

目 录

第一部分 基础	1
第1章 Delphi 7及其IDE	1
Delphi的版本	1
IDE概述	2
Delphi编辑器	8
窗体设计器	16
组件面板的秘密	20
管理项目	23
附加及外部Delphi工具	29
系统生成的文件	30
对象库	35
Delphi 7中调试器的更新	37
小结	37
第2章 Delphi编程语言	38
核心语言特性	38
类与对象	39
封装	42
构造器	48
Delphi的对象引用模型	49
继承已有类型	53
滞后绑定与多态性	56
类型安全的转换	60
使用接口	61
异常处理	63
记录错误	67
类引用	68
小结	71
第3章 运行时库	72
RTL的单元	72
数据转换	85
关于货币转换的问题	88
使用SysUtils来管理文件	91
TObject类	93

	小结	96
第4章	核心库类	97
	RTL包、VCL与CLX	97
	TPersistent类	100
	TComponent类	104
	事件	110
	列表与容器类	114
	流	121
	核心VCL与BaseCLX单元小结	131
	小结	133
第5章	可视控件	134
	VCL与VisualCLX	134
	TControl与派生类	142
	打开组件工具框	146
	控件相关的技术	158
	Listview与TreeView控件	169
	小结	179
第6章	建立用户界面	181
	多页面窗体	181
	工具栏控件	192
	主题与样式	198
	ActionList组件	200
	工具栏容器	208
	ActionManager体系结构	218
	小结	226
第7章	使用窗体	227
	TForm类	227
	窗体直接输入	234
	在窗体中绘图	240
	特殊技巧: 字母混合、颜色键和动画API	242
	位置、大小、滚动和缩放比例	243
	建立和关闭窗口体	251
	对话框和其他二级窗体	253
	建立对话框	256
	预定义对话框	260
	About框与Splash屏幕	262
	小结	265

第二部分 Delphi面向对象的体系结构	267
第8章 Delphi应用程序的结构	267
Application对象	267
从事件到线程	273
检查应用程序以前的实例	278
建立MDI应用程序	281
Delphi中的框架与子窗口	282
带有不同子窗口的MDI应用程序	286
可视窗体继承	289
理解框架	294
基窗体和界面	300
Delphi的内存管理器	304
小结	306
第9章 编写Delphi组件	307
扩充Delphi库	307
创建自己的第一个组件	310
建立复合组件	318
复杂的图形组件	326
定制Windows控件	335
组件中的对话框	344
集合属性	348
定义定制的动作	351
编写属性编辑器	354
编写组件编辑器	358
小结	361
第10章 库与组件包	362
DLL在Windows中的作用	362
使用现有的DLL	364
在Delphi中创建DLL	367
Delphi DLL的高级特性	371
内存中的DLL: 代码与数据	375
使用Delphi组件包	379
组件包内的窗体	381
组件包的结构	387
小结	390
第11章 建模和OOP编程 (使用ModelMaker)	391
了解ModelMaker的内部模型	392
建模和UML	392

ModelMaker的编码功能	397
文档和宏	403
重分解代码	405
小花絮	410
小结	410
第12章 从COM到COM+	411
OLE和COM技术历史简介	412
实现IUnknown	413
第一个COM服务器	416
自动化	422
编写一个自动化服务器	426
使用复合文档	434
介绍ActiveX控件	438
编写ActiveX控件	440
介绍COM+	447
Delphi 7中的COM和.NET	452
小结	454
第三部分 Delphi面向数据库的体系结构	457
第13章 Delphi的数据库体系结构	457
访问数据库: dbExpress、本地数据及其他	457
MyBase: 独立的ClientDataSet	461
使用Delphi的数据敏感控件	467
数据集(DataSet) 组件	471
数据集的字段	476
定位数据集	487
自定义数据库网格	490
带有标准控件的数据库应用程序	495
分组与合计	499
主/详结构	502
处理数据库错误	503
小结	504
第14章 使用dbExpress的客户机/服务器编程	505
客户机/服务器的体系结构	505
数据库设计的元素	507
InterBase简介	510
dbExpress库	515
dbExpress组件	517
一些dbExpress演示	523

包和缓存	532
使用InterBase Express	538
实际问题	546
小结	559
第15章 使用ADO	560
Microsoft数据访问组件 (MDAC)	561
使用dbGo组件	563
使用Jet引擎	568
光标	573
事务处理	578
更新数据	580
断开的记录集合	585
关于ADO.NET	589
小结	589
第16章 多层DataSnap应用程序	590
Delphi发展历史中的一、二、三层	590
建立一个范例应用程序	595
向服务器添加约束	599
向客户端添加特性	601
高级的DataSnap特性	604
小结	610
第17章 编写数据库组件	611
数据链接	611
编写面向字段的数据敏感控件	612
创建定制的数据链接	619
定制DBGrid组件	624
建立定制的数据集合	627
数据集合中的目录	645
对象的数据集合	650
小结	653
第18章 使用Rave做报表	654
Rave介绍	654
Rave设计器的组件	660
高级Rave	667
小结	671
第四部分 Delphi、因特网以及.NET预览	673
第19章 因特网编程：套接字和Indy组件	673
建立套接字应用程序	673

	发送和接收邮件	683
	使用HTTP工作	686
	生成HTML	694
	小结	700
第20章	使用WebBroker和WebSnap进行Web编程	701
	动态Web页面	701
	Delphi的WebBroker技术	703
	实际范例	715
	WebSnap	719
	WebSnap和数据库	730
	会话、用户和许可	736
	小结	740
第21章	使用IntraWeb进行Web编程	741
	IntraWeb简介	741
	建立IntraWeb应用程序	747
	Web数据库应用程序	756
	小结	763
第22章	使用XML技术	764
	XML简介	764
	用DOM编程	768
	XML和InternetExpress	786
	使用XSLT	792
	处理大型的XML文档	797
	小结	801
第23章	Web服务与SOAP	802
	Web服务	802
	建立网络服务	805
	SOAP上的DataSnap	814
	处理附件	817
	支持UDDI	819
	小结	822
第24章	从Delphi的角度看微软.NET体系结构	823
	安装Delphi for .NET Preview	823
	Microsoft的.NET平台	826
	中间语言	830
	无用存储单元收集	833
	部署和版本确定	838
	小结	839

第25章 Delphi for .NET Preview: 语言和RTL	841
去除的Delphi语言特性	841
Delphi语言的新增特性	843
运行时库及VCL	850
VCL	852
Microsoft库的使用	854
利用Delphi语言实现ASP.NET	860
小结	863
附录A 作者提供的其他Delphi工具	864
附录B 其他来源提供的Delphi工具	869
附录C 本书配套的Delphi免费读物	871

第一部分 基础

第1章 Delphi 7及其IDE

在像Delphi这样的可视化编程工具中，集成开发环境所扮演的角色有的时候甚至比编程语言还要重要。Delphi 7在Delphi 6丰富的IDE基础之上又提供了一些新的有趣的特性。本章将主要介绍这些新的特性，以及在其他各早期Delphi版本中所添加的一些特性。同时我们也会讨论许多对新入门的读者来说并不太熟悉的传统的Delphi特性。由于篇幅所限本章并未给出一份完整的IDE指南，而是针对普通的Delphi用户介绍一些技巧和并提供一些建议。

如果读者是一位初级程序员，不要担心，Delphi的集成开发环境（IDE）使用起来是相当直观易用的。Delphi本身也提供了一份参考手册（在Delphi Companion Tool光碟中可以得到Acrobat格式的文件），为大家介绍Delphi应用程序的开发。读者也可以在我的Essential Delphi在线书籍中（参见附录C“本书配套的Delphi免费读物”）找到对Delphi及其IDE的简单介绍。在整个学习过程中，我们假设各位已经知道了如何执行一些IDE实际的动手操作；本书后面的其他章节将会侧重在编程问题和技巧上。

本章主要包括以下内容：

- IDE概览
- 编辑器
- Code Insight技术
- 设计窗体
- 项目管理器
- Delphi文件

Delphi的版本

在开始深入研究Delphi编程环境的具体细节之前，让我们先来强调两个关键的概念。首先，并不是只有一种Delphi版本，而是存在许多种Delphi版本。第二，任何版本的Delphi环境都可以进行定制。因此，本章示例所用的Delphi屏幕可能与读者计算机上的那些Delphi屏幕并不完全相同。下面是当前的一些Delphi版本：

- 个人版（Personal）主要针对的是新入门的Delphi用户和不常使用Delphi的程序员。该

版本既不支持数据库编程，也不支持任何Delphi中的各种高级特性。

- 专业版（Professional）主要针对专业开发人员。该版本包括了Delphi所有的基本特性，并增加了对数据库编程的支持（包括ADO支持）、基本Web服务器的支持（WebBroker），以及一些外挂工具（包括ModelMaker和IntraWeb）的支持。本书假设读者正在使用的至少应该是Delphi专业版。
- 企业版（Enterprise）主要针对的是开发企业应用的开发人员。该版本包括所有的XML和高级Web服务技术、CORBA支持、国际化以及许多其他工具。本书中的一些章节会专门介绍企业版中特有的特性，并给予特别标注。
- 体系版（Architect）在企业版的基础上添加了对Bold的支持，Bold是一种开发环境，用于建立在运行时由一个UML模型驱动的应用程序，而且因为其中包含一些高级组件，故能够将其对象映射到一个数据库和用户界面中。Bold支持在本书中并没有涉及。

除了这些不同的版本之外，还有一些可以定制Delphi环境的方法。对本书中的屏幕图示，笔者将尽量使用一个标准的用户界面。但是我当然也有一些自己的选择，通常安装一些附加工具，这些有可能会反映到书中的屏幕快照上。

在Delphi 7的专业版及其以上版本的Delphi语言版本中，已经包括了一个Kylix 3的工作拷贝。除了Delphi对CLX库的引用和跨平台的特性之外，本书并没有涉及其他的Kylix和Linux的开发问题。读者可以参考“Mastering Kylix 2”（Sybex, 2002，中译本《Kylix 2从入门到精通》一书已由电子工业出版社出版）一书，以获得关于该主题的更多信息（对Delphi语言版本来说，其中的Kylix 2和Kylix 3实际上并没有什么太大的区别，Kylix 3中最重要的新特性就是其对C++语言的支持）。

IDE概述

当使用一种可视化的开发环境时，我们的时间将主要花费在该应用的两个不同部分：可视设计器和代码编辑器。设计器使程序员可以以可视的层次（如将一个按钮放置在一个窗体上）或非可视化的层次（如将一个数据集合组件放置在一个数据模块上）对组件进行操作。在图1.1中，可以看到正在工作的一个窗体和一个数据模块。在这两种情况下，设计器允许程序员选择需要的组件并设置该组件属性的初始值。

代码编辑器就是编写代码的地方。在一个可视环境中，最常见的编写代码的方式主要涉及响应一个事件、开始运行程序用户执行的操作所对应的事件，如单击一个按钮或从列表框中选择一个条目。可以使用相同的方法处理一个内部事件，如涉及到数据库改变或操作系统通知的事件。

当程序员获得更多的关于Delphi的知识时，他们经常会首先编写主要的事件处理代码，然后接着开始编写自己的类和组件，并且常常将他们的时间花费在编辑器上。因为本书不仅涉及可视编程，还想要帮助读者精通整个Delphi强大的功能，所以在接下去的学习中，大家将会看到更多的是代码而不是窗体。

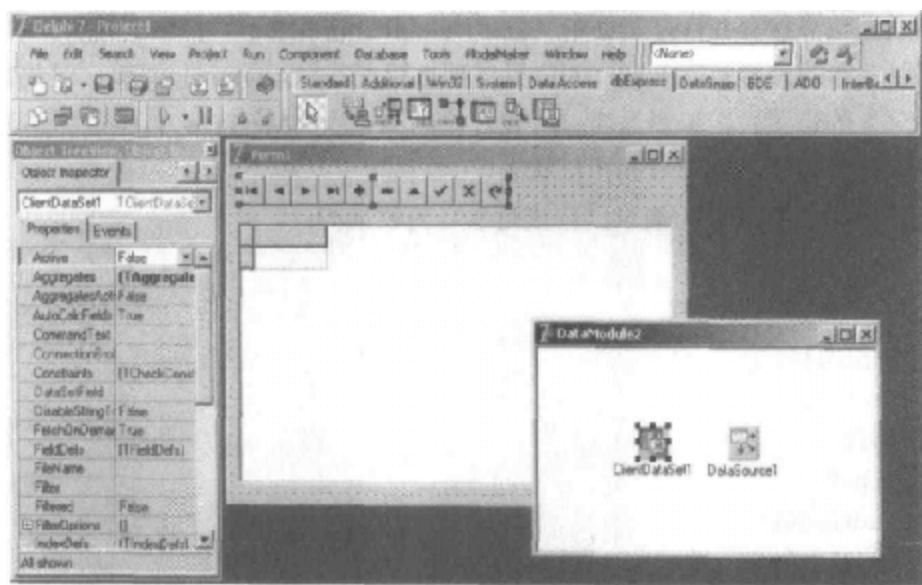


图1.1 Delphi 7 IDE中的一个窗体和一个数据模块

一个IDE用于两个库

这是在Delphi 6中首次出现的一种重要的特性。IDE现在允许用户使用两个不同的可视库：VCL（可视组件库）和CLX（跨平台组件库）。当我们创建一个新的项目时，只需要选择两个库中想使用的那一个就可以了：使用File►New►Application命令选择经典的基于VCL的Windows程序，或使用File►New►CLX Application命令选择新的基于CLX的可移植应用程序。

说明：CLX是Delphi的跨平台库，使我们能够用Kylix重新编辑自己的代码，使其能够运行在Linux下。读者可以在第5章中了解到更多的关于VCL和CLX的信息。在Delphi 7中使用CLX将会更加有趣，因为Kylix的Delphi语言版本是随Windows产品一起出售的。

当建立一个新的项目或打开一个现存的项目时，组件面板将会重新排列，以便只显示与当前库相关的控件（尽管大多数控件是共享的）。而使用一个非可视设计器（如一个数据模块）时，那些只能处理可视组件的组件面板选项卡将会在视图中隐藏。

桌面设置

程序员可以用不同的方法定制Delphi IDE——典型地是打开多个窗口，将它们排列并衔接起来。然而，我们也经常需要在设计时打开一批窗口，而在调试的时候打开另外一批窗口。与之相似，在操作窗体的时候需要一种布局，而在只使用编辑器编写组件或底层代码时却需要完全不同的另外一种布局。针对这些不同的需求而重新安排IDE是一项很繁琐的任务。

因此，Delphi允许用户保存一套带有名称的IDE窗口的特定排列（称为一个桌面或全局性桌面，后者区别于一个项目的桌面），并且可以很容易地恢复它。我们也可以将这些组中的任意一组作为默认的调试设置，这样在启动调试器时会自动地恢复。所有这些特性都可以在新的Desktops工具栏中使用，用户也可以使用View►Desktops菜单进行桌面设置。

桌面设置信息保存在DST文件中（位于Delphi的bin目录下），该文件实际上是一个INI文件。可以保存的设置包括主窗口的位置、项目管理器、Alignment面板、对象监视器（包括其属性分类的设置）、编辑器窗口（包含代码浏览器和Message View的状态）以及其他很多信息，此外还有不同窗口的衔接状态。

这里有一段DST文件的摘录，应该是很容易读懂的：

```
[Main Window]
Create=1
Visible=1
State=0
Left=0
Top=0
Width=1024
Height=105
ClientWidth=1016
ClientHeight=78

[ProjectManager]
Create=1
Visible=0
State=0
...
Dockable=1

[AlignmentPalette]
Create=1
Visible=0
...
```

桌面设置将会覆盖项目设置（以相似的结构存放在一个DSK文件中）。桌面设置有助于避免我们在计算机之间（或开发人员之间）移动一个项目和必须按照自己的习惯重新安排窗口时发生问题。Delphi可将每个用户的全局性桌面设置和每个项目的桌面设置独立开来，以更好地支持团队开发。

提示：如果打开Delphi时不能看到窗体或其他窗口，我建议你检查（或删除）桌面设置（从Delphi的bin目录中）。如果打开一个从其他用户来的项目时，不能看到任何窗口或者不喜欢桌面的布局，可以装载自己的全局性桌面设置或者删除该项目的DSK文件。

环境选项

最近，相当一部分更新是关于常用环境选项对话框的。该对话框的页面在Delphi 6中被重新安排，窗体设计器选项从Preferences页面移到了新的设计器页面。在Delphi 6中，也有一些新的选项和页面：

- 环境选项对话框的Preferences页面中有一个复选框，该复选框可以防止Delphi窗口相互间自动产生衔接。
- 环境变量页面允许我们查看系统的环境变量（如标准的路径名称和OS设置）并能够设置用户定义的变量。其优点是在IDE的每个对话框中既可以使用系统定义的环境变

鼠，也可以使用用户定义的环境变量——例如，避免使用常用的固定的路径名称，而用变量来代替。也就是说，环境变量的工作原理与\$DELPHI变量相似，既可以引用Delphi的基本目录，同时还可以由用户来定义。

- 在Internet页面中，可以选择HTML和XML文件所使用的默认文件扩展名（主要是通过WebSnap框架），也可以将一个外部编辑器与每个扩展名关联起来。

菜单

尽管用户可能会通过使用快捷键和快捷菜单完成大部分的任务，但Delphi的主菜单栏（在Delphi 7中看起来会更加美观）仍然是一种与IDE交互的重要方式。就我们目前的操作来说，不需要对菜单栏做太多的改变：单击鼠标右键，就可以得到一个完整的列表，其中包含针对当前窗口或组件所能执行的所有操作。

而根据安装的第三方工具或向导的不同，菜单栏则可能会有相当大的改变。在Delphi 7中，ModelMaker会有其自己的菜单。通过安装诸如Gexperts这样的通用附件甚至是自己设计的向导（分别参见附录B和附录A，以获得更多的信息），读者将会看到其他的菜单。

在最近的Delphi版本中，新添加了一个相应的菜单，即IDE中的Window菜单。这个菜单列出了打开的窗口；而在以前，我们可能需要使用Alt+O组合键或View►Window List菜单命令才能获得这个列表。因为该窗口总会隐藏在其他窗口的下面，故很难被找到；这个时候，Window菜单是相当好用的。只需使用Windows注册表中的一个设置，就可以按字母表的顺序控制该菜单的显示：在Delphi中找到Main Window子键（在HKEY_CURRENT_USER\Software\Borland\Delphi\7.0中）。该注册表键值使用一个字符串（布尔变量）表示，用“-1”和“True”表示真，而用“0”和“False”表示假。

提示：在Delphi 7中，Window菜单以一个新的命令作为结束：Next Window。以Alt+End快捷键形式使用这个命令特别有效。在IDE的不同窗口之间切换从来没有现在这样简单（至少，在没有附加工具的情况下）。

环境选项对话框

正如笔者曾经提到的那样，一些IDE的设置需要我们直接编辑注册表。本章将会讨论一些这样的设置。当然，那些最常见的设置可以通过使用环境选项对话框来设定，该对话框可以从编辑器选项和调试器选项的Tools菜单中得到。这些设置中的大部分都是相当直观的，并且Delphi的帮助文件也对其有详细的描述。图1.2给出了该对话框Preference页面中的标准设置。

to-do列表

在Delphi 5中引进的，但我们现在仍然需要很好了解的另一个特性就是to-do列表。这是完成一个项目必须要执行的任务列表，是为程序员（或程序员们，这个工具在团队开发中是非常方便的）提供的一组注释。尽管这个概念不是什么新东西，但在Delphi中，to-do列表的关键就在于其工作起来像是一个两用工具。

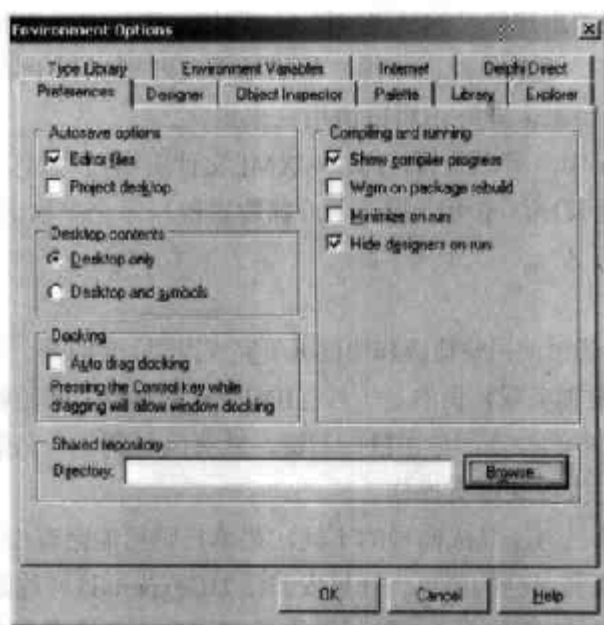


图1.2 Environment Options对话框中的Preferences页面

可以通过向一个项目的任何文件的源代码添加特定的TODO注释来添加或修改to-do条目，并且在列表中看到它们。另外，还可以可视编辑该列表中的条目来修改对应的源代码注释。例如，对于源代码中的下面这项to-do列表的条目：

```
procedure TForm1.FormCreate(Sender: TObject);
begin
    // TODO -oMarco: Add creation code
end;
```

我们还可以在图1.3所示的To-Do列表窗口中对其进行可视编辑。

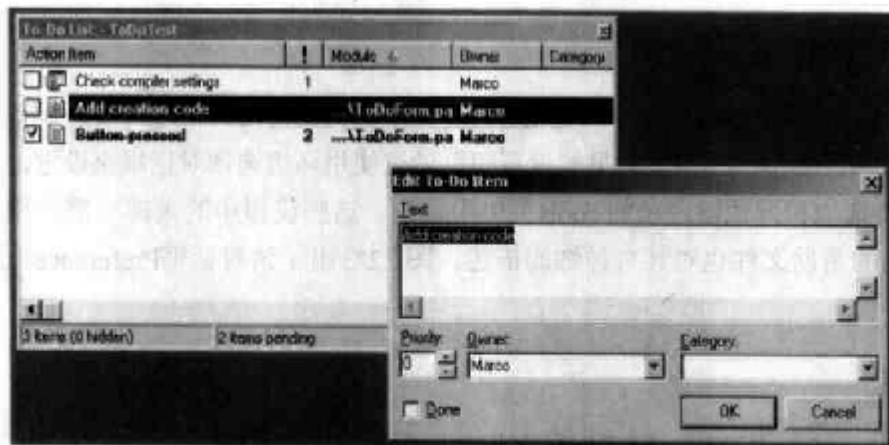


图1.3 Edit To-Do Item窗口可以用来修改一个to-do条目，一个可以直接在源代码中所做的操作

这种两用规则也有一个例外，就是在对整个项目范围的to-do条目进行定义时，必须要将这些条目直接添加到列表中。为此，应在To-Do列表窗口中使用Ctrl+A组合键，或者使用鼠标右键单击窗口并从快捷菜单中选择Add命令。这些条目被保存在一种特殊的文件中，这

种文件的名称与根项目文件相同，但是扩展名是TODO。

一个TODO注释可以与多个选项一起使用：用-o（就像前面的代码摘录中所提到的那样）表示所有者（即输入注释的程序员），用-c表示一个类别，或者简单地使用从1到5的数字表示优先级（0或没有数字表示没有设置优先级别）。例如，若使用编辑器快捷菜单中的Add To-Do Item命令（或Ctrl+Shift+T快捷键）生成注释如下：

```
{ TODO 2 -oMarco : Button pressed }
```

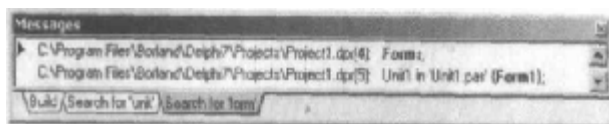
则Delphi会根据注释的类型，将冒号之后到一行结束或大括号之间的所有字符都看做是to-do条目的文本。

另外，在To-Do列表窗口中，我们不仅可以核对一个条目来表明其所完成的操作——这时源代码注释将会从TODO变为DONE，而且还可以手工地改变源代码中的注释——这时在To-Do列表窗口中会出现复选标记。

该结构中功能最强的元素之一是主To-Do列表窗口，它可以在用户键入、分类和过滤源代码文件时从其中自动地收集to-do信息，并将其作为普通文本或HTML表格输出到剪贴板上。

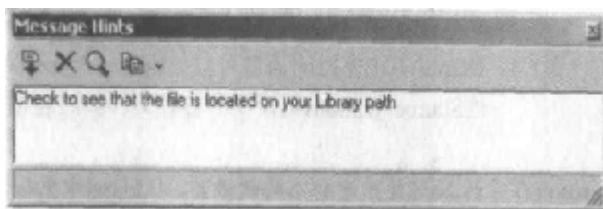
Delphi 7中的扩展编译器消息和搜索结果

默认情况下，在编译器的下方会有一个小的消息窗口；该窗口既可以显示编译器信息，也可以显示搜索的结果。这个窗口在Delphi 7中已有相当大的修改。首先，搜索结果会在不同的选项卡中显示，这样就不会像以前一样与编译器的消息互相影响。第二，我们每次做不同的搜索时，可以要求Delphi将各自的结果以不同的页面显示，这样以前所做的搜索操作将仍然可以使用：



Alt+Page Down和Alt+Page Up组合键用于在这个窗口的不同选项卡之间来回地切换（对其他的选项卡式视图，该组合键也同样有效）。

如果编译器有错误产生，可以通过View►Additional Message Info命令激活另一个新的窗口。当编译一个程序的时候，该消息提示窗口将会提供关于常见错误消息的一些额外信息，并且提供如何修正这些错误的建议。



这种类型的帮助主要针对新入门的程序员，而且该窗口的操作也是非常方便的。其中非常重要的一点就是，它的信息是完全可以定制的：一个开发项目的领导可以专门针对新加入的开发人员将一些常见错误的描述放入到一个窗体中。要想实现这一功能，只需在Delphi的bin文件夹下，遵循msginfo70.ini中的注释信息即可，该文件用于这一特性的设置。

Delphi编辑器

从外观上看起来，Delphi编辑器在IDE版本7中没有什么大的改变。但是，在这种表像的背后，Delphi编辑器已经完全是一个新的工具了。除了用该工具以Object Pascal语言（或Delphi语言，Borland现在更愿意这样称呼）对文件进行操作以外，我们现在还可以使用它操作其他用于Delphi开发的文件（如SQL、XML、HTML和XSL文件）以及用其他语言写的文件（包括C++和C#）。Delphi 6就已经支持对XML和HTML进行编辑了，但是新版本中的改变还是相当大的。例如，当编辑一个HTML文件时，可以获得的支持包括语法亮显和代码自动生成。

编辑器适用于各个文件的设置（包括诸如Tab等键的功能）。我们可以在Editor Properties对话框的新的Source Options页面中对这些设置进行相应的配置，如图1.4所示。该特性已经被扩展，并且更加开放，所以我们甚至可以通过为基于XML的文件格式提供一个DTD来配置编辑器，或者通过编写一个能够为其他编程语言提供语法亮显的定制向导来配置编辑器。编辑器的另一个特性就是代码模板，该特性是当前语言所特有的（预定义的Delphi模板对HTML或C#没什么意义）。

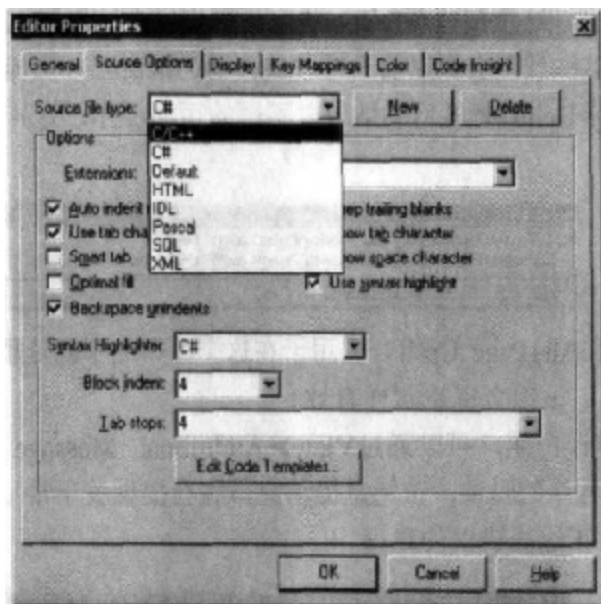


图1.4 Delphi IDE支持的语言可以与Editor Properties对话框Source Options页面中的各种文件扩展名相关联

说明：C#是Microsoft在它的.NET体系中引入的新语言。Borland希望在其自己的.NET环境下支持C#，其当前代码称为Galileo。

仅仅就Delphi语言而言，IDE环境下的编辑器在最近版本中并没有太大的改变。但是，仍然有一些新的特性对很多Delphi程序员来说并不了解，也不太使用，所以在这里我还是想简单介绍一下。

通过使用“带选项卡的记事本”，Delphi编辑器允许用户同时对多个文件进行处理。而使用Ctrl+Tab（或按Ctrl+Shift+Tab向相反的方向移动）组合键，可以从编辑器的一个页面跳

到下一个页面。如果拖拽编辑器上部的带有单元名称的选项卡以改变它们的顺序,则在任何时候,只需简单地使用`Ctrl+Tab`组合键就可以在正在操作的不同单元之间来回移动。编辑器的快捷菜单也有一个`Pages`命令,用于列出一个子菜单中所有可以使用的页面(当很多的单元同时被装载时,使用这一特性是非常方便的)。

也可以同时打开多个编辑器的窗口,每个窗口会有多个选项卡。这是可以同时查看两个单元的源代码的惟一途径(实际上,当我需要对比两个Delphi单元时,我总是使用Beyond Compare——www.scootersoftware.com——一个用Delphi写得非常好的、又非常便宜的文件比较工具)。

正如读者从图1.4所示的编辑器属性对话框中看到的那样,还有一些可以影响编辑器的选项。然而,用户必须翻到环境变量选项对话框(参见图1.2)的`Preferences`页面,才能设置编辑器的`AutoSave`特性。这一选项会强迫编辑器在用户每次运行一个程序时,保存所有的源代码文件,以防止万一程序在调试器中出现意外时丢失数据。

Delphi编辑器还可以提供很多命令,包括一些其早期的WordStar仿真命令(隶属于早期的Turbo Pascal编译器)。我并不想在这里讨论编辑器的这些不同设置,因为它们都非常直观,并且在联机帮助中都有说明。要注意的是,只有当我们查找shutcuts目录条目时,那些对键盘快捷键进行描述的帮助页面才可以使用。

提示:要记住的一个技巧就是使用`Cut`和`Paste`命令不再是移动源代码的惟一方法了。可以选择和拖动字、表达式或者整行源代码。另外,只需在拖动的时候按住`Ctrl`键,就可以复制文本,而不是仅仅移动它了。

代码浏览器

代码浏览器的窗口通常会出现在编辑器的旁边,列出一个单元中定义的所有类型、变量和例程,以及出现在`uses`声明中的其他单元。对于复杂的类型,如类,代码编辑器可以列出详细的信息,包括字段的列表、属性和方式。只要在编辑器中开始输入代码,所有的信息就会被更新。

可以使用代码浏览器在编辑器中进行浏览。如果双击代码浏览器中的条目,编辑器就会跳到相应的声明处。也可以在代码浏览器中直接修改变量、属性和方式名称。但是,就像读者看到的那样,如果对类进行操作时需要使用一个可视化的工具,ModelMaker将会提供更多的特性。

尽管在使用Delphi几分钟后,所有这些功能都会非常明显,但代码浏览器中还有一些特性却不是那么直观的。为此,需要完全地控制信息的布局。并且通过定制代码编辑器,减少在这个窗口中显示的树的层次深度(这样有助于加快做出选择的速度)。可以通过使用环境变量对话框中相应的页面来配置代码浏览器,如图1.5所示。

要注意的是,从该对话框的这个页面的右侧栏内取消一个`Explorer Categories`条目时,浏览器将不会从视图中删除相应的元素——它只是简单地在树状结构上添加一个节点。例如,如果取消`Uses`复选框,Delphi就不会把已经使用的单元列表从代码浏览器中隐藏起来;相反,已使用的单元将会作为主节点被列出,而不是列在`Uses`文件夹中。我通常都会禁用类型、类和变量/常量选项。

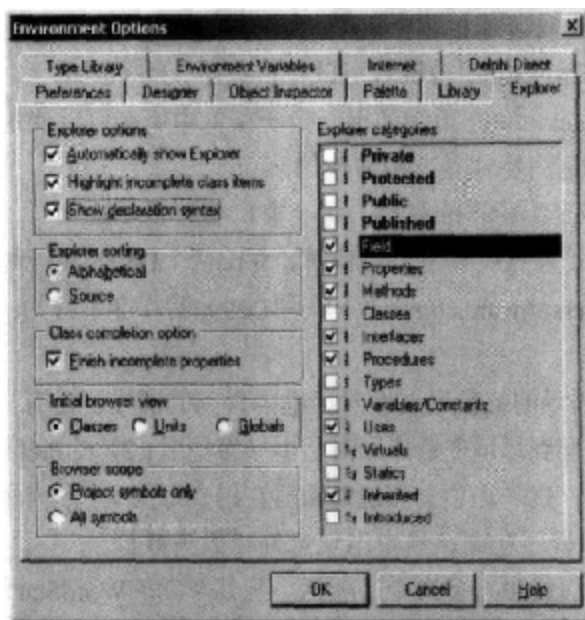


图1.5 可以配置Environment Options对话框中的Code Explorer

因为代码浏览器树形结构上的每一项都有一个图标表示其类型，所以按对象方法和字段进行安排并不像按访问标识符进行安排那样重要。笔者建议把所有的条目都显示在一个组中，因为这样安排只需要最少的鼠标单击次数就可以访问每个条目。事实上，在用户选择条目时，代码浏览器为其浏览一个大单元的源代码提供了非常方便的方法——双击代码浏览器中的一个对象方法，光标就会移动到类声明的定义处。也可以使用Module Navigation (Ctrl+Shift和上下箭头的组合键)，从一个对象方法或过程定义处跳到其完整的定义处（会返回）。

说明：在图1.5所示的一些Explorer Categories是由项目浏览器所使用的，而不是由代码浏览器使用的，其中包括Virtuals、Statics、Inherited和Introduced分组选项。

在编辑器中浏览

编辑器的另一个特性就是“工具提示符号识别” (Tooltip symbol insight)。将鼠标移动到编辑器的一个符号上，工具提示便会告诉用户该标识符是在哪里被声明的。当需要在正在编写的一个应用程序中跟踪标识符、类和函数时，这个特性是特别重要的，该特性也可以用于引用库的源代码。

警告：尽管看起来这可能是一个好主意，但实际上，并不能使用工具提示符号来查找哪个单元声明了我们想使用的标识符。事实上，如果没有包括相应的单元，工具提示将不会出现。

该特性真正的优点在于我们可以利用它实现“代码浏览”。当按住Ctrl键并把鼠标移动到标识符上时，Delphi将创建一个可以激活的链接，而不是显示工具提示。这些链接将以蓝色下划线形式显示，这是Web浏览器中链接的典型形式，并且当鼠标位于该链接之上时，鼠标的形状将会变为手的形状。

例如，按下Ctrl键并单击TLabel标识符，就可以打开其在VCL源代码中的定义。当选择引用时，编辑器会跟踪用户到达不同的位置，而且还可以在这些位置之间前后移动——这也

与Web浏览器一样——在编辑器窗口中使用右上方的Browse Back和Browse Forward按钮，或者使用Alt+左箭头/Alt+右箭头进行相同的操作。为了更好地控制前后的移动，也可以单击Back和Forward按钮旁边的下拉箭头，来查看已经到过的源文件的详细清单。

如果VCL源代码不是项目的组成部分，如何能直接跳到该代码中呢？答案是使用编辑器，它不仅可以查找位于Search路径中的单元（作为项目的一部分被编译），还可以查找Delphi的Debug Source中的单元。这些目录是按照我刚才所列的顺序进行查找的，但是我們可以在项目选项对话框的Directories/Conditionals页面和环境选项对话框的Library页面中对其进行设置。默认情况下，Delphi会在Browsing路径中添加VCL源代码目录。

类的自动生成

Delphi编辑器可以自动产生一些源代码来帮助用户完成正在编写的程序。该特性称为类的自动生成，可以通过使用Ctrl+Shift+C组合键将其激活。向应用程序添加事件处理器是一种很快的操作，因为Delphi会在类中自动添加一个新的对象方法声明来处理事件，并在单元的实现部分为用户提供对象方法的框架。这就是Delphi对可视编程提供支持的一部分。

新版本的Delphi还为那些需要在事件处理器后面编写一些附加代码的程序员提供了另一种相似的方法。这种代码生成特性适用于常用的对象方法、消息处理方法以及属性。例如，如果在类的声明中输入下列代码：

```
public
    procedure Hello (MessageText: string);
```

然后按Ctrl+Shift+C组合键，Delphi将会在单元的实现部分提供对象方法的定义，并生成下列代码：

```
{ TForm1 }

procedure TForm1.Hello(MessageText: string);
begin
end;
```

与许多Delphi程序员使用的传统方法（复制和粘贴一个或多个声明，添加类的名称，最后为每个对象方法复制begin...end代码）相比，这个特性是非常方便的。类的自动生成还可以以其他方式使用：直接编写实现对象方法的代码并按Ctrl+Shift+C组合键，从而在类的声明中生成所需的条目。

类的自动生成最重要而且最有用的例子就是自动生成代码，以支持类中的属性声明。例如，如果在一个类中输入

```
property Value: Integer;
```

并且按下Ctrl+Shift+C组合键，Delphi将会将这一行转换为

```
property Value: Integer read fValue write SetValue;
```

Delphi也将会向类声明中添加SetValue方式，并为其提供一个默认的实现。在下一章中，读者可学习到更多的关于属性的知识。

代码识别

除了代码浏览器、类的自动生成以及导航特性之外，Delphi编辑器还支持代码识别（code insight）技术。总的来说，代码识别技术是基于背景分析的，这种分析一方面来自用户编写的源代码，另一方面来自其源代码引用的系统单元的源代码。

代码识别由5项功能构成：代码完成、代码模板、代码参数、Tooltip表达式估值和Tooltip符号识别。最后一个特性已经由“在编辑器中浏览”小节介绍过了，其他的四个特性将在下面各小节中讨论。读者可以在编辑器属性对话框的Code Insight页面中启用、禁用和配置这些特性。

代码完成

代码完成特性允许用户通过查看一个列表或者键入对象的首字母来选择对象的属性和方法。要想激活这个列表，只需要输入一个对象的名称，如Button1，然后添加圆点并等待就可以了。要想强制显示该列表，可以按住Ctrl+空格键；当不需要该列表并且要移除它时，只需按Esc键。代码完成还允许用户在赋值语句中查询相应的值。

当开始输入时，该列表会根据插入元素的初始位置过滤其内容。代码完成列表可利用颜色及更多的细节来帮助用户识别不同的项。在Delphi中，用户还可以在编辑器选项对话框的Code Insight页面中定制这些颜色。另一个特性就是，对于具有参数的函数，圆括号一旦输入到生成的代码中，参数列表提示就会立刻显示出来。

在一个变量或属性后输入:=时，Delphi将列出相同类型的所有变量或对象，以及具有该类型属性的对象。一旦该列表显示出来，用户就可以右键单击它来改变条目的顺序，按作用范围或名称对其分类，而且还可以改变窗口的大小。

从Delphi 6开始，代码完成还用于一个单元的接口部分。如果我们在鼠标位于类定义中的时候按下Ctrl+空格键，就会得到一个列表，其中含有可以覆盖的虚拟对象方法（包括抽象的对象方法）、实现接口的对象方法、基类属性，以及可以处理的系统消息。只需选择它们之中的任何一个，即可向类声明中添加相应的对象方法。在这种特殊的情况下，代码完成列表提供了多种选择。

提示：当编写的代码出现错误时，代码识别将不能工作，而且程序员只能看到说明情况的普通错误消息。可以在Message窗格中显示特定的代码识别错误（该窗格必须已经打开；它不会自动打开来显示编译错误）。为了激活该特性，需要设置一个未公开的注册表条目，将字符串键值\Delphi\7.0\Compiling\ShowCodeInsightErrors设为“1”。

代码完成还包括一些高级的特性，并不容易被发现。其中有一个特别有用的特性与项目未使用单元中的符号发现相关。当在一个空白行上使用它（通过Ctrl+空格键）时，列表还包含来自同一单元的符号（如Math、StrUtils和DateUtils），而且该单元还没有包含在当前单元的uses语句中。通过选择这些外部符号中的一个，Delphi可以将单元添加到uses语句中。该特性（不能在表达式中工作）由外部单元的定制列表驱动，存放在注册表键值\Delphi\7.0\CodeCompletion\ExtraUnits中。

提示：Delphi 7中增加了一个新的功能，在按住Ctrl键的同时单击代码完成列表中的任何标识符，就可以浏览该列表中项目的声明。

代码模板

该特性使用户能够插入事先定义好的代码模板，如一个带有内部begin...end块的复杂语句。代码模板必须要通过手工才能够被激活，使用Ctrl+J组合键可以显示所有模板的列表。如果在按Ctrl+J组合键之前输入了几个字母（如一个关键字），Delphi将只会列出以这些字母打头的模板。

我们可以添加自己定制的代码模板，这样就可以为通用的代码块建立自己的快捷方式。例如，如果经常使用MessageDlg函数，则可以为其添加一个模板。为了修改模板，需进入Editor Option对话框的Source Options页面中，从Source File Type列表中选择Pascal，并单击Edit Code Templates按钮。这样将会打开一个新的Delphi 7的Code Templates对话框。在这里，单击Add按钮，键入一个新的模板名称（如mess），输入描述说明，然后向Code备注控件的模板中添加下列文本：

```
MessageDlg ('', mtInformation, [mbOK], 0);
```

现在，每当需要创建一个消息对话框时，我们只需简单地输入mess，然后按Ctrl+J组合键，就可以获得完整的文本信息。竖线（或管道）符号表示在展开模板后，光标在编辑器源代码中的位置。应该选择开始输入的位置来完成由模板生成的代码。

尽管代码模板可能看起来与语言的关键字没有直接的对应关系，但它们是一种更通用的机制。这些模板都被保存在DELPHI32.DCU文件中，这是一种格式相当简单的文本文件，可以直接对其进行编辑。Delphi 7也允许用户将对一种语言的设置输出到一个文件中，反之亦然。这样对开发人员来说可以很容易地定制自己的模板。

代码参数

当输入一个函数或方法时，代码参数在一个提示或Tooltip窗口中会显示这个函数或方法参数的数据类型。只需输入函数或方法的名称与左括号，参数的名称和类型就会立刻显示在一个弹出提示窗口中。要想强行显示代码参数，可以按Ctrl+Shift+空格键。作为进一步的帮助，当前参数会用黑体显示。

Tooltip表示式估值

Tooltip表示式估值是一个调试特性，用来显示鼠标所指的标识符、属性或表达式的值。对于表达式，我们通常需要在编辑器中进行选择，然后将鼠标移到亮显的文字上。

更多的编辑器快捷键

根据用户所选择的编辑器风格的不同，编辑器中还有其他很多的快捷键。这里是大家不太了解的一些快捷键：

- Ctrl+Shift加上从0到9中的一个数字键，就可以激活一个书签，它显示在编辑器边上的“装订线”的空白处。要想回到该书签，按住Ctrl加上那个数字就可以了。书签的用途在编辑器中受到限制，因为新书签将会覆盖原有的书签，而且书签并非持续不变（当关闭文件时，书签将会丢失）。

- **Ctrl+E**可以激活增量查询。按住**Ctrl+E**组合键，然后直接输入想要查询的单词（而不需要打开某个特定的对话框）并按**Enter**键即可进行查找。
- **Ctrl+Shift+I**组合键用于同时缩进多行代码。缩进的空格数要通过编辑器选项对话框中编辑器页面的**Block Indent**选项进行设定。**Ctrl+Shift+U**是用来对应地进行反操作的组合键。
- **Ctrl+O+U**用于切换所选择的代码的大小写；也可以使用**Ctrl+K+E**组合键将所选代码转换为小写，而用**Ctrl+K+F**组合键将其转换为大写。
- **Ctrl+Shift+R**组合键用于开始记录一个宏，之后使用**Ctrl+Shift+P**快捷键执行这个宏。宏记录了在源代码文件中所有的输入、移动以及删除操作。执行宏就可以简单而顺序地重复所有记录的操作——但换到另一个不同的源代码文件后，该操作就没有什么意义了。编辑器宏在需要重复执行多步骤操作时是相当有用的，如重新格式化源代码，或在源代码中更合理地安排数据。
- 当按下**Alt**键时，可以通过拖动鼠标来选择编辑器的矩形区域，而不只是选择连续的代码行和代码。

可装载的视图

在Delphi 6中引入的另一个重要的特性就是在编辑器中支持多个视图。对于装载到IDE中的任何一个文件，编辑器可以显示多个视图，而且通过编程方式定义并添加到系统中，然后针对特定的文件进行装载——因此称为可装载的视图。

最常用的视图是Diagram页面，该页面在Delphi 5的数据模块中已经可以使用了，尽管其功能还不够强。另一系列的视图可以用于网络应用程序中，包括HTML脚本视图、HTML Result预览和许多其他的视图，我们将在第20章“使用WebBroker和WebSnap进行Web编程”和第22章“使用XML技术”中进行讨论。可以使用**Alt+Page Down**和**Alt+Page Up**组合键在编辑器底部所示的选项卡间来回切换；**Ctrl+Tab**组合键用来改变顶部选项卡中出现的页面（或文件）。

Diagram视图

Diagram视图显示了组件之间的从属关系，包括父/子关系、属主关系、链接属性和类属关系。对于数据集合组件，它还支持主/从关系与查找连接。用户甚至可以在与指定组件相连的文本块中添加自己的注释。

Diagram并不是自动创建的。必须从树状视图将各组件拖拽到图表中，才会自动地显示所拖放组件之间已有的关系。也可以从对象树状视图中选择多个项目，同时将它们都拖到Diagram页面中。

可以通过在组件之间绘制箭头来设置属性。例如，在将一个编辑框和一个选项卡移动到图表中之后，就可以选择Property Connector图标，单击该选项卡，并将鼠标拖到编辑框上。当释放鼠标按钮时，Diagram视图将基于FocusControl属性（这是选项卡惟一可以引用编辑控件的属性）建立一个属性关系。图1.6描述了这种情况。

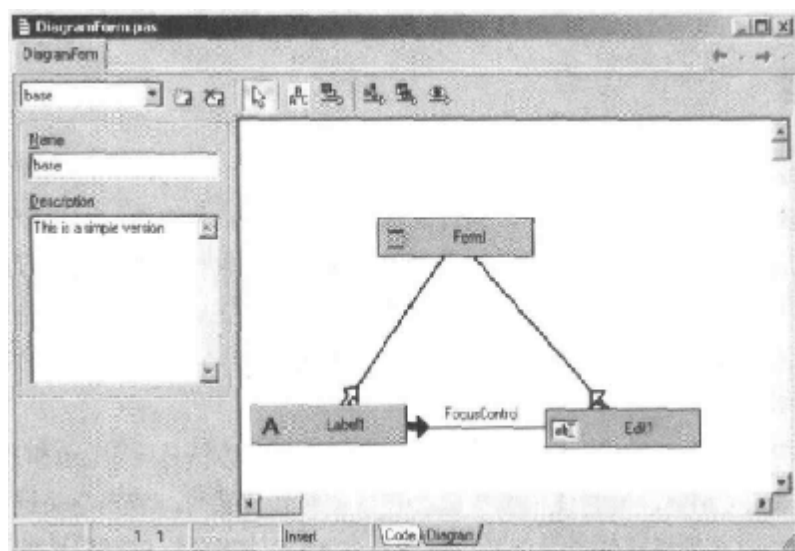


图1.6 Diagram视图显示了组件之间的关系（且允许用户对它们进行设置）

正如读者所看到的那样，设置属性是直接的：如果从编辑控件向选项卡中拖动属性关系线，将会导致使用选项卡作为编辑框一个属性的值。因为这是不可能的，所以我们将看到一个错误消息提示，指明该问题并提供反方向的组件连接。Diagram视图允许用户为每个Delphi单元，即每个窗体或数据模块，创建多个图表。只需要为图表提供一个名称，添加一些描述，然后单击New Diagram按钮，并准备另一个图表，就可以使用Diagram视图工具栏中的可用组合框来前后切换图表。

尽管可以使用Diagram视图来建立关系，但其主要作用是归档设计方案。因此，可以打印该视图的内容。在Diagram视图处于活动状态时，使用File►Print命令，Delphi将会提示如图1.7所示的相关选项，用于用多种方式定制输出。

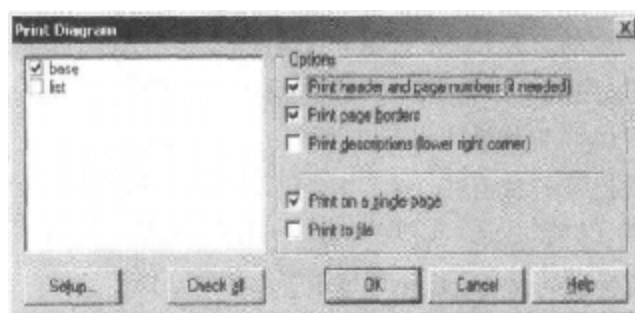


图1.7 Diagram视图的Print选项

Diagram视图中的信息将会被保存在一个独立的文件中，而不作为DFM文件的一个部分。Delphi 5使用设计时信息（DTI）文件，该文件格式与INI文件相似。Delphi 6和Delphi 7仍然能够读取旧的DTI格式，但它们使用了新的Delphi Diagram Portfolio格式（DDP）。这些文件使用DFM二进制格式（或相似的格式），所以它们不能作为文本来编辑。很显然，所有这些文件在运行时都是没有用的（把它们包括在可执行文件的编译中是没有意义的）。

说明：如果想要实践Diagram视图，可以打开包含在本章示例中的DiagramDemo项目。该程序窗体有两个相关联的图表：一个如图1.6所示，另一个则比较复杂，包含有下拉菜单及其条目的图表。

窗体设计器

用户经常会用到的另一个Delphi窗口就是窗体设计器，一种用来将组件放置在窗体上的可视工具。在窗体设计器中，我们可以通过鼠标直接地选择一个组件；而当一个控件隐藏在另一个控件之后或者该控件非常小时，使用对象监视器或对象树状视图这种方法就显得非常方便了。如果一个控件完全地覆盖了另一个控件，可以使用Esc键选择当前控件的父控件。为此，需按Esc键一次或多次来选择窗体；或者按住Shift键，同时单击所选的组件，以取消所选的组件，并默认地选择窗体。

有两种方法可以代替使用鼠标来设置组件的位置：一种方法是设置Left和Top属性的值，另一种方法是按住Ctrl键，同时使用箭头键。在微调组件位置时，使用箭头键是特别有用的（当Snap To Grid选项激活的情况下），就像按住Alt键使用鼠标移动组件的操作一样。如果使用Ctrl+Shift+箭头这样的组合键，则组件将只能以网格间隔进行移动。

按住Shift键的同时使用箭头键可以微调组件的大小。当然，也可以通过Alt键和鼠标的组合完成同样的操作。

为了对齐多个组件或将它们的大小调成一致，可以选定多个组件，并且同时设置所有组件的Top、Left、Width或Height属性。要想选择多个组件，只需要在按下Shift键的同时用鼠标单击这些组件就可以了。如果要选择的所有组件都在一个矩形区域内，可以拖动鼠标“画”一个矩形围住它们。如果要选择子控件（就是说，在面板中的按钮），可以按住Ctrl键，同时在面板内拖动鼠标；否则，就会变成移动面板的操作。当选择了多个组件之后，还可以使用Alignment对话框（与窗体快捷菜单中的Align命令）或Alignment面板（通过View Alignment面板菜单命令访问）来设置它们的相对位置。

当完成了一个窗体的设计后，可以使用编辑菜单中的Lock Controls命令，以避免由于意外造成组件位置的变动。这个命令是非常有帮助的，因为在窗体中，Undo操作的功能非常有限（只能使用Undelete），但是这种设置并不是永久性的。

在其他的特性当中，窗体设计器还提供了一些工具提示：

- 将鼠标移动到一个控件上时，提示框会显示该组件的名称和类型。从Delphi 6开始，Delphi便提供了扩展提示，可以详细地显示控件的位置、大小、选项卡顺序等等。这是为Show Component Captions环境设置添加的新特性。
- 改变一个控件的大小时，提示框会显示当前的大小（Width和Height属性）。当然，该特性只适用于控件，而不适用于非可视组件（在窗体设计器中用图标表示）。
- 移动一个组件时，提示框则会说明当前位置（Left和Top属性）。

最后，可以将DFM（Delphi Form Module）文件保存为普通的文本格式，而不是默认的二进制源格式；并且使用窗体设计器的快捷菜单为一个特定的窗体切换该操作，或者在Environment Options对话框的Preferences页面中为新建立的窗体设置默认值。在同一个页面中，还可以指定一个程序的二级窗体是否在启动时自动创建，而且为每个独立的窗体做出相反的决定（使用Project Options对话框的Form页面）。

将DFM文件以文本格式保存能够更有效地对版本控制系统进行操作。但程序员并不能从该特性上真正受益，因为现在已经可以在Delphi编辑器中使用设计器快捷菜单的特定命令

来打开二进制DFM文件。在另一方面，版本控制系统则需要保存DFM文件的文本版本，这样才能够对比它们，并找出同一个文件的两个版本之间的区别。

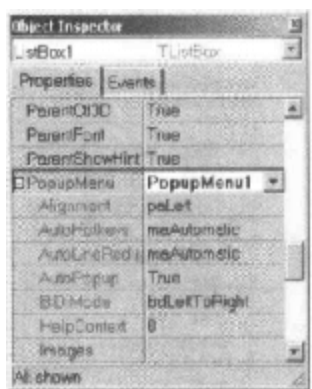
在任何情况下，如果使用文本格式的DFM文件，在将它们编译进程序的可执行文件之前，Delphi都会将其转换为二进制资源格式的DFM。DFM文件会以二进制格式链接到可执行文件中，以减小可执行文件的大小（尽管它们并没有被真正压缩），并改善程序运行时的性能（可以被更快速地装载）。

说明：文本的DFM文件比其二进制版本更容易在不同的Delphi之间移植。当Delphi的旧版本不接受新版本所建立的DFM中某个控件的属性时，旧版本的Delphi仍然可以读取文本DFM文件的其他部分。如果新版本中添加了新的数据类型，则旧版本不能读取新版本的二进制DFM。尽管这听起来好像不太可能，但要知道，64位的操作系统就要出来了。如果拿不准，最好保存为文本的DFM格式。还要注意的，所有版本的Delphi使用bin目录下的命令行工具Convert，都可以支持文本的DFM。最后要注意的是，在Delphi和Kylix中，CLX库都使用XFM扩展名，而不是DFM扩展名。

对象监视器

在设计时要想查看和改变放置在一个窗体（或另一个设计器）中的组件属性，可以使用对象监视器。与早期的Delphi版本相比，目前的对象监视器已经添加了很多的新特性。而对于Delphi 7中引入的最新特性，我们将会用粗体字标注出那些与默认值不同的属性。

另一个重要的变化（在Delphi 6中引入）就是对象监视器可以作为扩展组件被引用。引用其他组件的属性将会以不同的颜色来显示，并且可以通过选择左边的+号来进行扩展，就像在内部子组件中一样。随后，修改被引用的其他组件的属性时，便不必再选择它了。这里当我们对另一个组件（列表框）进行操作时，将会看到在对象监视器中扩展出一个连接组件（一个弹出菜单）。



这种接口－扩展特性也支持子组件，就像新的LabeledEdit控件所显示的那样。对象监视器的一个相关特性就是为被属性所引用的组件提供选择。要想使用该特性，按住Ctrl键并用鼠标左键双击属性值。例如，如果窗体中有一个MainMenu组件，并且我们正在对象监视器中查看窗体的属性，则可以通过将鼠标移动到窗体的Menu属性上，选择MainMenu组件，并按住Ctrl键然后双击该属性值，来选择对象监视器中被标明为该属性的主菜单。

这里是其他一些新近添加到对象监视器中的变化：

- 用对象监视器顶部的列表显示对象的类型，并且允许用户选择一个组件。鉴于用户能够在对象树状视图选择组件（默认情况下放在对象监视器的顶部），所以可以

删除这个列表以节省一些空间。

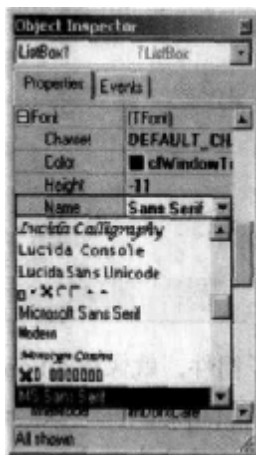
- 引用对象的属性现在用不同的颜色表示，可以在不需要改变选择的情况下被扩展。
- 可以有选择性地查看对象监视器的只读属性。当然，这些属性显示为灰色。
- 对象监视器中有一个新的属性对话框，用于定制不同类型属性的颜色以及该窗口的整体特性。
- 从Delphi 5开始，属性的下拉菜单中就包含图形元素。这种特性可以用于诸如Color和Cursor这样的属性，而且对于连接到一个ImageList的组件的ImageIndex属性来说特别有用。

说明：现在，界面属性在设计时可以通过使用对象监视器来进行配置。这种功能使用了在Kylix/Delphi 6中引入的Interfaced Component Reference模型，其中只要界面由组件实现，组件就可以实现并保存对界面的引用。Interfaced Component References的工作原理与旧的简单组件引用相似，只是界面属性可以绑定到实现所需界面的任何组件上。与组件属性不同的是，界面属性并不局限于一个特定的组件类型（某个类或其派生类）。在对象监视器编辑器中单击某个界面属性的下拉列表时，当前窗体（及相链接的窗体）中的所有组件都将会显示出来。

对象监视器中的下拉字体

Delphi对象监视器具有某些属性的图形化下拉菜单。读者可能想添加一个这样的列表来显示正在选择的字体的实际图像，以便与Font属性中的Name子选项相对应。实际上，Delphi中已经建立了这种功能，但不能使用，因为大多数计算机中并没有安装大量的字体，而且显示这些字体将会降低计算机的运行速度。如果读者想使用该特性的话，就必须在Delphi中安装一个软件包。笔者曾通过OiFontPk软件包实现了这一特性，读者可以在为本书设计的程序范例中找到它。

一旦安装了该软件包，大家就可以将鼠标移动到任何组件的Font属性上，并使用图形化的Name下拉菜单，如下所示：



笔者喜欢并经常使用的是对窗体进行更复杂的二次定制：对整个对象监视器使用一种定制字体，使其文本更加醒目。这种特性对于公开演示是特别有用的。参见附录A可以了解如何获得该附加软件包。

属性类别

Delphi中包含了属性类别的概念，可以通过对象监视器的快捷菜单上的Arrange选项将其激活。如果设置了这个选项，属性将会按照组别来排列，而不是按照字母表顺序排列，且每个属性都可能出现在多个组中。使用类别的概念有助于减小对象监视器的复杂性。可以使用快捷菜单中的View子菜单隐藏给定类别的属性，而不管显示它们的方法（这就是说，即使用户喜欢使用传统的根据名称排列的方式，仍然可以隐藏某些类别的属性）。

对象树状视图

Delphi 5中引入了一种用于数据模块的树状视图，通过这种视图，我们可以看到非可视组件之间的关系，如数据集合、字段、行为等等。Delphi 6扩展了这一思想，为每个设计器（包括普通窗体）都提供一个对象树状视图，而且在默认情况下，将其放置在对象监视器上。

对象树状视图以一种树状的形式显示了窗体中所有组件和对象之间的关系。其中最明显的就是父/子关系：如果将一个面板放到一个窗体当中，并且假设有两个按钮，一个按钮放在这个面板里面，另一个按钮放在这个面板的外面，则这时关系树就将显示出一个按钮属于这个窗体，而另一个按钮属于面板。



要注意的是，树状视图是与对象监视器和窗体设计器同步的。所以，只要在这三种工具的任何一种之中选择了一个条目并且做了修改，那么这个修改也将会同时应用在其他两种工具中。

除了父/子关系之外，对象关系视图还显示了其他的一些关系，如宿主/寄宿关系，组件/子对象关系以及集合/条目关系，另外还有各种特定的关系等等，包括数据集合/连接关系和数据源/数据集合关系。在这里，可以看到一个以关系树形式显示的菜单的结构范例：



有时，树状视图还可以显示“哑”节点，这种节点并不对应一个实际的对象，而是对应一个预定义的对象。作为该特性的一个范例，拖出一个Table组件（从BDE页面中），将可以看到两个关于会话和别名的图标。从技术角度上来讲，对象树状视图使用灰色的图标来

表示不具备设计持久性的组件。它们是真正的组件（在设计的时候和在运行的时候），但是因为它们是在运行时建立的默认对象，不具备可以在设计时被编辑的持久性数据，所以Data Module Designer将不允许编辑它们的属性。如果把一个Table拖放到窗体上，将会看到有些条目的旁边会出现包含在黄色圆圈内的红色问号。这种符号表示部分未定义条目。

对象树状视图支持多种类型的拖放：

- 从组件面板中选择一个组件（通过单击而不是拖拽该组件），并将鼠标移动到关系树上，然后单击组件就可以将其放到这里。这种技巧能够用于将一个组件拖放到合适的容器中（窗体、面板以及其他容器），而不必考虑其表面将可能会被其他的组件完全地覆盖——这样，就可以防止没有事先重新排列的组件被拖放到设计器中。
- 在树状视图中拖放组件，例如，将一个组件从一个容器移动到另一个容器中。而在窗体设计器中，只能通过剪切-粘贴动作来完成这个操作。用移动来代替剪切的好处在于，这些组件之间的连接将不会丢失，而在进行剪切操作时，删除组件将会丢失这些连接。
- 将组件从树状视图拖放到Diagram视图当中，稍后读者将会看到这样的情况。

用鼠标右键单击树状视图中的任何元素，都将显示一个快捷菜单，该快捷菜单与组件在窗体中得到的组件菜单非常相似（在这两种情况下，快捷菜单都可以包含与定制组件编辑器相关的条目）。我们甚至可以从关系树上将这些条目删除。树状视图还可以用做集合编辑器，就像大家在这里看到的一个ListView控件的Columns属性一样。在这种情况下，我们不仅能够重新排列并删除条目，而且还能够向集合中添加新的条目。



提示：通过选定一个窗口，并使用File►Print命令，就可以将对象树状视图的内容打印出来以备归档所用（在快捷菜单中并没有Print这个命令）。

组件面板的秘密

组件面板可以用来选择想要添加到当前设计器中的组件。将鼠标移动到一个组件上便能看到该组件的名称。在Delphi 7中，同时还会显示定义该组件的单元的名称。

组件面板中有很多的选项卡——真的是非常非常的多。所以我们需要隐藏那些与无需使用的组件相关的选项卡，并确认是否组件面板符合要求。在Delphi 7中，也可以通过拖拽选项卡来记录它们。使用环境选项对话框中的Palette页面，可以完全重新安排不同页面上的组件，添加新的元素或将它们从一个页面移动到另一个页面上。

当组件面板上有太多的页面时，需要翻动页面才能找到一个组件。在这种情况下，可以使用一个简单的技巧：以一个较短的名称为这些页面重新命名，这样所有的页面都会显示在屏幕上（显然，各位也已经想到了）。

Delphi 7提供了另外一种新的特性。当一个页面中有太多的组件时，Delphi将会显示一个双下指箭头，通过单击它就可以显示剩下的组件，而不必滚动面板页面。

组件面板的快捷菜单中还有一个Tabs子菜单，该菜单按字母顺序列出了所有面板页面。可以使用这个子菜单来改变活动页面，特别是当需要的页面在屏幕上看不到的时候。

提示：可以设置在组件面板快捷菜单的Tabs子菜单中的条目顺序，使之与其在面板本身中的顺序相同，而不是以字母顺序排列。要想这么做，需要进入Delphi中的Main Window注册表部分（对当前用户来说位于\Software\Borland\Delphi\7.0下），并将Sort Palette Tabs Menu键的值设为0（假的）。

组件面板中真正未公开的特性是“热跟踪（hot-track）”。设置了注册表中的特殊键之后，就可以通过简单地选项卡上移动鼠标来选择一页，而不需要单击操作。同样的特性还可以应用到面板两边的组件滚动按钮上，当一个页面上有太多的组件时该按钮将会出现。要想激活这一隐藏的特性，需要在注册表的HKEY_CURRENT_USER\Software部分的Borland\Delphi\7.0键下添加一个Extras键，并且输入两个字符串值，AutoPaletteSelect和AutoPaletteScroll，同时将这两个值都设为字符串“1”。

复制和粘贴组件

窗体设计器的一个有趣的特性就是能够从一个窗体向另一个窗体复制和粘贴组件或在窗体内复制一个组件。在这个操作过程中，Delphi将会复制所有的属性，并保留相连的事件处理器，而且在必要时还会改动控件的名称（每个窗体中控件的名称必须惟一）。

也可以在窗体设计器和编辑器之间相互复制组件。在将一个组件复制到粘贴板上时，Delphi会把原文描述也放在那里。甚至可以编辑组件的文本版本，将该文本复制到剪贴板上，并且随后把它作为一个新的组件粘贴回窗体当中。例如，如果我们在一个窗体中放置了一个按钮，复制它，然后将其粘贴到一个编辑器中（既可以是Delphi自己的源代码编辑器，也可以是任何其他文字处理器），则会得到如下描述：

```
object Button1: TButton
  Left = 152
  Top = 104
  Width = 75
  Height = 25
  Caption = 'Button1'
  TabOrder = 0
end
```

现在，如果改变了对象的名称，例如它的标题或位置，或者添加了一个新的属性，则这些改变将被复制并粘贴回一个窗体中。这里是一些改动的范例：

```
object Button1: TButton
  Left = 152
  Top = 104
  Width = 75
  Height = 25
  Caption = 'My Button'
```



```
TabOrder = 0  
Font.Name = 'Arial'  
end
```

复制这个描述并将其粘贴回窗体当中，就会在指定位置创建一个按钮，该按钮的标题为Arial字体的My Button。

要想使用这一技巧，需要知道如何编辑一个组件的原文表述，哪些属性对特定的组件是有效的，以及怎样编写字符串属性、已设置属性和其他一些特殊属性的值。当Delphi解释某个组件或窗体的原文描述时，它可能会改变一些与该改变组件相关的其他属性值，也可能会改变这些组件的位置。这样，它就不会与以前的复制重叠。当然，如果我们写入一些完全不正确的东西，并试图将其粘贴到窗体当中，Delphi将会显示一个错误信息，说明什么地方出错了。

可以一次选择几个组件并复制它们到另一个窗体或文本编辑器。当需要对一系列相似的组件进行操作时，这种方法可能会非常有用。也可以复制一个组件到编辑器中，然后重复若干次，并进行相应的改动，然后再次将整个组粘贴到窗体当中。

从组件模板到框架

从一个窗体向另一个窗体复制一个或多个组件时，将会复制它们的所有属性。一种功能更强大的方法是创建一个组件模板，用以复制事件处理器的属性和源代码。通过选择组件面板中的伪组件（pseudo-component）将该模板粘贴到一个新的窗体时，Delphi将会把事件处理器的源代码也复制进去。

要创建一个组件模板，首先要选择一个或多个组件，然后使用Component►Create Component Template菜单命令。这个命令将会打开组件模板信息对话框，用户需要在对话框中输入该模板的名称、该模板在组件面板中应当出现的页面以及一个图标。



默认情况下，组件模板的名称由第一个被选组件名加上单词Template构成。而默认的模板图标则是第一个被选组件的图标，但是也可以用一个图标文件来替换这个默认的图标。为该组件模板所取的名称将用于在组件面板中描述这个模板（当Delphi显示弹出式提示框时）。

所有关于组件模板的信息都保存在一个单独的DELPHI32.DCT文件中，但是并没有办法提取该信息并编辑一个模板。然而，可以将一个组件模板放置到一个新的窗体中并编辑它，然后使用相同的名称作为一个组件模板再次安装。通过这种方式，便可以覆盖以前的定义。

提示：通过将组件模板保存到一个公共目录中，并向注册表中的\Software\Borland\Delphi\7.0\Component Templates键下添加CCLibDir条目以后，一组Delphi程序员就可以共享这个组件模板了。

当不同的窗体需要同一组组件和相关联的事件处理器时，组件模板就显得非常方便了。但问题是，一旦将模板中的一个引用放入窗体后，Delphi就会复制组件及其代码，而复制的

内容不再与模板相关联。由于无法修改模板定义本身，所以也就不可能在使用该模板的所有窗体中进行效果相同的改变。是我们的要求太多了么？并不完全是。为此，我们要在Delphi中引入框架技术。

所谓框架就是一种面板，在设计时它操作起来与窗体相似。首先需要简单地创建一个新框架，将一些控件放入其中，并在事件处理器中添加相应的代码。当一个框架准备好后，打开一个窗体，在组件面板的Standard页面中选择Frame伪组件，并选择一个当前项目可以使用的框架。在将该框架放入窗体之后，就会看到它好像组件一样被复制到窗体中。如果修改了原始框架（在其自己的设计器中），其改变将会反映到框架的每一个引用当中。

在图1.8中有一个简单的例子，称为Frames1。当然，一个屏幕快照并不能真正说明太多问题；如果读者想要深入了解如何使用框架，应当打开该程序或重建一个相似的程序。

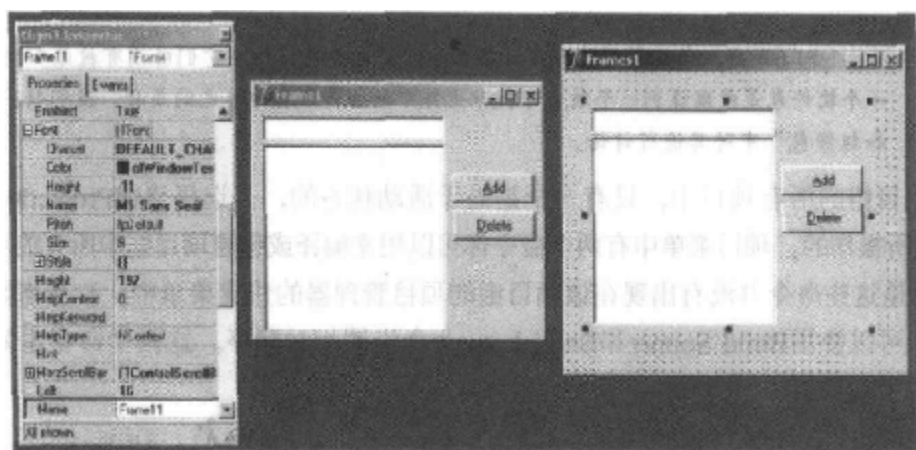


图1.8 范例Frames1演示说明了框架的用法。左边的框架及其右侧窗体中的实例保持同步

与窗体一样，框架也定义了类，所以它们比组件模板更易于适应VCL面向对象模型。第8章“Delphi应用程序的结构”中将会深入探讨VCL，以及关于框架的更具体的描述。读者根据这段简短的介绍会想像出，框架是一种功能非常强大的技术。

管理项目

Delphi的多目标项目管理器（View►Project Manager）是针对项目组来工作的，一个项目组中可以包含有一个或多个项目。例如，一个项目组可以包括一个动态链接库和一个或者多个可执行文件。所有打开的软件包都将在项目管理器的视图中显示为项目，即使它们并没有被添加到项目组中。

在图1.9中，读者可以看到带有一个简单项目组的项目管理器，该项目组包括了当前章节中的所有范例。正如大家能看到的那样，项目管理器基于一个树状视图，该树状视图显示了项目组的分层结构、项目以及构成每个项目的所有窗体和单元。可以使用简单的工具栏与项目管理器中较复杂的快捷菜单对组中各个项目进行操作。快捷菜单是上下文相关的，其选项依选定项目的不同而有所不同。这些菜单选项包括将一个新的或现存的项目添加到一个项目组中、编译或创建一个特定的项目，以及打开一个单元等等。

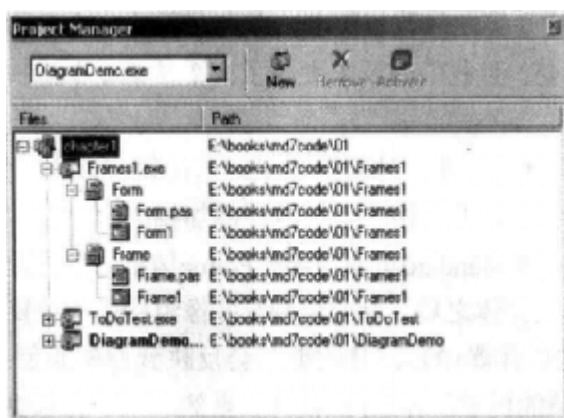


图1.9 Delphi的多目标项目管理器

提示：从Delphi 6开始，项目管理器也可以显示打开的软件包，即使它们并没有被添加到项目组中。

一个软件包是被编译到一个独立可执行文件中的组件或其他单元的集合，我们将在第10章“库和组件包”中对其进行讨论。

在项目组的所有项目中，只有一个处于活动状态的，是选择诸如**Project►Compile**这样的命令所操作的。项目菜单中有两个命令你可以用来编译或创建项目组中所有的项目。（非常奇怪的是这些命令并没有出现在该项目组的项目管理器的快捷菜单中。）当需要创建多个项目时，可以使用**Build Sooner**和**Build Later**命令设置相对顺序。这两个命令主要用于重新排列列表中的项目。

提示：在Delphi 7中，项目管理器中的**local**菜单允许用户通过使用**Make All From Here**或**Build All From Here**命令基于一个给定的项目开始编译。

项目管理器的高级特性之一是，允许用户从Windows文件夹或Windows浏览器中将源代码文件拖拽到项目管理器窗口内的一个项目上，从而将它们添加到这个项目当中（这种拖拽操作方式也支持在代码编辑器中打开文件）。借此，也可以很容易地看到哪一个项目被选定并通过项目管理器窗口顶部的**combo**框或者使用Delphi工具栏中与运行按钮相邻的下指箭头来改变它。

除了添加Pascal文件和项目外，还可以向项目管理器添加Windows资源文件；这些文件将会与项目一起被编译。为此，只需简单地移动到一个项目上，选择**Add**快捷菜单，并选择**Resource**文件（*.rc）作为文件类型。这种资源文件将自动地与项目绑定在一起，甚至没有相应的**\$R**指令也可以。

Delphi使用.BPG扩展名来保存项目组（.BPG代表Borland Project Group）。这种特性源于C++ Builder和以前的Borland C++编译器；当打开一个项目组的源代码时，就会清晰地看到这一点，该代码基本上是C/C++开发环境中的makefile的源代码。这里是一个简单的例子：

```
#-----
VERSION = BWS.01
#-----
ifndef ROOT
ROOT = $(MAKEDIR)\..
```

```

endif
#-----
MAKE = $(ROOT)\bin\make.exe -$(MAKEFLAGS) -f$**
DCC = $(ROOT)\bin\dcc32.exe $**
BRCC = $(ROOT)\bin\brcc32.exe $**
#-----
PROJECTS = Project1.exe
#-----
default: $(PROJECTS)
#-----
Project1.exe: Project1.dpr
    $(DCC)

```

项目选项

项目管理器不能同时设置两个不同项目的选项。因此，开发人员只能从每个项目自己的项目管理器中调用**Project Options**对话框。**Project Options**的第一个页面（**Forms**）列出了在程序启动时应当自动被创建的窗体以及被这个程序手工创建的窗体。其下一个页面（**Application**）用于设定应用程序的名称及其帮助文件的名称，并选择其相应的图标。其他的项目选项则与**Delphi**编译器和链接器、版本信息、以及运行软件包的使用相关。

有两种方法可以用来设置编译器选项。第一种方法是使用**Project Options**对话框中的**Compiler**页面。另外一种方法则是使用**(\$X+)**和**(\$X-)**（**X**代表想要设置的选项）命令在源代码中设置或删除特定的选项。相比而言，第二种方法更为灵活一些，因为这种方法允许开发人员只针对一个特定的源代码文件，甚至是仅仅针对几行代码来修改一个选项。源级别的选项会覆盖编译级别的选项。

所有的项目选项都会与项目一起自动保存到一个带**.DOF**扩展名的独立文件当中。这种文件是一种文本文件，很容易编辑。如果改变了某些默认的选项，则不应将该文件删除。**Delphi**也会将编译器选项以另一种格式保存到一个扩展名为**.CFG**的文件中，用于命令行编译。这两个文件的内容相似，但是格式不同：**dcc**命令行编译器不能使用**.DOF**文件，但是却需要**.CFG**文件。

另一种保存编译器选项的方式是按**Ctrl+O+O**（按住**Ctrl**键的同时按**O**键两次）。这个组合键命令将会在当前单元的顶部插入与当前项目选项相对应的编译器命令（包括所有新的编译器警告设置），如下面的程序清单所示：

```

{$AB,B-,C+,D+,E-,F-,G+,H+,I+,J-,K-,L+,M-,N+,O+,P+,Q-,R-,S-,T-,U-,V+,W-,X+,Y+,Z1}
{$MINSTACKSIZE $00004000}
{$MAXSTACKSIZE $00100000}
{$IMAGEBASE $00400000}
{$APPTYPE GUI}
{$WARN SYMBOL_DEPRECATED ON}
{$WARN SYMBOL_LIBRARY ON}
{$WARN SYMBOL_PLATFORM ON}
{$WARN UNIT_LIBRARY ON}

```

```
{ $WARN UNIT_PLATFORM ON }
{ $WARN UNIT_DEPRECATED ON }
{ $WARN HRESULT_COMPAT ON }
{ $WARN HIDING_MEMBER ON }
{ $WARN HIDDEN_VIRTUAL ON }
{ $WARN GARBAGE ON }
{ $WARN BOUNDS_ERROR ON }
{ $WARN ZERO_NIL_COMPAT ON }
{ $WARN STRING_CONST_TRUNCED ON }
{ $WARN FOR_LOOP_VAR_VARPAR ON }
{ $WARN TYPED_CONST_VARPAR ON }
{ $WARN ASG_TO_TYPED_CONST ON }
{ $WARN CASE_LABEL_RANGE ON }
{ $WARN FOR_VARIABLE ON }
{ $WARN CONSTRUCTING_ABSTRACT ON }
{ $WARN COMPARISON_FALSE ON }
{ $WARN COMPARISON_TRUE ON }
{ $WARN COMPARING_SIGNED_UNSIGNED ON }
{ $WARN COMBINING_SIGNED_UNSIGNED ON }
{ $WARN UNSUPPORTED_CONSTRUCT ON }
{ $WARN FILE_OPEN ON }
{ $WARN FILE_OPEN_UNITSRC ON }
{ $WARN BAD_GLOBAL_SYMBOL ON }
{ $WARN DUPLICATE_CTOR_DTOR ON }
{ $WARN INVALID_DIRECTIVE ON }
{ $WARN PACKAGE_NO_LINK ON }
{ $WARN PACKAGED_THREADVAR ON }
{ $WARN IMPLICIT_IMPORT ON }
{ $WARN HPPEMIT_IGNORED ON }
{ $WARN NO_RETVAL ON }
{ $WARN USE_BEFORE_DEF ON }
{ $WARN FOR_LOOP_VAR_UNDEF ON }
{ $WARN UNIT_NAME_MISMATCH ON }
{ $WARN NO_CFG_FILE_FOUND ON }
{ $WARN MESSAGE_DIRECTIVE ON }
{ $WARN IMPLICIT_VARIANTS ON }
{ $WARN UNICODE_TO_LOCALE ON }
{ $WARN LOCALE_TO_UNICODE ON }
{ $WARN IMAGEBASE_MULTIPLE ON }
{ $WARN SUSPICIOUS_TYPECAST ON }
{ $WARN PRIVATE_PROPACCESSOR ON }
{ $WARN UNSAFE_TYPE OFF }
{ $WARN UNSAFE_CODE OFF }
{ $WARN UNSAFE_CAST OFF }
```

编译与建立项目

有多种方法都可以用来编译一个项目。如果运行项目（按F9或单击工具栏上的Run图标），Delphi就会首先编译它。当Delphi编译项目时，将只会编译那些发生了改变的文件。如果选择了Project►Build All，那么不管文件是否发生改变，所有的文件都将会被编译。但是不应经常使用该命令，因为Delphi通常会自动确定哪些文件发生了改变，同时会根据需要它们进行编译。惟一例外的情况是，改变的某些项目选项必须要使用Build All命令才能生效。

为了创建一个项目，Delphi首先要编译每个源代码文件，生成一个Delphi编译单元（DCU）（只有当DCU文件已经不是最新版本时才需要执行这一步骤）。第二步由链接器来执行，将所有的DCU文件合并到一个可执行文件中，有时还需要从VCL库中合并编译代码（如果还没有决定在运行时是否使用软件包）。第三步就是将一些可选的资源文件绑定到可执行文件中，这些资源文件包括项目的RES文件（用于管理项目的主图标）和窗体的DFM文件。如果读者使用Environment Options对话框的Preferences页面中的Show Compiler Progress选项，就可以更好地理解编译步骤以及操作过程中发生的事件。

警告：Delphi并不总是能够根据用户曾经修改过的其他单元来正确地理解什么时间重建相关的单元。特别是因为用户的干涉而引起编译器逻辑混乱的情况下，如改动文件名称、在IDE外部修改源文件、向磁盘复制回的源文件或DCU文件，或者在搜索路径中有同一单元源文件的多个拷贝，就都会中断编译过程。每当编译器显示一些奇怪的错误信息时，我们首先应该试一下Build All命令，以使所用特性与磁盘上的当前文件重新同步起来。

只有在编辑器中装载了一个项目之后，才可以使用Compile命令。如果没有处于活动状态的项目，而只是装载了一个Pascal源文件，将不能进行编译。然而，如果将源代码文件当做项目装载的话，就可以编辑它了。做法很简单，只需要选择Open Project工具栏按钮，并装载一个PAS文件即可。现在，可以建立一个DCU文件，检查其语法或者编译它。

笔者在前面曾经提到过，Delphi允许用户使用运行软件包，这对程序发布的影响要比编译过程大得多。Delphi软件包是包含有Delphi组件的动态链接库。通过使用软件包，可以极大地减小可执行文件的大小。当然，在这种情况下，只有当运行该程序的计算机中有相应的动态链接库时（如lvcl70.bpl，该文件是相当大的），该可执行程序才可以正确地运行。

如果将小型可执行文件的大小加上该文件所需动态链接库的大小后，则看似较小但是运行软件包所需要的磁盘空间将要远远大于一个看似较大的单一可执行文件所需要的磁盘空间。当然，如果在一个系统中有多多个应用程序，则最终将会节约很多的资源，包括磁盘空间以及在运行时所需的内存消耗。因此，软件包并不是在任何情况下都推荐使用的。我们将在第10章中详细讨论组件包的各种细节问题。

在这两种情况下，Delphi可执行文件的编译速度都是非常快的，并且其相应的应用程序的运行速度与C或C++程序相比也所差无几。经Delphi编译过的代码运行起来要比解释或半编译工具中的同样代码至少快5倍。

编译器消息

正如我们在本章开始时提到的那样（参见“Delphi 7中的扩展编译器消息和搜索结果”小节），除了经典的编辑器消息之外，Delphi 7中还提供了一种新的窗口，其中含有一些错误消息的附加信息。通过使用View►Additional Message Info菜单命令就可以激活这种窗口。该窗口显示的信息都存放在本地的一个文件中，可以从Borland网站上下载到该文件最新的版本。

在Delphi 7中的另一个变化是增强了对编译器警告的控制。Project Options对话框目前包括了一个编译器消息页面，用户可以在这个页面中选择许多个人的警告消息。之所以引入这一特性是因为Delphi 7中有一套新的警告，这套警告与将来Delphi兼容.NET的工具密切相关。

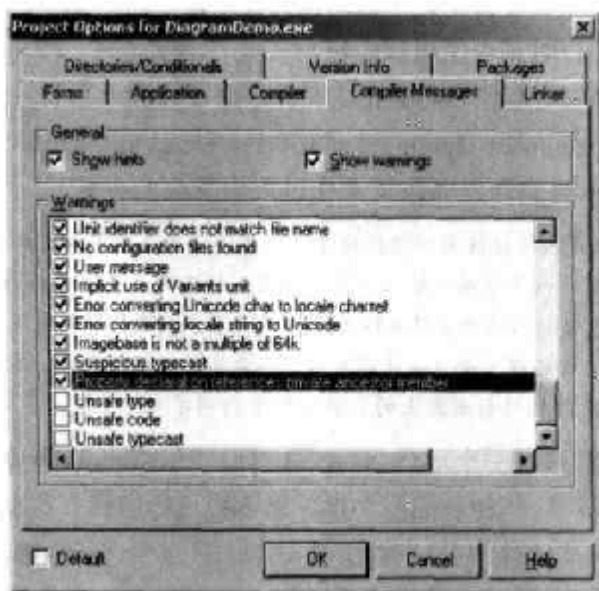


图1.10 Project Options对话框中的新的Compiler Messages页面

也可以通过使用以下这些编译器选项来激活或关闭一些此类警告：

```
{Warn UNSAFE_CODE OFF}  
{Warn UNSAFE_CAST OFF}  
{Warn UNSAFE_TYPE OFF}
```

通常说来，最好把这些设置与程序的源代码相互独立起来——Delphi 7最终将会允许用户做一些事情。

浏览一个项目的类

Delphi中总会包括一种工具用于浏览一个编译项目的符号，尽管这种工具曾经多次易名（从Object Browser到Project Explorer，以及现在的Project Browser）。在Delphi 7中，可以通过View►Browser菜单命令激活项目浏览器（Project Browser）窗口，该窗口如图1.11所示。利用该浏览器能够查看项目类的分层结构，并且查找其符号和源代码行被引用的位置。

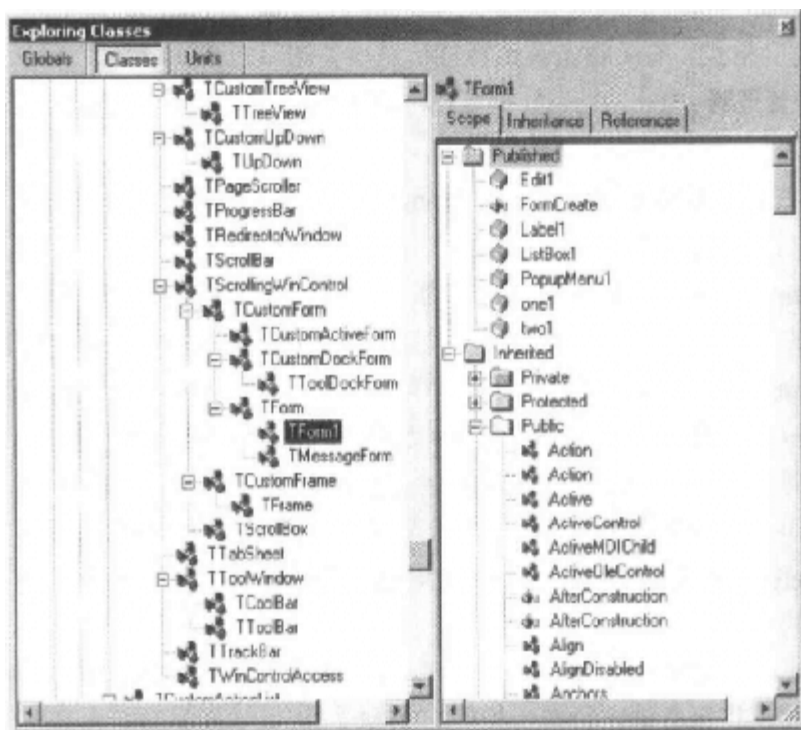


图1.11 项目浏览器

与Code Explorer不同的是，项目浏览器只有在用户重编译项目的时候才会被更新。这种浏览器能够列出类、单元、全局变量等等，并且允许用户选择是否只查看在项目定义的符号，或查看项目以及VCL中所有的符号。可以通过Environment Options的Explorer页面改变Project Browser或Code Explorer的设置，或通过选择Project Explorer快捷菜单中的Properties命令改变其设置。从该窗口中可以看到一些类别是Project Browser特有的，而其他类别则是与这些工具都相关的。

附加及外部Delphi工具

除了IDE之外，安装Delphi时，还会得到其他一些外部工具。其中一些工具，如Database Desktop、Package Collection编辑器（PCE.exe）以及Image编辑器（ImagEdit.exe），都可以直接在IDE的Tool菜单中获得。另外，企业版中还包含有与SQL Monitor（SqlMon.exe）的链接。其他不能直接从IDE中访问的工具还包括许多命令行应用程序，这些程序可以在Delphi的bin目录下找到。例如，这些工具包括一个命令行Delphi编译器（DCC32.exe）、Borland资源编译器（BRC32.exe和BRCC32.exe）以及可执行的阅读器（TDump.exe）。

最后，Delphi本身所带的一些范例程序实际上也是一些很有用的工具，用户可以编译并使用这些范例。在本书中，我们将根据需要讨论一些这样的工具。下面简要介绍一些有用的高级工具，其中的大部分都可以在\Delphi7\bin文件夹及Tools菜单中找到：

Web应用调试器（WebAppDbg.exe）调试Web服务器是在Delphi 6中引进的，用于跟踪发送给应用程序的请求并调试它们。这种调试器在Delphi 7中被重新修改：现在是一种CLX应用，并且其连接性是基于套接字的。我们将在第20章中讨论该工具。

XML映射器 (XmlMapper.exe) 一种用于创建XML转换的工具, 适用于ClientDataSet组件生成的格式。有关该工具的详细信息请参阅第22章。

外部转换管理器 (etm60.exe) 是集成转换管理器的单机版本, 在Delphi 6被首次引入。该外部工具可以被赋予外部转换器。

Borland注册表清除工具 (D7RegClean.exe) 可以帮助用户删除Delphi 7添加到计算机中的所有注册表条目。

TeamSource Delphi提供的一种高级版本控制系统, 在Delphi 5中首次引入。该工具与其以前的形式非常相似, 并且是独立于Delphi安装的。Delphi 7使用的是1.01版的TeamSource。Delphi 6如果安装了相应的补丁程序后, 也可以使用1.01版的TeamSource。

WinSight (Ws32.exe) 一种Windows“消息侦察”程序, 该程序存在于bin目录下。

数据库浏览器 使用bin目录下的DBExplor.exe程序时, 该浏览器可以通过Delphi IDE激活或作为一种单独的工具使用。因为该工具需要BDE, 因此该工具目前并不常用。

OpenHelp (oh.exe) 一种可以管理Delphi帮助文件的结构并将第三方文件集成到帮助系统中的工具。

Convert (Convert.exe) 一种命令行工具, 既可以用来将DFM文件转换成相同的原文描述, 也可以将原文描述转换为相应的DFM文件。

Turbo Grep (Grep.exe) 一种命令行搜索工具, 比内置的Find In Files机制更快, 但不太容易使用。

Turbo注册服务器 (TRegSvr.exe) 用于注册ActiveX库和COM服务器的一种工具。该工具的源代码位于\Demos\ActiveX\TregSvr目录下。

资源浏览器 一种功能非常强大的资源查看工具(但不是一个成熟的资源编辑器), 该工具存放于\Demos\ResXplor目录下。

资源工作室 一种旧的16位资源编辑器, 但是也可以管理Win32资源文件。Delphi安装光碟包括一个该工具单独的安装程序。该工具以前包含在Windows的Borland C++和Pascal编译器中, 但比后来出现的标准Microsoft资源编辑器要好得多。尽管其用户界面没有被更新, 并且不能处理长文件名, 但该工具在建立定制或特定资源时仍然是非常有用的。它还允许用户研究现有可执行文件的资源。

系统生成的文件

Delphi为每个项目都生成一些不同的文件, 大家应当知道这些文件都是什么, 以及它们是怎样被命名的。一般来说, 对文件如何命名有影响的因素有两个: 为项目及其单元所起的名字, 以及Delphi中使用的预定义的文件扩展名。表1.1列出了在Delphi项目目录中可以看到的文件扩展名。从该表中也可以看出这些文件是在什么时间、什么情况下被创建的, 以及它们对将来编译的重要性如何等等。

表1.1 Delphi项目文件扩展名

扩展名	文件类型和描述	创建时间	是否需要编译
.BMP、.ICO、 .CUR	位图文件、图标文件和鼠标文件； 用于存储位图的标准Windows文件	开发：图像 编辑器	通常不需要，但是在运行时或 以后编辑时可能会需要
.BPG	Borland项目组：新的多目标项目管 理器使用的文件，这是一种make file文件	开发	同时重编辑组中所有项目时需 要
.BPL	Borland程序包库：在Delphi环境设 计或应用程序运行时用到的包含有 VCL组件的一种文件（这种文件在 Delphi 3中使用.DPL扩展名）	编译：链接	向其他的Delphi程序开发人员 或最终用户（可选）发布软件 包时需要
.CAB	Delphi中用于开发Web配置时使用 的一种Microsoft Cabinet压缩文件 格式。一个CAB文件可以存储多个 个压缩文件	编译	向用户发布时需要
.CFG	带项目选项的配置文件。与DOF文 件相类似	开发	只在特定编辑选项已被设定的 情况下需要
.DCP	Delphi编辑软件包：一种包含关于 被编译进软件包当中的代码符号信 息的文件。这种文件并不包含编译 代码，这些代码实际上存储在DCU 或BPL文件中	编译	当运行实时软件包时需要。只 向其他与BPL文件相关的开发 人员发布。只要拥有DCP文件 和BPL文件（没有DCU文件）， 就可以用一个软件包中的单元 来编译一个应用
.DCU	Delphi编译单元：一个Pascal文件 编译的结果	编译	只有当源代码不能使用的时候 才需要。DCU文件对于编写单 元来说就是一个中间步骤，所 以它可以加快编译速度
.DDP	Delphi Diagram Portfolio，用于编 辑器的图表视图中（在Delphi 5中 是.DTI）	开发	不需要。这种文件只用于存放 “design-time only”信息，结 果程序并不需要这种文件， 但程序员却非常需要它
.DFM	Delphi窗体文件：一种用于描述一 个窗体（或一个数据模块）及其组 件属性的二进制文件	开发	需要。每个窗体被同时存放在 PAS和DFM文件中
.~DF	Delphi窗体文件（DFM）的备份	开发	不需要。当保存一个与窗体及 窗体文件相关的单元的新版本 时，该文件才会被创建
.DFN	集成转换环境的支持文件（每个窗 体和每个目标语言都需要一个DFN 文件）	开发（ITE）	需要（这些文件包含有在转换 管理器中编辑的已转换的字符 串）
.DLL	动态链接库：可执行文件的另一种 版本	编辑：链接	参见.EXE

(续表)

扩展名	文件类型和描述	创建时间	是否需要编译
.DOF	Delphi选项文件: 包含当前项目选项设置的一种文本文件	开发	只有当特定编辑器选项设定时才需要
.DPK, 现名为.DPKW和.DPKL	Delphi软件包: 一个软件包的项目源代码文件(或Windows或Linux所用的一个特定项目文件)	开发	需要
.DPR	Delphi项目文件(该文件实际上包含了Pascal源代码)	开发	需要
.~DP	Delphi项目文件(.DPR)的备份	开发	不需要。当保存一个项目文件的新版本时, 该文件自动生成
.DSK	桌面文件: 包含Delphi窗口位置、编辑器中打开的文件, 以及其他桌面设置等信息	开发	不需要。如果将项目复制到新目录中, 则应当删除该文件
.DSM	Delphi符号模块: 存储所有的浏览器符号信息	编译(只在保存符号选项被设置时)	不需要。当不能重新编译一个项目时, 对象浏览器将会使用这种文件而不是内存中的数据
.EXE	可执行文件: 开发人员创建的Windows应用程序	编译: 链接	不需要。这是由开发人员发布的文件, 包括了所有编译了的单元、窗体和资源
.HTM	或HTML, 超文本标记语言: 该文件格式用于互联网网页	一个Active窗体的Web应用	不需要。该格式文件和项目编译无关
.LIC	与一个OCX文件相关的许可文件	ActiveX向导和其他工具	不需要。在其他开发环境中使用控件时需要
.OBJ	对象(已编译)文件, C/C++中的典型文件	编译的中间步骤, 通常Delphi中不使用	在将Delphi与C++编译代码合并到一起时需要
.OCX	OLE控制扩展: 这是DLL的一种特殊版本, 包含ActiveX控件或窗体	编译: 链接	参见.EXE
.PAS	Pascal文件: 一个Pascal单元的源代码, 可以是一个与窗体相关的单元, 也可以是一个独立的单元	开发	需要
.~PA	Pascal备份文件(.PAS)	开发	不需要。当保存源代码的新版本时, 由Delphi自动生成
.RES、.RC	资源文件: 与项目相关联的二进制文件, 通常包含项目的图标。可以将这种类型的其他文件添加到一个项目中。当创建定制资源文件时, 也可能使用原文格式.RC	开发选项对话框。ITE(集成转换环境)可生成带有特殊注释的资源文件	需要。一个应用程序的主RES文件可以根据Project Options对话框中Application页面上的信息由Delphi来重新建立

(续表)

扩展名	文件类型和描述	创建时间	是否需要编译
.RPS	翻译库 (集成转换环境的组成部分)	开发 (ITE)	不需要。管理转换时需要
.TLB	类型库: 自动建立或由类型库编辑器为OLE服务器应用所建立的一种文件	开发	这是一种其他OLE程序可能需要的文件
.TODO	to-do列表文件, 包含与整个项目相关的条目	开发	不需要。该文件包含有编程人员的注释
.UDL	Microsoft数据链接	开发	ADO用该文件引用数据提供者。与BDE中的别名 (参见第12章) 相类似

在Delphi中, 除了这些在项目开发时产生的文件之外, 还有很多其他的由IDE本身产生和使用的文件。在表1.2中, 我们列出了一些需要了解的扩展名。其中大部分文件格式都是专有的或未公开的, 所以开发人员可以处理的文件很少。

表1.2 其他Delphi IDE定制文件扩展名

扩展名	文件类型
.DCI	Delphi代码模板
.DRO	Delphi的对象库 (该对象库应当可以通过Tools►Repository命令进行修改)
.DMT	Delphi菜单模板
.DBI	数据库浏览器信息
.DEM	Delphi编辑掩码 (带有用于编辑掩码的国家专有格式的文件)
.DCT	Delphi组件模板
.DST	桌面设置文件 (每个已经定义的桌面设置都有一个.DST文件)

查看源代码文件

刚刚已经列出了与Delphi应用程序开发相关的一些文件, 下面我们将花一些时间来探讨一下它们的实际格式。基本的Delphi文件都是Pascal源代码文件, 而且实际上是普通的ASCII文本文件。在编辑器中看到的那些粗体、斜体以及彩色的文本都只是为了突出语法, 其实并没有存于文件当中。

提示: 在本书的程序清单中, 编辑器中的粗体表示关键字, 而斜体则用来表示字符串和注释。

对于一个窗体来说, Pascal文件包含窗体类声明和事件处理器的源代码。在Object Inspector (对象检验器) 中设置的属性值存储在一个单独的窗体描述文件中 (扩展名为.DFM)。惟一的例外就是Name属性, 该属性在窗体声明中引用窗体的组件。

默认情况下, DFM文件是窗体的文本表示, 但是这种文件也能够被存储为二进制Windows Resource格式。用户可以在Environment Options对话框的Designer页面中为新项目设置自己想要使用的格式, 或使用窗体快捷菜单中的Text DFM命令转换不同的窗体格式。普通文本编辑器只能读取文本格式, 但是通过Delphi编辑器则可以装载两种格式的DFM文件。

在需要的时候，首先应将其转化为一种原文描述。要打开一个窗体的原文描述（不管是哪种格式），最简单的方法就是从窗体设计器的快捷菜单上选择**View As Text**命令。这个命令会关闭并保存窗体，然后在编辑器中打开DFM文件。当然，随后还可以使用编辑器窗口的快捷菜单上的**View As Form**命令返回到窗体中。

也可以编辑一个窗体的原文描述，尽管这样做需要非常小心才行。只要保存文件，该文件就会被分析，以重新产生窗体。如果已经做了不正确的修改，编译过程就会停止，并产生一条错误消息。在能够重新打开一个窗体之前，需要首先更正DFM文件中的内容。由于这个原因，在各位精通Delphi编程之前，我们建议读者不要手工改动窗体的原文描述。

提示：在本书中我们经常会给读者提供一些DFM文件的摘录。在大部分摘录中，将只选择相关的组件或属性；通常会删除掉位置的属性、二进制值以及一些没有提供有用信息的行。

除了这两种描述窗体的文件（PAS和DFM）之外，第三种对重新建立应用程序至关重要的文件是Delphi项目文件（DPR），这是又一种Pascal源代码文件。该文件可自动生成，而且很少需要开发人员手工改动。可以通过**Project►View Source**菜单命令来查看该文件。

其他一些由IDE生成的不太相关的文件使用Windows INI文件结构。在这些文件中，每段由一个名字（用一个方括号间隔）表示。这里是一段选项文件（DOF）代码举例：

```
[Compiler]
A=1
B=0
ShowHints=1
ShowWarnings=1

[Linker]
MinStackSize=16384
MaxStackSize=1048576
ImageBase=4194304

[Parameters]
RunParams=
HostApplication=
```

桌面文件（DSK）也使用相同的结构，用于保存特定项目的Delphi IDE状态、列出每个窗口的位置。这里是一小段摘录：

```
[MainWindow]
Create=1
Visible=1
State=0
Left=2
Top=0
Width=800
Height=97
```

对象库

可以使用Delphi中的菜单命令来创建新窗体、新应用程序、新数据模块、新组件等等。这些命令都位于File►New菜单和其他一些下拉菜单中。如果仅仅选择File►New►Other, Delphi将会打开对象库(Object Repository)。可以用它创建任何种类的新元素:窗体、应用程序、数据模块、线程对象、库、组件、白动对象等等。

New对话框(如图11.2所示)包含有几个页面,包括了用户可以创建的所有新元素、存储在库中的现有窗体和项目、Delphi向导以及当前项目的窗体(用于可视窗体继承)。该对话框中的页面和条目视Delphi版本的不同而会有所不同,所以这里我们就不再对其一一列举了。

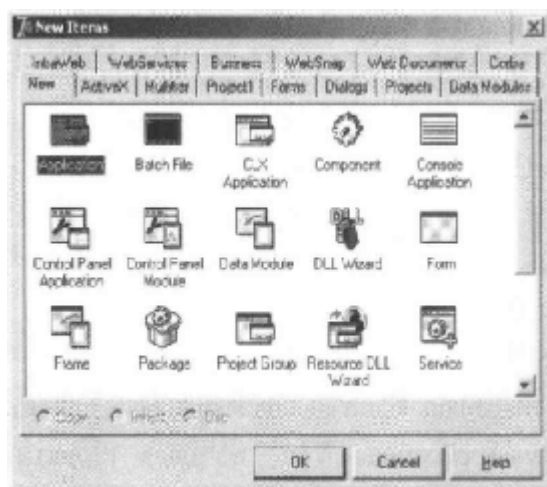


图1.12 New Items对话框的首页,通常称为对象库

提示: 可以通过使用对象库提供了一种快捷菜单来根据不同的方式排列这些条目(根据名字、作者、日期或描述来排列),或以不同的视图方式来显示(大图标、小图标、列表和详细资料),其中的详细资料视图方式可以提供关于工具的描述、作者以及日期等信息。当需要查找自己以前已经添加到库中的向导、项目或窗体时,这些信息就会特别重要了。

最简单定制对象库的方法就是将新的项目、窗体和数据模块作为模板来添加。也可以添加一些新的页面并排列这些页面上的条目(除了New和当前项目这两种页面)。向Delphi对象库中添加新的模板就像是使用现成的模板来创建一个应用程序那样,非常简单。当已经有一个现成的应用程序,并且想以此为基础进一步开发相类似的程序时,就可以将该程序的当前状态保存到一个模板中,待将来直接使用。要实现上述操作,只需要使用Project►Add To Repository命令,并填写其对话框就可以了。

和向一个对象库中添加新的项目模板一样,还可以添加新的窗体模板。只需移动到想添加的窗体上,并从它的快捷菜单中选择Add To Repository命令,然后在对话框中填写相应的标题、描述、作者、页和图标就可以了。要注意的是,将一个项目或窗体模板复制到库中,接着又将其复制到另一个目录中时,只需要简单地进行一次复制-粘贴的操作,这与手工复制文件没有什么两样。

空项目模板

当开始一个新的项目时，Delphi也会自动打开一个空白窗体。但是，如果需要一个基于窗体对象或向导的新项目时，就不需要这种窗体了。为了解决这个问题，可以将一个空的项目模板添加到Gallery中。

实现这一目的所需的步骤很简单：

1. 像平常一样创建一个新的项目。
2. 从项目中删除其惟一的窗体。
3. 将该项目添加到模板中，并命名它为Empty Project。

从对象库中选择这个项目有两个好处：项目不含窗体，并且我们可以选择一个目录，将该项目模板的文件复制到这个目录下。当然这种方法也存在一个缺点——必须要记着使用File►Save Project As命令来为该项目重新起一个名字，否则无论怎样保存项目，Delphi都将会自动使用模板中默认的名字。

如果要进一步定制对象库，可以使用Tools►Repository命令。该命令会打开对象库的对话框，供用户将条目移动到不同的页面，添加新的元素或者删除现有元素，甚至是添加新的页面、改名或者删除、修改其顺序。对象库设置中一个重要的因素就是使用默认配置：

- 当创建一个新的窗体时（File►New Form），使用项目列表下的New Form复选框来指定一个窗体作为要使用的窗体。
- 如果没有特定的新项目选择，而要创建一个新应用程序的主窗体时（File►New Application），使用Main Form复选框来指定窗体的类型。
- 当使用File►New Application命令时，使用New Project复选框（选择一个项目后才会出现），标记Delphi将要使用的默认项目。

在对象库中只有一个窗体和一个项目可以同时拥有这三种设置，并会在其图标上用一个特殊的符号进行标记。如果没有任何项目被选做新项目，Delphi就会基于被标记为主窗体的窗体来创建一个默认的项目。如果没有窗体被标记为主窗体，Delphi就会用一个空的窗体来创建一个默认项目。

在作用于对象库时，所使用的是保存在Delphi主目录下OBJREPOS子目录中的窗体和模块。同时，如果直接使用而不是复制一个窗体或其他任何对象，则最终在该目录下就会出现一些项目文件。了解对象库的工作方法是非常重要的，因为在修改存储于对象库中的一个项目或一个对象时，最好的办法就是直接操作源文件，而不是来回地向对象库中复制数据。

安装新的动态链接库（DLL）向导

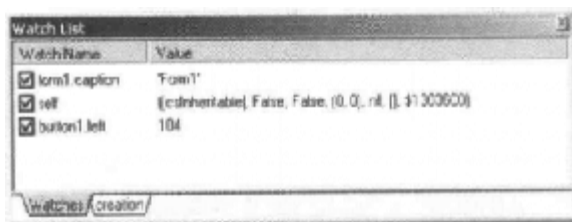
从技术角度上来看，新向导可以划分为两种不同的形式：作为组件或软件包的一部分发布，或作为单独的动态链接库发布。在第一种情况下，安装它们和安装组件或软件包的方式相同——使用Components►Install Packages菜单命令，并单击Add按钮。而当收到一个单独的动态链接库时，应当将这个动态链接库的名字添加到Windows注册表的\Software\Borland\Delphi\7.0\Experts键中，并将向导动态链接库的路径和文件名作为文本使用就可以了。通过查看Experts键下已经存在的条目能够知道如何输入路径。

Delphi 7中调试器的更新

当我们在Delphi的集成开发环境中运行一个程序时，通常你都会在一个集成的调试器中启动。你可以设置中断点、逐行执行代码并研究其内部细节问题，包括所执行的汇编程序以及CPU寄存器的使用情况。本书由于篇幅所限，并没有对Delphi中的调试器进行介绍。请参阅附录C，可以获得关于该主题的相关信息。尽管如此，这里我还是需要简要地介绍一些新的调试器的特性。

首先，在Delphi 7中的运行参数对话框可以让你为正在调试的程序设置工作目录。这就意味着你可以指定当前目录，而不必使用已经编辑进程序的那个目录。

另一个相应的改变是与监视列表（Watch List）相关的。Delphi现在提供了多个选项卡，使你能够同时观察多个正在运行的监视，这些不同的监视分别处理你正在调试的程序的不同区域。这样就可以避免在一个窗口中显示多个监视的混乱情况。你还可以使用快捷菜单向监视列表中添加一个新的组，然后同样可以调整列标题栏的外观以及使用复选框来逐个选取相应的监视。



说明：本书并没有介绍Delphi调试器，但这也是一个非常重要的主题。请参阅附录C，其中提到了一些在线资料，并介绍了如何下载关于Delphi调试问题的免费资料。

小结

本章主要针对Delphi 7编程环境中出现的新特性和更高级特性做了简单概述，同时针对一些在旧的Delphi版本中不常用到的特性提供了使用技巧及建议。我们这里并没有对IDE进行详细描述，部分原因是因为，通常来说直接开始使用Delphi比阅读如何使用它来得更容易一些。而且Delphi中还有一份详细的帮助文件，它描述了新的简单项目的开发和环境。同时，我们也认为读者应该已经对旧版本的Delphi或与其相似的开发环境有所了解。

我们将在下一章中研究Delphi编程语言，然后进一步学习Delphi中的运行时库（RTL）和类库。

第2章 Delphi编程语言

Delphi开发环境是基于Pascal编程语言Object Pascal的面向对象的一种扩展。近来，Borland声称之所以将这种语言称为“Delphi语言”，是因为本公司想要说明Kylix使用的就是Delphi语言，并且Borland将把Delphi语言用到Microsoft.NET平台上。由于多年的习惯，我将交替地使用这两种叫法。

大多数现代编程语言都支持面向对象编程（OOP）。OOP语言基于三个基本概念：封装（通常通过类实现）、继承、多态（或滞后绑定）。尽管人们可以在不了解OOP语言的这些核心特性的情况下编写Delphi程序，但只有充分理解这种编程语言后才能完全掌握这种编程环境。

说明：由于版面限制，加上这门语言近几年也没有太大的变化，故本章将只对这种语言做一个非常简单的介绍。在笔者的网站上，读者可以从以前版本的书中看到更详细的描述（详见附录C“免费的Delphi辅助用书”），其中也包括Essential Pascal——标准Pascal语言的一个完整介绍。

本章主要包括以下内容：

- 类与对象
- 封装：private和public
- 使用属性
- 构造器
- 对象与内存
- 继承
- 虚拟方法与多态性
- 运行时类型信息
- 接口
- 异常
- 类引用

核心语言特性

Delphi语言是经典Pascal语言的一种OOP扩展，Borland公司多年来一直致力于Turbo Pascal编译器的研究。与C语言相比，Pascal语言的语法非常详细并且具有更强的可读性。它的OOP扩展遵循了相同的方法，与近来其他OOP种类的语言（从Java到C#）功能一样强大。

即使核心语言在不断地变化，但这些变化很少会影响到程序员每天编程的需要。例如，在Delphi 6中，Borland公司对核心语言添加了如下一些新的特性，它们或多或少都与Kylix——Delphi的Linux版本有关：

- 一个新的条件编译提示指令（\$IF）

- 一系列提示指令（`platform`、`deprecated`和`library`，其中只有第一个可以被用于任何范围）和用于关闭它们的新的`$WARN`提示指令
- 一个`$MESSAGE`提示指令，用于在编译器消息之间发布定制信息

而Delphi 7添加了三个额外的编译器警告：`unsafe type`、`unsafe code`和`unsafe cast`。当用户的操作不能在Microsoft.NET平台（更多信息请参阅第25章“Delphi for .NET Preview: 语言和RTL”）上产生安全“托管”代码的情况下，这些警告就会被发布。

另一个变化与单元名称有关，现在单元名称可以由被点号分隔的多个单词构成，如`marco.test`单元，被保存在`marco.test.pas`文件中。这个特性将有助于支持命名空间，并且在用于.NET的Delphi和用于Windows的Delphi编译器的将来版本中，提供更灵活的单元引用，但是在Delphi 7中该特性只被有限地使用。

类与对象

Delphi是基于OOP概念的，特别是在新类类型的定义中。OOP的使用在可视开发环境中得到了部分增强，这是因为，对于每个在设计时定义的一个新窗体来说，Delphi会自动定义一个新的类。另外，每个可视放置在一个窗体中的组件实际上是一个类的类型对象，而且该类可以在系统库中获得，或者被添加到系统库中。

说明：类和对象这两个词是很常用的，但往往会用错，所以让我们先来统一对它们的定义。一个类是一种用户定义的数据类型，有一种状态（它的表现或内部数据）或一些操作（它的行为或它的方法）。一个对象是类的一个实例，或由类定义的数据类型的一个变量。对象是真正的实体。当程序运行时，对象会为其内部表现占用一些内存。对象和类之间的关系与变量和类型之间的关系是相同的。

就像在大多数其他现代OOP语言（包括Java和C#）中一样，在Delphi中，一个类的类型变量不会为对象提供存储，而只是在内存中为对象提供一个指针或引用。在使用对象之前，必须创建一个新的实例或将一个现有的实例分配给变量，以此来为其分配内存：

```
var
  Obj1, Obj2: TMyClass;
begin
  // assign a newly created object
  Obj1 := TMyClass.Create;
  // assign to an existing object
  Obj2 := ExistingObject;
```

对`Create`的调用会激活一个默认的构造器，该构造器对每个类都有效，除非某个类重新定义了它（稍后将会介绍）。为了在Delphi中声明一个新的带有一些本地数据字段和一些方法的类数据类型，可使用下面的语法：

```
type
  TDate = class
    Month, Day, Year: Integer;
    procedure SetValue (m, d, y: Integer);
    function LeapYear: Boolean;
  end;
```

说明：Delphi中有一个约定，使用字母T作为每个用户编写的类及其他各种类型（T代表Type）的前缀。这只是一个约定——对编译器来说，T和别的字母没什么两样——不过，这种做法很常见，因此遵循它会使开发人员更容易理解他人的代码。

方法可以使用**function**或**procedure**关键字定义，这取决于它是否含有一个返回值。在类的定义中，方法只能被声明；它们接着会在同一个单元的实现部分中被定义。如下所示，可以使用所属类的名字作为每个方法名称的前缀，并使用点号做为分隔符：

```
procedure TDate.SetValue (m, d, y: Integer);
begin
    Month := m;
    Day := d;
    Year := y;
end;

function TDate.LeapYear: Boolean;
begin
    // call IsLeapYear in SysUtils.pas
    Result := IsLeapYear (Year);
end;
```

提示：当光标位于类定义中时，如果按下Ctrl+Shift+C组合键，Delphi编辑器的类自动生成特性将会为一个类中声明的方法构造定义的基干。

此例说明的是如何使用前面定义的类的对象：

```
var
    ADay: TDate;
begin
    // create an object
    ADay := TDate.Create;
    try
        // use the object
        ADay.SetValue (1, 1, 2000);
        if ADay.LeapYear then
            ShowMessage ('Leap year: ' + IntToStr (ADay.Year));
    finally
        // destroy the object
        ADay.Free;
    end;
end;
```

注意，ADay.LeapYear是与ADay.Year相似的一种表达形式，尽管前者是一种函数调用，而后者是一个直接数据访问。可以有选择性地，在没有参数的一个函数调用之后添加圆括号。读者可以在Dates1范例的源代码中发现前面的代码片段；惟一的差别在于，该程序将会基于编辑框中提供的年份来创建一个日期。

说明：上面的代码片段使用了一个try/finally块，以保证万一代码中出现异常时能够析构对象。读者可以在本章结束时找到对异常这一主题的介绍。

关于方法的更多信息

还有很多关于方法的信息。这里是一些Delphi中一些有用特性的简单介绍：

- Delphi支持overloading方法。这意味着如果使用overload关键字标志该方法，你就可以有两个具有相同名字的方法，而且两个方法的参数列表是相当不同的。通过检查参数，编译器可以确定用户想调用那个版本。
- 方法可以有一个或多个带有默认值的参数。如果这些参数在方法调用时被忽略，它们将会得到默认值。
- 在一个方法中，可以使用Self关键字访问当前对象。当引用对象的本地数据时，对Self的引用就是隐含的了。例如，在前面所列的Tdate类的SetValue方法中，使用了Month来引用当前对象的一个字段，并且编译器会将Month转换成Self.Month。
- 可以定义由class关键字标志的类方法。一个类方法必须能够作用于所有的对象实例，这是因为要将其应用到类的一个对象或整个类中。Delphi没有任何方式可以定义类数据（当前），但是，程序员可以通过在定义类的单元的实现部分添加全局数据来模拟这个功能。
- 默认情况下，方法使用register调用约定：（简单的）参数和返回值要从调用代码传递到函数中，并且使用CPU寄存器而不是堆栈返回。这个过程可以使方法调用更快捷。

动态地创建组件

为了强调Delphi组件与其他对象没有太多区别的这个事实（同时说明Self关键字的使用方法），笔者编写了一个CreateComps范例。这个程序有一个不带组件的窗体和一个用于其OnMouseDown事件的处理器，之所以选择它是因为它将鼠标单击的位置作为一个参数接收（与OnClick事件不同）。我们需要这个信息，以便在那个位置创建一个按钮组件。下面是该方法的代码：

```
procedure TForm1.FormMouseDown (Sender: TObject;  
    Button: TMouseButton; Shift: TShiftState; X, Y: Integer);  
var  
    Btn: TButton;  
begin  
    Btn := TButton.Create (Self);  
    Btn.Parent := Self;  
    Btn.Left := X;  
    Btn.Top := Y;  
    Btn.Width := Btn.Width + 50;  
    Btn.Caption := Format ('Button at %d, %d', [X, Y]);  
end;
```

这段代码的作用是在鼠标单击的位置创建按钮，正如读者在图2.1中所看到的那样。在该代码中，特别要注意Self关键字的使用——同时作为Create方法的参数（以指定组件的所有者）和Parent属性的值。在第4章“核心库类”中将讨论这两个要素（所有权和Parent属性）。

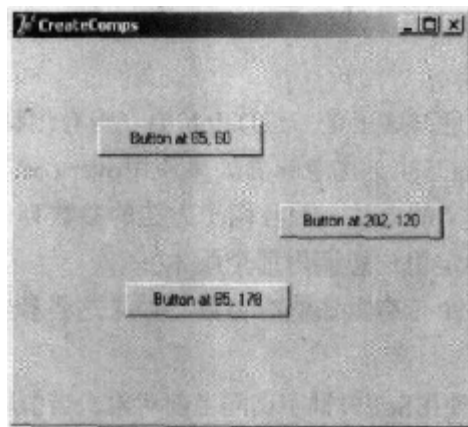


图2.1 CreateComps的示例输出，用于在运行时创建按钮组件

当如此编写代码时，大家可能想到使用Form1变量而不是Self。在这个特定的例子中，这样更改不会造成任何实际的区别，但是如果一个窗体有多个实例，那么使用Form1将会是错误的。实际上，如果Form1变量引用第一个照此创建的窗体，那么通过单击相同类型的另一个窗体，新的按钮将会被显示在第一个窗体中。此时，按钮的Owner和Parent将是Form1，而不是用户单击过的那个窗体。通常来说，在请求当前对象时，引用类的一个特殊实例是不好的OOP行为。

封装

一个类可以含有任意数量的数据和任意多个对象方法。然而，对于一个好的面向对象的方法来说，使用的数据应被隐含或封装在使用它的类中。例如，当访问一个日期时，由它自己改变日期的值是没有意义的。事实上，改变日期的值可能导致一个无效的日期，如2月30日。而使用对象方法来访问一个对象的内部表现会减少产生错误的机会，因为对象方法可以检测日期是否有效，并拒绝改动无效的新值。封装是很重要的，因为它允许类的编写者在将来新的版本中，修改内部的表现。

封装的概念十分简单，经常可以把其想像为一个“黑盒子”。人们并不知道其内部是什么：只能知道怎样与黑盒子接口，或使用它而不管其内部结构。“怎样使用”这部分，称为类的接口，它允许程序的其他部分访问和使用该类的对象。然而，使用对象时，对象的大部分代码都是隐含的。用户很少能知道对象有哪些内部数据，而且一般没有办法可以直接访问数据。当然，可以使用对象方法来访问数据，这与非法的访问不同。相对于信息隐含这样一个经典的编程概念来说，这就是面向对象的方法。然而，在Delphi中，通过属性会有其他等级的隐含，我们将在本章稍后部分看到。

Delphi实现了这种基于类的封装，但仍支持经典的、基于模块的、使用单元结构的封装。假如有一个uses语句引用回定义标识符的单元，则在一个单元的接口部分中声明的每个标识符将会为程序其他单元可见。另一方面，声明在单元的实现部分中的标识符对那个单元来说是本地的。

private、protected和public

对基于类的封装来说，Delphi使用了三个访问标识符：**private**、**public**和**protected**。第四个是**published**，控制着运行时类型信息（RTTI）和设计时信息（其详细情况将在第4章介绍），但是它提供了和**public**相同的程序访问性。这里是三个经典的访问标识符：

- **private**指令专门用于一个类的字段和对象方法，在声明类的单元外这个类不能被访问。
- **protected**指令用于表示对象方法和字段具有有限的可见性。**protected**元素只能被当前类和它的子类访问。更精确地说，只有同一个单元中的类、子类和任何代码可以访问**protected**成员，这将会在本章稍后的“访问其他类的保护数据（或保护的数据窃用）”小节展开一个有趣的讨论。我们将在“保护字段和封装”小节中讨论这个关键字。
- **public**指令专门用于表示那些可以被程序代码中的任意部分访问的数据和对象方法，当然也包括在定义它们的单元中。

一般情况下，一个类的字段应该是专用的，而对象方法则应该是公用的。然而，也不总是这种情况。如果某对象方法只需要在内部完成一些部分地计算或实现属性，那么该方法也可以是专用或受保护的。字段可以被声明为受保护的，这样就可以在子类中对它们进行处理，尽管这并不是一个好的OOP行为。

警告：访问标识符只能限制单元外面的代码访问单元接口部分中声明的一些确定的类成员。这意味着，如果两个类在同一个单元中，则它们的专用字段将不受保护。

作为一个例子，考虑这个TDate类的新版本：

```
type
  TDate = class
  private
    Month, Day, Year: Integer;
  public
    procedure SetValue (y, m, d: Integer); overload;
    procedure SetValue (NewDate: TDateTime); overload;
    function LeapYear: Boolean;
    function GetText: string;
    procedure Increase;
  end;
```

读者可能还会考虑添加其他一些函数，如GetDay、GetMonth和GetYear，它们只返回相应的专用数据，但并不总是需要相似的直接数据访问函数。为每个字段提供访问函数可能会减小封装，并使得对类的内部改动变得困难起来。只有当访问函数属于类（正在实现）的逻辑接口时，这些函数才应该被提供。

另一个新的对象方法是Increase过程，用于将日期增加一天。这种计算实现起来并不简单，因为需要考虑不同月份的日期数不同，并判断是否是闰年。为了使编写代码更为容易，我们要把该类的内部实现改变为适用于内部实现的Delphi的TDate类型。之后，该类的定义将如下所示（完整的代码在DateProp例子中）：

```

type
  TDate = class
  private
    fDate: TDateTime;
  public
    procedure SetValue (y, m, d: Integer); overload;
    procedure SetValue (NewDate: TDateTime); overload;
    function LeapYear: Boolean;
    function GetText: string;
    procedure Increase;
  end;

```

注意，因为唯一的改变在类的专用部分中，所以我们不必改动那些使用它的现有程序的任何部分。这就是封装的优点。

说明：TDateTime类型实际上是一个浮点数。该浮点数的整数部分表示从12/30/1899开始的日期，OLE Automation与Microsoft Win32应用程序也使用相同的基准日期（使用负值表示该日期之前的年份）。小数部分以一个分数形式表示时间。例如，值3.75代表1900年1月2日的6:00PM（一天的四分之三）。为了加减日期，只需加或减天数，这要比用日/月/年表示法增加天数更容易实现。

用属性封装

属性是一种非常有效的OOP机制，或者说非常适合实现封装的思想。从本质上讲，属性就是用一个名称来完全隐藏它的实现细节。这使得程序员可以任意修改类，而不会影响使用它的代码。关于什么是属性，有一个较好的定义，亦即属性就是虚拟字段。从定义它们的类的用户角度来看，属性完全就像字段一样，因为用户可以读取或编写它们的值。例如，可以用下列代码读取一个按钮的Caption属性值，并将其赋给一个带有下列代码的编辑框的Text属性：

```
Edit1.Text := Button1.Caption;
```

这看起来就像读写字段一样。然而，属性可以直接与数据以及访问对象的方法对应起来，用于读写数值。当属性与对象方法对应时，它们访问的数据可以是对象或其外部的某部分内容，而且它们可以产生附加影响，如在改变了一个属性值后，重新绘制一个控件。从技术上讲，一个属性就是对应数据或对象方法（使用一个read或一个write子句）的标识符。例如，下面是一个日期类的Month属性的定义：

```
property Month: Integer read FMonth write SetMonth;
```

为了访问Month属性的值，该程序语句要读取专用字段FMonth的值；同时为了改变该属性值，它调用了对象方法SetMonth（当然，必须在类的内部定义它）。

不同的组合都是可能的（例如，还可以使用一个对象方法来读取属性值或直接修改write指令中的一个字段），但使用对象方法修改一个属性的值是很常见的。下面是属性的两种可替换定义，它们分别对应两个访问方法或在两个方向上直接对应着数据：

```

property Month: Integer read GetMonth write SetMonth;
property Month: Integer read FMonth write FMonth;

```

通常，属性是公共的，而实际数据与访问对象方法是专用的（或受保护的）。这意味着必须使用属性访问那些对象方法或数据，这种技术提供了封装的扩展与简化版本。它是一个扩展的封装，因为不但可以改变数据的表示方法与访问函数，而且还可以添加与删除访问函数，而不会影响到调用代码。一个用户只需要重新编译使用属性的程序即可。

提示：在定义属性时，可以利用Delphi编辑器扩展的类完成特性，该特性可以通过Ctrl+Shift+C组合键激活。在写完属性名称、类型和分号之后，按下Ctrl+Shift+C，Delphi将提供setter方法的一个完整的定义和基干。在read关键字后面的标识符名称之前写上Get，几乎不用任何输入就可以得到一个getter方法。

TDate类的属性

作为一个范例，我向前面讨论过的TDate类的一个对象中添加用于访问年、月、日的属性。这些属性没有与特定的字段对应起来，但它们都对应着存储完整日期信息的单个fDate字段。这就是所有属性都有getter和setter方法的原因：

```
type
  TDate = class
  public
    property Year: Integer read GetYear write SetYear;
    property Month: Integer read GetMonth write SetMonth;
    property Day: Integer read GetDay write SetDay;
```

这些方法通过使用在DateUtils单元中可获得的函数，可以很容易地实现（更多细节请参阅第3章“运行时库”）。这里是用于其中两种方法的代码（其他的都非常相似）：

```
function TDate.GetYear: Integer;
begin
  Result := YearOf (fDate);
end;

procedure TDate.SetYear(const Value: Integer);
begin
  fDate := RecodeYear (fDate, Value);
end;
```

该类的代码可以在DateProp范例中找到。程序使用了一个二级单元来定义TDate类，以便增强封装，并创建一个保存在窗体变量中的单个日期对象，同时为了整个程序的执行，将它保存在内存中。作为一种标准的方法，该对象在窗体的OnCreate事件处理器中建立，并在窗体的OnDestroy事件处理器中删除。这一程序的窗体（如图2.2所示）有三个编辑框与按钮，用于这些编辑框与日期对象属性之间的数值复制。

警告：在赋值时，程序使用SetValue对象方法而不是设置每个属性。事实上，分别对月日赋值，在月份对于当前日期无效的情况下，会引起错误。例如，当前日期是1月31日，却想赋给它2月20日时。如果首先赋月份，赋值的这一部分可能会无效，因为2月31日不存在。如果首先对日子赋值，当进行反方向赋值时也可能会出现问题。由于日期的有效性规则所限，最好还是一次性进行赋值较为妥当。

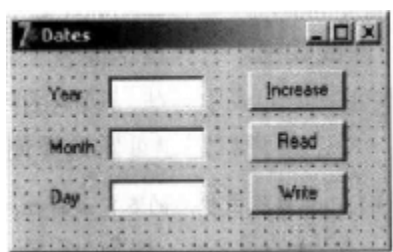


图2.2 DateProp示例的窗体

属性的高级特性

属性有一些高级特性，我们将在以后的章节中详细讨论，特别是在第4章中，将介绍Tpersistent类、RTTI和流；在第9章“编写Delphi组件”中讨论编写定制的Delphi组件。下面是对这些高级特性的一个简要介绍：

- 一个属性的write指令可以被省略，使之变成一个“只读”属性。如果试图修改这个属性值，编译器将会报告一个错误。还可以省略read指令并定义一个“只写”属性，但这种方法没有太大的意义，而且很少被使用。
- Delphi IDE对待“设计时”属性比较特殊，需用公布的访问标识符声明，并且通常显示在所选组件的对象检验器中。有关published关键字及其效果的更多讨论参见第4章。
- 另一种方法是使用公共的访问标识符来声明属性，那些通常被称做“只用于运行时”属性。这些属性可以在程序代码中使用。
- 可以定义基于数组的属性，它们使用带方括号的典型符号来访问一个列表中的元素。基于字符串列表的属性，如一个列表框的Lines，就是这类属性的一个典型范例。
- 属性有一些特殊指令，包括stored与default，它们控制着组件的流系统（第4章将介绍并在第9章中详细描述）。

说明：通常可以向一个属性赋予一个值或读取它，甚至在表达式中使用属性，但不能将属性作为参数传递给过程或对象方法。这是因为属性不是一个内存位置，所以它不能被用做var或out参数；也不能通过引用传递。

封装与窗体

封装的一个关键思想就是减少程序所使用的全局变量的数目。由于一个全局变量可从程序的任何部位进行访问，因此，一旦某个全局变量发生改变就会影响所有程序。另一方面，当改变类中字段的表示时，只需改变该类的一些对象方法的代码，而不用再做其他的事了。因此可以说信息隐含指的就是封装改变。

下面用一个范例来澄清该思想。编写一个具有多个窗体的程序时，为了使每个窗体都能使用部分数据，可以在其中一个窗体单元的接口部分将其声明为全局变量：

```
var
    Form1: TForm1;
    nClicks: Integer;
```

这样做虽可以运行，但是数据将与整个程序相连，而不是与窗体的一个特定实例相连。如果建立相同类型的两个窗体，它们就可以共享数据。如果想要相同类型的每个窗体都有自己的数据，则唯一的解决方法是将其添加到窗体类中：

```
type
  TForm1 = class(TForm)
  public
    nClicks: Integer;
  end;
```

向窗体添加属性

前面的类使用了公共数据，为了进行封装，应当修改它，以使用专有数据和数据访问函数。一个甚至更好的解决方案是向窗体添加一个属性。每当大家想使窗体的一些信息为其他窗体也可以使用，应该使用一个属性，上一节“用属性封装”中解释了所有的原因。只需改变窗体的字段声明（见前面的代码清单），在它前面添加关键字**property**，然后使用**Ctrl+Shift+C**组合键激活代码完成功能即可。Delphi将自动生成需要的所有额外代码。

该窗体类的完整代码见FormProp范例，如图2.3所示。该程序可以建立窗体的多个实例（也就是说，多个基于相同窗体类的对象），且每个都有自己的单击计数。

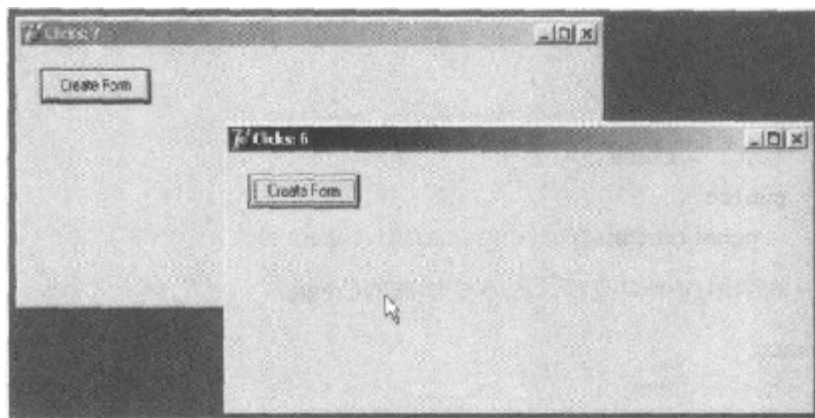


图2.3 FormProp示例在运行时的两个窗体

说明：注意，向一个窗体添加属性时，并不能将其添加到对象检验器的窗体属性列表中。

在笔者看来，属性还可在窗体类中用来封装对一个窗体组件的访问。例如，如果主窗体中有一个状态条用于显示一些信息（并将SimplePanel属性设置为Tree），并且此时我们想从一个二级窗体上更改显示文本，则可以这样编写代码：

```
Form1.StatusBar1.SimpleText := 'new text';
```

这是Delphi中的一个标准方法，但并不是一种好方法，因为它不能提供窗体结构或组件的任何封装。如果在一个应用程序的很多地方都有相似的代码，而且以后决定要更改窗体的用户界面（用其他控件替换StatusBar或激活多个面板），就必须修正很多地方的代码。还有一种方法是使用对象方法或最好使用一个属性，以隐藏具体的控件。这种属性通过读取或写入状态栏的SimpleText属性的GetText和SetText方法（或其一个面板的标题）可以被定义为：

```
property StatusText: string read GetText write SetText;
```

而在程序的其他窗体中，可以引用窗体的StatusText属性，如果用户界面改变，则只有属性的setter和getter方法会被影响。

说明：参见第4章可以了解如何避免公布组件的窗体字段，因为这将改善封装。但是现在不要着急：这需要有良好的Delphi知识，并且上面讨论的技术也有一些缺点。

构造器

到目前为止，为了给对象分配内存，我们调用了Create对象方法。这是一种构造器——一种可以用在类中、为该类的一个实例分配内存的特殊对象方法。该实例通过构造器返回，并可以分配给一个变量，用于存储对象以备后用。Create构造器会将新实例的所有数据初始化为零。如果我们想使用特殊的值启动实例数据，那么就需要编写一个定制的构造器来实现它。

在构造器前面应使用constructor关键字。尽管可以为一个构造器使用任何名称，但还是应该坚持使用标准名称，Create。如果使用Create之外的名称，TObject基类的Create构造器将仍可以使用，但调用该默认构造器的程序员可能会忽略后来所提供的初始化代码，因为他们不认识该名称。

通过使用一些参数定义Create构造器，可以用新定义代替默认定义并强制使用它。例如，做如下定义：

```
type
  TDate = class
  public
    constructor Create (y, m, d: Integer);
```

之后，将只能调用这个构造器，而不是标准的Create：

```
var
  ADay: TDate;
begin
  // Error, does not compile:
  ADay := TDate.Create;
  // OK:
  ADay := TDate.Create (1, 1, 2000);
```

为定制组件编写构造器的规则有所不同，读者将在第9章中看到这一点。其原因是在这种情况下，必须覆盖一个虚拟的构造器。过载特别地与构造器相关，因为用户可以将多个构造器添加到一个类中，并调用它们所有的Create；这种方法使构造器更易于被记住，并且会跟着一个由其他OOP语言提供的标准路径，其中所有构造器都有相同的名字。作为一个范例，我们向该类添加了两个独立的Create构造器：一个不带参数，这将会隐藏默认的构造器；另一个则带有初始值。不带参数的构造器使用当天的日期作为默认值（就像大家在DataView范例的完整代码中所看到的那样）：

```
type
  TDate = class
  public
    constructor Create; overload;
    constructor Create (y, m, d: Integer); overload;
```

析构器和Free方法

类可以有定制的构造器，同样还可以有定制的析构器——这是由`destmctor`关键字声明的一种方法，称为`Destroy`。就像一个构造器调用可以为一个对象分配内存一样，一个析构器调用将会释放内存。只有当对象在其构造器中需要外部资源或者在它们的生命周期内，才需要析构器。你可以为一个析构器编写定制代码，通常会覆盖默认的`Destroy`析构器，以便在对象被删除前，执行一些资源清除工作。

`Destroy`是`TObject`类的一个虚拟析构器。并不需要定义一个不同的析构器，因为对象通常是通过调用`Free`对象方法被解除的，而且该对象方法会调用特定类的`Destroy`虚拟析构器（虚拟的对象方法将在本章稍后进行讨论）。

`Free`是`TObject`类的一个对象方法，可以由所有其他类继承。`Free`对象方法会在调用`Destroy`虚拟析构器之前检查当前对象（`Self`）是否为`nil`。`Free`不会将对象自动地设置为`nil`；这个工作需要由你自己来做。对象并不知道哪一个变量可能正在引用它，所以无法将它们都设置为`nil`。

Delphi 5引进了一种`FreeAndNil`过程，你可以用来释放一个对象并设置其同时引用`nil`。调用`FreeAndNil (Obj1)`而不是编写下列代码：

```
Obj1.Free;
Obj1 := nil;
```

说明：本章稍后的“只能解除一次对象”中还会详细讨论该问题。

Delphi的对象引用模型

在一些OOP语言中，可通过声明一个类的变量建立该类的实例。而Delphi则不同，它以对象引用模型为基础。它的基本思想是，类的每个变量，如上面`ViewDate`范例中的`TheDay`变量，并不会保存对象的值，而是保存一个引用或一个指针，以说明对象被存储的内存位置。其结构如图2.4所示。

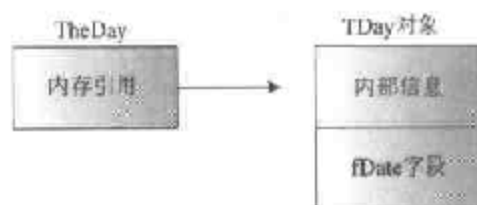


图2.4 内存中一个对象结构的表示，有一个变量指向它

使用这种方法惟一的问题是，当声明一个变量时，不会在内存中建立一个对象（这和所有其他变量是不一致的，可能会使Delphi的新用户感到迷惑），而只是保存一个对象引用

的内存地址。对象实例必须人工建立，至少对于用户自己定义的类的对象如此。用户放置在窗体上的组件实例可由Delphi库自动建立。

前面已经说明了如何通过向对象的类应用一个构造器来建立一个对象实例。一旦建立了一个对象并完成了使用，就需要解除它（以避免内存占用，这会引起称为内存泄漏的问题）。这可以通过调用Free对象方法来实现。只要在需要对象时建立它们，并在完成使用时释放它们，对象引用模式就会正常运转。对象引用模式在对象分配与内存管理上有很多用处，有关内容将在下面两节中介绍。

赋值对象

如果一个保存对象的变量只含有一个内存中对象的引用，复制该变量的值会发生什么情况呢？假设用下列方法编写ViewDate范例的BtnTodayClick对象方法：

```
procedure TDateForm.BtnTodayClick(Sender: TObject);  
var  
    NewDay: TDate;  
begin  
    NewDay := TDate.Create;  
    TheDay := NewDay;  
    LabelDate.Caption := TheDay.GetText;  
end;
```

也就是说，将NewDay对象的内存地址复制给TheDay变量（如图2.5所示），而没有将对象的数据复制给其他对象。在这种特殊情况下，这并不是一个很好的方法——因为每当按下按钮时，都会为一个新对象分配内存，但却不能释放TheDay变量以前指向的对象的内存。

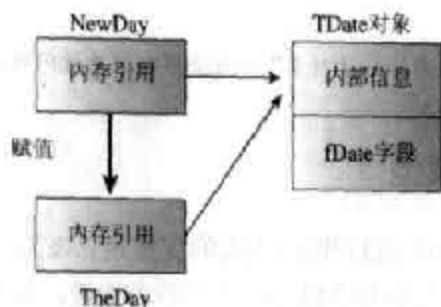


图2.5 将一个对象引用赋值给另一个对象。这不同于将一个对象的实际内容复制给另一个对象

这种特殊问题可以通过释放旧的对象来解决，如下面的代码（它也被简化了，没有对新建对象使用显式变量）所示：

```
procedure TDateForm.BtnTodayClick(Sender: TObject);  
begin  
    TheDay.Free;  
    TheDay := TDate.Create;
```

要记住重要的是，当将一个对象赋给另一个对象时，Delphi会将内存中对该对象的引用复制给新的对象引用。不应该否定这种现象；在很多情况下，能够定义引用已有对象的变量

可能是件好事。例如，可以通过访问一个属性来存储返回的对象，并在后续语句中使用它，如下列代码段所示的那样：

```
var
    ADay: TDate;
begin
    ADay := UserInformation.GetBirthDate;
    // use a ADay
```

如果将一个对象作为参数传递给函数，也会出现相同的情况：没有建立新对象，但在代码的两个不同位置引用了相同的对象。例如，通过编写该过程并如下调用它，来修改Button1对象的Caption属性，而不是内存中其数据复制的属性（没有什么用处）：

```
procedure CaptionPlus (Button: TButton);
begin
    Button.Caption := Button.Caption + '+';
end;

// call...
CaptionPlus (Button1)
```

这意味着对象被引用所传递，而不需要使用var关键字，并且不需要任何其他明显的pass-by-reference语义指示，这对初学者来说也是很容易困惑的。如果真想修改一个已有对象中的数据，使它与另一个对象的数据相匹配，又该如何呢？此时必须复制对象的每个字段，只有当它们都是公共字段时，才可能达到这一目的；或提供一个特殊的对象方法来复制内部数据。VCL的一些类拥有Assign对象方法，它可以执行这种复制操作。更准确地讲，大多数继承自TPersistent，而并非继承自TComponent的VCL类都拥有Assign对象方法。其他源自TComponent的类也拥有该对象方法，但在被调用时会引起异常。

在DateCopy范例中，我已经将一个Assign方法添加到Tdate类中，并使用下列代码从Today按钮中调用它：

```
procedure TDate.Assign (Source: TDate);
begin
    fDate := Source.fDate;
end;

procedure TDateForm.BtnTodayClick(Sender: TObject);
var
    NewDay: TDate;
begin
    NewDay := TDate.Create;
    TheDay.Assign(NewDay);
    LabelDate.Caption := TheDay.GetText;
    NewDay.Free;
end;
```

对象与内存

至少，在没有Access Violations和消耗不需要的内存的情况下，如果我们允许系统协调工作的话，Delphi中的内存管理须遵循三条规则：

- 每个对象在其可以使用之前必须被创建。
- 每个对象在其使用之后必须被解除。
- 每个对象必须只能被解除一次。

是在代码中必须执行这些操作，还是可以让Delphi来处理内存管理，都取决于我们在Delphi提供的不同方法中所选择的模块。

Delphi支持下列针对动态元素进行内存管理的三种类型：

- 每当在应用程序代码中明确地建立一个对象时，还应该释放它（一些系统对象和通过接口引用的对象例外）。如果不这样做，在程序终止之前，该对象使用的内存将不会被释放给其他对象。
- 当建立一个组件时，可以指定一个宿主组件，将宿主传递给组件构造器。宿主组件（通常是窗体）负责解除其拥有的全部对象。换句话说，当释放窗体时，也会释放其拥有的所有组件。所以，如果建立一个组件并为它分配一个宿主，就不必惦记着解除该组件了。这是在一个窗体或日期模块上设计并创建组件时应采用的标准方法。然而，选择一个可以保证将被解除的宿主确实是强制的：例如，窗体通常被全局的Application对象所拥有，当程序结束时它将被库所解除。
- 当Delphi的RTL为字符串和动态数组分配内存时，如果引用超出了范围，Delphi会自动释放内存。用户不需要释放字符串：当执行不到它时，其内存就会被释放。

只解除对象一次

如果两次调用一个对象的Free方法（或调用解除程序），就会出现错误。然而，如果记住将对象设置为nil，就能调用Free两次而不会出现任何问题。

说明：你可能想知道，如果对象引用是nil，为什么就可以安全地调用Free，而不能调用Destroy呢。原因是，Free是给定内存位置已知的对象方法，而虚拟函数Destroy要在运行时通过检查对象类型来确定，如果对象已不存在，这将是非常危险的操作。

总之，应该注意下面两条规则：

- 调用Free来解除对象，而不要调用Destroy析构器。
- 使用FreeAndNil，或在调用Free后将对象引用设置为nil，除非引用超出了范围。

通常，可以通过Assigned函数来检查一个对象是否为nil。下列两条语句在大多数情况下都是相同的：

```
if Assigned (ADate) then ...  
if ADate <> nil then ...
```

注意，这些语句只测试指针是否为nil，它们并不检查其是否为一个有效的指值针。如果编写下列代码，测试将会令人满意，并且随着对对象方法的调用，会在行中得到一个错误：

```
ToDestroy.Free;  
if ToDestroy <> nil then  
    ToDestroy.DoSomething;
```

重要的是认识到调用Free并不能将对象设置为nil。

继承已有类型

我们经常需要在现有类的基础上稍做修改。例如，向现有类中添加一个新对象方法或对其做细微改动。如果是复制并粘贴原始的类并且随后修改它（确实是一种可怕的编程行为，除非有特殊的原因这么做），那么所有代码、缺陷和头疼之事就会一并复制。相反，在这种情况下，可以使用OOP的一个关键特性：继承。

在Delphi中，为了继承一个现有类，只需要在新类声明的开始处指出该类。例如，当每次用户建立新窗体时，Delphi会自动产生以下代码：

```
type  
    TForm1 = class(TForm)  
    end;
```

这个简单的定义表示TForm1类继承了TForm类的所有对象方法、字段、属性和TForm类的事件。用户可以为TForm1类型的对象调用TForm类的所有公用对象方法。同时TForm也会从其他类继承一些对象方法，如TObject基类。

作为继承的一个简单例子，可以从TDate中派生出一个新类并改动它的GetText函数。为此，从NewDate范例的Dates单元中找到这个代码：

```
type  
    TNewDate = class(TDate)  
    public  
        function GetText: string;  
    end;
```

为了实现GetText函数的新版本，笔者使用了复杂的FormatDateTime函数，它使用了（在其他特性中）在Windows中可用的预定义的月份名；这些名称取决于用户的地区与语言设置（大部分这些地区设置都由Delphi复制为库里定义的常量，如LongMonthNames、ShortMonthNames和其他很多常量，可以在Delphi帮助文件中的Currency and date/time formatting variables主题中找到）。下面是GetText对象方法，其中“dddddd”代表长数据格式：

```
function TNewDate.GetText: string;  
begin  
    GetText := FormatDateTime ('dddddd', fDate);  
end;
```

提示：使用区域信息，程序NewDate会自动适应不同的Windows用户设置。如果在一台区域设置不是英语的计算机上运行该程序，它会自动使用所设置的语言来显示月份。要测试这种行为，只需改变区域设置就可以了。注意，地区设置的变化会立刻影响运行中的程序。

定义了新类后，就需要在NewDate范例的窗体代码中使用这种新的数据类型。只需定义TNewDate的TheDay对象，并在FormCreate对象方法中创建这种新类的一个对象，而无需改

变使用方法调用的代码，这是因为继承的方法仍然会以相同的方法运行；尽管正像新的输出显示的那样，其效果已经发生了改变（如图2.6所示）。

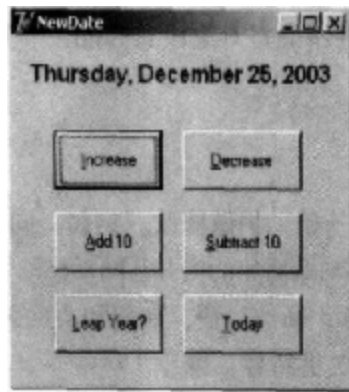


图2.6 NewDate程序的输出，带有月份名称和Windows区域设置的日期

保护字段与封装

TNewDate类的GetText对象方法的代码只在与TDate类处于相同单元中时才能被编译。事实上，它访问了父类的fDate专用字段。如果要将派生类放到新单元中，就必须将fDate元素声明为保护类型，或者在父类中添加一个简单的对象方法来读取专用元素的值。

很多开发人员会认为第一种方案总是最好的，因为将大部分元素声明为保护类型，将使得类更容易扩展，而且子类更容易编写。然而，这种方法却违背了封装的思想。在层次众多的类中，改变基类一些保护元素的定义会和改变一些全局数据结构一样困难。如果十个派生类访问该数据，改变它的定义就意味着可能需要改动十个类的代码。

换句话说，灵活性、扩展性与封装经常会成为冲突的目标。当这种情况发生时，应该倾向于封装。如果可以既考虑封装又不牺牲灵活性，那就更好了。通常这种折衷方案可以通过使用虚拟的对象方法来实现，本书将在“滞后绑定与多态性”一节中详细讨论该问题。如果为了更快地编写子类的代码而不选择使用封装的话，那么这种设计就背离面向对象的原则了。

访问其他类的受保护数据（或“盗用受保护的数据”）

读者已经在Delphi中了解到，一个类的私有和受保护数据都可以由与该类在同一个单元中的函数或方法访问。例如，考虑这个类（Protection范例的一部分）：

```
type
  TTest = class
  protected
    ProtectedData: Integer;
  end;
```

一旦将这个类放入到它自己的单元中，就不能再从其他单元直接访问它的受保护部分。相应地，如果编写下列代码：

```
var
  Obj: TTest;
begin
  Obj := TTest.Create;
  Obj.ProtectedData := 20; // won't compile
```

编译器将会发布一个错误消息，“未声明的标识符：受保护数据。”那么在这种情况下，就没有方法去访问一个定义在不同单元中的类的受保护数据吗？不，有一种方法。如果创建一个明显没有用的派生类，如下所示，我们来将会发生什么：

```
type
  TTestHack = class (TTest);
```

现在，如果针对一个类进行对象的直接计算，并通过它访问受保护数据，则可以编写代码如下：

```
var
  Obj: TTest;
begin
  Obj := TTest.Create;
  TTestHack (Obj).ProtectedData := 20; // compiles!
```

这个代码编译和工作起来都正常，就像大家通过运行Protection程序所看到的那样。这种方法是如何才能工作的呢？如果仔细思考，则不难发现TTestHack类会自动地继承TTest基类的受保护字段，而TTestHack类和试图访问继承字段数据的代码在相同的单元中，所以受保护字段是可以访问的。就像我们所期望的，如果将TTestHack类的声明移动到一个二级单元中，程序将不再编译。

