第四部分　打 包 实 例

软件的功能既可以以单个可执行程序的方式发布，也可以以库的方式发布：即把可执行程序中可以分开的部分放进功能库中，以便其他应用程序也可使用。库可以被静态或者动态地链接到其他应用程序中。一个动态链接库可以立刻与多个应用程序共享其功能，它可以限制访问硬驱的次数。若不应用动态链接库，即使在内存很大的情况下，各应用程序访问内存的次数都可能很频繁。

这部分的例子都是涉及到以几种库的形式打包用户软件，包括静态、动态链接库及资源库的例子。

第15章　库

代码库是一种应用程序与其他应用程序共享功能的途径。本章中的所有例子都涉及到库，包括静态、动态链接库和资源库。动态链接库可以在执行时被多个应用程序共享。

例82　静态链接C/C++库　使用Developer Studio进行一些简单的编辑，创建一个MFC应用程序可以访问的静态C和C++库。

例83　动态链接C/C++库　创建一个执行时不需要MFC的DLL。这样，虽然特征少一些，但使得DLL很小。

例84　动态链接MFC扩展类库　创建一个完全的MFC DLL，它具有MFC类的所有功能。

例85　资源库　创建一个MFC DLL，它没有任何功能，只是作为文本串、对话框模板等资源的储存库形式存在。使用资源库，多个应用程序可以共享对话框模板、图标或位图等。具有多语言接口的应用程序也可以使用资源库存储应用程序的特定语言资源。

15.1　例82　静态链接C/C++库

目标

打包C或C++功能到库中，该库静态地与应用程序进行链接。静态链接使得最终执行程序比动态链接的要大。但是在安装应用程序时，不必考虑是否包含了所需的 DLL文件。

注意　本例子假定在库中不使用任何MFC类。若想创建使用MFC类的函数库，参见例84。

策略

使用Developer Studio创建工程工作空间。 Studio创建了一个带有正确编译设置的工作空间，但没有资源文件，甚至连资源文件的框架也没有。因此，必须创建或引进资源文件到工

程中。还要配置C库，以便通过使用_cplusplus编译指令，使它能被C++应用程序直接使用。

步骤

1. 创建一个静态库

1) 选择New菜单命令打开Developer Studio的New对话框。选择Project标签和Win32 Static Library选项。给工程命名之后单击OK按钮。现在，Studio将创建一个带有正确编译C或C++静态库的设置的空工程。

一个静态库工程创建.lib文件的发行版本或者调试版本。发行版本进行了性能优化，而调试版本因带有调试符号而过于庞大。一个好的建议是改换旧文件的调试版本的名字，以使得在应用程序使用时不至于混淆。库的调试版本将不能正确地与应用程序的发行版本链接或执行，反之亦然。

2) 给库的调试版本一个新名字，选择 Studio的Project/Settings菜单命令打开 Project/Settings对话框。找到Settings组合框，然后选择Win32 Debug。最后选择Library标签，改换Output File Name编辑框中显示的名字。典型的做法是在所列的名字后面加上字母“ d ”以标识调试版本。

StdAfx.cpp和StdAfx.h文件定义了应用程序可能需要的每一个运行库和 MFC类。这些文件被应用程序预编译过，以便只需编译其原始代码，而不是每次需要编译资源模块时，再编译这些大量的文件。对于 C++文件，自动假定有这个特性。然而，本工程的 C++文件不使用MFC类库，因此，需要使用如下方法关掉这个特性。

3) 对于C++文件，必须改变工程中的设置，不使用预编译头。如前所示打开 Project Settings对话框，改变所有配置的设置，在文件树中选取 C++文件，然后选择C/C++标签。从Category组合框中，选择 Precompiled header，对于每一个 C++文件，单击 Not using precompiled headers。

