

# Delphi 中的 DLL 封装和调用对象技术

本文刊登 2003 年 10 月份出版的《Dr.Dobb's 软件研发》(第 3 期)

刘 艺

## 摘 要

DLL 是一种应用最为广泛的动态链接技术。但是由于在 DLL 中封装和调用对象受到对象动态绑定机制的限制,使得 DLL 在封装对象方面有一定的技术难度,导致有些 Delphi 程序员误以为 DLL 只支持封装函数,不支持封装对象。本文着重介绍了 DLL 中封装和调用对象的原理和思路,并结合实例给出了多种不同的实现方法。

**关键字:** 动态链接库 (DLL)、对象、接口、虚方法、动态绑定、类引用、面向对象

## 1、 物理封装与动态链接

物理上的封装意味着将程序封装成若干个独立的物理组成部分,各部分之间通过动态链接共同完成系统的功能,而且各个物理组成部分可以单独维护和编译,不影响其他部分。要理解物理封装首先要搞清楚静态链接和动态链接。

在 Delphi 中,如果程序的各个模块分别保存在不同的单元文件中,并通过 uses 指令来互相调用,这就是一个典型的静态链接。于是各个静态的子例程编译之后,连接器从 Delphi 编译过的单元(或静态库)中取出子例程编译代码并添加到执行文件中。最终 EXE 文件包括了程序及其所属单元的所有代码。显然,静态链接的单元或模块最终以一个独立的物理形式(可执行文件)存在。除了自己编写的单元文件,Delphi 还自动 uses 了一些预设的单元,如: Windows、Messages 等,这些都是静态链接。

静态链接无法实现物理上的切割和封装,而且一旦其中某个单元或模块改动,其他所有单元或模块都得随之重新编译和连接。

用于实现物理切割和封装的 bpl 包、DLL 动态链接库或 COM+ 组件都是一种动态链接的形式。在动态链接情况中,连接器只使用子例程 external 声明中的信息在执行文件中产生一些数据表格。当 Windows 向内存中装载执行文件时,它首先装载所有必需的 DLL,然后程序才会启动。在装载过程中,Windows 用函数在内存中的地址填充程序的内部表格。

每当程序调用一个外部函数时,它就会使用该内部数据表格直接对 DLL 代码(它当前装载在程序的地址空间中)进行调用。注意,该模式不会涉及两个不同的应用程序,DLL 已经变成了应用程序的一部分,并装载在同一地址空间。所有参数的传递都发生在堆栈上,与其它任何函数调用一样。这里我们不打算讨论 DLL 的编译,因为我们首先想重点介绍 Delphi 中的 DLL 封装和调用对象技术。

## 2、 用 DLL 封装对象

DLL (Dynamic Link Library, 动态链接库) 就目前来讲已经不再是什么新技术, 读者可以在书店过时的 Delphi 书籍里随便找到讨论 DLL 编程的章节。但这些涉及 DLL 编程的书中几乎都是谈论用 DLL 来封装函数的, 实际上大量的程序员也是在使用 DLL 来封装函数, 或面向过程的一个模块 (一个函数集合)。而在这里, 我只想讨论如何用 DLL 来封装对象, 这可能是读者未曾有过的 DLL 使用经验, 但这却是这本完全围绕面向对象编程的书中重要的部分之一, 或许你能从中发现一些与众不同的实用技巧。

参见: 考虑到目前关于DLL的现成资料很多, 这里我省略了DLL的基本知识和编写方法, 假设读者已经有了一定的DLL编程基础。如果你没有这样的基础, 建议参阅拙作《Delphi6企业级解决方案及应用剖析》“DLL编程技术”一节 (P271)。

一般来说, 使用 DLL 封装对象主要有以下好处:

- 节约内存。多个程序可以使用同一个 DLL 时, 该 DLL 只需加载一次, 而且可以只在使用时加载, 不用时销毁。
- 使程序代码实现复用。这就是说用 DLL 封装的对象可以重复使用, 甚至可以让不同的程序语言调用。
- 使程序模块化、组件化。这样利于团队开发, 维护和更新方便。

然而 DLL 在封装对象方面却有一定的技术难度, 这方面资料极少, 甚至有的程序员误以为 DLL 只支持封装函数, 不支持封装对象。

通过研究, 我们发现 DLL 在封装对象上主要的限制在于:

- 调用 DLL 的应用程序只能使用 DLL 中对象的动态绑定的方法。
- DLL 封装对象的实例只能在 DLL 中创建。
- 在 DLL 和调用 DLL 的应用程序中都需要对封装的对象及其被调用的方法进行声明。