虽然已经向读者解释了该方法的原理，但必须警告大家，该方法违背了类保护机制，可能会在程序中引起错误（如访问了不应该访问的数据），并且它也与优秀的OOP技术背道而驰。然而，有时使用该技术却是最好的解决方案，研究一下VCL源代码与很多Delphi组件的代码就会发现这一点。其中两个简单的例子是访问TControl类的Text属性与DBGrid控件的Row与Col位置属性。这两个想法可以用TextProp与DBGridCol范例分别解释（这些范例非常复杂，所以只建议具有良好Delphi编程背景的程序员在此阅读它们的文本——其他读者请稍后再阅读）。尽管第一个范例是一个使用类型转换的合理范例，但Row与Column的DBGrid范例实际上是相反的——该范例说明访问类编写者所不希望暴露的数据是有危险性的。一个DBGrid的行与列与其DrawGrid或StringGrid（基类）中的意义不同。首先，DBGrid不会像实际单元那样计算固定单元（它根据修饰来识别数据单元），所以在索引中的行与列必须根据当前在网格上的有效修饰（并且不工作的时候可以被修改）进行调整。第二，DBGrid是数据的一种虚拟显示。当用户在一个DBGrid中向上滚动时，数据可能会移到下面，但当前选中的行不会改变。

这种技术——只声明一个本地类型，使我们可以访问一个类保护的数据成员——通常被称为非法修改，并且无论何时，只要可能的情况下都应该避免使用它。问题的出现不是因为相同单元中访问一个类的保护数据，而是出于一个原因——访问不同类中已有对象的受保护数据时，必须声明一个类！这种技术的危险性在于要将对象从一个类向另一个类进行硬编码类型转换。

继承与类型的兼容性

Pascal是一种具有严格类型定义的语言。这意味着，例如，不能将一个整型数值赋给一个布尔变量，除非使用一个显式的类型强制转换。两个值兼容的规则是，只有它们的数据类型相同时，或（更精确地说）只有它们的数据类型作为单个类型引用时，才能视它们为兼容。为了简化用户的工作，Delphi做了一些预定义的类型分配兼容：通过升级或降级，大家可以将一个Extended分配给一个Double，反之亦然。

警告：如果在两个不同单元中重定义了相同的数据类型，则即使它们的名字是相同的，也不能兼容。对一个使用了两个不同单元的两个相等名类型的程序来说，对其进行编辑和调试是一件可怕的事。

该规则有一个重要的例外，那就是类的类型确实存在。如果用户声明一个类，如TAnimal，并派生出一个新类TDog，就可以把一个TDog类型的对象赋给TAnimal类型的变量。那是因为狗也是动物！作为一个通用规则，可以在任何时间使用子类的一个对象。然而，反方向的操作是不合法的：在一个子类的一个对象被期望时，不能使用一个父类的对象。为了简化这种解释，下面再次以代码形式来表示：

```
var
    MyAnimal: TAnimal;
    MyDog: TDog;
begin
    MyAnimal := MyDog; // This is OK
    MyDog := MyAnimal; // This is an error!!!
```

滞后绑定与多态性

Pascal函数和过程通常基于静态绑定或超前绑定。即由编译器或连接器来解析对象方法的调用，将请求转换为调用函数或过程所占用的特殊内存地址（也就是例程的地址）。面向对象的编程语言允许使用另一种绑定方式，叫做动态绑定或滞后绑定。在这种方式下，只有在运行时才能确定（根据用于进行调用的实例类型）所调用对象方法的实际地址。

这种技术被称做多态性（一个希腊单词，意思是多个窗体）。多态性的基本思想是，用户调用一个对象方法，并将其应用于一个变量，但Delphi具体调用哪个方法将依赖于变量相关的对象类型。而Delphi直到运行时才能确定与变量相关的对象的实际类，而这只是为了满足前面讨论的类型兼容法则。多态性的优势是能够编写简单的代码，对待完全不同的对象类型，但前提是它们是相同的并可取得正确的运行时行为。

例如，假设一个类和它的子类（如TAnimal和TDog）都定义了一个对象方法，并且该方法采用的是滞后绑定。现在用户可以将该方法应用于一个通用变量，如MyAnimal。在运行时，它既可以引用TAnimal类的对象，也可以引用TDog类的对象。根据当前对象的类，在运行时才决定实际调用的对象方法。

PolyAnimals范例演示了这项技术。TAnimal类和TDog类有一个新的对象方法Voice，用来以文本或声音形式输出所选动物发出的声音（调用定义在MMSystem单元中的PlaySound API函数）。该Voice方法使用关键字virtual和override，在TAnimal类中被定义为Virtual（虚

拟)类并在TDog类的定义中被覆盖:

```
type
  TAnimal = class
  public
    function Voice: string; virtual;

  TDog = class (TAnimal)
  public
    function Voice: string; override;
```

现在来看看MyAnimal.Voice的调用效果。如果变量MyAnimal当前引用了TAnimal类的一个对象,则它会调用MyAnimal.Voice对象方法。如果它引用TDog类的对象,它将调用TDog.Voice方法。这只因为函数是虚拟的(读者可通过删除和重编辑该关键字来试验)。

TAnimal类任何子类的对象都可调用MyAnimal.Voice,甚至是在其他单元中定义的类——或者是还没有编写的类!编译器不需要知道所有子类,以使调用与之相兼容,此时,只有父类是需要的。换句话说,对MyAnimal.Voice的调用与将来所有的TAnimal子类兼容。

说明:这正是面向对象的编程语言喜爱复用性的关键性技术原因。读者可以编写一些使用不同层次类的代码,而不需要知道这些不同层次类的任何实际信息。换句话说,层次(和程序)仍然是可以扩展的,即使已经完成了上千行的代码也仍可以使用它。当然,需要满足一个条件:层次中的父类需要非常小心地设计。

在图2.7中,读者可以看到PolyAnimals程序输出的一个范例。通过运行它,将会听到由PlaySound API调用所产生的相应的声音。

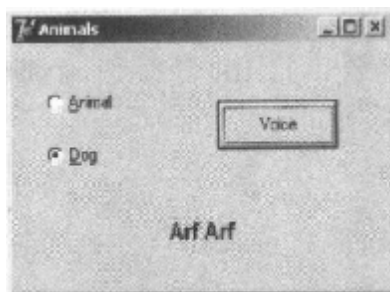


图2.7 PolyAnimals例子的输出

覆盖、重定义方法

像前面看到的那样,在子类中覆盖一个滞后绑定的对象方法,需要使用关键字override。然而,值得注意的是,只有在父类中将对象方法定义为虚拟类型(或动态)后,才能进行覆盖。否则,如果是一个静态类型的对象方法,就没有办法激活滞后绑定,而只有改变其父类的代码。

规则非常简单:一个定义为静态的对象方法会在每个子类中保持静态,除非用一个同名的新虚拟方法屏蔽它。一个被定义为虚拟的方法在每个子类中都将保持滞后绑定(除非使用一个静态方法隐藏它)。这是无法改变的,因为编译器会为滞后绑定方法创建不同的代码。

要重定义一个静态对象方法,用户只需向子类添加该对象方法,它的参数可以与原来方法的参数相同或不同,也没有进一步的规范。覆盖一个虚拟方法,必须指定相同的参数并

使用关键字override:

```
type
  TMyClass = class
    procedure One; virtual;
    procedure Two; (static method)
  end;

  TMyDerivedClass = class (TMyClass)
    procedure One; override;
    procedure Two;
  end;
```

典型地, 可以使用两种手法覆盖一个方法: 用一个新版本取代父类的方法, 或将更多代码添加到现有的方法中。这可以通过使用inherited关键字调用父类相同的方法来完成。例如:

```
procedure TMyDerivedClass.One;
begin
  // new code
  ...
  // call inherited procedure TMyClass.One
  inherited One;
end;
```

当覆盖基类的一个已有虚拟对象方法时, 必须使用相同的参数。在派生类中引入一个新版本的对象方法时, 可以用想要的参数声明它。事实上, 这将是一个与同名父对象方法无关的新对象方法, 它们只是名称相同而已。下面是一个范例:

```
type
  TMyClass = class
    procedure One;
  end;

  TMyDerivedClass = class (TMyClass)
    procedure One (S: string);
  end;
```

说明: 使用上面的类定义创建TMyDerivedClass类的一个对象时, 可以使用带有字符串参数的One对象方法, 而不是在基类中缺少参数的那个版本。如果需要, 可以用overload关键字标记重新声明的对象方法 (在派生类中)。如果对象方法与基类中的版本具有不同的参数, 它就实际变成了一个被重载的对象方法; 否则它会代替基类对象方法。注意对象方法不需要在基类中被标记为overload。然而, 如果基类中的对象方法是虚拟的, 则编译器会发出警告 “Method 'One' hides virtual method of base type 'TMyClass.'” 为了避免该消息, 更准确地按照自己的意图指示编译器, 可以使用reintroduce指令。如果读者对该专题感兴趣, 可以在Reintro范例中找到它的代码, 并且进一步研究。

虚拟方法与动态方法

在Delphi中,启动滞后绑定有两种基本办法:声明它为虚拟方法(**virtual**),或声明它为动态方法(**dynamic**)。这两个关键字的语法相同,而且其结果也相同。差别在于,编译器用来实现滞后绑定的内部机制不同。

虚拟对象方法基于一个虚拟对象方法表(**VMT**,也叫**vtable**)。虚拟方法表是一个对象方法的地址数组。为调用虚拟方法,编译器将产生代码,用以跳到相应的地址中,该地址储存在虚拟对象方法表内的第*n*个格中。**VMT**允许方法调用的快速执行,但是它们需要每个子类的各个虚拟方法都有一个入口,即使该方法没有在被继承类中覆盖。

另一方面,在调用动态对象方法时,使用惟一数值来表示对象方法。寻找相应的函数与查找虚拟方法表相比要慢。但其长处是,当子类覆盖对象方法时,动态对象方法的条目只在子类中传播。

消息处理

滞后绑定的对象方法也可用于处理一条Windows消息,尽管技术有些不同。为此,Delphi还提供了另一个**message**指令,用以定义消息处理对象方法。该方法必须是带有**var**参数的过程。**message**指令后要加上对象方法想要处理的Windows消息编号。

警告: Delphi还提供了专为Kylix设计的**message**指令,且Delphi语言和RTL完全支持该指令,但是CLX应用程序主框架的可视区却不使用消息方法向控件发送消息。因此,只要可能,都最好使用库中提供的虚拟方法,而不要直接处理Windows消息。当然,只有当程序员想要使代码更加灵活时,才需要考虑这个问题。

例如,通过下列代码,使用由**wm_User**这一Windows常量指出的数字值来处理一个用户定义的消息:

```
type
  TForm1 = class(TForm)
    ...
    procedure WMUser (var Msg: TMessage);
    message wm_User;
  end;
```

尽管不同的Windows消息有大量事先定义的记录类型,但还是应该由用户来决定过程名和参数的实际类型。随后用如下语句调用相应的方法,以产生消息:

```
PostMessage (Form1.Handle, wm_User, 0, 0);
```

对于那些了解Windows消息和API函数的Windows熟练程序员,这项技术也是绝对有用的。还可以通过调用**SendMessage** API或**VCL Perform**方法立刻发送一条消息。

抽象方法

关键字**abstract**用于声明只在当前类的子类中定义的对象方法。**abstract**指令可完整地定义对象方法,它不是正向声明语句。如果试图定义此对象方法,编译器将会发出警告。在Delphi中,可以建立一个含有抽象(**abstract**)对象方法的类实例(对象)。但在Delphi的32

位编译器中这样做时，编译器将发出一个警告消息：“Constructing instance Of<class name>containing abstractmethods”。如果偶然在运行时调用了抽象的对象方法，Delphi会发出异常，如下面的AbstractAnimals范例所示（PolyAnimals范例的一个扩展）：

```
type
  TAnimal = class
  public
    function Voice: string; virtual; abstract;
```

说明：大多数其他OOP语言使用了一个更严格的方法：通常不能创建包含有抽象方法的类实例。

读者也许会问，为什么要使用抽象方法？原因很简单，这是多态性的需要。如果类TAnimal含有抽象方法Voice，那么每个子类都能重定义它。如果有抽象方法Voice，则每个继承类都必须重新定义它。

在Delphi的早期版本中，如果一个方法覆盖了称为inherited的抽象方法，则会导致抽象的对象方法调用。从Delphi 6开始，编译器的功能被增强，它会注意到抽象对象方法的出现，并忽略inherited调用。这意味着可以在每个覆盖的对象方法中安全地使用继承，除非专门想禁止基类一些代码的执行。

类型安全的转换

Delphi派生类的类型兼容原则允许程序员在希望使用父类的位置使用派生类。前面曾提到，反过来是不可能的。现在假设TDog类有一个Eat对象方法，它没有出现在TAnimal类中。如果变量MyAnimal引用的是狗，它应该可以调用函数。但如果这样做，而且变量引用的是另一个类，结果则会出现错误。进行显式的类型转换时，还可能会引起一个危险的运行时错误（或更糟糕，一个轻微的内存重写问题），因为编译器不能确定对象的类型是否正确，以及调用的对象方法是否实际存在。

为了解决上述问题，可以使用基于运行时类型信息（RTTI）的技术。从本质上讲，因为每个对象都“了解”自己的类型与父类，所以我们可以用is操作符或是TObject类的InheritsFrom对象方法来查询该信息。is操作符的参数是一个对象与一个类的类型，返回值是个Boolean值：

```
if MyAnimal is TDog then ...
```

只有在MyAnimal对象当前正引用TDog类的一个对象或派生自TDog的类型时，is表达式才会返回True。这意味着，如果检查一个TDog对象是否属于TAnimal类型，测试将会成功。换句话说，如果可以安全地将对象（MyAnimal）赋给数据类型的变量，该表达式会返回True。

既然我们已经知道动物是一只狗，则可以进行一个安全的类型专换。例如，通过编写下列代码来完成这种直接的转换：

```
var
  MyDog: TDog;
begin
  if MyAnimal is TDog then
  begin
```

```
MyDog := TDog (MyAnimal);  
Text := MyDog.Eat;  
end;
```

同样的操作也可以直接通过第二个RTTI操作符as来实现, 如果需要的类与实际类兼容, 它可以只转换对象。as操作符的参数是一个对象和一个类的类型, 结果是被转换为新类类型的对象。可以编写下列代码:

```
MyDog := MyAnimal as TDog;  
Text := MyDog.Eat;
```

如果只想调用Eat函数, 还可以使用一个甚至更简短的语句:

```
(MyAnimal as TDog).Eat;
```

该表达式的结果是得到一个TDog类数据类型的对象, 所以可以向它应用该类的任何对象方法。传统转换与使用as转换之间的区别是, 如果对象的类型与试图转换的类型不兼容, 则第二种方法会引起异常——EInvalidCast (异常将在本章最后描述)。

为了避免这种异常, 可使用is操作符, 如果成功, 则进行一个普通的类型转换 (事实上, 无法依次使用is与as来进行两次类型检测):

```
if MyAnimal is TDog then  
    TDog(MyAnimal).Eat;
```

这两种RTTI操作符在Delphi中都非常有用, 因为我们经常需要编写通用代码, 用于相同类型的多个组件, 甚至是不同类型的组件。当一个组件作为参数传递给一个事件响应对象方法时, 要使用通用的数据类型 (TObject), 所以通常需要将它转换回原始的组件类型:

```
procedure TForm1.Button1Click(Sender: TObject);  
begin  
    if Sender is TButton then  
        ...  
end;
```

这在Delphi中是常用的技术, 而且会在本书的范例中使用它。两个RTTI操作符, is与as, 功能很强, 而且可以将它们看做是标准的编程构造。尽管它们的功能确实很强, 但也应该限制它们在特殊情况中的使用。当需要解决涉及多个类的复杂问题时, 首先可以尝试使用多态性。只有在特殊情况中, 多态性单独不能解决问题时, 才应该使用RTTI操作符来补充它。不要使用RTTI代替多态性。这是不好的编程习惯, 会导致程序速度缓慢。事实上, RTTI对性能会产生负面影响, 因为它必须遍历类的层次结构, 检查类型转换是否正确。而虚拟的对象方法调用只需要检查内存, 它的速度更快些。

说明: 实际上运行时信息远不止is与as操作符所提供的信息。我们可以在运行时访问详细的类与类型信息, 特别是关于公布的属性、事件与对象方法的信息。更多的相关主题参见第4章。

使用接口

当定义一个抽象类来表示带层次结构的基类时, 会发现抽象类是如此抽象, 以至于它只列出了一系列虚拟函数, 而没有提供任何实际的实现代码。这种纯抽象类还可以使用一种

特殊的技术——接口——来定义。因此，也将这些类称为接口。

从技术上讲，一个接口并不是一个类，尽管它类似于类。接口不是类，因为它们被看做是完全独立的元素，具有与众不同的特性：

- 接口类型对象是引用计数的，当对象不再有引用时会被自动销毁。该机制与Delphi管理长字符串的方法相似，使得内存管理几乎完全自动化。
- 一个类可以继承自一个基类，但它还可以实现多个接口。
- 因为所有类都派生自TObject，所有接口都派生自IInterface，故形成一个总的独立层次。

说明：接口基类在Delphi 5之前一直是IUnknown，但Delphi 6为它引入了一个新名称，IInterface，这更清楚地表明该语言特性独立于Microsoft的COM（使用IUnknown做为其基接口）。事实上，Delphi接口还可以在产品Kylix中使用。

重要的是，接口支持的OOP模型与类稍微有些不同。接口对多态性的实现提供了更少的限制。对象引用多态性是基于层次的一个特定分支。接口多态性可以在整个层次中工作。确实，接口更愿意使用封装，与继承相比，接口会在类之间提供一个更松散的连接。注意，最近的OOP语言，从Java到C++，都有接口的概念。

这里是一个接口声明的语法（按照惯例，用字母I开头）：

```
type
  ICanFly = interface
    ['{EAD9C4B4-E1C5-4CF4-9FA0-3B812C880A21}']
    function Fly: string;
  end;
```

这个接口含有一个GUID，这是一个数字ID，后面跟随着它的声明并基于Windows约定。可以在Delphi编辑器中使用Ctrl+Shin+G组合键生成这些标识符。

说明：尽管可以在不指定GUID的情况下编译并使用接口，但通常还是应该为它们指定GUID，因为不这样的话，就需要使用该接口类型执行QueryInterface或动态的as类型转换。因为接口就是为了在运行时利用扩展类型的灵活性，如果与类的类型比较的话，则没有GUID的接口就没有太大的用处。

在声明了一个接口之后，就可以定义一个类来实现它，如：

```
type
  TAirplane = class (TInterfacedObject, ICanFly)
    function Fly: string;
  end;
```

RTL已经提供了一些基类，以实现IInterface接口需要的基本性能。对于内部对象来说，笔者在这个代码中使用的是TInterfacedObject类。

可以用静态对象方法（如上面的代码）或虚拟对象方法来实现接口对象方法，通过使用override指令覆盖子类中的虚拟对象方法。如果不使用虚拟的对象方法，仍可以在子类中，通过重新声明接口类型，重新将接口绑定到新版本的静态对象方法中，以提供新的实现。使用虚拟的对象方法实现接口看起来更便于在子类中编写流畅的代码，但两种方法在功能与灵活性上是对等的。然而，使用虚拟对象方法会影响代码的大小与内存。

说明：编译器必须生成占位子例程来安排接口调用入口，指向实现类的匹配对象方法，并调整self指针。占位静态对象方法的接口对象方法非常简单：调整self并跳到类中的真正对象方法上即可。占位虚拟对象方法的接口对象方法稍微复杂些，与静态情况相比，在每个占位过程中需要四倍的代码（20~30字节）。另外，向实现类和其子类中添加更多虚拟对象方法会使虚拟对象方法表（VMT）增大。接口已经拥有了自己的VMT，并在派生中重新声明接口，将接口重新绑定到派生的新对象方法中，这与使用虚拟对象方法实现多态性一样，但所需代码量却要少得多。

既然已经定义了接口的一个实现，就可以编写一些代码，通过一个接口类型变量来使用这个类的一个对象：

```
var
  Flyer1: ICanFly;
begin
  Flyer1 := TAirplane.Create;
  Flyer1.Fly;
end;
```

只要向接口类型变量分配了一个对象，Delphi就会自动地检查是否该对象使用as操作符实现这个接口。可以像下面那样显式地表达这个操作符：

```
Flyer1 := TAirplane.Create as ICanFly;
```

说明：当通过接口或类使用as操作符时，编译器会为它生成不同的代码。通过类，编译器会引入“运行检测”，以证实对象与给定对象的类型是有效兼容的。通过接口，编译器在编译时如果发现可以从可使用的类类型中得到需要的接口，那么将完成相应的操作。该操作就像一个“编译时as”，而不是运行时的操作。

无论使用直接赋值还是as语句，Delphi都将调用对象的_AddRef方法（由TInterface定义）。这种方法的标准实现就像TInterfaceObject提供的那个一样，用于增加对象的引用次数。同时，只要Flyer1变量超出范围时，Delphi就会调用_Release对象方法来减小引用次数，并检测引用次数是否为零，而且如果必要的话就会清除对象。因此，在上面的代码中，并没有释放所创建对象的代码。

换句话说，在Delphi中，接口变量引用的对象是作引用计数的，并且当不再有接口变量引用它们时，它们便自动被释放。

警告：当使用基于接口的对象时，通常只应该使用对象变量或接口变量访问它们。混合两种方法会违反Delphi提供的引用计数规则，并会导致很难追踪的内存错误。实际上，如果决定使用接口，则应该专门使用基于接口的变量。如果想要混合它们，则需要通过编写自己的基类而不是使用TInterfacedObject来禁用引用计数。

异常处理

本章将涉及Delphi的另一个关键特性，即对异常的支持。异常处理的思想是为了使程序更稳定，用一种标准的方法通知并处理错误和不期望的条件。这样，异常将会使程序更易于编写、读取和调试，因为异常还可以允许用户把错误处理代码从正常代码中分离出来，而避免它们混杂在一起。加强代码和错误处理之间的分离并将错误处理器分隔出来可以自动地使

实际的逻辑更干净。异常能通过减少与实际编程目的无关的杂务，使用户编写更精练的代码。

在运行时遇到错误的情况下，Delphi库就会引发异常（在运行时代码中、在组件中或者在操作系统中）。从在代码中出现的地点，异常会被传递给它的调用代码。最终，如果用户代码中没有哪一部分能够处理该异常，VCL就会显示一个标准的错误消息，并通过处理下一个系统信息或用户请求来试着继续运行程序。

整个异常处理机制基于四个关键字：

try 定义代码保护块的起始。

except 定义代码保护块的结尾，并引入异常处理语句。

finally 用于指定必须执行的代码块，即使当异常出现时。这个块通常被用来执行总是需要的清除操作，如关闭文件或数据库表格、释放对象并释放在相同程序块中所要求的内存和其他资源。

raise 用于触发一个异常。在Delphi编程中，用户遇到的大部分异常都是由系统产生的，但当运行时发现有无效或非一致数据时，也能在自己的代码中引起异常。关键字**raise**还能用于在一个处理程序中重新触发异常，也就是说，把它向下一个处理程序发送。

提示：在程序中，没有合适的控制流可替代异常处理，只能继续使用if语句测试用户输入和其他可预见的错误条件。应当只对不正常的或非期望的情况使用异常。

程序流程与finally段

Delphi提供的异常处理功能可以将异常从子例程或对象方法传递给调用函数，其路径是一直向上，直到全局处理程序（如果程序提供的话，如Delphi应用程序通常所做的那样），而不是按程序的标准异常路径行进。所以真正的问题不是如何停止异常，而是如何在出现异常时执行一些代码。

下列代码用于执行一些耗时的操作，并且使用沙漏鼠标来显示该用户正在做一些事情：

```
Screen.Cursor := crHourglass;  
// long algorithm...  
Screen.Cursor := crDefault;
```

一旦在算法中出现一个错误（就像我故意在TryFinally范例的事件处理器中所做的那样），程序将会中断，但并不会重置默认光标。这就说明了一个try/finally块是做什么用的：

```
Screen.Cursor := crHourglass;  
try  
    // long algorithm...  
finally  
    Screen.Cursor := crDefault;  
end;
```

当程序执行这个函数时，它总会重置光标，而不管是否有（任何种类的）异常发生。

这个代码并不能处理异常；它仅仅是让程序在异常出现时更加健壮。一个try块后可以跟随一条except或一条finally语句，但这两条语句不能同时出现；所以，如果想要处理异常，则典型的解决方案是使用两个try块。将内部块与一个finally语句关联起来，将外部块与一个

except语句关联起来，如果情况需要，反之亦然。这里是在TryFinally范例中第三个按钮的代码摘录：

```
Screen.Cursor := crHourglass;
try try
    // long algorithm...
finally
    Screen.Cursor := crDefault;
end;
except
    on E: EDivByZero do ...
end;
```

通常在对象方法的最后，都有一些终结性代码。每次都应该使用**finally**语句来保护程序块，以免一旦引起异常而造成资源和内存浪费。

提示：处理异常通常没有使用**finally**段重要，因为Delphi可以处理大多数异常。而且如果代码中出现太多的异常处理段可能意味着程序流程中有错误，也可能是对语言中异常作用的误解。在本书余下部分的范例中，我们将看到很多try/finally块和一些raise语句，而几乎不会看try/except块。

异常类

上面的异常处理语句捕获了由Delphi的RTL定义的“EDivByZero”异常。其他这样的异常与“运行时间问题”（如错误的动态转换）、Windows资源问题（如内存溢出错误）或组件错误（如错误的索引）有关。程序员也可以定义自己的异常，只需要建立默认异常类或其子类的一个新子类即可：

```
type
    EArrayFull = class (Exception);
```

当我们向一个数组添加新元素而且数组已满（可能是由于程序中的一个逻辑错误造成的）时，就可以通过建立该类的一个对象来触发相应的异常：

```
if MyArray.Full then
    raise EArrayFull.Create ('Array full');
```

这个Create构造器（继承自Exception类）有一个字符串参数来为用户说明产生的异常。因为它会被异常处理程序自动删除，所以用户不必担心会破坏已建立的对象。

前面出现的代码属于一个范例程序，它叫做Exception1。一些子例程实际上已被稍加修改，如下面的DivideTwicePlusOne函数：

```
function DivideTwicePlusOne (A, B: Integer): Integer;
begin
    try
        // error if B equals 0
        Result := A div B;
        // do something else... skip if exception is raised
        Result := Result div B;
```

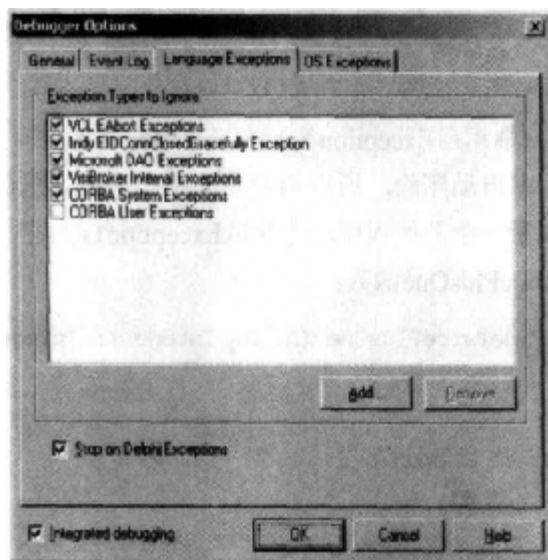
```
Result := Result + 1;  
except  
on EDivByZero do  
begin  
Result := 0;  
MessageDlg ('Divide by zero corrected.', mtError, [mbOK], 0);  
end;  
on E: Exception do  
begin  
Result := 0;  
MessageDlg (E.Message, mtError, [mbOK], 0);  
end;  
end; // end except  
end;
```

调试和异常

当从Delphi环境中启动一个程序（例如，按F9键）时，通常是在调试器中运行它。当遇到异常时，调试器在默认情况下会终止程序。当然，这通常是程序员希望的，因为这样将了解异常发生的位置，并可以逐步地了解处理程序的调用。还可以使用Delphi的Stack Trace特性，了解函数与对象方法调用的顺序，找出引起程序出现异常的原因。

但在本节的Exception1测试程序中，这种行为会使程序员不知道Delphi的调试器是怎样工作的。事实上，即使代码可以适当地处理异常，调试器还会在距异常出现处最近的源代码行停止程序运行，然后一步步地执行代码，而用户可以看到它是怎样处理异常的。

如果用户在异常被适当处理时想让程序继续运行而不停下来，可以从Windows Explorer上运行程序，或在Debugger Options对话框（使用Tools>Debugger Options命令）Language Exceptions页中暂停Stop on Delphi Exceptions命令，显示在Debugger Options对话框中的Language Exceptions异常页如下图所示（也可以用其来禁用调试器）。



在Exception1这段代码中，同一个try块后有两个不同的异常处理程序。用户可以拥有任意多个这样的处理程序，它们会按顺序进行检测。

实际上，使用异常的层次结构，处理程序也会被它引用的类型的子类所调用，这和其他过程所做的一样。因此需要将这个处理器（Exception父类处理器）放置到最后。但要清楚，对每个异常都使用处理程序通常来说并不是一个好的选择。最好的办法是，把不知道的异常都留给Delphi处理。VCL中默认的异常处理程序会在一个消息框中显示异常的错误消息，然后重新开始程序的正常操作。程序员实际可以使用Application.OnException事件改动常规的异常处理程序，就像本章稍后ErrorLog范例中解释的那样。

上述代码中另一个重要的元素是，在处理程序中使用异常对象（见on E: Exception do）。类Exception的引用E引用了由raise语句传递的异常对象值。当用户使用异常处理功能时，要记住这一规则：通过建立一个对象引起一个异常，并通过说明其类型来处理它。这样做有一个明显的好处，当用户处理一种类型的异常时，实际上是在处理指定类型以及每个派生类型的异常。

记录错误

大多数情况下，程序员不知道哪一步操作会引起异常，而且又不能（也不应该）将所有的代码都装在try/except块中。通常的方法是通过全局Application对象的OnException事件，让Delphi处理所有异常，并将它们传递给用户。使用ApplicationEvents组件可以很容易地予以实现。

在ErrorLog范例中，笔者向主窗体添加了一个ApplicationEvents组件的实例，并为其OnException事件添加了一个处理程序：

```
procedure TFormLog.LogException(Sender: TObject; E: Exception);
var
  Filename: string;
  LogFile: TextFile;
begin
  // prepares log file
  Filename := ChangeFileExt (Application.Exename, '.log');
  AssignFile (LogFile, Filename);
  if FileExists (FileName) then
    Append (LogFile) // open existing file
  else
    Rewrite (LogFile); // create a new one
  try
    // write to the file and show error
    Writeln (LogFile, DateTimeToStr (Now) + ':' + E.Message);
    if not CheckBoxSilent.Checked then
      Application.ShowException (E);
  finally
    // close the file
```

```

    CloseFile (LogFile);
end;
end;

```

说明: ErrorLog范例使用了由传统的Turbo Pascal TextFile数据类型提供的简单文本文件支持。可以将一个文本文件变量赋给一个实际文件, 然后读写它。在电子版“Essential Pascal”一书的第12章中, 读者会看到更多的关于TextFile操作的信息, 详见附录C。

在全局异常处理程序中, 可以写入记录, 例如, 事件的日期与时间, 还可以决定是否像Delphi通常那样显示异常信息(执行TApplication类的ShowException对象方法)。默认情况下, 只有当未安装OnException处理程序时, Delphi才会默认执行ShowException。在图2.8中, 大家可以看到运行的ErrorLog程序和ConTEXT中打开的一个异常记录示范(一位优秀程序员使用Delphi创建的编辑器, 可以从www.fixedsys.com/context中获得)。

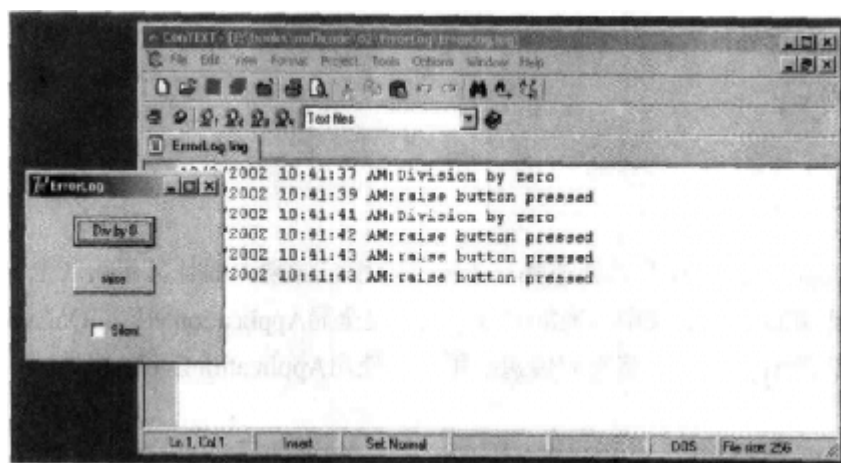


图2.8 ErrorLog示例和它产生的日志

类引用

本章最后讨论的语言特征是类引用, 它暗示代码是对类本身(不是类实例)进行处理。第一点需要注意的是, 类的引用不是一个对象, 它只是对类的类型的引用。一个类的引用类型确定了一个类引用变量的类型。听起来很含混吧? 不过几行代码将会使这个概念更清楚些。

假设用户定义了一个类TMyClass。现在可以定义一个新类的引用类型并与此类相关:

```

type
    TMyClassRef = class of TMyClass;

```

接下来可以声明这两种类型的变量了。第一个变量引用一个对象, 而第二个变量则引用一个类:

```

var
    AnObject: TMyClass;
    AClassRef: TMyClassRef;
begin
    AnObject := TMyClass.Create;
    AClassRef := TMyClass;

```

读者可能想知道类引用的作用是什么,一般说来,类的引用允许用户在运行时处理数据类型。用户可以在数据类型使用合法的任何表达式中使用类的引用。实际上,这样的表达式并不多,但即使这样,这些情况也非常有意义,最简单的一种情况是创建对象。可以重写上面两行代码如下:

```
AnObject := AClassRef.Create;
```

这次用类的引用来替代实际的类,对其应用Create构造程序,以使用类的引用建立该类的对象。

如果类的引用类型不支持与应用于类类型相同的类型匹配规则,那么它的用处也不会太大。当用户声明类引用变量时,如前面的MyClassRef,就可以赋予它具体的类以及任何子类。所以,如果TMyNewClass是TMyClass的子类,用户也可以这样写:

```
AClassRef := TMyNewClass;
```

Delphi在“运行时库”和VCL中声明了许多类的引用,包括:

```
TClass = class of TObject;  
TComponentClass = class of TComponent;  
TFormClass = class of TForm;
```

特别地,TClass类的引用类型可用来存储用户在Delphi中所编写的对任何类的引用,因为每个类最终都派生自TObject。而TFormClass引用被用在大多数Delphi项目的源代码中。事实上,Application对象的CreateForm对象方法需要把要建立的窗体类作为参数:

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

第一个参数是类引用,第二个参数是变量,用于存储对所建对象实例的引用。

最后,当拥有一个类引用时,可以将相关类的对象方法应用于它。鉴于每个类均继承自TObject,故可以将TObject的一些对象方法应用于每个类引用,读者将在第3章中看到。

使用类引用创建组件

在Delphi中,类引用的实际用处是在运行时处理数据类型,它是Delphi环境提供的一项基本功能。当用户从ComponentPalette上选择一个新组件并将其添加至窗体时,实际上就选择了一个数据类型,并建立了该数据类型的一个对象(这其实是Delphi在幕后为用户所做的工作)。换句话说,类引用为用户提供了对象结构的多态性。

为了使读者对引用类的工作原理有一个更好的认识,这里提供了一个简单的例子,名为ClassRef。该例的窗体非常简单,有三个单选按钮,置于窗体上部的面板中。选择其中之一并单击窗体,将建立由按钮标签指明的三种类型的新组件:单选按钮、可单击按钮、编辑框。

为使该程序正常地运行,读者需要改变三个组件的名称。窗体还必须有一个类的引用字段,声明为ClassRef:TControlClass。它保存了每次读者单击三个单选按钮时一个新的数据类型,如ClassRef:=Tedit。当读者单击窗体时,代码的有关部分将会执行。并且,该代码已选择了窗体的OnMouseDown事件来跟踪鼠标位置上的单击:

```
procedure TForm1.FormMouseDown(Sender: TObject; Button: TMouseButton;  
    Shift: TShiftState; X, Y: Integer);
```



```
var
    NewCtrl: TControl;
    MyName: String;
begin
    // create the control
    NewCtrl := ClassRef.Create (Self);
    // hide it temporarily, to avoid flickering
    NewCtrl.Visible := False;
    // set parent and position
    NewCtrl.Parent := Self;
    NewCtrl.Left := X;
    NewCtrl.Top := Y;
    // compute the unique name (and caption)
    Inc (Counter);
    MyName := ClassRef.ClassName + IntToStr (Counter);
    Delete (MyName, 1, 1);
    NewCtrl.Name := MyName;
    // now show it
    NewCtrl.Visible := True;
end;
```

该对象方法的第一行代码非常关键，它建立了一个存储在ClassRef字段中的类数据类型的对象，对类的引用应用Create构造程序即可完成这一工作。现在，可以设置Parent属性的值和新组件的位置，为新组件命名（也会自动地被Caption或Text使用），以及使其可视化。可以在图2.9中看到这个程序的输出范例。

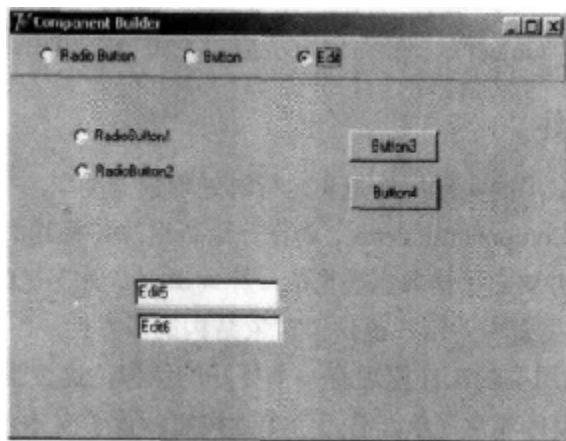


图2.9 ClassRef输出的一个例子

说明：为了实现多态构造，类引用的基类类型必须有一个虚拟的构造器。如果使用虚拟构造器（如范例中那样），应用于类引用的构造器调用将调用类引用变量当前引用的那个类型的构造器。但是如果没有虚拟构造器，代码将调用类引用声明中指示的固定类类型的构造器。虚拟构造器对多态构造的作用就像虚拟对象方法对多态性的作用一样。

小结

在这一章中，我们讨论了Delphi中面向对象编程（OOP）的基础。我们也讨论了类的定义和方法的使用、封装和内存管理，还有一些更高级的概念，如属性和组件的动态创建。接着又介绍了继承、虚拟和抽象方法、多态性、安全转换、接口、异常和类引用。

如果你是一个初学者，这里确实有很多的信息。但是如果你对另一种OOP语言非常熟悉，或者已经使用过Delphi以前的版本，就能够将本章所介绍的概念应用到编程中。

理解Delphi语言的秘密和库对于要想成为一个专业的Delphi程序员来说是至关重要的。这些主题组成了VCL和CLX类库工作的基础；在下面两章对其介绍之后，我们将使用Delphi提供的各种组件研究真实应用程序的开发。

同时，第3章将给读者提供Delphi运行时库（RTL）的一个概述（主要是OOP很少涉及的功能集合）。RTL是Delphi用于执行基本任务的例程集合。第4章将提供关于该语言的更多信息，讨论与Delphi类库结构相关的特性，如published关键字的效果和事件的作用。而且第4章也将讨论组件库的整体结构。

第3章 运行时库

Delphi使用Object Pascal作为其编程语言，并支持面向对象的方法，同时结合了可视化的开发风格。这也是Delphi引人注目的地方，本书将介绍基于组件与可视化的开发；然而，必须强调这样一个事实，Delphi很多可用的特性都来自于它的“运行时库”，简称RTL。这是一个大型的函数集合，程序员可以使用这些函数在Pascal代码中执行简单的任务以及那些复杂的任务（在这里使用了“Pascal”，是因为运行时库中主要包含的是过程与函数，它们都是使用传统的编程语言结构编写的，而不是使用由Borland向编程语言加入的面向对象扩展）

实际上在本章介绍运行时库还有另外一个原因：Delphi 6对该库进行了大量的改进，而在Delphi 7中，则提供了更多的改进。一些新的函数组添加进来，有些函数被移到了新单元中，而且还修改了其他一些元素，建立了一些与已有代码不兼容的函数。所以即使读者使用过Delphi的老版本，并对于RTL很自信，也还是应该阅读本章的某些部分。

本章主要包括以下内容：

- RTL综述
- Delphi的RTL函数
- 转换引擎
- 数据、字符串与其他新的RTL单元
- TObject类
- 在运行时显示类信息

RTL的单元

在最新版本的Delphi中，RTL（运行时库）拥有新结构与一些新单元。添加新单元的原因是，增加了很多新函数。在大多数情况下，程序员会发现已有函数还在原来的单元中，但新函数出现在特定的单元中。例如，与日期有关的新函数出现在DateUtils单元中，但已有的日期函数没有从SysUtils中移出，这是为了避免与已有代码不兼容。

该规则的例外与一些变量支持函数有关，它们被移出了System单元，以避免与不需要的某些Windows库进行连接，甚至是在不使用这些特性的程序中。这些变量函数现在是新建Variants单元的一部分，稍后将在本章介绍。

警告：有些已有的Delphi 5或者Delphi 4的代码可能需要使用这个新建的Variants单元进行重新编译。

Delphi已经很敏锐地响应了这个需求，它会在使用Variant类型的项目中自动包含Variants单元，并且只是发出一个警告。

但还是应该稍稍调整一下，以尽可能减小可执行文件的大小，因为有时包含不必要的全局变量或初始化代码会增大可执行文件的大小。

Microscope下的可执行文件的大小

在调整RTL时, Borland工程师已经对每个Delphi应用程序实施了一些“减肥”措施。对于每天使用的所有臃肿的应用程序来说, 减少少许KB的程序看起来有些多余, 但对于开发人员却是一项很好的服务。有些情况下, 甚至减少很少KB的数据量(很多应用程序累积起来)也可以减小应用程序的大小, 并节省下载时间。

作为简单的测试, 下面建立一个MiniSize程序, 它的目的不是建立一个尽可能最小的程序, 而是要建立一个非常小的程序, 做一些有趣的事情: 它会报告自己可执行文件的大小。范例的全部代码如下所示:

```
program MiniSize;

uses
  Windows;

{$R *.RES}

var
  nSize: Integer;
  hFile: THandle;
  strSize: String;

begin
  // open the current file and read the size
  hFile := CreateFile (PChar (ParamStr (0)),
    0, FILE_SHARE_READ, nil, OPEN_EXISTING, 0, 0);
  nSize := GetFileSize (hFile, nil);
  CloseHandle (hFile);

  // copy the size to a string and show it
  SetLength (strSize, 20);
  Str (nSize, strSize);
  MessageBox (0, PChar (strSize),
    'Mini Program', MB_OK);
end.
```

该程序从第一个命令行参数 (ParamStr(0)) 中检索了其名称后, 打开它自己的可执行文件, 读取程序的大小, 使用简单Str函数将它转换成字符串, 并用一条消息显示结果。该程序没有提供窗口, 而且使用Str函数进行整数向字符串的转换, 以避免包含SysUtils——用于定义所有比较复杂的格式化子例程, 当然同时也会带来一点额外的资源耗资。

如果用Delphi 5编译该程序, 会获得18432字节大小的可执行文件。Delphi 6将该文件大小减少到15360字节, 节省出了大约3KB的内容。而Delphi 7仅将程序的大小缩减为15872字节。通过使用短字符串代替长字符串, 并稍稍修改代码, 还可以进一步清理程序, 将其大小减到小于10K字节。这是因为, 最终将删除字符串支持的子例程以及内存分配程序, 在程序中只有使用绝对低级的调用时才会使用到它。读者在范例的源代码中可发现这个版本的文件。

注意, 这种决定总暗示着某种权衡或折衷。例如, 在从使用变体的Delphi应用程序中消除它们的消耗时, Borland却为这些Delphi应用程序额外增加了一些开销。这种操作的真正

优点在于减少了不使用变体的Delphi应用程序的内存消耗，其结果是不必将数以兆字节计的Ole2系统库带入程序中。

所以，依笔者的观点来看，真正重要的是基于“运行时库”且已经成熟的Delphi应用程序的大小。范例MiniPack是一个简单的并不完成实际任务的程序，它简单测试显示了一个17 408字节的可执行文件大小。

随后的小节提供了一个Delphi RTL单元的列表，其中包括Delphi目录中位于Source\Rtl\Sys子文件夹下所有可以使用的单元（带有完整的源代码），以及位于Source\Rtl\Common子文件夹下的一些可以使用的单元。这个新目录包含了补充新RTL包的单元的源代码，由基于函数的库与核心类组成，将在本章结尾及第4章“核心库”中讨论。

说明：Delphi 5中的源VCL包现在被分成了VCL与RTL包，所以使用运行时包的非可视化应用程序就无需再为VCL可视化部分分配资源，从而节省这部分开销。另外，这种变化也有助于与Linux兼容，因为新包是在VCL与CLX库之间共享的。还要注意，Delphi 6和7中的包的名称中将不再包含版本号了。当它们被编译时，BPL会在其文件名中附带版本号，具体细节见第10章“库与组件包”。

我们下面将简单介绍每个单元的作用，以及所含每组函数的特性。此外，还将用更多篇幅讨论新添的Delphi单元。本书不提供所含函数的详细列表，因为在线帮助包含了相似的参考材料。在此将选出一些有意义或大家不太了解的函数，并简单讨论它们。

System与SysInit单元

System是RTL的核心单元，会自动包含在所有编译过程中（自动并隐式地使用了指向它的uses语句）。实际上，如果试图将该单元添加到程序的uses语句中，会得到一个编译时错误：

```
[Error] Identifier redeclared: System
```

这些系统单元包括：

- TObject类，这是Object Pascal语言中定义的所有类的基类，包括所有VCL的类（该类将在本章稍后讨论）。
- IInterface、IInvokable、IUnknown与IDispatch接口，以及比较简单的实现类TInterfacedObject。在Delphi 6中添加IInterface的目的主要是用来强调Delphi语言定义中的接口类型是独立于Windows操作系统的。IInvokable则是为了支持基于SOAP的请求而加入到Delphi 6中的。
- 一些变量支持代码，包括变体类型常量、TVarData记录类型与新的TVariantManager类型、大量的变量转换子例程以及变量记录与动态数组支持。这一部分与Delphi 5相比有很大的变化。变量的基本信息将在“Essential Pascal”电子图书的第10章中进行介绍（详细信息见本书附录C）。
- 很多基本数据类型，包括指针与数组类型以及在第2章“Delphi编程语言”中曾介绍过的TDateTime类型。
- 内存分配子例程，例如GetMem与FreeMem，以及实际的内存管理器，它由TMemoryManager记录定义，并由GetMemoryManager与SetMemoryManager函数访

问。为了获取信息，GetHeapStatus函数会返回一个THeapStatus数据结构。两个新的全局变量（AllocMemCount¹与AllocMemSize）保存了被分配内存段的号码与总量。更多有关内存以及这些函数使用的讨论见第8章“Delphi应用程序结构”中的ObjsLeft范例。

- 包（Package）与模块支持代码，包括PackageInfo指针类型、GetPackageInfoTable全局函数以及EnumModules过程（包内部特性的讨论见第12章）。
- 一个更长的全局变量列表，包括Windows应用程序实例MainInstance；IsLibrary，用来表明可执行文件是一个库、还是一个标准的程序；IsConsole，表明是一个控制台应用程序；IsMultiThread，说明是否存在二级线程；此外还有命令行字符串CmdLine（该单元还包括ParamCount与ParamStr，以便于访问命令行参数）。有些变量为Windows平台专用，有一些也可以用于Linux平台，还有一些变量是专用于Linux环境中的。
- 线程支持的代码，带有BeginThread¹与EndThread函数；文件支持记录和与文件相关的子例程；宽字符串与OLE字符串转换子例程；以及其他很多低级例程与系统例程（包括一些自动转换函数）。

System的伴随单元被称为SysInit，它包含了系统初始化代码，以及一些很少直接使用的函数。该单元也总是被隐式包含使用，因为它由System单元使用。

System单元中的新添特性

我们在上面的列表中已经介绍了System单元的一些有意思的新添特性，Delphi最新版本中的大多数修改都是为了使核心Delphi RTL更具跨平台的可移植性，用目前被Delphi与Kylix共享的更加通用的实现方法来代替Windows专用的特性。沿着这个方向，出现了一些新的名称来对应不同的接口类型，这种彻底修改是为了支持变量、新指针类型、动态数组以及用于定制异常对象管理的函数。

说明：如果查看System.pas的源代码，我们会注意到其中大量使用了条件编译，还包括很多{\$IFDEF LINUX}与{\$IFDEF MSWINDOWS}实例，它们用来在两个不同的操作系统之间进行区别。请注意对于Windows来说，Borland公司使用了MSWINDOWS的定义来指明其平台信息。因为WINDOWS是被用于16位操作系统中的（可以与WIN32符号进行比较）。

例如，在Linux与Windows的兼容性问题上，增强了与文本文件中的换行功能有关的特性。DefaultTextLineBreakStyle变量影响到那些包括大多数文本流例程在内的、进行文件读写操作的例程的行为。对于这个全局变量来说，它的默认值在Kylix环境中为tlbsLF，而在Dephi中的默认值是tlbsCRLF。换行形式还可以用SetTextLineBreakStyle函数逐个文件地进行设置。

相似地，全局模式字符串常量sLineBreak在Windows版本的IDE中具有值#13#10，而在Linux版本中具有值#10。其他的变化是在System单元中目前包括了TfileRec和TtextRec结构，这些结构在早期Delphi版本的SysUtils单元中进行了定义。

SysUtils与SysConst单元

SysConst单元定义了一些由其他RTL单元用于显示消息的常量字符串。这些字符串用resourcestring关键字来声明，并保存在程序资源中。与其他资源一样，它们可以通过Inte-

grated Translation Manager或External Translation Manager进行转换。

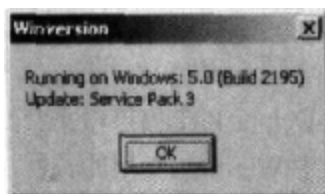
SysUtils单元是不同类型系统公用函数的一个集合。与其他RTL单元不同，它在很大程度上是一个与操作系统有关的单元。SysUtils单元没有专门的侧重点，但它包罗万象，从字符串管理到与多字节字符的本地化支持，从Exception类及其他一些派生异常类到大量的字符串格式化常量与子例程。特别地，我们将在本章的稍后部分重点介绍一些与文件管理例程相关的内容。

SysUtils的一些特性可能每天都在被每个程序员所使用，如IntToStr或Format字符串格式化函数；其他函数则很少为人所知，例如一些Windows版本信息全局变量。这些变量可说明Windows平台的类型（Windows 9x或NT/2000），操作系统版本与构件号，以及安装在NT上的服务包。可以如下列代码那样使用它们，这段代码摘自WinVersion范例：

```
case Win32Platform of
  VER_PLATFORM_WIN32_WINDOWS: ShowMessage ('Windows 9x');
  VER_PLATFORM_WIN32_NT:      ShowMessage ('Windows NT');
end;

ShowMessage ('Running on Windows: ' + IntToStr (Win32MajorVersion) + '.' +
  IntToStr (Win32MinorVersion) + ' (Build ' + IntToStr (Win32BuildNumber) +
  ') ' + #10#13 + 'Update: ' + Win32CSDVersion);
```

第二个代码段产生的消息如下图所示（当然，前提是安装了操作系统版）。



另一个鲜为人知的特性是一个支持多线程的类，它可能是拥有最长名称的VCL类，TMultiReadExclusiveWriteSynchronizer。Borland公司为这个类定义了别名，相比之下就简短很多：TMREWSync（这两个类是相同的）。这个类支持多线程操作，它允许程序员处理可以同时由多个线程读取的资源，但在编写时，必须由单个线程使用。这意味着写操作在所有读操作结束之前不能启动。

TMultiReadExclusiveWriteSynchronizer类的应用在Delphi 7中进行了改进和更新，但是这个改进在Delphi 6的Update 2之后的一个非正式补丁文件中已经存在了。这个新版本的类被进一步优化，并且不像原来那么容易死锁，这通常对于同步代码来说是一个问题。

说明：多线程读同步程序在支持递归加锁与促进读加锁与写加锁方面是独一无二的。该类的主要目标是允许多个线程从共享资源中轻松快速地读取，同时仍允许一个线程在相对很少更新的情况下独自控制资源。在Delphi中还有其他同步化的类，声明在SyncObjs单元中（可以在Source/Rtl/Common下找到），并紧密对应着操作系统的同步化对象（如Windows中的事件与临界区）。

新增的SysUtils函数

在最近几个Delphi版本中，SysUtils单元新添了一些函数，其中一个新领域与Boolean到字符串的转换有关。BoolToStr函数通常会返回“-1”与“0”，分别表示真与假值。如果

第二个可选的参数被确定, 该函数会在TrueBoolStrs与FalseBoolStrs数组中返回第一个字符串(默认情况下是“TRUE”与“FALSE”):

```
BoolToStr (True) // returns '-1'
BoolToStr (False, True) // returns 'FALSE' by default
```

相反功能的函数是StrToBool, 它可以转换包含两个Boolean数组(上面提到的)的一个值或一个数字值的字符串。在后一种情况中, 结果将是真, 除非数字值是零。在本章稍后的StrDemo范例中, 读者会看到一个使用Boolean转换函数的简单示例。

SysUtils的其他新函数都与浮点类型向货币与日期时间类型的转换有关: FloatToCurr与FloatToDateTime可以用于避免显式的类型转换。TryStrFloat与TryStrToCurr函数可以将字符串转换为浮点类型或货币值, 一旦出现错误, 它不会生成异常, 而是返回False值(如典型的StrToFloat与StrToCurr函数那样)。

Delphi 6提供了一个AnsiDequoteStr函数, 它可从字符串中删除引号, 与Delphi 5中添加的AnsiQuoteStr函数匹配。说到字符串, Delphi 6在宽字符串的支持上有很大改进, 提供了一系列新的子例程, 包括WideUpperCase、WideLowerCase、WideCompareStr、WideSameStr、WideCompareText、WideSameText与WideFormat。所有这些函数的操作都与AnsiString函数相似。

另外三个函数(TryStrToDate、TryEncodeDate与TryEncodeTime)用于将字符串转换为日期或对日期或时间进行编码, 但不会引起异常, 与前面提到的Try函数相似。而且, DecodeDateFully函数会返回更多的详细信息, 如今天是星期几; CurrentYear函数会返回今天的日期。

还有一个可移植的、友好的GetEnvironmentVariable函数的重载版本。这个新版本使用字符串参数代替PChar参数, 而且比使用PChar指针的老版本更容易使用:

```
function GetEnvironmentVariable(Name: string): string;
```

其他新函数与接口支持有关。Support函数中两个新添的重载版本允许程序员检查一个对象或类是否支持给定的接口。该函数在性能上对应着类的is操作符, 以及QueryInterface对象方法。下面是一个范例代码:

```
var
  W1: IWalker;
  J1: IJumper;
begin
  W1 := TAthlete.Create;
  // more code...
  if Supports (w1, IJumper) then
  begin
    J1 := W1 as IJumper;
    Log (J1.Walk);
  end;
```

还有一个IsEqualGUID函数以及两个用于将字符串与GUID相互转换的函数。函数CreateGUID已经被移到SysUtils单元中, 这样它可以在Linux上使用(这是一个定制的实现方式)。

最后, Delphi的近期版本中还包括有一些增强跨平台支持的特性。AdjustLineBreaks函数现在提供不同类型的回车与换行顺序的调整,并在System单元中为文本文件引入了新的全局变量,这一点我们在前面已经讨论过了。FileCreate函数也具有一个重载的版本,可以用它以UNIX的方式指定文件访问权利。ExpandFileName函数可以确定文件的位置(在区分大小写的文件系统上),即使它们的大小写没有严格对应。与路径定界符(反斜线与斜线)有关的函数变得更通用了,并相应地更改了名称(例如,IncludeTrailingBackslash函数现在被称做IncludingTrailingPathDelimiter)。

说到文件的问题, Delphi 7在SysUtils单元中加入了GetFileVersion函数,它可以从Windows可执行文件的附加可选版本信息中读取版本号(这也就是为什么这个函数不能够用于Linux环境中的原因)。

Delphi 7中对字符串格式例程的扩展

大多数的Delphi字符串格式化例程(参考附录C)使用全局变量来决定位数、千位分隔符以及时间/日期格式等等。在程序启动的时候,这些变量的值首先从系统中读取出来(Windows的本地化设置),然后程序员就可以根据自己的需要来对这些数值进行改变。但是,如果这个时候我们通过Windows的控制面板对这些设定进行了改变,那么应用程序将会响应这个信息并且更改自身变量的值,从而可能丢失我们在代码中所做的改变。

如果我们需要在应用程序的不同位置提供不同的输出格式,则可以利用重载字符串格式化例程的方法。它们使用了TFormatSettings的一个额外参数,其中包括了所有相关的设定。例如,这里有两个版本的Format:

```
function Format(const Format: string;  
    const Args: array of const): string; overload;  
function Format(const Format: string; const Args: array of const;  
    const FormatSettings: TFormatSettings): string; overload;
```

数十个函数带有这个新的参数,它可以用来代替全局设定。但是,也可以用运行程序的计算机上的默认配置来设定这个参数值,亦即调用一个新函数GetLocaleFormatSettings,这个函数只能够用在Windows平台上,它不支持Linux环境。

数学单元

Math单元含有一系列数学函数:大约40个三角函数、对数与指数函数、舍入函数、多项式计算,几乎30个统计函数与十多个金融函数。

描述该单元的所有函数有些过于繁琐,尽管有些读者可能对Delphi的数学能力特别感兴趣。出于上述原因,下面只介绍一些在Delphi中(特别是Delphi 6中)比较新的数学函数。同时我们也将特别介绍一个经常使Delphi程序员非常困惑的主题——舍入问题。

新的数学函数

Delphi向Math单元添加了一些新特性,包括支持无穷大常量(Infinity与NegInfinity)和相关的比较函数(IsInfinite与IsNan),用于余割与余切运算的新三角函数,以及新的角度转换函数。

一个方便的特性是能够使用重载的IfThen函数，它根据Boolean表达式返回两个可能值中的一个（有一个相似的函数现在可以用于字符串类型）。例如可以使用它计算出两个值中的最小值：

```
nMin := IfThen (nA < nB, nA, nB);
```

说明：IfThen函数与C/C++语言中的?:运算符相似，它非常方便，因为可以用一个更加精练的表达式代替一个完整的if/then/else语句，需要编写的代码也少得多，也不需要声明太多的临时变量。

我们可以使用RandomRange与RandomFrom来代替传统的Random函数，用以对RTL生成的随机值进行更多的控制。第一个函数会在两个指定的极值之间返回一个数值，第二个函数会从一个可能数值的数组（作为参数传递给它）中选择一个数值。

InRange布尔类型函数可以用于检查一个数值是否在其他两个数值之间。而另一方面，EnsureRange函数会迫使一个数值处于指定范围之间，如果数值在该范围之外，返回值将是数值自己或者是上限或下限。下面是一个范例：

```
// do something only if value is within min and max
if InRange (value, min, max) then
    ...

// make sure the value is between min and max
value := EnsureRange (value, min, max);
...
```

另一个非常有用的函数集合与比较操作有关。浮点值基本上是不精确的；浮点值是对理论实数值的近似。当对浮点值进行数学运算时，原值的不精确性会在结果中累加。乘以并除以相同的值可能不会严格地返回原值，而是一个非常接近的值。SameValue函数允许程序员检查两个值是否足够接近，以至被认为是相等的。我们可以指定两个值如何靠近才算是相等，或让Delphi使用表达式计算出一个合理的误差范围（这是函数被重载的原因）。同样，IsZero函数会比较一个值是否为零，它使用的是同样的“模糊逻辑”。

CompareValue函数对浮点值使用了相同规则，而且可以用于整型值，它会返回三个常量LessThanValue、EqualsValue与GreaterThanValue（分别对应着-1、0、1）中的一个。同样，新添的Sign函数会返回-1、0、1说明负值，零与正值。

DivMod函数等于div与mod运算，一次返回整除值与余数（或者称模）。RoundTo函数允许指定舍入位——例如，舍入到最靠近的千位或小数点后两位：

```
RoundTo (123827, 3); // result is 124,000
RoundTo (12.3827, -2); // result is 12.38
```

警告：注意RoundTo函数使用整数说明要舍入的十位的权数（例如，2表示百），负数表示小数点后的位数。这与电子表格，如Excel中使用的Round函数相反。

对Round函数提供的标准舍入运算也做了一些修改：现在可以通过调用SetRoundMode函数控制FPU（CPU的浮点单元）如何进行舍入。还有一些函数可以控制FPU的精度模式及其异常。

舍入问题

Delphi经典的Round函数以及更新的RoundTo函数都是与CPU/FPU的舍入算法相映射的。默认情况下，Intel的CPU使用银行家舍入算法，这个算法也常用于电子数据表格应用程序。

银行家舍入是基于下面这样一个假设的：如果在我们对两个值的（准确的）中间值（.5数值）进行舍入，将它们全部向上或者向下舍入的时候，可能会造成总数额的统计增加或者减少。出于这个原因，银行家舍入的法则规定，对于一个这样的中间值来说，是向上还是向下舍入要取决于这个数（不包括小数部分）是奇数还是偶数。通过这种方法，舍入操作将会趋于平衡，至少是统计上的平衡。读者可以在图3.1中看到银行家舍入的输出，这个例子中我们构建了不同的舍入类型进行演示。

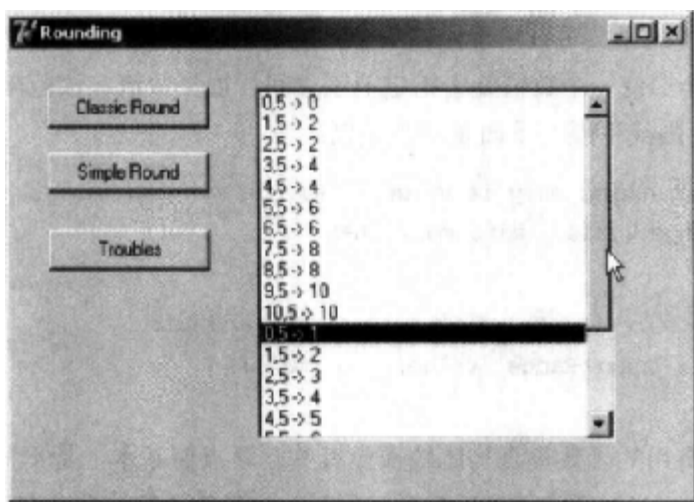


图3.1 Rounding示例，银行家的舍入和算法舍入

这个程序还使用了Math单元中的SimpleRoundTo函数提供的一种舍入方法，它使用非对称算术舍入（asymmetric arithmetic rounding）。在这个例子中，所有的 .5数值都被向上舍入，但是，请注意示例中加重的部分，这个函数对于小数位的舍入并不能像我们希望的那样进行（当给出一个负值的时候）。在这个情况下，由于在符点数的操作中屡次出现错误，舍入的计算就修剪了这些数值。例如，它将1.15舍入为1.1而不是1.2。这个解决方案是在进行舍入之前先将数值乘以10，将它暂时转化成一个没有小数位的数字，然后再进行操作，正如下列范例所示的那样：

```
SimpleRoundTo (d * 10, 0) / 10)
```

ConvUtils与StdConv单元

新添的ConvUtils单元包含了由Delphi 6引入的转换引擎的核心，它使用第二个单元StdConv定义的转换常量。本章稍后会介绍这两个单元，说明如何利用新添的度量单元来扩展它们。

说明：Delphi 7在转换单位方面只有很简单的一个改进：它加入了对“石”（Stone）的支持（英制单位，等于14磅）。在任何情况下，如果我们在代码中必须对单位进行转换，则就会对这个引擎中的特性非常欣赏。

DateUtils单元

DateUtils单元是新添的日期和时间函数的集合。它包含了用于从TDateTime变量中提取值的新函数或从给定时间间隔中计算值的函数，如：

```
// pick value
function DayOf(const AValue: TDateTime): Word;
function HourOf(const AValue: TDateTime): Word;
// value in range
function WeekOfYear(const AValue: TDateTime): Integer;
function HourOfWeek(const AValue: TDateTime): Integer;
function SecondOfHour(const AValue: TDateTime): Integer;
```

其中一些函数实际上很古怪，如MilliSecondOfMonth或SecondOfWeek，但Borland的开发人员决定提供完整的函数集合，无论它们看起来如何不切合实际。我们在第2章中实际上曾使用过其中的一些函数来建立TDate类。

有一些函数用于计算给定时间间隔（日、周、月、年）的初值或终值，包括当前日期、时间范围检查与查询；例如：

```
function DaysBetween(const ANow, AThen: TDateTime): Integer;
function WithinPastDays(const ANow, AThen: TDateTime;
    const ADays: Integer): Boolean;
```

其他函数涉及可能时间间隔的递增与递减，编码与“重编码”（用新元素代替TDateTime值的一个元素，如“日”数值），进行“模糊”比较（近似比较，可以认为仅差1毫秒的两个日期是相同的）。总起来说，DateUtils非常有意思，而且使用起来并不很困难。

StrUtils单元

StrUtils单元是Delphi 6中的一个新添单元，带有一些新添的、与字符串有关的函数。该单元的主要特性之一是具有很多新添的字符串比较函数。这些函数均基于“探测”（soundex）算法（AnsiResembleText），其中一些则提供了对字符串数组的查找（AnsiMarchText与AnsiIndexText）、子字符串定位以及文本替代（包括AnsiContainsText与AnsiReplaceText）。

说明：探测法是一种比较名称的算法，基于它们如何发音，而不是它们如何拼写。该算法会为每个词的发音计算一个数值，这样，比较两个数值就可以确定两个名称听起来是否相似。该系统最早在1880年由U.S.的人口普查局使用，1918年获得专利，现在广泛地应用于公共领域。探测法代码是一个索引系统，会将名称翻译成一个四字符的代码，由一个字母与三个数值组成。如需了解更多信息，请参见www.nara.gov/genealogy/coding.html。

除了比较功能外，其他函数还提供了一个双向检测（IfThen函数，与前面介绍的数值函数相似）、复制与反转字符串，以及替代子字符串。在这些字符串函数中，大多数是为了让Visual Basic程序员能够方便地转移到Delphi上来而添加的。

本书选配光碟中的StrDemo范例使用了其中一些函数，该范例还使用了一些在SysUtils单元中新定义的Boolean类型到字符串类型的转换函数。该程序实际上可以用于测试其中的

一些函数。例如，它在比较两个编辑框中的字符串时使用了“探测法”，然后将Boolean结果转换为字符串并显示它：

```
ShowMessage (BoolToStr (AnsiResemblesText
  (EditResemble1.Text, EditResemble2.Text), True));
```

该程序在使用一个列表框内的字符串填充一个动态字符串数组 (strArray) 之后，也展现了AnsiMatchText与AnsiIndexText函数。我们可以使用更简单的TStrings类的IndexOf方法，但是这样做将会与本范例的目的背道而驰。关于两个列表的比较请参考下例：

```
procedure TForm1.ButtonMatchesClick(Sender: TObject);
begin
  ShowMessage (BoolToStr (AnsiMatchText(EditMatch.Text, strArray), True));
end;

procedure TForm1.ButtonIndexClick(Sender: TObject);
var
  nMatch: Integer;
begin
  nMatch := AnsiIndexText(EditMatch.Text, strArray);
  ShowMessage (IfThen (nMatch >= 0, 'Matches the string number ' +
    IntToStr (nMatch), 'No match'));
end;
```

请注意在最后几行中IfThen函数的使用，它根据初始化测试的结果 (nMatch>=0) 具有两个不同的输出字符串。

其他三个按钮完成对其他三个新函数的调用，其代码如下：

```
// duplicate (3 times) a string
ShowMessage (DupeString (EditSample.Text, 3));
// reverse the string
ShowMessage (ReverseString (EditSample.Text));
// choose a random string
ShowMessage (RandomFrom (strArray));
```

从Pos到PosEx

Delphi 7只对StrUtils单元稍做加强。新的PosEx函数对于很多开发人员来说将会显得更加方便，因此这里我们做一个简短的介绍。当在一个字符串中查找另外一个字符串的多个存在的时候，一个经典的Delphi解决办法是使用Pos函数，对字符串的剩余部分进行反复搜索。例如，使用下面的代码来统计一个字符串中其他字符串出现的次数：

```
function CountSubstr (text, sub: string): Integer;
var
  nPos: Integer;
begin
  Result := 0;
  nPos := Pos (sub, text);
  while nPos > 0 do
```

```
begin
  Inc (Result);
  text := Copy (text, nPos + Length (sub), MaxInt);
  nPos := Pos (sub, text);
end;
end;
```

新的PosEx函数允许用户在一个字符串中指定一个开始搜索的位置，因此我们能够不必更换源字符串（这样会浪费很多的时间）。这样，可以对上述代码进行如下简化：

```
function CountSubstrEx (text, sub: string): Integer;
var
  nPos: Integer;
begin
  Result := 0;
  nPos := PosEx (sub, text, 1); // default
  while nPos > 0 do
  begin
    Inc (Result);
    nPos := PosEx (sub, text, nPos + Length (sub));
  end;
end;
```

这两个代码片断均可以用在前面讨论过的StrDemo范例中。

Types单元

Types单元是一个新的Pascal文件，包含了多个操作系统公用的数据类型。在Delphi过去的版本中，相同类型都由Windows单元定义，现在它们被移到该公共单元中，由Delphi与Kylix共享。这里定义的类型都比较简单，包含了TPoint、TRect与TSmallPoint记录结构，以及与它们相关的指针类型。

警告：请注意我们必须升级引用TRect和TPoint的旧Delphi程序，亦即在uses语句中添加Type单元，否则编译将不能够进行。

Variants与VarUtils单元

Variants与VarUtils是Delphi 6中引入的两个与变体有关的新单元。Variants单元包含了变体的通用代码。前面曾提到，该单元中的有些子例程是从System单元中移植到这里的。函数包括通用的变体支持、变体数组、变体复制以及动态数组向变体数组的转换。还有TCustomVariantType类，它定义了定制的变体数据类型。

Variants单元总体上是与平台无关的，并使用VarUtils单元，它包含了与OS相关的代码。在Delphi中，该单元使用系统API处理变体数据；而在Kylix中，它使用一些由RTL库提供的定制代码。

说明：在Delphi 7中，这些单元均有所扩展，并且一些缺陷也被排除掉。在增加这种技术的速度与减少代码的内存占用的同时，对变量的实现部分也进行了大量的修改。

Delphi 7中的一个显著改进是具备了控制变量实现的能力，特别是比较法则。Delphi 6中，**null**值不能够与其他的值进行比较。这个行为从正规的观点来看是正确的，特别是对于一个数据集合（一个变量被大量使用的地方）来说。但是这个改变的副作用是破坏了现有的代码。现在，我们可以通过**NullEqualityRule**和**NullMagnitudeRule**全局变量控制这个行为，它们中的每一个都假设具有下面的值：

ncrError 任何类型的比较都会导致一个异常的产生，因为不能比较没有被定义的数值；这是Delphi 6中的默认设置。

ncrStrict 任何类型的比较总是失败（返回**False**），无论是什么值。

ncrLoose 相等测试只在**null**值之间成功（一个**null**与其他任何的数值不同）。在比较中，**null**值被认为是空或者是零。

其他的设定，如**NullStrictConvert**和**NullAsStringValue**，控制了在使用**null**值的情况下转换完成的方式。我们建议读者根据自己的经验来使用本章中**VariantComp**范例的代码。正如我们在图3.2中看到的那样，这个程序具有一个**RadioGroup**，可以使用它来改变全局变量**NullEqualityRule**及**NullMagnitudeRule**的设定。另外还有一些按钮可执行各种比较操作。

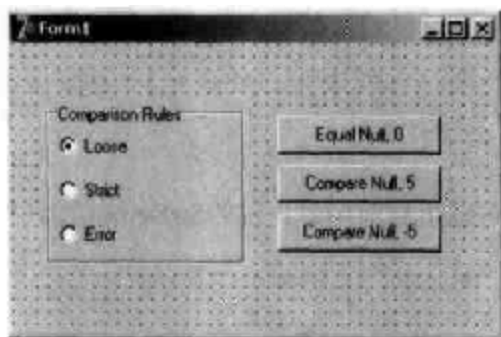


图3.2 VariantComp示例的设计时窗体

定制的变量与复数

用定制变体来扩展类型系统的可能性是Delphi中新添的特性，它允许程序员定义新的数据类型，与一个类相反，它重载了标准的算术运算符。

事实上，变量是一种既含有类型描述又含有实际值的类型。一个变量可以包含字符串，另一个变量可以包含数值。系统定义了变量类型之间的自动转换，允许在操作中组合它们（包括定制变体）。这种灵活性带来了高额代价：对变量做运算要比原本的类型慢得多，而且变量也使用了过多内存。

作为定制变量类型的一个范例，Delphi 6携带了一个复数的有趣定义，见**VarCmplx**单元（**RtlCommon**文件夹的源代码格式中可用）。程序员可以使用重载的**VarComplexCreate**函数建立复数变体，并在任何表达式中使用它们，如下面的代码所示：

```
var
  v1, v2: Variant;
begin
  v1 := VarComplexCreate (10, 12);
  v2 := VarComplexCreate (-10, 1);
  ShowMessage (v1 + v2 + 5);
```

复数实际是用类定义的，但它们通过从TCustomVariantType类（在Variants单元中定义）继承新类，覆盖一些虚拟的抽象函数，并建立一个全局对象来负责系统中的注册工作。

除了这些内部定义外，Variants单元还包含了大量的变体运算符例程，包括数学与三角运算。我们将这些函数留给读者来研究，因为并不是所有的读者都对复数的使用感兴趣。

警告：建立一个定制的变体不是容易的事，而且我们也几乎找不出使用它们代替对象与类的合理性。

事实上，使用定制变体可以获得在自己的数据结构上重载运算符的好处，但却会失去编译时的检查功能，使代码速度变慢，失去一些COP特性，同时，还必须编写大量相对复杂的代码。

DelphiMM与ShareMem单元

DelphiMM与ShareMem单元与内存管理有关。实际的Delphi内存管理器声明在System单元中。

DelphiMM单元定义了另外一个内存管理器库，以便从可执行文件向DLL（Windows动态链接库）传递字符串时使用，二者都由Delphi建立。这个内存管理库在默认条件下是由Borlndmm.dll库文件编译的，读者需要在自己的程序中部署这个文件。

该内存管理器的接口定义在ShareMem单元中，该单元必须包含在可执行文件与库的项目中（要求作为第一个单元）。第10章“库与组件包”中会详细介绍。

说明：与Delphi不同，Kylix中不具有DelphiMM与ShareMem单元，因为内存管理的功能是由Linux本身的库来实现的（具体说来，Kylix使用的是glibc中的malloc），并且在不同的模块中被有效地共享使用。但是，在Kylix中，具有多个模块的应用程序必须使用ShareExcept单元，它允许某个模块中产生的异常不会影响到其他的模块。

COM相关的单元

与COM相关的单元ComConts、ComObj和ComServ提供了低级的COM支持。因为这些单元其实不属于RTL，出于考虑，将不在这里详细讨论它们。读者可以在第12章参考有关信息。这些单元在Delphi的最近版本中没有太多改变。

数据转换

就像前面提到的那样，Delphi包含一个新的转换引擎，定义在ConvUtils单元中。该引擎自己没有包含任何对实际度量单位的定义，而是为终端用户提供了--系列核心函数。

关键的函数是实际转换调用，即Convert函数，只需提供数量、它的度量单位以及要转换成的单位即可。下面的代码将把摄氏31度转换成华氏温度：

```
Convert (31, tuCelsius, tuFahrenheit)
```

Convert函数的一个重载版本允许转换具有两个单位的值，如速度（它具有长度与时间单位）。例如，用下列调用可以将每小时英里数转换成每秒米数：

```
Convert (20, duMiles, tuHours, duMeters, tuSeconds)
```

单元中的其他函数允许转换一个相加或相减的结果，检查转化是否可以应用，甚至列出可用转换的系列与单位。

StdConv单元中提供了一个度量单位的预定义集合，该单元含有转换系列与一些实际值，如下面的摘录所示：

```
// Distance Conversion Units
// basic unit of measurement is meters
cbDistance: TConvFamily;

duAngstroms: TConvType;
duMicrons: TConvType;
duMillimeters: TConvType;
duMeters: TConvType;
duKilometers: TConvType;
duInches: TConvType;
duMiles: TConvType;
duLightYears: TConvType;
duFurlongs: TConvType;
duHands: TConvType;
duPicas: TConvType;
```

这个系列与变量单元都被单元的初始化部分注册在转换引擎中，提供了转换比率（存储在一系列常数中，例如下面代码中的MetersPerInch）：

```
cbDistance := RegisterConversionFamily('Distance');
duAngstroms := RegisterConversionType(cbDistance, 'Angstroms', 1E-10);
duMillimeters := RegisterConversionType(cbDistance, 'Millimeters', 0.001);
duInches := RegisterConversionType(cbDistance, 'Inches', MetersPerInch);
```

为了测试这个转换引擎，我们构建了一个通用的范例（ConvDemo），它允许用户对全部变量转换进行操作。这个程序中包含有一个组合框，它列举了一些可用的转换系列以及一个列表框，其中显示了活动系列中的可用单元，下面是相关的代码：

```
procedure TForm1.FormCreate(Sender: TObject);
var
    i: Integer;
begin
    GetConvFamilies (aFamilies);
    for i := Low(aFamilies) to High(aFamilies) do
        ComboFamilies.Items.Add (ConvFamilyToDescription (aFamilies[i]));
    // get the first and fire event
    ComboFamilies.ItemIndex := 0;
    ChangeFamily (self);
end;

procedure TForm1.ChangeFamily(Sender: TObject);
var
    aTypes: TConvTypeArray;
    i: Integer;
begin
    ListTypes.Clear;
```

```

CurrFamily := aFamilies [ComboFamilies.ItemIndex];
GetConvTypes (CurrFamily, aTypes);
for i := Low(aTypes) to High(aTypes) do
  ListTypes.Items.Add (ConvTypeToDescription (aTypes[i]));
end;

```

变量aFamilies和CurrFamily在窗体的私有部分被声明如下:

```

aFamilies: TConvFamilyArray;
CurrFamily: TConvFamily;

```

到目前为止, 一个用户可以输入两个度量单元, 在相应的编辑框中输入一个数量, 如我们在图3.3中看到的。为了使这个操作更加快捷, 读者可以在列表框中选择一个数值并且将其拖进两个类型编辑框之一。对这个拖拽功能的支持我们将在下面一节中进行讨论。

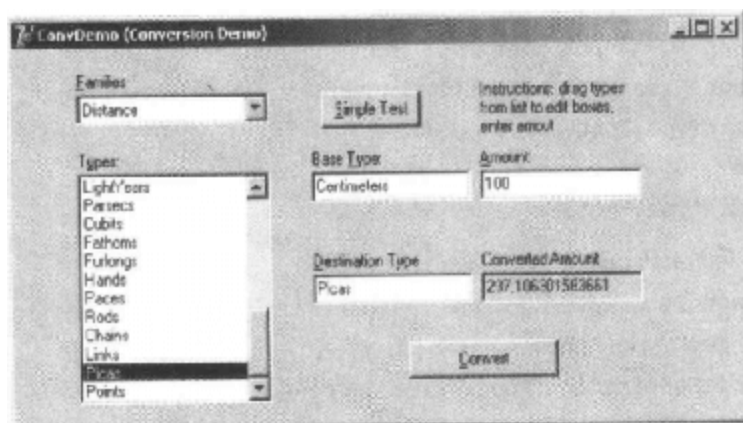


图3.3 运行时的ConvDemo示例

Delphi中的简单拖动

本书建立的ConvDemo范例显示了如何使用Delphi的转换引擎, 其中使用了一项有意思的技术: 拖动。事实上, 可以将鼠标移到列表框上, 选择其中一项, 然后按住鼠标左键, 将该项拖到窗体中央的一个编辑框中。

为了实现该操作, 必须将列表框(源组件)的DragMode属性设置为dmAutomatic, 并实现目标编辑框的OnDragOver与OnDragDrop事件(两个编辑框都与相同的事件处理程序相连, 共享相同的代码)。在第一个对象方法中, 程序说明编辑框总是接受拖动操作, 而不考虑源组件。在第二个对象方法中, 程序将把在列表框(拖动操作的Source控件)中选择的文本复制到可触发事件(Sender对象)的编辑框中。下面是两个对象方法的代码:

```

procedure TForm1.EditTypeDragOver(Sender, Source: TObject;
  X, Y: Integer; State: TDragState; var Accept: Boolean);
begin
  Accept := True;
end;

procedure TForm1.EditTypeDragDrop(Sender, Source: TObject;
  X, Y: Integer);
begin
  (Sender as TEdit).Text := (Source as TListBox).Items

```

```
[(Source as TListBox).ItemIndex];
end;
```

其单位必须与当前系列中可以使用的单位匹配。如果出现错误，Type编辑框的文本将显示为红色。这是窗体DoConvert对象方法第一部分的作用，只要包含单位或数量的任何一个编辑框中的值发生变化，它就会被触发。在检查了编辑框中的类型之后，DoConvert对象方法将执行实际的转换操作，并在第四个，变灰的编辑框中显示结果。如果出现错误，就会在同一编辑框中显示相应的消息。下面是有关代码：

```
procedure TForm1.DoConvert(Sender: TObject);
var
  BaseType, DestType: TConvType;
begin
  // get and check base type
  if not DescriptionToConvType(CurrFamily, EditType.Text, BaseType) then
    EditType.Font.Color := clRed
  else
    EditType.Font.Color := clBlack;
  // get and check destination type
  if not DescriptionToConvType(CurrFamily, EditDestination.Text,
    DestType) then
    EditDestination.Font.Color := clRed
  else
    EditDestination.Font.Color := clBlack;
  if (DestType = 0) or (BaseType = 0) then
    EditConverted.Text := 'Invalid type'
  else
    EditConverted.Text := FloatToStr (Convert (
      StrToFloat (EditAmount.Text), BaseType, DestType));
end;
```

如果所有这些都不足以引起读者的兴趣，可以考虑转换类型只是提供了一个演示范例：大家完全可以通过提供自己感兴趣的度量单位来定制自己的引擎，如下一节中描述的那样。

关于货币转换的问题

转换货币与转换度量单位不同，因为货币比率变化非常快。理论上讲，可以用Delphi的转换引擎注册一个转换比率。有时，可能会检查新的兑换比率，不注册已有的转换比率，而注册新的转换比率。然而，跟上实际比率变化意味着要更加频繁地修改转换比率，这可能会导致操作变得没有太大意义了。另外，还必须进行三角转换：必须定义一个基本单位（如果生活在美国，可能是美元），为了在其他两种货币之间进行转换，首先要进行它们与基本单位之间的转换。

更有意思的是使用引擎转换欧元的成员货币，这里有两个原因。第一，转换比率是固定的（直到单一欧元货币实际接管）。第二，欧元货币之间的转换从法律上讲应该首先将一种货币转换成欧元，然后从欧元转换为其他货币，这一操作完全符合Delphi转换引擎的行为。只有一个小问题，因为在每一步转换中应该使用舍入算法，故在用Delphi 6的转换引擎为统一的欧元货币提供基础代码之后，将考虑该问题。

说明：Delphi中的ConvertIt演示范例为欧元转换提供了支持，它使用一种稍稍不同的舍入方法，但是还不能如同欧洲货币兑换规则要求的那样精确。本书决定保留该范例，因为它对于演示如何建立一个新的度量系统有好处。

范例名为EuroConv，它的实际意义在于演示如何使用引擎注册任何新度量单位。下面是StdConvs单元提供的模板，它建立了一个新单元（叫做EuroConvConst），而且在接口部分，为系列与特定单位声明了变量，如下所示：

```
interface

var
    // Euro Currency Conversion Units
    cbEuroCurrency: TConvFamily;

    cuEUR: TConvType;
    cuDEM: TConvType; // Germany
    cuESP: TConvType; // Spain
    cuFRF: TConvType; // France
    // and so on...
```

该单元的实现部分为不同的官方转换比率定义了一些常量：

```
implementation

const
    DEMPerEuros = 1.95583;
    ESPPerEuros = 166.386;
    FRFPerEuros = 6.55957;
    // and so on...
```

最后，该单元的初始化代码将这个系列与各种货币进行了注册，每一个都使用它们自己的转换比率以及一个可读的名称：

```
initialization
    // Euro Currency's family type
    cbEuroCurrency := RegisterConversionFamily('EuroCurrency');

    cuEUR := RegisterConversionType(
        cbEuroCurrency, 'EUR', 1);
    cuDEM := RegisterConversionType(
        cbEuroCurrency, 'DEM', 1 / DEMPerEuros);
    cuESP := RegisterConversionType(
        cbEuroCurrency, 'ESP', 1 / ESPPerEuros);
    cuFRF := RegisterConversionType(
        cbEuroCurrency, 'FRF', 1 / FRFPerEuros);
```

说明：这个引擎使用了一个基本单位的转换因子来获得从属单位，并使用常量MetersPerInch。标准的欧元货币比率的定义方法是相反的。出于这个原因，我们将保留官方的数值常量（如DEMPerEuros）并且将1/DEMPerEuros传递给引擎。

通过注册这个单位，我们可以将120个德国马克转换为意大利里拉，如下所示：

```
Convert (120, cuDEM, cuITL)
```

这个演示程序还完成了其他的工作：提供两个列表框，其中显示了可用的货币，正如从前面的范例中提取的那样；还有一个编辑框，用于输入和显示最终的结果。我们可以在图3.4中看到。

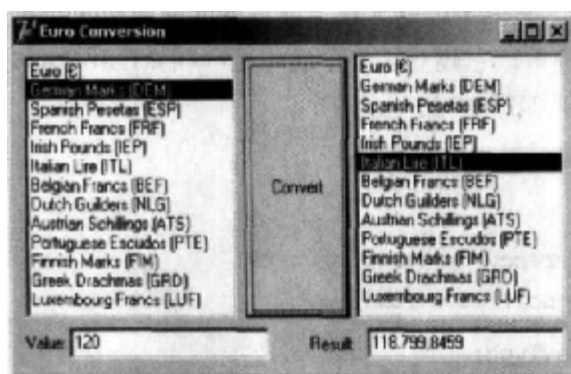


图3.4 EuroConv单元的输出，显示了Delphi的转换引擎的使用

该程序可以正常运行，但还不完美，因为没有使用合适的舍入操作。事实上，不仅要转换的结果进行舍入，而且要对中间值进行舍入。直接使用转换引擎实现该操作并不容易。引擎允许提供一个定制的转换函数或转换比率，但为所有货币编写同样的转换函数看来不是好的想法，所以决定采取一种不同的方法（读者可以在StdConv单元中看到定制转换函数的范例——在与温度相关的小节中）。

在EuroConv范例中，向含有转换比率的单元添加了一个定制的函数，名为EuroConv，它可以进行正确的转换。只需要调用该函数，而不是标准的Convert函数即可（实际上该方法没有缺点，因为在这样的程序中，很难用米或温度固定货币）。作为一种可供选择的方法，可以从TConvTypeFactor中继承一个新类，提供新版本的FromCommon与ToCommon对象方法；或者调用RegisterConversionType的重载版本，它接受两个函数作为参数。然而，这些技术都不允许处理特殊情况，例如一种货币向自身的转换。

下面是EuroConv函数的代码，它使用内部的EuroRound函数舍入Decimals参数中指定的位数（它必须在3到6之间，对应着官方规则）：

```
type
  TEuroDecimals = 3..6;

function EuroConvert (const AValue: Double;
  const AFrom, ATo: TConvType;
  const Decimals: TEuroDecimals = 3): Double;

function EuroRound (const AValue: Double): Double;
begin
```

```

    Result := AValue * Power (10, Decimals);
    Result := Round (Result);
    Result := Result / Power (10, Decimals);
end;

begin
    // check special case: no conversion
    if AFrom = ATo then
        Result := AValue
    else
        begin
            // convert to Euro, then round
            Result := ConvertFrom (AFrom, AValue);
            Result := EuroRound (Result);
            // convert to currency then round again
            Result := ConvertTo (Result, ATo);
            Result := EuroRound (Result);
        end;
    end;
end;

```

当然，我们可能希望通过提供与其他非欧元货币的转换来扩展这个范例，最终实现自动地从Web站点上获得数值。我们将把这个问题作为一个练习题目留给我们的读者。

使用SysUtils来管理文件

为了访问文件和文件信息，我们通常依靠SysUtils单元中的标准函数。使用这些很传统的Pascal库将会使代码可以很容易地在不同的操作系统中进行移植，尽管需要仔细考虑各种文件系统体系结构的差异，特别是Linux平台上的大小写敏感的问题。

例如，FilesList范例中使用了FindFirst、FindNext和FindClose的组合，用以从一个文件夹中获得符合过滤条件的文件列表，同样的代码也可以使用在Kylix和Linux环境中。该范例的输出如图3.5所示。

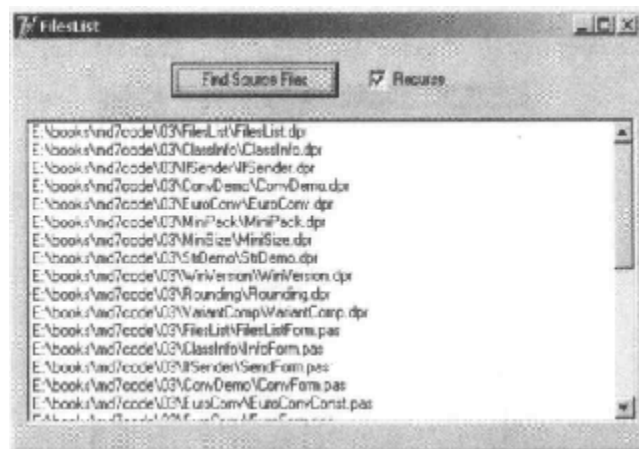


图3.5 FilesList应用程序输出的一个例子

下面的代码向列表框lbFiles添加了文件名:

```

procedure TForm1.AddFilesToList(Filter, Folder: string;
  Recurse: Boolean);
var
  sr: TSearchRec;
begin
  if FindFirst (Folder + Filter, faAnyFile, sr) = 0 then
    repeat
      lbFiles.Items.Add (Folder + sr.Name);
    until FindNext(sr) <> 0;
  FindClose(sr);

```

如果Recurse参数被设定, AddFilesToList过程将通过再次检查本地文件取得一个子文件夹的列表, 然后针对每一个子文件夹调用它自己。这个文件夹的列表被放在一个字符串列表对象中, 其代码如下所示:

```

procedure GetSubDirs (Folder: string; sList: TStringList);
var
  sr: TSearchRec;
begin
  if FindFirst (Folder + '.*', faDirectory, sr) = 0 then
    try
      repeat
        if (sr.Attr and faDirectory) = faDirectory then
          sList.Add (sr.Name);
        until FindNext(sr) <> 0;
      finally
        FindClose(sr);
      end;
    end;

```

最后, 这个程序使用了一种有趣的技术来请用户选择进行文件搜索的初始化目录——通过调用SelectDirectory过程(如图3.6所示):

```

if SelectDirectory ('Choose Folder', '', CurrentDir) then ...

```

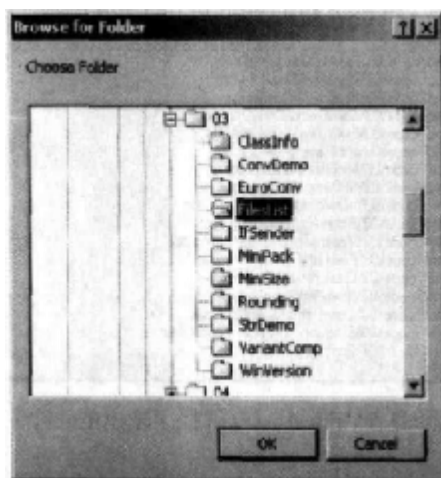


图3.6 SelectDirectory过程的对话框, 由FindFile应用程序显示

TObject类

如前面所说的，系统单元的关键元素是TObject类的定义，它是所有Delphi类之母。系统中的每一个类都是TObject类的子类，无论是直接的（如果指定TObject为基类）、隐含的（如果不指定任何基类）或者是间接的（如果指定其他的类）。所以Pascal对象整个类的层次式结构只有一个根类。这允许用户在系统中用TObject数据类型替代任何类的数据类型。当然，还要遵循第2章中“继承与类型兼容”一节介绍的原则。

例如，事件处理程序通常有一个TObject类型的Sender参数，这意味着Sender对象可属于任何类，因为每个类都是从TObject类派生出来的。这种方式的一个典型缺陷是，在对象上进行操作时，用户需要知道它的数据类型。事实上，TObject类型的变量或参数只能应用于由TObject定义的对象方法和属性。例如，如果该变量或参数偶然引用了一个TButton类型的对象，用户将不能直接访问它的Caption属性。该问题的解决方案要使用第2章讨论的安全强制下载或运行时类型信息（RTTI）操作符（is和as）。

还有另一种方法。对于任意对象，都可调用定义在TObject类本身上的对象方法。例如，ClassName对象方法用类的名称返回一个字符串。因为它是类的对象方法（参考第2章获得详细信息），故可以将其应用于对象和类。假如用户定义了TBunon类和该类的Button1对象，那么，下面两个语句的效果是相同的：

```
Text := Button1.ClassName;  
Text := TButton.ClassName;
```

使用类名的机会不是很多，但在检索对类本身或其基类的引用时，它就可能非常有用。事实上，类引用允许在运行时对类进行操作（如上一章中所述），而类名称只不过是字符串罢了。可以使用ClassType和ClassParent对象方法来得到这些类引用。第一个方法返回一个该对象类的类引用；第二个方法返回一个该对象基类的类引用。一旦有类引用，可以把它当做对象使用。例如，调用ClassName对象方法。

另一个有用的对象方法是InstanceSize，它返回对象运行时的大小，尽管可以使用SizeOf全局函数，但实际上，该函数返回的是一个对象引用，即一个指针的大小，它的大小总是四个字节，而不是这个对象自身的实际大小。

在程序清单3.1中，我们可以找到完整的关于TObject类的定义，它来自System单元。除了我们提到的那些方法外，请注意InheritsFrom，它提供了与is操作符类似的测试功能，但是也可以应用于类以及类引用（is的第一个参数必须是一个对象）。

程序清单3.1 TObject类的定义（在System RTL单元中）

```
type  
  TObject = class  
    constructor Create;  
    procedure Free;  
    class function InitInstance(Instance: Pointer): TObject;  
    procedure CleanupInstance;  
    function ClassType: TClass;
```



```

class function ClassName: ShortString;
class function ClassNameIs(
    const Name: string): Boolean;
class function ClassParent: TClass;
class function ClassInfo: Pointer;
class function InstanceSize: Longint;
class function InheritsFrom(AClass: TClass): Boolean;
class function MethodAddress(const Name: ShortString): Pointer;
class function MethodName(Address: Pointer): ShortString;
function FieldAddress(const Name: ShortString): Pointer;
function GetInterface(const IID: TGUID; out Obj): Boolean;
class function GetInterfaceEntry(
    const IID: TGUID): PInterfaceEntry;
class function GetInterfaceTable: PInterfaceTable;
function SafeCallException(ExceptObject: TObject;
    ExceptAddr: Pointer): HRESULT; virtual;
procedure AfterConstruction; virtual;
procedure BeforeDestruction; virtual;
procedure Dispatch(var Message); virtual;
procedure DefaultHandler(var Message); virtual;
class function NewInstance: TObject; virtual;
procedure FreeInstance; virtual;
destructor Destroy; virtual;
end;

```

说明: ClassInfo方法返回了一个指针, 指向的是内部的运行时类型信息(RTTI), 我们将在下面一章进行介绍。

这些TObject方法对于每一个类的对象都可用, 因为TObject是所有类的共同祖先类。下列代码演示了我们如何使用这些方法来访问类的信息:

```

procedure TSenderForm.ShowSender(Sender: TObject);
begin
    Memo1.Lines.Add ('Class Name: ' + Sender.ClassName);

    if Sender.ClassParent <> nil then
        Memo1.Lines.Add ('Parent Class: ' + Sender.ClassParent.ClassName);
    Memo1.Lines.Add ('Instance Size: ' + IntToStr (Sender.InstanceSize));
end;

```

这段代码检查了在我们使用一个没有基类型的TObject类型实例时, ClassParent是否是nil。

ShowSender方法是IfSender范例的一部分。这个方法与很多控件的OnClick事件相连接: 三个按钮、一个检查框以及一个编辑框。当单击每一个控件的时候, 其上的ShowSender方法就被激活。按钮之一是位图按钮, 是一个TButton子类的对象。我们可以在图3.7中看到这个范例的程序输出。

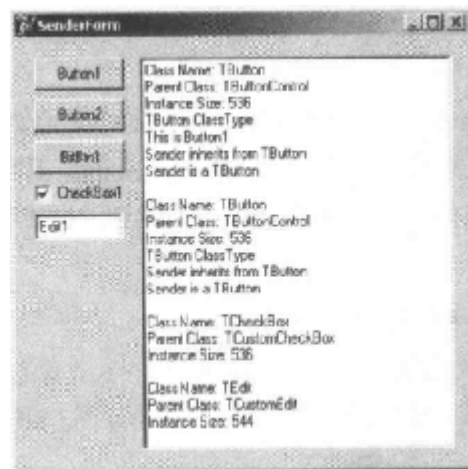


图3.7 IfSender示例的输出

我们可以使用其他方法来完成这个测试，例如，通过下面的代码检查是否Sender对象属于一个指定的类型：

```
if Sender.ClassType = TButton then ...
```

我们还可以检查Sender参数是否与一个给定的对象相关联：

```
if Sender = Button1 then...
```

除了检测特殊类或对象外，通常还需要检查对象与给定类的类型是否兼容，也就是说，需要检查对象的类是否是给定类或其子类。该项检查使我们知道，是否可以使用为该类定义的对象方法对其对象进行操作。可以使用InheritsFrom对象方法完成该检查，当使用is操作符时也调用了该对象方法。下面两条检查语句是等效的：

```
if Sender.InheritsFrom (TButton) then ...
```

```
if Sender is TButton then ...
```

显示类信息

IfSender范例可以进一步扩展，以显示已给对象或类的基类的完整列表。事实上，一旦拥有了类引用，就可以使用下列代码将其所有基类添加到ListParent列表框中：

```
with ListParent.Items do
begin
  Clear;
  while MyClass.ClassParent <> nil do
  begin
    MyClass := MyClass.ClassParent;
    Add (MyClass.ClassName);
  end;
end;
```

读者会注意到，while循环中使用了一个类引用，该循环用于测试当前类是否没有父类（也就是说，测试当前类是否为TObject类）。还可以使用下面两种方式编写while语句：

```
while not MyClass.ClassNameIs ('TObject') do...  
while MyClass <> TObject do...
```

ListParent列表框的with语句中使用的代码是ClassInfo范例的一部分，该范例用于显示父类列表和其他一些与VCL组件（基本上都是组件板Standard页上的组件）有关的信息。这些组件被手工添加到存储类的一个动态数组中，该数组声明如下：

```
private  
  ClassArray: array of TClass;
```

当程序启动时，该数组用于在一个列表框中显示所有的类名。选择列表框中的某个选项会触发详细信息及其基类的显示，该程序输出的如图3.8所示。

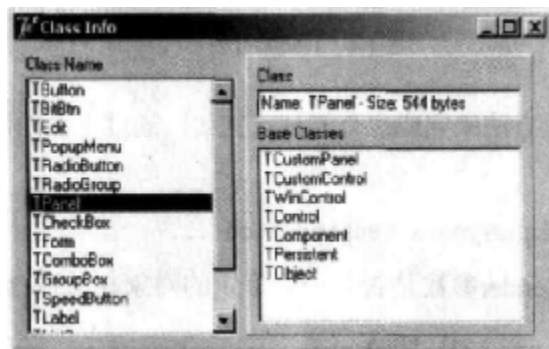


图3.8 ClassInfo示例的输出

说明：作为对该范例的进一步扩展，可以创建显示一个层次中各种组件的所有基类树。为此，我们建立了VclHierarchy向导，附录A中将进行讨论。

小结

在本章中，重点讨论了Delphi基于函数的运行时库的新特性。这里只提供了整个RTL的一个总结，而没有全面评述，因为这需要太多的篇幅。读者可以在笔者Web站点上的免费电子书中找到更多有关Delphi基础RTL函数的范例。

在下一章中，将从基于函数的RTL移植到基于类的RTL上，它是Delphi类库的核心。这里不想争论核心类是否对于VCL与CLX通用，如TObject，它实际上属于RTL或类库。本章已经介绍了定义在System、SysUtils中的内容，以及其他单元中所包含的函数与过程。下一章，重点将放在Classes单元与其他定义类的核心单元上。

结合前两章关于Delphi语言的内容，第4章将为讨论可视的与面向数据库的类（或组件）提供基础。查看各种库单元，我们将发现更多全局函数，它们都不属于核心RTL，但仍非常有用。

第4章 核心库类

在上一章中，我们可以看出Delphi包括大量的函数与过程，但Delphi真正的可视编程依赖于它所携带的巨型类库。Delphi的标准类库包含了数百个类以及数以千计的方法，而且它是如此之大，以至于不能在本书中提供一个详细的参考清单。从本章开始介绍该库的各个部分，并在随后的章节中继续介绍。

本章主要介绍库的核心类，以及一些标准的编程技术，如事件定义。我们将研究一些常用的类，如列表、字符串列表、集合与流。其中大部分时间将用来研究Classes单元的内容，但也将探讨库的其他核心单元。

Delphi类完全可以在代码中或在可视窗体设计器中使用。其中有些类是组件类，它们显示在组件面板上，而其他一些类的任务更通用些。术语类与组件在Delphi中几乎可以被看做是同义词。组件是Delphi应用程序的核心元素，当编写一个程序时，主要是选择一些组件，并定义它们的相互作用。这就是Delphi的可视编程。

在阅读本章之前，读者需要很好地理解Object Pascal编程语言，包括继承、属性、虚拟对象方法、类引用等，这些内容在本书第2章“Delphi编程语言”中已讨论过。

本章主要讨论以下内容：

- RTL包、CLX与VCL
- TPersistent与published
- TComponent基类及其属性
- 组件与所有权关系
- 事件
- 列表、容器类与集合
- 流
- RTL包的单元

RTL包、VCL与CLX

在版本5之前，Delphi的类库都被称做VCL，代表Visual Components Library（可视组件库）。在Kylix、Linux的Delphi版本中，引入了一个新的组件库，名为CLX（发音是“clicks”，代表Component Library for X-Platform或Cross Platform）。Delphi 6是第一个包含了VCL与CLX库的版本。对于可视组件，这两个类库是可供选择的。然而，两个库的核心类与数据库及Internet部分基本上是共享的。

VCL被看做是一独立的大型库，尽管程序员常常引用它的不同部分（组件、控件、非可视组件、数据集合、数据感应控件、Internet组件，等等）。CLX与VCL的区别表现在四个方面：BaseCLX、VisualCLX、DataCLX与NetCLX。但只有在VisualCLX中，该库才在两种平台上使用完全不同的方法，而其余代码都被移植到Linux中。在随后的小节中，将讨论这

两个库的各个部分，其余小节将讨论常用的核心类。

最近的Delphi版本强调了这种区别，因为库的核心非可视组件与类都属于新的RTL包，它可以同时被VCL与CLX使用。而且，在非可视应用程序（例如，网络服务器程序）中使用该包可以减小用于配置与装载到内存中的文件大小。

VCL的传统部分

Delphi程序员过去常通过名称引用VCL的不同部分（Borland原来在其文档中建议这样做），而且后来不同组的组件使用了相同的名称。从技术上讲，组件是TComponent类的子类，它是整个层次化体系结构的根类，如图4.1所示。实际上，TComponent类继承自TPersistent类，这两个类的作用将在下一节中解释。

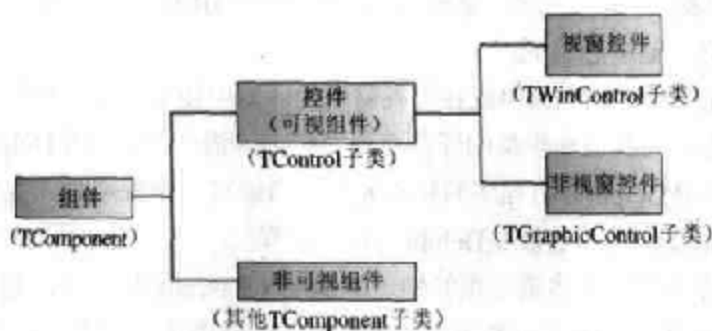


图4.1 VCL组件的图形表示

除了组件外，该库中还包括了直接继承自TObject与TPersistent的类。这些类在文档中总称为Objects，这更容易引起混乱。这些非组件类通常用于属性值或作为代码中使用的工具类，而不是继承自TComponent，因而这些类不能直接用于可视编程。

说明：更准确地讲，非组件类不能在组件板上使用，也不能直接拖入窗体中，但可以通过对象检验器对它们采取可视管理，作为其他属性的子属性或不同类型集合的项。所以即使是非组件类，在与Form Designer交互作用时也经常会用到。

组件类可以进一步划分为两大组：控件与非可视组件。控件组的所有类都派生自TControl。

控件 所有的控件类都派生自TControl类。控件在屏幕上有位置与大小，并且设计时在窗体中显示的位置与运行时的相同。控件有两个不同的子规格，基于窗口或图形化，但我们要在第5章“可视控件”中再详细讨论它们。

非可视组件 所有非可视组件都不是控件——其所有的类都派生自TComponent，而不是TControl。在设计时，非可视组件在窗体上显示为一个图标（其下面可选择性地带有标题）。在运行时，其中有些组件可以显示（例如，标准对话框），有些则不能显示（例如，数据库表格组件）。

提示：在Form Designer中可以将鼠标移到一个控件或组件上，通过一个Tooltip查看它的名称与类类型以及其他一些扩展信息。还可以使用一个环境选项，Show Component Captions，来查看非可视组件的名称。

这是对传统VCL的子划分，Delphi程序员都很熟悉。即使引入CLX与一些新的命名方案，传统的名称也可能保留下来并融入到Delphi程序员的行话中。

CLX的结构

Borland现在使用一套术语机制在Linux下称呼CLX库的不同部分，它是与Delphi稍有不同的命名结构。这是对与跨平台兼容的库的子划分，它与类库的层次结构相比，展示了更多的逻辑领域：

BaseCLX BaseCLX形成了类库的核心，其中包括最顶级的类（如TComponent），以及一些通用的工具类（包括列表、容器、集合、流）。与VCL的相应类比较，BaseCLX没有很大的变化，在Windows与Linux平台之间具有高度的可移植性。本章主要解释BaseCLX与常用的VCL核心类。

VisualCLX VisualCLX是可视组件的集合，通常被称做控件。库的这一部分更紧密地与操作系统相关：VisualCLX是在Qt库的顶部实现的，可以用于Windows与Linux。使用VisualCLX可以将应用程序的可视部分在Windows的Delphi上与Linux的Kylix上完全移植。然而，大多数VisualCLX组件都有相应的VCL控件，所以还可以轻易地将代码从一个库中移到另一个库中。我们将在第5章讨论VisualCLX与VCL的控件。

DataCLX DataCLX由库中所有与数据库相关的组件组成。实际上，DataCLX是Delphi 6与Kylix中所含新dbExpress数据库引擎的前端。如果可以将所有这些组件考虑为DataCLX的一部分，则只有dbExpress前端与IBX在Windows与Linux之间是可移植的。DataCLX还包含了ClientDataSet组件，现在被称做MyBase，以及其他相关的类。Delphi的数据访问组件将在本书第三部分中讨论。

NetCLX NetCLX包含了与Internet相关的组件，从WebBroker框架到HTML生成器组件，从Indy（Internet Direct）到Internet Express，从新的Site Express到XML支持。库的这一部分在Windows与Linux之间也具有高度的可移植性。Internet支持将在本书的第四部分中讨论。它的名字是Internet CLX的缩写，与Microsoft的.NET技术无关。

库的VCL专用部分

库的前面这些部分在Delphi与Kylix上都可以使用，但也有一些区别。然而，VCL还提供其他一些部分，出于某种原因，是Windows专用的：

- Delphi ActiveX（DAX）框架为COM、OLE Automation、ActiveX以及其他与COM相关的技术提供了支持。Delphi该部分的详细介绍见第12章“从COM到COM+”。
- Decision Cube组件提供了在线分析操作（OLAP）的支持，但与BDE有关，最近没有新的变化。本书将不讨论Decision Cube。

最后，默认的Delphi安装包含了一些第三方组件，如用于商务图形的TeeChart与用于报表和打印的RAVE，以及用于Internet开发的IntraWeb。这些组件将在本书中提到，但它们不是严格的VCL部分。RAVE和IntraWeb也可以用于Kylix。

TPersistent类

我们要讨论的第一个Delphi库中的核心类是TPersistent类，它是很奇怪的一个类：代码非常少，几乎没有任何直接的用途，但它为可视编程提供了基础。我们可以在程序清单4.1中看到该类的定义。

程序清单4.1 TPersistent类的定义，来自类单元

```
{SM+}
TPersistent = class(TObject)
private
    procedure AssignError(Source: TPersistent);
protected
    procedure AssignTo(Dest: TPersistent); virtual;
    procedure DefineProperties(Filer: TFile); virtual;
    function GetOwner: TPersistent; dynamic;
public
    destructor Destroy; override;
    procedure Assign(Source: TPersistent); virtual;
    function GetNamePath: string; dynamic;
end;
```

就像其名称暗示的那样，该类处理的是持续性，也就是说，将对象的值保存到文件中，以备今后用相同状态与相同数据重建对象。持续性是可视编程的一个要素。事实上（参见第1章“Delphi 7及其IDE”），在Delphi中，在设计时可以处理实际对象，而这些对象被保存为DFM文件；运行时，当特定的组件容器即窗体或数据模块建立时，这些对象被重新建立。

说明：我们所提到的所有关于DFM文件的内容也同样可以应用于XFM文件，这个文件格式被CLX应用程序所使用。它的格式是相同的。我们介绍扩展名的区别是因为Delphi要使用它来决定这个窗体是基于CLX/Qt还是基于VCL/Windows的。在Kylix环境中，每一个窗体是一个CLX/Qt窗体，无论它使用什么样的扩展名；因此，在Kylix环境中，XFM/DFM文件的扩展是无关紧要的。

流支持尽管没有嵌套在TPersistent类中，但由其他类提供，把TPersistent及其派生作为目标。换句话说，可以用Delphi默认的Streaming-Only对象（其类继承自TPersistent）实现“持续性”。这是因为类可以用特殊的选项{\$M+}来编译。该标记会为类的公布部分生成扩展的RTTI信息。

事实上，Delphi的流系统没有试图保存对象在内存中的数据，这非常复杂，因为有很多指向其他内存位置的指针，而且当对象被重新装载时没有什么意义。Delphi通过列出类的所有标记着特殊关键字published的属性值来保存对象。当一个属性引用另一个对象时，Delphi会根据其类型以及同主对象的关系来保存对象的名称或整个对象（使用相同的原理）。作为与其他方法的比较，请参考“对象流与代码生成”附加说明。

在TPersistent类的对象方法中，经常使用的只有一个Assign过程，它可以用于复制一个对象的实际值。在库中，该对象方法由很多非组件类以及少量的组件类来实现。实际上，大多数子类都会重新实现虚拟的、受保护的AssignTo对象方法，它由Assign的默认实现来调用。

其他对象方法包括DefineProperties, 用于定制流系统并添加额外的信息(伪属性); GetOwner与GetNamePath对象方法由集合与其他特殊类使用, 用于向对象检验器标识自己。

对象流与代码生成

Delphi与Kylix使用的方法与其他的可视开发工具及语言有所不同。例如, 在Java中, 在IDE中定义一个窗体是为了生成用于创建组件并且定义其属性的Java源代码。在一个监视器中设定这些属性会影响到代码本身。在C#中情况也很类似, 虽然在这种语言中, 属性的概念与在Delphi中的很接近。我们已经在Delphi中看到, 可以编写代码来生成组件, 而不必依靠流, 但是因为没有在IDE中的特定支持, 我们将不得不手工编写代码。

这两个方法中的每一个都具有各自的优缺点。当生成源代码的时候, 我们能够对将要发生什么情况以及创建和初始化的精确步骤进行有效的控制。Delphi中重载了这些对象以及它们的属性, 但是有些分配要到下一个修订期才能实现, 以避免引用尚未初始化的对象所造成的错误。这个操作更加复杂, 但是它是不可见的, 因此对于程序员来说相对要简单一些。

Java语言允许类似JBuilder的工具在每次发生变化的时候编译一个窗体类, 并且将它装入一个运行程序。在一个类似Delphi的编译系统中, 这个方法就变得很复杂(在设计时, Delphi使用该窗体的一个假的版本或者被称为代理, “proxy”, 而不是真正的窗体)。

Delphi的这个方法的一个优点是DFM文件可以被转移到不同的语言中, 而不必更改源代码。这也是Java提供窗体的XML持续性的原因。另一个不同之处是Delphi将组建的图像嵌入到DFM文件中, 而不是引用外部文件。这样做使部署变得简单(因为所有的东西都被包含在可执行文件之中), 但是也会使可执行文件变得比较大。

published关键字

除了public、protected与private访问指令外(我们在第2章“Delphi编程语言”中进行了讨论), 还可以使用第四个指令published。对于任何公布的字段、属性或对象方法, 编译器都会生成扩展的RTTI信息, 这样Delphi的运行时环境或程序就可以为一个公布的接口查询它的类了。例如, 每个Delphi组件都有一个公布的接口供IDE、特别是对象检验器来使用。在编写组件时, 公布字段的规范使用是很重要的。通常, 组件的公布部分中不包含字段或对象方法, 而只包含属性与事件。

当Delphi生成一个窗体或数据模块时, 它会将其组件与对象方法(事件处理程序)的定义放在其定义的第一部分中, 即public与private关键字之前。类初始部分的这些字段与对象方法都是公布的。当组件类的一个元素前没有添加特殊的关键字时, published是默认的关键字。

更准确地讲, 只有当类用\$M+编译器指令编译或派生自用\$M+编译的类时, published才是默认关键字。因为该指令用于TPersistent类中, 所以VCL的大多数类与所有组件类的关键字都是published。然而, Delphi中的非组件类(如TStream与TList)用\$M-编译, 默认关键字是public。

赋给任何事件的对象方法都应该是公布的对象方法, 而且对应着窗体中组件的字段应该是公布的, 以自动与DFM文件中描述的对象相连, 随着窗体一起建立(本章稍后将详细介绍这种情况及其引起的问题)。

访问公布的字段与对象方法

正如我们前面曾提到的,有三个不同的声明用于类的公布部分中:字段、对象方法与属性。在我们的代码中,常常会用到published关键字,就像使用public关键字那样,通过在代码中引用相关的标识符来完成。在某些特殊的场合,也可以在运行时通过名称来访问published对象。笔者将在“通过名称访问属性”一节中讨论属性,而这里首先介绍在运行时与字段及对象方法互操作的方式。事实上,TObject在这方面有三个对象方法:MethodAddress、MethodName与FieldAddress。

第一个函数,MethodAddress,会返回对象方法(在一个字符串中作为参数传递)被编译代码的内存地址(一种函数指针)。通过将该对象方法的地址赋给TMethod结构的Code字段并将对象赋给Data字段,可以获得一个完整的对象方法指针。在这里,为了调用对象方法需要将它转换成相应的对象方法指针类型。下面的代码段强调了该技术的要点:

```
var
    Method: TMethod;
    Evt: TNotifyEvent;
begin
    Method.Code := MethodAddress ( Button1Click' );
    Method.Data := Self;
    Evt := TNotifyEvent(Method);
    Evt (Sender); // call the method
end;
```

Delphi在装载DFM文件时,使用相同的代码指定一个事件处理程序,因为这些文件存储着用于处理事件的对象方法的名称,而组件则实际存储着对象方法指针。第二个对象方法,MethodName,执行相反的转换,返回给定内存地址的对象方法名称。该对象方法可以用于获得一个事件处理程序的名称——给它的值,当将组件流到一个DFM文件中时,Delphi也会这样做。

最后,TObject的FieldAddress对象方法会返回一个公布字段的内存位置,给出它的名称。该对象方法被Delphi用于将从DFM文件建立的组件与其同名称的所有者(例如,一个窗体)的字段联系起来。

注意,这三个对象方法在“正常”程序中很少使用,但在促使Delphi正常工作方面起到了关键作用,并与流系统密切相关。只有当编写非常动态的程序或特殊目的的向导或其他Delphi扩展时,才需要使用这些对象方法。

通过名称访问属性

对象检验器显示了对象公布属性的一个列表,甚至包括程序员编写的组件。为此,它要依赖于为公布属性生成的RTTI信息。使用一些高级技术,应用程序可以检索对象公布属性的列表并使用它们。

尽管这种功能不太为人所知,但在Delphi中简单地通过名称访问属性还是可能的,只需使用含有属性名称的字符串,然后取回它的值即可。访问属性的RTTI信息是通过一组未归档的子例程(属于TypInfo单元)来提供的。

警告：这些子例程在Delphi的老版本中总是不被归档，所以Borland保留了修改它们的权力。然而，从Delphi 1到Delphi 7，修改实际是非常有限的，而且只与支持的新特性有关，并且具有很好的向后兼容性。在Delphi 5中，Borland实际添加了很多有趣的内容和一些“Helper”的子例程，并且受到官方的鼓励（即使仍没有完全在帮助文件中归档，而只是在单元中提供了提示）。

这里没有探讨完整的TypeInfo单元，只是查看了通过名称访问属性所需的最少量的代码。在Delphi 5之前，需要使用GetPropInfo函数来检索指向一些内部属性信息的指针，然后向该指针应用一个访问函数，如GetStrProp，此外还必须检查属性的存在与类型。

现在，Delphi引入了一套新的TypeInfo例程，包括方便的GetPropValue，它返回带有属性值的一个变体，如果属性不存在，则产生一个异常。为了避免异常的发生，我们可以先调用IsPublishedProp，而且简单地向该函数传递对象与一个带有属性名称的字符串即可。进一步选择GetPropValue参数还允许我们选择集合类型属性的返回值格式（可以是一个字符串或者数值）。例如，可以调用

```
ShowMessage (GetPropValue (Button1, 'Caption'));
```

该调用的效果与调用ShowMessage相同，传递Button1.Caption作为参数。惟一真正的区别是该版本的代码速度慢一些，因为编译器通常会用一种更有效的方法决定对属性的正常访问。运行时访问的优点在于它非常灵活，如下面的RunProp范例中所示的那样。

该程序在一个列表框中可为窗体的每个组件显示任何类型属性的值。要查找的属性名称由一个编辑框提供，使得程序非常灵活。除了编辑框与列表框外，窗体还有一个按钮，用于生成输出，添加的其他一些组件只是为了测试它们的属性。当单击该按钮时，下列代码会被执行：

```
uses
    TypeInfo;

procedure TForm1.Button1Click(Sender: TObject);
var
    I: Integer;
    Value: Variant;
begin
    ListBox1.Clear;
    for I := 0 to ComponentCount - 1 do
    begin
        if IsPublishedProp (Components[I], Edit1.Text) then
        begin
            Value := GetPropValue (Components[I], Edit1.Text);
            ListBox1.Items.Add (Components[I].Name + '.' +
                Edit1.Text + ' = ' + string (Value));
        end
        else
            ListBox1.Items.Add ('No ' + Components[I].Name + '.' +
                Edit1.Text);
        end;
    end;
end;
```

如图4.2所示, 在编辑框中使用默认Caption值时, 可以看到单击Fill List按钮的效果。此外, 也可以测试其他任何属性名称。通过变体转换, 数值将被转换为字符串。对象(如Font属性的值)将被显示为内存地址。

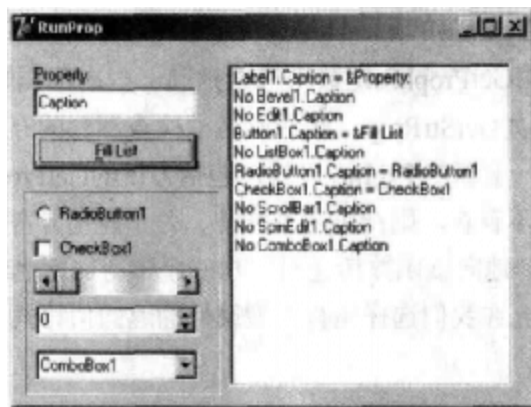


图4.2 RunProp示例的输出, 它在运行时根据名称访问属性

警告: 不要经常使用TypeInfo单元代替多态性与其他属性访问技术。首先要使用基于类的属性访问, 或在需要时使用安全的as类型转换, 并将RTTI访问属性保留作为一种快速访问的手段。使用TypeInfo技术会使代码的速度减慢且更复杂, 也更容易产生人为的错误, 事实上, 它省略了编译时的类型检查。

TComponent类

如果TPersistent类真的比其看起来要更加重要的话, 那么处于Delphi基础组件类库核心位置的关键类就要算是TComponent了, 它继承自TPersistent(与TObject)。TComponent类定义了组件的许多核心元素, 但不像人想像的那么复杂, 因为基类与语言已经提供了大多数实际需要的东西。

我们将不深入探讨TComponent类的所有细节, 因为有些细节对组件设计者来说比对组件用户更重要。我们将只讨论所有权(它解释了类的一些公共属性)与类的两个公布属性, Name与Tag。

所有权

TComponent类的一个核心特性是所有权的定义。当建立一个组件时, 它可以被赋给一个所有者组件, 同时也要负责消除这个组建。所以每个组件都有一个所有者, 并还可以作为其他组件的所有者。

类的一些公共对象方法与属性实际就是用于处理所有权的两方。下面这个程序清单摘自类声明(在VCL的Classes单元中):

```
type
  TComponent = class(TPersistent, IInterface, IInterfaceComponentReference)
  public
    constructor Create(AOwner: TComponent); virtual;
    procedure DestroyComponents;
```

```

function FindComponent(const AName: string): TComponent;
procedure InsertComponent(AComponent: TComponent);
procedure RemoveComponent(AComponent: TComponent);

property Components[Index: Integer]: TComponent read GetComponent;
property ComponentCount: Integer read GetComponentCount;
property ComponentIndex: Integer
    read GetComponentIndex write SetComponentIndex;
property Owner: TComponent read FOwner;

```

如果建立一个组件并且将它赋给某个所有者的话，那么它将被添加给组件列表（InsertComponent），并使用Components数组属性来访问。特定的组件有一个Owner，并通过ComponentIndex属性了解自己在所有者组件列表中的位置。最后，所有者的析构器将负责其所有对象的解除，此时，可调用DestroyComponents。还有一些受保护的成员方法，这只是为了给读者一个初步的认识。

需要强调的是，组件所有权可以解决应用程序的大部分内存管理问题，如果使用适当的话。如果建立了一个带有所有者的组件——使用IDE的可视设计器，这是默认操作方式——那么只需要注意不在需要它们时，解除这些组件容器即可，而且也无需牢记它们包含的组件。同时，我们需要使用一个窗体或者数据模块的所有者来创建组件，甚至是在代码之中。例如，删除一个窗体就会一次性删除它包含的所有组件，与必须记着单独释放每个对象相比，这真是大大地简化了。从更大的范围来看，窗体与数据模块通常都是由Application对象所有的，而VCL的关闭代码可将该对象取消，并且释放所有的组件容器。

组件数组

Components属性还可以用于访问另一个组件（例如窗体）所有者的组件。这对于编写通用代码来说非常方便（与直接使用特定组件相比），可一次对所有或很多组件进行操作。例如，使用下列代码向一个列表框添加窗体中所有组件的名称（这段代码实际是下一节中ChangeOwner范例的一部分）：

```

procedure TForm1.Button1Click(Sender: TObject);
var
    I: Integer;
begin
    ListBox1.Items.Clear;
    for I := 0 to ComponentCount - 1 do
        ListBox1.Items.Add (Components [I].Name);
end;

```

这段代码使用了ComponentCount属性（该属性用于存储当前窗体拥有的所有组件的总数）以及Components属性（该属性一般是窗体所有组件的列表）。当用户从该列表中访问某值时，得到的将是TComponent类型的值。因此，用户仅可以直接使用对所有组件都通用的属性，如Name属性。使用其他特定的属性时，必须进行相应的类型转换（as）。

说明: 在Delphi中, 有一些组件同时也是组件容器: GroupBox组件、Panel组件, 当然还有Form组件。当用户使用这些控件时, 可以向它们添加其他组件。在这种情况下, 组件容器就是组件的父组件(用Parent属性说明), 同时窗体是它们的所有者(用Owner属性说明)。用户可以使用窗体的Controls属性或组合框来遍历子控件, 而且可以使用窗体的Components属性遍历所拥有的组件, 而不论其父组件是什么。

利用Components属性总可以访问窗体的每个组件。如果用户需要对一个特定组件进行访问, 而不是用正寻找的组件的名称来比较所有名称, 那么, 就可以通过使用窗体的FindComponent对象方法, 让Delphi来完成这项工作。该对象方法会扫描Components数组, 寻找出与之匹配的名称。关于组件Name属性使用原则的更多信息将在随后的“名称属性”一节中进行介绍。

改变所有者 (Owner)

我们已经看到每个组件通常都有所有者。当在设计时(或从已有DFM文件中)建立一个组件时, 它的所有者总是它所在的窗体。当在运行时建立一个组件时, 所有者被作为参数传递给Create构造器。

所有者是一个只读属性, 所以不能改变它。在创建时所有者被设置并且在组件的生存期内不应改变。为了理解为什么在设计时既不应改变组件的所有者也不能改变它的名字, 可阅读下面的讨论。注意, 涉及的题目不简单, 因此如果你是刚刚才接触使用Delphi的, 那么应该稍后再回到这部分重新学习。

为了改变组件的所有者, 可以通过调用所有者自己的InsertComponent与RemoveComponent对象方法来影响该值(将当前组件作为参数传递)。使用这些对象方法, 可以改变组件的所有者。但是, 不能直接将它们应用到窗体的事件处理程序中, 如我们可以如下进行操作:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
    RemoveComponent (Button1);
    Form2.InsertComponent (Button1);
end;
```

这段代码将会产生一个内存访问冲突, 因为当调用RemoveComponent的时候, Delphi切断了这个组件与窗体字段(Button1)的连接, 并将它设为nil(我们将在“删除窗体字段”一节中详细讨论窗体字段)。解决的办法是编写下列过程:

```
procedure ChangeOwner (Component, NewOwner: TComponent);
begin
    Component.Owner.RemoveComponent (Component);
    NewOwner.InsertComponent (Component);
end;
```

这个方法(从ChangeOwner范例中提取的)改变了组件的所有者。它可以用更简单的代码进行调用, 以改变父组件; 这两个命令结合使用, 可使这个按钮完全移动到另外一个窗体上, 并且改变它的所有者:

```
procedure TForm1.ButtonChangeClick(Sender: TObject);
begin
```

```
if Assigned (Button1) then
begin
    // change parent
    Button1.Parent := Form2;
    // change owner
    ChangeOwner (Button1, Form2);
end;
end;
```

这个方法检查了Button1字段是否还与这个控件相关联，因为在移动组件的同时，Delphi将会把Button1设为nil。我们可以在图4.3中看到这段代码的效果。

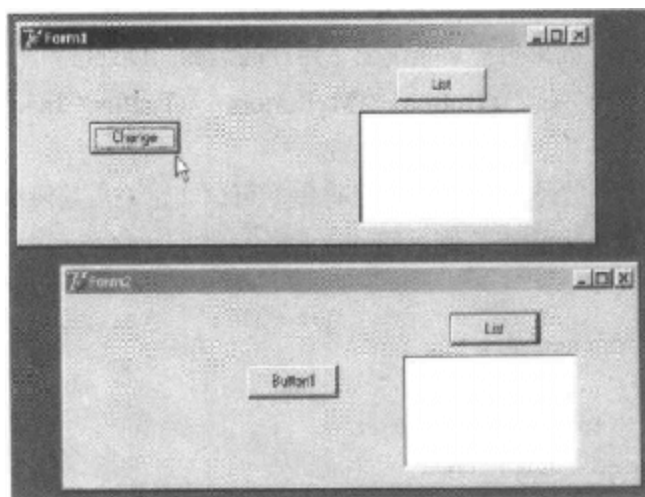


图4.3 在ChangeOwner示例中，单击Change按钮将把Button1组件移动到第二个窗体

为了演示Button1组件所有者的实际变化，我们已经在这两个窗体中添加了另外一个特性。List按钮使用一个列表框来列举每一个窗体所有者的名称，这里使用了在前面一节中用到的过程。在移动组件之前与之后单击这两个List按钮，将可以看到屏幕后发生的操作。作为一个最终的特性，Button1组件针对它的OnClick事件具有一个简单的事件处理程序，以显示这个所有者窗体的标题：

```
procedure TForm1.Button1Click(Sender: TObject);
begin
    ShowMessage ( 'My owner is ' + ((Sender as TButton).Owner as TForm).Caption);
end;
```

名称属性

Delphi中的每个组件都应有名称，在主组件中，这个名称必须是唯一的，这个主组件通常是用来放置其他组件的窗体。这意味着一个应用程序可以有两个不同的窗体，位于不同窗体内的两个组件的名称可以相同，尽管用户会尽力避免这种情况，因为这会造成一些混乱。一般应提倡在整个应用程序中保持组件名的唯一性。

为Name属性设置相应的值是很重要的：如果太长，则需要键入太多的代码；如果太短，可能会混淆不同的对象。组件名通常用组件类型作前缀，这样可增强代码的可读性，并且允

在Delphi的ObjectInspector的组合框内对组件按名称进行分组。有三个重要的元素与组件的Name属性有关：

- 第一，在设计时，Name属性的值在窗体类的声明中用于定义对象的名称，这一般是用户在引用此对象的代码中打算使用的名称。因此，Name属性的值必须是一个合法的Delphi语言标识符，它不能包括空格字符并且必须以字母开头，而不是数字。
- 第二，如果在改变控件的Caption属性或者Text属性之前设置它的Name属性，新名称会自动复制给标题。也就是说，如果控件的名称与标题相同，那么改变它的名称也会改变标题。
- 第三，Delphi利用组件名来建立与事件相关的对象方法的默认名称。如果有一个Button1组件，则OnClick事件处理方法的默认名称将是Button1Click，除非用户对其指定一个不同的名称。如果用户后来改变了组件的名称，Delphi将相应改变相关对象方法的名称。例如，如果改变Button1为MyButton，则Button1Click对象方法会自动变成MyButtonClick。

前面曾提到，对于一个含组件名称的字符串，可以通过调用其所有者的FindComponent来获得其实例，如果没有发现组件则返回nil。例如，我们可以编写如下的代码：

```
var
  Comp: TComponent;
begin
  Comp := FindComponent ('Button1');
  if Assigned (Comp) then
    with Comp as TButton do
      // some code...
```

说明：Delphi还包含了一个FindGlobalComponent函数，它用于查找顶级组件，基本上是窗体或数据模块，具有给定的名称。更准确地讲，FindGlobalComponent函数要调用一个或多个安装的函数，所以在理论上，可以修改该函数的工作方式。然而，因为FindGlobalComponent由流系统使用，所以我们不建议安装自己的替代函数。如果想用一种定制的方法在其他容器上查找组件，只需要用定制的名称编写一个新函数即可。

删除窗体字段

每当向窗体添加组件时，Delphi会向DFM文件添加其完整的描述，包括其所有属性，并向Pascal文件的窗体类声明中添加相应的元素。当建立窗体时，Delphi会装载DFM文件，并使用它重建所有组件而且还要设置它们的属性。然后它会使用与其Name属性相应的窗体元素连接新对象。这说明了在我们的代码中能够使用窗体字段来对相应的组建进行操作。

因此，拥有一个没有名称的组件是完全可能的。如果应用程序在运行时不处理组件或改动它，则可以从对象检验器中删除组件名称。例如，显示固定文本的静态标签，或某个菜单项，或菜单项分隔符。通过删除名称，可以从窗体类声明中删除相应的元素。这会减小窗体对象的大小（对象引用的大小只有四个字节），并由于不包括无用的字符串（组件名称），从而减小DFM文件的大小。减小DFM文件还意味着会减小最终的EXE文件，即使只是稍稍减小。

警告：如果删除组件名称，要确保窗体上使用的每个类至少保留一个有名称的组件，这样链接器才会将它链接到所需的代码中。例如，如果从窗体中删除引用TLabel组件的所有元素，则Delphi链接器会从可执行文件中删除TLabel类的实现。因为当系统在运行时装载窗体时，不能建立无名类的对象，所以会显示错误消息说明该类不能使用。正如我们在下一节中看到的，我们能够调用RegisterClass或者RegisterClasses例程来避免这样的错误。

还可以保留组件名并手工删除窗体类的相应元素。即使组件没有相应的窗体元素，也总会建立它，尽管使用它（例如，通过窗体的FindComponent对象方法）会有些困难。

隐藏窗体字段

很多OOP纯粹派人士抱怨Delphi没有真正地遵循封装的规则，因为窗体的所有组件都对应着公用元素，并可以从其他的窗体和单元访问。组件的字段被列在一个类声明的第一个未命名的部分，它具有一个默认的发布可视性。然而，Delphi这样做无非是为了让初学者尽快地使用Delphi的可视开发环境，让程序员可以遵循一种不同的方式并使用属性与对象方法在窗体上操作。然而问题是，同小组的另一个程序员可能会无意地绕过该方式，直接访问组件（如果它们留在published部分中）。解决方法（很多程序员并不了解）是将组件移到类声明的专用部分中。

作为范例，下面建立一个非常简单的窗体，其中有一个编辑框、一个按钮与一个列表框。当编辑框含有文本、而用户单击按钮时，文本会被添加到列表框中。当编辑框为空时，按钮将处于禁用状态。下面是HideComp范例的简单代码：

```
procedure TForm1.Button1Click(Sender: TObject);
begin
    ListBox1.Items.Add (Edit1.Text);
end;

procedure TForm1.Edit1Change(Sender: TObject);
begin
    Button1.Enabled := Length (Edit1.Text) <> 0;
end;
```

我们列举出这些方法只是为了展示，在一个窗体的代码中，通常要引用可用组件，并且定义它们的互操作。出于这个原因，要去掉这些与组件相关的字段看上去是不可能的。但是，能够隐藏它们，将它们从默认的发布部分移动到窗体类声明的专用部分：

```
TForm1 = class(TForm)
    procedure Button1Click(Sender: TObject);
    procedure Edit1Change(Sender: TObject);
    procedure FormCreate(Sender: TObject);
private
    Button1: TButton;
    Edit1: TEdit;
    ListBox1: TListBox;
end;
```

现在，如果运行程序，会遇到麻烦：窗体会正常装载，但因为专用元素没有初始化，故上面的事件将使用nil对象引用。Delphi通常使用从DFM文件建立的组件初始化窗体的

published元素。如果使用下列代码会出现什么情况呢？

```
procedure TForm1.FormCreate(Sender: TObject);
begin
    Button1 := FindComponent ('Button1') as TButton;
    Edit1 := FindComponent ('Edit1') as TEdit;
    ListBox1 := FindComponent ('ListBox1') as TListBox;
end;
```

它几乎可以运行，但它生成了一个系统错误，与前一节中讨论的问题相似。这一次，专用声明会使链接器与那些类的实现相连，但问题是，流系统需要知道类名称来确定类引用的位置，而这个类引用用来在装载DFM文件时建立组件。

最后需要一些注册代码，以便在运行时通知Delphi想要使用的组件类是存在的。这应该在建立窗体前完成，所以通常将下列代码放置在单元的初始化部分中：

```
initialization
    RegisterClasses ([TButton, TEdit, TListBox]);
```

现在的问题是，这些努力值得吗？我们获得了高度的封装特性，使窗体的组件与其他窗体（以及其他编写它们的程序员）隔离开来。必须说明的是，为每个窗体重复这些步骤是很繁琐的，所以最好是有一个向导，以便在Delphi中进行标准操作时，为程序员生成这些代码。然而，采用向导的办法也远不够完美，因为它不能够自动地处理变化，但是至少它是可以使用的。参考附录A可以获得更多的相关信息。对于大型项目来说，根据面向对象编程的原理，建议读者考虑该方法或相似的技术。

可定制的Tag属性

Tag属性是一个奇怪的属性，因为它根本就没什么效果，它只是一个附加内存地址，出现在每个组件类中，用于存储专用的值。它存储信息的种类及其使用的方式完全由用户来决定。

在不需要定义组件类的前提下，由一个附加内存地址来附带组件的信息通常显得非常有用。从技术上讲，Tag属性存储一个长整型值，例如，一个与对象相对应的数组或列表的入口值。使用类型转换，可以在Tag属性中存储指针、对象或任何四字节宽的值。这允许程序员使用其选项卡将组件与任何事物虚拟相连。在以后章节的多个例子中都将看到该属性的用法。

事件

现在，已经介绍完了TComponent类，Delphi还有一个元素，必须要介绍。事实上，Delphi组件是使用“PME”即属性、对象方法与事件进行编程的。现在，我们只对对象方法与属性进行了论述，而事件的内容还没有介绍。原因是事件没有新添语言特性，它只是一种标准的编程技术。事实上，事件从技术上讲就是属性，二者的唯一区别就是事件引用了对象方法（更精确地讲是一种对象方法指针类型），而不是其他类型的数据。

Delphi中的事件

当用户对组件进行一些操作时，如单击它，该组件就会产生一个事件。其他事件由系统产生，如响应一个对象方法调用或修改该组件的某个属性（甚至是其他组件等）。例如，如果用户将焦点设置在一个组件上，那么当前有焦点的组件就会失去它，这样就触发了相应的事件。

严格地讲，大多数Delphi事件是在收到相应的Windows消息后被触发的，尽管事件与消息不是一一对应的。Delphi事件的级别比Windows消息的级别要高，而且Delphi还另外提供了一些组件内部消息。

从理论角度看，事件是向一个组件或者控件发送请求的结果，并且该组件或者控件可以响应该消息。这样的话，为处理按钮的单击事件，就需要把TButton类划分为若干子类，并在这个新类内添加新的事件处理程序。

在实践中，创建新类是很复杂的，不是合理的方法。在Delphi中，组件的事件处理程序一般是控制组件所在窗体的对象方法，而不是组件自己的。换句话说，组件依赖于它的所有者，即窗体来处理它的事件。这项技术叫授权（delegation），它是Delphi基于组件模式的基础。也就是说，不用修改TButton类，除非想定义一个新类型的组件，简单自定义它的所有者来修改按钮的行为。

说明：如我们将要在下一节中介绍的，Delphi中的事件是基于方法指针的。这与Java非常不同，它使用某个系列事件的带有方法的监听器类。由这些监听器方法调用事件处理程序。C#和.NET使用类似的类授权思想，我们将在第24章“从Delphi的角度看微软.NET体系结构”中进行介绍。请注意“授权”这个术语与传统的Delphi用语是相同的，用来解释事件处理程序的思想。

对象方法指针

事件要依赖于Delphi语言的一个特定性质：对象方法指针。对象方法指针类型就像一种过程类型，但它引用的是对象方法。从技术上讲，对象方法指针类型就是具有隐含Self参数的过程类型。换句话说，一个过程类型的变量存储了调用函数的地址，倘若它有一个给定的参数集合。对象方法指针变量存储了两个地址：对象方法代码的地址与对象实例（数据）的地址。当使用该对象方法指针调用对象方法代码时，对象实例的地址将显示为对象方法体中的Self。

说明：这解释了Delphi通用TMethod类型的定义，一个含有Code字段与Data字段的记录。

对象方法指针类型的声明与过程类型相似，只是它在声明末尾有对象的关键字：

```
type
  IntProceduralType = procedure (Num: Integer);
  IntMethodPointerType = procedure (Num: Integer) of object;
```

当声明这样一个对象方法指针时，我们可以声明该类型的一个变量并将它赋给另一个对象的兼容对象方法，并且具有相同的标记（参数、返回类型、调用惯例等等）。

当为一个按钮添加OnClick事件处理程序时，Delphi会正确地完成这些工作。该按钮有一个对象方法指针类型的属性，名为OnClick，而且可以直接或间接地把它赋给另一个对象的对

象方法，如窗体。即使在另一个类中定义了它，当用户单击按钮时，也会执行该对象方法。接下来的一个Delphi用来定义事件处理程序的代码构架，以及一个窗体的相关方法：

```

type
  TNotifyEvent = procedure (Sender: TObject) of object;

  MyButton = class
    OnClick: TNotifyEvent;
  end;

  TForm1 = class (TForm)
    procedure Button1Click (Sender: TObject);
    Button1: MyButton;
  end;

var
  Form1: TForm1;

```

现在，在一个过程内，可以编写下面的代码：

```
MyButton.OnClick := Form1.Button1Click;
```

这个代码片断与VCL代码之间惟一真正的区别是，OnClick是一个属性名称，而且它引用的实际数据被称做FOnClick。在对象检验器的Events页中显示的事件，事实上就是一个对象方法指针类型的属性。举例来说，这表示可以在设计时动态地修改附加在一个组件上的事件处理程序，或者是在运行时建立一个新组件并将一个事件处理程序赋给它。

事件就是属性

我们曾经提到过，事件就是属性。这意味着，要处理组件的事件，可以赋予对象方法相应的事件属性。当在对象检验器中双击一个事件的时候，在所有者窗体中就添加了一个新的对象方法，并赋给组件相应的事件属性。

这就使多个事件可共享同一事件处理程序，或者在运行时改变事件处理程序。为了利用这一特性，用户就不需要知道有关语言的更多知识。事实上，当用户在对象检验器中选择某事件后，就可以按事件名右边的箭头按钮来查看“匹配”方法的下拉列表——拥有相同对象方法指针类型的对象方法列表。使用对象检验器，我们很容易为不同组件的相同事件选择相同的对象方法，或者为相同组件但不同或者兼容的事件选择相同的对象方法。

与我们在第2章中向TDate类添加属性一样，还可以向它添加一个事件。该事件将非常简单，叫做OnChange，可用来提醒组件的用户，日期值发生了变化。为了定义这个事件，只需定义与之相应的属性，并添加一些数据来存储事件所引用的实际对象方法指针即可。下列代码向TDate类中添加了一些新定义，可在DateEvt范例中找到它们：

```

type
  TDate = class
  private
    FOnChange: TNotifyEvent;
    ...
  protected

```

```

procedure DoChange; dynamic;
...
public
  property OnChange: TNotifyEvent
    read FOnChange write FOnChange;
...
end;

```

属性定义其实非常简单，该类的用户可以向属性赋一个新值，并因此向FOnChange专有字段赋一个新值。该类不会向这个FOnChange元素赋值，赋值的工作应由组件的用户来完成。当日期值改变时，TDate类只调用存储在FOnChange元素中的对象方法。当然，只有在事件属性已经被赋值的情况下，调用才会发生。这由下面的DoChange对象方法（声明为动态对象方法，因为它是传统的带有事件触发的对象方法）来完成：

```

procedure TDate.DoChange;
begin
  if Assigned (FOnChange) then
    FOnChange (Self);
end;

```

接下来，每当这些数值改变的时候，DoChange方法都会被调用，如下面的方法所示：

```

procedure TDate.SetValue (y, m, d: Integer);
begin
  fDate := EncodeDate (y, m, d);
  // fire the event
  DoChange;

```

如果我们检查这段使用了该类的程序，就可以大大地简化这些代码。首先，在窗体类中加入一个新的定制方法：

```

type
  TDateForm = class (TForm)
    ...
    procedure DateChange(Sender: TObject);

```

这个方法的代码只是简单地用当前TDate对象的Text属性值来更新那个标签：

```

procedure TDateForm.DateChange;
begin
  LabelDate.Caption := TheDay.Text;
end;

```

然后事件处理程序被安装到FormCreate方法中：

```

procedure TDateForm.FormCreate(Sender: TObject);
begin
  TheDay := TDate.Init (2003, 7, 4);
  LabelDate.Caption := TheDay.Text;
  // assign the event handler for future changes

```

```
TheDay.OnChange := DateChange;  
end;
```

这么做看上去反而增加了不少工作，难道我说该事件处理程序能够节省大量代码是在说谎吗？当然不是。现在，在添加了一些代码后，当改变了对象的数据时，我们就可以完全忘掉更新标签的操作了。例如，如下编写该按钮的OnClick事件处理程序：

```
procedure TDateForm.BtnIncreaseClick(Sender: TObject);  
begin  
    TheDay.Increase;  
end;
```

在其他事件处理程序中，也有同样简化的代码。一旦安装了该事件处理程序，就不必时刻惦记着更新标签了，更新标签在程序中是潜在的错误源。还要注意，必须在起始处编写一些代码，因为这不是装入Delphi的组件，而只是一个类。对于组件，用户只需在对象检验器中选择事件处理程序，并写一行更新标签的代码即可。

说明：这里，对定义事件只是进行了简单的介绍。对每一位Delphi程序员来说，大概理解这些特性是重要的。如果想创建新的带有复杂事件的组件，可参考第9章“编写Delphi组件”，届时将会有大量更为详细的信息介绍这些主题。

列表与容器类

处理组件或对象组通常是很重要的。除了使用标准数组与动态数组外，还有一些VCL类可以显示其他对象的列表。这些类可以分为三组：简单的列表、集合与容器。

列表和字符串列表

列表类可以由对象的通用列表TList以及两个字符串列表TStrings与TStringList来表示：

- TList定义指针列表，用于存储任意类的对象。TList远比一个动态数组要灵活得多，因为它可以通过增加新的项目而自动扩展。动态数组超越TList之处在于，动态数组允许为所含对象指定特定的类型，并在编译时执行相应的类型检查。
- TStrings是一个代表所有字符串列表形式的抽象类，而不论它们的存储器实现是什么。该类定义了字符串的抽象列表。因此，TStrings对象只用做可存储字符串自身的组件属性，如列表框。
- TStringList是TStrings的子类，用自己的存储器定义字符串列表。用户可以在程序中使用该类定义自己的字符串列表。

TStringList和TStrings对象都有与字符串相关的对象列表和字符串列表。这些类有大量不同的用法。例如，用户可以把它们用做相关对象字典，或者用于存储位图或其他在列表框中使用的元素。

字符串列表的两个类还有一些备用的对象方法，如用来向文本文件中存储数据的SaveToFile，以及用来从文本文件中提取内容的LoadFromFile。要在列表上进行循环，可在其索引上使用一个简单的for语句，就好像列表是数组一样。

所有这些列表都有许多对象方法和属性。程序员能使用数组符号（[和]）在列表上进行读取和改变元素的操作。此外Count属性也是典型的访问方法（如Add、Insert、Remove、Delete）、查找对象方法（如IndexOf）以及排序支持。TList类具有Assign对象方法，除了可拷贝源代码外，还能在两个列表上完成集合操作，包括逻辑与、逻辑或和逻辑异或操作。

为了用条目填充字符串列表并且稍后检查它是否被显示，可编写代码如下：

```
var
    sl: TStringList;
    idx: Integer;
begin
    sl := TStringList.Create;
    try
        sl.Add ('one');
        sl.Add ('two');
        sl.Add ('three');
        // later
        idx := sl.IndexOf ('two');
        if idx >= 0 then
            ShowMessage ('String found');
        finally
            sl.Free;
        end;
    end;
```

名称值匹配（以及Delphi 7扩展）

TStringList类还具有其他很好的特性：如支持名称值匹配。如果我们在一个列表中加入一个与“lastname=john”类似的字符串，就可以使用IndexOfName函数或者Values数组属性来搜索这个匹配，检查它是否存在。例如，通过调用Values ['lastname']来获得“John”这个值。

我们可以使用这个特性来建立更复杂的数据结构，例如字典；并且仍然从将一个对象附加到一个字符串的可能性中受益。这个数据结构可直接映射到初始化文件以及其他的通用格式中。

Delphi 7中对名称值匹配的支持做了进一步的扩展，这是通过允许定制分割符，而不是只允许使用等号来实现的。这里使用了新的NameValueSeparator属性。除此之外，使用新的ValueFromIndex属性可以直接访问位于指定位置的字符串的值；这样，我们就不必再手工使用笨重而且缓慢的表达式在整个字符串中搜索名称值了：

```
str := MyStringList.Values [MyStringList.Names [I]]; // old
str := MyStringList.ValueFromIndex [I]; // new
```

使用对象的列表 •

我们可以编写一个例子来集中讨论通用TList类的用法。当用户需要一个任意类型数据的列表时，一般可以声明一个TList对象并填入数据，在将其转换为相应的类型后访问其中

的数据。ListDemo范例显示了这一过程，还显示了该方法的缺陷。它的窗体含有一个专用变量，保存了一个日期列表：

```
private
    ListDate: TList;
```

这个列表对象是在窗体本身被创建时创建的：

```
procedure TForm1.FormCreate(Sender: TObject);
begin
    Randomize;
    ListDate := TList.Create;
end;
```

窗体上的一个按钮可为这个列表添加随机生成的日期值（当然，我们已经在这个工程中包括了具有上一章中构建的日期组件的单元）：

```
procedure TForm1.ButtonAddClick(Sender: TObject);
begin
    ListDate.Add (TDate.Create (1900 + Random (200), 1 + Random (12),
    1 + Random (30)));
end;
```

当我们从这个列表中提取表项的时候，不得不将它们设回正确的类型，就像在下面的方法中那样，这里是与List按钮进行连接的，我们可以在图4.4中看到它的效果：

