第4章　应用程序框架和设计

本章内容：

- 理解Delphi环境和项目的体系结构
- 构成Delphi 5项目的文件
- 项目管理提示
- Delphi 5项目的框架类
- 定义公共体系结构 —— 使用对象库
- 一些项目管理的功能

本章主要介绍Delphi的项目管理和体系结构，将告诉你怎样正确使用窗体 (Form)以及怎样运用它们的功能和可视化特征。本章将讨论应用程序启动 /初始化过程、窗体重用 /继承以及增强的用户界面等技术。本章还要介绍下列构成Delphi 5应用程序的类：TApplication、TForm、TFrame和TScreen。我们将向你证明掌握这些概念有助于正确建立应用程序。

4.1　理解Delphi环境和项目的体系结构

要正确建立和管理Delphi 5项目，至少有两个重要因素。第一是要了解创建项目的开发环境的细节，第二是要知道构成Delphi 5项目的体系结构。这一章并不带你深入了解Delphi 5的开发环境(Delphi文档介绍了怎样使用这个环境)；相反，这一章只讲述Delphi 5 IDE的特点，以帮助你更有效地管理项目。本章还要解释Delphi应用程序的体系结构。这不但使你能够充分发挥开发环境的特点，而且使你能够正确使用其固有的体系结构。

我们的第一个建议是：熟练掌握 Delphi 5开发环境。本书假设你已经熟悉 Delphi 5的IDE。其次，本书假设你已经读过Delphi 5的文档。但是，你仍然应该把Delphi 5的菜单和对话框都打开一遍。如果遇到什么不清楚的选项、设置或操作，应该打开在线帮助从中找到答案。在这里所花费的时间将被证明是值得的(更不用说还学会了如何有效地使用在线帮助)。

提示　在所有参考工具里面，Delphi 5的帮助系统无疑是最有价值和最快的。学会如何利用它在数千页的帮助主题中进行浏览是非常有用的。

Delphi 5的帮助包括了从怎样使用Delphi 5的环境到Win32 API的细节以及复杂的Win32结构等内容。要快速获得帮助，可以在编辑器中键入帮助主题，并使光标落在该主题上，然后按下Ctrl+F1键，帮助信息屏幕就会立即显示出来。也可以在对话框中单击 Help按钮，或者在一个组件上按下F1键来获得帮助信息。还可以选择Delphi的Help菜单来浏览帮助系统。

4.2　构成Delphi 5项目的文件

一个Delphi 5项目由若干个相关的文件构成。一些文件是在设计时 (如定义主窗体)创建的。其他文件是在编译项目的时候生成的。要有效地管理 Delphi 5的项目，必须知道其中每一个文件的用途。Delphi 5的文档和在线帮助都详细介绍了项目中的文件。让我们先回顾一下这些文档，以确保你已经熟悉了这些文件。

4.2.1 项目文件

项目文件是在设计时创建的，它的扩展名是 .dpr。这个文件也是主程序文件。项目文件是主窗体以及其他自动创建的窗体实例化的地方。一般不需要编辑项目文件，除非要执行程序初始化例程、显示启动画面或执行其他必须在程序启动时运行的例程。下面的代码是一个典型的项目文件：

```
program Project1;
uses
  Forms,
  Unit1 in 'Unit1.pas' {Form1};
{$R *.RES}
begin
  Application.Initialize;
  Application.CreateForm(TForm1, Form1);
  Application.Run;
end.
```

Pascal程序员会把项目文件看作是标准的Pascal源文件。注意uses子句列出了主窗体单元Unit1。项目文件以同样的方式列出项目的所有窗体单元。下面这行代码用于引用项目的资源文件：

```
{$R *.RES}
```

这一行告诉编译器去链接一个资源文件，该资源文件名与项目文件相同，但扩展名是 .res。项目的资源文件中包含了程序图标和版本信息。

最后，begin..end之间的语句是应用程序要执行的主代码。在这个例子中，创建了主窗体即 Form1。当Application.Run这条语句执行后，Form1作为主窗体显示出来。后面将会介绍，可以在 begin..end之间加入自己的代码。

4.2.2 单元文件

单元文件是Pascal源文件，它的扩展名是 .pas。有三种类型的单元文件：窗体 /数据模块和框架的单元文件、组件的单元文件和通用的单元文件。

- 窗体/数据模块和框架单元文件是由 Delphi 5自动生成的。每个窗体 /数据模块或框架都有一个对应的单元文件。例如，不能让两个窗体共用一个单元文件。为了解释窗体文件，我们不在窗体、数据模块和框架之间进行区分。
- 组件的单元文件是由程序员或Delphi 5创建新的组件时生成的。
- 通用的单元文件是由程序员创建的，用于声明在应用程序中要访问的数据类型、变量、过程、类等。

后面将详细介绍这些单元的细节。

4.2.3 窗体文件

窗体文件存储了窗体的二进制信息。当创建一个窗体时， Delphi 5将同时创建一个窗体文件(扩展名为.dfm)和一个Pascal单元文件(扩展名为.pas)。如果打开一个窗体的单元文件，会看到下面这行语句：

```
{$R *.DFM}
```

这一行告诉编译器去链接对应的窗体文件(名称与单元文件相同，但扩展名是.dfm)到项目中。

一般不用直接编辑窗体文件(尽管可以这么做)。可以用Delphi 5编辑器打开一个窗体文件，这样就能够查看或编辑文本形式的窗体文件了。要打开一个窗体文件，先选择 File | Open菜单命令，然后选择只打开窗体文件(.dfm)的选项。也可以在窗体设计器上单击鼠标右键，在弹出的菜单中选择 View as Text命令。当打开文件后，会看到窗体的文本形式。

查看窗体的文本形式会带来一些方便，因为可以从中看出非缺省的属性设置以及窗体上有那些组件。编辑窗体文件是修改组件类型的方法之一。例如，假设一个窗体上有一个 TButton组件：

```
object Button1: TButton
    Left = 8
    Top = 8
    Width = 75
    Height = 25
    Caption = 'Button1'
    TabOrder = 0
  end
```

如果把object Button1: TButton这一行改为“ object Button1: TLabel ”，那么组件的类型就被改为TLabel。当打开这个窗体时，将会看到一个标签而不是一个按钮。

注意 在窗体文件中修改组件类型可能会导致错误。例如，TButton原来有TabOrder属性，如果把TButton改为TLabel，由于TLabel没有TabOrder属性，这就会导致错误。不过，不用手工去更正它，因为当保存这个窗体时，Delphi会自动进行更正。

警告 编辑窗体文件时要特别小心。如果操作失误，可能会导致Delphi 5无法打开这个窗体文件。

注意 Delphi 5的新功能允许以文本文件格式保存窗体。这样就可以利用其他像记事本这样的通用工具来编辑窗体。只要在窗体上单击右键打开关联菜单，然后选择Text DFM命令。

4.2.4 资源文件

资源文件(.res)中包含了二进制数据，也称为资源，这些资源将链接到应用程序的可执行文件中。.res文件是Delphi 5自动创建的，包含应用程序的图标、应用程序版本信息及其他信息。要把资源加入到应用程序中，可以先创建一个单独的资源文件，然后把它链接到项目中。要创建资源文件，可以使用专门的资源编辑器，例如Delphi 5提供的Image Editor或Resource Workshop等。

警告 不要编辑由Delphi在编译时自动生成的资源文件。如果那样的话，下次编译时所做的修改有可能丢失。如果要在应用程序中加入其他资源，应当创建另外一个和项目文件不同名的资源文件，然后参照下面这一行把资源链接到项目中：

```
{$R MYRESFIL.RES}
```

4.2.5 项目选项及桌面设置文件

项目选项文件(扩展名为.dof)存储了Project | Options菜单命令所设置的项目选项。它是在第一次保存项目时创建的，以后每次保存项目时都会保存这个文件。

桌面设置文件(扩展名为.dsk)存储了Tools | Options菜单命令所设置的桌面选项。桌面设置与项目选项不同，项目选项与具体项目有关，而桌面设置作用于 Delphi 5环境。

提示 错误的.dsk或.dof文件在编译时可能导致像GPF这样不可预测的错误。如果真的出现这种情况，应当把.dof和.dsk文件都删除掉。当保存项目或退出Delphi 5时会重新生成这两个文件。IDE和项目又恢复到默认设置。

4.2.6 备份文件

自第二次保存开始， Delphi 5为项目文件和PAS单元文件生成备份文件。备份文件是上次保存的文

件的副本。项目文件的备份文件的扩展名是.~dp。单元文件的备份文件的扩展名是.~pa。

窗体文件的二进制备份文件也是在第二次保存时创建的。窗体文件的备份文件的扩展名是 .~df。

删除备份文件一般不会有什么问题。如果不想生成备份文件，可以在 Editor Properties对话框的Display页上不选中Create Backup File选项。

4.2.7 包文件

包类似于动态链接库，它的代码可以被几个应用程序共享。不过，包是 Delphi特有的，用于共享组件、类、数据和代码。 把组件放到包中，而不是直接链接到应用程序中，可以大大减少应用程序的长度。后面的章节将进一步介绍包。包的源文件的扩展名是 .dpk(Delphi package的缩写)，编译后就会生成一个.bpl文件(一个.bpl文件类似于一个动态链接库)。这个.bpl文件由若干个单元或 .dcu(Delphi compiled units的缩写)文件组成。与源文件对应的中间文件其扩展名是 .dcp(Delphi compiled package的缩写)。这些内容如果一时搞不清楚就先不要管它，我们会在后面详细介绍。

4.3 项目管理提示

通过良好的组织和代码重用，可以优化开发过程。下面提供一些有关项目管理的建议。

4.3.1 一个项目一个目录

应当把一个项目中的文件与另一个项目中的文件分开。这样可以避免一个项目的文件覆盖另一个项目的文件。

本书附带的光盘中每个项目都是单独一个目录。你最好也把一个项目放在一个单独的目录中。

文件名约定

应当规划好一个项目中文件的命名约定。可以参考光盘中的“ DDG Coding Standrad Do ment ”以及本书中用到的项目(参见第6章“ 代码标准文档 ”)。

4.3.2 共享代码的单元

最好把那些需要被其他应用程序共享的例程，放到一个单独的单元中。一般地，先在磁盘中创建一个目录，然后把需要共享的单元放到这个目录中。当一个项目要共享其中的某个单元时，只要把那个单元的名称加到uses子句中就行了。

