China-pub.com
下载

第9章　动态链接库

本章内容:

- 究竟什么是DLL
- 静态链接与动态链接
- 为什么使用DLL
- 创建和使用DLL
- 显示DLL中的模式窗体
- 在Delphi应用程序中使用DLL
- DLL的入口和出口函数
- DLL中的异常
- 回调函数
- 从DLL中调用回调函数
- 在不同的过程中共享DLL数据
- 引出DLL中的对象

本章讨论了Win32动态链接库，也就是DLL。DLL是用来编写Windows应用程序的关键组成部分。本章讨论了使用和创建DLL的几个方面，它给出了DLL怎样工作的概述并讨论了怎样创建和使用DLL，你将学会怎样调入DLL和链接由它们引出的过程和函数的不同方法。本章还包括回调函数的使用并举例说明在不同调用进程中如何实现共享数据。

9.1　究竟什么是DLL

动态链接库是程序模块，它包括代码、数据或资源，能够被其他的 Windows应用程序共享。DLL的主要特点之一是应用程序可以在运行时调入代码执行，而不是在编译时链接代码，因此，多个应用程序可以共享同一个DLL的代码。事实上，文件Kernel32.dll、User32.dll、GDI32.dll就是核心Win32系统的动态链接库。Kernel.dll负责内存、进程和线程的管理。USER32.DLL包含了一些程序，是创建窗口和处理Win32消息的用户接口。 GDI32.DLL负责处理图形。你还会听说其他的系统 DLL，譬如AdvAPI32.dll和ComDlg32.dll，它们分别处理对象安全性/注册操作和通用对话框。

使用动态链接库的另一个特点是有利于应用程序的模块化。这样就简化了应用程序的修改，因为一般只需要修改DLL，而不是整个应用程序。 Windows环境自身就是模块化类型的典型实例。每当安装一个新设备，就安装一个设备驱动程序(即DLL)，使设备能够与Windows相互通信。

在磁盘上，一个DLL基本上类似于一个Windows可执行文件(*.EXE)。一个主要的区别是,DLL不是一个独立的可执行文件，尽管它可能包含了可执行代码。大部分 DLL文件的扩展名是.dll，也有的可能是.drv(设备驱动程序)、.sys(系统文件)、.fon(字体文件)，这些不包含可执行代码。

注意　Delphi引入了一种叫做程序包的特殊用途的DLL，它应用于Delphi和C++编程环境。我们将在第21章深入探究程序包。

DLL通过动态链接技术(dynamic linking)与其他应用程序共享代码，这将在本章后面部分讨论。总之，当一个应用程序使用了一个 DLL，Win32系统会确保内存中只有一个该 DLL的拷贝，这是通过内

China-pub.com
下载

存映射文件来实现的。DLL首先被调入Win32的全局堆，然后映射到调用这个DLL进程的地址空间。在Win32系统中，每个进程都被分配有自己的32位线性地址空间。当一个DLL被多个进程调用时，每个进程都会获得该DLL的一份映像。因此，在16位Windows中，程序代码、数据、资源不被进程共享，而在Win32中，DLL是可以被看作是属于调用该DLL进程自己的代码。为得到关于Win32概念的更多信息，请参阅第3章“Win32 API”。

但这并不意味着，如果多进程调用一个DLL，物理内存就分配有该DLL的每个实例。通过从系统的全局堆到调用该DLL的每一进程的地址空间的映射，DLL映像置于每个进程的地址空间。至少在理想情况下应这样。

设置DLL的首选基地址

如果DLL被调入进程的地址空间时设置了基地址，这样DLL数据就可以被共享。如果DLL的基地址与已经分配的DLL地址重叠的话，Win32重新分配基地址。这样，每一个重新分配的DLL实例都有自己的物理上的内存空间和交换文件空间。

这是很关键的，通过使用$IMAGEBASE指示符，给每个DLL都设置一个基地址，这样不会引起冲突或不会出现地址重叠。

如果有多个应用程序都调用同一个DLL，设置一个唯一的基地址，这样无论是在进程的低端地址或者是在一般的DLL(如VCL包)的高端地址，都不会引起冲突。一般可执行文件(EXE和DLL)缺省的基地址为$400000，这就意味着，除非修改DLL的基地址，否则就会与主程序的基地址引起冲突，因此进程间也就不能共享DLL的数据。

在调用时，DLL不需要重新分配或安装，因为它保存在本地磁盘上，DLL的内存页面被直接映射到磁盘上的DLL文件。DLL代码不需占用系统页面文件(也叫交换文件)的空间。这就是为什么系统提交页的总数和大小可能比系统交换文件加内存要大。

你可以参阅Delphi 5的在线帮助中的“Image Base Address”，那里有使用$IMAGEBASE指示符的详细介绍。

有关DLL的一些术语如下：

- 应用程序，一个扩展名为.exe的Windows程序。
- 可执行文件，一个包含可执行代码的文件，它包括.dll文件和.exe文件。
- 实例，当提到应用程序和DLL时，在内存中出现的可执行文件就是实例。Win32系统通过实例句柄的方式来引用实例。例如，如果一个应用程序运行两次，就会有应用程序的两个实例，同时就有两个实例句柄。当一个DLL被调入时，实例及其相应的实例句柄同时产生。应该注意的是，这里所提的实例与类的实例不能混淆。
- 模块，在32位Windows系统中，模块和实例可以说是同义的。而在16位的Windows系统中，是建立一个模块数据库来管理模块的，一个模块对应一个模块句柄。在Win32中，应用程序的每一个实例都拥有自己的地址空间；所以，没有必要为模块单独指定标识符。不过，微软仍然保留了它自己的术语。注意一点，模块和实例是同一个概念。
- 任务，Windows是一个多任务(或任务切换)环境，所以它必须能够为运行的多个实例合理分配系统资源和时间。于是，Windows建立一个任务数据库，这个数据库包括任务的实例句柄和其他必要信息，以此实现任务切换功能。任务是Windows用来管理和分配资源与时间段的重要元素。

9.2 静态链接与动态链接

静态链接是指Delphi编译器把要调用的函数和过程编译成可执行代码。函数的代码可存留在应用程序的.dpr文件或一单元中。当链接用户的应用程序时，这些函数与过程便成为最终的可执行文件的

China-pub.com 下载

一部分。也就是说，函数和过程都在程序的.exe文件中。

程序运行时，函数和过程随程序一起调入内存，它们的位置与程序的位置是相关的。当主程序需要调用函数或过程时，流程将跳转到函数或过程所在的位置，执行完函数或过程的代码，将返回主程序调用位置。而函数或过程的相对位置，在链接时就已经确定了。

以上是对Delphi编译器进行静态链接这一复杂过程的简单描述。不过，本书并不是要你了解编译器在背后的具体操作。

注意 Delphi实现一个智能链接器，可以自动地把项目中没有引用的函数、过程和有类型的常量去掉，那么，最后的可执行文件就不会有冗余的代码。

假设有两个应用程序，都要调用一个单元的同一个函数，当然，这两个应用程序都要在其 uses子句中包含该单元。如果这两个程序要同时运行，那么内存中就存在两份该函数，如果还有第三个这样的应用程序，内存中就会有第三份该函数的实例，这样，就会三次占据内存。这个小例子就表明了动态链接的优越性之一。函数通过动态链接，被放到一个 DLL中。那么如果一个应用程序把该函数调入内存，其他应用程序就可以通过映射 DLL的映像到自己进程内存空间来共享代码。理论上讲，最终结果是内存中只存在该DLL的一份实例。

对于动态链接，在程序运行时，通过引用一个外部函数 (该函数包含在DLL中)而将该函数链接到可执行文件中。其中的引用可以在应用程序中声明，但是通常情况下是放在一个专门的引入 (import)单元里，在这个单元里可以声明引入的函数、过程以及 DLL所需的多种类型的定义。

例如，假设有一个叫MaxLib.dll的动态链接库，其中包含一个函数：

```
function Max(i1, I2: integer): integer;
```

这个函数返回两个整数中较大的一个数，一个典型的引入单元如下：

```
unit MaxUnit;
interface
function Max(I1, I2: integer): integer;
implementation
function Max; external 'MAXLIB';
end.
```

你也许注意到了，这看上去类似于一般的单元，但这个单元没有定义 Max()函数。关键字external后面的字符串就是该DLL的名称。要使用这个单元，应用程序只需把MaxUnit加到它的uses子句中即可。当这个程序运行时，该 DLL就会自动地被调入内存，并且任何需要调用 Max()的程序都被链接到这个DLL中的Max()函数。

调用DLL有两种方式，这是其中一种，叫隐式调用，就是让 Windows在应用程序调入时自动地调入所要调用的DLL；另一种是显式调用，这个将在本章节的后面讨论。

9.3 为什么要使用DLL

使用DLL有若干理由，其中有一些前面已经提到过了。大体说来，使用动态链接库可以共享代码、系统资源，可以隐藏实现的代码或底层的系统例程、设计自定义控件。下面将分别讨论这几个方面的内容。