2. 添加函数到静态库

1) 引入或者创建.c或.cpp文件。

2) 若要允许库函数使用标准的C运行库，包含如下文件：

```
# include <stdlib.h>
```

3) 若要允许库函数使用Windows API，包含如下文件：

```
# include <windows.h>
```

4) 若想创建C库，必须在C包含文件的头和尾部加上如下代码，以便这个 C包含文件可被C或C++文件使用。我们需要加入这个指令是因为 C++编译器给完全相同的函数名不同的内部符号，而C却不是。符号名被链接器用来把应用程序对象链接在一起。为了强迫 C++编译器给函数赋一个C符号名，需要使用extern “ c ”指令。然而，既然extern “ c ”指令将引发C编译器产生一个语法错误，则在 _cplusplus被定义的情况下只能有条件地处理它。使用 _cplusplus指令是因为当编译C++文件时它为TRUE，而编译C文件时为FALSE。最后的结果是，当这个文件包含在C文件中，它正常处理。然而，在 C++文件中被编译时， C++函数原型被强迫转换为C函数原型。

```
/* at the start of your C .h file */
#ifdef __cplusplus
extern "C" {
```

```
#else
#endif  /* __cplusplus */
:   :   :
your C function prototypes
:   :   :
/* at the end of your C .h file */
#ifdef __cplusplus
}
#endif
```

5) 关于C和C++静态库函数的例子，参见本节的“清单——静态C库实例”和15.1.6节“清单——静态C++库实例”。

3. 使用新静态库

通过打开目标工程的Project Settings对话框，选择Link标签，把静态.lib文件包含在其他应用程序中。然后，填加.lib文件名到Object/Library modules编辑框。并确认对于目标应用程序的调试版本加入的是库的调试版本，对于目标应用程序的发行版本加入的是库的发行版本。否则，目标应用程序不进行连接。

说明

- 对于MFC应用程序来讲，静态C或者C++库的经典应用是从旧的应用程序中把遗留代码转换到新的MFC应用程序。然而，若遗留代码是用C写的，把每个函数转换到静态C++函数可能更好。那样，遗留代码就可容易地调用 MFC函数。(一个C过程调用C++过程不是一件容易的事)。如何转换C源代码到C++呢？简单地把文件名从.c变为.cpp，然后编译，修改语法错误，通常是C例子使用了C++保留字(如class、new等)导致的语法错误。
- 关于库操作的详细情况，参见附录D。

清单——静态C库实例

```
/* WzdStatic.h : NonMFC Static Library
 *
 ****************************************************************/

#if !defined WZDSTATIC_H
#define WZDSTATIC_H

#ifdef __cplusplus
extern "C" {
#else
#endif  /* __cplusplus */

void WzdMessageBox(LPSTR pszString);
void DestroyWzdWindow(HWND hWnd);
void WzdFunc3(BOOL b);

#ifdef __cplusplus
}
#endif
```

```
#endif

/*
 * WzdStatic.c : NonMFC "C" Static Library Using the Win32 API directly
 */

#include <stdlib.h>
#include <windows.h>
#include "WzdStatic.h"

BOOL flag=FALSE;

// must call Win32 API directly
void WzdMessageBox(LPSTR pszString)
{
   MessageBox(NULL,pszString,"Wzd Static Library",MB_OK);
}

void DestroyWzdWindow(HWND hWnd)
{
   DestroyWindow(hWnd);
}

void WzdFunc3(BOOL b)
{
   flag=b;
}
```

清单——静态C++库实例

```
// WzdCpp.h : C++ Static Library
//
#if !defined WZDCPP_H
#define WZDCPP_H

class CWzdDllCpp
{
   BOOL  m_bFlag;

public:
   CWzdCpp();
   ~CWzdCpp();

   void WzdMessageBox(LPSTR pszString);
   void DestroyWzdWindow(HWND hWnd);
   void WzdFunc3(BOOL b);

};


#endif
```

```
// WzdCpp.cpp : C++ Static library using the Win32 API directly
//

#include <windows.h>
#include <stdlib.h>
#include "WzdCpp.h"

CWzdCpp::CWzdCpp()
{
    m_bFlag=FALSE;
}

CWzdCpp::~CWzdCpp()
{
}

// must call the Win32 API directly
void CWzdCpp::WzdMessageBox(LPSTR pszString)
{
    MessageBox(NULL,pszString,"Wzd DLL Cpp",MB_OK);
}

void CWzdCpp::DestroyWzdWindow(HWND hWnd)
{
    DestroyWindow(hWnd);
}

void CWzdCpp::WzdFunc3(BOOL b)
{
    m_bFlag=b;
}
```