另外，必须把共享单元所在的目录加到 Project Options对话框的Directories/Conditionals页上的Search Path框中。这样， Delphi 5就知道到哪里找这个单元。

提示 通过项目管理器，可以把其他目录中的单元加到当前项目中，Delphi会自动添加搜索路径中。

为了解释如何使用共享单元，清单4-1列出了一个短小的单元文件StrUtils.pas，其中定义了一个简单的字符串处理函数。实际上，一个单元可能要包含多个例程，这里仅仅用来举个例子。注释部分解释了函数的用途。

清单4-1 StrUtils.pas单元

```
unit strutils;
interface
function ShortStringAsPChar(var S: ShortString): PChar;
implementation
```

```
function ShortStringAsPChar(var S: ShortString): PChar;
{ 该函数使一个短字符串以null结束，这样它可以传递给需要PChar类型的函数。
如果字符串长度超过254个字符，则被截为254个字符
}
begin
  if Length(S) = High(S) then Dec(S[0]); { 如果S太长，就截掉多余的部分 }
  S[Ord(Length(S)) + 1] := #0;           { 在字符串的末尾加上null }
  Result := @S[1];                       { 返回PChar化的字符串 }
end;
end.
```

假设有一个单元SomeUnit.pas，需要使用这个ShortStringAsPChar()函数。

那么只要把StrUtils加到这个单元的uses子句。例如：

```
unit SomeUnit;
interface
...
implementation
uses
  strutils;
...
end.
```

还要使用Project | Option菜单命令把StrUtils单元所在的目录加到搜索路径中，以确保Delphi 5能够找到这个单元。

这样，就可以在SomeUnit.pas单元的Implementation部分使用ShortStringAsPChar()函数。必须把StrUtils加到所有需要使用ShortStringAsPChar()函数的单元的uses子句中。如果要想在整个应用程序中都能使用这个函数，只把StrUtils加到一个项目的一个单元甚至项目文件中是不够的。

提示 因为ShortStringAsPChar()函数比较实用，值得把它放到一个共享单元中，以便其他应用程序重用，这样就不必记住曾经在哪里使用过它。

1. 全局标识符单元

一个单元可以专门用来声明全局标识符。前面提到过，一个项目通常由若干个单元组成，包括窗体的单元文件、组件的单元文件和通用的单元文件。如果需要声明一个所有单元都可以访问的变量，该怎么办？按照下面的步骤可以建立一个专门声明全局标识符的单元：

1) 在Delphi 5中创建一个单元。

2) 给这个单元起一个能够体现出它是用来声明全局标识符的名字，例如Globals.pas或者ProjGlob.pas。

3) 在这个单元的Interface部分声明变量、类型等。这些标识符将可被所有单元访问。

4) 为了使这些标识符能够被其他单元访问，把这个单元名称加到那些需要访问这个单元的 uses子句(就像本章前面关于共享代码中介绍的那样)。

2. 使一个窗体能够被其他窗体调用

每一个窗体都包含在一个单元文件中，这并不意味着它不能访问其他窗体的变量、属性和方法。Delphi会在窗体的单元文件中生成代码，声明窗体的一个全局变量实例。所有需要访问另一个窗体的单元，都要把该窗体的单元加到uses子句中。例如，假设窗体1的单元文件是Unit1.pas，它需要访问窗体2，对应的单元文件是Unit2.pas，只要把Unit2加到Unit1的uses子句。如下所示：

```
unit Unit1;
interface
...
implementation
```

```
uses
  Unit2;
...
end.
```

现在，Unit1的Implementation部分就可以引用Form2了。

注意 如果在Unit1中引用了Unit2的窗体(就叫它Form2)，那么编译项目时会问是否把Unit2加到Unit1的uses子句中；对于Unit1引用Form2这种情况，这是必须的。

4.3.3 多项目管理

通常，一个产品由几个项目组成 (这些项目可能互相依赖)。典型的例子是多层应用程序中的每一层，或者被其他项目调用的DLL，因为DLL本身也可能是一个独立的项目。

Delphi 5提供了管理项目组的功能。项目管理器允许把几个项目组织在一起，形成一个项目组。这里不想详细介绍项目管理器的使用，因为 Delphi的文档已经详细介绍过它。这里主要强调组织项目组是非常重要的，以及项目管理器是如何帮你实现项目组的。

在一个项目组中，每个项目的文件最好放在一个单独的目录中。所有需要共享的单元、窗体等都应当放在一个公共的目录中，以便被所有的项目共享。例如，下面是一个典型的目录结构：

```
\DDGBugProduct
\DDGBugProduct\BugReportProject
\DDGBugProduct\BugAdminTool
\DDGBugProduct\CommonFiles
```

根据上面的结构可以看出，有两个专门的目录各自存放一个项目： BugReportProject和BugAdminTool。这两个项目可能需要共享一些单元和窗体，这些单元和窗体应当放在 CommonFiles目录中。

项目管理在开发过程中是至关重要的，尤其是在一个团队开发环境中。强烈建议在整个小组投入开发之前建立一个标准，否则，以后的管理将很困难。可以使用项目管理器来建立项目组的结构。

4.4 Delphi 5项目的框架类

大多数 Delphi 5应用程序至少有一个 TForm的实例。 Delphi 5 VCL应用程序也只能有一个TApplication的实例和一个TScreen的实例。这三个类在Delphi 5中扮演着重要的角色。下面几节就介绍这几个类，掌握了这些知识后就可以根据需要修改它们的缺省功能。

4.4.1 TForm 类

TForm类是Delphi 5应用程序的焦点。大多数情况下，整个应用程序都是围绕着主窗体转的。从主窗体，可以打开其他窗体，通常要通过菜单命令或按钮。也可以让 Delphi 5自动创建窗体，这样就不用操心什么时候创建、什么时候删除窗体。也可以选择在运行期间动态创建窗体 。

注意 Delphi可以创建不使用窗体的程序(例如控制台程序、服务和COM服务器)。这种情况下，TForm就不再是应用程序的焦点。

显示给用户的窗体有两种：有模式的和无模式的。具体使用哪一种窗体，取决于是否希望用户能够同时与这个窗体和其他窗体交互。

1. 显示一个模式窗体

当打开一个模式窗体后，用户无法与应用程序的其他部分交互，直到用户关闭了这个窗体。模式窗体通常是对话框，就好像 Delphi 5本身的对话框一样。事实上，大多数情况下应当使用模式窗体。要显示一个模式窗体，只要调用 ShowModal()就可以了。下面的代码演示了怎样创建 TmodalForm的实

例，并把它作为模式窗体打开：

```
Begin
  //创建TModalForm的实例
  ModalForm := TModalForm.Create(Application);
try
  if ModalForm.ShowModal = mrOk then    //显示这个窗体
   {do something};                      //执行一些代码
finally
  ModalForm.Free;                       //释放窗体的实例
  ModalForm := nil;                     //把窗体变量设为nil
  end;
end;
```

上面的代码演示了怎样动态创建TModalForm的实例以及把实例赋值给ModalForm变量。注意：如果需要动态创建一个窗体的实例，就要把这个窗体从Project Options对话框的Auto-Create框中去掉。如果窗体的实例已经存在，可以调用ShowModal()函数打开它，其他代码可以删掉，变成下面这样：

```
begin
  if ModalForm.ShowModal = mrOk then      // 如果ModalForm已经存在
    { do something };
end;
```

ShowModal()函数的返回值是ModalForm的ModalResult属性值。缺省情况下，ModalResult属性的值为0，相当于预定义的常量mrNone。如果ModalResult属性被赋值为其他非零值，则窗体将关闭。

可以在运行时对窗体的ModalResult属性进行赋值：

```
begin
  ModalForm.ModalResult := 100; // Assigning a value to ModalResult
  // 使窗体关闭
end;
```

表4-1列出了预定义的ModalResult值。

表4-1 ModalResult值

常 量	值
mrNone	0
mrOk	idOk
mrCancel	idCancel
mrAbort	idAbort
mrRetry	idRetry
mrIgnore	idIgnore
mrYes	idYes
mrNo	idNo
mrAll	mrNo+1

2. 打开无模式窗体

要打开一个无模式窗体，可以调用Show()。无模式窗体与有模式窗体的区别是，用户可以在无模式窗体和其他窗体之间切换。这样，用户就可以同时工作于一个应用程序的几个部分。下面的代码演示了怎样动态创建一个无模式的窗体：

```
Begin
// 检查无模式窗体的实例是否已经存在
  if not Assigned(Modeless) then
    Modeless := TModeless.Create(Application);  // 创建窗体
```

```
    Modeless.Show                                    // 显示无模式窗体
  end;                                               // 实例已经存在
```

上述代码同时演示了怎样防止一个窗体的多个实例存在。记住，无模式的窗体允许用户与应用程序的其他部分交互。这样，用户可以照常使用菜单命令，或者创建 TModeless的另一个实例。因此，需要考虑这些实例的创建和删除问题。

要特别注意窗体的实例：当通过窗体的系统菜单或者窗体上的 Close按钮关闭这个窗体时，窗体并没有真正从内存中释放。它仍然还在内存中，除非关闭了主窗体 (即应用程序)。在上面这个程序示例中，then后面的语句只会执行一次，前提是这个窗体不是自动创建的。如果希望用户关闭了窗体就在内存中释放它，必须处理它的 OnClose事件，并且把Action参数设为caFree，这样，VCL就会在这个窗体关闭时释放它。

```
procedure TModeless.FormClose(Sender: TObject;
  var Action: TCloseAction);
begin
  Action := caFree;  // 当关闭时，释放窗体实例
end;
```

上述代码解决了窗体实例在内存释放的问题。不过，还有一个问题，注意下面这行代码：

```
if not Assigned(Modeless) then begin
```

这行代码检查TModeless的实例是否已经由Modeless变量引用，这实际上就是检查Modeless是否为nil。尽管第一次进入例程的时候，Modeless可能是nil，但第二次进入这个例程的时候，它已经不是nil。这是因为VCL并没有把Modeless变量设为nil。因此，必须手工把这个变量设为nil。

与模式窗体不同的是，无法在代码中判断无模式窗体什么时候将删除。因此，无法在创建窗体实例的例程中删除窗体的实例。用户有可能在应用程序正在运行的任何时候关闭无模式窗体。因此，无模式窗体本身一定要把Modeless变量设为nil，而且最好在处理窗体的OnDestroy事件的处理过程中设置这个变量：

```
procedure TModeless.FormDestroy(Sender: TObject);
begin
  Modeless := nil; // 把Modeless变量设为nil
end;
```

这样就能保证每次关闭窗体时，Modeless变量总是被设为nil，从而防止Assigned()函数失败。记住，同一时刻只能创建TModeless的一个实例。

注意 对于无模式窗体来说，要避免出现下列有缺陷的代码：

```
begin
  Form1 := TForm1.Create(Application);
  Form1.Show;
end;
```

上述代码会导致每次都创建窗体的实例，重复了被 Form1引用的以前的实例，从而消耗了大量的内存。尽管通过 Screen.Forms可以访问这些实例，但最好还是尽量避免使用上述代码。向构造器Create( )传递nil造成无法在Form1实例变量被覆盖后，无法再引用这个窗体实例指针。

本书附带的光盘中有一个项目ModState.dpr，演示了怎样同时使用模式窗体和无模式窗体 .