9.3.1 共享代码、资源和数据

在本章节前面已经提到，共享代码是创建动态链接库的主要目的所在。但与单元的代码共享不同，DLL的代码可以被任何Windows应用程序共享，而单元代码的共享局限于Delphi应用程序。

另外，DLL提供了共享资源的途径，诸如位图、字体、图标等等这些都可以放到一个资源文件中，并直接链接到应用程序。如果把这些资源放到 DLL中，那么就可以让许多应用程序使用，而不必在内

China-pub.com
下载

存里重复装入这些资源。

在16位的Windows中，DLL有自己的数据段，于是，所有要调用同一个 DLL的应用程序能够访问同一个全局变量和静态变量。但在 Win32系统中，这就不同了。因为 DLL的映像被映射到每个进程的地址空间，该DLL的所有数据属于映射到的进程。值得一说的是，尽管进程间不能共享 DLL的数据，但是同一个进程的所有线程可以共享，因为线程是相互独立的，所以在访问某一 DLL的全局变量时，务必小心，防止引起冲突。

但这并不意味着没有办法实现在进程间共享一个 DLL的数据。一个技术可以通过内存映射文件的方法在内存中创建一个共享的区域。一切需调用 DLL的应用程序都可以读这些存储在内存中的共享区域的数据。这个技术将在本章的后面详细介绍。

9.3.2 隐藏实现的细节

有些时候，你可能想隐藏例程实现的细节， DLL就可以实现这一点。不管为何要隐藏你的代码，DLL可以使函数被应用程序访问，而其中的代码细节不被显现，你所要做的只是提供别人能访问DLL的接口。你也许认为Delphi的编译单元(DCU)也可以隐藏细节，但是DCU只适用于Delphi应用程序，而且还受版本的局限。而DLL与语言无关，所以，创建的DLL可以被C++、VB或其他任何支持DLL的语言调用。

Windows单元是Win32 DLL的接口单元。Delphi 5提供了Win32 API的源文件，其一是Windows单元的源文件windows.pas，在该文件的interface部分有如下定义：

```
function ClientToScreen(Hwnd: HWND; var lpPoint: TPoint): BOOL; stdcall;
```

为链接到相应的DLL，在其implementation部分，有如下的例子：

```
function ClientToScreen; external user32 name 'ClientToScreen';
```

这行代码的意思表明，ClientToSreen()在动态链接库User32.dll中,它的名称叫ClientToSreen.

9.3.3 自定义控件

自定义控件通常放在DLL中。这些控件不同于Delphi的自定义组件。自定义控件是在 Windows下注册，并且可以在任何Windows开发环境中使用。将这些类型的自定义控件加进 DLL中，是考虑到即使有多个应用程序要使用这些自定义控件，内存中也只有该控件的一份实例。

注意 其实将自定义控件加进DLL这种机制已经过时，现在，微软使用OLE和ActiveX控件，自定义控件已很少见了。

9.4 创建和使用DLL

下面是在Delphi中创建一个DLL的全部过程，你将看到怎样创建一个接口单元，使之可以被其他的应用程序访问。并且将学会怎样把Delphi的窗体加进DLL中。

9.4.1 数美分：一个简单的DLL

下面是包含一个例程的DLL例子。该例程是将以美分计算的货币换算成五分镍币、一角硬币的数目。

1. 一个简单的DLL

该DLL中包含PenniesToCoins()函数，清单9-1完整地显示了该DLL项目文件代码。

清单9-1 一个把美分转换为硬币的DLL

```
library PenniesLib;
{$DEFINE PENNIESLIB}
```

China-pub.com 下载

```
uses
  SysUtils,
  Classes,
  PenniesInt;

function PenniesToCoins(TotPennies: word;
   CoinsRec: PCoinsRec): word; StdCall;
begin
  Result := TotPennies;  // 结果存放于Result
  { 计算Quarters、Dimes、Nickels、Pennies的值 }
  with CoinsRec^ do
  begin
    Quarters    := TotPennies div 25;
    TotPennies  := TotPennies - Quarters * 25;
    Dimes       := TotPennies div 10;
    TotPennies  := TotPennies - Dimes * 10;
    Nickels     := TotPennies div 5;
    TotPennies  := TotPennies - Nickels * 5;
    Pennies     := TotPennies;
  end;
end;

{ 引出函数名 }
exports
  PenniesToCoins;
end.
```

注意，该DLL使用了PenniesInt单元，这将在后面详细介绍。

在Exports子句引出了DLL中应用程序要调用的函数或过程。

2. 定义接口单元

接口单元通过把引入单元的名字加到 uses子句中，实现调用 DLL的应用程序能够静态地引入 DLL的例程。接口单元也允许定义成 DLL和调用应用程序都能使用的公共结构。下面就有一个接口单元论证了这点。清单9-2列出了PenniesInt.pas的源代码。

清单9-2 PenniesInt.pas，PenniesLib.dll的接口单元

```
unit PenniesInt;
{ PENNIES.DLL的接口例程 }

interface
type

  { 这个记录将保存转换后的货币数 }

  PCoinsRec = ^TCoinsRec;
  TCoinsRec = record
    Quarters,
    Dimes,
    Nickels,
    Pennies: word;
  end;

{$IFNDEF PENNIESLIB}
{ 由关键词export声明函数 }

function PenniesToCoins(TotPennies: word;
  CoinsRec: PCoinsRec): word; StdCall;
```

China-pub.com
下载


```
{$ENDIF}

implementation

{$IFNDEF PENNIESLIB}
{ 定义引入的函数 }
function PenniesToCoins; external 'PENNIESLIB.DLL' name 'PenniesToCoins';
{$ENDIF}

end.
```

在这个项目中的type部分，声明了一个叫TCoinsRec的记录以及指向该记录的指针。这个记录保存传递给Pennies To Coins()的货币数转换后的结果。函数PenniesToCoins()带有两个传递参数：以美分为单位的货币数和指向TCoinsRec变量的指针。函数的返回值是换算后的货币数。

PenniesInt.pas在其接口单元中声明了函数，该函数要从DLL中引出。PenniesToCoins()函数的定义放在implementation部分，这个定义指明该函数是存在于 DLL文件(PenniesLib.dll)中的一个外部函数。链接是按函数的名称进行的。请注意：这里用了一条编译指令 PENNIESLIB，用于有条件地编译PenniesToCoins()函数的声明。这样做是因为在编译该接口单元时，对 DLL来说，是没有必要编译函数的声明。这样，DLL和调用DLL的应用程序就可以共享接口单元，如果要改变二者使用的结构，只需修改接口单元。

提示 要定义一个应用程序范围的条件指令，可以在Options对话框Directories/Conditionals页上指定该条件。值得注意的是，为使条件指令有效，必须重新编译程序项目，这是因为 make逻辑中需要加入条件定义。

注意 下面定义是引入一个DLL例程的两种方法之一：

```
function PenniesToCoins; external 'PENNIESLIB.DLL' index 1;
```

这种方法称为按序号引入，另一种是按名称引入，如下：

```
function PenniesToCoins; external 'PENNIESLIB.DLL' name 'PenniesToCoins';
```

按名称引入时，紧跟在关键字name后面的标识符就是例程在DLL中的名称。

因为按序号引入例程不必在DLL的名称表中查找，所以减少了DLL的调入时间。然而，在Win32中，最好的方法不使用这个方法，而是按名称引入例程。因为当DLL有所改动时，应用程序不必理睬DLL整体序号的变动。如果按序号引入，就捆绑到DLL的具体位置；如果按名称引入，则捆绑到例程名，而不必管它在DLL中的位置。

如果把上面的DLL共享出去，就必须向你的用户提供 PenniesLib.dll和PenniesInt.pas。这样，就使他们可以通过定义这里有 PenniesLib.dll所要求在PenniesInt.pas中定义的类型和例程使用该DLL。再者如果程序员使用的是其他语言，譬如 C++,这就需要将PenniesInt.pas转换为他所使用的语言，这样，在这些开发环境下才可以使用该DLL。本书附带的CD-ROM上有个使用PenniesLib.dll动态链接库的项目例子，你可以参阅。

9.4.2 显示DLL中的模式窗体

在这一节，你将看到怎样显示DLL中的模式窗体。把窗体放到DLL的优点是可以扩展其使用范围，DLL中的窗体可以被其他任何Windows应用程序或其他开发环境(譬如C++和Visual Basic)使用。

要把窗体放到DLL中，auto-created列表中不得有该窗体。

我们在前面已经创建了一个窗体，该窗体包含了一个 TCalendar组件。应用程序调用该窗体，是通过调用DLL中的例程。当用户选择某一日期时，该日期将被返回给应用程序。