下面我先通过一个简单的例子来演示如何使用 DLL 封装对象, 并在应用程序中调用该对象。然后再讨论相关的技术细节。

## 3、 一个简单的例子

读者一定还记得我们在前面章节中演示了车的继承关系和合成关系。这个程序由逻辑单元的 Demo 和界面单元 frmDemo 组成。我们现在就用 DLL 封装 Demo 单元的所有对象, 并在 frmDemo 单元实现调用。读者可以通过这个具体的例子来学习如何使用 DLL 封装对象。

打开项目文件 ObjDemo.dpr, 如图 1 所示, 在项目管理器 (Project Manager) 中, 鼠标右击 ProjectGroup1, 然后在弹出菜单中选择 Add New Project...菜单项。此时弹出如图 2 所示的 New Items 对话框。选择 DLL Wizard, Delphi 的 DLL 向导将创建一个 DLL 项目。我们将该项目重新命名为 DemoSvr, 并保存在项目组同一目录下。

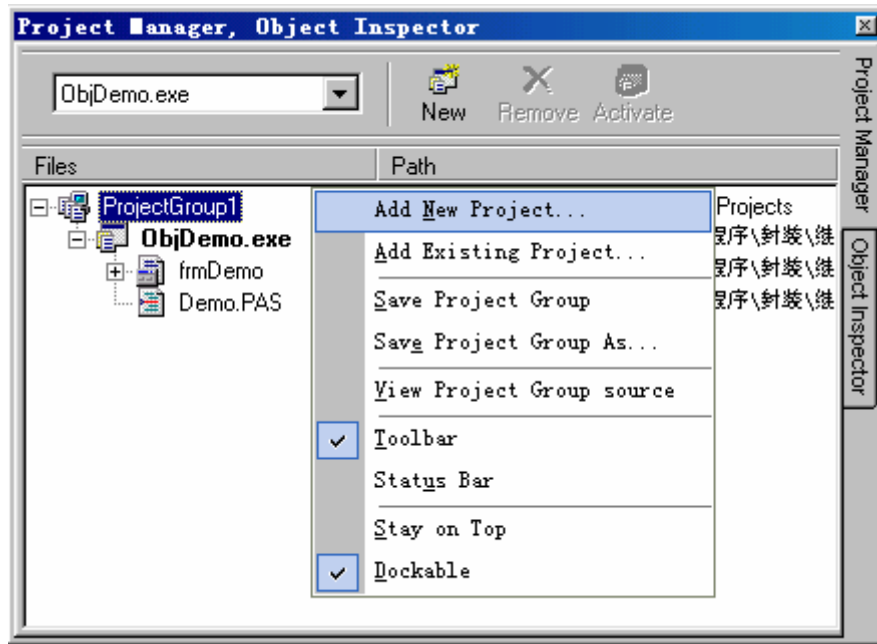


图 1 鼠标右击 ProjectGroup1，在弹出菜单中选择 Add New Project...菜单项。

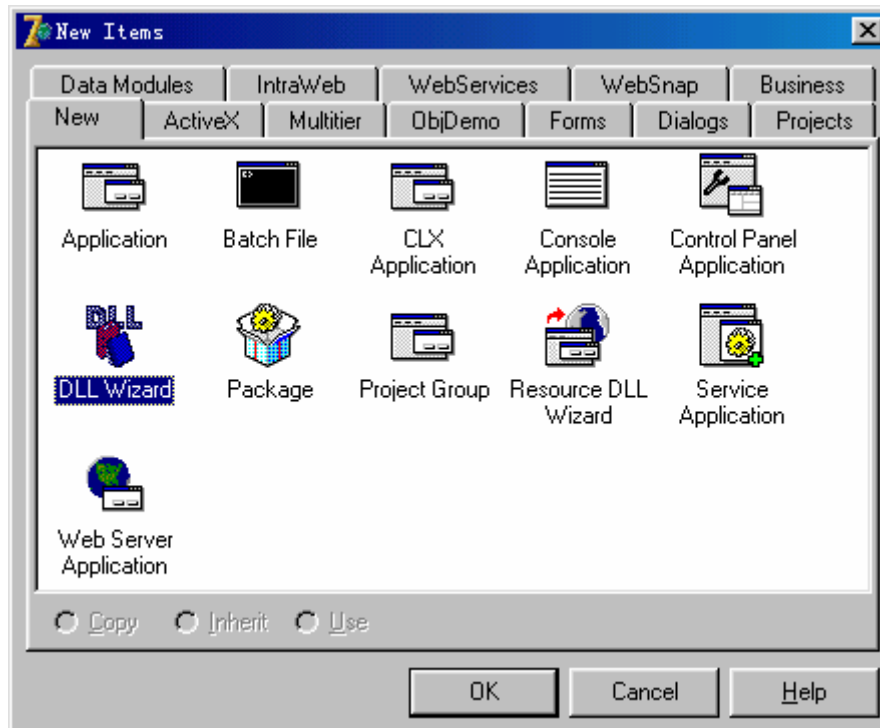


图 2 在 New Items 对话框中选择 DLL Wizard

修改 DemoSvr 中的代码如示例程序 1 所示。

示例程序 1 动态链接库 DemoSvr 的主程序

```
library DemoSvr;
```

```
{ Important note about DLL memory management: ShareMem must be the
```

```

first unit in your library's USES clause AND your project's (select
Project-View Source) USES clause if your DLL exports any procedures or
functions that pass strings as parameters or function results. This
applies to all strings passed to and from your DLL--even those that
are nested in records and classes. ShareMem is the interface unit to
the BORLNDMM.DLL shared memory manager, which must be deployed along
with your DLL. To avoid using BORLNDMM.DLL, pass string information
using PChar or ShortString parameters. }

uses
    ShareMem,
    SysUtils,
    Classes,
    Demo in 'Demo.PAS';

{$R *.res}