3. 管理窗体的图标和边框

TForm有一个BorderIcons属性，它是一个集合，包含下列元素： biSystemMenu、biMinimize、biMaximize和biHelp。只要让这个集合不包含其中的某个元素，就可以使窗体上不出现相应的系统菜单、最大化按钮和最小化按钮，但窗体上总是有关闭按钮。

还可以通过BorderStyle属性改变窗体的非客户区。BorderStyle属性的定义如下：

```
TFormBorderStyle = (bsNone, bsSingle, bsSizeable, bsDialog,
➥bsSizeToolWin, bsToolWindow);
```

BorderStyle属性可以使窗体具有下列特征：

- bsDialog 不能重设大小，只有关闭按钮。
- bsNone 没有边框，不能重设大小，没有按钮。
- bsSingle 不能重设大小，有所有按钮。如果 biMinimize和biMaximize中只有一个按钮被设为False，那么窗体上有两个按钮。但设为False的按钮不可用。如果二者均为False，那么没有按钮在窗体上显示。如果biSystemMenu，则没有按钮显示。
- bsSizeable 有边框，有所有按钮，按钮情况与bsSingle一样。
- bsSizeToolWin 可以重设大小，只有关闭按钮和标题栏。
- bsToolWindow 不能重设大小，只有关闭按钮和标题栏。

注意 在设计时修改BorderIcon和BorderStyle属性并不立即反映出来。这些变化要到运行期才能看到效果。其实，TForm的大部分属性都是这样的，这是因为在设计时修改窗体的外观没有多大意义。例如，假设把Visible属性设为False，如果窗体不再显示出来，那么就无法操作窗体上的组件。

粘上标题！

你可能注意到了，上面的选项没有一个选项能够创建一个没有标题但可以重设大小的窗体。要实现这种特殊窗体，需要覆盖窗体的CreateParams()方法，然后设置相关的风格。下面的代码演示了这一点：

```
unit Nocapu;
interface
uses
  SysUtils, WinTypes, WinProcs, Messages, Classes,
  Graphics, Controls, Forms, Dialogs;
type
  TForm1 = class(TForm)
public
    { override CreateParams method }
    procedure CreateParams(var Params: TCreateParams); override;
end;
var
  Form1: TForm1;
implementation
{$R *.DFM}
procedure TForm1.CreateParams(var Params: TCreateParams);
begin
  inherited CreateParams(Params);  { Call the inherited Params }
  { 设置Form的风格 }
  Params.Style := WS_THICKFRAME or WS_POPUP or WS_BORDER;
end;
end.
```

第21章“编写自定义组件”将进一步介绍CreateParams()的用法。

本书附带的光盘中有一个NoCaption.dpr项目，演示了无边框但可重设大小的窗体，同时还演示了怎样捕捉WM_NCHITTEST消息，这样，即使是一个没有标题栏的窗体，也可以通过拖动窗体本身移动它。

注意光盘中的BrdrIcon.dpr项目。该项目演示了怎样在运行时修改 BorderIcon和BorderStyle属性。清单4-2列出了这个项目主窗体单元的有关代码：

清单4-2 BorderStyle/BorderIcon项目的主窗体单元

```
unit MainFrm;

interface

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

type
  TMainForm = class(TForm)
    gbBorderIcons: TGroupBox;
    cbSystemMenu: TCheckBox;
    cbMinimize: TCheckBox;
    cbMaximize: TCheckBox;
    rgBorderStyle: TRadioGroup;
    cbHelp: TCheckBox;
    procedure cbMinimizeClick(Sender: TObject);
    procedure rgBorderStyleClick(Sender: TObject);
  end;

var
  MainForm: TMainForm;

implementation

{$R *.DFM}

procedure TMainForm.cbMinimizeClick(Sender: TObject);
var
  IconSet: TBorderIcons;  // 用于存放值的暂时变量
begin
  IconSet := [];  // 初始化为空集
  if cbSystemMenu.Checked then
    IconSet := IconSet + [biSystemMenu]; // 为窗体增加系统菜单按钮
  if cbMinimize.Checked then
    IconSet := IconSet + [biMinimize];   // 为窗体增加最小化按钮
  if cbMaximize.Checked then
    IconSet := IconSet + [biMaximize];   // 为窗体增加最大化按钮
  if cbHelp.Checked then
    IconSet := IconSet + [biHelp];

  BorderIcons := IconSet;                // 为BorderIcons属性赋值
end;

procedure TMainForm.rgBorderStyleClick(Sender: TObject);
begin
  BorderStyle := TBorderStyle(rgBorderStyle.ItemIndex);
end;

end.
```

注意 TForm的部分属性用于设置窗体的外观，部分属性用于设置窗体的行为。要想进一步了解TForm的属性，请查找Delphi 5帮助系统。

4. 窗体重用性：可视化窗体继承

Delphi 5的一个重要功能就是可视化窗体继承。在Delphi的第一个版本中，可以创建一个窗体，然

后把它保存为模板，以后可以在模板的基础上创建新的窗体。但这并不是真正的继承，因为无法访问祖先窗体的组件、方法和属性。而对于真正的继承来说，派生的窗体与祖先窗体可以共享相同的代码。继承的优势在于，应用程序可以做得比较精巧。另外，当祖先窗体的代码改变时，派生窗体会跟着改变。

对象库

Delphi 5提供了对象管理功能，允许程序员共享窗体、对话框、数据模块和项目模板。这个功能称为“对象库(Object Kepository)”。通过对象库，开发者可以共享其他项目的对象。另外，通过继承对象库中已有的对象，可以最大程度地实现代码重用。第4章讲到了对象库。最好要熟悉对象库的功能。

提示 在网络环境中，可能要与其他程序员共享窗体模板，这时需要创建一个共享的库。在Environment Options对话框中，可以指定共享库的位置。每一个程序员必须把一个网络驱动器映射到这个位置。以后当程序员使用File | New菜单命令时，Delphi就会检查共享库所在的目录。

继承另一个窗体是很简单的，因为这已成为Delphi 5环境内置的功能。要基于一个已有的窗体创建一个新的窗体，只要使用File | New菜单命令，Delphi将打开New Items对话框。这个对话框列出了对象库中的所有对象。翻到Forms页，这里列出了所有已经加到对象库中的窗体。

注意 不必在对象库中查找就能继承一个本项目中的窗体。使用File | New菜单命令，然后选择Project页。在那里，可以选择一个本项目中已有的窗体。显示在Project页中的窗体并不在对象库中。

有一些列出的窗体就是以前加入到对象库中的。可能注意到，有三个选项用于把窗体加到项目中：Copy、Inherit和Use。

如果选择Copy，则意味着把所选窗体的副本加到当前项目中。如果对象库中的窗体发生变化，不会影响到当前项目中的副本。

如果选择Inherit，则意味着从所选窗体派生出一个新的窗体加到当前项目中。如果对象库中的窗体发生变化，则派生的窗体也会跟着变化。

如果选择Use，则意味着所选的窗体直接加到当前项目中，就好像这个窗体是当前项目创建的一样。在设计时对这个窗体的任何修改都会影响以Inherit方式使用这个窗体的项目。

4.4.2 TApplication类

任何基于窗体的Delphi 5程序都包含一个全局变量Application，它的类型是TApplication。TApplication封装了一些属性和方法，使应用程序能够正确地在Windows环境下运行。这些方法中，有的用于建立窗口类定义，有的用于创建应用程序的主窗口、激活应用程序、处理消息、添加上下文敏感的帮助以及处理VCL的异常。

注意 只有基于窗体的Delphi应用程序才有全局Application对象，而控制台程序和服务程序没有Application对象。

一般不需要关心TApplication在背后到底做了些什么，不过，有时需要了解TApplication的详细情况。

由于TApplication并不在Object Inspector中出现，所以不能在设计时修改它的属性，但可以用Project | Options菜单命令，翻到Application页，设置一些有关TApplication的属性。在大多数情况下，只能在运行时工作于TApplication的实例即Application。也就是说，只能在运行时设置Application的属

性、方法和事件过程。

1. TApplication的属性

TApplication具有几个属性，可以在运行时访问它们。下面将介绍这些属性以及怎样通过它们改变Application的默认行为。这些属性在Delphi 5的在线帮助中也有介绍。

(1) TApplication.ExeName属性

ExeName属性能够返回应用程序的全路径和文件名。这个属性在运行时是只读的，不能修改它。但是可以读它，以使用户知道应用程序是从哪儿运行的。例如，下面的代码把 ExeName属性的值显示在主窗体的标题栏上：

```
Application.MainForm.Caption := Application.ExeName;
```

提示 使用ExtractFileName()函数可以从ExeName属性中得到文件名：

ShowMessage(ExtractFileName(Application.ExeName));

使用ExtractFilePath()函数可以从ExeName属性中得到全路径：

ShowMessage(ExtractFilePath(Application.ExeName));

使用ExtractFileExt()函数可以从ExeName属性中得到文件扩展名：

ShowMessage(ExtractFileExt(Application.ExeName));

(2) TApplication.MainForm属性

在前面的例子中，可以看到怎样访问 MainForm来修改它的 Caption属性以显示应用程序的ExeName值。MainForm的类型是TForm，可以通过MainForm访问TForm的任何属性和方法。也可以访问加到派生窗体的属性，只要把MainForm强制转换为该窗体的类型：

```
(MainForm as TForm1).SongTitle := 'The Flood';
```