China-pub.com 下载

清单9-3列出了DLL项目文件CalendarLib.dpr的源代码。“显示DLL中的模式窗体”这一部分列出的清单9-4是DllFrm.pas的源代码，它演示如何将窗体加进DLL中。

清单9-3 CalendarLib.dpr的源代码

```
unit DLLFrm;

interface

uses
  SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls,
  Forms, Dialogs, Grids, Calendar;

type

  TDLLForm = class(TForm)
    calDllCalendar: TCalendar;
    procedure calDllCalendarDblClick(Sender: TObject);
  end;

{ 声明要引出的函数 }
function ShowCalendar(AHandle: THandle; ACaption: String):
  TDateTime; StdCall;

implementation
{$R *.DFM}

function ShowCalendar(AHandle: THandle; ACaption: String): TDateTime;
var
  DLLForm: TDllForm;
begin
  // 复制应用程序的句柄给DLL的TApplication 对象
  Application.Handle := AHandle;
  DLLForm := TDLLForm.Create(Application);
  try
    DLLForm.Caption := ACaption;
    DLLForm.ShowModal;
    // 结果被返回
    Result := DLLForm.calDLLCalendar.CalendarDate;
  finally
    DLLForm.Free;
  end;
end;

procedure TDLLForm.calDllCalendarDblClick(Sender: TObject);
begin
  Close;
end;

end.
```

这个DLL的主窗体包含在引出函数里。注意：该 DLLForm的变量声明不是在interface部分，而是在函数ShowCalendar()内部。

该DLL函数首先将AHandle参数的值赋给Application.Handle属性。在第4章的一个Delphi项目里，讲到过一个全局对象叫 Application。DLL中的Application对象与调用它的应用程序是分离的，为使DLL中的窗体真正成为应用程序模式窗体，必须将应用程序的句柄赋给 DLL的Application.Handle属性，正如上面代码所述。要是不这样做，结果难以预料，尤其是当想最小化该 DLL的窗体时，必须确保赋

China-pub.com
下载

给Application.Handle属性的AHandle值不为nil。

窗体创建后，参数ACaption的值被赋给窗体的Caption属性，这样该窗体就可以作为模式窗体打开。当关闭该窗体时，用户在TCalendar组件里选择的日期返回给调用函数。双击窗体上的 TCalendar组件，该窗体就被关闭。

警告 如果DLL中的导出函数或过程以字符串或动态数组作为参数或返回值，那么 ShareMem必须是DLL和项目的uses子句中的第一单元。这应用于应用程序和 DLL的一切字符串的传递，甚至隐含在记录和类中的字符串。 ShareMem是共享的内存管理器 Borlndmm.dll的接口单元，Borlndmm.dll必须与DLL一起发布。要避免使用Borlndmm.dll，就得用PCar或者ShortString来传递字符串信息。

只有当模块间传递字符串和动态数组并且需要传递内存的隶属关系时，才需要 ShareMem单元。将一个内部字符串强制类型转换为PChar并将其作为PChar传递给另一个模块时，不传递字符串内存的隶属关系到调用模块，那么就不需要ShareMem单元了。

注意ShareMem单元只应用于Delphi/BCB DLL之间或其与EXE之间以字符串或动态数组为传递参数时使用。反之，如果是在非 Delphi的DLL或宿主应用程序之间的话，也就不需要ShareMem单元了。

还有由于包与包之间的共享内存分配器是隐含的，所以包之间也不需要ShareMem。

这是将模式窗体加到DLL中所要求的。下一节，将讨论怎样显示DLL中的无模式窗体。

9.5 显示DLL中的无模式窗体

为了讨论将无模式窗体加到DLL中，下面就以前一节中包含日历的窗体为例。

要显示DLL中的无模式窗体， DLL必须提供两个例程。一个创建和显示窗体，一个释放窗体。清单9-4列出了含有无模式窗体的DLL的源代码。

清单9-4 DLL中的无模式窗体

```
unit DLLFrm;

interface

uses
  SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls,
  Forms, Dialogs, Grids, Calendar;

type

  TDLLForm = class(TForm)
    calDllCalendar: TCalendar;
  end;

{ Declare the export function }
function ShowCalendar(AHandle: THandle; ACaption: String):
  Longint; stdCall;
procedure CloseCalendar(AFormRef: Longint); stdcall;


implementation
{$R *.DFM}
```

China-pub.com 下载

```
function ShowCalendar(AHandle: THandle; ACaption: String): Longint;
var
  DLLForm: TDllForm;
begin
  // Copy application handle to DLL's TApplication object
  Application.Handle := AHandle;
  DLLForm := TDLLForm.Create(Application);
  Result := Longint(DLLForm);
  DLLForm.Caption := ACaption;
  DLLForm.Show;
end;

procedure CloseCalendar(AFormRef: Longint);
begin
  if AFormRef > 0 then
    TDLLForm(AFormRef).Release;
end;

end.
```

上述清单包括ShowCalendar()和CloseCalendar()例程。ShowCalendar()跟清单9-3里的ShowCalendar()函数相似，都是把应用程序的句柄赋给Application.Handle属性后才创建窗体。只不过，这里调用的是Show()，而不是ShowModal()。请注意：ShowModal()不释放窗体，同时，该函数返回一个长整型的值，将这个值赋给DLL Form实例。这是因为已创建的窗体的引用必需维护，最好是由调用的应用程序来管理这个实例，这样就可以防止创建窗体的多个实例。

CloseCalendar()过程比较简单，首先判断引用是否有效，如果有效，就调用 Release()方法来释放它。在这里，应用程序应该将ShowCalendar()的返回值传递给该过程。

请务必小心，不可以随便释放该窗体，只有调用的应用程序才能释放，否则，调用CloseCalendar()将会失败。

本书附带的光盘，有模式窗体和无模式窗体的例子，请参阅。

9.6 在Delphi应用程序中使用DLL

前面章节说到调用DLL有两种方式：隐式和显式。下面就以刚创建的 DLL为例，来介绍这两种方式。

本章创建的第一个DLL包含一个接口单元。下面就用该接口单元来隐式链接 DLL。这个工程的主窗体上有一个TMadkEdit、一个TButton和九个TLabel。

在这个应用程序里，用户输入美分的数目，然后单击按钮，标签上将按条目显示其结果。这结果是来源于PenniesLib.dll中引出的例程PenniesToCoins()。

主窗体是在MainFrm.pas单元里定义的，代码见清单9-5。

清单9-5 Pennies 程序的主窗体

```
unit MainFrm;

interface

uses
  SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls,
  Forms, Dialogs, StdCtrls, Mask;
```

China-pub.com
下载


```
type

  TMainForm = class(TForm)
    lblTotal: TLabel;
    lblQlbl: TLabel;
    lblDlbl: TLabel;
    lblNlbl: TLabel;
    lblPlbl: TLabel;
    lblQuarters: TLabel;
    lblDimes: TLabel;
    lblNickels: TLabel;
    lblPennies: TLabel;
    btnMakeChange: TButton;
    meTotalPennies: TMaskEdit;
    procedure btnMakeChangeClick(Sender: TObject);
  end;

var
  MainForm: TMainForm;

implementation
uses PenniesInt;  // Use an interface unit

{$R *.DFM}

procedure TMainForm.btnMakeChangeClick(Sender: TObject);
var
  CoinsRec: TCoinsRec;
  TotPennies: word;
begin
  { Call the DLL function to determine the minimum coins required
    for the amount of pennies specified. }
  TotPennies := PenniesToCoins(StrToInt(meTotalPennies.Text), @CoinsRec);
  with CoinsRec do
  begin
    { Now display the coin information }
    lblQuarters.Caption := IntToStr(Quarters);
    lblDimes.Caption    := IntToStr(Dimes);
    lblNickels.Caption  := IntToStr(Nickels);
    lblPennies.Caption  := IntToStr(Pennies);
  end
end;

end.
```

注意MainFrm.pas使用了接口单元PenniesInt。PenniesInt.pas包括了PenniesLib.dpr中的函数的外部声明。当应用程序运行时，Win32系统会自动地调入PenniesLib.dll，并将其映射到该应用程序的进程地址空间。

import接口单元其实是可选的，可以不用PenniesInt单元，而在MainFrm.pas的 implementation部分外部声明PenniesToCoins()函数即可，代码如下：

```
implementation

function PenniesToCoins(TotPennies: word; ChangeRec: PChangeRec): word;
```

China-pub.com 下载

```
  ➥StdCall external 'PENNIESLIB.DLL';
```

不过，在MainFrm.pas还要再次定义PChangeRec 和TChangeRec，要不然也可以使用编译指令PENNIESLIB编译你的应用程序。在只需要访问DLL中的少数例程时，这种技术较适合。不过，大多数情况是既要访问DLL中的例程，又要引用DLL中定义的数据类型，这时最好使用接口单元。

