 压缩）。所有这些例程都由笔者编写并在 VC++6.0 中调试通过。

1.概论

先来阐述一下 DLL(Dynamic Linkable Library) 的概念，你可以简单的把 DLL 看成一种仓库，它提供给你一些可以直接拿来用的变量、函数或类。在仓库的发展史上经历了 ¡°无库－静态链接库－动态链接库¡±的时代。

静态链接库与动态链接库都是共享代码的方式， 如果采用静态链接库， 则无论你愿不愿意， lib 中的指令都被直接包含在最终生成的 EXE 文件中了。但是若使用 DLL，该 DLL 不必被包含在最终 EXE 文件中，EXE 文件执行时可以 ¡°动态¡±地引用和卸载这个与 EXE 独立的 DLL 文件。静态链接库和动态链接库的另外一个区别在于静态链接库中不能再包含其他的动态链接库或者静态库，而在动态链接库中还可以再包含其他的动态或静态链接库。

对动态链接库，我们还需建立如下概念：

（1）DLL 的编制与具体的编程语言及编译器无关

只要遵循约定的 DLL 接口规范和调用方式，用各种语言编写的 DLL 都可以相互调用。譬如 Windows 提供的系统 DLL（其中包括了 Windows 的 API），在任何开发环境中都能被调用，不在乎其是 Visual Basic、Visual C++还是 Delphi。

（2）动态链接库随处可见

我们在 Windows 目录下的 system32文件夹中会看到 kernel32.dll、user32.dll和 gdi32.dll，windows 的大多数 API 都包含在这些 DLL 中。kernel32.dll 中的函数主要处理内存管理和进程调度； user32.dll中的函数主要控制用户界面； gdi32.dll 中的函数则负责图形方面的操作。

一般的程序员都用过类似 MessageBox的函数，其实它就包含在 user32.dll 这个动态链接库中。由此可见 DLL 对我们来说其实并不陌生。

(3)VC 动态链接库的分类

Visual C++支持三种 DLL，它们分别是 Non-MFC DLL（非 MFC 动态库）、MFC Regular DLL（MFC 规则 DLL）、MFC Extension DLL（MFC 扩展 DLL）。

非 MFC 动态库不采用 MFC 类库结构，其导出函数为标准的 C 接口，能被非 MFC 或 MFC 编写的应用程序所调用； MFC 规则 DLL 包含一个继承自 CWinApp 的类，但其无消息循环；MFC 扩展 DLL 采用 MFC 的动态链接版本创建， 它只能被用 MFC 类库所编写的应用程序所调用。

由于本文篇幅较长， 内容较多， 势必需要先对阅读本文的有关事项进行说明， 下面以问答形式给出。

问：本文主要讲解什么内容？

答：本文详细介绍了 DLL 编程的方方面面， 努力学完本文应可以对 DLL 有较全面的掌握，并能编写大多数 DLL 程序。

问：如何看本文？

答：本文每一个主题的讲解都附带了源代码例程，可以随文下载（每个工程都经 WINRAR 压缩）。所有这些例程都由笔者编写并在 VC++6.0 中调试通过。

在具体进入各类 DLL 的详细阐述之前，有必要对库文件的调试与查看方法进行一下介绍，因为从下一节开始我们将面对大量的例子工程。

由于库文件不能单独执行， 因而在按下 F5（开始 debug模式执行） 或 CTRL+F5（运行）执行时，其弹出如图 3 所示的对话框，要求用户输入可执行文件的路径来启动库函数的执行。这个时候我们输入要调用该库的 EXE 文件的路径就可以对库进行调试了，其调试技巧与一般应用工程的调试一样。

图 3 库的调试与 ¡¶运行¡±

通常有比上述做法更好的调试途径， 那就是将库工程和应用工程 （调用库的工程） 放置在同一 VC 工作区，只对应用工程进行调试，在应用工程调用库中函数的语句处设置断点，执行后按下 F11，这样就单步进入了库中的函数。第 2 节中的 libTest 和 libCall 工程就放在了同一工作区，其工程结构如图 4 所示。

图 4 把库工程和调用库的工程放入同一工作区进行调试

上述调试方法对静态链接库和动态链接库而言是一致的。 所以本文提供下载的所有源代码中都包含了库工程和调用库的工程， 这二者都被包含在一个工作区内， 这是笔者提供这种打包下载的用意所在。

动态链接库中的导出接口可以使用 Visual C++的 Depends工具进行查看，让我们用 Depends 打开系统目录中的 user32.dll，看到了吧？红圈内的就是几个版本的 MessageBox了！原来它真的在这里啊，原来它就在这里啊！