MainForm是一个只读的属性。只能在设计时通过 Project Options对话框上的Forms页把一个窗体指定为主窗体。

(3) TApplication.Handle属性

Handle属性是一个 HWND(一个用于 Win32 API的窗口句柄)。一般情况下，用不着访问Handle属性，除非要修改Application的默认行为，而Delphi又没有提供相应的方法。此外，调用某些Win32 API时可能也需要用到 Handle属性，因为那些 API需要传递应用程序的窗口句柄。后面将更详细地讨论Handle属性。

(4) TAppllcation.Icon和 TApplication.Title属性

Icon属性用于设置当应用程序最小化时代表应用程序的图标。可以修改 Icon属性来改变应用程序的图标。后面的4.6.1节“ 在项目中添加资源 ”将详细介绍这一点。

在Windows 95/98的任务栏上，显示在图标右边的文字通过Title属性设置。如果应用程序在Wndows NT下运行，则文字显示在图标下面。下面的代码演示了怎样修改 Title属性：

```
Application.Title := 'New Title';
```

(5) 其他属性

Active是一个只读的属性，它的值表明应用程序是否激活和具有输入焦点。

ComponentCount属性表明应用程序所包含的组件数，如果 Application.ShowHint属性设为 True的话，那么这些组件主要是指窗体和 THintWindow实例(即提示窗口)。对于那些没有拥有者(owner)的组件来说，ComponentIndex属性总是 -l。因此，TApplication.ComponentIndex总是 -1。这个属性主要用于窗体和窗体中的组件。

Components属性是一个数组，它的元素就是那些属于 Application的组件。Components数组的元素个数就是TApplication.ComponentCount属性的值。下面的代码演示了在一个列表框中列出所有组件的类名：

```
var
  i: integer;
begin
  for i := 0 to Application.ComponentCount - 1 do
    ListBox1.Items.Add(Application.Components[i].ClassName);
end;
```

HelpFile属性用于指定帮助文件的文件名。需要向TApplication的HelpContext方法以及其他类似的方法传递帮助文件的文件名。

TApplication.Owner属性总是nil，因为TApplication不可能被另个组件拥有。

ShowHint属性用于设置是否允许显示提示条。Application.ShowHint属性覆盖其他组件的ShowHint值。如果Application.ShowHint属性设为False，则所有组件的提示条都不会显示。

如果应用程序的主窗体被关闭，或者调用了TApplication.Terminate()，则Terminated属性为True。

4.4.3 TApplication的方法

应当熟悉TApplication的有些方法。下面就讨论这些方法。

1. TApplication.CreateForm()方法

TApplication.CreateForm()是这样声明的：

```
procedure CreateForm(InstanceClass: TComponentClass; var Reference)
```

这个方法用于创建一个窗体的实例，InstanceClass参数用于指定这个窗体的类，创建的实例由Reference参数返回。在项目文件中你可能看到过CreateForm的用法。例如，下面的代码创建TForml类型的实例Form1：

```
Application.CreateForm(TForm1, Form1);
```

如果Form1出现在Project Options对话框的Auto-Create列表中，Delphi 5会自动生成上面这一行代码。不过，对于那些没有出现在Auto-Create列表中的Form，可以在程序的任何一个地方调用CreateForm()来创建它的实例。其实，CreateForm()相当于窗体本身的Create()，但TApplication.CreateForm()会检查MainForm属性是否为nil。如果是的话，CreateForm()会把新创建的窗体作为主窗体。一般情况下，不要调用CreateForm()，而要调用窗体本身的Create()。

2. TApplication.HandleException()方法

HandleExcePtion()用于显示项目中出现的异常的有关信息。信息将显示在一个由VCL定义的标准的异常信息框中。如果希望用自己的方式来显示异常信息，可以响应Application.OnException事件。后面的4.6.5节“覆盖应用程序的异常处理”将详细介绍。

3. TApplication的HelpCommand()、HelpContext()和HelpJump()方法

HelpCommand()、HelpContext()和HelpJump()分别提供了一种让应用程序与WINHELP.EXE程序提供的帮助系统交互的方式。其中，HelpCommand()用于执行一条WinHelp宏命令和帮助文件中定义的宏；HelpContext()用于打开一个帮助主题，主题的编号由Context参数传递；HelpJump()类似于HelpContext()，但它的JumpID参数需要传递一个字符串。

4. TApplication.ProcessMessages()方法

ProcessMessages()用于从Windows消息队列中检索任何等待处理的消息并进行处理。如果程序正在执行一个很长的循环而又不希望中断其他代码的执行（诸如响应“放弃”按钮），这时候就要用到ProcessMessages()。相反，TApplication.HandleMessages()如果发现没有消息，它就使应用程序处于空闲状态，而ProcessMessages()则不会使应用程序处于空闲状态。在第10章中将用到ProcessMessages()方法。

5. TApplication.Run()方法

Delphi 5会自动把Run()放到项目文件的主代码块中。不要自己去调用这个方法，但需要知道这个方法到底干了些什么。首先，TApplication.Run()建立一个退出过程，以保证当应用程序退出运行时所有的组件都会得到释放。然后，它就建立一个循环来处理消息，直到应用程序终止。

6. TApplication.ShowException()方法

ShowException()需要传递一个异常类作为参数，它将显示有关该异常的信息的消息框。后面的4.6.5节“覆盖应用程序的异常处理”将详细介绍ShowException()。

7. 其他方法

TApplication.Create()用于创建TApplication的实例。不过，这个方法是 Delphi 5内部调用的，应用程序本身不应调用它。

TApplication.Destroy()用于删除TApplication的实例。不过，这个方法是 Delphi 5内部调用的，应用程序本身不应调用它。

TApplication.MessageBox()用于打开一个Windows消息框。不过，这个方法不需要像 Windows的MessageBox()函数那样传递窗口句柄。

TApplication.Minimize()用于把应用程序的主窗口最小化。

TApplication.Restore()用于把应用程序的主窗口恢复为最大化或最小化之前的大小。

TApplication.Terminate()用于终止应用程序的执行。与 Halt()不同的是，Terminate()会隐含调用PostQuitMessage()看看还有什么消息要处理。

注意 调用TApplication.Terminate()可以终止应用程序。Terminate()会调用Windows的PostQuitMessage()函数向应用程序的消息队列中发一个消息。VCL据此释放应用程序创建的所有对象。要说明的是，并不是一调用Terminate()就马上使应用程序终止，而是当应用程序检索到WM_QUIT消息时才会真正终止。而Halt()立即终止应用程序的执行，但不释放先前创建的对象，也不会返回到调用Halt()的地方。

4.4.4 TApplication的事件

TApplication有一些事件，可以建立处理这些事件的处理方法。在旧版本 Delphi中，TApplication的事件在Object Inspector上是找不到的(例如针对组件面板上的组件和窗体的事件)。要建立处理一个事件的处理方法，必须先声明一个作为处理方法的方法，然后在运行期动态地把此方法赋给某个事件属性。后面将以 OnException事件为例说明怎样建立处理 TApplication事件的处理方法。表 4-2列出了TApplication的所有事件。

表4-2 TApplication和TApplication Events的事件

事 件	描 述
onActivate	当应用程序被激活时将触发这个事件。相应地，当应用程序被挂起 (例如切换到了其他应用程序)时将触发onDeactivate事件
onException	当一个未处理的异常发生时，将触发这个事件。你可以为未处理的异常添加默认处理过程
onHelp	当用户请求帮助的时候将触发这个事件。例如，用户按下F1键，或者程序调用HelpCommand()、HelpContext()或Helpjump()
onMessage	当应用程序接收到一个消息时将触发这个事件。特别要注意，所有的消息都会触发这个事件，因此，这个事件有可能造成瓶颈
onHint	当鼠标指向某个控件时将触发这个事件，这样就可以显示提示信息
onIdle	当应用程序进入空闲状态时将触发这个事件。处于空闲状态后，需要收到一个消息才能把应用程序唤醒

本章的后续内容以及其他章节还将使用TApplication。

注意 TApplication.OnIdle事件适合于做一些不需要用户交互的动作，例如，在应用程序空闲的时候根据应用程序的状态更新菜单栏和工具栏。

4.4.5 TScreen类

TScreen类封装了有关应用程序运行于的屏幕的状态信息。 TScreen既不能作为组件加到窗体上，也不能在运行时动态地创建它。 Delphi 5会自动创建一个TScreen类型的全局变量叫Screen。TScreen的有些属性是很有用的，这些属性如表4-3所列。

表4-3 TScreen的属性

ActiveControl	这是一个只读的属性，表明当前屏幕上哪个控件具有焦点。当焦点从一个控件切换到另一个控件时，ActiveControl将在失去焦点的哪个控件发生onExit事件之前切换为新的控件
ActiveForm	表明屏幕上哪个窗体具有焦点。不管是在同一个应用程序内还是不同的应用程序之间切换，都会使这个属性发生变化
Cursor	这个属性用于设置整个应用程序的光标形状。它的默认值是CrDefault。每个控件有自己的Cursor属性，可以单独修改。不过，如果TScreen的Cursor属性设为其他值，则所有控件的光标形状都会跟着修改，除非 ScreenCursor又恢复为crDefault。换句话说，ScreenCursor=crDefault意味着每个控件都可以自己设置光标形状
Cursors	这是一个列表，列出了屏幕所支持的各种光标形状
DataModuleCount	应用程序中数据模块的个数
DataModules	应用程序的数据模块的列表
FormCount	应用程序中窗体的个数
Forms	应用程序中的窗体的列表
Fonts	这是一个列表，列出了屏幕所支持的各种字体名称
Height	屏幕的高度(以像素为单位)
PixelsPerlnch	表示系统字体的相对缩放比例
Width	屏幕的宽度(以像素为单位)

4.5 定义公共体系结构：使用对象库

Delphi使应用程序的开发变得容易，以前，要花费很大的精力用于建立应用程序的体系结构，但现在可以轻松多了。问题是，很多开发者往往急于写代码而很少考虑应用程序的结构，这使得一个项目往往以失败而告终。

4.5.1 考虑应用程序的体系结构

本书不打算专门讲述体系结构或面向对象的分析和设计。不过，我们认为这是非常重要的。附录C“参考读物”列出了一些关于面向对象的主题。在开始编写代码之前最好先阅读附录C的内容。