注意 有些时候，需要调用其他软件厂商的DLL，这时就不会有Pascal的接口单元，不过，也许有C/C++的引入库。如果是这样的话，就只有将该库转换为等效的Pascal接口单元。

你可以在附带的光盘上找到该演示过程。

显式调用

尽管隐式调用很方便，但不是最理想的方法。假使有一个包含多个例程的 DLL，可能这些例程却用不着，这样的话，调入该DLL显然浪费了内存。尤其是一个应用程序需要使用多个 DLL时，这就更浪费了。另一种情形是，一个多版本的标准函数，如一组打印驱动程序，分别是由多个 DLL来实现的。在这种情形下，如果能需要哪一个版本就调用其DLL，就最好不过了。这种方式就是显式调用。

为了阐明显式调用DLL，现在回到那个模式窗体的DLL。清单9-6列出了一个应用程序的主窗体的代码，演示了显式调用DLL。该应用程序的项目文件在附带的光盘上。

清单9-6 日历DLL的主窗体单元

```
unit MainFfm;

interface

uses
  SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls,
  Forms, Dialogs, StdCtrls;

type
  { First, define a procedural data type, this should reflect the
    procedure that is exported from the DLL. }
  TShowCalendar = function (AHandle: THandle; ACaption: String):
    TDateTime; StdCall;

  { Create a new exception class to reflect a failed DLL load }
  EDLLLoadError = class(Exception);

  TMainForm = class(TForm)
    lblDate: TLabel;
    btnGetCalendar: TButton;
    procedure btnGetCalendarClick(Sender: TObject);
  end;

var
  MainForm: TMainForm;

implementation

{$R *.DFM}

procedure TMainForm.btnGetCalendarClick(Sender: TObject);
```

China-pub.com
下载


```
var
  LibHandle   : THandle;
  ShowCalendar: TShowCalendar;
begin

  { Attempt to load the DLL }
  LibHandle := LoadLibrary('CALENDARLIB.DLL');
  try
    { If the load failed, LibHandle will be zero.
      If this occurs, raise an exception. }
    if LibHandle = 0 then
      raise EDLLLoadError.Create('Unable to Load DLL');
    { If the code makes it here, the DLL loaded successfully, now obtain
      the link to the DLL's exported function so that it can be called. }
    @ShowCalendar := GetProcAddress(LibHandle, 'ShowCalendar');
    { If the function is imported successfully, then set
      lblDate.Caption to reflect the returned date from
      the function. Otherwise, show the return raise an exception. }
    if not (@ShowCalendar = nil) then
      lblDate.Caption := DateToStr(ShowCalendar(Application.Handle, Caption))
    else
      RaiseLastWin32Error;
  finally
    FreeLibrary(LibHandle); // Unload the DLL.
  end;
end;

end.
```

上面这个单元首先定义了一个过程数据类型TShowCalendar,该类型就是CalendarLib.dll中的函数类型。接着，又声明一个特殊的异常类，当调用失败时，就会引发这个异常。在 btnGetCalendarClick()事件处理过程中，使用了三个Win32 API函数：LoadLibrary()、FreeLibrary()、GetProcAddress()。

LoadLibrary()声明如下：

```
function LoadLibrary(lpLibFileName: PChar): HMODULE; stdcall;
```

上述函数调入由lpLibFileName参数指定的DLL模块，并将其映射到调用进程的地址空间。如果调用成功，函数将返回该模块的句柄；若失败，返回值为 0，并触发一异常。你可以查阅在线帮助中LoadLibrry()函数的详细说明以及可能返回的错误值。

FreeLibrary()声明如下：

```
function FreeLibrary(hLibModule: HMODULE): BOOL; stdcall;
```

FreeLibrary()函数减小LibModule指定的库的实例计数。当该DLL的实例计数是零时，调用的DLL就会被释放。实例计数记录使用这个DLL的任务数。

GetProcAddress()是这样定义的：

```
function GetProcAddress(hModule: HMODULE; lpProcName: LPCSTR):
  FARPROC; stdcall
```

GetProcAddress()返回的是一个函数在模块中的地址，其中由 hModule参数指定模块。hModule是从LoadLibrary()函数返回的结果 THandle。如果GetProcAddress()调用失败，则返回 nil。你只有调用GetLastError()才能获得详细的错误信息。

在处理Button1的OnClick事件处理过程中，首先调用LoadLibrary()去调用CALDLL。如果失败，则

China-pub.com 下载

触发异常。如果成功，调用GetProcAddress()来获得函数ShowCalendar()的地址。在变量ShowCalendar前加上取址运算符@，这样就不会在编译时出现类型转换错误。得到函数ShowCalendar()的地址后，就可以按TShowCalendar定义的使用它了。最后不再使用时，一定要在 finally块中调用FreeLibrary()来释放它。

你也许明白，每次调用该函数时， DLL既要调入，又要释放。如果在应用程序运行时，只需要调用一次，显式调用的优势就很明显，它能够有效地减少内存的消耗。但是，如果该函数要被频繁地调用，那么频繁地调入和释放会增加系统负担。

9.7 DLL的入口函数和出口函数

当进程或线程初始化和终止时，可以给DLL加进入口函数和出口函数。

9.7.1 进程/线程初始化和终止例程

典型的初始化操作包括注册窗口类、初始化全局变量和初始化入口/出口函数。这是在DLLEntryPoint函数里进行的。该函数实际上在DLL项目文件的begin..end之间调用。这是应该放置入口函数和出口函数的地方。DLLEntryPoint需要一个DWord类型的参数。

全局变量DLLProc是一个过程的指针，该指针指定入口 /出口函数。该变量初始值为nil，除非建立了入口/出口函数。表9-1列出了入口/出口函数事件对应的用途。

表9-1 DLL的入口/出口事件

事 件	用 途
DLL_PROCESS_ATTACH	在进程启动或调用LoadLibrary()时，DLL映射到当前进程的地址空间。在这个事件期间，DLL初始化实例数据
DLL_PROCESS_DETACH	DLL正从进程的地址空间分离出来，这也许是进程本身退出或调用了FreeLibrary()。在该事件里，DLL也许没有初始化任何实例。
DLL_THREAD_ATTACH	进程创建了一个新线程。这时，系统会调用所有和这个进程相关联的DLL入口函数。这个调用在新线程的环境中进行，用于分配线程特定的数据
DLL_THREAD_DETACH	一个线程正在退出。在该事件里，DLL将释放线程特定的初始化数据

警告 调用TerminateThread()非正常终止线程时，没有调用DLL_THREAD_DETACH。

9.7.2 DLL入口/出口示例

清单9-7演示了怎样通过变量DLLProc来建立DLL的入口/出口函数。

清单9-7 DLLEntry.dpr的源代码

```
library DllEntry;
uses
  SysUtils,
  Windows,
  Dialogs,
  Classes;
procedure DLLEntryPoint(dwReason: DWord);
begin
  case dwReason of
    DLL_PROCESS_ATTACH: ShowMessage('Attaching to process');
```

China-pub.com
下载


```
    DLL_PROCESS_DETACH: ShowMessage('Detaching from process');
    DLL_THREAD_ATTACH:  MessageBeep(0);
    DLL_THREAD_DETACH:  MessageBeep(0);
  end;
end;
begin
  { First, assign the procedure to the DLLProc variable }
  DllProc := @DLLEntryPoint;
  { Now invoke the procedure to reflect that the DLL is attaching to the
    process }
  DLLEntryPoint(DLL_PROCESS_ATTACH);
end.
```

你也许会发现，入口/出口函数被赋给DLLProc变量是在DLL的项目文件的begin..end之间进行的。在过程DLLEntryPoint()中，检查其参数dwReason的值来判定被调用的事件。这些事件对应的用途在表9-1中列出了。当DLL被调用和删除时，分别显示一个信息框。当一个线程被创建或退出时，会发出“嘀”的一声。

为了说明该DLL的用法，清单9-8列出了一个例子的源代码。

清单9-8 DLL的入口/出口函数演示示范

```
unit MainFrm;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls,
  Forms, Dialogs, StdCtrls, ComCtrls, Gauges;

type

  { Define a TThread descendant }

TTestThread = class(TThread)
    procedure Execute; override;
    procedure SetCaptionData;
  end;

  TMainForm = class(TForm)
    btnLoadLib: TButton;
    btnFreeLib: TButton;
    btnCreateThread: TButton;
    btnFreeThread: TButton;
    lblCount: TLabel;
    procedure btnLoadLibClick(Sender: TObject);
    procedure btnFreeLibClick(Sender: TObject);
    procedure btnCreateThreadClick(Sender: TObject);
    procedure btnFreeThreadClick(Sender: TObject);
    procedure FormCreate(Sender: TObject);
  private
    LibHandle   : THandle;
    TestThread  : TTestThread;
    Counter     : Integer;
```