```
procedure TForm1.ButtonListDateClick(Sender: TObject);
var
    I: Integer;
begin
    ListBox1.Clear;
    for I := 0 to ListDate.Count - 1 do
        Listbox1.Items.Add ((TObject(ListDate [I]) as TDate).Text);
end;
```

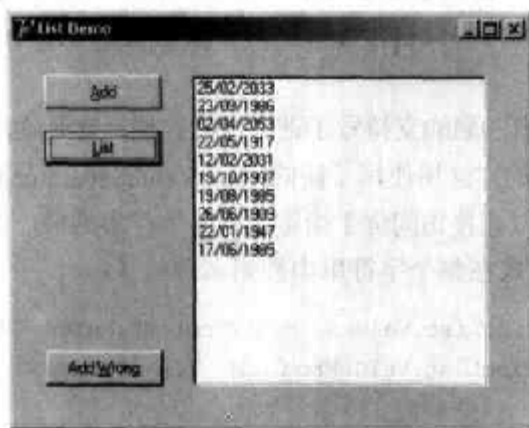


图4.4 ListDemo示例显示的日期列表

在代码的最后，在我们进行一个as改变之前，我们首先需要硬转换通过TList到TObject引用返回的指针。这种表达式将会导致一个不可用的类型异常，或者在指针并没有指向一个对象时，可能产生一个内存错误。

提示：如果没有可能在列表中包含任何非日期对象的数据，那么使用静态转换而不是as转换提取它将会更有效。但是，即使存在错误对象的可能性微乎其微，我们还是建议使用as操作。

为了演示发生错误的情况，我们添加了另外一个按钮，它调用ListDate.Add(Sender)向列表添加一个TButton对象。如果单击这个按钮，并且更新某个列表，那么将会得到一条错误信息。最后，请记住撤销一个对象列表时，应该首先撤销所有列表中的对象。ListDemo程序在窗体的FormDestroy方法中实现了这个任务：

```
procedure TForm1.FormDestroy(Sender: TObject);
var
  I: Integer;
begin
  for I := 0 to ListDate.Count - 1 do
    TObject(ListDate[I]).Free;
  ListDate.Free;
end;
```

集合

第二组，集合，只包含了两个VCL类，TCollection与TCollectionItem。TCollection定义了一个同类的对象列表，都归集合类所有。集合中的对象必须派生自TCollectionItem类。如果需要一个存储特定对象的集合，就必须建立TCollection的一个子类与TCollectionItem的一个匹配子类。

集合用于确定组件的属性值。处理用于存储对象的集合非常不同寻常，所以在这里将不讨论它们。我们将在第9章讨论集合的话题。

容器类

最近版本的Delphi引入了一个新的容器类系列，定义在Containers单元中。Delphi通过添加散列的相应列表来扩展这些类，如下面各小节描述的。这些类通过增加所有权的思想，并定义特殊的提取规则（模仿堆栈与队列）或排序能力，而扩展了TList类。

例如，TList与TObjectList新类之间的根本区别在于，后者被定义为TObject对象的列表，而不是指针的列表。然而，更重要的是，如果将对象列表的OwnsObjects属性设置为True，那么当使用一个对象代替另一个对象时，它会自动删除后者；当解除列表时，它会自动删除每个对象。下面列出所有的新容器类：

- TObjectList类（我们已经描述过）代表一个对象列表，最终属于列表本身。
- 派生类TComponentList代表一个组件列表，完全支持解除通知（当使用其属性连接两个组件时，也就是说，当一个组件是另一个组件的属性值时，这是一个重要的安全特性）。

- **TClassList**类是一个类引用列表。它继承自**TList**，不要求解除。因为在Delphi中我们不必解除类的引用。
- **TStack**类与**TObjectStack**类代表指针与对象的列表，从列表中取出元素时，只能从最后插入的元素开始。堆栈遵循LIFO规则（后进先出）。堆栈的典型对象方法是用于插入操作的**Push**、**Pop**（用于取出）以及**Peek**（用于预览第一项，而不删除它）方法，而且我们仍可以使用基类**TList**的所有对象方法。
- **TQueue**类与**TObjectQueue**类代表指针与对象的列表，我们总是可以从列表中删除插入的第一项（FIFO，先入先出）。这些类的对象方法与堆栈类相同，但工作方式不同。

警告：与**TObjectList**不同，**TObjectStack**与**TObjectQueue**不拥有插入的对象，当它被解除时，留在数据结构中的那些对象将不会被解除。我们只要取出（**Pop**）所有项，一旦使用完毕就解除它们，然后解除容器。

为了演示这些类的用法，下面将前面的ListDate范例改为新的Contain范例。首先，将ListDate变量的类型改为**TObjectList**。在FormCreate对象方法中，将列表的建立改为下列代码形式（它会激活列表的所有权）：

```
ListDate := TObjectList.Create (True);
```

这时，可以简化解除代码，因为将**Free**应用于列表将自动释放其存储的日期。

此外我们还向程序添加了一个堆栈与一个队列对象，并使用数字充填它们。该窗体的两个按钮之一用于在每个容器中显示数字列表，而另一个用于删除最后一项（显示在一个消息框中）：

```
procedure TForm1.btnQueueClick(Sender: TObject);
var
  I: Integer;
begin
  ListBox1.Clear;
  for I := 0 to Stack.Count - 1 do begin
    ListBox1.Items.Add (IntToStr (Integer (Queue.Peek)));
    Queue.Push(Queue.Pop);
  end;
  ShowMessage ('Removed: ' + IntToStr (Integer (Stack.Pop)));
end;
```

单击两个按钮，可以看到为每个容器调用**Pop**会返回最后一项。不同之处是，**TQueue**类在开始处插入元素，而**TStack**类在末尾插入它们。

哈希相关列表

在Delphi 6中，又将容器类的思想向前推进了一步，引入了一个新的列表集合，特别是**TBucketList**与**TObjectBucketList**。这两个列表是相关联的，这意味着它们有一个关键字与实际表项。关键字用于标识项目并查找它们。为了添加一个项目，可以调用**Add**对象方法，该对象方法有两个参数，关键字与实际数据。当使用**Find**对象方法时，传递关键字并取回数据。使用**Data**数组属性并将关键字作为参数来传递，也可以获得相同的效果。

这些列表基于一个哈希系统，作用是建立项目的一个内部数组——称做桶，每个桶内有一个列表元素的子列表。当添加一项时，其关键字值用于计算哈希值，确定加入项目的桶。在查找项目时，也需要计算哈希值，而且列表会马上获得包含项目的子列表，并在其中查找它。这使得插入与查找变得非常快，但只有当哈希算法在不同桶中平均分配项目，而且在数组中有足够的不同入口时才会这样。事实上，当很多元素都在相同的桶中时，查找速度很慢。

出于这个原因，当建立TObjectBucketList时，可以为列表指定表项的数量，使用构造器的参数，选择2到256之间的一个值。桶的值要通过将指针的第一个字节作为关键字传递，并与入口相应的数值做逻辑“与”运算来获得。

说明：我们没有发现该算法对于哈希系统非常有效，但如果用自己的函数替换它，将只能覆盖BucketFor虚拟函数，最终通过为BucketCount属性设置不同的值，改变数组中的入口数。

另一个有趣的特性，但不适用于列表，是ForEach对象方法，它允许针对列表中包含的每一项而执行一个给定的函数。可以向ForEach对象方法传递一个指向数据的指针和一个过程，该过程接收四个参数，包括定制的指针、每个关键字、列表的对象以及一个Boolean参数，可以将它设置为False来停止执行。换句话说，有两个信号：

type

```
TBucketProc = procedure (AInfo, AItem, AData: Pointer;  
    out AContinue: Boolean);
```

```
function TCustomBucketList.ForEach(AProc: TBucketProc;  
    AInfo: Pointer): Boolean;
```

说明：除了这些容器外，Delphi还包含了一个THashedStringList类，它继承自TStringList。该类与哈希列表没有直接的关系，甚至定义在不同的单元IniFile中。哈希字符串列表有两个相关的哈希表（TStringHash类型），每当字符串列表的内容发生变化时，它就会完全被刷新。所以该类只用于读取固定字符串的一个大型集合，而不适用于处理一个经常变化的字符串列表。另一方面，由TStringHash支持的类在一般情况下看起来非常有用，并且有一个很好的算法用于计算字符串的哈希值。

类型安全容器与列表

容器与列表存在一个问题：它们不是类型安全的，前面已经介绍过，在两个范例中向日期列表添加按钮对象时，会出现问题。为了确保列表中的数据是同类的，在插入数据前，可以检查取出的数据类型，但作为附加的安全措施，在取出数据时也应该检查数据类型。然而，添加运行时类型检查会减慢程序的运行速度，而且具有冒险性——在某些情况下程序员可能并未进行类型检查。

为了解决这两个问题，可以为给定的数据类型和形式建立具体的列表类，从已有TList或TObjectList类（或其他容器类）中改写代码。有两种方法可以实现该目的：

- 从列表类中派生新类并定制Add对象方法和访问对象方法（与Items属性有关）。这也是Borland用于容器类（都继承自TList类）的方法。

说明：Delphi容器类使用了静态覆盖来执行简单的类型转换（所需类型的参数与函数结果）。静态覆盖与多态性不同，通过TList变量使用容器类将不会调用容器的特殊函数。静态覆盖是一种简单而有效的技术，但它有一个非常重要的约束条件：派生类中的对象方法只能做简单的类型转

换, 因为不能保证派生的对象方法一定会被调用。列表可以使用父对象方法来访问与处理, 与派生的对象方法相似, 所以它们的实际操作必须是等同的。惟一区别在于派生对象方法中使用的类型, 它允许我们避免额外的类型转换。

- 建立包含一个TList对象的全新类, 并使用适当的类型检查将新类的对象方法与内部列表对应起来。该方法定义了一个包装类, 它“包装”了一个已有对象, 用以为其对象方法提供一种不同的或受限的访问方式(在我们的范例中, 是执行类型转换的)。

在DateList范例中实现了这两种解决方法, 该范例定义了TDate对象的列表。在下面的代码中, 读者会发现这两个类(基于继承的TDateListI类与包装类TDateListW)的声明:

```

type
  // inheritance-based
  TDateListI = class (TObjectList)
  protected
    procedure SetObject (Index: Integer; Item: TDate);
    function GetObject (Index: Integer): TDate;
  public
    function Add (Obj: TDate): Integer;
    procedure Insert (Index: Integer; Obj: TDate);
    property Objects [Index: Integer]: TDate
      read GetObject write SetObject; default;
  end;

  // wrapper based
  TDateListW = class (TObject)
  private
    FList: TObjectList;
    function GetObject (Index: Integer): TDate;
    procedure SetObject (Index: Integer; Obj: TDate);
    function GetCount: Integer;
  public
    constructor Create;
    destructor Destroy; override;
    function Add (Obj: TDate): Integer;
    function Remove (Obj: TDate): Integer;
    function IndexOf (Obj: TDate): Integer;
    property Count: Integer read GetCount;
    property Objects [Index: Integer]: TDate
      read GetObject write SetObject; default;
  end;

```

显然, 编写第一个类比较简单, 因为它没有多少对象方法, 而且它们只调用继承的对象方法。好在TDateListI对象可以被传递给等待TList类型的参数, 问题是通过一个通用的TList变量处理该列表实例的代码不会调用指定的对象方法, 因为它们不是虚拟的, 而且可能会添加给其他数据类型的列表对象。

如果决定不使用继承，最终需要编写大量代码，因为需要重新建立每一个原始TList对象方法（只是调用内部TList对象的对象方法）。缺点是，TDateListW类与TList类型不兼容，这限制了它的使用。它不能作为参数传递给需要TList类型的对象方法。

这两种方法提供了很好的类型检查手段。在建立了某个列表类的一个实例后，可以只添加适当类型的对象，而取出的对象自然是正确类型的。这可以由DateList范例来演示，该范例有一些按钮、一个组合框（让用户选择列表中显示的项）与一个列表框（显示列表的实际值）。程序通过向TDate对象的列表添加按钮来扩展列表，为了将不同类型的对象添加到TDateListI列表中，可以将列表转换成基类TList。当将列表作为参数传递给需要基类对象的对象方法时，偶然才会发生这种情况。相比而言，对于TDateListW列表，必须显式地将对象转换为TDate类（在插入它之前），通常程序员不应该像下面这样做：

```
procedure TForm1.ButtonAddButtonClick(Sender: TObject);
begin
    ListW.Add (TDate(TButton.Create {nil}));
    TList(ListI).Add (TButton.Create {nil});
    UpdateList;
end;
```

UpdateList调用会触发一个异常，此异常被直接显示在列表框中，这是因为在定制的列表类中使用了as类型转换。明智的程序员是不会编写以上代码的。总而言之，为指定类型编写定制的列表会使程序更稳固。编写包装列表而不是基于继承的类会更安全些，虽然需要更多的代码。

说明：除了针对不同类型重新编写包装形式的列表类外，还可以使用笔者编写的List Template Wizard，更详细的信息参见附录A。

流

Delphi类库的另一个核心部分是对流的支持，流包括文件管理、内存、套接字与其他按序列排列的信息源。流的理念是在读取数据的同时进行数据的传递，更像Pascal语言使用的传统读与写函数（详见“Essential Pascal”电子图书的第12章，参见附录C）。

TStream类

VCL定义了抽象的TStream类与多个子类。父类TStream只有一些属性，而且不会建立它的实例，但它提供了一组有趣的对象方法列表，当需要处理派生流类时，通常需要使用这个对象方法列表。

TStream类定义了两个属性，Size与Position。所有的流对象都有一个特定的大小（如果在流末尾写入信息的话，通常会增大），而且必须在流中为进行读或写信息操作指定一个位置。

读与写字节要根据正使用的实际类来决定，但在这两种情况下，都只需要知道流的大小，以及要读或写的的数据在流中的相对位置。事实上，这是使用流的一个优点。基本的接口决定我们是处理一个磁盘文件、一个二进制的大型对象（BLOB）字段，还是内存中的一段

长序列字节。

除了Size与Position属性外，TStream类还定义了一些重要的对象方法，其中大部分是虚拟的与抽象的（换句话说，TStream类没有定义这些对象方法的操作，所以，派生的类要负责实现它们）。其中一些对象方法只有在流中读或写组件时才有用（例如，ReadComponent与WriteComponent），而另一些则在其他场景中有用。在程序清单4.2中，可以看到TStream类的声明，提取自Classes单元。

程序清单4.2 TStream类定义的公共部分

```
TStream = class(TObject)
public
    // read and write a buffer
    function Read(var Buffer; Count: Longint): Longint; virtual; abstract;
    function Write(const Buffer; Count: Longint): Longint; virtual; abstract;
    procedure ReadBuffer(var Buffer; Count: Longint);
    procedure WriteBuffer(const Buffer; Count: Longint);

    // move to a specific position
    function Seek(Offset: Longint; Origin: Word): Longint; overload; virtual;
    function Seek(const Offset: Int64; Origin: TSeekOrigin): Int64;
        overload; virtual;

    // copy the stream
    function CopyFrom(Source: TStream; Count: Int64): Int64;

    // read or write a component
    function ReadComponent(Instance: TComponent): TComponent;
    function ReadComponentRes(Instance: TComponent): TComponent;
    procedure WriteComponent(Instance: TComponent);
    procedure WriteComponentRes(const ResName: string; Instance: TComponent);
    procedure WriteDescendent(Instance, Ancestor: TComponent);
    procedure WriteDescendentRes(
        const ResName: string; Instance, Ancestor: TComponent);
    procedure WriteResourceHeader(const ResName: string; out FixupInfo: Integer);
    procedure FixupResourceHeader(FixupInfo: Integer);
    procedure ReadResHeader;

    // properties
    property Position: Int64 read GetPosition write SetPosition;
    property Size: Int64 read GetSize write SetSize64;
end;
```

一个流的基本用途包括调用ReadBuffer和WriteBuffer方法，这是非常强大的功能，但是并不太容易使用。第一个参数是一个无类型的缓冲区，在里面我们可以传递存储或者载入的变量。例如，在一个文件中存储一个数字（二进制格式的）以及一个字符串时，使用如下代码：

```
var
    stream: TStream;
```

```
n: integer;  
str: string;  
begin  
  n := 10;  
  str := 'test string';  
  stream := TFileStream.Create ('c:\tmp\test', fmCreate);  
  stream.WriteBuffer (r, sizeof(integer));  
  stream.WriteBuffer (str[1], Length (str));  
  stream.Free;
```

另一个办法是使用特定的组件与流之间进行存储或者载入的操作。很多VCL类定义了一个LoadFromStream或者SaveToStream方法, 包括TStrings、TStringList、TBlobField、TMemoField、TIcon和TBitmap。

特定的流类

建立一个TStream实例没有什么意义, 因为该类是抽象的, 而且没有提供对数据保存的直接支持。但可以使用一个派生类从实际文件、BLOB字段、套接字或内存段中下载数据, 或将数据保存在这些媒介中。当希望对文件操作时, 可以使用TFileStream, 将文件名与一些文件访问选项传递给Create对象方法。使用TMemoryStream可处理内存中的流, 而不是实际文件。

一些单元定义了TStream派生的类, 在Classes单元中, 包括以下一些类:

- THandleStream定义的流, 用来处理由Windows文件句柄代表的磁盘文件。
- TFileStream定义的流, 用来处理由文件名代表的磁盘文件(本地或网络磁盘上的文件)。它继承自THandleStream。
- TCustomMemoryStream是存储在内存中的流的基类, 但不能直接使用。
- TMemoryStream定义的流, 用来处理内存中的一系列字节。它继承自TCustomMemoryStream。
- TStringStream提供的简单方法, 用来将流与内存中的字符串联系起来, 这样就可以用TStream接口访问字符串, 还可以与另一个流相互复制字符串。
- TResourceStream定义的流, 用来处理内存中的一系列字节, 并对与应用程序可执行文件相连的资源数据提供只读访问(这些资源数据的一个例子就是DFM)。它继承自TCustomMemoryStream。

在其他单元中定义的流类包括:

- TBlobStream定义的流, 用来为数据库BLOB字段提供简单的访问。除了BDE以外, 对于其他数据访问技术存在着相似的BLOB流, 包括TSQLBlobStream和TClientBlobStream。请注意, 对于BLOB字段, 每一个类型的数据集合使用一个特定的流类。所有这些类都派生自TMemoryStream。
- TOleStream定义的流, 用于通过OLE对象提供的流的接口进行读与写信息操作。
- TWInSocketStream为套接字连接提供流支持。

使用文件流

建立与使用文件流非常简单，只需建立派生自TStream的类型变量，并调用对象方法从文件中下载内容即可：

```
var
  S: TFileStream;
begin
  if OpenFileDialog1.Execute then
  begin
    S := TFileStream.Create (OpenDialog1.FileName, fmOpenRead);
    try
      Mem1.Lines.LoadFromStream (S);
    finally
      S.Free;
    end;
  end;
end;
```

在这段代码中，可以看到文件流的Create对象方法有两个参数：文件的名称与说明所需访问对象方法的标记。在这种情况下，由于想读取文件，所以使用了fmOpenRead标记（其他可以使用的标记见Delphi帮助文件）。

说明：在不同的模式中，最重要的是fmShareDenyWrite，当从共享文件中读取数据时将使用它；还有fmShareExclusive，当向共享文件写入数据时将使用它。在TFileStream.Create中有一个第三方的参数，叫做Rights，当访问模式是fmCreate时（即创建一个新的文件时），这个参数被用来向Linux文件系统传递文件访问许可。这个参数在Windows环境中被忽略。

流与其他文件访问技术相比的一大优点是，它们具有非常好的可互换性，所以可以对内存流进行操作，然后将它们保存到文件中，或执行相反的操作。这样可以改进一个文件程序的速度。下面一段代码是一个文件复制函数，它可为大家提供另一种使用流的思路：

```
procedure CopyFile (SourceName, TargetName: String);
var
  Stream1, Stream2: TFileStream;
begin
  Stream1 := TFileStream.Create (SourceName, fmOpenRead);
  try
    Stream2 := TFileStream.Create (TargetName, fmOpenWrite or fmCreate);
    try
      Stream2.CopyFrom (Stream1, Stream1.Size);
    finally
      Stream2.Free;
    end;
  finally
    Stream1.Free;
  end;
end;
```

流的另一个重要用处是直接处理数据库BLOB字段或其他大型字段。事实上，可以向流输出这样的数据，或从流中读取这样的数据，只需调用TBlobField类的SaveToStream与LoadFromStream对象方法即可。

说明：Delphi对于流的支持中增加了一个新的异常基类，EFileStreamError。它的构造器使用一个文件名作为错误报告的参数。这个类使流中的文件错误通告更加标准化和简单化。

TReader与TWriter类

VCL的流类自己没有对读或写数据提供很多支持。事实上，流类除了简单地读写数据块之外，没有实现更多的功能。如果想在流中装载或保存特殊的数据类型（并且不想执行大量的类型转换），可以使用TReader与TWriter类，它们派生自通用的TFileer类。

基本上，设计TReader与TWriter类的目的就是为了根据流数据（不只是序列数据）的类型简化其装载与保存操作。为此，TWriter在流中嵌入了特殊的符号，为每个对象的数据指定类型。反过来，TReader类从流中读取这些符号，建立相应的对象，然后使用来自流中的顺序数据初始化这些对象。

例如，利用下面的代码可以向流写入一个数值与一个字符串：

```
var
    stream: TStream;
    n: integer;
    str: string;
    w: TWriter;
begin
    n := 10;
    str := 'test string';
    stream := TFileStream.Create ('c:\tmp\test.txt', fmCreate);
    w := TWriter.Create (stream, 1024);
    w.WriteInteger (n);
    w.WriteString (str);
    w.Free;
    stream.Free;
```

这次，实际文件还将包含额外的符号字符，所以我们只需要使用一个TReader对象，就可以读回该文件。因此，使用TReader与TWriter通常被限定为组件流，它很少在通常的文件管理中使用。

流与持续性

在Delphi中，流对于持续性有着重要的作用。因此，TStream的很多对象方法都与保存一个组件及其子组件有关。例如，可以编写下列代码将窗体保存在一个流中：

```
stream.WriteComponent (Form1);
```

如果我们研究一下Delphi DFM文件的结构，会发现这其实就是一个包含了定制格式资源的资源文件。在该资源中，将发现窗体或数据模块，及其所含每个组件的组件信息。正如希望的那样，流类提供了两个对象方法，用于读与写这种定制的组件资源数据：WriteComponentRes用于存储数据，ReadComponentRes用于装载数据。

根据内存使用经验（不包括实际的DFM文件），利用WriteComponent通常更适合些。在建立了内存流并将当前窗体保存给它之后，问题就剩下如何显示它了。这可以通过将窗体的二进制表达转换为文本表达来实现。尽管Delphi IDE从版本5开始，就可以将DFM文件保存为文本格式，但编译代码内部使用的表达仍只能是二进制格式的。

窗体转换可以由IDE来实现，通常使用窗体设计器的View as Text命令以及其他方法。还有一个命令行工具，CONVERT.EXE，在Delphi的Bin目录中。在本书的代码里，获得转换的标准方法是调用VCL的专用对象方法。有四个函数可以用于转换WriteComponent对象方法获得的内部对象格式：

```
procedure ObjectBinaryToText(Input, Output: TStream); overload;
procedure ObjectBinaryToText(Input, Output: TStream;
    var OriginalFormat: TStreamOriginalFormat); overload;
procedure ObjectTextToBinary(Input, Output: TStream); overload;
procedure ObjectTextToBinary(Input, Output: TStream;
    var OriginalFormat: TStreamOriginalFormat); overload;
```

四个不同的函数具有相同的参数与名称，其中包含了名称Resource而不是Binary（如在ObjectResourceToText里那样），转换由WriteComponentRes获得的资源格式。最后一个对象方法，TestStreamFormat，说明DFM存储为二进制格式还是文本格式。

在FormToText程序中，我们使用ObjectBinaryToText方法将一个窗体的二进制定义复制到其他的流中，接下来我们将在一个备注中显示这个最终生成的流，正如读者在图4.5中看到的那样。下面是包含有这两个方法的程序代码：

```
procedure TFormText.btnCurrentClick(Sender: TObject);
var
    MemStr: TStream;
begin
    MemStr := TMemoryStream.Create;
    try
        MemStr.WriteComponent (Self);
        ConvertAndShow (MemStr);
    finally
        MemStr.Free
    end;
end;

procedure TFormText.ConvertAndShow (aStream: TStream);
var
    ConvStream: TStream;
begin
    aStream.Position := 0;
    ConvStream := TMemoryStream.Create;
    try
        ObjectBinaryToText (aStream, ConvStream);
        ConvStream.Position := 0;
        MemoOut.Lines.LoadFromStream (ConvStream);
    finally
```

```

ConvStream.Free
end;
end;

```



图4.5 窗体组件的文本描述，由FormToText示例显示

请注意，重复单击**Current Form Object**按钮，将获得越来越多的文本，演示的文本包含在流中。在单击多次之后，整个操作变得缓慢下来，看上去就像是程序被停滞了。在该代码中，我们开始看到使用流的一些灵活性——可以编写一个通用的过程，用于转换任何流。

说明：在向流中写入数据后，必须在进一步使用这个流之前，显式地回到开始处（或将Position属性设置为0），除非希望向流附加数据。

另一个按钮是**Panel Object**，用于显示指定组件（panel）的文本表示——通过向WriteComponent对象方法传递该组件。第三个按钮，**Form in Executable File**，执行了不同的操作。它不是将已有对象流入内存，而是将窗体的设计时表达——也就是其DFM文件——从嵌入可执行文件的相应资源中转载到TResourceStream对象中：

```

procedure TFormText.btnResourceClick(Sender: TObject);
var
    ResStr: TResourceStream;
begin
    ResStr := TResourceStream.Create(hInstance, 'TFORMTEXT', RT_RCDATA);
    try
        ConvertAndShow (ResStr);
    finally
        ResStr.Free
    end;
end;

```

通过依次单击按钮（或修改程序的窗体），可以对比保存在DFM文件中的窗体与当前运行时对象。

编写定制的流类

除了使用已有的流类之外, Delphi程序员还可以编写自己的流类, 并使用它们代替已有的流类。为此, 只需要指定通用的原数据块如何保存与装载即可, 且VCL可以使用新类, 无论在哪里调用它。我们不需要建立崭新的流类来处理新的媒介类型, 只需要定制已有的类。在这种情况下, 所有必须做的工作就是编写相应的读与写对象的方法。

作为一个范例, 我们建立了一个类, 用于对一个通用的文件流进行编码与解码。尽管该范例受到它所用的编码机制的局限, 但它完全可以与VCL融合并正常运行。新的流类只声明了两个核心读与写对象方法, 以及一个用来存储关键字的属性:

```
type
  TEncodedStream = class (TFileStream)
  private
    FKey: Char;
  public
    constructor Create(const FileName: string; Mode: Word);
    function Read(var Buffer; Count: Longint): Longint; override;
    function Write(const Buffer; Count: Longint): Longint; override;
    property Key: Char read FKey write FKey;
  end;
```

这个关键字的值被加到所保存文件的每一个字节中, 而当数据被读出的时候, 再从中减去。下面是一个完整的Write和Read方法, 它大量地使用了指针:

```
constructor TEncodedStream.Create( const FileName: string; Mode: Word);
begin
  inherited Create (FileName, Mode);
  FKey := 'A'; // default
end;

function TEncodedStream.Write(const Buffer; Count: Longint): Longint;
var
  pBuf, pEnc: PChar;
  I, EncVal: Integer;
begin
  // allocate memory for the encoded buffer
  GetMem (pEnc, Count);
  try
    // use the buffer as an array of characters
    pBuf := PChar (@Buffer);
    // for every character of the buffer
    for I := 0 to Count - 1 do
    begin
      // encode the value and store it
      EncVal := ( Ord (pBuf[I]) + Ord(Key) ) mod 256;
      pEnc [I] := Chr (EncVal);
    end;
    // write the encoded buffer to the file
```

```

Result := inherited Write (pEnc^, Count);
finally
    FreeMem (pEnc, Count);
end;
end;

function TEncodedStream.Read(var Buffer; Count: Longint): Longint;
var
    pBuf, pEnc: PChar;
    I, CountRead, EncVal: Integer;
begin
    // allocate memory for the encoded buffer
    GetMem (pEnc, Count);
    try
        // read the encoded buffer from the file
        CountRead := inherited Read (pEnc^, Count);
        // use the output buffer as a string
        pBuf := PChar (@Buffer);
        // for every character actually read
        for I := 0 to CountRead - 1 do
            begin
                // decode the value and store it
                EncVal := ( Ord (pEnc[I]) - Ord(Key) ) mod 256;
                pBuf [I] := Chr (EncVal);
            end;
        finally
            FreeMem (pEnc, Count);
        end;
        // return the number of characters read
        Result := CountRead;
    end;
end;

```

这段比较复杂的代码提供了相应的注释，应该有助于读者理解细节。

现在，已经有有了一个编码的流，可以试着在范例程序中使用它了，范例程序名为 EncDemo。该程序的窗体有两个Memo组件与三个按钮，如下图所示。



第一个按钮会将普通文本文件装载到第一个Memo中，第二个按钮会将第一个Memo的文本保存到一个编码的文件中，最后一个按钮会重新将编码的文件装载到第二个Memo中并解码。在范例中，对文件编码后，将它作为普通文件装载到第一个Memo中，这当然是无法阅读的。

因为有编码流类可以使用，所以该程序的代码与其他使用流的程序非常相似。例如，下面是用于保存编码文件的对象方法（我们可以将其代码与基于流的其他范例的代码进行比较）：

```
procedure TFormEncode.BtnSaveEncodedClick(Sender: TObject);
var
  EncStr: TEncodedStream;
begin
  if SaveDialog1.Execute then
  begin
    EncStr := TEncodedStream.Create(SaveDialog1.Filename, fmCreate);
    try
      Mem1.Lines.SaveToStream (EncStr);
    finally
      EncStr.Free;
    end;
  end;
end;
```

使用ZLib压缩流

Delphi 7中的一个新特性是提供对ZLib压缩库的官方支持（查阅www.gzip.org/zlib）。一个ZLib接口单元已经在Delphi的CD上存在了很长的时间，但是现在它才被包含进核心发布版本中，并且成为了VCL源（ZLib和ZLibConst单元）的一部分。除了提供一个到这个库（这是一个C语言库，可以直接将它嵌入Delphi程序，而不需要使用DLL）的接口外，Delphi 7还定义了一些帮助流类：TCompressStream和TDecompressStream。

作为一个使用这些类的范例，我们编写了一个叫做ZCompress的小程序，用来进行文件的压缩与解压缩。这个程序包括两个编辑框，在其中可以输入需要压缩的文件名，以及目标文件名，如果这个文件尚不存在的话，它会被自动建立。当单击Compress按钮的时候，源文件被转化为目标文件；当单击Decompress按钮的时候，将会把这个压缩文件返回给内存流。在这两种情况下，最终获得的压缩或者解压缩文件，将在一个备注中显示。图4.6显示了这个压缩文件（它就是当前程序窗体的源代码）。

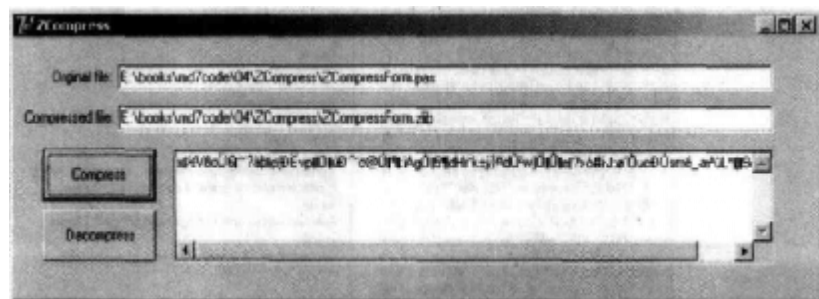


图4.6 ZCompress示例可以用ZLib库压缩文件

为了使这段程序可以重复使用，我们编写了两个函数，用于压缩与解压缩一个流到另一个流，下面是具体的代码：

```

procedure CompressStream (aSource, aTarget: TStream);
var
    comprStream: TCompressionStream;
begin
    comprStream := TCompressionStream.Create(
        clFastest, aTarget);
    try
        comprStream.CopyFrom(aSource, aSource.Size);
        comprStream.CompressionRate;
    finally
        comprStream.Free;
    end;
end;

procedure DecompressStream (aSource, aTarget: TStream) ;
var
    decompStream: TDecompressionStream;
    nRead: Integer;
    Buffer: array [0..1023] of Char;
begin
    decompStream := TDecompressionStream.Create(aSource);
    try
        // aStreamDest.CopyFrom (decompStream, size) doesn't work
        // properly as you don't know the size in advance,
        // so I've used a similar "manual" code
        repeat
            nRead := decompStream.Read(Buffer, 1024);
            aTarget.Write (Buffer, nRead);
        until nRead = 0;
    finally
        decompStream.Free;
    end;
end;

```

正如我们在代码注释中看到的，解压缩的操作要更加复杂一些，因为我们不能够使用 CopyFrom 方法：事先不能够知道目标流的大小。如果我们传递 0 给这个方法，它将尝试获得这个源流（TDecompressionStream）的大小。但是这个操作将导致一个异常，因为只能从头开始读取压缩和解压缩流，而不能搜索这个文件的结尾。

核心 VCL 与 BaseCLX 单元小结

本章大部分篇幅讨论了该库的 Classes 单元中的类。该单元非常重要，但它并不是该库惟一的核心单元（尽管不是很多）。在本节中，我们将简单介绍 Classes 单元及其内容以及其

他的核心库单元。

Classes单元

Classes单元处于VCL与CLX库的核心位置，尽管它与Delphi老版本相比有很多内部的变化，但对于一般用户来说没有多少新东西（大多数变化都与修改的IDE集成相关，面对的是专家级的组件编写者）。

下面是在Classes单元中发现的一个列表，每个Delphi程序员都应该花费一些时间来研究该单元：

- 很多枚举类型，标准的对象方法指针类型（包括TNotifyEvent），以及很多异常类。
- 核心库类，包括TPersistent与TComponent，以及其他一些较少直接使用的类。
- 列表类，包括TList、TThreadList（列表的线程安全版本）、TInterfaceList（内部使用的接口列表）、TCollection、TCollectionItem、TOwnedCollection（它是一个仅有一个所有者的集合）、TStrings与TStringList。
- 在上一节中讨论过的所有流类不再列出了。另外还有TFile、TReader与TWriter类，以及在内部用于DFM分析的TParser类。
- 工具类，如用于二进制处理的TBits与一些工具子例程（例如，点与矩形的构造器，字符串列表处理子例程，如LineStart与ExtractStrings）。还有很多注册类，通知系统存在着一些可以替换的组件、类、特殊工具函数等等。
- TDataModule类，窗体可以选择的一个简单的对象容器。数据模块只能包含非可视组件，通常用于数据库与Web应用程序中。

说明：在Delphi的早期版本中，TDataModule类定义在Forms单元中，从Delphi 6开始，它被移到了Classes单元中。这样做是为了从非可视应用程序（例如，Web服务器模块）中消除GUI类的代码耗费，以便更好地将非可移植的Windows代码从独立于OS的类（如TDataModule）中隔离出来。其他变化也与数据模块有关，例如，允许用多个数据模块建立Web应用程序。

- 新添的与接口有关的类，如TInterfacedPersistent，用于为接口提供进一步的支持。该类允许Delphi代码获得对TPersistent对象或任何派生实现接口的引用，它是对象检验器中对接口对象提供最新支持的核心元素（参见第9章“编写Delphi组件”中的例程）。
- 新添的TRecall类，用于维护一个对象的临时拷贝，这个类对于基于图形的资源特别有用。
- 新添的TClassFinder类，用于发现一个注册过的类，代替了FindClass对象方法。
- TThread类，它为多线程的应用程序提供了独立于操作系统的核心支持。

Classes单元中的新元素

在Delphi 7中，Classes单元只有很少的改动。本章已经提到了一些变化，如TStringList类中的名称-值对的扩展支持，此外，还有一对新全局函数，它们是AncestorIsValid和IsDefaultPropertyValue。

引入这两个函数是为了支持对象检验器中的非默认属性。它们不适合其他目的，我无法肯定能从它们当中获益——除非要保存组件和窗体的状态，或者编写自己的定制流机制。

其他核心单元

典型的Delphi程序员在使用Classes的时候，不会直接使用RTL包中的其他单元。这里是其他单元的列表：

- **TypeInfo**单元包括对公布属性的用于访问RTTI信息的支持，如在“通过名称访问属性”一节中介绍的那样。
- **SyncObjs**单元为线程同步化包含了一些通用的类。
- **ZLib**单元包括压缩和解压缩流，如同前面“压缩流和ZLib”小节中所讨论的那样。
- **ObjAuto**单元中包含的代码可通过名称来调用对象发布的方法——在一个变量数组中传递参数。此单元是SOAP和其他Delphi新技术增加的动态方法扩展支持的一部分。

当然，RTL包还包含了带有前一章中讨论过的函数与过程的单元，如**Math**、**SysUtils**、**Variants**、**VarUtils**、**StrUtils**、**DateUtils**等等。

小结

正如我们在本章中看到的，Delphi类库中有一些根类起到了很重要的作用，读者应该最大限度地掌握它们。有些程序员趋向于精通他们每天使用的组件，这也很重要，但如果不理解核心类（以及像所有权和流这样的思想），就不能轻松掌握Delphi的全部功能。

当然，在本书中，还需要讨论可视与数据库类，这将在下一章中进行。现在，已经介绍了Delphi的所有基本要素（语言、RTL、核心类），以后准备讨论如何利用该工具进行实际应用程序的开发。第5章将要介绍组件库可视部分的结构。

从下一章开始，完全致力于介绍各种组件的使用范例，特别是通过现代的用户界面建立应用程序以及有效地使用窗体。本书将从传统控件与菜单的高级使用开始，讨论行为结构，介绍TForm类，然后是工具栏、状态栏以及MDI应用程序。

第5章 可视控件

前几章已经介绍了Delphi的环境并综述了Delphi语言，以及组件库的基础元素，现在我们准备学习组件的使用和应用程序用户界面的开发。这是Delphi的实际工作内容。使用组件进行可视编程是这种开发环境的主要特性。

Delphi带有大量的备用组件。本书将不会详细描述每个组件并研究其每个属性与对象方法，如果读者需要这些信息，可以在帮助系统中查找它们。本章的目的是向读者展示如何使用一些Delphi预定义组件提供的高级特性建立应用程序，并讨论特殊的编程技术。

我们首先将对VCL与VisualCLX库，并介绍核心类（如TControl）。然后，试图列出所有可视组件，因为选择正确的基础控件通常是开发项目的一条捷径。

本章主要包括以下内容：

- VCL与VisualCLX
- TControl、TWinControl与TWidgetControl
- 标准组件综述
- 基础与高级菜单的构造
- 修改系统菜单
- 菜单中的图形与列表框
- 自绘制与式样

VCL与VisualCLX

正如在上一章中介绍的那样，Delphi有两个可视类库：CLX库与传统的VCL库。它们之间有很多区别，甚至表现在RTL与核心库类的使用上，以及专为Windows开发的程序之间，或是使用跨平台的态度上，但用户界面是其中的最大区别。

VCL的可视部分是Windows API的一个包装器，它包括本机Windows控件（如按钮与编辑框）、常用控件（如TreeViews与ListViews），以及一些与窗口的Windows概念有关的原有Delphi控件。还有一个TCanvas类，包装了基本的图形调用，通过它可以轻松地在窗口的表面上绘图。

VisualCLX，CLX的可视部分，是Qt（读做“cute”）库的一个包装器。它包含了本机Qt饰件的包装器，范围从基本控件到高级控件，与Windows自己的标准和常用控件非常相似。它还使用另一个类似的TCanvas类来提供绘图支持。Qt是一个C++类库，由Trolltech（www.trolltech.com）开发，这是一家与Borland关系密切的挪威公司。

在Linux上，Qt是一个实际的标准用户界面库，是KDE桌面环境的基础。在Windows上，Qt为本机API的使用提供了一种选择方法。事实上，与VCL不同，VCL为本机控件提供了一个包装器，而Qt为这些控件提供了另一种实现途径。即使它们都是基于Windows的窗口，Qt按钮不是Windows的BUTTON类控件（可以运行WinSight32来看看）。这允许程序真正地具

有可移植性,因为操作系统没有产生隐含的区别。此外使用Qt无需添加额外的层次,CLX在Qt的顶层,Qt在Windows本机控件的顶层,一共三层,但事实上在每个解决方案中只有两层(CLX在Qt的顶层,VCL控件在Windows的顶层)。

说明:在Windows上配置Qt应用程序实际上是在完成Qt库自己的配置。通常在Linux平台上,需要获得Qt库的许可,但还是可以用接口库配置。同时,Linux下Borland的CLX与Qt的特殊版本相联系(Borland已经在Kylix 3中专门做了修订),所以它可能还需要发布。同专业应用程序(与开放源代码的项目不同)一起配置Qt库通常表明要向Trolltech购买许可证。可以使用Delphi或Kylix建立Qt应用程序,因为Borland已经为我们向Trolltech购买了许可证。然而,必须使用CLX类包装Qt,如果仅使用Qt类(没有CLX),那么在使用时仍需要获得Qt的许可,即使在Delphi或Kylix中。

从技术上讲,用VCL建立的Windows应用程序与用VisualCLX开发的、可移植的Qt程序之间实质上有很大区别。可以说,在低级别,Windows使用API函数调用和消息与控件联系,而Qt使用类对象方法和直接的对象方法回调,而且没有内部消息。从技术上讲,Qt类提供了高级的面向对象结构,而Windows API仍固守它的C遗产和一个1985年(Windows颁布的时间)发布的、基于消息的系统。VCL在低级的API顶层提供了一种面向对象的抽象,而VisualCLX将已经是高级的接口重新对应到更常见的类库上(参见接下来的“从Qt到CLX”附加说明,以了解Qt的结构)。

说明:Microsoft显然开始放弃传统的低级Windows API,而转向固有的高级类库,它属于.Net体系结构的一部分。读者可以在本书的第四部分看到更详细的内容。

如果Windows API的基础结构和Qt有关,那么由Borland建立的两个类库(VCL和CLX)会抹平大部分的差异,使Delphi和Kylix应用程序的代码非常相似。能在全新的平台顶部有一个熟悉的类库,这对于在Linux上使用VisualCXL的Delphi程序员是个好事。从外表看,按钮对于两个库来说都是TButton类的对象,而且它或多或少具有相同的方法、属性与事件的集合。在很多情况下,可以用很短的时间为新类库重新编译已有的程序,如果它们并未直接调用低级的API(依靠平台的ADO或COM,或遗存的BDE)。

从Qt到CLX

Qt是一个C++类库,包含有一个完整的窗口小部件集合,与Windows核心组件(按钮、列表框等)类似,也与大多数普通控件(例如TreeView和ListView控件)类似。因为Qt是一个C++库,故它不能够直接包含在Delphi语言代码中。访问Qt API是通过一个绑定层来实现的,定义在Qt.pas单元内。

这个绑定层包括很长的一个Qt类的包装器列表,这些类带有一个后缀H,例如,Qt QPainter类成为绑定层的QPainterH类型。Qt.pas单元还包括了这些类的所有公共方法的列表,而且这些公共方法已被转化为标准函数(不是类方法),作为它们应用的第一个参数。惟一的例外是类的构造器,它被转化为函数,可以返回类的新实例。

请注意,使用映射层中至少一个以上的类对于Delphi(和Kylix)所带的Qt许可证来说是必需的。Qt对于非商业应用是免费的,在X Window下使用(被称为Qt Free版本),但是我们必须向Trolltech付费来开发一个商用应用程序。当我们购买Delphi的时候,Qt的许可证费用已经由Borland支付了,但是我们必须通过CLX来使用Qt(即使在CLX应用中允许

对Qt的底层调用)。我们不能直接使用CLX.pas单元, 并且不包括QForms单元(这是命令式的)。Borland通过从Qt接口中忽略QFormH和QApplicationH构造器而强制这个限制。

在大多数关于Delphi编程的书中, 我们仅使用CLX对象和方法。了解这一点很重要, 如果可能, 我们就可以直接使用Qt特性, 或者进行底层调用, 以避免CLX中的缺陷。Qt文档并不包括在Delphi的帮助文件中, 我们可以在Trolltech的网站(www.trolltech.com)中找到它, 包括HTML和PDF格式的文件。

Delphi 6双重的库支持

Delphi在设计时与运行时全面支持这两个库。当开始开发一个新应用程序时, 可以使用File►New Application命令建立一个基于VCL的新程序, 使用File►New CLX Application命令建立一个基于CLX的新程序。在给出其中一个命令之后, Delphi的IDE将建立一个VCL或CLX设计时窗体并更新组件面板, 这样它将只显示与所选应用程序类型兼容的可视组件(如图5.1所示)。事实上, 不能将VCL按钮放入CLX窗体中, 而且甚至不能在一个可执行文件内混合库的形式。换句话说, 每个应用程序的用户界面都必须完全使用这两个库中的一个来建立, (抛开技术上的因素)这对我们有很大的实际意义。

如果读者没有这样做, 那么先试着建立一个CLX应用程序, 研究可以使用的控件, 并尝试着使用它们。你会发现在组件的使用上没有太大的区别, 而且如果已经使用Delphi一段时间了, 可能会立刻适应CLX。



图5.1 基于CLX(上半部)和VCL(下半部)开发应用时, 组件面板中的前三个页面

相同的类, 不同的单元

在CLX与VCL代码之间, 源代码兼容性的一个基础是两个库中相似的类具有完全相同的类名称。每个库都有一个叫TButton的类, 代表一个按钮; 其对象方法与属性也非常相似, 以至于下列代码适用于两个库:

```
with TButton.Create (Self) do
begin
  SetBounds (20, 20, 80, 35);
  Caption := 'New';
  Parent := Self;
end;
```

两个TButton类具有相同的名称，而且这是可能的，因为它们保存在两个不同的单元（StdCtrls与QStdCtrls）中。当然，设计时的组件面板上不能同时有两个组件，因为Delphi IDE只能用惟一的名称注册组件。整个VisualCLX库由VCL单元对应的单元来定义，但要使用Q作为前缀——所以就有了QForms单元、QDialogs单元、QGraphics单元，等等。还有一些奇特的单元，如QStyle单元，但在VCL中没有对应的单元，因为在Windows API中没有对应的Qt的特征。

注意，无法使用编译设置或其他隐含技术来区别这两个库，关键是在要明确代码中引用的单元集合。注意，这些引用必须是一致的，因为不能在一个窗体中混合两种库的可视控件，甚至在一个程序中都不行。

DFM与XFM

在设计时建立一个窗体时，可将其保存到一个窗体定义文件中。传统的VCL应用程序使用DFM扩展名，代表Delphi窗体模块的意思。CLX应用程序使用XFM扩展名，代表跨平台窗体模块（Cross-platform Form Modules，X表示跨平台）的意思。窗体模块是流化窗体及其组件的结果，而且两个库共享流代码，所以它们会生成很相似的效果。DFM和XFM文件的格式是一样的，是基于文本或二进制的。

之所以具有两个不同扩展名的原因不是由于内部编译器技巧所致或存在不兼容的格式，它只是为了向程序员说明并向IDE说明应该在该定义中查找的组件类型（因为该说明没有包含在文件自己的内部）。

如果想将DFM文件转换成XFM文件，只需重新命名文件即可。然而，为了发现属性、事件和可使组件之间的一些区别而重新打开另一个库的窗体定义，可能会引出一些警告。

提示：显然，Delphi的IDE只通过查看窗体模块的扩展名来选择实际的库，它省略了uses语句中的引用。因此，如果我们计划使用CLX，则应改变扩展名。在Kylix上，不同的扩展名是没有用处的，因为任何窗体在IDE中都作为CLX窗体打开，而不考虑扩展名。在Linux上，只有基于Qt的CLX库，它既跨平台，也是固有的库。

作为一个范例，我建立了两个相同的简单应用程序，LibComp与QLibComp，它们只含有少数一些组件与一个事件处理程序。程序清单5.1提供了这两个应用程序的文本窗体定义，在Delphi IDE中，在选择了CLX或VCL应用程序之后，使用相同的步骤建立。我们用黑体标出了区别：可以看到，两程序的差别非常少，大多数与窗体及其字体有关。OldCreateOrder是一个继承属性，用于同Delphi 3与更老的代码兼容，标准颜色具有不同的名称，而且CLX保存了滚动条的范围。

程序清单5.1 XFM文件（左）和等价的DFM文件（右）

object Form1: TForm1	object Form1: TForm1
Left = 192	Left = 192
Top = 107	Top = 107
Width = 350	Width = 350
Height = 210	Height = 210
Caption = 'QLibComp'	Caption = 'LibComp'

<pre> Color = clBackground VertScrollBar.Range = 161 HorzScrollBar.Range = 297 TextHeight = 13 TextWidth = 6 PixelsPerInch = 96 object Button1: TButton Left = 56 Top = 64 Width = 75 Height = 25 Caption = 'Add' TabOrder = 0 OnClick = Button1Click end object Edit1: TEdit Left = 40 Top = 32 Width = 105 Height = 21 TabOrder = 1 Text = 'my name' end object ListBox1: TListBox Left = 176 Top = 32 Width = 121 Height = 129 Rows = 3 Items.Strings = ('marco' 'john' 'helen') TabOrder = 2 end end </pre>	<pre> Color = clBtnFace Font.Charset = DEFAULT_CHARSET Font.Color = clWindowText Font.Height = -11 Font.Name = 'MS Sans Serif' Font.Style = [] TextHeight = 13 OldCreateOrder = False PixelsPerInch = 96 object Button1: TButton Left = 56 Top = 64 Width = 75 Height = 25 Caption = 'Add' TabOrder = 0 OnClick = Button1Click end object Edit1: TEdit Left = 40 Top = 32 Width = 105 Height = 21 TabOrder = 1 Text = 'my name' end object ListBox1: TListBox Left = 176 Top = 32 Width = 121 Height = 129 ItemHeight = 13 Items.Strings = ('marco' 'john' 'helen') TabOrder = 2 end end </pre>
--	---

uses语句

通过研究VCL或CLX应用程序的源代码，可以发现相关的差别很少，因为它们只是引用了uses语句。CLX应用程序的窗体具有下列初始代码：

```
unit QLibCompForm;  
interface  
uses  
    SysUtils, Types, Classes, QGraphics, QControls, QForms, QDialogs, QStdCtrls;
```

VCL程序的窗体拥有传统的uses语句:

```
unit LibCompForm;  
interface  
uses  
    Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,  
    Dialogs, StdCtrls;
```

该类与惟一事件处理程序的代码完全一样。当然,经典的编译器指令{\$R *.dfm}在程序的CLX版本中被{\$R *.xfm}代替了。

禁止双重的库帮助支持

当在编辑器中按下F1键查询Delphi库中的某个子例程、类、或对象方法的帮助文件时,通常要在相同特性的VCL与CLX声明之间做出选择。并需要选择相关的帮助页,它可能会引起程序员的烦恼(特别是因为两个页经常是相同的)。

如果不考虑CLX,只打算使用VCL(或相反),则可以通过下列步骤禁止相互替换:选择Help►Customize命令,在Contents、Index与Link的名称中删除与CLX有关的信息,并保存项目。再次重新启动Delphi IDE,Help引擎将不再询问有关CLX的信息。当然,一旦决定开始使用CXL,不要忘记再添加上这些帮助文件。同样,卸载全部CLX包可以减少内存占用和Delphi IDE的装载时间。

选择一个可视库

因为在Delphi中有两个不同的用户界面库,所以必须为每个可视应用程序选择一个库。必须综合判断多个标准,得出正确决定,这通常是不容易的。

第一个标准是可移植性。如果在Windows与Linux上运行程序,则显示相同的用户界面对我们非常重要,使用CLX可能会使操作更简单些,并可以保留一个使用限定IFDEF的源代码文件。如果要使用Linux作为主要平台,可采取同样方法。但是,如果大部分用户使用Windows,只想通过Linux版本扩展自己的程序,则可以考虑保留一个双重的VCL/CLX系统。这样做可能会产生两套不同的源代码文件或太多的IFDEF。

另一个标准就是感觉。通过在Windows上使用CLX,有些控件的行为将与用户希望的稍有不同——至少对于专家用户而言。对于简单的用户界面(编辑、按钮、网格),这可能没有太多问题,但如果使用的是TreeView与ListView控件,区别会非常明显。另一方面,使用CLX将可以让用户根据他们自己的感觉进行选择,亦即不同于基本的Windows显示,而且可跨平台使用它。使用Windows平台意味着喜欢图形界面的人能够选择该风格。这在Linux中是很灵活的,而在Windows中就没有这种感觉。

使用本机控件还意味着,只要得到一个新版本的Windows操作系统,应用程序就将(可能)适应它。这对用户是好事,但如果不兼容的话,可能会使我们非常头痛。Microsoft的常用控件库在过去几年中出现的差别已经给Windows程序员,包括Delphi程序员,带来了很多麻烦。

另一个标准是配置：如果使用CLX，就必须让Windows程序携带Qt库，它在Windows系统上不常使用。

笔者曾做了一个小测试，想看看VCL与CLX应用程序的运行速度有没有差异。为此，我建立了一千个组件，在屏幕上显示它们，结果发现速度差异很小，基于VCL的程序运行速度稍快些。读者可以使用应用程序LibSpeed与QLibSpeed进行测试。

另一个以CLX代替VCL的重要标准是Unicode支持。CLX的控件默认支持Unicode（在Win9x平台上也支持它，而Microsoft不支持）。然而，即使Windows支持Unicode，但VCL在Windows版本中也不支持它，这使得为不同国家建立VCL应用程序很困难。

在Linux上运行它

对于程序员与用户来说，选择库的真正问题在于确定Linux或Unicode的重要性。注意这一点很重要：如果建立一个CLX应用程序，那么使用Kylix可以在不改动的前提下，重新编译它而生成一个本机Linux应用程序，除非已经编写了Windows API代码，这时还需要进行编译。

作为一个范例，这里重新编译了前面介绍过的QLibComp程序，图5.2显示了它的运行情况，其中还可以看到Kylix IDE在KDE上的运行情况。

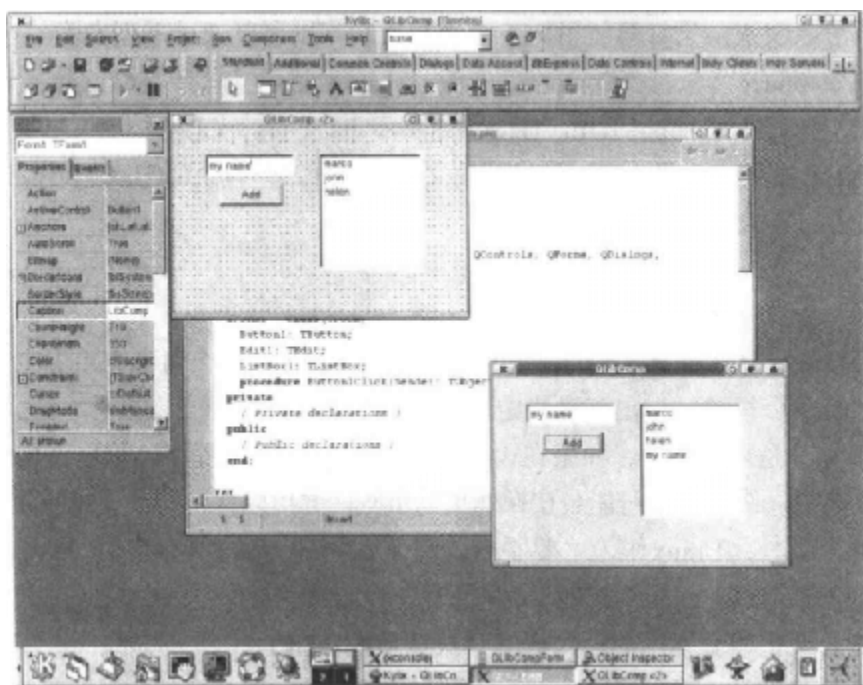


图5.2 用CLX编写的应用程序可以直接在Kylix环境中进行重编译（在背景显示）

库的条件编译

如果想保留一个源代码文件，但在Windows上要用VCL编译，在Linux上要用CLX编译，那么在使用条件编译的情况下，可以使用平台专用的符号（如\$IFDEF LINUX）来识别两种情况。但如果想在Windows上编译两个库的部分代码，该如何呢？

可以定义一个自己的符号，并使用条件编译，或检测只保留在VCL或CLX中的标识符，如：

```
{ $IF Declared(QForms) }
...CLX-specific code
{ $IFEND }
```

转换已有的应用程序

除了开始编写新的CLX应用程序外，我们还想将一些已有的VCL应用程序转换成新的类库。有一系列操作必须要执行，Delphi IDE中没有提供专门的帮助：

- 必须将DFM文件改名为XFM，并将所有的{\$R *.DFM}语句更新为{\$R *.XFM}。
- 必须更新程序中的所有uses语句（在单元与项目文件中），引用CLX单元代替VCL单元。注意，即使忽略任何一点，在运行应用程序时也会遇到麻烦。

提示：如果CLX应用程序中包含了对VCL单元的引用，为了防止编译它，可以将VCL单元移到lib下的不同目录中，避免将该文件夹包含在查找路径中。这样，最终对VCL单元的引用将引起一个“Unit not Found”错误。

表5.1对比了可视VCL与CLX单元的名称，不包括数据库部分与一些很少引用的单元。

表5.1 对应的VCL和CLX名称

VCL	CLX
ActnList	QActnList
Buttons	QButtons
Clipbrd	QClipbrd
ComCtrls	QComCtrls
Consts	QConsts
Controls	QControls
Dialogs	QDialogs
ExtCtrls	QExtCtrls
Forms	QForms
Graphics	QGraphics
Grids	QGrids
ImgList	QImgList
Menus	QMenus
Printers	QPrinters
Search	QSearch
StdCtrls	QStdCtrls

还可以将对Windows与Messages的引用转换为对Qt单元的引用。一些Windows数据结构现在还可以用于Type单元中（见第3章“运行时库”），所以必须将它添加到CLX程序中。然而，注意QTypes单元不是VCL Type单元的CLX版本，这两个单元完全无关。

警告：注意uses语句！如果编译一个包含CLX窗体的项目，但没有更新项目单元，则会留下对VCL Forms单元的引用，程序也将运行，但会立刻停止。原因是没有建立VCL窗体，所以程序立刻停止。在其他情况中，试图在一个VCL应用程序中建立CLX窗体将引起运行时错误。最后，Delphi IDE可能会不适当地添加对错误库的uses语句的引用，所以最终会有一个引用两个相同单元的uses语句，但只有第二个是有效的。这一错误不会影响程序编译，但程序却无法正常运行。

为了给转换我自己的一些程序提供帮助，笔者编写了一个简单的单元替换工具，名为VclToClx，其完整的源代码见本书选配光碟上的Tools文件夹。在附录A中可以看到这个程序的更多信息。

TControl与派生类

在第4章我们讨论了Delphi库的基类，特别强调了TComponent类。TComponent最重要的一个子类是TControl，它对应着可视组件。该类可以在CLX与VCL中使用，并定义了通用的概念，如控件的位置与大小、父控件控制它否，等等。在实际执行时，必须引用它的两个子类。在VCL中，它们是TWinControl与TGraphicControl；在CLX中，它们是TWidgetControl与TGraphicControl。下面是它们的主要特性：

基于Windows的控件（也叫窗口化控件） 是基于操作系统窗口的可视组件。VCL中的TWinControl有一个窗口句柄，这是一个引用内部Windows结构的数值。CLX中的TWidgetControl有一个Qt句柄，是对内部Qt对象的引用。从用户的角度来看，窗口化的控件可以检索输入焦点，有些还可以包含其他控件。这是Delphi库中最大的一组组件。可以进一步将窗口化控件划分为两组：Windows或Qt控件的包装器，以及定制控件，它们通常派生自TCustomControl。

图形控件（也叫非窗口化控件） 有一些可视组件，它们不基于操作系统窗口。所以，它们没有句柄，不能检索焦点，不能包含其他控件。这些控件继承自TGraphicControl，并由其父窗体来绘制，向它们发送与鼠标有关的事件以及其他事件。非窗口化控件的例子是Label与SpeedButton组件。该组中的控件不多，在Delphi早期（16位Windows），最小化系统资源的使用是非常重要的。使用图形控件来节省Windows资源在Win9x/Me中仍很有用，尽管现在已经在很大程度上受到了限制，但仍没有完全放弃它们（不像Windows NT/2000）。

Parent与控件

控件的Parent属性说明了哪一个控件负责显示它。当在Form Designer中将一个组件拖到一个窗体中时，窗体就变成了新控件的父控件与宿主。但如果将一个组件拖到一个Panel、ScrollBar或其他容器组件中，它们也会变成其父组件，但窗体仍是控件的宿主。

当在运行中建立控件时，需要设置宿主（使用Create构造器参数），但还必须设置Parent属性，或看不到的控件。

与Owner属性一样，可对Parent属性采用逆向操作。Controls数组事实上列出了当前控件的所有子控件，号码从0到ControlsCount-1。程序员可以扫描该属性，在另一个控件的所有子控件上操作，最终使用递归的方法，对每个子控件的子控件进行操作。

与组件大小及位置有关的属性

其他重要且对所有控件通用的属性是那些与组件大小及位置有关的属性。组件的位置由它的Left和Top属性确定，组件的大小（只针对控件而言）由它的Height和Width属性确定。所有组件都有位置，因为当用户在设计时重新打开一个已有窗体时，用户也许希望能够看到

非可视组件图标所处的位置就是用户放置它时的位置，该位置在文件中可以看到。

提示：当改变一些位置或大小属性时，最终会调用SetBounds方法。所以任何需要一次改变两个或多个这样的属性时，直接调用SetBounds都将加快程序的速度。另一个对象方法，BoundsRect，会返回控件的矩形边界，并相应地访问四个属性：Left、Top、Height和Width。

组件的位置有一个重要特性，与Windows中的其他任何坐标一样，它总是与其父组件的客户区（其Parent属性指示的组件）有关。对于窗体，客户区就是边框内的区域（但不包括边框本身）。尽管有一些现成的方法可以在窗体和屏幕之间进行坐标转换，但在屏幕坐标下工作还是显得有些混乱。

然而要注意，控件的坐标总是与父控件有关的，父控件通常是窗体，也可以是面板或其他容器组件。如果用户在窗体内放置一个面板，再在面板内放置一个按钮，那么按钮的坐标与面板有关，而不是与窗体有关。事实上，在这种情况下，按钮的父组件是面板。

激活和可见属性

有两个基本属性，可用来使用户激活或隐藏组件。最简单的是Enabled属性。当组件禁止使用（Enabled被设置为False）时，通常会有一些可以看到的提示。在设计时，“禁止使用”的属性并不总会被禁止，但在运行时，禁止使用的组件一般是灰的。

还有一种更为基本的方法，用户可以通过使用相应的Hide方法或将Visible属性设置为False来完全隐藏组件。然而需要注意，Visible属性的状态并不能确定一个控件是否可见。事实上，如果组件的父组件被隐藏，即使该组件被设置为Visible，用户也看不到它。因此，还有另一个属性，Showing，它是一个运行时的只读属性。用户可以读取Showing值来获知组件是否真的可见；也就是说，如果该组件是可见的，那么它的父控件也是可见的，父组件的父组件也是可见的，等等。

字体

定制组件用户界面时，经常使用的两个属性是Color与Font。与颜色有关的属性数量不少，Color属性自己通常引用组件的背景色。字体及其他很多图形元素也拥有Color属性。很多组件还有ParentColor和ParentFont属性，用来说明控件是否使用与它们的父组件（通常是窗体）相同的字体和颜色。只需通过设置窗体的Font属性，就可以在窗体上改变所有控件的字体。

当用户设置字体时，既可以在对象检验器中输入属性的特征值，也可以在标准字体选择对话框中选择系统安装的任何一种字体。Delphi允许用户使用系统内安装的所有字体既有好处也有弊端。主要的好处是，如果系统安装了一些优秀字体，则用户的程序可以任意使用它们。弊端是，当在其他计算机上使用该应用程序时，可能不兼容这些字体。

如果某应用程序使用的字体在它的用户机器上没有，Windows将选择其他字体来代替。这样，该程序精心设计的输出方式可能会被字体替代毁于一旦。因此，用户应该尽可能地使用Windows标准字体（如MS Sans Serif、System、Axial、Times New Roman等等）。另一种办法是，如果字体的用户许可证允许，可将一些字体载入应用程序中。

颜色

设置颜色值的办法很多。这种属性的类型是TColor。对于该类型的属性，用户可以从一系列预定义的名称常量中选择一个值或直接输入一个值。颜色常量包括clBlue、clSilver、clWhite、clGreen、clRed，以及其他很多常量（包括Delphi 6中的四种新的标准颜色：clMoneyGreen、clSkyBlue、clCream和clMedGray）。另一种更好的办法是使用系统表示特定元素所使用的颜色。这些颜色不同于VCL和CLX的。

VCL包括预先确定的Windows系统颜色，如某窗口的背景色（clWindow）、某高亮度菜单（clHighlightText）的文本颜色、某活动标题（clActiveCaption）以及按钮表面颜色（clBtnFace）等等。CLX包括一组不同并且不兼容的系统颜色集合，该集合包括窗体的标准色clBackground；编辑框和其他可视控件使用的颜色clBase；用于活动控制的前景颜色clActiveForeground；以及用于失效文字控件的背景颜色clDisabledBase。这里提及的所有颜色均显示在“TColor type”标题下的VCL和CLX帮助文件中。

还有一种方法是用数值（四字节的十六进制数）指定TColor，而不使用预定义值。如果用户使用这种方法，就应该知道该数值低位的三个字节分别代表RGB色度中的蓝、绿、红。例如，值\$00FF0000代表纯蓝色，值\$0000FF00代表绿色，值\$000000FF代表红色，值\$00000000代表黑色，值\$00FFFFFF代表白色。指定任一中间值，就能获得16 000 000种可能颜色中的一种。

如果不直接使用这些十六进制的数值，建议使用RGB函数，该函数有三个范围在0到255之间的参数，第一个表示红色的数量，第二个表示绿色的数量，第三个表示蓝色的数量。使用RGB函数与使用十六进制常数相比，增强了程序的可读性。实际上，RGB几乎可以说是Windows API函数。它是用与Windows有关的单元而不是Delphi单元定义的，但在Windows API中却不含该函数。在C中，有一个名称与作用都相同的宏指令，因此对于Pascal的Windows界面来说，添加该函数是一个受欢迎的举措。RGB不能在CLX上获得，因此笔者写了自己的版本如下：

```
function RGB (red, green, blue: Byte): Cardinal;  
begin  
  Result := blue + green * 256 + red * 256 * 256;  
end;
```

TColor类型的最高位字节用于指明对于最匹配的颜色应搜索哪个调色板，但这一论题过于高级，不适合在此论述（该字节还用于在复杂的图像程序中为屏幕上的每个显示元素传递透明度信息）。

在考虑调色板和颜色的匹配时需注意，Windows有时用最接近的纯色代替某一任意色，至少在使用调色板的视频方式中是这样的。对于字体、线等等的使用，也总是如此。还有有的时候，Windows使用抖动技术来模拟所需颜色。在16位适配器（VGA）中，还有更高级的解决方法，用户最终看到的是不同颜色像素的奇怪模式，而不是用户所想像的颜色。

TWinControl类 (VCL)

在Windows中, 用户界面的大多数元素都是窗口。从用户的角度看, 窗口是屏幕上由边框包围的区域, 拥有一个标题, 通常还有一个系统菜单。但从技术上讲, 窗口是内部系统表中的一个入口, 通常对应着屏幕上可以看到的元素, 具有一些相连代码。大多数这样的窗口都具有控件的作用; 其他窗口由系统临时建立 (例如, 显示一个下拉式菜单)。其他窗口仍由应用程序建立, 但隐藏了起来, 只作为接收消息的方式来使用 (例如, 套接字使用窗口与系统通信)。

所有窗口都被Windows系统所了解, 并为它们的行为引用一个函数, 每当系统中有事发生时, 一条通知消息就会传递给相应的窗口, 它通过执行一些代码做出响应。事实上, 系统的每个窗口都有一个与之相连的函数 (通常称为它的窗口过程), 负责处理窗口感兴趣的的不同消息。

在Delphi中, 任何TWinControl类都可以覆盖WndProc方法, 成为WindowProc属性定义一个新值。然而, 有意思的Windows消息可以通过提供专门的消息处理程序进行跟踪。VCL甚至可以将这些低级消息转换为事件。Delphi允许我们在高级别上工作, 使应用程序开发更容易, 同时也仍允许在需要时使用低级技术。

还要注意, 建立基于TWinControl类的实例不会自动建立与其相应的Window句柄。事实上, Delphi使用了一种惰性的初始化技术, 这样当需要时只建立低级控件, 通常只要有一个方法访问Handle属性就行。该属性的get对象方法最早被称做HandleNeeded, 然后叫做CreateHandle, 一直到CreateWnd、CreateParams与CreateWindowsHandle (顺序很复杂, 而且没有必要了解其细节)。反过来, 可以在内存中保留已有的控件 (或许是不可视的) 并解除其窗口句柄, 以节省系统资源。

TWidgetControl类 (CLX)

在CLX中, 每个TWidgetControl都有一个内部Qt对象, 程序员可通过Handle属性引用它。该属性具有同名且与之相应的Windows属性, 但两者的实质则完全不同。

TWidgetControl对象通常包含相应的Qt/C++对象。CLX类使用了迟滞构造 (内部对象只有需要其某一方法时才创建), 在InitWidget和其他方法中执行。当解除CLX类时, 会释放内部对象。还可以围绕已有的Qt对象建立一个窗口饰件; 在这种情况下, CLX对象不会拥有Qt对象, 也不会解除它。该行为由OwnHandle属性说明。

实际上, 每个VisualCLX组件都有两个相连的C++对象: Qt Handle与Qt Hook, 后者是接收系统事件的对象。使用当前Qt设计, 这必须是一个C++对象, 作为Delphi控件的事件处理程序的中间物。HookEvents方法用于将挂钩对象与CLX控件相连。

与Windows不同, Qt定义了两两种不同类型的事件:

- 事件 (Events) 是输入的转换或系统事件 (如按键、鼠标移动与绘图)。
- 信号 (Signals) 是内部组件事件 (对应着VCL内部或抽象的操作, 如OnClick与OnChange)。

然而, CLX组件的事件合并了事件和信号。一般的Delphi CLX控件包含OnMouseDown、OnMouseMove、OnKeyDown、OnChange、OnPaint和其他事件, 就像在VCL中一样 (调用大部

分事件就像响应Windows消息)。

说明：专业程序员会注意到CLX有一个很少使用的EventHandler方法，它多少对应着VCL中TWinControl类的WndProc对象方法。

打开组件工具框

当要编写Delphi应用程序时，打开一个新Delphi项目，会发现自己正面对着大量组件。问题是对于每一步操作都有多种方法，例如，使用列表框、组合框、单选按钮组、字符串网格甚至是树型预览（如果存在层次顺序）来显示一系列值。那么应该选择哪一种方法呢？这个问题是很难回答的，因为有很多因素需要考虑，要根据应用程序的任务而定。所以，下面针对一些常遇到的任务，将可以使用的各种方法进行归纳总结。

说明：对于下面介绍的一些控件，Delphi还包括了一个data-aware版本，通常由前缀DB来表示。在第13章中读者会看到，控件DB版本的典型功能与其“标准”版本相似，只是属性与使用的方法通常具有较大的差别。例如，在Edit控件中，使用了其Text属性；而在DBEdit组件中，却使用了访问相关字段对象的Value属性。

文本输入组件

尽管窗体和组件可以使用OnKeyPress事件来直接处理键盘输入，但这并不是其常用的功能。Windows提供了备用控件，可用于获得字符串输入，甚至可以建立简单的文本编辑器。Delphi在这方面提供了多种略有差别的组件。

Edit组件

Edit组件允许用户输入一行文本，也可以显示一行带Label或StaticText控件的文本，但是这些组件一般用来修改文本或生成程序输出，而不是输入。在CLX中，可以用一种LCD数字控件来显示数字。

Edit组件使用Text属性来引用其显示的文本，而在这方面，其他很多控件使用Caption属性。惟一可以影响用户输入的条件是，接受字符的数量。如果只想接受特殊字符，则可以处理编辑框的OnKeyPress事件。例如，编写一个方法来检测字符是数字键还是Backspace键（它的数值是8）。否则，就将该键的值改变为空字符（#0），这样编辑控件就不能处理它，并发出警告声：

```
procedure TForm1.Edit1KeyPress(  
  Sender: TObject; var Key: Char);  
begin  
  // check if the key is a number or backspace  
  if not (Key in ['0'..'9', #8]) then  
  begin  
    Key := #0;  
    Beep;  
  end;  
end;
```

说明：CLX的一个较小区别是，Edit控件没有内置的Undo机制。另一个区别是PasswordChar属性被EchoMode属性所替代。此时并不必确定要显示的字符，而是应确定要显示输入的文本，还是将输入文本显示为星号。

新添的LabeledEdit控件

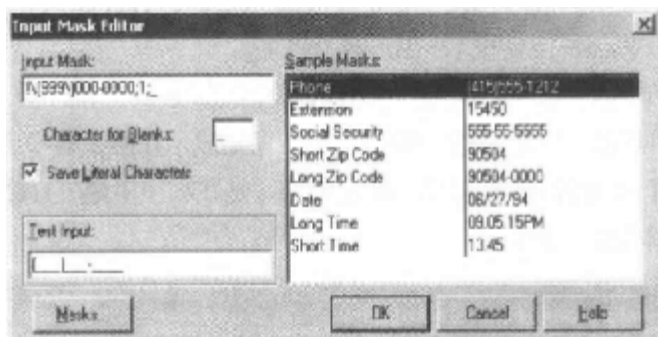
Delphi 6添加了一个非常有用的控件，叫LabeledEdit，它是一个带有标签的Edit控件。标签作为复合控件的一个属性，控件继承自TCustomEdit。

该组件非常方便，因为减少了窗体上组件的数量，便于更轻松地移动组件，并拥有了更标准的标签布局，特别是当它们放在编辑框上时。EditLabel属性会与子组件相连，它具有实际属性与事件。其他两个属性，LabelPosition与LabelSpacing，允许程序员配置两个控件的相对位置。

说明：该组件被添加到ExtCtrls单元中，说明在对象检验器中使用的是子组件。我们将在第9章“编写Delphi组件”中讨论这些组件的开发。还要注意，该组件不能在CLX上使用。

MaskEdit组件

为了进一步定制编辑框的输入，还可以使用MaskEdit组件，该组件具有EditMask属性，它说明每个字符是否为大写、小写或数字，以及其他相似条件的字符串。读者可以看到EditMask属性的编辑器如下所示。



Input Mask编辑器允许用户输入掩码，但它也要求用户为输入指出用做占位符的字符，并决定是否将出现在掩码中的字母与结果字符串一起保存。例如，用户可以选择只将电话号码区号两边的括号作为输入提示，或将它们与代表结果号码的字符串一起保存。Input Mask编辑器中的这两项对应着掩码的后两个部分（用分号分开）。

提示：单击Input Mask编辑器中的Masks按钮，可以为不同国家选择预定义的输入掩码。

Memo组件与RichEdit组件

前面讨论的所有控件都只允许单行输入。而Memo组件可以管理多行文本，但（在Win95/98平台上）它仍保留着16位Windows的32KB的文本限制，而且对于全部文本只提供一种字体。用户可以逐行对备注文本进行操作（使用Lines字符串列表）或一次访问所有文本（使用Text属性）。

如果用户想管理大量的文本或改变字体与段落对齐方式，在VCL中可使用RichEdit控件，这是一种基于RTF格式的Win32通用控件。读者可以在Delphi携带的范例程序中发现一个基

RichEdit组件的完整编辑器范例（范例的名称也是RichEdit）。

警告：RichEdit控件是一个很少在CXL和Kylix中不能用的Delphi控件。最新的Qt版本有一个相似的控件，所以在将来CLX版本中可以支持这些控件。

RichEdit组件有一个说明默认字体样式的DefAttributes属性及一个说明当前所选字体样式的SelAttributes属性。这两个属性不属于TFont类型，但它们与TFont兼容，所以可以使用Assign方法复制值，如下列代码段所示：

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  if RichEdit1.SelLength > 0 then
  begin
    FontDialog1.Font.Assign (RichEdit1.DefAttributes);
    if FontDialog1.Execute then
      RichEdit1.SelAttributes.Assign (FontDialog1.Font);
  end;
end;
```

TextViewer CLX控件

在所有常用控件中，CLX与Qt没有提供RichEdit控件。然而，它们提供了一个全新的HTML阅读器，可以显示格式化文本，但不能输入它。该HTML阅读器嵌入在两个不同的控件中，单页的TextViewer控件或TextBrowser控件。

作为一个简单的范例，我们向一个CLX窗体添加了一个备注控件与一个文本阅读器，并连接它们，这样在备注控件中输入的内容都会立刻显示在阅读器的中。该范例名为HtmlEdit；这不是因为它是真正的HTML编辑器，而是因为这是在程序中建立HTML预览最简单的方法。程序运行时的窗体如图5.3所示。

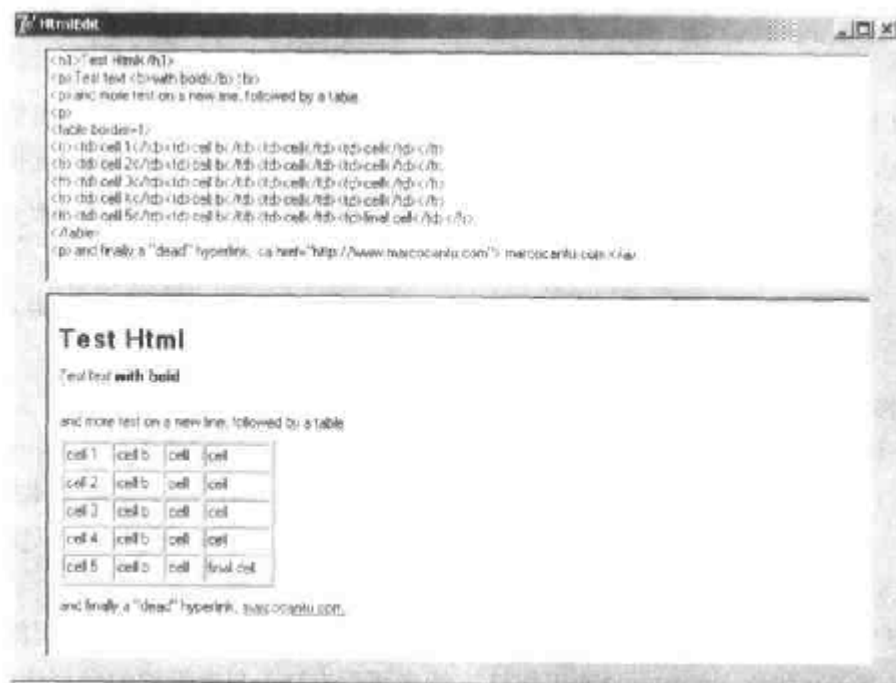


图5.3 运行时的HtmlEdit范例：当向备注控件中添加新的HTML文本时，会立即得到预览

提示：原来是用Kylix在Linux上建立该范例的，为了将它移植到Windows与Delphi中，必须做的所有工作就是复制文件与重新编译。

选择具体选项

有两个标准Windows控件允许用户选择不同的选项，还有另外两个控件用于对选项进行编组。

CheckBox组件与RadioButton组件

第一个组件是复选框，它对应着一个可供选择的选项，而不论其他复选框的状态如何。设置复选框的AllowGrayed属性，允许用户显示三种不同的状态（选择状态、未选择状态与变灰状态），当用户单击复选框时，这些状态会轮流变化。

第二个控件是单选钮，它对应着惟一选择。同一窗体或同一单选钮组容器中的两个单选钮不能同时选择，而且其中一个单选钮应总处于选择状态（作为程序设计员，在设计时应该负责选择一个单选钮）。

GroupBox组件

为了管理多组单选钮，可以使用GroupBox控件，从功能与视觉两方面将它们组合在一起。为了建立带有单选钮的组合框，只需将GroupBox组件放置在窗体上，然后向组合框添加单选钮即可，如下图所示。



可以单独处理单选钮，但也可以在组合框控制的控件数组中漫游，如上一章所讨论的那样。下面的一小段代码用于获得组合框中被选单选钮的文本：

```
var
  I: Integer;
  Text: string;
begin
  for I := 0 to GroupBox1.ControlCount - 1 do
    if (GroupBox1.Controls[I] as TRadioButton).Checked then
      Text := TRadioButton(GroupBox1.Controls[I]).Caption;
```

RadioGroup组件

Delphi还有一个相似的组件可以专门用于单选钮，这就是RadioGroup组件。RadioGroup组件是内部绘有一些单选钮副本的组合框。所不同的是内部的单选钮被容器控件自动管理。使用一组单选钮通常要比使用组合框容易些，因为各种条目是列表的一部分，就像在列表框中一样。下面这行代码说明了获取所选条目文本的方法：

```
Text := RadioGroup1.Items[RadioGroup1.ItemIndex];
```


另一个优点是RadioGroup组件还可以按一列或多列自动对齐它的单选钮（用Columns属性来说明），而且用户可以通过向Items字符串列表添加字符串，在运行时轻松地添加新选择项。相比之下，向组合框添加新单选钮则复杂得多。

列表

当有很多选择时，单选钮就难以胜任了。通常，单选钮的数量不应该超过五或六个，当需要更多选择时，可以使用列表框或其他可显示条目列表并允许对其中条目进行选择控件。

Listbox组件

在列表框中选择某个条目要使用Items与ItemIndex属性，其使用方法与前面介绍的RadioGroup控件一样。如果用户需要频繁访问所选列表框条目的文本，可以编写一个函数，如下所示：

```
function SelText (List: TListBox): string;
var
  nItem: Integer;
begin
  nItem := List.ItemIndex;
  if nItem >= 0 then
    Result := List.Items [nItem]
  else
    Result := '';
end;
```

另一个重要的特性是，通过使用列表框组件，可以决定是像在一组单选钮中那样进行单项选择，还是像在一组复选框中那样进行多项选择。通过指定MultiSelect属性的值，可以确定选择的方式。在Windows与Delphi列表框中有两种多项选择：多项选择与扩展选择。在第一种情况中，只需单击选项就可以选择多个条目；而在第二种情况中，可以使用Shift与Ctrl键选择多个连续或不连续的选项。第二种选择由ExtendedSelect属性确定。

对于多项选择列表框来说，程序可以使用SelCount属性检索与所选条目数量有关的信息，而且可以通过检测Selected数组确定被选定的条目。这个Boolean值数组的项数与列表框相同。例如，为了将所有被选条目连接到一个字符串中，可以像下面这样扫描Selected数组：

```
var
  SelItems: string;
  nItem: Integer;
begin
  SelItems := '';
  for nItem := 0 to ListBox1.Items.Count - 1 do
    if ListBox1.Selected [nItem] then
      SelItems := SelItems + ListBox1.Items[nItem] + ' ';
```

在CLX中与在VCL中不同，ListBox可使用Columns、Row、ColumnLayout、RowLayout属性来配置固定的行和列编号。当然，VCL中的ListBox只有Columns属性。

ComboBox组件

列表框需要占据大量的屏幕空间，而且它提供了固定的选择。也就是说，用户只能在列表框的条目范围内做出选择，而不能输入任何程序员没有预见到的选择。

可以使用ComboBox控件来解决这两个问题，该控件由一个编辑框与一个下拉式列表组成。ComboBox组件的行为根据其Style属性的值会发生很大的改变：

- csDropDown类型定义了一个典型的组合框，该组合框允许按需要直接编辑与显示列表框。
- csDropDownList类型定义了不允许编辑的组合框（使用按键输入来进行选择）。
- csSimple类型则定义了总是显示其下方列表框的组合框。

还需要注意，访问组合框中选项的文本要比在列表框中进行同样操作简单得多，因为程序员可以方便地使用Text属性。对于组合框来说，有一个有用而且常用的技巧，当用户输入一些文本并按回车键时，可以向列表中添加新选项。下面这个对象方法首先通过寻找数值为13（ASCII）的字符，测试用户是否按了回车键，然后检查并确定组合框文本不是空的，而且还没有添加到列表中——文本在列表中的位置是否小于零。下面是有关代码：

```
procedure TForm1.ComboBox1KeyPress(  
  Sender: TObject; var Key: Char);  
begin  
  // if the user presses the Enter key  
  if Key = Chr (13) then  
    with Sender as TComboBox do  
      if (Text <> '') and (Items.IndexOf (Text) < 0) then  
        Items.Add (Text);  
  end;
```

提示：在CLX中，用户按回车键时，组合框就能够自动将输入到编辑框的文本添加到下拉列表中。与VCL不同，一些事件被调用不同的次数。

从Delphi 6开始，组合框包含了两个新事件。OnCloseUp事件对应着下拉式列表的关闭行为，补充了已有的OnDropDown事件。OnSelect事件只有当用户在下拉式列表中进行选择时才会被触发，这与编辑部分中的输入不同。

另一个友好特性是AutoComplete属性。当设置它时，ComboBox组件（以及ListBox）会自动确定最靠近用户输入的字符串——建议作为文本的最后一部分。该特性（在CLX中）的核心在TCustomListBox.KeyPress对象方法中实现。

CheckListBox组件

列表框的另一个扩展是CheckListBox组件，该组件列表框的每个条目前都有一个复选框。



用户可以选择列表的单个条目，而且还可以单击复选框转换它们的状态。所以CheckListBox组件非常适用于多项选择或想突出一系列独立条目状态（就像在一系列复选框中一样）的情况。

为了选择每个条目的当前状态，可以使用Checked与State数组属性（如果复选框变灰则使用后者）。Delphi 5引入了ItemEnabled数组属性，用于使用或禁用列表的每一项。我们将在本章稍后的DragList范例中使用CheckListBox组件。

提示：大多数基于列表的控件都具有一个常见且非常重要的特性。列表的每个条目都有一个32位的相关值，通常由TObject类型表示。该值可以用做每个列表条目的标记，而且也可以用于存储每个条目的附加信息。注意，该方法与原Windows列表框控件（此控件为每个列表框条目提供了四字节的附加存储空间）的一个特性相关。我们将在本章稍后的ODList范例中使用该特性。

新的组合框：ComboBoxEx与ColorBox

ComboBoxEx（其中ex代表扩展的意思）是新Win32常用控件的包装器，它通过允许图像出现在靠近列表项的位置，扩展了传统的组合框。可以将一个图像列表附加到组合框中，然后为每个要显示的项选择图像。这种改变的效果是，简单的Items字符串列表被更复杂的集合ItemsEx属性所代替。在第7章的RefList2示例中将用到ComboBoxEx控件。

提示：在Delphi 7中，ComboBoxEx组件有一个新的AutoCompleteOptions属性，能够使组合框响应用户敲击键盘。

ColorBox控件是组合框的一个新版本，专门用于选择颜色。可以使用其Style属性，选择想在列表中看到的颜色组（标准颜色、扩展颜色和系统颜色等）。

List View与Tree View组件

如果需要更为复杂的列表，可以使用ListView控件，该控件可以使应用程序的用户界面看起来非常时髦。该组件使用起来稍微有些复杂，就像本章“ListView与TreeView控件”一节将说明的那样。还有一个通用控件是TreeView控件，它可以在一种层次式输出中显示条目，而StringGrid控件会为每一行显示多个元素。这个组件的运用实例可以在附录C中所提到的免费在线章节“Delphi中的图像”中看到。

如果在应用程序中使用这些公共控件，就应了解它们的使用方法，而且认识到我们的应用程序用户界面是最新的。TreeView与ListView是Windows Explorer的两个关键组件，可以假设很多用户已经非常熟悉它们，甚至超过了对传统Windows控件的熟悉程度。CLX也添加了一个IconView控件，它与VCL ListView部分特征相似。

警告：在CLX中，ListView控件没有Windows相应的大/小图标样式，但是可以用伴随控件IconView来实现。

新添的ValueListEditor组件

Delphi应用程序通常使用原来由字符串列表提供的名称/值结构，在上一章中讨论过。Delphi 6引入了一个StringGrid组件（TCustomDrawGrid子类）的版本，专门调整这类字符串列表。ValueListEditor有两列，其中可以显示并让用户编辑一帶有名称/值对的字符串列表，如图5.4所示。该字符串列表在控件的Strings属性中说明。

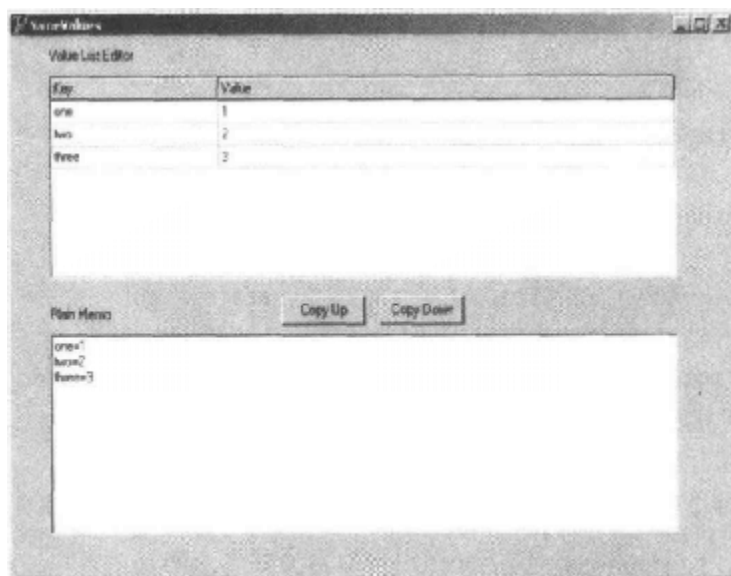


图5.4 NameValue范例有一个新ValueListEditor组件，它显示了一个字符串列表的名称/值或关键字/值对，还可以在一个普通的备注控件中查看

该控件的功能是可以为网格或每个关键字值定制编辑选项，使用只在运行时可用的ItemProps数组属性定制编辑选项。对于每一项来说，可以指明以下内容：

- 它是否只读
- 字符串中字符的最大个数
- 一个编辑屏蔽（在OnGetEditMask事件中尤其需要）
- 下拉式列表中的项（在OnGetPickList事件中尤其需要）
- 显示编辑对话框所需的显示按钮（在OnEditButtonClick事件中）

不用说，该行为类似于字符串网格与DBGGrid控件，以及对象检验器的行为。

ItemProps属性必须在运行时通过建立TItemProp类的一个对象，并将它赋给一个索引或字符串列表的关键字来设置。为了每行都有一个默认编辑器，可以多次赋给相同的项目属性对象。在范例中，这个共享的编辑器设置了一个三位数的编辑屏蔽：

```
procedure TForm1.FormCreate(Sender: TObject);
var
  I: Integer;
begin
  SharedItemProp := TItemProp.Create (ValueListEditor1);
  SharedItemProp.EditMask := '999;0;';
  SharedItemProp.EditStyle := esEllipsis;

  FirstItemProp := TItemProp.Create (ValueListEditor1);
  for I := 1 to 10 do
    FirstItemProp.PickList.Add(IntToStr (I));

  Memo1.Lines := ValueListEditor1.Strings;
  ValueListEditor1.ItemProps [0] := FirstItemProp;
  for I := 1 to ValueListEditor1.Strings.Count - 1 do
    ValueListEditor1.ItemProps [I] := SharedItemProp;
end;
```

只要行数改变，就必须重复相似的代码——例如，通过在备注中添加新元素并将它们复制给值列表：

```
procedure TForm1.ValueListEditor1StringsChange(Sender: TObject);
var
  I: Integer;
begin
  ValueListEditor1.ItemProps [0] := FirstItemProp;
  for I := 1 to ValueListEditor1.Strings.Count - 1 do
    if not Assigned (ValueListEditor1.ItemProps [I]) then
      ValueListEditor1.ItemProps [I] := SharedItemProp;
  end;
```

说明：显然，重复指定相同的编辑器两次会引起一些麻烦，所以应只向没有编辑器的行指定编辑器。

另一个属性，**KeyOptions**，仍然允许用户编辑关键字、添加新入口、删除已有项，并在字符串的第一部分复制名称。很奇怪，不能添加新关键字，除非也激活编辑选项，这使得让用户添加额外项目并保留基本项名称的工作变得很困难。

范围

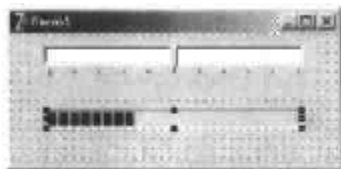
最后，还有些组件可以在某范围内选择值。在数字化输入以及在列表中选择元素时，可以使用范围。

ScrollBar组件

独立的滚动条控件是范围组件中最原始的，但它很少单独使用。滚动条通常与其他组件相关，如列表框与备注字段，或直接与窗体相连。在所有这些情况中，滚动条可以被看做是其他组件表面的一部分。例如，带有滚动条的窗体实际上在其边框上绘制了一个与滚动条相似的区域，是由窗体窗口的一种特殊**Windows**类型管理的特性。在这里使用了相似这个词，因为从技术上讲，它不是滚动条组件类型的独立窗口，而只是“伪”滚动条。在Delphi中，通常使用窗体和其他组件的两个特定属性控制这些伪滚动条：**VertScrollBar**和**HorzScrollBar**。

TrackBar组件与ProgressBar组件

直接使用滚动条组件的情况很少见，特别是在Windows 95引入**TrackBar**组件之后——该组件用于让用户在某个范围内选择值。在Win32常用控件中，还有一个配套的**ProgressBar**控件，它允许程序在某个范围内输出值，并显示操作进程。



UpDown组件

另一个相关控件是**UpDown**组件，它实际上与一个编辑框相连，允许用户输入数值或使用两个箭头按钮增减数值。两个控件之间的连接可以通过设置**UpDown**组件的**Associate**属性

来实现。可以单独使用UpDown组件，显示一个标题框的当前值或用其他方式显示。

说明：在CXL中没有UpDown控件，但SpinEdit实际上将Edit和UpDown捆绑为一个控件。

PageScroller组件

Win32 PageScroller组件是一个容器组件，允许程序员滚动内部控件。例如，如果在页滚动器中放置了一个工具栏，而且工具栏超出了可视区，则PageScroller控件会在两端显示两个小箭头。单击这两个箭头可以滚动内部区。该组件可以被用做滚动条，但只能部分地代替ScrollBar控件。

ScrollBar组件

ScrollBar组件表示了窗体的一个区域，该区域可以独立于窗体的其余部分进行滚动。因此，ScrollBar有两个滚动条用于移动内含的组件。程序员可以轻易地将任何组件放置到ScrollBar中，就像对待面板一样。事实上，ScrollBar就是一种带有滚动条（用于滚动其内部表面）的面板，是在很多Windows应用程序中使用的界面元素。当一个窗体中有很多控件及工具栏或状态栏时，可以使用ScrollBar覆盖窗体的中央区域，将其工具栏与状态栏留在滚动区外面。事实上，依靠窗体的滚动条可以允许用户将工具栏或状态栏移到视野外，这是一种非常奇怪的情况。

命令

最后一类组件并不是划分很清晰的，它们与命令相关。这个组中的基本组件是TButton（或者push按钮，在Windows的术语中）。除了单独的按钮，Delphi程序员还使用位于工具栏内的按钮（在早期的Delphi版本中，他们使用位于面板内的快速按钮）。除了按钮以及类似的控件，另外一个发出命令的关键技术是使用菜单项，可以是主窗体的下拉菜单或者通过右键激活的弹出菜单。

与菜单或工具栏相关的命令属于不同的类别，取决于它们的用途和提供给用户的响应：

Commands Commands是用于执行操作的菜单项或按钮。

State-setters State-setters是用于设置一个选项的开关状态的菜单项或按钮，用来改变一个特定元素的状态。命令的菜单项通常带有一个复选标记，用来指明它们是激活的状态（我们可以通过AutoCheck属性来获得这个特性）。通常用按钮被按下的状态来表示同样的状态（ToolButton控件具有一个Down属性）。

Radio Items 把菜单项放在一起用于显示可以替换的选项，与单选按钮相似。为了获得单选菜单项，将RadioItem属性设为True并且设置替换菜单项的GroupIndex属性为相同的值。使用类似的办法，我们可以将工具栏的按钮分成组，并且使它们相互排斥。

Dialog Openers 使对话框显示内容的项目。它们通常通过一个接在文本后面的省略号来表示。

命令与动作

正如我们将要在第6章中看到的，现代的Delphi应用程序倾向于使用ActionList组件或者它的ActionManager扩展来处理菜单和工具栏命令。简短地说，我们定义一系列的动作对象

并且将它们分别与一个工具栏按钮或者一个菜单项相关联。我们可以分别定义命令的执行，也可以把动作作为目标来更新用户界面；相关的可视控件将会自动地反映行为对象的状态。

菜单设计器

如果我们需要在应用程序中显示一个单独的菜单，则可以在窗体上放置一个MainMenu或者PopupMenu组件并且双击它来启动Menu Designer，如图5.5所示。可以添加新的菜单项并且提供Caption属性，使用一个“-”来分隔标题菜单项。

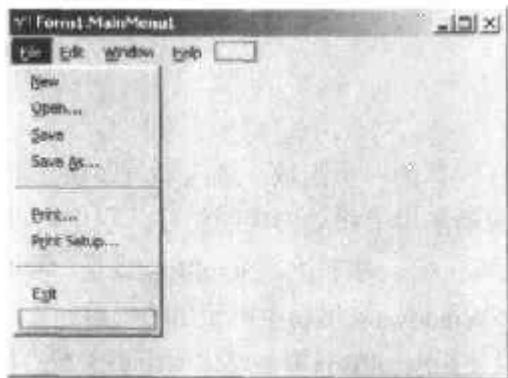


图5.5 Delphi的菜单设计器

对于每一个添加的菜单项，Delphi创建了一些新的组件。为了命名这些组件，Delphi使用了我们输入的标题并且附加了一个数字（因此Open变成了Open1）。在去掉标题中的空格以及其他特殊字符后，如果没有任何字符了，那么Delphi会添加一个字母N到这个名称中。最后，向它附加一个数字。因此，菜单项的分隔符被称为N1、N2等等。通过了解Delphi的默认工作方式，我们应该首先想到名称的编辑，这对于我们建立合理的命名计划非常重要。

警告：不要使用Break属性，它是用来在多个列上布置一个下拉菜单的。mbMenuBarBreak的值表明了这个项目将被显示在第二行或者接下来的行中，mbMenuBarBreak意味着这个项目将被添加到一个下拉菜单的第二列或者下面的列中。

为了获得一个更具现代感的菜单，我们可以向这个程序添加一个图像列表控件，带有一系列的位图，并且将这个图像列表与菜单连接——通过使用它的Images属性。我们可以为每一个菜单项设定一个图像，这是通过给它的ImageIndex属性设置一个正确的值来实现的。菜单图像的定义是非常灵活的，我们可以使用SubMenuImages属性将一个图像列表与任何特定的下拉菜单相关联（甚至是一个特定的菜单项）。应使用一个小型的图像列表对应每一个菜单项而不是使用一个大型的图像列表对应于整个菜单，这可以使我们在运行时更灵活地定制应用程序。

提示：在运行时创建菜单是很常见的，Delphi在Menus单元中提供了一些可供使用的函数。包括：NewMenu、NewPopupMenu、NewSubMenu、NewItem和NewLine。

弹出式菜单与OnContextPopup事件

用户在某组件上单击鼠标右键会显示该菜单，且该组件使用给定的弹出式菜单作为其PopupMenu属性值。然而，除了使用相应属性将弹出式菜单与组件连接之外，还可以调用其

Popup对象方法,该方法需要菜单在屏幕上弹出的坐标位置作为参数。使用本地组件的**ClientToScreen**对象方法,通过将该位置坐标点转换为屏幕坐标点,可以获得相应的值。下面是相应的代码:

```

procedure TForm1.Label3MouseDown(Sender: TObject;
  Button: TMouseButton; Shift: TShiftState; X, Y: Integer);
var
  ScreenPoint: TPoint;
begin
  // if some condition applies...
  if Button = mbRight then
  begin
    ScreenPoint := Label3.ClientToScreen (Point (X, Y));
    PopupMenu1.Popup (ScreenPoint.X, ScreenPoint.Y)
  end;
end;

```

另一种Delphi 5提供的方法是使用**OnContextMenu**事件。当用户用鼠标右键单击组件时,会触发该事件,确切地说,通过使用**If Button=mbRight**测试语句,进行跟踪检查操作。该方法的优点在于,还可以使用**Shift+F10**组合键以及一些键盘的快捷菜单键来触发相同的事件。程序员可以用少量的代码使用该事件触发一个下拉式菜单:

```

procedure TFormPopup.Label1ContextMenu(Sender: TObject;
  MousePos: TPoint; var Handled: Boolean);
var
  ScreenPoint: TPoint;
begin
  // add dynamic items
  PopupMenu2.Items.Add (NewLine);
  PopupMenu2.Items.Add (NewItem (TimeToStr (Now), 0, False, True, nil, 0, ''));
  // show popup
  ScreenPoint := ClientToScreen (MousePos);
  PopupMenu2.Popup (ScreenPoint.X, ScreenPoint.Y);
  Handled := True;
  // remove dynamic items
  PopupMenu2.Items [4].Free;
  PopupMenu2.Items [3].Free;
end;

```

该范例向快捷菜单添加了一些动态动作,并添加了临时项来说明何时显示弹出式菜单。这并不特别有用,但可以突出说明是否需要显示一个普通的弹出式菜单,程序员可以简单地使用控件或其父控件的**PopupMenu**属性。只有在想要进行一些额外处理时,使用事件**OnContextMenu**才有意义。

Handled参数被预初始化为**False**,这样如果没有在事件处理程序中添加其他代码,就会以常规的弹出式菜单进行处理。如果在事件处理程序中添加了代码来代替常规的弹出式菜单(在本例中,如弹出一个对话框或一个定制菜单),则应该将**Handled**设置为**True**,而系统

将停止处理消息。将Handled设置为True并不常见，因为程序员通常先处理OnContextPopup来动态地建立或定制弹出式菜单，然后让默认的处理程序显示菜单。

OnContextPopup事件的处理程序不止限于显示弹出式菜单，它可以进行任何操作，如直接显示一个对话框。下面是右击操作的例子，用于改变控件的颜色：

```
procedure TFormPopup.Label2ContextPopup(Sender: TObject;  
    MousePos: TPoint; var Handled: Boolean);  
begin  
    ColorDialog1.Color := Label2.Color;  
    if ColorDialog1.Execute then  
        Label2.Color := ColorDialog1.Color;  
    Handled := True;  
end;
```

本节中所有的代码段都摘自于VCL的CustPop范例和CXL的QCustPop范例。

控件相关的技术

在对大多数Delphi控件进行了概括性简介之后，我们将使用一些篇幅来介绍通用的核心技术，它并不是与某个特定组件关联的。我们将讨论输入焦点、控件的锚标志、分割条的使用以及提示信息的显示。当然，这些课题不能包括所有的关于可视控件的内容，但是它们提供了一个使用这种技术的开始点。

处理输入焦点

大多数控件使用TabStop与TabOrder属性来确定控件接受输入焦点的顺序（当用户按Tab键时）。除了手工设置窗体每个组件的定位顺序属性外，还可以使用窗体设计器的弹出菜单激活Edit Tab Order对话框，如图5.6所示。

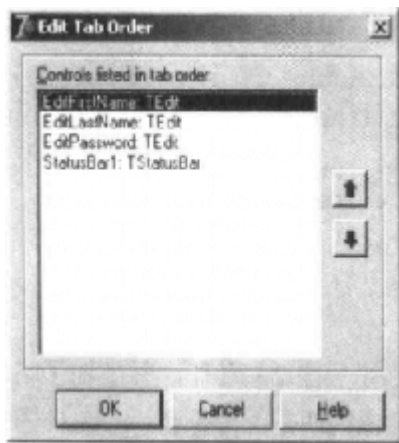


图5.6 Edit Tab Order对话框

除了这些基本设置外，还需要知道，每当组件接受或失去输入焦点时，它会接收到相应的OnEnter或OnExit事件。这允许程序员更好地调整与定制用户操作的顺序。其中一些技

术由InFocus范例演示，该范例中建立了一个典型的密码输入窗口。它的窗体中有三个编辑框与三个说明其含义的标签，如图5.7所示。窗口底部是一个用提示指导用户的状态区。每一项都需要按顺序依次输入。

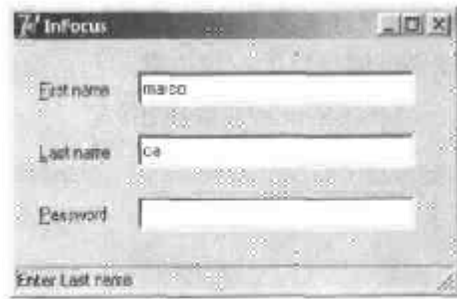


图5.7 运行时的InFocus范例

对于状态信息的输出，我使用了带有单独输出区的状态条（StatusBar）组件（将其SimplePanel属性设置为True而得到）。下面是该范例的属性概况。注意标题框中表示快捷键的&字符以及这些标题框与相应编辑框之间的连接（使用FocusControl属性）：

```
object FocusForm: TFocusForm
  ActiveControl = EditFirstName
  Caption = 'InFocus'
  object Label1: TLabel
    Caption = '&First name'
    FocusControl = EditFirstName
  end
  object EditFirstName: TEdit
    OnEnter = GlobalEnter
    OnExit = EditFirstNameExit
  end
  object Label2: TLabel
    Caption = '&Last name'
    FocusControl = EditLastName
  end
  object EditLastName: TEdit
    OnEnter = GlobalEnter
  end
  object Label3: TLabel
    Caption = '&Password'
    FocusControl = EditPassword
  end
  object EditPassword: TEdit
    PasswordChar = '*'
    OnEnter = GlobalEnter
  end
  object StatusBar1: TStatusBar
    SimplePanel = True
```

```

    end
end

```

程序非常简单，只有两步操作。第一步操作是在状态栏中显示拥有焦点的编辑控件名。该操作是通过处理控件的OnEnter事件实现的，可以使用一个通用事件处理程序来避免代码的重复。在范例中，并不存储每个编辑框的附加信息，而是检查窗体的每个控件，以确定与当前编辑框相连的标签是哪一个（用Sender参数指出）：

```

procedure TFocusForm.GlobalEnter(Sender: TObject);
var
    I: Integer;
begin
    for I := 0 to ControlCount - 1 do
        // if the control is a label
        if (Controls[I] is TLabel) and
            // and the label is connected to the current edit box
            (TLabel(Controls[I]).FocusControl = Sender) then
            // copy the text, leaving off the initial & character
            StatusBar1.SimpleText := 'Enter ' +
                Copy (TLabel(Controls[I]).Caption, 2, 1000);
    end;

```

窗体的第二个事件处理程序与第一个编辑框的OnExit事件相连。如果控件是空的，它会拒绝释放输入焦点，并在向用户显示一条消息前设置输入焦点返回。该对象方法还会查找一个给定的输入值，自动填充第二个编辑框并将焦点移到第三个编辑框中：

```

procedure TFocusForm.EditFirstNameExit(Sender: TObject);
begin
    if EditFirstName.Text = '' then
        begin
            // don't let the user get out
            EditFirstName.SetFocus;
            MessageDlg ('First name is required', mtError, [mbOK], 0);
        end
    else if EditFirstName.Text = 'Admin' then
        begin
            // fill the second edit and jump to the third
            EditLastName.Text := 'Admin';
            EditPassword.SetFocus;
        end;
    end;

```

提示：该范例的CLX版本与QInFocus程序具有相同的代码并可以一样使用。

控件的锚标志

为了建立完美与灵活的、可以适应窗体当前大小的用户界面，Delphi允许使用Anchors属性确定控件的相应位置。在Delphi 4引入该特性之前，每个放置在窗体上的控件都具有相

对于顶边与左边的坐标，除非它与底边或右边对齐。对齐操作适用于某些控件，但不是所有控件，特别是按钮。

通过使用锚标志，可以相对窗体的任何一边设置控件的位置。例如，使按钮的位置位于窗体右下角。只需将按钮放在所需位置，将它的Anchors属性设置为[akRight, akBottom]。当窗体大小改变时，按钮与锚标志的距离保持不变。换句话说，如果设置了这两个锚标志并删除了这两个默认值，按钮将保持其在右下角的位置。

另一方面，如果在窗体中央放置一个大组件，如Memo组件或ListBox组件，可以设置它们的Anchors属性使它包括所有的四边。这样，控件就会像对齐控件一样，随窗体的大小而增大或缩小，但控件与窗体边框之间将总保持一段空白。

提示：锚标志，与限制一样，可以在设计时与运行时使用。所以，应该尽早地设置它们，以便在运行时和在窗体设计中从该特性中获益。

本章建立了一个Anchors应用程序，它解释了这两种方法，该范例有两个位于窗体右下角的按钮与一个位于中央的列表框。如图5.8所示，当窗体大小改变时，控件会自动移动与伸展。为了使窗体正常工作，还必须设置其Constraints属性，否则，当窗体变得太小时，控件可能会重叠或消失。



图5.8 在Anchors范例中，当用户改变窗体的大小时，控件会自动移动与改变大小。移动控件不需要编写代码，只需要正确使用Anchors属性即可

如果删除所有锚标志，或两个相对的锚标志（如左与右），则调整大小的操作会导致控件四处浮动。该控件保持了当前大小，而且系统在控件每边上都添加或减去了相同数目的像素。这可以被定义为中央锚标志，因为如果组件一开始就在窗体中央，它会保持原来的位置不变。在任何情况下，如果想使控件位于中央位置，通常应该使用两个相对的锚标志，这样，如果用户改变窗体的大小，控件大小也会随着改变。事实上，在刚提到的情况中，窗体变大，其中央会留下一个小控件。

窗体分割技术

在Delphi中，有多种方法可以实现窗体分割技术，但是最简单的方法是使用Splitter组件，可以在组件面板的Additional页中找到它。为了使之更为有效，可以将分隔条与相关控件的Constraints属性结合起来使用。从下面的Split1范例中看到，这种做法允许程序员定义分隔条与窗体的最大位置、最小位置。要建立Split1范例，首先应在窗体中放置一个ListBox组

件，然后放置Splitter组件、第二个ListBox组件、另一个Splitter组件以及第三个ListBox组件。窗体的面板中还应有一个简单的工具栏。

通过简单放置两个Splitter组件，使窗体完全具备了在运行时移动并设置其所属组件大小的能力。分割条组件的Width、Beveled以及Color属性确定了它们的外观，在Split1范例中，可以使用工具栏来改变它们。另一个相关的属性是MinSize，它确定了窗体内不同组件的最小化尺寸。在分割操作中（如图5.9所示），有一条线标志着分割条的最终位置，但拖动该线不能超过一定的限制。Split1程序不让控件变得太小。另一种技术是将分割条的AutoSnap属性设置为True，当控件的大小低于MinSize限定时，该属性会让分割条隐藏控件。

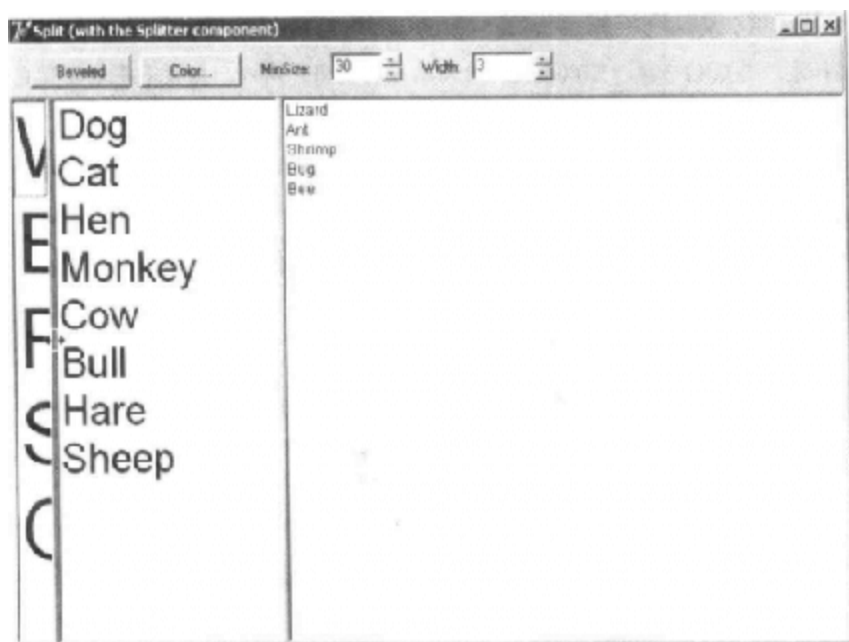


图5.9 Split1范例的Splitter组件确定了窗体上每个控件的最小尺寸，即使是那些不与Splitter相邻的控件

建议读者试着运行Split1范例，以便完全理解分割条是怎样影响与之相连的控件以及窗体的其他控件的。即使设置了MinSize属性，该程序的用户也可以将窗体的尺寸减到最小，隐藏一些列表框。而如果测试范例的Split2版本，会得到更好的结果。在Split2中，为ListBox控件设置了一些Constraints属性，如：

```
object ListBox1: TListBox
  Constraints.MaxHeight = 400
  Constraints.MinHeight = 200
  Constraints.MinWidth = 150
```

只有实际改变控件的大小时才会应用大小限定，所以为了使程序按一种令人满意的方式运行，必须将两个分隔条的ResizeStyle属性设置为rsUpdate。该值说明，每当移动分割条时（而不是在操作结束时），就会更新控件的位置。如果选择rsLine或rsPattern值，分割条只会在需要的位置画一条线，并检查MinSize属性而不是控件的限制。

提示：当设置Splitter组件的AutoSnap属性为True时，若相邻控件的大小小于Splitter组件为其设置的最小限定，则分割条会完全隐藏在相邻控件后。

水平分割

Splitter组件还可以用于水平分割，而不是默认的垂直分割。然而，该方式稍微有些复杂。基本作法是，将一个组件放置在窗体上，将其与窗体顶部对齐，然后在窗体上放置一个分割条，默认时它是左对齐的。选择Align属性的alTop值，然后手动重新设置组件的大小。在SplitH范例中，可以看到带有水平分割条的窗体。该程序有两个备注组件，可以在其中打开文件，它还有一个分割条将这两个备注组件分开，定义如下：

```
object Splitter1: TSplitter
  Cursor = crVSplit
  Align = alTop
  OnMoved = Splitter1Moved
end
```

该程序有一个状态条，用于跟踪两个备注组件的当前高度。该程序还会处理分割条的OnMoved事件（该组件的唯一事件）来更新状态条的文本。无论何时改变窗体的大小，都会执行相同的代码：

```
procedure TForm1.Splitter1Moved(Sender: TObject);
begin
  StatusBar1.Panels[0].Text := Format ('Upper Memo: %d - Lower Memo: %d',
    [MemoUp.Height, MemoDown.Height]);
end;
```

加速键

自从Delphi 5起，就不需要在菜单项的Caption中输入&字符了，如果用户省略快捷键，它会提供一个自动的快捷键。Delphi自动快捷键系统还可以判断是否输入了冲突的快捷键，并修正它们。这并不意味着应停止使用&字符添加定制的快捷键，因为自动系统只会使用第一个可以使用的字母，它不会遵循默认标准，而且与自动系统相比，我们可以找到更好记忆的快捷键。

这个新特性由AutoHotkeys属性控制，该属性可以用于主菜单组件以及每个下拉式菜单与菜单项中。在主菜单中，该属性的默认值为maAutomatic；在下拉式菜单及菜单项中，其默认值为maParent。这样，为主菜单组件设置的值会自动被所有子项使用，除非它们有特殊的maAutomatic值或maManual值。

该系统背后的引擎是TMenuItem类的RethinkHotkeys方法，以及配套的InternalRethinkHotkeys方法。还有一个RethinkLines方法，用于检查下拉式菜单是否有两个连续的分割符，或在开始或末尾有一个分割符。在所有这些情况中，分割符会被自动删除。

Delphi引入该新特性的原因之一是对转换提供支持。当需要转换一个应用程序的菜单时，如果无需处理快捷键是很方便的，至少不用担心同一菜单中是否有两个相冲突的项。有一个可以自动解决相似问题的系统显然大有好处。另一个动机是Delphi自己的IDE。使用所有动态装载的组件包在IDE主菜单或弹出式菜单中安装菜单项，以及使用装载在不同版本程序中的不同组件包，几乎不可能不在每个菜单中出现冲突的快捷键设置。这也说明了为什么该系统不能在设计时用做静态分析菜单的向导，而只能用于在运行时处理动态建立的菜单管理问题。

警告：这个新特性确实非常方便，但因为它可以默认使用，所以可能会影响已有代码。我们不得不改动本章两个程序范例的Delphi 4和5版本，就是为了避免该变化造成的运行时错误。问题在于在代码中使用了标题，以及额外的&会影响代码。尽管变化非常简单，所需要做的就是将主菜单组件的AutoHotkeys属性设置为maManual。

使用浮动提示

工具栏中另一个常用的元素是tooltip，也叫浮动提示——一些简要描述当前光标所指按钮的文本。这些文本通常当鼠标光标在一个按钮上停留一段时间之后会显示在一个小黄框中。为了向一组工具栏或组件添加提示，只需将其ShowHints属性设置为True并为每个按钮的Hint属性输入一些文本。可能需要让提示对窗体上的所有组件或工具栏、面板上的按钮都起作用。

如果需要更多的控件来显示提示信息，可以使用Application对象的一些属性和事件。其中，该全局对象有下列属性：

属性	定义
HintColor	提示窗口的背景颜色
HintPause	在提示被显示前，光标应停留在该组件上的时间
HintHidePause	提示将被显示多长时间
HintShortPause	如果其他提示正被显示，系统将等待多长时间来显示该提示

例如，程序允许用户用下列代码选择特殊的颜色来定制提示背景：

```
ColorDialog.Color := Application.HintColor;  
if ColorDialog.Execute then  
    Application.HintColor := ColorDialog.Color;
```

作为一种选择，可以通过处理Application对象的OnShowHint属性来改变提示颜色。该处理器能为特殊控件改变提示的颜色。OnShowHint事件用在本章稍后描述的CustHint示例中。

定制提示

正如向应用程序工具栏添加提示一样，还可以向窗体或向窗体的组件添加提示。对于大控件，提示显示在鼠标的旁边。在一些情况下，知道程序可以随意定制显示提示是很重要的。最简单的操作就是改变Application对象的属性值，像在上一小节中所做的一样。要更好地控制提示，可以赋予应用程序的OnShowHint事件一个对象方法，以便进一步定制提示。更好的方法是向窗体添加一个ApplicationEvents组件并处理其OnShowHint事件。

必须定义的方法有一些有趣的参数，如提示文本的字符串、提示启动的Boolean标志，以及带有更多信息的THintInfo结构，包括控件、提示位置和提示的颜色。每个参数都以引用的方式传递，因此程序员有机会改变它们，也能修改THintInfo结构的值，例如，在提示被展示前，可以改变提示窗口的位置。

如CustHint范例中，在标签中心显示的提示。

```
procedure TForm1.ShowHint (var HintStr: string; var CanShow: Boolean;  
    var HintInfo: THintInfo);  
begin
```

```

with HintInfo do
  // if the control is the label show the hint in the middle
  if HintControl = Label1 then
    HintPos := HintControl.ClientToScreen (Point (
      HintControl.Width div 2, HintControl.Height div 2));
  end;

```

这段代码必须检索通用控件的核心（HintInfo.HintControl），然后向该控件应用 ClientToScreen 对象方法，将控件坐标转换为屏幕坐标。

可以用另外一种方式更新 CustHint 范例。窗体中的 ListBox 控件有些相当长的文本项，所以可能在鼠标移到该项时需要显示全部的文本提示。当然，为列表框设置一个提示效果不好。

好的解决方法是通过动态提示鼠标下的列表框文本来定制提示系统。需要指出提示所属的系统，从而可以在鼠标移到下一行时显示新的提示。这些可以通过设置 CursorRect 的 THintInfo 记录实现，它只指示鼠标可以移过的组件区域，而不需要设置提示。当鼠标移出该区域时，Delphi 会隐藏提示。这是加入 ShowHint 属性的相关代码：

```

else if HintControl = ListBox1 then
begin
  nItem := ListBox1.ItemAtPos(
    Point (CursorPos.x, CursorPos.y), True);
  if nItem >= 0 then
  begin
    // set the hint string
    HintStr := ListBox1.Items[nItem];
    // determine area for hint validity
    CursorRect := ListBox1.ItemRect(nItem);
    // display over the item
    HintPos := HintControl.ClientToScreen (Point(
      0, ListBox1.ItemHeight * (nItem - ListBox1.TopIndex)));
  end
  else
    CanShow := False;
  end;

```

其效果是为每一行显示特定的提示，如图 5.10 所示；并且计算出提示框的位置，使它能覆盖当前项的文本和列表框的边界。

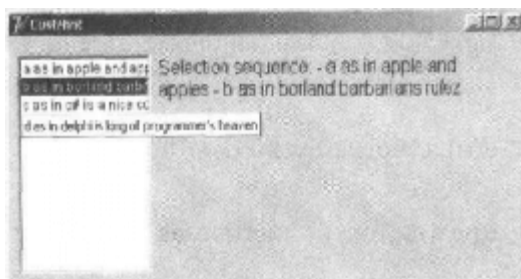


图 5.10 CustHint 范例的 ListBox 控件显示了鼠标所在列表项的不同提示

自绘制控件与样式

在Windows中，系统通常会负责绘制按钮、列表框、编辑框、菜单项以及相似组件，这些组件基本知道怎样绘制自己。然而，还有一种选择，即系统允许这些控件的宿主，一般是窗体，来绘制它们。这项适用于按钮、列表框、组合框以及菜单项的技术叫做自绘制。

VCL中的实际情形稍复杂些。组件在这种情况下也可以绘制自己（如位图按钮的TBitBtn类的情况），并可能激发相应的事件。通常系统会向宿主（通常是窗体）发出请求，要求绘制，而窗体会将事件返回到相应的控件上，并激活其事件处理程序。在CXL中，一些控件如ListBox、ComboBox，其表面事件与Windows的自绘制窗体相似，但是菜单缺少它们。Qt的本机方法是使用样式来确定系统中特定应用程序或给定控件上的所有控件的图像行为。稍后将简短介绍与样式相关的内容。

说明：大多数Win32常用控件都支持自绘制技术，也叫做定制绘图技术。程序员完全可以定制ListView、TreeView、TabControl、PageControl、HeaderControl、StatusBar或ToolBar的外观。ToolBar、ListView与TreeView还支持高级的定制绘图，这是从Microsoft最新版本的Win32常用控件库中引入的，是更为精细的绘图功能。自绘制技术的缺点是，在将来Windows用户界面样式发生改变时，完全适应当前用户界面样式的自绘制控件将会显得过时。因为现在正建立定制的用户界面，所以需要不断地更新它。相比而言，如果使用控件的标准输出，应用程序会自动适应这些控件的新版本。

自绘制菜单项

与传统的Windows API方法相比，使用VCL开发图形菜单项是很简单的。程序员可以将菜单项组件的OwnerDraw属性设置为True，并处理它的OnMeasureItem与OnDrawItem事件。在OnMeasureItem事件中可以确定菜单项的大小，当显示下拉式菜单时，需要分别为每个菜单项激活这个事件处理程序。该事件处理程序有两个可以设置的引用参数：Width和Height。在OnDrawItem事件中，绘制了实际的图像。每当必须重新绘制一个菜单项时，会激活OnDrawItem事件。这发生在Windows第一次显示菜单项及每次改变状态时，例如，当将鼠标移到一个菜单项上时，它应该变成高亮状态。

为了绘制菜单项，必须考虑所有的可能性，包括使用特殊颜色高亮显示菜单项、需要绘制复选标记等等。非常幸运的是，Delphi事件会向处理程序Canvas传递绘图位置，输出矩形和选项的状态（选中或未选中）。在ODMenu范例中，将处理高亮显示颜色，但略过了一些高级特性（如复选标记）。在程序中设置了菜单的OwnerDraw属性，并为一些菜单项编写了处理程序。为了编写三个与颜色有关的菜单项的事件处理程序，我已在窗体的OnCreate事件处理程序中将它们的Tag属性设置为实际颜色的值。这使得菜单项OnClick事件的处理程序变得非常简单：

```
procedure TForm1.ColorClick(Sender: TObject);
begin
  ShapeDemo.Brush.Color := (Sender as TComponent).Tag
end;
```

OnMeasureItem事件的处理程序不依靠实际菜单项，而是使用一个固定值（不同于其他下拉式菜单的处理程序）。该代码最重要的部分在OnDrawItem事件的处理程序中。对于颜

色，使用了标志的值来绘制给定颜色的矩形区，如图5.11所示。然而，在此之前，必须使用标准颜色为所选菜单项（clHighlight）的菜单（clMenu）充填背景色（作为参数传递的矩形区）：

```
procedure TForm1.ColorDrawItem(Sender: TObject; ACanvas: TCanvas;
  ARect: TRect; Selected: Boolean);
begin
  // set the background color and draw it
  if Selected then
    ACanvas.Brush.Color := clHighlight
  else
    ACanvas.Brush.Color := clMenu;
  ACanvas.FillRect (ARect);
  // show the color
  ACanvas.Brush.Color := (Sender as TComponent).Tag;
  InflateRect (ARect, -5, -5);
  ACanvas.Rectangle (ARect.Left, ARect.Top, ARect.Right, ARect.Bottom);
end;
```

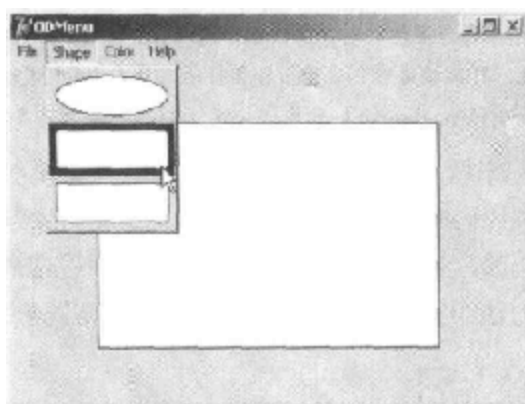


图5.11 ODMeno范例的自绘制菜单

Shape下拉式菜单项中该事件的三个处理程序各不相同，尽管它们使用了相似的代码：

```
procedure TForm1.Ellipse1DrawItem(Sender: TObject; ACanvas: TCanvas;
  ARect: TRect; Selected: Boolean);
begin
  // set the background color and draw it
  if Selected then
    ACanvas.Brush.Color := clHighlight
  else
    ACanvas.Brush.Color := clMenu;
  ACanvas.FillRect (ARect);
  // draw the ellipse
  ACanvas.Brush.Color := clWhite;
  InflateRect (ARect, -5, -5);
  ACanvas.Ellipse (ARect.Left, ARect.Top, ARect.Right, ARect.Bottom);
end;
```

说明：为了容纳Windows 2000用户界面样式中不断增多的语句，Delphi为其菜单引入了新的OnAdvancedDrawItem事件。

颜色列表框

我们已经知道，菜单与列表框都有自绘制特性，这意味着程序可以绘制列表框数据项。Delphi也为组合框提供了同样的支持，也可在CLX上获得。要建立自绘制列表框，首先需要将它的Style属性设置为lbOwnerDrawFixed或lbOwnerDrawVariable。第一个值说明：我们打算通过指定ItemHeight属性来设置列表框数据项的高度，并且这将是每个数据项的高度。第二个自绘制类型指定了一个具有不同高度的数据项列表框。在此例中，组件将触发每个数据项的OnMeasureItem事件，用以向程序查询它们的高度。

在ODList（和QODList版本）范例中，使用了第一种方法，也是更简单的方式。该范例的目的是将颜色信息与列表框数据项存储在一起，然后绘制彩色的数据项（替代单一颜色的列表）。

每个窗体的DFM或XFM文件，包括这个，都有一个TextHeight属性，它说明了显示文本所需的像素量。应该将该值用于列表框的ItemHeight属性。另一种方法是在运行时计算该值，这样，如果之后在设计时改变字体，就不必注意相应地设置项目的高度了。

说明：刚才将TextHeight描述为窗体的特征值，而不是属性。但它确实不是属性，而是窗体的位置值。如果它不是属性，可能读者会问，Delphi怎样将它保存在DFM文件中？答案是，Delphi的流机制是基于属性和由DennePropenks对象方法建立的特殊属性副本的。

因为TextHeight不是属性，所以尽管也将其列入了窗体的文本描述中，但是，还是不能直接访问它。研究VCL源代码能够发现通过调用窗体的专用对象方法，GetTextHeight，可以计算该值。因为它是专用的，所以不能调用该函数。但可以在选择了列表框的字体之后，在窗体的FormCreate对象方法中复制它的代码（其实代码非常简单）：

```
Canvas.Font := ListBox1.Font;  
ListBox1.ItemHeight := Canvas.TextHeight('0');
```

然后要做的事情是，向列表框添加一些数据项。因为这是一个彩色列表框，所以打算向列表框的Items添加颜色名，并向与每个列表数据项有关的Objects数据存储器添加相应的颜色值。没必要分别添加这两个值，我用了这个过程下面向列表添加数据项：

```
procedure TODListForm.AddColors (Colors: array of TColor);  
var  
  I: Integer;  
begin  
  for I := Low (Colors) to High (Colors) do  
    ListBox1.Items.AddObject (ColorToString (Colors[I]), TObject(Colors[I]));  
end;
```

该对象方法使用了一个开放式数组参数，即一个任意多相同类型元素的数组。对于每个作为参数传递的数据项，我们均要调用AddObject对象方法来向列表添加颜色名，并向相应的数据添加颜色值。通过调用ColorToString函数可以得到与颜色对应的字符串，该函数返回一个带有颜色常量的字符串，或者说是带有十六进制颜色值的字符串。将颜色数据值转换为TObject数据类型（这是AddObject对象方法需要的四字节引用）后，再将其添加到列表框中。

提示：除了ColorToString（它将一个颜色值转换为带有标识符或十六进制值的相应字符串），还有一个Delphi函数，StringToColor，用于将一定格式的字符串转换为颜色。

在ODList范例中，窗体的OnCreate事件处理程序调用了该对象方法（在设置数据项的高度后）：

```
AddColors ([clRed, clBlue, clYellow, clGreen, clFuchsia, clLime, clPurple,  
            clGray, RGB (213, 23, 123), RGB (0, 0, 0), clAqua, clNavy, clOlive, clTeal]);
```

为了编辑该代码的CXL版本，在本章先前描述的“Colors”中笔者添加了RGB函数。用于绘制数据项的代码并不太复杂。只不过必须检索与数据项相关的颜色，并将它设置为字体的颜色，然后绘制文本：

```
procedure TODListForm.ListBox1DrawItem(Control: TWinControl; Index: Integer;  
    Rect: TRect; State: TOwnerDrawState);  
begin  
    with Control as TListBox do  
    begin  
        // erase  
        Canvas.FillRect(Rect);  
        // draw item  
        Canvas.Font.Color := TColor (Items.Objects [Index]);  
        Canvas.TextOut(Rect.Left, Rect.Top, Listbox1.Items[Index]);  
    end;  
end;
```

该系统已经设置了相应的背景颜色，所以即使没有任何附加代码，也会相应地显示所选项。此外，可以通过在列表框上双击来添加新的项：

```
procedure TODListForm.ListBox1DbClick(Sender: TObject);  
begin  
    if ColorDialog1.Execute then  
        AddColors ([ColorDialog1.Color]);  
end;
```

如果读者试着使用该功能会发现，所添加的有些颜色被转换为颜色名（Delphi颜色常量），而有些则被转换为十六进制数。

List View与Tree View控件

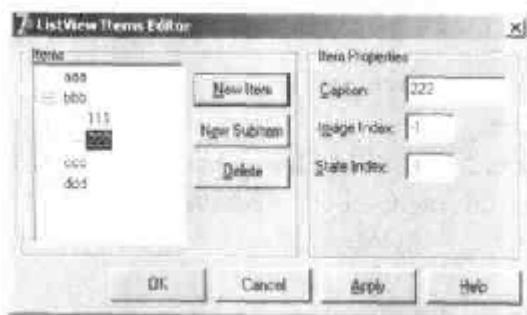
在前面“打开组件工具框”一节中介绍了所有能用来显示列表值的可视控件。标准列表框与组合框组件仍是最常用的，但它们经常被更强大的ListView和TreeView控件所取代。ListView和TreeView是Win32普通控件的一部分，存储在ComCtl32.DLL库中。并且在Windows与Linux上，相似的控件也可用于Qt与VisualCLX。

图形索引列表

当使用ListView组件时，可以提供位图来指定元素（例如，被选项）的状态，并用图形方式来描述选项的内容。

为了将图像与list或tree组件相连，需要求助于ImageList组件，前面已经对菜单的图像应用过该组件。ListView组件实际可以拥有三个图像列表，一个用于大图标（LargeImages属性），一个用于小图标（SmallImages属性），一个用于项目的状态（StateImages属性）。在RefList范例中，笔者使用两种不同的ImageList组件设置了头两个属性。

ListView的每一项都拥有ImageIndex，引用着列表中的图像。为了使其能正常运行，两个图像列表中的元素顺序应该相同。当准备好一个图像列表时，就可以使用Delphi的ListView Item编辑器（与Items属性相连）向列表添加图像了。在该编辑器中，可以定义项与子项。子项只能在详细视图中显示（将Viewstyle属性设置为vsReport时）并与Columns属性中设置的标题相连。



警告：在CLX中的ListView控件没有小/大图标。在Qt中用IconView组件来实现。

在RefList范例（对书籍、杂志、光碟、Web站点引用的一个简单列表）中，选项被存储在一个文件中，因为程序的用户可以编辑列表的内容，当程序退出时，列表会自动被保存。这样，由用户来完成的编辑就不会变了。保存与装载ListView的内容并不轻松，因为TListItem类型没有自动保存数据的机制。另一种简单的方法是，使用一种定制的格式，从或向一个字符串列表中复制数据。然后将字符串列表保存到一个文件中，并使用相同的命令重新装载。

文件格式是简单的，从下列保存的代码中可以看出这一点。对于列表的每一项，程序会在一行上保存标题，而在另一行上保存图像索引（前缀加字符@），最后在随后的行上保存子项，用制表符进行缩进：

```
procedure TForm1.FormDestroy(Sender: TObject);
var
  I, J: Integer;
  List: TStringList;
begin
  // store the items
  List := TStringList.Create;
  try
    for I := 0 to ListView1.Items.Count - 1 do
```

```

begin
    // save the caption
    List.Add (ListView1.Items[I].Caption);
    // save the index
    List.Add ('@' + IntToStr (ListView1.Items[I].ImageIndex));
    // save the subitems (indented)
    for J := 0 to ListView1.Items[I].SubItems.Count - 1 do
        List.Add (#9 + ListView1.Items[I].SubItems [J]);
    end;
    List.SaveToFile (ExtractFilePath (Application.ExeName) + 'Items.txt');
finally
    List.Free;
end;
end;

```

然后,将各项目加载到FormCreate方法中:

```

procedure TForm1.FormCreate(Sender: TObject);
var
    List: TStringList;
    NewItem: TListItem;
    I: Integer;
begin
    // stops warning message
    NewItem := nil;
    // load the items
    ListView1.Items.Clear;
    List := TStringList.Create;
    try
        List.LoadFromFile (
            ExtractFilePath (Application.ExeName) + 'Items.txt');
        for I := 0 to List.Count - 1 do
            if List [I][1] = #9 then
                NewItem.SubItems.Add (Trim (List [I]))
            else if List [I][1] = '@' then
                NewItem.ImageIndex := StrToIntDef (List [I][2], 0)
            else
                begin
                    // a new item
                    NewItem := ListView1.Items.Add;
                    NewItem.Caption := List [I];
                end;
            finally
                List.Free;
            end;
        end;
    end;
end;

```

该程序中含有一个菜单，可以用它来选择不同的视图（由ListView控件支持）以及向选项添加复选框，就像在CheckBox中一样。读者可以在图5.12中看到这些样式的不同组合

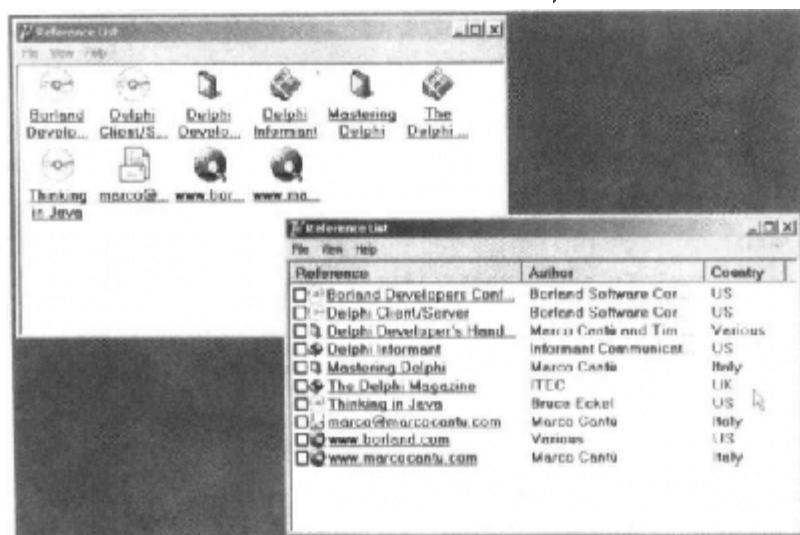


图5.12 RefList范例中ListView组件的不同输出实例，通过改变ViewStyle属性并添加复选框获得

该程序的另一个重要特性是允许用户将选项划分到不同的列中，这在控件的详细或报告视图中是常用的。在VCL中这个技术需要三步操作。第一步是将ListView的SortType属性设置为stBoth或stData。这样，ListView将不会基于标题进行分类，而是为必须进行分类的每项调用OnCompare事件。

第二步，因为要在详细视图的每列上进行分类，所以还需要处理OnColumnClick事件（当用户在详细视图中单击列标题时会触发该事件，但只有在ShowColumnHeaders属性被设置为True时）。每当单击某一列时，程序都会将该列的序号保存到窗体类的nSortCol专用元素中去：

```
procedure TForm1.ListView1ColumnClick(Sender: TObject;
  Column: TListColumn);
begin
  nSortCol := Column.Index;
  ListView1.AlphaSort;
end;
```

然后是第三步，分类代码使用与当前分类列相对应的标题或一个子项：

```
procedure TForm1.ListView1Compare(Sender: TObject;
  Item1, Item2: TListItem; Data: Integer; var Compare: Integer);
begin
  if nSortCol = 0 then
    Compare := CompareStr (Item1.Caption, Item2.Caption)
  else
    Compare := CompareStr (Item1.SubItems [nSortCol - 1],
      Item2.SubItems [nSortCol - 1]);
end;
```

在程序 (QRefList) 的 CLX 版本中, 不需要任何预先步骤。当单击控件的标题时, 控件能够适当地对自己分类。可以得到多个栏的自动分类 (包括升序和降序)。

向程序添加的最后一个特征与鼠标操作有关。当用户单击一个选项时, RefList 程序会显示该选项的说明。右键单击选项可以将它设置为编辑模式, 用户可以改变它 (当程序结束时, 所做改变将被自动保存)。下面是 ListView 控件的 OnMouseDown 事件处理程序中的有关代码:

```

procedure TForm1.ListView1MouseDown(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
var
  strDescr: string;
  I: Integer;
begin
  // if there is a selected item
  if ListView1.Selected <> nil then
    if Button = mbLeft then
      begin
        // create and show a description
        strDescr := ListView1.Columns [0].Caption + #9 +
          ListView1.Selected.Caption + #13;
        for I := 1 to ListView1.Selected.SubItems.Count do
          strDescr := strDescr + ListView1.Columns [I].Caption + #9 +
            ListView1.Selected.SubItems [I-1] + #13;
        ShowMessage (strDescr);
      end
    else if Button = mbRight then
      // edit the caption
      ListView1.Selected.EditCaption;
    end;

```

尽管还有未实现的特性, 但该范例展示了 ListView 控件的一些潜力。此外, 该程序还实现了 Windows 98 的 “hot-tracking (热追踪)” 特性, 它可以突出显示列表视图并向鼠标下的选项加下划线。ListView 的有关属性都列在下面的文本描述中:

```

object ListView1: TListView
  Align = alClient
  Columns = <
    item
      Caption = 'Reference'
      Width = 230
    end
    item
      Caption = 'Author'
      Width = 180
    end
  item

```



```

Caption = 'Country'
Width = 80
end>
Font.Height = -13
Font.Name = 'MS Sans Serif'
Font.Style = [fsBold]
FullDrag = True
HideSelection = False
HotTrack = True
HotTrackStyles = [htHandPoint, htUnderlineHot]
SortType = stBoth
ViewStyle = vsList
OnColumnClick = ListView1ColumnClick
OnCompare = ListView1Compare
OnMouseDown = ListView1MouseDown
end

```

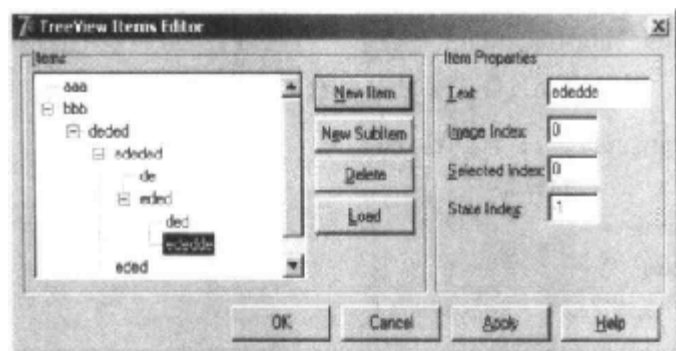
实际上，该程序非常有趣，本书将在第9章中进一步扩展它。

为了建立它的CLX版本，QRefList不得不只使用一个图像列表，禁用小图像和大图像菜单，就像ListView被限制到列表一样，并报告浏览风格。许多控制框都提供了叫做IconView的大图标和小图标。正如前面提到的，已经使用的分类支持可以保存大部分的示例代码。

数据的树形结构

前面已经研究了基于ListView组件的范例，程序员可以检测TreeView控件。TreeView控件有一个灵活的功能很强的用户界面（支持对元素的编辑与拖动），而且比ListView的用户界面更为标准，因为它是Windows Explorer的用户界面。有大量属性与各种不同的方法可以用来定制每一行或每一类行的位图。

为了在设计时定义TreeView的节点结构，可以使用TreeView的Items属性编辑器，如下所示。



然而，在本例中，我们决定在启动时将它装载到TreeView的数据中，方法与上一个范例相似。

TreeView组件的Items的属性有许多成员函数，可以用来改变字符串的层次结构。例如，用下列代码建立一个两级的树形结构：

```

var
  Node: TTreeNode;

```

```
begin
  Node := TreeView1.Items.Add (nil, 'First level');
  TreeView1.Items.AddChild (Node, 'Second level');
```

使用这两个Add与AddChild方法，可以在运行时建立复杂的结构。为了装载信息，这里再次在运行时使用了StringList，将信息装入一个文本文件，并分析它。

然而，因为TreeView控件含有LoadFromFile对象方法，所以DragTree和QDragTree范例使用了下列较简单的代码：

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  TreeView1.LoadFromFile (ExtractFilePath (Application.ExeName) +
    'TreeText.txt');
end;
```

LoadFromFile对象方法主要将数据装载到字符串列表中，并通过查看制表符的号码，来检查每一项的级别（如果读者愿意，可以查看Delphi VCL源代码中ComCtrls单元内的TTreeStrings.GetBufStart方法）。而且，为TreeView提供的数据是一个跨国公司的组织结构表，如图5.13所示。

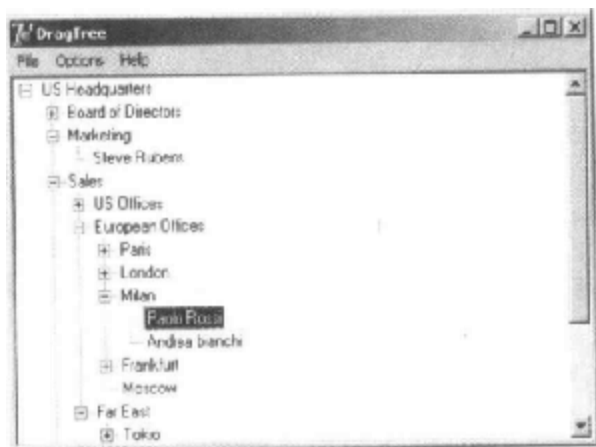


图5.13 在DragTree范例中，加载数据并展开分支

不用分别展开各节点项，可以使用程序的File►Expand All菜单，它调用TreeView控件的FullExpand方法或执行相应的代码（有带根的树的情况下）：

```
TreeView1.Items [0].Expand(True);
```

除了装载数据外，当程序终止时还需保存数据。该程序有一些菜单项用于定制TreeView控件的字体并改变其他一些简单设置。在该范例中专门实现的特性是为选项与整个子树提供拖动功能。我们将组件的DragMode属性设置为dmAutomatic，并为OnDragOver与OnDragDrop事件编写了处理程序。

在第一个事件处理程序中，程序确保了用户不会试图将一个选项拖到子项上（它会随着选项一起移动，导致死循环）：

```
procedure TForm1.TreeView1DragOver(Sender, Source: TObject;
  X, Y: Integer; State: TDragState; var Accept: Boolean);
```

```

var
  TargetNode, SourceNode: TTreeNode;
begin
  TargetNode := TreeView1.GetNodeAt (X, Y);
  // accept dragging from itself
  if (Source = Sender) and (TargetNode <> nil) then
  begin
    Accept := True;
    // determines source and target
    SourceNode := TreeView1.Selected;
    // look up the target parent chain
    while (TargetNode.Parent <> nil) and (TargetNode <> SourceNode) do
      TargetNode := TargetNode.Parent;
    // if source is found
    if TargetNode = SourceNode then
      // do not allow dragging over a child
      Accept := False;
    end
  else
    Accept := False;
  end;
end;

```

这段代码的效果是（除了需要避免的特殊情况），用户可以将TreeView的一个选项拖到另一个选项上。为选项移动操作编写实际代码是很简单的，因为TreeView控件为该操作提供了支持，可以使用TTreeNode类的MoveTo对象方法：

```

procedure TForm1.TreeView1DragDrop(Sender, Source: TObject; X, Y: Integer);
var
  TargetNode, SourceNode: TTreeNode;
begin
  TargetNode := TreeView1.GetNodeAt (X, Y);
  if TargetNode <> nil then
  begin
    SourceNode := TreeView1.Selected;
    SourceNode.MoveTo (TargetNode, naAddChildFirst);
    TargetNode.Expand (False);
    TreeView1.Selected := TargetNode;
  end;
end;

```

说明：在Delphi自带的演示程序中，有一个有趣的范例，它演示了自制的TreeView控件。该范例位于CustomDraw子目录中。

DragTree的可移植版本

因为我在很多的移植演示中都要使用这个程序，故我使用Delphi建立了一个可以编译为VCL应用程序的版本，并且使用Kylix建立了一个CLX应用程序。这也是与本书中其他程序

的不同之处，它可以通过使用VisualCLX被移植到Delphi或者是Windows上的Qt中。下面是另一个方法，在某些时候可能是必需的。

我们要做的第一件事是利用条件化编译使用uses语句的两个不同集合。PortableDrag-Tree范例的单元以如下方式开始：

```
unit TreeForm;

interface

uses
  SysUtils, Classes,

  {$IFDEF LINUX}
    Qt, Libc, QGraphics, QControls, QForms, QDialogs,
    QStdCtrls, QComCtrls, QMenus, QTypes, QGrids;
  {$ENDIF}

  {$IFDEF MSWINDOWS}
    Windows, Graphics, Controls, Forms, Dialogs,
    StdCtrls, ComCtrls, Menus, Grids;
  {$ENDIF}
```

一个类似的指令用在实现小节的初始化部分，用以包括这个窗体的正确的资源文件，这两个资源文件是不同的：

```
  {$IFDEF LINUX}
    {$R *.xfm}
  {$ENDIF}

  {$IFDEF MSWINDOWS}
    {$R *.dfm}
  {$ENDIF}
```

我们忽略了一些Windows的专有特性，因此代码中的惟一区别就是FormCreate方法。该程序从用户的默认文件夹装入数据文件，而不是从可执行文件所在的文件夹。根据操作系统的类型，用户的文件夹可能是主文件夹（隐含文件以一个点开头），或者是一个特定的My Documents区域（使用特殊的API调用）：

```
procedure TForm1.FormCreate(Sender: TObject);
var
  path: string;
begin
  {$IFDEF LINUX}
    filename := GetEnvironmentVariable('HOME') +
      '\.TreeText.txt';
  {$ELSE}
    SetLength (path, 100);
    ShGetSpecialFolderPath (Handle, PChar(path),
      CSIDL_PERSONAL, False);
    path := PChar (path); // fix string length
    filename := path + '\TreeText.txt';
```

```

($ENDIF)
TreeView1.LoadFromFile (filename);
end;

```

定制树节点

Delphi 6向TreeView控件添加了几个新特征, 包括多重选择器(参见MultiSelect¹与MultiSelectStyle属性和Selections数组)、改良的种类与几个新事件。创建定制树节点项的另一个好处是, 它有助于向每个节点添加额外的定制信息, 并且可以创建不同类的节点。能定制节点项意味着可以用简单的、面向对象的方式向节点添加定制数据。为了支持该技术, 使用了一种用于TTreeItems类的新的AddNode对象方法和新的OnCreateNodesClass特殊事件。在该事件的处理器中, 返回的类对象必须继承自TTreeNode。

因为这是一种非常通用的技术, 故下面通过一个示例来详细介绍它。CustomNodes示例没有任何实际意义, 但它显示了一种较复杂的情况, 即有两个不同的定制节点类从其他类派生出一个类。基类添加了一个ExtraCode属性, 它被映射到虚拟对象方法中, 子类覆盖了这些对象方法中的一个类。对于基类, GetExtraCode函数简单地返回值, 而对于派生类, 该值将被累加到父类节点值上。下面是这些类和第二种对象方法的代码:

```

type
  TMyNode = class (TTreeNode)
  private
    fExtraCode: Integer;
  protected
    procedure SetExtraCode(const Value: Integer); virtual;
    function GetExtraCode: Integer; virtual;
  public
    property ExtraCode: Integer read GetExtraCode write SetExtraCode;
  end;