下面列出了一些应当考虑的问题：

- 体系结构支持代码重用吗？
- 应用程序中的模块、对象等能够本地化吗？
- 修改体系结构非常容易吗？
- 用户界面和后端可以本地化吗？

• 体系结构支持团队开发吗？或者说，团队的成员可以工作于各自的模块吗？

上面这几个问题其实只是开发过程中要考虑的一部分问题。

关于相关内容的书籍很多，我们无意与它们竞争。下面将举例说明怎样设计一个数据库应用程序的通用用户界面。

4.5.2 Delphi固有的体系结构

你可能经常听到这样一句话，即作为一个Delphi开发者，没必要是一个组件编写者。尽管这句话是正确的，但下面这句话也是正确的：如果你是一个组件编写者，就一定是一个更优秀的Delphi开发者。

这是因为，组件编写者清楚地知道对象模式和 Delphi应用程序的体系结构，这意味着组件编写者能够更好地发挥它们的优势。你可能已经听说过，事实上 Delphi本身就是用组件编写的。 Delphi本身就是一个运用体系结构的例子。

即使并不想编写一个组件，但掌握体系结构还是有好处的。应当像熟悉 Win32操作系统那样熟悉VCL和Object Pascal模型。

4.5.3 体系结构的例子

为了证明窗体继承以及对象库的能力，下面将定义一个通用的应用程序体系结构。重点是代码重用性、修改的灵活性、一致性和易于团队开发。

窗体继承，更准确地说是框架，它们的典型应用是在数据库应用程序中。窗体应当对数据库的操作(编辑、添加或浏览)具有感知能力。窗体还应当包含一些通用控件，例如工具栏和状态栏，以便对数据库表进行操作。这些控件随窗体状态变化。另外，这些窗体还应当提供事件，以便跟踪窗体模式的变化。

应用程序的框架应当允许团队开发，每个成员可以各自工作于应用程序的一部分，而不至于出现重复和覆盖。

框架分为3个层次，后面将详细介绍这3个层次。表4-4描述了框架中每个窗体的用途。

表4-4 框架中的数据库窗体

窗 体	用 途
TChildForm = class(TForm)	可以插入到一个窗口中作为子窗口
T DBModeForm = class(TChildForm)	能够感知数据库的状态 (浏览、插入和编辑)以及在数据库的状态变化时触发事件
TDBNavStatForm=class(TDBBaseForm)	典型的数据库导航窗体，它能感知数据库的状态，包含一个标准的导航栏和状态栏

4.5.4 子窗体TChildForm

TChildForm是那些能够被单独打开的模式窗口无模式窗体并能成为其他窗口的子窗口的基类。

TChildForm支持团队开发，每个成员可以工作于应用程序的一部分。同时， TChildForm也实现了漂亮的用户界面，用户可以在应用程序内打开一个窗体，作为一个单独的实体。清单 4-3是TChildForm的源代码。这些代码可以在光盘中的\Code目录找到。

清单4-3 TChildForm的源代码

```
unit ChildFrm;

interface
```

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

type

  TChildForm = class(TForm)
  private
    FAsChild: Boolean;
    FTempParent: TWinControl;
  protected
    procedure CreateParams(var Params: TCreateParams); override;
    procedure Loaded; override;
  public
    constructor Create(AOwner: TComponent); overload; override;
    constructor Create(AOwner: TComponent;
        AParent: TWinControl); reintroduce; overload;

    // 下面的方法必须覆盖，要么返回窗体的主菜单，要么返回 nil。

    function GetFormMenu: TMainMenu; virtual; abstract;
    function CanChange: Boolean; virtual;
  end;

implementation

{$R *.DFM}
constructor TChildForm.Create(AOwner: TComponent);
begin
  FAsChild := False;
  inherited Create(AOwner);
end;

constructor TChildForm.Create(AOwner: TComponent; AParent: TWinControl);
begin
  FAsChild := True;
  FTempParent := aParent;
  inherited Create(AOwner);
end;

procedure TChildForm.Loaded;
begin
  inherited;
  if FAsChild then
  begin
    align := alClient;
    BorderStyle := bsNone;
    BorderIcons := [];
    Parent := FTempParent;
    Position := poDefault;
  end;
end;

procedure TChildForm.CreateParams(var Params: TCreateParams);
Begin
  Inherited CreateParams(Params);
  if FAsChild then
```

```
    Params.Style := Params.Style or WS_CHILD;
end;

function TChildForm.CanChange: Boolean;
begin
  Result := True;
end;

end.
```

上述代码演示了下列技术：首先是重载，这是对 Object Pascal语言的扩展；其次是怎样使一个窗体成为另一个窗口的子窗口。

1. 提供第二个构造器

你可能注意到了，上述代码中声明了两个构造器 (constructor)。第一个构造器用于创建一个普通的窗体，它需要传递一个参数。第二个构造器需要传递两个参数，它重载了第一个构造器。如果要使窗体成为子窗口，应当使用第二个构造器。其中，AParent参数用于传递父窗口。注意，这里用了reintroduce指示符，这样编译器就不会发出警告了。

第一个构造器只是简单地把FAsChild变量设为False，以保证创建的是一个普通的窗体。第二个构造器把这个变量设为True，并且把FTempParent设为AParent参数的值，这个值将在Loaded()方法中作为父窗口。

2. 使一个窗体成为子窗口

要使一个窗体成为子窗口，有几件事情需要做。首先，要确保窗体的属性已经正确设置，正如在TChildForm.Loaded()中看到的那样。清单4-3的代码能保证窗体变成一个子窗口而不是一个对话框，这是通过把边框隐去来实现的。如果这个窗体只用做子窗口，可以在设计时设置这些属性。如果这个窗体有可能要用作一个普通的窗体，应在FAsChild变量设为True的情况下才设置这些属性。

还要覆盖CreateParams()，以告诉Windows把窗体作为子窗口。要实现这一点，需要把Params.Style属性设为WS_CHILD风格。

TChildForm并不只限于数据库应用程序。事实上，可以把它用在任何需要把窗体作为子窗口的场合。本书附带的光盘的\Form Framework目录中有一个项目ChildTest.dpr，这个项目演示了一个窗体既可以作为普通的窗体，也可以作为子窗口。

4.5.5 数据库基础模式窗体TDBModeForm

TDBModeForm是从TChildForm继承下来的。它能够感知数据库的状态 (浏览、插入、编辑)。TDBModeForm还提供了一个事件，以跟踪数据库状态的变化。

清单4-4列出了TDBModeForm的源代码。

清单4-4 TDBModeForm

```
unit DBModeFrm;

interface

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

type
```

```
  TFormMode = (fmBrowse, fmInsert, fmEdit);

  TDBModeForm = class(TChildForm)
  private
    FFormMode      : TFormMode;
    FOnSetFormMode : TNotifyEvent;
  protected
    procedure SetFormMode(AValue: TFormMode); virtual;
    function  GetFormMode: TFormMode; virtual;
  public
    property FormMode: TFormMode read GetFormMode write SetFormMode;
  published
    property OnSetFormMode: TNotifyEvent read FOnSetFormMode
       write FOnSetFormMode;

  end;

var
  DBModeForm: TDBModeForm;

implementation

{$R *.DFM}

procedure TDBModeForm.SetFormMode(AValue: TFormMode);
begin
  FFormMode := AValue;
  if Assigned(FOnSetFormMode) then
    FOnSetFormMode(self);
end;

function TDBModeForm.GetFormMode: TFormMode;
begin
  Result := FFormMode;
end;

end.
```

TDBModeForm的实现比较简单。尽管这里使用了一些目前还没有介绍的技术，但你应当能看出它的作用。首先，这里声明了一个枚举类型TFormMode，用于表示窗体的状态。其次，TDBModeForm提供了一个FormMode属性以及它的读写方法。关于属性和读写方法将在第21章“编写自定义组件”中详细介绍。

在本书附带光盘的\Form Framework中有一个项目FormModeTest.DPR，它演示了TDBModeForm的用法。

4.5.6 数据库导航/状态窗体TDBNavstatForm

TDBNavstatForm具有框架的许多功能。TDBNavstatForm中包含了一些用于数据库应用程序的通用组件。特别是，它包含一个导航栏和状态栏，能够随数据库的状态而发生变化。例如，当数据库处于fsBrowse状态，导航栏上的Accept按钮和Cancel按钮就被禁止。当用户使数据库进入 fsInsert状态或fsEdit状态，这两个按钮将生效。状态栏上将显示数据库的状态。

下面的清单4-5列出了TDBNavstatForm的源代码。注意，这里去掉了组件的列表。当打开这个范例项目时会看到这些列表。

这里主要处理了一些TToolButton组件的事件，用于设置窗体的当前状态。这实际上是调用覆盖

SetFormMode()，再由SetFormMode()调用SetButtons()和SetStatusBar()实现的。SetButtons()能够根据窗体的状态来决定按钮的可用或不可用。

清单4-5 TDBNavstatForm

```
unit DBNavStatFrm;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  DBMODEFRM, ComCtrls, ToolWin, Menus, ExtCtrls, ImgList;

type
  TDBNavStatForm = class(TDBModeForm)
    { 组件没有在列表中列出 }
    procedure sbAcceptClick(Sender: TObject);
    procedure sbInsertClick(Sender: TObject);
    procedure sbEditClick(Sender: TObject);
  private
    { Private declarations }
  protected
    procedure Setbuttons; virtual;
    procedure SetStatusBar; virtual;
    procedure SetFormMode(AValue: TFormMode); override;
  public
     constructor Create(AOwner: TComponent); overload; override;
     constructor Create(AOwner: TComponent; AParent: TWinControl); overload;
     procedure SetToolBarParent(AParent: TWinControl);
     procedure SetStatusBarParent(AParent: TWinControl);
   end;

var
  DBNavStatForm: TDBNavStatForm;

implementation

{$R *.DFM}

{ TDBModeForm3 }

procedure TDBNavStatForm.SetFormMode(AValue: TFormMode);
begin
  inherited SetFormMode(AValue);
  SetButtons;
  SetStatusBar;
end;

procedure TDBNavStatForm.Setbuttons;

  procedure SetBrowseButtons;
  begin
    sbAccept.Enabled  := False;
    sbCancel.Enabled  := False;

    sbInsert.Enabled  := True;
    sbDelete.Enabled  := True;
```

```
    sbEdit.Enabled    := True;

    sbFind.Enabled    := True;
    sbBrowse.Enabled  := True;