function CarObj:TCar;
begin
    Result:=TCar.create;
end;

function BicycleObj:TBicycle;
begin
    Result:=TBicycle.create;
end;

exports
    CarObj,
    BicycleObj;
end.

```

由此可见，DLL 封装对象的实例是在 DLL 中创建的，CarObj 和 BicycleObj 函数创建并输出了 Car 对象和 Bicycle 对象的引用。这样 DemoSvr 动态链接库就可以通过 CarObj 和 BicycleObj 函数输出 Car 对象和 Bicycle 对象了。但是 Car 对象和 Bicycle 对象是在 Demo.pas 文件中声明和实现的，所以这里 uses 了 Demo.PAS。

为了能够使用 Demo.PAS，在项目管理器中直接把 Demo.pas 文件从 ObjDemo 项目中拖放到 DemoSvr 项目中，如图 3 所示。

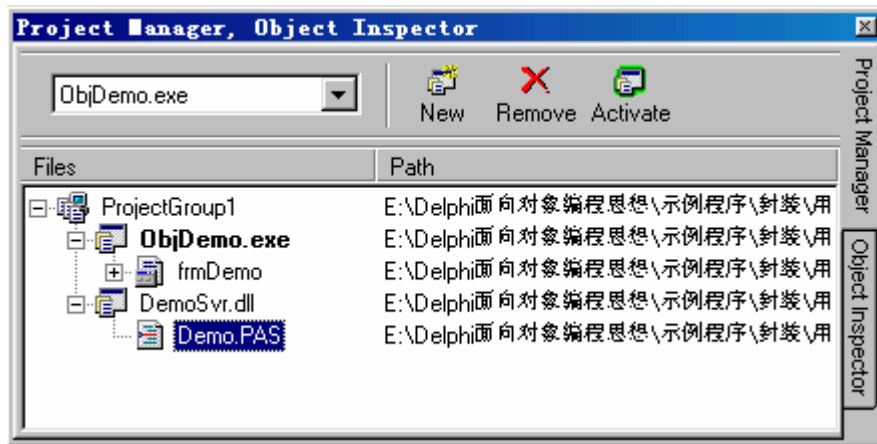


图 3 将 Demo.pas 文件从 ObjDemo 项目中拖放到 DemoSvr 项目中

打开 Demo.pas，修改 TBicycle 和 TCar 的声明如下：

```

TBicycle = class(TVehicle)
public
    constructor create;
    destructor Destory;
    procedure ride;virtual;
end;

TCar = class(TVehicle)
protected
    FEngine: TEngine;
public
    constructor create;
    destructor Destory;
    procedure drive;virtual;
end;

```

请注意，这里我把应用程序中需要调用的对象方法 ride 和 drive 改成了虚方法。显然，这么做不是为了让 TBicycle 和 TCar 的派生类来覆盖 ride 和 drive 方法。这是因为编译连接应用程序时，编译器无法知道（也无需知道）对象在 DLL 中的方法是如何实现的，这就意味着对于应用程序来说要使用动态绑定（晚绑定）技术，所以调用 DLL 的应用程序只能使用 DLL 中对象的动态绑定的方法。前面我们讲过，虚方法的动态绑定技术是把虚方法的入口放到虚方法表 VMT 中。VMT 是一块包含对象方法指针的内存区，通过 VMT 调用程序可以得到虚方法的指针。如果我们不把 ride 和 drive 声明为虚方法，VMT 中就不会有这些方法的入口指针，因此调用程序也就无法得到这个方法的入口指针。

接下来回到 frmDemo 单元，在调用 DLL 的应用程序中同步声明需要调用的对象及其被调用的方法。这里除了将 ride 和 drive 声明为虚方法外，还要声明为抽象方法。因为 frmDemo 单元不提供 ride 和 drive 方法的实现，不把它们声明为抽象方法则编译时无法通过。应用程序 frmDemo 单元的完整代码如示例程序 2 所示。