  TMySubNode = class (TMyNode)
  protected
    function GetExtraCode: Integer; override;
  end;

function TMySubNode.GetExtraCode: Integer;
begin
  Result := fExtraCode * (Parent as TMyNode).ExtraCode;
end;

```

通过让这些定制树节点类可用, 该程序通过使用第一种类型作为第一层节点以及第二种类型为其他节点而创建了一个项树。因为只有一个OnCreateNodeClass事件处理器, 所以该处理器使用了存储在窗体私有域中的类引用(TTreeNodeClass类型的CurrentNodeClass):

```

procedure TForm1.TreeView1CreateNodeClass(Sender: TCustomTreeView;
  var NodeClass: TTreeNodeClass);
begin
  NodeClass := CurrentNodeClass;
end;

```

例如，在创建每个类型的节点前，程序用下面代码设置类引用：

```
var
  MyNode: TMyNode;
begin
  CurrentNodeClass := TMyNode;
  MyNode := TreeView1.Items.AddChild (nil, 'item' + IntToStr (nValue))
    as TMyNode;
  MyNode.ExtraCode := nValue;
```

一旦整个树被创建，当用户挑选一个项时，就能把它的类型放到TMyNode中，并且访问额外属性（也是方法和数据）：

```
procedure TForm1.TreeView1Click(Sender: TObject);
var
  MyNode: TMyNode;
begin
  MyNode := TreeView1.Selected as TMyNode;
  Label1.Caption := MyNode.Text + ' [' + MyNode.ClassName + '] = ' +
    IntToStr (MyNode.ExtraCode);
end;
```

CustomNodes示例使用该代码将所选节点的描述显示到标签控件中，如图5.14所示。注意当挑选一项到树中时，它的值对于该项的每个父类节点是累加的。尽管有更简单的方法获得该效果，如使用一个来自同一层次不同类的项对象的树型窗口，但用它可提供面向对象的结构，不过使用这种结构需要编写一些复杂的代码。

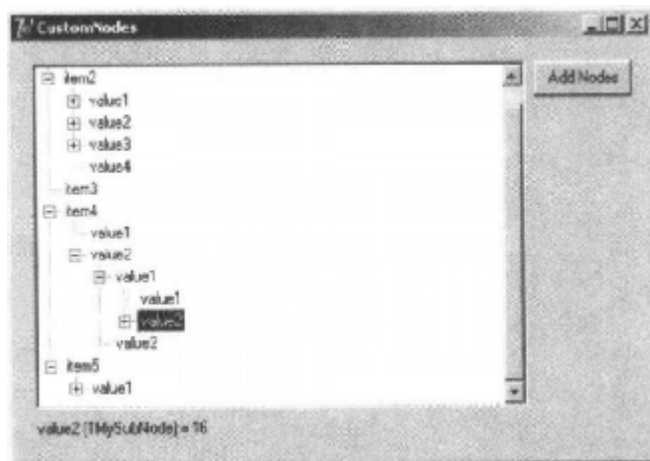


图5.14 有了OnCreateNodesClass事件，CustomNodes范例便有了基于不同定制类的节点对象的树型窗口

小结

在本章中，我们探讨了在Delphi中建立用户界面能够使用的库的基础，Windows VCL与基于Qt的CLX，讨论了TControl类及其属性，以及其最重要的派生类。

然后，我们通过介绍两个库研究了一些在Delphi中可以使用的基礎组件。这些组件对应着标准的Windows控件与一些常用控件，而且它们在应用中是非常一致的。我们还介绍了如何建立主菜单与弹出式菜单，以及如何向这些控件添加图形。

接下来，将进一步深入地介绍一个完整用户界面的各个元素，并讨论动作列表和Action Manager，并创建几个简单但完整的例子。这些是第6章的主题，在第7章将讨论窗体。

第6章 建立用户界面

在第5章中，我们讨论了TControl类的核心概念和VCL与VisualCLX库中相应的类。接下来，我们将对用于建立用户界面的关键控件做快速的介绍，这其中包括编辑组件、列表、范围选择器等。本章还要讨论其他用来定义窗体总体设计的控件，如PageControl和TabControl。在这些组件之后，我们将介绍工具栏和状态栏，包括一些比较高级的特性。这将给我们探讨动作和动作管理器的体系结构等内容提供基础知识。

现代的Windows应用程序通常使用多个方法来提供一个命令，包括菜单项、工具栏按钮、快捷菜单等等。为了将用户能够提供的命令与代表它们的用户界面相分离，Delphi使用了动作（Action）的概念。在最近的Delphi版本中，这个体系结构被扩展为在动作的顶部建立完全可视的用户界面结构。我们现在可以让程序用户轻松地定制这个接口，就像在很多专业程序中那样。最后，Delphi 7添加了支持Action Manager体系结构的可视控件，以及更现代的UI支持，而这是通过支持类似XP的样式和感觉而实现的。在Windows XP中，我们可以创建应用程序，使它们适应激活的主题，这要归功于很多新的内部VCL代码。

本章主要包括以下内容：

- 多页面窗体
- 页与页标
- ToolBars与StatusBars
- 主题与样式
- Actions与ActionList
- Delphi 6中的预定义动作
- ControlBar与CoolBar组件
- 工具栏与其他控件的停放
- Action Manager的体系结构

多页面窗体

当我们有大量信息与控件要显示在某对话框或窗体中时，可以使用多页面。该特性的原理与笔记簿一样：使用标签，供用户选择不同的页面。有两个控件可以用于在Delphi中建立多页应用程序：

- PageControl组件，它在每一页面边上有一个标签，而且有多个页面覆盖于其表面（类似于面板）。由于每页上都有标签，所以只需在设计时与运行时在每个页面上放置组件，就可以获得相应的效果。
- TabControl组件，它只含标签部分，而没有提供页面来管理信息。在这种情况下，需要使用一个或多个组件来模仿页面改变的操作，也可以放置包含标签和模仿页面的窗体。

第三种相关类, **TabSheet**, 表示**PageControl**的一个单页。这是一种独立组件, 没有出现在组件面板上。可以在设计时通过使用**PageControl**的局部菜单, 或在运行时使用同一控件的对象方法建立**TabSheet**。

说明: Delphi仍包含了(在Win3.1组件板的标签上) **Notebook**、**TabSet**和**TabbedNotebook**组件, 这些组件是在16位版本(从Delphi 2开始)时引入的。为了其他用途, **PageControl**和**TabControl**组件(封装了Win32常用控件)提供了更现代的用户界面。实际上, 在32位版本的Delphi中, 可以在内部使用Win32 **PageControl**重新实现**TabbedNotebook**组件, 以减少代码量并更新外观。

PageControls与TabSheels

与前面一样, 我们并不是想复制属性的**Help**系统列表与**PageControl**组件的对象方法, 而是建立一个范例来扩展它的功能, 并允许用户在运行时改变它的动作。该范例名为**Pages**, 包含一个拥有三个页面的**PageControl**控件。程序清单6.1给出了**PageControl**与其他主要组件的结构。

程序清单6.1 Pages范例的DFM文件的关键部分

```
object Form1: TForm1
  BorderIcons = [biSystemMenu, biMinimize]
  BorderStyle = bsSingle
  Caption = 'Pages Test'
  OnCreate = FormCreate
object PageControl1: TPageControl
  ActivePage = TabSheet1
  Align = alClient
  HotTrack = True
  Images = ImageList1
  MultiLine = True
object TabSheet1: TTabSheet
  Caption = 'Pages'
  object Label3: TLabel
  object ListBox1: TListBox
  end
object TabSheet2: TTabSheet
  Caption = 'Tabs Size'
  ImageIndex = 1
  object Label1: TLabel
  // other controls
  end
object TabSheet3: TTabSheet
  Caption = 'Tabs Text'
  ImageIndex = 2
  object Memo1: TMemo
    Anchors = [akLeft, akTop, akRight, akBottom]
    OnChange = Memo1Change
```

```

end
object BitBtnChange: TBitBtn
  Anchors = [akTop, akRight]
  Caption = '&Change'
end
end
object BitBtnPrevious: TBitBtn
  Anchors = [akRight, akBottom]
  Caption = '&Previous'
  OnClick = BitBtnPreviousClick
end
object BitBtnNext: TBitBtn
  Anchors = [akRight, akBottom]
  Caption = '&Next'
  OnClick = BitBtnNextClick
end
object ImageList1: TImageList
  Bitmap = {...}
end
end

```

注意，与位图相连的标签由ImageList控件提供，而且一些控件使用了Anchors属性保持与窗体右下角的固定距离。即使窗体不支持改变大小（由于建立了这么多控件，改变大小就太复杂了），当标签分多行（只增加标题的长度）显示或显示在窗体左端时，位置也会改变。

每个TabSheet对象都有自己的Caption，作为页面的标签显示。在设计时可以使用同样的局部菜单建立新页面或在两个页面间移动。可以在图6.1中看到PageControl组件的局部菜单，它带有第一个页面标签，该标签保存了一个列表框与一个小标题，并与其他页面一起共享两个按钮。

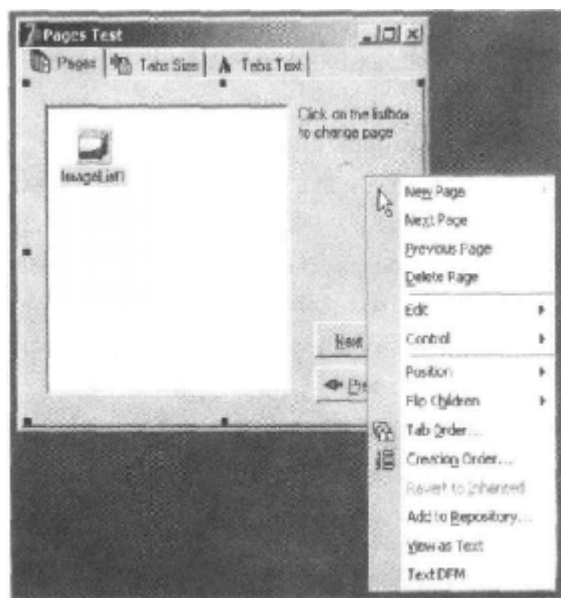


图6.1 Pages范例中PageControl的第一页及其弹出菜单

如果在一个页面上放置一个组件，那么它只能在该页面上使用。怎样才能在每个页面上都能使用相同的组件（在本例中是两个位图按钮），而又无需复制它呢？只需在窗体上放置组件，要放在PageControl的外边（或在将它与客户区对齐之前），然后调用窗体弹出菜单的Bring To Front命令，将它移到页面的前面。放置在每个页面上的两个按钮可以用于前后各个页面，并且用相同的方法使用标签。下面是与它们有关的一段代码：

```
procedure TForm1.BitBtnNextClick(Sender: TObject);
begin
    PageControl1.SelectNextPage (True);
end;
```

其他按钮调用了同样的过程，该过程将False作为它的参数传递以选择前一个页面。注意，不需要检测是否处于第一个页面或最后一个页面，因为SelectNextPage对象方法将最后一个页面作为第一个页面的前一个页面，并直接在两个页面之间移动。

再注意一下第一个页面。它有一个列表框，运行时它将保存标签的名称。如果用户单击该列表框的一个选项，当前页面会改变，这是用于改变页面的第三个对象方法（在标签与Next或Previous按钮之后）。列表框是在FormCreate对象方法中充填的，与窗体的OnCreate事件相连，并复制每个页面的标题（Page属性存储了TabSheet对象的一个列表）：

```
for I := 0 to PageControl1.PageCount - 1 do
    ListBox1.Items.Add (PageControl1.Pages.Caption);
```

当单击列表中的一项时，可选择相应的页：

```
procedure TForm1.ListBox1Click(Sender: TObject);
begin
    PageControl1.ActivePage := PageControl1.Pages [ListBox1.ItemIndex];
end;
```

第二个页面有两个编辑框（与两个UpDown组件相连）、两个复选框及两个单选钮，如图6.2所示。用户可以输入一个数值（或通过单击向上箭头键与向下箭头键或当焦点在相应编辑框上时按键来选择它）并选择相应的框或单选钮，然后单击Apply按钮来做出改变：

```
procedure TForm1.BitBtnApplyClick(Sender: TObject);
begin
    // set tab width, height, and lines
    PageControl1.TabWidth := StrToInt (EditWidth.Text);
    PageControl1.TabHeight := StrToInt (EditHeight.Text);
    PageControl1.MultiLine := CheckBoxMultiLine.Checked;
    // show or hide the last tab
    TabSheet3.TabVisible := CheckBoxVisible.Checked;
    // set the tab position
    if RadioButton1.Checked then
        PageControl1.TabPosition := tpTop
    else
        PageControl1.TabPosition := tpLeft;
end;
```

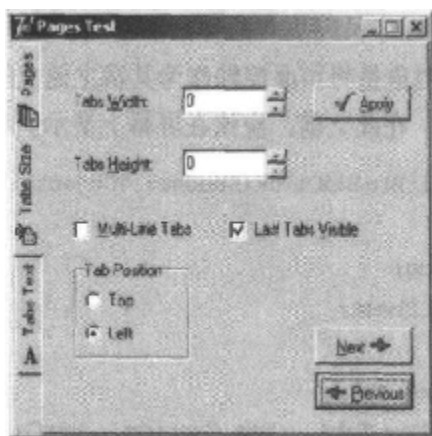


图6.2 范例的第二页面可以用于确定标签的大小与位置。从中读者可以看到Page控件左边的标签

使用这段代码可以改变每个页面的宽度与高度（记住0意味着从每个字符串所占的空间中自动计算尺寸），并选择标签的多行或两个小箭头来滚动标签区域，而且将它们移到窗口的左边。该控件还允许将标签放置到底部或右边；但该程序不允许这样，因为这会使其他控件的放置变得非常复杂。

还可以在PageControl上隐藏最后一个标签，它对应着TabSheet3组件。如果将标签的TabVisible属性设置为False来隐藏该标签，那么单击Next与Previous按钮就不能使用它，这两个按钮基于SelectNextPage对象方法。而这里使用的FindNextPage函数，即使标签不可见也将选中该页。FindNextPage函数的调用效果正如Next按钮组件中OnClick事件处理程序的新版本所示的一样：

```
procedure TForm1.BitBtnNextClick(Sender: TObject);
begin
  PageControl1.ActivePage := PageControl1.FindNextPage (
    PageControl1.ActivePage, True, False);
end;
```

最后一个页面含有一个备注组件，并带有页面的名称（在FormCreate方法中添加的）。可以编辑页面的名称并单击Change按钮来改变标签的文本，但字符串的序数必须与标签的序数匹配：

```
procedure TForm1.BitBtnChangeClick(Sender: TObject);
var
  I: Integer;
begin
  if Memo1.Lines.Count <> PageControl1.PageCount then
    MessageDlg ('One line per tab, please', mtError, [mbOK], 0)
  else
    for I := 0 to PageControl1.PageCount - 1 do
      PageControl1.Pages [I].Caption := Memo1.Lines [I];
    BitBtnChange.Enabled := False;
end;
```

最后, 按钮Add Page允许程序员向页面控件添加新的标签页, 尽管程序没有向其添加任何组件。该标签页(空的)对象是将页面控件作为其宿主建立的, 但它不能使用, 除非还设置了PageControl属性。然而, 在此之前, 应该在屏幕上显示新标签页。下面是有关代码:

```
procedure TForm1.BitBtnAddClick(Sender: TObject);
var
  strCaption: string;
  NewTabSheet: TTabSheet;
begin
  strCaption := 'New Tab';
  if InputQuery ('New Tab', 'Tab Caption', strCaption) then
  begin
    // add a new empty page to the control
    NewTabSheet := TTabSheet.Create (PageControl1);
    NewTabSheet.Visible := True;
    NewTabSheet.Caption := strCaption;
    NewTabSheet.PageControl := PageControl1;
    PageControl1.ActivePage := NewTabSheet;
    // add it to both lists
    Memo1.Lines.Add (strCaption);
    ListBox1.Items.Add (strCaption);
  end;
end;
```

提示: 无论何时建立基于PageControl的窗体, 都需要记住, 运行时第一个显示的页面是代码编译之前正在操作的页面。这意味着如果正位于第三个页面, 然后编译并运行程序, 则该程序将首先显示第三个页面。解决该问题的一个常用的方法是, 在FormCreate对象方法中添加一行代码, 将PageControl或笔记本设置到第一页。利用该方法, 设计时的当前页面将不决定运行时的初始页面。

带有自绘制标签的图像查看器

上一个范例中介绍了TabControl以及一种动态方法的用法, 该方法还可以应用到更多(更简单)的情况中。每当需要多个含有相同类型内容的页面时, 就可以使用TabControl, 并在选定新标签时改变其内容。这个范例, BmpViewer, 包含了多页面位图查看器中要实现的功能。出现在范例窗体TabControl(与窗体的整个客户区对齐)中的图像, 要根据用户所选标签而定(如图6.3所示)。

开始时, TabControl是空的。当用户选择File菜单下的Open命令后, 便可以在File Open对话框中选择一些文件, 而且带有文件名的字符串数组(OpenDialog1组件的Files属性)被用做标签的文本(TabControl1的Tabs属性):

```
procedure TFormBmpViewer.Open1Click(Sender: TObject);
begin
  if OpenDialog1.Execute then
  begin
    TabControl1.Tabs.AddStrings (OpenDialog1.Files);
```

```

TabControl1.TabIndex := 0;
TabControl1Change (TabControl1);
end;
end;

```

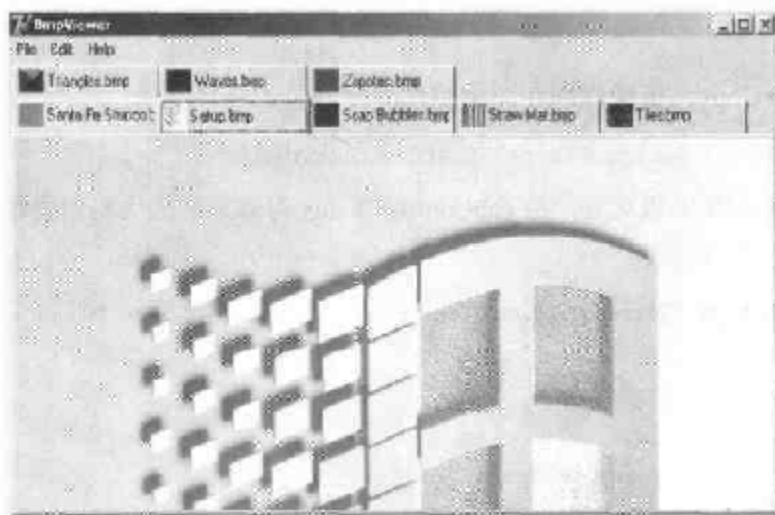


图6.3 BmpViewer范例中位图查看器的界面，注意自绘制标签

警告：在CLX中，TabControl的Tabs属性是个集合，而在VCL中是简单的字符串列表。

在显示新标签之后，必须更新图像，以使其与第一个标签匹配。为了实现该目的，程序调用了与TabControl的OnChange事件相连的对象方法，它将把与当前标签相关的文件装载到图像组件中：

```

procedure TFormBmpViewer.TabControl1Change(Sender: TObject);
begin
  Image1.Picture.LoadFromFile (TabControl1.Tabs [TabControl1.TabIndex]);
end;

```

该范例可以运行，除非选择的文件没有包含一幅位图。程序将使用一个标准异常信息来警告用户，忽略文件，并继续它的运行。

程序也允许在剪粘板上粘贴位图（尽管没有实际复制它，而是仅仅在选中时添加一个执行粘贴的标签）并且将当前位图复制到程序中。在Delphi上可通过定义在ClipBrd单元的全局Clipboard对象获得剪贴板支持。为了复制或粘贴位图，可以使用TClipboard与TBitmap类的Assign对象方法。当在示例中选择Edit►Paste命令时，一个叫做Clipboard的新标签将被添加到标签集中（除非它已经存在了）。然后新标签的编号被用来改变活动标签：

```

procedure TFormBmpViewer.Paste1Click(Sender: TObject);
var
  TabNum: Integer;
begin
  // try to locate the page
  TabNum := TabControl1.Tabs.IndexOf ('Clipboard');
  if TabNum < 0 then

```

```

        // create a new page for the Clipboard
        TabNum := TabControl1.Tabs.Add ('Clipboard');
        // go to the Clipboard page and force repaint
        TabControl1.TabIndex := TabNum;
        TabControl1Change (Self);
    end;

```

选择Edit►Copy, 该操作和在图像控件上复制位图一样简单:

```
Clipboard.Assign (Image1.Picture.Graphic);
```

由于可能需要显示剪贴板标签, 故TabControlChange对象方法的代码需要做相应的修改, 如下所示:

```

procedure TFormBmpViewer.TabControl1Change(Sender: TObject);
var
    TabText: string;
begin
    Image1.Visible := True;
    TabText := TabControl1.Tabs [TabControl1.TabIndex];
    if TabText <> 'Clipboard' then
        // load the file indicated in the tab
        Image1.Picture.LoadFromFile (TabText)
    else
        {if the tab is 'Clipboard' and a bitmap
         is available in the clipboard}
        if Clipboard.HasFormat (cf_Bitmap) then
            Image1.Picture.Assign (Clipboard)
        else
            begin
                // else remove the clipboard tab
                TabControl1.Tabs.Delete (TabControl1.TabIndex);
                if TabControl1.Tabs.Count = 0 then
                    Image1.Visible := False;
            end;
    end;

```

每次改变标签时, 该程序都从剪贴板粘贴位图。程序在一次只存储一个图像, 它没有办法来存储剪贴板位图。然而, 如果剪贴板中的内容发生改变, 将无法获得位图格式, 剪贴板标签被自动删除 (如上所示)。如果没有更多的标签被留下, 图像内容将被隐藏。

使用Cut或Delete菜单命令, 图像也能被删除。在复制一个位图到剪贴板后, 使用Cut将删除标签。实际上, Cut1Click对象方法除了调用Copy1Click和Delete1Click对象方法外, 什么也没做。Copy1Click对象方法负责将当前图像复制到剪贴板中, Delete1Click只简单地删除当前标签。下面是它们的代码:

```

procedure TFormBmpViewer.Copy1Click(Sender: TObject);
begin
    Clipboard.Assign (Image1.Picture.Graphic);
end;

```

```

procedure TFormBmpViewer.Delete1Click(Sender: TObject);
begin
  with TabControl1 do
  begin
    if TabIndex >= 0 then
      Tabs.Delete (TabIndex);
    if Tabs.Count = 0 then
      Image1.Visible := False;
    end;
  end;

```

该范例的一个特性是，TabControl将其OwnerDraw属性设置为True。这意味着，该控件不能绘制标签（在设计时是空的），但将通过调用OnDrawTab事件由应用程序来完成该操作。在程序代码中，它使用了DrawText API函数显示了纵向居中的文本。显示的文本不是文件的完整路径，而只是文件名。然后，如果文本不是None，程序会读取标签所引用的位图并在标签自身上绘制该位图。为了实现该操作，程序使用了TabBmp对象，它是TBitmap类型的对象，与窗体一起建立与解除。程序还使用了BmpSide常量以确定位图与文本的位置：

```

procedure TFormBmpViewer.TabControl1DrawTab(Control: TCustomTabControl;
  TabIndex: Integer; const Rect: TRect; Active: Boolean);
var
  TabText: string;
  OutRect: TRect;
begin
  TabText := TabControl1.Tabs [TabIndex];
  OutRect := Rect;
  InflateRect (OutRect, -3, -3);
  OutRect.Left := OutRect.Left + BmpSide + 3;
  DrawText (Control.Canvas.Handle, PChar (ExtractFileName (TabText)),
    Length (ExtractFileName (TabText)), OutRect,
    dt_Left or dt_SingleLine or dt_VCenter);
  if TabText = 'Clipboard' then
    if Clipboard.HasFormat (cf_Bitmap) then
      TabBmp.Assign (Clipboard)
    else
      TabBmp.FreeImage
    else
      TabBmp.LoadFromFile (TabText);
  OutRect.Left := OutRect.Left - BmpSide - 3;
  OutRect.Right := OutRect.Left + BmpSide;
  Control.Canvas.StretchDraw (OutRect, TabBmp);
end;

```

在显示了一个用户可挑选适当比例的页预览格式后，程序也支持打印当前位图。程序的这一额外部分在老版本图书中没有详细介绍，但笔者在程序中留下了该代码，因此读者可以随时查看它的代码。

向导的用户界面

正如可以使用不带页面的TabControl一样，还可以采用相反的方法，使用不带标签的PageControl。现在需要重点讨论的是向导的用户界面开发。在向导中，要指导用户通过一系列步骤，一次一种屏幕，而且每一步都要为下一步提供继续进行的选择，或返回上一步所需的正确输入。所以向导通常提供Next与Back按钮来代替按某个顺序选择的标签。这并不是一个复杂的范例，它的目的只是为了给读者一些指导。该范例名为WizardUI。

首先要在PageControl控件中建立一系列页面，并将每个TabSheet的TabVisible属性设置为False（而将Visible设为True）。与以前的版本相反，从Delphi 5开始，程序员就可以在设计时隐藏标签了。在本例中，我们需要使用页面控件的快捷菜单、对象检验器的组合框或Object Tree View以便移动到另一个页面，而非标签。但想在设计时查看标签该怎么办呢？为此，可将控件放在页面上，然后将额外的控件放在页面前（如在范例中所做的那样），这样就不会在运行时看到它们相对位置的改变。此外，还应删除标签没有用处的标题，它在内存与应用程序资源中占据了一定空间。

在第一页中，我在一边放置了一个图像组件与一个带斜面的控件，在另一边添加了一些文本、一个复选框与两个按钮。实际上，Next按钮在页面内，而Back按钮覆盖在它上面（由所有页面共享）。读者可以在图6.4中看到该页面设计时的外观。随后的页面都与此相似，右边有一个标题、复选框、按钮，左边什么也没有。



图6.4 WizardUI范例设计时的第一页

当单击第一页上的Next按钮时，程序会查看复选框的状态，并决定随后显示的页面。有关代码如下：

```
procedure TForm1.btnNext1Click(Sender: TObject);
begin
  BtnBack.Enabled := True;
  if CheckInprise.Checked then
    PageControl1.ActivePage := TabSheet2
  else
    PageControl1.ActivePage := TabSheet3;
  // move image and bevel
  Bevel1.Parent := PageControl1.ActivePage;
  Image1.Parent := PageControl1.ActivePage;
end;
```

在启用Back按钮之后，程序会改变当前页面，将图形部分移到新页面中。因为这段代码必须被每个按钮重复使用，所以在添加了两个附加特性后，可将它放置到一个对象方法中。

下面是实际代码：

```

procedure TForm1.btnNext1Click(Sender: TObject);
begin
    if CheckInprise.Checked then
        MoveTo (TabSheet2)
    else
        MoveTo (TabSheet3);
end;

procedure TForm1.MoveTo(TabSheet: TTabSheet);
begin
    // add the last page to the list
    BackPages.Add (PageControl1.ActivePage);
    BtnBack.Enabled := True;
    // change page
    PageControl1.ActivePage := TabSheet;
    // move image and bevel
    Bevel1.Parent := PageControl1.ActivePage;
    Image1.Parent := PageControl1.ActivePage;
end;

```

除了已经解释的代码外，MoveTo方法用于向访问页的列表（像一个堆栈）添加最后一个页面（改变页面之前的那个）。事实上，在程序启动时会建立TList类的BackPages对象，最后一页总是添加到末尾。当用户单击Back按钮（不依赖于页面）时，程序会从列表中抽取最后一页，删除它的记录，并移到相应的页面：

```

procedure TForm1.btnBackClick(Sender: TObject);
var
    LastPage: TTabSheet;
begin
    // get the last page and jump to it
    LastPage := TTabSheet (BackPages [BackPages.Count - 1]);
    PageControl1.ActivePage := LastPage;
    // delete the last page from the list
    BackPages.Delete (BackPages.Count - 1);
    // eventually disable the back button
    BtnBack.Enabled := not (BackPages.Count = 0);
    // move image and bevel
    Bevel1.Parent := PageControl1.ActivePage;
    Image1.Parent := PageControl1.ActivePage;
end;

```

使用这段代码，用户可以向后移动数页直到列表空了为止，这时就关闭了Back按钮。需要处理的复杂问题是，当从特殊页移出时，虽然知道它的“前”页与“后”页，但不知道

它来自何页，因为到达一页的路径可能有多条。所以只能通过使用一个列表来跟踪移动，以可靠地返回指定页。

程序余下的代码（只显示了一些Web网站的地址）非常简单。读者可以在自己的程序中重用该范例的导航结构，只需要改动图形部分与页面的内容即可。因为大部分程序的标签都显示HTTP地址，故用户可以单击标签来打开默认浏览器访问该网页。完成这些需要为标签扩展HTTP地址并调用ShellExecute函数：

```
procedure TForm1.LabelLinkClick(Sender: TObject);
var
  Caption, StrUrl: string;
begin
  Caption := (Sender as TLabel).Caption;
  StrUrl := Copy (Caption, Pos ('http://', Caption), 1000);
  ShellExecute (Handle, 'open', PChar (StrUrl), '', '', sw_Show);
end;
```

在上述的方法中，需要为窗体上的多个标签添加OnClick事件，这些标签被转换为链接的形式，当光标指向这些标签时，将显示为手指的形状，下面是其中一个标签的代码：

```
object Label2: TLabel
  Cursor = crHandPoint
  Caption = 'Main site: http://www.borland.com'
  OnClick = LabelLinkClick
end
```

工具栏控件

为了创建工具栏，Delphi引进了一种特殊组件，它封装了相应的Win32通用控件，或VisualCLX中相应的Qt小饰件。该组件提供了一个工具栏，它带有自己的按钮，并且有一些扩展功能。可以把它放置在一个窗体中，然后使用组件编辑器（通过单击鼠标右键激活的快捷菜单）来创建几个按钮和分隔符。

Toolbar是用TToolButton类的对象填充的。这些对象有一个基本属性（Style），该属性确定了它们的功能：

- tbsButton类型表示标准的按钮。
- tbsCheck类型表示具有复选框功能的按钮，或如果按钮与所在块中的（由分隔符确定）其他按钮一起编组的话，就表示为单选钮。
- tbsDropDown类型表示可单击式按钮，也是一种复选框。然而，在Delphi中，通过将PopupMenu控件与控件的DropDownMenu属性相连，按下的效果很容易实现。
- tbsSeparator类型与tbsDivider类型表示没有或带有不同竖线的分隔符（取决于工具栏的Flat属性）。

为了建立图形工具栏，可以向窗体添加ImageList组件，并向ImageList装载一些位图，然后将ImageList与工具栏的Images属性相连。默认地，图形应按显示顺序赋予按钮，但通过设置每个工具栏按钮的ImageIndex属性可轻易地改变该默认动作。还可以为按钮的特殊条件

准备更多的ImageList, 并将它们赋予工具栏的DisabledImages与HotImages属性。第一组用于关闭的按钮, 第二组用于当前鼠标下的按钮。

说明: 在一个实际的应用程序中, 可以使用在下一章讨论的ActionList或新的动作管理器结构创建工具栏。在此例中, 将为工具栏按钮粘贴细小的动作, 因为它们的属性和事件将通过动作组件来管理。此外, 将结束使用TActionToolBar类的工具栏。

RichBar示例

作为一个使用工具栏的示例, 笔者建立了RichBar应用程序, 该程序有一个能通过使用工具栏操作的RichBar组件, 而且还有一个装载和保存文件的按钮, 用于复制和粘贴操作, 并可改变当前字体的某些属性。

这里不想牵扯RichEdit控件的详细特征, 因为这些特征太多并涉及许多代码, 也没有详细涉及到该应用程序。笔者只想集中介绍那些在示例中使用的并在图6.5中看到的ToolBar的特殊特征。该工具栏包含按钮、分隔符、一个下拉菜单和两个组合框(下一节中讨论)。

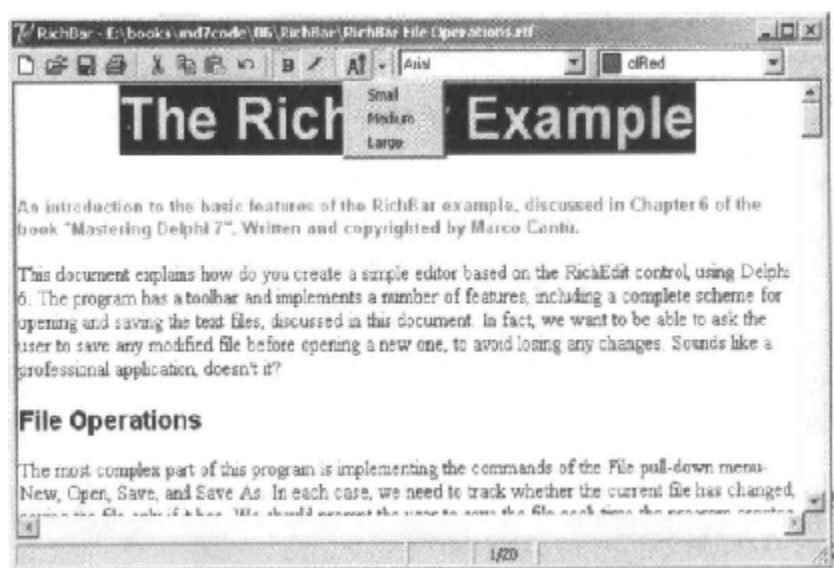


图6.5 RichBar中的工具栏, 注意其中的下拉菜单

不同的按钮实现不同的特征, 它们中的任一个对于打开、保存文本文件都是一个完整的设计, 包括在打开新文件前要求用户保存修改的文件以避免文件丢失。程序的文件处理部分十分复杂, 但是它值得探讨, 因为许多基于文件的应用程序将使用相似的代码。在RichBar File Operations.rtf文件中有该范例详细的源代码, 可以用RichBar程序打开该文件。

除了文件操作外, 该程序还支持复制、粘贴操作和字体管理。复制和粘贴操作不要求与VCL的Clipboard对象有实际的相互作用, 因为组件用以下简单的命令就可以处理它们:

```
RichEdit.CutToClipboard;  
RichEdit.CopyToClipboard;  
RichEdit.PasteFromClipboard;  
RichEdit.Undo;
```

当这些操作（和对应的按钮）可用时，有一点需提前了解。当一些文本被选中时，将激活OnSelectionChange事件的RichEdit控件的Copy和Cut按钮：

```
procedure TFormRichNote.RichEditSelectionChange(Sender: TObject);
begin
  tbtnCut.Enabled := RichEdit.SelLength > 0;
  tbtnCopy.Enabled := tbtnCut.Enabled;
end;
```

Copy操作不能被用户的动作决定，因为它依据剪贴板中的内容，也受其他应用程序的影响。一种方法是使用计时器并不时地检测剪贴板的内容。更好的方法是使用Application对象的OnIdle事件对象（或ApplicationEvents组件）。因为RichEdit支持多个剪贴板格式，所以该代码并不那么容易了解这一点，但是使用低级别特征而非Delphi控件的表面特性，可以询问组件本身：

```
procedure TFormRichNote.ApplicationEvents1Idle(Sender: TObject;
  var Done: Boolean);
begin
  // update toolbar buttons
  tbtnPaste.Enabled := RichEdit.Perform(em_CanPaste, 0, 0) <> 0;
end;
```

Bold和Italic按钮实现了基本的字体管理功能，它们有相似的代码。Bold按钮把属性绑定到挑选的文本上（或在当前编辑位置改变样式）：

```
procedure TFormRichNote.BoldExecute(Sender: TObject);
begin
  with RichEdit.SelAttributes do
    if fsBold in Style then
      Style := Style - [fsBold]
    else
      Style := Style + [fsBold];
end;
```

再次重申，按钮的当前状态由用户当前的选择来决定，因此需要添加下列行代码到RichEditSelectionChange对象方法中：

```
tbtnBold.Down := fsBold in RichEdit.SelAttributes.Style;
```

工具栏中的菜单和组合框

除了一系列按钮外，RichBar示例还有一个下拉菜单和一对组合框，这是许多普通应用程序共享的特征。下拉菜单用于选择字体大小，组合框用于选择字体系列和字体颜色。其中第二个组合框是使用ColorBox控件建立的。

Size按钮使用DropDownMenu属性被连接到PopupMenu组件（叫做SizeMenu）。用户可单击按钮，正常打开它的OnClick事件，或选择下拉箭头，打开弹出式菜单（见图6.5）并挑选选项之一。此例有三种由菜单定义的字大小：

```

object SizeMenu: TPopupMenu
  object Small1: TMenuItem
    Tag = 10
    Caption = 'Small'
    OnClick = SetFontSize
  end
  object Medium1: TMenuItem
    Tag = 16
    Caption = 'Medium'
    OnClick = SetFontSize
  end
  object Large1: TMenuItem
    Tag = 32
    Caption = 'Large'
    OnClick = SetFontSize
  end
end

```

每个菜单项有一个标签暗示实际的字体大小，由一个共享事件处理器激活：

```

procedure TFormRichNote.SetFontSize(Sender: TObject);
begin
  RichEdit.SelAttributes.Size := (Sender as TMenuItem).Tag;
end;

```

因为ToolBar控件是一个全特征容器，所以可以直接采用一个编辑框、组合框和其他控件，并把它们放置在工具栏内。工具栏中的组合框在FormCreate方法中初始化，该对象方法会从系统中提取可以使用的屏幕字体：

```

ComboFont.Items := Screen.Fonts;
ComboFont.ItemIndex := ComboFont.Items.IndexOf (RichEdit.Font.Name)

```

组合框开始会显示用于RichEdit控件中的默认字体名称，该控件在设计时设置。每当当前选择改变时，都会使用所选文本的字体重新计算该值：

```

procedure TFormRichNote.RichEditSelectionChange(Sender: TObject);
begin
  ComboFont.ItemIndex :=
    ComboFont.Items.IndexOf (RichEdit.SelAttributes.Name);
  ColorBox1.Selected := RichEdit.SelAttributes.Color;
end;

```

当从组合框中选择一种新字体时，相反的操作会同时发生。当前组合框选项的文本会被作为字体名称赋给RichEdit控件中所选的任何文本：

```

RichEdit.SelAttributes.Name := ComboFont.Text;

```

ColorBox中的一个颜色选择器将激活相应代码。

简单的状态栏

建立一个状态栏甚至比建立工具栏更简单。Delphi包含了一个专门的StatusBar组件，它基于相应的Windows通用控件（相似的控件还可以在VisualCLX中使用）。该组件在其SimplePanel属性被设置为True时，几乎可以用做面板。在这种情况下，可以使用SimpleText属性输出一些文本。然而，该组件的真正优点在于，它允许程序员定义一些子面板，这只需要激活其Panels属性的编辑器即可（还可以双击状态栏控件显示该属性）。每个子面板都有自己的图形属性，可以使用对象检验器定制它们。状态栏组件的另一个特性是添加到栏右下角的“大小调整”区，用于调整自己窗体的大小。这是Windows用户界面的典型元素，而且可以使用SizeGrip属性控制它（当窗体不能自动调整大小时它就不可用）。

状态栏有不同的用法。最常见的用法是显示当前用户所选菜单项的有关信息。除此之外，状态栏经常显示程序状态的其他信息：鼠标在图形程序中的位置，字处理器中文本的当前行，加锁键的状态，事件与日期，等等。为了在面板上显示信息，只需使用其Text属性，通常使用的表达式如下：

```
StatusBar1.Panels[1].Text := 'message';
```

在RichBar范例中，用三个面板建立了一个状态栏，用于显示命令提示、Caps Lock键的状态与当前编辑位置。范例的StatusBar组件实际有四个面板；需要定义第四个面板，用以限制第三个面板的区域。事实上，最后一个面板总是足够大的，可以覆盖状态栏的剩余表面。

提示：同样，关于RichBar程序的详细情况见本书范例程序代码的RTF文件。注意，因为提示显示在第一个面板的状态栏，故可以使用AutoHint属性使代码简化。由于我已经提供了很详尽的代码，所以读者可以自己定制。

面板不是一个独立的组件，所以不能通过名称访问它们，而只能通过前述的代码段来定位。一个增加程序可读性的方法是为每个要用的面板定义一个常量，然后通过常量调用面板。下面是段简单的代码：

```
const
  sbpMessage = 0;
  sbpCaps = 1;
  sbpPosition = 2;
```

在第一个面板的状态栏中显示工具栏的提示信息，该程序可以使用应用程序的OnHint事件实现，同时使用ApplicationEvents组件，将应用程序Hint属性的当前值复制给状态栏：

```
procedure TFormRichNote.ApplicationEvents1Hint (Sender: TObject);
begin
  StatusBar1.Panels[sbpMessage].Text := Application.Hint;
end;
```

默认情况下，这段代码会在状态栏中显示并浮动提示相同的文本。实际上，可以使用Hint属性为两种情况确定不同的字符串，为此，需要编写一个字符串，并用一个分隔符（|）将其划分为两部分。例如，可以输入下列代码作为Hint属性的值：

```
'New|Create a new document'
```

字符串的第一部分，New，被浮动提示使用；第二部分，Create a new document，由状态栏使用，如图6.6所示。

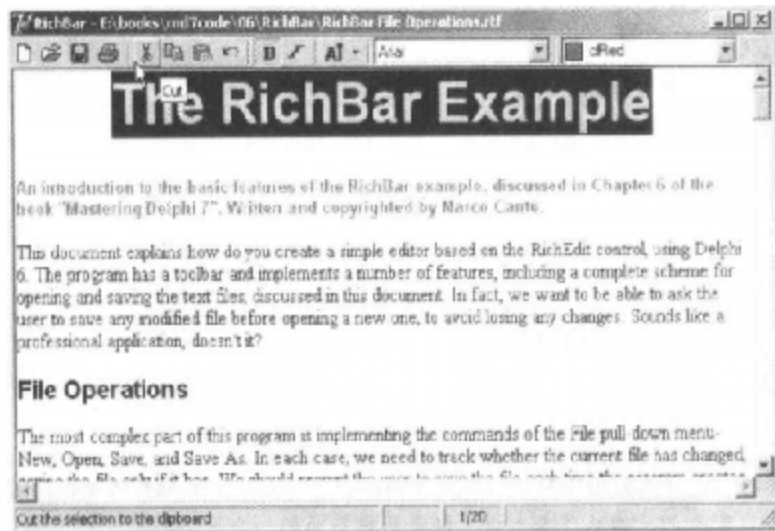


图6.6 RichBar范例的状态栏，可以显示更加详细的信息

提示：当用于控件的提示由两个字符串组成时，可以使用GetShortHint与GetLongHint方法从通过的字符串中抽取第一个（short）和第二个（long）子字符串作为参数，它通常是Hint属性的值。

第二个面板显示了Caps Lock键的状态，要通过调用GetKeyState API函数来获得，它返回一个状态值。如果设置了该数值的低位（也就是说，如果数值是奇数），就表示该键被按下了。我已决定在应用程序空闲时检查其状态，这样每当该键被按下以及有消息到达窗口（在处理另一个程序时用户改变了该设置的情况）时就将执行这项检查。这里向ApplicationEventsIdle处理程序添加了一个用于定制CheckCapslock对象方法的调用，实现如下：

```
procedure TFormRichNote.CheckCapslock;
begin
    if Odd (GetKeyState (VK_CAPITAL)) then
        StatusBar1.Panels[spbCaps].Text := 'CAPS'
    else
        StatusBar1.Panels[spbCaps].Text := '';
end;
```

最后，该程序使用第三个面板来显示每次选择变化时当前光标的位置。因为CaretPos值是基于一零的（即左上角是0线，且字符为0），所以决定为每个值都添加，使它们对临时用户更为合理：

```
procedure TFormRichNote.RichEditSelectionChange(Sender: TObject);
begin
    ...
    // update the position in the status bar
    StatusBar.Panels[spbPosition].Text := Format ('%d/%d',
        [RichEdit.CaretPos.Y + 1, RichEdit.CaretPos.X + 1]);
end;
```


主题与样式

过去, 基于GUI的操作系统为运行在其上的程序指定了全部用户界面的元素。近些年, Linux开始允许用户定制应用程序主窗体和用户界面控件的外观, 如按钮控件。同样的方法已经在许多程序中出现, 它的积极影响甚至使Microsoft开始接受(开始时用在程序中, 然后用于整个操作系统)。

CLX样式

在Linux(更准确地说是X-Windows)中, 用户一般能够选择控件的用户界面样式, 并且得到Qt和KDE系统的完全支持。Qt提供一些基础的样式, 如Windows样式、Motif样式, 等等。用户还可以在系统中安装新样式, 并将它们用于应用程序。

说明: 这里讨论的样式引用了控件的用户界面, 而不是窗体及其边框的界面。这通常在Linux系统中是可以配置的, 但从技术上讲, 这是用户界面的一个独立元素。

因为该技术嵌套在Qt中, 所以还可以在该库的Windows版本上使用。CLX允许它为Delphi开发人员所用, 以至在Microsoft操作系统中使用应用程序能够有Motif的感观。CLX的Application全局对象有一个Style属性, 可以用于设置定制的样式或默认的样式, 由DefaultStyle子属性说明。例如, 可以用下列代码选择一个Motif样式:

```
Application.Style.DefaultStyle := dsMotif;
```

在StylesDemo程序中, 我为各种范例控件添加了一个含有默认样式名的列表框, 如TDefaultStyle枚举说明的那样, 下面是其OnDblClick事件的代码:

```
procedure TForm1.ListBox1DblClick(Sender: TObject);  
begin  
    Application.Style.DefaultStyle := TDefaultStyle (ListBox1.ItemIndex);  
end;
```

其效果是, 通过双击列表框, 可以改变当前的应用程序样式, 并立即在屏幕上看到效果, 如图6.7所示。

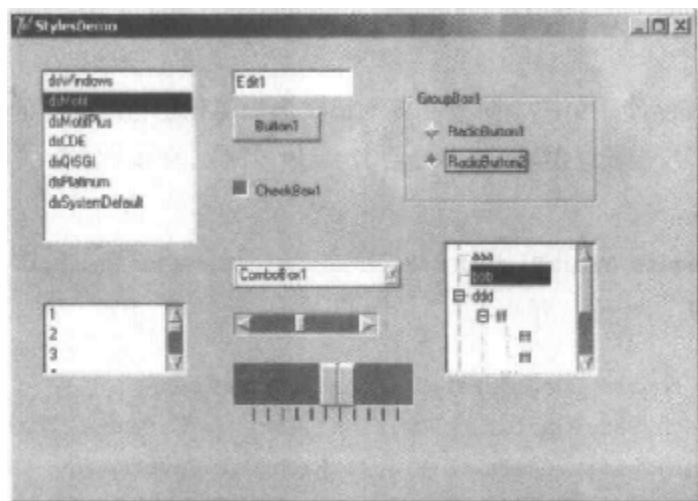


图6.7 StylesDemo程序, 一个特殊Motif布局的Windows应用程序

Windows XP主题

随着Windows XP的发布,微软引入了一个新的、独立版本的通用控件库。旧的库仍然出于兼容性的原因被保留,因此一个在XP上运行的程序可以选择其中之一,作为它希望使用的库。新的通用控件库的主要区别在于它不是一个固定的绘制引擎,而是要借助XP主题引擎并且针对当前的主题来代表组件的用户接口。

在Delphi 7中,VCL完全支持主题,因为它有大量的内部代码以及主题管理库,它们原是由Mike Lischke开发的。一些新的绘制特性被Action Manager体系结构中的可视控件所使用,它与运行它的操作系统无关。但是,完整的主题支持只可用于支持这个特性的操作系统,在目前,就是指Windows XP而已。

在XP中,Delphi应用程序默认情况下使用传统的方法。为了支持XP主题,我们必须包括一个证明文件在程序中。我们可以使用下面的办法来完成这个任务:

- 将一个证明文件放在与应用程序相同的文件夹中。这是一个XML文件,表明了标识以及程序的从属性。这个文件与可执行文件具有相同的文件名,但是具有一个额外的.manifest扩展名,如MyProgram.exe.manifest。我们可以在程序清单6.2中看到一个范例文件。
- 在一个应用程序中编译的资源文件中添加相同的信息。我们需要编写一个资源文件,其中包括一个证明文件。在Delphi 7中,VCL具有一个WindowsXP.res编译资源文件,它可以通过重新编译VCL源文件中的WindowsXP.rc文件获得。这个资源文件中包含有sample.manifest文件,也可以在VCL源文件中找到。
- 使用XpManifest组件,它是Borland添加到Delphi 7中用来进一步简化这个任务的。当我们将这个什么也不做的组件放入一个程序的窗体后,Delphi会自动地包括XPMan单元,它可导入前面提到的VCL资源文件。

警告: 当从应用程序中删除XpManifest组件的时候,也必须手工使用uses语句删除XPMan单元。而Delphi是不会替我们完成这个任务的。如果我们没有做这个操作,即使没有XpManifest组件,程序仍然会将证明资源文件绑定进来。使用这个单元是很必要的(我不明白,为什么Borland要创建该组件,而不是简单提供该单元或相关的资源文件,要知道组件根本不能归档)。

程序清单6.2 一个证明文件的范例(pages.exe.manifest)

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<assembly xmlns="urn:schemas-microsoft-com:asm.v1" manifestVersion="1.0">
  <assemblyIdentity
    version="1.0.0.0"
    processorArchitecture="X86"
    name="Pages.exe"
    type="win32"
  />
  <description>Mastering Delphi Demo</description>
  <dependency>
    <dependentAssembly>
      <assemblyIdentity
        type="win32"
```

```
name="Microsoft.Windows.Common-Controls"
version="6.0.0.0"
processorArchitecture="X86"
publicKeyToken="6595b64144ccf1df"
language="*"
/>
</dependentAssembly>
</dependency>
</assembly>
```

作为一个演示，我们添加了程序清单6.2中的证明文件到Pages范例的文件夹中。通过在Windows XP中使用标准的XP主题来运行它，我们将能够获得类似于图6.8的输出。我们可以用它和图6.1与图6.2进行比较，它们显示了该程序在Windows 2000中运行的情况。



图6.8 Pages范例使用了当前的Windows XP主题，它包含在一个证明文件中（请与图6.1进行比较）

ActionList组件

Delphi的事件结构是非常开放的：程序员可以编写一个事件处理程序，并将它与工具栏按钮或菜单的OnClick事件相连。如果事件处理程序可以使用Sender参数来引用激活事件的对象，则也可以将同一个事件处理程序与不同的按钮或菜单命令相连。但同步工具栏按钮与菜单项的状态还有一点困难。如果有一个菜单项和一个工具栏按钮的切换选项相同，那么每当切换一项时，都必须同时向菜单项添加复选标记并改变按钮的状态以显示它被按过。

为了克服这个问题，Delphi基于动作引入了一种新的事件处理机制。动作（或命令）既说明了单击某菜单项或按钮时所执行的操作，又确定了所有与动作有关的元素状态。动作与链接控件的用户界面之间的连接是非常重要的，应该予以重视，因为它决定了是否可以很好地利用该机制。

在这种事件处理机制中有很多环节，其中，动作对象起着核心作用。与其他任何组件一样，动作对象有自己的名称，而且拥有用于链接控件（被称做动作客户）的其他属性。这些属性包括标题、图形表示（ImageIndex）、状态（Checked、Enabled与Visible）以及用户反馈（提示与帮助）。此外，还有ShortCut和SecondaryShortCut列表，两种状态动作的

AutoCheck属性，帮助支持和用于在逻辑组合上安排动作的Category属性。

动作对象的基类是TBasicAction。它介绍了一个动作的抽象核心动作，对菜单项或控件不进行任何特殊绑定或者改正。派生类TContainedAction介绍了能使动作展现在动作列表或动作管理器上的属性和对象方法。派生类TCustomAction介绍了链接到动作对象上的菜单项和控件属性与对象方法。最后，是一个备用的TAction源类。

每个动作对象都通过ActionLink对象与一个或多个客户对象连接。注意，多个控件（可能是不同类型的）可以共享同一个动作对象，这由它们的Action属性来指定。从技术上讲，ActionLink对象维持着客户对象和动作的双向连接。之所以需要ActionLink对象，是因为连接工作是双向的。在对象上的操作（如单击）会触发动作对象，并导致对其OnExecute事件的调用；对动作对象状态的更新会反映到相连的客户控件中去。换句话说，一个或多个客户控件可以创建一个ActionLink对象，该对象会与动作对象一起注册。

程序员不应该设置与动作相连的客户控件属性，因为动作将改写客户控件的属性值。因此，通常应该首先编写动作，然后建立与之相连的菜单项以及按钮。还需要注意，当动作没有OnExecute事件处理程序时，客户控件会自动关闭（或变灰），除非已将DisableIfNoHandler属性设置为False。

与动作相连的客户控件通常是菜单项和各种类型的按钮（普通按钮、复选框、单选按钮、快捷按钮、工具栏按钮等等），但是也不排除使用该机制建立新组件。组件编写人员甚至可以定义新动作与新的链接动作对象，相关内容将在第9章“编写Delphi组件”中介绍。

除了客户端控件，有些动作还可以拥有目标组件。有些预定义动作与指定的目标组件相连。其他动作会自动在支持指定动作的窗体中寻找目标组件，使用激活控件来启动。

最后，动作对象是由ActionList或者ActionManager组件拥有的，这是该机制中惟一显示在组件面板上的类。动作列表组件会接收指定动作对象没有处理的可执行动作，触发OnExecuteAction事件。甚至，如果动作列表也无法处理某动作，则Delphi会调用Application对象的OnExecuteAction事件。动作列表对象有一个特殊的编辑器，可用于建立一些动作，如图6.9所示。

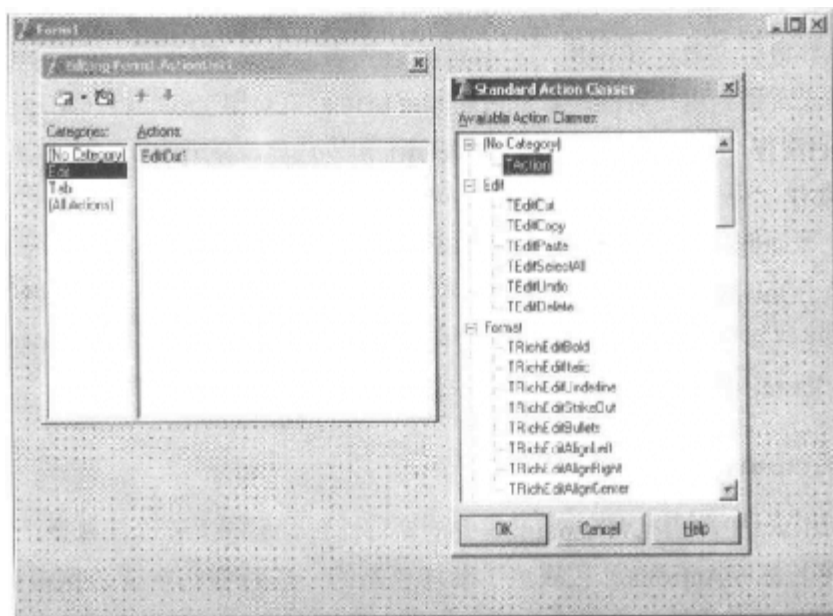


图6.9 ActionList组件编辑器，以及可以使用的一系列定义操作

在编辑器中，动作按不同组显示，由它们的Category属性指定。只需将该属性设置为新值，就可以命令编辑器建立新组。这些组基本上是逻辑组，尽管在某些情况下，一组动作只能作用于一种指定类型的目标组件。也可以为每个下拉式菜单定义一个组或按其他逻辑关系将它们分组。

Delphi中的预定义动作

使用动作列表和ActionManager编辑器可以建立新动作，或选择在系统中注册的已有动作。这些动作列在二级对话框中，如图6.9所示。预定义动作有很多，都被划分在不同的逻辑组中：

文件动作 包括Open、Save as、Open with、Run、Print Setup和Exit。

编辑动作 在下一个范例中说明。它们包括Cut、Copy、Paste、Select All、Undo和删除动作。

RichEdit动作 为RichEdit控件补充编辑动作，包括bold、italic、underline、strikeout、bullets和各种对齐动作。

MDI窗口动作 将在第8章中介绍，届时将介绍多文档界面（MDI）方法。它们包括了大部分常用的MDI操作：Arrange、Cascade、Close、Tile与Minimize等。

Dataset动作 与数据库表格以及查询有关，将在第13章中介绍。数据集动作很多，包括了在数据集上可能执行的所有主要操作。Delphi 7增加了核心数据集动作，它是一组为ClientDataSet组件量身定制的动作，包括apply、revert和undo。该动作将在第13章（包括普通的数据库编程和特别的ClientDataSet组件）和第14章（讨论数据库更新）继续讨论。

Help动作 允许激活应用程序配属的帮助文件内容页或索引。

Search动作 包括Find、Find First、Find Next和Replace。

Tab与Page控件动作 包括向前一页和下一页导航。

Dialog动作 激活颜色、字体、打开、保存和打印对话。

List动作 包括Clear、Copy、Move、Delete和Select All。这些动作可以使程序员与列表控件进行交互。另一组动作包括静态列表、虚拟列表和一些支持类，它们允许定义的列表连接到用户界面。本章最后的“使用列表动作”一节将详细地介绍相关内容。

Internet动作 包括浏览URL、下载URL和发送信件的动作。

Tools动作 只包括对定制动作框的对话。

除了处理动作的OnExecute事件及改变动作的状态以影响客户控件的用户界面外，动作还可以处理：OnUpdate事件，当应用程序空闲时会激活该事件处理程序。这使得程序员可以检测应用程序或系统的状态并相应地改变控件的用户界面。例如，仅当剪贴板中有一些文本时，标准的PasteEdit动作才会作用于客户控件。

实际范例Actions

前面已经介绍了动作（Delphi最重要的新特性）背后的主要设计思想，现在来研究一个范例。范例名称叫Actions，它演示了动作机制的一些特性。在该范例窗体中先放置一个ActionList组件，并添加三个标准编辑动作及一些定制动作。窗体中还有一个含有快捷按钮

的面板，一个主菜单以及一个Memo控件（编辑动作的自动目标）。程序清单6.3给出了这些动作的列表，摘自DFM文件。

程序清单6.3 Actions范例的动作

```
object ActionList1: TActionList
  Images = ImageList1
object ActionCopy: TEditCopy
  Category = 'Edit'
  Caption = '&Copy'
  ShortCut = <Ctrl+C>
end
object ActionCut: TEditCut
  Category = 'Edit'
  Caption = 'Cu&t'
  ShortCut = <Ctrl+X>
end
object ActionPaste: TEditPaste
  Category = 'Edit'
  Caption = '&Paste'
  ShortCut = <Ctrl+V>
end
object ActionNew: TAction
  Category = 'File'
  Caption = '&New'
  ShortCut = <Ctrl+N>
  OnExecute = ActionNewExecute
end
object ActionExit: TAction
  Category = 'File'
  Caption = 'E&xit'
  ShortCut = <Alt+F4>
  OnExecute = ActionExitExecute
end
object NoAction: TAction
  Category = 'Test'
  Caption = '&No Action'
end
object ActionCount: TAction
  Category = 'Test'
  Caption = '&Count Chars'
  OnExecute = ActionCountExecute
  OnUpdate = ActionCountUpdate
end
object ActionBold: TAction
  Category = 'Edit'
```

```

    AutoCheck = True
    Caption = '&Bold'
    ShortCut = <Ctrl+B>
    OnExecute = ActionBoldExecute
end
object ActionEnable: TAction
    Category = 'Test'
    Caption = '&Enable NoAction'
    OnExecute = ActionEnableExecute
end
object ActionSender: TAction
    Category = 'Test'
    Caption = 'Test &Sender'
    OnExecute = ActionSenderExecute
end
end
end

```

说明：存储在DFM文件中的快捷键使用了虚拟键号，此号还包括了Ctrl与Alt键的值。在本书的程序清单中，使用字母面值代替键号，将该值括在尖括号中。

所有这些动作都与MainMenu组件的菜单项以及面板上的快捷按钮相连。注意，在ActionList控件中选择的图像只影响编辑器中的动作，如图6.10所示。为使ImageList的图像也显示在菜单项和工具按钮中，还必须在MainMenu组件与ToolBar组件中选择图像列表。

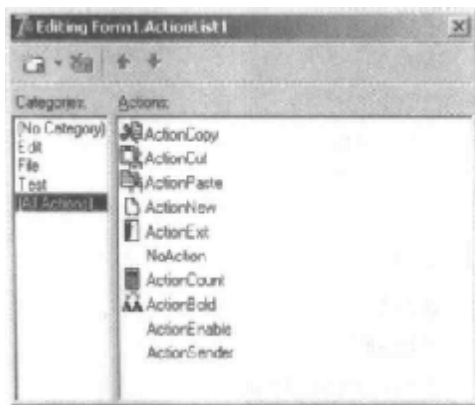


图6.10 Actions范例的ActionList编辑器

Edit菜单的三个预定义动作没有相关的处理程序，但这些特殊的对象拥有内部代码，在激活的编辑或备注控件上执行相关动作。这些动作还可以根据剪贴板上的内容以及当前编辑控件中所选文本的存在情况，激活或关闭自己。其他大部分动作都具有定制代码，但NoAction对象除外。因为没有代码，故与该命令相连的菜单项处于关闭状态，即使动作的Enabled属性被设置为True也是一样。

下面的代码向范例以及Test菜单中添加了另一个动作，它能激活与NoAction对象相连的菜单项：

```

procedure TForm1.ActionEnableExecute(Sender: TObject);
begin

```

```

NoAction.DisableIfNoHandler := False;
NoAction.Enabled := True;
ActionEnable.Enabled := False;
end;

```

只需将Enabled设置为True, 就会在非常短的时间内产生效果, 除非设置DisableIfNoHandler属性, 就像上一节讨论的那样。一旦执行该操作, 就会关闭当前动作, 因为没有必要再次发出相同的命令。

这不同于可以触发的动作, 如Edit►Bold菜单项及相应的快捷按钮。下面是Bold动作的代码 (因为AutoCheck属性已设为True, 故不必在代码中改变Checked属性的状态):

```

procedure TForm1.ActionBoldExecute(Sender: TObject);
begin
  with Mem01.Font do
    if fsBold in Style then
      Style := Style - [fsBold];
    else
      Style := Style + [fsBold];
    end;
end;

```

ActionCount对象的代码非常简单, 但它演示了OnUpdate处理程序; 当备注控件是空的时候, 它会被自动关闭。可以通过处理备注控件自己的OnChange事件获得同样的效果, 但只通过处理控件的某个事件来确定其状态通常不能或并不容易。下面是该动作两个处理程序的代码:

```

procedure TForm1.ActionCountExecute(Sender: TObject);
begin
  ShowMessage ('Characters: ' + IntToStr (Length (Mem01.Text)));
end;

procedure TForm1.ActionCountUpdate(Sender: TObject);
begin
  ActionCount.Enabled := Mem01.Text <> '';
end;

```

最后, 添加了一个特殊动作来测试动作事件处理程序的sender对象并获得其他一些系统信息。除了显示对象的类与名称外, 还添加了访问动作列表对象的代码。这样做的目的主要是为了向读者说明, 该信息是可以访问的, 以及访问的具体方法:

```

procedure TForm1.ActionSenderExecute(Sender: TObject);
begin
  Mem01.Lines.Add ('Sender class: ' + Sender.ClassName);
  Mem01.Lines.Add ('Sender name: ' + (Sender as TComponent).Name);
  Mem01.Lines.Add ('Category: ' + (Sender as TAction).Category);
  Mem01.Lines.Add (
    'Action list name: ' + (Sender as TAction).ActionList.Name);
end;

```


读者可以从图6.11中看到这段程序的输出，以及范例的用户界面。注意，**Sender**并不是选择的菜单项，即使事件处理程序与它连接也是如此。触发事件的**Sender**对象用于截取用户操作的动作。

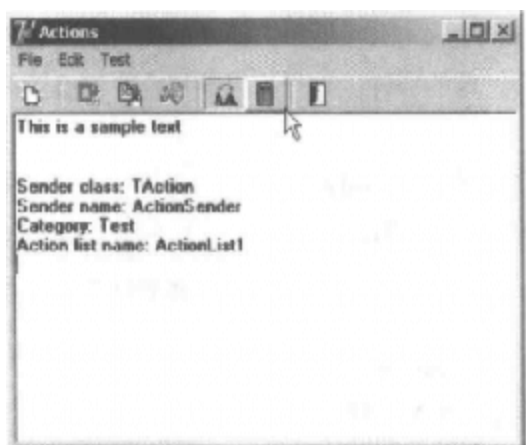


图6.11 Actions范例，以及Action对象OnExecute事件的Sender的详细说明

最后，需要牢记，还可以为ActionList对象自己的事件编写处理程序，它对于列表的所有动作和对于Application的全局对象来说，起着全局处理程序的作用。事实上，在调用动作的OnExecute事件之前，Delphi将首先激活ActionList的OnExecute事件和Application全局对象的OnActionExecute事件。这些事件能查看动作，甚至执行一些共享代码，然后停止执行（使用Handled参数）或让它到达下一级别。

如果没有为对应的动作或者动作列表、应用程序或动作级别指定事件处理器，那么应用程序将试图确定一个目标对象，以使动作能够得以处理。

说明：当一个动作被执行时，它通过在窗体上查看活动控件、活动窗体和其他控件来查找对动作目标起作用的控件。例如，当数据集控件查找与数据源相连的数据集集合时，编辑动作指向当前活动控件（如果继承自TCustomEdit）。其他动作有不同的方法来发现目标组件，但整个想法被多数标准动作共享。

编辑器的工具栏与ActionList

在第5章中，已介绍了用RichBar示例来说明使用工具栏和状态栏进行编辑器的开发。当然，我们应该向窗体添加一个菜单栏，但在向那些菜单项同步创建工具栏按钮时存在一些困难。解决这一问题比较好的方法是使用在MdEdit1示例中介绍的动作，本节将对此进行讨论。

应用程序是基于ActionList组件的，其中包括了文件处理和剪贴板支持的动作，使用与RichBar版本相似的代码。字体类型和颜色选择仍基于组合框，因此它与动作无关，下拉菜单的Size按钮也是如此。然而菜单有几个附加命令，包括用于字符计算和改变背景颜色的命令。它们都基于动作，新组中的三个按钮（和菜单命令）也是如此。

该新版本的主要不同之处是代码没有指向工具栏按钮的状态，但最后修改了动作的状态。在其他例子中也使用了动作的OnUpdate事件。例如，RichEditSelectionChange对象方法没有更新粗体按钮的状态，该按钮通过下面的OnUpuate处理器与动作相连：

```

procedure TFormRichNote.acBoldUpdate(Sender: TObject);
begin
    acBold.Checked := fsBold in RichEdit.SelAttributes.Style;
end;

```

同样，大多数动作都具有**OnUpdate**事件处理器，包括统计操作（仅当**RichEdit**控件总存在一些文本时）、保存操作（仅当文本被改动时），以及剪切和复制操作（仅当选中一些文本时）：

```

procedure TFormRichNote.acCountcharsUpdate(Sender: TObject);
begin
    acCountChars.Enabled := RichEdit.GetTextLen > 0;
end;

procedure TFormRichNote.acSaveUpdate(Sender: TObject);
begin
    acSave.Enabled := Modified;
end;

procedure TFormRichNote.acCutUpdate(Sender: TObject);
begin
    acCut.Enabled := RichEdit.SelLength > 0;
    acCopy.Enabled := acCut.Enabled;
end;

```

在以前的例子中，粘贴按钮的状态在应用程序对象的**OnIdle**事件中被更新。现在可以使用动作把它转换成另一个**OnUpdate**处理器（详细代码见第5章）：

```

procedure TFormRichNote.acPasteUpdate(Sender: TObject);
begin
    acPaste.Enabled := SendMessage (RichEdit.Handle, em_CanPaste, 0, 0) <> 0;
end;

```

三个段落对齐按钮以及相关的菜单项可以像单选按钮一样工作，它们互相排斥，即三个按钮中总是只能有一个被选中。因为这个原因，动作把**GroupIndex**设置为1，并将相关菜单项的**RadioItem**属性设置为**True**，三个工具栏按钮的**Grouped**属性设置为**True**，而将**AllowAllUp**属性设置为**False**（它们被安排在两个分隔符之间）。

这样安排是有必要的，以便程序能够对应当前的样式为动作设置**Checked**属性，以避免未检查其他两个动作的错误发生。因为要将该代码应用到多个动作，所以说它是动作列表的**OnUpdate**事件的一部分：

```

procedure TFormRichNote.ActionListUpdate(Action: TBasicAction;
    var Handled: Boolean);
begin
    // check the proper paragraph alignment
    case RichEdit.Paragraph.Alignment of
        taLeftJustify: acLeftAligned.Checked := True;
        taRightJustify: acRightAligned.Checked := True;
        taCenter: acCentered.Checked := True;

```

```
end;  
// checks the caps lock status  
CheckCapslock;  
end;
```

最后, 当这些按钮中的一个被选中时, 共享事件处理器将使用Tag的值, 设置对应的TAlignment枚举的值, 以决定恰当的队列:

```
procedure TFormRichNote.ChangeAlignment(Sender: TObject);  
begin  
  RichEdit.Paragraph.Alignment := TAlignment ((Sender as TAction).Tag);  
end;
```

工具栏容器

大部分现代应用程序都拥有多个工具栏, 通常由一个专门的容器来控制。Microsoft Internet Explorer、各种标准的商业应用程序, 以及Delphi IDE都采用这种办法。然而, 实现却各不相同。Delphi有两个备用的工具条容器:

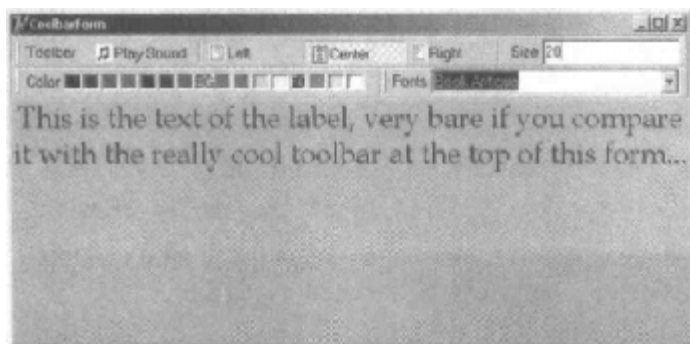
- CoolBar组件是一种由Internet Explorer引入的Win32通用控件, 用于一些Microsoft应用程序。
- ControlBar组件是基于VCL的, 不依赖于外部的库。

这两种组件都可以控制工具栏控件以及一些附加元素, 如组合框与其他控件。工具栏还代替了应用程序的菜单, 稍后将会介绍。因为CoolBar组件不常用于Delphi应用程序, 故仅在下面的附加说明中介绍一下CoolBar。下一节将强调Delphi的ControlBar。

真正酷的工具栏

Win32的CoolBar组件基本上是TCoolBand对象的一个集合, 可以通过Bands属性的编辑器来激活, 或通过该组件编辑菜单项或TreeView对象激活。可以用很多方法定制CoolBar。可以为它的背景设置一幅位图, 使用Bands属性编辑器添加一些区, 然后赋予每个区一个已有的组件或组件容器。可以使用任何基于窗口的控件(非图形控件), 但只有一部分可以适当地显示。例如, 如果想将CoolBar控件的背景设置为一幅位图, 就需要使用透明控件。用在CoolBar中的典型控件是Toolbar, 但也常使用组合框、编辑框以及动画控件。

我们可以每一行放置一个区或将所有的区都放在一行上。每个区使用一部分, 单击它的标题可以自动扩大它。使用这个新组件比解释它更容易, 读者可以自己练习使用该组件或打开CoolDemo范例。

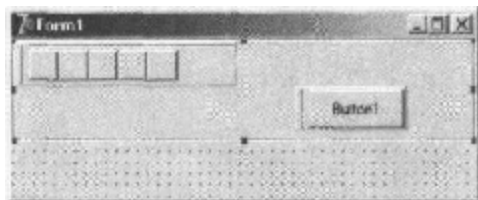


CoolDemo范例的窗体有一个包含四个区的TCoolBar组件，一共有两行，一行两个。第一区包括了前一个范例中工具栏的子集合，这次为高亮显示的图像添加了一个ImageList；第二区是用于设置文本字体的编辑框；第三区有一个ColorGrid组件，用于选择字体颜色以及背景颜色；最后一区有一个ComboBox控件，列出了可使用的字体。

CoolBar组件的用户界面非常漂亮，而且Microsoft也在自己的应用程序中越来越多地使用了它，然而，如果使用ControlBar组件也可以提供相似的没有问题的UI。Windows CoolBar控件有很多不同的和不兼容的版本，因为Microsoft使用了Internet Explorer的不同版本发布通用控件库的不同版本。其中一些版本与Delphi建立的已有程序是不兼容的，尽管它很稳定，但这也是不用它的好理由。

控件栏

控件栏是一个控件容器，而且建立它只需将其他控件放置到其中即可，就像面板一样（没有Bands列在其中）。每个放置在控件栏中的控件都有一个自己的拖动区（在控件左边，带有两条竖线的小面板），甚至单独的按钮，如下图所示。



因此，通常不能将具体的按钮放在控件栏中，而是再添加容器，将按钮放置在这些容器当中。一般应该为工具栏的每一部分都使用一个工具栏控件，而不是使用面板。

MdEdit2范例是本章中开发的演示程序的另一个版本。基本上将按钮分配到三个工具栏（而不是一个）中，并保留了两个组合框为独立控件。所有这些组件都在一个ControlBar中，这样用户可以在运行时根据自己的意愿安排它们，如图6.12所示。

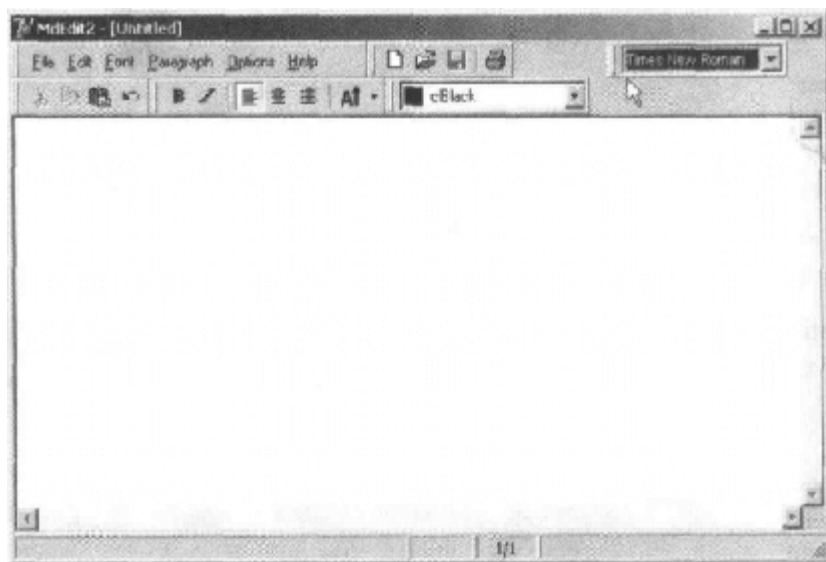


图6.12 MdEdit2范例运行时的情形，图中用户正重新安排控件栏中的工具栏

下列MdEdit2示例的DFM程序清单摘录显示了各种工具栏和控件是怎样嵌入到ControlBar组件中的:

```

object ControlBar1: TControlBar
  Align = alTop
  AutoSize = True
  ShowHint = True
  PopupMenu = BarMenu
object ToolBarFile: TToolBar
  Flat = True
  Images = Images
  Wrapable = False
object ToolButton1: TToolButton
  Action = acNew
end
  // more buttons...
end
object ToolBarEdit: TToolBar...
object ToolBarFont: TToolBar...
object ToolBarMenu: TToolBar
  AutoSize = True
  Flat = True
  Menu = MainMenu
end
object ComboFont: TComboBox
  Hint = 'Font Family'
  Style = csDropDownList
  OnClick = ComboFontClick
end
object ColorBox1: TColorBox...
end

```

为了获得标准效果,必须消除工具栏控件的边缘,并将其类型设置为扁平。将所有控件的大小设置为相同——这样可以获得一行或两行高度相同的控件——不像初见时那么简单。

一些控件可以自动设置大小或不同的限制,特别是为了使组合框与工具栏一样高,必须调整其字体的类型与大小。重新改变控件的大小不会有效。

ControlBar还有一个快捷菜单,允许程序员显示或隐藏其中当前的每个控件。不用特别为该范例编写代码,我已经实现了更为通用的解决方法。快捷菜单,叫做BarMenu,在设计时是空的,并在程序启动时填充:

```

procedure TFormRichNote.FormCreate(Sender: TObject);
var
  I: Integer;
  mItem: TMenuItem;
begin
  ...

```

```
// populate the control bar menu
for I := 0 to ControlBar.ControlCount - 1 do
begin
    mItem := TMenuItem.Create (Self);
    mItem.Caption := ControlBar.Controls [I].Name;
    mItem.Tag := Integer (ControlBar.Controls [I]);
    mItem.OnClick := BarMenuClick;
    BarMenu.Items.Add (mItem);
end;
```

BarMenuClick过程是一个事件处理程序，被菜单的所有选项使用，并在FormCreate方法中使用Sender菜单项的Tag属性来引用与菜单项有关的ControlBar的元素：

```
procedure TFormRichNote.BarMenuClick(Sender: TObject);
var
    aCtrl: TControl;
begin
    aCtrl := TControl ((Sender as TComponent).Tag);
    aCtrl.Visible := not aCtrl.Visible;
end;
```

最后，菜单的OnPopup事件用于更新菜单项的选取标记：

```
procedure TFormRichNote.BarMenuPopup(Sender: TObject);
var
    I: Integer;
begin
    // update the menu check marks
    for I := 0 to BarMenu.Items.Count - 1 do
        BarMenu.Items [I].Checked := TControl (BarMenu.Items [I].Tag).Visible;
    end;
```

控件栏中的菜单

如果看一下MdEdit2应用程序的用户界面，见图6.12，将注意到窗体的菜单实际上显示在工具栏内，由控件栏控制并停放在应用程序标题下。在Delphi的老版本中，这需要编写一些代码。而在Delphi 6，所要做的只是设置工具栏的Menu属性。必须从窗体的Menu属性中删除主菜单，以避免有两个菜单。

Delphi的停放支持

Delphi 4提供的另一个新特性是支持可停放工具栏与控件。换句话说，现在可以创立一个工具栏并将它移到窗体的任何一边，或甚至在屏幕上任意移动它，而不将其停放。然而建立一个获得该效果的程序并不容易。

首先，Delphi的停放支持与容器控件相连，而不是与窗体相连。面板、控件栏及其容器（从技术角度讲，任何控件都派生自TWinControl类）都可以通过设置它们的DockSite属性实现停放功能。还可以设置这些容器的AutoSize属性，这样仅在它们实际管理控件时才显示它们。

为了能将控件（任何TControl派生类的对象）拖进停放区，只需将其DragKind属性设为dkDock，将其DragMode属性设置为dmAutomatic。这样，可以将控件从其当前位置拖进新的定位容器。为了解除组件的停放状态并将它移到一个特殊窗体上，可以将FloatingDockSiteClass属性设置为TCustomDockForm（使用一个有小标题的预定义独立窗体）。

所有停放与解除停放的操作都可以被跟踪，这可以通过使用被拖组件（OnStartDock与OnEndDock）与接受停放组件（OnDragOver与OnDragDrop）的特殊事件来实现。这些停放事件与拖动事件非常相似。

还有一些命令可以用于在代码中实现定位操作并研究定位容器的状态。每个控件都可以使用Dock、ManualDock与ManualFloat对象方法被移到另一个位置。容器有一个说明停放控件数量的DockClientCount属性，以及一个含有这些控件的DockClients属性的数组。

而且，如果将定位容器的UseDockManager属性设置为True，就可以使用DockManager属性，该属性可以实现IDockManager界面，其中含有很多特性，可以用来定制停放容器的行为，甚至包括对流化处理状态的支持。

从这段简单的介绍中可以看出，Delphi中的停放支持基于大量的新属性、新事件、新对象方法以及新对象（如停放区与停放树）——我们无法在此详细地介绍所有这些新特征。下一章将介绍通常需要的主要特性。

说明：定位支持不能在任意平台的VisualCLX上获得。

在ControlBars中定位ToolBar

在已介绍过的MdEdit2示例中，我们介绍了定位支持。该程序在窗体的底部有第一个ControlBar，它可以实现在ControlBar顶部拖动一个工具栏的功能。因为两个工具栏容器的AutoSize属性均设置为True，所以当它们没有包含任何控件时，它们将被自动删除。我们也已将ControlBar的AutoDrag和AutoDock两个属性设置为True。

在实例中不得不将ControlBar和RichEdit控件放置在面板内。当激活和自动调整大小时，如果不使用这种技巧，ControlBar将继续移动下面的状态栏，我想这不是正确的动作。因为，在该示例中，ControlBar是与底部对齐的惟一控件，这一点不容质疑。

为了允许用户将工具栏拖出原始容器，程序员只需再次将DragKind属性设为dkDock并将DragMode属性设为dmAutomatic即可（与前面介绍的一样）。只有两个例外：一个是菜单工具栏，它不像组合框，该组件不暴露DragMode和DragKind属性（实际上，在示例的FormCreate对象方法中，将发现用于激活组件停放功能的代码，该代码是建立在第2章介绍的“盗用受保护的数据”理论基础上的）。字体组合框能被拖动，但这里不想让用户将它停放进较低的控件栏。为了实现这一限制，笔者通过只为工具栏接受停放操作，使用了控件栏的OnDockOver事件处理器：

```
procedure TFormRichNote.ControlBarLowerDockOver(Sender: TObject;  
    Source: TDragDockObject; X, Y: Integer; State: TDragState;  
    var Accept: Boolean);  
begin  
    Accept := Source.Control is TToolbar;  
end;
```

警告：不能直接把工具栏从上部的控件栏拖到下面的控件栏。在拖动过程中控件栏不能调整大小来适应工具栏，因为在拖动工具栏到浮动窗口再到下面的控件栏时，它调整了大小。这个问题发生在VCL中，并且很难更正。就像下一个范例，即使MdEdit3使用同样的代码，但还是如预想的那样：不同的VCL支持的代码将使用不同的组件。

当把一个工具栏移出任何容器时，Delphi会自动建立一个浮动窗体，大家可能会试着通过关闭浮动窗体将控件栏返回原地。但这种操作是不会成功的，因为浮动窗体会与其含有的工具栏一同被删除。然而，可以使用最顶层ControlBar的快捷菜单显示该隐藏工具栏。

Delphi创建的用于安放未锁控件的浮动窗体有一个窄标题，也叫做工具栏标题，在默认情况下，它不含任何文本。因为该原因，所以笔者向每个可停放控件的OnEndDock事件都添加了一些代码，以设置新创建的用于停放控件的标题文本。为了避免该信息有定制的数据结构，笔者使用了这些控件（基本上不使用）的Hint属性文本来提供一个适当的标题：

```
procedure TFormRichNote.EndDock(Sender: TObject; X, Y: Integer);
begin
    if Target is TCustomForm then
        TCustomForm(Target).Caption := GetShortHint((Sender as TControl).Hint);
end;
```

在图6.13的MdEdit2程序示例中，读者能看到该程序的效果。



图6.13 MsEdit2范例允许大家在窗体的顶部或底部停放工具栏（不是菜单）或让它们悬浮

另外一个笔者没有做的扩展是在窗体的两边添加附加的停放位置。该要求惟一的作用是将工具栏由水平变成垂直。在取消其自动调节大小的能力后，需要调整每个按钮的Left属性和Top属性。

控制停放操作

Delphi提供了很多事件与对象方法，用于对控件执行停放操作，包括一个停放管理器。为了研究这些特性，这里建立了DockTest范例，用于测试停放操作，如图6.14所示。

该程序处理了一个停放主面板的OnDockOver与OnDockDrop事件，用以向用户显示消息，如当前停放的控件数量：

```
procedure TForm1.Panel1DockDrop(Sender: TObject; Source: TDragDockObject;
    X, Y: Integer);
begin
    Caption := Docked: ' + IntToStr (Panel1.DockClientCount);
end;
```

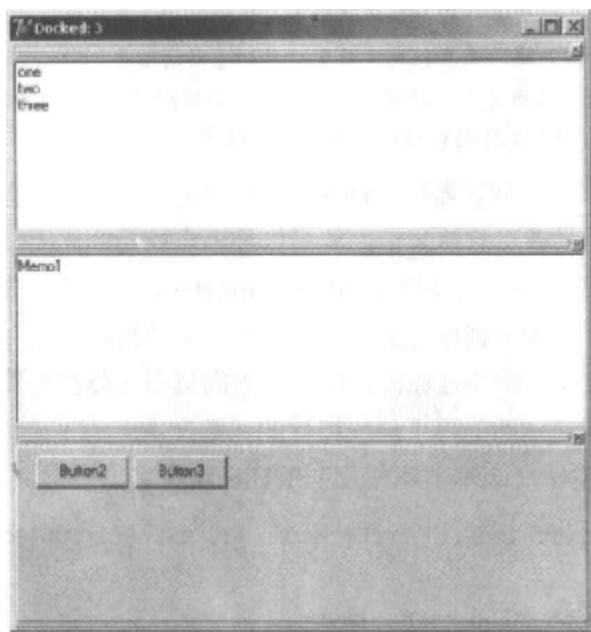



图6.14 DockTest范例，三个停放在主窗体中的控件

同样，该程序还处理了主窗体的停放事件。该控件具有快捷菜单，可用于在代码中执行停放与解除停放操作，而不用通常的鼠标拖动操作：

```

procedure TForm1.menuFloatPanelClick(Sender: TObject);
begin
    Panel2.ManualFloat (Rect (100, 100, 200, 300));
end;

procedure TForm1.Floating1Click(Sender: TObject);
var
    aCtrl: TControl;
begin
    aCtrl := Sender as TControl;
    // toggle the floating status
    if aCtrl.Floating then
        aCtrl.ManualDock (Panel1, nil, alBottom)
    else
        aCtrl.ManualFloat (Rect (100, 100, 200, 300));
    end;

```

为了使程序在开始时能正常执行，必须在初始代码中将控件停放在主面板上，否则，会得到不寻常的效果。奇怪的是，为了使程序执行正常，需要将控件添加到停放管理器中，并在面板中停放（这两个操作不能自动相互调用）：

```

// dock memo
Memo1.Dock(Panell, Rect (0, 0, 100, 100));
Panell.DockManager.InsertControl(Memo1, alTop, Panell);
// dock listbox
ListBox1.Dock(Panell, Rect (0, 100, 100, 100));

```

```

Panel1.DockManager.InsertControl(ListBox1, alLeft, Panel1);
// dock panel2
Panel2.Dock(Panel1, Rect (100, 0, 100, 100));
Panel1.DockManager.InsertControl(Panel2, alBottom, Panel1);

```

范例的最后一个特征可能也是最有趣和最难实现的。每当程序关闭时，它会通过停放管理器的支持功能保存面板的当前停放状态。当程序重新打开时，它会利用停放信息，恢复窗口的前一次配置。下面是用于保存与装载的代码：

```

procedure TForm1.FormDestroy(Sender: TObject);
var
    FileStr: TFileStream;
begin
    if Panel1.DockClientCount > 0 then
        begin
            FileStr := TFileStream.Create (DockFileName, fmCreate or fmOpenWrite);
            try
                Panel1.DockManager.SaveToStream (FileStr);
            finally
                FileStr.Free;
            end;
        end
    else
        // remove the file
        DeleteFile (DockFileName);
    end;

procedure TForm1.FormCreate(Sender: TObject);
var
    FileStr: TFileStream;
begin
    // initialization code above...
    // reload the settings
    DockFileName := ExtractFilePath (Application.Exename) + 'dock.dck';
    if FileExists (DockFileName) then
        begin
            FileStr := TFileStream.Create (DockFileName, fmOpenRead);
            try
                Panel1.DockManager.LoadFromStream (FileStr);
            finally
                FileStr.Free;
            end;
        end;
    Panel1.DockManager.ResetBounds (True);
end;

```

全部控件初始停放后代码就可以运行了。保存程序时如果有控件是浮动的，则再装载设置时它便不可见。然而，因为很早插入的初始化程序总能使控件在面板中定位，故拖走其他控件时它将显示出来。到此为止吧，可能有些乱。因此加载设置后，添加了以下代码：

```
for i := Panel1.DockClientCount - 1 downto 0 do
begin
  aCtrl := Panel1.DockClients[i];
  Panel1.DockManager.GetControlBounds(aCtrl, aRect);
  if (aRect.Bottom - aRect.Top <= 0) then
  begin
    aCtrl.ManualFloat (aCtrl.ClientRect);
    Panel1.DockManager.RemoveControl(aCtrl);
  end;
end;
```

完整的程序清单包含更多的注释代码，用于开发该程序。使用它可以理解发生了什么（经常不同于预想）。简单地说，在停放管理器中没有大小设置（知道没有停放的惟一方式）的控件显示在浮动窗口，并从停放管理器列表中删除。

如果分析OnCreate事件的完整代码，将会发现该代码很复杂停放，但只完成简单的功能。要向停放程序中添加更多功能需要删除其他的功能，因为可能有些会相互冲突。添加定制停放窗体减弱了停放管理器的功能。自动对齐与定位管理器用于恢复的代码就不能很好地配合。建议读者下载该程序，研究其性能，扩展它来支持自己喜欢的用户界面类型。

说明：记住，尽管定位面板功能使应用程序看上去很好，但是一些用户会被这样的事实所困惑，他们常用的工具栏可能没有出现，或出现在他们不习惯的位置。不要过分地使用定位特性，否则一些经验不多的用户会被迷惑。

定位到PageControl

页面控件的另一个有趣的特性是支持定位功能。当将一个控件定位到PageControl上时会自动添加一个新页面来管理它，就像在Delphi环境中常看到的那样。为了实现该操作，只需将PageControl设置为定位容器，并为客户控件激活定位特性即可。该技术最好用于有二级窗体需要放置的情况下。而且，如果可以将整个PageControl移到一个浮动窗口中，然后定位它，就需要在主窗体中设计一个定位面板。

这其实可以用DockPage范例来解释，其主窗体有以下设置：

```
object Form1: TForm1
  Caption = 'Docking Pages'
  object Panel1: TPanel
    Align = alLeft
    DockSite = True
    OnMouseDown = Panel1MouseDown
  object PageControl1: TPageControl
    ActivePage = TabSheet1
    Align = alClient
    DockSite = True
```

```

    DragKind = dkDock
    object TabSheet1: TTabSheet
        Caption = 'List'
        object ListBox1: TListBox
            Align = alClient
        end
    end
end
end
end
object Splitter1: TSplitter
    Cursor = crHSplit
end
object Memo1: TMemo
    Align = alClient
end
end
end

```

注意，面板将UseDockManager属性设置为True，而且PageControl可以使用一个列表框来管理一页，因为删除所有页面时，自动设置定位容器大小的代码会出现问题。

该程序另有两个窗体，具有相似的设置（尽管它们管理的控件不同）：

```

object Form2: TForm2
    Caption = 'Small Editor'
    DragKind = dkDock
    DragMode = dmAutomatic
    object Memo1: TMemo
        Align = alClient
    end
end
end

```

可以将这些窗体拖到页面控件上，向它添加新页面和对应着窗体标题的标题。还可以解除这些控件甚至是整个PageControl的定位功能。为了实现该操作，程序没有使用自动拖动特性，这样就不再能切换页面了。当用户单击PageControl（不带标签）的控件区（也就是下面的面板）时，会激活该特性：

```

procedure TForm1.PanelMouseDown(Sender: TObject; Button: TMouseButton;
    Shift: TShiftState; X, Y: Integer);
begin
    PageControl1.BeginDrag (False, 10);
end;

```

通过运行DockPage范例可以测试该操作，图6.15显示了它的效果。请注意，当从主窗体中删除PageControl时，我们不能直接将其他窗体停放在面板上，因为这是被程序的特定代码所禁止的。



图6.15 DockPage范例的主窗体，在一个窗体被定位到左边的页面控件之后生成

ActionManager体系结构

我们在前面已经看到，在开发Delphi应用程序的过程中，各种动作（操作）与ActionManager组件起了重要作用，因为它允许将用户界面从实际应用程序代码中分离出来。事实上，不需要过多地修改代码就可以轻易改变用户界面。该方法的缺点是程序员的工作量被加大了。为了获得新菜单项，首先需要添加对应的动作，然后移动到菜单，添加菜单项并将它连接到动作上。

为了解决该问题并向开发商和最终用户提供高级特征，Delphi 6基于ActionManager组件引入了一个新的体系结构，该结构大大地扩展了动作的作用。事实上，ActionManager不仅有一个动作的集合，也有一个工具栏和菜单集合连接到它们。这些工具栏和菜单的开发是直观的：将动作从ActionManager的特殊组件编辑器拖动到有需要按钮的工具栏中，而且，可让程序的最终用户有同样的操作，从提供给它们的动作开始重新排列它们自己的工具栏和菜单。

换句话说，使用该结构允许建立拥有现代用户界面的应用程序，而且由用户定制。菜单只能显示最近使用的项（如许多微软程序一样），允许显示动画以及其他内容。

该结构集中了ActionManager组件，但它也包括几个在调色板Additional页面最后的其他组件：

- ActionManager组件是ActionList的替代品（也能使用一个或更多现有的ActionList）。
- ActionMainMenuBar控件是一个工具栏，用于显示基于ActionManager组件动作的应用程序菜单。
- ActionToolBar控件是一个基于ActionManager组件动作的、用来放置按钮的工具栏。
- CustomizeDlg组件包括能让用户基于ActionManager组件定制应用程序用户界面的对话框。
- PopupActionBarEx组件是应该使用的特殊组件，它使弹出菜单与主菜单等用户界面保持一致。Delphi 7不包含该组件，但可以单独下载。

提示：可以在Borland的网上代码中心库（18870号）找到PopupActionBarEx组件（也称ActionPopupMenu）。此外，可以在组件作者的主页（homepages.borland.com/strefethen）看到更多信息。作者是Borland的Delphi研发组成员，站点上的组件并没有获得官方支持。

建立一个简单的演示范例

因为该结构主要是一个可视结构，所以通过演示比进行一般的讨论更有意义（虽然印制的图书不是介绍一系列高度可视操作的最好方法）。为了基于该结构创建一个程序范例，首先向窗体中拖入一个ActionManager组件，然后双击它，打开它的编辑组件，如图6.16所示。注意，该编辑器不是模式的，因此当在Delphi上进行其他操作时，可以使它始终保持打开状态。假设同一个对话框也被CustomizeDlg组件展示，尽管具有一些受限特征（例如，添加的新动作被设为无效）。

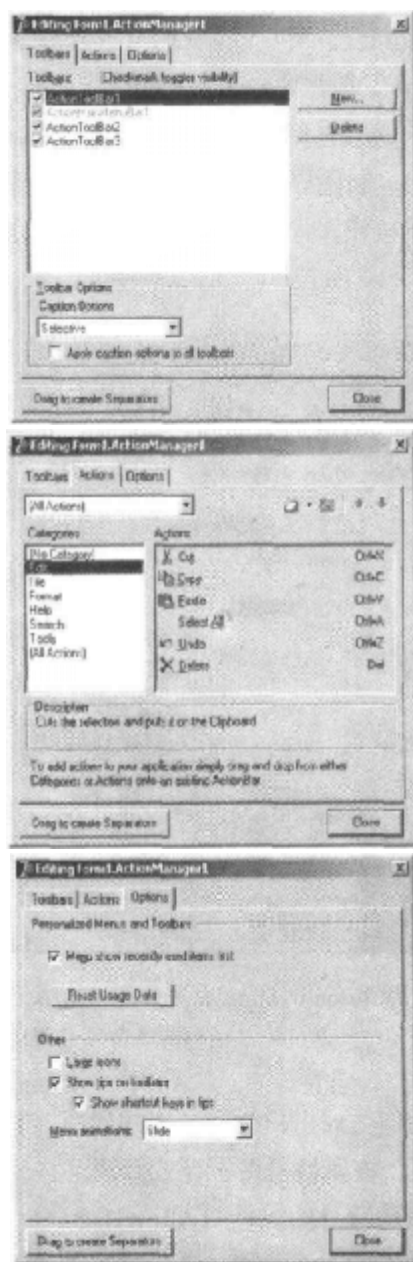


图6.16 ActionManager编辑器对话框的三个页面

下面介绍这个编辑器的三个页面：

- 该编辑器的第一页提供动作的可视容器列表（工具栏或菜单）。程序员能通过单击New按钮添加新工具栏。为了添加新菜单，我们必须向窗体添加对应组件，然后打开ActionManager的ActionBar集合，挑选动作栏或添加新栏，使用ActionBar属性使它与菜单相连。在运行时，将新工具栏连接到该结构也遵循相同的步骤。
- ActionManager编辑器的第二页与ActionList编辑器十分相似，提供一种新方法添加新的标准或定制动作，并根据种类处理它们并改变它们的顺序。该页的一个很好的特性是能从页上拖动类或单个动作，并把它拖动到动作栏控件中。如果将某种类拖动到菜单内，我们将获得一个包含该种类所有项的菜单；如果我们将它拖动到工具栏中，种类的每个动作在工具栏上获得与之相应的按钮。如果拖动它到菜单，将获得一条菜单命令，这是应避免的。
- ActionManager编辑器的最后一页允许我们（或者是一个末端用户）激活最近使用的菜单项的显示，并修改工具栏的一些可视属性。

AcManTest程序是一个示例，它使用一些标准动作和RichEdit控件来显示该结构的优点（这里没有使动作效果更好的定制代码，因为我只是想关注范例的动作管理）。在设计时可以实验它或运行它，单击定制按钮，了解最终用户在定制应用程序时能做什么（如图6.17所示）。

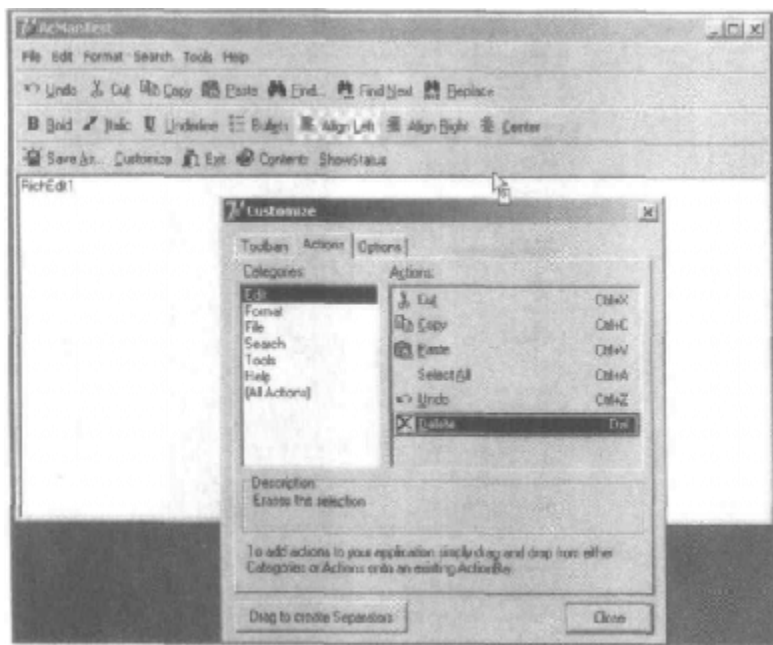


图6.17 使用CustomizeDlg组件能让用户定制工具栏和应用程序菜单——通过从对话框拖动各项或围绕动作栏移动它们

在程序中可以防止各动作项被用户改动。用户界面（TActionClient对象）的任何特殊元素都有ChangedAllowed属性，该属性可用于无效修改、移动和删除操作。任何动作客户容器（可视栏）都有一个取消隐藏自身的属性（AllowHiding默认设置为True）。每个ActionBar的Items集合都有一个能关闭的Customizable选项使所有用户都无法改变整个栏。

提示：我们所介绍的ActionBar并不意味着可视工具栏包含动作项，但ActionManager集合的ActionBar项在运行时有项集合。理解该结构的最好方法是看一下TreeView对象为ActionManager组件显示的子树。每个TActionBar收藏项都有一个连接实际TCustomizableBar的可视组件，但反之则不行（例如，如果挑选可视工具栏启动，就不能使用该定制属性）。由于这两个名字具有相似性，故理解Delphi的实际意义需要花费一些时间。

为了使用户设置持久，我连接了一个文件（叫settings）到ActionManager组件的FileName属性上。当赋值该属性时，应该输入所需文件的名称；当启动该程序时，ActionManager文件将被创建。每个连接动作管理器的ActionClientItem流确保了程序的持续性。因为这些动作客户项基于用户设置和保存的状态信息，且使用单独的文件来记录用户对界面的改变和使用的数据。

因为Delphi要在前面提供的文件中存储用户设置和信息状态，所以可在单个计算机上使用应用程序支持多个用户。当程序启动时，对每个用户（在MyDocuments或MySettings虚拟目录下）简单地使用设置文件并连接它到动作管理器上（使用计算机的当前用户或经过定制注册）。另一个可能的方法是在网络上存储这些设置，以便当用户移到不同的计算机时，当前的个人设置也能随着移动。

该程序将在其文件夹下的文件库中保存设置，这可通过将相应的路径（文件名）赋值给ActionManager的FileName属性来实现。为了便于找到文件并加载，该组件将添加包含程序目录的全部文件名。然而，该文件在它的数据中包括的是绝对路径的文件名。所以，当保存文件时，将使用旧的路径。这样就不能复制程序及其配置到不同的目录中（例如，这是AcManTest演示的核心问题）。可以在加载文件后重新设置FileName属性。更好的方式是在运行时，在窗体的OnCreate事件中设置文件名。在此例中，不得不指定该文件以再次加载它，因为在ActionManager组件和ActionBar被创建和加载后，正在设置该文件。然而，加载后可能要如下指定文件名：

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  ActionManager1.FileName :=
    ExtractFilePath (Application.ExeName) + 'settings';
  ActionManager1.LoadFromFile(ActionManager1.FileName);
  // reset the settings file name after loading it (relative path)
  ActionManager1.FileName :=
    ExtractFilePath (Application.ExeName) + 'settings';
end;
```

最近不常使用的菜单项

--一旦用于用户设置的文件可用，ActionManager将把它保存到用户的收藏夹中，并将使用它追踪用户活动。这是最基本的方法，可使系统删除不经常使用的菜单项，同时使它们可以出现在扩展菜单中，以保持与微软的用户界面具有相同的风格（请参考图6.18中的例子）。

ActionManager并不只是简单地显示最近不常使用的项：ActionManager还允许我们以一种很准确的方法定制该动作。每个动作工具栏都有跟踪应用程序被执行次数的SessionCount属性。每个ActionClientItem都有LastSession属性和UsageCount属性用于跟踪用户操作。顺便

提一下，使用定制对话框的Reset Usage Data按钮，可使用户重新设置所有动态信息。

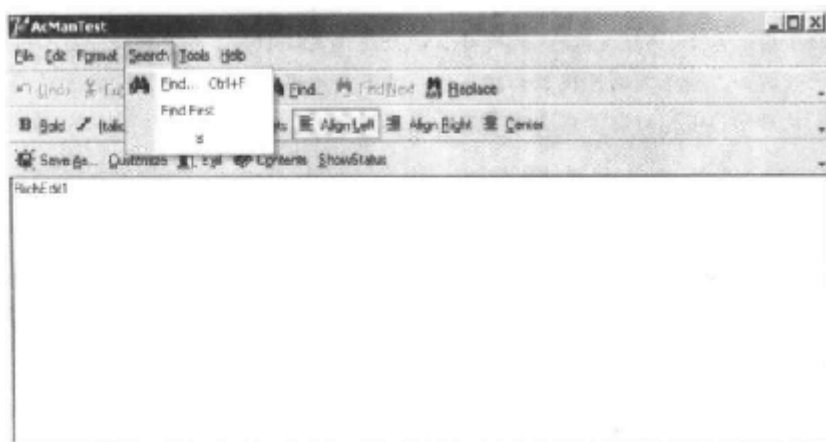


图6.18 ActionManager禁用了最近使用的菜单项，我们仍然可以通过选择菜单的扩展命令来看到它们

系统会计算没有使用某动作的会话次数，这是通过计算应用程序已被执行的次数（SessionCount）和动作被最后使用时的次数（LastSession）之差而得到的。Usage Count的值用于在PrioritySchedule中计算相应会话未被使用的次数，换句话说，PrioritySchedule返回了unused的会话次数。通过对PrioritySchedule做修改，程序员就可以得知它未被使用的次数，并决定何时删除它。

对于特殊动作或动作组，也能防止该系统被激活。ActionManager的ActionBar中的Items属性有一个HideUnused属性，对于整个菜单，该属性能使特征无效。为了使一个特定的项总是可见，无论它的用途是什么，都可设置UsageCount属性为-1。然而，用户的设置可以覆盖该值。

为了更好地理解该系统是怎样工作的，我已添加一个定制动作（ActionShowStatus）到AcManTest示例。该动作具有下列代码，这些代码能将当前动作管理器的设置保存到内存流中，转变它为文本并在备注中显示它（关于流的详细介绍见第4章）：

```
procedure TForm1.ActionShowStatusExecute(Sender: TObject);
var
  memStr, memStr2: TMemoryStream;
begin
  memStr := TMemoryStream.Create;
  try
    memStr2 := TMemoryStream.Create;
    try
      ActionManager1.SaveToStream(memStr);
      memStr.Position := 0;
      ObjectBinaryToText(memStr, memStr2);
      memStr2.Position := 0;
      RichEdit1.Lines.LoadFromStream(memStr2);
    finally
      memStr2.Free;
    end;
  end;
end;
```

```

    end;
  finally
    memStr.Free;
  end;
end;
end;

```

我们获得的输出是文本格式的settings文件，它在每一次程序被执行的时候进行自动的更新。下面是这个文件的一小部分，包括一个下拉菜单的详细内容以及丰富的注释信息：

```

item // File pulldown of the main menu action bar
Items = <
  item
    Action = Form1.FileOpen1
    LastSession = 19 // was used in the last session
    UsageCount = 4 // was used four times
  end
  item
    Action = Form1.FileSaveAs1 // never used
  end
  item
    Action = Form1.FilePrintSetup1
    LastSession = 7 // used some time ago
    UsageCount = 1 // only once
  end
  item
    Action = Form1.FileRun1 // never used
  end
  item
    Action = Form1.FileExit1 // never used
  end>
Caption = '&File'
LastSession = 19
UsageCount = 5 // the sum of the usage count of the items
end

```

移植现有程序

如果该结构是可用的，我们将要使用它的优点来重做大部分应用程序。然而，如果已经使用了动作（用ActionList组件），相应的变化将更简单。事实上，ActionManager有它自己的动作集合，它也能使用来自另一个ActionManager或ActionList的操作。ActionManager的LinkedActionList属性是动作容器的一个集合（ActionList或ActionManager），与当前的ActionManager相关联。联合所有各种动作对用户用单个对话框定制整个用户界面很有帮助。

如果连接外部动作并打开ActionManager编辑器，将看到在Actions页面中，一个组合框显示了当前的ActionManager以及其他指向它的动作容器。我们能选择这些容器中的一个来看一下它的动作设置并改变它们的属性。该组合框的All Action选项允许我们选择来自各种

容器的所有动作选项进行工作。重新选择它来查看所有的动作。

作为一个移植现存应用程序的示例，通过本章，笔者扩展了MdEdit3示例的程序结构。该示例使用老版本中的相同的动作列表连到ActionManager，它有一个额外的定制属性，能使用户重新编排用户界面。与以前的ActionDemo程序不同，对于动作栏，MdEdit3示例使用ControlBar作为容器（一个菜单、三个工具栏和常用的组合框）并支持像浮动栏那样的功能，拖动它们到容器外并把它们拖动到较低的ControlBar中。

为了完成它，我们只需简单地修改源代码，用ControlBarLowerDockOver对象方法引用容器的新类（也就是，用TCustomActionToolBar代替TToolBar）。读者也会发现当系统创建一个浮动窗体来放置控件时，ActionToolBar组件的OnEndDock事件将作为一个空目标参数被传递，因此不能轻易地给该窗体定制新标题（参考窗体的EndDock对象方法）。

使用列表操作（List Actions）

在MDI和数据库编程章节中（第8章和第13章）将看到更多有关该结构的使用示例。现在，只添加一个额外示例显示怎样使用Delphi介绍的更复杂的标准动作组，即列表操作。事实上，列表操作有两个不同的组。一些动作（如移动、复制、删除、清除与全部选择）是正常动作，工作在列表框或其他列表中。相反，VirtualListAction和StaticListAction元素基于多个选择来定义操作，它们作为组合框被显示在工具栏中。

ListActions演示程序演示了两组列表动作，因为它的ActionManager中可提供其中的五个，并显示在两个分开的工具栏中。下面是一个ActionManager的动作总结（笔者已删除部分组件的DFM文件的动作栏）：

```
object ActionManager1: TActionManager
  ActionBar.SessionCount = 1
  ActionBar = <...>
  object StaticListAction1: TStaticListAction
    Caption = 'Numbers'
    Items.CaseSensitive = False
    Items.SortType = stNone
    Items = <
      item
        Caption = 'one'
      end
      item
        Caption = 'two'
      end
    ...>
    OnItemSelected = ListActionItemSelected
  end
  object VirtualListAction1: TVirtualListAction
    Caption = 'Items'
    OnGetItem = VirtualListAction1GetItem
    OnGetItemCount = VirtualListAction1GetItemCount
    OnItemSelected = ListActionItemSelected
```

```

end
object ListControlCopySelection1: TListControlCopySelection
    Caption = 'Copy'
    Destination = ListBox2
    ListControl = ListBox1
end
object ListControlDeleteSelection1: TListControlDeleteSelection
    Caption = 'Delete'
end
object ListControlMoveSelection2: TListControlMoveSelection
    Caption = 'Move'
    Destination = ListBox2
    ListControl = ListBox1
end
end
end

```

程序的窗体中也有两个列表框，作为动作目标使用。通过它们的ListControl和Destination属性，复制和移动动作被连接到这两个列表框上。相反，在拥有输入焦点的列表框中，删除动作将自动生效。

StaticListAction在它的Items集合中定义了一系列可选项。这不是一个无格式的字符串列表，因为每一项都带有一个ImageIndex，它允许我们在显示这个列表的空间中添加图像元素。当然，我们能添加更多项到该列表程序中。然而，在更有活力的列表中，还可以使用组件VirtualListAction。该组件没有定义各项的列表，但有两个能为列表提供字符串和图片的事件。其中OnGetItemCount事件允许程序说明要显示的项数，而OnGetItem事件被每个特定的项调用。

在ListAction演示中，VirtualListAction对于它的定义提供了下面的事件处理程序，创建出如图6.19所示的活动组合框中的列表：

```

procedure TForm1.VirtualListAction1GetItemCount(Sender: TCustomListAction;
    var Count: Integer);
begin
    Count := 100;
end;

procedure TForm1.VirtualListAction1GetItem(Sender: TCustomListAction;
    const Index: Integer; var Value: String;
    var ImageIndex: Integer; var Data: Pointer);
begin
    Value := 'Item' + IntToStr (Index);
end;

```

说明：虚拟动作列表只在需要显示它们时才会被请求，这才是虚拟动作列表的真正含义。如果将VirtualListAction1GetItem方法中的注释代码全部激活（并非前面所列那些代码），那么所有的项目都将被正确地创建，因为这些代码将在用户请求与项目相关的字符串时，添加相应的项目。

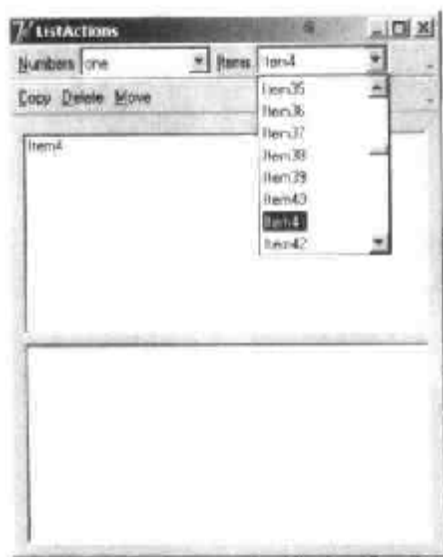


图6.19 ListActions应用程序带有一个工具栏，它有一个静态列表和一个虚拟列表

静态和虚拟列表都有一个**OnItemSelected**事件。在共享事件处理程序中，笔者编写了下列代码，它将当前项添加到窗体的第一个列表框中：

```
procedure TForm1.ListActionItemSelected(Sender: TCustomListAction;  
    Control: TControl);  
begin  
    ListBox1.Items.Add ((Control as TCustomActionCombo).SelText);  
end;
```

在此例中，发送方是定制的动作列表，但该列表的**ItemIndex**属性没用使用选中的项进行更新。然而，通过访问显示该列表的可视控件时，可以获得选中项的值。

小结

在本章中，我们介绍了动作、动作列表的使用，和**ActionManager**的结构。如读者所见到，这是一个十分强大的结构，能从应用程序的实际代码中分离出用户界面，而且使用并引用动作而不是菜单项或相关的工具栏按钮。该结构的最新扩展允许程序用户拥有许多控件，使程序达到最好的效果而不影响其他部分。同样该结构对于我们设计程序的用户界面也是十分有益的，不管是否会将该功能提供给用户。

本章还介绍了用户界面的其他技巧，如定位工具栏和其他控件。本章可以作为创建专业应用程序的第一步。在下面的章节中，我们将继续对此课题进行介绍，但读者已经知道了许多能使自己的应用程序具有**Windows**应用程序风格的方法，这对于我们的客户十分重要。

现在程序主窗体所需的元素已经建立好了，我们可以继续添加第二个窗体和对话框。这是下一章要讨论的问题，此外还有一些对窗体的综合介绍。第8章中将涉及**Delphi**应用程序的全部结构。

第7章 使用窗体

经过前面几章的学习，相信读者已经能够用Delphi的可视组件为自己的应用程序创建用户界面了。现在让我们把注意力转移到Delphi开发的另一个核心要素上，这就是窗体。在初始化的内容中已经使用了窗体，但是还没有详细描述如何用窗体，没有描述TForm类的哪些属性、方法应引起注意。

本章将介绍一些窗体的属性和样式、大小和定位，以及缩放和滚动。还将介绍有多个窗体的应用程序、对话框的使用（定制和预定义）、结构和可视窗体的继承性。此外，还将花些时间讨论使用鼠标和键盘在窗体上输入的问题。

本章主要包括以下内容：

- 窗体样式、边框样式和边框图标
- 鼠标和键盘输入
- 窗体绘图和特殊效果
- 定位、缩放和滚动窗体
- 创建与关闭窗口体
- 模态与非模态对话框和窗体
- 动态创建二级窗体
- 预定义对话框
- 建立Splash屏幕

TForm类

Delphi窗体是由TForm类定义的，包含在VCL的窗体单元中。当然，在可视CLX内有窗体的第二种定义。尽管本章将主要介绍VCL类，但也将强调与CLX提供的交叉版本的不同之处。

TForm类是窗口控件层的一部分，由TWinControl（或TWinWidgetControl）类开始。实际上，TForm继承了几乎整个TCustomForm类，而TCustomForm类又继承自TScrollingWinControl（或TScrollingWidget）。由于拥有它们的许多基类特征，故窗体有一大套对象方法、属性和事件。基于这个原因，在这里将不对它们过多地介绍，但本章还是要展示一些与窗体有关的有趣技术。我们首先要展示一种在设计时不定义程序窗体的技术——直接使用TForm类，然后探索几个窗体类的属性。

本章也将介绍VCL窗体和CLX窗体的几个不同之处。笔者已建立了一个CLX版本用于本章的大多数示例，因此可以用CLX和VCL一起来检测窗体和对话框。读者也许没有忘记，在前面几章中，CLX版本的每个示例都是以字母Q为前缀的。

使用简单窗体

通常，Delphi开发者往往是在设计时创建窗体的，这意味着要从基类派生一个新类并创建可视的窗体内容。这当然是一个合理的标准习惯，但并不是必须得创建一个TForm类的子类来显示窗体，特别是当它是一个简单窗体时。

考虑下面这种情况：要向用户显示一个相当长的信息（基于字符串），并且不想使用简单的预定义信息框，因为它显示起来太大，还不能提供滚动栏。此时需要创建一个含备注组件在内的窗体，并在里面显示字符串。没人能阻止以标准的可视方式创建该窗体，但应该考虑用代码创建该窗体，特别是当需要很大程度的弹性时。

DynaForm与QDynaForm示例（本书所带源代码）在某种程度上有些极端，它们没有在设计时定义窗体，但包含一个具有该功能的单元：

```
procedure ShowStringForm (str: string);
var
  form: TForm;
begin
  Application.CreateForm (TForm, form);
  form.caption := 'DynaForm';
  form.Position := poScreenCenter;
  with TMemo.Create (form) do
  begin
    Parent := form;
    Align := alClient;
    Scrollbars := ssVertical;
    ReadOnly := True;
    Color := form.Color;
    BorderStyle := bsNone;
    WordWrap := True;
    Text := str;
  end;
  form.Show;
end;
```

在此，必须使用Application的全局对象的CreateForm方法创建窗体（Delphi应用程序需求的条件将在下一章讨论）。除此之外，该代码动态地完成了通常窗体设计器要做的事情。毫无疑问，这些代码过于冗长，但该程序具有更大的灵活性，这是因为使用的所有参数都依赖外部设置。

上面示例的ShowStringForm函数不能被另一个窗体的事件执行，因为在该程序上没有传统窗体。以下是修改后的源代码：

```
program DynaForm;

uses
  Forms,
  DynaMemo in 'DynaMemo.pas';
```

```

{$R *.RES}

var
  str: string;

begin
  str := '';
  Randomize;
  while Length (str) < 2000 do
    str := str + Char (32 + Random (74));
  ShowStringForm (str);
  Application.Run;
end.

```

DynaForm程序的运行结果是一个充满任意字符的陌生窗体（如图7.1所示），其本身并不十分有用，但有助于读者理解它的整体思想。

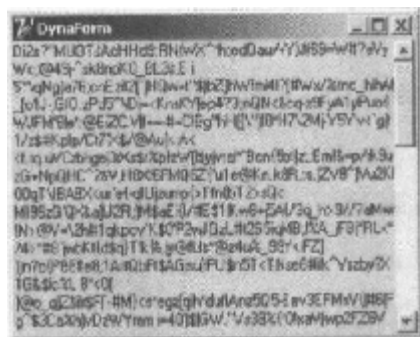


图7.1 由DynaForm范例生成的动态窗体在运行时被完全创立，不提供设计时支持

提示：与设计时窗体的DFM文件的使用相比较，该方法一个间接的优点是，对于一个外部程序员来说，很难理解应用程序结构的信息。在第5章中，我们知道能从当前的Delphi执行文件中抽取DFM，但是对于用Delphi编辑的任意可执行文件（我们并不知道源代码）来说，都能轻松完成这一工作。对于我们来说，如果要保存使用的特殊组件（也许在一个特定的窗体中），及其属性的默认值，则编写额外代码是值得考虑的。

窗体类型

FormStyle属性允许在一个正常窗体（fsNormal）和用以组成一个多文档界面（MDI）应用程序的窗口之间进行选择。在这种情况下，将使用fsMDIForm样式的MDI父窗口，它是MDI应用程序的框架窗口，并且将fsMDIChild样式用于MDI子窗口。参见第8章，可以了解MDI应用程序的开发。

fsStayOnTop样式是第四个选择，它决定窗体是否总保持在所有其他窗口的顶部（除了“stay-on-top”类型的窗口）。

要建立top-most窗体（窗口总处于前面的窗体），只需要像上面描述的那样设置FormStyle属性。该属性根据应用窗体的类型，可以有两种不同的效果：

- 应用程序的主窗体将始终处于其他任何应用程序之前（除非其他应用程序也有相同的top-most类型）。此时就会产生一个非常难看的可视结果，因此它只用于特殊目的

的警告程序。

- 二级窗体将处于它所属应用程序的其他任何窗体之前，此时其他应用程序不受影响。它经常用于浮动工具栏和其他应停在主窗口前的窗体。

警告：在VCL中，当这个属性应用于一个二级窗体时，该窗体仅仅出现在同一应用程序中其他窗体之前。而在CLX中，甚至是一个二级窗体也有可能出现在整个窗口系统的最前端，这是我们应该避免的。

边框类型

窗体的另一个重要属性是**BorderStyle**。该属性引用了窗体的一个可视元素，但对窗口有更为深刻的影响，如图7.2所示。

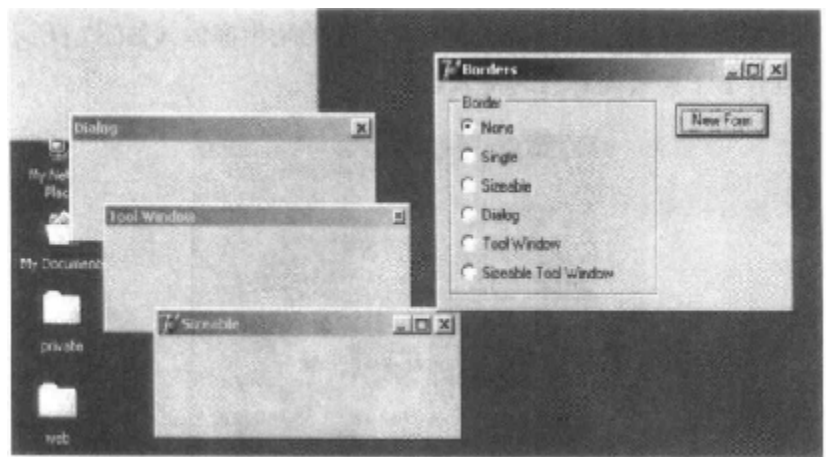


图7.2 由Borders范例创建的带有各种边界类型的范例窗体

在设计时，窗体总是使用**BorderStyle**属性的默认值**bsSizeable**来显示它，这对应着一个叫做**thick frame**的Windows样式。当主窗口有一个粗边框时，用户可以拖动边框改变窗口的大小。当用户将鼠标移动到窗口粗边框上时，会显示特殊的重定义大小的鼠标形状（一个双箭头的形状）来使这一操作更为清楚。

该属性的第二个重要选择是**bsDialog**，如果我们选择它，窗体将使用典型对话框边框——不能重新改变大小的粗边框。除了这一图形元素外，还要注意，如果我们选择了**bsDialog**值，窗体会变成一个对话框。这包括很多变换，例如，其系统菜单上的选项会改变，而且窗体将忽视**BorderIcons**设置属性的一些元素。

警告：设计时设置**BorderStyle**属性不会产生视觉上的影响。事实上，有些组件属性在设计时不会起作用，因为它们在程序开发时会阻止对用户组件的操作。那么举例来说，如果窗体变成了对话框，如何用鼠标改变它的大小呢？回答是：在运行应用程序时，窗体将具有我们所需的边框。

我们还可以为**BorderStyle**分配四个属性值：

- **bsSingle**值可以用来建立不能改变大小的主窗口。很多基于带有控件的窗口（如数据输入窗体）的应用程序和游戏都使用该值，只因为重新设置这些窗体大小没有意义。扩大一个窗体会使之看起来空旷，而缩小它又会使一些组件的可视程度降低，所以通常不能对程序的使用者有所帮助（尽管Delphi的自动滚动条会部分地解决后一个问题）。

- **bsNone**值只用于在其他窗体内非常特殊的情况，读者决不会见到应用程序的主窗口没有边框或标题（除了也许作为一本编程书的例子向读者表明它没有意义外）。
- **bsToolWindow**与**bsSizeToolWin**与特殊的Win32扩展类型**ws_ex_ToolWindow**有关。这种新样式将窗口转换为一个浮动工具框，带有小标题字体与关闭按钮。该样式不应用于应用程序的主窗口。

警告：在CLX中，**BorderStyle**属性的枚举使用不同的值，以字母**fb**s（窗体边框样式）为前缀。因此会有**fbSingle**、**fbDialog**等值。

为了测试**BorderStyle**属性不同值的效果及行为，本节将介绍一个简单的范例**Borders**，在CLX版本中是**QBorders**。读者在图7.2中已经看到了它的输出。然而，建议读者运行该范例，以体会和理解窗体中所有的不同。该程序的主窗体非常简单，只含有一组单选钮和一个按钮，还有一个二级窗体，其中不包含组件，**Position**属性被设置为**poDefaultPosOnly**。单击按钮会影响二级窗体的初始位置（本章稍后将讨论**Position**属性）。

程序的代码非常简单：当用户单击按钮时，根据单选钮组被选中的项，会建立一个新窗体：

```
procedure TForm1.BtnNewFormClick(Sender: TObject);
var
    NewForm: TForm2;
begin
    NewForm := TForm2.Create (Application);
    NewForm.BorderStyle := TFormBorderStyle (BorderRadioGroup.ItemIndex);
    NewForm.Caption := BorderRadioGroup.Items[BorderRadioGroup.ItemIndex];
    NewForm.Show;
end;
```

这段代码使用了一个技巧：它将选项的值转换为**TFormBorderStyle**枚举值。这之所以起作用是因为枚举的值与单选钮顺序相同。然后，由**BtnNewFormClick**方法向二级窗体的标题复制单选钮的文本。该程序引用了**TForm2**，这是在程序的第二个单元中定义的二级窗体，保存为**SECOND.PAS**。因此，为编译该程序，必须向主窗体单元的**implementation**部分添加下面两行代码：

```
uses
    Second;
```

提示：无论何时需要引用程序的另一个单元时，如果可能，可在**implementation**部分（而不是**interface**部分）放置相应的**uses**语句。这将使编译过程的速度加快，而且使代码更为整洁（因为用户使用的单元独立于Delphi使用的单元），并且也不会不同单元间产生循环引用。要实现这一操作，还可以使用File菜单的Use Unit命令。

边框图标

窗体的另一个重要元素是边框上出现的图标。默认情况下，窗口有一个连接到系统菜单的小图标、一个最大化按钮、一个最小化按钮，最右边有一个关闭按钮。程序员可以设置不同的选项来使用**BorderIcons**属性，该设置有四个可能值：**biSystemMenu**、**biMinimize**、**biMaximize**以及**biHelp**。

说明：新的biHelp边框图标可以激活“*What's this?*”帮助。当该类型被设置而不包括biMinimize与biMaximize类型时，一个问号会出现在窗体的标题栏中。如果用户单击该问号，然后单击窗体或组件，Delphi会激活一个弹出式窗口，显示有关对象的帮助信息。这可以用BIcons范例来演示，该程序有一个简单的Help文件，其页面与窗体中央位置按钮的HelpContext属性相连

为了演示带有不同边框图标的窗体行为以及显示怎样在运行时改变该属性，我又编写了一个范例，名为BIcons。该范例的窗体非常简单：它只有一个菜单，该下拉式菜单有四个选项，对应着边框图标设置的每一种可能元素。这里只编写一个对象方法，与三个命令相连，用以在菜单项上读取复选标记来确定BorderIcons属性的值。所以下面的代码也是对设置操作的很好练习：

```
procedure TForm1.SetIcons(Sender: TObject);
var
  BorIco: TBorderIcons;
begin
  (Sender as TMenuItem).Checked := not (Sender as TMenuItem).Checked;
  if SystemMenu1.Checked then
    BorIco := [biSystemMenu]
  else
    BorIco := [];
  if MaximizeBox1.Checked then
    Include (BorIco, biMaximize);
  if MinimizeBox1.Checked then
    Include (BorIco, biMinimize);
  if Help1.Checked then
    Include (BorIco, biHelp);
  BorderIcons := BorIco;
end;
```

运行BIcons范例，我们可以轻易地设置或删除窗体边框上不同的可视元素。还会看见一些元素是紧密相关的：如果我们删除系统菜单，所有的边框图标将消失；如果删除Minimize或Maximize按钮，它将变灰；如果删除这两个按钮，它们将消失。还要注意，在后两种情况中，系统菜单中相应的选项会自动关闭。这是任何Windows应用程序的标准行为。当Maximize与Minimize按钮被关闭时，我们可以激活Help按钮。在Windows 2000中，如果只有一个按钮被关闭，Help按钮将显示但不可用。作为获得该效果的捷径，可以在窗体内单击按钮。还可以在单击Help Menu图标之后，单击按钮来查看Help消息，如图7.3所示。另外作为附加功能，该程序还通过处理窗体的OnHelp事件，在标题中显示了调用Help的时间。

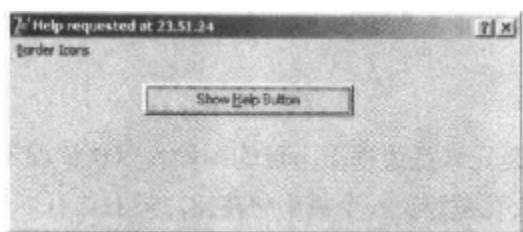


图7.3 BIcons范例。通过选择Help边界图标并且单击按钮，可以获得帮助

警告：通过查看由CLX建立的QBIcons版本，我们清楚地注意到库中有一个故障，它阻止了程序在运行时改变边框图标。也就是说，在运行时程序什么也不做。由于各种设计时的设置可以完善地工作，因此我们将需要在运行它来查看效果之前对程序进行修改。在运行时，程序什么事情都不做！

设置其他窗口样式

Delphi使用两个不同的属性指示边框类型与边框图标，而且它们可以用于设置相应的用户界面元素的初始值。我们可以看到，除了改变用户界面外，这些属性还影响着窗口的状态。需要知道，在VCL中这些与边框有关的属性以及FormStyle属性主要对应着窗口样式与扩展样式中的不同设置。这两个术语对应着Delphi用于建立窗体的CreateWindowEx API函数的两个参数

了解这一点是很重要的，因为Delphi允许程序员通过覆盖CreateParams虚拟函数来自由改变这两个参数：

```
public
    procedure CreateParams (var Params: TCreateParams); override;
```

这是使用某些特殊窗口样式的惟一办法，这些样式不能直接通过窗体属性来使用。窗口样式及扩展样式的列表见API帮助中的CreateWindow与CreateWindowEx专题。可以看到，Win32 API有大量样式与这些函数对应，包括那些与工具窗口相关的样式。

为了介绍如何使用该方法，需要编写一个简单范例，叫NoTitle，它允许我们使用定制的标题建立程序。首先必须删除标准标题，但同时需要通过设置相应的样式，保留改变了大小的窗体框架：

```
procedure TForm1.CreateParams (var Params: TCreateParams);
begin
    inherited CreateParams (Params);
    Params.Style := (Params.Style or ws_Popup) and not ws_Caption;
end;
```

为了删除标题，需要将重叠样式改为弹出样式，否则，仍将会出现标题。为了添加定制标题，这里放置了一个标题框，与窗体的上边框对齐，并在远端放置了一个小按钮。图7.4显示了该程序运行时的效果。

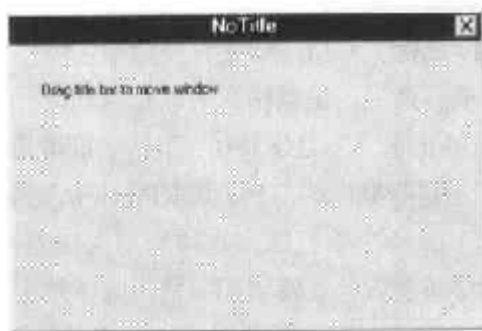


图7.4 NoTitle范例不具有实际的标题，但是使用了一个标签来作为一个假标题

为了使伪造的标题可以正常使用，必须通知系统，在该区域的鼠标操作实际上是对标题的鼠标操作。这可以通过截取`wm_NCHitTest` Windows消息来实现，该消息通常被发向Windows以确定鼠标当前的位置。当在客户区与标题框内单击时，我们可以通过设置相应的结果来表示光标在标题上：

```
procedure TForm1.WMNCHitTest (var Msg: TWMNCHitTest);
// message wm_NcHitTest
begin
  inherited;
  if (Msg.Result = htClient) and
    (Msg.YPos < Label1.Height + Top + GetSystemMetrics (sm_cyFrame)) then
    Msg.Result := htCaption;
end;
```

上述代码中使用的`GetSystemMetrics` API函数用于向操作系统询问各种可视元素的大小。每次都进行该查询（并不存储结果）是重要的，因为用户可以使用Desktop选项的Appearance标签以及其他Windows设置定制大多数这样的元素。而小按钮在其OnClick事件处理程序中有一个对Close对象方法的调用。即使使用Anchors属性的[akTop, akRight]值改变窗口的大小，按钮也会保持它的位置不变。窗体还具有大小限制，这样用户就不能使之太小，如稍后“窗体强制”一节中所述。

窗体直接输入

讨论完窗体的一些特殊功能后，现在来讨论一个非常重要的主题：在窗体内的用户输入。如果我们决定限制组件的使用，还不如编写复杂的程序，从鼠标与键盘接收输入，并直接在窗体表面进行输出。在本章中只介绍该主题。

键盘输入的管理

通常，窗体不会直接处理键盘输入。如果用户必须键入一些字符，应用程序窗体应该包括一个编辑组件或一个输入组件。如果我们想处理键盘的快捷操作，可以使用与菜单有关的快捷键（使用隐藏的弹出式菜单）。

然而，有些时候我们为了特殊目的可能会使用特殊方式处理键盘输入。在这些情况下，可以使用窗体的KeyPreview属性。然后，即使有一些输入控件，窗体的OnKeyPress事件也可以被任何键盘输入操作激活。最后，键盘输入的数据将到达一个目标组件，除非用户设置该字符值为零才能在窗体内停止它（不是字符0，而是字符设置的值为0，表示为#0）。

因此我建立了用于演示这些概念的范例，即KPreview，该范例有一个不含特殊属性的窗体（甚至不含KeyPreview）、一个带有四个选项的单选按钮组，以及几个编辑框（如图7.5所示）。默认状态下，该程序不会做什么特别的事情，除了单击不同的单选按钮激活Key preview时：

```
procedure TForm1.RadioPreviewClick(Sender: TObject);
begin
  KeyPreview := RadioPreview.ItemIndex <> 0;
end;
```

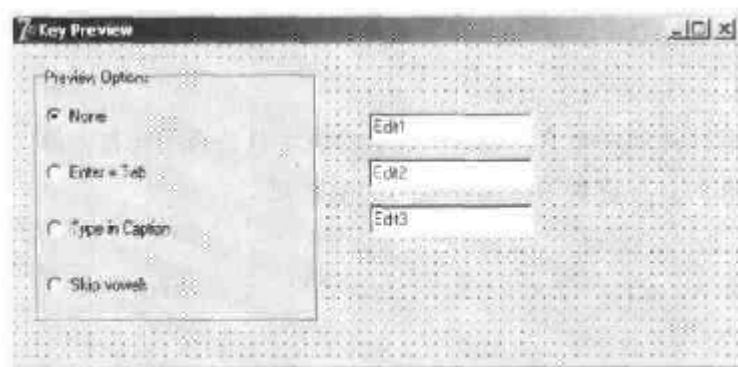


图7.5 设计时的Kpreview程序

现在，开始接收**OnKeyPress**事件，并且按单选按钮组的三个特殊按钮的要求进行相应的操作。这些操作将依赖于单选按钮组件的**ItemIndex**属性值，这就是事件处理程序基于**case**语句的原因：

```
procedure TForm1.FormKeyPress(Sender: TObject; var Key: Char);
begin
  case RadioPreview.ItemIndex of
    ...
```

在第一种情况中，如果**Key**参数的值为#13（对应着Enter键），将终止操作（设置**Key**为零），然后模拟激活Tab键。有很多方法可以实现这一操作，但在此选用了一种非常特殊的方法，向窗体发送**CM_DialogKey**消息，为Tab键（**VK_TAB**）传递代码：

```
1: // Enter = Tab
  if Key = #13 then
  begin
    Key := #0;
    Perform (CM_DialogKey, VK_TAB, 0);
  end;
```

说明：**CM_DialogKey**信息是内部Delphi的非正式文件信息。有几个相应信息可为一些特殊译码建立高级组件。但Borland公司从没有介绍过这些信息。该话题的更多介绍参见第9章中的“组件信息和通告”小节。还应注意该基于译码类型抽取的信息在CLXF不能获得。

要在窗体标题中键入字符，程序只需向当前的**Caption**添加字符即可。有两种特殊情况：当按Backspace键时，字符串的最后一个字符会被删除（通过向**Caption**复制当前**Caption**的所有字符，除了最后一个字符），而且当按Enter键时，程序会通过重新设置单选按钮组控件的**ItemIndex**属性来停止操作。代码如下：

```
2: // type in caption
begin
  if Key = #8 then // backspace: remove last char
    Caption := Copy (Caption, 1, Length (Caption) - 1)
  else if Key = #13 then // enter: stop operation
    RadioPreview.ItemIndex := 0
  else // anything else: add character
```

```
    Caption := Caption + Key;  
    Key := #0;  
end;
```

最后，如果选择最后一个单选项，代码会检查字符是否为元音字母（通过在常量vowel set中测试它的存在）。在这种情况下，字符将被跳过：

```
3: // skip vowels  
if UpCase(Key) in ['A', 'E', 'I', 'O', 'U'] then  
    Key := #0;
```

获得鼠标输入

当用户在窗体（或组件）上单击一个鼠标键时，Windows会向应用程序发送一些消息。Delphi定义了一些事件来响应这些消息，下面是两个基本事件：OnMouseDown是单击一个鼠标键时激发的事件，而OnMouseUp是释放一个鼠标键时激发的事件。还有一个基本的系统消息与鼠标移动有关，该事件是OnMouseMove。尽管这三个消息的意思很容易理解——按下、释放、移动——但问题是怎样将它们与目前为止经常使用的OnClick事件联系起来。

组件经常需要使用OnClick事件，但它对窗体也是适用的。通常的意思是，按下鼠标左键，然后在同一个窗口或组件上释放它。然而，在这两步操作之间，在鼠标左键被按下的同时，光标可以移出窗口或组件的区域。

OnMouseXX¹与OnClick事件的另一个不同之处是后者只与左按钮有关。大多数连接到Windows PC机上的鼠标都有两个按钮，有些是三个按钮。单击左按钮用于选择；右按钮用于局部菜单；中间按钮很少使用。

现在，多数新鼠标设备用一个“按钮轮”替代中间按钮。用户可使用“按钮轮”引用OnMouseWheelUp事件。但是它也能被按下（产生OnMouseWheelDown和OnMouseWheelUp事件）。鼠标轮事件自动变成滚动事件。

不使用鼠标对窗口进行操作

用户应该能在没有鼠标的情况下使用任何Windows应用程序。这不是一种选择，而是Windows编程的规范。当然，用鼠标对应用程序进行操作可能更简单，但这不应该成为强制性的。事实上，有一些用户由于一些原因，可能不能使用鼠标与他们的系统相连，如身处狭小空间的旅游者，工作环境中的工人，以及周围有很多办公用品的银行职员。

还有另一个原因支持键盘操作：使用鼠标是方便的，但它操作起来有时速度较慢。对于一个训练有素的打字员，他将不会使用鼠标拖动文本；而会使用快捷键来复制与粘贴它，作到手不离键盘。

由于这些原因，应该为窗体的组件设置相应的切换顺序，并记住为按钮与菜单项加上热键，这样可以使用键盘进行选择。在菜单命令上使用快捷键等等。

鼠标事件的参数

所有低级的鼠标事件都有相同的参数：通常的Sender参数、Button参数——指示哪个鼠标按钮被单击（mbRight、mbLeft或mbCenter）、Shift参数——当事件发生时，指示哪个相

关的鼠标键被按住（Alt、Ctrl和Shift，加三个鼠标按键）；以及鼠标在当前窗口的客户区坐标中的x和y位置值。

使用该信息，在单击鼠标左键的位置绘制一个小圆就变得很简单了：

```
procedure TForm1.FormMouseDown(Sender: TObject;
  Button: TMouseButton; Shift: TShiftState; X, Y: Integer);
begin
  if Button = mbLeft then
    Canvas.Ellipse (X-10, Y-10, X+10, Y+10);
end;
```

说明：要在窗体上绘图，需要使用一个非常特殊的属性：Canvas。TCanvas对象有两个突出的特性：它保存了一个绘图工具的集合（如Pen、Brush以及Font），而且它拥有一些绘图对象方法，这些方法使用当前的工具。该范例中的这种直接绘图代码是不正确的，因为屏幕上的图像不是连续不变的；将另一个窗口移到当前窗口上，会清除其输出。下一个范例会解释Windows的“store-and-draw（保存与绘制）”方法。

使用鼠标进行拖动与绘图操作

为了演示一些鼠标技术，我们基于一个不带任何组件的窗体建立了一个简单的范例，即VCL版本中的MouseOne和CLX版本中的QMouseOne。该程序的第一个特性是，它可以在窗体的Caption中显示鼠标的当前位置：

```
procedure TMouseForm.FormMouseMove(Sender: TObject; Shift: TShiftState;
  X, Y: Integer);
begin
  // display the position of the mouse in the caption
  Caption := Format ('Mouse in x=%d, y=%d', [X, Y]);
end;
```

我们可以使用程序的这个简单特性更好地理解鼠标的工作原理。完成下面这个测试：运行程序（此简单版本或完整的版本）并在桌面上改变窗口的大小，这样MouseOne或QMouseOne程序的窗体将处于另一个窗口的下面，并处于非激活状态，但可以看到其标题。在窗体上移动鼠标，此时将看到坐标在改变。这意味着，OnMouseMove事件是发往应用程序的，即使其窗口没有处于激活状态也是如此，而且证明了前面曾说过的话：鼠标消息总是发往鼠标下的窗口。惟一的例外是将在同一个范例中讨论鼠标捕获操作。

除了在窗口标题中显示鼠标的位置外，如果用户同时按下鼠标和Shift键，MouseOne/QMouseOne范例还可以通过在窗体上绘制小像素来跟踪鼠标的移动（同样，这种直接绘图代码产生的是非连续不变的输出）：

```
procedure TMouseForm.FormMouseMove(Sender: TObject; Shift: TShiftState;
  X, Y: Integer);
begin
  // display the position of the mouse in the caption
  Caption := Format ('Mouse in x=%d, y=%d', [X, Y]);
  if ssShift in Shift then
    // mark points in yellow
```



```
Canvas.Pixels [X, Y] := clYellow;
end;
```

提示：Kylix 1和Delphi 6中的CLX库的TCanvas类不包括Pixels数组。相反，在设置了画笔的适当颜色后，我们能像在QMouseOne示例中介绍的那样调用DrawPoint对象方法。Kylix 2和Delphi 7再次引入了Pixels数组属性。

该范例真正的特性是直接支持鼠标拖动，与想像的相反，Windows对于拖动没有系统支持，需要通过较低级的事件与操作在VCL中实现（把一个控件拖向另一个控件的例子已在第6章讨论过）。在VCL中，窗体不能发生拖动操作，所以在这种情况下，我们被迫使用低级方法。该范例的目的在于，从拖动操作的起点到终点绘制一个矩形，使用户对自己进行的操作能有视觉上的理解。

拖动操作的思想很简单。程序接收一系列鼠标键单击、鼠标移动与鼠标键释放的消息。当按下鼠标键时，拖动开始，尽管只有当用户移动鼠标（不释放鼠标键）并结束拖动（当鼠标键释放的消息到达）时，真正的操作才会发生。这种方法的问题是不太可靠。通常只有当鼠标移到窗口的客户区上时，它才接收鼠标事件；所以，如果用户按下鼠标键，将鼠标移到另一个窗口上，然后再释放鼠标键，第二个窗口将接收到鼠标释放的消息。

对于该问题有两个解决方案。一种方案（很少使用）是鼠标剪切。使用Windows API函数（ClipCursor），可以迫使鼠标不能离开屏幕上的某个区域。当用户要将它移到该区域之外时，会有一个看不到的障碍阻挡鼠标。第二种方法，也是一种更为常用的方法，就是捕获鼠标操作。当一个窗口捕获鼠标操作时，所有后来的鼠标输入都会发往该窗口。这也是我们用于MouseOne/QMouseOne范例的方法。

该范例的代码基于以下三个对象方法：**FormMouseDown**、**FormMouseMove**与**FormMouseUp**。在窗体上按下鼠标左键将启动拖动操作——通过设置窗体的**fDragging**布尔值来实现（说明拖动处理在其他两个对象方法中也在进行）。该对象方法还使用了一个**TRect**变量，用于跟踪拖动的起始位置与当前位置。下面是有关代码：

```
procedure TMouseForm.FormMouseDown(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  if Button = mbLeft then
  begin
    fDragging := True;
    Mouse.Capture := Handle;
    fRect.Left := X;
    fRect.Top := Y;
    fRect.BottomRight := fRect.TopLeft;
    dragStart := fRect.TopLeft;
    Canvas.DrawFocusRect (fRect);
  end;
end;
```

该对象方法的一个重要操作是调用**SetCapture** API函数，该函数通过设置全局对象**Mouse**的**Capture**属性获得。现在，即使用户将鼠标移到客户区之外，窗体仍可以接收到与鼠标有关的所有消息。我们可以看到，将鼠标移到屏幕左上角，标题中将显示负的坐标值。

提示：全局Mouse对象允许我们获得关于鼠标的全局信息，如它的存在、它的类型和当前位，并设置一些它的全局特征。该全局对象隐藏着几个API功能，使代码更简单更可移植。在VCL的Capture属性有一个Handle类型，而CLX中的类型为TControl（获得鼠标事件组件的对象）。所以，正如大家在QMouseOne示例中所看到的，这部分的代码将变为Mouse.Capture:=self。

当启动拖动特性且用户移动鼠标时，程序会根据实际位置绘制虚线矩形。实际上，程序调用了DrawFocusRect对象方法两次。第一次，调用该对象方法删除当前图像，这是由于两次连续的DrawFocusRect调用会复位初始状态的缘故。在更新矩形的位置之后，程序第二次调用该对象方法：

```
procedure TMouseForm.FormMouseMove(Sender: TObject; Shift: TShiftState;
  X, Y: Integer);
begin
  // display the position of the mouse in the caption
  Caption := Format ('Mouse in x=%d, y=%d', [X, Y]);
  if fDragging then
    begin
      // remove and redraw the dragging rectangle
      Canvas.DrawFocusRect (fRect);
      if X > dragStart.X then
        fRect.Right := X
      else
        fRect.Left := X;
      if Y > dragStart.Y then
        fRect.Bottom := Y
      else
        fRect.Top := Y;
      Canvas.DrawFocusRect (fRect);
    end
  else
    if ssShift in Shift then
      // mark points in yellow
      Canvas.Pixels [X, Y] := clYellow;
    end;
```

在Windows 2000（和其他版本）中，DrawFocusRect函数不会用负值画矩形，所以通过对比当前位置和拖动前的位置（保存在dragStart中）会修改程序代码（如上面看到的）。当释放鼠标键时，程序会通过重置Mouse对象的Capture属性（ReleaseCapture API函数）并将fDragging元素的值设置为False来终止拖动操作：

```
procedure TMouseForm.FormMouseUp(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  if fDragging then
    begin
      Mouse.Capture := 0; // calls ReleaseCapture
      fDragging := False;
```