    sbFirst.Enabled   := True ;
    sbPrev.Enabled    := True ;
    sbNext.Enabled    := True ;
    sbLast.Enabled    := True ;
  end;

  procedure SetInsertButtons;
  begin
    sbAccept.Enabled  := True;
    sbCancel.Enabled  := True;

    sbInsert.Enabled  := False;
    sbDelete.Enabled  := False;
    sbEdit.Enabled    := False;

    sbFind.Enabled    := False;
    sbBrowse.Enabled  := False;

    sbFirst.Enabled   := False;
    sbPrev.Enabled    := False;
    sbNext.Enabled    := False;
    sbLast.Enabled    := False;
  end;

  procedure SetEditButtons;
  begin
    sbAccept.Enabled  := True;
    sbCancel.Enabled  := True;

    sbInsert.Enabled  := False;
    sbDelete.Enabled  := False;
    sbEdit.Enabled    := False;

    sbFind.Enabled    := False;
    sbBrowse.Enabled  := True;

    sbFirst.Enabled   := False;
    sbPrev.Enabled    := False;
    sbNext.Enabled    := False;
    sbLast.Enabled    := False;
  end;

begin
  case FormMode of
    fmBrowse: SetBrowseButtons;
    fmInsert: SetInsertButtons;
    fmEdit:   SetEditButtons;
  end; { case }

end;
```

```
procedure TDBNavStatForm.SetStatusBar;
begin
    case FormMode of
      fmBrowse: stbStatusBar.Panels[1].Text := 'Browsing';
      fmInsert: stbStatusBar.Panels[1].Text := 'Inserting';
      fmEdit:   stbStatusBar.Panels[1].Text := 'Edit';
    end;

    mmiInsert.Enabled := sbInsert.Enabled;
    mmiEdit.Enabled   := sbEdit.Enabled;
    mmiDelete.Enabled := sbDelete.Enabled;
    mmiCancel.Enabled := sbCancel.Enabled;
    mmiFind.Enabled   := sbFind.Enabled;

    mmiNext.Enabled   := sbNext.Enabled;
    mmiPrevious.Enabled := sbPrev.Enabled;
    mmiFirst.Enabled  := sbFirst.Enabled;
    mmiLast.Enabled   := sbLast.Enabled;

 end;

 procedure TDBNavStatForm.sbAcceptClick(Sender: TObject);
 begin
   inherited;
    FormMode := fmBrowse;
 end;

 procedure TDBNavStatForm.sbInsertClick(Sender: TObject);
 begin
   inherited;
   FormMode := fmInsert;
 end;

 procedure TDBNavStatForm.sbEditClick(Sender: TObject);
 begin
   inherited;
   FormMode := fmEdit;
 end;

 constructor TDBNavStatForm.Create(AOwner: TComponent);
 begin
   inherited Create(AOwner);
   FormMode := fmBrowse;
 end;

 constructor TDBNavStatForm.Create(AOwner: TComponent; AParent: TWinControl);
 begin
   inherited Create(AOwner, AParent);
   FormMode := fmBrowse;
 end;

 procedure TDBNavStatForm.SetStatusBarParent(AParent: TWinControl);
 begin
   stbStatusBar.Parent := AParent;
 end;

 procedure TDBNavStatForm.SetToolBarParent(AParent: TWinControl);
```

```
begin
  tlbNavigationBar.Parent := AParent;
end;

end.
```

你可能注意到了，上面的代码中有两个过程用于修改 TToolBar组件和TStatusBar组件的父窗体。当窗体作为子窗口打开时，应当把 TToolBar组件和TStatusBar组件的父窗体设为主窗体。当运行随书附带光盘上\Form Framework目录中的项目时，就会知道这样做的意义。

正如前面提到的那样， TDBNavStatForm既可以是一个独立的窗体，也可以是一个子窗口。下面的代码演示了怎样把TDBNavstatForm作为一个独立的窗体调用：

```
procedure TMainForm.btnNormalClick(Sender: TObject);
var
  LocalNavStatForm: TNavStatForm;
begin
  LocalNavStatForm := TNavStatForm.Create(Application);
  try
    LocalNavStatForm.ShowModal;
  finally
    LocalNavStatForm.Free;
  end;
end;
```

下面的代码演示了怎样把TDBNavStatForm作为子窗口调用：

```
procedure TMainForm.btnAsChildClick(Sender: TObject);
begin
  if not Assigned(FNavStatForm) then
  begin
    FNavStatForm := TNavStatForm.Create(Application, pnlParent);
    FNavStatForm.SetToolBarParent(self);
    FNavStatForm.SetStatusBarParent(self);
    mmMainMenu.Merge(FNavStatForm.mmFormMenu);
    FNavStatForm.Show;
    pnlParent.Height := pnlParent.Height - 1;
  end;
end;
```

上面这个过程不仅把TDBNavStatForm作为pnlParent(TPanel组件)的子窗口，同时还使主窗体成为TToolBar组件和TStatusBar组件的父窗体。另外， TMainForm.mmMainMenu.Merge()这一行的作用是把TDBNavstatForm实例上的菜单合并到主窗体的菜单中。当然，当释放 TDBNavStatForm的实例时，必须这样调用TMainForm.mmMainMenu.UnMerge()：

```
procedure TMainForm.btnFreeChildClick(Sender: TObject);
begin
  if Assigned(FNavStatForm) then
  begin
    mmMainMenu.UnMerge(FNavStatForm.mmFormMenu);
    FNavStatForm.Free;
    FNavStatForm := nil;
  end;
end;
```

看一看随书光盘中的范例。图 4-1显示了TDBNavStatForm作为一个独立的窗体和作为一个子窗口的情况。实际上，这里可以用TImage组件来代替子窗口。

以后，我们将把这个框架扩展为功能齐全的数据库应用程序。

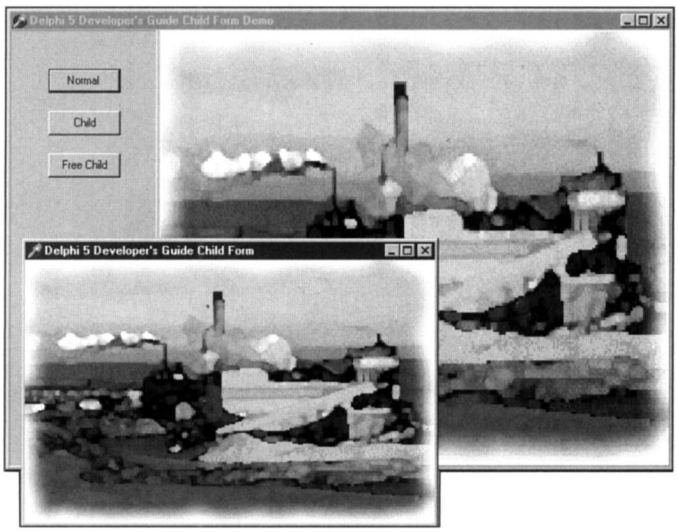

图4-1 TDBNavstatForm作为一个独立的窗体和作为一个子窗口

4.5.7 使用框架进行应用程序结构设计

Delphi 5提供框架功能，可以创建能被嵌入到其他窗体中的组件容器。这一点和我们用TChildForm进行的演示类似。然而，框架允许在设计时使用组件容器，并且可以把这些组件容器加到组件板中，以便将来重用。清单4-6演示了框架的功能。

清单4-6 框架演示

```
unit MainFrm;

interface

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

type
  TMainForm = class(TForm)
    spltrMain: TSplitter;
    pnlParent: TPanel;
    pnlMain: TPanel;
    btnFrame1: TButton;
    btnFrame2: TButton;
    procedure btnFrame1Click(Sender: TObject);
    procedure btnFrame2Click(Sender: TObject);
  private
    { Private declarations }
    FFrame: TFrame;
  public
    { Public declarations }
```

```
  end;

var
  MainForm: TMainForm;

implementation
uses Frame1Fram, Frame2Fram;

{$R *.DFM}

procedure TMainForm.btnFrame1Click(Sender: TObject);
begin
  if FFrame <> nil then
    FFrame.Free;
  FFrame := TFrame1.Create(pnlParent);
  FFrame.Align := alClient;
  FFrame.Parent := pnlParent;
end;

procedure TMainForm.btnFrame2Click(Sender: TObject);
begin
  if FFrame <> nil then
    FFrame.Free;
  FFrame := TFrame2.Create(pnlParent);
  FFrame.Align := alClient;
  FFrame.Parent := pnlParent;
end;

end.
```

在清单4-6的代码中，我们显示了一个主窗体，它包含两个由独立的面板构成的长方块。右边的面板用于包容框架。我们已经定义了两个独立的框架。在 private段，FFrame被定义为TFrame类型。由于两个框架都是直接来自TFrame，所以FFrame能够直接访问它们。位于主窗体上的两个按钮，各自创建一个不同的TFrame并且将其赋给FFrame。这和TChildForm的效果相同。可以在随书光盘中找到这个范例FrameDemo.dpr。

4.6 一些项目管理的功能

下面将介绍一些项目管理的功能，这对许多使用 Delphi 5的开发者是有帮助的。

4.6.1 在项目中添加资源

前面讲过.res文件是应用程序的资源文件，以及什么是Windows资源。要在项目中添加资源，可以创建一个单独的.res来存储要加到应用程序中的位图、图标、光标等资源。

必须使用专门的资源编辑器来创建.res文件。创建了.res后，只要在项目文件中加上下面这行语句，就能使资源链接到应用程序中：

```
{$R MYFILE.RES}
```

上面这行语句可以紧接在下面这行语句的后面。下面这行语句的作用是，把一个与项目文件同名的资源文件链接到应用程序中：

```
{$R *.RES}
```

如果已经这样做了，这时可以通过 TBitmap.LoadFromResourceName()或TBitmap.LoadFromReourceID()来调入资源文件中的资源。清单4-7演示了怎样从资源文件中调入一个位图、图标和光标。

可以在本书附带光盘的Resource.dpr项目中找到这些代码。注意，这里用到了一些Windows API函数，例如 LoadIcon()和 LoadCursor()，可以在 Windows API的帮助中找到它们的说明。

注意 Windows API中有一个LoadBitmap()函数，它可以调入一个位图，但它不能返回调色板，也就是说，它无法调入256色的位图。因此，建议使用 TBitmap.LoadFromResourceName()或TBitmap.LoadFromResourceID()。

清单4-7 从资源文件中调入资源的例子

