2011年计算机等级考试二级Delphi辅导讲义：动态链接库编程
 导读：本章主要介绍Windows的动态链接库原理、DLLs的编写和调用、利用DLLs实现数据传输、利用DLLs实现窗体重用。>>>>点击查看此系列辅导讲义汇总第十章 动态链接库编程 10.1 Windows的动态链接库原理 动态链接库(DLLs)是从C语言函数库和Pascal库单元的概念发展而来的。所有的C语言标准库函数都存放在某一函数库中，同时用户也可以用LIB程序创建自己的函数库。在链接应用程序的过程中，链接器从库文件中拷贝程序调用的函数代码，并把这些函数代码添加到可执行文件中。这种方法同只把函数储存在已编译的.OBJ文件中相比更有利于代码的重用。但随着Windows这样的多任务环境的出现，函数库的方法显得过于累赘。如果为了完成屏幕输出、消息处理、内存管理、对话框等操作，每个程序都不得不拥有自己的函数，那么Windows程序将变得非常庞大。Windows的发展要求允许同时运行的几个程序共享一组函数的单一拷贝。动态链接库就是在这种情况下出现的。动态链接库不用重复编译或链接，一旦装入内存，Dlls函数可以被系统中的任何正在运行的应用程序软件所使用，而不必再将DLLs函数的另一拷贝装入内存。 10.1.1 动态链接库的工作原理“ 动态链接 ”这几字指明了DLLs是如何工作的。对于常规的函数库，链接器从中拷贝它需要的所有库函数，并把确切的函数地址传送给调用这些函数的程序。而对于DLLs，函数储存在一个独立的动态链接库文件中。在创建Windows程序时，链接过程并不把DLLs文

件链接到程序上。直到程序运行并调用一个DLLs中的函数时，该程序才要求这个函数的地址。此时Windows才在DLLs中寻找被调用函数，并把它的地址传送给调用程序。采用这种方法，DLLs达到了复用代码的极限。 动态链接库的另一个方便之处是对动态链接库中函数的修改可以自动传播到所有调用它的程序中，而不必对程序作任何改动或处理。 DLLs不仅提供了函数重用的机制，而且提供了数据共享的机制。任何应用程序都可以共享由装入内存的DLLs管理的内存资源块。只包含共享数据的DLLs称为资源文件。如Windows的字体文件等。 10.1.2 Windows系统的动态链接库 Windows本身就是由大量的动态链接库支持的。这包括Windows API函数 (KRNLx86.EXE，USER.EXE，GDI.EXE，…)，各种驱动程序文件，各种带有.Fon和.Fot 扩展名的字体资源文件等。Windows还提供了针对某一功能的专用DLLs，如进行DDE编程的ddeml.dll，进行程序安装的ver.dll等。 虽然在编写Windows程序时必然要涉及到DLLs，但利用Delphi ，用户在大部分时候并不会注意到这一点。这一方面是因为Delphi提供了丰富的函数使用户不必直接去使用Windows API.另一方面即使使用Windows API，由于Delphi把API函数和其它Windows DLLs函数重新组织到了几个库单元中，因而也不必使用特殊的调用格式。所以本章的重点放在编写和调用用户自定义的DLLs上。 使用传统的Windows编程方法来创建和使用一个DLLs是一件很令人头痛的事，正如传统的Windows编程方法本身就令人生畏一样。用户需要对定义文件、工程文件进行一系列的修改以适应创建和使用DLLs的需要。Delphi的出现，在这一方面，正如在其它许多方面所做的那样，减轻了开发者的

负担。更令人兴奋的是Delphi利用DLLs 实现了窗体的重用机制。用户可以将自己设计好的窗体储存在一个DLLs中，在需要的时候可随时调用它。 10.2 DLLs的编写和调用 10.2.1 DLLs的编写 在Delphi环境中，编写一个DLLs同编写一个一般的应用程序并没有太大的区别。事实上作为DLLs 主体的DLL函数的编写，除了在内存、资源的管理上有所不同外，并不需要其它特别的手段。真正的区别在工程文件上。 在绝大多数情况下，用户几乎意识不到工程文件的存在，因为它一般不显示在屏幕上。如果想查看工程文件，则可以打开View菜单选择Project Source项，此时工程文件的代码就会出现在屏幕的Code Editor(代码编辑器)中。 一般工程文件的格式为： program 工程标题. uses 子句. 程序体 而DLLs工程文件的格式为： library 工程标题. uses 子句. exprots 子句. 程序体 它们主要的区别有两点： 1.一般工程文件的头标用program关键字，而DLLs工程文件头标用library 关键字。不同的关键字通知编译器生成不同的可执行文件。用program关键字生成的是.exe文件，而用library关键字生成的是.dll文件； 2.假如DLLs要输出供其它应用程序使用的函数或过程，则必须将这些函数或过程列在exports子句中。而这些函数或过程本身必须用export编译指令进行编译。 根据DLLs完成的功能，我们把DLLs分为如下的三类： 1.完成一般功能的DLLs； 2.用于数据交换的DLLs； 3.用于窗体重用的DLLs。 这一节我们只讨论完成一般功能的DLLs，其它内容将在后边的两节中讨论。 10.2.1.1 编写一般DLLs的步骤 编写一般DLLs的步骤如下： 1.利用Delphi的应用程序模板，建立一个DLLs程序框架。 对于Delphi 1.0的用户，由于没有DLLs模板，因此： (1).建立一个一般的应用