```
    Invalidate;  
    end;  
end;
```

最后一个调用是`Invalidate`会触发绘图操作，并执行下列代码：

```
procedure TForm1.FormPaint(Sender: TObject);  
begin  
    Canvas.Rectangle (fRect.Left, fRect.Top, fRect.Right, fRect.Bottom);  
end;
```

这使得窗体的输出很稳定，即使将它隐藏在其他窗体之后。图7.6显示了1. 一次的矩形以及进行中的拖动操作。

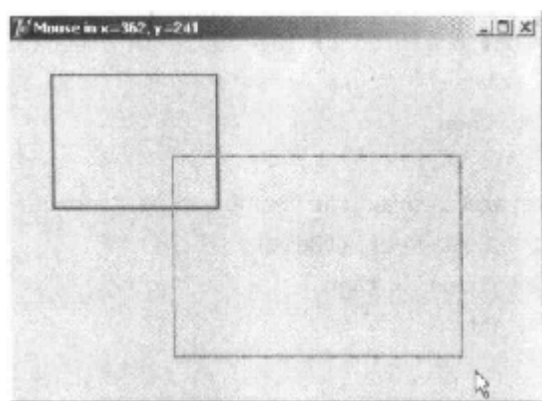


图7.6 在一个拖动操作中，`MouseOne`范例使用了一个虚线来表示最后的一个矩形区域

在窗体中绘图

为什么我们需要处理`OnPaint`事件来生成相应的输出，而且不能在窗体画布上直接绘图呢？这是由Windows的默认行为来决定的。当在窗口上绘图时，Windows不会存储结果图像，当窗口被覆盖时，其内容自然就被清除了。

其原因很简单：为了节省内存。Windows假定在长时间的运行中，使用代码重新绘制屏幕要比使用内存保存窗口的显示状态更节省内存。这是经典的内存与CPU周期协定。一幅 600×800 分辨率的256色位图大概需要480KB的存储空间。增加位图的颜色数量或像素数量，可以轻易地得到一幅16 000 000色、 1280×1024 分辨率的大约4MB的位图。

在通常的情况下，希望自己的应用程序具有一致的输出可以使用的技术有两种。常用的解决方法是存储与数据有关的足够数据，当系统发出一条重绘请求时可以重新生成输出。另一种方法是在生成窗体的输出时将它保存到一幅位图中，这需要在窗体上放置Image组件并在该Image组件的画布上绘图。

第一种技术，绘图，是Windows中处理输出的常用方法，但面向图形并在位图中存储窗体的整幅图像的特殊程序除外。用于实现绘图的方法有一个非常具有说明性的名称：保存并绘制（store and paint）。事实上，当用户单击鼠标键或执行任何其他操作时，都需要存储位置与其他元素；然后，在重绘对象方法中，使用这些信息画出相应的图形。

该方法的思想是让应用程序在任何可能的条件下重绘其整个界面。如果提供一个对象方法来重绘窗体，而在窗体的一部分被遮盖并需要重绘时，该对象方法能被自动调用，那么将可以正确地重建输出。

因为该方法分两步，所以必须连续执行这两步操作，要求系统重画窗口——而不是等着窗口要求重画。可以使用多种对象方法（可以应用于窗体与组件）来启动重新绘制功能：**Invalidate**、**Update**、**Repaint**和**Refresh**。前两个对应着Windows API函数，后两个由Delphi提供：

- **Invalidate**方法通知Windows，窗体的整个表面都应重新绘制。最重要的是**Invalidate**不立即执行重绘操作，Windows只是存储该请求，并在当前过程执行完成后而且没有其他事件有待系统解决时，再对它做出响应（除非调用**Application.ProcessMessages**或**Update**）。Windows有意推迟重绘操作是因为这是一种最耗时的操作。有时，利用这一延迟手段，只是在发生多次变化后才可能绘制窗体，这样可以避免对绘制对象方法（慢）多次连续的调用。
- **Update**对象方法要求Windows更新窗体的内容，立即重新绘制窗体。然而，只有存在无效区域时该操作才会执行。该操作只发生在**Invalidate**对象方法刚被调用之后或作为用户某步操作的结果时。如果不存在无效区域，对**Update**的调用根本不起作用。因此，常可以看到在**Invalidate**调用后紧跟着**Update**调用。这两步操作其实也可以用两个新的Delphi对象方法来实现，**Repaint**与**Refresh**。
- **Repaint**对象方法依次调用**Invalidate**与**Update**，结果是立即激活**OnPaint**事件。该对象方法还有一个稍微不同的版本叫**Refresh**，默认叫做**Repaint**。对这同一个操作具有两个方法这个事实可以追溯到Delphi 1的时代，那时，这两个方法有着微妙的区别。

当程序员需要对窗体执行重新绘制操作时，根据标准的Windows方式，一般都调用**Invalidate**。在需要频繁请求该操作时，这一点特别重要，因为如果Windows花费大量时间更新窗口，重新绘制的请求会积累成一次简单的重新绘制操作。在Windows中，**wm_Paint**消息是一种低优先级的消息。更精确地说，如果重新绘制的请求与其他消息同时等待处理，则其他消息会先于绘制请求处理。

而另一方面，如果多次调用**Repaint**，那么每次调用一定會在Windows处理其他消息前重新绘制屏幕，而且因为重绘操作非常慢，所以这会使用户程序的反应能力较差。然而，当程序员想使自己的应用程序尽可能快地重新绘制窗口时，在这些不常有的情况中，调用**Repaint**是可行的方法。

说明：另一个重要的原因是，在重绘操作期间，为了加快操作速度，Windows只重绘更新区域。因此，如果只让窗口的一部分无效，只有该区会被重绘。为了实现该操作，可以使用**InvalidateRect**与**InvalidateRegion**函数。实际上，该特性有利也有弊。它是一种功能非常强的技术，可以加快速度，减少由于频繁重绘操作造成的闪烁现象。而另一方面，它还会造成不正确的输出。典型的问题是，当仅有一些由用户操作影响的区域被实际改动时，其他区域会保持原样，即使系统执行了应更新它们的源代码。事实上，如果重绘操作在更新区外失效，系统会忽视这一点，把它当做窗口可视区之外发生的操作。

特殊技巧：字母混合、颜色键和动画API

Delphi关于窗体的几个新特征之一就是支持一些新的Windows API来显示窗体（在Windows 2000/XP中，但在Qt/CLX下不可用）。对于窗体，字母混合允许我们在屏幕上用它后面的窗体合并一个几乎不需要的窗体内容，至少在一个商务应用程序中这非常有用。该技巧被应用于位图时比应用于窗体本身更有意思（使用新的AlphaBlend属性和AlphaDIBBlend API函数）。在任意情况下，通过设置窗体的AlphaBlend属性为True，给AlphaBlendValue属性赋一个低于255的值，我们将看到窗体后面是什么。AlphaBlend属性的值越低，窗体越淡。图7.7显示了一个选自ColorKeyHole示例的字母混合示例。

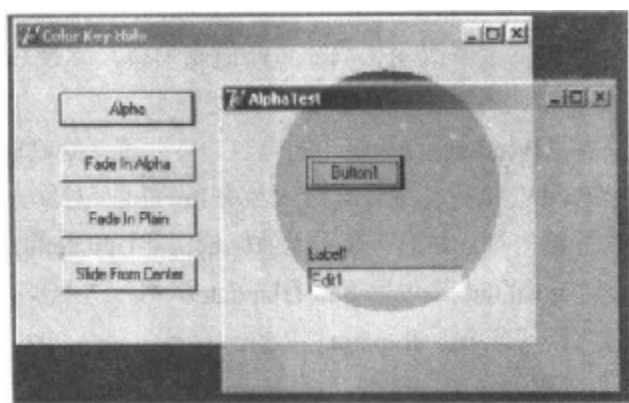


图7.7 ColorKeyHole的输出，显示了新的TransparentColor和AlphaBlend属性以及AnimateWindow API的效果

另一个独特的Delphi特性是新的TransparentColor布尔属性，该属性允许我们显示一个将被背景替代的透明颜色，在窗体上创建一类突破口。TransparentColorValue属性用于显示透明颜色。在图7.7可看到该效果的示例。

最后，也可使用Windows天然技巧，动画显示，而Delphi不直接支持这一技巧（超出提示显示）。例如，为了代替调用窗体的Show对象方法，可编与代码如下：

```
Form3.Hide;  
AnimateWindow (Form3.Handle, 2000, AW_BLEND);  
Form3.Show;
```

注意，为了使窗体行为正常，不得不在最后调用Show方法。通过在循环上改变AlphaBlendValue也能获得一个相似的动画效果。AnimateWindow API也能用于让窗体的显示从中央（用AW_CENTER标记）或从两边的一边（用AW_HOR_POSITIVE、AW_HOR_NEGATIVE、AW_VER_POSITIVE或AW_VER_NEGATIVE）开始，这就是常见的滑动显示。

同样的功能也能用于窗口的控件，以获得淡入的效果代替通常的直接显示效果。我一直怀疑这些动画会引起CPU周期的浪费，但不得不承认，如果它们被正确应用于优秀的程序中，一定能改进用户界面的效果。

位置、大小、滚动和缩放比例

一旦在Delphi上设计了一个窗体，就希望运行程序时窗体能正确显示我们设计的東西。然而，应用程序的用户可能有一个不同的屏幕决定或想调整窗体的大小（如果这是可能的，依靠边框类型），最终影响用户界面。前面已经讨论了一些关于控件的技巧，如队列和锚定。这里想将其作为一个整体再次着重强调一下关于窗体的元素。

除了用户系统的不同外，在该处有许多原因需要改变Delphi默认值。例如，可能运行程序的两个副本并避免所有窗体显示在同一地方。本章的这一部分收集了包括窗体滚动在内的所有其他相关元素。

窗体位置

这有几个属性，能用来设置窗体的位置。Position属性显示Delphi怎样决定窗体的初始位置。poDesigned默认值显示窗体在哪里设计和哪里能使用（Left和Top）以及窗体的大小属性（Width和Height）。

一些其他选择（如poDefault、poDefaultPosOnly和poDefaultSizeOnly）依赖操作系统的特征：使用特殊的标记。使用层叠布局，Windows能定位和/或调整新窗口。以该方法在设计时设置的位置和大小属性将被忽视，但两次运行应用程序将不能获得重叠窗口。当窗体有一个对话框类型时，默认位置被忽略。最后，通过使用poScreenCenter值，窗体以在设计时设置的大小被展示在屏幕中央。这是对话框和二级窗体的通常设置。

影响窗口初始大小与尺寸的另一个属性是它的状态。可以在设计时使用WindowState属性，在启动时显示最大化或最小化窗口。实际上，该属性只有三个值：wsNormal、wsMinimized以及wsMaximized。该属性的意思非常直观，如果设置最小化窗口状态，它会相应地显示在Windows任务栏中。对于应用程序的主窗体，该属性能被自动设置，通过捷径指定对应的应用程序属性。

当然，我们也可以在运行时最大化或最小化窗口。只需将WindowState属性改变为wsMaximized或wsNormal，就会产生所希望的效果。然而，将属性设置为wsMinimized，会生成最小化的窗口，此窗口放置在任务栏上，而不是在其中。这对于主窗体不是异常现象！该问题的解决方法是，调用Application对象的Minimize对象方法。TApplication类中还有一个Restore对象方法，当需要恢复窗体时可以使用该方法，尽管大多数用户通常会使用Restore系统菜单命令。

抓取屏幕（Delphi 7）

在Delphi 7中，窗体有两个新属性：

- 布尔变量ScreenSnap，用于确定当窗体的一个边界靠近屏幕的显示区域时，是否应该抓取。
- 整型变量SnapBuffer，用于确定到边界的距离。虽然不是一个特别的特征，但是它可以很方便地让用户将窗体抓取到屏幕的一边并充分利用整个屏幕表面；这对一个同时拥有多个可视窗体的应用程序来说特别方便。系统会拒绝太高的SnapBuffer属性值（大于屏幕）。

窗体及其客户区的大小

在设计时，有两种方法设置窗体的大小：设置Width与Height属性的值，或拖动它的边框。在运行时，如果窗体有一个可改变的边框，用户可以改变它的大小（产生OnResize事件，可以在该事件中执行定制的活动，让用户界面适应窗体的新大小）。

然而，如果在源代码或联机帮助中查看窗体的属性，可以发现有两个属性引用了窗体的宽度，有两个属性引用了窗体的高度。Height与Width引用了窗体的大小，不包括边框；ClientHeight与ClientWidth引用了窗体内部区的大小，包括边框、标题、滚动条与菜单栏。窗体的客户区是用户用来在窗体上放置组件、输出结果以及接收输入的区域。注意在CLX中，Height和Width表示窗体内部区的大小。

因为人家可能希望有一个可使用的组件区域，所以有时设置窗体的客户区大小而不是设置整个窗体的大小是有意义的。这很好理解，因为当我们设置两个客户区属性时，相应的窗体属性会跟着变化；亦即改变ClientHeight的值时，Height的值会立即改变。

提示：在Windows中，从窗体的非客户区——也就是说，从它的边框建立输出并接收输入也是可能的。当单击边框时，在它的上面绘制并接收输入实在是一个复杂的操作。如果读者对这一专题感兴趣，可以查看Help文件中的相应描述，如wm_NCPaint、wm_NCCalcSize和wm_NCHitTest等消息以及一系列与鼠标输入有关的非客户消息，如wm_NCLButtonDown。该方法的困难之处是，要将自己的代码与默认的Windows行为混合在一起。

窗体强制

当我们为窗体选择可变化的边框时，通常可以按自己的意愿来改变窗体的大小，而且将它最大化到全屏幕。Windows会告诉我们，窗体的大小已使用wm_Size消息改变了，该消息会产生OnResize事件，OnResize事件发生在窗体大小改变之后。在该事件中再一次改变大小（如果用户过分地缩小或增大窗体）是愚蠢的。有一个预防措施可解决该问题。

Delphi为窗体和所有控件提供了一个特殊属性：Constraints属性。只需为Constraints属性的子属性设置合适的最大值与最小值，就能建立一个大小改变不会超出那些限制的窗体。下面是一个范例：

```
object Form1: TForm1
  Constraints.MaxHeight = 300
  Constraints.MaxWidth = 300
  Constraints.MinHeight = 150
  Constraints.MinWidth = 150
end
```

注意，因为设置了Constraints属性，在设计时就会立即显示效果；如果超出了许可区域，则改变窗体的大小。

Delphi还为最大窗口使用了最大化强制，生成了一个糟糕的效果。因此，通常要禁止使用最大化窗口的Maximize按钮。在有些情况下，具有限定大小的最大化窗口是有意义的——这是Delphi主窗口的行为。一旦需要在运行时改变强制，还可以考虑使用两个专门的事件，OnCanResize与OnConstrainedResize。其中第一个事件还可以用于禁止改变给定环境内的窗体或控件大小。

滚动窗体

当建立简单的应用程序时，单个窗体可能会容纳所需的所有组件。当应用程序不断变大时，可能需要压缩组件，加大窗体或添加新窗体。如果压缩组件使用的空间，可能要添加一些功能，用于在运行时改变它们的大小，甚至将窗体分割为不同区域。如果选择改变窗体大小的方法，可能要使用滚动条来让用户在大于屏幕的窗体内移动（至少大于屏幕的可视部分）。

向窗体添加滚动条很简单。实际上，不需要做任何工作——如果在一个大窗体上放置一些组件，然后缩小它，只要不改变AutoScroll属性的值（它的默认设置为True），滚动条就会自动地添加到窗体上。

除了AutoScroll，窗体还有两个属性，HorScrollBar与VertScrollBar，它们可以设置与窗体有关的两个TFormScrollBar对象的一些属性。Visible属性表示是否显示滚动条，Position属性确定滚动块的初始位置，Increment属性确定单击滚动条两端箭头键的效果。然而，最重要的属性还是Range。

滚动条的Range属性确定了窗体一个方向上的虚拟大小，而不是滚动条值的实际范围。起初，这也许会令人迷惑，下面有一个例子来说明Range属性是如何工作的。假设需要一个含有多个组件的窗体，而且窗体需要1000个像素宽。可以使用该值设置窗体的“虚拟范围”，并改变水平滚动条的范围。

滚动条Position属性的范围从0到1000减去当前客户区的大小。实际上，如果窗体客户区有300个像素宽，就可以滚动700个像素来看窗体的最边缘（第1000个像素）。

滚动测试程序

为了说明正讨论的特殊情况，这里建立了一个范例，Scroll1，它的虚拟窗体有1000个像素。要实现该窗体，只需设置水平滚动条的范围为1000：

```
object Form1: TForm1
  HorzScrollBar.Range = 1000
  VertScrollBar.Range = 305
  AutoScroll = False
  OnResize = FormResize
  ...
```

该范例的窗体内包含了一些没有意义的列表框，通过放置最右端的列表也可以得到同样的滚动条范围，因此它的位置（Left）加上它的尺寸（Width）将等于1000。

该范例有趣的部分是有一个工具框窗口来显示窗体及其水平滚动条的状态。这个二级窗体有四个标题框，两个有固定文本，两个是实际输出。除了这些，二级窗体（叫Status）还有一个bsToolWindow边框样式并且是一个最上层的窗口。还应该设置它的Visible属性为True，以使其窗口在启动时自动显示：

```
object Status: TStatus
  BorderIcons = [biSystemMenu]
  BorderStyle = bsToolWindow
  FormStyle = fsStayOnTop
  Visible = True
```



```
object Label1: TLabel...
...
```

该程序的代码不多，它们的目的是，每当窗体大小改变或被滚动时，更新工具框内的值（如图7.8所示）。第一部分非常简单，我们可以处理窗体的OnResize事件，并简单地向两个标题框复制两个值。注意，标题框属于另一个窗体，所以需要使用窗体实例名，Status，作为它们的前缀：

```
procedure TForm1.FormResize(Sender: TObject);
begin
    Status.Label3.Caption := IntToStr(ClientWidth);
    Status.Label4.Caption := IntToStr(HorzScrollBar.Position);
end;
```

如果想在用户滚动窗体的内容时改变输出，则不能使用Delphi的事件处理程序，因为窗体没有OnScroll事件（尽管独立的ScrollBar组件有该事件）。因为Delphi窗体会用一种有效的方式自动处理滚动条，所以可以忽略该事件。相比较而言，在Windows中，滚动条是非常低级的元素，需要大量代码。处理滚动事件只有在特殊情况下才有意义，如要精确地跟踪用户所做的滚动操作时。

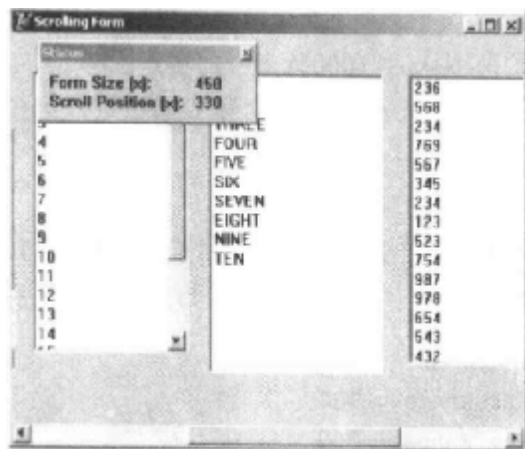


图7.8 Scroll1范例的输出

下面是需要编写的代码。首先，向类中添加一个对象方法声明，并将它与Windows水平滚动消息（wm_HScroll）相连接，然后，编写该过程的代码，该代码几乎与前面见过的FormResize对象方法的代码相同：

```
public
    procedure WMHScroll (var ScrollData: TWMScroll); message wm_HScroll;
procedure TForm1.WMHScroll (var ScrollData: TWMScroll);
begin
    inherited;
    Status.Label3.Caption := IntToStr(ClientWidth);
    Status.Label4.Caption := IntToStr(HorzScrollBar.Position);
end;
```

向inherited添加调用是很重要的，它在基类窗体中激活了与同一消息相关的方法。注意，Windows消息处理程序中的inherited关键字调用了正在覆盖的基类对象方法，它与相应的Windows消息有关（即使过程名不同）。没有该调用，窗体就不会具有默认的滚动功能，也就是说，根本就不会滚动。

说明：因为在CLX中不能处理低级滚动信息，因此看起来没有容易的方法来创建一个与滚动相似的程序。在现实世界应用程序中，这不是十分重要的，因为滚动系统是自动的，并且能通过较低层连到CLX库来完成。

自动滚动

滚动条的Range属性看起来有点陌生，但需要不断使用它时，情况就不同了。如果认真考虑一下，就可以理解这个“虚拟范围”方式的好处了。首先，当窗体客户区足够大到能容纳虚拟大小时，滚动条会自动从窗体上消失；而当缩小窗体时，滚动条又会自动出现。

当窗体的AutoScroll属性被设置为True时，该特性显得额外有意义。在此例中，最右下端控件的绝对位置会自动地复制到窗体两个滚动条的Range属性中去。在Delphi中，自动滚动可以正常工作。在上一个范例中，窗体的虚拟大小将被设置为最后一个列表的右边界。这是由以下属性定义的：

```
object ListBox6: TListBox
  Left = 832
  Width = 145
end
```

所以，窗体的水平虚拟大小将是977（它是上面两个值的和）。该数值会自动地复制到窗体HorzScrollBar属性的Range字段中去，除非手动地改变它使窗体变大（就像在Scroll1范例中做的一样，设置它为1000，以便在最后一个列表与窗体边界之间保留一些空隙）。在对象检验器中可以看到该值，或做以下测试：运行程序，随意设置窗体大小，并将滚动块移到最右端。当添加窗体的大小与滚动块的位置时，总会得到1000，无论大小如何，这总是窗体最右端像素的虚拟坐标。

滚动与窗体坐标

我们已经看到，窗体可以自动滚动它的组件。但如果直接在窗体表面绘制图像，情况又会是什么样呢？会出现一些问题，但它们的解决方法也是现成的。假设想在窗体虚拟表面画一些线，如图7.9所示。因为用户可能没有一个显示器可以在每个坐标轴上显示2000个像素，所以可以建立一个较小的窗体，添加两个滚动条，并设置它们的Range属性，像在Scroll2范例中做的一样。

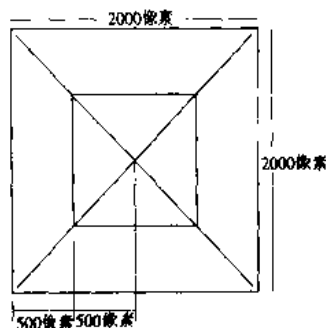


图7.9 绘于窗体的虚拟表面的线

如果只使用窗体的虚拟坐标画线, 图像不会正常地显示。实际上, 在**OnPaint**对象方法中, 需要自己计算虚拟坐标。幸运的是, 这并不麻烦, 因为我们知道客户区左上角的虚拟X1坐标与Y1坐标对应着两个滚动条的当前位置:

```
procedure TForm1.FormPaint(Sender: TObject);  
var  
    X1, Y1: Integer;  
begin  
    X1 := HorzScrollBar.Position;  
    Y1 := VertScrollBar.Position;  
  
    // draw a yellow line  
    Canvas.Pen.Width := 30;  
    Canvas.Pen.Color := clYellow;  
    Canvas.MoveTo (30-X1, 30-Y1);  
    Canvas.LineTo (1970-X1, 1970-Y1);  
  
    // and so on ...
```

还有一种更好的方法, 不必为每步输入操作计算正确的坐标, 可以调用**SetWindowOrgEx** API函数来移动Canvas自己的坐标原点。这样, 绘图代码会直接引用实际坐标并正确地显示:

```
procedure TForm2.FormPaint(Sender: TObject);  
begin  
    SetWindowOrgEx (Canvas.Handle, HorzScrollbar.Position,  
        VertScrollbar.Position, nil);  
  
    // draw a yellow line  
    Canvas.Pen.Width := 30;  
    Canvas.Pen.Color := clYellow;  
    Canvas.MoveTo (30, 30);  
    Canvas.LineTo (1970, 1970);  
  
    // and so on ...
```

读者在本书的源代码中会发现该版本的程序, 试着使用该程序并注释掉**SetWindowOrgEx**调用, 看一看不使用实际坐标的结果: 程序的输出将出现错误, 它将不能滚动, 而相同的图像将总是保留在相同的位置, 对滚动操作无动于衷。此外, 还应注意该程序的**Qt/CLX**版本, 即**QScroll2**, 它没有使用虚拟坐标, 而只是简单地从硬坐标中减去了滚动位置的当前值。

设置窗体比例

当我们建立一个带有多个组件的窗体时, 通常会设置窗体大小为不可改变的, 这样可以避免一些组件超出窗体的可视范围, 如我们已看到的。这可能发生, 因为应用程序的用户有一个显示驱动器, 该驱动器的像素数比程序员所使用的更小。

除了简单地缩小窗体尺寸和滚动内容外, 还可同时缩小每个组件大小。如果用户使用的系统字体, 其每英寸的像素值与开发时使用的字体系统不同时, 该种情况也能自动发生。为了解释这些问题, **Delphi**有一些好的缩放特征, 但它们都不十分直观。

窗体的ScaleBy方法允许程序员设置窗体与其每一个组件的比例。当应用程序在不同的系统字号下运行时，PixelsPerInch与Scaled属性允许Delphi自动改变应用程序窗体的大小。在这两种情况中，要使窗体与它的窗口成比例，还需要设置AutoScroll属性为False。否则，窗体内部会被设置比例，而窗体边框则不会被相应地设置。这两种方法将在下两小节讨论。

说明：窗体比例的计算基于运行时字体高度与设计时字体高度的差别。设置比例可以确保编辑及其他控件有足够的空间使用用户的字体参数选择来显示他们的文本而无需翻页。窗体比例的设置以及稍后介绍的内容，主要是为了使编辑及其他控件变得易读。

手工设置窗体比例

每当程序员要设置一个窗体（包括其组件）的比例时，可以使用ScaleBy方法，该方法有两个整型参数，分别是一个分数的乘数与除数。例如，要使当前窗体的大小被减小到原来大小的三分之一时，所用语句为：

```
ScaleBy (3, 4);
```

下面的形式可以获得同样的效果

```
ScaleBy (75, 100);
```

当设置窗体的比例时，所有的组件也将被设置，但如果高于或低于一定的限制，文本字符串会稍微改变这些组件的比例。注意，如果用户过于缩小窗体，大多数组件将变得不能使用甚至完全消失。这是因为，在Windows中，组件只能以整个像素为单位放置或改变大小，而设置的比例总会涉及分数，所以任何分数部分都会被舍入，这也就造成了上面所述的现象。

在这里建立了一个简单的范例，Scale或QScale，用来介绍根据需要手动设置窗体比例的方法。该应用程序的窗体含有两个按钮、一个标题框、一个编辑框以及UpDown控件。UpDown组件与编辑框相连（使用它的Associate属性）。使用此设置，用户可以在编辑框内键入数值或单击两个小箭头，以固定的数量（用Increment属性来表示）增大或减小编辑框内的数值。为了获取输入的数值，可以使用编辑框的Text属性或UpDown控件的Position属性。当单击Scale按钮时，当前的输入值用于决定窗体的缩放比例：

```
procedure TForm1.ScaleButtonClick(Sender: TObject);
begin
    AmountScaled := UpDown1.Position;
    ScaleBy (AmountScaled, 100);
    UpDown1.Height := Edit1.Height;
    ScaleButton.Enabled := False;
    RestoreButton.Enabled := True;
end;
```

该对象方法在窗体的AmountScaled专用元素中存储当前的输入值并激活Restore按钮——这需要关闭刚按过的按钮。然后，当用户按Restore按钮时，执行相反的比例设置操作。通过在另一个比例操作开始前存储窗体，避免了一个舍入错误的积累。此外还添加了一行代码来设置UpDown组件的Height与编辑框的Height，使它们具有相同的值，这会消除二者之间的微小差别，归功于UpDown控件的比例问题。

说明：如果大家想正确地设置窗体文本的比例，包括组件的标题、列表框中的选项等等，就应该只使用TrueType字体。用系统字体进行缩放不会取得很好的效果。选择字体非常重要，因为很多组件的尺寸要依赖标题的文本高度；而且，如果标题的比例设置不合适，组件可能不会正常工作。因此，在Scale范例中，使用了Arial字体。

相同的缩放技巧也应用在CLX中，如运行QScale示例所见。惟一真正不同之处在于用SpinEdit控件代替了UpDown组件，因为在Qt上前者不可用。

自动设置窗体比例

如果不使用ScaleBy对象方法，可以要求Delphi来做设置比例的工作。当Delphi启动时，它会向系统询问屏幕的分辨率，并将所得值存入Screen对象的PixelsPerInch属性里，这是VCL中一个特殊的全局对象，可以在任何应用程序中使用。

PixelsPerInch听上去好像与屏幕的像素分辨率有关，但遗憾的是，它们并没有关系。如果我们将屏幕分辨率从640×480改变为800×600或1024×768，甚至是1600×1280，就会发现Windows在所有的情况中都会报告同样的PixelsPerInch值，除非改变了系统字体。PixelsPerInch真正的含义是当前装载的系统字体被赋予的屏幕像素分辨率。当用户改变系统字体比例（通常是为了使菜单与其他文本更容易读取）时，会希望所有应用程序都遵循那些设置。没有遵循用户桌面参数选择的应用程序会看起来很别扭，在极端的情况下，对于那些需要大字体与强对比色的弱视力用户来说，甚至不能使用。

最常用的PixelsPerInch值是96（小字体）与120（大字体），也可以使用其他值，Windows系统甚至允许用户任意设置系统字体大小。在设计时，屏幕的PixelsPerInch值（只读属性）会被复制给应用程序的每个窗体。然后Delphi使用PixelsPerInch的值（如果Scaled属性被设置为True）在应用程序启动时改变窗体的大小。

前面曾提到，自动设置比例与执行ScaleBy对象方法设置比例都要通过改变字体的大小来对组件进行操作。事实上，每个控件的大小都以它使用的字体为根据。使用自动比例设置，窗体的PixelsPerInch属性值（设计时值）会与当前的系统值（由Screen对象的相应属性来指定）相比，而将结果用于改变窗体上组件的字体。实际上更精确地讲，字体最终的高度与字体设计时的高度相比，如果它们不匹配，就会调整其大小。

由于Delphi的自动支持，运行在具有不同系统字体大小的系统中的应用程序可以自动调整其比例，而不再需要其他任何代码。应用程序的编辑控件将以用户喜欢的字体大小来显示他们的文本，而窗体也会以相应的大小来容纳这些控件。尽管自动设置比例在一些特殊情况下会出现问题，但如果遵循以下原则，就可以得到好的效果：

- 设置窗体的Scaled属性为True。
- 只使用TrueType字体。
- 在使用的计算机上使用Windows小字体（96像素）来开发窗体。
- 如果大家想设置窗体的比例，而不只是它的控件，应设置AutoScroll属性为False（AutoScroll的默认值为True）。
- 将窗体的位置设置在靠近屏幕左上角或屏幕中央（使用poScreenCenter值），以避免出现超出屏幕的窗体。

建立和关闭窗体

到目前为止，我们一直都忽略了窗体建立的问题。要知道，当建立窗体时，会接收 **OnCreate** 事件，并可以改变或测试初始窗体的属性或字段。负责建立窗体的语句在项目的源文件中：

```
begin
  Application.Initialize;
  Application.CreateForm(TForm1, Form1);
  Application.Run;
end.
```

为了跳过窗体自动建立，也可以改动这段代码，或使用 **Project Options** 对话框中的 **Forms** 页（如图7.10所示）。在该对话框中，可以决定窗体是否应该自动建立。如果禁止自动建立，项目的初始化代码会变成：

```
begin
  Applications.Initialize;
  Application.Run;
end.
```

如果现在运行该程序，什么也不会发生，而且会立即终止，因为没有建立主窗口。调用应用程序的 **CreateForm** 方法的效果是建立一个类的新实例，并将该实例作为第一个参数传递，而且将它赋予作为第二个参数传递的变量。

在屏幕后面也发生了一些事情，当调用 **CreateForm** 时，如果当前没有主窗体，当前窗体会被赋予应用程序的 **MainForm** 属性。因此，该对话框（如图7.10所示）中 **MainForm** 表示的窗体对应着对应用程序 **CreateForm** 对象方法的第一次调用（也就是说，当启动时有多个窗体建立）。

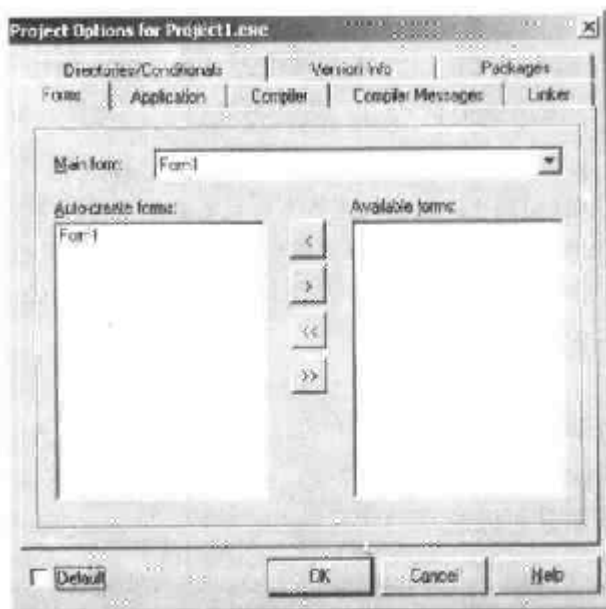


图7.10 Delphi Project Options对话框中的Forms页面

关闭应用程序的处理也相同。关闭主窗口会终止应用程序，而不论其他窗体为何状态。如果大家想从程序代码上执行该操作，只需调用主窗体的Close对象方法，就像在过去的范例会中多次所做的那样。

窗体建立事件

无论是手工还是自动建立窗体，当建立窗体时，都可以截取很多事件。按下列顺序依次触发窗体建立事件：

1. OnCreate指示正在建立窗体。
2. OnShow指示正在显示窗体。除了主窗体，该事件还在将窗体的Visible属性设置为True或调用Show及ShowModal对象方法之后发生。如果窗体被隐藏，然后又显示出来，也会触发该事件。
3. OnActivate指定窗体成为了应用程序中的活动窗体。每当用户从应用程序的一个窗体移到当前窗体时就会触发该事件。
4. 其他事件，包括OnResize与OnPaint，指定操作总发生在启动时，但之后可能会重复很多次。

说明：在Qt中，当窗体被创建时，就像在Windows中一样，OnResize事件不会被触发。为了使代码从Delphi移植到Kylix环境中更方便，CLX模拟了这个事件，虽然通过改变VCL来避免这个奇怪的行为才更有意义些（在CLX的源代码中有一个注释描述了这个现象）。

在上面的列表中可以看到，除了OnCreate事件（只有当建立窗体时才肯定调用它一次）外，每个事件都在窗体初始化中起着特定的作用。

然而，还有一个向窗体添加初始化代码的方法：覆盖构造器。通常，做法如下所示：

```
constructor TForm1.Create(AOwner: TComponent);
begin
    inherited Create (AOwner);
    // extra initialization code
end;
```

在调用基类的Create对象方法之前，窗体的属性还没有装载，而且内部组件还不能使用。为此，标准的方法是首先调用基类构造器，然后再进行定制操作。

说明：直到Delphi 3，才使用了不同的建立顺序，使得OldCreateOrder属性有VCL的兼容性。当把属性设为默认值False时，窗体构造器中的代码在OnCreate事件代码之前执行（被特定的AfterConstruction方法激活）。如果使用旧的建立顺序，构造器的inherited命令要调用OnCreate事件的命令。读者可以使用OldCreateOrder属性的两个值检验CreateOrd示例。

关闭窗体

当我们使用刚才描述的方法或通常的方法（Alt+F4，系统菜单或Close按钮）关闭窗体时，会调用OnCloseQuery事件。在该事件中，将会要求用户确定该操作，特别是在窗体中有未保存的数据时。下面是可以编写的代码结构：

```
procedure TForm1.FormCloseQuery(Sender: TObject; var CanClose: Boolean);
begin
```

```
if MessageDlg ('Are you sure you want to exit?', mtConfirmation,  
    [mbYes, mbNo], 0) = mrNo then  
    CanClose := False;  
end;
```

如果OnCloseQuery仍表示窗体应该关闭，会调用OnClose事件。第三步是调用OnDestroy事件，该事件与OnCreate事件的操作相反，通常被用于解除与窗体有关的对象，释放相应的内存。

说明：更精确地讲，BeforeDestruction方法在调用Destroy解除器之前可以生成OnDestroy事件。也就是说，除非将OldCreateOrder属性设置为True，在这种情况下，Delphi使用一个不同的关闭顺序。

那么，中间的OnClose事件有什么用呢？在该方法中，我们还有一次机会来避免关闭应用程序，或指定不同的“关闭操作”。事实上，该对象方法有一个由引用方式传递的Action参数，可以赋予该参数以下值：

caNone 不允许窗体关闭。该值相当于设置OnCloseQuery事件的CanClose参数为False。

caHide 窗体没有关闭，只是隐藏起来。如果在应用程序中还有其他窗体，该值会很有意义；否则，程序会终止。这是二级窗体的默认设置值，也是在前一个例子中必须处理OnClose事件来实际关闭二级窗体的原因。

caFree 窗体被关闭，并释放它所占用的内存；如果是主窗体，应用程序将终止。这是主窗体的默认行为，而且当动态建立多个窗体时（如果想消除窗口并在窗体关闭时解除相应的Delphi对象）应该使用的行为。

caMinimize 窗体没有关闭，而是最小化。这是MDI子窗体的默认行为。

说明：当用户关闭Windows时，会激活OnCloseQuery事件，而且用户可以使用它来停止关闭过程。在这种情况下，即使OnCloseQuery设置CanClose参数为True，也不会调用OnClose事件。

对话框和其他二级窗体

除了能定制的边框、边框图标和相似的用户界面元素外，当编写程序时，对话框和二级窗体之间没有人的区别。

用于连接用户和对话框的概念是模态窗口，即能引起注意的、并在用户移到主窗口前必须被关闭的窗口。这对于信息框以及通常的对话框是真实的。然而，也有需要使用非模态对话框的情况。

因此，如果想要对话框为模态窗体，单击右键，但我们的描述不简洁。在Delphi（在Windows）中，能拥有非模态对话框和模态窗体。此时要考虑两个不同元素：窗体的边框和它的用户界面决定它看上去是否像一个对话框；使用两种不同的对象方法（Show或ShowModal）来显示二级窗体决定它的行为（模态或非模态）。

向程序添加二级窗体

向应用程序添加二级窗体，只需单击Delphi工具栏上的New Form按钮或使用File菜单下的New Form菜单命令。还有一种方法，可以选择File►New命令，移到Forms或Dialogs页，并选择一种可使用的窗体模板或窗体向导。

如果在一个项目中有两个窗体，可以使用Delphi工具栏的View Form或View Unit按钮，在设计时遍历它们。还可以选择哪一个窗体作为主窗体，以及哪一个窗体应在启动时使用Project Options对话框中的Forms页自动建立，该信息在项目文件的源代码中反映了。

提示：根据Environment Options对话框Preferences页的Auto Create Forms复选框的状态，可以在项目源代码文件中自动建立二级窗体。尽管对于初学者与简单项目来说，自动建立是最简单与最可靠的方法，但是建议大家在重要的开发工作中禁用该复选框。因为当应用程序拥有上百个窗体时，其实不可能都在程序启动时建立它们，只在需要它们时，在恰当的位置建立二级窗体实例，并且使用完后释放它们。

一旦准备好了二级窗体，只需要将它的Visible属性设置为True，程序就会在启动时显示两个窗体。通常，应用程序的二级窗体应该保持隐藏状态，然后通过调用Show方法（或在运行时设置Visible属性）显示它。如果使用Show函数，二级窗体将被显示为非模态类型，所以当二级窗体仍然显示时可以返回主窗体。要关闭二级窗体，可以使用它的系统菜单或单击Close按钮。正如我们看到的，二级窗体默认的关闭操作（见OnClose事件）只是将它隐藏起来，所以当关闭二级窗体时，并没有解除它。它被保存在内存中（同样，也不总是最好的方法），以便再次显示它时可以使用。

在运行时建立二级窗体

除非在程序启动时建立窗体，否则将需要检查窗体是否存在以及是否有必要建立它。最简单的情况是，可在运行时建立同一个窗体的多份拷贝。在MultiWin/QMultiWin范例中，我编写了以下代码来实现该操作：

```
with TForm3.Create (Application) do  
    Show;
```

每当单击按钮时，就会建立窗体的一份新拷贝。注意，这里没有使用Form3全局变量，因为每当建立新窗体对象时赋予该变量新值是没有意义的。然而，重要的是不要在窗体自己的代码或应用程序的其他部分中引用Form3对象。事实上，Form3变量只是一个指向nil的指针，所以应该从程序中删除它，以避免造成混乱。

提示：在窗体的代码中，不能通过使用Delphi为它建立的全局变量明显地引用窗体。例如，假设在TForm3的代码中引用Form3.Caption。如果建立一个相同类型（TForm3类）的二级对象，表达式Form3.Caption将总是引用由Form3变量引用的窗体对象标题，而该变量可能不是执行代码的当前对象。为了避免该问题，只需在窗体的对象方法中引用Caption属性，说明当前窗体对象的标题，并且当需要特别引用当前窗体的对象时使用Self关键字。为了避免在建立窗体的多份拷贝时出现问题，建议从声明窗体单元的接口部分中删除全局窗体对象。该全局变量只有在自动建立窗体时才需要。

当动态建立窗体的多份拷贝时，要记住在关闭每个窗体时，还要通过处理相应的事件来解除窗体对象：

```
procedure TForm3.FormClose(Sender: TObject; var Action: TCloseAction);  
begin  
    Action := caFree;  
end;
```

如果不这样做,会导致大量的内存消耗,因为建立的所有窗体(窗口与Delphi对象)都将保存在内存中,只是看不到罢了。

建立二级窗体的单个实例

现在,在一次只考虑一份复制窗体的程序中,重点讨论窗体的动态建立。当建立模态窗体时,这是非常简单的,因为对话框在关闭时会被解除,使用的代码如下所示:

```
var
    Modal: TForm4;
begin
    Modal := TForm4.Create (Application);
    try
        Modal.ShowModal;
    finally
        Modal.Free;
    end;
```

因为ShowModal调用会引起一个异常,所以为了确保对象会被解除,应该在一个finally块中写入它。通常,该块还包括在显示之前用于初始化对话框的代码,以及提取用户设置值的代码。如果ShowModal函数的结果是mrOK,则最后的值是只读的,将在下一个范例中看到这一点。

当只想显示一份非模态窗体的拷贝时,情况就稍显复杂了。事实上,必须建立窗体——如果它还不能使用,然后显示它:

```
if not Assigned (Form2) then
    Form2 := TForm2.Create (Application);
Form2.Show;
```

使用这段代码,第一次需要的时候就会建立窗体,然后将它保存到内存中,使之在屏幕上可视或隐藏。为了避免不必要的内存浪费与系统资源浪费,可以在关闭二级窗体时解除它。为此,编写如下OnClose事件的处理程序:

```
procedure TForm2.FormClose(Sender: TObject; var Action: TCloseAction);
begin
    Action := caFree;
    // important: set pointer to nil!
    Form2 := nil;
end;
```

注意,在解除窗体之后,Form2全局变量会被设置为nil,但这与前面设定的规则相抵触。这些规则是为带有多个实例的窗体而设定的,但是这里只有一个实例。没有这些代码,关闭窗口会解除其对象,但Form2变量仍将引用原内存地址。这时,如果试图再一次使用以前出现过的btnSingleClick方法显示窗体,则if not Assigned()测试会成功,因为它用来检测Form2变量是否为nil。这样,代码将不会建立新对象,而此时使用Show方法(调用一个不存在的对象)将会导致一个系统内存错误。

做个实验，删除上面代码清单中的最后一行，就会生成该错误。可以看到，解决方法是在解除对象时将Form2对象设置为nil，这样正确编写的代码就可“发现”在使用新窗体之前，必须建立它。MultiWin/QMultiWin范例可以用于测试不同的条件。我们没有对该范例进行任何图示，因为它显示的窗体没有什么内容（除了主窗体和其中的三个按钮外）。

说明：如果在任何给定的时刻都只出现一个窗体实例，那么将窗体变量设置为nil是有意义的——而且程序可以正常运行。如果想建立多个窗体，就必须使用其他技术跟踪它们。还要记住，在这种情况下，不能使用新添的FreeAndNil过程，因为不能在Form2上调用Free。原因是，不能在窗体的事件处理程序执行完成之前解除它。

建立对话框

本章前面曾提过，对话框与其他窗体的区别并不很大。建立对话框（而不是窗体）有一个非常简单的技巧，只需选择窗体的BorderStyle属性为bsDialog即可。仅仅是一个简单的变化，窗体的用户界面就变得像对话框界面了，没有系统图标，没有Minimize或Maximize按钮，也没有右击标题激活的系统菜单。当然，这样一个窗体有典型的对话框粗边框，它的大小是不能改变的。

建立对话框窗体后，可以使用两个常用的显示方法（Show与ShowModal函数）来把它显示为模态窗口或非模态窗口。然而，模态对话框比非模态对话框更常用。这正好与窗体相反：模态窗体通常不应该使用，因为用户不愿意使用它们。

RefList范例的对话框

在第5章中，我们研究了一个RefList/QRefList范例，它使用了ListView控件显示对书籍、杂志、Web站点等的引用。在RefList2版本中（等价CXL的QRefList），我们将向该程序的基础版本添加一个对话框，用在两种不同的情况下：向列表添加新项，以及编辑已有项。

警告：CLX的ListView组件有个问题，一旦激活复选框之后再将其禁用，它们就会消失。这是第5章中QRefList示例的情况。在QRefList2版本中，已经对这个问题添加了代码，以重新指定各工作区项的ImageIndex属性。

在VCL示例中，这个窗体独特的特征仅仅是使用了ComboBoxEx组件，而且主窗体的ListView控件使用的ImageList也添加了该组件。用于选列表项的下拉菜单包括文本描述和相应的图像。

前面曾提到，该对话框用于两种情况。第一种情况发生在用户选择File菜单下的Add Items菜单命令时：

```
procedure TForm1.AddItem1Click(Sender: TObject);
var
  NewItem: TListItem;
begin
  FormItem.Caption := 'New Item';
  FormItem.Clear;
  if FormItem.ShowModal = mrOK then
  begin
```

```

NewItem := ListView1.Items.Add;
NewItem.Caption := FormItem.EditReference.Text;
NewItem.ImageIndex := FormItem.ComboType.ItemIndex;
NewItem.SubItems.Add (FormItem.EditAuthor.Text);
NewItem.SubItems.Add (FormItem.EditCountry.Text);
end;
end;

```

除了设置窗体适当的标题外，该过程还需要初始化对话框，就像输入一个新值一样。然而，如果用户单击OK按钮，程序会向列表添加新项并设置它的所有值。为了清空对话框的编辑框，程序需要调用定制的Clear对象方法，它会重新设置每个编辑框控件的文本：

```

procedure TFormItem.Clear;
var
  I: Integer;
begin
  // clear each edit box
  for I := 0 to ControlCount - 1 do
    if Controls [I] is TEdit then
      TEdit (Controls[I]).Text := '';
  end;

```

编辑已有项需要使用稍微不同的方法。首先，在显示当前值前将它移到对话框中，然后，如果用户单击OK按钮，程序会改动当前列表项，而不是建立一个新项。下面是有关代码：

```

procedure TForm1.ListView1DbClick(Sender: TObject);
begin
  if ListView1.Selected <> nil then
    begin
      // dialog initialization
      FormItem.Caption := 'Edit Item';
      FormItem.EditReference.Text := ListView1.Selected.Caption;
      FormItem.ComboType.ItemIndex := ListView1.Selected.ImageIndex;
      FormItem.EditAuthor.Text := ListView1.Selected.SubItems [0];
      FormItem.EditCountry.Text := ListView1.Selected.SubItems [1];

      // show it
      if FormItem.ShowModal = mrOK then
        begin
          // read the new values
          ListView1.Selected.Caption := FormItem.EditReference.Text;
          ListView1.Selected.ImageIndex := FormItem.ComboType.ItemIndex;
          ListView1.Selected.SubItems [0] := FormItem.EditAuthor.Text;
          ListView1.Selected.SubItems [1] := FormItem.EditCountry.Text;
        end;
      end;
    end;
  end;

```

读者可以在图7.11中看到这段代码的效果。注意，用来读取一个新项与改动项值的代码很相似。通常，大家应避免复制代码这种做法，并将共享的代码语句放到添加在对话框中的一个对象方法里。在本例中，对象方法可接受TListItem对象作为参数，并将正确的值复制到此对象中。



图7.11 使用编辑模式的RefList2范例的对话框。
请注意使用了ComboBoxEx图像组件

说明：当用户单击对话框的OK或Cancel按钮时，内部发生了什么事情呢？通过设置其ModalResult属性，模态对话框将被关闭，而且它会返回该属性的值。可以通过设置按钮的ModalResult属性指定返回值。当用户单击按钮时，其ModalResult值会被复制给窗体，由它关闭窗体并返回作为ShowModal函数结果的值。

非模态对话框

对话框的第二个范例显示了一个较复杂的模态对话框，它使用了标准方式以及一个非模态对话框。DlgApply范例（同样基于CLX的QDlgApply示例）的主窗体上有五个名称标题框，如图7.12所示；通过浏览示例的源代码也可看到这一点。

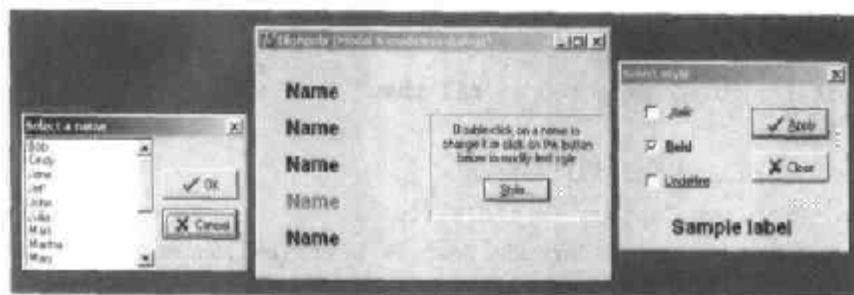


图7.12 DlgApply范例在运行时的三个窗体（一个主窗体和两个对话框窗口）

如果用户单击一个名称，它的颜色会变成红色；如果双击它，程序会显示一个带有一列供从中挑选名称的模态对话框。如果用户单击Style按钮，会显示一个非模态对话框，允许用户改变主窗体标题的字体样式。主窗体的五个标题框与两个方法相关，一个用于响应OnClick事件，另一个用于响应OnDoubleClick事件。前一个方法将用户单击的标题框颜色改变为红色，并重新设置其他标题框颜色为黑色（它们设置Tag属性为1）。注意，所有的标题框都与同一个对象方法相连：

```
procedure TForm1.LabelClick(Sender: TObject);
var
    I: Integer;
begin
```

```

for I := 0 to ComponentCount - 1 do
  if (Components[I] is TLabel) and (Components[I].Tag = 1) then
    TLabel (Components[I]).Font.Color := clBlack;
  // set the color of the clicked label to red
  (Sender as TLabel).Font.Color := clRed;
end;

```

同样，第二个对象方法也对所有标题框通用，它是OnDoubleClick事件的处理程序。LabelDoubleClick对象方法在对话框列表框中选择当前标题框（由Sender参数指定）的Caption，然后显示模态对话框。如果用户单击OK关闭对话框，并且同时某列表项被选中，那么该选择会被复制回标题框的标题：

```

procedure TForm1.LabelDoubleClick(Sender: TObject);
begin
  with ListDial.ListBox1 do
  begin
    // select the current name in the list box
    ItemIndex := Items.IndexOf (Sender as TLabel).Caption);
    // show the modal dialog box, checking the return value
    if (ListDial.ShowModal = mrOk) and (ItemIndex >= 0) then
      // copy the selected item to the label
      (Sender as TLabel).Caption := Items [ItemIndex];
    end;
  end;
end;

```

提示：注意，所有用于定制模态对话框的代码都包含在主窗体的LabelDoubleClick方法中。该对话框的窗体不含添加的代码。

而非模态对话框有很多代码。当单击Style按钮（注意按钮标题后有三个点，这表示当单击它时，程序将打开一个对话框）时，主窗体会调用它的Show方法来显示它。可以在前面的图7.12中看到该对话框的运行情况。

注意两个按钮的名称，Apply与Close，它们在非模态对话框中通常代表OK与Cancel按钮（得到这些按钮最快的方法是为Kind属性选择bkOK或bkCancel值，然后编辑Caption）。有时，可以看到Cancel按钮操作起来像Close按钮，但在非模态对话框中，OK按钮通常根本没有意义。而可能有一个或多个按钮在主窗体上执行特殊操作，如Apply、Change Style、Replace、Delete等等。

如果用户单击这个非模态对话框的某个复选框，则底部样本标题的文本类型会相应地改变。为了实现该操作，可以向设置中添加或从设置中清除特殊标志来标明类型，就像下面的OnClick事件处理程序所做的一样：

```

procedure TStyleDial.ItalicCheckBoxClick(Sender: TObject);
begin
  if ItalicCheckBox.Checked then
    LabelSample.Font.Style := LabelSample.Font.Style + [fsItalic]
  else
    LabelSample.Font.Style := LabelSample.Font.Style - [fsItalic];
end;

```

当用户选择Apply按钮时，程序会向窗体的每个标签复制样本标签的字体样式，而不考虑复选框的值：

```
procedure TStyleDialog.ApplyBtnClick(Sender: TObject);
begin
    Form1.Label1.Font.Style := LabelSample.Font.Style;
    Form1.Label2.Font.Style := LabelSample.Font.Style;
    ...
end;
```

还有一种办法，不用直接引用每个标签，而是调用窗体的FindComponent方法来寻找它，并将标签名作为参数传递，然后将结果转换为TLabel类型。该方法的优点是，可以使用for循环建立不同标签的名称：

```
procedure TStyleDialog.ApplyBtnClick(Sender: TObject);
var
    I: Integer;
begin
    for I := 1 to 5 do
        (Form1.FindComponent('Label' + IntToStr(I)) as TLabel).Font.Style :=
            LabelSample.Font.Style;
    end;
```

提示：ApplyBtnClick方法还可以通过在一个循环内扫描Controls数组编写，这已经在另一个范例中做过了。这里之所以使用FindComponent方法，只是想说明一种不同的技术而已。

代码的第二个版本执行起来慢了一些，因为它需要执行更多的操作，但用户将注意不到这一区别，因为无论如何它执行起来都称得上足够快了。当然，这第二种方法也更具灵活性：如果要添加新标签，只需要改变for循环的上限，为所有标签提供序号即可。

注意，当用户单击Apply按钮时，对话框不会关闭。只有Close按钮具备这样的功能。还要看到，该对话框不需要初始化代码，因为窗体不会被清除，每当对话框显示时，它的组件都会保持它们的状态。然而，在QDlgApply程序对应的CLX版本中，即使它被称为Show方法，该对话框也是模态的。

预定义对话框

除了建立我们自己的对话框外，Delphi还允许使用一些不同种类的默认对话框。其中一些是Windows预定义的，其他一些则是由一个Delphi例程显示的简单对话框。Delphi的组件面板包含了一个对话框组件页。这些对话框——被称做Windows通用对话框——定义在库ComDlg32.DLL中。

Windows通用对话框

在前面章节的一些范例中，曾经使用过这样的对话框，所以读者可能已经熟悉了它们。一般需要在窗体上放置相应组件，设置它的一些属性，运行对话框（使用Execute方法，返回布尔值），并在运行时检索设置过的属性。为了帮助读者使用这些对话框，我们建立了CommDlgTest程序。

我们只是想强调通用对话框的一些主要与不容易引起注意的特性，以便于读者自己研究范例代码的细节：

- **Open Dialog**组件可以通过设置不同的文件扩展名过滤器来定制**Filter**属性，该属性有一个方便的编辑器，可以直接赋予**Text File(*.txt) | *.txt**之类的字符串。另一个方便的特性是，让对话框检测所选文件的扩展名是否与默认的扩展名匹配（通过执行了对话框后检查**Options**属性的**ofExtensionDifferent**标志）。最后，该对话框通过设置其**ofAllowMultiSelect**选项而允许多重选择。在这种情况下，可以通过查看**Files**字符串列表属性获得所选文件的列表。
- **SaveDialog**组件可以按照相同的方法使用，并且具有相同的属性，尽管不能选择多个文件。
- **OpenPictureDialog**与**SavePictureDialog**组件提供了相似的特性，但有一个定制的窗体，可以预览图像。当然，只有在打开或保存图形文件时使用它们才有意义。
- **FontDialog**组件能用于显示并从所有字体类型中进行选择，可选择屏幕和所选打印机上可用的字体或仅仅选择**TrueType**字体。可以显示与特殊效果相关的部分或隐藏它，并通过设置其**Options**属性获得其他不同的版本。还可以为**Apply**按钮的**OnApply**事件提供一个事件处理程序来激活它，并可使用**fdApplyButton**选项。带有**Apply**按钮的**Font**对话框（如图7.13所示）几乎与非模态对话框一样。

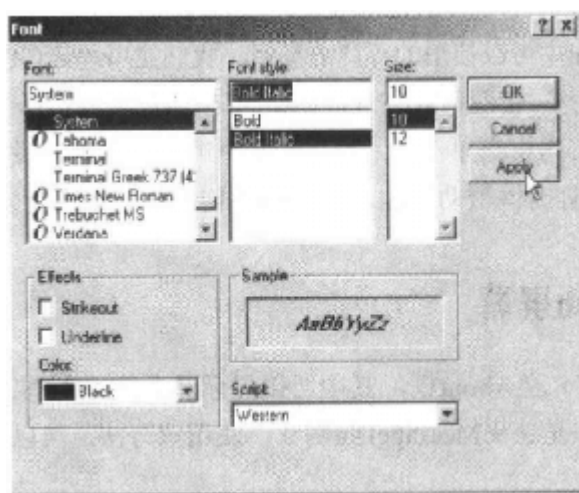


图7.13 带有Apply按钮的字体选择对话框

- **ColorDialog**组件带有不同的选项，用以显示开始时完全打开的对话框或阻止它完全打开。与这些设置相关的是**Options**属性的**cdFullOpen**或**cdPreventFullOpen**值。
- **Find**与**Replace**对话框是真正的非模态对话框，但我们必须自己实现查找与替换功能，就像在**CommDlgTest**范例中所做的一样。通过提供**OnFind**与**OnReplace**事件，将定制代码与两个对话框的按钮相连。

说明：Qt提供了一组相似的预定义对话框，只有选项类经常被限制。笔者已创建了能用于检测这些设置的示例的**QCommDlg**版本。CLX程序只有几个菜单项，因此一些选项在CLX上不可用，并且在源代码上有一些其他微小的变化。

消息框的用法

Delphi消息框与输入框是又一系列的预定义对话框。基本上有六个Delphi过程与函数可用于显示简单对话框：

- **MessageDlg**函数显示一个定制的消息框，该消息框有一个或多个按钮，并通常是位图按钮。**MessageDlgPos**函数与**MessageDlg**函数相似。不同之处是，该消息框在给定位置，而不是在屏幕中央显示（除非使用-1, -1使它在屏幕中央）。
- **ShowMessage**过程显示一个更为简单的消息框，该对话框以应用程序为标题，而且只有一个OK按钮。**ShowMessagePos**过程也一样，但是也要指明该消息框的位置。**ShowMessageFmt**过程是**ShowMessage**的变种，有与**Format**函数相同的参数。它会相应地在**ShowMessage**的调用中调用**Format**函数。
- **Application**对象的**MessageBox**方法允许程序员指定消息与标题，还可以提供不同的按钮与特性。这是对Windows API的**MessageBox**函数简单而直接的封装，该函数将**Application**对象的句柄作为主窗口参数传递。要求该句柄使消息框的行为与模态窗口相像。
- **InputBox**函数要求用户输入字符串。程序员可以提供标题、查询以及默认的字符串。**InputQuery**函数也要求用户输入字符串。它与**InputBox**函数的惟一差异是语法不同。**InputQuery**函数有一个布尔返回值，用来指示用户是否单击了OK或Cancel。

为了演示在Delphi中可以使用的不同消息框，我们使用与处理CommDlg范例相同的方式编写了又一个范例程序。在范例MBParade中，有相当数量的选择（单选钮、复选框、编辑框以及Spin编辑控件）用于在单击显示消息对话框的按钮前进行设置。相似地，QMbParade实例的不足之处是没有提供帮助按钮，这在CLX信息框中不可用。

About框与Splash屏幕

应用程序通常有一个About框，其中可以显示诸如产品版本、版权注意等信息。建立About框最简单的方法是使用**MessageDlg**函数。使用该方法，可以只显示有限数量的文本而不带特殊图形。

所以，建立About框通常的方法是使用一个简单的对话框，如一个使用Delphi默认模板产生的对话框。说简单是因为当设计带有徽标的窗体时，很少需要更多的代码，至多在显示系统信息时可能需要一些代码，如Windows的版本或闲置内存的大小，或一些用户信息，如登录用户的名称等。

建立Splash屏幕

在应用程序中使用的另一个典型技术是，建立一个初始窗口，在主窗体显示前显示。这使得应用程序看起来更具吸引力，因为当装载程序时，不但可以向用户显示一些信息，而且可以产生美观的视觉效果。有时，同样的窗口作为应用程序的About框来显示。对于本范例（其中Splash屏幕特别有用），我们建立了一个在列表框中显示素数的程序。

在程序启动时,运行for循环从1到30 000将素数计算出来。一旦窗体显示就可以看到它们。因为使用了一个很慢的函数计算素数,所以初始化代码运行很长时间,但实际就是需要它的慢速度来显示有关细节。用于显示素数的列表框覆盖窗体的整个客户区并允许多列显示这一现象,如图7.14所示。



图7.14 Splash范例的主窗体,带有Splash屏幕(Splash2版本)

Splash程序有三个版本(加上三个对应的CXL版本)。运行Splash0范例可以看到,该程序的问题是初始操作,它发生在FormCreate方法中,花费了很长时间。当启动程序时,显示主窗体会需要几秒钟的时间。如果用户的计算机非常快或非常慢,可以改变FormCreate对象方法中for循环的上限,使程序运行得快一些或慢一些。

该程序有一个带有图形组件的简单对话框、一个简单的标题以及一个位图按钮,都放在一个面板里,该面板占了About框的整个表面。当用户选择Help菜单下的About菜单项时,会显示该窗体。但真正想要的是在程序启动时显示这个About框。运行Splash1或Splash2范例就可以看到该效果,这两个范例显示了使用两种不同技术建立的splash屏幕。

首先向TAboutBox类添加一个对象方法。该对象方法叫MakeSplash,用于改变窗体的一些属性,使之适合于splash窗体。基本上讲,它的作用是删除窗体的边框与标题,隐藏OK按钮,加宽面板的边框(来代替窗体的边框),然后显示窗体,马上重新绘制它:

```
procedure TAboutBox.MakeSplash;
begin
  BorderStyle := bsNone;
  BitBtn1.Visible := False;
  Panel1.BorderWidth := 3;
  Show;
  Update;
end;
```

在Splash1范例的项目文件中建立窗体后,该对象方法将被调用。在建立其他窗体(在本例中仅有主窗体)前应执行这段代码,然后在运行应用程序前删除Splash屏幕。这些操作都是在一个try/finally块中完成的。下面是Splash2范例项目文件主块的完整源代码:

```

var
    SplashAbout: TAboutBox;

begin
    Application.Initialize;
    // create and show the splash form
    SplashAbout := TAboutBox.Create (Application);
    try
        SplashAbout.MakeSplash;
        // standard code...
        Application.CreateForm(TForm1, Form1);
        // get rid of the splash form
        SplashAbout.Close;
    finally
        SplashAbout.Free;
    end;

    Application.Run;
end.

```

该方式只有在以下情况下才有意义：启动代码较慢，而且建立应用程序的主窗体要花费一定时间，以执行它的启动代码（如在本例中），或打开数据库列表。注意：**Splash**屏幕是建立的第一个窗体，但因为程序没有使用**Application**对象的**CreateForm**方法，所以它没有成为应用程序的主窗体。在这种情况下，关闭**Splash**屏幕就等于终止程序！

还有一种方法是，在屏幕上保持**Splash**窗体稍长一段时间，并使用计时器在一段时间后清除它。我们在**Splash2**范例中实现了这种技术。该范例还使用一种不同的方式来建立**Splash**窗体：不在项目源文件中建立它，而在主窗体**FormCreate**对象方法的最开始建立它：

```

procedure TForm1.FormCreate(Sender: TObject);
var
    I: Integer;
    SplashAbout: TAboutBox;
begin
    // create and show the splash form
    SplashAbout := TAboutBox.Create (Application);
    SplashAbout.MakeSplash;
    // slow code (omitted)...
    // get rid of the splash form, after a while
    SplashAbout.Timer1.Enabled := True;
end;

```

计时器在该对象方法终止前启动，在延迟时间结束后（在本例中是3秒）激活**OnTimer**事件，**Splash**窗体通过关闭或清除自己来处理它，亦即调用**Close**和**Release**。

说明：窗体的**Release**方法与对象的**Free**对象方法相似，只是窗体的解除会延迟，直到所有事件处理程序都完成执行为止。在窗体中使用**Free**对象方法会造成访问错误，因为内部代码（会激活事件处理程序）可能又引用了窗体对象。

还要改正一个地方，主窗体稍后会显示在**Splash**窗体的前方，除非设置它为最顶层窗体。因此，向**Splash2**范例**About**框的**MakeSplash**对象方法添加了一行代码：

```
FormStyle := fsStayOnTop;
```

小结

本章研究了一些重要的窗体属性。现在，大家知道怎样处理窗体的大小和位置、怎样调整它的大小及怎样获得鼠标输入和给它着色。读者了解了许多关于对话框、模态窗体、预定义对话框、**Splash**屏幕的知识和许多其他技巧，包括字母混合的有趣效果。很好地了解窗体的工作状况对于正确使用**Delphi**是至关重要的，特别对于建立复杂应用程序（除非正在建立没有用户界面的服务或网络应用程序）更是如此。

下一章将继续研究**Delphi**应用程序的全局结构，涉及两个全局对象：**Application**和**Screen**。此外，也将介绍**MDI**开发，因为读者将学习一些如可视窗体继承这样的窗体高级特征。最后，还将介绍框架，这是类似于窗体的可视组件容器。

在本章，笔者也对直接绘画和**TCanvas**类的使用做了简短的介绍。关于在**Delphi**窗体中制图的详细论述，参见附录C中介绍的读物“**Delphi**中的图像”所提供的內容。

第二部分 Delphi面向对象的体系结构

第8章 Delphi应用程序的结构

尽管从本书一开始就已建立了Delphi应用程序，但从没有注意由Delphi类库建立的应用程序的结构和体系。例如，没有介绍全局Application对象、用于追踪已创建窗体的技术、系统中消息的流程及其他这样的元素。

在第7章“使用窗体”中，笔者介绍了怎样用多个窗体和对话框创建应用程序。然而，没有介绍这些窗体是怎样一个个联系起来的，怎样共享窗体的相似特征，以及怎样连贯地操作多个类似的窗体。

所有这些内容将是本章介绍的要点，既涉及基本技术，也涉及高级技术，包括可视窗体继承、框架使用、MDI开发以及用于建立复杂层次窗体类的界面的使用。

本章主要包括以下内容：

- Application和Screen全局对象
- Windows消息和多任务处理
- 后台进程和多线程
- 查找应用程序的上一个实例
- MDI应用程序
- 可视窗体继承
- 框架
- 基窗体和界面
- Delphi的内存管理器

Application对象

在多个场合笔者都提到过Application全局对象，本章将集中介绍Delphi应用程序的结构，同时应该更详细地介绍这个全局对象的有关内容和它对应的类。Application是TApplication类的一个全局对象，定义在窗体单元并在控件单元中被创建。TApplication类是一个组件，但在设计时并不能使用它。该类的一些属性能直接在Project Options对话框的Application页面中设置；其他的必须以代码形式赋值。

为了处理该事件，Delphi介绍了一个便利的ApplicationEvent组件。除了允许用户在设计时向处理器赋值外，这个组件的优点在于它允许被用于多个处理器。如果简单地将ApplicationEvent的两个示例放置在两个不同的窗体，则它们都能处理相同的事件，并且两个事件处理器都将被执行。换句话说，多个ApplicationEvent组件能链接处理器。

一些应用广泛的程序事件包括OnActivate、OnDeactivate、OnMinimize和OnRestore，它们允许程序员追踪应用程序的状态。而其他事件则期待应用程序通过控件接受它们，诸如OnActionExecute、OnActionUpdate、OnHelp、OnHint、OnShortCut和OnShowHint。最后是在第2章“Delphi编程语言”中使用的OnException全局异常处理器，用于背景计算的OnIdle事件，以及OnMessage事件——当一个消息被发布到任意窗口或应用程序的窗口控件时，该事件被触发。

虽然它的类直接继承自TComponent，但Application对象有一个与其相关联的窗口。该应用程序窗口将被隐藏，但会出现在任务栏中。这就是为什么Delphi命名窗口为Form1，相应的任务栏图标为Project1的原因。

与Application对象相关的窗口——应用程序窗口——用于将应用程序的所有窗口联系在一起。实际上，一个程序的所有顶层窗体其宿主窗口都有这个隐藏窗口，当应用程序被激活时，这是十分重要的。实际上，当用户程序的窗口在其他窗口下面时，单击用户应用程序中的一个窗口，该程序的所有窗口都会被带到前面。换句话说，隐藏的应用程序窗口可用于连接应用程序的不同窗体（应用程序窗口不是隐藏的，否则这会影响它们的行为，它只不过是高度与宽度设为零，所以就看不见了）。

提示：在Windows中，最大化和最小化操作默认情况下，是与系统声音和一个可视动画效果相关联的。用Delphi建立的应用程序，默认情况下将会产生声音并显示可视效果。

当创建一个全新的空白应用程序时，Delphi会为项目文件生成一些代码，包括下列代码：

```
begin
    Application.Initialize;
    Application.CreateForm(TForm1, Form1);
    Application.Run;
end.
```

就像大家可以在这个标准代码中看到的那样，Application对象能创建窗体，设置第一个窗体为MainForm（Application属性之一），并当该主窗体被破坏时关闭整个应用程序。程序的执行被封装在Run方法中，该方法内嵌了系统循环以处理系统消息。这个循环将不断进行，直到应用程序的主窗口（创建的第一个窗口）被关闭为止。

提示：就像我们在第7章的splash screen范例中看到的那样，主窗体并不一定是所创建的第一个窗体，而是通过调用Application.CreateForm所创建的第一个窗体。

内嵌入Run方法中的Windows消息循环会将系统消息传递到合适的应用程序窗口中。任何Windows应用程序都需要一个消息循环，但并不需要在Delphi中编写它，这是因为Application对象已经提供了一个循环。

除了执行这个主要任务之外，Application对象还管理一些其他有趣的地方：

- Hints（在第5章“可视控件”最后已做过介绍）

- 帮助系统，包括定义帮助浏览器的类型的能力（本书没有详细介绍）
- 应用程序激活、最小化和恢复
- 全局异常处理器，在第2章中的ErrorLog示例中介绍过
- 通用应用程序信息，包括MainForm、可执行文件名称和路径（ExeName）、图标以及当用Alt+Tab键浏览正在运行的应用程序时，显示在窗口任务栏中的标题

提示：为了避免两个标题间的差异，可在设计时改变应用程序的标题。为了防止在运行时主窗体的标题发生改变，可以使用代码Application.Title:=Form1.Caption，将其复制到应用程序的标题中。

在大多数应用程序中，除了设置其标题与图标，以及处理它的一些事件外，用户对应用程序窗口都不会太注意。然而，我们可以执行一些其他的简单操作。在项目源代码中，将ShowMainForm属性设置为False，表示在启动时不显示主窗体。在一个程序中，还可以使用Application对象的MainForm属性访问主窗体。

显示应用程序窗口

为了证明确实存在Application对象的窗口，最好的方法是将它显示出来，如ShowApp范例所示。实际上，并不需要显示它——只需要改变它的大小并设置两个窗口属性，如显示一个标题与一个边框。可以通过在Application对象的Handle属性指定的窗口上使用Windows API函数来执行这些操作：

```
procedure TForm1.Button1Click(Sender: TObject);
var
  OldStyle: Integer;
begin
  // add border and caption to the app window
  OldStyle := GetWindowLong (Application.Handle, gwl_Style);
  SetWindowLong (Application.Handle, gwl_Style,
    OldStyle or ws_ThickFrame or ws_Caption);
  // set the size of the app window
  SetWindowPos (Application.Handle, 0, 0, 0, 200, 100,
    swp_NoMove or swp_NoZOrder);
end;
```

GetWindowLong和SetWindowLong API函数用于访问与窗口有关的系统信息。在本例中，使用gwl_Style参数来读取或写入窗口的类型，包括它的边框、标题、系统菜单和边框图标等等。上面的代码获取了当前类型并向窗体添加（使用一个or语句）了一个标准边框与一个窗体的标题。

尽管没有必要在自己的应用程序中也这样做，但了解应用程序对象有一个窗口和其连接是非常重要的，还应了解Delphi应用程序的默认结构并能够在需要的时候修改它。

激活应用程序与窗体

为了清楚地演示激活窗体与应用程序是怎样工作的，这里编写了一个简单的解释性范例，叫做ActivApp。该范例有两个窗体，每个窗体含有一个Label组件（LabelForm），用于

显示窗体的状态。该程序为此使用了文本与颜色以指明这个状态信息，就像第一个窗体的OnActivate与OnDeactivate事件处理程序所演示的那样：

```
procedure TForm1.FormActivate(Sender: TObject);
begin
    LabelForm.Caption := 'Form2 Active';
    LabelForm.Color := clRed;
end;

procedure TForm1.FormDeactivate(Sender: TObject);
begin
    LabelForm.Caption := 'Form2 Not Active';
    LabelForm.Color := clBtnFace;
end;
```

第二个窗体具有一个相似的标签与相似的代码。

主窗体也显示了整个应用程序的状态。它使用了一个ApplicationEvents组件来处理Application对象的OnActivate与OnDeactivate事件；这两个事件处理程序与前面列出的两个处理程序相似，惟一的区别是它们修改了窗体中一个二级标签的文本与颜色，并且其中一个会发出一声蜂鸣。

如果运行这个程序，将看到该应用程序是否处于激活状态；如果是，它的窗体将处于激活状态。通过查看输出（如图8.1所示）并听一听提示声音，就可以理解每一个激活事件是如何由Delphi触发的。运行该程序并操作一段时间，将有助于理解它的工作原理。稍后，我们将返回与窗体激活有关的其他事件的话题。

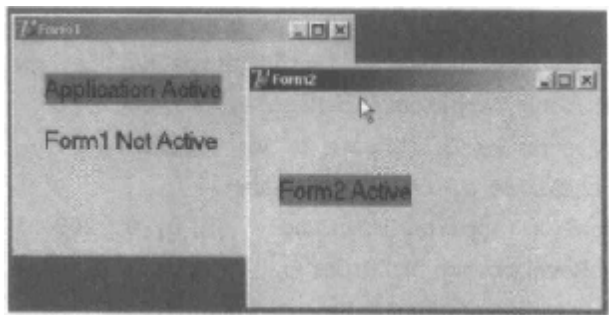


图8.1 范例显示出应用程序及其窗体是否是活动的

使用Screen对象跟踪窗体

我们在前面已经研究了Application对象的一些属性与事件。有关一个应用程序的其他全局信息可通过Screen对象得到，该对象的基类是TScreen。该对象负责处理与系统显示（屏幕大小与屏幕字体）以及一个运行应用程序中窗体的当前设置有关的信息。例如，可以通过编写以下代码显示屏幕大小与字体列表：

```
Label1.Caption := IntToStr (Screen.Width) + 'x' + IntToStr (Screen.Height);
ListBox1.Items := Screen.Fonts;
```

TScreen也会记录在一个多监视器系统中监视器的数量与分辨率。然而,我们现在想关注的是由Screen对象的Forms属性管理的窗体列表、由ActiveForm属性说明的最顶端窗体,以及相关的OnActiveFormChange事件。注意,Screen对象引用的窗体是应用程序的窗体,而不是系统的窗体。

这些特性可以用Screen范例来演示,该范例在一个列表框中列出了当前的窗体。因为每当建立一个新窗体、删除已有窗体或改变程序的活动窗体时,必须更新该列表。为了解释其工作原理,可以单击New按钮,建立二级窗体:

```
procedure TMainForm.NewButtonClick(Sender: TObject);
var
    NewForm: TSecondForm;
begin
    // create a new form, set its caption, and run it
    NewForm := TSecondForm.Create (Self);
    Inc (nForms);
    NewForm.Caption := 'Second ' + IntToStr (nForms);
    NewForm.Show;
end;
```

注意,需要通过使用Project Options对话框的Form页面来关闭二级窗体的自动创建功能。该程序的一个关键部分就是窗体的OnCreate事件处理器,该处理器负责列表的首次填写,然后将一个处理器连接到OnActiveFormChange事件中:

```
procedure TMainForm.FormCreate(Sender: TObject);
begin
    FillFormsList (Self);
    // set the secondary form's counter to 0
    nForms := 0;
    // set an event handler on the screen object
    Screen.OnActiveFormChange := FillFormsList;
end;
```

用于填充Forms列表框的代码位于一个二级进程FillFormList中,该进程也是被用做Screen对象的OnActiveFormChange事件的一个事件处理器而安装的:

```
procedure TMainForm.FillFormsList (Sender: TObject);
var
    I: Integer;
begin
    // skip code in destruction phase
    if Assigned (FormsListBox) then
    begin
        FormsLabel.Caption := 'Forms: ' + IntToStr (Screen.FormCount);
        FormsListBox.Clear;
        // write class name and form title to the list box
        for I := 0 to Screen.FormCount - 1 do
            FormsListBox.Items.Add (Screen.Forms[I].ClassName + ' - ' +
```

```

Screen.Forms[I].Caption);
ActiveLabel.Caption := 'Active Form : ' + Screen.ActiveForm.Caption;
end;
end;

```

警告：当主窗体正在被解除时，不能执行这个代码，这一点是非常重要的。为了测试列表框是否被设置为nil，一种方案就是测试用于csDestroying标记的窗体的ComponentState。另一个方法就是在退出程序之前，删除OnActiveFormChange事件处理器；就是说，处理主窗体的OnClose事件，并将nil值赋予Screen.OnActiveFormChange。

FillFormsList对象方法填充了列表框，并为它上面的两个标题框设置了一个值，这两个标题框用于显示窗体的数目以及活动窗体的名称。当单击New按钮时，程序会建立二级窗体的一个实例，赋予它一个新标题，并显示它。由于为OnActiveFormChange事件安装了处理程序，所以Forms列表框会自动更新。图8.2显示了当建立了一些二级窗口时，该程序的输出。

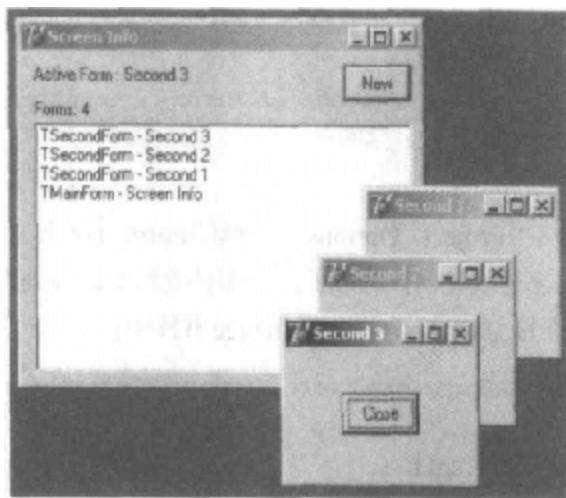


图8.2 范例Screen使用一些二级窗体进行输出

每个二级窗体都有一个Close按钮，可以单击它将其删除。该程序处理了OnClose事件，设置Action参数为caFree，这样当窗体关闭时，它就会被解除。这段代码关闭了窗体，但它没有相应地更新窗口属性的列表。系统会首先将输入焦点移到另一个窗口上，激活更新列表的事件，并在执行完这个操作之后解除旧的窗体。

为了合适地更新窗口列表，笔者的第一个想法是引入延迟，发送一条用户定义的Windows消息。因为发送的消息是排在队列中的，故不会马上被处理。如果在二级窗体生存期的最后可能时刻发送该消息，当其他窗体被解除时，主窗体将收到该消息。这里的问题是，在二级窗体的OnDestroy事件处理程序中发送这个消息。为了实现该想法，需要引用MainForm对象，可通过在该单元的执行部分添加相应的一条uses语句实现该操作。笔者已经发送了一条wm_User消息，它由主窗体的一个特定的message对象方法处理，如下所示：

```

public
  procedure ChildClosed (var Message: TMessage); message wm_User;
  procedure TMainForm.ChildClosed (var Message: TMessage);
begin
  FillFormsList (Self);
end;

```

这里的问题是，如果在关闭二级窗体之前关闭了主窗体，则主窗体仍然会存在，但它的代码却不能再执行了。为了避免另一个系统错误（一个非法访问错误），只有当主窗体没有被关闭时，才能发送消息。但如何才能确定主窗体是否被关闭呢？一个方法是向TMainForm类中添加一个标记，当主窗体关闭时改变它的值，这样就可以从二级窗体的代码中检测该标记。

这是一种好的办法——其实VCL也已经通过以前介绍的ComponentState属性和它的csDestroying标记提供了一些相似的功能。所以可以编写下列代码：

```
procedure TSecondForm.FormDestroy(Sender: TObject);
begin
  if not (csDestroying in MainForm.ComponentState) then
    PostMessage (MainForm.Handle, wm_User, 0, 0);
end;
```

使用这段代码，列表框总会列出应用程序中的所有窗体。

然而，在对该方法经过一定的思考之后，会发现另一种更面向Delphi的解决方法。每当解除一个组件时，就会通过调用在TComponent类中定义的Notification对象方法通知其宿主有关的事件。因为二级窗体由主窗体控制，就像在NewButtonClick对象方法的代码中指定的一样，可以覆盖这种对象方法，并简化代码（参见在Screen2文件夹中的这个版本的代码）：

```
procedure TMainForm.Notification(AComponent: TComponent;
  Operation: TOperation);
begin
  inherited Notification(AComponent, Operation);
  if (Operation = opRemove) and Showing and (AComponent is TForm) then
    FillFormsList;
end;
```

说明：如果二级窗体不属于主窗体，也可以使用FreeNotification对象方法使二级窗体在它们被解除时通知主窗体。FreeNotification会接收组件作为参数来通知当前组件何时被解除。效果是对Notification对象方法的一个调用来自组件而不是宿主组件。FreeNotification通常被组件编写人员用来与不同窗体或数据模块中的组件进行安全连接。

向程序添加的最后一个特性非常简单。当在列表框中单击一个条目时，会使用BringToFront对象方法激活相应的窗体。现在一切都完成了。如果在主窗体没有被激活时单击列表框，则首先会激活主窗体，并重新排列列表框；所以最终可能会选择一个与所期待不同的窗体。如果读者运行该程序，很快就会明白这个意思。程序中的小缺陷是当动态地更新一些信息，并且同时允许用户进行操作时，程序可能会出现故障。

从事件到线程

为了理解Windows应用程序内部是怎样工作的，让我们花一些时间来讨论一下在这种环境下是怎样支持多任务的。我们也需要了解计时器的角色（以及Timer组件）和后台计算，以及Application全局对象的ProcessMessages方法。

简而言之，我们需要更深入地研究Windows的事件驱动结构，以及它的多任务支持。因为这是一本关于Delphi编程的书，故不能详细地讨论这个主题，但是笔者会为那些对Windows API编程不是特别了解的读者提供一个大概的描述。

事件驱动的编程

基于事件驱动编程的基本思想是，用特定的事件确定应用程序的控制流程。一个程序要花费其大部分时间来等待这些事件，并提供代码响应这些事件。例如，当一个用户单击鼠标键时，事件就发生了，一条描述该事件的消息被发送到鼠标光标下的当前窗口中。该窗口响应事件的代码将接收该事件，处理它，并相应地作出响应。当程序完成了对事件的响应之后，它会返回等待或“空闲”状态。

如该解释所描述的一样，事件是串行的，每一个事件的处理都要等到前一个事件处理完毕之后。当应用程序正在执行事件处理代码时（也就是说，当它没有处于等待事件的状态时），该应用程序的其他事件必须在为该程序保留的消息队列中等待（除非应用程序使用多线程处理）。当应用程序响应一条消息之后并返回等待状态时，它将变成程序列表中的最后一位，等待着处理其他消息。在Win32（9x、NT、Me和2000）的每个版本中，等待一个固定时间后，系统将中断当前应用程序并立即将控制权交给列表中的下一个程序。只有当每个应用程序运行一次后，第一个程序将重新开始。这叫做抢先式多任务处理。

因此，想通过使应用程序在一个事件处理器上执行耗时操作，是无法正常地中断程序运行的（因为其他的进程会有其自己的CPU时间片），但是应用程序通常并不能正确地重绘自身的窗口，从而产生较差的效果。如果读者从没经历过这种问题，不妨自己试一下：按住按钮，编写一个时间消耗循环执行代码，并试着在它顶部移动窗体或移动另一个窗口。结果是令人讨厌的。现在试着在循环内添加Application.ProcessMessages调用，将看到操作变得十分缓慢，但窗体被立即刷新。

作为在一个时间消耗循环中使用Application.ProcessMessages的范例（并且没有这种调用），可以引用BackTask范例。这里是使用这种方法的代码（忽略了用于计算给定素数集合之和的技巧）：

```
procedure TForm1.Button2Click(Sender: TObject);
var
  I, Tot: Integer;
begin
  Tot := 0;
  for I := 1 to Max do
  begin
    if IsPrime (I) then
      Tot := Tot + I;
    ProgressBar1.Position := I * 100 div Max;
    Application.ProcessMessages;
  end;
  ShowMessage (IntToStr (Tot));
end;
```

提示：调用ProcessMessages可以用调用HandleMessage函数来替代。但会有两种不同之处：HandleMessage进程每次被调用时最多只有一条消息，而ProcessMessage会将正处理的消息保存在队列中；HandleMessage也会激活空闲时间处理，如行为更新调用。

如果应用程序响应它的事件并等待它返回来处理信息，就没有机会重新获得控制权，直到它收到另一个信息（除非它使用多线程）。这也是使用定时器的一个原因：系统组件将在每个时间间隔结束时向应用程序发送一条消息。使用一个定时器有时是使一个应用程序自动执行的惟一方法，即使这时用户并不在场或者并没有使用该程序（所以并没有处理任何事件）。

最后需要注意的是——当考虑事件时，要记住输入事件（使用鼠标或键盘）在一个Windows应用程序总信息流中只占很小比例。大部分信息是系统的内部信息或不同控件和窗口间交换的信息。诸如单击鼠标键这样熟悉的操作都能导致大量信息，它们中多数为内部的Windows信息。大家可使用Delphi中包含的WinSight工具测试该例子——在WinSight中选择浏览MessageTrace，并为所有的窗口挑选信息；单击Start，然后用鼠标执行一些正常操作。在几秒内将会看到数以百计的信息。

Windows消息发送

在研究一些实际范例之前，需要考虑消息处理的另一个关键要素。Windows有两种不同的方法向一个窗口发送一条消息：

PostMessage API函数 它将一条消息放置到应用程序的消息队列中。只有当应用程序有机会访问它的消息队列时（或者说，当它从系统获得控制权时）才会处理消息，而且只有在前一条消息处理完毕之后。这是一种异步调用，因为程序员不知道何时消息才会被实际接收到。

SendMessage API函数 用于立刻执行消息处理程序代码。SendMessage跳过了应用程序的消息队列，并直接向目标窗口或控件发送消息。这是一种同步调用。该函数甚至有一个返回值，它由消息处理程序代码传回。调用SendMessage的方法与直接调用另一对象方法或程序函数没有什么不同。

这两种发送消息方式之间的区别就像发一封信（它可能很早或很晚到达目的地）与发一份传真（它将马上到达接收者手中）之间的区别一样。尽管在Delphi中很少使用这些底层函数，但如果需要编写这种类型的代码，则这段介绍将有助于确定应该使用哪个函数。

后台处理与多任务

假设需要实现一种计算时间损耗的算法，如果编写该算法是为了响应一个事件，那么在处理该算法期间，应用程序将处于完全停顿的状态。为了使用户能够知道程序正处于处理过程中，可以显示沙漏光标或者一个进程条，但这并不是一种用户友好的解决方案。Win32允许其他程序继续它们的执行，但正在运行的程序将会停止；此时，如果有重新绘制操作请求，甚至可能不能更新自己的用户界面。事实上，当执行该算法时，应用程序不能接收与处理其他任何消息，包括绘制消息。

该问题最简单的解决方法就是调用ProcessMessages和HandleMessage方法，就像前面讨论的那样。然而，该方法的问题是，当启动该算法时，用户可能会再次单击按钮或再次按下

键盘。为了修正这种可能性，可以禁用不想让用户选择的按钮与命令，而且可以显示沙漏光标（从技术上讲，这不会防止鼠标单击事件，但它会提示用户在进行其他任何操作之前，应该耐心等待）。

对于一些低优先级的后台处理来说，还有一种方法是将算法分成小块，依次执行它们，让应用程序在各块执行间隔中响应未决的消息。可以使用一个计时器来让系统在一段时间间隔之后通知你。虽然可以使用计时器实现一些后台计算的窗体，但这也不是一个好的解决方法。更好的方法应该是在Application对象接收OnIdle事件时，执行程序的一个步骤。

调用ProcessMessages函数与使用OnIdle事件之间的区别是，程序通过调用ProcessMessages函数可以获得更多的处理时间。调用ProcessMessages函数是在程序正在执行之时，让系统执行其他操作的一种方法；使用OnIdle事件是当应用程序没有接到用户的任何请求时，让应用程序执行后台任务。

Delphi的多线程

当需要执行后台操作或任何与用户界面没有严格相关的处理时，可以遵循技术上基本正确的方法：在进程中分化一个执行的独立线程。多线程编程可能看起来是一个过时的话题，但是它确实不是那么复杂，即使必须要仔细地考虑它。至少，了解多线程的基础知识是非常必要的，这是因为在套接字或Internet编程中，没有线程便几乎不能做任何事情。

Delphi的RTL库提供了一种TThread类，将允许程序员创建和控制线程。但是不会直接使用TThread类，这是因为它是一种抽象类——一个带有虚拟抽象方法的类。为了使用线程，应总是将TThread子类化，并使用这个基类的特性。

TThread类有一个带有单个参数的构造器（CreateSuspended），用以选择是立刻开始这个线程还是延缓一下。如果线程对象自动启动，或当其重新开始时，它将运行它的Execute方法，直到结束。该类提供了一种保护界面，包括用于线程子类的两种关键方法：

```
procedure Execute; virtual; abstract;  
procedure Synchronize(Method: TThreadMethod);
```

Execute方法作为一个虚拟抽象过程被声明，必须由每个线程类重新定义。它包含了线程的主代码——当使用系统函数时，通常会将该代码放置在一个线程函数中。

Synchronize方法用来避免并发地访问VCL组件。VCL代码在程序的主线程中运行，我们需要同步访问VCL，以避免出现重复输入问题（在前一个调用完成之前再次输入一个函数），并且能同时访问共享资源。Synchronize唯一的参数就是一个方法，该方法没有参数，通常是同一线程类的一个方法。因为不能向该方法传递一个参数，所以常见的做法是将线程数据中的一些值保存在Execute方法中，并在被同步的方法中使用这些值。

说明：Delphi 7包含有Synchronize的两种新版本，允许用户在不从线程对象中调用的情况下将一个方法与主线程同步起来。新的过载Synchronize和StaticSynchronize都是TThread的类方法，并且需要一个线程做为参数。

另一种避免冲突的方法是用操作系统提供的同步技巧。SyncObjs定义了一些新的VCL类，用于那些低层次的同步对象，如事件（带TEvent类和TSingleEvent类）和关键部分（带TCriticalSection类）（同步事件不能与Delphi事件相混淆，因为这两个概念是不相关的）。

线程的一个例子

为了针对线程举例，我们再次引用BackTask范例。这个例子分化出一个二级线程用于计算素数之和。这个线程类拥有典型的Execute方法、一个在public属性（Max）中传递的初始值以及两个内部值（Ftotal和FPosition），用于同步在ShowTotal和UpdateProgress方法中的输出。以下列是用于定制线程对象的完整的类描述：

```

type
  TPrimeAdder = class (TThread)
  private
    FMax, FTotal, FPosition: Integer;
  protected
    procedure Execute; override;
    procedure ShowTotal;
    procedure UpdateProgress;
  public
    property Max: Integer read FMax write FMax;
  end;

```

Execute方法与前面提到的BackTask例子中用于按钮的代码非常相似。惟一的不同在于最终对Synchronize的调用，可以从下面列出的两个片段中看到：

```

procedure TPrimeAdder.Execute;
var
  I, Tot: Integer;
begin
  Tot := 0;
  for I := 1 to FMax do
  begin
    if IsPrime (I) then
      Tot := Tot + I;
    if I mod (FMax div 100) = 0 then
    begin
      FPosition := I * 100 div FMax;
      Synchronize(UpdateProgress);
    end;
  end;
  FTotal := Tot;
  Synchronize(ShowTotal);
end;

procedure TPrimeAdder.ShowTotal;
begin
  ShowMessage ('Thread: ' + IntToStr (FTotal));
end;

procedure TPrimeAdder.UpdateProgress;
begin
  Form1.ProgressBar1.Position := FPosition;
end;

```


当一个按钮被单击时，线程对象即被创建。而当它的Execute方法完成时，将立刻自动将其解除：

```
procedure TForm1.Button3Click(Sender: TObject);
var
  AdderThread: TPrimeAdder;
begin
  AdderThread := TPrimeAdder.Create (True);
  AdderThread.Max := Max;
  AdderThread.FreeOnTerminate := True;
  AdderThread.Resume;
end;
```

这里并未使用属性来设置最大数，而是将这个值做为定制构造器的一个额外参数进行传递的；避免这样做只是为了更加关注使用线程的范例。读者将会在其他章中看到更多的关于线程的例子，特别是在第19章“因特网编程：套接字和Indy组件”中，将专门讨论套接字的使用。

检查应用程序以前的实例

多任务处理的窗体之一可反映出同一应用程序的两个或多个实例的执行情况。任何应用程序通常都会被一个用户以多于一个实例的方式执行，并且它需要检查前面已经在运行的实例，以便能够禁用这种默认的行为，同时只允许最多一个实例。这一章将介绍实现这样一个检查的几种方式，并讨论一些有趣的Windows编程技巧。

搜索主窗口的一个副本

为了查找前一个实例主窗口的一份拷贝，可使用FindWindow API函数并向它传递窗口类的名称（在系统中，用于登记窗体窗口类型的名称，或WNDCLASS）与正在搜索的窗口标题。在Delphi应用程序中，WNDCLASS窗口类的名称与用于窗体类（例如，TForm1）的Object Pascal名称是相同的。FindWindow函数的结果是一个窗口句柄或零（如果没有找到匹配窗口）。

Delphi应用程序的主要代码应该用下述方式编写，这样只有当FindWindow结果为零时才执行它：

```
var
  Hwnd: THandle;
begin
  Hwnd := FindWindow ('TForm1', nil);
  if Hwnd = 0 then
  begin
    Application.Initialize;
    Application.CreateForm(TForm1, Form1);
    Application.Run;
  end
```

```
else  
    SetForegroundWindow (Hwnd)  
end.
```

为了激活应用程序已有实例的窗口，可以使用SetForegroundWindow函数，它也可用于其他过程拥有的窗口。该调用只有当作为参数传递的窗口没有被最小化时才会有效。事实上，当Delphi应用程序的主窗体最小化时，它会被隐藏，因此，此时激活代码无效。

遗憾的是，如果在Delphi IDE中运行一个使用FindWindow调用的程序，则一个带有标题及类的窗口可能已经存在：设计时窗体。这样，程序将一次都不会启动。然而，如果关闭窗口及其相应的源代码文件（只关闭窗口，实际只是隐藏了窗口），或是关闭项目并从Windows Explorer中运行程序，它将可以运行。也要考虑到一个称为Form1的窗体相当有可能并不像想像的那样运行，因为许多Delphi应用程序可能都会有相同名字的窗体。这一点将会在下面版本的代码中被修正。

使用互斥对象

使用互斥信号或称互斥对象是一种完全不同的方法。这是一种典型的Win32方法，通常用于同步线程。下面打算使用互斥对象，以便同步两个不同的应用程序，或更精确地说是同步一个应用程序的两个实例。

应用程序建立了带有给定名称的互斥对象之后，它就可以调用WaitForSingleObject Windows API函数来检测另一个应用程序是否也拥有该对象。如果互斥对象没有主应用程序，那么调用该函数的应用程序将成为其主程序。如果互斥对象已经有了主程序，则应用程序要一直等到规定时间（函数的第二个参数）过后，再返回一个错误代码。

为了实现该技术，可以使用下列项目源代码：

```
var  
    hMutex: THandle;  
begin  
    HMutex := CreateMutex (nil, False, 'OneCopyMutex');  
    if WaitForSingleObject (hMutex, 0) <> wait_TimeOut then  
    begin  
        Application.Initialize;  
        Application.CreateForm(TForm1, Form1);  
        Application.Run;  
    end;  
end.
```

如果运行两次该范例，将可以看到程序会临时创建该应用程序的新拷贝（会在任务栏中出现它的图标），当规定时间过后，再解除它。该方法要比前一种方法稳定，但它缺少了一个特性：怎样激活应用程序的已有实例呢？这仍需要寻找其窗体，但可以使用更好的方法。

查找Windows列表

若想在系统中查找一个特殊的主窗体，可以使用EnumWindows API函数。在Windows中，枚举函数非常特殊，因为它们通常需要另一个函数作为参数。这些枚举函数需要指向一

个函数（经常被称为是一个回调函数）的指针作为参数。该函数被应用于列表中的每个元素（在本例中是主窗口的列表），直到列表末尾或函数返回False值。以下是OneCopy范例中的枚举函数：

```
function EnumWndProc (hwnd: THandle; Param: Cardinal): Bool; stdcall;
var
  ClassName, WinModuleName: string;
  WinInstance: THandle;
begin
  Result := True;
  SetLength (ClassName, 100);
  GetClassName (hwnd, PChar (ClassName), Length (ClassName));
  ClassName := PChar (ClassName);
  if ClassName = TForm1.ClassName then
  begin
    // get the module name of the target window
    SetLength (WinModuleName, 200);
    WinInstance := GetWindowLong (hwnd, GWL_HINSTANCE);
    GetModuleFileName (WinInstance,
      PChar (WinModuleName), Length (WinModuleName));
    WinModuleName := PChar(WinModuleName); // adjust length
    // compare module names
    if WinModuleName = ModuleName then
    begin
      FoundWnd := Hwnd;
      Result := False; // stop enumeration
    end;
  end;
end;
```

该函数（为系统的每个非子窗口调用）要检查每个窗口类的名称——通过查找TForm1类的名称。当发现一个窗口的类名称中有该字符串时，将使用GetModuleFilename读取拥有匹配窗体的应用程序的可执行文件名称。如果模块名称也匹配，就可以确定，已经找到了同一程序的前一个实例。以下是调用枚举函数的代码：

```
var
  FoundWnd: THandle;
  ModuleName: string;
begin
  if WaitForSingleObject (hMutex, 0) <> wait_TimeOut then
  ...
  else
  begin
    // get the current module name
    SetLength (ModuleName, 200);
```

```

GetModuleFileName (HInstance, PChar (ModuleName), Length (ModuleName));
ModuleName := PChar (ModuleName); // adjust length
// find window of previous instance
EnumWindows (@EnumWndProc, 0);

```

处理用户定义的Windows消息

前面曾提到, 如果程序的主窗体最小化, **SetForegroundWindow**调用将不会执行。现在该解决这个问题了。我们可以要求另一个应用程序的窗体, 在本例中, 是同一个程序的前一个实例, 通过向窗体发送用户定义的Windows消息来恢复主窗体。然后检测该窗体是否最小化, 并向老窗口发送一条用户定义的新消息。下面是**OneCopy**程序中的代码, 是上一节中最后一段代码的延续:

```

if FoundWnd <> 0 then
begin
    // show the window, eventually
    if not IsWindowVisible (FoundWnd) then
        PostMessage (FoundWnd, wm_App, 0, 0);
    SetForegroundWindow (FoundWnd);
end;

```

PostMessage API函数向拥有目的窗口的应用程序的消息队列发送一条消息。在窗体代码中, 可以添加一个用于处理这条信息的特殊函数:

```

public
    procedure WMAp (var msg: TMessage); message wm_App;

    procedure TForm1.WMAp (var msg: TMessage);
    begin
        Application.Restore;
    end;

```

说明: 该程序使用了wm_App消息, 而不是wm_User消息, 所以不能保证其他的应用程序或系统将不会发送这条消息。这就是为什么Microsoft引入wm_App用于限制应用程序解释消息的原因。

建立MDI应用程序

对于一个应用程序的结构来说, MDI (Multiple Document Interface) 是一种常见的方法。一个MDI应用程序由多个窗体组成, 这些窗体显示在一个主窗体内。如果使用Windows Notepad, 只能打开一个文本文档, 因为Notepad不是MDI应用程序。但如果使用字处理器, 就可以打开多个不同的文档, 且每个文档都有自己的子窗口, 因为字处理器是一种MDI应用程序。所有这些文档的窗口通常都由一个框架或应用程序的窗口控制。

说明: Microsoft正逐渐地从Windows 3时代倡导的MDI模型中分离出来。Office的新版本更趋向于为每个文档使用一个特殊的主窗口: 经典的SDI (单个文档界面) 方法。不过在某些情况下, MDI并没有被淘汰, 而且很有用处, 如Opera和Mozilla浏览器所展示的那样。

Windows中的MDI：技术概述

尽管MDI结构有些复杂，但它也给程序员自动地提供了一些方便。例如，Windows是在MDI应用程序的一个下拉式菜单中处理子窗口列表的，而有一些特定的Delphi对象方法可以激活相应的MDI功能来平铺或级联子窗口。下面是Windows中一个MDI应用程序的技术结构：

- 使用应用程序主窗口作为一个框架或一个容器。
- 有一个叫做MDI客户的特殊窗口，它覆盖了框架窗口的整个客户区。该MDI客户窗口是一个Windows预定义控件，就像编辑框或列表框一样。MDI客户窗口不具有窗口界面的典型元素，如标题或边框，但它是可视的。事实上，可以在Windows的DisplayProperties对话框的Appearance页中改变MDI工作区的标准系统颜色（调用“Application Background”）。
- 有一些子窗口，类型可以相同也可以不同。这些子窗口没有直接放置在框架窗口上，但每个都被定义为MDI客户窗口的子类，反过来它们也应该算做是框架窗口的子类。

Delphi中的框架与子窗口

Delphi使得MDI应用程序的开发变得容易了，甚至不需要使用Delphi中的MDI应用程序模板（见File>New>Other对话框的Application页面）。只需要建立至少两个窗体——将一个窗体的FormStyle属性设置为fsMDIForm，另一个窗体的相同属性设置为fsMDIChild——几乎就足够了。只需在一个简单程序中设置这两个属性，并运行该程序，就会看到两个以典型MDI形式嵌套在一起的窗体。

然而，子窗体通常不是在启动时建立的，这样，就需要提供一种方法来建立一个或多个子窗口了。通过使用New菜单选项添加一个菜单并编写下列代码就可以实现：

```
var
    ChildForm: TChildForm;
begin
    ChildForm := TChildForm.Create (Application);
    ChildForm.Show;
```

另一个重要特性是添加一个“Window”下拉式菜单，并使用它作为窗体WindowMenu属性的值。该下拉式菜单将自动列出所有可使用的子窗口（当然，可以为该菜单选择其他名称，但“Window”是标准的）。

为了使该程序能够正常工作，可以在子窗体建立时为它的标题添加一个序号：

```
procedure TMainForm.New1Click(Sender: TObject);
var
    ChildForm: TChildForm;
begin
    WindowMenu := Window1;
    Inc (Counter);
    ChildForm := TChildForm.Create (Self);
    ChildForm.Caption := ChildForm.Caption + ' ' + IntToStr (Counter);
```

```
ChildForm.Show;  
end;
```

读者还可以打开多个子窗口，最小化或最大化它们，关闭它们，并使用Window下拉式菜单在它们之间进行切换。现在，假设要关闭一些子窗口，以整理程序的客户区。单击一些子窗口的Close框，它们会最小化！这是怎么回事呢？记着，当关闭一个窗口时，通常只是隐藏它。在Delphi中关闭的窗口仍然存在，尽管看不到。在子窗口的情况中，只是隐藏它们使之不能工作，因为MDI Window菜单以及窗口列表仍然列出现有子窗口，即使它们处于隐藏状态。因此，当用户试图关闭MDI子窗口时，Delphi只是最小化了它们。要解决这个问题，就应当在关闭子窗口时删除它们，将OnClose事件的Action引用参数设置为caFree。

建立一个完整的Window菜单

第一项工作是为范例定义一个较好的菜单结构。典型的Window下拉式菜单至少应有两个选项，相应的标题分别是Cascade、Tile以及ArrangeIcons。要处理菜单命令，可以使用一些TForm预定义的对象方法，该对象方法只用于MDI结构窗体中：

Cascade方法 Cascade方法用于打开MDI子窗口，而且窗口一个叠着一个。图标化的子窗口也要排列（见下面的ArrangeIcons过程）。

Tile方法 Tile方法用于平铺打开的MDI子窗口；子窗体被排列开来，互不叠置。默认行为是横向平铺，如果有多个子窗口，它们将被排列为几列。使用TileMode属性，可以改变这个默认设置（或者是tbHorizontal，或者是tbVertical）。

ArrangeIcons过程 ArrangeIcons过程用于排列所有图标化的子窗口。打开的窗体不会被移动。

还有一种更好的选择，就是调用这些对象方法，在窗体中放置一个ActionList组件，并向它添加一系列预定义的MDI行为。相关的类有TWindowArrange、TWindowCascade、TWindowClose、TWindowTileHorizontal、TWindowTileVertical与TWindowMinimizeAll。相连的菜单项将执行相应的操作，如果没有可用的子窗体，则禁用这些命令。MdiDemo范例（下一个介绍的范例）介绍了MDI行为的用法以及其他特性。

在Delphi中，还有其他一些有意思的对象方法和属性与MDI相关：

ActiveMDIChild属性 ActiveMDIChild是包含活动子窗口的MDI框架窗体的运行时、只读属性。用户可以选择一个新的子窗口来改变该值，或在程序中使用Next与Previous过程来改变它，这两个过程可用于激活当前窗口之前或之后的子窗口。

ClientHandle属性 ClientHandle属性保存着MDI客户窗口的Windows句柄，该窗口覆盖了主窗体的整个客户区。

MDIChildren属性 MDIChildren属性是一个子窗口数组。可以使用它与MDIChildCount属性在所有子窗口之间进行循环，这个属性在寻找一个特殊子窗口或对每个子窗口进行操作时显得非常有用。

注意，子窗口的内部顺序与激活顺序是相反的，这意味着，最后一个选择的子窗口是当前激活的窗口（在内部列表中排在第一位），倒数第二个选择的子窗口在内部列表中排在第二位，第一个选择的子窗口是内部列表中的最后一个。该顺序确定了窗口在屏幕上的排列方式。列表中的第一个窗口排列在所有其他窗口的前面，同时最后一个窗口排列在所有其他窗

口的后面，并很可能被遮住。读者可以想像一个从屏幕指向自己的坐标轴（z轴），当前激活的窗口具有最高的z坐标值，所以它覆盖在其他窗口的上面。因此，Windows排序模式也叫做z-order。

说明：从Delphi 7开始，Window菜单可以和ActionManager以及ActionMainMenuBar控件一起用来处理菜单。这种控件有一个特殊的属性，WindowMenu，用户必须设置它来指定将要列出MDI子窗口的菜单。

MdiDemo范例

本章建立的第一个范例演示了简单MDI应用程序的大部分特性。MdiDemo实际上是一个完整的MDI文本编辑器，因为每个子窗口中都有一个Memo组件，所以可以打开与保存文本文件。子窗体有一个Modified属性，用于说明备注组件的文本是否有改动（它在备注的OnChang事件的处理程序中被设置为True）。Modified在Save和Load方法中被设为False，同时当窗体被关闭（将提示保存文件）时也将选中Modified标记。

前面曾提到过，该范例的主窗体基于ActionList组件。这一动作可以通过一些菜单项与工具栏来实现，如图8.3所示。读者可以在范例的源代码中看到ActionList的细节。下面重点讨论定制动作的代码。

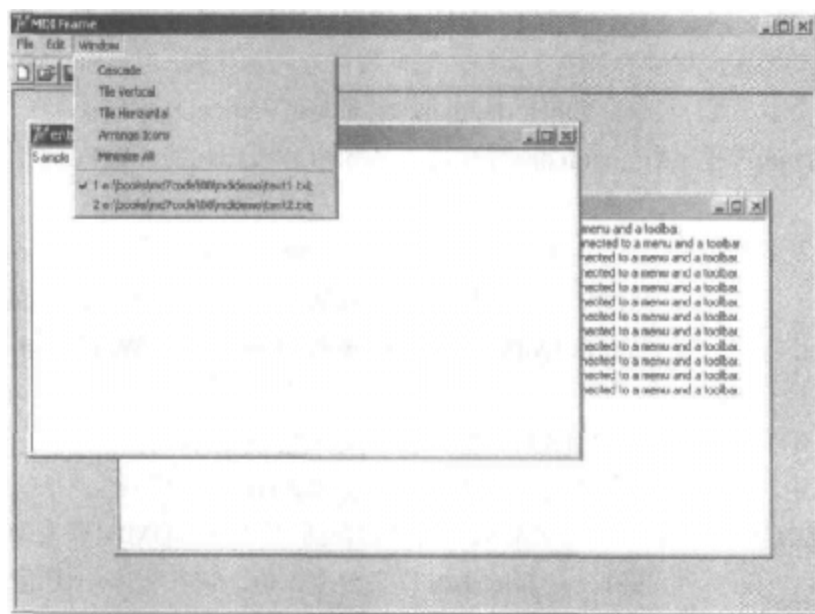


图8.3 MdiDemo程序使用一系列预定义的Delphi动作连接菜单和工具栏

一个最简单的动作是ActionFont对象，该对象有一个OnExecute处理器（使用一个FontDialog组件）和一个OnUpdate处理器（当没有子窗体时关闭该动作，并因此将菜单条目和工具栏按钮关联起来）：

```
procedure TMainForm.ActionFontExecute(Sender: TObject);
begin
    if FontDialog1.Execute then
        (ActiveMDIChild as TChildForm).Memo1.Font := FontDialog1.Font;
end;
```

```

procedure TMainForm.ActionFontUpdate(Sender: TObject);
begin
    ActionFont.Enabled := MDIChildCount > 0;
end;

```

名为New的动作创建了一个子窗体，并设置了一个默认的文件名。Open动作在装载文件之前调用了ActionNewExecute方法：

```

procedure TMainForm.ActionNewExecute(Sender: TObject);
var
    ChildForm: TChildForm;
begin
    Inc (Counter);
    ChildForm := TChildForm.Create (Self);
    ChildForm.Caption :=
        LowerCase (ExtractFilePath (Application.Exename)) + 'text' +
        IntToStr (Counter) + '.txt';
    ChildForm.Show;
end;

procedure TMainForm.ActionOpenExecute(Sender: TObject);
begin
    if OpenDialog1.Execute then
        begin
            ActionNewExecute (Self);
            (ActiveMDIChild as TChildForm).Load (OpenDialog1.FileName);
        end;
end;

```

装载的文件被窗体的Load方法所执行。同样地，子窗体的Save方法被Save和Save As动作所使用。注意，Save动作的OnUpdate处理器，只有当用户改变了备忘录的文本时才会激活：

```

procedure TMainForm.ActionSaveAsExecute(Sender: TObject);
begin
    // suggest the current file name
    SaveDialog1.FileName := ActiveMDIChild.Caption;
    if SaveDialog1.Execute then
        begin
            // modify the file name and save
            ActiveMDIChild.Caption := SaveDialog1.FileName;
            (ActiveMDIChild as TChildForm).Save;
        end;
end;

procedure TMainForm.ActionSaveUpdate(Sender: TObject);
begin
    ActionSave.Enabled := (MDIChildCount > 0) and
        (ActiveMDIChild as TChildForm).Modified;
end;

```



```
procedure TMainForm.ActionSaveExecute(Sender: TObject);  
begin  
    (ActiveMDIChild as TChildForm).Save;  
end;
```

带有不同子窗口的MDI应用程序

在复杂的MDI应用程序中，一个常用的方式是包含不同类型的子窗口（也就是说，基于不同的子窗体）。下面将建立一个新范例，称为**MdiMulti**，强调一下使用该方式可能遇到的一些问题。对于该范例，需要建立两个不同类型的子窗口：第一种类型的子窗口会在鼠标单击的最终位置画一个圆，而第二种类型的子窗口中有一个跳动的正方形。我们向主窗体添加的另一个特性是，定制的背景可以通过在窗体中绘制一幅平铺的图像来实现。

子窗体与合并菜单

第一种类型的子窗体可以在用户单击鼠标键的位置显示一个圆。图8.4显示了**MdiMulti**程序输出的一个例子。该程序还包括一个**Circle**菜单，它允许用户改变圆表面的颜色及其边界的颜色与尺寸。这里有意思的是为子窗体编程序，我们不需要考虑其他窗体或框架窗口，只编写当前窗体的代码就足够了。惟一特殊的需要是两个窗体的菜单。

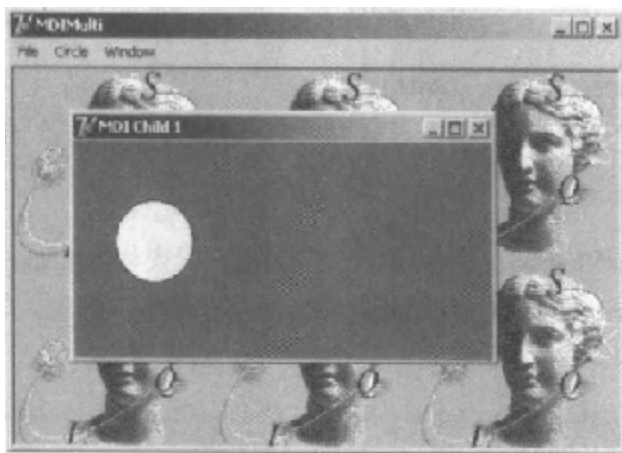


图8.4 MdiMulti范例的输出——用子窗口显示圆

如果为子窗体准备一个主菜单，当子窗体被激活时，它就会代替框架窗口的主菜单。实际上，MDI子窗口不能拥有自己的菜单。但子窗口不能拥有任何菜单的事实不应该令人不安，因为这只是MDI应用程序的标准行为。可以使用框架窗口的菜单栏来显示子窗体的菜单。甚至有更好的方法，亦即将框架窗口的菜单与子窗体的菜单合并。例如，在本程序中，子窗体的菜单放置在框架窗口的**File**与**Window**下拉式菜单之间。可以使用下面的**GroupIndex**值实现该合并：

- **File**下拉式菜单，主窗体：1
- **Circle**下拉式菜单，子窗体：2
- **Window**下拉式菜单，主窗体：3

对菜单组的索引使用这些设置，框架窗口的菜单栏中将出现两个或三个下拉式菜单。启动时，菜单栏有两个菜单。只要建立了子窗口，就会有两个菜单，并且当最后一个子窗口关闭（消除）时，**Circle**下拉式菜单将消失。通过运行程序来测试该行为时还需要花费一些时间。

第二种类型的子窗体可以显示一个移动的图形。正方形，即一个**Shape**组件，它会在固定时间间隔（使用**Timer**组件）上在窗体客户区周围移动，并在窗体边缘改变方向弹回。该过程由一个复杂的算法确定，本书没有对此进行过多的讨论。范例的重点是向读者演示，当一个MDI框架有多个不同类型的子窗体时，菜单如何合并（读者可以通过研究源代码看到其工作原理。）

主窗体

现在，让我们将两个子窗口集成到一个MDI应用程序中去。**File**下拉式菜单有两个独立的**New**菜单项，用于为每种类型建立一个子窗口。该代码使用了一个子窗口计数器。还有一种方法，就是对于两种类型的子窗口使用两种不同的计数器。**Window**菜单又一次使用了预定义MDI动作。

只要这种类型的窗体显示在屏幕上，它的菜单栏就会自动与主菜单栏合并。当在两种类型的子菜单之间进行选择时，菜单栏会相应地改变。一旦所有子窗口都被关闭，便会恢复主窗体原来的菜单栏。通过使用相应的菜单组索引，可以让Delphi来自动完成每一件工作，就像在图8.5显示的两个窗口中看到的一样。

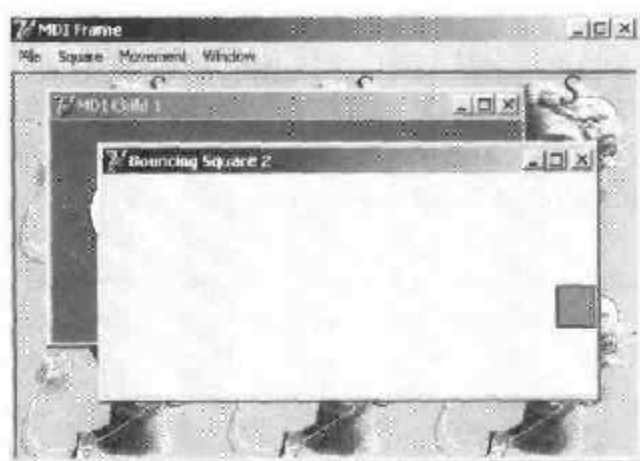


图8.5 Md Multi应用程序可自动变化，以反映当前所选的子窗口（可对比图8.4中的菜单栏）

在主窗体中还添加了一些菜单项，用于关闭每个子窗口和显示一些关于子窗口的统计信息。在这个过程中，与**Count**命令有关的对象方法会扫描**MDIChildren**数组，以计算每种子窗口的数量（使用**RTTI**操作符**is**）：

```
for I := 0 to MDIChildCount - 1 do
  if MDIChildren[I] is TBounceChildForm then
    Inc (NBounce)
  else
    Inc (NCircle);
```

子类化MDI客户窗口

范例程序还包含对一幅平铺的背景图像的支持。该位图来自一个Image组件，并且应绘制在wm_EraseBkgnd Windows消息的处理程序中的窗体上。问题是，不能简单地将代码与主窗体连接，因为表面覆盖着一个独立的窗口（MdiClient窗口）。

对于这个窗口，并没有相应的Delphi窗体，所以怎样处理它的消息呢？必须借助于一种低级的Windows编程技术，叫做子类化（除了名字以外，该技术与OOP继承没有什么关系）。其基本思想是，使用一个新过程替换窗口过程（它接受了窗口的所有消息）。这可以通过调用SetWindowLong API函数并提供过程的内存地址（函数指针）来实现。

说明：一个窗口过程是一个接受窗口所有消息的函数。每个窗口都必须有一个窗口过程，并且只能有一个。甚至Delphi窗体也有一个窗口过程；尽管它隐含在系统中，但它会调用用户可以使用的WndProc虚拟函数。然而，VCL有一种预定义的消息处理机制，在一些预处理之后会用到窗体的消息处理对象方法。使用所有这些支持，则只有当处理非Delphi窗口时才需要显式地处理窗口过程，就像本节的范例一样。

除非有理由改变系统窗口的该默认功能，否则只能存储原过程，并调用它获得默认的处理。引用两个过程（老过程与新过程）的两个函数指针存储在窗体的两个本地元素中：

```
private
    OldWinProc, NewWinProc: Pointer;
    procedure NewWinProcedure (var Msg: TMessage);
```

该窗体还有一个对象方法，可将它用做一个新的窗口过程，其中含有用于绘制窗口背景的实际代码。因为这是一个对象方法，而不是普通的窗口过程，所以程序必须调用MakeObjectInstance对象方法向该对象方法添加一个前缀，让系统将它用做函数。所有这些描述总结起来只需要两条复杂的语句：

```
procedure TMainForm.FormCreate(Sender: TObject);
begin
    NewWinProc := MakeObjectInstance (NewWinProcedure);
    OldWinProc := Pointer (SetWindowLong (ClientHandle, gw1_WndProc, Cardinal
        (NewWinProc)));
    OutCanvas := TCanvas.Create;
end;
```

所安装的窗口过程会调用默认的过程。然后，如果消息是wm_EraseBkgnd，而且图像不是空的，就可以使用临时canvas的Draw对象方法将它绘制到屏幕上（可以很多次）。程序启动时会建立该canvas对象（见上面的代码），并通过消息与作为wParam参数传递的句柄相连。使用该方法，不必为每个绘制操作需要的背景建立一个新TCanvas对象，这样如果频繁操作的话会节省一些时间。下面是有关代码，其输出如图8.5所示：

```
procedure TMainForm.NewWinProcedure (var Msg: TMessage);
var
    BmpWidth, BmpHeight: Integer;
    I, J: Integer;
begin
```

```
// default processing first
Msg.Result := CallWindowProc (OldWinProc, ClientHandle, Msg.Msg, Msg.WParam,
    Msg.LParam);

// handle background repaint
if Msg.Msg = WM_EraseBkgnd then
begin
    BmpWidth := MainForm.Imagel.Width;
    BmpHeight := MainForm.Imagel.Height;
    if (BmpWidth <> 0) and (BmpHeight <> 0) then
    begin
        OutCanvas.Handle := Msg.WParam;
        for I := 0 to MainForm.ClientWidth div BmpWidth do
            for J := 0 to MainForm.ClientHeight div BmpHeight do
                OutCanvas.Draw (I * BmpWidth, J * BmpHeight,
                    MainForm.Imagel.Picture.Graphic);
            end;
        end;
    end;
```

可视窗体继承

当需要建立两个或更多相似的、可能有不同事件处理程序的窗体时，就可以使用动态技术，在运行时隐藏或创建新组件，改变事件处理程序，并使用if或case语句。此外，也可应用面向对象技术，这要感谢可视窗体继承。简要地说，程序员可以从一个已有窗体简单地继承一个窗体，而不是基于TForm创建一个窗体，并添加新组件或改变已有窗体的属性。但可视窗体继承真正的优点是什么呢？

总起来说，要根据建立的应用程序的类型来定。如果应用程序有多个窗体，而且其中一些互相非常相似或含有相同的元素，那么就可以在基窗体中放置通用组件与通用事件处理程序，然后向子窗体添加特殊的功能与组件。例如，准备一个带有工具栏、徽标的标准父窗体，用于默认地设置大小与关闭的代码，以及一些Windows消息处理程序，然后用它作为每个应用程序窗体的父类。

还可以利用可视窗体继承为不同客户定制应用程序，而不需要复制任何源代码或窗体定义代码——只需要从标准窗体中继承用于一个客户的具体版本即可。需要记住，可视继承的主要优点是，改变父窗体后会自动更新所有的子窗体。这是继承在面向对象编程语言中一个著名的优点。但也存在着副作用：多态。可以在基窗体中添加一个虚拟对象方法，并在一个子类化的窗体中覆盖它，然后引用两个窗体，并为它们每一个调用该对象方法。

说明：Delphi新添了一个类似于可视窗体继承的特性，叫做框架（Frames）。在这两种情况中，都可以在设计时对一个窗体/框架的两个版本进行操作。然而，在可视窗体继承中，定义了两个不同的类（父类与派生类），而在框架中，只对一个类与一个实例进行操作。框架的详细内容将在本章稍后部分中进行介绍。

从基窗体继承

如果对什么是继承有一个清晰的概念，那么要理解可视窗体继承的规则就非常简单了。基本上，一个子类窗体含有与父类窗体相同的组件以及一些新组件。不能删除基类的一个组件，尽管（如果它是可视控件）可以使它不可见。重要的是，可以轻松地改变继承组件的属性。

注意，如果在继承窗体内改变某组件的一个属性，则父窗体内相同属性的任何改动将不会产生任何影响。改变组件的其他属性也将影响继承版本。使用对象检验器中的**Revert to Inherited**本地菜单命令可以使两个属性值重新同步。为两个属性设置相同的值并重新编译代码也可以达到同样的目的。在改动了多个属性之后，可以应用组件的本地菜单的**Revert to Inherited**命令使它们与基版本重新同步。

除了继承组件外，新窗体还继承了基窗体的全部对象方法，包括事件处理程序。也可以在继承窗体内添加新处理程序，或覆盖已有处理程序。

为了解释可视窗体继承是怎样工作的，这里建立了一个非常简单的范例，称为VFI。要想建立这个范例，首先要启动一个新项目，并向其主窗体添加四个按钮。然后在New Items对话框中选择**File>New>Other**中命令，并打开带有项目名称的页面（如图8.6所示）。

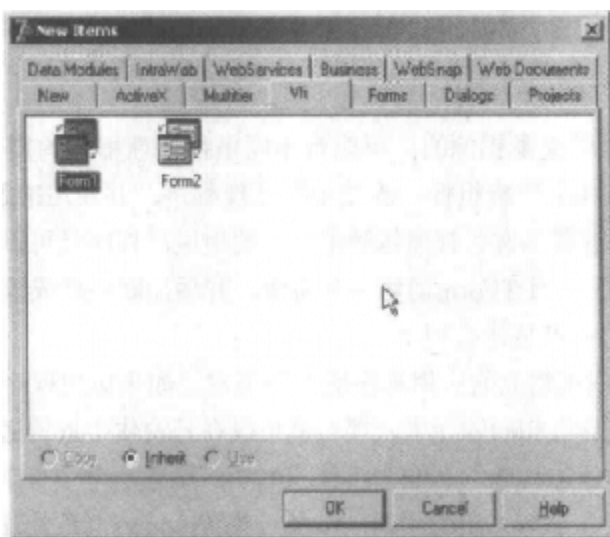


图8.6 New Items对话框允许用户创建继承窗体

在New Items对话框中，用户可以选择想要继承的窗体。新窗体也会有相同的四个按钮。下面是新窗体的原始文本描述：

```
inherited Form2: TForm2
  Caption = 'Form2'
end
```

下面是其初始类描述，在这里可以看到，基类并不是通常的TForm，而是基类窗体：

```
type
  TForm2 = class(TForm1)
  private
```

```

    { Private declarations }
public
    { Public declarations }
end;

```

注意在文本描述中**inherited**关键字的出现；也要注意带有一些组件的窗体，尽管它们是在基类窗体中被定义的。如果移动了窗体并添加了按钮的标题，文本描述将会相应地改变：

```

inherited Form2: TForm2
    Left = 313
    Top = 202
    Caption = 'Form2'
    inherited Button2: TButton
        Caption = 'Beep...'
    end
end

```

只有带有不同值的属性（通过从继承窗体的文本描述中删除这些属性，可以重新把它们复位为基窗体的值，就像前面提到的一样）会被列出来，这里实际改变了大部分按钮的标题，如图8.7所示。

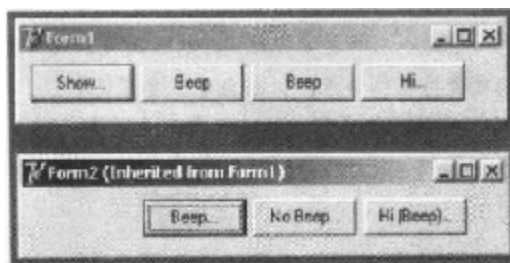


图8.7 VFI范例运行时的两个窗体

第一个窗体的每一个按钮都有一个带有简单代码的OnClick处理程序。第一个按钮通过调用其Show对象方法来显示继承的窗体；第二个以及第三个按钮调用Beep过程；最后一个按钮用于显示一条简单的消息。

在继承的窗体中，我们要做的第一件事情是，清除Show按钮，因为已经显示了二级窗体。然而，不能从继承窗体内删除组件。另一个解决方法是设置它的Visible属性为False——此时按钮将保留在原处，但不可见（如图8.7所示）。其他三个按钮将是可见的，但带有不同的处理程序。如果在继承窗体内选择某个按钮的OnClick事件（双击该按钮），会得到一个与默认对象方法稍有不同空对象方法。因为它包含了**inherited**关键字。该关键字代表了对基窗体中相应事件处理程序的一个调用。注意，该关键字总是由DelPhi添加，甚至处理程序在父类中没有定义（这是有道理的，因为它可能会在后面定义）或组件没有在父类中出现（这不是什么好主意）也是这样。执行父窗体的代码并执行其他一些操作非常简单：

```

procedure TForm2.Button2Click(Sender: TObject);
begin
    inherited;
    ShowMessage ('Hi');
end;

```

这不是惟一的选择。还有一种方法，就是编写一个崭新的事件处理程序而不执行基类的代码，就像为VFI范例的第三个按钮所做的那样；为了完成它，可简单地删除关键字 `inherited`。

还有另一种选择，包括在执行一些定制的代码后调用基类的对象方法，亦即当条件满足时调用它，或调用基类的一个不同事件的处理程序，如第四个按钮：

```
procedure TForm2.Button4Click(Sender: TObject);
begin
    inherited Button3Click (Sender);
    inherited;
end;
```

人们可能并不经常需要从一个不同的处理器中继承，但必须知道可以这样做。当然，可以将基窗体的每个对象方法看做是子窗体的对象方法，并自由地调用它们。该范例允许读者研究可视窗体继承的一些特性；但要了解其真正的功能，还需要研究实际应用中更复杂的范例。下面来介绍可视窗体的多态技术。

说明：可视窗体继承和集合不能很好地在一起工作：我们不能在一个继承窗体中扩展一个组件的集合属性。这种限制可以防止一系列组件的实际使用，如Toolbars或ListViews。当然，我们可以在父窗体或继承窗体中使用那些组件，但是不能扩展它们所包含的元素，这是因为它们被保存在一个集合中。这个问题的一个解决方案就是，为了避免在设计时分配这些集合，要使用一种运行时技术。我们仍然使用窗体继承，但将会丢失其可视部分。如果试图使用Action Manager组件，将会发现我们甚至不能从拥有它的一个窗体中继承。Borland禁用了这个特性，因为这将会给用户带来太多的麻烦。

多态的窗体

如果向窗体添加一个事件处理程序，然后在一个继承窗体内改变它，那么使用基类的一个通用变量来引用两个对象方法是行不通的，因为事件处理程序默认情况下使用静态绑定技术。

不太明白？这里有一个例子，这是专门为有经验的Delphi程序员设计的。假设想在同一个程序中建立一个位图浏览窗体与一个文本浏览窗体，且两个窗体有相似的元素，相似的工具栏、相似的菜单、OpenDialog组件以及不同的组件来浏览实际数据，那么就可以建立一个包含通用元素的基类窗体，并从这个基类窗体中继承这两个窗体。从图8.8中可以看到设计时的两个窗体。

主窗体包含一个拥有几个按钮的工具栏组（使用可视窗体继承实现真实的工具栏时可能会出现一些问题）、一个菜单和一个打开的对话组件。两个继承窗体只有一些很小的差别，但它们使用了一个新组件：或者是图像浏览器（TImage）或者是文本浏览器（TMemo）。它们也改变了OpenDialog组件的设置，用以引用不同的文件类型。

主窗体包括了一些通用代码。Close按钮和File►Close命令均调用窗体的Close对象方法。Help►About命令可以显示一个简单的消息框。基窗体的Load按钮含有一个ShowMessage调用显示一个错误消息。File►Load命令会调用另一个方法：

```
procedure TViewerForm.Load1Click(Sender: TObject);
begin
```

```
LoadFile;
end;
```

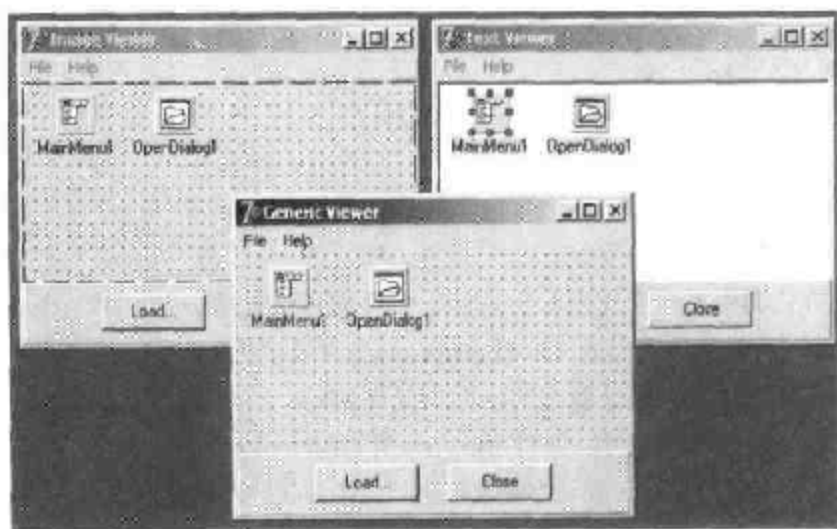


图8.8 PoliForm范例在设计时拥有的一个基类窗体和两个继承窗体

该对象方法是在TViewerForm类中作为一个虚拟抽象方法定义的（因此父窗体类实际上是一个抽象类）。因为这是一个抽象对象方法，所以需要在继承窗体内重新定义（并覆盖它）。这个LoadFile对象方法的代码只使用OpenDialog1组件，它请求用户选择一个输入文件，并将它装入到图像组件中：

```
procedure TImageViewerForm.LoadFile;
begin
  if OpenDialog1.Execute then
    Image1.Picture.LoadFromFile (OpenDialog1.FileName);
end;
```

其他继承类的代码与此相似，即将文本装入memo组件。项目中还有一个窗体，一个带有两个按钮的主窗体，用于在每个查看器窗体中重新装载文件。主窗体是启动时惟一由项目建立的窗体。不会建立通用的查看器窗体：它只是一个通用基类，包含两个子类的通用代码及组件。两个子类的窗体在主窗体的OnCreate事件处理程序中建立：

```
procedure TMainForm.FormCreate(Sender: TObject);
var
  I: Integer;
begin
  FormList [1] := TTextViewerForm.Create (Application);
  FormList [2] := TImageViewerForm.Create (Application);
  for I := 1 to 2 do
    FormList[I].Show;
  end;
```

FormList是通用TViewerForm对象的一个多态数组，在TMainForm类中被声明。注意，为了在类中做此声明，需要在主窗体界面部分的uses子句中添加Viewer单元（不是具体的窗

体)。窗体的数组用于当单击某个按钮时,在每个浏览窗体中装入新文件。两个按钮的OnClick事件处理程序使用了不同的方法:

```
// ReloadButton1Click
for I := 1 to 2 do
  FormList [I].ButtonLoadClick (Self);

// ReloadButton2Click
for I := 1 to 2 do
  FormList [I].LoadFile;
```

第二个按钮只调用了--个虚拟对象方法,并且它可以正常运行。第一个按钮调用了--个事件处理程序,并将总是访问通用的TFormView类(显示其ButtonLoadClick对象方法的错误消息)。这是因为对象方法是静态的,而不是虚拟的。

为了使该方法正常运行,可以将TFormView类的ButtonLoadClick对象方法声明为虚拟方法,并在每个继承窗体类中将它声明为覆盖方法,就像对其他任何虚拟对象方法所做的一样:

```
type
  TViewerForm = class(TForm)
    procedure ButtonLoadClick(Sender: TObject); virtual;
  public
    procedure LoadFile; virtual; abstract;
  end;

type
  TImageViewerForm = class(TViewerForm)
    procedure ButtonLoadClick(Sender: TObject); override;
  public
    procedure LoadFile; override;
  end;
```

这种小技巧确实可行,尽管在Delphi文档中从没有提到过它。这种用于虚拟事件处理程序的功能就是可视窗体多态性的实际含义。换句话说,可以将虚拟对象方法分配到事件属性上,而该事件属性依据在运行时可获得的示例解释对象方法。

理解框架

第1章“Delphi 7及其IDE”中简要介绍了框架(frames)。我们知道,程序员可以建立一个新框架,放入一些组件,为组件编写一些事件处理程序,然后将框架添加到一个窗体中。换句话说,框架与窗体相似,但它只定义了窗口的一部分,而不是整个窗口。当然,创建框架特性并不只是为此目的,框架的真正意义在于,可以在设计时建立一个框架的多个实例,并同时改动类与实例。这使得框架成为在设计时建立定制复合控件的有效工具,即与某种可视组件构造工具相似。

根据可视窗体继承这一理论,程序员可以在设计时对一个基窗体及其派生窗体进行操作,对基窗体所做的任何改变都会传递给派生窗体(除非这覆盖了一些属性或事件)。使用框架也是在对类进行操作,但不同之处在于,还可以在设计时定制所建类的一个或多个实例。

当使用一个窗体时，不能在设计时修改用于这个窗体的一个具体实例的TForm1类的属性，而使用框架则可以。

一旦认识到在设计时正对类及其一个或多个实例进行操作，就完全可以理解框架了。实际上，当想要在一个应用程序的多个窗体中使用同一组组件时，便可以使用框架。事实上，在这种情况下，可以在设计时定制每个实例。组件模板不是已经完成这些任务了吗？是的，但组件模板是建立在复制与粘贴一些组件及其代码的思想上的，无法改变它的原始定义并了解所有使用的效果。而使用框架完全可以做到这一点（并使用一种与可视窗体继承不同的方式），对原始版本（类）的改动会反映在复制（实例）中。

现在用一个范例Frames2来讨论框架的一些原理。该程序有一个框架，其中包括了一个列表框、一个编辑框和三个按钮，以及一些对组件进行操作的简单代码。该框架还有一个角尺用于排列其客户区，因为框架没有边框。当然，框架也对应着相应的类，该类看上去类似于窗体类：

```
type
  TFrameList = class(TFrame)
    ListBox: TListBox;
    Edit: TEdit;
    btnAdd: TButton;
    btnRemove: TButton;
    btnClear: TButton;
    Bevel: TBevel;
    procedure btnAddClick(Sender: TObject);
    procedure btnRemoveClick(Sender: TObject);
    procedure btnClearClick(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;
```

不同之处在于，可以将框架添加到窗体中。在该范例中使用了框架的两个实例（如图8.9所示）并稍微改动了它们的状态，框架的第一个实例含有已分类的列表框项。当改变框架中一个组件的属性时，主窗体的DFM文件将列出区别，就像它处理可视窗体继承一样：

```
object FormFrames: TFormFrames
  Caption = 'Frames2'
  inline FrameList1: TFrameList
    Left = 8
    Top = 8
    inherited ListBox: TListBox
      Sorted = True
    end
  end
  inline FrameList2: TFrameList
    Left = 232
```

```

    Top = 8
    inherited btnClear: TButton
        OnClick = FrameList2btnClearClick
    end
end
end
end

```

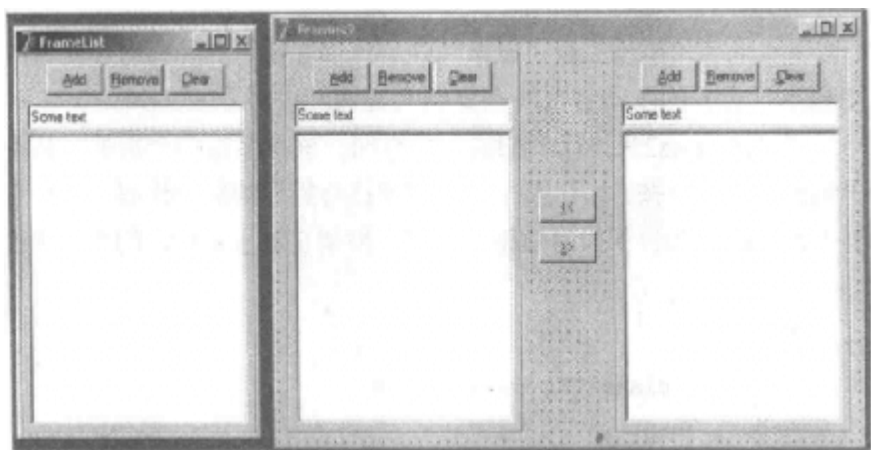


图8.9 Frames2范例在设计时拥有的一个框架及其两个实例

从上面的代码清单中可以看出，含有框架的窗体DFM文件使用了一个新的DFM关键字 **inline**。而对框架中被改动组件的引用使用了 **inherited** 关键字，尽管该术语的使用使其意义被扩展了。在此，**inherited** 引用的不是正在继承的基类，而是正在实例化对象的类。虽然使用可视窗体继承的已有特性，并将其应用到新概念中也许是一种好思想，但事实上该方法的效果是，可以使用对象检验器或窗体的 **Revert to Inherited** 命令取消所做的修改，并恢复属性的默认值。

还要注意，框架类中未改动的组件没有列在窗体（使用框架）的DFM文件中，而且窗体拥有两个不同名称的框架，但两个框架中的组件名称却相同。事实上，这些组件不属于窗体，而属于框架。这意味着，窗体必须通过框架引用这些组件，如下列按钮的代码（用于将一个列表框中的项复制到另一个列表框中）所示：

```

procedure TFormFrames.btnLeftClick(Sender: TObject);
begin
    FrameList1.ListBox.Items.AddStrings (FrameList2.ListBox.Items);
end;

```

最后，除了修改一个窗体的任何实例的属性外，可以改变其任何事件处理器的代码。当对窗体进行操作时，如果双击框架的任何一个按钮，则Delphi将产生这个代码：

```

procedure TFormFrames.FrameList2btnClearClick(Sender: TObject);
begin
    FrameList2.btnClearClick(Sender);
end;

```

由Delphi自动添加的代码行与调用继承的事件处理程序相对应。然而这一次，为了获得框架的默认行为，需要调用一个事件处理程序并将它应用于一个具体实例（框架对象自己）。

事实上, 当前窗体没有包括该事件处理程序, 而且对它一无所知。不管正确地放置这个调用还是删除它, 都依赖于用户所寻求的效果。

提示: 注意, 因为事件处理程序有一些代码, 所以保留其空状态并保存窗体将不会像通常那样删除它。

事实上, 它不是空的。相反, 如果只是想省略一个事件的默认代码, 则至少需要向它添加一条注释, 以避免它被系统自动删除。

框架与页面

当一个对话框由充满控件的多个页面组成时, 窗体背后的代码会变得非常复杂, 因为所有控件与对象方法都要声明在一个窗体中。而且, 建立所有这些组件 (并初始化它们) 可能会使对话框的显示延时。框架实际上并不缩减装载窗体所需的创建和初始化时间, 相反, 装载框架对于流系统来说比简单装载组件更复杂。然而使用框架, 只能装载一个多页对话框的可视页, 从而大大缩小初始的装载时间, 这正是用户所期望的。

框架技术可以很好地解决这些问题: 将一个复杂窗体的代码划分给每页的框架, 而窗体只是在一个PageControl中管理所有框架。这确实有助于建立更简单与重点更集中的单元, 并使得在不同对话框或应用程序中重复使用具体页面更容易。如果不使用框架或嵌套的窗体, 则重复使用PageControl的某一页是很困难的 (对于另一种方法来说, 参见附加说明“页面中的窗体”)。

为了举例说明该方法, 我们在这里建立了FramePage范例, 它含有一些框架放置在PageControl的三个页中, 如图8.10所示。所有的框架都与客户区对齐, 并占用整个页面。实际上, 有两页使用了相同的框架, 只是框架的两个实例在设计时有些区别。该范例的框架Frame3中有一个列表框, 在启动时填入一个文本文件, 其中的按钮用于改动列表中的条目并将它们保存进文件。文件名显示在一个标签中, 这样可以在设计时通过改动标签的Caption轻易地为框架选择文件。

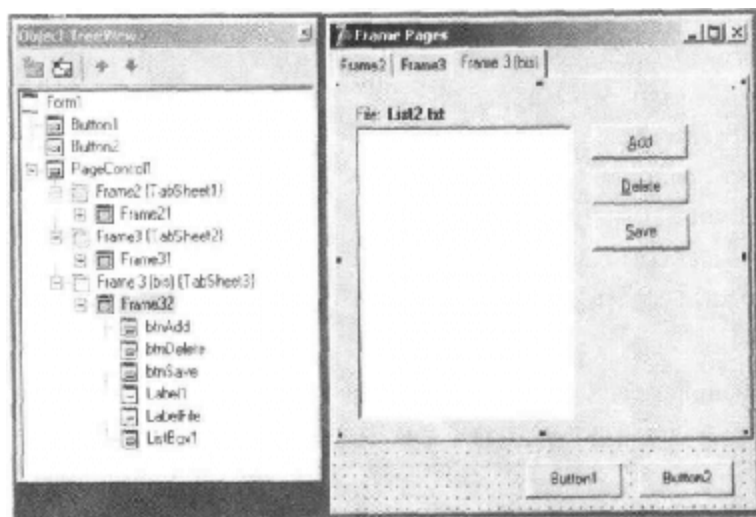


图8.10 FramePag范例包含一个框架, 这样可将复杂窗体的代码分割成多个可管理的块

可以使用一个框架的多个实例是引入该技术的一个原因, 而在设计时定制框架甚至更重要。因为在设计时向框架添加属性并使它们可以使用, 需要一些定制的复杂代码, 而使用

组件来管理这些定制值是很方便的。如果它们不适合用户界面，还可以隐藏这些组件（如该范例中的标签）。

在该范例中，我们需要在建立框架实例时装载文件。因为框架没有OnCreate事件，故最好的选择是覆盖CreateWnd对象方法。事实上，编写一个定制的构造器不起作用，因为过早（在指定的标签文本可用之前）地执行它是不行的。可以从一个文件中装载列表框内容。

说明：当提到窗体丢失OnCreate事件处理器问题时，Borland研发人员已经声明他们不能触发它以响应wm_Create消息，这是因为它随窗体一起发生。框架窗口的创建（对大部分控件来说是正确的）被性能原因所延缓。在拥有框架的窗体中继承会导致更多的困难，这种特性已经被禁用——程序员可以编写他们认为合理的代码。

页面中的窗体

尽管可以使用框架在设计时定义PageControl的页面，但笔者通常在运行时使用其它的窗体。这种方法会带来很大的灵活性，使页面在独立的单元（和DFM文件）中被定义，但同时允许将那些窗体做为独立的窗口使用。另外，也避免了框架行为之间轻微的差异。

一旦拥有一个带有一个页面控件或多个二级窗体的主窗体，所有必须要做的就是编写下列代码，以创建二级窗体并将其放置在页面上：

```
var
  Form: TForm;
  Sheet: TTabSheet;
begin
  // create a tabsheet within the page control
  Sheet := TTabSheet.Create(PageControl1);
  Sheet.PageControl := PageControl1;
  // create the form and place it in the tabsheet
  Form := TForm2.Create (Application);
  Form.BorderStyle := bsNone;
  Form.Align := alClient;
  Form.Parent := Sheet;
  Form.Visible := True;
  // activate and set title
  PageControl1.ActivePage := Sheet;
  Sheet.Caption := Form.Caption;
end;
```

可以在FormPage范例中找到这个代码，但是这是程序所能做的。对于一个应用程序来说，参见第14章“使用dbExpress的客户机/服务器编程”中的RWBlocks示例。

不含页面的多个框架

另一种方法是避免建立所有的页面连同控制它们的窗体。具体做法是，使PageControl为空状态，只有当一个页面被显示时，再建立框架。实际上，当将框架放在PageControl的多个页面上时，只有第一次显示它们时，这些用于框架的窗口才被建立，如果在上一个范例的代码中设置一个断点，就会发现该现象。

作为一种更极端的方法，可以不使用页面控件，转而使用TabControl。使用该方法，选项卡标没有连接的页面，但可以一次只显示一条信息。因此，需要建立当前框架，并解除前一个框架，或将它的Visible属性设置为False，或调用新框架的BringToFront隐藏它。尽管听起来需要很多工作，但是在一个大型的应用程序中，使用该技术可以减少资源与内存的消耗。

为了演示该方法，下面建立了一个FrameTab范例，与上一个范例相似，该范例基于一个TabControl并动态地建立框架。运行时显示的主窗体如图8.11所示，它只有一个TabControl控件，其中每个框架一页：

```
object Form1: TForm1
  Caption = 'Frame Pages'
  OnCreate = FormCreate
  object Button1: TButton...
  object Button2: TButton...
  object Tab: TTabControl
    Anchors = (akLeft, akTop, akRight, akBottom)
    Tabs.Strings = ( 'Frame2' 'Frame3' )
    OnChange = TabChange
  end
end
```

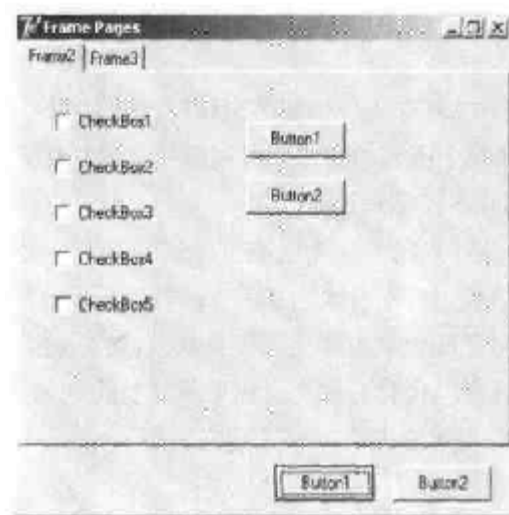


图8.11 FrameTab范例运行时的首页。选项卡中的框架在运行时创建

根据相应的框架名称赋给每个选项卡一个标题，这是因为打算使用该信息建立新页面。一旦窗体建立后，而且只要当用户改变当前选项卡时，程序就会读取选项卡的当前标题，并将它传递给定制的ShowFrame对象方法。该对象方法的代码检查被请求的框架是否已经存在（该范例中的框架名称遵循Delphi标准，在类名称后附加数字），如果存在，就将它显示出来。如果框架不存在，它会使用框架名称寻找出相关的框架类，建立该类的一个对象，并向它分配一些属性。此代码中使用了类引用与动态建立技术：

```
type
  TFrameClass = class of TFrame;
```

```

procedure TForm1.ShowFrame(FrameName: string);
var
  Frame: TFrame;
  FrameClass: TFrameClass;
begin
  Frame := FindComponent (FrameName + 'I') as TFrame;
  if not Assigned (Frame) then
  begin
    FrameClass := TFrameClass (FindClass ('T' + FrameName));
    Frame := FrameClass.Create (Self);
    Frame.Parent := Tab;
    Frame.Visible := True;
    Frame.Name := FrameName + 'I';
  end;
  Frame.BringToFront;
end;

```

为了使这段代码可以运行，必须记住在定义框架每个单元的初始化部分时，添加一个对RegisterClass的调用。

基窗体和界面

当需要在一个应用程序中含有两个相似的窗体时，可使用可视窗体的继承性从一个通用的基窗体那里创建一个或两个所需的窗体。可视窗体继承的优点是能使用它来继承可视定义，即DFM。然而，DFM并不经常被请求。

有时，想要几个窗体具有共同的行为或响应相同的命令，而不需要有任何共享组件或用户界面元素。此时，可以利用可视窗体的继承性，并使用没有额外组件的基窗体来实现这一目的。笔者更喜欢定义自己的定制窗体类，该类继承自TForm，然后手工地编辑窗体类声明，使之继承该定制基窗体类以代替标准类。如果所需要的是定义一些共享的对象方法，或以连续的方式覆盖TForm可视对象方法，那么定义定制窗体类是一个好办法。

使用基窗体类

在FormIntf演示中可获得该技巧的一个简单示范，它介绍了窗体界面的使用。在一个叫做SaveStatusForm的新单元中，已定义了下列窗体类（没有相关DFM文件也不使用新窗体命令，但在其上创建一个新单元并打印代码）：

```

type
  TSaveStatusForm = class (TForm)
  protected
    procedure DoCreate; override;
    procedure DoDestroy; override;
  end;

```

这两个重载对象方法在事件处理器的同一时间被调用,因此可以连接额外代码(允许事件处理器像平常一样被定义)。在两个对象方法中,简单地在应用程序的INI文件中装载或保存窗体位置。下面是这两个对象方法的代码:

```
procedure TSaveStatusForm.DoCreate;
var
  Ini: TIniFile;
begin
  inherited;
  Ini := TIniFile.Create (ExtractFileName (Application.ExeName));
  Left := Ini.ReadInteger(Caption, 'Left', Left);
  Top := Ini.ReadInteger(Caption, 'Top', Top);
  Width := Ini.ReadInteger(Caption, 'Width', Width);
  Height := Ini.ReadInteger(Caption, 'Height', Height);
  Ini.Free;
end;

procedure TSaveStatusForm.DoDestroy;
var
  Ini: TIniFile;
begin
  Ini := TIniFile.Create (ExtractFileName (Application.ExeName));
  Ini.WriteInteger(Caption, 'Left', Left);
  Ini.WriteInteger(Caption, 'Top', Top);
  Ini.WriteInteger(Caption, 'Width', Width);
  Ini.WriteInteger(Caption, 'Height', Height);
  Ini.Free;
  inherited;
end;
```

再者,上述代码对于窗体是一个简单的普通行为,但在这里定义了一个比较复杂的类。为了将该类作为创建窗体的父类,应简单地让Delphi像平常一样创建窗体(没有继承性),然后更新窗体声明,如下所示:

```
type
  TFormBitmap = class(TSaveStatusForm)
    Image1: TImage;
    OpenPictureDialog1: TOpenPictureDialog;
    ...
```

该技巧看起来很简单,但它是一个十分强大的技术,因为所有需要做的是改变应用程序窗体的定义来引用该父类。如果这样做太单调,也可以在我们整个程序中改变该父类,这里可以使用一个特殊技巧“*interposer*”类。

Delphi中的INI文件和注册

如果想保存关于应用程序状况的信息,便于在下次程序被执行时恢复它,可正式地使用Windows提供的支持来存储该类信息。INI文件,即老的Window标准再一次优先用来保存应用程序数据。另外,也可选择目前十分流行的注册方法。Delphi提供了备用类来处理两者。

TIniFile类

对于INI文件,Delphi有一个TIniFile类。一旦创建了该类的对象并把它连接到文件,就能向它读写信息。为了创建该对象,需要调用构造器,并向它传递一个文件名,如下所示:

```
var
  IniFile: TIniFile;
begin
  IniFile := TIniFile.Create ('myprogram.ini');
```

INI文件的位置有两种选择。在Window 2000中,刚才显示的代码将被保存在Windows目录下的文件中,或者被保存在用户设置的用户文件夹下。为了本地储存数据到应用程序(与本地相反是到当前用户),我们应提供一个到构造器的完整路径。

INI文件被分成几部分,每一部分用方括号中的名字说明。每部分能包含三个可能种类的多个项:字符串、整型和布尔型。TIniFile类有三个读取对象方法,分别用于读取各种数据:ReadBool、ReadInteger和ReadString。对于写数据也对应三种对象方法:WriteBool、WriteInteger和WriteString。其他对象方法允许读取或删除整个部分。在读取对象方法中,如果对应输入不存在于INI文件中,也可指定一个默认值。

注意,Delphi经常使用INI文件,但它们使用的名字不同。例如,桌面文件(.dsk)和选项文件(.dof)都作为INI文件被构造。

TRegistry和TRegIniFile类

Registry是有关计算机、软件配置和用户界面的信息分级数据库。Windows提供了一套API函数访问Registry:打开一个键(或文件夹),然后用子键(或子文件夹)和值(或项)工作,但必须了解Registry的结构和细节。

Delphi主要提供了两种使用Registry的方法。TRegistry类提供了Registry API的一般封装,而TRegIniFile类提供了TIniFile类的用户界面,并在Registry中保存信息。基于INI和基于Registry版本的同一个程序就其便携性而言,最好应该选择TRegIniFile类。当创建TRegIniFile对象时,数据止于当前用户信息,因此通常使用一个构造器,如下所示:

```
IniFile := TRegIniFile.Create ('Software\MyCompany\MyProgram');
```

通过使用VCL提供的TIniFile和TRegistryIniFile类,可以从一个本地模型和每个用户存储的模型移到另一个上。不应该过多地使用Registry,因此为每个应用程序设置有一个集中仓库的想法是一个结构上的错误。甚至连Microsoft都认识到这一问题(还没承认错误),所以Window 2000兼容性要求用户不再使用Registry设置程序,但可以在当前用户的Documents和Setting文件夹中使用INI文件。

一个额外技巧: Interposer类

与Delphi VCL组件相比,该类有惟一的名字,通常Delphi类在它的单元内其名字必须是惟一的。这就意味着可以在两个不同的单元中使用相同的名称来定义类。这看起来很不可

思议，但十分有用。例如，Borland公司正用该技术提供VCL和VisualCLX类间的兼容性。两者都有一个TForm类，一个定义在窗体单元，另一个定义在QForm单元。

说明：该技术比CLX/VCL还要老。例如，服务和控件面板程序定义它们自己的TApplication对象，它与TApplication无关，被VCL可视GUI应用程序使用并定义在窗体单元。

在一期“Delphi杂志”中，曾经介绍过名为“插入类”的技术，它建议用自己版本的名字代替标准的Delphi类名。因此程序员能在设计时使用Delphi设计器来引用Delphi的标准组件，但在运行时只能使用自己创建的类。

该方法是简单的。在SaveStatusForm单元，定义了新窗体类，如下所示：

```
type
  TForm = class (Forms.TForm)
  protected
    procedure DoCreate; override;
    procedure DoDestroy; override;
```

该类叫做TForm，继承自窗体单元的TForm（该最新引用是被强制使用的，用以避免递归定义）。程序的余下部分不需要改变窗体类的定义，但要在单元定义Delphi类后，在使用声明中简单地添加单元定义interposer类（在此例中是SaveStatusForm单元）。在使用use语句的声明单元中顺序是很重要的，因为用户不知道进行到哪里了，这也是一些人批评它的原因。笔者也赞成该意见：interposer类使用起来非常方便（对组件比对窗体更容易），但它们的使用使程序不易阅读，有时甚至较难调试。

使用界面

另一个技术更复杂，但也比一个普通父类窗口类的定义更强大，该技术可用于创建执行特殊界面的窗体。以这种方式程序员就可以为一个或更多界面创建窗体，查询它执行界面的每个窗体并调用支持的对象方法。

作为一个示例（可在后面部分讨论的FormIntf程序中获得），这里定义了一个简单的用于装载和存储的界面：

```
type
  IFormOperations = interface
    ['{DACFDB76-0703-4A40-A951-10D140B4A2A0}']
    procedure Load;
    procedure Save;
  end;
```

每个窗体都可选地实现这个界面，就像下列TFormBitmap类所示：

```
type
  TFormBitmap = class(TForm, IFormOperations)
    Image1: TImage;
    OpenPictureDialog1: TOpenPictureDialog;
    SavePictureDialog1: TSavePictureDialog;
  public
    procedure Load;
```

```

    procedure Save;
end;

```

读者可以看到用于Load和Save对象方法的实际代码，它使用标准对话框来装载或保存图像（在示例的代码中，窗体也继承自TSaveStatusForm类）。

当一个应用程序有一个或多个窗体执行界面时，就可对它支持的所有窗体应用已给的界面对象方法，代码如下（取自FormIntf示例的主窗体）：

```

procedure TFormMain.btnLoadClick(Sender: TObject);
var
    i: Integer;
    iFormOp: IFormOperations;
begin
    for i := 0 to Screen.FormCount - 1 do
        if Supports (Screen.Forms [i], IFormOperations, iFormOp) then
            iFormOp.Load;
    end;
end;

```

考虑一个商务应用程序，当可以将所有窗体同步到一个指定公司的数据上或一个指定商务事件上时，还要考虑不同的继承，每次执行多个界面，对可能出现的组合不加以限制。这就是为什么使用这样的结构能大大改进一个复杂的Delphi应用程序的原因，这样可使它更灵活且易于适应实现的变化。

Delphi的内存管理器

笔者将在这一节专门讲述内存管理，以结束对Delphi应用程序结构的专门讲解。这个主题是相当复杂的，并且可能值得专门用一章的篇幅来介绍，这里我只是简单地对其进行介绍。要想了解更多的内存分析，读者可以参考许多Delphi的附加工具，了解内存验证和控制，包括MemCheck、MemProof、MemorySleuth、Code Watch和AQTime。

Delphi有一个内存管理器，使用System单元的GetMemoryManager和SetMemoryManager函数来访问。这些函数允许用户提取当前内存管理器的记录或修改为自己的定制内存管理器。

一个内存管理器记录是用来分配、解除分配和再分配内存的三个函数的集合：

```

type
    TMemoryManager = record
        GetMem: function(Size: Integer): Pointer;
        FreeMem: function(P: Pointer): Integer;
        ReallocMem: function(P: Pointer; Size: Integer): Pointer;
    end;

```

重要的是知道这些函数在我们创建一个对象时是怎样被调用的，这是因为需要连接两个不同的步骤。就像调用一个构造器一样，Delphi会激活在TObject中定义的NetInstance虚拟类函数。因为这是一个虚拟函数，故通过覆盖技术，可以为一个具体类修改内存管理器。然而，为了执行内存分配操作，NewInstance通常会结束活动内存管理器的GetMem函数的调用，这为用户提供了又一个机会来定制标准的行为。

除非有特别的需求，否则通常并不需要连接到内存管理器中以修改内存分配的工作。然而，笔者发现连接到内存管理器中来确定是否内存分配工作正常是非常有用的——这就是说，能够保证程序没有内存遗漏。例如，可以覆盖一个类的NewInstance和FreeInstance方法，来保持被创建和解除的类对象的数量，并检查总数是否为0。

一个相当简单的技术是执行相同的测试，以确定被整个内存管理器分配的对象的数量。在Delphi的早期版本中，这样做需要额外的代码，但是内存管理器提供了两个全局变量（AllocMemCount和AllocMemSize），以帮助用户确定系统中正在进行什么。

说明：要想了解更多的关于内存管理器内部是如何工作的信息，可以使用GetHeapStatus函数。这只能在Windows中使用，因为它提供了关于内存分配器状态的信息。在Linux中，RTL使用系统分配器，而不是一个定制的内存分配器。

确定程序是否正确地处理内存的最简单的方法就是，测试AllocMemCount是否归零。一个程序通过执行其单元的初始化部分开始运行，这通常会通过各自的最终部分释放内存。为了保证代码被执行，必须在一个单元的最终部分编写它，并将其放置在项目源代码文件的单元列表的最前面。读者可以在程序清单8.1中看到这样一个单元。这是ObjLeft范例的SimpleMemTest单元，该单元会有一个带按钮的范例窗体，其中一个按钮能够显示当前的分配数，另一个按钮用于创建一个内存遗漏（当程序终止时会被发现）。

程序清单8.1 测试内存遗漏的简单单元，摘自ObjLeft范例

```
unit SimpleMemTest;

interface

implementation

uses
  Windows;

var
  msg: string;

initialization

finalization
  if AllocMemCount > 0 then
  begin
    Str (AllocMemCount, msg);
    msg := msg + ' heap blocks left';
    MessageBox (0, PChar(msg), 'Memory Leak', MB_OK);
  end;
end.
```

提示：当编写涉及检查、实现或扩展内存管理器的代码时，必须避免使用任何高级函数，因为它们可能会影响内存管理器。例如，在SimpleMemTest单元中，不能包含SysUtils单元，这是因为它会分配内存。因此必须求助于传统的Turbo Pascal Str函数，而不是Delphi的标准的IntToStr转化。

这种程序是方便的，但是不能真正帮助读者理解发生了什么错误。因此，需要使用强大的第三方工具（其中一些是免费版本），或者引用笔者定制的内存管理器，这将跟踪内存的分配（在本书的附录A中进行介绍）。

小结

在前面章节详细介绍了窗体和二级窗体后，本章将重点放在应用程序的结构上，讨论Delphi应用程序对象怎样工作的和怎样构造有多个窗体的应用程序。

这里特别介绍了MDI、可视窗体继承和框架。本章最后还用窗体继承性和界面介绍了定制结构。现在进一步介绍Delphi应用程序的另一个关键元素：建立程序中使用的定制组件。有关这些方面的知识可以编写一本书，因此，在此就不详细描述了，但应该对该问题有一个综合的概述。

另一个与Delphi应用程序结构相关的元素是包的使用，应把它作为一个与组件相关的技术介绍，但它已超过了本书的范围。事实上，能在多个包上构造一个应用程序的代码，包括窗体和其他单元。程序的开发基于多个可执行文件、库和包，将在第10章介绍它们。

随后将专门介绍Delphi数据库编程，它是Borland开发环境的另一个关键元素，而且对于很多开发人员来说是主要的侧重点。

第9章 编写Delphi组件

虽然多数Delphi程序员已经熟悉现有组件的使用，不过，有时编写自己的组件或定制现有组件也是有用的。Delphi最有意义的一面就是创建组件并不比编写程序难。由于这个原因，尽管本书是为Delphi应用程序程序员，而不是为Delphi工具编写者所写的，但本章仍将涉及创建组件的有关内容并介绍Delphi提供的附加功能，如组件和属性编辑器。

本章将综述Delphi组件的编写并展示一些简单示例。尽管没有足够的空间展示非常复杂的组件，但笔者的那些想法将覆盖相关内容的所有基本概念。

说明：读者将在第17章“编写数据库组件”中找到更多关于编写组件的信息，包括怎样建立数据敏感组件。

本章主要包括以下内容：

- 扩充Delphi库
- 编写组件包
- 复合组件
- 使用界面类型属性
- 定义定制事件
- 定制已有组件
- 使用数组属性
- 在一个组件上放置对话框
- 集合属性
- 编写属性和组件编辑器

扩充Delphi库

类是Delphi的组件，可视组件库（VCL）是所有定义Delphi组件类的集合。可以通过向一个组件包中写入新的组件类并将其安装到Delphi中来扩充VCL。这些新类将来自一个现有组件相关的类或一般的TComponent类，向那些它继承的组件添加新功能。

可以从现有组件或抽象组件类（不对应一个可用组件）中获得新的组件。VCL层级包括许多这样的中间类（经常用TCustom作为它们的名字前缀），允许用户为新的组件选择一个默认行为并改变它的属性。

组件包

组件被添加到组件包中。每个组件包基本上都是一种扩展名为BPL（代表Borland Package Library）的DLL（一个动态链接库）。

组件包可以分成两种：Delphi IDE使用的设计时组件包，以及应用程序使用的运行时组件包。组件包的类型由设计时或运行时组件包选项确定。当试图安装一个组件包时，IDE

会检测它是否具有**Design-only**或**Run-only**标记，并决定是否允许用户安装组件包，以及是否将它添加到运行时组件包列表中去。因为有两个非互斥选项，每一个又有两种可能的状态，所以就有四种不同的组件包——两个主要变体与两种特殊情况：

- **Design-only**组件包可以被安装在Delphi环境中。这些**Design-only**组件包通常包含了一个组件的设计时部分，如它的属性编辑器与注册代码。通常它们还可以包含组件自己，尽管这不是最专业的方式。一个**Design-only**组件包的代码通过使用相应的DCU（Delphi编译单元）文件的代码静态地被链接到执行文件中。然而，应当记住的是，从技术可行性角度看，**Design-only**组件包也可以作为一个运行时组件包使用。
- **Run-only**组件包在运行时由Delphi应用程序使用。它们不能安装到Delphi环境中，但当已安装的一个**Design-only**组件包需要它们时，会将它们自动地添加到运行时组件包列表中。**Run-only**组件包通常包含了组件类的代码，但不能在设计时使用（这样做是为了尽可能减小可执行文件所带组件库的大小）。**Run-only**组件包是重要的，这是因为它们可以与应用程序一起被免费使用，但其他的Delphi程序员将不能在环境中安装它们以建立新程序。
- 简单的组件包（既没有**Design-only**选项设置也没有**Run-only**选项设置）不能被安装，也不能被自动添加到运行时组件包列表中去。只有被其他组件包当做工具组件包时，这个选项才有意义，但这种组件包是相当少的。
- 两个标记都被设置的组件包既可以被安装也可以被自动添加到运行时组件包列表中去。通常这些组件包包含了很少需要或根本不需要设计时支持的组件（除了有限的组件注册代码）。

提示：Delphi自己的**Design-only**组件包文件名以字母DCL打头（例如，DCLSTD60.BPL）；**Run-only**组件包的文件名以字母VCL打头（例如，VCL60.BPL）。如果愿意，也可以对自己的组件包使用相同的方法命名。

在第1章“Delphi 7及其IDE”中，我们曾经讨论过组件包对程序可执行文件大小的影响。现在，我们将介绍如何建立组件包，因为这是在Delphi中创建或安装组件的一个必需的步骤。

当编译一个运行时组件包时，会产生一个带有已编译代码的动态链接库（BPL文件）和一个只带有符号信息的文件（一个DCP文件），但不含有已编译的机器码。Delphi编译器使用后一种文件来收集那些属于组件包的一部分而不用访问单元（DCU）文件（这些文件既包含符号信息也包含已编译机器码）的有关单元。这个过程减少了编译时间并允许用户只发布组件包而不需要预编译单元文件。预编译单元在静态地将组件链接到应用程序中时还是需要的。预编译的DCU文件（或源代码）的发布要根据所开发的组件种类而定。在我们讨论一些通用规则并创建一个简单组件之后，大家将会看到如何创建一个组件包。

说明：DLL是包含了函数与类集合的可执行文件，它可以由应用程序或其他DLL在运行时使用。其典型的优点是，如果很多的应用程序使用同一个DLL，在磁盘上或内存中只需要一个拷贝，而且每个执行文件所占空间会大大缩小。这也是使用新型Delphi组件包可以实现的优点。第10章“库与组件包”中将更为详细地介绍DLL和组件包。

编写组件的规则

编写组件要遵循一些通用的规则。读者在“Delphi组件编写者指南”(Delphi Component Writer's Guide)帮助文件中可以发现大部分有关这些规则的详细描述,该手册对于Delphi组件编写者来说应该是必读的。

下面是笔者为组件编写者汇总的一个组件编写规则:

- 用心研究Object Pascal语言。特别重要的概念是继承、对象方法覆盖和重载、类中public部分与published部分之间的区别以及属性与事件的定义。如果读者对Delphi语言或基础的VCL概念不是特别熟悉,可以参考本书第一部分对语言和库的概述,特别是第2章“Delphi编程语言”和第4章“核心库类”。
- 研究VCL类层次式的结构并在于手边保留一份类的图表(如Delphi中包括的图表)。
- 遵循标准的Delphi命名规则。对于组件来说,就像大家将要看到的那样,命名规则有很多,并且遵循这些规则可以使其他程序员更容易地与他人开发的组件进行交互,并进一步地扩展它们。
- 保持组件的简单化,模拟其他组件,避免相关性。这三个规则的基本意思是,当一个程序员使用他人编写的组件时,应该能够像使用Delphi预安装的组件那样容易。无论何时,要尽可能地使用相似的属性、对象方法以及事件名称。如果用户不需要学习他人组件的复杂的使用规则(也就是说,如果对象方法或属性之间的相关性是有限的),并且可以使用有意义的名称访问属性,那么他们又何乐而不为呢?
- 使用异常。当出现一些错误时,组件应该会引起一个异常。在分配任何种类的资源时,必须使用try/finally块,并且根据需要,调用解除程序来保护它们。
- 为了完成一个组件,需要向它添加一个位图,以备Delphi的组件面板使用。如果打算让更多的人使用自己的组件,最后还应该考虑再添加一个帮助文件。
- 准备编写纯代码并且要忘掉Delphi的可视部分。编写组件通常意味着编写不含可视支持的代码(尽管Class Completion可以大大加快普通类代码的编写速度)。这个规则的例外是,可以使用框架可视地编写组件。

说明:也可使用一个第三方组件编写工具来创建自己的组件,或提高它的开发速度。我所知道的一个最强大的用于创建Delphi组件的第三方开发工具组件就是Class Developer Kit (CDK),该工具由Eagle Software (www.eaglesoftware.com)提供。除此之外,当然也还有其他很多的工具可以利用。

基本组件类

为了建立一个新的组件,通常应该从一个现有组件开始或从VCL的某个基类开始。在这两种情况中,组件应该属于三种主要组件类型中的一种(在第4章介绍过),由组件层次式结构的三个基类来确定:

- TWinControl是任何基于一个窗口的组件的父类。继承自该类的组件可以从系统接受输入焦点并从系统中获取Windows消息。当调用API函数时,还可以使用它们的窗口句柄。当建立一个崭新的窗口控件时,通常需要从TCustomControl派生类(有一些特别有用的特性,特别是支持绘制控件)中继承新类。

- TGraphicControl是不含Windows句柄（保存了一些Windows资源）的可视组件的父类。这些组件不能接受输入焦点，或不能直接响应Windows消息。当建立一个崭新的图形控件时，将直接从该类（它有一系列与TCustomControl非常相似的特性）继承。
- TComponent是所有组件的父类（包括控件），并可以用做一个非可视组件的直接父类。

在本章的其他各节中，我们将使用不同的父类建立一些组件，并且研究它们之间的不同之处。让我们从继承自现有组件或位于层次式结构底层类的组件开始，然后介绍直接继承上面提到的父类来建立组件的范例。

创建自己的第一个组件

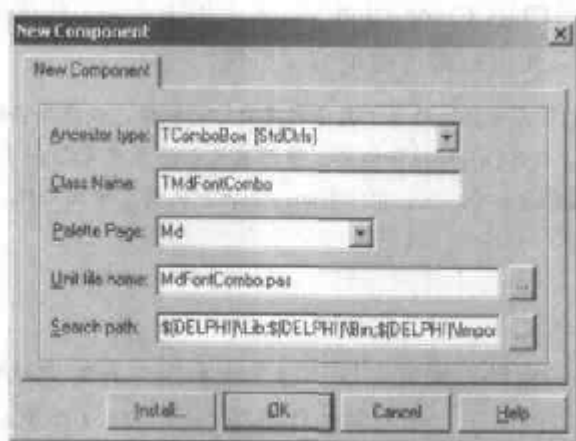
建立组件实际上是Delphi程序员的一项重要工作。基本思想是，当在一个应用程序的两个不同位置或在两个不同的应用程序中实现相同的性能时，可以将共享代码放置到一个类中——放到一个组件中更好。

在这一节中，笔者将介绍两个简单的组件，以此讨论建立组件所需的步骤。同时介绍在使用有限的代码定制已有组件时，需要做的各种事情。

字体组合框

很多应用程序都有一个带组合框的工具栏，可以使用该组合框选择一种字体。如果经常使用这样一个定制的组合框，为什么不将它转化为一个组件呢？花上不到一分钟的时间就可能完成该转化。

一开始，在Delphi环境中关闭任何激活的项目，通过选择Component►New Component命令启动组件向导，或选择File►New►Other菜单打开对象存储库，然后在New页面中选择Component。就像读者可以看到的那样，这个组件向导需要下列信息：



- 父类型的名称：要继承的组件类名称。在本例中，使用TComboBox。
- 正建立的新组件的类名称：本例使用TMdFontCombo。
- 想要显示新组件的组件面板的页面，该组件可以是一个新页面或是一个现有的页面。可以创建一个新页面，叫做Md。
- Delphi将放置新组件源代码的单元文件名：可以键入MdFontBox。

- 当前的搜索路径（此项应自动建立）。

单击OK按钮，组件向导将会产生下列简单的源文件，如程序清单9.1所示，它定义了组件的结构。另一个Install按钮可以用来立刻在一个组件包中安装组件。让我们先来查看代码，然后再讨论安装。

程序清单9.1 TMdFontCombo类的代码，由组件向导生成

```
unit MdFontCombo;

interface

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

type
  TMdFontCombo = class (TComboBox)
  private
    { Private declarations }
  protected
    { Protected declarations }
  public
    { Public declarations }
  published
    { Published declarations }
  end;

procedure Register;

implementation

procedure Register;
begin
  RegisterComponents('Md', [TMdFontCombo]);
end;

end.
```

这段代码中一个关键的要素就是类定义，它从指定父类开始。惟一的其他相关部分是Register过程。事实上，可以看出组件向导没有做太多的工作。

警告：Register过程必须使用大写字母R来编写。这个要求显然与C++ Builder的兼容性有关（标识符在C++中是大小写敏感的）。

提示：当建立组件时应使用一个命名规则。所有在Delphi中安装的组件都应当有不同的类名。因此，大多数Delphi组件开发人员都选择两到三个字母标记作为他们组件名称的前缀。我们也应该这样，使用Md（用于Mastering Delphi）来标识在本书中创建的组件。这种方法的优点是，即使已经安装了一个叫TFontCombo的组件，也可以再安装我们开发的TMdFontCombo组件。注意，对于在系统中安装的所有组件，单元名称必须是惟一的，所以对单元名称应用了相同的前缀。

这就是建立一个组件的全部工作。当然，在该范例中，代码非常简单。只需要在启动时向组合框的Items属性复制所有的系统字体。为了实现该想法，可以试着在类声明中覆盖

Create对象方法，添加语句Items:=Screen.Fonts。然而，这并不是正确的方法。问题是，不能在组件的窗口句柄可以使用之前，访问组合框的Items属性；而且组件在设置其Parent属性之前，不能拥有一个窗口句柄；并且该属性没有在构造器中设置。

因此，不能在Create构造程序中赋予新的字符串，而必须在CreateWnd过程中执行这个操作，即在组件构建完成后、Parent属性设置后、其窗口句柄可用后，再调用该过程来建立窗口控件。执行了默认行为，就可以编写所需的定制代码了。虽然可以跳过Create构造程序在CreateWnd中编写所有代码，但这里决定使用这两个启动对象方法来演示它们之间的区别。下面是该组件类的声明：

```
type
  TMdFontCombo = class (TComboBox)
  private
    FChangeFormFont: Boolean;
    procedure SetChangeFormFont(const Value: Boolean);
  public
    constructor Create (AOwner: TComponent); override;
    procedure CreateWnd; override;
    procedure Change; override;
  published
    property Style default csDropDownList;
    property Items stored False;
    property ChangeFormFont: Boolean
      read FChangeFormFont write SetChangeFormFont default True;
  end;
```

而启动时执行的那两个方法的源代码如下：

```
constructor TMdFontCombo.Create (AOwner: TComponent);
begin
  inherited Create (AOwner);
  Style := csDropDownList;
  FChangeFormFont := True;
end;

procedure TMdFontCombo.CreateWnd;
begin
  inherited CreateWnd;
  Items.Assign (Screen.Fonts);

  // grab the default font of the owner form
  if FChangeFormFont and Assigned (Owner) and (Owner is TForm)
  then
    ItemIndex := Items.IndexOf ((Owner as TForm).Font.Name);
end;
```

注意，除了在Create对象方法中赋予组件的Style属性一个新值外，还可以用default关键字通过设置一个值来重新定义该属性。而且必须进行这两步操作，因为向一个属性声明添加

default关键字不会对属性的初始值产生直接影响。为什么要指定属性的默认值呢？因为具有默认值的属性不用包含在窗体定义中（而且它们也不会出现在窗体文本描述——DFM文件中）。**default**关键字会向流代码说明，组件初始化代码将设置该属性的值。

提示：为公布的属性指定默认值是非常重要的，目的是为了减小DFM文件，进而减小可执行文件（其中包括DFM文件）的大小。

另一个重新定义的属性**Items**被设置为一个根本无需保存在DFM文件中的属性，而不论其实际值是什么。这种行为可以通过后跟**False**值的**stored**指令获得。当程序启动时，将重新建立组件及其窗口，所以将以后会丢弃的信息（被字体的新列表代替）保存在DFM文件中是没有意义的。

说明：甚至可以编写CreateWnd对象方法的代码，只在运行时将字体复制到组合框中。可以用如下语句来实现：`if not(csDesigning in ComponentState)`，但对于我们建立的第一个组件，效果稍差但更为直观的对象方法可以更清楚地解释基本过程。

第三个属性**ChangeFormFont**不被继承，但将通过组件引入。它用来确定组合框当前字体的选择是否拥有组件窗体的字体。另外，该属性被声明为在构造器中设置的默认值。**ChangeFormFont**属性用在以前提到的CreateWnd对象方法代码中。这通常是组件的拥有者，尽管仍需通过Parent树来查找窗体组件。该代码并不是完美的，但Assigned和is测试提供了一些额外的安全保障。

ChangeFormFont属性和相同的if测试在Changed对象方法上起到了关键作用，它们在基类上触发OnChange事件。通过重设该对象方法可提供一个默认行为（该属性可通过开关值被设置为禁用），并且允许OnChange事件执行，因此该类的用户能完全定制它的行为。最后，SetChangeFormFont对象方法已被修改，以便当该属性打开时刷新窗体的字体。完整的代码如下所示：

```
procedure TMDFontCombo.Change;
begin
    // assign the font to the owner form
    if FChangeFormFont and Assigned (Owner)
        and (Owner is TForm)
    then
        TForm (Owner).Font.Name := Text;
    inherited;
end;

procedure TMDFontCombo.SetChangeFormFont(const Value: Boolean);
begin
    FChangeFormFont := Value;
    // refresh font
    if FChangeFormFont then
        Change;
end;
```

建立一个组件包

现在，必须使用一个组件包将组件安装到环境中。对于这个范例，可以创建一个新的组件包，或使用一个现有的组件包，例如默认用户的组件包。

在这两种情况中，只要选择**Component►Install Component**菜单命令，结果对话框中就会出现一个允许将组件安装到已有组件包中的页面，以及一个用于建立新组件包的页面。在后一种情况中，只需键入一个文件名并为组件包输入一个描述即可。单击**OK**，打开**Package Editor**（如图9.1所示），它由两部分组成：

- **Contains**列表指出了组件包中包括的组件（或更精确地说，是定义这些组件的单元）。
- **Requires**列表指出了这个组件包所需的组件包。这一组件包通常需要使用**rtl**和**vcl**组件包（主要的运行时库组件包和核心**VCL**组件包），但如果新组件包的组件要进行任何数据库相关操作的话，还可能需要**vcldb**组件包（它包括了大多数与数据库相关的类）。

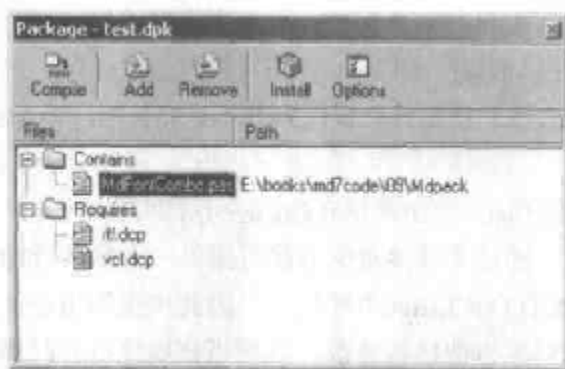


图9.1 组件包编辑器

说明：从Delphi 6开始，组件包名称不再与版本有任何关系，即使编译包在文件名中仍有版本编号。

详情参见第10章中的“修改项目和库名称”小节。

向刚定义的新组件包中添加组件，然后简单地编译该组件包并安装它（使用组件包编辑器的两个相应的工具栏按钮），将马上会看到在组件面板的**Md**页面中显示的新组件。组件单元文件的**Register**过程会告诉**Delphi**向哪里安装新组件。默认情况下，使用的位图将与父类的位图相同，因为我们在该程序中没有提供一个定制的位图（将在以后的范例中进行该操作）。还要注意，如果将鼠标移到新组件上，**Delphi**会显示不带首字母**T**的类名称作为提示。

一个组件包的幕后有什么呢

组件包编辑器为组件包项目创建了基本的源代码：在**Delphi**中建立的一个特殊种类的**DLL**。组件包项目保存在一个扩展名为**DPK**（用于**Delphi PacKage**）的文件中。一个典型的组件包项目如下所示：

```
package MdPack;  
  
{ $R *.RES }  
{ $ALIGN ON }
```

```
{SBOOLEVAL OFF}  
{DEBUGINFO ON}  
...  
{DESCRIPTION 'Mastering Delphi Package'}  
{IMPLICITBUILD ON}  
  
requires  
    vcl;  
  
contains  
    MdfontBox in 'MdfontBox.pas';  
  
end.
```

可以看到，Delphi对组件包使用了特殊的语言关键字：第一个是**package**关键字（它与在下一章中讨论的**library**关键字相似），该关键字引入一个新组件包项目。然后是所有编译器选项的列表，我们已经从列表中略去了其中的部分选项。通常，Delphi项目的选项存储在一个独立的文件中；与之相对，组件包却在源代码中包含所有的编译器选项。在编译器选项中有个**DESCRIPTION**编译器指令，用于使组件包的描述在Delphi环境中可用。实际上，在安装新组件包后，它的描述将显示在Project Options对话框的Package页中，还可以通过选择Component►Install Packages菜单命令激活该页。该对话框如图9.2所示。

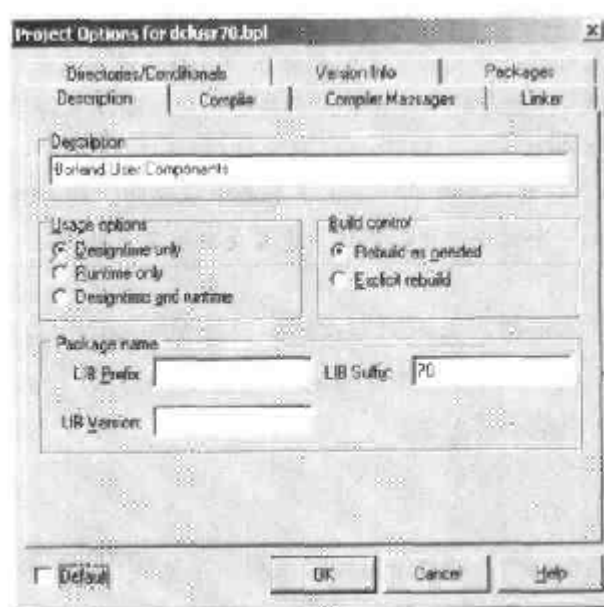


图9.2 组件包的Project Options对话框

除了诸如**DESCRIPTION**这样的通用指令外，还有其他一些组件包专用指令。这些最通用的选项可以通过Package Editor的Options按钮轻松地访问到。在选项列表之后是**requires**与**contains**关键字，它们列出的选项可以在组件包编辑器的两个页面中看到。**requires**列出了当前组件包所需的组件包，**contains**列出了由该组件包安装的单元。

让我们考虑一下建立一个组件包所产生的技术性后果。除了带有源代码的DPK文件外，Delphi还会生成带有组件包动态链接库版本的BPL文件，以及带有符号信息的DCP文件。实际上，这个DCP文件是组件包内所含单元的DCU文件的符号信息汇总。

在设计时, Delphi需要BPL与DCP文件, 因为BPL文件拥有在设计窗体上建立的组件实际代码与代码识别技术所需的符号信息。如果动态地链接组件包(将它用做一个运行时组件包), DCP文件还要被连接器使用, 而且BPL文件应该与应用程序的主可执行文件链接在一起。如果静态地链接组件包, 连接器会引用DCU文件, 而且只需要发布最终的可执行文件。

因此, 作为一个组件设计者, 通常至少应该分配BPL文件、DCP文件、组件包中所含单元的DCU文件与一些相应的DFM文件, 以及帮助文件。当然, 作为一种选择, 还可以使用组件包单元的源代码文件(PAS文件)与组件包自己的源代码文件(DPK文件)。

警告: 默认情况下, Delphi将所有编译包文件(BPL和DCP)放置在\Projects\BPL文件夹下而不是在包的源代码文件夹下。这样IDE能轻松定位它们并不会带来特殊问题。然而, 当不得不使用在这些包上声明的组件编译项目时, Delphi将向用户报告, 它不能发现存储在包源代码文件夹中的对应DCU文件。这一问题能通过在庫路径中(在Environment Options中指定, 它影响所有项目)指定包的源代码文件夹解决, 或通过查找路径中为当前项目(在Project Options中)指定该文件夹而解决。如果选择第一种方法, 在单个文件夹放置不同组件和包可能会真正地节省时间。

安装本章中创建的组件

建立第一个组件包后, 就可以使用向它添加的组件了。然而, 在此之前, 应该考虑扩展MdPack组件包, 以包括在本章中建立的所有组件, 并包括相同组件的不同版本。建议读者安装该组件包。最好的方法是将它复制到自己所在路径的目录中, 这样无论是对Delphi环境还是对自己使用Delphi建立的程序来说, 都能使用它。这里选择了所有的组件源代码文件与一个子目录中的组件包定义, MdPack。这允许Delphi环境(或一个特定项目)在查找组件包的DCU文件时只引用一个目录。如上面警告中建议的, 读者应将所有出现在本书中的组件收集到网站上的一个文件夹中; 不过, 笔者决定按章节进行组织会更易于读者的理解。

需要牢记, 如果使用诸如运行时库这样的组件包编译应用程序, 则需要在自己的客户端安装这些新的组件库。如果静态地链接组件包来编译程序, 那么只有开发环境需要组件包库, 而应用程序的用户是不需要的。

使用字体组合框

现在, 让我们创建一个新的Delphi程序来测试字体组合框。移到组件面板上, 选择新组件, 将它添加到一个新窗体上, 之后会出现一个传统模样的组合框。然而, 如果打开Items属性编辑器, 将看到一个安装在计算机上的字体列表。为了建立一个简单的范例, 这里向带有一些文本的窗体内添加了一个Memo组件, 通过保留ChangeFormFont属性的设置状态, 无需再为该程序编写其他代码, 如例中所见到的那样。或者, 也可关闭该属性, 并通过以下代码处理该组件的OnChange事件:

```
Memol.Font.Name := MdFontCombol.Text;
```

这个简单范例的目的只是为了测试新建组件的功能。组件的作用还不太大——本可以向窗体添加两行代码来达到一些效果——但研究两个简单的组件, 将有助于读者对建立组件所涉及的内容有一个较好的认识。

建立复合组件

组件不能孤立地存在。程序员通常将组件与其他的组件结合在一起使用，在一个或多个事件处理器中编写它们的关系。另一个方法就是编写复合组件，该方法可以封装这种关系，并使它易于处理。这里有两种不同的复合组件类型：

内部组件 由主组件创建和管理，也会带有一些它们自己的属性和事件。

外部组件 使用属性连接。这样一种复合组件会自动化两个独立组件之间的交互作用，而两个组件可以在相同的或不同的窗体或设计器上。

在这两种情况下进行开发将遵循一些标准的规则。

第三种方法不太常用，它涉及了组件构造器的开发和子控件的交互作用，这是一个更高级的主题，笔者在这里对其将不做介绍。

内部组件

下面要讨论的组件是一个数字时钟。该范例有一些有趣的特性，首先，它将一个组件（Timer组件）嵌入另一个组件中；第二，它显示了处理动态数据的方法：大家将能够看到一个即使在设计时也是动态的行为（时钟的事件被不断更新），就像它用数据敏感组件所做的那样。

说明：第一个特性从Delphi 6起已变得更加好用，因为对象检验器允许用户直接访问子组件的属性。作为一个结果，与本书的Delphi 5相比，在该部分显示的示例已被修改（并简化）。当介绍时，笔者将提及不同之处。

因为该数字时钟将提供一些文本输出，所以我考虑从TLabel类继承。然而，如果这样做会允许一个用户改变标题框的标题——也就是说，时钟的文本。为了避免该问题，可以直接使用TCustomLabel组件作为父类。除了几个Published属性外，TCustomLabel对象具有与TLabel对象相同的功能。换句话说，一个从TCustomLabel继承而来的子类可以决定什么属性应该使用，什么属性应该保持隐藏状态。

说明：大多数的Delphi组件，特别是基于Windows的组件，都有一个TCustomXxx基类，它实现了所有的功能，但只暴露出有限的属性集合。从这些基类继承，是在组件定制版本中暴露一些属性的标准方法。事实上，无法隐藏基类中的public与published属性，除非通过在继承类中使用相同的名字定义一个新的属性来隐藏它们。

在Delphi的老版本中，组件必须定义一个新的属性Active——通过捆绑定时器组件的Enabled属性。一个Wrapper属性意味着该属性的get和set对象方法可以读取并编写属于一个内部组件的Wrapper属性的值（一个Wrapper属性通常没有本地数据）。在这种特定情况下，其代码如下：

```
function TMDClock.GetActive: Boolean;
begin
    Result := FTimer.Enabled;
end;

procedure TMDClock.SetActive (Value: Boolean);
```

```
begin
    FTimer.Enabled := Value;
end;
```

公布子组件

从Delphi 6开始，用可以在子组件自己的属性中简单揭示全部子组件（定时器），它们将被对象检验器有规律地扩充，并允许用户设置其每一个子属性，甚至处理其事件。

下面是用于TMDClock组件的全部类型声明，其中包括在private数据中声明的并作为一个published属性展示的子组件（在最后一行中）：

```
type
    TMDClock = class (TCustomLabel)
    private
        FTimer: TTimer;
    protected
        procedure UpdateClock (Sender: TObject);
    public
        constructor Create (AOwner: TComponent); override;
    published
        property Align;
        property Alignment;
        property Color;
        property Font;
        property ParentColor;
        property ParentFont;
        property ParentShowHint;
        property PopupMenu;
        property ShowHint;
        property Transparent;
        property Visible;
        property Timer: TTimer read FTimer;
    end;
```

Timer属性是只读的，因为笔者在这里不想让读者在对象检验器（或通过清除该属性的值分离组件）中为该组件挑选另一个值。子组件开发集合的选择性可被作为一种可替换的解决方案，但添加用于安全地写该属性的功能非常复杂（考虑到使用所创组件的用户并非都是专业的Delphi程序员）。因此我建议读者对子组件使用只读属性。

为了创建Timer，必须覆盖时钟组件的构造程序。Create对象方法可调用相应的基类对象方法并建立Timer对象——通过为它的OnTimer事件安装处理程序：

```
constructor TMDClock.Create (AOwner: TComponent);
begin
    inherited Create (AOwner);
    // create the internal timer object
    FTimer := TTimer.Create (Self);
    FTimer.Name := 'ClockTimer';
```

```

FTimer.OnTimer := UpdateClock;
FTimer.Enabled := True;
FTimer.SetSubComponent (True);
end;

```

该代码给出了组件名字，以展示在对象检验器上（如图9.4所示），并调用特殊的SetSubComponent对象方法。这里不需要解除程序，只是因为FTimer对象将TMdClock组件做为宿主（如其Create构造程序的参数说明的一样），所以当时钟组件解除时，它将会自动地被解除。

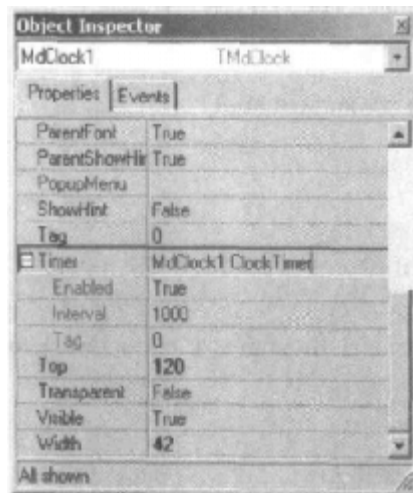


图9.4 对象检验器可自动扩展子组件，显示其属性，如TMdClock组件的Timer属性

说明：在上面代码中，调用SetSubComponent对象方法设置了一个内部标记，保存在ComponStyle属性中。标记（csSubComponent）会影响流系统，允许子组件和它的属性被保存在DFM文件中。事实上，默认情况下，流系统就会忽视不被窗体拥有的组件。

该组件代码的一个关键部分就是UpdateClock进程，它只是一条语句：

```

procedure TMdLabelClock.UpdateClock (Sender: TObject);
begin
    // set the current time as caption
    Caption := TimeToStr (Time);
end;

```

这种对象方法使用了Caption，这是一个非公布属性，所以该组件的一个用户就不能在对象检验器中改动它。这段代码语句的结果是连续不断地显示当前的时间，因为该对象方法是与Timer的OnTimer事件相连的。

说明：子组件的事件可以在对象检验器中被编辑，这样可以使一个用户对它们进行处理。如果你在内部处理这些事件，就像我在TmdLabelClock中那样做的，则一个用户可以通过处理事件来覆盖这种行为，在本例中是OnTimer。通常，解决方案是确定一个用于内部组件的导出类，覆盖它的虚拟方法，诸如TTimer类的Timer方法。尽管如此，在这个例子中，这种技巧将不能正常工作，这是因为，Delphi只有在一个事件处理器被附加到其上时才会激活一个定时器。如果覆盖虚拟方法而并没有提供事件处理器（在一个子组件中将是正确的），则定时器将不会正常工作。

外部组件

当一个组件引用一个外部组件时，它不会自己创建这个组件（这就是为什么称为外部的原因），而是要由程序员使用组件分别独立地创建这两种组件（例如，将它们从组件面板拖拽到一个窗体中），并使用它们的属性之一将这两个组件连接到一起。所以我们可以说，是组件的属性引用了外部链接的组件。这个属性必须是继承自TComponent的一个类的类型。

为了更好地说明，笔者已经建立了一个非可视组件，可以在一个标签上显示关于一个人的数据，并自动地刷新数据。该组件拥有下面这些published属性：

```
type
  TmdPersonalData = class(TComponent)
  ...
  published
    property FirstName: string read FFirstName write SetFirstName;
    property LastName: string read FLastName write SetLastName;
    property Age: Integer read FAge write SetAge;
    property Description: string read GetDescription;
    property OutLabel: TLabel read FLabel write SetLabel;
  end;
```

这里有一些基础的数据，加上一个只读的Description属性，将会返回所有的信息。OutLabel属性被连接到称为FLabel的一个本地私有字段。在组件代码中，我通过内部的FLabel引用来使用这个外部标签，如下所示：

```
procedure TmdPersonalData.UpdateLabel;
begin
  if Assigned (FLabel) then
    FLabel.Caption := Description;
end;
```

这种UpdateLabel方法每当其他属性改变时（读者可以在图9.5中看到设计时的情况）就会被触发，如下所示：

```
procedure TmdPersonalData.SetFirstName(const Value: string);
begin
  if FFirstName <> Value then
  begin
    FFirstName := Value;
    UpdateLabel;
  end;
end;
```

当然，如果没有分配，就不能使用标签；因此需要进行初始化测试。然而，这种测试并不能保证标签在被解除后不能使用（或者在运行时，或者在设计时）。当编写一个引用外部组件的组件时，需要在开发的组件中（使用外部引用的组件）覆盖Notification方法。当一个兄弟组件（拥有相同宿主）被创建或解除时，这种方法会被触发。考虑TmdPersonalData类的情况，将会收到Label组件的解除（opRemove）通知：

```

procedure TMdPersonalData.Notification(
  AComponent: TComponent; Operation: TOperation);
begin
  inherited;
  if (AComponent = FLabel) and (Operation = opRemove) then
    FLabel := nil;
end;

```

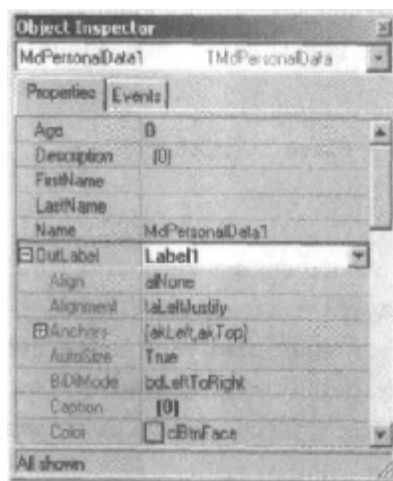


图9.5 一个组件在设计时引用外部标签

这个代码足以避免在相同的窗体或设计器中（如数据模块）由组件所带来的问题，这是因为当一个组件被解除时，它的宿主将会通知其拥有的所有其他组件（被解除组件的兄弟组件）。然而，为了说明连接到窗体或数据模块的组件，需要执行一个额外的步骤。每个组件都有一个内部的通知列表，列有它必须通知的一个或多个组件。而我们的组件可以通过调用它的**FreeNotification**方法将自己添加到其组件通知列表中（在本例中，是标签）。所以，即使外部引用的标签在一个不同的窗体上，它也将通过触发**Notification**方法来通知组件它被解除了（该方法已经过处理，并且不需要更新）：

```

procedure TMdPersonalData.SetLabel(const Value: TLabel);
begin
  if FLabel <> Value then
    begin
      FLabel := Value;
      if FLabel <> nil then
        begin
          UpdateLabel;
          FLabel.FreeNotification (Self);
        end;
    end;
end;

```

提示：也可以使用相反的通知（opInsert）来自动地将其关联，就像它们被添加到相同的窗体或设计器中一样。我并不能确定为什么这种技巧很少使用，这是因为在许多情况下它都是非常有用的。建立特定的属性和组件编辑器来支持设计时操作将会更有意义，而不是将特定的代码嵌入到组件中。

使用接口引用组件

当引用外部组件时，我们通常被限制到一个子层次上。例如，前一小节中创建的组件只能引用TLabel类或继承自TLabel的类的对象，尽管它也能将数据输出到其他组件中。Delphi 6添加了对一个有趣特性的支持，能有潜力对VCL的一些领域带来革命性的变化，这就是接口类型的组件引用。

说明：在Delphi 7中，这个特性被保守地使用。因为使用界面来更新Delphi的数据敏感组件结构可能是太晚了，所以我们只是希望在库中用它来表示将来其他的复杂关系。

如果组件支持一个给定的接口（即使它们并不是相同子层次的一部分），则可以使用一个接口类型来声明一个属性，并将任何这些组件分配给它。例如，假设有一个非可视组件附加在一个控件上以支持它的输出，与我在前一小节中提到的相似。当时我使用了一个传统的方法将一个组件关联到一个标签上，但是现在可以像下面这样定义一个接口：

```
type
  TMDViewer = interface
    ['{9766860D-8E4A-4254-9843-59B98FEE6C54}']
    procedure View (const str: string);
  end;
```

一个组件可以使用这个viewer接口来提供对另一个（任何类型的）控件的输出。程序清单9.2显示了如何声明一个组件，并使之使用这个接口应用一个外部组件。

清单9.2 使用接口引用外部组件的组件

```
type
  TMDIntfTest = class(TComponent)
  private
    FViewer: IViewer;
    FText: string;
    procedure SetViewer(const Value: IViewer);
    procedure SetText(const Value: string);
  protected
    procedure Notification(AComponent: TComponent;
      Operation: TOperation); override;
  published
    property Viewer: IViewer read FViewer write SetViewer;
    property Text: string read FText write SetText;
  end;

{ TMDIntfTest }

procedure TMDIntfTest.Notification(AComponent: TComponent;
  Operation: TOperation);
```

```

var
  intf: IMdViewer;
begin
  inherited;
  if (Operation = opRemove) and
    (Supports (AComponent, IMdViewer, intf)) and (intf = FViewer) then
  begin
    FViewer := nil;
  end;
end;

procedure TMDIntfTest.SetText(const Value: string);
begin
  FText := Value;
  if Assigned (FViewer) then
    FViewer.View(FText);
end;

procedure TMDIntfTest.SetViewer(const Value: IMdViewer);
var
  iComp: IInterfaceComponentReference;
begin
  if FViewer <> Value then
  begin
    FViewer := Value;
    FViewer.View(FText);
    if Supports (FViewer, IInterfaceComponentReference, iComp) then
      iComp.GetComponent.FreeNotification(Self);
  end;
end;

```

相对于引用外部组件的一个类类型的传统用法来说，接口的使用暗示出两个相关性的不同。首先，在Notification方法中，必须从组件中提取接口，作为一个参数传递，并与已经使用的接口对比。第二，为了调用FreeNotification方法，必须看到是否该对象作为参数传递，以支持IInterfaceComponentReference接口。这是在TComponent类中声明的，并提供了一种方式来引用组件（GetComponent），调用其方法。没有这种帮助，从对象中提取接口时，将不会有自动的方式来引用对象。

既然已经有了一个带有接口属性的组件，通过将其添加到IViewer接口并部署View方法，就可以将其分配给任何组件（从VCL层次的任何部分中）。这里是一个例子：

```

type
  TViewerLabel = class (TLabel, IViewer)
  public
    procedure View(str: String);
  end;

procedure TViewerLabel.View(const str: String);

```

```
begin
    Caption := str;
end;
```

使用框架建立复合组件

除了在代码中建立复合组件并手工连接计时器事件之外，还可以使用框架获得相似的效果。框架可以使复合组件（带有定制的事件处理程序）的开发变成可视操作，这样将大大简化相关过程。可以将该框架添加到存储库（Repository）中，并使用框架快捷菜单的Add to Palette命令建立一个模板，以实现对其的共享。

还有一种方法，亦即将框架放入一个组件包中，注册它为一个组件，以共享框架。从技术上讲，这并不困难：只需向框架的单元中添加一个Register过程，再将单元添加到一个组件包中，并建立它。新的组件/框架将出现在组件面板上，与其他任何组件一样。把该组件/窗体放置在一个窗体上时，将看到它的子组件。在窗体设计器中使用鼠标单击并不能来挑选这些子组件，但在对象TreeView上是可行的。然而，当运行程序或保存和重载窗体时，任何在设计时对组件所做的改变将不可避免地丢失。这是因为对那些子组件的改变将不被流化，相反放置在窗体内的标准框架将有事件发生。

如果这不是读者所期待的话，可以参考笔者发现的一个合理的方法，亦即在组件包中使用框架，该框架通过MdFramedClock组件说明（参见Sybex网站上本章中的部分示例）。该想法通过调用SetSubComponent对象方法把窗体拥有的组件变成实际的子组件。此外，笔者还将用属性展示内部组件（因为它们能在ObjectTreeView中被挑选，即使这一步骤并不是强制性的）。下面是该组件的声明及其对象方法的代码：

```
type
    TMdFramedClock = class(TFrame)
        Label1: TLabel;
        Timer1: TTimer;
        Bevel1: TBevel;
        procedure Timer1Timer(Sender: TObject);
    public
        constructor Create(AOwner: TComponent); override;
    published
        property SubLabel: TLabel read Label1;
        property SubTimer: TTimer read Timer1;
    end; constructor TMdFramedClock.Create(AOwner: TComponent);
begin
    inherited;
    Timer1.SetSubComponent (True);
    Label1.SetSubComponent (True);
end; procedure TMdFramedClock.Timer1Timer(Sender: TObject);
begin
    Label1.Caption := TimeToStr (Time);
end;
```


与先前建立的时钟组件相比较，不需要设置定时器的属性或手动地将定时器事件连接到与之相应的处理函数上。这是因为该过程是可视的，并且保存在框架的DFM文件中。也要注意，这里没有展示Bevel组件（没在它上调用SetSubComponent）。我故意这样做的目的是让读者能看到这种错误行为——试图在设计时编辑它，并会发现所有的修改结果都已消失了，如上面提到的。

在安装这个框架/组件之后，便可以在任何应用程序中使用它。在这里，只要把框架拖入窗体中，定时器就将启动并用当前时间更新标签。然而，此时我们仍能处理它的OnTime事件，并且Delphi IDE（确认组件在一个框架中）将用这种预定义的代码定义对象方法：

```
procedure TForm1.MdFramedClock1Timer1Timer(Sender: TObject);
begin
    MdFramedClock1.Timer1Timer(Sender);
end;
```

一旦该定时器被连接（甚至在设计时），活动锁就将停止，这是因为它的原始事件处理器并未连接。然而，在编译和运行程序后，原始行为将恢复（至少如果没有删除前面的行）；并且额外定制的代码将同时执行。这种行为正是框架可以完成的操作。读者能在FrameClock示例中看到该框架/组件使用的完整演示。

可以说这种方法仍然是远离线性的。它比Delphi以前的版本要好得多，因为在Delphi以前的版本中，框架在包中是不可用的。问题是，它值得努力吗？我想不。如果单独工作或小组团队工作，最好使用存储在Repository上的普通框架。在大公司或向大用户群分配程序时，大多数人还是愿意用传统方式建立他们的组件，而不是使用框架。我希望Borland公司会对基于框架的组件包可视开发给予更全面的支持。

复杂的图形组件

这里，笔者将介绍如何建立一个图形化的箭头组件——用户常常用这样一个组件来指示信息的一个流程或一步操作。因为这个组件是相当复杂的，故笔者将按照连续步骤来建立它，而不是直接向读者显示该组件的完整代码。向MdPack组件包添加的组件只是此过程的最终版本。在建立该组件的过程中，将熟悉一些重要的概念：

- 基于定制枚举数据类型的新型枚举属性的定义。
- 组件Paint对象方法的实现，该对象方法提供了组件的用户接口，并且是通用的，以适应不同属性的所有可能值，包括组件的Width与Height。Paint对象方法在该图形组件中起着一种重要的作用。
- TPersistent派生类属性的使用，例如TPen与TBrush和与之建立及解除相关的问题，以及组件内部对它们的OnChange事件处理相关的问题。
- 一种定制事件处理程序的组件定义，用于响应用户输入（在本例中，是双击箭头）。这需要直接对Windows消息进行处理并对图形区域使用Windows API函数。
- 对象检验器类别中属性的注册与定制类别的定义。

定义枚举属性

在使用组件向导创建新组件，并选择TGraphicControl作为父类之后，就可以开始定制组件了。箭头可以指向四个方向的任意一个：上、下、左、右。为了表示这些选择，定义一个枚举类型：

```
type
    TmdArrowDir = (adUp, adRight, adDown, adLeft);
```

这个枚举类型定义了组件的一个专用数据成员，以及用于改变其过程的一个参数与相应属性的类型。

ArrowHeight属性确定箭头头部的大小，Filled属性指定是否用颜色填充箭头：

```
type
    TmdArrow = class (TGraphicControl)
    private
        FDirection: TmdArrowDir;
        FArrowHeight: Integer;
        FFilled: Boolean;
        procedure SetDirection (Value: Tmd4ArrowDir);
        procedure SetArrowHeight (Value: Integer);
        procedure SetFilled (Value: Boolean);
    published
        property Width default 50;
        property Height default 20;
        property Direction: Tmd4ArrowDir
            read FDirection write SetDirection default adRight;
        property ArrowHeight: Integer
            read FArrowHeight write SetArrowHeight default 10;
        property Filled: Boolean read FFilled write SetFilled default False;
```

说明：一个图形化控件没有默认的大小，所以将其放置在一个窗体中时，它的大小是一个单个的像素。因此，在类构造器中为Width和Height属性添加一个默认值并将类字段设置为默认的属性值是非常重要的。

这三种定制属性可以从相应字段中直接读取，并通过三种set方法被编写，所有这些都具有相同的标准结构：

```
procedure TmdArrow.SetDirection (Value: TmdArrowDir);
begin
    if FDirection <> Value then
    begin
        FDirection := Value;
        ComputePoints;
        Invalidate;
    end;
end;
```

注意，只有当属性真正改变其值，并调用ComputePoints对象方法（它计算了箭头的三角形边界）之后，才可以要求系统重新绘制组件（调用Invalidate对象方法）。否则，会跳过这段代码，对象方法也会马上结束。该段代码的结构很常见，本章将使用它作为大多数属性的set过程。

必须记住，应在组件的构造器中设置属性的默认值：

```
constructor TMDArrow.Create (AOwner: TComponent);  
begin  
    // call the parent constructor  
    inherited Create (AOwner);  
    // set the default values  
    FDirection := adRight;  
    Width := 50;  
    Height := 20;  
    FArrowHeight := 10;  
    PFilled := False;
```

就像前面提到的那样，在属性声明中指定的默认值只用于确定是否向磁盘保存属性。Create构造程序在新组件类型定义的公共部分中定义，并由关键字override标记该构造器，就像它放置TComponent的虚拟Create构造器那样。牢记override标识符是非常重要的；否则，当Delphi创建一个该类的新组件时，将不会调用为其派生类编写的构造程序，而是调用基类的构造程序。

属性命名约定

在Arrow组件的定义中，注意对属性、访问对象方法以及字段的几个命名约定的使用。

下面对它们进行总结：

- 一个属性应该有一个有意义而且易读的名称。
- 当一个专用数据元素字段用于保存一个属性的值时，该字段名称应该以一个大写字母F (field) 打头，后面跟着相应属性的名称。
- 当一个过程被用于改变属性的值时，函数名称应以Set打头，后面跟上相应的属性名称。
- 用于读取属性的一个相应的函数应该以Get作为名称的前缀，后面跟上属性的名称。

这样做只是为了增强程序的可读性，编译器不会强迫人们这样做。这些约定在“Delphi 组件编写指南”中有述，且其后还有Delphi类的完整机制。

编写绘图对象方法

绘制各个方向和各种类型的箭头需要相当大量的代码。为了实现定制的绘图操作，只需要覆盖Paint对象方法并使用受保护的Canvas属性即可。

这里没有在频繁执行的绘制代码中计算箭头顶点的位置，而是编写了一个专门的函数来计算箭头区，并将它存储在一个点数组中，该数组定义在组件的专用字段中，如下所示：

```
FArrowPoints: array [0..3] of TPoint;
```

这些点通过ComputePoints专有方法来确定, 这种方法每当一个组件属性改变时就会被调用。这里是其代码的一个摘录:

```

procedure TMDArrow.ComputePoints;
var
    XCenter, YCenter: Integer;
begin
    // compute the points of the arrowhead
    YCenter := (Height - 1) div 2;
    XCenter := (Width - 1) div 2;
    case FDirection of
        adUp: begin
            FArrowPoints [0] := Point (0, FArrowHeight);
            FArrowPoints [1] := Point (XCenter, 0);
            FArrowPoints [2] := Point (Width-1, FArrowHeight);
        end;
    // and so on for the other directions

```

该代码计算了组件区域的中心(只需用Height与Width属性除以2), 然后使用该中心来确定箭头的位置。注意, 除了改变方向或其他属性外, 当组件大小改变时, 还需要更新箭头头部的位置。可以覆盖组件的SetBounds对象方法, 每当一个组件的Left、Top、Width和Height属性改变时, VCL会调用它:

```

procedure TMDArrow.SetBounds(ALeft, ATop, AWidth, AHeight: Integer);
begin
    inherited SetBounds (ALeft, ATop, AWidth, AHeight);
    ComputePoints;
end;

```

一旦组件知道了箭头头部的位置, 它的绘制代码就变得非常简单了。这里是Paint方法的一个摘录:

```

procedure TMDArrow.Paint;
var
    XCenter, YCenter: Integer;
begin
    // compute the center
    YCenter := (Height - 1) div 2;
    XCenter := (Width - 1) div 2;
    // draw the arrow line
    case FDirection of
        adUp: begin
            Canvas.MoveTo (XCenter, Height-1);
            Canvas.LineTo (XCenter, FArrowHeight);
        end;
    // and so on for the other directions
end;

```

```

// draw the arrow point, eventually filling it
if FFilled then
  Canvas.Polygon (FArrowPoints)
else
  Canvas.PolyLine (FArrowPoints);
end;

```

读者可以在图9.6中看到这个组件输出的一个范例。

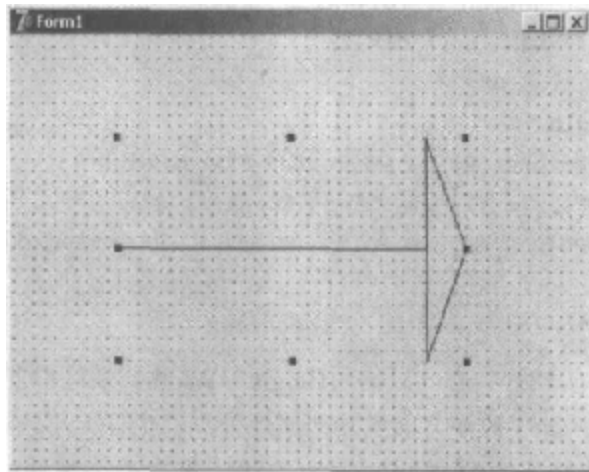


图9.6 箭头组件的输出

添加TPersistent属性

为了使组件的输出更加灵活，向它添加了两个新的属性，**Pen**和**Brush**，并使用一种数据类型（一种**TPersistent**数据类型，其定义的对象可以自动由Delphi简化）来定义。这些属性处理起来稍显复杂，因为组件现在必须建立或解除这些内部对象。然而，这一次，还使用一些属性输出了内部对象，以便用户可以直接在对象检验器上改变这些内部对象。当这些子对象改变时，为了更新组件，还需要处理它们的内部**OnChange**属性。下面是**Pen**属性的定义以及其他对组件类的定义的改变（**Brush**属性的代码是相似的）：

```

type
  TMDArrow = class (TGraphicControl)
  private
    FPen: TPen;
    ...
    procedure SetPen (Value: TPen);
    procedure RepaintRequest (Sender: TObject);
  published
    property Pen: TPen read FPen write SetPen;
  end;

```

首先在构造器创建对象，并设置它的**OnChange**事件处理器：

```

constructor TMDArrow.Create (AOwner: TComponent);
begin
  ...

```

```

    // create the pen and the brush
    FPen := TPen.Create;
    // set a handler for the OnChange event
    FPen.OnChange := RepaintRequest;
end;

```

这些OnChange事件当画笔的任何一个属性发生改变时就会被触发，所有我们必须做的就是要求系统重新绘制组件：

```

procedure TmdArrow.RepaintRequest (Sender: TObject);
begin
    Invalidate;
end;

```

必须向组件中添加一个解除器，以便从内存中删除图形化的对象（并释放它的系统资源）。所有解除器所要做的就是调用Pen对象的Free方法。

与持续对象相关的一个属性要求特定的处理：必须复制作为一个参数传递的对象的内部数据，而不是将指针复制到对象中。标准的:=操作将复制指针，所以在这种情况下，必须使用Assign方法：

```

procedure TmdArrow.SetPen (Value: TPen);
begin
    FPen.Assign(Value);
    Invalidate;
end;

```

许多TPersistent类有一个Assign方法，当需要更新这些对象的数据时，可使用这个方法。现在，为了使用画笔来绘制，在画一根横线之前，必须修改Paint方法，设置组件Canvas的相应属性为内部对象的值（参见图9.7中组件的新输出范例）：

```

procedure TmdArrow.Paint;
begin
    // use the current pen
    Canvas.Pen := FPen;

```

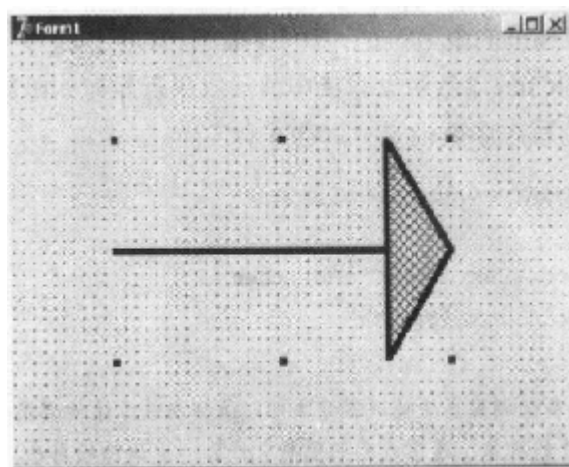


图9.7 箭头组件使用粗画笔和特殊阴影画刷进行输出

就像Canvas使用一个setter例程来Assign画笔对象一样，不仅要将一个对画笔的引用保存到Canvas的一个字段中，而且还要复制所有数据。这就是说，可以自由地解除本地Pen对象（FPen），而且修改FPen将不会影响Canvas，除非Paint被调用，以及上面的代码被再次执行。

定义定制的事件

为了完成对Arrow组件的开发，让我们添加一个定制的事件。大多数时间，新组件使用其父类的事件，例如，在这个组件中，只需要通过在类的公共部分重新声明一些标准事件就可以将它们变为可用的：

```
type
  TMDArrow = class (TGraphicControl)
  published
    property OnClick;
    property OnDragDrop;
    property OnDragOver;
    property OnEndDrag;
```

由于有了这种声明，当安装组件时，在对象检验器中就可以使用上面的事件（最初声明在一个父类中）。

然而有些时候，一个组件需有一个定制事件。为了定义一个全新的事件，首先需要确保已经有一个方法指针类型，适合事件所使用。这种类型是一个方法指针类型（参见第5章“可视控件”，可以看到更详细的信息）。在这两种情况下，需要向类中添加事件类型的一个字段。下面是在TMDArrow类的专用部分中添加的定义：

```
FArrowDbClick: TNotifyEvent;
```

在这里，我使用了TNotifyEvent类型，它只有一个Sender参数，并可用于Delphi的多个事件，包括OnClick与OnDbClick事件。使用该字段，通过直接访问它，定义了一个非常简单的公用属性：

```
property OnArrowDbClick: TNotifyEvent
  read FArrowDbClick write FArrowDbClick;
```

（需要再一次注意标准的命名约定，事件名称应以On打头。）fArrowDbClick对象方法指针在特定的ArrowDbClick动态对象方法中激活（执行相应的函数）。但只有一个事件处理程序在使用此组件的程序中被指定时，这才会发生：

```
procedure TMDArrow.ArrowDbClick;
begin
  if Assigned (FArrowDbClick) then
    FArrowDbClick (Self);
end;
```

提示：使用Self作为参数调用事件处理器方法可以保证当该方法被调用时，它的Sender参数将实际上引用触发事件的对象，通常我们期望将它作为一个组件的用户。

使用低层次的Windows API调用

定义在类型定义的受保护部分的fArrowDbClick方法允许将来继承类调用并修改它。基本上,这种方法被wm_LButtonDown Windows消息的处理器调用,但是只有当双击发生在箭头点的内部时才会出现。为了测试这个条件,可以使用一些Windows API区域函数。

说明: 一个区域是被任何图形封闭起来的一个屏幕区域。例如,可以通过使用三角形的三个点建立一个多边形。惟一的问题是,要正确地填充这个表面,必须以顺时针方向定义一个TPoints数组(参见Windows API帮助文件中对CreatePolygonalRgn的描述,了解这种奇怪方法的具体信息)。这就是我在ComputePoints方法中所做的。

一旦定义了一个区域,就可以使用PtInRegion API调用来测试双击点是否发生在区域内。这个过程的全部代码如下所示:

```
procedure TMDArrow.WMLButtonDownDbClick (
  var Msg: TWMLButtonDownDbClick); // message wm_LButtonDownDbClick;
var
  HRegion: HRgn;
begin
  // perform default handling
  inherited;

  // compute the arrowhead region
  HRegion := CreatePolygonRgn (FArrowPoints, 3, WINDING);
  try // check whether the click took place in the region
    if PtInRegion (HRegion, Msg.XPos, Msg.YPos) then
      ArrowDbClick;
  finally
    DeleteObject (HRegion);
  end;
end;
```

CLX版本: 调用Qt本身的函数

前面的代码并不适用于Linux, 并且对组件的CLX/Qt版本也没有任何意义。如果想要为CLX类库建立一个相似的组件, 可以使用直接(低层次)调用Qt层来取代Win32 API调用, 创建QRegion类的一个对象, 如下面的程序清单所示:

```
procedure TMDArrow.DbClick;
var
  HRegion: QRegionH;
  MousePoint: TPoint;
begin
  // perform default handling
  inherited;

  // compute the arrow head region
  HRegion := QRegion_create (PPointArray(FArrowPoints), True);
  try
    // get the current mouse position
```



```

    GetCursorPos (MousePoint);
    MousePoint := ScreenToClient(MousePoint);
    // check whether the click took place in the region
    if QRegion_contains(HRegion, PPoint(@MousePoint)) then
        ArrowDbClick;
    finally
        QRegion_destroy(HRegion);
    end;
end;

```

注册属性类别

我们向这个组件添加了一些定制的属性与一个新事件。如果在对象检验器中按照类别排列属性，那么所有的新元素都会显示在Miscellaneous类别中。当然，这样并不理想，但可以很容易地在一个可以使用的类别中注册新属性。

可以通过调用RegisterPropertyInCategory函数（定义在DesignIntf单元中）的四个重载版本之一，在一个类别中注册属性（或事件）。在调用此函数时，要指明类别名称，并指定属性名称、类型或属性名称及属于它的组件。例如，向单元的Register过程中添加以下代码，在Input类别中注册OnArrowDbClick事件，在Visual类别中注册Filled属性：

```

uses
    DesignIntf;

procedure Register;
begin
    RegisterPropertyInCategory ('Input', TMDArrow, 'OnArrowDbClick');
    RegisterPropertyInCategory ('Visual', TMDArrow, 'Filled');
end;

```

第一个参数是表示类别名称的一个字符串——相对于最初Delphi 5中使用类别类的方法，这是一个非常简单的解决方案。可以通过简单地把它名字作为RegisterPropertyInCategory函数的第一个参数，以一种简单的方式定义一个新的类别：

```

RegisterPropertyInCategory ('Arrow', TMDArrow, 'Direction');
RegisterPropertyInCategory ('Arrow', TMDArrow, 'ArrowHeight');

```

也可对组件的特定属性创建一个新的类别，以使用户定位其指定特征的操作更简单。注意，虽然使用DesignIntf单元，但应在设计时的组件包而不是运行时的组件包中包括这些注册在内的单元（事实上，请求的DesignIde单元不能被分配）。由于这个原因，应在一个独立单元内编写该代码（包括所有设计时单元），而不是在定义组件的单元中编写代码，并向MdDesPk页添加新的MdArrReg单元。这将在后面的“安装属性编辑器”小节介绍。

警告：为一个组件的特定属性使用一个类别是否是好的想法，还有待讨论。一方面，组件的一个用户可以轻易地发现特定属性。同时，一些新属性可能不属于任何已有类别。然而另一方面，类别可能会被滥用。如果每个组件都引入新类别，用户可能会感到困惑。还可能要面对类别与属性一样多的风险。

注意, **Filled**属性已经注册在两个不同类别中了。这并不是一个问题, 因为相同属性可以在对象检验器的不同类别下显示多次, 如图9.8所示。为了测试**Arrow**组件, 笔者编写了一个非常简单的范例程序, **ArrowDemo**, 它允许在运行时改动该组件的大部分属性。这种测试在编写了一个组件之后, 或正在编写组件时, 是非常重要的。

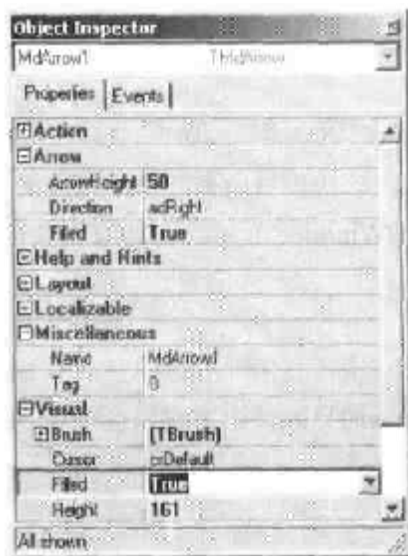


图9.8 **Arrow**组件定义了一个定制属性类别, **Arrow**, 在对象检验器中可看到它。注意, 该属性在多个部分均可见, 诸如本例中的**Filled**属性

说明: **Localizable**属性种类有一个与ITE (集成翻译环境) 有关的特殊功能。当一个属性是该种类的一部分时, 它的值将作为一个能译成其他语言的属性而列在ITE中。

定制Windows控件

定制已有组件最常用的方法之一就是向它们的事件处理程序添加一些预定义性能。每当需要将同一个事件处理程序配属给不同窗体的组件时, 应该考虑将事件的代码直接添加到组件的子类中。一个明显的例子是, 编辑框的子类只接收数字输入。不用为每个组件都配属一个通用的**OnChar**事件处理程序, 程序员可以定义一个简单的新组件。

然而, 这种组件将不会处理事件; 事件只用于组件的用户。相反, 组件可以直接处理**Windows**消息, 或覆盖一个对象方法, 通常称为一个二级消息处理器。前面一种技术通常在过去使用, 但是它专门用于**Windows**平台的组件。为了创建一个适用于**CLX**和**Linux**的组件——并且, 在将来, 用于**.NET**结构——应该避免使用低层次的**Windows**消息, 而应当覆盖基组件和控件类的虚拟方法。

说明: 当大多数**VCL**组件处理一个**Windows**消息时, 它们会调用一个二级消息处理器 (通常是一个动态方法), 而不是在消息响应方法中直接地执行代码。这种方法更易于程序员在一个派生类中定制组件。典型地, 一个二级处理器将会做它自己的工作, 并调用组件用户分配的任何事件处理器。所以, 应当总是调用**inherited**, 以使组件像期望的那样触发事件。

为什么覆盖现有的二级处理器通常比直接处理**Windows**消息的方法要好些? 除了适用性之外, 还有一些其他的原因。首先, 这种技术从面向对象的角度来说更加正确。程序员可以

覆盖VCL设计器计划的一个虚拟方法调用，而不是从基类中复制消息响应代码并定制它。其次，如果某些人需要从你的组件类中派生另一个类，你应当尽可能地使它们更容易被定制，并且覆盖二级处理器并不容易导致错误（只是因为你编写了更少的代码）。例如，我通过处理wm_Char系统消息，编写了下列数字编辑框控件：

```
type
  TMDNumEdit = class (TCustomEdit)
  public
    procedure WmChar (var Msg: TWmChar); message wm_Char;
```

然而，如果覆盖KeyPress方法，该代码将会更方便，就像我在下一个组件中所做的那样（在稍后的例子中，将处理定制的Windows消息，这是因为没有相应的用于覆盖的方法）。

数字编辑框

为了定制一个编辑框组件来限制它接收的输入，所需做的就是覆盖它的KeyPress方法，当组件收到wm_Char Windows消息时，该方法将会被调用。这里是用于TMDNumEdit类的代码：

```
type
  TMDNumEdit = class (TCustomEdit)
  private
    FInputError: TNotifyEvent;
  protected
    function GetValue: Integer;
    procedure SetValue (Value: Integer);
    procedure KeyPress(var Key: Char); override;
  public
    constructor Create (Owner: TComponent); override;
  published
    property OnInputError: TNotifyEvent read FInputError write FInputError;
    property Value: Integer read GetValue write SetValue default 0;
    property AutoSelect;
    property AutoSize;
    // and so on...
```

这个组件继承自TCustomEdit而不是TEdit，所以它可以隐藏Text属性而显示Integer Value属性。注意，这里并没有创建一个新字段以保存这个值，因为可以使用已有（但现在没有公布）的Text属性。为此，只需将数字值与文本字符串相互转换即可。TCustomEdit类（或它实际包装的Windows控件）会在组件表面上自动绘制Text属性的信息：

```
function TMDNumEdit.GetValue: Integer;
begin
  // set to 0 in case of error
  Result := StrToIntDef (Text, 0);
end;
```

```
procedure TMDNumEdit.SetValue (Value: Integer);  
begin  
    Text := IntToStr (Value);  
end;
```

最重要的对象方法是对KeyPress方法的重定义,该方法会滤出所有的非数字字符,并在出现错误时触发一个特殊的事件:

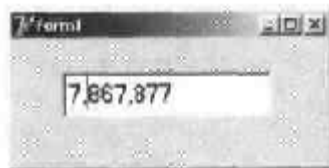
```
procedure TMDNumEdit.KeyPress (var Msg: TWMChar);  
begin  
    if not (Key in ['0'..'9']) and not (Key = #8) then  
    begin  
        Key := #0; // pretend that nothing was pressed  
        if Assigned (FInputError) then  
            FInputError (Self);  
    end  
    else  
        inherited;  
end;
```

这种对象方法要检测用户输入的每个字符,检查数字键与空格键(ASCII值为8)。除了系统键(箭头键与Del键)之外,用户还应该可以使用空格键,所以也需要检查该值。

现在,如果在一个窗体上放置这个组件,则可以在编辑框中键入一些值并查看它的行为。还可以将一个对象方法配属给OnInputError事件,当输入一个错误键时,会为用户提供反馈消息。

使用千位分隔符进行数字编辑

作为该示例的进一步扩展,当用户键入大量数字时(作为浮点数保存在内部,相对于整数来说可能会更大,并带有小数位),最好能自动显示千位分隔符,并像用户输入所要求的那样更新它们。



此外,也可以通过覆盖内部Change对象方法并正确设置数字的格式来完成该功能。只有两个小问题需要考虑。第一个是为了格式化数字,需要有一个字符串,其中包含一个数字,但编辑器中的文本并不是一个Delphi能认出的数字字符串,这是因为它有千位分隔符,并且不能直接转换为一个数字。为了举例说明,我编写了一个StringToFloat功能的修改版本,称为StringToFloatSkipping,用以完成这个转换。

第二个小问题是如果修改编辑框中的文本,则鼠标当前位置将会丢失。因此,需要保存最初的鼠标位置,重新格式化数字;如果一个分隔符已被添加或删除,则鼠标位置将会相应地改变,所以应考虑重新应用鼠标位置。

通过下面这些用于TMdThousandEdit类的全部代码，将所有这些想法总结一下：

```

type
  TMdThousandEdit = class (TMdNumEdit)
  public
    procedure Change; override;
  end;

function StringToFloatSkipping (s: string): Extended;
var
  s1: string;
  l: Integer;
begin
  // remove non-numbers
  s1 := '';
  for i := 1 to length (s) do
    if s[i] in ['0'..'9'] then
      s1 := s1 + s[i];
  Result := StrToFloat (s1);
end;

procedure TMdThousandEdit.Change;
var
  CursorPos, // original position of the cursor
  LengthDiff: Integer; // number of new separators (+ or -)
begin
  if Assigned (Parent) then
  begin
    CursorPos := SelStart;
    LengthDiff := Length (Text);
    Text := FormatFloat ('#,###',
      StringToFloatSkipping (Text));
    LengthDiff := Length (Text) - LengthDiff;
    // move the cursor to the proper position
    SelStart := CursorPos + LengthDiff;
  end;
  inherited;
end;

```

声音按钮

下一个组件，TMdSoundButton，可以在按下按钮时发出一种声音，当松开按钮时发出另一种声音。用户通过改动两个字符串属性（为各自的声音指定相应的WAV文件）可指定每一种声音。而程序员需要截取一些系统消息（wm_LButtonDown和wm_LButtonUp），或者覆盖相应的二级处理程序。

下面是用于TMdSoundButton类的代码，其中有两个受保护对象方法以及两个确定声音文件的字符串属性。在属性声明中，读写相应的专用字段，这是因为当用户改变这些属性时，

不需要做任何特别的操作:

```

type
  TMdSoundButton = class(TButton)
  private
    FSoundUp, FSoundDown: string;
  protected
    procedure MouseDown(Button: TMouseButton;
      Shift: TShiftState; X, Y: Integer); override;
    procedure MouseUp(Button: TMouseButton;
      Shift: TShiftState; X, Y: Integer); override;
  published
    property SoundUp: string read FSoundUp write FSoundUp;
    property SoundDown: string read FSoundDown write FSoundDown;
  end;

```

这里是用于两种方法之一的代码:

```

uses
  MMSystem;

procedure TMdSoundButton.MouseDown(Button: TMouseButton; Shift: TShiftState;
  X, Y: Integer);
begin
  inherited MouseDown (Button, Shift, X, Y);
  PlaySound (PChar (FSoundDown), 0, snd_Async);
end;

```

读者会注意到, 在做其他事情之前, 先调用了对象方法的继承版本。对于大多数二级处理程序来说, 这是一种好的习惯, 因为它确保了在执行任何定制操作之前, 执行标准操作。然后, 调用了PlaySound Win32 API函数来播放声音。可以使用这个函数(在MmSystem单元中定义)来播放WAV文件或系统声音, 就像SoundB范例所演示的那样。下面是该范例程序窗体的文本描述(摘自DFM文件):

```

object MdSoundButton1: TMdSoundButton
  Caption = 'Press'
  SoundUp = 'RestoreUp'
  SoundDown = 'RestoreDown'
end

```

说明: 为这些声音属性选择一个合适的值并不简单。本章稍后会向读者介绍如何向组件中添加一个属性编辑器, 以简化这种操作。

处理内部消息: Active按钮

Windows界面正向一种新的标准进化, 包括当鼠标移到组件上时, 会突出显示它们。Delphi为其内置的许多组件提供了相似的支持, 但对于一个简单的按钮组件来说, 如何模仿该特性呢? 这看起来可能是一个复杂的任务, 但其实不然。一旦你知道了哪个虚拟函数用来覆盖或哪个消息用来连接的话, 开发组件可能会变得非常简单。

下一个组件，**TMdActiveButton**类，通过处理一些内部Delphi消息以一种非常简单的方法实现了该任务（如果想了解这些内部Delphi消息出自何处的相关内容，可阅读随后的“组件信息和通告”小节）。ActiveButton组件用于处理**cm_MouseEnter**和**cm_MouseExit**内部Delphi消息，当鼠标进入或离开组件的相应区域时，会接受到这两条内部Delphi消息：

```
type
  TMdActiveButton = class (TButton)
  protected
    procedure MouseEnter (var Msg: TMessage); message cm_mouseEnter;
    procedure MouseLeave (var Msg: TMessage); message cm_mouseLeave;
  end;
```

为这两个对象方法编写的代码可以实现预期的目的。对于该范例，我决定只切换按钮字体的粗体样式。读者可以在图9.9中看到鼠标在这些组件上移动的效果。

```
procedure TMdActiveButton.MouseEnter (var Msg: TMessage);
begin
  Font.Style := Font.Style + [fsBold];
end;

procedure TMdActiveButton.MouseLeave (var Msg: TMessage);
begin
  Font.Style := Font.Style - [fsBold];
end;
```

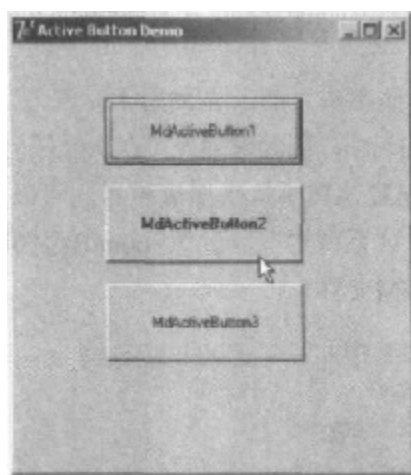


图9.9 ActiveButton组件的应用举例

可以根据需要添加其他效果，包括扩大字体，使按钮处于默认状态，或稍微增加它的大小等。最好的效果通常涉及颜色，但必须从**TBitBtn**类中获得这种支持（**TButton**控件有固定的颜色）。

组件信息和通告

为了建立ActiveButton组件，我使用了两个有**cm**前缀的Delphi内部组件。如示例强调的，这些信息是十分有趣的，但它们未被Borland公司正式证实。此外还有第二组内部Delphi信息

作为组件通告信息，并由前缀cm区别。本书没有足够的空间介绍它们中包含的每一个项目或提供一个详细的分析；如果读者想了解更多的知识，可以浏览VCL源代码。

警告：因为这是一个相对高级的主题，所以读者如果是编写Delphi组件的新手，可以略过这一部分。但Delphi帮助文件没有提供与组件信息有关的文档，因此在这里介绍它们十分重要。

组件信息

一个Delphi组件通过向其他组件传递组件消息来显示该状态的任何变化，这些将影响其他组件。这些消息的大部分由Window信息开始，但它们中的一些更复杂，例如高级翻译和复杂的重变换。另外，组件发送它们自己的信息与转发从Windows接受的信息一样。例如，改变一个属性值或组件的一些其他属性与告诉其他组件有关的变化都是十分必要的。

可以将这些消息分为几类：

- 激活和输入焦点消息，发送到被激活或无效的组件中——取决于接收或丢失输入焦点：

cm_Activate	对应于窗体和应用程序的OnActivate事件
cm_Deactivate	对应于OnDeactivate
cm_Enter	对应于OnEnter
cm_Exit	对应于OnExit
cm_FocusChanged	无论何时，当焦点在相同窗体的组件之间改变时发送（稍后，将看到使用这种消息的例子）
cm_GotFocus	已声明但没有使用
cm_LostFocus	已声明但没有使用

- 当一个属性改变时发送到子组件中的消息：

cm_BiDiModeChanged	cm_IconChanged
cm_BorderChanged	cm_ShowHintChanged
cm_ColorChanged	cm_ShowingChanged
cm_Ctl3DChanged	cm_SysFontChanged
cm_CursorChanged	cm_TabStopChanged
cm_EnabledChanged	cm_TextChanged
cm_FontChanged	cm_VisibleChanged

监控这些消息有助于跟踪在一个属性中发生的改变。或许你需要响应在一个新组件中的这些信息，但是这并不可能。

- 与ParentXxx属性相关的消息：cm_ParentFontChanged、cm_ParentColorChanged、cm_ParentCtl3DChanged、cm_ParentBiDiModeChanged和cm_ParentShowHintChanged。这些均与以前组中的消息相似。
- 在Windows系统中改变的通知：cm_SysColorChange、cm_WinIniChange、cm_TimeChange和cm_FontChange。只有在需要跟踪系统颜色或字体的特殊组件中，处理这些消息才是非常有用的。
- 鼠标消息：cm_Drag在拖拽操作期间被发送多次。cm_MouseEnter和cm_MouseLeave在光标输入或离开其表面时，会被发送给控件，但是它们被Application对象作为低层次消息发送。cm_MouseWheel对应于wheel_based操作。

应用程序消息:

<code>cm_AppKeyDown</code>	发送给Application对象, 以使其确定是否一个键对应于一个菜单快捷键
<code>cm_AppSysCommand</code>	对应于 <code>wm_SysCommand</code> 消息
<code>cm_DialogHandle</code>	在一个DLL中发送, 以提取DialogHandle属性的值 (被一些对话框使用, 但不是Delphi内置的)
<code>cm_InvokeHelp</code>	被一个DLL中的代码发送, 以调用InvokeHelp方法
<code>cm_WindowHook</code>	在一个DLL中发送, 以调用HookMainWindow和UnhookMainWindow方法

我们已经很少需要使用这些消息了。还有一个`cm_HintShowPause`消息, 在VCL中从来没有被处理过。

• Delphi内部消息:

<code>cm_CancelMode</code>	终止特定操作, 如显示一个组合框的下拉列表
<code>cm_ControlChange</code>	在添加或删除一个子控件之前发送给每个控件 (由公共控件处理)
<code>cm_ControlListChange</code>	在添加或删除一个子控件之前发送给每个控件 (由DBCtrlGrid组件处理)
<code>cm_DesignHitTest</code>	确定一个鼠标操作应该针对组件还是窗体设计器
<code>cm_HintShow</code>	显示提示之前发送给一个控件 (只有当ShowHint属性为True时)
<code>cm_HitTest</code>	当父控件试图在给定的鼠标位置定位一个子控件时发送给一个控件
<code>cm_MenuChanged</code>	在MDI或OLE菜单合并操作之后发送

• 与特殊键有关的消息:

<code>cm_ChildKey</code>	发送给父控件, 以处理一些特殊的键 (在Delphi中, 这个消息只能被DBCtrlGrid组件处理)
<code>cm_DialogChar</code>	发送给一个控件, 以确定是否一个给定的输入键是它的加速器字符
<code>cm_DialogKey</code>	由需要执行特殊操作的模式窗体和控件所处理
<code>cm_IsShortCut</code>	当前没有使用 (因为大部分代码只是调用IsShortCut), 但是它设计用来确定是否一个快捷键被一个窗体支持, 通过OnShortCut事件, 一个菜单条目或一个行动
<code>cm_WantSpecialKey</code>	由控件处理, 以一种不常见的方式来截取特殊键 (例如, 使用Tab键导航, 就像一些Grid组件那样)

• 用于特定组件的消息:

<code>cm_GetDataLink</code>	由DBCtrlGrid控件使用 (在第17章“编写数据库组件”中讨论)
<code>cm_TabFontChanged</code>	由TabbedNotebook组件使用

- | | |
|-------------------------------|---|
| <code>cm_ButtonPressed</code> | 由SpeedButtons使用, 以通知其他的兄弟SpeedButton组件(增强radio-button的行为) |
| <code>cm_DeferLayout</code> | 由DBGrid组件使用 |
- OLE容器消息: `cm_DocWindowActivate`, `cm_IsToolControl`, `cm_Release`, `cm_UIActivate`和`cm_UIDeactivate`。
 - 与锁定相关的消息, 包括`cm_DockClient`, `cm_DockNotification`, `cmFloat`和`cm_UndockClient`。
 - 方法实现消息: 如`cm_RecreateWnd`, 由TControl的RecreateWnd方法调用; `cm_Invalidate`, 由TControl.Invalidate调用; `cm_Changed`, 被TControl.Changed调用; 以及`cm_AllChildrenFlipped`, 由TWinControl和TScrollingWinControl的DoFlipChildren方法调用。在相似的组中, 还有两个与列表相关的动作消息: `cm_ActionUpdate`和`cm_ActionExecute`。

组件通告

组件通告消息是那些由一个父类窗体或组件发送给其子类的消息。这些通告对应于Windows向其父控件窗口发送的消息, 但逻辑上只用于子控件。例如, 按钮、编辑或列表框等控件间的相互作用能引起Windows向父控件发送一个`wm_Command`信息。当一个Delphi程序收到这些信息时, 它将这些消息转发到控件自身, 就像一个通告一样。Delphi控件能处理消息并最终触发一个事件。相似的分派操作发生在许多其他消息中。

Windows消息和组件通告消息之间的连接很紧密, 以至于大家通常从通告消息的名字中就可认出Windows消息的名字, 并用`wm`简单地代替最初使用的`cn`。下面是几组不同的组件通告消息:

- 普通键盘消息: `cn_Char`, `cn_KeyUp`, `cn_KeyDown`, `cn_SysChar`和`cn_SysKeyDown`。
 - 特殊的键盘消息, 只被`lbs_WantKeyboardInput`风格的列表框使用: `cn_CharToItem`和`cn_VKeyToItem`。
 - 有关宿主拖动技巧的消息: `cn_CompareItem`, `cn_DeleteItem`, `cn_DrawItem`和`cn_MeasureItem`。
 - 滚动消息, 只被滚动栏和追踪栏控件使用: `cn_HScroll`和`cn_VScroll`。
 - 普通的通告消息, 为多数控件使用: `cn_Command`, `cn_Notify`和`cn_ParentNotify`。
 - 有关控件颜色的消息: `cn_CtlColorBtn`, `cn_CtlColorDlg`, `cn_CtlColorEdit`, `cn_CtlColorListbox`, `cn_CtlColorMsgbox`, `cn_CtlColorScrollbar`和`cn_CtlColorStatic`。
- 其他的控件通告定义用于普通的控件支持(在ComCtrls单元)。

一个组件消息示例

作为一个十分简单的使用组件消息的示例, 本章编写了CMNTest程序。该程序含有一个含有三个编辑框的窗体并与标签相关联。它处理的第一个消息是`cm_DialogyKey`, 允许它像处理Tab键一样处理输入键。该对象方法的代码用于检查输入键的代码并发送相同的消息, 但传递`vk_Tab`键代码。为了停止输入键的进一步操作, 我们将该消息的结果设置为1:

```
procedure TForm1.CMDialogKey(var Message: TCMDialogKey);
begin
  if (Message.CharCode = VK_RETURN) then
  begin
    Perform (CM_DialogKey, VK_TAB, 0);
    Message.Result := 1;
  end
  else
    inherited;
end;
```

第二个消息，`cm_DialogChar`，用于监控加速器键。这种技巧在没有定义一个额外的菜单时，对于提供定制的快捷键非常有用的。注意，尽管这个代码对于一个组件是正确的，但是在一个正常的应用程序中，通过处理窗体的`OnShortCut`事件可能更容易获得相同的效果。在这种情况下，可以在一个标签中记录特殊的键：

```
procedure TForm1.CMDialogChar(var Msg: TCMDialogChar);
begin
  Label1.Caption := Label1.Caption + Char (Msg.CharCode);
  inherited;
end;
```

最后，窗体会处理`cm_FocusChanged`消息，以响应焦点变化，而不必处理其每个组件的`OnEnter`事件。再次强调的是，这个动作将显示对焦点组件的描述：

```
procedure TForm1.CmFocusChanged(var Msg: TCmFocusChanged);
begin
  Label5.Caption := 'Focus on ' + Msg.Sender.Name;
end;
```

这种方法的优点是它独立于添加到窗体上的组件的类型和数量，并且在这部分没有任何其他的特殊动作。再者，对于这样高级的主题来说，这是一个琐碎的示例，但如果向`ActiveButton`组件添加该代码，那么至少有几个原因需要我们注意这些特殊的、未正式说明的信息。有时，编写没有它们支持的相同代码很复杂。

组件中的对话框

将创建的最后一个组件与读者看到的前面建立的组件完全不同。在建立了基于窗口的控件以及简单的图形组件之后，现在将介绍如何建立一个非可视组件。

基本思想就是，窗体就是组件。当建立一个可能在很多项目中都格外有用的窗体时，可以将它添加到对象存储库中，或将它制作成一个组件。第二个方法比第一个方法更复杂，但它使新窗体的使用更容易了，而且它还允许我们不使用窗体源代码来分布窗体。作为一个范例，本章建立了一个基于定制对话框的组件，并尽可能地模仿标准Delphi对话框组件的功能。

在组件中建立一个对话框的第一步是,使用标准的Delphi方式来编写对话框自身的代码。只需定义一个新窗体并像往常一样对它进行加工即可。当组件基于窗体时,我们几乎可以可视地设计组件。当然,一旦建立对话框的工作完成之后,必须按非可视的方式以它为核心定义一个组件。

在这个例子中,我们想建立的标准对话框基于一个列表框,因为允许用户从字符串列表中选择值是很常见的。这里在对话框中定制了一个通用行为,然后使用它建立一个组件。我们建立的简单ListBoxForm窗体含有一个列表框与典型的OK与Cancel按钮,如下列文本描述所示:

```
object MdListBoxForm: TMdListBoxForm
  BorderStyle = bsDialog
  Caption = 'ListBoxForm'
  object ListBox1: TListBox
    OnDblClick = ListBox1DblClick
  end
  object BitBtn1: TBitBtn
    Kind = bkOK
  end
  object BitBtn2: TBitBtn
    Kind = bkCancel
  end
end
```

该列表框窗体惟一的对象方法与列表框的双击事件有关,它可以像用户单击OK按钮一样来关闭对话框,只需设置窗体的ModalResult属性为mrOk即可。一旦该窗体正常工作,就可以开始改变它的源代码,添加一个组件的定义,删除对窗体全局变量的声明。

说明: 对于基于窗体的组件来说,可以使用两个Pascal源代码文件:一个对应窗体,另一个对应封装窗体的组件。而在这个范例中,我们决定将窗体与组件放置在一个单元中。从理论上讲,最好是在该单元的`执行部分`中声明窗体类,还可以将它隐藏起来使组件的用户看不到。但实际上,这不是什么好主意。为了在窗体设计器中可视地处理窗体,窗体类声明必须出现在单元的`接口部分`中。Delphi IDE的这种行为的基本原理是,该约束会最小化模块管理器必须扫描的代码量(为了寻找窗体声明),该操作必须经常执行,以维持可视窗体与窗体类定义的同步。

这些操作中最重要的是对TMDListBoxDialog组件的定义,而且将其做为一个非可视组件,这是因为它的直接父类是TComponent。此组件有三个published属性与一个public属性:

- Lines是一个TStrings对象,它通过两个对象方法GetLines与SetLines访问。其中第二个对象方法使用了Assign过程向与该属性对应的专有字段复制新值。这个内部对象是在Create构造程序中初始化的,并在Destroy对象方法中解除。
- Selected是一个整型数,它直接访问相应的专用字段。它存储着字符串列表中的所选元素。
- Title是用于改变对话框标题的字符串。

公用属性是SelItem,这是一个只读属性,可以自动地提取字符串列表中的所选元素。注意,该属性没有存储器,也没有数据;它只是访问其他属性,提供数据的一个虚拟表示:

```

type
  TMDListBoxDialog = class (TComponent)
  private
    FLines: TStrings;
    FSelected: Integer;
    FTitle: string;
    function GetSelItem: string;
    procedure SetLines (Value: TStrings);
    function GetLines: TStrings;
  public
    constructor Create(AOwner: TComponent); override;
    destructor Destroy; override;
    function Execute: Boolean;
    property SelItem: string read GetSelItem;
  published
    property Lines: TStrings read GetLines write SetLines;
    property Selected: Integer read FSelected write FSelected;
    property Title: string read FTitle write FTitle;
  end;

```

该范例的大多数代码位于Execute对象方法中，该函数根据对话框的模态结果返回True或False。这与大部分标准Delphi对话框组件的Execute对象方法是一致的。Execute函数动态地建立窗体，使用组件的属性设置它的一些值，显示对话框，并在结果正确时更新当前选择。

```

function TMDListBoxDialog.Execute: Boolean;
var
  ListBoxForm: TListBoxForm;
begin
  if FLines.Count = 0 then
    raise EStringListError.Create ('No items in the list');
  ListBoxForm := TListBoxForm.Create (Self);
  try
    ListBoxForm.ListBox1.Items := FLines;
    ListBoxForm.ListBox1.ItemIndex := FSelected;
    ListBoxForm.Caption := FTitle;
    if ListBoxForm.ShowModal = mrOk then
    begin
      Result := True;
      Selected := ListBoxForm.ListBox1.ItemIndex;
    end
    else
      Result := False;
  finally
    ListBoxForm.Free;
  end;
end;

```

注意, 代码包含在一个try/finally块中, 这样当对话框显示时, 如果出现运行时错误, 窗体将被无条件解除。当运行它时, 如果列表是空的, 还可使用异常来引起一个错误。该错误是专门设计的, 而且使用异常是强制产生它的较好技术。组件的其他对象方法非常简单。构造器建立了FLines字符串列表, 由解除器删除, GetLines与SetLines对象方法作为整体在字符串列表中进行操作, 而GetSelItem函数会返回所选的文本:

```
function TMDListBoxDialog.GetSelItem: string;
begin
  if (Selected >= 0) and (Selected < FLines.Count) then
    Result := FLines [Selected]
  else
    Result := '';
end;
```

当然, 因为这是手工编写组件代码, 并且将其添加到原始的窗体源代码中, 因此必须记住编写Register过程。

一旦编写了Register过程, 并且组件准备好了, 就必须提供一个位图。对于非可视组件来说, 位图是非常重要的, 这是因为它们不仅被组件面板所使用, 而且当我们把组件放置到一个窗体上时也要使用它们。

使用非可视组件

当位图准备好并且组件已经安装之后, 可编写一个简单的项目来测试它。该测试程序的窗体只有一个按钮、一个编辑框以及MdListDialog组件。在程序中, 可以编写以下几行代码来响应按钮的OnClick事件:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  // select the text of the edit, if corresponding to one of the strings
  MdListDialog1.Selected := MdListDialog1.Lines.IndexOf (Edit1.Text);
  // run the dialog and get the result
  if MdListDialog1.Execute then
    Edit1.Text := MdListDialog1.SelItem;
end;
```

这就是运行放置在组件中的对话框所需要的所有代码, 如图9.10所示。对于开发一些通用对话框来说, 这是一个有趣的方法。



图9.10 ListDialogDemo范例显示了封装在ListDialog组件中的对话框

集合属性

有时我们需要一个属性来保存一系列值，而不仅仅是一个值。有时可以使用一个基于TStringList的属性，但是它只用于说明文本数据（即使一个对象可以被附加到每个字符串上）。当需要一个属性处理一组对象时，最适合VCL的解决方案就是使用集合。集合的角色是专门设计用来建立包含一系列值的属性。Delphi集合属性的例子包括DBGrid组件的Columns属性，以及TStatusBar组件的Panels属性。

集合基本上就是给定类型的对象的容器。因此，为了定义一个集合，需要从TCollection类中继承一个新类，并且从TCollectionItem类中继承一个新类。第二个类定义了集合使用的对象；通过向其传递将包含的对象的类，可以创建集合。

集合类不仅要处理集合的条目，而且当其Add方法被调用时，还要负责创建新的对象，但是不能创建一个对象，然后将其添加到一个现有的集合中。程序清单9.3显示了用于条目和一个集合的两个类，并带有它们最相关的代码。

程序清单9.3 集合及其条目的类

```

type
  TMyItem = class (TCollectionItem)
  private
    FCode: Integer;
    FText: string;
    procedure SetCode(const Value: Integer);
    procedure SetText(const Value: string);
  published
    property Text: string read FText write SetText;
    property Code: Integer read FCode write SetCode;
  end;

  TMyCollection = class (TCollection)
  private
    FComp: TComponent;
    FCollString: string;
  public
    constructor Create (CollOwner: TComponent);
    function GetOwner: TPersistent; override;
    procedure Update(Item: TCollectionItem); override;
  end;

{ TMyCollection }

constructor TMyCollection.Create (CollOwner: TComponent);
begin
  inherited Create (TMyItem);
  FComp := CollOwner;
end;

```

```
function TMyCollection.GetOwner: TPersistent;  
begin  
    Result := FComp;  
end;  
  
procedure TMyCollection.Update(Item: TCollectionItem);  
var  
    str: string;  
    i: Integer;  
begin  
    inherited;  
    // update everything in any case...  
    str := '';  
    for i := 0 to Count - 1 do  
    begin  
        str := str + (Items [i] as TMyItem).Text;  
        if i < Count - 1 then  
            str := str + '-';  
        end;  
        FCollString := str;  
    end;  
end;
```

该集合必须定义**GetOwner**方法，以便能够在Delphi IDE提供的集合属性编辑器中正确地显示。因此，它需要被连接到处理它的组件，即集合的宿主中（在该代码中保存于FComp字段）。读者可以在图9.11中看到这种范例组件的集合。

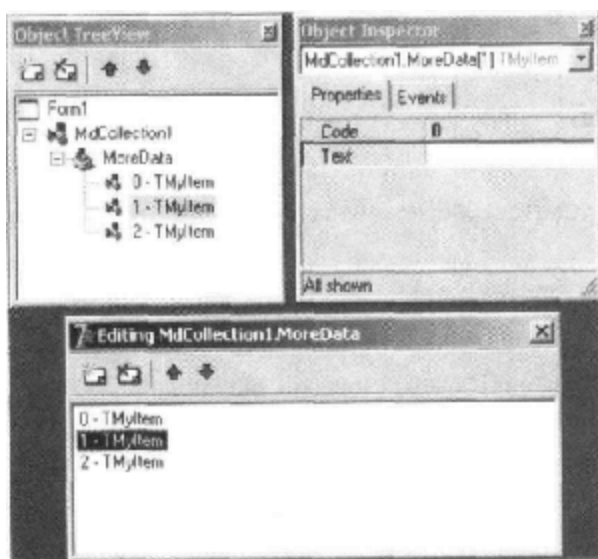


图9.11 集合编辑器，对集合条目使用对象的树状视图和对象检验器

每次在集合表项中发生数据改变时，它的代码都会调用**Changed**方法（传递True或False，指出是改变本地的条目还是在集合中引用整个条目集），这种调用的结果是由**TCollection**类调用虚拟方法**Update**，在请求更新单个条目时它作为一个参数，在所有条目改

变时它作为nil（当True做为一个参数，Changed方法被调用时）。可以覆盖这种方法，以更新集合中其他元素的值、集合本身的值或目标组件的值。

在这个例子中，我们使用添加到集合中的集合数据汇总来更新一个字符串，并且主持组件将作为一个属性出现。在一个组件中使用集合是非常简单的——只需声明一个集合，在构造器中创建它，在结束时将其释放，并通过一个属性显示它：

```

type
  TCanTest = class(TComponent)
  private
    FColl: TMyCollection;
    function GetCollString: string;
  public
    constructor Create (aOwner: TComponent); override;
    destructor Destroy; override;
  published
    property MoreData: TMyCollection read FColl write SetMoreData;
    property CollString: string read GetCollString;
  end;

constructor TCanTest.Create(aOwner: TComponent);
begin
  inherited;
  FColl := TMyCollection.Create (Self);
end;

destructor TCanTest.Destroy;
begin
  FColl.Free;
  inherited;
end;

procedure TCanTest.SetMoreData(const Value: TMyCollection);
begin
  FColl.Assign (Value);
end;

function TCanTest.GetCollString: string;
begin
  Result := FColl.FCollString;
end;

```

注意，集合条目在DFM文件中通过使用特殊的item标记器和尖括号，随着处理它们的组件一起被流化，就像下面的例子所示：

```

object MdCollection1: TMdCollection
  MoreData = <
    item
      Text = 'one'
      Code = 1

```

```

    end
    item
        Text = 'two'
        Code = 2
    end
    item
        Text = 'three'
        Code = 3
    end>
end

```

定义定制的动作

除了定义定制的组件之外，你还可以定义并注册新的标准操作，这可以在 Action List 组件的动作编辑器中实现。建立新动作并不复杂，必须继承 TAction 类，并覆盖基类的一些对象方法。

必须覆盖下面三种对象方法：

- **HandlesTarget** 函数，用于返回操作对象是否要为当前目标处理操作（默认情况下当前目标用焦点控制）。
- **UpdateTarget** 过程，用于设置与操作相连的控件用户界面，最终如果操作在当前不能执行，则禁用该动作。
- **ExecuteTarget** 对象方法，用于确定执行的实际代码，这样用户只需选择操作，而不必实现它。

为了向读者实际演示该方法，在此为一个列表框实现了三种操作，剪切、复制与粘贴，使用的方法与 VCL 实现编辑框的方法相似（尽管稍微简化了代码）。这里编写了一个基类，它从 ExtActns 单元的 TListControlAction 类继承而来。这种基类，TmdCustomListActions，添加了一些为所有特定动作共享的公共代码，并发布了一些动作属性。这三种派生类都带有它们自己的 ExecuteTarget 代码。下面是这四种类：

```

type
    TmdCustomListAction = class (TListControlAction)
    protected
        function TargetList (Target: TObject): TCustomListBox;
        function GetControl (Target: TObject): TCustomListControl;
    public
        procedure UpdateTarget (Target: TObject); override;
    published
        property Caption;
        property Enabled;
        property HelpContext;
        property Hint;
        property ImageIndex;
        property ListControl;
    end;

```

```

    property ShortCut;
    property SecondaryShortCuts;
    property Visible;
    property OnHint;
end;

TMDListCutAction = class (TMDCustomListAction)
public
    procedure ExecuteTarget(Target: TObject); override;
end;

TMDListCopyAction = class (TMDCustomListAction)
public
    procedure ExecuteTarget(Target: TObject); override;
end;

TMDListPasteAction = class (TMDCustomListAction)
public
    procedure UpdateTarget (Target: TObject); override;
    procedure ExecuteTarget (Target: TObject); override;
end;

```

HandlesTarget方法是动作类的三种关键方法之一，由TListControlAction类用下面这个代码提供：

```

function TListControlAction.HandlesTarget(Target: TObject): Boolean;
begin
    Result := ((ListControl <> nil) or
        (ListControl = nil) and (Target is TCustomListControl)) and
        TCustomListControl(Target).Focused;
end;

```

UpdateTarget方法有两种不同的实现。默认的实现由基类提供，并被复制和剪切动作所使用。这些动作当且仅当目标列表框中当前只有一个表项被选择时才能激活。粘贴行为的状态依赖于剪贴板的状态：

```

procedure TMDCustomListAction.UpdateTarget (Target: TObject);
begin
    Enabled := (TargetList (Target).Items.Count > 0)
        and (TargetList (Target).ItemIndex >= 0);
end;

function TMDCustomListAction.TargetList (Target: TObject): TCustomListBox;
begin
    Result := GetControl (Target) as TCustomListBox;
end;

function TMDCustomListAction.GetControl(Target: TObject): TCustomListControl;
begin
    Result := Target as TCustomListControl;
end;

```

```

procedure TMdListPasteAction.UpdateTarget (Target: TObject);
begin
    Enabled := Clipboard.HasFormat (CF_TEXT);
end;

```

TargetList函数使用TListControlAction类的GetControl函数，该函数要么在设计时返回与动作相连的列表框，要么返回目标控件（带有输入焦点的列表框控件）。

最后，这三个ExecuteTarget方法在目标列表框中执行相应的行为：

```

procedure TMdListCopyAction.ExecuteTarget (Target: TObject);
begin
    with TargetList (Target) do
        Clipboard.AsText := Items [ItemIndex];
end;

procedure TMdListCutAction.ExecuteTarget (Target: TObject);
begin
    with TargetList (Target) do
        begin
            Clipboard.AsText := Items [ItemIndex];
            Items.Delete (ItemIndex);
        end;
end;

procedure TMdListPasteAction.ExecuteTarget (Target: TObject);
begin
    (TargetList (Target)).Items.Add (Clipboard.AsText);
end;

```

一旦大家在一个单元中编写了这种代码，并将其添加到一个组件包中（在这里，是MdPack组件包），最终的步骤就是将新的定制行为注册到一个给定的类别中。这个类别会被RegisterActions过程作为第一个参数表示；第二个参数是用于注册的动作类的列表：

```

procedure Register;
begin
    RegisterActions ('List',
        [TMdListCutAction, TMdListCopyAction, TMdListPasteAction], nil);
end;

```

为了测试这三个定制动作，可使用本书选配光碟上的ListTest范例。该程序有两个列表框与一个工具栏，其中含有三个按钮，分别与三个定制的操作相连，还有一个用于输入新值的编辑框。该程序允许用户剪切、复制与粘贴列表框选项。读者可能会认为，这没有什么特别之处，但奇怪的事实是该程序没有代码！

警告：为了给一个动作设置一个映像（并且通常为了定义默认的属性），需要使用RegisterActions过程的第三个参数，这是一个使用预先定义的值来处理映像列表和动作列表的数据模块。因为在可以设置这样一个数据模块之前，必须注册该动作，所以在开发这些动作时，将需要重复注册。这个问题是相当复杂的，所以在这里我将不对其进行介绍，但读者可以在<http://www.blong.com/Conferences/BorCon2002/Actions/2110.htm>的“注册标准动作”和“标准动作和数据模块”部分中找到其详细的描述。

编写属性编辑器

编写组件确实是定制Delphi环境的一种有效方式，它可以帮助开发人员更快速地建立应用程序，而不需要了解有关的具体的低级技术知识。Delphi环境对第三方扩展也是非常开放的，特别是通过编写定制的属性编辑器可以轻易地扩展对象检验器，通过添加组件编辑器可以轻易地扩展窗体设计器。

与这些技术一起，Delphi还为附加工具开发者提供了内部的接口。为了使用这些称为OpenTools API的接口，需要对Delphi环境的工作原理有一个深入的理解，并且对本书中没有讨论的一些高级技术有相当好的认识。关于这些技术的信息以及一些范例，参见附录A。

说明：Delphi中的OpenTools API已经发生了明显的改变。例如，从Delphi 5开始，DsgnIntf已经被分割为DesignIntf单元、DesignEditors和其他一些特殊单元。Borland还引入了用于定义每种编辑器方法的界面。不过大部分简单的示例，如本书中出现的示例，其编译过程与Delphi的早期版本相比都没有发生变化。想要了解与此相关的更多信息，读者可参阅Delphi目录下的\Source\ToolsApi中的扩展源代码。注意，在Delphi 6 Update Pack 2中，Borland随OpenTools API的文档附带了一份Help文件。

每个属性编辑器必须是抽象类TPropertyEditor的子类，此类在DesignEditors系统单元中定义并提供了一个用于IProperty接口的标准实现。Delphi已经定义了一些特殊的属性编辑器，这些编辑器分别对应着字符串（TStringProperty类）、整型数（TIntegerProperty类）、字符（TCharProperty类）、枚举（TEnumProperty类）和集合（TSetProperty类），所以我们可以从工作的任何属性类型之中继承属性编辑器。

在任何定制的属性编辑器中，必须重新定义GetAttributes函数，这样它会返回一系列表示编辑器功能的值。最重要的属性是paValueList与paDialog。paValueList属性指定对象检验器将显示一个带有一列值（通过覆盖GetValues对象方法提供）的组合框（如果设置paSortList属性，最终还会排序）。paDialog属性类型会激活对象检验器中的省略号按钮，它用于执行编辑器的Edit对象方法。

声音属性的编辑器

前面建立的声音按钮有两个与声音有关的属性，SoundUp与SoundDown。它们实际是字符串，所以可以在对象检验器中使用一个默认的属性编辑器显示它们。然而，需要用户输入系统声音或外部文件名是不友好的，而且容易出错。

我们可以编写一个通用编辑器来处理文件名，但是也可以选择一个系统名称（系统声音被预定义连接到用户操作的声音名字，并和Windows控制面板的声音面板中实际的语音关联在一起）。因此，我建立了一个更复杂的属性编辑器。而用于声音字符串的编辑器允许用户从下拉列表选择一个值或显示一个装载和测试声音的对话框（从一个声音文件或一个系统声音）。该属性编辑器提供Edit和GetValues方法：

```
type
    TSoundProperty = class (TStringProperty)
```

```

public
    function GetAttributes: TPropertyAttributes; override;
    procedure GetValues(Proc: TGetStrProc); override;
    procedure Edit; override;
end;

```

提示：默认的Delphi规则是，用以Property结尾的名字命名一个属性编辑器类，并用以Editor结尾的一个名字命名所有的组件编辑器。

GetAttributes函数结合了paValueList（用于下拉列表）和paDialog（用于定制编辑框）属性，并归类列表，允许用于多个组件属性的选择：

```

function TSoundProperty.GetAttributes: TPropertyAttributes;
begin
    // editor, sorted list, multiple selection
    Result := [paDialog, paMultiSelect, paValueList, paSortList];
end;

```

GetValues方法多次调用它做为一个参数接收的过程，每个它打算添加到下拉列表中的字符串各一次（读者在图9.12中可以看到）：

```

procedure TSoundProperty.GetValues(Proc: TGetStrProc);
begin
    // provide a list of system sounds
    Proc ('Maximize');
    Proc ('Minimize');
    Proc ('MenuCommand');
    Proc ('MenuPopup');
    Proc ('RestoreDown');
    ...
end;

```

一种更好的方法是从Windows注册表中提取这些值，此注册表中列出了所有这些名称。Edit对象方法非常简单，因为它只是建立与显示一个对话框。读者会注意到，本可以直接显示Open对话框，但却添加了一个中间步骤，允许用户测试声音。这与Delphi测试图形属性有些相似：首先打开预览，在确定其正确之后，装载文件。最重要的一步是加载文件，然后在将它应用于该属性之前首先对它进行测试。下面是Edit对象方法的代码：

```

procedure TSoundProperty.Edit;
begin
    SoundForm := TSoundForm.Create (Application);
    try
        SoundForm.ComboBox1.Text := GetValue;
        // show the dialog box
        if SoundForm.ShowModal = mrOK then
            SetValue (SoundForm.ComboBox1.Text);
    finally
        SoundForm.Free;
    end;
end;

```



图9.12 声音列表为用户提供了提示，使之可以输入属性值或双击菜单命令，以激活编辑器

上面调用的`GetValue`与`SetValue`对象方法是由基类、字符串属性编辑器定义的。它们只读写正在编辑的当前组件的属性值。

还有一种方法，可以通过使用`GetComponent`对象方法（需要一个参数指示正在使用组件，0说明是第一个组件）访问正在编辑的组件。当直接访问该组件时，还需要调用`Designer`对象（基类属性编辑器的一个属性）的`Modified`对象方法。在本范例中，不需要这个`Modified`调用，因为基类`SetValue`对象方法会自动执行相关操作。

上面的`Edit`对象方法显示了一个对话框，它是一个通过可视方式建立的标准Delphi窗体，并被添加到组件包中处理设计时组件。该窗体非常简单，一个组合框显示了由`GetValues`对象方法返回的值，四个按钮允许打开一个文件、测试声音、通过接收或取消值终止对话框。图9.13显示了对话框的一个例子。它为编辑属性提供一个下拉式列表与一个对话框，这导致对象检验器只显示箭头按钮（标志下拉式列表），并忽略表示对话框编辑器可以使用的省略号按钮。在这里，因为它是为默认的`Color`属性编辑器产生的，故通过双击当前值或按下`Ctrl+Enter`组合键可获得这个对话框。

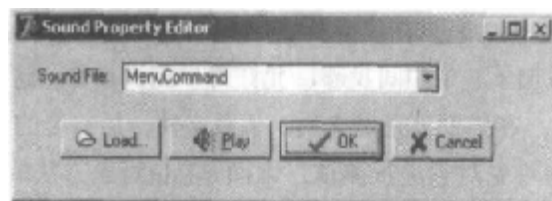


图9.13 声音属性编辑器窗体用于显示可用声音列表，并允许用户装载文件和倾听所选的声音

该窗体的头两个按钮有一个分配给它们的`OnClick`事件的方法：

```
procedure TSoundForm.btnLoadClick(Sender: TObject);
begin
  if OpenFileDialog1.Execute then
    ComboBox1.Text := OpenFileDialog1.FileName;
end;
```

```

procedure TSoundForm.btnPlayClick(Sender: TObject);
begin
    PlaySound (PChar (ComboBox1.Text), 0, snd_Async);
end;

```

遗憾的是，还没有发现一种简单的方法来确定声音是否适当地被定义和使用（检测文件是可能的，但系统声音会引起一些问题）。如果PlaySound函数不能找到要求的声音，当同步播放，而它又不能发现试图播放的默认系统声音时，该函数会返回一条错误代码。如果所需的聲音不能使用，它会播放默认的系统声音，而不会返回错误代码。PlaySound首先会在注册表中寻找声音，如果没有发现，它会检测指定的声音是否存在。

提示：如果想要进一步扩展该示例，可以在对象检验器中向下拉菜单添加图像——如果能够确定要向图像添加哪些声音的话。

安装属性编辑器

在编写完这段代码后，可以在Delphi中安装该组件及其属性编辑器。为了完成该过程，必须在单元的Register过程中添写下列语句：

```

procedure Register;
begin
    RegisterPropertyEditor (TypeInfo(string), TMdSoundButton, 'SoundUp',
        TSoundProperty);
    RegisterPropertyEditor (TypeInfo(string), TMdSoundButton, 'SoundDown',
        TSoundProperty);
end;

```

为了使用String类型的属性（第一个参数），该调用注册了在最后一个参数中指定的编辑器，但只适合特定组件与使用了特定名称的属性。后两个值可以省略，以便提供更通用的编辑器。注册该编辑器会允许对象检验器显示一系列值，而且对话框会被Edit对象方法调用。

为了安装这个组件，需将它的源代码文件添加到已有的或新的组件包中。在这里，没有将该单元以及本章中的其他单元添加到MdPack组件包中，而是该决定建立第二个组件包，包含本章中所建立的所有单元，并命名为MdDesPk（它代表了“Mastering Delphi design package”的意思）。有关该组件包，我们使用了最新的{\$DESIGNONLY}编译指令来编译它。该指令用于标记与Delphi环境相互作用的组件包，安装组件与编辑器，但在运行时，建立的应用程序并不需要它。

说明：所有在本章建立的附加工具源代码都保存在MdDesPk子目录中，该目录中还有用于安装这些附加工具的组件包代码。没有演示如何使用这些设计时工具的范例，因为只需在Delphi环境中选择相应的组件，就可以看到它们的行为。

属性编辑器的单元使用SoundB单元，定义了TMdSoundButton组件。因此，新的组件包引用了现有的组件包。这里是它的初始代码（我在本章稍后将添加其他的单元）：

```

package MdDesPk;

{$R *.RES}

{$ALIGN ON}

```



```
...
{$DESCRIPTION 'Mastering Delphi DesignTime Package'}
{$DESIGNONLY}

requires
    vcl,
    MdPack,
    designide;

contains
    PeSound in 'PeSound.pas',
    PeFSound in 'PeFSound.pas' {SoundForm};
```

编写组件编辑器

使用属性编辑器可令开发人员做出更友好的组件。事实上，对象检验器代表了Delphi环境用户界面的一个主要部分，而且Delphi开发人员对它的使用很频繁。然而，可以定制组件与Delphi交互的方式：编写一个定制的组件编辑器。

就像属性编辑器扩展了对象检验器一样，组件编辑器也扩展了窗体设计器。事实上，当在设计时用鼠标右键单击窗体时，可以看到一些默认的菜单项，还有所选组件的组件编辑器添加的菜单选项。这些菜单项的例子是那些用于激活MenuDesigner、FieldsEditor、Visual Query Builder以及其他环境编辑器的选项。有时，当被双击时，显示这些特殊编辑器成为了组件的默认行为。

组件编辑器常见的用处包括添加一个带有组件开发人员信息的About框，添加组件名称，以及为建立其属性提供特定的向导。特别地，最初的目的是允许一个向导或一些直接代码来一次设置多个属性，而不是分别设置它们。

TComponentEditor类的子类

一个组件编辑器通常应该从TComponentEditor类中继承，该类提供了TComponentEditor接口的基本实现。该接口最重要的对象方法包括：

- **GetVerbCount**对象方法，当选中一个组件时，它返回向窗体设计器弹出菜单添加的菜单选项数目。
- **GetVerb**对象方法，要对每个新菜单项调用一次，而且它将返回每个菜单项在弹出菜单中的文本。
- **ExecuteVerb**对象方法，当选中一个新菜单项时，应调用它。而且将该菜单项的数目作为参数传递。
- **Edit**对象方法，当用户双击窗体设计器中的组件以启动默认操作时，调用该方法。

一旦熟悉了这样一个概念，动词不过是带有要执行功能的相应操作的新菜单项，那么就直观地理解该界面的对象方法名称了。实际上该界面比前面见过的那些属性编辑器的界面简单得多。

说明：像属性编辑器一样，组件编辑器从Delphi 5到Delphi 6被大量地修改了，现在被定义在DesignEditor和DesignIntf单元中。

ListDialog的组件编辑器

现在，读者已经知道了有关编写组件编辑器的主要思路，下面我们来研究一个实际范例，即前面建立的ListDialog组件的编辑器。在组件编辑器中，我只想显示一个About框，向菜单添加一个版权声明（一种不恰当但很常见的组件编辑器用法），并允许用户执行一步特殊操作：预览与对话框组件相连的对话框。此外还想改变默认操作，在蜂鸣声（这并没有特别用处，但它演示了一项技术）之后简单地显示。

为了实现这个组件编辑器，程序必须覆盖在前面部分列出的四种方法：

```
uses
    DesignIntf;

type
    TMDListCompEditor = class (TComponentEditor)
        function GetVerbCount: Integer; override;
        function GetVerb(Index: Integer): string; override;
        procedure ExecuteVerb(Index: Integer); override;
        procedure Edit; override;
    end;
```

第一种方法返回菜单选项的数目，以添加到快捷菜单中，在本例中为3。这种方法只会被调用一次：在显示菜单之前。第二种方法为每个菜单选项调用一次，所以在这个例子种，它被调用了三次：

```
function TMDListCompEditor.GetVerb (Index: Integer): string;
begin
    case Index of
        0: Result := 'MdListDialog ( @Cantù )';
        1: Result := '&About this component...';
        2: Result := '&Preview...';
    end;
end;
```

这个代码将菜单选项添加到窗体的快捷菜单中，就像读者在图9.14中看到的那样。选择任何这些菜单选项，将会激活组件编辑器的ExecuteVerb方法：

```
procedure TMDListCompEditor.ExecuteVerb (Index: Integer);
begin
    case Index of
        0: ; // nothing to do
        1: MessageDlg ('This is a simple component editor'#13 +
            'built by Marco Cantù '#13 +
            'for the book "Mastering Delphi"', mtInformation, [mbOK], 0);
        2: with Component as TMDListDialog do
            Execute;
    end;
end;
```

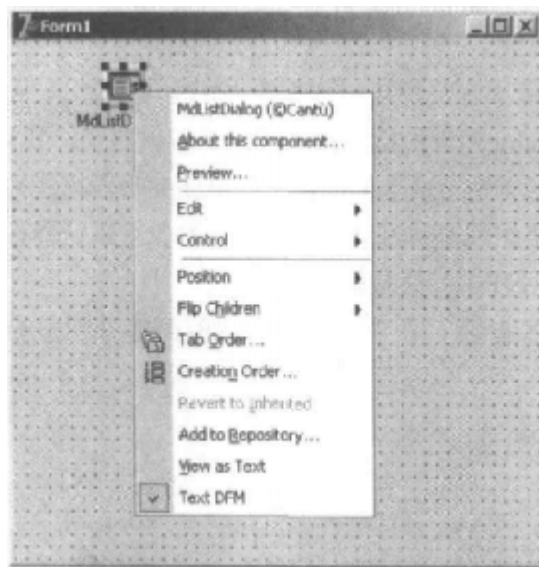


图9.14 定制菜单选项由ListDialog组件的组件编辑器添加

这里决定在case语句的一个单独分支中处理前两个菜单选项，尽管略去了版权注意选项的代码。其他命令改变调用了正在编辑组件的Execute对象方法，由TComponentEditor类的Component属性确定。知道了组件的类型后，就可以在动态类型转换后轻易地访问它的对象方法。

最后一个对象方法引用了组件的默认操作，而且通过在窗体设计器中双击组件来激活该对象方法：

```
procedure TMdListCompEditor.Edit;
begin
    // produce a beep and show the about box
    Beep;

    ExecuteVerb (0);
end;
```

注册组件编辑器

为了使该组件编辑器可以适用于Delphi环境，还需要注册它。可以向它的单元添加一个Register过程，并为组件编辑器调用一个特殊的注册过程：

```
procedure Register;
begin
    RegisterComponentEditor (TMdListDialog, TMdListCompEditor);
end;
```

这里向MdDesPk组件包添加了该单元，该组件包包括了本章所有的设计时扩展。在安装并启动该组件包之后，就可以建立一个新项目，在项目里面放置一个带制表符的列表框组件，并测试它。

小结

在本章中，我们已经了解了如何定义属性的不同类型、如何添加事件，以及如何定义并重载组件对象方法等相关内容。此外，还研究了不同组件的范例，包括对已有组件的简单改变，新的图形组件，以及最后一节中封装在组件内的一个对话框。在建立这些组件的同时，还面对了一些新的Windows编程的挑战。一般情况下，当编写Delphi新组件时，程序员必须直接使用Windows API函数。

对于软件再使用来说，编写组件是非常方便的技术，但为了使自己的组件使用起来更方便，应该尽可能地将它们与Delphi环境融为一体。为了实现该目的，可以编写属性编辑器与组件编辑器。还可以编写更多的Delphi IDE的扩展，包括定制向导。笔者自己建立了一些Delphi扩展程序，其中一些在附录A中讨论。

第10章侧重讨论Delphi DLL。大家已经在前面很多章节中用过DLL，现在是详细讨论它们的作用以及如何建立它们的时候了。在这一章中，还将进一步讨论Delphi组件包的使用，这是一种特殊类型的DLL。为了学习更多的组件开发知识，请参阅第17章，届时将专门侧重讨论数据敏感控件和定制数据集合组件。

第10章 库与组件包

Windows的可执行文件可以划分为两种形式：程序（EXE）与动态链接库（DLL）。当编写一个Delphi的应用程序时，典型的情况是产生一个程序文件。然而，Delphi的应用程序经常还调用存储在DLL中的函数。每当直接调用一个Windows API函数时，实际上是在访问一个动态链接库。在Delphi环境中，产生一个DLL是非常容易的。然而，DLL的天然属性会带来一些问题。在Windows中编写一个动态库并不总像它看起来那么简单，这是因为动态库和调用程序需要一致同意调用约定、参数类型和其他一些细节。本章将从Delphi的角度介绍DLL编程的基本概念。

本章的第二节将讨论动态链接库的一种特定类型：Delphi组件包。Delphi组件包将为简单的DLL提供一个好的方法，尽管并没有那么多的Delphi程序员在组件编写之外利用它们。这里笔者将与读者共享使用组件包分割一个大应用程序的一些有趣的技巧与技术。

本章主要讨论以下内容：

- 在Delphi中建立与使用DLL
- 在运行时调用DLL函数
- 在DLL中共享数据
- Delphi组件包的结构
- 在组件包中放置窗体

DLL在Windows中的作用

在探讨如何在Delphi以及其他编程语言中开发DLL之前，我们首先概括性地介绍一下Windows中DLL的技术问题，重点放在主要元素上。我们将先来研究动态链接，然后看一看Windows是如何使用DLL的，最后是编写DLL时应遵守的一些通用规则。

什么是动态链接

首先，重要的是需要完全理解函数或过程的静态链接与动态链接之间的区别。当一个子例程在一个源文件中不能被直接使用时，编译器会将这个子例程添加到一个内部的符号表格中。当然，Delphi编译器必须要先查到子例程的声明，并知道它的参数与类型，否则它将产生一个错误。

在一个正常的——静态的——子例程编译之后，连接器从Delphi编译过的一个单元（或静态库）中取出子例程的编译代码并将它添加到可执行文件中。最终的可执行文件包括了程序及其所属单元的所有代码。Delphi连接器是非常灵活的，它只提取程序所使用单元中最少数量的代码，而且只链接程序实际使用的函数与对象方法。这就是为什么称之为“灵活连接器”的原因。

说明：该规则有一个例外值得注意，即含有虚拟对象方法的情况。编译器不能事先确定程序将要调用的虚拟对象方法，所以它必须把所有的都包括在内。因此，拥有太多虚拟函数的程序与库会生成巨大的可执行文件。在开发VCL时，Borland开发人员必须平衡虚拟函数的灵活性与通过限制使用虚拟函数减小可执行文件大小之间的关系。

在动态链接的情况（当代码调用基于DLL的函数时）中，连接器只使用在子例程external声明中的信息于可执行文件内建立一个重要的数据表格。当Windows把这些可执行文件装载到内存中时，它首先装载所有必需的DLL，然后程序才会启动。在这个装载过程中，Windows用函数在内存中的地址充填程序的导入表格。如果由于某些原因没有找到DLL或者在找到的DLL中没有出现一个被引用的子例程，则程序甚至不能启动。

每次程序调用一个外部函数时，它就会使用这种导入数据表格将调用转发给DLL代码（它当前位于程序的地址空间中）。注意，该模式不会涉及两个不同的应用程序，DLL已经变成了运行程序的一部分，并且装载在同一地址空间。所有的参数传递都发生在应用程序的堆栈上（因为DLL没有单独的堆栈）或者在CPU寄存器上。把一个DLL装载进应用程序的地址空间，意味着任何DLL的内存分配或任何其创建的数据将会驻留在主进程的地址空间。这样，数据和内存的指针可以直接从DLL传递到程序中，反之亦然。这也可以扩展到传递对象引用，这是非常困难的，因为EXE和DLL可能会有一个不同的编辑类（并且程序员可以使用专门用于该目的的组件包，我们在本章稍后部分将会看到）。

另一种可以使用DLL的方法甚至比我刚才讨论的那个更加动态，那就是：在运行时，程序员可以在内存中装载一个DLL，查找一个函数（根据提供的名称），并且通过名称调用这个函数。这种方法需要更复杂的代码，并需要花费额外的时间以定位函数。然而，函数的执行速度会与调用一个暗中装载的DLL的速度相同。从积极的方面来说，不需要用该DLL来启动这个程序。我们将在本章稍后部分的DynaCall范例中使用这种方法。

为什么要使用DLL

既然读者已经了解了DLL是如何工作的一般概念，我们可以再来讨论一下为什么使用它们。首先第一个优势就是，如果不同的程序使用相同的DLL，则只需在内存中装载DLL一次即可，这样将节省系统内存。DLL会映射到每个进程（每运行一次应用程序）的专用地址空间中，但它们的代码只在内存中装载一次。

说明：为了更精确，操作系统会将同一个地址上的DLL装载到每个应用程序的地址空间中（使用由DLL指定的首选基址）。但是如果该地址在某个特殊应用程序的虚拟地址空间内不能使用，则该过程的DLL代码映像必须被重新定位——从性能和内存角度来看开销都是非常大的一种操作，这是因为重新定位只以每个过程为基础，而不是整个系统。

另一个有趣的特性就是，程序员可以提供一种不同版本的DLL，代替当前的DLL，而不需要重新编译就可以使用。当然，只有当DLL中的函数与旧版本中的函数参数一样时，这种方法才是可行的。如果DLL中有新的函数时，这不会有什么影响。只有当新版本在行为上不能实现顺应老版本中DLL的操作功能，而且类、基础类甚至编译器版本出现不匹配时，才会有问题。

第二个优点特别适用于复杂的应用程序。如果我们有一个非常大而且需要不断更新或改正错误的应用程序，可以将它划分成多个可执行的部分与动态库，这样允许我们只对需要

改变的部分进行操作，而不用对整个大的可执行文件进行改动。这样做对于Windows系统库格外有意义。如果微软提供给用户一个Windows系统库的更新版本——例如，推出一个操作系统或服务包的新版本，那么程序员将不需要重新编译他们的代码了。

另一个常用的技术是，使用动态库去存储资源。可以建立一个DLL的不同版本来保存不同语言的字符串，然后在运行时改变语言；或者准备一个图标库与位图库，然后在不同的应用程序中使用它们。开发一个程序的专用语言版本是非常重要的，而且Delphi可通过Integrated Translation Environment (ITE) 为此功能提供支持。

还有一个主要的优点就是，DLL是独立于编程语言的。大多数Windows编程环境，包括在终端用户应用程序中的大部分宏语言，都允许程序员调用存储在DLL中的函数。尽管这种灵活性只应用于函数的使用。要想在不同的编程语言之间，在一个DLL中共享一个对象，应该求助于COM基础架构或.NET结构。

Delphi DLL编写者应遵守的规则

对于Delphi DLL程序员来说，应该遵守一些规则。被外部程序调用的一个DLL函数或过程必须遵照以下这些规则：

- 它必须列在DLL的exports子句中。这使得子例程对外部可见。
- 输出函数还应该被声明为stdcall，以使用标准的Win32参数传递技术来代替优化的register参数传递技术（它在Delphi中是默认的）。如果只想从其他Delphi应用程序使用这些库时，该规则可能会产生异常。当然也可以使用另外一个由其他理解它的编译器提供的调用约定（就像cdecl，在C编译器中默认提供）。
- 一个DLL的参数类型应是默认的Windows类型（大部分为兼容C的数据类型），至少是在希望该DLL能够应用于其他开发环境中的情况下。对于输出字符串来说，还有更进一步的规则，读者将在FirstDLL范例中看到。
- 一个DLL可以使用全局数据，该数据将不会由调用应用程序来共享。每当一个应用程序装载一个DLL时，它便会在自己的地址空间中存储DLL的全局数据，稍后在DllMem范例中读者将会看到。
- Delphi的库应当触发所有的内部异常，除非程序员打算只从其他Delphi程序中使用该库。

使用现有的DLL

在本书的许多例子中，当调用Windows API函数时，我们已经使用过现有的DLL。读者可能还记着，所有的API函数都在系统Windows单元中声明。函数在单元的interface部分中声明，如下所示：

```
function PlayMetaFile(DC: HDC; MF: HMETAFILE): BOOL; stdcall;
function PaintRgn(DC: HDC; RGN: HRGN): BOOL; stdcall;
function PolyPolygon(DC: HDC; var Points; var nPoints; p4: Integer):
    BOOL; stdcall;
function PtInRegion(RGN: HRGN; p2, p3: Integer): BOOL; stdcall;
```

接着，在implementation部分，由单元引用DLL中的外部定义，而不是提供每个函数的代码：

```
const
  gdi32 = 'gdi32.dll';

function PlayMetaFile; external gdi32 name 'PlayMetaFile';
function PaintRgn; external gdi32 name 'PaintRgn';
function PolyPolygon; external gdi32 name 'PolyPolygon';
function PtInRegion; external gdi32 name 'PtInRegion';
```

说明：在Windows.PAS中，大量地使用了新指令{\$EXTERNALSYM identifier}。这实际与Delphi本身没有太大关系；因为它应用于C++ Builder中。该指令避免了相应的Delphi符号出现在C++翻译的头文件中。这种行为有助于Delphi与C++标识符保持同步，这样就可以在两种语言之间共享代码了。

这些函数的外部定义引用了它们使用的DLL名称。DLL的名称必须包括DLL扩展名，否则程序将无法在Windows NT/2000/XP下工作（即使它可以工作在Windows 9x下）。另一个元素是DLL函数本身的名称。如果Delphi函数（或过程）的名称与DLL函数的名称相匹配（区分大小写），就不需要name指令了。

为了调用驻留在DLL中的一个函数，可以像上面介绍的那样，在部署部分的一个单元中或外部定义的接口部分中提供函数声明，或者在一个单元的部署部分中将这两个合并到一个声明中。一旦正确定义了函数，就可以像调用其他任何函数一样，在Delphi应用程序代码中调用它。

提示：Delphi中包含有大量Windows API的Delphi语言转换，读者在Delphi的Source\Rtl\Win文件夹中可以看到许多文件都是这样的。更多的引用其他API的Delphi单元在www.delphi-jedi.org的Delphi Jedi项目中。

使用C++ DLL

作为例子，笔者已经用C++编写了一个非常简单的DLL，用来说明如何从一个Delphi应用程序中调用DLL。我在这里将不会详细地讨论C++代码（它基本是C代码），而只是重点介绍Delphi应用程序与C++ DLL之间的调用。在Delphi编程中，使用C或C++编写的DLL是常见的事情。

假设我们有一个用C或C++建立的DLL。通常这意味着我们手头上将会有有一个DLL文件（编译库）、一个H文件（库中函数的声明）以及一个LIB文件（C/C++连接器输出函数列表的另一个版本）。该LIB文件在Delphi中根本没有用处；而DLL文件的使用都一样；H文件必须用相应的声明转化为一个Delphi的单元。

在下面的程序清单中，读者可以看到用于建立CppDll库的C++函数的声明。C++ DLL的编译版本和完整的源代码，以及使用它的Delphi应用程序的源代码都在CppDll目录中。可以用任何C++编译器编译这段代码；而我只是使用了Borland C++编译器对它进行测试。下面是函数的C++声明：

```
extern "C" __declspec(dllexport)
int WINAPI Double (int n);
```



```
extern "C" __declspec(dllexport)
int WINAPI Triple (int n);
__declspec(dllexport)
int WINAPI Add (int a, int b);
```

三个函数对参数执行了一些基本计算并返回结果。注意，所有的函数都是用WINAPI修改程序定义的，该修改程序设置了相应的参数调用约定；并在__declspec(dllexport)声明之后进行，此声明使得函数可以在外部使用。

在这些C++函数中，还有两个使用了C的命名约定（由extern “C” 语句表示），但第三个函数，Add，则没有使用。读者将看到，这个区别会影响我们在Delphi中调用这些函数的方式。除了Add函数外，这些函数的内部名称与它们在C++源代码文件中的名称是一致的。因为没有为Add函数使用extern “C” 子句，所以C++编译器使用了名称矫正技术（name mangling）。这项技术用于将与参数序数以及类型有关的信息包括在函数名称中，C++语言需要这些信息来执行函数重载。使用Borland C++编译器的结果是产生一个有趣的函数名称：@Add\$qqsii。在Delphi中必须使用这个名称来调用Add DLL函数（这揭示了为什么在输出函数中总是避免使用C++名称矫正技术，以及为什么通常会声明这些函数为extern “C” 的原因）。下面是Delphi CallCpp范例中这三个函数的声明：

```
function Add (A, B: Integer): Integer;
    stdcall; external 'CPPDLL.DLL' name '@Add$qqsii';
function Double (N: Integer): Integer;
    stdcall; external 'CPPDLL.DLL' name 'Double';
function Triple (N: Integer): Integer;
    stdcall; external 'CPPDLL.DLL';
```

读者可以看到，能够为外部函数提供或省去别名。这里为第一个函数（没有其他选择，因为输出DLL函数的名称是@Add\$qqsii，这是无效的Delphi标识符）与第二个函数起了一个别名，尽管对于第二个函数并不是必要的。事实上，如果两个名称相匹配，就可以略去name指令，就像上面对第三个函数所做的一样。如果不能确定DLL调用的函数的实际名称，则可以利用Borland的TDump命令程序，通过-ec命令行切换，在Delphi BIN文件夹中使用这个程序。

需要记住向每个定义添加stdcall指令，以便调用模块（应用程序）与被调用模块（DLL）使用相同的参数传递约定。如果不能这样做，将会得到一个以随机值作为参数的调用，且程序错误很难发现。

说明：当必须将一个大的C/C++头文件转换成相应的Delphi声明时，除了进行手工转换外，可以使用第三方工具部分自动地进行转换。其中的一个工具是HeadConv，由Bob Swart编写。大家可在他的站点<http://www.drbob42.com>上找到这个工具。该工具通过Project Jedi进行了扩展，在DARTH项目下（www.delphi-jedi.org/team_darth_home）。注意，从C/C++到Delphi的自动头文件翻译是不可能的；因为Delphi比C++强大得多，因此必须更精确地使用类型。

为了使用该C++ DLL，我建立了一个Delphi范例，名为CallCpp。其简单的窗体中只有用于调用DLL各个函数的按钮和一些用于输入和输出参数的可视组件（如图10.1所示）。注意，要想运行该程序，就需要在项目所在的相同目录下有该DLL，或者在Windows的主目录（\Windows、\WinNT）或其系统目录（\Windows\System、\WinNT\System32）下有该

DLL。如果将可执行文件移动到一个新目录下并试图运行它，就会获得一个运行时错误，显示DLL丢失。

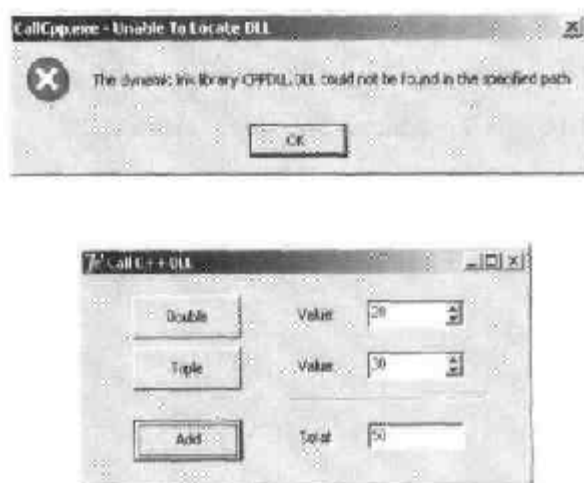


图 10.1 单击所有按钮之后CallCpp范例的输出显示

在Delphi中创建DLL

除了使用在其他环境中编写的DLL外，还可以使用Delphi来建立DLL，该DLL可以供Delphi程序使用，或由其他任何支持DLL的开发工具使用。在Delphi中建立DLL是非常简单的，以至于程序员可能会过多地使用该特性。一般来说，笔者建议读者尽量建立组件包而不是一般的DLL。正如我在本章前面所讨论的那样，组件包通常会包含组件，但它们也包含普通的非组件类，它允许程序员编写面向对象的代码并有效地重新使用它。当然，组件包也可以包含简单的例程、常量、变量等等。

正如我前面曾提到过，当一个程序代码的某部分需要频繁地改变时，建立一个DLL是非常有用的。在这种情况下，可以不断地替换DLL，同时保持程序的其他部分不变。同样，当需要编写一个程序为不同组的用户提供不同的性能时，可以为不同的用户分配一个DLL的不同版本。

创建自己的第一个Delphi DLL

在开始研究用Delphi开发DLL之前，我先来介绍一个在Delphi中建立的库。该范例的首要目的是解释在Delphi中用于定义一个DLL的语法，但它也可解释有关传递字符串参数的一些问题。首先，选择File►New►Other命令，并在对象存储库的New页面上选择DLL选项。这样将创建一个非常简单的源文件，它以下面这行定义开头：

```
library Project1;
```

这个库声明表明要建立一个DLL，而不是一个可执行文件。现在可以向库中添加例程，并在一个exports声明中列出它们：

```
function Triple (N: Integer): Integer; stdcall;  
begin  
    try
```

```
    Result := N * 3;  
except  
    Result := -1;  
end;  
end;  
  
function Double (N: Integer): Integer; stdcall;  
begin  
    try  
        Result := N * 2;  
    except  
        Result := -1;  
    end;  
end;  
  
exports  
    Triple, Double;
```

在这个DLL的基本版本中，不需要使用uses语句；但是通常来说，主项目文件只包括uses和exports语句，而函数声明放在另一个独立单元中。FirstDll范例源代码的最终版本与列在这里版本的相比稍有不同，我已经做了一点改动，以便每当调用函数时会显示一条消息。显示消息有两种方法；最简单的方法是使用Dialog单元并调用ShowMessage函数。

这段代码需要Delphi将大量的VCL代码链接到应用程序中。如果静态地链接VCL到这个DLL中，最终文件大小将达到几百KB。原因是，ShowMessage函数显示了一个VCL窗体，其中包含了VCL控件，并使用了VCL图形类；同时间接引用了VCL流系统、VCL应用程序与屏幕对象等。对于这个简单的范例，还有一种更好的方法，就是使用Windows单元并调用MessageBox函数，通过直接API调用来显示消息，这样就不需要VCL代码了。该代码中的这个改动使应用程序的大小降到了仅仅50KB。

说明：大小上的巨大差别强调了这样一个事实，就是不能在Delphi中过度使用DLL，以避免在多个可执行文件中编译VCL的代码。当然，可以通过使用运行时组件包减小Delphi DLL，在本章稍后将会详细讨论该技术。

如果通过使用基于API的DLL版本运行一个像CallFrst范例（以后说明）这样的测试程序，它的行为将会不正常。事实上，可以多次单击调用DLL函数的按钮，而不用首先关闭DLL显示的消息框。出现这种情况的原因是因为MessageBox API调用的第一个参数是零，它的值应该是程序主窗体或应用程序窗体的句柄——尚未介绍的DLL信息。

在Delphi DLL中重载函数

C++创建一个DLL时，重载函数使用名称矫正技术来为每个函数生成一个不同的名称。参数的类型包在名称中，就像读者在CppDll范例中曾看到的一样。

用Delphi创建一个DLL，并使用重载函数（也就是说，多个函数使用相同的名称，并用overload指令标记）时，Delphi只允许导出这些有原始名称的重载函数中的一个，表明它的参数列在exports子句中。如果需要输出多个重载函数，就需要在exports子句中指定多个名称以区分这些重载函数。这种技巧在FirstDll代码这一部分中说明：

```
function Triple (C: Char): Integer; stdcall; overload;
function Triple (N: Integer): Integer; stdcall; overload;

exports
  Triple (N: Integer),
  Triple (C: Char) name 'TripleChar';
```

说明：反过来也是可能的：可以从一个DLL中调入一系列相似的函数，并在Delphi声明中将它们都定义为重载函数。Delphi的OpenGLPAS单元中包含了这种技巧的一系列范例。

从DLL中导出字符串

通常，在一个DLL中的函数可以使用任何参数的类型，并返回任何类型的值。但这个规则会有两个例外：

- 如果计划从其他编程语言中调用DLL，就应该使用Windows原数据类型而不是Delphi的专有类型。例如，为了表达颜色值，可以使用整数或Windows ColorRef类型而不是Delphi的TColor类型来进行相应的转换（将在下一节的FormDLL中详细介绍）。为了兼容性的需要，应该避免使用其他一些Delphi类型，包括对象（根本就不能被其他语言使用）、Delphi字符串（可以由PChar字符串代替）。换句话说，每个Windows开发环境都必须支持API的基本类型，并且如果遵守这一点，那么所建立的DLL就可以用于其他的开发环境中。Delphi文件变量（记录的文本文件或二进制文件）不能在DLL之外使用，但可以使用Win32文件处理。
- 即使只打算在Delphi应用程序中使用DLL，也不能不加小心地在DLL边界之间传递Delphi字符串（和动态数组）。这是因为，Delphi在内存中管理字符串的方法是——自动地分配、重新分配与自动释放它们。解决该问题的方法是，在DLL中和使用它的程序中包含ShareMem系统单元。该单元必须作为每个项目的第一个单元。而且，必须随程序和特定库一起使用BorlndMM.DLL文件（该名字代表Borland Memory Manager）。

在FirstDLL范例中，实际使用了两种方法：一个函数接收并返回一个Delphi字符串，另一个函数接收一个PChar指针（然后由函数自己充填）作为参数。第一个函数在Delphi通常被写为：

```
function DoubleString (S: string; Separator: Char): string; stdcall;
begin
  try
    Result := S + Separator + S;
  except
    Result := '[error]';
  end;
end;
```

第二个函数非常复杂，因为PChar字符串没有一个简单的+操作符，而且与字符不能直接兼容；在添加分隔符之前必须将它转化为字符串。下面是完整的代码；它使用了输入与输入PChar缓冲区，这两个缓冲区与任何Windows的开发环境都兼容：

```

function DoublePChar (BufferIn, BufferOut: PChar;
  BufferOutLen: Cardinal; Separator: Char): LongBool; stdcall;
var
  SepStr: array [0..1] of Char;
begin
  try
    // if the buffer is large enough
    if BufferOutLen > StrLen (BufferIn) * 2 + 2 then
      begin
        // copy the input buffer in the output buffer
        StrCopy (BufferOut, BufferIn);
        // build the separator string (value plus null terminator)
        SepStr [0] := Separator;
        SepStr [1] := #0;
        // append the separator
        StrCat (BufferOut, SepStr);
        // append the input buffer once more
        StrCat (BufferOut, BufferIn);
        Result := True;
      end
    else
      // not enough space
      Result := False;
    except
      Result := False;
    and;
  end;