```
unit MainFrm;
interface
uses
  Windows, Forms, Controls, Classes, StdCtrls, ExtCtrls;

const
  crXHair = 1; // 声明一个光标常量，其值必须是一个正数或 -20以内的负数
type

  TMainForm = class(TForm)
    imgBitmap: TImage;
    btnChemicals: TButton;
    btnClear: TButton;
    btnChangeIcon: TButton;
    btnNewCursor: TButton;
    btnOldCursor: TButton;
    btnOldIcon: TButton;
    btnAthena: TButton;
    procedure btnChemicalsClick(Sender: TObject);
    procedure btnClearClick(Sender: TObject);
    procedure btnChangeIconClick(Sender: TObject);
    procedure btnNewCursorClick(Sender: TObject);
    procedure btnOldCursorClick(Sender: TObject);
    procedure btnOldIconClick(Sender: TObject);
    procedure btnAthenaClick(Sender: TObject);
  end;

var
  MainForm: TMainForm;

implementation

{$R *.DFM}

procedure TMainForm.btnChemicalsClick(Sender: TObject);
begin
  { Load the bitmap from the resource file. The bitmap must be
    specified in all CAPS! }
  imgBitmap.Picture.Bitmap.LoadFromResourceName(hInstance, 'CHEMICAL');
end;

procedure TMainForm.btnClearClick(Sender: TObject);
begin
  imgBitmap.Picture.Assign(nil); // Clear the image
end;

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

```
begin
  { 从资源文件中调入位图资源，位图的标识符必须全部大写 }
  imgBitmap.Picture.Bitmap.LoadFromResourceName(hInstance, 'CHEMICAL');
end;

procedure TMainForm.btnClearClick(Sender: TObject);
begin
  imgBitmap.Picture.Assign(nil); // 清除图像
end;

procedure TMainForm.btnChangeIconClick(Sender: TObject);
begin
  { 从资源文件中调入图标，图标的标识符必须全部大写 }
  Application.Icon.Handle := LoadIcon(hInstance, 'SKYLINE');
end;

procedure TMainForm.btnNewCursorClick(Sender: TObject);
begin
  { 把一个新的光标加到Screen的Cursors数组中 }
  Screen.Cursors[crXHair] := LoadCursor(hInstance, 'XHAIR');
  Screen.Cursor := crXHair;  // 改变光标
end;

procedure TMainForm.btnOldCursorClick(Sender: TObject);
begin
  // 恢复为默认的光标
  Screen.Cursor := crDefault;
end;

procedure TMainForm.btnOldIconClick(Sender: TObject);
begin
  { 从资源文件中调入图标，图标的标识符必须全部大写 }
  Application.Icon.Handle := LoadIcon(hInstance, 'DELPHI');
end;

procedure TMainForm.btnAthenaClick(Sender: TObject);
begin
  { 从资源文件中调入位图，位图的标识符必须全部大写 }
  imgBitmap.Picture.Bitmap.LoadFromResourceName(hInstance, 'ATHENA');
end;

end.
```

4.6.2 改变屏幕光标

Cursor属性可能是TScreen最常用的属性之一，它的作用是改变应用程序的光标。例如，下面的代码把光标改为砂漏状，表示现在正在进行一个较长时间的操作。

```
Screen.Cursor := crHourGlass
{ 做一些较长时间的操作 }
Screen.Cursor := crDefault;
```

crHourGlass是一个预定义的常量，其他预定义的常量有 crBeam和crSize等。这些常量值的范围是从0到 -20(crDefault到crHelp)。可以从在线帮助中查找Cursors属性的详细说明，那里列出所有的光标

常量。要改变光标形状，只要把一个常量赋值给Screen.Cursor。

也可以创建一个自定义的光标，然后把它加到Cursors数组中。为此，必须声明一个光标常量，这个光标常量的值不能与已有的光标常量重复。预定义的光标常量的值是从0到 -20，而自定义的光标常量最好用正数，负数是Borland保留的。例如：

```
crCrossHair := 1;
```

可以使用资源编辑器(例如 Delphi 5附带的 Image Editor)来创建自定义的光标。创建的光标必须保存到一个资源文件中。要注意的是， Delphi 5会为一个项目自动创建一个资源文件。因此，自定义的资源文件不能与项目文件原有的资源文件重名。另外，自定义的资源文件要放在与项目文件相同的目录中，这样编译器才会找到这个资源文件。要使 Delphi 5能够把资源文件链接到应用程序中，可以参照下面这行语句：

```
{$R CrossHairRes.RES}
```

最后，可以参照下面的代码把自定义的光标调入，加到Cursors数组中，并指定使用这个光标：

```
procedure TMainForm.FormCreate(Sender: TObject);
begin
  Screen.Cursors[crCrossHair] := LoadCursor (hInstance, 'CROSSHAIR');
  Screen.Cursor := crCrossHair;
end;
```

这里使用了LoadCursor()函数来调入光标。LoadCursor()需要传递两个参数：一个是需要使用这个光标的模块的句柄，另一个是在.res文件中指定的光标的名字(必须全部大写)。

hInstance代表当前运行的应用程序。接着将从FormCreate返回的值赋给Cursor属性中由crCrossHair指定的位置。最后将当前光标赋给Screen.Cursor。

可以在光盘中找到一个叫作CrossHair.dpr的例子。这个例子从资源文件中调入一个光标，然后指定用这个光标。

如果需要的话，可以使用 Toolsl | Image Editor菜单命令打开 Image Editor，然后打开资源文件CrossHairRes.res，看看这个光标到底是怎么创建的的。

4.6.3 避免创建一个窗体的多个实例

如果使用Application.CreateForm()或TForm.Create()来创建窗体的实例，最好确保当前没有相同的实例存在。下面的代码演示了这一点：

```
begin
  if not Assigned(SomeForm) then begin
    Application.CreateForm(TSomeForm, SomeForm);
    try
      SomeForm.ShowModal;
    finally
      SomeForm.Free;
      SomeForm := nil;
    end;
  end
  else
    SomeForm.ShowModal;
end;
```

上面的代码中，必须在释放SomeForm变量后把它赋值为nil，否则， Assigned()函数将无法正常工作。不过，上面的代码不适用于无模式窗体，因为对于无模式窗体来说，程序代码并不知道什么时候删除窗体实例。因此，必须在处理onDestroy事件的处理方法中把窗体的实例赋值为nil。本章前面介绍过这种方法。

4.6.4 在DPR文件中增加代码

可以主窗体创建之前向项目文件中增加一些代码。这常用于做一些初始化工作，也可以根据需要终止应用程序。清单4-8列出了一个项目文件，它要求用户输入一个口令。项目文件 Initialize.dpr可以在光盘中找到。

清单4-8 演示项目初始化的Initialize.dpr文件

```
program Initialize;

uses
  Forms,
  Dialogs,
  Controls,
  MainFrm in 'MainFrm.pas' {MainForm};

{$R *.RES}

var
  Password: String;
begin
  if InputQuery('Password', 'Enter your password', PassWord) then
    if Password = 'D5DG' then
    begin
      // 其他初始化代码
      Application.CreateForm(TMainForm, MainForm);
    Application.Run;
    end
    else
      MessageDlg('Incorrect Password, terminating program', mtError, [mbok],
0);
end.
```

4.6.5 覆盖应用程序的异常处理

Win32系统具有强大的异常处理能力。缺省情况下，当一个异常发生时，应用程序会自动处理，并显示一个标准的错误框。

当开发一个大型的应用程序时，可能需要定义自己的异常。 Delphi 5默认的异常处理不能满足需要，因为应用程序往往需要对异常进行特殊的处理。这种情况下，需要覆盖 TApplication的默认异常处理，用自己的方法来代替默认的异常处理方法。

TApplication提供了一个OnException事件，可以响应这个事件并加入代码。当一个异常发生时，就会触发这个事件，这样就可以进行特殊的处理，同时，原有的标准错误框不会出现。

但是，由于TApplication的属性和事件都无法在Object Inspector上列出来，必须在应用程序中使用TApplicationEvents组件增加指定的异常处理方法。

清单4-9演示了怎样覆盖应用程序的默认异常处理。

清单4-9 演示覆盖异常处理的主窗体

```
unit MainFrm;

interface

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

```
  Forms, Dialogs, StdCtrls, AppEvnts, Buttons;

type
  ENotSoBadError = class(Exception);
  EBadError      = class(Exception);
  ERealBadError  = class(Exception);

  TMainForm = class(TForm)
    btnNotSoBad: TButton;
    btnBad: TButton;
    btnRealBad: TButton;
    appevnMain: TApplicationEvents;
    procedure btnNotSoBadClick(Sender: TObject);
    procedure btnBadClick(Sender: TObject);
    procedure btnRealBadClick(Sender: TObject);
    procedure appevnMainException(Sender: TObject; E: Exception);
  public
  end;

var
  MainForm: TMainForm;

implementation

{$R *.DFM}

procedure TMainForm.btnNotSoBadClick(Sender: TObject);
begin
  raise ENotSoBadError.Create('This isn''t so bad!');
end;

procedure TMainForm.btnBadClick(Sender: TObject);
begin
  raise EBadError.Create('This is bad!');
end;

procedure TMainForm.btnRealBadClick(Sender: TObject);
begin
  raise ERealBadError.Create('This is real bad!');
end;

procedure TMainForm.appevnMainException(Sender: TObject; E: Exception);
var
  rslt: Boolean;
begin
  if E is EBadError then
  begin
  { 显示一个自定义的消息框，提示应用程序将终止 }
    rslt := MessageDlg(Format('%s %s %s %s %s', ['An', E.ClassName,
         'exception has occurred.', E.Message, 'Quit App?']),
          mtError, [mbYes, mbNo], 0) = mrYes;
    if rslt then
       Application.Terminate;
  end
  else if E is ERealBadError then
  begin // 显示一个自定义的消息框，然后终止程序
    MessageDlg(Format('%s %s %s %s %s', ['An', E.ClassName,
```