China-pub.com 下载

```
    GoThread    : Boolean;
  end;

var
  MainForm: TMainForm;

implementation

{$R *.DFM}

procedure TTestThread.Execute;
begin
  while MainForm.GoThread do
  begin
    Synchronize(SetCaptionData);
    Inc(MainForm.Counter);
  end;
end;

procedure TTestThread.SetCaptionData;
begin
  MainForm.lblCount.Caption := IntToStr(MainForm.Counter);
end;

procedure TMainForm.btnLoadLibClick(Sender: TObject);
{ This procedure loads the library DllEntryLib.DLL }
begin
  if LibHandle = 0 then
  begin
    LibHandle := LoadLibrary('DLLENTRYLIB.DLL');
    if LibHandle = 0 then
       raise Exception.Create('Unable to Load DLL');
   end
   else
     MessageDlg('Library already loaded', mtWarning, [mbok], 0);
end;

procedure TMainForm.btnFreeLibClick(Sender: TObject);
{ This procedure frees the library }
begin
  if not (LibHandle = 0) then
  begin
    FreeLibrary(LibHandle);
    LibHandle := 0;
  end;
end;

procedure TMainForm.btnCreateThreadClick(Sender: TObject);
{ This procedure creates the TThread instance. If the DLL is loaded a
  message beep will occur. }
begin
  if TestThread = nil then
  begin
```

China-pub.com
下载


```
    GoThread   := True;
    TestThread := TTestThread.Create(False);
  end;
end;

procedure TMainForm.btnFreeThreadClick(Sender: TObject);
{ In freeing the TThread a message beep will occur if the DLL is loaded. }
begin
  if not (TestThread = nil) then
  begin
    GoThread   := False;
    TestThread.Free;
    TestThread := nil;
    Counter    := 0;
  end;

end;
procedure TMainForm.FormCreate(Sender: TObject);
begin
  LibHandle  := 0;
  TestThread := nil;
end;

end.
```

这个程序包含的主窗体上有四个TButton组件。BtnLoadLib按钮用于调入DllEntryLib.dll。BtnFreeLib按钮用于释放该 DLL。BtnCreateThread按钮用于创建一个 TThread派生对象，它再创建线程。BtnFreeThred释放TThread对象。lblCount表明线程的运行情况。

在btnLoadLibClick()事件处理过程中，调用LoadLibrary()来调入DllEntryLib.dll。这样，DLL被调入并且映射到进程的地址空间。另外，DLL中的初始化代码得到执行。这些代码也是在 DLL的begin..end之间出现，用来建立DLL的入口/出口函数：

```
begin
  { 首先，给DLLProc变量赋值 }
  DllProc := @DLLEntryPoint;
  { 现在调用这个过程以反映DLL已映射到进程 }
    process }
  DLLEntryPoint(DLL_PROCESS_ATTACH);
end.
```

这个初始化部分将只被每个进程调用一次。如果还有别的进程需要调用它，这部分会被再调用，除非进程不共享DLL实例。

btnFreeLibClick()事件处理过程调用FreeLibrary(),释放DLL。这时将调用DLLProc指针所指的过程DLLEntryProc()，传递的参数是DLL_PROCESS_DETACH。

btnCreateThreadClick()事件处理过程创建一个TThread派生对象，这使DLLEntryProc()被调用，传递的参数为DLL_THREAD_ATTACH。btnFreeThreadClick()也调用DLLEntryProc()，传递的参数是DLL_THREAD_DETACH。

尽管这些事件发生时只是打开信息框，但在必要情况下，你可以使用这些事件对进程、线程进行初始化或者清空。后面有一个例子，用这种技术设置一个共享的全局变量。你可以在附带的光盘上看到DLLEntryTest.dpr的代码。

China-pub.com 下载

9.8 DLL中的异常

这一节将讨论关于DLL和Win32异常的问题。

9.8.1 在16位Delphi中捕捉异常

在16位的Delphi中，异常跟语言是分不开的。如果DLL触发了一个异常，必须在DLL中捕获；否则，这个异常蔓延到调用该DLL的模块栈中，从而导致该模块崩溃。所以，必须在每个DLL的入口点加一个异常处理的外套，代码如下所示：

```
procedure SomeDLLProc;
begin
  try
    { 做一些别的事情 }
  except
     on Exception do
        { 不让异常蔓延，处理掉，并且不让其再触发 }
  end;
end;
```

Delphi 2就能解决此类事件了，Delphi 5可以把异常映射为Win32的异常，这样，异常就跟编译器/语言无关了，而交由Win32系统来处理。

不过，必须保证SysUtils包含在DLL的uses子句里。否则，就不能处理DLL中的异常。

警告 大多数的Win32应用程序都没有考虑处理异常。因此，尽管Delphi异常被变成Win32的异常，但要是让异常蔓延到主程序，也可能导致主模块崩溃。

如果主程序是由Delphi或C++ Builder开发的，这将不会有问题，但是可能会多出一些C和C++的代码。

因此，要使主程序不受影响，最好采用16位的方式，在DLL的入口函数外加上try..except块以捕获DLL中的异常。

注意 当一个非Delphi应用程序调用一个Delphi编写的DLL时，Delphi特定的异常类就不能生效。不过，它会被当成Win32系统的异常处理。该异常的地址是Win32系统的EXCEPTION_RECORD的ExceptionInformation数组的第一项。第二项包含Delphi异常对象。查看附带的Delphi在线帮助EXCEPTION_RECORD。

9.8.2 异常和Safecall指示符

Safecall指示符用于COM和异常处理。如果函数用Safecall声明，就可确保任何异常都被传播给函数的调用者，并且将异常转换为HResult类型的返回值。Safecall也意味着采用StdCall调用约定。因此，Safecall函数声明如下：

```
function Foo(i: integer): string; Safecall;
```

编译器是这样看这行代码的：

```
function Foo(i: integer): string; HResult; StdCall;
```

然后编译器就会给整个函数内容加上try..except外套，并且捕获触发的任何异常。except块调用SafecallExceptionHandler()，将异常转换为HResult类型值。这个类似于16位的捕捉异常并返回错误值的方法。

9.9 回调函数

回调函数不是被应用程序调用，而是被Win32 DLL或其他DLL调用。基本上，Windows有几个API

China-pub.com
下载

函数需要调用回调函数。当调用这些函数时，传递函数的地址。也许你想知道这跟 DLL有什么关系？请记住：Win32 API的相当多的例程都是从系统DLL引出的。事实上，将一回调函数传递给Win32 API函数时，就是将其传递给DLL。

这里有一个EnumWindows() API函数，该函数列举所有的顶层窗口。该函数将传递每个窗口的句柄给回调函数。需要定义回调函数的地址并将其传递给 EnumWindows()函数。这个回调函数定义如下：

```
function EnumWindowsProc(Hw: HWnd; lp: lParam): Boolean; stdcall;
```

本书附带的光盘上有一个CallBack.dpr项目，演示了EnumWindows()函数的使用。清单9-9是其源代码。

清单9-9 回调函数示例 MainForm.pas

```
unit MainFrm;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls,
  Forms, Dialogs, StdCtrls, ComCtrls;

type

  { Define a record/class to hold the window name and class name for
    each window. Instances of this class will get added to ListBox1 }
  TWindowInfo = class
    WindowName,          // The window name
    WindowClass: String; // The window's class name
  end;

  TMainForm = class(TForm)
    lbWinInfo: TListBox;
    btnGetWinInfo: TButton;
    hdWinInfo: THeaderControl;
    procedure btnGetWinInfoClick(Sender: TObject);
    procedure FormDestroy(Sender: TObject);
    procedure lbWinInfoDrawItem(Control: TWinControl; Index: Integer;
      Rect: TRect; State: TOwnerDrawState);
    procedure hdWinInfoSectionResize(HeaderControl: THeaderControl;
      Section: THeaderSection);
  end;

var
  MainForm: TMainForm;

implementation

{$R *.DFM}
function EnumWindowsProc(Hw: HWnd; AMainForm: TMainForm):
   Boolean; stdcall;
{ This procedure is called by the User32.DLL library as it enumerates
  through windows active in the system. }
var
```