示例程序 2 调用 DLL 对象的应用程序

```

unit frmDemo;

interface

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

type

//---这里声明需要用到的 DLL 中对象的方法---

  TVehicle = class(TObject);
  TCar = class(TVehicle)
  public
    procedure drive;virtual;abstract;
  end;
  TBicycle = class(TVehicle)
  public
    procedure ride;virtual;abstract;
  end;

//-----

  TForm1 = class(TForm)
    Button1: TButton;
    Button2: TButton;
    procedure Button2Click(Sender: TObject);
    procedure Button1Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

//---这里导入 DLL 文件及其函数---
  function CarObj:TCar ;external 'DemoSvr.dll';
  function BicycleObj:TBicycle ;external 'DemoSvr.dll';

implementation

{$R *.dfm}

```

```

procedure TForm1.Button2Click(Sender: TObject);
var MyCar:TCar;
begin
  MyCar:=CarObj;
  if Mycar=nil then exit;
  try
    MyCar.drive;
  finally
    MyCar.Free;
  end;
end;

procedure TForm1.Button1Click(Sender: TObject);
var Bicycle:TBicycle;
begin
  Bicycle:=BicycleObj;
  try
    Bicycle.ride;
  finally
    Bicycle.Free;
  end;
end;

end.

```

最后，选择 Build All Projects 菜单项，编译和连接所有的项目，如图 4 所示，我们就得到了需要的应用程序可执行文件以及 DLL。运行测试，可以看到这个程序实现了和原先一样的功能。

但是我对这样的 DLL 封装对象实现不是太满意。因为在 DLL 和应用程序中都需要声明封装的对象，还要使用好 virtual 和 abstract 限定符（很容易造成阅读程序理解上的错觉）。如果一旦对象发生变化，就需要分别在两边修改对象声明，以保持同步，稍有不慎就会出错。对此，Steve Teixeira 在《Delphi6 开发人员指南》（机械工业出版社 2003 年出版，相关内容参见该书 209 页）一书中提出了使用头文件的方法，并通过加上编译指令来控制 DLL 和应用程序分别读到不同的头文件内容。这个方法虽然可以通过只修改头文件来保持声明的同步，但编译指令和头文件使得阅读程序更加困难。

在这里我有一个更好的方法供读者分享，这就是使用接口的方法。

#### 4、 利用 Delphi 接口实现 DLL 中对象的动态绑定

前面我们分析过，调用 DLL 的应用程序只能使用 DLL 中对象的动态绑定的方法，理解这一点是实现 DLL 封装和使用对象的关键。那么，Delphi 接口技术为我们提供了一个最佳选择。

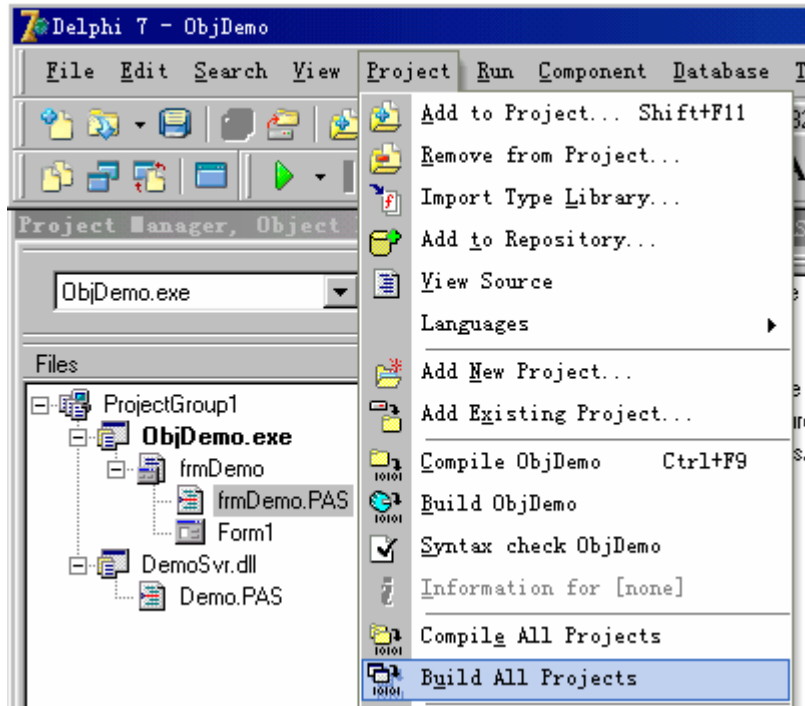


图 4 选择 Build All Projects 菜单项，编译和连接所有的项目

为此我们创建一个接口单元 IDemo，分别声明 ICar 和 IBicycle 接口，接口方法分别是应用程序要用到的 Drive 和 Ride。完整代码如示例程序 3 所示。

示例程序 3 接口单元 IDemo 的代码。

```

unit IDemo;

interface

type
  ICar = interface (IInterface)
    ['{ED52E264-6683-11D7-B847-001060806215}']
    procedure Drive;
  end;

  IBicycle = interface (IInterface)
    ['{ED52E264-6683-11D7-B847-001060806216}']
    procedure Ride;
  end;

implementation

end.
  