```
        'exception has occured.', E.Message, 'Quitting Application']),
         mtError, [mbOK], 0);
      Application.Terminate;
    end
    else // 进行默认的处理
      Application.ShowException(E);
  end;

  end.
```

在清单4-9中，appevnMainException()方法对TApplicationEvent组件的OnException事件进行处理。首先使用RTTI检查异常的类型，然后各自进行特殊的处理。代码中的注释介绍了处理的过程。在本书附带光盘中可以找到项目onException.dpr。

提示 如果选中Debugger Options对话框(通过Options|Debugger Options菜单项进入)的Language Exceptions页上的Stop on Delphi Exceptions复选框，当一个程序在调试运行时，如果出现异常，调试器将先报告这个异常，应用程序中的异常处理再起作用。尽管对于调试这很有用，但在查看自定义的异常处理时这很烦人。关闭这个选项，让项目正常运行。

4.6.6 显示一个封面

假设要为项目创建一个封面。封面能够在应用程序启动时显示，并在应用程序初始化期间一直停留在屏幕上。显示封面是一件很简单的事情。

下面是创建一个封面的基本步骤：

1) 在创建了主窗体后，再创建一个窗体来作为封面。把这个窗体叫做 SplashForm。

2) 使用Project | Options菜单命令，确保 SplashForm没有出现在 Auto-Create列表中。

3) 把SplashForm的 BorderStyle属性设为 bsNone，BorderIcons属性设为[]。

4) 把一个TImage 组件放到SplashForm上，把它的 Align属性设为 alClient。

5) 选择Picture属性，在TImage组件中调入一个位图。

现在已经设计好这个封面了，只要在项目文件中加入代码来显示它。清单 4-10列出了一个项目文件，其中包含了显示封面的代码。在随书附带的光盘中可以找到这个项目 Splash .dpr。

清单4-10 一个带有封面的项目文件

```
program splash;

uses
  Forms,
  MainFrm in 'MainFrm.pas' {MainForm},
  SplashFrm in 'SplashFrm.pas' {SplashForm};

{$R *.RES}
begin
  Application.Initialize;
  { 创建封面 }
  SplashForm := TSplashForm.Create(Application);
  SplashForm.Show;    // 显示封面
  SplashForm.Update; // 强制更新封面

  { 下面通过一个定时器来延时 }

  while SplashForm.tmMainTimer.Enabled do
```

```
    Application.ProcessMessages;

  Application.CreateForm(TMainForm, MainForm);
  SplashForm.Hide; // 隐藏封面
  SplashForm.Free; // 释放封面实例
  Application.Run;
end.
```

注意代码中有这样一个循环：

```
while SplashForm.tmMainTimer.Enabled do
    Application.ProcessMessages;
```

这个循环是为了造成延时。窗体上有一个TTimer组件，它的Interval属性设为3000。当TTimer组件的OnTimer事件发生时，就执行下面这行代码：

```
tmMainTimer.Enabled := False;
```

上面这行代码使while循环的条件为False，并结束循环。

4.6.7 使窗体尺寸最小

为了说明怎样改变窗体的尺寸，下面将介绍一个项目，它的主窗体具有蓝色的背景，上面放有一个面板。当用户改变窗体的尺寸时，这个面板总是位于窗体的中心，并且不允许用户把 窗体的尺寸设得比面板还要小。清单4-11列出了有关代码。

清单4-11 模板窗体的源代码

```
unit BlueBackFrm;

interface

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

type
  TBlueBackForm = class(TForm)
    pnlMain: TPanel;
    bbtnOK: TBitBtn;
    bbtnCancel: TBitBtn;
    procedure FormResize(Sender: TObject);
  private
    Procedure CenterPanel;
    { 处理 WM__WINDOWPOSCHANGING消息}
    procedure WMWindowPosChanging(var Msg: TWMWindowPosChanging);
       message WM_WINDOWPOSCHANGING;
  end;

var
  BlueBackForm: TBlueBackForm;

implementation
uses Math;
{$R *.DFM}

procedure TBlueBackForm.CenterPanel;
{ 这个过程用于使面板总是在窗体的客户区的中心 }
```

```
begin
  { 水平居中 }
  if pnlMain.Width < ClientWidth then
    pnlMain.Left := (ClientWidth - pnlMain.Width) div 2
  else
    pnlMain.Left := 0;

  { 垂直居中 }
  if pnlMain.Height < ClientHeight then
    pnlMain.Top := (ClientHeight - pnlMain.Height) div 2
  else
    pnlMain.Top := 0;
end;

procedure TBlueBackForm.WMWindowPosChanging(var Msg: TWMWindowPosChanging);
var
  CaptionHeight: integer;
begin
  { 计算标题的高度 }
  CaptionHeight := GetSystemMetrics(SM_CYCAPTION);
  { 这个过程没有考虑窗体框架的高度和宽度，通过 GetSystemMetrics()可以获得这两个值 }
  // 使窗体的宽度不小于面板的宽度
  Msg.WindowPos^.cx := Max(Msg.WindowPos^.cx, pnlMain.Width+20);

  // 使窗体的高度不小于面板的高度
  Msg.WindowPos^.cy := Max(Msg.WindowPos^.cy, pnlMain.Height+20+CaptionHeight);

  inherited;
end;

procedure TBlueBackForm.FormResize(Sender: TObject);
begin
  CenterPanel; // 当窗体改变尺寸时使面板居中
end;

end.
```

上面的代码演示了怎样捕捉Windows消息，特别是WM_WINDOWPOSCHANGING消息。这个消息是当窗口的尺寸将要改变时发生的，这样就有机会阻止尺寸的改变。第 5章“ 理解Windows消息 ”将进一步介绍Windows消息。在光盘中可以找到相应的示范程序 TempDemo.dpr。

4.6.8 运行没有窗体的项目

窗体是Delphi 5应用程序的焦点。不过，完全可以创建一个没有窗体的应用程序。为此要创建一个新的项目，然后使用Project | Remove From Project菜单命令把主 窗体移走。此时的项目文件如下所示：

```
program Project1;
uses
 Forms;
{$R *.RES}
begin
  Application.Initialize;
  Application.Run;
end.
```

事实上，甚至可以把uses子句、Application.Initialize 和Application.Run都删掉：

```
program Project1;
begin
end.
```

当然，这么简单的项目肯定是没有任何实际意义的，但要记住，可以在 begin..end之间加入代码，这将成为Win32控制台程序的起点。

4.6.9 退出Windows

有些情况下往往需要退出 Windows。例如，应用程序可能修改了系统配置，而这些配置要在Windows重新启动后才有效。当然可以提示用户自己去重新启动 Windows，但常规的做法是询问用户要不要重新启动Windows；如果要的话，就通过程序重新启动 Windows。不过要记住，系统重启不是很好的行为，应当尽量避免。

要退出Windows，需要用到下面两个API函数中的一个：ExitWindows()或ExitWindowsEx()。

ExitWindows()函数是从16位Wndows移植过来的。在16位Wndows中，需要设置有关选项，以允许退出Windows后再重新启动Windows。不过，在Win32中，这个函数只是注销当前用户，然后让其他用户登录。

现在最好用 ExitWindowsEx()函数，这个函数能够注销当前用户、关闭 Windows，或者在关闭Windows后重新启动它。清单4-12演示上述函数的用法。

清单4-12 用ExitWindows()或ExitWindowsEx()退出Windows

```
unit MainFrm;

interface

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

type
  TMainForm = class(TForm)
    btnExit: TButton;
    rgExitOptions: TRadioGroup;
    procedure btnExitClick(Sender: TObject);
  end;

var
  MainForm: TMainForm;

implementation

{$R *.DFM}

procedure TMainForm.btnExitClick(Sender: TObject);
begin
  case rgExitOptions.ItemIndex of
    0: Win32Check(ExitWindows(0, 0)); // 退出，然后以另外一个用户身份登录

    1: Win32Check(ExitWindowsEx(EWX_REBOOT, 0));   // 退出并重启
    2: Win32Check(ExitWindowsEx(EWX_SHUTDOWN, 0));// 退出并关闭系统
    // 退出/注销/以另外一个用户身份登录
    3: Win32Check(ExitWindowsEx(EWX_LOGOFF, 0));
  end;
```

```
end;

end.
```

上面的代码用一组单选按钮来让用户选择退出 Windows的方式。第一个选项将调用ExitWindows()注销当前用户，然后以另外一个用户身份登录。

剩下的两个选项都是使用 ExitWindowsEx()函数。第二个选项退出Windows然后重新启动计算机。第三个选项退出并关闭系统，这样用户就可以断开电源。第四个选项与第一个选项相同，但它使用的是ExitWindowsEx()函数。

不论是ExitWindows()还是ExitWindowsEx()函数，如果调用成功就返回 True，否则就返回False。可以使用SysUtils.pas中的Win32Check()函数，它将调用Win32 API函数GetLastError()来显示错误信息。

注意 如果运行在Windows NT环境下，使用ExitWindowsEx()函数并不会关闭系统；这需要特殊的权限。必须使用Win32 API函数AdjustTokenPrivleges()授予SE_SHUTDOWN_NAME权限。有关这方面内容的详细介绍请查找Win32的在线帮助。

可以在本书附带的光盘中找到一个示范程序 ExitWin.dpr。

4.6.10 防止关闭Windows

有时候不允许关闭Windows。例如，假设一个应用程序正在编辑一个文件而且还没有存盘，这时如果另一个应用程序调用了ExitWindowsEx()，则会关闭Windows，从而导致数据丢失，除非应用程序知道Windows将要退出。这其实很简单，只要响应主窗体的 OnCloseQuery事件，然后参照下面的代码进行处理：

```
procedure TMainForm.FormCloseQuery(Sender: TObject; var CanClose: Boolean);
begin
  if MessageDlg('Shutdown?', mtConfirmation, mbYesNoCancel, 0) = mrYes then
    CanClose := True
  else
    CanClose := False;
end;
```

如果把CanClose设为 False，表示不允许关闭Windows。如果把CanClose设为 True，将提示用户保存文件。可以在光盘中找到一个示例程序 NoClose.dpr。

提示 如果运行的是一个无窗体的程序，那么应当捕捉WM_QUERYENDSESSION消息。只要有一个应用程序调用了 ExitWindows()或ExitWindowsEx()，每个应用程序就会收到 WM_QUERYENDSESSION消息。如果应用程序从这个消息返回非零值，表示允许 Windows关闭。如果返回零，表示不允许Windows关闭。在第5章“理解Windows消息”中将进一步讲解有关Windows的消息处理。

4.7 总结

本章重点介绍了项目管理技术和体系结构。主要讨论了组成 Delphi 5项目的关键要素：TForm、TApplication和TScreen。我们通过建立一个通用的体系结构来演示怎样开始设计一个应用程序。这一章还介绍了一些其他有用的例程。