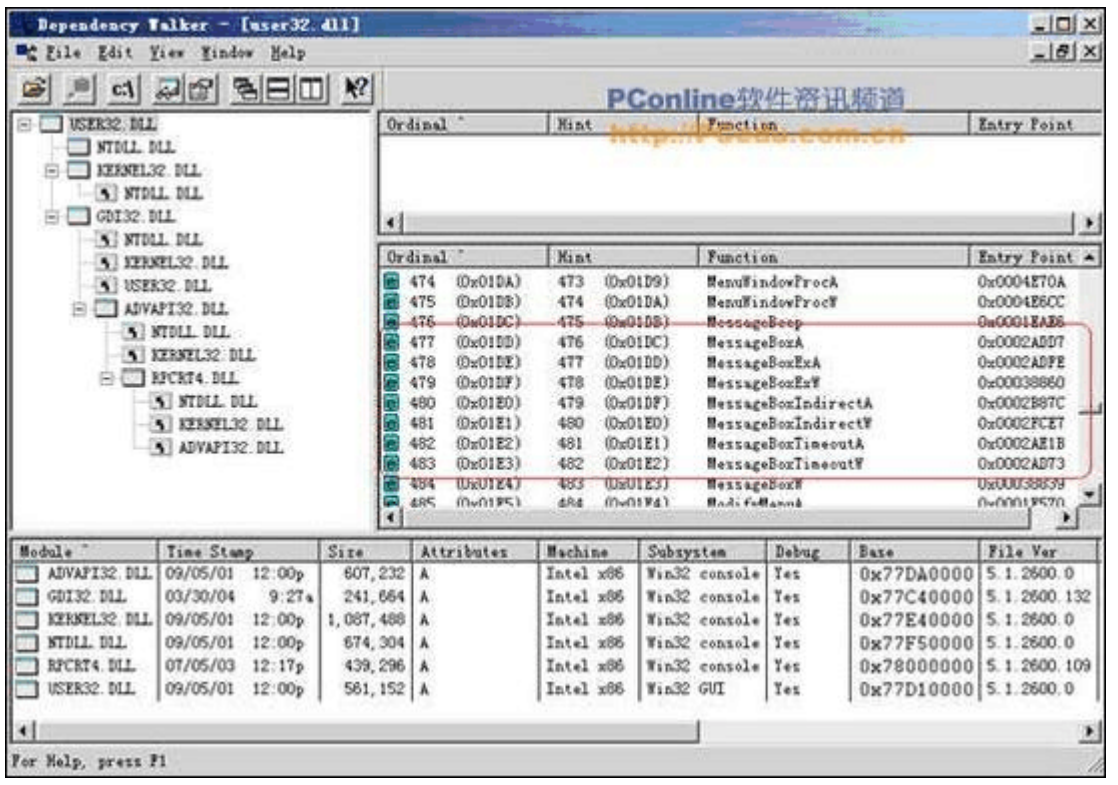

图 5　用 Depends查看 DLL

当然 Depends工具也可以显示 DLL 的层次结构，若用它打开一个可执行文件则可以看出这个可执行文件调用了哪些 DLL 。

好，让我们正式进入动态链接库的世界，先来看看最一般的 DLL，即非 MFC DLL(待续...)

上节给大家介绍了静态链接库与库的调试与查看（动态链接库 (DLL) 编程深入浅出 (一)），本节主要介绍非 MFC DLL 。

4.非 MFC DLL

4.1 一个简单的 DLL

第 2 节给出了以静态链接库方式提供 add 函数接口的方法，接下来我们来看看怎样用动态链接库实现一个同样功能的 add函数。

如图 6，在 VC++中 new 一个 Win32 Dynamic-Link Library 工程 dllTest（单击此处下载本工程附件）。注意不要选择 MFC AppWizard(dll) ，因为用 MFC AppWizard(dll) 建立的将是第 5、6 节要讲述的 MFC 动态链接库。