```

注意，接口单元 IDemo 中没有（也不能有）任何实现，它同时被应用程序和 DLL 所用（Use），这样当需要修改应用程序调用的对象方法时，只要在一个地方（即该接口单元）

修改即可，避免了可能出现的声明不一致错误。

使用接口还带来了更多的好处。首先无需使用 `virtual` 和 `abstract` 限定符修改对象方法声明，避免了程序阅读上的错觉；其次，利用接口实例计数器自动管理对象的生命期，避免了程序员遗忘销毁对象造成的内存泄漏。

为了使用接口，我将 Demo 单元的类型声明部分作了以下修改，以便 `TBicycle` 和 `TCar` 类能够实现接口方法。值得高兴的是，该单元仅仅需要修改声明部分，而程序实现部分根本不需要做任何改动。

```
unit Demo;

interface

uses
  SysUtils, Windows, Messages, Classes, Dialogs, IDemo;

type
  TVehicle = class(TInterfacedObject)
  protected
    FColor: string;
    FMake: string;
    FTopSpeed: Integer;
    FWheel: TWheel;
    FWheels: TList;
    procedure SlowDown;
    procedure SpeedUp;
    procedure Start;
    procedure Stop;
  end;

  TBicycle = class(TVehicle, IBicycle)
  public
    constructor create;
    destructor Destory;
    procedure ride;
  end;

  TCar = class(TVehicle, ICar)
  protected
    FEngine: TEngine;
  public
    constructor create;
    destructor Destory;
    procedure drive;
  end;
```

最后检查一下项目管理器，确保在应用程序项目和 DLL 项目中都添加了 IDemo 单元，如图 5 所示。

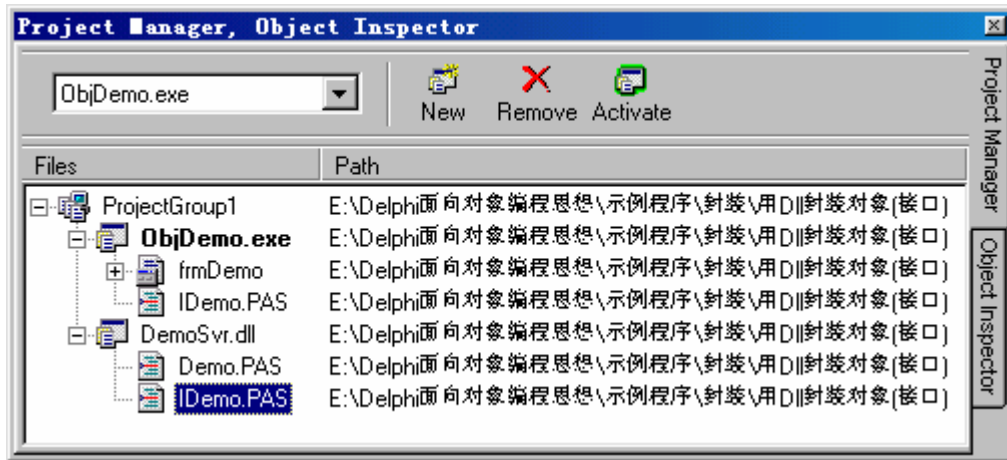


图 5 确保在应用程序项目和 DLL 项目中都添加了接口单元 IDemo

#### 示例程序 4 使用接口技术调用 DLL 对象的应用程序

```

unit frmDemo;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
  Dialogs, StdCtrls, IDemo; //在这里 Use IDemo 单元

type
  TForm1 = class(TForm)
    Button1: TButton;
    Button2: TButton;
    procedure Button2Click(Sender: TObject);
    procedure Button1Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;
  function CarObj:ICar ;external 'DemoSvr.dll';
  function BicycleObj:IBicycle ;external 'DemoSvr.dll';

```

```

implementation

{$R *.dfm}

procedure TForm1.Button2Click(Sender: TObject);
var MyCar:ICar;
begin
    MyCar:=CarObj;
    MyCar.drive;
    Mycar:=nil;
end;

procedure TForm1.Button1Click(Sender: TObject);
var Bicycle:IBicycle;
begin
    Bicycle:=BicycleObj;
    Bicycle.ride;
    Bicycle:=nil;
end;

end.

