Win32环境下动态链接库(DLL)编程原理

比较大[应用](http://search.yesky.com/search.do?wd=%E5%BA%94%E7%94%A8" \t "_blank)程序都由很多模块组成，这些模块分别完成相对独立的功能，它们彼此协作来完成整个软件系统的工作。其中可能存在一些模块的功能较为[通用](http://product.yesky.com/financialsoftware/tongyong/)，在构造其它软件系统时仍会被使用。在构造软件系统时，如果将所有模块的[源代码](http://mydown.yesky.com/soft/83/83685.html)都静态编译到整个应用程序EXE文件中，会产生一些问题：一个缺点是增加了应用程序的大小，它会占用更多的磁盘空间，程序运行时也会消耗较大的[内存](http://product.yesky.com/memory/" \t "_blank)空间，造成系统资源的浪费；另一个缺点是，在编写大的EXE程序时，在每次修改重建时都必须调整编译所有源代码，增加了编译过程的复杂性，也不利于阶段性的单元[测试](http://product.yesky.com/saas/cs/" \t "_blank)。   
  
　　Windows系统平台上提供了一种完全不同的较有效的编程和运行环境，你可以将独立的程序模块创建为较小的DLL([Dynamic](http://mydown.yesky.com/soft/2/2896.html) Linkable Library)文件，并可对它们单独编译和测试。在运行时，只有当EXE程序确实要调用这些DLL模块的情况下，系统才会将它们装载到内存空间中。这种 方式不仅减少了EXE文件的大小和对内存空间的需求，而且使这些DLL模块可以同时被多个应用程序使用。[Microsoft](http://product.yesky.com/database/microsoft/" \t "_blank) Windows自己就将一些主要的系统功能以DLL模块的形式实现。例如IE中的一些基本功能就是由DLL文件实现的，它可以被其它应用程序调用和集成。  
  
　 　一般来说，DLL是一种磁盘文件（通常带有DLL扩展名），它由全局数据、服务函数和资源组成，在运行时被系统加载到进程的虚拟空间中，成为调用进程的 一部分。如果与其它DLL之间没有冲突，该文件通常映射到进程虚拟空间的同一地址上。DLL模块中包含各种导出函数，用于向外界提供服务。Windows 在加载DLL模块时将进程函数调用与DLL文件的导出函数相匹配。  
  
　　在Win32环境中，每个进程都复制了自己的读/写全局变量。如果想要与其它进程共享内存，必须使用内存映射文件或者声明一个共享数据段。DLL模块需要的堆栈内存都是从运行进程的堆栈中分配出来的。  
  
　　DLL现在越来越容易编写。Win32已经大大简化了其编程模式，并有许多来自AppWizard和MFC类库的支持。  
  
　　**一、导出和导入函数的匹配**  
　 　DLL文件中包含一个导出函数表。这些导出函数由它们的符号名和称为标识号的整数与外界联系起来。函数表中还包含了DLL中函数的地址。当应用程序加载 DLL模块时时，它并不知道调用函数的实际地址，但它知道函数的符号名和标识号。动态链接过程在加载的DLL模块时动态建立一个函数调用与函数地址的对应 表。如果重新编译和重建DLL文件，并不需要修改应用程序，除非你改变了导出函数的符号名和参数序列。  
  
　　简单的DLL文件只为应用程序提供导出函数，比较复杂的DLL文件除了提供导出函数以外，还调用其它DLL文件中的函数。这样，一个特殊的DLL可以既有导入函数，又有导入函数。这并不是一个问题，因为动态链接过程可以处理交叉相关的情况。  
  
　　在DLL代码中，必须像下面这样明确声明导出函数：  
  
\_\_declspec(dllexport) int MyFunction(int n);  
  
　　但也可以在模块定义(DEF)文件中列出导出函数，不过这样做常常引起更多的麻烦。在应用程序方面，要求像下面这样明确声明相应的输入函数：  
  
\_\_declspec(dllimport) int MyFuncition(int n);  
  
　　仅有导入和导出声明并不能使应用程序内部的函数调用链接到相应的DLL文件上。应用程序的项目必须为链接程序指定所需的输入库（LIB文件）。而且应用程序事实上必须至少包含一个对DLL函数的调用。  
  
　　**二、与DLL模块建立链接**  
  
　　应用程序导入函数与DLL文件中的导出函数进行链接有两种方式：隐式链接和显式链接。所谓的隐式链接是指在应用程序中不需指明DLL文件的实际存储路径，程序员不需关心DLL文件的实际装载。而显式链接与此相反。  
  