15.2 例83 动态链接C/C++库

目标

打包C或C++函数到一个动态链接库中，它允许其他应用程序共享这些功能，并有可能减少总内存需求。在这些函数中不需要任何 MFC类。

注意 本例子假定在库中不使用任何MFC类。创建一个使用MFC类的函数库，参见例84。

策略

使用Developer Studio创建工程工作空间。 Studio创建了一个带有正确编译设置的工作空间，但是没有代码文件，甚至连代码文件框架也没有。因此，必须创建或引入源文件到工程中。也将配置C库，以便通过使用_cplusplus编译指令，使它能直接被C++应用程序使用。

步骤

1. 创建动态链接库

1) 选择New菜单命令打开 Developer Studio的New对话框。选择 Project标签和Win32 Dynamic-Linked Library选项。给工程命名，单击OK按钮。Studio将创建一个带有正确编译动态链接库的设置的空工程。

一个动态链接库工程创建.lib和.dll文件的调试或发行版本。发行版本在性能上进行了优化，而调试版本因调试符号而过于庞大。一个好的建议是改变 .lib和.dll文件调试版本的名称，以使得应用程序使用时不至于混淆。库的调试版本将不能正确地与应用程序的发行版本链接或执行，反之亦然。

2) 为了重新命名库的调试版本，选择 Studio的Project/Settings菜单命令打开 Project Settings对话框。找到Settings组合框，选择Win32 Debug。然后，选择Library标签，改变显示在Output File Name编辑框中的名字。典型的做法是在所列的字符串后面加上一个字母“ d”以标识调试版本。

StdAfx.cpp和StdAfx.h文件定义了应用程序可能需要的每一个运行库和 MFC类。这些文件被应用程序预编译过，以便只有原始代码被编译，而不是每次资源模块需编译时，再编译这些大量的文件。对于 C++文件，自动假定为有这种特性。然而，本工程的 C++文件不使用MFC类库，因此，必须使用如下方法关掉这个特性。

3) 对于C++文件，必须改变工程中的设置，不使用预编译头。如前所示打开 Project Settings对话框，改变所有配置的设置，在文件树中选取 C++文件，然后选择C/C++标签。从Category组合框中，选择 Precompiled header，对于每一个 C++文件，单击 Not using precompiled headers。

2. 添加函数到动态链接库中

1) 引入或者创建.c或.cpp文件。

2) 若要允许库函数使用标准的C运行库，包含如下文件：

```
# include <stdlib.h>
```

3) 若要允许库函数使用Windows API，包含如下文件：

```
# include <windows.h>
```

动态链接库中有的函数可以被库外的其他函数所访问，而有的函数只能在内部被访问。为了通知编译器和链接器，使函数可被外界所访问，需要用 _declspec (dllexport)函数类型声明该函数。然而，当要在应用程序中包含库的原型时，用户可能不想包含这个函数类型。为了解决这个问题，我们创建了一个宏，当创建库时，有条件地表示 declspec(dllexport)；而当应用程序包含这些原型时，该宏无意义。

4) 添加下面的宏到定义库中函数原型的包含文件。

```
#ifdef WZDDLL_BLD
#define DLL __declspec(dllexport)
#else
#define DLL
#endif
```

5) 为了使DLL宏正常工作，需要在 DLL工程设置中定义 WZDDLL_BLD符号。选择Developer Studio的Project/Settings菜单命令，打开Project Settings对话框。找到Settings组合框，

选择All Configurations。然后选择C/C++标签，添加WZDDLL_BLD到Preprocessor definitions编辑框。

6) 使用DLL宏声明一个外部C函数，在函数的定义和实现处插入DLL宏。

```
void DLL WzdMessageBox(LPSTR pszString);
void DLL WzdMessageBox(LPSTR pszString)
{
   :  :  :
}
```