程序，并打开工程文件；(2).移去窗体和相应的代码单元；(3).在工程文件中，把program改成library，移去Uses子句中的Forms，并添加适当的库单元（一般SysUtils、Classes是需要的），删去begin...end之间的所有代码。 2.以适当的文件名保持文件，此时library后跟的库名自动修改； 3.输入过程、函数代码。如果过程、函数准备供其它应用程序调用，则在过程、函数头后加上export 编译指示； 4.建立exports子句，包含供其它应用程序调用的函数和过程名。可以利用标准指示 name、Index、resident以方便和加速过程/函数的调用； 5.输入库初始化代码。这一步是可选的； 6.编译程序，生成动态链接库文件。 10.2.1.2 动态链接库中的标准指示 在动态链接库的输出部分，用到了三个标准指示：name、Index、resident。 1.name name后面接一个字符串常量，作为该过程或函数的输出名。如： exports InStr name MyInstr. 其它应用程序将用新名字(MyInstr)调用该过程或函数。如果仍利用原来的名字(InStr)，则在程序执行到引用点时会引发一个系统错误。 2.Index Index指示为过程或函数分配一个顺序号。如果不使用Index指示，则由编译器按顺序进行分配。 Index后所接数字的范围为1...32767。使用Index可以加速调用过程。 3.resident 使用resident，则当DLLs装入时特定的输出信息始终保持在内存中。这样当其它应用程序调用该过程时，可以比利用名字扫描DLL入口降低时间开销。 对于那些其它应用程序常常要调用的过程或函数，使用resident指示是合适的。例如： exports InStr name MyInStr resident. 10.2.1.3 DLLs中的变量和段 一个DLLs拥有自己的数据段(DS)，因而它声明的任何变量都为自己所私有。调用它的模块不能直接使用它定义的变量。要

使用必须通过过程或函数界面才能完成。而对DLLs来说，它永远都没有机会使用调用它的模块中声明的变量。 一个DLLs没有自己的堆栈段(SS)，它使用调用它的应用程序的堆栈。因此在DLL中的过程、函数绝对不要假定DS = SS。一些语言在小模式编译下有这种假设，但使用Delphi可以避免这种情况。Delphi绝不会产生假定DS = SS的代码，Delphi的任何运行时间库过程/函数也都不作这种假定。需注意的是如果读者想嵌入汇编语言代码，绝不要使SS和DS登录同一个值。 10.2.1.4 DLLs中的运行时间错和处理 由于DLLs无法控制应用程序的运行，导致很难进行异常处理，因此编写DLLs时要十分小心，以确保被调用时能正常执行 。当DLLs中发生一个运行时间错时，相应DLLs并不一定从内存中移去（因为此时其它应用程序可能正在用它），而调用DLLs的程序异常中止。这样造成的问题是当DLLs已被修改，重新进行调用时，内存中保留的仍然可能是以前的版本，修改后的程序并没有得到验证。对于这个问题，有以下两种解决方法： 1.在程序的异常处理部分显式将DLL卸出内存； 2.完全退出Windows，而后重新启动，运行相应的程序。 同一般的应用程序相比，DLL中运行时间错的处理是很困难的，而造成的后果也更为严重。因此要求程序设计者在编写代码时要有充分、周到的考虑。 10.2.1.5 库初始化代码的编写 传统Windows中动态链接库的编写，需要两个标准函数：LibMain和WEP，用于启动和关闭DLL。在LibMain中，可以执行开锁DLL数据段、分配内存、初始化变量等初始化工作；而WEP在从内存中移去DLLs前被调用，一般用于进行必要的清理工作，如释放内存等。Delphi用自己特有的方式实现了这两个标准函数的功能。这就是在工程

文件中的begin...end部分添加初始化代码。和传统Windows编程方法相比，它的主要特色是： 1.初始化代码是可选的。一些必要的工作（如开锁数据段）可以由系统自动完成。所以大部分情况下用户不会涉及到； 2.可以设置多个退出过程，退出时按顺序依次被调用； 3.LibMain和WEP对用户透明，由系统自动调用。 初始化代码完成的主要工作是： 1.初始化变量、分配全局内存块、登录窗口对象等初始化工作。
在(10.3.2)节“ 利用DLLs实现应用程序间的数据传输 ”中，用于数据共享的全局内存块就是在初始化代码中分配的。 2.设置DLLs退出时的执行过程。Delphi有一个预定义变量ExitProc用于指向退出过程的地址。用户可以把自己的过程名赋给ExitProc。系统自动调用WEP函数，把ExitProc指向的地址依次赋给WEP执行，直到ExitProc为nil。 下边的一段程序包含一个退出过程和一段初始化代码，用来说明如何正确设置退出过程。 library Test. {$S-} uses WinTypes, WinProcs. var SaveExit: Pointer. procedure LibExit. far. begin if ExitCode = wep_System_Exit then begin { 系统关闭时的相应处理 } end else begin { DLL卸出时的相应处理 } end. ExitProc := SaveExit. { 恢复原来的退出过程指针 } end. begin {DLL的初始化工作 } SaveExit := ExitProc. { 保存原来的退出过程指针 } ExitProc := @LibExit. { 安装新的退出过程 } end. 在初始化代码中，首先把原来的退出过程指针保存到一个变量中，而后再把新的退出过程地址赋给ExitProc。而在自定义退出过程LibExit结束时再把ExitProc的值恢复。由于ExitProc是一个系统全局变量，所以在结束时恢复原来的退出过程是必要的。 退出过程LibExit中使用了一个系统定义变量ExitCode，用于标志退出时的状态。 ExitCode

的取值与意义如下： 表10.1 ExitCode的取值与意义

取 值	意 义
WEP_System_Exit	Windows关闭
WEP_Free_DLLx	DLLs被卸出
退出过程编	

译时必须关闭stack_checking，因而需设置编译指示{$S-}。