```

该代码的第一个版本确实是非常复杂的，但第一个版本只能在Delphi中使用。而且，正如以前讨论的，第一个版本需要在DLL中（以及使用它的程序中）包含ShareMem单元，并与程序以及特定的库一起配置BorlandMM.DLL，就像我们在前面讨论过的那样。

调用Delphi DLL

那么，该怎样使用刚才建立的库呢？可以从另一个Delphi项目或其他环境中调用它。作为一个例子，我已经建立了CallFrst项目（存储在FirstDLL目录中）。为了访问DLL函数，必须将它们声明为**external**，就像对C++ DLL所做的一样。然而，这一次可以从Delphi DLL的源代码中复制并粘贴函数的定义，添加**external**子句如下：

```

function Double (N: Integer): Integer;
  stdcall; external 'FIRSTDLL.DLL';

```

这个声明与用于调用C++ DLL的声明相似。然而，这一次没有函数名称的问题。一旦它们被重新声明为**external**，DLL的函数就可以作为局部函数使用。下面是两个例子，带有关于字符串相关函数的调用（在图10.2中输出的一个例子是可视的）：

```

procedure TForm1.BtnDoubleStringClick(Sender: TObject);
begin
    // call the DLL function directly
    EditDouble.Text := DoubleString (EditSource.Text, '');
end;

procedure TForm1.BtnDoublePCharClick(Sender: TObject);
var
    Buffer: string;
begin
    // make the buffer large enough
    SetLength (Buffer, 1000);
    // call the DLL function
    if DoublePChar (PChar (EditSource.Text), PChar (Buffer), 1000, '/') then
        EditDouble.Text := Buffer;
end;

```

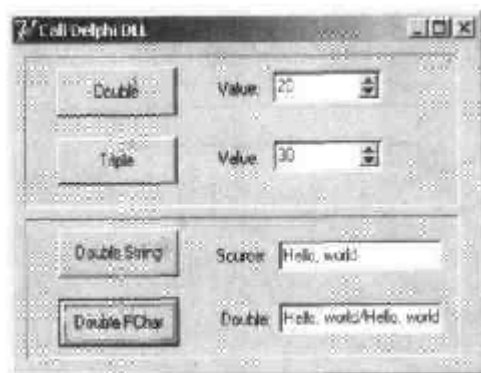


图10.2 CallFirst范例调用我们刚建立的Delphi DLL所得到的输出

Delphi DLL的高级特性

除了这个介绍性的例子之外，我们可以在Delphi中对动态库做一些其他的事情。例如，使用一些新的编译器命令来影响库的名称，在运行时调用一个DLL，将整个Delphi窗体放置在一个动态库中，等等。这些将是下面各小节所讨论的主题。

改变项目和库名称

对于一个库，作为一个标准的应用程序来说，需要以一个与Delphi项目文件名称相匹配的库名称作为结尾。在Kylix中介绍过这样一个技巧，可用来兼容共享对象库（在Linux中与Windows DLL等效）的标准Linux命名约定；而下面这个技巧与其相似，即Delphi 6为用户引入了特殊的编译指令，可以在库中使用这些指令来确定它们的可执行文件的名称。这种命令其中一部分在Linux系统下会比在Windows中的意义更大，但它们已经都被加入到各种版本中：

- **\$LIBPREFIX**，用于在库的名称前面添加一些内容。与Linux技术在库名称前添加lib相似，Kylix使用该指令在组件包名称前加上前缀bpl。这样做主要是因为Linux为库

使用一个单独的扩展名（.SO），而在Windows中，却可以使用不同的扩展名，如Borland给组件包使用.BPL扩展名

- **\$LIBSUFFIX**，用于在库名称之后和扩展名之前添加文本。这种文本可以用来指定各种不同版本的信息或其他一些关于库名的变化。在Windows系统上这是非常有用的。
- **\$LIBVERSION**，用于在扩展名之后添加一个版本信息——这一作法在Linux中是很常见的，不过通常在Windows中需要避免使用这种命令

在IDE中，这些指令可以从Project Options对话框的Application页面中进行设置，读者可以在图10.3中看到这一点。作为一个例子，读者应认真地考虑下面这些指令，它们将生成一个被调用的库文件MarcoNameTest60.dll：

```
library NameTest;  
{$LIBPREFIX 'Marco'}  
{$LIBSUFFIX '60'}
```

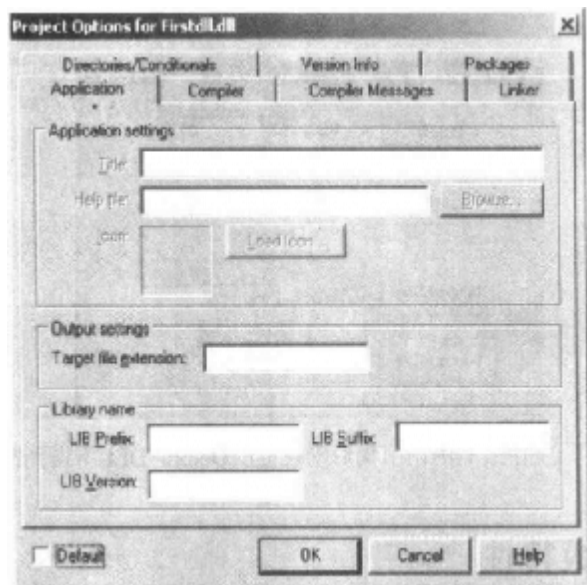


图10.3 Project Options对话框的Application页面现在拥有Library Name部分

说明：Delphi 6组件包广泛地使用\$LIBSUFFIX指令。由于这个原因，VCL包产生VCL.DCP文件和VCL70.BPL文件。该方法的优点就是程序员不需要为Delphi的每个新版本改变组件包的requires部分。当然，将项目从Delphi 6移植到Delphi 7中时，这是非常有帮助的，因为Delphi以前的版本不提供这个特性。当重新打开一个Delphi 5的组件包时，仍然需要升级它们的源代码，Delphi IDE并不能自动地提供这一操作。

在运行时调用DLL函数

到现在为止，我们已经在代码中引用了库输出的函数，因此DLL将会随程序一起装载。我以前曾提到过，可以推迟装载DLL，也就是说，直到需要它时才进行装载，这样在DLL不能获得时，还能继续使用程序的余下部分。

在Windows中动态装载DLL可通过调用LoadLibrary API函数完成，它会在程序文件夹、文件夹路径和一些系统文件夹中查找DLL。如果DLL没有被找到，Windows将显示一个错误

信息,有时可以通过调用Delphi的SafeLoadLibrary函数略过它。该函数和它封装的API有同样的效果,但它会抑制标准Windows错误信息并优先在Delphi中装载动态库。

如果库被发现并装载(有时通过检查LoadLibrary或SafeLoadLibrary的返回值可知晓),则程序可以调用GetProcAddress API函数,它会查找DLL的导出表格,查找作为参数传递的函数名字。如果GetProcAddress发现了一个匹配,它将向请求过程返回一个指针。现在,可以将这个函数指针指到正确的数据类型并调用它了。

不管使用哪种装载函数,不要忘记最后都要调用FreeLibrary,以便DLL能从内存中被正确地释放出来。事实上,系统会对库使用一种引用计数技术,这样,在每个装载请求后,将会跟着一个释放请求以释放它们。

我已经建立的例子显示了动态DLL装载的过程,其名称为DynaCall。该DLL使用我们在本章先前建立的FirstDLL库(为了使该程序工作,必须将DLL从其源文件夹复制到与DynaCall示例相同的文件夹中)。在取代了声明Double和Triple函数并直接使用它们之后,该示例获得了与一些更复杂的代码相同的结果。然而,其优点就是该程序甚至可以在没有DLL的情况下运行。如果一个新的compatible函数被添加到DLL中,则不得不修订程序的源代码并重新编译它来访问那些新函数。下面是该程序的核心代码:

```

type
  TIntFunction = function (I: Integer): Integer; stdcall;

const
  DllName = 'Firstdll.dll';

procedure TForm1.Button1Click(Sender: TObject);
var
  HInst: THandle;
  FPointer: TFarProc;
  MyFuncnt: TIntFunction;
begin
  HInst := SafeLoadLibrary (DllName);
  if HInst > 0 then
  try
    FPointer := GetProcAddress (HInst,
      PChar (Edit1.Text));
    if FPointer <> nil then
    begin
      MyFuncnt := TIntFunction (FPointer);
      SpinEdit1.Value := MyFuncnt (SpinEdit1.Value);
    end
    else
      ShowMessage (Edit1.Text + ' DLL function not found');
  finally
    FreeLibrary (HInst);
  end
  else
    ShowMessage (DllName + ' library not found');
end;

```


警告: 由于库使用Borland内存管理器, 故动态装载它的程序必须做相同的事。所以需要在DynaCall范例的项目中添加ShareMem单元。奇怪的是, Delphi的旧版本并不是这样的, 以防库不能有效地使用字符串。需要提醒的是, 如果忽略了这个内涵, 将获得一个明显的系统错误, 这个错误甚至可以使在FreeLibrary调用上的调试器停止运转。

在Delphi中如果有一个指向一个过程的指针, 那么如何调用它呢? 一种解决方案就是, 将指针转化为一个过程类型, 然后调用这个使用过程类型变量的过程, 就像在上面的代码清单中所做的一样。注意, 定义的过程类型必须与DLL中的过程定义相兼容。但该方法的缺点是——没有真正的关于参数类型的检测。

那么这种方法的优点是什么呢? 从理论上讲, 可以使用它在任何时候访问任何DLL的任何函数。实际上, 就像在这个例子中那样, 当拥有含有匹配函数的多个DLL时, 或者含有多个匹配函数的单个DLL时, 该方法都可以使用——通过在编辑框中输入Double与Triple对象方法的名称来调用它们。现在, 如果有人给你一个含有新函数的DLL, 该函数接受整型数作为参数而且返回整型数, 则你可以通过在编辑框中输入该函数的名称调用它, 甚至不需要重新编译应用程序。

使用这段代码, 编译器与连接器会忽视DLL存在与否。当程序装载时, 并不会马上装载DLL。甚至可以使程序更灵活, 并让用户输入所需DLL的名称。在一些情况中, 这有很大的好处。一个程序可能会在运行时切换DLL, 而这是采用直接方式所不允许的。注意, 这种装载DLL函数的方法在宏语言中是常见的, 并且也可以被很多可视编程环境使用。

只有基于编译器与连接器的系统, 如Delphi, 才可以使用这种直接的方法, 一般说来, 这种方法更可靠、更安全, 也稍稍快一些。我的观点是, DynaCall范例中的间接装载方法只在特殊情况中 useful, 但它的功能确实很强大。另一方面, 在组件包(包括窗体)的动态装载应用中有很多的值, 这一点读者将在本章的结尾看到。

将Delphi窗体放置在库中

除了编写一个带有函数及过程的库之外, 还可以将用Delphi建立的一个完整窗体放置到一个动态库中。该窗体可以是一个对话框或其他任何类型的窗体, 其他Delphi程序和其他能够使用动态链接库的开发环境或宏语言都可以使用它。一旦创建了一个新的库项目, 所有需要我们做的就是将一个或多个窗体添加到项目中, 然后编写将创建和使用那些窗体的导出函数。

例如, 激活一个模式对话框以选择一种颜色的一个函数可以编写如下:

```
function GetColor (Col: LongInt): LongInt; cdecl;
var
  FormScroll: TFormScroll;
begin
  // default value
  Result := Col;
  try
    FormScroll := TFormScroll.Create (Application);
    try
      // initialize the data
```

```
FormScroll.SelectedColor := Col;  
// show the form  
if FormScroll.ShowModal = mrOK then  
    Result := FormScroll.SelectedColor;  
finally  
    FormScroll.Free;  
end;  
except  
    on E: Exception do  
        MessageDlg ('Error in library: ' + E.Message, mtError, [mbOK], 0);  
    end;  
end;
```

是什么造成了这种与你在一个程序中通常所编写的代码不同呢？这就是异常处理的使用。

- 整个函数由一个try/except块保护，由函数产生的任何异常都将会被捕获，并显示一条相应的消息。处理每个可能异常的原因是，调用的应用程序可能是用任何语言编写的——特别是不知道如何处理异常的某种语言。即使调用的应用程序是Delphi程序，使用相同的保护方法有时也是有用的。
- 窗体上的操作由一个try/finally块保护，它确保了只要引起了一个异常，窗体对象就将会被正确地解除。

通过检查ShowModal对象方法的返回值，程序可以确定函数的结果。我在这里在输入try块之前设置了默认值，以确保该代码块总是可以被执行（而且还避免了编译器警告，它会提示函数的结果没有定义）。

读者可以在FormDLL和UseCol项目中发现这个代码摘录，这些项目在FormDLL文件夹中（还有一个WORDCALL.TXT文件会显示怎样从一个Word宏中调用例程）。这个例子也说明，可以向DLL中添加一个非模态的窗体，但是这样做将会造成太多的困难。非模态窗体和主窗体不能同步，这是因为DLL在它自己的VCL拷贝中有它自己的全局Application对象。通过将该应用程序的Application对象复制到库的Application对象的Handle中，可以部分地修正这种情况。读者在例子中找到的代码并不能解决所有的问题。一个更好的解决方案可能是编译程序和库，以便能够使用Delphi组件包，这样VCL代码和数据将不会被复制。但是这种方法还是会引起一些新的问题：通常建议大家不要同时使用Delphi DLL和组件包。那么，我给各位最好的建议是什么呢？为了使一个库的窗体能被其他的Delphi程序所使用，最好使用组件包而不是简单的DLL！

内存中的DLL：代码与数据

在我开始讨论组件包之前，想先介绍一下动态库的一个技术要素：它们怎样使用内存。让我们从库的代码部分开始，然后，讨论它的全局数据。当Windows装载一个库的代码时，就像其他任何的代码模块一样，它必须要进行一个fixup操作。这个fixup包括跳出的地址和带有被装载的实际地址的内部函数调用。这种操作的效果是代码装载内存取决于它从哪里被装载。

对于可执行文件来说这并不是一个问题，但是对于库来说，这可能会带来很大的问题。如果两个可执行文件在相同的基地址上调用相同的库，在机器的RAM（物理内存）中将会只有一份DLL代码的物理拷贝，这样将会节约内存空间。如果第二次装载库的内存地址已经被使用，它需要重定位，也就是说，由另一个被应用的fixup所移动。所以我们将以在RAM中DLL代码的第二个物理拷贝作为结束。

基于GetProcAddress API函数，可以使用动态装载技术来检查函数对应的当前过程的内存地址，其代码如下所示：

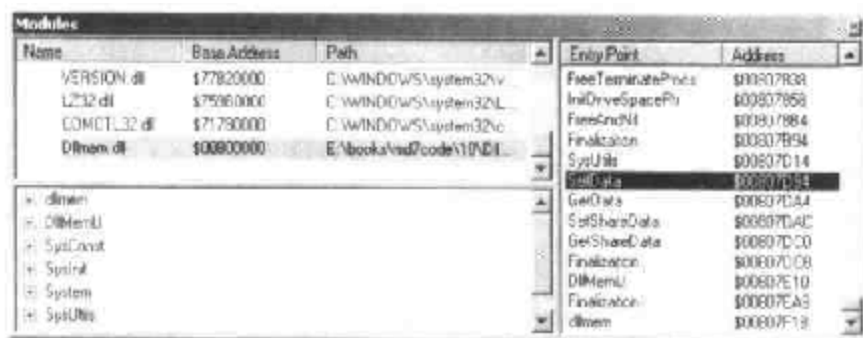
```

procedure TForm1.Button3Click(Sender: TObject);
var
    HDLLInst: THandle;
begin
    HDLLInst := SafeLoadLibrary ('dllem');
    Label1.Caption := Format ('Address: %p', [
        GetProcAddress (HDLLInst, 'SetData')]);
    FreeLibrary (HDLLInst);
end;

```

这段代码在一个标题框中显示了调用应用程序地址空间内函数的内存地址。如果使用这段代码运行两个程序，它们通常将显示相同的地址。这说明代码在相同的内存地址中只能被装载一次。

用来获得更多信息的另一种技术就是使用Delphi的Modules窗口，该窗口可以显示被模块引用的每个库的基地址，以及库中每个函数的地址，如下图所示。



重要的是要知道一个DLL的基地址可以通过设置基地址选项来要求。在Delphi中，这个地址可以通过Project Options对话框连接器页面中的Image Base值来确定。例如，在DIEMem库中，我已经将其设为\$00800000。读者需要为自己的每个库指定一个不同的值，并验证它不会与可执行文件所使用的任何系统库或其他库（组件包、ActiveX等等）冲突。再次强调一下，可以通过使用调试器的Module窗口来确定它。

尽管这并不能保证定位惟一，但为库设置一个基地址总是比不设置要好一些；在这种情况下，重定位操作总会发生，但是两个不同的可执行文件将相同的库重分配到相同地址上的机会并不会很多。

说明：也可以使用<http://www.sysinternals.com>中的Process Explorer来检查任何机器上的任何过程。这个工具还会有一个选项用于强调重分配的DLL。检查同一程序在不同的操作系统（Windows 2000、Windows XP和Windows ME）上使用它的库的运行效果，并确定一个未被使用的区域。

这是DLL代码的一种情况，但是对于全局数据呢？主要地，DLL的每个拷贝在调用的应用程序的空间中都有其自己的数据拷贝。用于共享数据的最常见的技术就是使用内存映像文件。我将对一个DLL使用这种技术，但是它也可用于在应用程序之间直接共享数据。

这个例子在针对库时被叫做DllMem，而针对演示程序时被叫做UseMem。DLL代码有一个项目文件，将会导出四个子例程：

```
library dllmem;

uses
  SysUtils,
  DllMemU in 'DllMemU.pas';

exports
  SetData, GetData,
  GetShareData, SetShareData;

end.
```

实际的代码在二级单元中（DllMemU.PAS），该单元包含用于读取或编写两个全局内存定位的四个例程的代码。这些内存定位可将整型数和指针保存到整型数中。这里是变量声明和两个Set例程：

```
var
  PlainData: Integer = 0; // not shared
  ShareData: ^Integer; // shared

procedure SetData (I: Integer); stdcall;
begin
  PlainData := I;
end;

procedure SetShareData (I: Integer); stdcall;
begin
  ShareData^ := I;
end;
```

使用内存映像文件共享数据

对于没有被共享的数据，没有什么其他的事要做。然而，为了访问共享数据，DLL必须创建一个内存映像文件，然后获得指向这个内存区的指针。这两步操作需要两个Windows API调用：

- **CreateFileMapping**需要的参数有文件名（或\$FFFFFFFF，以便在内存中使用一个虚拟文件）、一些安全与保护属性、数据的大小以及一个内部名称（必须相同，以使多个调用应用程序共享映像文件）。
- **MapViewOfFile**需要的参数有内存映像文件的句柄、一些属性与偏移量以及数据的大小。

下面是initialization部分的源代码，每当DLL被装载到一个新过程空间中时，它就会执行（也就是说，每个使用DLL的应用程序执行一次）：

```

var
    hMapFile: THandle;

const
    VirtualFileName = 'ShareDllData';
    DataSize = sizeof (Integer);

initialization
    // create memory mapped file
    hMapFile := CreateFileMapping (FFFFFFFF, nil,
        Page_ReadWrite, 0, DataSize, VirtualFileName);
    if hMapFile = 0 then
        raise Exception.Create ('Error creating memory-mapped file');

    // get the pointer to the actual data
    ShareData := MapViewOfFile (
        hMapFile, File_Map_Write, 0, 0, DataSize);

```

当应用程序终止并且DLL被释放时，它必须将指针释放到被映像的文件中，而且文件映像为：

```

finalization
    UnmapViewOfFile (ShareData);
    CloseHandle (hMapFile);

```

UseMem演示程序的窗体有4个编辑框（两个由一个UpDown控件连接着）、5个按钮、和一个标签。第一个按钮保存有DLL数据中的第一个编辑框的值——从连接着的UpDown控件取得：

```
SetData (UpDown1.Position);
```

如果单击第二个按钮，程序将会把DLL数据复制到第二个编辑框中：

```
Edit2.Text := IntToStr(GetData);
```

第三个按钮用于显示一个函数的内存地址，在本节一开始显示了它的源代码。最后两个按钮基本上与前两个按钮的代码相同，只是它们调用了SetShareData过程与GetShareData函数。

如果运行这个程序的两份拷贝，就可以看到每一份程序中DLL的普通全局数据都有其各自的值，但共享数据的值是相同的。在两份程序中设置不同的值，然后读取它们，就会明白我的意思是什么。这种情况如图10.4所示。

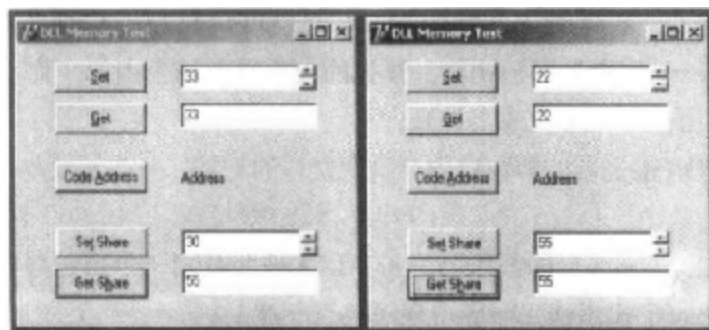


图10.4 运行UseMem程序的两个拷贝，就会知道DLL中的全局数据并未共享

警告：内存映像文件保留了一个64KB的最小虚拟地址范围，并以4KB的页面消耗物理内存。范例在共享内存中使用的4字节整型数据更耗费内存，特别是在使用相同的方法共享多个值时。如果需要共享多个变量，则应该将它们都放在一个共享内存区中（使用指针访问不同的变量或为它们都建立一个记录结构）。

使用Delphi组件包

在Delphi中，组件包是DLL的一种重要类型。组件包允许人们包装一组组件，然后静态（向应用程序的可执行文件中添加它们的编译代码）或动态（将组件代码保存在一个DLL中，把要发布的运行时组件包和应用程序以及所需要的其他所有组件包放在一起）地链接组件。在第9章“编写Delphi组件”中读者看到了建立一个组件包的方法，现在我们来讨论组件包链接的两种方式的优缺点。有很多要素需要注意：

- 使用诸如DLL那样的组件包可以减小可执行文件的大小。
- 将组件包单元链接到程序允许程序员只分配组件包代码的一部分。通常，应用程序可执行文件的大小加上所需组件包DLL的大小要远远大于静态链接程序的大小。链接器只包含了程序实际使用的代码，而一个组件包必须与组件包内所含单元interface部分中声明的所有函数和类相链接。
- 如果基于同一个组件包分配多个Delphi应用程序，结果可能会分配较少的代码，这是因为运行时组件包是共享的。换句话说，只要应用程序的用户拥有标准的Delphi运行时组件包，就可以生成非常小的可执行程序。
- 如果基于同一个组件包运行多个Delphi应用程序，则可以在运行时节省一些内存空间；运行时组件包的代码在多个Delphi应用程序之间只会在内存中被装载一次。
- 不要太担心大型的可执行文件的分配问题。记住，对一个程序进行较少改动时，可以使用各种工具创建一个补丁（patch）文件，这样只需分配一个包含改动的文件，而不是整个文件。
- 如果在一个运行时组件包中放置一些程序的窗体，就可以在程序中共享它们。然而，改动这些窗体时，通常需要重新编译主程序，并将它们重新发布给用户。下一节将详细讨论这个复杂的问题。
- 一个组件包就是编译单元（包括类、类型、变量、例程）的一个集合，这些单元与程序中的单元没有任何不同。惟一的不同在于建立过程。组件包单元的代码和主程序单元中使用时的代码是相同的。这是组件包的可以论证的一个关键优势。

组件包版本

一个非常重要但又常常被误解的要素是更新组件包的分配。当更新一个DLL时，可以建立新版本，而且该DLL的可执行文件仍可运行（除非删除了现有的输出函数或改变了它们的一些参数）。

然而，当分配一个Delphi组件包时，如果更新组件包并在组件包的任何单元的interface部分中改动任何代码，就都需要重新编译所有使用组件包的应用程序。如果需要向一个类中添加方法或属性，也需要重新编译，不过如果只是添加新的全局符号（或修改任何没有被客

户程序使用的部分)，则无需这样做。如果修改只影响了组件包单元的implementation部分，则不会有任何问题。

Delphi中的DCU文件具有一个基于其时间戳和校验和（从单元的接口部分中计算得出）的版本标记。当改变一个单元的interface部分时，基于它的每个单元都应该被重新编译。编译器会将旧编译单元的时间戳和校验和与新的时间戳和校验和进行比较，并确定从属单元是否必须重新编译。这就是当获得改动了的系统单元的Delphi新版本时，需要重新编译每个单元的原因。

在Delphi 3中（组件包在这个版本中第一次被引入），编译器向组件包库（以组件包的一个校验和命名，其校验和来自于其所含单元的校验和与其所需组件包的校验和）添加了一个额外的入口函数，这个校验和函数被使用组件包的程序所调用，这样任何基于旧版本组件包的可执行文件将在启动时失效。

Delphi 4与随后版本直到Delphi 7，都放宽了组件包的运行时限制（尽管对DCU文件的设计时限制依然未变）。组件包的校验和不再需要检测，所以可以直接改动属于组件包的单元并配置组件包的一个新版本，以用于已有的可执行文件。因为对象方法是通过名称引用的，所以不能删除任何已有的对象方法。因为名称矫正技术被添加到包中以防止对参数的改变，所以甚至不能改动其参数。

从调用程序中删除一个引用的对象方法会在装载过程中停止程序。然而，如果做任何其他的改变，程序都会在其执行期间意外地出现故障。例如，如果使用一个相似的组件来代替在一个组件包中编译的窗体中的另一个组件，则调用程序在该内存位置上仍可以访问该组件，尽管它现在是不同的组件了！

如果决定使用这种不可靠的方法，在组件包内改变单元的接口，而不重新编译使用它的所有程序，那么至少应该对所做的改变进行限制。向窗体中添加新属性或非可视对象方法时，应该与已经使用组件包的已有程序保持完全的兼容性。而且，添加字段与虚拟的对象方法可能影响类的内部结构，导致已有程序（希望一个不同的类数据与VMT设计）出现问题。

警告：这里讨论的是编译程序的分配（可以划分为EXE与组件包的分配），而不是将组件向其他Delphi程序员分配的相关内容。在后一种情况中，版本规则更严格，必须格外地小心组件包的版本。

说到这里，我建议大家绝不要改变被组件包导出的任何单元接口。为此，可以向组件包添加一个带有窗体创建函数的单元（像前面出现的带有窗体的DLL一样），并使用它访问定义窗体的另一个单元。尽管无法隐藏链接入组件包的单元，但如果从未直接使用在单元中定义的类，则通过其他例程，将可以更为灵活地改动它。还可以使用窗体继承来改动组件包内的一个窗体，而不用实际影响原版本。

对于组件包来说，组件编写者应该遵循的最严格的规则是：为了组件包内代码的长期配置与维护，应该有一个主要版本，与一个次要的维护版本。组件包的一个主要版本需要所有客户程序重新编译，组件包文件自己也应该使用新版本号重新命名，而且单元的接口部分可以改动。该组件包的维护版本应该仅被限制为实现上的改动，以保持完全与已有的可执行文件及单元兼容，就像Borland使用它的Update Packs通常所做的那样。

组件包内的窗体

第9章已经介绍过在Delphi应用程序中使用组件包，现在我们来介绍用于配置一个应用程序的组件包和DLL的使用，首先讨论包含窗体的组件包的开发。在本章前面，笔者已经提到过，可以在DLL内使用窗体，但这样做将会引起一些问题。如果正在Delphi中建立库和可执行文件，则使用组件包将是较好的解决方案。

第一眼看起来，大家可能认为Delphi组件包是惟一一种将组件发布到其安装环境中的方法。然而，也可以使用组件包作为构造代码的一种方式，但是，与DLL不同的是，仍然保留有Delphi的OPP的全部能力。假设，一个组件包是一种编译单元的集合，并且程序使用几个单元。那么，程序引用的单元将会在可执行文件中编译，除非要求Delphi将它们放置在一个组件包内。就像在前面讨论过的那样，这是为什么使用组件包的一个主要的原因。

因此，为了设置一个应用程序以使它的代码可被分配到一个或多个组件包和主可执行文件中，只需要在一个组件包中编译一些单元，然后建立主程序的选项来动态地链接这个组件包。例如，我复制了一个“平常”颜色的选择窗体，并重命名它的单元为PackScrollF；然后，创建了一个新的组件包并向它添加该单元，如图10.5所示。

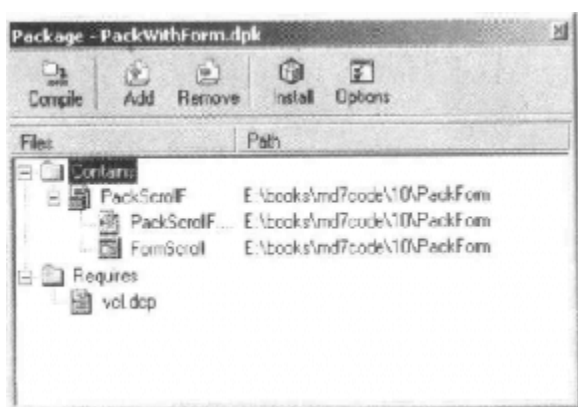


图10.5 在Delphi的组件包编辑器中放置组件包的结构

在编译这个组件包之前，应该改变它的默认输出目录，以引用当前的文件夹，而不是Delphi的标准的/Projects/Bpl子文件夹。要想这样做，需要进入组件包Project Options的Directories/Condition页面，并为Output目录（为BPL）设置当前目录（简写为一个点）和DCP输出目录。然后编译组件包，此时不要在Delphi中安装它，因为没有必要这样做。

这时，可以创建一个正常的应用程序，并编写将在程序中使用的标准代码来显示一个二级窗体，如下所示：

```
uses
    PackScrollF;

procedure TForm1.BtnChangeClick(Sender: TObject);
var
    FormScroll: TFormScroll;
begin
```



```

FormScroll := TFormScroll.Create (Application);
try
  // initialize the data
  FormScroll.SelectedColor := Color;
  // show the form
  if FormScroll.ShowModal = mrOK then
    Color := FormScroll.SelectedColor;
finally
  FormScroll.Free;
end;
end;

procedure TForm1.BtnSelectClick(Sender: TObject);
var
  FormScroll: TFormScroll;
begin
  FormScroll := TFormScroll.Create (Application);
  // initialize the data and UI
  FormScroll.SelectedColor := Color;
  FormScroll.BitBtn1.Caption := 'Apply';
  FormScroll.BitBtn1.OnClick := FormScroll.ApplyClick;
  FormScroll.BitBtn2.Kind := bkClose;
  // show the form
  FormScroll.Show;
end;

```

这种方法的一个优点是，可以将一个窗体编译进一个包中，此时该窗体的代码与在程序中编译一个窗体所使用的代码完全相同。事实上，如果编译这个程序，窗体的单元将会和程序绑定在一起。为了将窗体的单元保持在包中，必须使用用于应用程序的运行时报，并手工地将PackWithForm包添加到运行时包的列表中（Delphi IDE并不提倡这种做法，这是因为在开发环境中并没安装这个包）。

一旦执行了这个步骤，编译程序，它将像平常一样运行。但是，现在窗体在一个DLL包中，可以在包中修改窗体，重新编译它并简单运行该程序来看一下结果。注意，由于多数改变影响包的单元的interface部分（例如，向窗体中添加一个组件或一个方法），所以也应该重新编译调用该包的可执行文件。

说明：读者可以在与当前章节有关的源代码的PackForm文件夹上找到测试它的包和程序。下一示例的代码也在相同的文件夹中。组件包和项目都由文件夹中的项目组（BPG）文件所引用。

在运行时装载组件包

在上面的例子中，我们说明了PackWithForm包是一个运行时组件包，被应用程序所使用。这意味着该组件包被请求来运行该程序，并且当程序启动时装载，就像DLL的典型使用那样。可以通过动态地装载组件包来避免这两个方面，正如我们对DLL所做的那样。程序的结果将更灵活，启动更快，使用内存更少。

需要记住的一个重要要素就是，调用LoadPackage和UnloadPackage Delphi函数，而不是loadLibrary和FreeLibrary Windows API函数。不同之处是Delphi提供的函数不但装载组件包，还调用它们的正确的初始化和最终代码。

除了这个重要的元素之外——一旦你知道了，将会很容易完成——程序将要求一些额外的代码，因为不能从主程序引用到支持窗体的单元中。不能直接的使用窗体类，也不能访问它的属性和组件——至少不是标准Delphi代码。然而，这两个问题可以通过使用类引用、类注册和RTTI（运行时类型信息）来解决。

让我们从第一个方法开始。在包的窗体单元中添加这个初始化代码：

```
initialization
  RegisterClass (TFormScroll);
```

由于组件包已经装载，故主程序可以使用Delphi的GetClass函数来注册类的类引用，然后为这个类引用调用Create构造器。

为了解决第二个问题，在组件包中创建窗体的SelectedColor属性作为一个公开的属性，所以可以通过RTTI访问它。然后使用下面的代码取代访问这个属性（FormScroll.Color）的代码：

```
SetPropValue (FormScroll, 'SelectedColor', Color);
```

总结所有这些改变，这里是主程序（DynaPackForm应用程序）所使用的代码，用于显示动态装载的组件包中的模式窗体：

```
procedure TForm1.BtnChangeClick(Sender: TObject);
var
  FormScroll: TForm;
  FormClass: TFormClass;
  HandlePack: HModule;
begin
  // try to load the package
  HandlePack := LoadPackage ('PackWithForm.bpl');
  if HandlePack > 0 then
    begin
      FormClass := TFormClass(GetClass ('TFormScroll'));
      if Assigned (FormClass) then
        begin
          FormScroll := FormClass.Create (Application);
          try
            // initialize the data
            SetPropValue (FormScroll, 'SelectedColor', Color);
            // show the form
            if FormScroll.ShowModal = mrOK then
              Color := GetPropValue (FormScroll, 'SelectedColor');
          finally
            FormScroll.Free;
          end;
        end;
    end;
```

```

end
else
    ShowMessage ('Form class not found');
    UnloadPackage (HandlePack);
end
else
    ShowMessage ('Package not found');
end;

```

注意，程序一旦完成，就不再装载组件包。这个步骤不是强制的。我们可以在窗体的 OnDestory 处理器中移动 UnloadPackage 调用，并避免第一次之后重装组件包。

现在，可以尝试在没有组件包的情况下运行这个程序。大家将看到它会正常启动，只有在单击 Chang 按钮时，会显示无法找到组件包的信息。在该程序中，不需使用运行时组件包（除该单元的可执行文件之外，因为没有引用代码中的单元）。PackWithForm 页面也不需要被列入运行时组件包中。然而，必须为其使用运行时组件包，或者把程序包含在 VCL 全局变量中（如 Application 对象），并且动态装载的组件包将包括另一个版本，因为它将引用 VCL 页。

警告：当动态装载组件包的程序被关闭时，我们可能要面对存取违例问题。它们会频繁地发生，这是因为即使组件包被卸载后，定义在组件包中的类对象仍然会保存在内存中。当程序关闭后，可能会通过调用一个不存在的 VMT 的 Destroy 方法来试图释放那个对象，并且会引起错误。笔者通过经验知道，这些错误的类型是非常难以追踪和修复的。我建议大家要保证在卸载组件包之前释放所有的对象。

在组件包中使用界面

通过对象方法和属性访问窗体的类比使用各处的 RTTI 访问类更简单。为了建立一个大的应用程序，我试图使用界面并拥有多个窗体，且每个都执行几个程序定义的标准界面。一个示例并不能真正地判断这种结构的类型，它对于大程序是相对的，但我这里仍试图建立一个程序来显示该想法是怎样应用在实际中的。

说明：如果读者对界面不太了解，我建议你在学习这一小节之前，先参阅第2章“Delphi 编程语言”的相关部分。

为了构造 IntfPack 项目，我使用了三个组件包加上一个演示应用程序。三个组件包中有两个（分别叫做 IntfFormPack 和 IntfFormPack2）都用于定义挑选颜色的可选窗体。第一个组件包（叫做 IntfPack）负责其他两个组件包都使用的一个共享单元。这个单元基本包括界面的定义。我不能添加它到两个组件包的任何一个之中，这是因为不能使用具有相同名称的单元来加载两个组件包（即使在运行时装载）。

IntfPack 组件包的唯一文件就是 IntfColSel 单元，显示在程序清单 10.1 中。该单元定义了通用的接口（在真正的应用程序中可能会有其中一些），以及一个注册类列表；它模仿 Delphi 的 RegisterClass 方法，但可获得整个列表以方便浏览。

程序清单 10.1 IntfPack 组件包的 IntfColSel 单元

```

unit IntfColSel;

interface

```

```

uses
    Graphics, Contnrs;

type
    IColorSelect = interface
    ['{3F961395-71F6-4822-BD02-3B475FF516D4}']
        function Display (Modal: Boolean = True): Boolean;
        procedure SetSelColor (Col: TColor);
        function GetSelColor: TColor;
        property SelColor: TColor
            read GetSelColor write SetSelColor;
    end;

procedure RegisterColorSelect (AClass: TClass);

var
    ClassesColorSelect: TClassList;

implementation

procedure RegisterColorSelect (AClass: TClass);
begin
    if ClassesColorSelect.IndexOf (AClass) < 0 then
        ClassesColorSelect.Add (AClass);
    end;

initialization
    ClassesColorSelect := TClassList.Create;

finalization
    ClassesColorSelect.Free;

end.

```

一旦获得了这种界面，就可以定义窗体来实现它，就像下面这个从IntfFormPack中提取的例子中看到的-一样：

```

type
    TFormSimpleColor = class(TForm, IColorSelect)
    ...
    private
        procedure SetSelColor (Col: TColor);
        function GetSelColor: TColor;
    public
        function Display (Modal: Boolean = True): Boolean;

```

这两种存取方法将会读写窗体（在本例中是一个ColorGrid控件）中一些组件的颜色值，而Display方法将根据参数的不同，内部调用Show或ShowModal：

```

function TFormSimpleColor.Display(Modal: Boolean): Boolean;
begin
    Result := True; // default
    if Modal then

```

```

        Result := (ShowModal = mrOK)
    else
    begin
        BitBtn1.Caption := 'Apply';
        BitBtn1.OnClick := ApplyClick;
        BitBtn2.Kind := bkClose;
        Show;
    end;
end;

```

就像读者从这个代码中看到的那样，当窗体是非模态的时候，OK按钮将变成一个Apply按钮。最后，单元在initialization部分中有注册代码，因此当该组件包动态装载时，它能够执行：

```
RegisterColorSelect (TFormSimpleColor);
```

第 4 个组件包，IntfFormPack2，有一个相似的结构，但有一个不同的窗体。可以在源代码中查找到它（我并没有在这里讨论第二个窗体，因为它的代码并没有向本例的结构中添加太多的东西）。

该结构建立后，可以建立一个更优雅更灵活的主程序，该主程序基于单个窗体。当窗体被创建时，它定义了一个组件包列表（HandlePackages），并将它们全部装载。在示例代码中有一个含有硬编码的组件包，但仍可以查找当前文件夹的组件包或使用一个配置文件来使程序结构更灵活。最后，装载完组件包之后，该程序在一个列表框中显示注册类。下面是LoadDynaPackage和FormCreat对象方法的代码：

```

procedure TFormUseIntf.FormCreate(Sender: TObject);
var
    I: Integer;
begin
    // loads all runtime packages
    HandlesPackages := TList.Create;
    LoadDynaPackage ('IntfFormPack.bpl');
    LoadDynaPackage ('IntfFormPack2.bpl');

    // add class names and select the first
    for I := 0 to ClassesColorSelect.Count - 1 do
        lbClasses.Items.Add (ClassesColorSelect [I].ClassName);
        lbClasses.ItemIndex := 0;
    end;

procedure TFormUseIntf.LoadDynaPackage(PackageName: string);
var
    Handle: HModule;
begin
    // try to load the package
    Handle := LoadPackage (PackageName);
    if Handle > 0 then

```

```

    // add to the list for later removal
    HandlesPackages.Add (Pointer(Handle))
  else
    ShowMessage ('Package ' + PackageName + ' not found');
  end;

```

保持组件包句柄列表的主要原因是能够在程序结束的时候卸载它们。不需要这些句柄来访问在那些组件包中定义的窗体；用来创建和显示一个窗体的运行时代码将使用对应的组件类。这里是用于显示一个非模态窗体（由一个复选框控制的一个选项）的代码摘录：

```

var
  AComponent: TComponent;
  ColorSelect: IColorSelect;
begin
  AComponent := TComponentClass
    (Classes.ColorSelect[LbClasses.ItemIndex]).Create (Application);
  ColorSelect := AComponent as IColorSelect;
  ColorSelect.SelColor := Color;
  ColorSelect.Display (False);

```

该程序通过**Supports**函数来检查在使用之前窗体确实能支持界面，并且说明窗体的模式版本；但其本质是在前面四个语句中正确地描述它。

顺便提一下，要注意的是，该代码不需要窗体。一个好的练习将能向该结构中添加一个组件包，该组件包中有一个封装有颜色选择对话框或继承它的组件。

警告：主程序会引用负责界面定义的单元，但是没有链接这个文件。而且，它应当使用包含这个单元的运行时组件包，就像动态装载组件包那样。另外，主程序将使用相同代码的不同拷贝，包括全局类的不同列表。它是一个全局类的列表，不能在内存中复制。

组件包的结构

读者可能会问，是否可以知道一个单元是被链接到可执行文件中，还是被用做运行时组件包？在Delphi中这不但是可能的，而且可以全面剖析一个应用程序的结构。一个组件可以使用非正式的**ModuleIsPackage**全局变量（在**SysInit**单元中声明）。我们虽需要该变量，但从技术上讲，对于拥有不同代码（根据它是否打包而定）的一个组件来说是可能的。下列代码用于提取运行时组件包的名称：

```

var
  fPackageName: string;
begin
  // get package name
  SetLength (fPackageName, 100);
  if ModuleIsPackage then
  begin
    GetModuleFileName (HInstance, PChar (fPackageName), Length (fPackageName));
    fPackageName := PChar (fPackageName) // string length fixup
  end;

```

```
end  
else  
    fPacName := 'Not packaged';
```

除了在组件中访问组件包信息外（如上述代码那样），还可以从组件包库的特殊入口点进行访问，**GetPackageInfoTable**函数这样做。该函数会返回一些特殊的组件包信息，Delphi将这些信息作为资源来存储，并包括在组件包DLL中。幸运的是，不需要使用低级技术访问该信息，因为Delphi为此提供了一些高级函数：

可以使用两个函数来访问组件包信息：

- **GetPackageDescription**会返回一个字符串，其中包含了对组件包的描述。为了调用该函数，必须提供模块（组件包库）的名称作为唯一的参数。
- **GetPackageInfo**不会直接返回有关组件包的信息。要传递给它一个函数，它才会为组件包中内部数据结构的每个入口调用该函数。实际上，**GetPackageInfo**将会为组件包所含的每个单元及所需组件包而调用函数。而且，**GetPackageInfo**会在一个整型变量中设置一些标记。

这两个函数调用允许我们访问一个组件包的内部信息，但怎么知道应用程序使用的是哪个组件包呢？可以通过使用低级函数查看可执行文件来确定这个信息，但Delphi又一次提供了一种较为简单的方法。**EnumModules**函数不会直接返回应用程序模块的有关信息；但它允许传递给它一个函数，它会为应用程序的每个模块、主要的可执行文件以及应用程序依靠的每个组件包调用该函数。

为了演示该方法，我这里建立了一个简单的范例程序，在一个**TreeView**组件中显示模块与组件包信息。每个第一级节点对应着一个模块；每个模块中建立了一个子树，用以显示该模块包含与需要的组件包，以及组件包描述与编译器标记（**RunOnly**与**DesignOnly**）。读者可以在图10.6中看到该范例的输出。



图10.6 PackInfo范例的输出，含有所用组件包的详情

除了TreeView组件外，我还向主窗体添加了一些其他组件，但从视图中隐藏了它们：一个DBEdit、一个Chart与一个FilterComboBox。添加这些组件只是为了在应用程序中包括更多的运行时组件包，超过Vcl和Rtl组件包。此窗体类唯一的对象方法是FormCreate，它调用了模块的枚举函数：

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  EnumModules(ForEachModule, nil);
end;
```

EnumModules函数接收两个参数：第一个是回调函数（在本例中是ForEachModule），第二个是指向数据结构的指针（回调函数将使用它，在本例中是nil，因为不需要它）。回调函数必须接收两个参数——一个HInstance值与一个无类型指针——而且必须返回Boolean值。EnumModules函数将会依次为每个模块调用回调函数，将每个模块的实例句柄作为第一个参数，数据结构指针（在本例中是nil）作为第二个参数：

```
function ForEachModule (HInstance: Longint;
  Data: Pointer): Boolean;
var
  Flags: Integer;
  ModuleName, ModuleDesc: string;
  ModuleNode: TTreeNode;
begin
  with Form1.TreeView1.Items do
  begin
    SetLength (ModuleName, 200);
    GetModuleFileName (HInstance,
      PChar (ModuleName), Length (ModuleName));
    ModuleName := PChar (ModuleName); // fixup
    ModuleNode := Add (nil, ModuleName);

    // get description and add fixed nodes
    ModuleDesc := GetPackageDescription (PChar (ModuleName));
    ContNode := AddChild (ModuleNode, 'Contains');
    ReqNode := AddChild (ModuleNode, 'Requires');

    // add information if the module is a package
    GetPackageInfo (HInstance, nil, Flags, ShowInfoProc);
    if ModuleDesc <> '' then
    begin
      AddChild (ModuleNode, 'Description: ' + ModuleDesc);
      if Flags and pfDesignOnly = pfDesignOnly then
        AddChild (ModuleNode, 'Design Only');
      if Flags and pfRunOnly = pfRunOnly then
        AddChild (ModuleNode, 'Run Only');
    end;
  end;
  Result := True;
end;
```


从上面的代码中可以看到, **ForEachModule**函数一开始添加模块名称作为树的主要节点(通过调用**TreeView1.Items**对象的**Add**对象方法并将**nil**作为第一个参数传递)。然后, 添加了两个子节点, 它们存储在这个单元实现部分声明的**ContNode**与**ReqNode**变量中。

然后, 程序会调用**GetPackageInfo**函数并传递给它另一个回调函数, **ShowInfoProc**, 稍后将会介绍这个函数, 该函数可以提供一系列应用程序或组件包的单元。在该函数末尾, 如果模块是一个组件包, 就添加更多信息, 如对它的描述与编译器标记(如果其描述不是空字符串, 它就是一个组件包)。

在前面, 我曾提到过向**GetPackageInfo**函数传递另一个回调函数, **ShowInfoProc**过程;**GetPackageInfo**函数会依次为模块所含与需要的每个组件包调用我们的回调函数。该过程会建立一个描述组件包及其主要标记(添加在圆括号中)的字符串, 然后根据模块的类型, 将该字符串插入到一个节点(**ContNode**或**ReqNode**)之后。可以通过检测**NameType**参数确定模块类型。下面是第二个回调函数的完整代码:

```
procedure ShowInfoProc (const Name: string; NameType: TNameType; Flags: Byte;
  Param: Pointer);
var
  FlagStr: string;
begin
  FlagStr := ' ';
  if Flags and ufMainUnit <> 0 then
    FlagStr := FlagStr + 'Main Unit ';
  if Flags and ufPackageUnit <> 0 then
    FlagStr := FlagStr + 'Package Unit ';
  if Flags and ufWeakUnit <> 0 then
    FlagStr := FlagStr + 'Weak Unit ';
  if FlagStr <> ' ' then
    FlagStr := ' (' + FlagStr + ')';
  with Form1.TreeView1.Items do
    case NameType of
      ntContainsUnit: AddChild (ContNode, Name + FlagStr);
      ntRequiresPackage: AddChild (ReqNode, Name);
    end;
  end;
```

小结

这一章介绍了如何调用驻留在DLL中的函数, 怎样使用Delphi创建DLL。在讨论动态库之后, 又主要介绍了Delphi的组件包, 特别是如何将窗体和其他类放到组件包中。这是一个方便的技巧, 可以将一个Delphi应用程序分割为多个可执行文件。当讨论组件包时, 我解释了如何使用这些高级技巧, 包括RTTI和界面, 以获得动态的、灵活的应用程序结构。

在第12章“从COM到COM+”中介绍COM和OLE时, 将返回到揭示对象和类库的话题上来。届时关注另一个关于Delphi应用程序的主题: 模块工具的使用和OOP的相关技巧。

第11章 建模和OOP编程（使用ModelMaker）

当Borland决定向Delphi 7的Enterprise和Architect版提供UML设计方案时，它选择了绑定ModelMaker，并且使用Holland（www.modelmakertools.com）的ModelMaker Tools。ModelMaker是集成在Delphi IDE中的高质量UML设计工具。然而，随着读者逐渐熟悉ModelMaker和学习本章课程，就会发现ModelMaker不仅仅是一个UML图表工具。当然，在这里我们将主要讨论ModelMaker的图表功能，但是我们还会利用一点时间来介绍该工具的其他一些功能。

ModelMaker在早期的Delphi中就已经出现，随着时间的推移，它增添了很多选项来支持Delphi语言并最大程度地来方便程序员，因此它就有了一个很大的功能集。ModelMaker用户界面包含100多个窗体，如果没有很好的基础，很快就会感到灰心。跟着我们一起学习吧，相信大家很快就不会害怕ModelMaker了。

虽然ModelMaker通常是指UML图表设计工具，但是我更喜欢将它描述为Delphi特定的、用户化的、可扩展的、完整的UML图表设计和CASE工具。说它是Delphi特定的，是因为它被设计用来处理Delphi代码。例如，当使用某个属性时，ModelMaker中的对话框会显示特定于Delphi语言的选项——关键字和概念。说ModelMaker是用户化的，是因为有上百个选项可以控制从对象模型中如何生成Delphi代码。说ModelMaker是可扩展的，是因为它包含功能强大的OpenTools API，它允许创建插件来扩展产品功能。说ModelMaker是完整的，是因为它提供了开发周期所有阶段的功能。最后，将ModelMaker说成是CASE工具，是因为它可以为Delphi类自动生成很多可见的代码，由程序员提供类的操作代码。

说明：本章是与Robert Leahey一起编写的。Robert对ModelMaker有深入的了解和丰富的经验。在软件领域，Robert是一个设计师、程序员、作家和演说家。作为一个音乐人，他有20多年的演奏经历并是North Texas大学音乐理论系的研究生。在其公司——Thoughtsmithy（www.thoughtsmithy.com），Robert提供咨询和培训服务，开发商业软件、音频产品，并从事大型LEGO设计。Robert与其妻子和女儿一起住在North Texas。

本章主要包括以下内容：

- 了解ModelMaker
- 建模和UML
- ModelMaker的编码功能
- 文档和宏
- 重分解代码
- 实现模型

了解ModelMaker的内部模型

在讨论ModelMaker的UML支持以及其他功能之前，关键的是要了解ModelMaker如何管理代码模型。不同于Delphi和其他编辑器，ModelMaker不会连续地解析源代码来显示内容。考虑一下Delphi：用来更改代码的任何IDE工具都会直接更改源代码文件的内容（为了持续修改必须保存内容）。然而，ModelMaker能够维持表示类、代码、文档等的内部模型。当通过ModelMaker中的多个编辑器来编辑模型时，这些更改会被应用到内部模型上——而不是外部的源代码文件，至少在ModelMaker重新生成外部文件之前是这样的。理解这种区别对读者很有帮助。

要了解的另一个概念是ModelMaker能够在用户界面的多个视图中表示一个单个的内部代码模型。例如，把这个模型作为类分层或作为包含类的单元列表来查看或编辑。类成员可以用多种方法来分类、筛选、分组和编辑。有很多用于ModelMaker的插件可以查看这些视图。但是最重要的是，UML图表编辑器本身是模型中的另一个视图。当在图表中显示模型元素（例如类）时，就是在创建代码模型元素的可视表示；如果从图表中删除一个符号，实际上并没有从模型中删除元素——只是从图表中简单删除了表示而已。

在ModelMaker中有关图表的最后一个考虑事项是：虽然ModelMaker提供了几个可视的向导程序和自动功能，但是它并不能够读取程序代码，并且也不能产生很具有吸引力的UML图表。在导入源代码和向图表中添加类时，需要排列符号，以便创建可用的UML图表。

建模和UML

UML（Unified Modeling Language，统一建模语言）是一种图形符号，用来表示软件项目的分析和设计及通信。UML是独立的语言，但是它可以描述面向对象的项目。UML不仅仅是一种方法，虽然读者可能更喜欢设计过程，但是UML还可以用做一种描述工具。

我的目标是从使用ModelMaker的Delphi程序员的角度来查看UML图表。对于UML的更深入讨论会超出本章的范围。

说明：我所看到的有关UML的最好资料是Martin Fowler的“UML Distilled”（Addison-Wesley，1999）。

类图表

ModelMaker所支持的最常用的UML图表是类图表。类图表可以显示多种类的关系，而且还可以显示一组类或对象以及它们之间的静态关系。例如，图11.1是一个类图表，它包含第2章“Delphi编程语言”中NewDate程序的类。当大家将这些类导入到ModelMaker中并创建自己的类图表时，结果可能会有所不同，记住我们前面所讨论的很多选项。许多设置可以控制可视类的显示。可以从相应的源代码文件夹中打开用于图11.1的ModelMaker文件（MPB文件）。

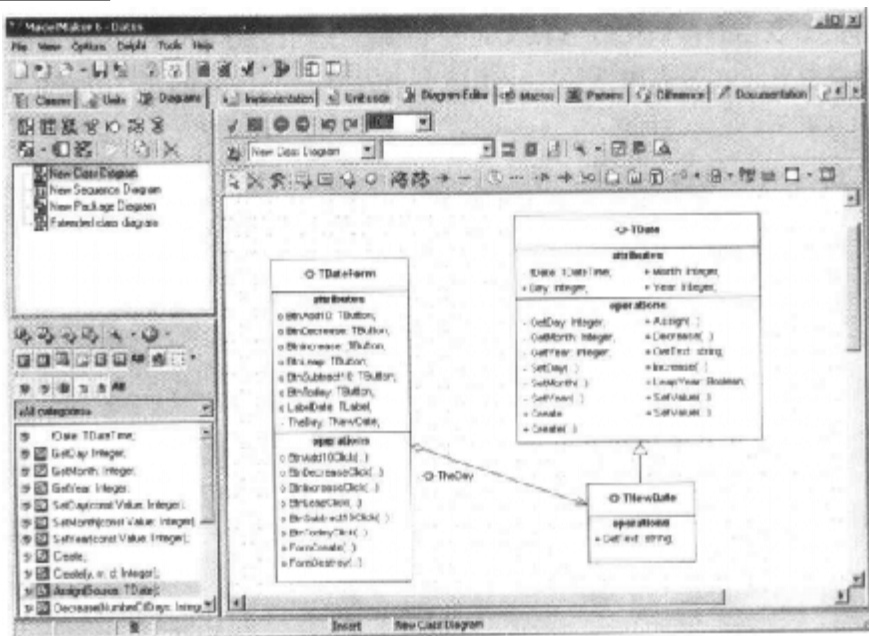


图11.1 ModelMaker中的类图表

前面曾经提到过，ModelMaker图表编辑器只是内部模型中的另一个视图。ModelMaker图表中的有些符号可直接映射到代码模型元素，而有些符号是不可以的。操作这些符号可以改变ModelMaker所产生的代码。在类图表中，可以向模型中添加新类、接口、字段、属性以及文档。同样，可以从图表内部更改模型中类的继承。同时，还可以向代码模型中没有逻辑表示的类图表添加一些符号。

ModelMaker中的类图表还允许编写接口代码和在更高级的抽象层上工作。图11.2用一个接口使用（IntfDemo）的复杂例子演示了类和接口的关系。附录C中所讨论的联机书籍介绍了这个例子，本章的源代码示例中也包含这个例子。

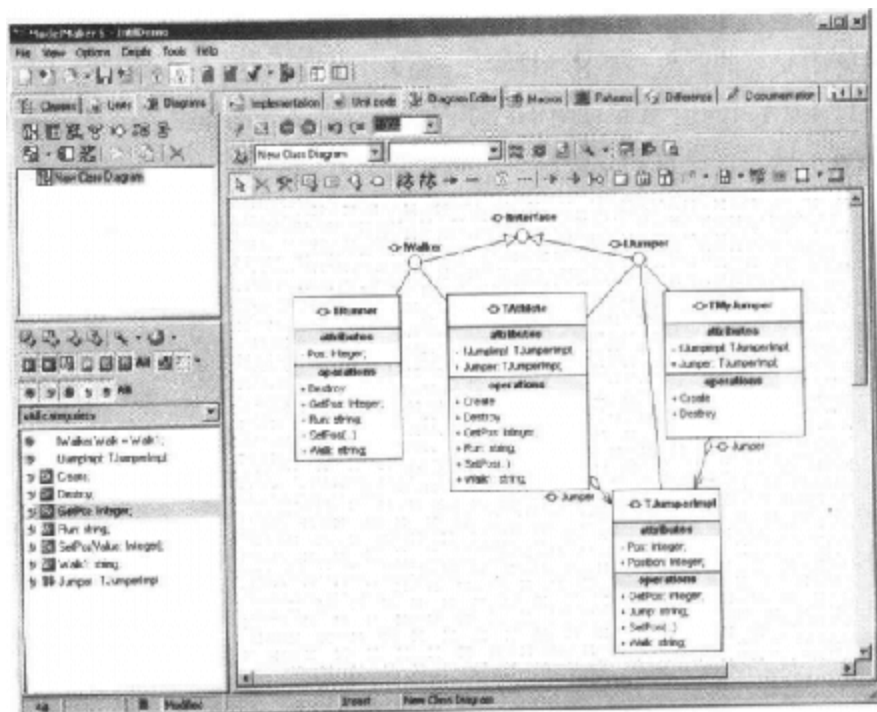


图11.2 包含接口、类和接口委派类图表

类图表和时序图都属于ModelMaker所支持的与代码模型关系最密切的UML图表。可以在ModelMaker中创建多个图表，并且图表符号与代码模型没有直接的关系，但是在时序图中，可以直接影响代码，就像影响模型类及其方法一样。例如，当为对象间的某个消息创建一个符号时，可以从属于容器对象的方法列表中选择，或者向对象添加新的方法，然后将这个新方法添加到代码模型中。

注意，正如以前所提到的，ModelMaker不会自动地根据输入代码创建时序图，需要开发人员自己创建。

应用分支和其他图表

我们已经讨论了两个最低层的UML图表，ModelMaker还支持其他更高级的UML图表，它们被设计用来提供从高级用户交互（应用分支图表建模）到低级类和时序图。应用分支图表属于最常用的图表之一，虽然这些符号与代码模型元素没有关系。这些图表可以建立软件功能的模型，并且它们使用起来非常简单。

一个简单的应用分支图表包含动作者（用户或应用程序子系统）以及应用分支（动作者要做的事情）。有关应用分支中的一个最常见问题是如何处理ModelMaker中的应用分支文本。应用分支文本是预分析时的下一步操作。例如，应用分支是对动作者要采取的动作的简单描述（“Preview Sales Report”或“Resize Window”）；应用分支文本是一个详细的、较长的描述性文字。ModelMaker不支持较长的应用分支文本，可以在连接到应用分支符号的图表中使用注释符号，或者将应用分支符号连接到包含这些应用分支文本的外部文件。本章后面将详细讨论这些技术。

ModelMaker所支持的其他UML图表如下：

协作图表 交互性图表，类似时序图。然而它们又有区别，在协作图表中，消息的顺序是由编号而不是时间来指定的，这就导致一个不同的图表布局。在这里，对象之间的关系有时看起来会更清晰。

状态图表 这种图表通过将所有对象的状态识别为接收到的消息来描述系统行为。一个状态信息应该列出一个对象的所有状态变换，并指出每次变换的起始和结果状态。

活动图表 这种图表描述系统的工作流程，特别适合于可视并行处理。

组件和部署图表 是一种实现图表。这种图表允许建立组件（实际上是执行模块、COM对象、DLL等等）之间的关系模型。在部署图表中，就是建立实际资源（节点）之间的关系模型。

非UML图表

ModelMaker支持三种非UML标准的图表，它们非常有用：

思想图表 由Tony Buzan在20世纪60年代创建，它是一种智力爆发、探索分支主题的出色方法，或者可以作为快速记录相关想法的方法。在表达过程中，常将这种图表用于一般数据的显示。

单元相关图表 该图表常用来显示ModelMaker的Unit Dependency Analyzer（单元相关性分析器）的结果。这些图表可以显示一个Delphi项目中单元关系的分支。

强健性图表 值得怀疑的是，UML规范为什么没有包含这种图表。这种图表可用来帮助桥接接口应用分支建模和特定实现时序之间的空隙，并能够帮助分析小组来验证他们的应用分支和查看实现细节。

公共图表元素

ModelMaker所支持的每种类型的图表都会包含特定于图表类型的符号，但是在Diagram Editor中的有些元素是所有图表都具有的。可以向图表添加图像和外形以及包符号（用来添加其他符号的容器）。

超链接符号允许向图表添加一个链接到其他实体的标签。实际上，多数图表符号都支持超链接功能。可以链接到其他图表（单击链接将在编辑器中打开所链接的图表），链接到代码模型中的元素（可以是类、单元、方法、接口，等等——单击链接将打开所链接元素的相应编辑器），或者链接到外部文档（这将在相应的应用程序中打开所链接的文档）。

每种图表都有三种类型的注解工具。有些文档解释了这些工具，可以添加一个独立的注解符号；还可以添加一个能够自动显示所链接对象的内部文档的注解符号；或者添加一个注解符号，输入到文本中，并将这个符号链接到对象。如果这样做，对象的内部文档将被更新来匹配注解符号的文本。

关系或关联符号默认是直线，但是通过选择符号并按Ctrl+O，可以转换它为垂直线。通过按Ctrl键并单击线条，可以向线条添加节点。ModelMaker会尽可能让这些符号相互垂直。

ModelMaker还提供了一些可视符号样式集。可以通过分层的方式来定义字体和颜色样式，并根据名称将它们应用到图表符号中。更详细的信息参见联机帮助文件中的“样式管理器”。

一个要注意的最常见功能是分层排列Diagrams列表（如图11.5所示）。可以添加一些文件夹用于组织，并“重父化”图表，为此，用Ctrl并用鼠标拖拉图表到新的父体。



图11.5 图表视图的组织

ModelMaker的图表功能包含了很多功能集，只要花时间来学习这些功能集，就会发现它能够转变我们的开发过程。对于Delphi开发者来说，Diagram Editor比多数UML编辑器（只是简单地产生静态图表）提供了更加动态的图示体验。

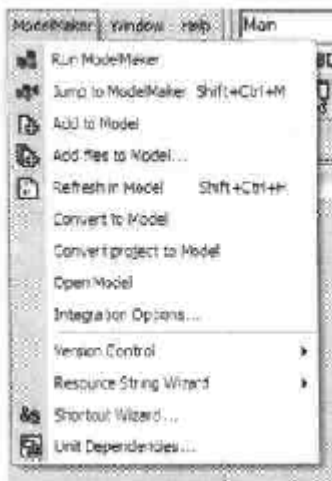
ModelMaker的编码功能

ModelMaker是一个功能强大的UML图表工具，可用于分析和设计工作，接下来我们将介绍ModelMaker所提供的其他功能。许多开发者将ModelMaker作为主要的开发环境来代替Delphi，其部分原因是ModelMaker是以可视方法来执行Delphi编程任务的，能自动完成编写Delphi类代码的重复任务。然而，这也是由于Delphi简化了代码开发，模糊了其中表示域和问题域之间的分隔。换句话说，很容易编写应用程序实现代码（实际所填入的代码）以及将它们放到窗体的事件处理程序中。这通常被认为不是一种很好的面向对象设计，但是ModelMaker有利于创建和重分解问题域对象，并便于从用户界面来重新组合这些对象。

在继续讨论这些问题之前，我们将首先讨论Delphi和ModelMaker是如何一起工作的。

Delphi/ModelMaker集成

正确安装Delphi之后，ModelMaker会向Delphi IDE添加一个菜单，标记为ModelMaker。



如果没有看到这个菜单，则需要基于Delphi中的向导程序来安装ModelMaker的DLL，在第1章末尾“安装新的动态链接库（DLL）向导”中详细介绍了安装方法。从该菜单可以控制ModelMaker，并迅速将代码导入到ModelMaker项目中。

该菜单的多数选项只有在运行ModelMaker时才可用。启动了ModelMaker之后（可以从Run ModelMaker菜单启动，或者以正常的方式启动），其他项才可用。这种集成菜单包含了添加代码到模型的多种方法：Add to Model、Add Files to Model、Convert to Model以及Convert Project to Model等都能使ModelMaker导入指定的单元。Add选项将单元导入到ModelMaker当前所装载的模型中，Covert选项将创建一个新模型，并把单元导入到其中。

Convert Project to Model是一个开始的好地方——确保备份了代码，然后选择该菜单项，同时在Delphi中打开项目。整个项目将被导入到ModelMaker的新模型中。

VCL位于何处？在ModelMaker中继承和导入代码

通过检查在ModelMaker中新导入的代码，可以发现，只导入了项目的部分单元。ModelMaker不会自动导入项目所使用的所有单元。如果这样做，则会创建一个很大的模型，其中包含许多不必要的类。但是，ModelMaker的继承功能需要祖先类存在于代码类型中（例如，如果正确设置，祖先类的更改会自动应用到其子孙）。

幸运的是，可以有很多选项来导入代码。虽然可以通过Delphi中的ModelMaker集成菜单（或者将单元从Windows Explorer中拖拉到ModelMaker中）来导入，但是ModelMaker的主工具栏有更多灵活的方法。照这样导入单元将会打开Import Source File对话框，这里可以设置导入代码的选项。利用这些选项，可以将部分VCL作为占位符类来导入，从而能够利用ModelMaker的继承工具，而不会增大模型。

该菜单中的Refresh in Model可以使得ModelMaker重新导入当前单元，这个时候最好讨论一下前面所提及的ModelMaker内部代码模型的结果。因为ModelMaker操作其内部模型，而不会操作外部源文件，所以通常可以发现模型和源文件都被编辑，但是结果是模型没有与源文件同步。当源文件被修改而模型没有被修改时，可以通过重新导入源单元来重新同步模型。但是如果模型和源文件都被修改，此时情况会更复杂。幸运的是，ModelMaker提供了一个强健的工具来处理同步问题。更详细的信息参见“差别视图”小节。

此集成菜单中的另一个选项是Jump to ModelMaker。选择这一项，ModelMaker会找到载入模型中的当前代码位置，并将ModelMaker放到窗口的最前面。

虽然可以从Delphi中控制ModelMaker，但是这种集成是两方面的。类似于在Delphi IDE中集成ModelMaker菜单，Delphi菜单也会出现在ModelMaker中。这个菜单中的命令包括允许从当前所选择的模型元素跳到Delphi中源代码的相应位置，并让Delphi执行同步检查和编译。这样就可以在ModelMaker中编辑、生成和编译源代码。

管理代码模型

现在该讨论ModelMaker中的编码问题了。由于ModelMaker内部代码模型具有对象特性，所以编辑代码模型元素通常要比在Delphi中更直观。例如，编辑类属性可以通过Property Editor对话框来实现，如图11.6所示。这是ModelMaker自动化的最好例子。当添加一个新属性时，没有必要担心添加私有状态字段的开销以及读或写方法，甚至是属性的声明。所需要做的就是编辑器中选择相应的设置，ModelMaker会创建支持类成员所需要的一切。这要比在Delphi IDE中通过Class Completion所提供的功能更可扩展。

注意，手动输入的属性是由对话框中的多种控件来表示的。Visibility、Type、Read和Write等都在该编辑器中管理，这有利于重分解（并不是排除一些重复输入性任务）。例如，因为ModelMaker将属性作为代码模型中的一个对象来管理，更改属性（例如，类型）都会导致ModelMaker将这些改变应用到它所知道的引用上。如果将读访问权限从字段更改为方法，则编辑器也会发生变化，ModelMaker会小心处理获取方法并更改属性的声明。最好的做法是，如果决定重命名属性或将它移动到另一个类中，则属性要拥有其自己支持的类成员：它们会被相应地重命名或移动。

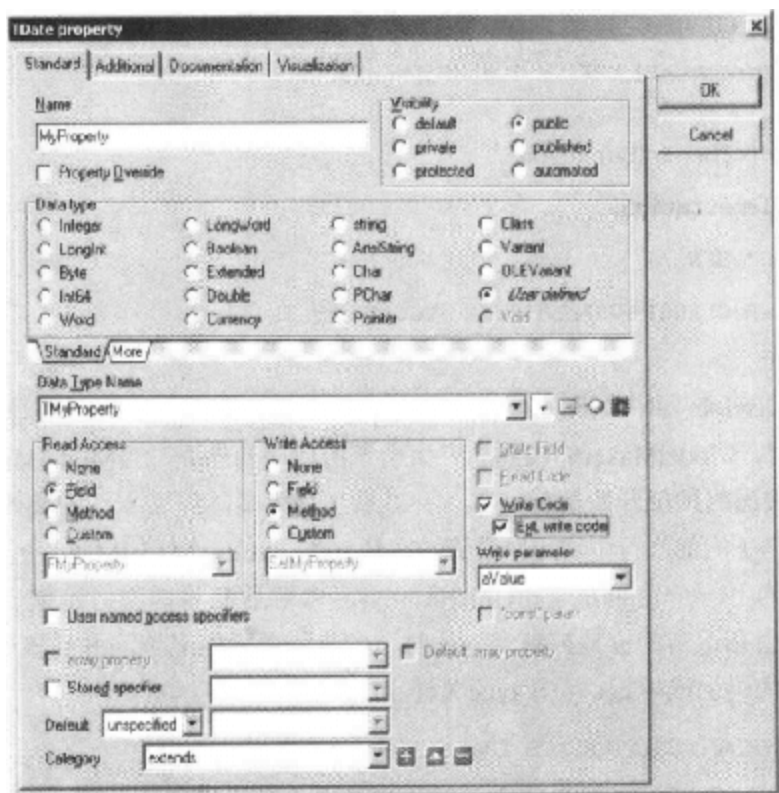


图11.6 Property Editor对话框

相同的方法可用于类成员类型；方法、时间、字段以及方法归结语句都有类似的编辑器。

当使用ModelMaker开发时，开发者在编辑类成员时无需考虑实现细节，只需要考虑接口的关系，因为ModelMaker会处理实现成员的多数重复任务（这不要与编写方法实现相混淆，仍然需要编写方法实现）。

单元代码编辑器

ModelMaker包含两种代码编辑器：实现类方法的编辑器（我们将在下一节讨论），以及Unit Code Editor（单元代码编辑器），这需要解释一下。实际上，ModelMaker是一种面向类/对象的工具——其有利于管理类级别的代码。对于不属于类的代码（非类的类型声明、元类声明、单元方法和变量，等等），ModelMaker利用了一种实用的办法。当ModelMaker导入一个单元时，保存在代码模型中的所有内容都会被相应处理，留下来未处理的都会在Unit Code Editor中显示（通常，对于新用户来说，这包含方法实现中没有的一些文档——但是ModelMaker也能够将项目反向转换为文档）。

下面是在Unit Code Editor中所看到的一个例子：

```
unit <!UnitName!>;

interface

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

```

type
MMWIN:CLASSINTERFACE TDateForm; ID=37;

var
    DateForm: TDateForm;

implementation
    {$R *.DFM}

MMWIN:CLASSIMPLEMENTATION TDateForm; ID=37;
end.

```

这些代码对于Delphi程序员来说是含糊不清的，显然它不能够编译。可以看到，当扩充或产生单元代码时，ModelMaker的代码会产生引擎所使用的外壳。当ModelMaker产生一个单元时，它从该代码的顶部开始解析文本，并查看三种事情：纯文本、宏或代码生成标记。

在这个例子的前几行中，可以看到一些纯文本：**unit**。ModelMaker会原封不动地发出纯文本。该行的下一个符号是宏<!UnitName!>，本章后面将详细讨论宏。现在，只需知道ModelMaker会在适当的位置扩展宏。此时，宏表示单元的名称，并且该文本也会被发出。

最后是代码生成标记，位于**type**关键字的下面：

```
MMWIN:CLASSINTERFACE TDateForm; ID=37;
```

在这里，该标记告诉ModelMaker在单元代码的这个地方展开TDataForm的类接口。

这样，当在Unit Code Editor中编辑代码时，可以看到由我们自己管理的和ModelMaker管理的代码混合体。当编辑该代码时，要注意不要混淆ModelMaker所管理的代码，除非知道自己在做什么。这与在Delphi的DPR文件中编辑代码类似——如果不小心，很快就会有麻烦。然而，这里是添加非类类型声明（例如，枚举类型）的好地方。可以像在Delphi中做处理一样，应在单元的**type**部分添加类型声明：

```

unit <!UnitName!>;

interface

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

type
    TmyWeekday = (wdSunday, wdMonday, wdTuesday, wdWednesday,
        wdThursday, wdFriday, wdSaturday);

MMWIN:CLASSINTERFACE TDateForm; ID=37;

var
    DateForm: TDateForm;

implementation
    {$R *.DFM}

MMWIN:CLASSIMPLEMENTATION TDateForm; ID=37;
end.

```

在这个例子中，ModelMaker会像我们所输入的那样发出类型声明，然后开始展开TDateForm类声明。

ModelMaker在Unit Code Editor中提供了一些工具来管理单元层的方法，当处理大型的例程库类型的单元时，这些工具可以提供很大的便利。也就是说，使用ModelMaker可以利用其强大的重分解功能来对象化一些例程。

方法实现代码编辑器

ModelMaker的Method Implementation Code Editor（方法实现代码编辑器，如图11.7所示）完全不同于Unit Code Editor。该编辑器利用三分之二的屏幕。在这里的例子中，我们已经添加了一个虚构的属性，名为MyNewProperty，并允许ModelMaker产生状态字段和访问方法。写访问方法在编辑器中处于激活状态。

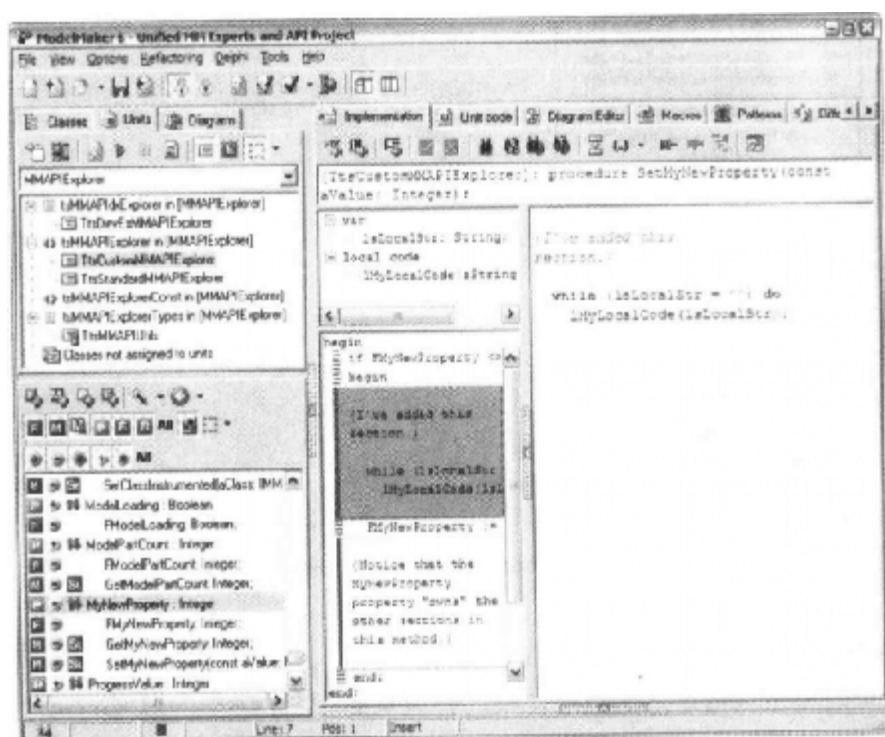


图11.7 ModelMaker的Implementation选项卡

右边紧靠代码编辑器的是两个有趣的窗口。上部的三视图是局部代码浏览器，可用来管理局部变量和局部过程。下面是Section List，ModelMaker允许用户在方法实现中将代码划分为片段。在某种程度上，这是一种便于组织的方法，但是更为重要的是，它允许ModelMaker控制代码的特定片段。就像ModelMaker能够有部分模型一样（类似于为属性自动生成的属性访问方法），它还可以在方法中拥有代码片段。这常常发生在已经选择了让ModelMaker在属性访问方法中生成读或写代码时。注意，在这个例子中，第一个、第三个和第五个片段都有红色和白色的虚线左边框，表示它们属于ModelMaker。具有绿色边框的片断属于用户。当ModelMaker生成该方法时，这个代码将按照顺序一个片段接一个片段地发出。

警告：在ModelMaker Implementation中编写代码的一个最大缺点就是没有Delphi IDE所提供的Code Completion窗体。

差别视图

前面我们曾经提及，很容易就会发生模型与源文件不同步的情况。如果模型和源文件都被编辑，则不能够简单地从模型中再次生成文件，以免覆盖源文件的修改。同样，也不能够重导入单元，以免消除模型的更改。幸运的是，ModelMaker提供了强大的区分差异的工具。当发现模型不同步时，就可以查看Difference选项卡（如图11.8所示）。

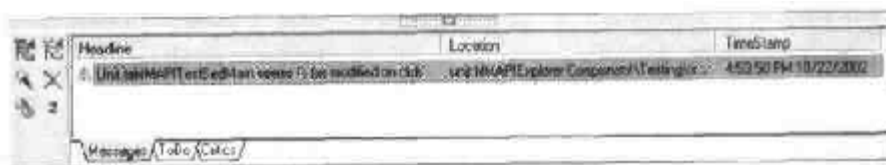


图11.8 ModelMaker的Difference视图

ModelMaker提供了多种方法来查看差别。可以查看标准文本文件的比较、时间戳的区别或者甚至是模型内两个所选类的比较。我喜欢的是图11.8中的显示——结构化的差别。ModelMaker会临时导入磁盘上的源文件（这就像内部代码模型一样使之具体化），并在对象和属性层上比较导入文件与模型中的相同单元和类，而不是比较文本，因此这种比较相对要快一些，而且更加精确。

注意图11.8的树状视图中表示差别的图标。红色◇表示模型和源文件都包含指定的类成员（例子中是btnUITestClick），但是两个实例是有差别的。有差别的代码显示在右边的备忘录控件中。树视图中的绿色“+”表示指定的类成员只存在于模型中，磁盘上没有。蓝色“-”表示类成员只存在于磁盘上，模型中没有。利用这些信息，可以选择如何继续处理模型的再同步问题。一个比较好的功能是可以从Difference视图中重新导入所选方法（而不是整个单元）。

这种方法意味着，重要的是知道何时模型不同步，以便在重新生成源文件时不会覆盖磁盘上的更改。幸运的是，ModelMaker提供了几个安全措施可以防止这种情况发生。一种措施是Design Critic（参见本章后面的“小花絮”）。如果启用了Design Critic，则当磁盘上的文件发生变换时，ModelMaker的Message View会发出警告，如下图所示。



事件类型视图

ModelMaker在Events选项卡中管理事件类型，在这里可以编辑事件类型的声明。要记住，虽然新的事件类型可以存在于ModelMaker的内部代码模型中，但是在将事件类型的声明添加到单元之前，它是不可能存在于源文件中的。管理该过程的最简单方法是将Events视图列表中的事件类型声明拖曳到Unit列表中，并将该项投放到单元中。

文档和宏

ModelMaker支持软件文档功能。在使用该功能之前，需要掌握一个重要的概念（幸运的是，它并不复杂）。在ModelMaker中，文档不等同于注释。开发人员可以利用源代码注释来完成复杂的事情，但是必须采取一些步骤，使得ModelMaker发出（或导入）这些注释。实质上，每个模型元素（类、单元、成员、图表符号，等等）都具有文档，但是元素的文档输入不会自动使得文档出现在源代码中。文本是附加在模型中元素上的，但是必须使ModelMaker生成包含文档的源代码注释。

文档与注释

ModelMaker代码模型中的每个元素都具有两种类型的文档：标准的、大型文本块文档和简短的一行文档（如图11.9所示）。这些文本在ModelMaker的上下文中有许多目的。正如前面所提到的，它们并不直接等同于代码注释，虽然有些注释可以从这些文档中产生。

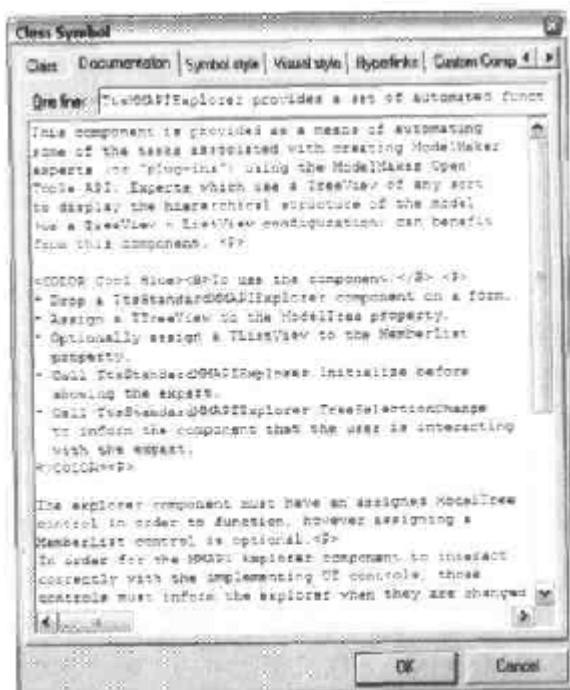


图11.9 Class Symbol的Documentation选项卡

在这个例子中，一个类具有两种类型的文档。这个文档可以出现在图表中，作为附加的批注（可以使用标准文档或一行文档），或者可以指定一种或两种类型的文档作为源文件的注释。为此，需要使用ModelMaker的功能强大的宏（下一节将讨论），并更改一些项目选项。这里，我们不必担心宏（接受默认设置），而只查看项目的一些选项。

从Option菜单中选择Project Options，可以打开Project Options对话框，并选择Source Doc Generation选项卡。这里有许多选项是关于从ModelMaker文档中生成源代码注释的（有关更详细的信息，可参见联机的ModelMaker帮助）。要查看当前的源代码注释，可以从In Source Documentation Generation组的Method Implementation部分选择Before Declaration。现在，包含文档的任何方法都会使用默认的宏来产生源代码注释。

ModelMaker可以从源文件单元中导入注释，并将这些注释关联到相应的代码模型元素。为此，必须在Document Import Signature（参见Project Options对话框中的Source Doc Import选项卡）中使得注释有意义，并告诉ModelMaker要将哪些行导入到模型中。因而，如果方法实现具有类似下面的注释，则可以告诉ModelMaker忽略前5行，并只导入实际的注释文本：

```
{*****  
TMyClass.DoSomething  
Returns:   String  
Visibility: Public  
Comment:  
    This is the actual comment that we want ModelMaker to import. The  
    first 5 lines of this comment block should not be imported into  
    the model.}
```

在为源注释配置ModelMaker时，要密切注意注释蔓延。这常常发生于注释的导入和导出设置不匹配时。例如，如果控制源注释输出的宏向注释添加6行，然后再添加文档文本，但是导入设置时仅删除了5行，则每个导入/生成周期会向注释中添加一个冗余的文本行。

使用宏

宏表示ModelMaker的某个关键功能，学习宏比较容易，但是掌握它比较难。一个宏就是一个标识符，它表示了一个文本块。当ModelMaker遇到宏时，它会用宏所表示的文本来替换宏名。

前面在Unit Code Editor中已经看到了这个过程。<!UnitName!>在代码生成时被所生成的代码名称所替换。这是一个实体特定的宏示例，根据所生成单元的不同，替换结果也是不同的。宏UnitName是预先定义好的，但是结果会随着上下文而变化。

ModelMaker包含许多预定义的宏（用户手册的第75页有完整的宏列表，Delphi随配光碟中的usermanual620.pdf文件就是用户手册）。在Macros选项卡中可以创建自己的宏（甚至嵌套宏）。还可以覆盖某些预定义的文档扩展宏。例如，如果启用方法实现文档，但是没有提供宏，那么ModelMaker会使用内置的宏来生成注释。但是，如果声明并定义了名为MemberImpDoc的宏，则ModelMaker在生成方法注释时就会使用该宏（有关用于源文档生成的宏的详细内容，可参见ModelMaker联机帮助中的主题“文档宏”）。

不仅可以在代码生成时使用宏，而且还可以在代码编辑器中输入时扩展宏。此时，可以参数化宏，以便当ModelMaker扩展宏时，可提示输入值。这些值将被插入到所扩展的文本中。

重分解代码

重分解代码是计算机编程中比较流行的术语之一。实际上，重分解是改进现有代码并且不改变外部行为的过程。重分解过程不是单一的过程，它只是一个任务——尝试改进现有代码并且不破坏当前环境。

有关重分解概念的讨论很多，这里我们只介绍ModelMaker帮助用户重分解代码的特定方法。再次重申，ModelMaker的内部代码模型起着重要的作用，记住，在ModelMaker中进行开发不仅仅是开发面向对象的代码，而是借助对象定向的一种开发过程。因为所有这些代码模型元素都被内部保存为对象（相互之间有引用的对象），而且每次选择产生代码时，源代码单元都要完全从该模型中重新生成，所以代码元素的任何变化都会被立刻传递到类中。

比较好的例子是类的属性。如果有一个名为MyNewProperty的属性并有读/写方法（由ModelMaker维护，相应的名称为GetMyNewProperty和SetMyNewProperty），且想重命名该属性为MyProperty，则只需一步即可完成：重命名属性。ModelMaker会处理剩下的工作——将访问方法自动重命名为GetMyProperty和SetMyProperty。如果该属性在图表中显示，图表将会自动更新来显示此变化。（警告：ModelMaker不会自动搜索代码中的MyNewProperty实例——需要用户自己使用ModelMaker执行全局搜索并替换。）这个例子虽然很简单，但是它演示了ModelMaker如何简化重分解的任务：只需要移动和重命名代码元素，其他细节便由ModelMaker来完成。现在我们来查看详细的情况：

简单的重命名 这个任务非常简单，上面我们已讨论过，但是不应该频繁使用。代码模型元素名称的变化由ModelMaker通过代码模型传递到它所知道的所有元素实例。

重父化类 这个非常简单的过程可以有多种方法来完成。最常见的是可以在Class视图中将类从一个父体拖拉到另一个父体（在类图表中也可以，就是将一般箭头从老的父体拖拉到新的父体），现在该类就具有一个新父体。如果继承被限制，ModelMaker会自动更新子类的继承方法来匹配父体的声明。下次生成代码时，这些更改会自动出现。

在单元之间移动类 移动类到一个新单元中也非常简单。在Unit视图中，将类从当前位置拖拉到新的单元，便会在新单元中重新生成所有相关代码（声明、实现和注释）。

在类之间移动成员 在重分解过程中，这个过程被称为“对象之间的移动功能（或职责）”。想法很简单：随着开发的进展，可以发现某些功能（由类成员实现）可能需要移动到其他类中。可以通过拖放鼠标来实现。从成员列表中选择所需要的类成员，拖放它们到Classes的新类中（如果同时按下Shift键，则是移动，而不是复制）。

转换成员 这是ModelMaker的一个比较出色的功能。在Member List中右键单击某个成员，将显示一个弹出菜单，其中包含Convert To菜单项和子菜单。选择其中一个子菜单，可以转换现有类成员的类型。例如，把一个私有字段名FMyInteger转换为属性时，ModelMaker将自动创建一个公共属性MyInteger，它将读写FMyInteger。同样，可以将这个字段转换为一种方法——返回值是整数的一种私有函数，名为MyInteger。

有限继承 在方法编辑器对话框中有一个**Inheritance Restricted**复选框。如果选中该复选框，**ModelMaker**就不允许更改方法的多数属性，因为这些属性是基于祖先类的重载方法实现而设置的。如果更改祖先类中的方法声明，这些更改将会被自动应用到子类中，这里重载方法被设置为有限继承。

如果读者有过重分解的体验（或者熟悉最新版本的**JBuilder**），就不会感到重分解工具很特别。然而，如果单独与**Delphi**中的功能相比较，则会发现它是一个出色的功能集。如果对**ModelMaker**所提供的功能不满意，可以根据需要扩展重分解功能。

说明：此外，我可以偷偷地告诉你们，我已经看到了**ModelMaker**未来版本的测试版（在编写本书的时候），其中包含了新的重分解工具集，它们的功能非常出色。

应用设计模型

ModelMaker将所有功能组合在一起支持设计模型（有关设计模型的详细讨论，已经超出了本章的范围。如果读者不熟悉这个模型，可以参见下面的附加说明“设计模型101”）。**ModelMaker**提供了应用模型实现的简单方法，只需要单击鼠标即可。根据所选择的模型，操作方法有所不同——有些模型在添加代码之前会显示一个向导，有些模型只是简单地向所选类添加成员。正如前面所讨论的，这些新成员都属于**ModelMaker**，并且很容易更新。此外，如果选择不应用模式，则**ModelMaker**会删除添加到模型中的所有类成员。

我们看下面一个**Singleton**模型的例子，假设有一个类，并至少需要该类的一个实例，下面是示例类：

```
type
  TOneTimeData = class (TObject)
  private
    FGlobalCount: Integer;
    procedure SetGlobalCount(const Value: Integer);
  public
    property GlobalCount: Integer read FGlobalCount write SetGlobalCount;
  end;
```

Singleton模型要求使用单个入口点（这种模型的**ModelMaker**实现中的类函数**Instance**）来获得类的单个实例的访问权限。如果实例并不存在，则要创建并返回它，否则将返回现有实例。因为**Instance**是入口点，所以可以禁止使用这个类的**Create**。一旦对**ModelMaker**中的类应用**Singleton**模型，它就会出现：

```
type
  TOneTimeData = class (TObject)
  private
    FGlobalCount: Integer;
    procedure SetGlobalCount(const Value: Integer);
  protected
    constructor CreateInstance;
    class function AccessInstance(Request: Integer): TOneTimeData;
  public
```

```

constructor Create;
destructor Destroy; override;
class function Instance: TOneTimeData;
class procedure ReleaseInstance;
property GlobalCount: Integer read FGlobalCount write SetGlobalCount;
end;

```

这里我没有列出方法实现；要查看它们，可以应用模型，或者查看PatternDemo示例的源代码。

警告：ModelMaker用来实现Singleton模型的代码要基于方法中恒量的使用来模仿每个类数据。但是，如果没有启用Assignable Typed Constants Delphi编译器选项（默认时这个选项是禁用的），这个代码会编译失败。

设计模型101

程序员关心的是实现特定的类，而设计者关心的是使得不同的类/对象能够一起工作。虽然很难给出软件设计的准确定义，但是本质上它是程序整体结构的组织。查看不同人对不同问题的设计方案，可以注意到一些相似的和共同的元素。模型就是一个通用的设计，用一种标准的方法来表示，并且可以应用到很多情形中。与设计模型关联更多的是设计重用，而不是代码重用。虽然模型的代码解决方案来自于程序员的灵感，但是在设计中我们实际所关注的是：即使可能必须重写代码，也要从一个清晰的被验证过的设计开始，这可以节省大量时间。设计模型不涉及原始构件块（例如，哈希表或链接的列表）或特定于域的问题（例如，分析模型）。

模型运动的创始人不是软件设计者而是建筑师，他更注重的是建筑中的模型使用。“每个模型都描述了环境中发生多次的问题，然后描述该问题解决方案的核心，这样就可以多次使用这种解决方案，而无需重复工作两次。”此话出于Erich Gamma、Richard Helm、Ralph Johnson和John Vlissides所编写的有关软件模型的书《设计模型：重用面向对象软件的元素》（Addison-Wesley, 1995），它掀起了软件行业的模型运行。这些作者被称为“Gamma et al.”，但是更常见的称呼是“Gang of Four”（四人邦）或简称GoF。本书通常也被称为“GoF书”。

本书描述了软件模型的概念，指出了描述它们的方法，并提供了23种模型，分为3组：创建、结构和行为。许多GoF模型可以在C++中实现，有些可以通过Smalltalk来实现，但是经过提取和移植，它们也可在Java或Delphi中使用。

模型的代码结构如下：

- pattern name（模型名）是重要的，当与其他程序员和设计者讨论时，用名称表示模型。
- problem（问题）描述了何时应用模型，实际上包含上下文和条件。
- solution（解决方案）描述了设计中的元素及其关系。这不是实现，只是类功能和协作的抽象描述。
- consequences（结果）是应用模型之后的结果，包括空间和时间限制。

当前，没有一本书从Delphi的角度来介绍模型。但是，Delphi杂志（包括Delphi Informant和The Delphi Magazine）上有很多关于此问题的文章。经典的GoF模型来自于很多文章的灵感，并且ModelMaker文档中有对这些模型的详细说明（参见其Web站点，并下载它）。

对于有些标准模型的Delphi实现我并不认可。实际上，我强调的是底层设计以及在将GoF实现（通常是C++或Java）移植到Delphi中时，如何保留它以及如何使用其特定的语言功能。其他作者可能倾向于移植代码，这是惟一实现设计的方法。学习模型是重要的，因为这将与其他程序员一起开发一个常用的术语，并学习应用OOP技术的更好方法（特别是封装和低耦合）。

最后一点要说明的是，在Delphi中多数模型是使用接口而不是使用类来实现的（ModelMaker也是这样做的，这遵循标准的方法）。

ModelMaker提供了几个其他模型的实现，包括Visitor、Observer、Wrapper、Mediator和Decorator。它们被硬编码在ModelMaker中，并以特定的方法来使用，其中有些实现方法会更好，这在开发者中有争论，起因是ModelMaker支持另外的应用模型方式：代码模板（下小节将讨论）。但是，不要小看ModelMaker对现有模型的支持；它们为这些常见问题的Delphi实现提供了很好的解决方法。

代码模板

ModelMaker中的另一个强大功能是代码模板，使用该技术可创建自己的设计模型实现。代码模板类似于部分类（应用到其他类的类）的快照。换句话说，它是类成员的集合，一起保存在可以添加到其他类的模板中。然而，这些模板可以被参数化（类似宏），这样当对类应用这些模板时，会弹出一个对话框，要求用户输入值，然后该值将被应用为模板的一部分。

数组属性就是这样的一个例子。在ModelMaker中声明属性非常简单，但是完全实现需要几个步骤：不仅必须具有数组属性，而且必须具有TList或其子类来包含数组元素，以及提供保存元素的方法。即使是这个简单的例子，也需要一些工作来获得数组属性。进入到数组属性模板。在ModelMaker中打开一个模型（或者新建一个新模型并添加TObject子节点），然后选择要添加新数组属性的类。右键单击Member List，并选择Code Templates。现在应该会出现一个浮动的工具栏，名为Code Templates（注意，在Patterns选项卡中会有相同的工具栏）。单击Apply Array Property Template按钮，打开Code Template Parameters对话框。它包含一个项目列表，这些项目被指定到要应用的模板中，如图11.10所示。选中左边列表中的项目，然后按F2键，以编辑这个参数的值，接收默认设置，并单击OK。

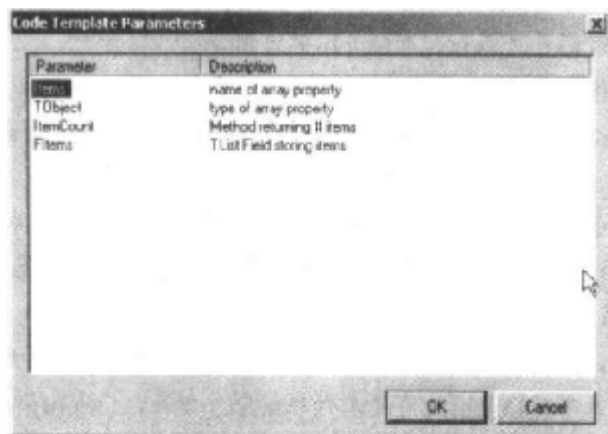


图11.10 ModelMaker的Code Template Parameters对话框

现在的类应该包含下列成员：

```
private
    FItems: TList;
protected
    function GetItemCount: Integer;
    function GetItems(Index: Integer): TObject;
public
    property ItemCount: Integer read GetItemCount;
    property Items[Index: Integer]: TObject read GetItems;
```

可以看出这项技术非常灵活。其他一些常见任务和自己的设计模型实现都是很容易实现的，下面介绍如何实现。

要创建自己的代码模板，可以从现有类开始，这个类中有我们想转换为模板的成员。选择该类，然后在Member List中选择希望使用的成员（这可以是任何类型的成员）。在Member List中右键单击并选择Create Code Template，Save Code Template对话框会出现。它非常类似于Save As 对话框（在这里是指定模板要保存的地方），但是我们还可以详细指出模板出现的地方。记下最后的确认消息，如果愿意，可以更改调色板位图。

新模板现在位于模板调色板中，可以将这个模板添加到类。要参数化该模板，必须改变保存模板时所创建的PAS文件。例如，下面是用于Array Property模板的ArrayProp_List.pas文件的一部分：

```
unit ArrayProp_List;

//DEFINEMACRO:Items=name of array property
//DEFINEMACRO:TObject=type of array property
//DEFINEMACRO:ItemCount=Method returning # items
//DEFINEMACRO:FItems=TList Field storing items

TCodeTemplate = class (TObject)
private
    <!FItems!>: TList;
protected
    function Get<!ItemCount!>: Integer;
    function Get<!Items!>(Index: Integer): <!TObject!>;
public
    property <!ItemCount!>: Integer read Get<!ItemCount!>;
    property <!Items!>[Index: Integer]: <!TObject!> read Get<!Items!>;
end;
```

注意以//DEFINEMACRO开始的行，这是声明参数的地方，它们将出现在Code Template Parameters对话框中。每一行都是Name/Value对：“=”符号左边的元素是可编辑的值，右边的元素用来解释参数的描述。

提供了参数之后，它们就可以作为模板代码中的宏，如下面的一行代码所示：

```
property <!Items!>[Index: Integer]: <!TObject!> read Get<!Items!>;
```

当这个属性被作为模板的一部分而添加到类时，宏（例如<!Items!>）将被相应参数的值所替换。这样一来，就可以使用参数来自定义自己的代码模板。

小花絮

为了帮助大家学习，这里列出一些有趣的功能：

垂直重思考 通过按Ctrl+O，可以将Diagram Editor中的默认直线对角线更改为垂直线。通过按下Shift+Ctrl+O，还可以强迫ModelMaker查找最短的可能垂直路径。

可视样式管理器 这个管理器（位于图表视图的快捷菜单中）值得花一些时间来学习。可以为图表符号定义很广泛的层次关联的可视样式，并随时应用它们。而且，不要忘记单击Diagram Editor中的Use Printing Style按钮，删除不可打印的元素，使得图表看上去与打印出来的一样。

设计检查 设计检查是ModelMaker中给人影响较深的QA功能。它们是在后台运行的校对程序，帮助检查代码。要启用它，首先要确保启用了Show Messages (Shift+Ctrl+M)，右键单击Message视图，并选择Show Critics Manager。建议打开时间戳设计检查，因为如果磁盘上的源文件在ModelMaker外面发生变化，它会发出警告。此外，还可以通过ModelMaker的OpenTools API来创建自己的设计检查。

创建向导 对于繁忙的Delphi程序员来说，这是又一个自动化的功能。Creational Wizard（创建向导，可以通过Member List中的Wizards按钮来访问它）将检查模型中被实例化的类成员，并将它们添加到相应的构造函数或析构函数中。它还可以完成其他一些事情，并且会有一些提示；在该向导中时按F1键可以访问联机帮助。

Open Tools API 这非常类似于Delphi的Tools API，这个ModelMaker功能允许创建插件。该API功能强健，并且可以访问图表和整个代码模型。以这种方式展开ModelMaker可能有点太过极端。

小结

本章概述了ModelMaker的功能，并讨论了一些不相关的主题，例如UML图表、模型、重分解和开发者文档。当然，粗略讨论这些方面的内容是有原因的：ModelMaker能够在这些方面帮助用户，而Delphi IDE本身没有提供对这些功能的支持。

访问其Web站点和其他一些文档可以更多了解其他技术。前面提及过，ModelMaker提供了很多文档，包括图表和模型等材料。访问该产品Web站点www.modelmakertools.com，可以下载比默认安装更多的附件。

如果想知道所讨论技术的更多内容，或想知道如何开始，可以参见ModelMaker的联机帮助以及用户手册（位于Delphi 7的随配光碟上）。我们还推荐了Thoughtsmithy Web站点，这里有Robert Leahey编写的“开始使用ModelMaker”学习指南，其地址如下：

www.thoughtsmithy.com/mmjump/MMGettingStarted_Intro.html

接着，我们将讨论Delphi的应用程序结构。在第12章深入分析Windows操作系统中Delphi所完全支持的COM相关技术。之后，将深入研究数据库编程。

第12章 从COM到COM+

从Windows 3.0操作系统发布至今的10年中, Microsoft 一直致力于将其开发的操作系统和API建立在真实的对象模型上, 从而取代传统编程方式中建立在函数上的做法。基于这种指导思想, Windows 95及其后续产品Windows 2000操作系统都应该是建立在这种具有革命性的方法之上的。然而, 实际情况并非如此, 不过Microsoft却依然始终如一地推销着其COM(组件对象模型, 即Component Object Model)技术。例如, 它们将该技术内置于Windows 95操作系统的外壳程序顶端, 使应用程序与COM紧密地结合在一起并从COM中派生出相应的设计技术(例如, 自动化功能), 而这一技术发展到了Windows 2000操作系统时已达到了顶峰。

现在, 在发布了高级COM编程所需的全部基础要求后, Microsoft又下定决心转而开发一种新的核心技术, 而这一技术也正好是新版.NET操作系统的一部分。就笔者的感觉而言, 尽管COM技术成功地提供了一个用来集成应用程序或大型对象的体系结构, 但是在实际应用当中, 它并不是很有利于对象间集成化工作的。

在本章中, 读者将要构建自己的第一个COM对象, 并了解一些基础的元素, 以便在无需过于深入学习的前提下就可以理解这种技术所扮演的角色; 而本书在以后还将会继续介绍与自动化功能(Automation)和类型库有关的内容。通过这些介绍, 读者将逐渐掌握利用自动化服务器和客户端来处理Delphi 7数据类型的方法。

本章的最后还将介绍嵌入对象的使用方法, 并学习OleContainer组件的使用方法, 以及如何开发自己的ActiveX控件。此外, 本章还将引入无状态的COM(MTS和COM+)技术, 以及其他一些高级编程思想, 其中包括Delphi 7为.NET操作系统所提供的集成支持功能。

本章主要包含以下内容:

- 什么是COM
- COM、GUID和类厂(class factory)
- Delphi接口和COM
- VCL COM支持的类
- 创建和使用自动化服务器
- 使用类型库
- 容器(Container)组件
- 创建ActiveX和ActiveForm
- 介绍COM+
- COM和Delphi 7中的.NET

OLE和COM技术历史简介

COM技术一直存在着困扰人们的模糊地方,即由于早期出于市场的考虑,Microsoft使用了多个不同的名称。实际上,所有以对象链接和嵌入(简称OLE)技术开发的部件,都是DDE(动态数据交换,即Dynamic Data Exchange)模型的扩展。使用剪贴板(Clipboard)可以为用户提供拷贝原始数据的功能,而使用DDE技术则允许用户将两份文档中的部分数据连接起来。OLE允许用户从服务器应用程序中将数据复制到客户应用程序中,其中包括那些与服务器相关的数据,或其他一些保存于Windows注册表中的引用信息。原始数据有可能会与其链接一起被复制(即对象嵌入),或者始终与原有文件中的内容保持一致。现在,OLE文档通常又被称为活动文档(active document)。

随后,Microsoft将OLE更新为OLE 2,并重新实现了这一技术,OLE 2并不仅仅是DDE技术的进一步扩展,而是在该技术中又添加了很多新技术,例如,OLE自动化功能和OLE控件。接着还为用户创建了大量应用了OLE技术的Windows 95外壳程序和接口,并将OLE控件重新命名为ActiveX控件(以前的OLE控件被称为OCX)。此外,在OLE 2中还更改了相应的规范,以便于对借助Internet进行发布的操作进行一些简单的控制。在一段时间内,Microsoft曾经想将其ActiveX控件推广为Internet上的主要控件,但是这一计划一直没有能够完全地被软件开发界所接受——最起码ActiveX技术并不适合于网络开发的要求。

由于这一技术得到了很大的扩展,所以它逐渐成为Windows平台中的重要组成部分,于是,Microsoft又再次将其名称更改回OLE,随后又改为COM,并在Windows 2000操作系统中将其最终更改为COM+。实际上,更改这一技术名称的原因只有很少的一部分是因为对该技术进行了修改,相反市场需求才是促使这一现象发生的重要因素。

那么COM的含义到底是什么呢?从本质上来看,组件对象模型(Component Object Model)技术为客户模块和服务器模块之间借助指定接口进行通信这一工作定义了一种标准的方法。这里模块(module)一词指的是应用程序或库(例如DLL)。相互通信的两个模块也可能位于相同的计算机上,或者位于借助网络相连的不同计算机上。此时,也可能会出现多接口的情况,这取决于客户端和服务器端各自所扮演的角色。此外,开发人员也可以根据自己的具体目标来添加新的接口。而这些接口是由服务器对象完成的。服务器对象通常都会实现多个接口,而且所有的服务器对象都会拥有一些公共的功能,因为它们都必须实现IUnknown接口(相关内容可参见第2章“Delphi编程语言”)。

令人高兴的是Delphi现在可以与COM技术完全兼容。早在Delphi 3时,它所提供的COM功能就已经非常简单了。并且在COM技术的集成方面,它比C++或其他语言做得更好,以至于Windows 2000 R&D小组都这样评价:“我们应该像Delphi那样来提供COM功能。”Delphi之所以可以轻松提供COM功能,主要是因为它将各种接口类型集成到了Delphi语言中(相应地,这些接口还被用于集成Windows平台中的Java和COM技术)。

正如前面所介绍的那样,创建COM接口的目的就是为了实现两个软件模块之间的通信功能。这两个软件模块可以是可执行文件,也可以是DLL。通过DLL来实现COM对象非常简单。因为,在Win32中,一个应用程序和其使用的DLL都将被保留于相同的内存地址空间中。这意味着,如果应用程序向DLL传递一个内存地址,那么这一做法是合法而有效的。而

当需要在两个可执行文件中实现通信时，则需要COM接口在后台完成大量的工作才能实现程序间的通信操作。该机制也被称为配置编组（准确地讲，如果客户端是多线程的，也许要应用这一技术）。注意，用于实现COM对象的DLL被称为进程内（in-process）服务器，而那些分别执行的服务器则被称为进程外（out-of-process）服务器。但是，当DLL在另一台计算机（DCOM）上或主机环境（MTS）中执行时，它们通常也被称为进程外服务器。

实现IUnknown

在开始介绍COM开发示例之前，首先引入一些与COM相关的基础知识。每个COM对象必须实现IUnknown接口。而在Delphi中则通常将那些非COM接口简称为IInterface（如在第2章中介绍的那样）。该接口是一个基础接口，Delphi中的每一接口都是在该接口基础上派生出来的，同时Delphi还为IUnknown/IInterface接口提供了一组不同的、现成的实例，其中包括TInterfacedObject和TComObject。这里介绍的第一个实例可以用来创建与COM无关的内部对象，而第二个示例则可以用于创建那些被服务器导出的对象，正如本章后面所介绍的那样。此外，还有其他一些类也是从TComObject那里继承而来的，它们提供了更多的接口，而这些接口正是Automation（自动化）服务器或ActiveX控件所需要的。

正如在第2章所介绍的那样，IUnknown接口提供了三个方法，即：_AddRef、_Release和QueryInterface方法。IUnknown接口的定义如下所示（以下代码是从系统单元中提取的）：

```
type
  IUnknown = interface
    ['{00000000-0000-0000-C000-000000000046}']
    function QueryInterface(const IID: TGUID;
      out Obj): Integer; stdcall;
    function _AddRef: Integer; stdcall;
    function _Release: Integer; stdcall;
  end;
```

其中_AddRef和_Release方法用于实现引用计数。而QueryInterface方法则用于处理信息类型和对象类型的兼容问题。

说明：实际上，前面的代码也介绍了out参数的使用方法，该参数是由方法回送给调用程序的。不过，当应用程序首先将该参数传递给方法时，该参数并没有被初始化成任何值。out参数已经加入到Delphi语言中，用以支持COM技术。但是它们也可以用于正常的应用程序中。而且在指定的环境中使用它可以提高参数的传递效率（尤其是对那些接口、字符串和动态数组的帮助更大）。此外，还应该明白的是，尽管Delphi语言在对接口类型进行定义时，尽量使其与COM技术相互兼容，但是实际上，Delphi接口在工作时并不需要使用COM技术。有关这方面的内容在第2章中也进行了介绍，而且还为读者编写了一个没有使用COM技术的基础接口示例。

通常情况下，开发人员无需自己实现这些方法，因为可以从Delphi提供的现有支持类中派生出这些方法。其中Delphi所提供的、最重要的一个类就是TComObject，该类在ComObj单元中定义。当创建一个COM服务器时，通常需要继承该类。

TComObject实现了IUnknown接口（它将其方法与ObjAddRef、ObjQuery和ObjRelease接口相映射）和ISupportErrorInfo接口（此时需要借助于InterfaceSupportsErrorInfo方法）。

注意, TComObject类的引用计数实现方法是具有线程安全性的。这主要是因为, 这里使用了InterlockedIncrement和InterlockedDecrement这两个API函数, 而不只是使用普通的Inc和Dec进程。

如果还记得在第2章中所介绍的那些有关引用计数的内容, 就应该明白TInterfacedObject所提供的_Release方法可以在没有任何引用的情况下销毁一个对象。TComObject类的作用与此相同。此外, 还应该牢记, 当使用基于变量的接口(包括COM变量)时, Delphi会自动地向编译代码中添加引用计数调用。

最后, 还应注意与QueryInterface方法相关的以下两方面内容:

- QueryInterface方法用于类型检查操作。应用程序可以向某个对象询问以下一些问题: 是我所感兴趣的类型吗? 是否实现了相应的接口和我想调用的指定方法? 如果回答是否定的, 该应用程序就将寻找其他对象, 甚至有可能会询问其他服务器。
- 如果回答是肯定的, QueryInterface通常就会借助其引用输出参数(Obj)来会返回一个指向该对象的指针。

要想理解QueryInterface方法的作用, 就需要切记一个COM对象可以实现多个接口这一特性, 正如TComObject类那样。当调用QueryInterface时, 可以首先使用TGUID参数来寻找一个可供使用的对象接口。

除了TComObject类外, Delphi还预先提供了其他一些COM类。下面列出Delphi VCL所提供的其他一些重要的COM类, 以及它们所适用的场合:

- TTypedComObject, 该类在ComObj单元中定义, 它是从TComObject那里继承而来的, 而且实现了IProvideClassInfo接口(而IUnknown和ISupportErrorInfo接口已经由基类TComObject实现了)。
- TAutoObject, 该类在ComObj单元中定义, 从TTypedComObject那里继承而来, 并且实现了IDispatch接口。
- TActiveXControl, 该类在AxCtrls单元中定义, 继承于TAutoObject, 并且实现了一些接口(IPersistStreamInit、IPersistStorage、IOleObject和IOleControl等)。

全局惟一标识符

QueryInterface方法拥有一个TGUID类型的方法。该类型代表着一个用于标识COM对象类(在这种情况下, GUID通常被称为CLSID)、接口(此时通常被称为项目ID, IID)以及其他COM和系统实体的惟一ID。当想了解一个对象是否支持指定的接口时, 一般可以询问对象是否实现了具有指定IID的接口(即由Microsoft所定义的默认COM接口)。另外一种有关的ID则用于指明具体的类或CLSID。Windows注册表可以通过指明与之相关的DLL或可执行文件来保存该CLSID。COM服务器的开发者定义了这种类标识符。

这两种ID都被称为GUID, 或全局惟一标识符。如果每位开发者都使用一个数字来标识一个COM服务器, 那么如何才能确保这个数字不会发生重复使用的情况呢? 回答非常简单, 无法确保这种情况是否会发生。真实的答案是, GUID是一个长整数(16比特, 或128比特即一个长达38位的数!)。在这种情况下两个随机生成的数字相等的机率是非常小的。更重要的是, 编程人员还会使用名为CoCreateGuid的指定API函数(直接或借助它们的开发环境), 以便最终可以形成一个合法的、同时可以反映出一些系统信息的GUID。

借助网卡在计算机上创建的GUID可以被确保是惟一的。因为，网卡中含有一个惟一的序列号，该序列号可以用于生成GUID。此外，借助CPU ID（例如Pentium III）创建出的GUID甚至在不使用网卡的情况下也肯定是惟一的。如果没有任何惟一的硬件标识符，也不太可能出现GUID重复的情况。

警告：注意如果想从其他应用程序那里复制GUID，那么一定要小心（因为此时可能会导致不同的两个COM对象使用同一GUID的情况）。此外还应该尽量避免在生成自己的ID时随意输入任意顺序的数字这一做法。如果想尽可能地避免任何故障的发生，也可在Delphi编辑器中按下Ctrl+Shift+G，以获得一个新的、正确定义的、真正惟一的GUID。

在Delphi中，TGUID类型（定义于系统单元中）是一个记录结构。该结构有些古怪，但它是Windows操作系统所必需的。值得庆幸的是，Delphi编译器提供了与之相关的功能，从而大大简化了一些本来比较乏味或耗时的工作。例如，可以使用保存于字符串中的标准十六进制符号将值赋予一个GUID。如下面这段示例代码段所示：

```
const
```

```
Class_ActiveForm1: TGUID = {1AFA6D61-7B89-11D0-98D0-444553540000}';
```

此外，程序员也可以在需要使用GUID的时候，传递一个由IID标识的接口。此时，Delphi将会自动地从中提取出需要引用的IID。如果需要手动地创建一个GUID，而且是需要Delphi环境之外来完成这一工作，也可以调用CoCreateGuid这个Windows API函数，如同NewGuid示例显示的那样（如图12.1所示）。由于本示例非常简单，所以在此没有列出其代码。



图12.1 由NewGuid示例所生成的GUID。此时的GUID值取决于正在使用的计算机和运行应用程序的时间

为了处理GUID，Delphi提供了GUIDToString函数和功能与之相反的StringToGUID函数。此外，程序员也可以使用相应的Windows API函数，例如StringFromGuid2，但是在这种情况下，必须使用WideString类型而不是普通的字符串类型。无论何时，一旦涉及到COM，就不得不使用WideString类型，除非使用了可以自动转换类型的Delphi函数。如果不希望使用那些直接调用COM API函数的Delphi函数，也可以使用PWideChar类型（该类型的变量将指向一个没有终止的宽位字符），或者将一个WideString转换为PWideChar（这就像在调用低级别的Windows API时需要将字符串转换为PChar类型一样）。

类厂的作用

当将一个COM对象的GUID注册到注册表中时，可以使用一个具体的API函数来创建该对象，例如CreateComObject API:

```
function CreateComObject (const ClassID: TGUID): IUnknown;
```

该API函数将查看注册表，同时使用指定的GUID来查找注册了该对象的服务器，并装载它。如果该服务器是一个DLL，也可调用该DLL的DLLGetClassObject方法。该函数是每一个进程内服务器所必须提供和导出的方法之一：

```
function DllGetClassObject (const CLSID, IID: TGUID;  
    var Obj): HRESULT; stdcall;
```

该API函数将接收所需的类和接口，并将它们作为参数，同时在其引用参数中返回一个对象。注意，该函数所返回的对象是一个类厂（class factory）。

正如其名称所表达的含义那样，一个类厂是一个可以创建其他对象的对象。每个服务器都可以拥有多个对象。Delphi为实现COM开发所提供的众多优点之一就是，系统可以为开发人员提供一个类厂。所以，在此也就无需再为本章的示例添加额外的自定义类厂了。

调用CreateComObject API函数的工作不会随着类厂的创建而停止。在获取类厂后，CreateComObject将调用IClassFactory接口的CreateInstance方法。该方法将创建所需的对象，并返回该对象。如果没有出现任何错误，该对象将被作为CreateComObject API函数所返回的值。

通过上面这一机制（包括类厂和DLLGetClassObject调用）的设置，Delphi使得编程人员可以创建COM对象。同时，Windows也为用户提供了一个简单的CreageComObect函数，不过实际上，该函数需要在后台完成大量复杂的操作。而Delphi的出色之处则在于，大多数复杂的COM机制都将交由RTL来完成。下面将详细地介绍一些Delphi所具有的、用于灵活控制COM的方法。

对于每个核心的VCL COM类，Delphi也为它们定义了一个类厂。这些类厂类形成了一个层级，其中包括TComObjectFactory、TTypedComObjectFactory、TAutoObjectFactory和TActiveXControlFactory。类厂非常重要，而且每一COM服务器都需要使用到它们。通常，在创建对象时，Delphi应用程序都需要在其单元初始化部分中使用类厂来定义相应的服务器对象类。

第一个COM服务器

实际上，如果想要了解COM，那么除了通过COM服务器DLL来创建一个简单的COM服务器外，再没有其他更好的方法了。在Delphi中，一个包含着COM对象的库通常被标识为一个ActiveX库。所以此时，读者可以通过选择File►New►Other来打开ActiveX Library页，并从中选择相应的ActiveX Library选项。通过上述步骤就可以生成那个前面已经保存于FirstCom示例中的项目文件了。该项目文件的完整源代码如下：

```
library FirstCom;

uses
  ComServ;

exports
  DllGetClassObject,
  DllCanUnloadNow,
  DllRegisterServer,
  DllUnregisterServer;

{$R *.RES}

begin
end.
```

由DLL所导出的四个函数可以实现与COM技术之间兼容的工作，这些函数由下面的系统组件所使用：

- 用于访问类厂（DllGetClassObject）
- 用于检查服务器是否已经销毁了其所有对象，以及是否可将自己从内存中卸载（DllCanUnloadNow）
- 用于在Windows注册表中添加或删除与服务器相关的信息（DllRegisterServer和DllUnregisterServer）

通常情况下，开发人员无需自己实现这些函数，因为Delphi在ComServ单元中提供了默认为实现代码。因此，在服务器代码中，只需根据需要导入相应的代码就可以了。

COM接口和对象

在定义好COM服务器的结构后，就可以开始部署它了。首先需要编写服务器中的接口代码。下面介绍一个简单接口的代码，读者需要将它添加到一个独立的单元中（在本例中为NumInft）：

```
type
  INumber = interface
    ['{B4131140-7C2F-11D0-98D0-444553540000}']
    function GetValue: Integer; stdcall;
    procedure SetValue (New: Integer); stdcall;
    procedure Increase; stdcall;
  end;
```

在声明了自定义的接口后，可以将该对象添加到服务器中。要完成这一工作，可以使用COM对象向导（选择File►New►Other，并打开ActiveX页面就可以找到该向导）。该向导的对话框如图12.2所示。根据需要提供服务器类的名称和相关的描述信息。本示例在前面的设置中禁用了类型库生成功能（所以在Delphi 7中，向导也将禁用相应的接口区域，这一点与Delphi 6有所不同），以避免一次引入过多的项目。此外，读者也可以根据自己的需要，按照相关附加说明中介绍的内容来选择一个实例和线程模型。

由COM对象向导所生成的代码非常简单。接口中包含着该类的定义，其中包括相应的方法和数据：

```
type
  TNumber = class(TComObject, INumber)
  protected
    {Declare INumber methods here}
  end;
```

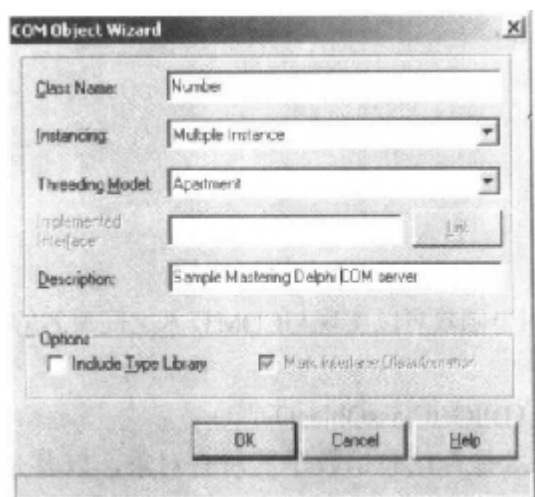


图12.2 COM对象向导

除了用服务器的GUID（保存于Class_Number常量中）外，代码单元中的initialization部分还拥有着与其自身相关的代码，该代码与前面大家在向导对话框中所做的选择密切相关：

initialization

```
TComObjectFactory.Create(ComServer, TNumber, Class_Number, 'Number',  
  'Number Server', ciMultiInstance, tmApartment);
```

该代码可创建一个TComObjectFactory类的对象，并且将以参数的方式来传递全局ComServer对象、一个指向前面所定义类的类以及该类的GUID、服务器名称、服务器描述和想要使用的实例和线程模型。

在ComServ单元中所定义的全局ComServer对象是服务器库中相应类厂的管理者。它将使用它自己的ForEachFactory方法来跟踪被分配对象的数目。正如前面所介绍的那样，ComServ单元将实现使用DLL构造COM库时所需的那些函数。

在对向导所生成的源代码进行检查后，可以继续向TNumber类添加需要使用的方法，以便最终实现INumber接口，并编写它们的代码，之后，就可以在自己的服务器中使用COM对象了。

COM实例和线程模型

创建一个COM服务器时，应该选择恰当的实例和线程模型，这一工作会大大地影响COM服务器的动作。

实例的选择将直接影响进程外服务器（任何一个位于单独的可执行文件中而不是DLL中的COM服务器都属于进程外服务器）。此时，可以选择的实例类型值有以下三种：

多重（Multiple） 选择该值意味着当多个客户应用程序需要请求该COM对象时，系统将启动该服务器的多个实例。

单一 (Single) 选择该值意味着即使有多个客户应用程序请求该COM对象, 系统也只会提供该服务器应用程序的一个实例, 此时可以为这些请求服务创建多个内部对象。

内部 (Internal) 选择该值意味着该对象仅能够被创建于服务器内部, 客户应用程序无法请求任何一个实例 (这种特殊设置只应用于过程内服务器中)。

第二个需要讨论的问题是COM对象的线程支持功能, 该功能只对那些进程内服务器 (通过DLL实现的服务器) 有效。线程模型是连接客户和服务器应用程序的关键点: 如果两端都同意使用相同的模型, 那么该模型就将用于连接操作中。如果不同意, COM将使用配置编组机制来建立连接, 采用这种方式来建立连接将大大地降低操作的速度。此外, 还需要注意的是, 一个服务器无需将其自己的线程模型发布到注册表中 (这取决于在向导中所做的设置); 此外在代码中还必须遵循与线程模型相关的规则。下面重点介绍一些线程模型:

单一模型 (Single) 实际上, 目前线程并不支持这种模型。到达COM服务器的请求已被串行化, 所以客户端每次只能执行一种操作。

单元模型或“单线程单元” (Apartment Model或“Single-Threaded Apartment”) 在这种模型中, 只有那些创建对象的线程可以调用自己的方法。这意味着发往每一服务器对象的请求都将被串行化, 但是同一服务器的其他对象也可同时接收这些请求。因此, 服务器对象只能特别小心地访问服务器的那些全局数据 (例如使用严格的、互斥的或除同步方法以外的其他一些技术)。这就是常用于Internet Explorer内ActiveX控件中的线程模型。

自由模型或“多线程单元” (Free Model或“Multithreaded Apartment”) 这种模型对客户没有特殊的限制。这意味着多个线程可以同时应用于相同的对象中。因此, 每一对象中的每一方法都必须对其自己及其使用的局部数据进行保护, 以避免它们被同时地多重调用。与Single (单一) 和单元 (Apartment) 模型相比, 对于服务器一端来说, 这种线程模型更加复杂。因为此时即使是访问对象自己的实例数据, 也必须注意线程安全问题。

两者 (Both) 在这种模型下, 服务器对象既可以提供单元 (Apartment) 模型也可以提供自由模型 (Free模型)。

中立模型 (Neutral) 该模型是在Windows 2000中引进的, 且仅出现于COM+中。该模型表明多个客户可以同时调用不同线程中的对象, 但是COM技术可以确保同一方法不会被同时调用两次。这种模型是并发访问对象数据所必需的模型。在COM中, 该模型将被直接映射转换为单元 (Apartment) 模型。

初始化COM对象

如果查看TComObject类的定义, 就会发现它具有实构造函数。实际上, 它拥有多个构造函数, 而每一构造函数都将调用一个虚Initialize方法。因此, 如果想要正确地设置COM对象, 就无需再去自定义新构造函数 (该构造函数永远都不会被调用), 而只需要像前面在TNumber中那样重载其Initialize方法就可以了。该类的最终代码如下所示:

```
type
    TNumber = class(TComObject, INumber)
    private
        fValue: Integer;
    public
```

```

function GetValue: Integer; virtual; stdcall;
procedure SetValue (New: Integer); virtual; stdcall;
procedure Increase; virtual; stdcall;
procedure Initialize; override;
destructor Destroy; override;
end;

```

正如读者所见的那样，上述代码还重载了该类的构造函数，因为该示例还将对Delphi提供的COM对象自动销毁功能进行测试。

测试COM服务器

到此为止已经完成了COM服务器对象的代码编写工作。接下来需要首先注册它，然后才可以使用该服务器。编译其代码，然后使用Delphi提供的Run►Register ActiveX Server菜单命令。通过上述步骤就可以将该服务器注册到自己的计算机中，并更新局部注册表。

分发该服务器时，应该首先将它安装到客户端所在的计算机上。要完成这一工作，可以编写一个REG文件来将该服务器安装到注册表中。但是，这并不是最好的方法，因为服务器中已经提供了可以激活注册服务器操作的函数。该函数可以被Delphi环境所激活，正如读者以前所见过的那样，此外也可以通过以下几种方法激活服务器注册操作：

- 将COM服务器DLL作为Microsoft命令行命令RegSvr32.exe的参数，该命令可以在\Windows\System目录下找到。
- 使用Delphi自带的TregSvr.exe示例应用程序来完成这一工作（在\Bin目录下可以找到已编译好的应用程序，其源代码则位于\Demos\ActiveX目录下）。
- 让安装编译器应用程序调用服务器的注册函数。

在注册完服务器后，就可以切换到该示例的客户端了。此时，该示例的客户端名为TestCom，该程序保存于单独的目录下。该应用程序可以通过COM机制装载服务器DLL，由于服务器已经在注册表中提供了足够多的信息，所以客户应用程序无需了解服务器所处的目录。

该应用程序的窗体与第10章“库和组件包”中用于测试DLL的示例程序的窗体非常相似。在客户应用程序中，还必须在源代码中加入相应的接口代码，同时重新声明COM服务器的GUID。在应用程序的初始状态下，所有的按钮都处于禁用（即设计时的状态）状态，只有在创建一个对象后，这些按钮才会有效。采用这种方式后，如果在创建一个对象时出现了异常，那么与该对象相关的所有按钮都将被禁用：

```

procedure TForm1.FormCreate(Sender: TObject);
begin
    // create first object
    Num1 := CreateComObject (Class_Number) as INumber;
    Num1.SetValue (SpinEdit1.Value);
    Label1.Caption := 'Num1: ' + IntToStr (Num1.GetValue);
    Button1.Enabled := True;
    Button2.Enabled := True;

    // create second object

```

```

Num2 := CreateComObject (Class_Number) as INumber;
Label2.Caption := 'Num2: ' + IntToStr (Num2.GetValue);
Button3.Enabled := True;
Button4.Enabled := True;
end;

```

此外，尤其需要注意指向CreateComObject的调用，和其随后as关键字的作用。该API调用将启动COM对象的构造机制。前面已经对此进行了较为详细的介绍。这也被称为动态装载服务器DLL。该调用返回的值为IUnknown类型的对象。在该对象被赋予Num1和Num2字段之前，必须将其转换为正确的接口类型。在本例中，该接口的类型为INumber。

警告：指定接口的类型时，通常需要使用as关键字，接口将会根据它在后台执行相应的QueryInterface调用。此外，也可以直接执行QueryInterface或Supports调用。在使用接口这种情况下，as（或具体的函数调用）是从其他接口中提取接口的惟一方法。企图将一个接口指针的类型直接转换为另一接口指针类型的做法是错误的——永远不要那样做。

此外，该程序还提供了带有事件处理器的按钮（即窗体底部的按钮），该按钮可以创建一个新COM对象，用于获取100以后的数字。如果想弄清楚为什么需要在示例中添加这一功能，可以单击该按钮，相应的答案将显示在弹出的信息窗口中。此外，用户也可能会看到第二段信息，用于指明对象已经被销毁。通过它们就可以清楚地看到，当释放一个接口变量时，会调用对象的_Release方法，并同时减少对象的引用计数，而且当引用计数减少至0时，还将销毁该对象。

当终止应用程序时，其他两个对象也需要执行相同的操作。尽管该应用程序没有显式地执行这些操作，这两个对象也是需要被销毁的。此时，它们的Destroy析构函数所显示的信息将清楚地演示这一过程。出现上述这种情况的原因是由于这两个对象是以实例的类型被声明的，而Delphi会通过引用计数来计算它们。此外，当想通过一个接口来销毁指向COM对象的某个引用时，不能直接调用Free方法（因为接口没有提供Free调用），但是却可以将nil赋予接口变量。这一赋值操作同样可以达到删除引用的作用，甚至还可以销毁相应的对象。

使用接口属性

在接下来的步骤中，可以通过向INumber接口添加相应的属性，对该示例进行扩展。当向一个接口添加属性时，需要首先指明数据类型，然后指明是需要执行read还是write指令。此外，也可以指定只读或只写属性。但是read和write从句必须始终指向一个方法，因为接口除了会保存方法外，不会再保存其他任何内容了。

更新后的接口代码如下所示，该代码是PropCom示例的一部分：

```

type
  INumberProp = interface
    ['{B36C5800-8E59-11D0-98D0-444553540000}']
    function GetValue: Integer; stdcall;
    procedure SetValue (New: Integer); stdcall;
    property Value: Integer read GetValue write SetValue;
    procedure Increase; stdcall;
  end;

```


这里为新接口进行了重新命名,更重要的是为它提供了一个新的接口ID。此时,也可以通过继承以前的接口来生成这个新的接口类型,但是如果这样做,那么这个新接口也就没有任何优越性了。COM自身并不支持继承功能,而且从COM这一方面来看,所有的接口都是不同的,因为它们各自拥有不同的接口ID。无需多说,在Delphi中,编程人员可以使用继承功能来改进接口代码的结构和那些用来实现服务器对象的代码结构。

在PropCom示例中,更新了服务器类的声明部分,使其引用了一个新的接口,并为之提供了一个新的服务器对象ID。客户程序(即本例中的TestProp)现在就可以使用Value属性来替代原来的SetValue和GetValue方法了。下面这一小段代码摘录于FormCreate方法的代码:

```
Num1 := CreateComObject (Class_NumPropServer) as INumberProp;  
Num1.Value := SpinEdit1.Value;  
Label1.Caption := 'Num1: ' + IntToStr (Num1.Value);
```

实际上在处理接口时选择方法还是属性只是语法上的差别。因为接口属性是无法像Delphi类属性那样访问私有数据的。如果使用属性这种形式,那么代码可能会更加易读一些。

调用虚方法

前面已经建立了一组基于COM的示例。不过读者可能仍然会觉得调用在DLL中所创建的对象的方法不太安全。如果无法将这些方法导出DLL又该怎么办呢?COM服务器(DLL)会首先创建一个对象,然后再将其返回调用的应用程序中。通过这样的步骤,DLL就可以创建一个具有虚方法表(即virtual method table,简称VMT)的对象了。更准确地说,该对象为其类提供了一个VMT,同时也为其实现的每个接口提供了一个虚方法表。

此时,主程序可能会接收到一个带有请求该接口的虚方法表的接口变量。该VMT除了可以用来调用相应的方法外,还可以用来查询其他那些支持该COM对象的接口(这是由于QueryInterface方法也是IUnknown接口VMT的一部分)。

主应用程序根本无需知道这些方法的内存地址,因为这些对象已经知道了,这就像是它们正在执行多态调用一样。不过COM技术的强大功能还不只于此,编程人员甚至无需了解创建对象时所使用的编程语言是什么。使用COM技术并按照标准的指令就可以提供VMT。

提示: 使用与COM兼容的VMT会产生一种奇怪的效果。即,方法的名称并不重要,它们的地址将被置于VMT中的恰当位置上。这就是为什么可以将接口的方法与实现它的函数映射起来的原因。

总而言之,COM技术为对象提供了独立于语言的二进制标准。用于各个模块中的共享对象将被编译,它们的VMT拥有一个独特的结构,该结构是由COM所决定,而不是由原来曾经使用过的开发环境所决定的。

自动化

到目前为止,可以使用COM技术让一个可执行文件和库共享对象。大多数情况下,用户希望应用程序间能够相互通信。实现这一功能的其中一种方法就是使用Automation(以前被称为OLE自动化)功能。下面首先为向读者介绍一组示例,通过它们来介绍在类型库的基础上自定义接口的方法。接下来还将介绍Word和Excel自动化控制器的开发工作,向读者介绍如何将数据库中的信息传递给这些应用程序。

说明：当前的Microsoft文档使用了Automation（自动化）这一术语，而不再使用原有的OLE自动化术语了。此外，Microsoft还使用了活动文档和复合文档两种术语来替代原有的OLE文档这种叫法。本书趋向于使用新的术语，尽管老的“OLE”术语始终被认为更具有说明性。

在Windows操作系统中，应用程序无法一直存在于孤立的环境中，用户通常希望它们之间能够相互通信。剪贴板（Clipboard）和DDE功能为应用程序间数据交换的实现提供了一个简单的方法。用户可以借助它们在应用程序间执行复制和粘贴操作。但是，越来越多的应用程序提供了自动化（Automation）接口，通过该接口其他应用程序就可以驱动它。与手动操作相比，除了编程自动化本身所具有的优点外，这些接口还具有独立于语言的优点。所以，用户可以使用Delphi、C++、Visual Basic或宏语言来驱动一个自动化服务器，而无需了解该服务器是由什么语言编写的。Delphi提供了实现自动化功能的途径，其编译器为此提供了强大的功能，同时VCL也大大简化了开发者们所需要执行的操作。为了向自动化功能提供支持，Delphi提供了一个向导和一个功能强大的类型库编辑器，该编辑器可以支持双接口。当使用进程内DLL时，客户应用程序可以使用服务器，并直接调用其方法。因为，此时它们位于同一地址空间中。而当使用自动化功能时，所处的环境就相对复杂一些了。通常情况下，客户端（被称为控制器）和服务端是两个分别运行于不同地址空间中的独立应用程序。所以，系统必须使用复杂的参数传递机制（被称为配置编组机制，在此不做过多的介绍）才能完成调用相应方法的工作。

从技术上来讲，在COM中提供自动化功能意味着需要实现IDispatch接口，该接口是在System（系统）单元中声明的：

```
type
  IDispatch = interface(IUnknown)
    ['{00020400-0000-0000-C000-000000000046}']
    function GetTypeInfoCount(out Count: Integer): HRESULT; stdcall;
    function GetTypeInfo(Index, LocaleID: Integer;
      out TypeInfo): HRESULT; stdcall;
    function GetIDsOfNames(const IID: TGUID; Names: Pointer;
      NameCount, LocaleID: Integer; DispIDs: Pointer): HRESULT; stdcall;
    function Invoke(DispID: Integer; const IID: TGUID;
      LocaleID: Integer; Flags: Word; var Params;
      VarResult, ExcepInfo, ArgErr: Pointer): HRESULT; stdcall;
  end;
```

前两个方法用于返回相应的类型信息。而最后两个函数则用于激活自动化服务器上的实际的方法。实际上，激活操作是由最后一个方法Invoke完成的，而GetIDsOfNames可以根据方法的名称来决定分派ID。在Delphi中创建一个自动化服务器时，需要编程人员完成的惟一工作就是定义类型库，并实现其接口。Delphi还通过其编译器和VCL代码为用户提供了其他一些功能（实际上，VCL的一部分原来被称做DAX框架）。

如果考虑到一个控制器可以通过三种途径来调用自动化服务器上的方法，那么就会发现IDispatch方法的作用更加明显了：

- 首先，它可以请求执行一个方法，以字符串来传递其名称，这与调用DLL的动态调用有些相似。当使用变体类型的变量（参见随后的说明）来调用自动化服务器时，

Delphi就将采取上面这些操作。这一技术在使用上非常简单，不过相对而言速度较慢，而且它所能提供的编译类型检查也比较有限。在调用了Invoke之后该方法还将隐式地调用GetIDsOfNames。

- 其次，它还可以为服务器上的对象导入一个Delphi分派接口（dispinterface）的定义，而且以更加直接的方式来调用其自己的方法（分派一个数字，它可直接调用‘Invoke’，就好像编译时已经知道每一方法的DispId一样）。这一技术是建立在接口基础上的，而且允许编译器检查参数的类型，并快速地生成相应的代码。不过，它需要编程人员的更多参与（也就是需要使用类型库）。此外，编程人员还需要终止控制器应用程序与指定服务器版本之间的绑定状态。
- 此外，该方法还可以通过接口变量等来直接地调用该接口，并将它作为常规的COM对象。这种途径适用于大多数情况下，主要是因为大部分自动化服务器接口提供了双接口（这种情况可以支持IDispatch和普通的COM接口）。

下面的示例将要使用到这些技术，同时还将进一步对这些技术进行比较。

说明：可以使用变体变量来保存指向自动化对象的引用。在Delphi语言中，变体是一种可变化的数据类型，这种类型的变量可以用于保存各种类型的值。变体数据类型包括基础类型（例如整数、字符串、字符和布尔值），此外，也可以是IDispatch接口类型。变体类型可在运行时完成类型检查工作，这也是为什么编译器甚至能在不知道自动化服务器提供了哪些方法的情况下编译代码的原因，稍后将就相关内容进行具体的介绍。

分派自动化调用

这两种方法之间的重要不同之处在于第二种方法通常需要使用类型库，它是COM组件的基础之一。一个类型库实际上是一个类型信息的集合，通常也可以在COM对象（而无需支持分派功能）中找到它们。该集合通常可以用来描述构成通用COM对象或自动化服务器的所有元素（对象、接口和其他类型信息）。类型库和这些元素的其他描述信息（例如C或Pascal代码）间的主要区别在于，类型库是独立于语言的。而类型元素则是由COM所定义的，它们是编程语言中标准元素的一部分子集，任何开发工具都可以使用它们。为什么编程人员需要使用这些信息呢？

正如笔者在前面曾经介绍过的那样，如果使用变体变量调用一个自动化对象的方法，那么在编译时，Delphi编译器就根本无需了解该与该方法相关的任何信息。下面这小段代码使用了Word应用程序所提供的老板自动化接口，并将其注册为Word.Basic，通过该示例就可以看出对于开发人员来说完成这一工作是非常简单的：

```
var
    VarW: Variant;
begin
    VarW := CreateOleObject ('Word.Basic');
    VarW.FileNew;
    VarW.Insert ('Mastering Delphi by Marco Cant ');
```

说明：正如读者将在后面看到的那样，Word近期版本所注册的内容仍然是Word.Basic接口。它与其内部的WordBasic宏语言相对应，但是它们还将注册新的Word.application接口，该接口与VBA宏语言相对应。Delphi还提供了相应的组件，该组件有一个简单的链接，用于连接Microsoft Office应用程序，相关内容将在本章稍后进行介绍。

上述三行代码将启动Word（除非它已经处于运行状态），然后创建一个新文档，并在该新文档中添加一条句子。该应用程序的运行效果如图12.3所示。

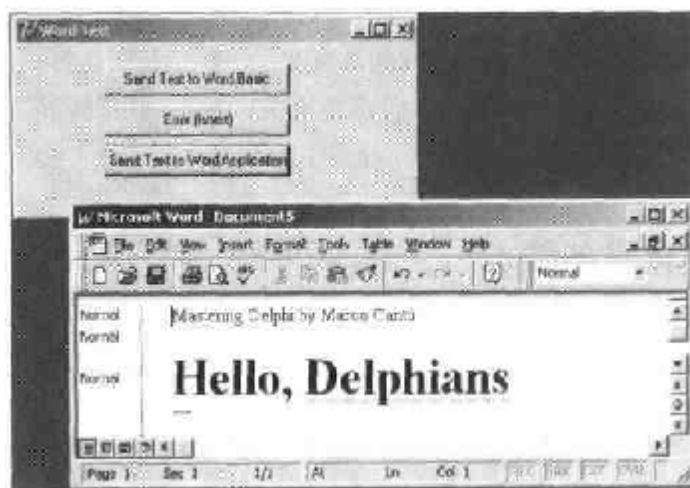


图12.3 WordTest Delphi应用程序将创建并组合Word文档

遗憾的是，Delphi编译器没有提供用来检查该方法是否存在的方法。此外，在运行时执行所有的输入检查操作风险是比较大的。因为，即使出现的是比较轻微的函数名称拼写错误，也只有在运行该应用程序时，用户才能得到与该错误相关的警告信息，并被定位到相应的代码行上。例如，即使键入了`VarW.Isnert`，那么编译器也不会报告与该拼写相关的错误信息，但是如果到了运行时刻，用户将会得到相应的错误信息。因为编译器无法识别该名称，此时，Word将假设该方法并不存在。

尽管IDispatch接口提供了上面所介绍的方法，但是服务器仍然能够并且安全地导出其接口的描述信息，以及那些使用了该类型库的对象。然后再通过特殊的工具（例如Delphi）将这种类型库转换为相应的定义形式。定义这些类型库时，可以使用与编写客户软件或控制器程序（例如Delphi语言）相同的编程语言。如果采用了相同的语言，那么编译器就能检查出代码是否正确，并且在Delphi编辑器中为用户提供代码自动完成（Code Completion）功能和代码参数自动填写（Code Parameter）功能。

一旦编译器完成了拼写检查，就可以使用两种技术中的任何一种向服务器发送请求信息了。此时可以使用Vtable（即接口类型声明中的一个入口），也可以使用dispinterface（即分派接口）。本章前面也曾经使用过接口类型声明，所以读者应该对此有所了解。dispinterface是用来在接口内每一入口和一个数字间建立映像的基本方法。那些发往服务器的调用只有借助调用IDispatch.Invoke方法才能通过数字被分派，并且无需再额外地调用IDispatch.GetIDsOfNames。读者也可以将它看做为一种中间技术，它介于以函数名称进行分派的操作和VTable中的直接调用操作之间。

说明：术语dispinterface是一个关键字，是类型库编辑器为每一接口自动生成的关键字。与dispinterface一起，Delphi还使用了其他一些关键字，其中dispid指明了与每一元素相关的数字，而readonly和writeonly则是用于描述属性的可选说明符。

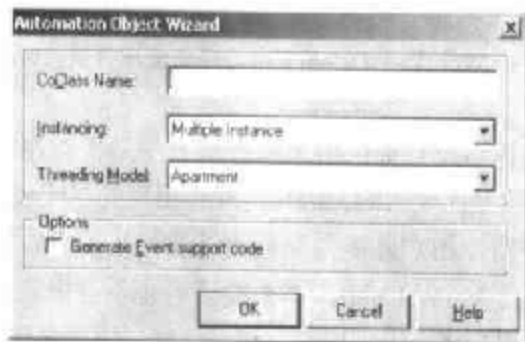
用于描述与服务器相连能力的方法有两种，即尽量多地使用动态或静态方法，该方法也被称为双接口。在编写一个COM控制器时，可以选择两种方式来访问服务器上的方法：

使用后期绑定或dispinterface所提供的机制，或者使用前期绑定和VTable所提供的机制，即接口类型。

另一个需要读者牢记的重要内容是（同时需要考虑其他一些因素），采用不同技术时的运行速度有快有慢。通过名称来查找函数（然后再在运行时执行拼写检查）是最慢的一种方法，使用dispinterface比较快一些，而直接使用VTable调用则是最快的方法。本章稍后将要介绍的TlibCli示例将会测试上述这三种方法的性能。

编写一个自动化服务器

下面我们将编写一个自动化服务器。要创建Automation对象，可以使用Delphi的Automation Object Wizard。从一个新的应用程序开始，选择File►New►Other，打开对象存储库，移动到ActiveX页面，选择Automation Object。可以看见Automation Object Wizard，如下所示。



在该向导中，输入类的名称（起始字母不要使用T，因为它将被自动添加到Delphi实现类），并单击OK。Delphi将打开类型库编辑器。

提示：Delphi可以产生导出事件的Automation服务器。选取该向导中相应的复选框，Delphi将在类型库中和其所生产的源代码中添加合适的项目。

类型库编辑器

可以使用类型库编辑器在Delphi中定义一个类型库。图12.4显示了添加一些元素之后的窗口。类型库编辑器允许用户向已创建的自动化服务器对象添加方法和属性，或者向使用COM Object向导创建的COM对象添加方法和属性。如果这样做，则会生成类型库（TLB）文件和相应的Delphi语言资源代码，其保存在称为类型库输入单元的单元中。

为了更好地使用Delphi的类型库编辑器，有两个建议。第一个也是最简单的是：如果在工具栏上右键单击，则会看到Text Labels选项，每个工具栏都有一个标题，这使得编辑器更容易使用。第二个建议是进入到Delphi的Environment Options对话框的Type Library页面，选择Pascal语言单选按钮，而不选择IDL语言单选按钮。这个设置确定了类型库编辑器所使用的符号，用以显示方法和参数，甚至用来编辑方法参数类型或属性类型。除非读者过去用C或C++编写过COM代码，否则可能会更喜欢用Delphi的术语来思考，而不是使用IDL的术语来思考。

警告：在本书这一部分，我们将讨论在这个设置下如何使用类型库编辑器，因为另外提供的IDL术语描述会导致不必要的混淆和复杂性。

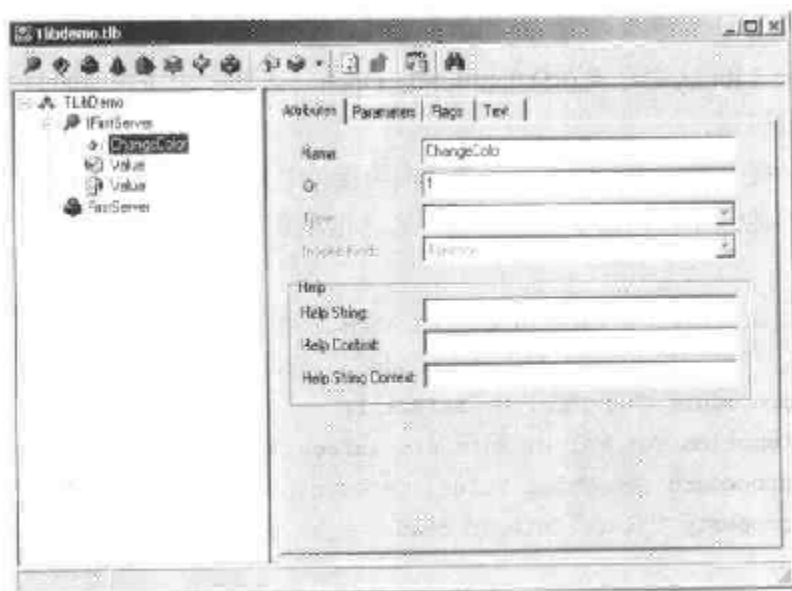


图12.4 类型库编辑器，显示了接口的细节

要建立第一个例子，可以使用该编辑器的相应工具栏按钮，并在窗口左边的Tree View控件中或右边的Name编辑框中输入名称，用于向服务器添加一个属性和方法。向接口添加这两个元素，并命名该接口为IFirstServer。

在Parameters页面可以定义一些参数并设置返回值。这里有一个特例，ChangeColor方法没有参数，其Delphi定义如下：

```
procedure ChangeColor; safecall;
```

说明：在Delphi中，Automation接口中的方法使用safecall调用约定。它在每个方法上封装了一个try/except块，并提供默认的返回值来表示错误或成功。此外，它还建立了COM丰富的错误对象，其中包含一些异常消息，这样，感兴趣的客户（例如Delphi客户）就可以在客户端重新创建服务器异常。

现在，通过单击类型库编辑器工具栏上的Property按钮，可以向接口添加一个属性。此外，可以输入一个名称，保存为Value，并且为其在Type组合框中选择一个数据类型。除了可以选择已经列出的类型之外，还可以直接输入其他类型，尤其是其他对象的接口。

示例中Value属性的定义对应于Delphi接口的下列元素：

```
function Get_Value: Integer; safecall;
procedure Set_Value(Value: Integer); safecall;
property Value: Integer read Get_Value write Set_Value;
```

单击类型库编辑器工具栏上的Refresh按钮，可以产生（或更新）具有该接口的Delphi单元。

服务器代码

现在，可以关闭类型库编辑器，并保存更改。这个操作将三个条目添加到项目中：类型库文件、相应的Delphi定义以及服务器对象的声明。类型库使用资源包含语句连接到项目，并将其添加到项目文件的源代码中：

```
{SR *.TLB}
```

使用View►Type Library命令或在Delphi的File Open对话框中选择相应的TLB文件，可以重新打开类型库编辑器。

正如前面所提及的，类型库还可以被转换为接口定义，并添加到新的Delphi单元中。该单元非常长，因此这里只列出了一些关键元素。很多重要的部分是新接口声明：

```
type
  IFirstServer = interface(IDispatch)
    ['{89855B42-8EFE-11D0-98D0-444553540000}']
    procedure ChangeColor; safecall;
    function Get_Value: Integer; safecall;
    procedure Set_Value(Value: Integer); safecall;
    property Value: Integer read Get_Value write Set_Value;
  end;
```