7) 使用DLL宏声明一个外部C++类，及声明它的所有函数为外部函数，只要在类的声明处插入DLL宏。

```
class DLL CWzdDllCpp
{

};
```

8) 若想创建C库，必须在C包含文件的头和尾部加上如下代码，以便这个 C包含文件可被C或C++文件使用。我们需要加入这个指令是因为 C++编译器给完全相同的函数名不同的内部符号，而C却不是。符号名被链接器用来把应用程序对象链接在一起。为了强迫 C++编译器给函数赋一个C符号名，需要使用extern"c"指令。然而，既然extern"c"指令将引发C编译器产生一个语法错误，则在 _cplusplus被定义的情况下只能有条件地处理它。使用 _cplusplus指令是因为当编译C++文件时它为TRUE，而编译C文件时为FALSE。最后的结果是，当这个文件包含在C文件中，它正常处理。然而，在 C++文件中被编译时， C++函数原型被强迫转换为C函数原型。

```
/* at the start of your C .h file */
#ifdef __cplusplus
extern "C" {
#else
#endif  /* __cplusplus */
:  :  :
your C function prototypes
:  :  :
/* at the end of your C .h file */
#ifdef __cplusplus
}
#endif
```

9) 关于C和C++动态链接库函数，参见本节的“ 清单 —— 动态链接C库实例 ”和“ 清单 —— 动态链接C++库实例 ”。

3. 使用新动态链接库

通过打开目标工程的Project Settings对话框，选择Link标签，包含动态链接.lib文件在其他应用程序中，加入.lib名称到Object/library modules编辑框中。并确认对于目标应用程序的调试版本加入的是库的调试版本，对于目标应用程序的发行版本加入的是库的发行版本。否则的话，目标应用程序不能被链接。为了运行应用程序，创建的. dll文件必须放到系统执行路径下，或者是由系统的 PATH环境参数指定的一个路径之下。关于 DLL的详细情况，请看附录D。

说明

- 动态链接库允许它的函数同时被当前所执行的应用程序共享。因此，潜在地能够节省内存需求，因为否则的话，每个应用程序要自己拥有同样的函数。然而，既然Windows操作系统利用虚拟内存，因此 DLL的真正优势在于，应用程序所使用的函数在主存中的机率比在磁盘上的交换文件中高得多。
- 关于库操作的详细情况，请看附录 D。

CD说明

使用Testdll工程测试这个例子。

清单 —— 动态链接C库实例

```
/* WzdDll.h : NonMFC Dll
 *
 ************************************************************/

#if !defined WZDDLL_H
#define WZDDLL_H

#ifdef __cplusplus
extern "C " {
#else
#endif  /* __cplusplus */


#ifdef WZDDLL_BLD
#define DLL __declspec(dllexport)
#else
#define DLL
#endif

void DLL WzdMessageBox(LPSTR pszString);
void DLL DestroyWzdWindow(HWND hWnd);
void DLL WzdFunc3(BOOL b);

#ifdef __cplusplus
}
#endif

#endif

/*
 * WzdDll.c : NonMFC "C" Dll Using the Win32 API directly
 */

#include <windows.h>
#include <stdlib.h>
#include "WzdDll.h"
```

```
BOOL flag=FALSE;

// must call Win32 API directly
void DLL WzdMessageBox(LPSTR pszString)
{
   MessageBox(NULL,pszString,"Wzd DLL",MB_OK);
}

void DLL DestroyWzdWindow(HWND hWnd)
{
   DestroyWindow(hWnd);
}

void DLL WzdFunc3(BOOL b)
{
   flag=b;
}
```