China-pub.com 下载

```
  WinName, CName: array[0..144] of char;
  WindowInfo: TWindowInfo;
begin
  { Return true by default which indicates not to stop enumerating
    through the windows }
  Result := True;
  GetWindowText(Hw, WinName, 144); // Obtain the current window text
  GetClassName(Hw, CName, 144);    // Obtain the class name of the window
  { Create a TWindowInfo instance and set its fields with the values of
    the window name and window class name. Then add this object to
    ListBox1's Objects array. These values will be displayed later by
    the listbox }
  WindowInfo := TWindowInfo.Create;
  with WindowInfo do
  begin
    SetLength(WindowName, strlen(WinName));
    SetLength(WindowClass, StrLen(CName));
    WindowName := StrPas(WinName);
    WindowClass := StrPas(CName);
  end;
  // Add to Objects array
  MainForm.lbWinInfo.Items.AddObject('', WindowInfo); end;

procedure TMainForm.btnGetWinInfoClick(Sender: TObject);
begin
  { Enumerate through all top-level windows being displayed. Pass in the
    call back function EnumWindowsProc which will be called for each
    window }
  EnumWindows(@EnumWindowsProc, 0);
end;

procedure TMainForm.FormDestroy(Sender: TObject);
var
  i: integer;
begin
  { Free all instances of TWindowInfo }
  for i := 0 to lbWinInfo.Items.Count - 1 do
    TWindowInfo(lbWinInfo.Items.Objects[i]).Free
end;
procedure TMainForm.lbWinInfoDrawItem(Control: TWinControl;
  Index: Integer;Rect: TRect; State: TOwnerDrawState);
begin
  { First, clear the rectangle to which drawing will be performed }
  lbWinInfo.Canvas.FillRect(Rect);
  { Now draw the strings of the TWindowInfo record stored at the
    Index'th position of the listbox. The sections of HeaderControl
    will give positions to which to draw each string }
  with TWindowInfo(lbWinInfo.Items.Objects[Index]) do
  begin
    DrawText(lbWinInfo.Canvas.Handle, PChar(WindowName),
      Length(WindowName), Rect,dt_Left or dt_VCenter);
    { Shift the drawing rectangle over by using the size
      HeaderControl1's sections to determine where to draw the next
```

China-pub.com
下载


```
        string }
      Rect.Left := Rect.Left + hdWinInfo.Sections[0].Width;
      DrawText(lbWinInfo.Canvas.Handle, PChar(WindowClass),
        Length(WindowClass), Rect, dt_Left or dt_VCenter);
    end;
  end;

  procedure TMainForm.hdWinInfoSectionResize(HeaderControl:
     THeaderControl; Section: THeaderSection);
  begin
    lbWinInfo.Invalidate; // Force ListBox1 to redraw itself.
  end;

  end.
```

上面这个应用程序使用EnumWindows()API函数列举所有的顶层窗口的名称和类名，并显示在列表框里。类名和名称是以分栏形式显示的。首先，我们将讨论该回调函数的用法，然后介绍怎样创建分栏列表框。

9.9.1 使用回调函数

清单9-9中，我们定义了EnumWindowsProc()过程，该过程的第一个参数传递的是窗口句柄，第二个参数是用户定义的数据，所以可以传递任何数据，只要是一个Integer类型的数。

EnumWindowsProc()是一个回调函数，可以被传递给EnumWindows() API函数。必须用StdCall指示符声明，以示遵循Win32调用约定。当EnumWindows()API调用时，对每一个顶层窗口都会调用EnumWindowsProc()一次，EnumWindowsProc()的第一个参数传递窗口的句柄。通过窗口的句柄获得每个窗口的名称和类名。然后，创建一个TWindowInfo类的实例，并用这些信息设置它的字段。将WindowInfo类的实例加进lbWinInfo.Objects数组中，列表框的数据将分栏显示。

请注意，在处理主窗体的OnDestroy事件时，必须把TWindowInfo类的实例删除。

btnGetWinInfoClick()事件处理过程调用EnumWindows()，并将EnumWindowsProc()的地址作为它的第一个参数。

当这个程序运行时，单击按钮，列表框里列出每一个窗体的名称和类名。

9.9.2 拥有者绘制的列表框

上面这个例子中，通过自定义分栏式的列表框来显示窗口的名称和类名。 TListBox组件的Style属性应该设为lbOwnerDraw。这样设置后，TListBox每画一个项时，TListBox.OnDrawItem事件就会触发。你不妨可以参照这个例子画一个项。

在清单9-9中，事件处理过程lbWinInfoDrawItem()包含绘制列表框项的代码，就是把lbWinInfo.Objects数组中的类TWindowInfo的实例中的字符串画出来。这些值是从回调函数EnumWindowsProc()中获得的。你可以参照代码中的注释判断事件处理过程的行为。

9.10 从DLL中调用回调函数

既然可以把一个回调函数传递给DLL，DLL也就可以调用回调函数。这一节讨论怎样建立一个以回调函数作为其引出函数的参数的DLL。然后，根据是否把回调函数传递给了DLL，调用该函数。清单9-10列出了这个DLL的源代码。

China-pub.com 下载

清单9-10 一个调用回调函数的演示：StrSrchLib.dll

```
library StrSrchLib;

uses
  Wintypes,
  WinProcs,
  SysUtils,
  Dialogs;

type
 { declare the callback function type }
 TFoundStrProc = procedure(StrPos: PChar); StdCall;

function SearchStr(ASrcStr, ASearchStr: PChar;  AProc: TFarProc):
  Integer; StdCall;
{ This function looks for ASearchStr in ASrcStr. When founc ASearchStr,
  the callback procedure referred to by AProc is called if one has been
  passed in. The user may pass nil as this parameter. }
var
  FindStr: PChar;
begin
  FindStr := ASrcStr;
  FindStr := StrPos(FindStr, ASearchStr);
  while FindStr <> nil do
  begin
    if AProc <> nil then
      TFoundStrProc(AProc)(FindStr);
    FindStr := FindStr + 1;
    FindStr := StrPos(FindStr, ASearchStr);
  end;
end;

exports
  SearchStr;
begin

end.
```

这个DLL定义了一个回调函数的类型即TFoundStrProc，这将在回调函数被调用时用于类型强制转换。

引出的SearchStr()是回调函数被调用的地方。清单的注释部分解释了该过程的作用。

关于该DLL的使用，可以参阅本书附带的光盘的\DLLCallBack目录中的CallBackDemo.dprd。清单9-11列出了主窗体的源代码。

清单9-11 DLL使用回调函数示例的主窗体单元

```
unit MainFrm;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls,
```

China-pub.com
下载


```
  Forms, Dialogs, StdCtrls;

type
  TMainForm = class(TForm)
    btnCallDLLFunc: TButton;
    edtSearchStr: TEdit;
    lblSrchWrd: TLabel;
    memStr: TMemo;
    procedure btnCallDLLFuncClick(Sender: TObject);
  end;

var
  MainForm: TMainForm;
  Count: Integer;

implementation

{$R *.DFM}

{ Define the DLL's exported procedure }
function SearchStr(ASrcStr, ASearchStr: PChar; AProc: TFarProc):
   Integer; StdCall external
  'STRSRCHLIB.DLL';

{ Define the callback procedure, make sure to use the StdCall directive }
procedure StrPosProc(AStrPsn: PChar); StdCall;
begin
  inc(Count); // Increment the Count variable.
end;

procedure TMainForm.btnCallDLLFuncClick(Sender: TObject);
var
  S: String;
  S2: String;
begin
   Count := 0; // Initialize Count to zero.
   { Retrieve the length of the text on which to search. }
   SetLength(S, memStr.GetTextLen);
   { Now copy the text to the variable S }
   memStr.GetTextBuf(PChar(S), memStr.GetTextLen);
   { Copy Edit1's Text to a string variable so that it can be passed to
     the DLL function }
   S2 := edtSearchStr.Text;
   { Call the DLL function }
   SearchStr(PChar(S), PChar(S2), @StrPosProc);
   { Show how many times the word occurs in the string. This has been
     stored in the Count variable which is used by the callback function }
   ShowMessage(Format('%s %s %d %s', [edtSearchStr.Text,
     'occurs', Count, 'times.']));
end;

end.
```

China-pub.com 下载

这个应用程序有一个TMemo控件。EdtSearchStr.Text中包含了要在memStr中搜索的字符串。调用SearchStr()，需要传递EdtSearchStr.Text作为搜索字符串，并传递memStr的内容作为源字符串。

StrPosProc()是实际的回调函数。该函数是将全局变量Count加1，用来统计搜索字符串出现的次数。

9.11 在不同的进程间共享DLL数据

16位的Windows与32位的Win32处理DLL内存的方式是不同的。在16位的Windows中，显然，不同的应用程序可以共享16位DLL的全局数据。也就是说，如果DLL定义了一个全局变量，所有调用这个DLL的应用程序都可以访问这个变量，而其中的一个应用程序对变量的修改，将会影响其他应用程序。

事实上，这种做法是很危险的，因为这样很容易引起冲突。

而Win32就不再共享DLL的全局变量。因为每个应用程序都是将 DLL映射到自己的地址空间，同时DLL的数据也就随之被映射。这样，每个应用程序都有自己 DLL数据实例，在一个应用程序中修改DLL中的全局变量，就不会影响其他应用程序。

如果想把一个16位的应用程序移植到32位的Win32下，而仍然以16位的共享方式来共享全局变量，这就要求以一种内存映射文件的技术来实现。将在第12章详细介绍这种技术。