　 　采用隐式链接方式，程序员在建立一个DLL文件时，链接程序会自动生成一个与之对应的LIB导入文件。该文件包含了每一个DLL导出函数的符号名和可选 的标识号，但是并不含有实际的代码。LIB文件作为DLL的替代文件被编译到应用程序项目中。当程序员通过静态链接方式编译生成应用程序时，应用程序中的 调用函数与LIB文件中导出符号相匹配，这些符号或标识号进入到生成的EXE文件中。LIB文件中也包含了对应的DLL文件名（但不是完全的路径名），链 接程序将其存储在EXE文件内部。当应用程序运行过程中需要加载DLL文件时，Windows根据这些信息发现并加载DLL，然后通过符号名或标识号实现 对DLL函数的动态链接。  
  
　　显式链接方式对于集成化的开发语言（例如VB）比较适合。有了显式链接，程序员就不必再使用导入文件，而是 直接调用Win32 的LoadLibary函数，并指定DLL的路径作为参数。LoadLibary返回HINSTANCE参数，应用程序在调用 GetProcAddress函数时使用这一参数。GetProcAddress函数将符号名或标识号转换为DLL内部的地址。假设有一个导出如下函数的 DLL文件：  
  
extern "[C](http://product.yesky.com/product/225/225032/)" \_\_declspec(dllexport) double SquareRoot(double d);  
  
　　下面是应用程序对该导出函数的显式链接的例子：  
c====》应用

win/linux系统编程

api  
typedef double(SQRTPROC)(double);  
HINSTANCE hInstance;  
SQRTPROC\* pFunction;  
VERIFY(hInstance=::LoadLibrary("c:\\winnt\\system32\\mydll.dll"));  
VERIFY(pFunction=(SQRTPROC\*)::GetProcAddress(hInstance,"SquareRoot"));  
double d=(\*pFunction)(81.0);//调用该DLL函数  
  
　 　在隐式链接方式中，所有被应用程序调用的DLL文件都会在应用程序EXE文件加载时被加载在到内存中；但如果采用显式链接方式，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。例如，可以将一个带有字符串资源的DLL模块以英语加载，而另一个以西班牙语加载。应用程 序在用户选择了合适的语种后再加载与之对应的DLL文件。  
  
　　**三、使用符号名链接与标识号链接**  
　 　在Win16环境中，符号名链接效率较低，所有那时标识号链接是主要的链接方式。在Win32环境中，符号名链接的效率得到了改善。Microsoft 现在推荐使用符号名链接。但在MFC库中的DLL版本仍然采用的是标识号链接。一个典型的MFC程序可能会链接到数百个MFC DLL函数上。采用标识号链接的应用程序的EXE文件体相对较小，因为它不必包含导入函数的长字符串符号名。

**四、编写DllMain函数**  
　　DllMain函数是DLL模块的默认入口点。当Windows加载 DLL模块时调用这一函数。系统首先调用全局对象的构造函数，然后调用全局函数DLLMain。DLLMain函数不仅在将DLL链接加载到进程时被调 用，在DLL模块与进程分离时（以及其它时候）也被调用。下面是一个框架DLLMain函数的例子。  
  
HINSTANCE g\_hInstance;  
extern "[C](http://product.yesky.com/product/225/225032/)" int APIENTRY DllMain(HINSTANCE hInstance,DWORD dwReason,LPVOID lpReserved)  
{  
if(dwReason==DLL\_PROCESS\_ATTACH)  
{  
TRACE0("EX22A.DLL Initializing!\n");  
//在这里进行初始化  
}  
else if(dwReason=DLL\_PROCESS\_DETACH)  
{  
TRACE0("EX22A.DLL Terminating!\n");  
//在这里进行清除工作  
}  
return 1;//成功  
}  
  
　　如果程序员没有为DLL模块编写一个DLLMain函数，系统会从其它运行库中引入一个不做任何操作的缺省DLLMain函数版本。在单个线程启动和终止时，DLLMain函数也被调用。正如由dwReason参数所表明的那样。  
  