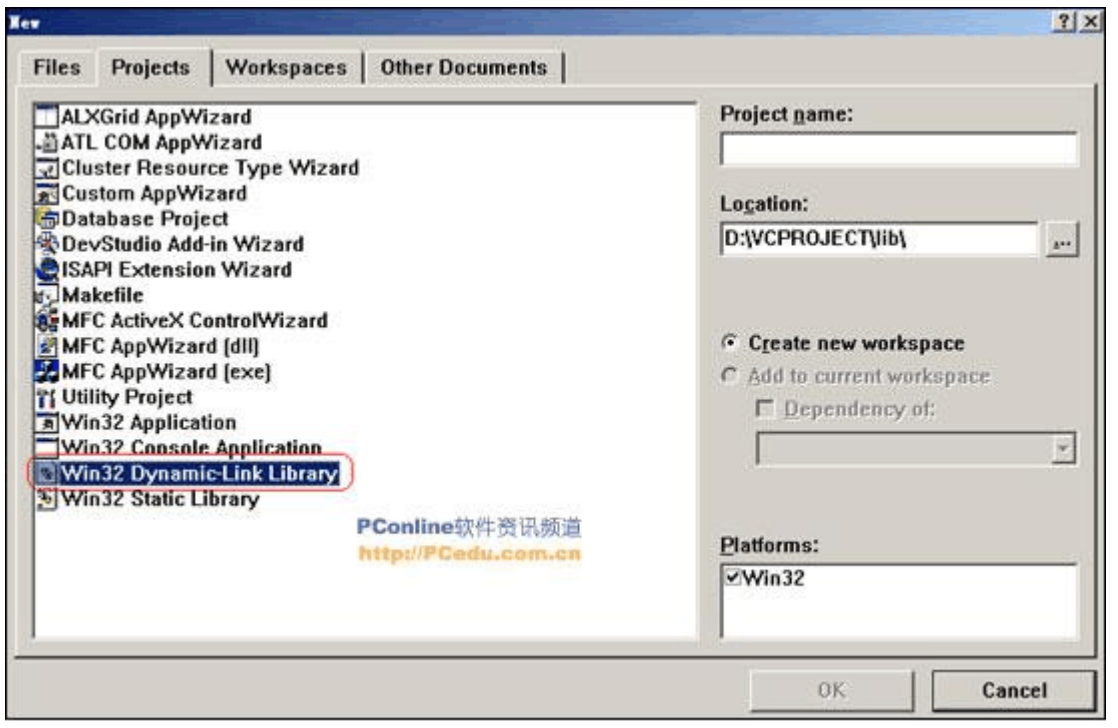

图 5　用 Depends查看 DLL

当然 Depends工具也可以显示 DLL 的层次结构，若用它打开一个可执行文件则可以看出这个可执行文件调用了哪些 DLL 。

好，让我们正式进入动态链接库的世界，先来看看最一般的 DLL，即非 MFC DLL(待续...)

上节给大家介绍了静态链接库与库的调试与查看（动态链接库 (DLL) 编程深入浅出 (一)），本节主要介绍非 MFC DLL 。

4.非 MFC DLL

4.1 一个简单的 DLL

第 2 节给出了以静态链接库方式提供 add 函数接口的方法，接下来我们来看看怎样用动态链接库实现一个同样功能的 add函数。

如图 6，在 VC++中 new 一个 Win32 Dynamic-Link Library 工程 dllTest（单击此处下载本工程附件）。注意不要选择 MFC AppWizard(dll) ，因为用 MFC AppWizard(dll) 建立的将是第 5、6 节要讲述的 MFC 动态链接库。

图 5　用 Depends查看 DLL

当然 Depends工具也可以显示 DLL 的层次结构，若用它打开一个可执行文件则可以看出这个可执行文件调用了哪些 DLL 。

好，让我们正式进入动态链接库的世界，先来看看最一般的 DLL，即非 MFC DLL(待续...)

上节给大家介绍了静态链接库与库的调试与查看（动态链接库 (DLL) 编程深入浅出 (一)），本节主要介绍非 MFC DLL 。

4.非 MFC DLL

4.1 一个简单的 DLL

第 2 节给出了以静态链接库方式提供 add 函数接口的方法，接下来我们来看看怎样用动态链接库实现一个同样功能的 add函数。

如图 6，在 VC++中 new 一个 Win32 Dynamic-Link Library 工程 dllTest（单击此处下载本工程附件）。注意不要选择 MFC AppWizard(dll) ，因为用 MFC AppWizard(dll) 建立的将是第 5、6 节要讲述的 MFC 动态链接库。

图 5　用 Depends查看 DLL

当然 Depends工具也可以显示 DLL 的层次结构，若用它打开一个可执行文件则可以看出这个可执行文件调用了哪些 DLL 。

好，让我们正式进入动态链接库的世界，先来看看最一般的 DLL，即非 MFC DLL(待续...)

上节给大家介绍了静态链接库与库的调试与查看（动态链接库 (DLL) 编程深入浅出 (一)），本节主要介绍非 MFC DLL 。

4.非 MFC DLL

4.1 一个简单的 DLL

第 2 节给出了以静态链接库方式提供 add 函数接口的方法，接下来我们来看看怎样用动态链接库实现一个同样功能的 add函数。

如图 6，在 VC++中 new 一个 Win32 Dynamic-Link Library 工程 dllTest（单击此处下载本工程附件）。注意不要选择 MFC AppWizard(dll) ，因为用 MFC AppWizard(dll) 建立的将是第 5、6 节要讲述的 MFC 动态链接库。

.def 文件的规则为：

(1)LIBRARY 语句说明 .def 文件相应的 DLL ；

(2)EXPORTS 语句后列出要导出函数的名称。可以在 .def 文件中的导出函数名后加 @n, 表示要导出函数的序号为 n（在进行函数调用时，这个序号将发挥其作用） ；

(3).def 文件中的注释由每个注释行开始处的分号 (;) 指定，且注释不能与语句共享一行。

由此可以看出，例子中 lib.def 文件的含义为生成名为 ¡°dllTest¡± 的动态链接库，导出其中的 add 函数，并指定 add 函数的序号为 1。

4.3 DLL 的调用方式

在 4.1 节的例子中我们看到了由 ¡°LoadLibrary-GetProcAddress-FreeLibrary¡± 系统 Api 提供的三位一体 ¡°DLL 加载-DLL 函数地址获取 -DLL 释放¡± 方式，这种调用方式称为 DLL 的动态调用。