9.11.1 一个可以被共享数据的DLL

清单9-12列出了一个DLL项目文件的源代码，这个DLL允许应用程序共享它的全局数据，即全局变量GlobalData。

清单9-12 一个允许共享数据的DLL：ShareLib

```
library ShareLib;

uses
  ShareMem,
  Windows,
  SysUtils,
  Classes;
const

  cMMFileName: PChar = 'SharedMapData';

{$I DLLDATA.INC}

var
  GlobalData : PGlobalDLLData;
  MapHandle  : THandle;

{ GetDLLData will be the exported DLL function }
procedure GetDLLData(var AGlobalData: PGlobalDLLData); StdCall;
begin
  { Point AGlobalData to the same memory address referred to by GlobalData. }
  AGlobalData := GlobalData;
end;

procedure OpenSharedData;
var
   Size: Integer;
```

China-pub.com
下载


```
begin
  { Get the size of the data to be mapped. }
  Size := SizeOf(TGlobalDLLData);

  { Now get a memory-mapped file object. Note the first parameter passes
    the value $FFFFFFFF or DWord(-1) so that space is allocated from
    the system's paging file. This requires that a name for the memory-mapped
    object get passed as the last parameter. }

  MapHandle := CreateFileMapping(DWord(-1), nil, PAGE_READWRITE, 0,
    Size, cMMFileName);

  if MapHandle = 0 then
    RaiseLastWin32Error;
  { Now map the data to the calling process's address space and get a
    pointer to the beginning of this address }
  GlobalData := MapViewOfFile(MapHandle, FILE_MAP_ALL_ACCESS, 0, 0, Size);
  { Initialize this data }
  GlobalData^.S := 'ShareLib';
  GlobalData^.I := 1;
  if GlobalData = nil then
  begin
    CloseHandle(MapHandle);
    RaiseLastWin32Error;
  end;
end;

procedure CloseSharedData;
{ This procedure un-maps the memory-mapped file and releases the memory-mapped
  file handle }
begin
  UnmapViewOfFile(GlobalData);
  CloseHandle(MapHandle);
end;

procedure DLLEntryPoint(dwReason: DWord);
begin
  case dwReason of
    DLL_PROCESS_ATTACH: OpenSharedData;
    DLL_PROCESS_DETACH: CloseSharedData;
  end;
end;

exports
  GetDLLData;

begin
  { First, assign the procedure to the DLLProc variable }
  DllProc := @DLLEntryPoint;
  { Now invoke the procedure to reflect that the DLL is attaching
    to the process }
  DLLEntryPoint(DLL_PROCESS_ATTACH);
end.
```

China-pub.com 下载

GlobalData类型是PGlobalDLLData，这个类型是在包含文件DLLData.inc里定义的。这个包含文件，通过编译指示符$I链接到项目文件。该文件有以下类型定义：

```
type

  PGlobalDLLData = ^TGlobalDLLData;
  TGlobalDLLData = record
    S: String[50];
    I: Integer;
  end;
```

上面这个DLL也如前面介绍的建立了入口/出口函数，即清单里的DLLEntryPoint()函数。当进程调用DLL时，就会调用OpenSharedData()。当进程与DLL分离时，就调用CloseSharedData()。

在这里，不打算详细讨论内存映射文件的用法，第 12章将会详细介绍。不过，为了更好地理解DLL用途，这里做简单介绍。

内存映射文件提供的是一种方法，就是在 Win32系统的地址空间里保留一块区域，物理存储可以向其中提交。这类似于分配内存并用指针来访问内存。不过，内存映射文件可以把磁盘上的文件映射到这个地址空间，并用指针访问该文件，就如同用指针访问内存一样。

内存映射文件必须先获得磁盘文件的句柄，然后，将内存映射对象映射到该文件。在本章的开头就介绍过，多个应用程序共享DLL的数据，首先将该DLL 调入内存，然后每个应用程序都有自己的该DLL映像，就好像都有一个该DLL的一份实例。事实上，内存中只有一份 DLL。这就是内存映射文件所做的。你可以用Win32 API来创建和访问内存映射文件。

现在假设有一个应用程序，它称之为 App1，创建了内存映射文件，它映射的磁盘文件叫 MyFile.dat。App1可以读写该文件的数据。如果在App1运行时，App2也映射到同一个磁盘文件，那么App1对该文件的修改会对App2产生影响，反之亦然。不过，事实上，肯定要复杂些，需要设置一些标志，使修改及时反馈给应用程序，这样修改对两个应用程序都产生影响。

不过，内存映射文件对象不只是磁盘文件，也可以是Win32的页面调度文件，而且后者比前者好。这意味着可以像访问一个磁盘文件那样访问内存中的一个区域，而不用创建临时的磁盘文件，用完后还得删除它。Win32系统自己管理页面调度文件，当不再需要该页面调度文件时，系统会自动将有关区域释放。

前面讲到，通过内存映射文件，两个应用程序可以访问同一个文件。其实，应用程序和 DLL也可以同时访问同一个文件。实际上，当 DLL被调入时，如果该DLL创建了内存映射文件，而同时又被另一个应用程序调用时，那么它们访问的是同一个文件。当其中一个应用程序修改文件中的数据时，另一个应用程序立即受到影响，因为，它们引用的是相同的数据，即同一个内存映射文件对象的实例。

在清单9-12中，OpenShareDate()用来创建内存映射文件。首先使用 CreateFileMapping()函数创建文件映射对象，然后传递给MapViewOfFile()函数。MapViewOfFile()函数把文件的视图映射到调用进程的地址空间。该函数的返回值就是该地址的首地址。请记住，这是调用进程的地址。两个应用程序调用该DLL，返回的地址应该是不同的，尽管引用的数据是相同的。

注意 CreateFileMapping()的第一个参数是内存映射文件对象的句柄。如果映射一个系统的页面调度文件，第一个参数值应为$FFFFFFFF(它等于DWord(-1))。最后一个参数用于给出文件映射对象的名称。该名称由系统用于引用这个文件映像。如果多进程使用相同的名称来创建内存映射文件，就可以访问相同的系统内存。

调用了MapViewOfFile()函数后，GlobalData全局变量指向内存映射文件的地址空间。函数GetDLLData()把变量AGlobalData(全局变量)参数返回给调用DLL的应用程序。因此，应用程序可以访问这些

China-pub.com
下载

数据了。

CloseShareData()过程解除映射关系，并释放文件映射对象。这不影响其他文件映射对象或其他应用程序的文件映射关系。

9.11.2 访问DLL中的共享数据

这一节用两个应用程序来举例说明怎样共享数据。第一个应用程序 APP1.dpr用于修改DLL中的数据。第二个应用程序APP2.dpr，也是访问DLL中的数据，并通过定时器来刷新两个标签。当同时运行两个应用程序时，你将看见，APP1对数据进行修改时，APP2会立即受到影响。

清单9-13列出了APP1的源代码。

清单9-13 APP1.dpr的主窗体单元

```
unit MainFrmA1;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls,
  Forms, Dialogs, StdCtrls, ExtCtrls, Mask;

{$I DLLDATA.INC}

type

  TMainForm = class(TForm)
    edtGlobDataStr: TEdit;
    btnGetDllData: TButton;
    meGlobDataInt: TMaskEdit;
    procedure btnGetDllDataClick(Sender: TObject);
    procedure edtGlobDataStrChange(Sender: TObject);
    procedure meGlobDataIntChange(Sender: TObject);
    procedure FormCreate(Sender: TObject);
  public
    GlobalData: PGlobalDLLData;
  end;

var
  MainForm: TMainForm;

{ Define the DLL's exported procedure }
procedure GetDLLData(var AGlobalData: PGlobalDLLData);
  StdCall External 'SHARELIB.DLL';

implementation

{$R *.DFM}

procedure TMainForm.btnGetDllDataClick(Sender: TObject);
begin
  { Get a pointer to the DLL's data }
  GetDLLData(GlobalData);
```

China-pub.com 下载

```
  { Now update the controls to reflect GlobalData's field values }
  edtGlobDataStr.Text := GlobalData^.S;
  meGlobDataInt.Text  := IntToStr(GlobalData^.I);
end;

procedure TMainForm.edtGlobDataStrChange(Sender: TObject);
begin
  { Update the DLL data with the changes }
  GlobalData^.S := edtGlobDataStr.Text;
end;

procedure TMainForm.meGlobDataIntChange(Sender: TObject);
begin
  { Update the DLL data with the changes }
  if meGlobDataInt.Text = EmptyStr then
    meGlobDataInt.Text := '0';
  GlobalData^.I := StrToInt(meGlobDataInt.Text);
end;
procedure TMainForm.FormCreate(Sender: TObject);
begin
  btnGetDllDataClick(nil);
end;

end.
```