```

在示例程序 4 中，改动不是很多。这里 Use 了 IDemo 单元，而没有额外的声明。实现部分通过接口调用了 DLL 中的接口方法，也可以说是对象方法。运行示例程序 4 和运行示例程序 2，实现的功能完全一样。

## 5、 使用抽象类实现 DLL 中对象的动态绑定

既然 DLL 中封装和调用对象受到了对象动态绑定机制的限制，那么除了利用 Delphi 接口技术外，我们还可以考虑使用抽象类来实现 DLL 中对象的动态绑定机制。

图 6 显示了一个基于数据库应用的示例程序的面向对象设计。我将界面部分设计成一个瘦客户机的形式，这是一个供用户交互的可执行文件（distributabel2.exe），它封装了外观类 TfrmUsers。我把业务部分（包括数据模块）设计成提供服务的服务器，这是一个动态链接库文件（UserSvr.dll），它封装了业务类 TuserMaint 和数据库访问类 TuserDM。这种设计体现了界面和业务分离的思想。

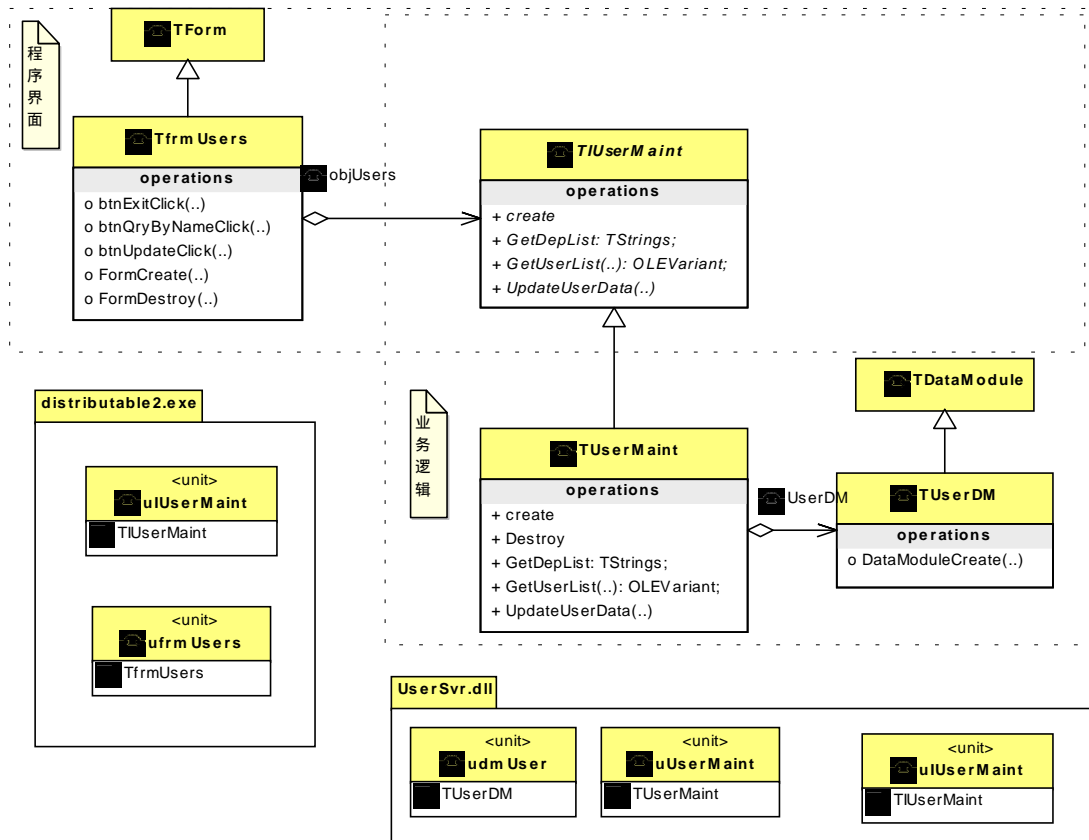


图 6 界面和业务的物理分离的设计

由于原来的逻辑独立的类和代码存放在不同的单元文件中, 我们很容易重新将它们划分到不同的项目里, 如图 7 所示。

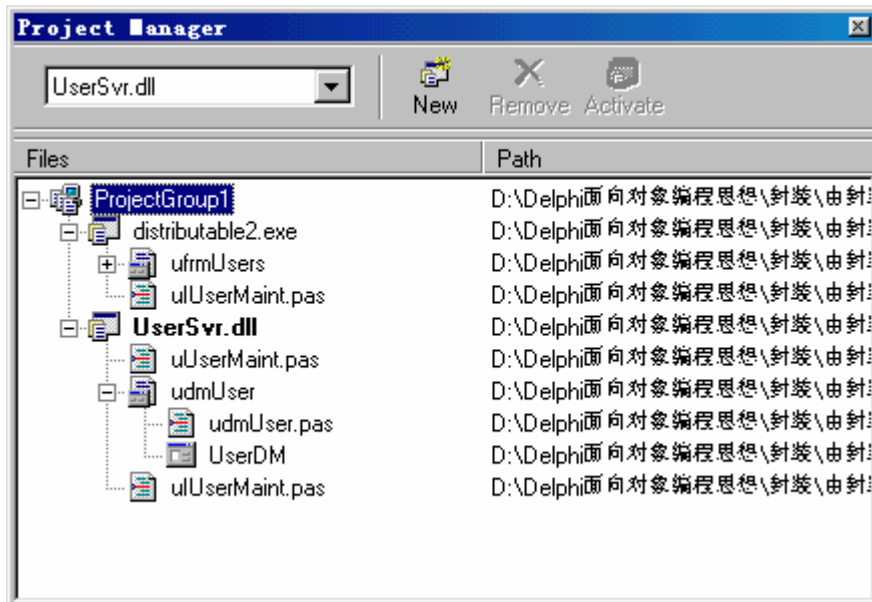


图 7 重新将逻辑独立的单元文件划分到不同的项目里

瘦客户机其实上就是一个空壳, 只提供交互的界面, 它的外观类 TfrmUsers 向 TUserMaint 的实例对象请求服务, 该对象封装在 DLL 中。前面我已经讲过如何用 DLL 来封

装对象。除了前面讲过的两种方法外，这里我想介绍第三种方法，即使用抽象类做接口的方法。

由于调用 DLL 的应用程序只能使用 DLL 中对象的动态绑定的方法，我们不妨专门设计一个抽象类 TIUserMaint 作为提供对象方法的接口。在抽象类 TIUserMaint 中，有供应用程序使用的对象方法，不过它们都是虚抽象方法，目的是支持动态绑定而又无需提供实现。

我将新增的 TIUserMaint 放在抽象类接口单元 uUserMaint.pas 文件中，其源代码如示例程序 5 所示。这个单元将作为接口文件分别包含在 UserSvr 和 Distributable2 项目中，如图 7 所示。

#### 示例程序 5 抽象类接口单元 uUserMaint 代码

```
unit uUserMaint;  
  
interface  
  
uses  
  Classes;  
  
type  
  TIUserMaint = class (TObject)  
  public  
    function GetDepList: TStrings;virtual;abstract;  
    function GetUserList(strName:String): OLEVariant;virtual;abstract;  
    procedure UpdateUserData(UserData:OleVariant; out ErrCount: Integer);  