清单——动态链接C++库实例

```
// WzdDllCpp.h : C++ NonMFC Dll
//


#if !defined WZDDLLCPP_H
#define WZDDLLCPP_H

#ifdef WZDDLL_BLD
#define DLL __declspec(dllexport)
#else
#define DLL
#endif

class DLL CWzdDllCpp
{
   BOOL  m_bFlag;

public:
   CWzdDllCpp();
   ~CWzdDllCpp();

   void WzdMessageBox(LPSTR pszString);
   void DestroyWzdWindow(HWND hWnd);
   void WzdFunc3(BOOL b);

};


#endif
```

```
// WzdDllCpp.cpp : NonMFC "C++" Dll Using the Win32 API directly
//

#include <windows.h>
#include <stdlib.h>
#include "WzdDllCpp.h"

CWzdDllCpp::CWzdDllCpp()
{
    m_bFlag=FALSE;
}

CWzdDllCpp::~CWzdDllCpp()
{
}

// must call the Win32 API directly
void CWzdDllCpp::WzdMessageBox(LPSTR pszString)
{
    MessageBox(NULL,pszString,"Wzd DLL Cpp",MB_OK);
}

void CWzdDllCpp::DestroyWzdWindow(HWND hWnd)
{
    DestroyWindow(hWnd);
}

void CWzdDllCpp::WzdFunc3(BOOL b)
{
    m_bFlag=b;
}
```

15.3 例84 动态链接MFC扩展类库

目标

打包C++函数到动态链接库中，同时在这个函数中使用 MFC类。

策略

使用Developer Studio创建一个MFC DLL工程工作空间，然后创建或引入函数文件。每个输出的函数都具有_declspec (dllexport)函数声明。

步骤

1. 创建MFC扩展库

1) 选择New菜单命令打开 Developer Studio的New对话框。选择 Project标签和 MFC

AppWizard (dll)选项。给工程命名，然后单击OK按钮。

2) AppWizard提示创建三种DLL类型的哪一种。Regular DLL可以被不使用MFC的应用程序使用；MFC Extension DLL必须被使用MFC的应用程序使用；第三个选项是与 MFC静态链接的Regular DLL。选择与MFC静态链接将使你的DLL很庞大，但是减少应用程序发布时另带文件数目。对于本例子，选择 MFC Extension DLL。

注意 若用户应用程序是与MFC库静态链接的，不能使用MFC Extension DLL。用户必须改变其应用程序以便与MFC动态链接。这个问题的症状在于，重复定义的MFC符号引起上百个链接错误。

一个MFC扩展类库工程创建 .lib和.dll文件的调试或发行版本。发行版本在性能上进行了优化，而调试版本因调试符号而过于庞大。一个好的建议是改变.lib和.dll文件调试版本的名称，以使得应用程序使用时不至于混淆。库的调试版本将不能正确地与应用程序的发行版本链接或执行，反之亦然。

3) 为了重新命名库的调试版本，选择 Studio的Project/Settings菜单命令，打开 Project Settings对话框。找到Settings组合框，选择Win32 Debug。然后，选择Library标签，改变显示在Output Name编辑框中的名字。典型的做法是在所列的字符串后面加上一个字母“ d ”以标识调试版本。

2. 添加函数到MFC扩展库中

1) 现在，可以忽略 AppWizard创建的文件。需要添加到 DLL中的函数必须是创建的或引入的，也可以使用 ClassWizard从MFC基类创建一个新类，因此称为“ MFC扩展库 ”。关于手工添加函数的例子，参见本节的“ 清单 —— 扩展类实例 ”。

在动态链接库中，有的函数可以被库外的其他函数所访问，而有的函数只能在内部被访问。为告知编译器和链接器使函数可被外界访问，用户需要用 _declspec (dllexport)函数类型声明该函数。然而，当要在应用程序中包含库的原型时，不包含该函数类型。为了解决这个问题，我们创建了一个宏，当创建库时，有条件地表示 _declspec (dllexport)；而当应用程序包含这些原型时，该宏无意义。

2) 添加下面的宏到定义库中函数原型的包含文件。

```
#ifdef _WINDLL
#define DLL __declspec(dllexport)
#else
#define DLL
#endif
```

_WINDLL符号在DLL工程设置中已经定义，它是由 AppWizard创建的。因此，与上两例不同的是，不用自己添加符号。

3) 为了使用DLL宏声明一个外部C++类及声明其所有函数为外部函数，只要在类的声明中加入DLL宏。

```
class DLL CWzdDllCpp
{

};
```