动态调用方式的特点是完全由编程者用 API 函数加载和卸载 DLL，程序员可以决定 DLL 文件何时加载或不加载，显式链接在运行时决定加载哪个 DLL 文件。

与动态调用方式相对应的就是静态调用方式， ¡°有动必有静¡± 这来源于物质世界的对立统一。¡°动与静¡± 其对立与统一竟无数次在技术领域里得到验证，譬如静态 IP 与 DHCP、静态路由与动态路由等。从前文我们已经知道，库也分为静态库与动态库 DLL，而想不到，深入到 DLL 内部，其调用方式也分为静态与动态。 ¡°动与静¡± 无处不在。《周易》已认识到有动必有静的动静平衡观，《易．系辞》曰：¡°动静有常，刚柔断矣¡± 哲学意味着一种普遍的真理，因此，我们经常可以在枯燥的技术领域看到哲学的影子。

静态调用方式的特点是由编译系统完成对 DLL 的加载和应用程序结束时 DLL 的卸载。当调用某 DLL 的应用程序结束时，若系统中还有其它程序使用该 DLL，则 Windows 对 DLL 的应用记录减 1，直到所有使用该 DLL 的程序都结束时才释放它。静态调用方式简单实用，但不如动态调用方式灵活。

下面我们来看看静态调用的例子（单击此处下载本工程附件） ，将编译 dllTest 工程所生成的.lib 和.dll 文件拷入 dllCall 工程所在的路径， dllCall 执行下列代码：

//.lib 文件中仅仅是关于其对应 DLL 文件中函数的重定位信息

.def 文件的规则为：

(1)LIBRARY 语句说明 .def 文件相应的 DLL ；

(2)EXPORTS 语句后列出要导出函数的名称。可以在 .def 文件中的导出函数名后加 @n, 表示要导出函数的序号为 n（在进行函数调用时，这个序号将发挥其作用） ；

(3).def 文件中的注释由每个注释行开始处的分号 (;) 指定，且注释不能与语句共享一行。

由此可以看出，例子中 lib.def 文件的含义为生成名为 ¡°dllTest¡± 的动态链接库，导出其中的 add 函数，并指定 add 函数的序号为 1。

4.3 DLL 的调用方式

在 4.1 节的例子中我们看到了由 ¡°LoadLibrary-GetProcAddress-FreeLibrary¡± 系统 Api 提供的三位一体 ¡°DLL 加载-DLL 函数地址获取 -DLL 释放¡± 方式，这种调用方式称为 DLL 的动态调用。

动态调用方式的特点是完全由编程者用 API 函数加载和卸载 DLL，程序员可以决定 DLL 文件何时加载或不加载，显式链接在运行时决定加载哪个 DLL 文件。

与动态调用方式相对应的就是静态调用方式， ¡°有动必有静¡± 这来源于物质世界的对立统一。¡°动与静¡± 其对立与统一竟无数次在技术领域里得到验证，譬如静态 IP 与 DHCP、静态路由与动态路由等。从前文我们已经知道，库也分为静态库与动态库 DLL，而想不到，深入到 DLL 内部，其调用方式也分为静态与动态。 ¡°动与静¡± 无处不在。《周易》已认识到有动必有静的动静平衡观，《易．系辞》曰：¡°动静有常，刚柔断矣¡± 哲学意味着一种普遍的真理，因此，我们经常可以在枯燥的技术领域看到哲学的影子。

静态调用方式的特点是由编译系统完成对 DLL 的加载和应用程序结束时 DLL 的卸载。当调用某 DLL 的应用程序结束时，若系统中还有其它程序使用该 DLL，则 Windows 对 DLL 的应用记录减 1，直到所有使用该 DLL 的程序都结束时才释放它。静态调用方式简单实用，但不如动态调用方式灵活。

下面我们来看看静态调用的例子（单击此处下载本工程附件） ，将编译 dllTest 工程所生成的.lib 和.dll 文件拷入 dllCall 工程所在的路径， dllCall 执行下列代码：

//.lib 文件中仅仅是关于其对应 DLL 文件中函数的重定位信息

.def 文件的规则为：

(1)LIBRARY 语句说明 .def 文件相应的 DLL ；

(2)EXPORTS 语句后列出要导出函数的名称。可以在 .def 文件中的导出函数名后加 @n, 表示要导出函数的序号为 n（在进行函数调用时，这个序号将发挥其作用） ；

(3).def 文件中的注释由每个注释行开始处的分号 (;) 指定，且注释不能与语句共享一行。

由此可以看出，例子中 lib.def 文件的含义为生成名为 ¡°dllTest¡± 的动态链接库，导出其中的 add 函数，并指定 add 函数的序号为 1。

4.3 DLL 的调用方式

在 4.1 节的例子中我们看到了由 ¡°LoadLibrary-GetProcAddress-FreeLibrary¡± 系统 Api 提供的三位一体 ¡°DLL 加载-DLL 函数地址获取 -DLL 释放¡± 方式，这种调用方式称为 DLL 的动态调用。

动态调用方式的特点是完全由编程者用 API 函数加载和卸载 DLL，程序员可以决定 DLL 文件何时加载或不加载，显式链接在运行时决定加载哪个 DLL 文件。

与动态调用方式相对应的就是静态调用方式， ¡°有动必有静¡± 这来源于物质世界的对立统一。¡°动与静¡± 其对立与统一竟无数次在技术领域里得到验证，譬如静态 IP 与 DHCP、静态路由与动态路由等。从前文我们已经知道，库也分为静态库与动态库 DLL，而想不到，深入到 DLL 内部，其调用方式也分为静态与动态。 ¡°动与静¡± 无处不在。《周易》已认识到有动必有静的动静平衡观，《易．系辞》曰：¡°动静有常，刚柔断矣¡± 哲学意味着一种普遍的真理，因此，我们经常可以在枯燥的技术领域看到哲学的影子。