      virtual;abstract;  
    constructor create;virtual;abstract;  
  end;  
  
  TIUserMaintClass=class of TIUserMaint;  
  
implementation  
  //没有实现代码  
end.
```

在示例程序 5 中还定义了 TIUserMaintClass 类型，它是 TIUserMaint 的类引用。这对于把实现类从 DLL 传递到进行调用的应用程序是必要的。

一般抽象类只定义接口，它由虚抽象方法组成而没有实际的数据。为了实现抽象类 TIUserMaint 的抽象方法，原来的 TUserMaint 类需要继承 TIUserMaint 类，并覆盖其所有的虚抽象方法。新的 TUserMaint 类声明如下：

```
TUserMaint = class (TIUserMaint)  
private  
  UserDM:TUserDM;  
public  
  function GetDepList: TStrings;override;
```

```

function GetUserList(strName:String): OLEVariant;override;
procedure UpdateUserData(UserData:OleVariant; out ErrCount: Integer);
    override;
constructor create;override;
destructor Destroy;override;
end;

```

但实际上 TUserMaint 类原有的实现部分并不需要改动，所以我们的工作量不大。

示例程序 6 是动态链接库 UserSvr.dll 的源代码。这里我使用了 TObjUsers 函数，该函数返回了一个类型为 TIUserMaintClass 的类引用而不是对象引用，所以在应用程序中可以使用这样的代码来创建 DLL 封装的对象：

```
objUsers:=TObjUsers.Create;
```

但这不意味着 TObjUsers 是一个类，记住这里 TObjUsers 是一个 DLL 输出的函数，它的返回类型是一个类引用类型。

#### 示例程序 6 动态链接库 UserSvr.dll 的源代码

```

library UserSvr;

uses
    ShareMem,
    SysUtils,
    Classes,
    uUserMaint in 'uUserMaint.pas',
    udmUser in 'udmUser.pas' {UserDM: TDataModule},
    uIUserMaint in 'uIUserMaint.pas';

{$R *.res}
function TObjUsers:TIUserMaintClass;
begin
    result:=TUserMaint;
end;

exports
    TObjUsers;

begin
end.