3. 使用新动态链接库

1) 通过打开目标工程的Project Settings对话框，选择Link标签，包含动态链接.lib文件在其他应用程序中；然后，添加.lib名称到Object/library modules编辑框中。并确认对于目标应用程序的调试版本加入的是库的调试版本，对于目标应用程序的发行版本加入的是库的发行版本。否则，目标应用程序不能被链接。

2) 为了运行应用程序，库工程创建的.dll文件必须放到系统执行目录下。可以把它放到应用程序可执行文件的相同目录中，或者放到由系统的PATH环境参数指定的一个路径之下。关于DLL的详细情况，请看附录D。

说明

- 若用Shared MFC DLL创建Regular DLL，则必须在访问MFC类的函数中加入AFX_MANAGE_STATE()宏，该宏必须位于函数中的第一行。

```
void CWzdDllMFC::DestroyWzdWindow(HWND hWnd)
{
   AFX_MANAGE_STATE(AfxGetStaticModuleState());
   // rest of function here
}
```

- 关于库操作的详细情况，请看附录D。

CD说明

使用Testdll工程测试本例子。

清单——扩展类实例

```
// WzdDllMFC.h : MFC Dll
//

#if !defined WZDDLLMFC_H
#define WZDDLLMFC_H

#ifdef _WINDLL
#define DLL __declspec(dllexport)
#else
#define DLL
#endif

class DLL CWzdDllMFC
{
   BOOL  m_bFlag;

public:
   CWzdDllMFC();
   ~CWzdDllMFC();

   void WzdMessageBox(LPSTR pszString);
   void DestroyWzdWindow(HWND hWnd);
   void WzdFunc3(BOOL b);
```

```
};


#endif

// WzdDllMFC.cpp : MFC Dll
//

#include "stdafx.h"
#include "WzdDllMFCx.h"

CWzdDllMFC::CWzdDllMFC()
{
    m_bFlag=FALSE;
}

CWzdDllMFC::~CWzdDllMFC()
{
}

// can use MFC classes and static functions
void CWzdDllMFC::WzdMessageBox(LPSTR pszString)
{
    AfxMessageBox(pszString);
}

void CWzdDllMFC::DestroyWzdWindow(HWND hWnd)
{
    CWnd wnd;
    wnd.Attach(hWnd);
    wnd.DestroyWindow();
}

void CWzdDllMFC::WzdFunc3(BOOL b)
{
    m_bFlag=b;
}
```

15.4 例85 资源库

目标

打包所有资源到DLL中，以便其他应用程序可以共享。分离出所有字符串和对话框模板，并把它们放到DLL中，以便可以创建支持不同语言 (如French、Italian、Esperanto等)的不同版本。

策略

创建只包含资源的 MFC扩展DLL。通过资源ID，应用程序可以像访问自己的资源一样访

问MFC扩展库中的资源。从DLL中访问资源时，唯一要考虑的是应用程序中的资源 ID不能与DLL中的相冲突。

步骤

创建资源库

1) 遵循上个例子的步骤，创建一个MFC扩展DLL。

2) 至少添加一个类到DLL中，这些类可以不包含函数或变量。

3) 使用Developer Studio编辑器添加资源到工程中，并保证资源ID不与任何其他DLL或主应用程序中的资源ID冲突。可赋一个值给ID，或者使用文本ID代替数字ID。要创建一个文本ID，只要用双引号封装ID即可。例如：

```
"MYDLL_RESOURCE1"
```

4) 从resource.h文件中剪切新资源ID，把它们粘贴到一个新的.h文件，这个文件可以包含在使用资源DLL的工程中。

5) 为使应用程序支持多语种，决不应当直接在代码中使用字符串。相反，应把它们放到资源DLL的字符串表中，并以以下方式访问。同样，应把所有对话框模板放到资源 DLL中。

```
CString str;
str.LoadString(id);
AfxMessageBox(str);
```

说明

在主要开发工作都完成之后，再去考虑增加多语种的支持，可能是一种更谨慎的方法。经常性地更新一个字符串表，特别是涉及到多个开发者时，往往使得源文件混乱不堪。把这项工作与主要开发任务混在一起显然是不必要，而可以在工程的末期很容易地完成。

CD说明

使用Testdll工程来测试本例子。