静态调用方式的特点是由编译系统完成对 DLL 的加载和应用程序结束时 DLL 的卸载。当调用某 DLL 的应用程序结束时，若系统中还有其它程序使用该 DLL，则 Windows 对 DLL 的应用记录减 1，直到所有使用该 DLL 的程序都结束时才释放它。静态调用方式简单实用，但不如动态调用方式灵活。

下面我们来看看静态调用的例子（单击此处下载本工程附件） ，将编译 dllTest 工程所生成的.lib 和.dll 文件拷入 dllCall 工程所在的路径， dllCall 执行下列代码：

//.lib 文件中仅仅是关于其对应 DLL 文件中函数的重定位信息

.def 文件的规则为：

(1)LIBRARY 语句说明 .def 文件相应的 DLL ；

(2)EXPORTS 语句后列出要导出函数的名称。可以在 .def 文件中的导出函数名后加 @n, 表示要导出函数的序号为 n（在进行函数调用时，这个序号将发挥其作用） ；

(3).def 文件中的注释由每个注释行开始处的分号 (;) 指定，且注释不能与语句共享一行。

由此可以看出，例子中 lib.def 文件的含义为生成名为 ¡°dllTest¡± 的动态链接库，导出其中的 add 函数，并指定 add 函数的序号为 1。

4.3 DLL 的调用方式

在 4.1 节的例子中我们看到了由 ¡°LoadLibrary-GetProcAddress-FreeLibrary¡± 系统 Api 提供的三位一体 ¡°DLL 加载-DLL 函数地址获取 -DLL 释放¡± 方式，这种调用方式称为 DLL 的动态调用。

动态调用方式的特点是完全由编程者用 API 函数加载和卸载 DLL，程序员可以决定 DLL 文件何时加载或不加载，显式链接在运行时决定加载哪个 DLL 文件。

与动态调用方式相对应的就是静态调用方式， ¡°有动必有静¡± 这来源于物质世界的对立统一。¡°动与静¡± 其对立与统一竟无数次在技术领域里得到验证，譬如静态 IP 与 DHCP、静态路由与动态路由等。从前文我们已经知道，库也分为静态库与动态库 DLL，而想不到，深入到 DLL 内部，其调用方式也分为静态与动态。 ¡°动与静¡± 无处不在。《周易》已认识到有动必有静的动静平衡观，《易．系辞》曰：¡°动静有常，刚柔断矣¡± 哲学意味着一种普遍的真理，因此，我们经常可以在枯燥的技术领域看到哲学的影子。

静态调用方式的特点是由编译系统完成对 DLL 的加载和应用程序结束时 DLL 的卸载。当调用某 DLL 的应用程序结束时，若系统中还有其它程序使用该 DLL，则 Windows 对 DLL 的应用记录减 1，直到所有使用该 DLL 的程序都结束时才释放它。静态调用方式简单实用，但不如动态调用方式灵活。

下面我们来看看静态调用的例子（单击此处下载本工程附件） ，将编译 dllTest 工程所生成的.lib 和.dll 文件拷入 dllCall 工程所在的路径， dllCall 执行下列代码：

//.lib 文件中仅仅是关于其对应 DLL 文件中函数的重定位信息

.def 文件的规则为：

(1)LIBRARY 语句说明 .def 文件相应的 DLL ；

(2)EXPORTS 语句后列出要导出函数的名称。可以在 .def 文件中的导出函数名后加 @n, 表示要导出函数的序号为 n（在进行函数调用时，这个序号将发挥其作用） ；

(3).def 文件中的注释由每个注释行开始处的分号 (;) 指定，且注释不能与语句共享一行。

由此可以看出，例子中 lib.def 文件的含义为生成名为 ¡°dllTest¡± 的动态链接库，导出其中的 add 函数，并指定 add 函数的序号为 1。

4.3 DLL 的调用方式

在 4.1 节的例子中我们看到了由 ¡°LoadLibrary-GetProcAddress-FreeLibrary¡± 系统 Api 提供的三位一体 ¡°DLL 加载-DLL 函数地址获取 -DLL 释放¡± 方式，这种调用方式称为 DLL 的动态调用。

动态调用方式的特点是完全由编程者用 API 函数加载和卸载 DLL，程序员可以决定 DLL 文件何时加载或不加载，显式链接在运行时决定加载哪个 DLL 文件。

与动态调用方式相对应的就是静态调用方式， ¡°有动必有静¡± 这来源于物质世界的对立统一。¡°动与静¡± 其对立与统一竟无数次在技术领域里得到验证，譬如静态 IP 与 DHCP、静态路由与动态路由等。从前文我们已经知道，库也分为静态库与动态库 DLL，而想不到，深入到 DLL 内部，其调用方式也分为静态与动态。 ¡°动与静¡± 无处不在。《周易》已认识到有动必有静的动静平衡观，《易．系辞》曰：¡°动静有常，刚柔断矣¡± 哲学意味着一种普遍的真理，因此，我们经常可以在枯燥的技术领域看到哲学的影子。