　　**五、模块句柄**  
  
　 　进程中的每个DLL模块被全局唯一的32字节的HINSTANCE句柄标识。进程自己还有一个HINSTANCE句柄。所有这些模块句柄都只有在特定的 进程内部有效，它们代表了DLL或EXE模块在进程虚拟空间中的起始地址。在Win32中，HINSTANCE和HMODULE的值是相同的，这个两种类 型可以替换使用。进程模块句柄几乎总是等于0x400000，而DLL模块的加载地址的缺省句柄是0x10000000。如果程序同时使用了几个DLL模 块，每一个都会有不同的HINSTANCE值。这是因为在创建DLL文件时指定了不同的基地址，或者是因为加载程序对DLL代码进行了重定位。  
模块句柄对于加载资源特别重要。Win32 的FindResource函数中带有一个HINSTANCE参数。EXE和DLL都有其自己的资源。如果[应用](http://search.yesky.com/search.do?wd=%E5%BA%94%E7%94%A8" \t "_blank)程序需要来自于DLL的资源，就将此参数指定为DLL的模块句柄。如果需要EXE文件中包含的资源，就指定EXE的模块句柄。  
  
　　但是在使用这些句柄之前存在一个问题，你怎样得到它们呢？如果需要得到EXE模块句柄，调用带有Null参数的Win32函数GetModuleHandle；如果需要DLL模块句柄，就调用以DLL文件名为参数的Win32函数GetModuleHandle。  
  
　　**六、应用程序怎样找到DLL文件**  
　　如果应用程序使用LoadLibrary显式链接，那么在这个函数的参数中可以指定DLL文件的完整路径。如果不指定路径，或是进行隐式链接，Windows将遵循下面的搜索顺序来定位DLL：  
  
　　1． 包含EXE文件的目录，  
　　2． 进程的当前工作目录，  
　　3． Windows系统目录，  
　　4． Windows目录，  
　　5． 列在[Path](http://mydown.yesky.com/soft/utilitie/systems/19/423519.shtml)环境变量中的一系列目录。  
  
　 　这里有一个很容易发生错误的陷阱。如果你使用VC＋＋进行项目开发，并且为DLL模块专门创建了一个项目，然后将生成的DLL文件拷贝到系统目录下，从 应用程序中调用DLL模块。到目前为止，一切正常。接下来对DLL模块做了一些修改后重新生成了新的DLL文件，但你忘记将新的DLL文件拷贝到系统目录 下。下一次当你运行应用程序时，它仍加载了老版本的DLL文件，这可要当心！  
  
　　**七、调试DLL程序**  
　　[Microsoft](http://product.yesky.com/database/microsoft/) 的VC＋＋是开发和[测试](http://product.yesky.com/saas/cs/" \t "_blank)DLL 的有效工具，只需从DLL项目中运行调试程序即可。当你第一次这样操作时，调试程序会向你询问EXE文件的路径。此后每次在调试程序中运行DLL时，调试 程序会自动加载该EXE文件。然后该EXE文件用上面的搜索序列发现DLL文件，这意味着你必须设置Path环境变量让其包含DLL文件的磁盘路径，或者 也可以将DLL文件拷贝到搜索序列中的目录路径下。

**DLL分配的内存如何在EXE里面释放**

总结下面几个要点：

1. 保证内存分配和清除的统一性：如果一个DLL提供一个能够分配内存的函数，那么这个DLL同时应该提供一个函数释放这些内存。数据的创建和清除应该在同一个层次上。

曾经遇到过这样的例子：在dll中分配了一块内存,通过PostMessage将其地址传给应用。然后应用去释放它,结果总是报异常。

2.如果exe用 MFC Appwizard方式生成， dll用win32方式生成，则运行时会出现错误。进一步用单步跟踪，发现mfc方式和win32方式下的new操作符是用不同方式实现的，源程序分别在VC目录的文件 Afxmem.cpp和new.cpp中。有兴趣的话可以自已跟踪一下。

因为dll输出函数后，并不知道是哪一个模拟调用它，因此new和delete配对时最好在一个文件中，这样可以保证一致性。

3. 问题主要在于DLL和EXE主程序中分配内存的堆不一样，你可以不用new和delete，而是用

1） ::HeapAlloc(::GetProcessHeap(),...)和::HeapFree(::GetProcessHeap(),...)

2） ::GlobalAlloc()和::GlobalFree()

这两对API，这样无论在DLL中还是在主程序中都是在进程默认堆中分配，就不会出错了。

4. 还有一个办法，就是把dll的Settings的C/C++选项卡的Code Generation的Use Run-time liberary改成Debug Multithreaded DLL，在Release版本中改成Multithreaded DLL，就可以直接使用new和delete了。不过MFC就不能用Shared模式了。