这个应用程序也链接了包含文件DLLData.inc，该文件定义了TGlobalDLLData数据类型及其指针。btnGetDLLDataClick()事件处理过程调用GetDLLData()，获取DLL的数据指针，然后访问DLL中的数据。OnChange事件在编辑框数据改变时修改DLL中的数据。

清单9-14列出了APP2.dpr的源代码。

清单9-14 APP2.dpr的主窗体单元源代码

```
unit MainFrmA2;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  ExtCtrls, StdCtrls;

{$I DLLDATA.INC}

type

  TMainForm = class(TForm)
    lblGlobDataStr: TLabel;
    tmTimer: TTimer;
    lblGlobDataInt: TLabel;
    procedure tmTimerTimer(Sender: TObject);
  public
    GlobalData: PGlobalDLLData;
  end;
```

China-pub.com
下载


```
{ Define the DLL's exported procedure }
procedure GetDLLData(var AGlobalData: PGlobalDLLData);
  StdCall External 'SHARELIB.DLL';

var
  MainForm: TMainForm;

implementation

{$R *.DFM}

procedure TMainForm.tmTimerTimer(Sender: TObject);
begin
  GetDllData(GlobalData);  // Get access to the data
  { Show the contents of GlobalData's fields.}
  lblGlobDataStr.Caption := GlobalData^.S;
  lblGlobDataInt.Caption := IntToStr(GlobalData^.I);
end;

end.
```

这个应用程序有两个TLabel组件，它们在定时器tmTimer的OnTimer事件处理过程里定时刷新。用户在APP1中修改数据时，APP2受到的影响在两个标签里反应出来。

你可以用这两个应用程序做一下实验。在本书附带的光盘里可找到它们。

9.12 引出DLL中的对象

可以访问包含在DLL中的对象及其方法。不过，对于DLL内的对象的定义和使用有一些规则需要注意。这里讨论的技术非常重要。通常，就是可以通过包或者接口实现类似的功能。

引出DLL中的对象，需要注意下列规则：

- 应用程序只能访问对象中的虚拟方法。
- 对象实例只能在DLL中创建。
- 应用程序和DLL必须都有对象及其方法的定义，其方法的顺序必须一致。
- DLL中的对象不能继承。

另外，还有些规则，但主要是以上4条。

下面创建一个简单的DLL，以阐明怎样引出DLL中的对象。这个对象包含一个函数，该函数用于字母的大小写的转换。在清单9-15中有对象定义。

清单9-15 从DLL中引出对象

```
type

  TConvertType = (ctUpper, ctLower);

  TStringConvert = class(TObject)
{$IFDEF STRINGCONVERTLIB}
  private
    FPrepend: String;
    FAppend : String;
{$ENDIF}
  public
```

China-pub.com 下载

```
    function ConvertString(AConvertType: TConvertType; AString: String):
String;
      virtual; stdcall; {$IFNDEF STRINGCONVERTLIB} abstract; {$ENDIF}
{$IFDEF STRINGCONVERTLIB}
    constructor Create(APrepend, AAppend: String);
    destructor Destroy; override;
{$ENDIF}
  end;

{ For any application using this class, STRINGCONVERTLIB is not defined and
  therefore, the class definition will be equivalent to:

  TStringConvert = class(TObject)
  public
    function ConvertString(AConvertType: TConvertType; AString: String):
String;
      virtual; stdcall; abstract;
  end;
}
```

清单9-15实际上就是包含文件StrConvert.inc。把这个对象定义放在该包含文件中，是因为这个对象需要同时在DLL和应用程序中定义。这样，DLL和应用程序只需包含这个包含文件，就相当于都定义了该对象。如果要修改这个对象，不需要分别对DLL和应用程序修改，只需修改包含文件，然后把DLL和应用程序的项目重新编译即可。

下面是ConvertString()方法的定义：

```
function ConvertString(AConvertType: TConvertType; AString: String):
  ➥String; virtual; stdcall;
```

之所以把这个方法声明为虚拟的，不是为了使派生类可以重载该方法，是为了让这个方法可以加到虚拟方法表(VMT)中。这里，不打算详细介绍VMT，在第13章将详细讨论。现在，VMT可以看成是内存中的一块区域，存储了指向对象的虚拟方法的指针。调用的应用程序可以从VMT中获得该虚拟方法的指针。如果声明的方法不是虚拟的，那该方法就不会加进VMT中，应用程序就无法获得该方法的指针。实际上，应用程序调用函数其实是获得一个函数的指针。有了对象方法的基地址，Delphi就可以自动地处理任何情况，如传递方法中的隐含参数self。

请注意，上面的包含文件使用了条件指示字STRINGCONVERTLIB。如果要引出该对象，必须在应用程序里重新声明这个DLL对象中的虚拟方法。为了避免在编译时出错，可以给虚拟方法加上abstract指示字。在运行时，这些方法在DLL中得到执行。

清单9-16列出了TStringConvert对象的实现代码。

清单9-16 实现TStringConvert对象的代码

```
unit StringConvertImp;
{$DEFINE STRINGCONVERTLIB}

interface
uses SysUtils;
{$I StrConvert.inc}

function InitStrConvert(APrepend, AAppend: String): TStringConvert; stdcall;

implementation
```

China-pub.com
下载


```
constructor TStringConvert.Create(APrepend, AAppend: String);
begin
  inherited Create;
  FPrepend := APrepend;
  FAppend  := AAppend;
end;

destructor TStringConvert.Destroy;
begin
  inherited Destroy;
end;

function TStringConvert.ConvertString(AConvertType:
  TConvertType; AString: String): String;
begin
  case AConvertType of
    ctUpper: Result := Format('%s%s%s', [FPrepend, UpperCase(AString),
    FAppend]);
    ctLower: Result := Format('%s%s%s', [FPrepend, LowerCase(AString),
    FAppend]);
  end;
end;

function InitStrConvert(APrepend, AAppend: String): TStringConvert;
begin
  Result := TStringConvert.Create(APrepend, AAppend);
end;

end.
```

正如前面所说，该对象必须在DLL中创建。DLL输出了一个例程为InitStrConvert()，它用于创建TStringConvert对象，需要传递两个参数来构造TStringConvert。下面通过一个接口函数来讨论怎样传递这两个参数。

请注意，在这个单元里也用了条件指示字STRINGCONVERTLIB。清单9-17列出了该DLL的代码。

清单9-17 项目文件StringConvertLib.dll

```
library StringConvertLib;
uses
  ShareMem,
  SysUtils,
  Classes,
  StringConvertImp in 'StringConvertImp.pas';

exports
  InitStrConvert;
end.
```

总体上，这个DLL的例子没有包含还没讲过的内容。不过，请注意ShareMem单元。这个单元必须在DLL的项目文件和调用该DLL的应用程序里都声明。

清单9-18演示了怎样使用DLL中引出的对象来转换一个字符串的大小写。在本书附带的光盘里有该项目，项目文件是StrConvertTest.dpr。

China-pub.com 下载

清单9-18 字符串转换的对象

```
unit MainFrm;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  StdCtrls;

{$I strconvert.inc}

type


  TMainForm = class(TForm)
    btnUpper: TButton;
    edtConvertStr: TEdit;
    btnLower: TButton;
    procedure btnUpperClick(Sender: TObject);
    procedure btnLowerClick(Sender: TObject);
  private
  public
  end;
var
  MainForm: TMainForm;

function InitStrConvert(APrepend, AAppend: String): TStringConvert; stdcall;
  external 'STRINGCONVERTLIB.DLL';

implementation

{$R *.DFM}

procedure TMainForm.btnUpperClick(Sender: TObject);
var
  ConvStr: String;
  FStrConvert: TStringConvert;
begin
  FStrConvert := InitStrConvert('Upper ', ' end');
  try
      ConvStr := edtConvertStr.Text;
      if ConvStr <> EmptyStr then
        edtConvertStr.Text := FStrConvert.ConvertString(ctUpper, ConvStr);
  finally
    FStrConvert.Free;
  end;
end;

procedure TMainForm.btnLowerClick(Sender: TObject);
var
  ConvStr: String;
  FStrConvert: TStringConvert;
```

China-pub.com
下载


```
begin
  FStrConvert := InitStrConvert('Lower ', ' end');
  try
      ConvStr := edtConvertStr.Text;
      if ConvStr <> EmptyStr then
        edtConvertStr.Text := FStrConvert.ConvertString(ctLower, ConvStr);
  finally
    FStrConvert.Free;
  end;
end;

end.
```

9.13 总结

DLL是创建Windows应用程序、实现代码重用的重要手段。这章讲述了怎样在 Delphi应用程序中创建和使用DLL，以及调用DLL的不同方法。除此之外，本章还讨论了使用 DLL的一些规则及在不同的应用程序间如何实现DLL的数据共享。

现在，你应该能够创建 DLL，并且可以在应用程序里调用它。在其他章节，你将碰到更多关于DLL方面的知识。