静态调用方式的特点是由编译系统完成对 DLL 的加载和应用程序结束时 DLL 的卸载。当调用某 DLL 的应用程序结束时，若系统中还有其它程序使用该 DLL，则 Windows 对 DLL 的应用记录减 1，直到所有使用该 DLL 的程序都结束时才释放它。静态调用方式简单实用，但不如动态调用方式灵活。

下面我们来看看静态调用的例子（单击此处下载本工程附件） ，将编译 dllTest 工程所生成的.lib 和.dll 文件拷入 dllCall 工程所在的路径， dllCall 执行下列代码：

//.lib 文件中仅仅是关于其对应 DLL 文件中函数的重定位信息

```
#define APIENTRY WINAPI //DllMain 的入口就在这里

#define APIPRIVATE __stdcall

#define PASCAL __stdcall
```

在 lib.h 中，应这样声明 add 函数：

```
int __stdcall add(int x, int y);
```

在应用工程中函数指针类型应定义为：

```
typedef int(__stdcall *lpAddFun)(int, int);
```

若在 lib.h 中将函数声明为 __stdcall 调用，而应用工程中仍使用 typedef int (*lpAddFun)(int,int)，运行时将发生错误（因为类型不匹配，在应用工程中仍然是缺省的 __cdecl 调用），弹出如图 7 所示的对话框。

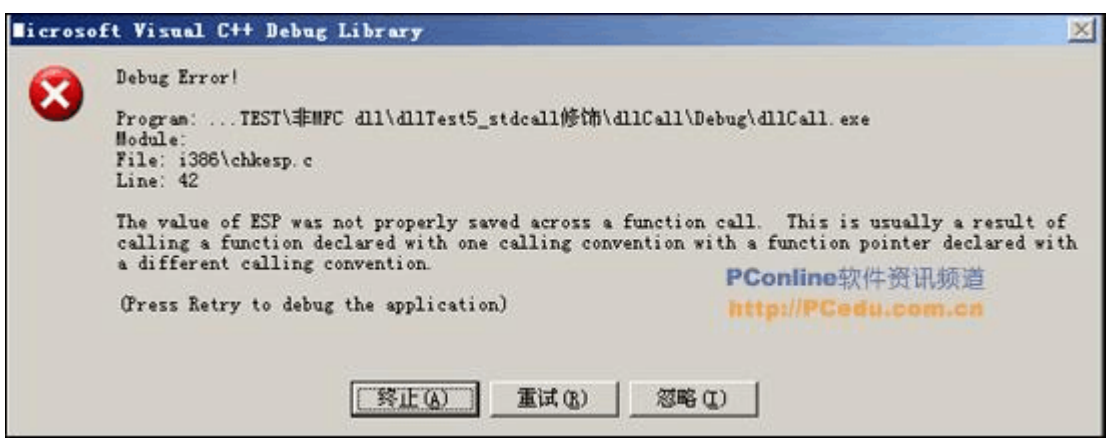

图 7 调用约定不匹配时的运行错误

图 8 中的那段话实际上已经给出了错误的原因，即 ¡°This is usually a result of ¡-¡±。

单击此处下载 __stdcall调用例子工程源代码附件。

4.6 DLL 导出变量

DLL 定义的全局变量可以被调用进程访问； DLL 也可以访问调用进程的全局数据，我们来看看在应用工程中引用 DLL 中变量的例子（单击此处下载本工程附件）。

```
/*  文件名： lib.h   */

#ifndef LIB_H

#define LIB_H

extern int dllGlobalVar;
```

```
#define APIENTRY WINAPI //DllMain 的入口就在这里

#define APIPRIVATE __stdcall

#define PASCAL __stdcall
```

在 lib.h 中，应这样声明 add 函数：

```
int __stdcall add(int x, int y);
```

在应用工程中函数指针类型应定义为：

```
typedef int(__stdcall *lpAddFun)(int, int);
```

若在 lib.h 中将函数声明为 __stdcall 调用，而应用工程中仍使用 typedef int (*lpAddFun)(int,int)，运行时将发生错误（因为类型不匹配，在应用工程中仍然是缺省的 __cdecl 调用），弹出如图 7 所示的对话框。

图 7 调用约定不匹配时的运行错误

图 8 中的那段话实际上已经给出了错误的原因，即 ¡°This is usually a result of ¡-¡±。

单击此处下载 __stdcall调用例子工程源代码附件。

4.6 DLL 导出变量

DLL 定义的全局变量可以被调用进程访问； DLL 也可以访问调用进程的全局数据，我们来看看在应用工程中引用 DLL 中变量的例子（单击此处下载本工程附件） 。

```
/* 文件名： lib.h */

#ifndef LIB_H

#define LIB_H

extern int dllGlobalVar;
```

```
#define APIENTRY WINAPI //DllMain 的入口就在这里

#define APIPRIVATE __stdcall

#define PASCAL __stdcall
```