```

细心的读者可能已经发现，既然 TObjUsers 函数的返回类型为 TIUserMaintClass，TIUserMaintClass 在示例程序 5 中声明为：

```
TIUserMaintClass=class of TIUserMaint;
```

那么 result:=TUserMaint 会不会是写错了呢？

没有写错！我们在应用程序中声明了 TIUserMaint 类型的对象 objUsers，通过传递类引用，那条 objUsers:=TObjUsers.Create 语句实现的是 objUsers:= TUserMaint.Create 功能，这里面隐含了 TUserMaint 向 TIUserMaint 转型的过程。当然，TIUserMaint 作为抽象类本身也无法直接创建自己的实例，所以必须通过转型才行。另外，在 TIUserMaint 的派生类中可以随意改变方法的实现，却不会影响到方法的接口。这就是说，你以后通过进一步修改 DLL 封装对象的实现方法来升级 DLL，无需重新修改和编译应用程序。因为 TIUserMaint 作为抽象类提供的方法接口没有改变。

示例程序 7 是应用程序实现界面和业务的物理分离后的界面单元的源代码。这里要注意几点：

- 在 interface 部分要 Uses 抽象类接口单元 uIUserMaint;
- objUsers 声明为 TIUserMaint 类型;
- 声明 DLL 函数: function TObjUsers:TIUserMaintClass; external 'UserSvr.dll';

除此之外，几乎不需要进行其他的改动。由此可见，从界面和业务的逻辑分离演化到界面和业务的物理分离实际上并不是想象的那样困难。

#### 示例程序 7 物理分离后的界面单元代码

```
unit ufrmUsers;  
  
interface  
  
uses  
    Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,  
    Dialogs, DB, DBClient, StdCtrls, DBCtrls, Grids, DBGrids, Mask, ExtCtrls,  
    Buttons, uIUserMaint;  
  
type  
    TfrmUsers = class(TForm)  
        btnExit: TButton;  
        btnQryByName: TSpeedButton;  
        Label1: TLabel;  
        Label2: TLabel;  
        Label3: TLabel;  
        Label5: TLabel;  
        Label6: TLabel;  
        Label7: TLabel;  
        edtQryByName: TLabeledEdit;  
        DBEdit1: TDBEdit;  
        DBEdit2: TDBEdit;  
        DBEdit3: TDBEdit;  
        DBEdit4: TDBEdit;  
        DBGrid1: TDBGrid;  
        dbcbSex: TDBComboBox;
```

```

    dbcbDep: TDBComboBox;
    DataSource1: TDataSource;
    cdsUserMaint: TClientDataSet;
    cdsUserMaintID: TWideStringField;
    cdsUserMaintNAME: TWideStringField;
    cdsUserMaintSEX: TWideStringField;
    cdsUserMaintJOB: TWideStringField;
    cdsUserMaintTEL: TWideStringField;
    cdsUserMaintCALL: TWideStringField;
    cdsUserMaintDEP: TWideStringField;
    cdsUserMaintGROUP_ID: TWideStringField;
    cdsUserMaintPASSWORD: TWideStringField;
    btnUpdate: TBitBtn;
    procedure btnUpdateClick(Sender: TObject);
    procedure btnQryByNameClick(Sender: TObject);
    procedure FormCreate(Sender: TObject);
    procedure btnExitClick(Sender: TObject);
    procedure FormDestroy(Sender: TObject);
private
    objUsers: TIUserMaint;
public
    { Public declarations }
end;

var
    frmUsers: TfrmUsers;

const
    M_TITLE=' 操作提示'; //所有提示对话框的标题

implementation

{$R *.dfm}
function TObjUsers: TIUserMaintClass;
    external 'UserSvr.dll';

procedure TfrmUsers.btnUpdateClick(Sender: TObject);
var
    nErr: integer;
begin
    if cdsUserMaint.State=dsEdit then cdsUserMaint.Post;
    if (cdsUserMaint.ChangeCount > 0) then
        begin
            objUsers.UpdateUserData(cdsUserMaint.Delta, nErr);
        end;
    end;
end;

```

```

    if nErr>0 then
        application.MessageBox('更新失败!',M_TITLE,MB_ICONWARNING)
    else
        begin
            application.MessageBox('更新成功!',M_TITLE,MB_ICONINFORMATION) ;
            btnQryByNameClick(nil);
        end;
    end;
end;

procedure TfrmUsers.btnQryByNameClick(Sender: TObject);
begin
    btnUpdate.Enabled:=true;
    dbcDep.Items.AddStrings(objUsers.GetDepList);
    cdsUserMaint.Active:=false;
    cdsUserMaint.Data:=objUsers.GetUserList(edtQryByName.Text);
    cdsUserMaint.Active:=True;
end;

procedure TfrmUsers.FormCreate(Sender: TObject);
begin
    objUsers:=TObjUsers.Create;
end;

procedure TfrmUsers.btnExitClick(Sender: TObject);
begin
    close;
end;

procedure TfrmUsers.FormDestroy(Sender: TObject);
begin
    objUsers.Free;
end;

end.

```

---

### 作者简介:

刘艺, 海军工程大学副教授, 美国 Borland 公司授予的 Delphi 产品专家, 计算机技术作家。著有 10 多部计算机专著, 出版重点大学计算机教材 2 部。这篇文章摘编自他的新书《Delphi 面向对象编程思想》。作者个人网站: <http://www.liu-yi.net>