在 lib.h 中，应这样声明 add 函数：

```
int __stdcall add(int x, int y);
```

在应用工程中函数指针类型应定义为：

```
typedef int(__stdcall *lpAddFun)(int, int);
```

若在 lib.h 中将函数声明为 __stdcall 调用，而应用工程中仍使用 typedef int (*lpAddFun)(int,int)，运行时将发生错误（因为类型不匹配，在应用工程中仍然是缺省的 __cdecl 调用），弹出如图 7 所示的对话框。

图 7 调用约定不匹配时的运行错误

图 8 中的那段话实际上已经给出了错误的原因，即 ¡°This is usually a result of ¡-¡±。

单击此处下载 __stdcall调用例子工程源代码附件。

4.6 DLL 导出变量

DLL 定义的全局变量可以被调用进程访问； DLL 也可以访问调用进程的全局数据，我们来看看在应用工程中引用 DLL 中变量的例子（单击此处下载本工程附件） 。

```
/* 文件名： lib.h */

#ifndef LIB_H

#define LIB_H

extern int dllGlobalVar;
```

```
#define APIENTRY WINAPI //DllMain 的入口就在这里

#define APIPRIVATE __stdcall

#define PASCAL __stdcall
```

在 lib.h 中，应这样声明 add 函数：

```
int __stdcall add(int x, int y);
```

在应用工程中函数指针类型应定义为：

```
typedef int(__stdcall *lpAddFun)(int, int);
```

若在 lib.h 中将函数声明为 __stdcall 调用，而应用工程中仍使用 typedef int (*lpAddFun)(int,int)，运行时将发生错误（因为类型不匹配，在应用工程中仍然是缺省的 __cdecl 调用），弹出如图 7 所示的对话框。

图 7 调用约定不匹配时的运行错误

图 8 中的那段话实际上已经给出了错误的原因，即 ¡°This is usually a result of ¡-¡±。

单击此处下载 __stdcall调用例子工程源代码附件。

4.6 DLL 导出变量

DLL 定义的全局变量可以被调用进程访问； DLL 也可以访问调用进程的全局数据，我们来看看在应用工程中引用 DLL 中变量的例子（单击此处下载本工程附件） 。

```
/* 文件名： lib.h */

#ifndef LIB_H

#define LIB_H

extern int dllGlobalVar;
```

```
#define APIENTRY WINAPI //DllMain 的入口就在这里

#define APIPRIVATE __stdcall

#define PASCAL __stdcall
```

在 lib.h 中，应这样声明 add 函数：

```
int __stdcall add(int x, int y);
```

在应用工程中函数指针类型应定义为：

```
typedef int(__stdcall *lpAddFun)(int, int);
```

若在 lib.h 中将函数声明为 __stdcall 调用，而应用工程中仍使用 typedef int (*lpAddFun)(int,int)，运行时将发生错误（因为类型不匹配，在应用工程中仍然是缺省的 __cdecl 调用），弹出如图 7 所示的对话框。

图 7 调用约定不匹配时的运行错误

图 8 中的那段话实际上已经给出了错误的原因，即 ¡°This is usually a result of ¡-¡±。

单击此处下载 __stdcall调用例子工程源代码附件。

4.6 DLL 导出变量

DLL 定义的全局变量可以被调用进程访问； DLL 也可以访问调用进程的全局数据，我们来看看在应用工程中引用 DLL 中变量的例子（单击此处下载本工程附件） 。

```
/* 文件名： lib.h */

#ifndef LIB_H

#define LIB_H

extern int dllGlobalVar;
```

```
#define APIENTRY WINAPI //DllMain 的入口就在这里

#define APIPRIVATE __stdcall

#define PASCAL __stdcall
```

在 lib.h 中，应这样声明 add 函数：

```
int __stdcall add(int x, int y);
```

在应用工程中函数指针类型应定义为：

```
typedef int(__stdcall *lpAddFun)(int, int);
```

若在 lib.h 中将函数声明为 __stdcall 调用，而应用工程中仍使用 typedef int (*lpAddFun)(int,int)，运行时将发生错误（因为类型不匹配，在应用工程中仍然是缺省的 __cdecl 调用），弹出如图 7 所示的对话框。

图 7 调用约定不匹配时的运行错误

图 8 中的那段话实际上已经给出了错误的原因，即 ¡°This is usually a result of ¡-¡±。

单击此处下载 __stdcall调用例子工程源代码附件。

4.6 DLL 导出变量

DLL 定义的全局变量可以被调用进程访问； DLL 也可以访问调用进程的全局数据，我们来看看在应用工程中引用 DLL 中变量的例子（单击此处下载本工程附件） 。

```
/* 文件名： lib.h */

#ifndef LIB_H

#define LIB_H

extern int dllGlobalVar;
```

```
#define APIENTRY WINAPI //DllMain 的入口就在这里

#define APIPRIVATE __stdcall

#define PASCAL __stdcall
```

在 lib.h 中，应这样声明 add 函数：

```
int __stdcall add(int x, int y);
```

在应用工程中函数指针类型应定义为：

```
typedef int(__stdcall *lpAddFun)(int, int);
```

若在 lib.h 中将函数声明为 __stdcall 调用，而应用工程中仍使用 typedef int (*lpAddFun)(int,int)，运行时将发生错误（因为类型不匹配，在应用工程中仍然是缺省的 __cdecl 调用），弹出如图 7 所示的对话框。

图 7 调用约定不匹配时的运行错误

图 8 中的那段话实际上已经给出了错误的原因，即 ¡°This is usually a result of ¡-¡±。

单击此处下载 __stdcall调用例子工程源代码附件。

4.6 DLL 导出变量

DLL 定义的全局变量可以被调用进程访问； DLL 也可以访问调用进程的全局数据，我们来看看在应用工程中引用 DLL 中变量的例子（单击此处下载本工程附件） 。

```
/* 文件名： lib.h */

#ifndef LIB_H

#define LIB_H

extern int dllGlobalVar;
```

```
#define APIENTRY WINAPI //DllMain 的入口就在这里

#define APIPRIVATE __stdcall

#define PASCAL __stdcall
```

在 lib.h 中，应这样声明 add 函数：

```
int __stdcall add(int x, int y);
```

在应用工程中函数指针类型应定义为：

```
typedef int(__stdcall *lpAddFun)(int, int);
```

若在 lib.h 中将函数声明为 __stdcall 调用，而应用工程中仍使用 typedef int (*lpAddFun)(int,int)，运行时将发生错误（因为类型不匹配，在应用工程中仍然是缺省的 __cdecl 调用），弹出如图 7 所示的对话框。

图 7 调用约定不匹配时的运行错误

图 8 中的那段话实际上已经给出了错误的原因，即 ¡°This is usually a result of ¡-¡±。

单击此处下载 __stdcall调用例子工程源代码附件。

4.6 DLL 导出变量

DLL 定义的全局变量可以被调用进程访问； DLL 也可以访问调用进程的全局数据，我们来看看在应用工程中引用 DLL 中变量的例子（单击此处下载本工程附件） 。

```
/* 文件名： lib.h */

#ifndef LIB_H

#define LIB_H

extern int dllGlobalVar;
```

```
#define APIENTRY WINAPI //DllMain 的入口就在这里

#define APIPRIVATE __stdcall

#define PASCAL __stdcall
```

在 lib.h 中，应这样声明 add 函数：

```
int __stdcall add(int x, int y);
```

在应用工程中函数指针类型应定义为：

```
typedef int(__stdcall *lpAddFun)(int, int);
```

若在 lib.h 中将函数声明为 __stdcall 调用，而应用工程中仍使用 typedef int (*lpAddFun)(int,int)，运行时将发生错误（因为类型不匹配，在应用工程中仍然是缺省的 __cdecl 调用），弹出如图 7 所示的对话框。

图 7 调用约定不匹配时的运行错误

图 8 中的那段话实际上已经给出了错误的原因，即 ¡°This is usually a result of ¡-¡±。

单击此处下载 __stdcall调用例子工程源代码附件。

4.6 DLL 导出变量

DLL 定义的全局变量可以被调用进程访问； DLL 也可以访问调用进程的全局数据，我们来看看在应用工程中引用 DLL 中变量的例子（单击此处下载本工程附件） 。

```
/* 文件名： lib.h */

#ifndef LIB_H

#define LIB_H

extern int dllGlobalVar;
```

```
#define APIENTRY WINAPI //DllMain 的入口就在这里

#define APIPRIVATE __stdcall

#define PASCAL __stdcall
```

在 lib.h 中，应这样声明 add 函数：

```
int __stdcall add(int x, int y);
```

在应用工程中函数指针类型应定义为：

```
typedef int(__stdcall *lpAddFun)(int, int);
```

若在 lib.h 中将函数声明为 __stdcall 调用，而应用工程中仍使用 typedef int (*lpAddFun)(int,int)，运行时将发生错误（因为类型不匹配，在应用工程中仍然是缺省的 __cdecl 调用），弹出如图 7 所示的对话框。

图 7 调用约定不匹配时的运行错误

图 8 中的那段话实际上已经给出了错误的原因，即 ¡°This is usually a result of ¡-¡±。

单击此处下载 __stdcall调用例子工程源代码附件。

4.6 DLL 导出变量

DLL 定义的全局变量可以被调用进程访问； DLL 也可以访问调用进程的全局数据，我们来看看在应用工程中引用 DLL 中变量的例子（单击此处下载本工程附件） 。

```
/* 文件名： lib.h */

#ifndef LIB_H

#define LIB_H

extern int dllGlobalVar;
```

#else

class _declspec(dllimport) class_name导入类

#endif

实际上，在 MFC DLL 的讲解中，您将看到比这更简便的方法，而此处仅仅是为了说明 _declspec(dllexport)与 _declspec(dllimport)匹对的问题。

由此可见，应用工程中几乎可以看到 DLL 中的一切，包括函数、变量以及类，这就是 DLL 所要提供的强大能力。只要 DLL 释放这些接口，应用程序使用它就将如同使用本工程中的程序一样！

本章虽以 VC++为平台讲解非 MFC DLL，但是这些普遍的概念在其它语言及开发环境中也是相同的，其思维方式可以直接过渡。

接下来，我们将要研究 MFC 规则 DLL(待续...)