接着是dispinterface，这使得IFirstServer接口中的每个元素都关联到一个数字：

```
type
  IFirstServerDisp = dispinterface
    ['{89855B42-8EFE-11D0-98D0-444553540000}']
    procedure ChangeColor; dispid 1;
    property Value: Integer dispid 2;
  end;
```

该文件的最后一部分包含构造类，它用来在服务器上创建一个对象（这个对象用在应用程序的客户端，而不是服务器端）：

```
type
  CoFirstServer = class
    class function Create: IFirstServer;
    class function CreateRemote(const MachineName: string): IFirstServer;
  end;
```

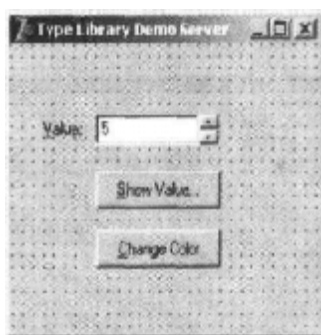
该文件中的所有声明（这里省略了其他一些声明）都被认为是内部的，支持隐式实现。对于多数Automation应用程序来说，没有必要全部了解它们。

最后，Delphi产生了一个包含Automation对象实现的文件。这个单元被添加到应用程序，可以使用它来完成程序。该单元声明了服务器对象的类，它实现了前面所定义的接口：

```
type
  TFirstServer = class(TAutoObject, IFirstServer)
  protected
    function Get_Value: Integer; safecall;
    procedure ChangeColor; safecall;
    procedure Set_Value(Value: Integer); safecall;
  end;
```

Delphi已经提供了方法的框架代码，因此我们只需要完善其中的一些行。在这个例子中，三个方法指向已经添加到窗体中的一个属性和两个方法。通常，不应该在服务器对象类中直

接添加与用户接口相关的代码。然而在这里这样做，是因为我想更改Value属性，但是会出现一个很显然的弊端（在编辑框中显示值）。在设计时，可以看到这个窗体。



注册自动化服务器

包含服务器对象的单元具有更多语句，这是由Delphi添加到initialization部分的：

```
initialization  
  TAutoObjectFactory.Create(ComServer, TFirstServer, Class_FirstServer,  
    ciMultiInstance);  
end.
```

说明：这里选择了多个实例。对于COM中不同的实例样式，可以参见本章前面的附加说明的“COM实例和线程模型”。

这与本章开始所介绍的类厂的创建不同。ComServer单元要利用InitPoc系统函数，将所有COM对象注册为COM服务器应用程序启动的一部分。调用Application.Initialize即可执行这个代码，Delphi默认时将Application.Initialize添加到任何程序的项目源代码中。

通过在目的机器（就是安装Automation服务器的计算机）上运行应用程序，或者在命令行上用参数/regserver来运行应用程序，就可以将该服务器信息添加到Windows注册表中。要这样做，可以选择Start>Run，或者在Explorer中创建一个快捷方法，或者输入一个命令行参数后在Delphi中运行该程序（使用Run>Parameters命令）。另一个命令行参数/unregserver可用来删除注册表的该服务器。

编写服务器的客户端

我们已经建立了一个服务器，现在可以准备一个客户端程序来测试它。这个客户通过使用变量或新的类型库来连接到服务器。第二种方法可以手动实现，或者使用Delphi技术来封装Automation服务器的组件。不妨尝试所有这些方法。

创建一个新应用程序——称之为TLibCli——并使用Delphi IDE的Project>Import类型库菜单命令来导入服务器的类型库。这个命令将打开Import Type Library对话框，如图12.5所示。这个对话框在上半部分列出了已注册的具有一个类型的COM服务器。按Add按钮并找到合适的文件模块，可以将其他项目添加到该列表中。Import Type Library对话框的下半部分显示了所选库的某些细节（例如，服务器对象的列表）。当按下Create Unit按钮（或Install按钮）时，该对话框会产生一个类型库导入单元，有关该单元的细节也包含在该Import Type Library对话框的下面。



图12.5 Delphi的类型库导入对话框

警告：不要向客户端应用程序添加类型库，因为现在正在编写Automation控制器，而不是服务器。控制器的Delphi项目不应该包含其所连接的服务器上的类型库。

类型库导入单元由Delphi所命名，并以_TLB作为结尾。这里，单元名是TlibdemoLib_TLB。前面曾经说过，creation类也是由类型库编辑器所产生的单元中的元素之一。我们已经给出了这个类的接口，这里是两个函数中第一个函数的实现：

```
class function CoFirstServer.Create: IFirstServer;
begin
    Result := CreateComObject(Class_FirstServer) as IFirstServer;
end;
```

可以使用它在相同计算机上创建一个服务器对象（并可启动服务器应用程序）。从上述代码中可以看出，该函数是CreateComObject调用的一个简化操作，它允许创建一个COM对象的实例（如果知道GUID）。当然，也可以使用CreateOleObject函数，它需要一个参数ProgID，这是服务器的注册名。这两个创建函数之间的另一个差别是：CreateComObject返回IUnknown类型的对象，而CreateOleObject返回IDispatch类型的对象。

在这个例子中，可使用CoFirstServer.Crcate简写形式。当创建服务器对象时，得到一个返回值IFirstServer接口。可以直接使用它，或者保存在派生变量中。下面是第一种方法的示例：

```
var
    MyServer: Variant;
begin
    MyServer := CoFirstServer.Create;
    MyServer.ChangeColor;
```

这个代码（基于派生）完全不同于本章前面所建立的第一个控制器代码（用于Microsoft Word）。下面的代码具有相同的效果：

```
var
  IMyServer: IFirstServer;
begin
  IMyServer := CoFirstServer.Create;
  IMyServer.ChangeColor;
```

大家已经看到了如何使用接口和变体。那Dispatch接口又如何呢？可以声明Dispatch接口类型的变量如下：

```
var
  DMyServer: IFirstServerDisp;
```

通过释放构造类返回的对象给接口分配一个对象之后，就可以像往常一样使用它来调用方法：

```
DMyServer := CoFirstServer.Create as IFirstServerDisp;
```

接口、变体和Dispatch接口：测试速度差别

在讨论类型库的一节中我们提到过，这些方法的差别之一就是速度。要知道这些技术的确切性能是非常复杂的，因为包含很多因素。在本章的TLibCli示例中添加了一个简单的测试，以给读者一个思路。这个测试代码是一个循环，它访问服务器的Value 100次。这个程序的输出显示了时间，它通过在执行该循环之前和之后调用GetTickCount API函数来确定。（有两种替换的方法，一是使用Delphi的自有时间函数，但不是很精确；或者使用非常精确的多媒体支持单元的计时函数，MMSystem。）

使用这个程序可以粗略地比较输出。要得到这些输出，可以基于接口调用方法，基于变体调用相应的版本和基于dispatch接口调用第三个版本。通过观察这个例子的计时，应该可以看到接口要快些，变体要慢些，而dispatch接口介于两者之间，并接近于接口的速度。

自动化对象的范围

另一个要记住的重要元素是Automation对象的范围（scope）。接口对象使用引用计数技术，因此如果关联到接口对象的变量在方法中被本地声明，则该对象在方法的末尾会被销毁，并且服务器也会终止（如果服务器所创建的所有对象都已经被销毁）。例如，用下列代码编写一个方法会产生很少的影响：

```
procedure TClientForm.ChangeColor;
var
  IMyServer: IFirstServer;
begin
  IMyServer := CoFirstServer.Create;
  IMyServer.ChangeColor;
end;
```

除非服务器已经被激活，否则将会创建程序的副本，并更改其颜色，但是只要接口类型的对象超出范围，服务器将立即关闭。在TLibCli例子中所使用的另一种方法是将对象声

明为窗体的一个字段，并在开始时创建COM对象，过程如下：

```
procedure TClientForm.FormCreate(Sender: TObject);
begin
    IMyServer := CoFirstServer.Create;
end;
```

使用这个代码，当客户端启动时，服务器程序将会立即被激活。当程序终止时，该窗体字段也会被销毁并且服务器关闭。另外一种方法是在窗体中声明对象，但是只有在使用的时候才创建，代码片段如下：

```
// MyServerBis: Variant;
if varType (MyServerBis) = varEmpty then
    MyServerBis := CoFirstServer.Create;
MyServerBis.ChangeColor;

// IMyServerBis: IFirstServer;
if not Assigned (IMyServerBis) then
    IMyServerBis := CoFirstServer.Create;
IMyServerBis.ChangeColor;
```

说明：在创建时，变体被初始化为varEmpty类型。如果给变体赋值null，则该类型变为varNull。varEmpty和varNull都表示没有赋值的变体，但是它们在表达式中的行为是不一样的。varNull值在表达式中会传播（使之称为一个null表达式），而varEmpty值则会悄悄地消失。

组件中的服务器

当为服务器或其他Automation服务器创建客户端程序时，可以使用更好的方法：在COM服务器上封装Delphi组件。如果查看TlibdemoLib_TLB文件的最后一部分，可以发现从TOLeServer继承来的TFirstServer类的声明。这是在导入库时所产生的组件，并且在单元的Register过程进行注册。

如果向数据包添加这个单元，新的服务器组件将会出现在Delphi组件面板中（默认时，在ActiveX页面）。该组件代码的生成受到Import Type Library对话框底部的复选框的控制，如图12.5所示。

我创建了一个新包PackAuto，在目录中具有相同的名字。在该包以及Project Options对话框的Directories/Conditional页面中，我添加了指令LIVE_SERVER_AT_DESIGN_TIME。这个指令将启用一个额外的功能（默认时没有该功能）：在设计时，服务器组件将有额外的属性，作为Automation服务器所有属性的子项，如下图所示。



警告：LIVE_SERVER_AT_DESIGN_TIME指令应该用在最复杂的Automation服务器中（包括像Word、Excel、PowerPoint和Visio等程序中）。对于有些服务器，在使用它们的自动化接口的一些属性之前，它们必须处于特定的模式。对于多数服务器来说，这个功能在设计时有些问题，因此默认时没有激活它。

从对象检验器中可以看出，组件没有属性。AutoConnect表示何时激活COM服务器。当该值为True时，创建封装组件（运行时和设计时）会加载服务器。若AutoConnect属性被设置为False，Automation服务器只有在其方法第一次被调用时才会加载。另一个属性ConnectKind表示如何建立与服务器的连接。它总是可以启动一个新实例（ckNewInstance），使用运行时实例（如果服务器还没有运行，ckRunningInstance会显示一个错误消息），或者选择当前实例，或者在没有实例时启动一个新的实例（ckRunningOrNew）。最后，可以用ckRemote询问远程服务器，并且直接在代码中用ckAttachToInterface手动连接到服务器。

说明：要连接到现有对象，必须在Running Object Table (ROT) 中注册。这种注册必须通过调用RegisterActiveObject API函数的服务器来执行。当然，每个COM服务器在给定时刻只能注册一个实例。

COM数据类型

COM配送不支持Delphi中的所有可用数据类型。这对于Automation来说特别重要，因为客户端和服务端通常会在不同的地址空间执行，并且系统必须将数据从一端移动到另一端。记住，用任何语言编写程序都可以访问COM接口。

COM数据类型包含基本的数据类型，例如Integer、SmallInt、Byte、Single、Double、WideString、Variant和WordBool（但是不包含Boolean）。

除了这些基本数据类型外，还可以使用复杂元素的COM类型，例如字体、字符串列表和位图，利用IFontDisp、IStrings和IPictureDisp接口。下面描述的服务器将向客户端提供一个字符串列表和一个字体。

显露字符串列表和字体

ListServ例子演示了如何在Delphi所编写的Automation服务器中显露两个复杂类型，例如字符串列表和字体。选择这两个类型，是因为Delphi支持它们。

Windows提供了IFontDisp接口，并且可用于ActiveX单元。AxCtrls Delphi单元通过提供转换方法（例如GetOleFont和SetOleFont）来扩展这个支持。Delphi支持StdVCL中的IStrings接口，AxCtrls单元提供这种类型（以及我们这里没有使用的另一个类型，IPicture）的转换函数。

警告：要运行这个应用程序和类似的应用程序，必须在客户端计算机上安装和注册StdVCL库。在安装Delphi时，它将被自动注册。

复杂类型属性的Set和Get方法可将COM接口的信息复制给本地数据，或反过来操作。例如，字符串的这两种方法是通过调用GetOleStrings和SetOleStrings Delphi函数来实现的。用于演示这个功能的客户端应用程序称为ListCli。这两个程序比较复杂，我们在这里列出了源代码，读者可以自己研究，因为Delphi程序员很少使用这种高级的技术。

使用Office程序

至此，我们已经建立了Automation连接的客户端和服务端。如果想让两个应用程序能够协同工作，有一种技术非常有用，就是第10章中所讨论的使用内存映射（另外还有一种技术不在本书范围之内，就是使用wm_CopyData消息）。Automation的实际价值是它是一种标准，使用它可以在其他应用程序中继承Delphi程序。一个典型的例子就是与Office应用程序集成，例如Microsoft Word和Microsoft Excel，或者是独立的应用程序，例如AutoCAD。

与这些应用程序集成能够提供两方面的优点：

- 让用户在自己熟悉的环境中工作——例如，以他们所熟悉的格式从数据库数据中生成报表和备忘录。
- 避免从头开始实现复杂功能，例如编写自己的字处理代码。不仅仅是重用组件，还可以重用更复杂的应用程序。

这种方法有一些缺点，需要提醒大家注意：

- 用户必须具有集成的应用程序，而且必须具备最新的版本才可以支持应用程序所提供的所有功能。
- 必须学习新的编程结构，而手头有可能会缺少这些新技术的文档资料。虽然仍然使用的是Delphi语言，但是所编写的代码取决于数据类型（也就是服务器所支持的类型），以及一些很难理解的相关类集。
- 如果想使用接口来代替变体以优化调用，则会终止一些只能在特定版本服务器上运行的程序。尤其是，Microsoft没有想维持Word和其他Office应用程序各个版本之间的脚本兼容性。

通过预安装一些封装了服务器Automation接口的可用组件，Delphi简化了Microsoft Office应用程序的使用。这些组件位于Palette的Servers页面中，安装它们的技术与前一节介绍的一样。实际上所补充的是创建组件来封装现有的Automation服务器，而不是利用预定义的服务器组件。注意，这些Office组件存在于不同的版本中，这取决于各位所使用的Office版本：所有组件都将安装，但是在设计时只注册了一个集，这取决于安装Delphi时的选择。以后要更改该设置，可以通过删除相关的组件包并添加一个新的包来实现。

在这里，没有实际的示例，这是因为编写一个适合Office不同版本的程序是很难的。在Essential Delphi中会有一些示例代码（有关如何下载本书电子图书的说明，参见附录C）。

使用复合文档

复合文档（Compound documents）是Microsoft的技术名称，其允许编辑不同的文档（例如，Word文档中的图片）。这种技术源于术语OLE，但是它的作用已经远远超出了Microsoft在20世纪90年代初引入该技术时的预想。复合文档具有两个功能：对象链接和嵌入（object linking and embedding，简称为OLE）：

- 在复合文档中嵌入一个对象相当于利用剪贴板所进行的复制和粘贴操作。主要的区别在于当复制服务器应用程序的OLE对象并粘贴到容器应用程序中时，可以复制数据和服务器的其他信息（GUID）。这允许开发人员从容器内激活服务器应用程序来

编辑数据。

- 链接一个对象到复合文档只是复制到数据和服务器信息的引用。通常，通过使用剪贴板和执行Paste Link操作，可以激活对象链接。当在容器应用程序中编辑数据时，可以修改原始数据，其保存在不同的文件中。

因为服务器程序指的是整个文件（只有部分被链接到客户文档），所以服务器会在单独的窗口中激活，并且操作整个原始文件，而不是复制的副本。当使用嵌入对象时，容器支持可视编辑，这意味着我们可以在容器主窗口中修改其中的对象。服务器和容器应用程序窗口、菜单以及工具栏会自动混合，允许我们在一个窗口中处理多个不同的对象类型——因而可以具有多个不同的OLE服务器——而不必离开容器应用程序窗口。

嵌入和链接的另一个关键区别是嵌入的对象数据由容器应用程序所保存和关联。容器用自己的文件保存嵌入的对象。相反，链接对象实际上会保存在不同的文件中，它由服务器来处理，即使是链接只引用文件的一小部分。在这两种情况中，如果没有服务器的帮助，容器应用程序不知道如何处理对象及其数据——甚至也不知道如何显示它。考虑到OLE相对较慢，而且开发COM服务器的工作量也比较大，因此可以理解为什么这种技术没有流行起来。

复合文档容器可以支持多种级别的COM。将对象放到容器的方法有多种：插入新对象，或者粘贴或粘贴-链接剪贴板中的对象，或者将对象拖动到另一个应用程序中，等等。一旦对象被放到容器中，就可以使用服务器的可用操作来处理它。通常，编辑操作是默认的操作。对于有些对象，例如视频或声音剪辑，播放是默认的操作。通过右键单击插入容器的对象可以看到其所支持的操作列表，而通过选择Edit►Object菜单项，其上面的子菜单也会列出当前对象的可用操作。

容器组件

在Delphi中，要创建COM容器应用程序，可以将OleContainer组件放到一个窗体中，然后选择该组件并右键单击，以打开一个弹出菜单，这将包括Insert Object命令。当选择这个命令时，Delphi将显示标准的OLE Insert Object对话框。这个对话框允许选择该计算机中已经注册的服务器应用程序。

一旦COM对象被插入到容器中，控制容器组件的快捷菜单将包含几个更多的自定义菜单项目。新菜单项目提供了一些命令，用于更改COM对象的属性，插入另一个对象，复制现有对象或删除现有对象。该列表还包含对象的一些操作（例如Edit、Open或Play）。一旦将COM对象插入到容器中，相应的服务器会启动以允许编辑新对象。只要关闭服务器应用程序，Delphi就更新容器中的对象，并在设计时在所进行开发的Delphi应用程序中显示它。

如果查看包含组件的窗体的文字描述，可以注意到一个Data属性，它包含COM对象的数据。虽然客户程序保存了对象的数据，但是如果没有合适服务器（它必须位于运行程序的计算机中）的帮助，它便不知道如何处理和显示数据。这就意味着COM对象被嵌入。

要完全支持复合文档，程序应该提供一个菜单和工具栏或面板。这些额外的组件非常重要，因为当场编辑意味着客户端用户界面与服务器程序的用户界面是融合的。当COM对象被激活时，服务器应用程序菜单栏的下拉菜单会被添加到容器应用程序的菜单栏上。

Delphi几乎可以自动处理所有菜单混合。只需要设置容器菜单项目的正确索引,使用GroupIndex属性。奇数索引编号的菜单项目将被当前OLE对象的相应元素所替换。特别是,File(0)和Windows(4)下拉菜单属于容器应用程序。Edit(1)、View(3)和Help(5)下拉菜单(或具有这些索引号的下拉菜单组)属于COM服务器。当COM对象处于激活状态时,第二组Object(2)可以由容器用来显示处于Edit和View组之间的另一个下拉菜单。我所编写的OleCont演示程序演示了这些功能,它允许用户调用TOleContainer类的InsertObjectDialog方法来创建一个新对象。

创建了新对象之后,可以使用DoVerb方法来执行原始的动作。这个程序允许显示具有一些位图按钮的小工具栏,我已将一些TWinControl组件放到窗体中,以使用户选择它们,并禁用了OleContainer。在当场编辑中要想保持这个工具栏/面板可见,应该设置Locked属性为True。这个设置将强制面板在应用程序中显示,并且不会被服务器的工具栏所替换。

为了演示不使用这个方法会发生的情况,我在程序中添加了具有更多按钮的第二个面板。因为没有设置Locked属性,所以这种新工具栏将会被活动服务器中的工具栏所替换。如果当场编辑启动一个服务器应用程序,并显示一个工具栏,则这个服务器的工具栏会替换容器的工具栏,如图12.6所示。

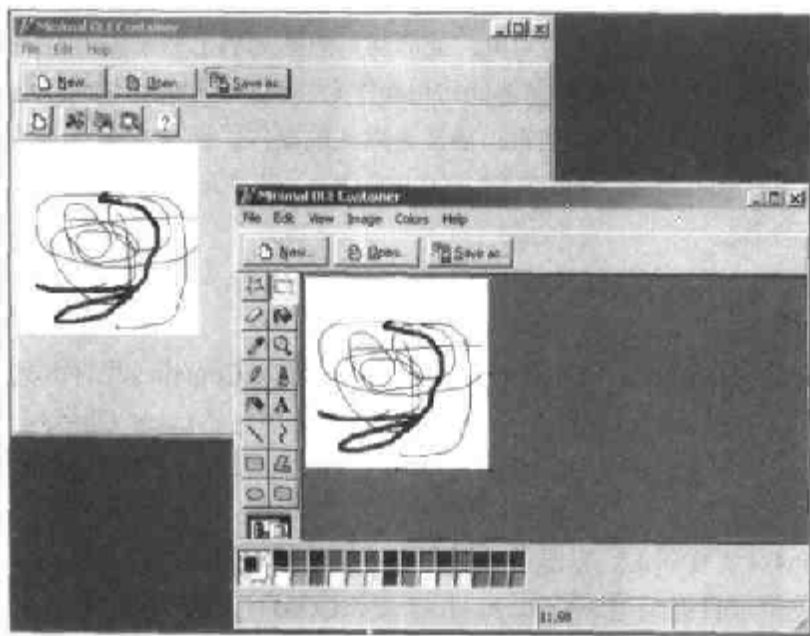


图12.6 OleCont例子的第二个工具栏(上面)被服务器的工具栏所替换(下面)

提示:要使得所有自动调整大小的操作顺利进行,则应该将OLE容器组件放到面板组件中,并将它们与窗口的客户区域对齐。

或者,可以创建一个使用PasteSpecialDialog方法的COM对象,在例子的PasteSpecialClick事件处理器中调用。另一个标准的COM对话框封装在Delphi函数中,它显示了对象的属性;通过调用OleContainer组件的ObjectPropertiesDialog方法,使用Edit下拉菜单中的Object Properties项目可以激活该对话框。

OleCont程序的最后功能是支持文件。这是最简单的功能之一,因为OLE容器组件已经提供了文件支持。

使用内部对象

在前面的程序中，用户确定了由程序所创建的内部对象类型。在这里，为了与内部对象交互，还有一点事情要做。假设，想在Delphi应用程序中嵌入一个Word文档，并用代码修改它。为此，可以使用具有嵌入对象的Automation，就如WordCont例子所演示的。

警告：因为WordCont例子包含了特定类型的对象（Microsoft Word文档），所以如果没有安装该应用程序，它便不会运行。具有不同版本的服务器也会导致一些问题（我已经用Office 97测试了本章中的例子）。对于其他版本，可能需要按照我所介绍的步骤来重新建立程序。

在该例子的窗体中，我们添加了一个OleContainer组件，设置它的AutoActive数据为aaManual（这样惟一可能的交互就在我的代码中），并且添加了一个按钮的工具栏。这个代码很简单，只要知道嵌入的对象对应于Word文档即可。下面是这个例子的代码（图12.7显示了其结果）：

```
procedure TForm1.Button3Click(Sender: TObject);
var
  Document, Paragraph: Variant;
begin
  // activate if not running
  if not (OleContainer1.State = osRunning) then
    OleContainer1.Run;
  // get the document
  Document := OleContainer1.OleObject;
  // add paragraphs, getting the last one
  Document.Paragraphs.Add;
  Paragraph := Document.Paragraphs.Add;
  // add text to the paragraph, using random font size
  Paragraph.Range.Font.Size := 10 + Random (20);
  Paragraph.Range.Text := 'New text (' +
    IntToStr (Paragraph.Range.Font.Size) + ')'#13;
end;
```

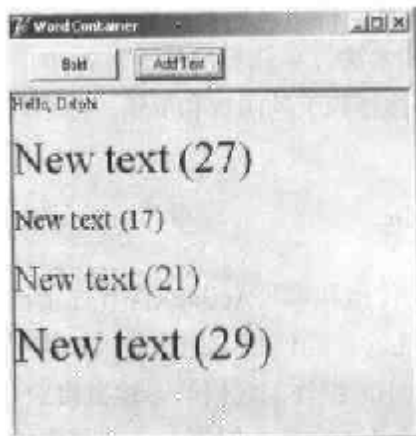


图12.7 WordCont示例演示了如何使用具有嵌入对象的Automation

介绍ActiveX控件

Microsoft的Visual Basic是第一个引入支持软件组件的程序开发环境，虽然重用软件组件的概念要比Visual Basic悠久。Visual Basic是完全基于面向对象的编程（OOP）。Visual Basic所提出的第一个技术标准是VBX，16位规格（Delphi 1支持）。在移植到32位平台时，Microsoft使用了功能更强大、更开放的ActiveX控件来代替VBX标准。

说明：ActiveX控件以前称为OLE控件（OCX）。名称的变换反映了Microsoft的新市场策略，而不是新的技术革新。所以，ActiveX控件常常被保存为.ocx扩展名的文件。

一般说来，ActiveX控件与Windows控件没有很大的区别。主要的区别是控件接口——控件与应用程序其他部分的交互。通常的Windows控件使用基于消息的接口；Automation对象和ActiveX控件使用属性、方法和事件（类似于Delphi的组件）。

用COM的术语来讲，ActiveX控件“被作为进程内服务器DLL的复合文档对象，并支持Automation、可视编辑和随时激活。”这样说明清楚吗？下面我们将解释这个定义。实现COM服务器有三种方法：

- 作为单独的应用程序（例如，Microsoft Excel）
- 作为进程外的服务器——也就是说，执行文件不能够自己运行，只能够被服务器所调用（例如，Microsoft Graph和类似的应用程序）
- 作为进程内服务器，例如在程序使用的时候装载到相同内存空间中的DLL

ActiveX控件只能够使用最后一个技术来实现，这也是最快的一种技术：作为进程内服务器。而且，ActiveX控件是Automation服务器。这意味着可以访问这些对象的属性，并调用其方法。使用ActiveX控件的应用程序总可以看到它们，并可以直接与容器应用程序窗口交互。这就是可视编辑的意思。一次单击可激活控件，而不是OLE所使用的双击。无论何时显示该控件，都表示它处于激活状态并且无需双击它（这就是随时激活的意思）。

在ActiveX控件中，属性能够识别状态，但是它们还可以激活方法。属性可以引用集合值、数组、子物体，等等。ActiveX控件的属性被划分为几组：多数控件需要实现的保留属性；提供容器信息的环境属性（类似于Delphi中的ParentColor和ParentFont属性）；容器管理的扩展属性，例如对象的位置；自定义属性，可以是其他任何东西。

事件和方法就是事件和方法。事件相关的有鼠标单击、键被按下、组件激活以及其他特定的用户行为。方法是与控件相关的函数和步骤。在事件和方法方面，ActiveX和Delphi没有多大的区别。

ActiveX控件与Delphi组件

在演示在Delphi中如何使用和编写ActiveX控件之前，我们先来讨论两种控件之间的区别。ActiveX控件是基于DLL的：当使用它们的时候，需要在使用它们的应用程序中发布代码（OCX文件）。在Delphi中，组件的代码静态地链接到可执行文件，或动态地依靠运行时包链接到可执行文件，这样可以选择是部署一个大型文件，还是部署很多较小模块。

有很多个单独文件允许开发人员可以在不同应用程序中共享代码，就像共享DLL那样。如果两个应用程序使用相同的控件（或运行时包），则只需要在硬盘上有它的一个拷贝并在

内存中也有一个拷贝。然而,缺点是如果两个程序使用ActiveX控件的两个不同类型的版本,则会出现兼容性问题。具有一个自包含执行文件的另一个优点是没有安装问题。

使用Delphi组件的缺点是Delphi组件要比ActiveX组件少,但是如果购买Delphi组件,则只能够在Delphi和Borland C++ Builder中使用它。如果购买ActiveX控件,则可以在很多开发环境中使用。即使这样,如果主要是利用Delphi来开发,并且有基于这两种技术的两个类似组件,我还是建议购买Delphi组件——它与Delphi环境结合更好,因此容易使用。而且,本机的Delphi组件可能有更好的文档,并且它还可以充分利用普通ActiveX接口(一般是基于C和C++)中没有的Delphi语言特性。

说明:在.NET领域,这种情形会完全发生变化。不仅能够用更无缝的方法来使用任何系统组件,而且还使得Delphi组件可用于其他.NET编程语言和工具中。

在Delphi中使用ActiveX控件

Delphi提供了一些预安装的ActiveX控件,也可以购买和安装第三方的ActiveX控件。我们先来描述ActiveX控件的一般工作方法,然后用一个例子来演示。

Delphi的安装过程非常简单:

1. 在Delphi菜单中选择Component►Import ActiveX Control,以打开Import ActiveX对话框,这里我们将看到已在Windows中注册的ActiveX控件库列表。
2. 选择一个库,Delphi将读取其类型库,列出其控件,并给出一个单元的建议名称。
3. 如果信息正确,单击Create Unit按钮,查看Delphi语言源代码文件,它由IDE创建并作为ActiveX控件的封装器。
4. 单击Install按钮,向Delphi包和组件面板添加一个新单元。

使用WebBrowser控件

为了建立一个例子,我们使用了预安装在Delphi中的ActiveX控件。不同于第三方控件,它并不位于面板上的ActiveX页面,而是位于Internet页面。这种控件称为WebBrowser,是Microsoft Internet Explorer引擎的封装器。WebDemo示例是一个功能非常有限的Web浏览器;在其客户区域有一个TWebBrowserActiveX控件,顶部有一个控制栏,底部有一个状态栏。要移动到给定的Web页面,用户可以在工具栏的组合框中输入URL,选择一个有效的URL(从组合框中),或者单击Open File按钮来选择一个本地文件。图12.8显示了这个程序的示例。

用于选择一个Web或本地HTML文件的代码实现是在GotoPage方法中:

```
procedure TForm1.GotoPage(ReqUrl: string);
begin
    WebBrowser1.Navigate (ReqUrl, EmptyParam, EmptyParam, EmptyParam,
        EmptyParam);
end;
```

EmptyParam是预定义好的OleVariant,可用来作为一个引用参数来传递默认值。利用这种简写形式,在每次需要一个类似的参数时,可以避免创建一个空的OleVariant变量。当用户按下Open File按钮时,或当在组合框中按下Enter键或单击Go按钮时,程序将调用GotoPage

方法，然后处理WebBrowser控件的四个事件。当下载操作结束时，该程序将更新状态栏的文字以及组合框中的下拉列表：

```

procedure TForm1.WebBrowser1DownloadComplete(Sender: TObject);
var
    NewUrl: string;
begin
    StatusBar1.Panels[0].Text := 'Done';
    // add URL to combo box
    NewUrl := WebBrowser1.LocationURL;
    if (NewUrl <> '') and (ComboURL.Items.IndexOf (NewUrl) < 0) then
        ComboURL.Items.Add (NewUrl);
end;

```



图12.8 选择Delphi开发者所知页面后的WebDemo程序

另一个有用的事件是OnTitleChange，它用来更新具有HTML文档的标题，而OnStatus-TextChange事件用来更新状态栏的第二个部分。实际上，这个代码复制了状态栏第一部分的显示信息。

编写ActiveX控件

除了可以使用Delphi中的现有ActiveX控件之外，使用下面两种技术，还可以很容易地开发新的控件：

- 使用ActiveX Control Wizard将VCL控件转换为ActiveX控件。可以从现有的VCL组件开始，但是该组件必须是TWinControl的继承体（并且必须没有不合适的属性，如

果有，就需要从向导程序中删除），然后Delphi在VCL组件上封装一个ActiveX，并向该控件添加一个类型库。（在Delphi组件上封装ActiveX控件与在Delphi中使用ActiveX控件是相反的。）

- 可以创建一个ActiveForm，在其中放几个控件，并将整个窗体（不包括边界）作为ActiveX控件。这种技术常用来建立Internet应用程序，但是它也可以用来基于多个Delphi控件或基于继承自TWinControl的Delphi组件来构造ActiveX控件。

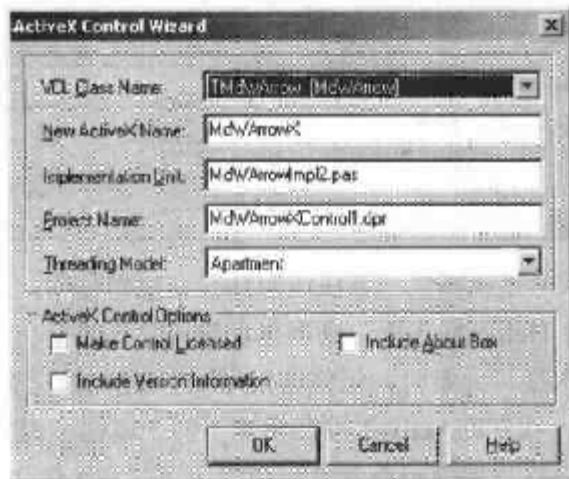
在这两种技术中，可以有选择性地准备一个控制属性页面，以使用属性编辑器在开发环境中设置控制属性的初始值——用来替换Delphi中的对象检验器。因为多数开发环境只允许有限的编辑，所以编写一个属性页面要比为Delphi控件编写一个组件或数据编辑器更重要。

建立ActiveX箭头

作为ActiveX控件的开发示例，我们决定利用第9章“编写Delphi组件”中的Arrow组件，并将它转换为ActiveX控件。但是不可以直接使用这个组件，因为它是一种图形控件（TGraphicControl的子类）。然而，将图形控件转换为基于窗口的控件则非常简单。

这里将基类名更改为TCustomControl（并且更改控件名为TMdWArrow，以避免名称冲突），正如XArrow文件夹中的源文件所示。在Delphi中安装了这个组件后，就可开发新的例子。要创建新的ActiveX库，选择File►New►Other，移动到ActiveX页面，并选择ActiveX库。Delphi创建了DLL的基本框架，如本章开始所介绍的。将该库保存为XArrow，并用相同的名称保存在一个目录中。

现在，可以使用ActiveX Control Wizard，其位于Object Repository（Delphi的新对话框）的ActiveX页面中。



在这个向导中，可以选择所感兴趣的VCL，自定义编辑框中的显示名，并单击OK，Delphi会自动地完成ActiveX控件的源代码。

ActiveX Control Wizard窗口底部的三个复选框也许并不很明显。如果要构造控件的许可证，Delphi可以在代码中包含一个许可证密钥，并且在单独的.LIC文件中提供相同的GUID。如果在设计环境中使用这个控件，但是没有该控件的正确许可证密钥，或者想在Web页面中使用它，则需要这个许可证文件。第二个复选框包含了OCX文件中ActiveX控件的版本细节。如果选中第三个复选框，ActiveX Control Wizard会自动为该控件添加一个About

方框。

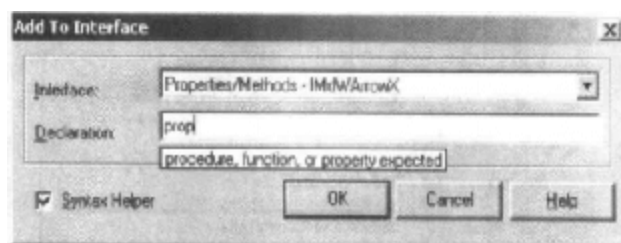
查看ActiveX Control Wizard产生的代码。该向导的主要作用是生成一个类型库，与接口定义（disinterface）一致的类型库导入单元和其他类型及常量。在这个例子中，导入文件名称是XArrow_TLB.PAS：建议仔细研究它，以理解Delphi如何定义ActiveX控件。该单元包含控件的GUID、对应于Delphi控件（例如TxMdWArrowDir）属性所使用的COM枚举类型的常量值定义，以及IMdWArrowX接口的声明。导入单元的最后部分包含TMdWArrowX类的声明。这是TOleControl继承类，使用它可以安装Delphi中的控件，就如本章最后一部分将讨论的。没有必要使用该类来建立ActiveX控件，但是需要它在Delphi中安装ActiveX控件。

剩下的代码以及自己输入的代码位于主单元中，在XArrow例子中，这个主单元就是MdWArrowImpl1。这个单元具有ActiveX服务器对象TMdWArrowX的声明，它继承自TActiveXControl，并实现了特定的IMdWArrowX接口。

在定制这个控件之前，先看看它的工作方式。编译ActiveX库，然后使用Delphi的Run>Register ActiveX Server菜单命令注册它。现在安装ActiveX控件，而且必须为新类指定一个不同的名称，以避免名称冲突。如果使用这个控件，则看上去它与原始VCL控件没有很大差别，但是现在要将相同的控件安装在其他开发环境中。

添加新属性

创建了Active控件之后，就可以为它添加新属性、事件或方法，这要比为VCL组件执行相同操作简单得多。为向ActiveX控件添加属性、方法或事件，Delphi提供了可视的界面。可以打开具有ActiveX控件实现的Delphi单元，并选择Edit>Add To Interface。或者使用编辑器快捷菜单中的相同命令。Delphi将打开Add To Interface对话框。



在组合框中，可以选择新属性、方法或事件。在编辑框中，可以输入新接口元素的声明。如果打开了Syntax Helper复选框，则当出现输入错误时会有提示并会加亮显示错误部分。在定义新的ActiveX接口元素时，记住这只局限于COM数据类型。

在XArrow示例中，我们向ActiveX控件添加了两个属性。因为原始Delphi组件的Pen和Brush属性不可用，所以这里要使得其颜色可用。以下是在Add To Interface对话框的编辑框中所输入文本的示例（执行两次）：

```
property FillColor: Integer;  
property PenColor: Integer;
```

说明：因为TColor是一个特定于Delphi的定义，所以不能够使用它。TColor是一个整数范围（默认是整数大小），因此我们直接使用标准的Integer类型。

在Add To Interface对话框中输入的声明将被自动添加到控制类型库（TLB）文件、导入库单元以及实现单元中。要完成ActiveX控件，所必须做的就是填写实现中的Get和Set方

法。如果在Delphi中再次安装这个ActiveX控件，则会出现两个新类型。该属性的惟一问题是Delphi使用一个纯整数编辑器，使得手动输入新颜色值比较困难。然而，程序可以很容易地使用RGB函数来创建合适的颜色值。

添加属性页面

如上所述，其他开发环境根本不能够使用这个组件，因为它没有属性页——没有属性编辑器。属性页是最基本的，以便使用该控件的程序员能够编辑其属性。然而，添加一个属性页要比向控件添加窗体复杂一些。属性页面将显示主机环境的属性页对话框，并提供OK、Cancel和Apply按钮，以及一些选项卡来显示多个属性页（有些属性页面是主机环境提供的。）

幸运的是Delphi支持属性页面，因此添加一个属性页面很简单。打开一个ActiveX项目，然后打开New Items对话框，移动到ActiveX页面，并选择Property Page。此时得到的与窗体完全不同——TPropertyPage1类（默认时创建的）继承自VCL的TPropertyPage类，而它又继承自TCustomForm。

提示：Delphi提供了四种内置的属性页面：颜色、字体、图片和字符串。这些类的GUID有下列常量表示：Class_DColorPropPage、Class_DFontPropPage、Class_DPicturePropPage以及Class_DStringPropPage，其位于AxCtrls单元中。

在属性页面中，可以向在一个正常的Delphi窗体添加控件，并且编写代码，以允许控件交互。在XArrow示例中，我使用Direction属性的可能值向属性页面添加了一个组合框，一个含Filler属性的复选框、一个具有UpDown控件的编辑框（用于设置ArrowHeight属性）以及两个有相应颜色的按钮状方块。当使用这个ActiveX控件时，可以在Delphi IDE中看到这个窗体，如图12.9所示。

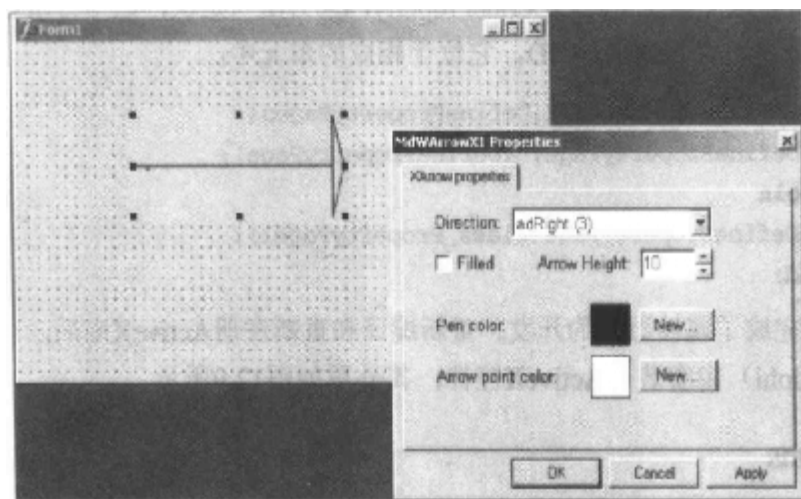


图12.9 XArrow ActiveX控件及其属性页面

添加到窗体中的惟一代码与两个按钮相关，这两个按钮被用来更改两个形状组件的颜色，并提供ActiveX控件颜色的预览。按钮的OnClick事件使用ColorDialog组件如下：

```
procedure TPropertyPage1.ButtonPenClick(Sender: TObject);
begin
  with ColorDialog1 do
  begin
    Color := ShapePen.Brush.Color;
    if Execute then
    begin
      ShapePen.Brush.Color := Color;
      Modified; // enable Apply button!
    end;
  end;
end;
```

注意该代码中对TPropertyPage类的Modified方法的调用。这个调用需要让属性页面对话框知道已修改了其中一个值，并启用了Apply按钮。当用户与窗体中的其他控件交互时，Modified会自动调用TPropertyPage类方法来处理内部的cm_Changed消息。作为一个用户，不可以更改这些控件的按钮，但是，你必须自己添加代码行。

提示：关于属性页面窗体的Caption还有另一个提示，它被用在主机环境的属性对话框中，作为属性页面选项卡的标题。

下一步是将属性页面的控件与ActiveX控件的属性相关联。这个属性页面自动就会有这个功能的两个方法：UpdateOleObject和UpdatePropertyPage。如名称所示，这些方法能将属性页面中的数据复制到ActiveX控件中，或反向操作，如示例代码所示。

最后一步是将属性页面连接到ActiveX控件。当创建控件时，Delphi ActiveX Control Wizard自动向实现单元添加一个DefinePropertyPage方法的声明。在这个方法中，可以调用要添加到控件中的每个属性页面的DefinePropertyPage方法（这次，这个方法名是唯一的），这个方法的参数是属性页面的GUID，它位于相应的单元中：

```
procedure TMDWArrowX.DefinePropertyPages(
  DefinePropertyPage: TDefinePropertyPage);
begin
  DefinePropertyPage(Class_PropertyPage1);
end;
```

至此就完成了属性页面的开发。重新编译和重新注册ActiveX库后，可以在主机开发环境（包括Delphi）中安装此ActiveX控件，其外观如图12.9所示。

ActiveForms

上面曾提到过，Delphi提供了另外一种方法代替使用ActiveX Control Wizard来创建ActiveX控件。可以使用ActiveForm，这种ActiveX控件基于一个窗体，并能够容纳一个或多个Delphi组件。Visual Basic使用这种技术来建立新的控件，并且要在创建复合组件时，它才有意义。

在XClock例子中，我给ActiveForm放置了一个标签（一个不能作为ActiveX控件起点的图形控件）和一个计时器，并用很少的代码来连接它们。窗体/控件成为其他控件的容器，

这样很容易就可以建立复合组件（要比VCL复合组件更容易）。

要建立这样的组件，需在File New对话框的ActiveX页面中，选择ActiveForm图标。Delphi会在ActiveForm Wizard对话框中询问一些信息，这类似于ActiveX Control Wizard对话框。

ActiveForm的本质

在继续这个例子之前，我们先看看ActiveForm Wizard所产生的代码，这与普通Delphi窗体的主要区别是新窗体类的声明，它继承自TActiveForm类并实现一个特定的ActiveForm接口。活动窗体类产生的代码实现了非常少的Set和Get方法，这些方法可以更改或返回Delphi窗体的相应属性；这个代码也可以实现事件，也就是窗体的事件。

TForm事件被设置为创建窗体时的内部方法。例如：

```
procedure TAXForm1.Initialize;
begin
  OnActivate := ActivateEvent;
  ...
end;
```

然后，每个事件将自身映射到外部的ActiveX事件，如下列方法所示：

```
procedure TAXForm1.ActivateEvent(Sender: TObject);
begin
  if FEvents <> nil then FEvents.OnActivate;
end;
```

由于有这个映射，所以不需要直接处理窗体的事件。相反，可以向这些默认处理程序添加代码，或者覆盖TForm方法（终止调用该事件）。这种映射问题只与窗体本身的事件有关，而与窗体组件的事件无关。可以照常处理组件的事件。

说明：XForm1示例演示了这些问题（以及解决办法）。这里不想详细讨论，由读者自己研究示例。

XClock ActiveX控件

现在已经讨论了一些基础知识，下面将返回到XClock例子的开发：

1. 在窗体上放置一个计时器，一个具有大字体的标签以及与客户端区域对齐的居中文本。
2. 为计时器的OnTimer事件编写事件处理程序，以便该控件每秒更新一次标签的输出。

```
procedure TXClock.Timer1Timer(Sender: TObject);
begin
  Labell.Caption := TimeToStr (Time);
end;
```

3. 编译此库，注册它，并将它安装到软件包中，以便于到Delphi环境中进行测试。

注意凹陷边框的效果，这是由活动窗体的AxBroderStyle属性所控制的，该属性是普通窗体不可使用的活动窗体的几个属性之一。

Web页面中的ActiveX

在前一个例子中，使用了Delphi的ActiveForm技术来创建新的ActiveX控件。ActiveForm是一种基于窗体的ActiveX控件。Borland文档暗示了ActiveForms应该用于HTML页面中，但是可以在Web页面中使用任何ActiveX控件。实际上，每当创建一个ActiveX控件，Delphi都应该启用Project►Web Deployment Options和Project►Web Deploy菜单项。

警告：由于错误（我的观点），在Delphi 7中，这些命令都只能由ActiveForm来激活。如果它们被禁用，则可以使用一个技巧：向当前的ActiveX库添加ActiveForm，这可启用这些菜单命令；然后立即删除ActiveForm，这些菜单项目仍然可用。问题是，在每次重新打开项目文件时，都需要重复这个步骤——在Borland更正这个错误之前，只能这样。

第一个命令允许程序员指定如何和在哪里递交合适的文件。在这个对话框中，可以设置要部署ActiveX组件的服务器目录、该目录的URL以及部署HTML文件的服务器目录（它有一个到使用此URL的ActiveX库的引用）。

还可以指定压缩CAB文件的使用，这些文件保存在OCX文件和其他辅助文件中，例如包文件。这就使得将应用程序交付给用户更容易、更快捷。压缩文件意味着下载更快。我们在相同的目录中产生了XClock的HTML文件和CAB文件。用Internet Explorer打开HTML文件将产生如图12.10所示的输出。如果有一个红色的X标记，则表示下载控件失效，出现这个问题的可能性有几个：Internet Explorer不允许下载控件，它不匹配未署名控件的安全级别，控制版本号不匹配，等等。

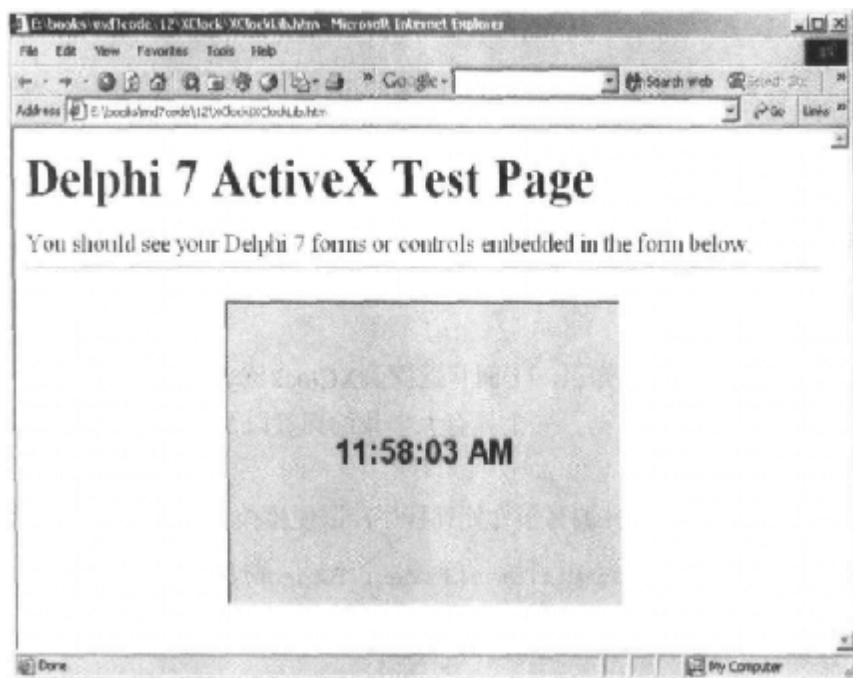


图12.10 HTML页面示例中的XClock控件

注意，在指向控件的HTML文件中，可以使用特殊的param标志来自定义控件的属性。例如，在XArrow控件的HTML文件中，我们用二个param标志修改了自动产生的HTML文件（在XArrwoCust.htm文件中）：

```
<object classid="clsid:482B2145-4133-11D3-B9F1-00000100A27B"  
  codebase="./XArrow.cab" width="350" height="250" align="center"  
  hspace="0" vspace="0">  
  <param name="ArrowHeight" value="100">  
  <param name="Filled" value="-1">  
  <param name="FillColor" value="111829">  
</object>
```

虽然这一技术看上去很有用,但更重要的是要考虑Web页面中ActiveX窗体的(有限)作用。它允许用户下载,并执行自定义的Windows应用程序,这将带来很多安全考虑。ActiveX控件可以访问计算机的系统信息,例如用户名、目录结构,等等。

介绍COM+

除了普通的COM服务器之外,Delphi还允许创建增强的COM对象,包括无状态的对象和事务支持。在Windows NT和98中,Microsoft首先引入了这种类型的COM对象,并称之为MTS(Microsoft事务服务器),但在Windows 2000/XP它被重命名为COM+(我们这里说的COM+特指MTS和COM+)。

Delphi支持建立标准的无状态对象和基于无状态对象的DataSnap远程数据模块。要开始开发它们,可以使用可用的Delphi向导程序,或者使用New Items对话框并选择ActiveX页面的Transactional Object图标或Multitier页面的Transactional Data Module图标。必须将这些对象添加到ActiveX库项目中,而不是普通应用程序中。另一个图标,COM+ Event Object,用于支持COM+事件。

COM+提供了运行时环境,支持数据库事务服务、安全、资源池以及对DCOM应用程序的整体改进。运行时环境管理称为COM+组件的对象。有些COM对象被保存在进程服务器内(例如,DLL)。而有些COM对象直接在客户应用程序中运行,COM+对象是由这类运行时环境来处理的,这个环境中安装了COM+库。COM+对象必须支持特定的COM接口,首先是IObjectControl,这是基本接口(类似于COM对象的IUnknown)。

在讨论许多技术和底层细节之前,我们先从一个不同的角度来考虑COM+: 这种方法的优点。COM+提供了几个有趣的功能,包括:

基于角色的安全 分配给客户端的角色确定了它是否有权限访问数据模块的接口。

减小数据库资源 可以减少数据库连接的数量,因为中间层登录到服务器,并使用多个客户的相同连接(虽然不可以有超过服务器许可证数量的客户连接)。

数据库事务 COM+事务支持包括多个数据的操作,虽然只有很少的SQL服务器支持Microsoft的COM+事务。

创建COM+组件

创建COM+组件要从创建ActiveX库项目开始,然后执行下列步骤:

1. 在New Items对话框的ActiveX页面选择一个新的Transactional Object。
2. 在打开的对话框中(如图12.11所示),输入新组件的名称(在我们的ComPlus1例子中是ComPlus1Object)。

New Transactional Object对话框允许输入COM+对象类、线程模块（因为COM+会序列化请求，故Single或Apartment通常会这样做）以及事务模型的名称：

Requires a Transaction（请求事务） 表示客户端每次对服务器的调用都被认为是一个事务（除非调用者提供了现有事务的上下文）。

Requires a New Transaction（请求新事务） 表示每次调用都被认为是一个新事务。

Supports Transactions（支持事务） 表示客户端必须显式提供一个事务上下文。

Does Not Support Transaction（不支持事务）（我所使用的默认选择。）表示远程数据模块不会包含在任何事务中。如果调用对象的客户端有一个事务，则这个选项能够防止这个对象被激活。

Ignores Transactions（忽略事务） 表示对象不会参与事务，但是不论客户端是否有事务，它都可以被使用。

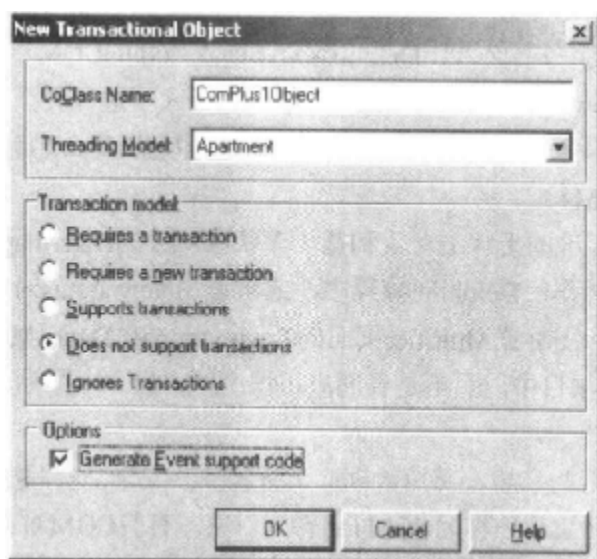
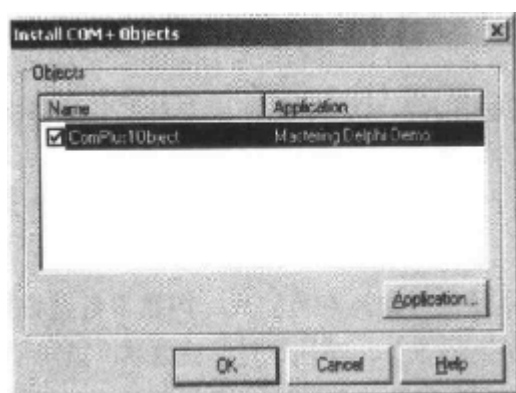


图12.11 New Transactional Object对话框，用来创建COM+对象

3. 当关闭这个对话框时，Delphi向项目添加一个类型库和一个实现单元，并打开类型库编辑器，这里可以定义新COM对象的接口。例如，添加一个Value整数属性，Increase方法（有一个增量参数）以及AsText方法（返回具有格式值的WideString）。
4. 当接受类型库编辑器中的编辑时（单击Refresh按钮或关闭窗口），Delphi显示Implementation File Update Wizard，但是只有在Environment Options对话框的Type Library页面设置了Display updates before refreshing选项，才会显示该向导。在向该类添加四个方法（包括属性的set和get方法）之前，该向导会让用户确认。现在可以编写COM对象的代码，在我们的例子中它非常简单。

编译了ActiveX库或COM库（其包含COM+组件）之后，可以使用Component Services管理工具（在Microsoft管理控制台或MMC中）来安装和配置COM+组件。最好使用Delphi IDE来安装COM+组件，并使用Run►Install COM+ Object菜单项。在随后的对话框中，可以选择要安装的组件（可以包含多个组件的库），并在要安装组件的地方选择COM+应用程序，如下图所示。



COM应用程序不仅是一种分组COM+组件的方法，它还是一个程序（为什么不称为应用程序，这我也不清楚）。因此，在Install COM+ Object对话框中，可以选择现有的应用程序/组，选择Install Into New Application页面，并输入一个名称和说明。

我们将这个COM+应用程序称为Mastering Delphi Com+ Test，如图12.12所示，其位于Microsoft的Component Services管理控制台。这个前端可用来精确调整COM+组件的行为，设置激活模型（随时激活、对象池，等等）、事务支持和要使用的安全和并发模型。还可以使用这个控制台来监视对象和方法调用（以防它们长时间执行）。如图12.12所示，可以看到两个同时活动的对象。

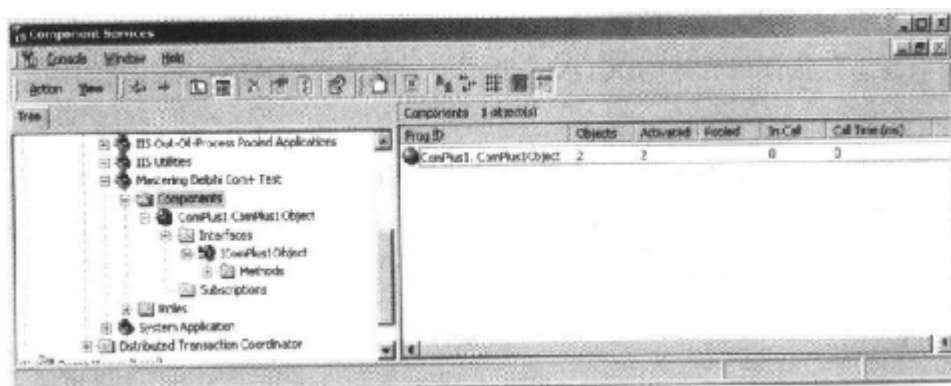


图12.12 用自定义COM+应用程序新安装的COM+组件（就像Microsoft的组件服务工具一样）

警告：因为创建了一个或多个模型，所以COM库仍然保留在COM环境中，并且某些对象可能还在缓存中，即使是没有客户端连接到它们。因此，通常不能够在使用COM库之后再编译它，除非使用MMC将其关闭，或者在MMC中设置Transaction Timeout为0。

我们已经为COM+对象创建了一个客户端程序，其类似于其他的Delphi COM客户端。导入类型库之后（类型库在安装组件时会被自动注册），我们创建了一个接口类型的变量来引用它，并调用其方法。

事务数据模块

当创建一个事务数据模块（COM+组件中的一个远程数据模块）时，可使用相同类型的功能。一旦创建了事务数据模块，就可以建立Delphi DataSnap应用程序（参见第16章“多层DataSnap应用程序”）。可以添加一个或多个数据集组件，添加一个或多个供应器，并导

出供应器。通过编辑类型库或使用Add To Interface命令,还可以向数据模块类型库添加自定义方法。

在COM+组件或事务数据模块中,可以使用支持事务的特定方法。这些方法在底层由GetObjectContext方法返回的IObjectContext接口提供:

- SetComplete告诉COM+环境,对象已经完成工作,并且应该被钝化,这样事务才可以提交。
- EnableCommit表示虽然对象还没有完成,但是事务应该提交。
- DisableCommit停止提交操作,即使执行了该方法,禁用了方法调用之间的对象钝化。
- SetAbort通知对象已经完成,并且被激活,但是事务没有提交。
- IsInTransaction检查对象是否是事务的一部分。

IContextObject接口的其他方法包括CreateInstance,它用来在相同环境中和当前事务中创建另一个COM+对象;IsCallerInrole,它用来检查对象的调用者是否是安全角色;以及IsSecurityEnabled(顾名思义)。

在服务器库中建立了事务数据模块之后,就可以安装它,就像安装普通的COM+对象一样。安装了事务数据模块之后,其他应用程序就可以直接使用它,并且它也会位于管理控制台中。

COM+的一个重要特征是它更容易配置使用这种环境的DCOM支持。客户端计算机的COM+环境可以获得服务器计算机上COM+环境的信息,包含要在网络中调用的COM+对象的注册信息。如果使用普通的DCOM,而不使用MTS或COM+,则相同的网络配置会更复杂。

提示:虽然COM+配置要比DCOM配置好,但是只局限于最新版本的Windows操作系统。考虑到就连Microsoft也正在逐渐放弃DCOM技术,所以在基于DCOM技术建立大型系统之前,应该评估一下SOAP所提供的替换技术(第22章“使用XML技术”将讨论)。

COM+事件

使用传统COM对象和Automation服务器的客户端应用程序可以调用这些服务器的方法,但是这不是一种查看服务器是否更新客户端数据的有效方法。因而,客户端可以定义实现了callback接口的COM对象,将该对象传递给服务器,并让服务器来调用它。传统的COM事件(使用IConnectionPoint接口)由Delphi的Automation对象所简化,但是处理起来仍然很复杂。

COM+引入了简化的事件模型,在这里事件是COM+组件,且COM+环境管理着连接。在传统的COM回调中,服务器对象必须跟踪它所通知的多个客户端,Delphi没有自动提供这个功能(默认的Delphi事件代码被限制为单个客户端)。要支持多个客户端的COM回调,必须添加代码来保存到每个客户端的引用。在COM+中,服务器会调用单个事件接口,并且由COM+环境将该事件转发给对此感兴趣的所有客户。这样一来,客户端和服务器很少耦合,使得客户端可以接受不同服务器的通知,而不需要更改其代码。

说明:有些人认为,Microsoft引入这个模型只是因为Visual Basic开发者用传统的方法很难处理COM事件。Windows 2000提供了一些特定于VB开发者的操作系统功能。

要创建一个COM+事件，应该创建一个COM库（或ActiveX库），并使用COM+ Event Object向导。最终的项目将包含一个类型库，它有助于发出事件的接口定义，以及一些伪实现代码。接受事件通告的服务器将提供接口实现。伪代码在这里只用来支持Delphi的COM注册系统。

当建立MdComEvents库时，我们向类型库添加了具有两个参数的一个方法，代码如下所示（在接口定义文件中）：

```
type
  IMdInform = interface(IDispatch)
    ['{202D2CC8-8E6C-4E96-9C14-1FAAE3920ECC}']
    procedure Informs(Code: Integer; const Message: WideString); safecall;
  end;
```

主单元包含伪COM对象（注意这个方法是抽象的，因此它没有实现）及其类厂，以允许服务器注册自己。此时，可以编译该库，并在COM+环境中安装它，步骤如下：

1. 在Microsoft的Component Services控制台中，选择COM+应用程序，移动到Components文件夹，并使用快捷菜单来添加新的组件。
2. 在COM Component Install Wizard中，单击Install New Event Class按钮，并选择刚才所编译的库。此COM+事件定义会被自动安装。

要测试它是否工作，必须建立这个事件接口的实现和调用它的客户端。这个实现可以被添加到另一个ActiveX库，包含一个普通的COM对象。在Delphi的COM Object Wizard中，单击List按钮，可以从显示的列表中选择要实现的接口。

我们例子中的库最终被命名为EvtSubscriber，它显露了一个Automation对象：实现了IDispatch接口的COM对象（这对于COM+对象来说是强制的）。该对象具有下列定义和代码：

```
type
  TInformSubscriber = class(TAutoObject, IMdInform)
  protected
    procedure Informs(Code: Integer; const Message: WideString); safecall;
  end;

  procedure TInformSubscriber.Informs(Code: Integer; const Message: WideString);
  begin
    ShowMessage ('Message <' + IntToStr (Code) + '>: ' + Message);
  end;
```

编译该库之后，可以首先将它安装在COM+环境中，然后将它绑定到事件。第二步是在Component Services控制台中通过选择事件对象注册项下面的Subscriptions文件夹并使用New►Subscription快捷菜单完成的。在最后的向导中，选择要实现的接口（这里可能是COM+事件库中的惟一接口）；可以看到实现该接口的COM+组件的一个列表。选择一个或多个组件建立预绑定，这将列在Subscriptions文件夹下。图12.13显示了绑定的配置示例。

最后，可以着重处理激发该事件的应用程序，我们称之为Publisher（因为它将公布其他COM对象所感兴趣的信息）。这是最简单的进程，因为它是使用事件服务器的普通COM客户。导入了COM+事件类型后，可以添加发布者代码如下：

```
var  
    Inform: TMDInform;  
begin  
    Inform := CoMdInform.Create;  
    Inform.Informs (20, Edit1.Text);
```

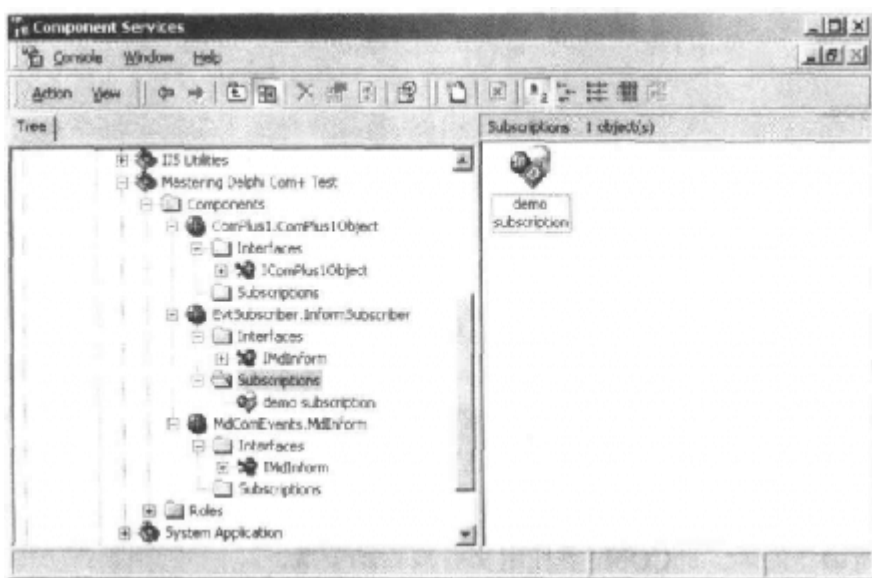
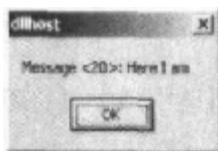


图12.13 在Component Services管理控制台中有两个预定的COM+事件

我们的例子在FormCreate方法中创建了COM对象来保留引用，但是这个效果是一样的。现在，客户端程序认为它正在调用COM+事件对象，但是这个对象（由COM+环境提供）将调用所有当前预定者的方法。在这里，可以看到一些消息框。



要使得事情更加有趣，可以向事件接口预定相同的服务器两次。实际效果是如果没有连接客户代码，则会得到两个消息框，每个预定的服务器有一个。显然，当有多个不同COM组件处理该事件时，这种效果非常有趣，因为可以很容易地在管理控制台中启用和禁用它们，更改COM+环境，而不需要修改任何程序代码。

Delphi 7中的COM和.NET

随着.NET结构的推出，Microsoft试图帮助各公司继续使用现有程序。一种移植途径是利用.NET对象与COM对象的兼容性。在.NET应用程序中可以使用现有COM对象，虽然这在管理和安全代码领域是不可以实现的。此外，还可以从Windows应用程序中使用.NET应用程序，就像它们是本机COM对象一样。这种功能的出现要感谢Microsoft所提供的封装程序。

Borland声明在Delphi 7中支持COM/.NET互操作性，这主要是因为Delphi所编译的COM对象不会给.NET导入器带来任何麻烦。此外，Delphi的类型库导入器还能够与.NET无缝地结

合，就像使用标准的COM库一样。

除非当前对COM有很大的投资，否则不建议采取这种方法。如果想将赌注压在Microsoft技术上，则将来肯定是采用.NET解决方案。如果不喜欢Microsoft技术，或者想有一个跨平台的解决方案，则选择.NET仍然要比COM好（将来，我们可以有其他操作系统的.NET框架）。

提示：这里建议的步骤在Delphi 6中也能够工作。Delphi 7添加了一个自动导入系统，利用.NET Preview版本的Delphi编译器所产生的代码有时会出现问题。

为了演示.NET导入功能，我们使用一个接口和实现它的类创建了一个.NET库。这个接口和类类似本章开始所讨论的FirstCom示例中的接口和类。这是该库的代码，在.NET Preview版本的Delphi编译器中编译。必须创建一个对象，否则链接器将删除已编译的库（用.NET术语讲，就是连编）中的所有内容：

```
library NetLibrary;

uses
  NetNumberClass in 'NetNumberClass.pas';

{$R *.res}

begin
  // create an object to link all of the code
  TNumber.Create;
end.
```

该代码位于NetNumber单元中，它定义了一个接口和实现它的类：

```
type
  INumber = interface
    function GetValue: Integer;
    procedure SetValue (New: Integer);
    procedure Increase;
  end;

  TNumber = class(TObject, INumber)
  private
    fValue: Integer;
  public
    constructor Create;
    function GetValue: Integer;
    procedure SetValue (New: Integer);
    procedure Increase;
  end;
```

注意，不同于COM服务器，该接口不需要GUID，并遵循.NET规则（虽然它有一个使用GuidAttributes类的属性）。这个系统将自动生成一个GUID。使用.NET Preview版本的Delphi（不是Delphi 7！）编译这个代码后（在本章代码的NetImport文件夹中），需要执行两个步骤：首先，运行Microsoft的.NET Framework Assembly Registration Utility——regasm；接着，运行Borland的Type Library Importer——tlibimp（理论上，应该可以省略这个步骤，并直接使用Import Type Library对话框，但是tlibimp程序需要使用这些库）。

实际上，就是进入编译库的文件夹，并从命令行输入这两条黑体所示的命令（应该还可以看到更多的文本）：

```
C:\md7code\NetImport>regasm netlibrary.dll
Microsoft (R) .NET Framework Assembly Registration Utility 1.0.3705.0
Copyright (C) Microsoft Corporation 1998-2001. All rights reserved.

Types registered successfully

C:\md7code\NetImport>tlbimp netlibrary.dll
Borland TLBIMP Version 7.0
Copyright (c) 1997, 2002 Borland Software Corporation
Type library loaded ....
Created E:\books\md7code\12\NetImport\mscorlib_TLB.dcr
Created E:\books\md7code\12\NetImport\mscorlib_TLB.pas
Created E:\books\md7code\12\NetImport\NetLibrary_TLB.dcr
Created E:\books\md7code\12\NetImport\NetLibrary_TLB.pas
```

其结果是创建项目类型库的单元和导入的Microsoft .NET Core Library (mscorlib.dll) 的单元。现在，可以创建一个新的Delphi 7应用程序（标准的Win 32程序）并使用.NET对象，就像它们是COM对象一样。下面是NetImport例子中的代码，该例如图12.14所示：

```
uses
    NetLibrary_TLB;

procedure TForm1.btnAddClick(Sender: TObject);
var
    num: INumber;
begin
    num := CoTNumber.Create as INumber;
    num.Increase;
    ShowMessage (IntToStr (num.GetValue));
end;
```

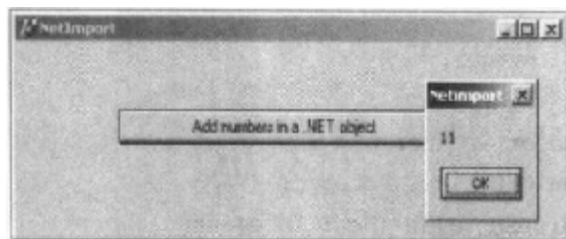


图12.14 NetImport程序使用一个.NET对象来统计数字

小结

在本章，我们讨论了Microsoft COM技术的应用，内容包括：自动化、文档、控件，等等。学习了Delphi如何使得Automation服务器和客户端以及ActiveX控件的开发更加简单。Delphi甚至允许在Automation服务器上封装组件，例如Word和Excel。此外，我们还介绍了COM+的元素，简单讨论了在浏览器中如何使用ActiveForms。我们还说明了这不是一个很好

的Internet Web编程方法——本书后面将详细讨论。

前面我们曾经提及，如果说COM在Windows 2000/XP中起着关键的作用的话，那么Microsoft将来版本的操作系统会贬低其作用，而大力推行.NET基础结构（包括SOAP和XML）。但是，本书到第23章才会详细讨论Delphi SOAP的支持。

第三部分 Delphi面向数据库的体系结构

第13章 Delphi的数据库体系结构

Delphi对数据库应用程序的支持是其编程环境的主要特性之一。很多程序员常常要花费大量时间来编写数据访问代码，因为这些代码必须是数据库应用程序中最稳定的部分。本章将综述Delphi对数据库编程的广泛支持。

这里不再讨论数据库设计的理论，我们认为读者已经掌握了数据库设计的基础知识，并曾经从事过数据库构建和设计的工作。我们也不会过深地涉及数据库专有的问题；我们的目标是帮助读者了解Delphi对数据库访问的支持。

我们首先介绍Delphi为进行数据访问提供的各种方法，然后对Delphi中可以使用的数据库组件进行综述。本章将深入分析TClientDataSet组件的使用，介绍如何用它进行本地数据访问。而客户机/服务器数据库访问在稍后的第14章“使用dbExpress的客户机/服务器编程”中进行介绍；本章中还包含对TDataSet类的介绍，并深入分析TField组件以及数据敏感控件的使用。随后的章节将讨论更高级的数据库编程专题，如使用dbExpress库（以及InterBase Express组件）的客户机/服务器编程。

最后，请注意本章中所讨论的所有内容几乎都是可跨平台的。就是说，这些例子可以使用在CLX环境中，或者通过重编译并且引用正确文件夹中的CDS文件在Linux环境中运行。

本章主要包括以下内容：

- Delphi的数据库组件
- 数据库访问方法
- 使用数据敏感控件
- DBGrid控件
- 处理表字段
- 使用标准控件的数据库应用程序

访问数据库：dbExpress、本地数据及其他

在被迅速公认为一种数据库应用程序开发工具的Delphi的早期版本中，惟一可以用于数据库数据访问的技术是Borland Database Engine (BDE)。从Delphi 3开始，与数据库访问有关的VCL部分经过改造可用于多种数据库访问的解决方案，其中包括有Microsoft的ActiveX

数据对象(ADO)、本机InterBase组件、dbExpress库以及BDE。许多第三方厂商目前已经提供了针对不同数据格式的数据库访问机制(包括一些尚不能被Borland组件访问的数据格式),并且提供了与Delphi的VCL相集成的解决方案。

提示:在Kylix中,整个情况稍有不同。Borland公司决定不将旧的BDE技术引入Linux平台,而是将重点放在新的数据库访问层,即dbExpress上

作为进一步的解决方案,为了能够简化应用程序,我们可以使用Delphi的ClientDataSet组件,将表格存储到本地的文件中,即Borland鼓吹的MyBase。请注意典型的Delphi简单应用程序是基于Paradox表格的,并且由于缺乏BDE的支持,不能够被移植到Kylix环境中。

dbExpress库

Delphi近年来适时提出的一个新特性是为Windows平台和Linux平台引入dbExpress库(DBX)。这里我们使用“库”(Library)一词而不是使用数据库引擎(DE)是因为与其他的解决方案不同,dbExpress使用了一种轻量级的方法,并且在用户机器上它基本上不需要进行配置

轻量级与可移植实际上是dbExpress的两个主要特征,也是被Borland同Kylix项目的开发一同引入的原因。与其他库相比,dbExpress实际上在能力方面是有限的。它只能访问SQL服务器(而不是本地文件),没有缓存能力并只提供了对数据的单向访问,且只能通过SQL查询来工作,而不能生成相应的SQL更新语句。

乍看起来,读者可能认为这些局限使该库不会有太大用处。恰恰相反,这些特性使之更有意义。如果我们需要生成报表,包括显示数据库内容的HTML页的话,对单向数据集进行间接更新是一种规范。如果我们想建立一个用户界面来编辑数据,可以考虑Delphi包含的特定组件(特别是ClientDataSet与Provider组件),它们提供了缓存以及对查询的解析。这些组件允许基于dbExpress的应用程序比使用独立数据库引擎的应用程序拥有更多的控制,独立的数据库引擎能够做更多额外的工作,但通常是采取它自己的方式而不是我们希望的方式。

dbExpress允许我们避开各种不同的SQL语法来编写应用程序,并且访问多个不同的数据库引擎而不需要进行太多的代码修改。受支持的SQL服务器包括Borland自己的InterBase、Oracle数据库服务器、MySQL数据库(它在Linux环境中非常流行)、Informix数据库、IBM DB2以及Delphi 7中的Microsoft SQL Server。第14章将更加详细地描述dbExpress、相关的VCL组件,并给出更多的范例。而本章主要着眼于数据库的体系结构基础。

提示:在Delphi 7中提供的支持Microsoft SQL Server的dbExpress驱动程序弥补了一个很大的缺陷。这个数据库正在广泛使用在Windows平台上,因此在开发人员进行不同数据库之间的移植的时候,也越来越多地要考虑到MS SQL Server的问题。现在,继续使用BDE的原因比以前又少了一个,因为Borland公司与Delphi 7一起发布的dbExpress驱动程序已经对很多的缺陷进行了修补。

Borland数据库引擎(BDE)

Delphi中仍然包含有BDE,它允许我们访问本地的数据库格式(如Paradox和dBase)以及SQL服务器和任何能够通过ODBC驱动程序访问的东西。这是早期Delphi版本中的标准数据库技术,但是,现在Delphi认为这个技术已经过时了,尤其是对于使用BDC通过SQL Link

驱动程序访问SQL服务器来说，更是如此。使用BDE访问本地表格仍然受到官方支持，这仅仅是因为对于这种应用程序而言，Borland还没有提出一个简单的迁移方法。

在一些情况下，一个本地表格能够由ClientDataSet组件（MyBase）替代，特别是那些临时的或者小的查询表格。但是，这种应用对于大型表格是行不通的，因为MyBase需要将整个表格装入内存空间进行访问，哪怕只是为了得到一条记录。我们建议将大型表格转移到安装在客户端计算机的SQL服务器之中。对于这种情况，InterBase是一个很好的解决办法。这种类型的迁移还能够提供对Linux平台的支持，因为，在这里BDE是不可以使用的。

当然，如果我们拥有正在使用BDE的应用程序，我们可以继续使用它们。Delphi的组件面板中的BDE页还保留了表格、查询、存储过程以及其他BDE专用的组件。我们不鼓励大家使用旧的技术来开发新的应用程序。最终，如果我们的应用程序需要一个类似的结构，或者需要保持与旧的数据库文件格式兼容，则应该使用第三方引擎来替代BDE。

说明：这是为什么笔者在这本书中去掉关于BDE内容介绍的原因。本章在以前的版本中介绍了表格与查询组件的相关信息。我们重新编写了本章，用来介绍使用ClientDataSet组件的Delphi数据库应用程序的体系结构。

InterBase Express

Borland公司为Delphi创建了另外一个数据库访问组件集：InterBase Express（IBX）。这些组件是专为Borland自有的InterBase服务器量身定制的。与dbExpress不同，这不是一种独立于服务器的数据库引擎，而是一系列用来访问特定数据库服务器的组件。如果我们只打算将InterBase用做后端RDBMS，则使用特定集合的组件可以让我们对服务器具有更多的控制，提供最好的性能，并在定制的客户机应用程序中配置与维护服务器。

说明：InterBase Express的使用强化了对特定数据库定制数据集的使用，这些数据集由第三方厂商提供，可以用于多种服务器（对于InterBase也有其他数据集组件，就像Oracle、Access、dBase文件等等一样）。

简单地讲，如果能够确保不会改变自己的数据库，并且想获得最佳性能和对灵活性与可移植性的控制的话，可以考虑使用InterBase Express（或其他同量级的组件集合）。但消极的一面是，我们获得的额外性能与控制可能是有限的，而且我们还必须学习如何使用另一个组件集合中提供的特定功能，而不是仅仅是学习如何使用通用的引擎，并将知识应用到不同的情况中去。

MyBase和ClientDataSet组件

ClientDataSet是一个用于访问内存中数据的数据集。内存中的数据可以是临时的（由程序创建，并且在退出程序的同时丢弃），从一个本地文件中载入然后再存回这个文件，或者使用一个供应器组件从其他的数据集中导入。

Borland指出，应该使用映射到一个文件并且名为MyBase的ClientDataSet组件，以指明它可以被认为是一个本地数据库解决方案。我们将在“MyBase：独立的ClientDataSet”一节中进行讨论。

从一个供应器访问数据是一个比较普通的办法，无论对于客户机/服务器体系结构（参见第14章）还是对于多层体系结构（在第16章“多层DataSnap应用程序”中进行讨论）。如

果我们使用的数据库访问组件不能提供或者只能提供有限的缓存能力，则ClientDataSet组件就变得尤其有用，这也是使用dbExpress引擎的情况。

用于ADO的dbGo

ADO代表ActiveX Data Objects的意思，是Microsoft用于数据库访问的高级接口。ADO在Microsoft的数据访问OLE DB技术上实现，提供了关系型与非关系型数据库以及电子邮件、文件系统与定制事务对象的访问。ADO作为一个引擎，其所含特性比得上BDE：数据库服务器独立支持本地与SQL服务器，是真正的重量级引擎，且简化了配置（因为它没有集中化）。理论上，安装应该不是问题，因为引擎是Windows当前版本的一部分。然而，在ADO不同版本之间有限的兼容性将迫使我们更新用户计算机上的版本，使之与我们正用于开发程序的版本相同——而且MDAC（Microsoft Data Access Components）的完全安装需要更新操作系统的大部分内容，使得该操作非常复杂。

如果我们打算使用Access或SQL Server的话，ADO有一些明确的优点，因为对于自己的数据库来讲Microsoft的驱动程序要比普通的OLE DB供应器有更好的品质。对于Access数据库来说，使用Delphi的ADO组件是一个很好的解决方案。但如果打算使用其他SQL服务器，则首先要检查具有优秀品质的驱动程序是否可以使用，否则我们可能会遇到一些令人惊讶的事。ADO的功能非常强大，但我们必须学习适应它，因为它真正地处于程序与数据库之间，提供服务但偶尔也发出我们意想不到的命令。但消极的一面是，如果打算在将来进行跨平台开发，千万不能考虑使用ADO：这种Microsoft专有的技术不能在Linux或其他操作系统上使用。

简单地讲，如果只打算在Windows上使用Access或其他Microsoft数据库，就使用ADO。否则的话，应该为打算使用的数据库服务器寻找一个好的OLE DB供应器（例如，有时要把InterBase与很多其他SQL服务器排斥在外）。

ADO组件（现在Delphi称之为dbGo）都集中在组件面板的ADO页中。其中三个核心组件是ADOConnection（用于数据库连接）、ADOCommand（用于执行SQL命令）以及ADODataset（用于执行返回结果集的请求）。还有三个兼容的组件——ADOTable、ADOQuery与ADOStoredProc——我们可以用它们将基于BDE的应用程序移植到ADO中。最后，还有一个RDSConnection组件，用于访问远程多层应用程序中的数据。

说明：第15章“使用ADO”将详细介绍ADO与相关技术。请注意微软正在使用它的.NET版本来更新ADO，它也是基于相同的核心理念的。因此，使用ADO将会提供一个很好的通往.NET应用程序的通道（虽然Borland也计划将dbExpress迁移到这个平台上）。

定制数据集组件

作为进一步的选择，我们可以编写自己定制的数据集组件，或者选择众多可用组件中的一个。开发定制的数据集组件是Delphi编程中最复杂的问题之一。我们将在第17章“编写数据库组件”中进行详细的介绍。届时，我们将能够学习TDataSet类的内部工作情况。

MyBase: 独立的ClientDataSet

如果我们希望使用Delphi编写一个单用户的数据库应用程序,则最简单的方法是使用ClientDataSet组件并且将它映射到一个本地文件中。这个本地文件映射不同于传统的数据到本地文件的映射。传统的方法是一次从文件中读取一个记录,而且可能还有一个文件用来存储索引。而ClientDataSet要将整个数据表格(可能是一个主/详结构)映射到文件中。当程序启动的时候,整个文件被装入内存,之后所有的内容都被一次性存储。

警告: 这解释了为什么我们不能在一个多用户或者多应用的环境中使用这种方法。如果两个程序或者一个程序的两个实例将相同的ClientDataSet装入内存并且修改数据,那么最后进行的存储操作将会把前面的应用程序的修改覆盖掉。

对于保存ClientDataSet内容的这种支持是在数年之前创建的,被称为公文包模型。一个用户可以从他的数据库服务器向客户机下载数据,保存一部分数据,离线工作(例如在旅行中使用一个膝上电脑进行工作),最后重新连接并提交这些变化。

连接到现有的本地表格

为了映射一个ClientDataSet到本地文件上,我们要设置它的FileName属性。为了建立一个最小的程序(叫做MyBase1),我们需要做的是连接ClientDataSet组件到一个CDS文件(在\Program Files\Common Files\Borland Shared下的Data目录中有一些可用)、一个数据源(DataSource)以及一个DBGrid控件。将ClientDataSet通过DataSource的DataSet属性连接到DataSource,并且通过DataSource属性连接DataSource到DBGrid,如程序清单13.1中显示的那样。这时,打开ClientDataSet的Active属性,将使得这个应用程序即使在设计时也可以显示数据库的数据,如图13.1所示。

程序清单13.1 MyBase1范例程序的DFM文件

```
object Form1: TForm1
  ActiveControl = DBGrid1
  Caption = 'MyBase1'
  OnCreate = FormCreate
  object DBGrid1: TDBGrid
    DataSource = DataSource1
  end
  object DataSource1: TDataSource
    DataSet = cds
  end
  object cds: TClientDataSet
    FileName = 'C:\Program Files\Common Files\Borland
      Shared\Data\customer.cds'
  end
end
```

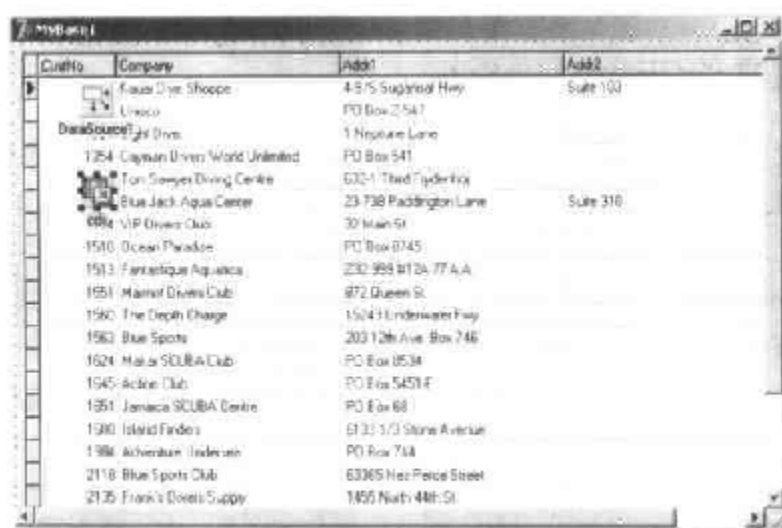



图13.1 在Delphi IDE中,设计时激活的一个范例本地表格

当我们想变更和关闭应用程序时,数据将被自动保存到文件中(可以禁用变更记录以减小数据的大小)。这时数据集也有我们在代码中使用的**SaveToFile**和**LoadFromFile**方法。

我们还做了另外一个改动,在设计时禁用了**ClientDataSet**,以避免在程序的DFM中以及编译好的可执行文件中包含它的所有数据。我们希望将这些数据保存在一个单独的文件之中。为此,在设计时关闭数据集,测试后,在窗体的**OnCreate**事件中加入一条代码来打开它:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  cds.Open;
end;
```

从Midas DLL到MidasLib单元

为了运行这个使用了**ClientDataSet**组件的应用程序,我们还需要部署由**DSIntf.pas**单元引用的**midas.dll**动态库。**ClientDataSet**组件的核心代码不是VCL的直接部分,而且在源代码格式中也不可用。这是很遗憾的事情,因为很多开发者习惯于调试VCL的源代码并且使用它作为最后的引用。

警告: **midas.dll**库在它的名称中没有版本号。因此,如果一台计算机具有一个旧的版本,则我们的程序将可以运行在它上面,但是可能不能正常工作。

Midas库是一个C语言库,但是从**Delphi 6**开始,它便可以被直接绑定进入一个可执行文件——通过将特定的**MidasLib**单元包含进来(C编译器产生的一个特殊的DCU)。在这种情况下,我们不需要使用DLL格式发布这个库。

XML和CDS格式

ClientDataSet组件支持两种不同的流格式:本机格式以及基于XML的格式。**Borland SharedDemo**文件夹中保存有两种格式的不同版本的表格。默认情况下,**MyBase**将数据集存为XML格式。**SaveToFile**方法具有一个参数,允许我们指定格式;而**LoadFromFile**方法则在

两种方法中进行自动的选择。

使用XML格式具有保证数据稳定性的好处，同时也可以保证数据能通过一个编辑器以及其他不基于ClientDataSet组件的程序来进行访问。但是，这个方法暗示着变换数据的操作，因为CDS格式与内部存储器的表现形式类似，并且总是由组件使用，而不管是哪一种流格式。同时，XML格式会生成更大的文件，因为它们是基于文本的。通常来说，一个MyBase的XML文件大约是相应的CDS文件大小的两倍。

提示：如果内存中有一个ClientDataSet，则我们可以通过XMLData属性来获取它的XML表现形式，而不必用流导出数据。下面的例子会把这个技术应用在实际之中。

定义新的本地表格

除了使我们能够连接到存储在本地文件之中的现有数据库表格外，ClientDataSet组件允许我们方便地创建新的表格。我们需要做的只是使用它的FieldDefs属性来定义这个表格的结构。这之后，可以为这个表格创建物理文件——使用Delphi IDE内的ClientDataSet组件的快捷菜单中的Create DataSet命令，或者在运行时调用CreateDataSet方法。

这是一个从MyBase2范例的DFM文件中提取的代码片断，它定义了一个新的本地数据库表格：

```
object ClientDataSet1: TClientDataSet
  FileName = 'mybase2.cds'
  FieldDefs = <
    item
      Name = 'one'
      DataType = ftString
      Size = 20
    end
    item
      Name = 'two'
      DataType = ftSmallint
    end>
  StoreDefs = True
end
```

注意StoreDefs属性，当我们编辑这个字段定义集合的时候，它被自动设为True。默认情况下，一个Delphi中的数据集在打开之前需要装入它的元数据（MetaData）。只有当一个本地定义存储在DFM文件中的时候，本地元数据才被使用（在DFM文件中存储字段定义同样有助于在客户机/服务器体系结构中对元数据进行缓存）。

为了说明可选数据集的创建，禁用日志信息（稍候介绍），以及在一个Memo控件中显示初始化数据的XML版本信息，应使程序的窗体类具有下面的OnCreate事件的处理程序：

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  if not FileExists (cds.FileName) then
    cds.CreateDataSet;
  cds.Open;
```

```

cds.MergeChangeLog;
cds.LogChanges := False;
Memo1.Lines.Text := StringReplace (
    Cds.XMLData, '>', '>' + sLineBreak, [rfReplaceAll]);
end;

```

最后一条语句包含了对于StringReplace的调用,用以提供可怜的XML格式。这段代码通过在关闭尖括号之后添加一个新行,在每个XML标记底部添加了一行代码。我们可以在图13.2中看到带有一些记录的XML格式显示的表格。我们将在第22章“使用XML技术”中讨论更多的关于Delphi中XML的内容。



图13.2 用XML格式显示范例MyBase2的CDS文件。表格的结构由程序定义,这个程序在它第一次执行的时候创建了一个文件

索引

如果有一个ClientDataSet组件在内存中,则我们可以在上面执行很多的操作。其中最简单的是索引、过滤和搜索记录,比较复杂的操作包括分组、定义合计值以及管理变化日志信息。这里,我们将只讨论最简单的技术,更复杂的内容将在本章最后进行介绍。

过滤ClientDataSet组件其实就是设置IndexFieldNames属性。这是当用户在DBGrid组件中单击字段标题时完成的(触发OnTitleClick事件)。如范例MyBase2所示:

```

procedure TForm1.DBGrid1TitleClick(Column: TColumn);
begin
    cds.IndexFieldNames := Column.Field.FieldName;
end;

```

与本地其他的数据库不同,因为索引在内存中被计算,故一个ClientDataSet能获得这个类型的动态索引,而不需要任何的数据库配置。

提示:该组件还支持基于计算字段的索引,特别是内部的计算字段,只对该数据集可用(我们将在本章后面进行讨论)。与普通的计算字段不同,普通计算字段在每一次使用记录的时候被计算,内部计算字段的值只被计算一次然后便保存在内存之中。因此,索引认为它们是普通字段。

除了给IndexFieldNames属性赋一个新值外,我们可以使用IndexDefs属性来定义一个索引。这样做允许我们定义多个索引并且保存在内存里,使得从一个索引向另一个索引转移的时候,速度会更快。

提示：定义一个不同的索引是获得递减索引而不是递增索引的惟一方法。

过滤

和用其他数据集一样，Filter属性可以在数据组件的数据集部分指定其组成部分。过滤操作发生在内存中，在载入全部记录之后。因此这是一个给用户显示较少数据的办法，不必限制一个大型本地数据集的内存使用。

当我们从一台服务器（在一个客户机/服务器体系结构中）中取回大量的数据时，我们应该尽量尝试使用一个正确的查询，这样就可以不从SQL服务器上取回大的数据集。在服务器上做前端过滤应该是我们的首选。对于本地数据，我们应该考虑将大量的记录分为一系列不同的文件，这样我们可以只载入那些我们需要的文件，而不是所有的数据。

然而，在ClientDataSet上的本地过滤十分有用，因为我们使用的过滤器表达式比那些我们能用于其他数据集使用的表达式更丰富。特别地，我们可使用下面这些：

- 标准的比较以及逻辑运算符：例如，Population > 10和Area < 1000
- 算术运算符，例如：Population / Area < 10
- 字符串函数，例如：Substring(Last_Name, 1, 2) = 'Ca'
- 日期与时间函数，例如：Year (Invoice_Date) = 2002
- 其他，包括Link函数、通配符以及In操作符

VCL帮助文件给出了这些过滤功能的说明。我们应该寻找与TClientDataSet类的Filter属性的描述连接的“限制记录显示”（Limiting what records appear）页；或者从Help的Content页，遵循下面的路径进行查找：Developing Database Applications、Using client datasets、Limiting what records appear。

定位记录

过滤允许我们限制显示给程序用户的记录输出，但是很多时候，我们希望显示所有的记录，并且只移动到某一个特定的目录上，Locate方法正是用来完成这个任务的。如果读者从未使用过Locate，首先会发现其帮助文件不是很清楚。该想法是我们必须提供一系列想要查寻的字段和一系列值，每个字段对应一个值。如果只传递一个字段，值就直接被传递，就像范例中的情况一样（在这里，搜索字符串在EditName组件中）：

```
procedure TForm1.btnLocateClick(Sender: TObject);
begin
    if not cds.Locate ('LastName', EditName.Text, []) then
        MessageDlg (''' + EditName.Text + '' not found', mtError, [mbOk], 0);
end;
```

如果我们查寻多个字段，就必须传递一个变体数组，其中含有希望进行匹配的值列表。变体数组可以使用VarArrayOf函数从常量数组中建立，或使用VarArrayCreate函数重新建立。下面是一条摘自范例的代码：

```
cds.Locate ('LastName;FirstName', VarArrayOf ([ 'Cook', 'Kevin' ]), [])
```

最后，使用相同的方法查寻记录，即使我们只知道查寻字段的初始部分。我们只需将loPartidKey标记添加到Locate调用的Options参数（第三个）中即可。

说明：使用Locate对于本地数据表格特别有意义，但不能很好地向客户机/服务器应用程序移植。在SQL服务器上，类似的客户端技术意味着要首先将所有数据移到客户机应用程序中，然后开始检索一个特定的记录。这显然不是好主意。数据定位应该通过严格的SQL语句来执行。我们仍可以在检索了一个有限的数据集后调用Locate。例如，在选择了一个给定城镇或地区的所有客户后，通过名称查寻一个客户，获得一个有限大小的结果集合。有关该话题的更多讨论见第14章，该章主要讨论客户机/服务器的开发。

Undo和SavePoint

当一个用户更改一个ClientDataSet组件中的数据的时候，这个更新将被存储在一个内存区域中，这个区域被称为Delta。追踪用户变化而不是保持最终结果表格的原因是，这些更新是发生在一个客户机/服务器体系结构中的。在这样的情况下，程序不需要发送整个表格回服务器，只需要发送一系列用户的变更（使用特定的SQL语句，正如我们将在第14章中看到的）。

因为ClientDataSet组件能够跟踪变化，故我们可以丢弃这些变化，从Delta中删除这些表项。ClientDataSet组件中有一个特殊的UndoLastChange方法，可以用来实现这种想法。这个方法FollowChange参数允许我们跟随Undo的操作：客户端数据集将移动到使用Undo操作恢复的记录上。我们使用下面的代码来连接Undo按钮：

```
procedure TForm1.ButtonUndoClick(Sender: TObject);
begin
    cds.UndoLastChange (True);
end;
```

Undo的一个扩展可以支持用于保存日志信息位置（当前状态）变更的书签，并且通过撤销所有成功的变更来恢复它。SavePoint属性既能用于在记录上保存变化的数量，也能重设日志到过去的位置。注意，我们只能从变更日志中删除记录，而不能重新插入变更。换句话说，SavePoint属性在日志中与位置相对应，因此它只能返回到前面几个记录的位置。这个记录位置反映了一些变化，如果我们存储当前的位置，撤销一些变化，然后进行更多的编辑，我们将不能够回到做过标记的位置。

提示：Delphi 7中具有新的映射到ClientDataSet的Undo操作的标准动作。其他新的动作包括Revert和Apply，当把组件连接到一个数据集以实现数据库访问的时候，我们需要这些操作。

使日志生效（Enabling）和无效（Disabling）

如果我们需要发送更新数据返回到服务器数据库，则应注意追踪变化。在把数据存于MyBase文件的本地应用程序中，追踪该日志记录既无效还浪费内存。由于该原因，我们可以用LogChanges属性禁用记录。这样，将会同时停止撤销的操作。

我们也能调用MergeChangesLog方法从变化日志记录上删除所有当前编辑，同时确认以往的编辑操作。注意，如果我们需要在单个会话中保留撤销日志，然后在不保留变化日志的情况下保存最终数据集，那么这就很有意义了。

说明：MyBase2范例中禁用了（Disable）变化日志，读者可以删除这些代码，然后重新使它生效，以察看编辑数据之后，CDS文件和XML文本的大小差异。

使用Delphi的数据敏感控件

一旦我们设置了正确的数据访问组件,就能够建立一个用户接口,以允许用户浏览数据并编辑数据。为此,Delphi提供了很多组件,它们与一般的Windows控件相似,但却是数据敏感(dataaware)控件。例如,DBEdit组件与Edit组件相似,DBCheckBox组件与CheckBox组件相似。在Delphi组件面板的Data Controls页面中,可以发现所有这些组件。

所有这些组件都使用相应的属性,DataSource,与数据资源连接。其中一些组件与整个数据集相关,如DBGrid与DBNavigator组件,而其他一些组件引用了数据资源的特殊字段,正如DataField属性表示的一样。一旦选择了DataSource属性,DataField属性将有一个可用值的列表。

注意,如果数据访问组件继承自TDataSet的话,则所有数据敏感组件都完全与数据访问技术无关。这意味着我们在用户界面上的投入在改变数据访问技术时完全可以保留下来。然而事实是,一些查询组件以及DBGrid的扩展使用(显示大量数据)只对本地数据操作才有意义,而且通常应该避免出现在客户机/服务器的情况下,这些将在第14章中介绍。

网格中的数据

DBGrid是一个可以一次显示整个数据表格的网格。它允许滚动与定位,而且可以编辑网格的内容。它是其他Delphi网格控件的扩展。

我们可以通过设置DBGrid的Options属性的各种标记并修改其Columns集合来定制它。网格允许用户使用滚动条确定数据的位置,并执行所有主要操作。用户可以直接编辑数据,在给定位置按下Insert键插入新记录,按动Down箭头键在尾部附加一个新记录,或者使用Ctrl+Del组合键删除当前记录。

Columns属性是一个集合,其中可以选择想在网格中查看的数据表格字段,并为每个字段设置列与标题属性(包括颜色、字体、宽度、对齐、标题等等)。某些更高级的属性,如ButtonStyle与DropDownRows,可以用于为网格单元或数值的下拉式列表(在列的PickList属性中指定)提供定制的编辑器。

DBNavigator与数据集操作

DBNavigator是一个按钮的集合,用于在数据库上进行定位并执行操作。我们可以通过删除VisibleButtons集合属性中的一些元素,禁用DBNavigator控件上的按钮。

这些按钮在相连的数据集上执行基本操作,所以我们可以用自己的工具栏代替它们,特别是当我们使用具有Delphi预定义数据库操作的ActionList组件的时候。事实上,在这种情况下,我们会得到所有的标准操作,但只有当这些操作合法时,我们才会看到各种可以使用的按钮。这种做法的好处是可以将这些按钮放在自己所希望它们出现的场合,并将它们与应用程序中的其他按钮放在一起并且使用多个客户控件,其中包括主菜单与弹出菜单。

提示:如果我们使用标准操作,可以避免将它们连接到特定的DataSource组件上,而且操作将应用到与(当前具有输入焦点的)可视控件相连的数据集上。这样,单个工具栏可以用于窗体显示的多个数据集,如果使用者对此不甚留意,将可能造成一些困惑。

基于文本的数据敏感控件

面向文本的组件有以下几个：

DBText 显示一个不能被用户修改的字段的内容。它是一个数据敏感的Label图形控件。它的作用很大，但用户可能会将它与（用来说明每个基于字段的控件内容的）普通标签相混淆。

DBEdit 可以使用户通过一个Edit控件来编辑字段（改变当前值）。有时，我们可能想禁止编辑功能，并将DBEdit作为一个DBText来使用，但要强调出该数据是来自数据库的数据。

DBMemo 让用户查看并修改一个大型的文本字段，最终存储在一个备注或BLOB（二进制大型对象）字段中。它类似于Memo组件，并具有全面的编辑功能，但所有文本都显示为一种字体。

基于列表的数据敏感控件

为了让用户在预定义列表中选择一个值（这样可以减少输入的错误），我们可以使用众多不同的组件。DBListBox、DBComboBox与DBRadioGroup具有一些相似之处，如在Items属性中提供一个字符串列表。但它们之间也存在有一些区别：

DBListBox 允许选择预定义项（“封闭选择”），但不能进行文本输入，而且可以用于列出很多元素。通常最好只显示六或七项，避免在屏幕上使用过多的空间。

DBComboBox 可以用于封闭选择与用户输入。DBComboBox的csDropDown样式事实上除了选择一个可用值外，还允许用户输入新值。该组件也使用了较小的窗体面积，因为下拉式列表通常只在请求时才显示。

DBRadioGroup 表现为一个单选钮（只允许一种选择），只允许封闭选择，而且应该只用于有限数量的选择范围。该组件的一个特性是，显示的值可以是那些要插入到数据库中的值，但我们还可以选择提供一些映射（Mapping）。用户界面的值（存储在Items属性中的一些描述性字符串）将映射到在数据库中存储的相应值上（列在Values属性中的一些基于数字或字符的代码）。例如，将一些部门的数字代码映射为一些描述性的字符串：

```
object DBRadioGroup1: TDBRadioGroup
  Caption = 'Department'
  DataField = 'Department'
  DataSource = DataSource1
  Items.Strings = (
    'Sales'
    'Accounting'
    'Production'
    'Management')
  Values.Strings = (
    '1'
    '2'
```

```

    '3'
    '4')
end

```

一种稍稍不同的组件是DBCheckBox，用于显示与切换一个选项，对应着Boolean字段。它是一个有限的列表，因为它只有两个可能值，加上表示不确定状态的空值。我们可以通过设置该组件的ValueChecked与ValueUnchecked属性确定返回数据库的是哪些值。

DbAware范例

DbAware范例的目的是为了强调DBRadioGroup控件以及DBCheckBox控件的使用，其中前者的设置我们在前一节已讨论过。这个范例与前面的相比并不很复杂，但是它具有一个带有面向字段的数据敏感控件的窗体，而不是使用一个表格来包括它们。我们在图13.3中可看到设计时的窗体范例。

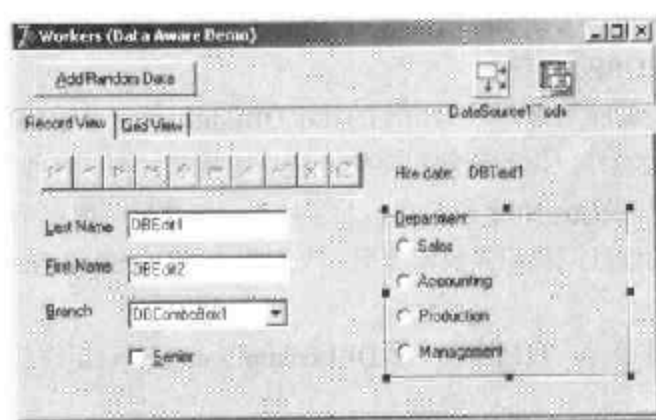


图13.3 设计时的DbAware范例的数据敏感控件

如同在MyBase2程序中那样，该应用程序定义了它自己的表格结构，并且使用了ClientDataSet的FieldDefs集合属性。表13.1提供了一个简短的字段定义摘要。

表13.1 DbAware范例中的数据表字段

名称	数据类型	大小
LastName	ftString	20
FirstName	ftString	20
Department	ftSmallint	
Branch	ftString	20
Senior	ftBoolean	
HireDate	ftDate	

程序中有一些代码使用随机值充填数据表格。因为这段代码有些冗长但不太复杂，所以我们不在此详细讨论了，如果读者有兴趣，可以查看DbAware范例的源代码。

使用查寻控件

如果数值取自另一个数据集，那么我们不能使用DBListBox¹与DBComboBox控件，而应该使用专门的DBLookupListBox或DBLookupComboBox组件。每当我们想为字段选择另一个数据集的记录时（并且不显示不同的记录），可以使用这些组件。

例如，我们在建立一个用于获取订单的标准窗体时，通常会在订单数据集中包含一个存储号码的字段，用以说明订货的客户。直接对客户号码进行操作不是最常见的方法，大多数用户喜欢对客户名称进行操作。然而，在数据库中，客户的名称存储在另一个不同的数据表格中，以避免对相同客户的每个订单重复复制客户数据。为了避免这样的情况，对于本地数据库或小的查寻表格来说，我们可以使用DBLookupComboBox控件（该技术不能很好地用于含有大型查寻表格的客户机/服务器结构，详细内容见下一章）。

DBLookupComboBox组件可以同时与两个数据源相连，一个源包含实际数据，另一个包含显示数据。我们使用了Delphi范例数据文件夹中的orders.cds文件创建了一个标准的窗体，该窗体中包含有一些DBEdit控件。

在这里，我们想删除与客户号码相连的标准DBEdit组件，并用DBLookupComboBox组件（以及一个DBText组件，用于理解程序的进程）代替它。查寻组件（和DBText组件）与订单信息DataSource以及CustNo字段相连。为了让查寻组件显示取自另一个数据文件（名为CUSTOMER.cds）的信息，我们需要添加另一个引用它的ClientDataSet组件，以及与数据表格相连的新数据源。

为了使程序正常运行，我们需要设置DBLookupComboBox1组件的多个属性。下面是相关数值的列表：

```
object DBLookupComboBox1: TDBLookupComboBox
  DataField = 'CustNo'
  DataSource = DataSourceOrders
  KeyField = 'CustNo'
  ListField = 'Company;CustNo'
  ListSource = DataSourceCustomer
  DropDownWidth = 300
end
```

和通常一样，前两个属性确定了主连接。其他四个属性确定了二级源（ListSource）、用于连接的字段（KeyField）以及需要显示的信息（ListField）。除了输入单个字段的名称外，我们可以提供多个字段，与范例中做的一样。只有第一个字段被显示为组合框文本，但如果我们为DropDownWidth属性设置一个大值，组合框下拉式列表将包含数据的多列。其输出如图13.4所示。

提示：如果我们将包含订单数据的ClientDataSet的IndexFieldNames属性设置到Company字段中，那么下拉式列表将按字母表顺序（而不是客户号码顺序）显示公司。这就是我们在范例中实现的。

图形数据敏感控件

Delphi包含了一个图形数据敏感控件，DBImage。它是Image组件的一个扩展，可以显示存储在BLOB字段中的图画，假设数据库使用了一种Image组件支持的图形格式，如BMP

与JPEG（你需要添加JPEG单元）。

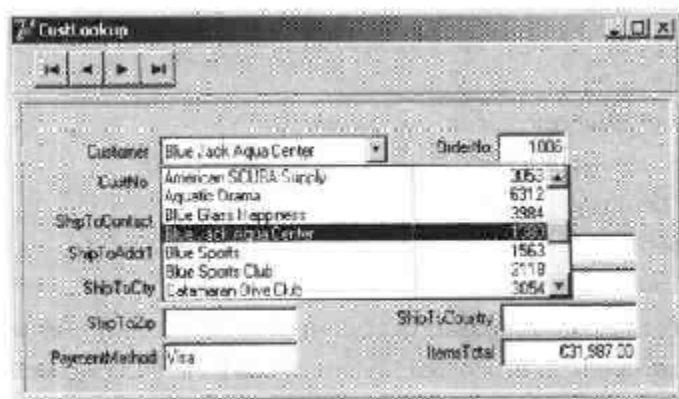


图13.4 CustLookup范例的输出，使用DBLookup组合框来显示多个字段

如果拥有的表格中包含有BLOB存储的图像，并且这个图形文件是兼容的图形格式，那么将它与组件进行挂钩就很简单。但是如果这个图形格式需要经过用户的转换后才能正确显示，那么最好使用一个非数据敏感的标准图形组件并且编写代码，以使得图像能够根据当前记录的改变而相应改变。但是，在我们对这个问题进行讨论之前，需要了解更多的关于TDataSet类以及数据集字段类的内容。

数据集（DataSet）组件

现在我们准备开始介绍TDataSet类的特性，这些特性由所有继承的数据访问类共享。DataSet组件非常复杂，所以我们将不列出它的所有功能，而只是讨论它的核心元素。

该组件提供对一系列记录的访问，这些记录读自于某些数据源，保存在内部缓冲区中（出于性能考虑），最终会被用户修改，并且将修改写回持久性的存储中。该方法非常通用，可以应用于不同类型的数据（甚至是非数据库数据），但使用这种方法也要遵循一些规则：

- 同一时间只能有一个活动的记录，所以如果需要访问多个记录中的数据，就必须在这些记录间逐个移动，以读取数据。读者将在“定位数据集”一节中发现该组件的范例与相关技术。
- 只能编辑活动记录：不能像关系型数据库中那样，同时修改一个记录集合。
- 可以修改活动缓冲区中的数据，但只有通过明确声明（通过赋给数据集Edit命令）之后才行。还可以使用Insert命令建立一个崭新的空记录，并通过Post命令关闭这两个操作（插入或编辑）。

我们要在随后小节中解释其他有趣的数据集元素，包括它的状态（以及状态改变事件）、定位与记录位置以及字段对象的作用。作为DataSet组件功能的总结，我们在程序清单13.2中包含了其类的公共方法（代码已经经过编辑与注释）。并不是所有这些列举出的方法都会被直接使用，但我们仍然决定将它们都保存在列表中。

程序清单13.2 TDataSet类的公共接口 (节选)

```

TDataSet = class(TComponent, IProviderSupport)
...
public
    // create and destroy, open and close
    constructor Create(AOwner: TComponent); override;
    destructor Destroy; override;
    procedure Open;
    procedure Close;
    property BeforeOpen: TDataSetNotifyEvent read FBeforeOpen write FBeforeOpen;
    property AfterOpen: TDataSetNotifyEvent read FAfterOpen write FAfterOpen;
    property BeforeClose: TDataSetNotifyEvent
        read FBeforeClose write FBeforeClose;
    property AfterClose: TDataSetNotifyEvent read FAfterClose write FAfterClose;

    // status information
    function IsEmpty: Boolean;
    property Active: Boolean read GetActive write SetActive default False;
    property State: TDataSetState read FState;
    function ActiveBuffer: PChar;
    property IsUniDirectional: Boolean
        read FIsUniDirectional write FIsUniDirectional default False;
    function UpdateStatus: TUpdateStatus; virtual;
    property RecordSize: Word read GetRecordSize;
    property ObjectView: Boolean read FObjectView write SetObjectView;
    property RecordCount: Integer read GetRecordCount;
    function IsSequenced: Boolean; virtual;
    function IsLinkedTo(DataSource: TDataSource): Boolean;

    // datasource
    property DataSource: TDataSource read GetDataSource;
    procedure DisableControls;
    procedure EnableControls;
    function ControlsDisabled: Boolean;

    // fields, including blobs, details, calculated, and more
    function FieldByName(const FieldName: string): TField;
    function FindField(const FieldName: string): TField;
    procedure GetFieldList(List: TList; const FieldNames: string);
    procedure GetFieldNames(List: TStrings); virtual; // virtual since Delphi 7
    property FieldCount: Integer read GetFieldCount;
    property FieldDefs: TFieldDefs read FFieldDefs write SetFieldDefs;
    property FieldDefList: TFieldDefList read FFieldDefList;
    property Fields: TFields read FFields;
    property FieldList: TFieldList read FFieldList;
    property FieldValues[const FieldName: string]: Variant
        read GetFieldValue write SetFieldValue; default;

```

```

property AggFields: TFields read FAggFields;
property DataSetField: TDataSetField
    read FDataSetField write SetDataSetField;
property DefaultFields: Boolean read FDefaultFields;
procedure ClearFields;

function GetBlobFieldData(FieldNo: Integer;
    var Buffer: TBlobByteData): Integer; virtual;
function CreateBlobStream(Field: TField;
    Mode: TBlobStreamMode): TStream; virtual;
function GetFieldData(Field: TField;
    Buffer: Pointer): Boolean; overload; virtual;
procedure GetDetailDataSets(List: TList); virtual;
procedure GetDetailLinkFields(MasterFields, DetailFields: TList); virtual;
function GetFieldData(FieldNo: Integer;
    Buffer: Pointer): Boolean; overload; virtual;
function GetFieldData(Field: TField; Buffer: Pointer; NativeFormat: Boolean):
    Boolean; overload; virtual;
property AutoCalcFields: Boolean
    read FAutoCalcFields write FAutoCalcFields default True;
property OnCalcFields: TDataSetNotifyEvent
    read FOnCalcFields write FOnCalcFields;

    // position, movement
procedure CheckBrowseMode;
procedure First;
procedure Last;
procedure Next;
procedure Prior;
function MoveBy(Distance: Integer): Integer;
property RecNo: Integer read GetRecNo write SetRecNo;
property Bof: Boolean read FBOF;
property Eof: Boolean read FEOF;
procedure CursorPosChanged;
property BeforeScroll: TDataSetNotifyEvent
    read FBeforeScroll write FBeforeScroll;
property AfterScroll: TDataSetNotifyEvent
    read FAfterScroll write FAfterScroll;

    // bookmarks
procedure FreeBookmark(Bookmark: TBookmark); virtual;
function GetBookmark: TBookmark; virtual;
function BookmarkValid(Bookmark: TBookmark): Boolean; virtual;
procedure GotoBookmark(Bookmark: TBookmark);

function CompareBookmarks(Bookmark1, Bookmark2: TBookmark): Integer; virtual;
property Bookmark: TBookmarkStr read GetBookmarkStr write SetBookmarkStr;

    // find, locate

```

```

function FindFirst: Boolean;
function FindLast: Boolean;
function FindNext: Boolean;
function FindPrior: Boolean;
property Found: Boolean read GetFound;
function Locate(const KeyFields: string; const KeyValues: Variant;
  Options: TLocateOptions): Boolean; virtual;
function Lookup(const KeyFields: string; const KeyValues: Variant;
  const ResultFields: string): Variant; virtual;

  // filtering
property Filter: string read FFilterText write SetFilterText;
property Filtered: Boolean read FFiltered write SetFiltered default False;
property FilterOptions: TFilterOptions
  read FFilterOptions write SetFilterOptions default {};
property OnFilterRecord: TFilterRecordEvent
  read FOnFilterRecord write SetOnFilterRecord;

  // refreshing, updating
procedure Refresh;
property BeforeRefresh: TDataSetNotifyEvent
  read FBeforeRefresh write FBeforeRefresh;
property AfterRefresh: TDataSetNotifyEvent
  read FAfterRefresh write FAfterRefresh;
procedure UpdateCursorPos;
procedure UpdateRecord;
function GetCurrentRecord(Buffer: PChar): Boolean; virtual;
procedure Resync(Mode: TResyncMode); virtual;

  // editing, inserting, posting, and deleting
property CanModify: Boolean read GetCanModify;
property Modified: Boolean read FModified;
procedure Append;
procedure Edit;
procedure Insert;
procedure Cancel; virtual;
procedure Delete;
procedure Post; virtual;
procedure AppendRecord(const Values: array of const);
procedure InsertRecord(const Values: array of const);
procedure SetFields(const Values: array of const);

  // events related to editing, inserting, posting, and deleting
property BeforeInsert: TDataSetNotifyEvent
  read FBeforeInsert write FBeforeInsert;
property AfterInsert: TDataSetNotifyEvent
  read FAfterInsert write FAfterInsert;
property BeforeEdit: TDataSetNotifyEvent read FBeforeEdit write FBeforeEdit;

```

```

property AfterEdit: TDataSetNotifyEvent read FAfterEdit write FAfterEdit;
property BeforePost: TDataSetNotifyEvent read FBeforePost write FBeforePost;
property AfterPost: TDataSetNotifyEvent read FAfterPost write FAfterPost;
property BeforeCancel: TDataSetNotifyEvent
    read FBeforeCancel write FBeforeCancel;
property AfterCancel: TDataSetNotifyEvent
    read FAfterCancel write FAfterCancel;
property BeforeDelete: TDataSetNotifyEvent
    read FBeforeDelete write FBeforeDelete;
property AfterDelete: TDataSetNotifyEvent
    read FAfterDelete write FAfterDelete;
property OnDeleteError: TDataSetErrorEvent
    read FOnDeleteError write FOnDeleteError;
property OnEditError: TDataSetErrorEvent
    read FOnEditError write FOnEditError;
property OnNewRecord: TDataSetNotifyEvent
    read FOnNewRecord write FOnNewRecord;
property OnPostError: TDataSetErrorEvent
    read FOnPostError write FOnPostError;

// support utilities
function Translate(Src, Dest: PChar;
    ToOem: Boolean): Integer; virtual;
property Designer: TDataSetDesigner read FDesigner;
property BlockReadSize: Integer read FBlockReadSize write SetBlockReadSize;
property SparseArrays: Boolean read FSparseArrays write SetSparseArrays;
end;

```

数据集的状态

在Delphi的数据集上进行操作时，可以在不同的状态下工作，这些状态由一个特殊的State属性确定，该属性有一些不同的值：

dsBrowse 表示数据集处于正常的浏览模式下，用于查看数据和扫描记录。

dsEdit 表示数据集处于编辑模式下。当程序调用Edit方法或DataSource的AutoEdit属性被设置为True，并且用户开始编辑数据敏感控件，如DBGrid或DBEdit时，数据集将处于该状态。当登记完改变的记录后，数据集将退出dsEdit状态。

dsInsert 表示正向数据集添加一个新记录。当调用Insert方法、移到DBGrid的最后一行，或使用DBNavigator组件的相应命令时，就会处于该状态。

dsInactive 表示已关闭的数据集的状态。

dsCalcFields 表示正在进行一个字段的计算，也就是说，在调用OnCalcFields事件处理程序期间的一种状态。

dsNewValue、**dsOldValue**和**dsCurValue** 表示正在进行缓存更新。

dsFilter 是设置过滤器时的数据集状态，也就是调用OnFilterRecord事件处理程序期间的一种状态。

在简单的范例中，这些状态之间的转换会自动处理，但理解它们是重要的，因为很多事件会引用这些状态的转换。例如，每个数据集都会在状态改变前后触发事件。当一个程序请求Edit操作时，组件会在进入编辑模式（我们可以通过引起一个异常停止该操作）之前触发BeforeEdit事件。在进入编辑模式之后，数据集会立刻接收到AfterEdit事件。在用户完成编辑并通过执行Post命令之后请求存储数据，数据集会触发BeforePost事件；而在向数据库发送数据前检查输入，在操作成功完成后触发AfterPost事件。

另一种更通用的状态改变追踪技术是处理DataSource组件的OnStateChange事件。作为一个非常简单的范例，我们可以用下面的代码显示当前状态：

```
procedure TForm1.DataSource1StateChange(Sender: TObject);
var
    strStatus: string;
begin
    case cds.State of
        dsBrowse: strStatus := Browse ;
        dsEdit: strStatus := 'Edit';
        dsInsert: strStatus := 'Insert';
    else
        strStatus := 'Other state';
    end;
    StatusBar.Panels[0].Text := strStatus;
end;
```

这段代码只考虑了三个最常见的数据集组件状态，并未考虑inactive状态和其他特殊情况。

数据集的字段

我们在前面提到，一个数据集只能有一个记录可以是当前的或活动的。记录存储在缓冲区中，而且我们可以用一些通用的方法对它进行操作，但如果要访问记录的数据，我们需要使用数据集的字段对象。这解释了字段组件（从技术上讲，类的实例都派生自TField类）在每个Delphi数据库应用程序中起着基础作用的原因。数据敏感控件直接与对应着数据库字段的字段对象连接。

默认情况下，Delphi会在运行时（即每当程序打开数据集组件时）自动建立TField组件。这是在读取了与数据表格相关的元数据或数据集引用的查询之后完成的。这些字段组件存储在数据集的Fields数组属性中。我们可以通过编号（直接访问数组）或名称（使用FieldByName方法）访问这些值。每个字段可以被用于读取或修改当前记录的数据，这是通过使用它的Value属性或类型专用属性，如AsDate、AsString、AsInteger等等实现的：

```
var
    strName: string;
begin
    strName := Cds.Fields[0].AsString
    strName := Cds.FieldByName('LastName').AsString
```

Vaule是一种变体类型属性，所以使用类型专用访问属性的效果不是很大。数据集组件还有一个快捷属性，用于访问字段的变体类型值，默认是**FieldValues**属性。默认属性意味着我们可以直接向数据集应用方括号，从而在代码中省略它：

```
strName := Cds.FieldValues ['LastName'];  
strName := Cds ['LastName'];
```

在数据集打开时建立字段组件只是一个默认行为。作为替代方法，我们还可以在设计时使用**Fields**编辑器（双击数据集可以启动**Fields**编辑器，也可以激活数据集或者是**TreeView**对象的快捷菜单并且选择**Field Editor**命令）建立字段组件。例如，在为数据表格的**LastName**列建立一个字段后，我们可以通过将**AsXxx**方法应用于相应的字段对象来引用它的值：

```
strName := CdsLastName.AsString;
```

除了用于访问字段的值外，每个字段对象还具有用于控制可视化和编辑其值的属性，包括值的范围、编辑屏蔽、显示格式、约束等等。这些属性当然要依赖于字段的类型——也就是说，依赖于字段对象的特定类。如果我们建立持久性字段，可以在设计时设置一些属性，而不用编写代码在运行时更改（可能在数据集的**AfterOpen**事件中）。

说明：尽管**Fields**编辑器与Delphi所使用的集合的编辑器相似，但字段不属于集合。它们是在设计时创建的组件，列在窗体类的**published**部分中，并可以在对象检验器顶部的下拉式组合框中使用。

当我们打开一个数据集的**Fields**编辑器时，它会显示为空。我们必须激活该编辑器或者**TreeView**对象中**Fields**伪节点的快捷菜单来访问它的功能。我们可以做的最简单的操作是选择**Add**命令，它允许我们将数据集中的任何其他字段添加到字段列表中。图13.5显示了**Add Fields**对话框，它列出了可以用于数据表格中的所有字段。这些字段是还没有出现在编辑器字段列表中的数据库表格字段。

Fields编辑器的**Define**命令用于定义新的计算字段、查寻字段或带有修改类型的字段。在该对话框中，我们可以输入一个描述性的字段名称，它可以包含空格。Delphi会生成一个内部名称——字段组件的名称——我们可以进一步定制它。然后，为字段选择一个数据类型。如果这是一个计算字段或查寻字段，而不只是被重新定义来使用新数据类型的字段副本，只需选择相应的单选按钮即可。我们将在“添加计算字段”一节中解释如何定义一个计算字段，在“查寻字段”一节中解释如何定义查寻字段。

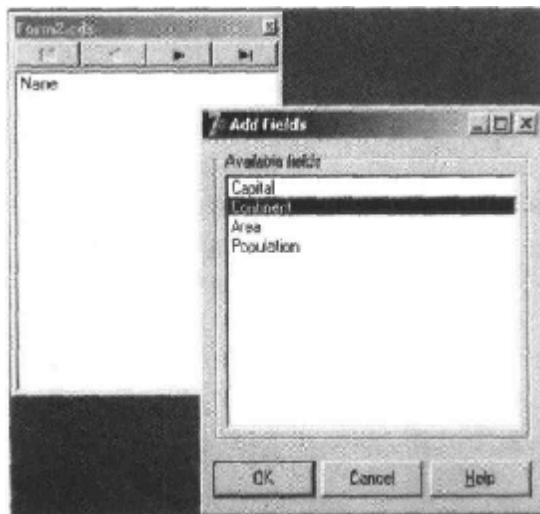


图13.5 带有Add Fields对话框的Fields编辑器

说明: TField组件具有Name属性与FieldName属性。Name属性是通常的组件名称。FieldName属性是数据库表格中列的名称或为计算字段定义的名称,它可以比Name更具描述性,并允许包含空格。TField组件的FieldName属性默认情况下被复制给DisplayLabel属性,但该字段名可以被改变为任何适当的文本。它在TDataSet类的FieldByName方法中,并在使用数组符号时用于查寻一个字段。

我们添加或定义的所有字段都包含在Fields编辑器中,并可以被数据敏感控件使用或显示在数据库网格中。如果原始数据库表格的字段不在该列表中,它将不能被访问。当我们使用Fields编辑器时,Delphi会向窗体的类添加可使用字段的声明,与新组件一样(更像是MenuDesigner向窗体添加TMenuItem组件)。TField类的组件,特别是其子类,都是窗体的字段,而且我们可以直接在程序代码中引用这些组件,在运行时改变它们的属性,或获得或设置它们的值。

在Fields编辑器中,我们还可以拖动字段改变它们的顺序。在定义网格时,适当的字段排序非常重要,网格会按照该顺序安排它的列。

提示: Fields编辑器一个甚至更好的特性是我们可以从该编辑器向窗体表面拖动字段,使用IDE来创建可视组件。这是一个非常便捷的特性,当进行数据库相关窗体的创建时,它能够为我们节省大量的时间。

使用Field对象

在研究范例之前,我们先复习一下TField类的使用。不要忽视该组件的重要性:尽管它通常在后台使用,但它在数据库应用程序中的作用是基础性的。我们曾谈到,即使我们没有定义这种专门的对象,我们也总是可以使用Fields数组属性、fieldValues索引属性或FieldByName方法访问数据表格的字段或查询。Fields属性与fieldByName函数都会返回TField类型的对象,所以有时我们必须在访问这些子类的特殊属性之前,使用as操作符将它们的结果转换为实际类型(如TFloatField或TDateField)。

FieldAcc范例是由Database Form Wizard生成的窗体的一个简单扩展。我们在工具栏中向它添加了一个快速按钮,用于在运行时访问不同的字段属性。第一个按钮用于改变网格中的Population列的格式。为此,我们必须访问DisplayFormat属性,它是TFloatField类的一个特定属性:

```
procedure TForm2.SpeedButton1Click(Sender: TObject);
begin
  (cds.FieldByName ('Population') as
    TFloatField).DisplayFormat := '###,###,###';
end;
```

当设定与数据输入或输出相关的字段属性的时候,这个变化将会应用于表格中的每一个记录。但是,当设定与字段的Value相关的属性时,通常只是引用当前的纪录。例如,通过如下代码,在一个消息框中输出当前国家的人口数量:

```
procedure TForm2.SpeedButton2Click(Sender: TObject);
begin
  ShowMessage (string (cds ['Name']) + ': ' + string (cds ['Population']));
end;
```

当访问一个字段值时, 可以使用一系列的As属性以及一个特定的数据类型(如果这个数据类型是可用的, 否则, 将会产生一个异常)来处理当前字段的值:

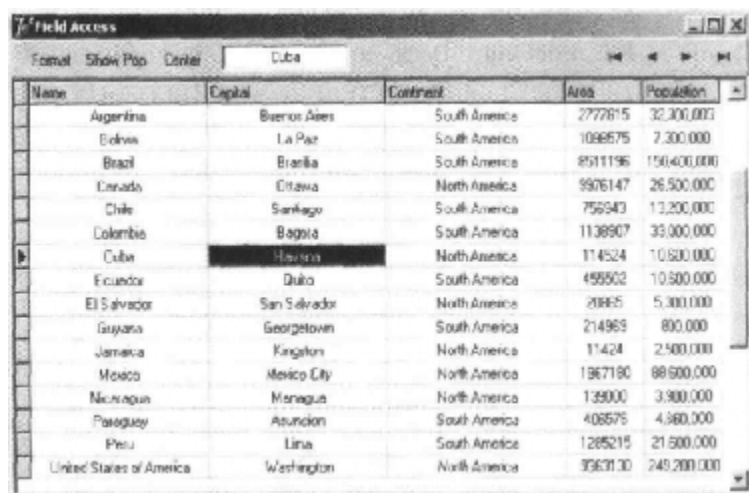
```
AsBoolean: Boolean;
AsDateTime: TDateTime;
AsFloat: Double;
AsInteger: LongInt;
AsString: string;
AsVariant: Variant;
```

这些属性可以用于读取或改变字段值。改变字段值只有在数据集处于编辑模式下才是可能的。作为上面所述As属性的一种替代方法, 我们还可以使用字段的Value属性(被定义为一个变量)来访问它的值。

TField组件的其他大多数属性, 如Alignment、DisplayLabel、DisplayWidth与Visible, 都反映了字段的用户界面中的元素, 并被各种数据敏感控件(特别是DBGrid)所使用。在FieldAcc范例中, 单击第三个快速按钮会改变每个字段的Alignment属性:

```
procedure TForm2.SpeedButton3Click(Sender: TObject);
var
  I: Integer;
begin
  for I := 0 to cds.FieldCount - 1 do
    cds.Fields[I].Alignment := taCenter;
end;
```

这个变化影响到我们加入工具栏中用来显示国家名称的DBGrid和DBEdit控件的输出。大家可以看到这个效果, 以及新的显示格式, 如图13.6所示。



The screenshot shows a window titled "Field Access" with a menu bar (Format, Show Pop, Center) and a toolbar. The "Center" button is highlighted. Below the toolbar is a table with the following data:

Name	Capital	Continent	Area	Population
Argentina	Buenos Aires	South America	2777615	32,300,000
Bolivia	La Paz	South America	1098575	7,300,000
Brazil	Brasilia	South America	8511196	150,400,000
Canada	Ottawa	North America	9976147	26,500,000
Chile	Santiago	South America	756943	12,200,000
Colombia	Bogota	South America	1138907	33,000,000
Cuba	Havana	North America	114524	10,600,000
Ecuador	Quito	South America	455502	10,500,000
El Salvador	San Salvador	North America	20665	5,300,000
Guyana	Georgetown	South America	214963	800,000
Jamaica	Kingston	North America	11424	2,500,000
Mexico	Mexico City	North America	1967190	88,600,000
Nicaragua	Managua	North America	139000	3,900,000
Paraguay	Asuncion	South America	408575	4,960,000
Peru	Lima	South America	1285215	21,500,000
United States of America	Washington	North America	3563130	245,200,000

图13.6 单击Centerand Format按钮后, FieldAcc范例的输出

Field类的层次式结构

在VCL中有一些字段类类型。在运行时打开一个数据表格或设计时使用Fields编辑器时, Delphi会自动根据数据库中的数据定义使用其中一个类型。表13.2显示了TField类完整的子类列表。

表13.2 TField类的子类

子类	基类	定义
TADTField	TObjectField	一个抽象数据类型 (ADT) 字段, 在对象关系数据库中对应一个对象字段
TAgregateField	TField	表示一个被维护的集合体。它被用在ClientDataSet组件中, 我们将在第14章讨论它
TArrayField	TObjectField	在一个对象关系数据库中的对象数组
TAutoIncField	TIntegerField	一个连接到Paradox表格的自动递增字段 (一个特殊的根据每一个记录而赋给不同的值的字段) 的正整数。请注意Paradox的AutoInc字段并不总能工作得很好, 正如在第14章中讨论的那样
TBCDField	TNumericField	小数点后具有固定位数的实数
TBinaryField	TField	通常不被直接使用, 这是下面两个类的基类
TBlobField	TField	没有大小限制的二进制数据 (Binary Large Object, BLOB)。理论上的最大限制是2GB
TBooleanField	TField	一个布尔值
TBytesField	TBinaryField	任意固定大小的大型数据 (最大为64KB)
TCurrencyField	TFloatField	货币值, 与Real数据类型有同样的范围
TDataSetField	TObjectField	一个与对象关系数据库中的单独表格相关的对象
TDateField	TDateTimeField	一个数据值
TDateTimeField	TField	一个日期和时间值
TFloatField	TNumericField	浮点数 (8字节)
TFMTBCDField	TNumericField	Delphi 6中的新字段类型。一个真的二进制编码的十进制数 (BCD), 与现有的TBCDField类型正相反, 它将BCD值转化为货币类型。这个字段类型只能够被dbExpress数据集自动使用
TGraphicField	TBlobField	一个图形的任意长度
TGuidField	TStringField	一个与COM的全局唯一标识符相关的字段 (是ADO支持的一部分)
TIDispatchField	TInterfaceField	一个与指向IDispatch COM接口的指针相关的字段 (是ADO支持的一部分)
TIntegerField	TNumericField	长整数 (32位)
TInterfacedField	TField	通常不直接使用。它是包含指向接口 (IUnknown) 的指针字段的基类
TLargeIntField	TIntegerField	极大的整数 (64位)
TMemoField	TBlobField	任意长度的文本
TNumericField	TField	通常不直接使用。这是所有数值字段类的基类

(续表)

子类	基类	定义
TObjectField	TField	通常不直接使用。这是为对象关系数据库提供支持的字段的基类
TReferenceField	TObjectField	一个指向对象关系数据库中对象的指针
TSmallIntField	TIntegerField	一个整数 (16位)
TSQLTimeStampField	TField	(Delphi 6中的新字段类型。) 支持dbExpress驱动程序中的日期/时间表达方式
TStringField	TField	具有固定长度的文本数据 (8192字节)
TTimeField	TDateTimeField	一个时间值
TVarBytesField	TBytesField	一个任意数据, 最大64KB。与TBytesField基类很相似
TVariantField	TField	一个代表变量数据类型的字段 (ADO支持的一部分)
TWideStringField	TStringField	一个代表Unicode (16位每字符) 字符串的字段
TWordField	TIntegerField	无符号 (unsigned) 整数 (16位) 或者字 (Word)

任何特定字段类型的可用性, 以及与数据定义的对应关系都要根据所使用的数据库来决定。特别是对于为对象关系数据库提供支持的新字段类型尤其如此。

添加计算字段

我们已经讨论了TField对象的使用, 并研究了一个它们运行时使用的范例, 现在该是在设计时使用Fields编辑器来建立一个基于字段对象声明的简单范例的时候了。可以从我们建立的一个范例开始, 再添加一个计算字段。我们访问的country.cds数据集中包括每个国家的人口与面积, 所以我们可以使用该数据计算人口密度。

为了建立新范例, 称为Calc, 我们要完成下面三个步骤:

1. 添加一个ClientDataSet组件到窗体。
2. 打开Fields编辑器。在这个编辑器中, 单击右键, 选择Add Field命令, 然后选择一些字段 (我们这里选择了所有的字段)。
3. 选择New Field命令并且为这个新的计算字段输入一个正确的名称以及数据类型 (将TFloatField设为浮点类型), 如图13.7所示。

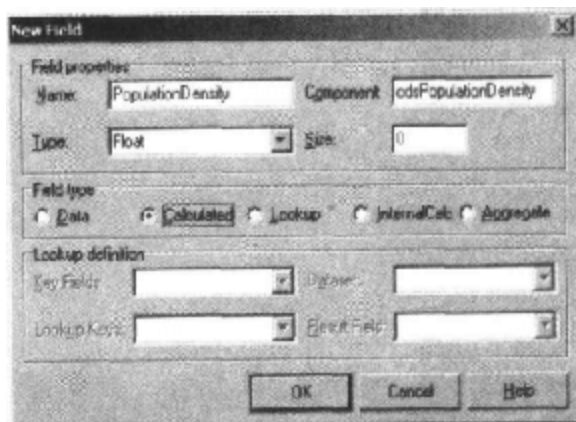


图13.7 在Calc范例中一个计算字段的定义

警告：显然，当我们使用Fields编辑器在设计时建立一些字段组件时，忽略的字段将不会获得相应的对象，正如我们强调过的，忽略的字段甚至在运行时也不可用（通过Fields或FieldByName）。当程序在运行时打开数据表格时，如果没有设计时的字段组件，Delphi会建立与数据表格定义相应的字段对象。但是，如果有设计时字段，Delphi将使用那些字段，而不添加任何额外的字段。

当然，我们还需要提供计算新字段的方法；这可以在ClientDataSet组件的OnCalcFields事件中实现，它具有下列代码（至少第一个版本是这样的）：

```
procedure TForm2.cdsCalcFields(DataSet: TDataSet);
begin
    cdsPopulationDensity.Value := cdsPopulation.Value / cdsArea.Value;
end;
```

说明：计算字段会对每个记录进行计算，并且在每次记录被装载到内部缓冲区时重新计算，一遍遍触发OnCalcFields事件。因此，该事件的处理程序执行速度应该非常快，而且不能通过访问不同记录来改变数据集的状态。计算字段的一个时间效率更高（但消耗内存更大）的版本由ClientDataSet组件通过“内部计算”字段来提供，它只在装载时计算一次，结果存储在内存中以备用。

到现在也并没有万事俱备。如果输入一个新记录而不设置人口与面积的值，或如果不小心设置面积为0，则计算会引起一个异常，这使得继续使用程序非常成问题。对此有一种解决方法，我们可以处理运算表达式的每个异常，并简单地将结果设置成0：

```
try
    cdsPopulationDensity.Value := cdsPopulation.Value / cdsArea.Value;
except
    on Exception do
        cdsPopulationDensity.Value := 0;
end;
```

然而，我们还可以做得更好：检测面积值是否被定义了——它是否是一个空值——以及它是否等于0。当可以预料可能的错误条件时，避免发生异常当然更好：

```
if not cdsArea.IsNull and (cdsArea.Value <> 0) then
    cdsPopulationDensity.Value := cdsPopulation.Value / cdsArea.Value
else
    cdsPopulationDensity.Value := 0;
```

上述cdsCalcFields方法的代码（在三个版本中都是这样）直接访问了一些字段。这是可能的，因为我们使用了Fields编辑器，而且它能自动产生相应的字段声明，就像在下面窗体界面声明中看到的一样：

```
type
    TCalcForm = class(TForm) *
        cds: TClientDataSet;
        cdsPopulationDensity: TFloatField;
        cdsArea: TFloatField;
        cdsPopulation: TFloatField;
        cdsName: TStringField;
```

```

cdsCapital: TStringField;
cdsContinent: TStringField;
procedure cdsCalcFields(DataSet: TDataSet);
...

```

每当在Fields编辑器中添加或删除字段时，可立即在窗体内出现的网格中看到操作的效果（除非网格具有自己定义的列对象，在这种情况下我们将不会看到任何变化）。当然，在设计时不会看到计算字段的值，它们只有在运行时才可以使用，因为它们是编译后的Delphi代码执行的结果。

因为我们已经为字段定义了一些组件，所以可以使用它们定制网格的一些可视元素。例如，要设置每一个千位用一个逗号分隔的显示格式，可以使用对象检验器改变一些字段组件的DisplayFormat属性为“###, ###, ###”。该变化会立即在设计时对网格产生影响。

说明：我们刚提到的（以及在前面的范例中使用过的）显示格式使用Windows International Settings来格式化输出。当Delphi将该字段的数字值转换为文本时，格式化字符串中的逗号会被相应的ThousandSeparator字符代替。因此，程序的输出会自动地使自己适应不同的International Settings。例如，在具有Italian配置的计算机上，逗号由句点代替。

设置数据表格组件与字段后，我们使用DBGrid的Columns属性编辑器对它进行了定制：设置Population Density列为只读性质，并设置它的ButtonStyle属性为cbsEllipsis，以提供一个定制的编辑器。设置该值后，当试图编辑网格单元时，会显示一个带省略号的小按钮，单击该按钮会激活DBGrid的OnEditButtonClick事件：

```

procedure TCalcForm.DBGrid1EditButtonClick(Sender: TObject);
begin
  MessageDlg (Format (
    'The population density (%.2n)'#13 +
    'is the Population (%.0n)'#13 +
    'divided by the Area (%.0n).'#13#13 +
    'Edit these two fields to change it.',
    [cdsPopulationDensity.AsFloat,
     cdsPopulation.AsFloat,
     cdsArea.AsFloat]),
    mtInformation, [mbOK], 0);
end;

```

实际上，我们并没有提供真正的编辑器，这只不过是一条简单的描述位置的消息，如图13.8所示，其中可以看到计算字段的值。要创建编辑器，可建立一个二级窗体来处理特殊的数据输入。

查寻字段

作为在窗体中放置DBLookupComboBox组件（在本章前面“使用查寻控件”一节中讨论过）的替代方法，我们还可以定义一个查寻字段，用于在一个DBGrid组件中通过下拉式查寻列表显示数据。我们曾介绍过为了向DBGrid添加一个固定的选择项，只需编辑Columns属性的PickList子属性。为了自定义带有即时查寻的网格，必须使用Fields编辑器定义一个查寻字段。

Name	Capital	Continent	Population	Area	Population Density
Argentina	Buenos Aires	South America	32,300,000	2,777,815	11.63
Bolivia	La Paz	South America	7,300,000	1,099,575	6.64
Brazil	Brasilia	South America	150,400,000	8,511,156	17.57
Canada	Ottawa	North America	28,500,000	9,976,147	2.86
Chile	Santiago	South America	13,200,000	756,543	17.44
Colombia	Bogota	South America	33,000,000	1,139,507	28.96
Cuba	Havana	North America	10,600,000	114,524	92.56
Ecuador	Quito	South America	10,600,000	455,502	23.27
El Salvador	San Salvador	North America	5,200,000	20,865	254.01

图13.8 Calc范例的输出。请注意，当我们进行编辑的时候，Population Density计算列和省略号按钮显示出来了

作为范例，我们建立了FieldLookup程序，它具有一个显示订单信息的网格，并且通过查寻字段显示完成该订单的员工名称，而不是显示该员工的员工号码。为了完成该任务，我们向数据模块添加了一个引用employee.db数据表格的ClientDataSet组件。然后，为Orders数据集打开Fields编辑器，并添加所有字段。我们选择EmpNo字段，并将它的Visible属性设置为False，从而在网格中删除它（但不能完全删除它，因为它还要用于同Employee数据集的相应字段建立交叉引用）。

现在，应该定义查寻字段了。如果读者已经完成了前面的步骤，可以使用Orders数据集的Fields编辑器，并选择New Field命令打开一个New Field对话框，我们在这里指定的数值将影响添加到数据表格中的新TField的属性，如该字段的DFM描述中解释的那样：

```
object cds2Employee: TStringField
  FieldKind = fkLookup
  FieldName = 'Employee'
  LookupDataSet = cds2
  LookupKeyFields = 'EmpNo'
  LookupResultField = 'LastName'
  KeyFields = 'EmpNo'
  Size = 30
  Lookup = True
end
```

这就是我们使用下拉式列表（如图13.9所示）并在设计时查看交叉引用字段值所需要做的全部工作。注意，不需要定制网格的Columns属性，因为下拉式按钮与七行值会默认获得。但是，这并不意味着我们不能使用该属性进一步定制网格的可视元素。

这个程序具有另外一个特性。两个ClientDataSet组件与两个DataSource组件并没有被放置在一个窗体中，而是放在一个不可见组件的容器中，被称为数据模块（参阅加说明“数据访问组件的数据模块”一节）。我们可以从Delphi的File/New菜单中获得一个数据模块。当向它添加了组件之后，我们可以从其他窗体的控件连接到它们——通过使用File/Use Unit命令。



图13.9 FieldLookup范例的输出，在网络中使用下拉列表，显示从另外一个数据库表格中获得的数据

数据访问组件的数据模块

为了建立一个Delphi数据库应用程序，我们可以在一个窗体中放置数据访问组件与数据敏感控件。这对于一个简单的程序是很方便的，但对于在单独的单元中拥有用户界面与数据访问及数据模型的程序来说不是好的想法。因此，Delphi实现了数据模型的思想，这是一个非可视组件的容器。

在设计时，数据模块与窗体相似，但在运行时，它只存在于内存中。TDataModule类直接派生自TComponent，所以它完全与窗口的Windows概念无关（完全可以在不同的操作系统之间移植）。与窗体不同，数据模块只有几个属性与事件。因此，将数据模块看做是组件与对象方法容器更有用些。

与窗体或框架类似，数据模块有一个设计器。这意味着Delphi为数据模块建立了一个专门的单元，用于定义它的类并列出其组件与属性的窗体定义文件。

使用数据模块有一些理由。最简单的理由是在多个窗体之间共享数据访问组件，就像我们要在下一章开始时要解释的那样。该技术同可视窗体链接一起使用，能够在设计时访问另一个窗体或数据模块的组件（使用File>Use Unit命令）。第二个原因是，将数据与用户界面分隔开，改进应用程序的结构。Delphi中的数据模块甚至也存在于多层应用程序（远程数据模块）与服务器端HTTP应用程序（Web数据模块）特定的版本中。

用字段事件处理零（Null）值

除了一些有趣的属性外，字段对象还有一些关键的事件。OnValidate事件可以用于提供字段值的扩展确认，而且只要我们需要复杂规则，也就是不能表达字段提供的范围与约束的时候，就应该使用它。该事件在数据写入记录缓冲区之前触发，而OnChange事件在输入被写入之后马上触发。

此外还有两个事件，OnGetText与OnSetText，可以用于定制字段的输出。这两个事件有很强的功能：它们允许使用数据敏感控件，甚至在我们要显示的字段表达方式不同于Delphi默认提供的表达方式时。

使用这些事件的一个范例是处理零值。在SQL服务器上，为字段存储空值或零值是两种独立的操作。后者更正确些，但默认情况下Delphi使用空值，并且空值字段或者零值字段的

输出是相同的。虽然它大多数情况下用于字符串和数值，但对于日期却非常重要，因为对于日期很难设置一个合理的默认值，而且如果用户空出字段，则输入是无效的。

NullDates程序会为具有零值的日期显示专门的文本，并在用户输入空字符串时清理字段（将它设置为零值）。下面是这两个事件处理程序的有关代码：

```

procedure TForm1.cdsShipDateGetText(Sender: TField;
  var Text: String; DisplayText: Boolean);
begin
  if Sender.IsNull then
    Text := '<undefined>'
  else
    Text := Sender.AsString;
end;

procedure TForm1.cdsShipDateSetText(Sender: TField; const Text: String);
begin
  if Text = '' then
    Sender.Clear
  else
    Sender.AsString := Text;
end;

```

图13.10显示了一个程序输出的例子，例子中对于某些运送日期数据，使用了未定义值或者零值。

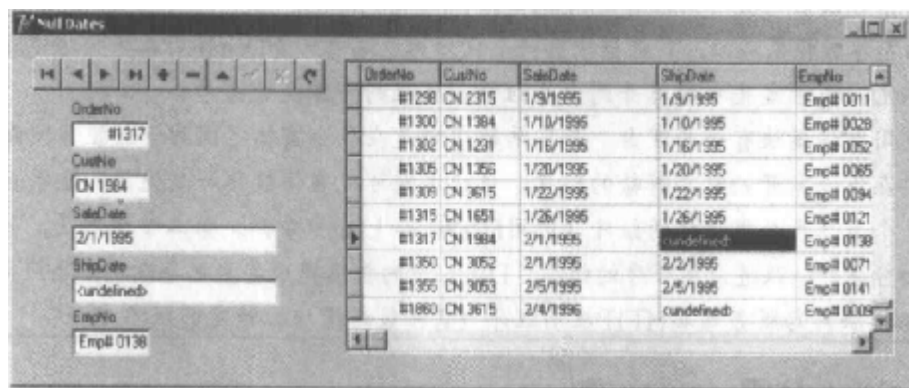


图13.10 通过处理一个数据字段的OnGetText和OnSetText事件，NullDates范例显示了空值的特殊输出

警告：在Delphi 6与7中对于零值的处理，可以通过改变零值变量工作方式的办法进行影响。正如在第3章的“变量与VarUtils单元”一节中介绍过的那样，将一个零值的字段与其他字段进行比较，在最新版本的Delphi中将会产生与以前版本不同的效果。在Delphi 7中，可以使用全局变量来优化变量间比较的效果。

定位数据集

我们曾经介绍过，一个数据集内只能有一个活动记录；而且可以想像到，活动记录经常变化，以响应用户的操作或因为赋给数据集的内部命令产生了变化。为了在数据集中来回移动并改变活动记录，TDataSet类提供了一些对象方法，读者可以查看程序清单13.2，特别是被注释为“position, movement”的部分。我们可以移到下一个或前一个记录，可以前后跳过给定数量的记录（使用MoveBy），或直接移到数据集的第一个或最后一个记录上。数据集的这些操作通常可以在DBNavigator组件或标准的数据集动作中实现，而且它们理解起来也不复杂。

可是，数据集如何处理极端位置仍然不是太清楚。如果我们打开带有导航器的任意数据集，可以看到当逐个地在记录上移动时，Next按钮会保持可使用状态，即使到达最后一个记录。只有当我们试图在最后一个记录上继续向前移动，而当前记录显然不会变化时，按钮才处于禁用状态。这是因为只有当光标移动到最后一个记录之后的一个特殊位置上时，Eof（文件尾）测试才会成功。如果用Last按钮跳到尾部，我们将达到真正的文件尾。对于第一个记录，也会存在同样的情况（即Bof测试）。接下来，我们将看到这种作法会带来很大的便利，因为我们可以扫描数据集，检查Eof是否为True，从而知道自己是否到达了数据集的最后一个记录。

除了前后移动记录外，程序还需跳到特定的记录或位置。有些数据集支持RecordCount属性，而且允许使用RecNo属性移到数据集中给定位置的记录上。这些属性只能用于本身就支持定位的数据集，它们基本上不包括所有的客户机/服务器结构，除非我们将所有记录都放在一个本地缓存中（通常我们都避免这种操作），然后在缓存上进行定位操作。我们在下一章中会看到，当我们在SQL服务器上打开一个查询时，我们只读取正使用的记录，所以Delphi不知道记录数量，至少是不能预先知道。

可以使用两种不同的方法来引用数据集中的记录，而不用考虑它的类型：

- 保存对当前记录的引用，然后在移动后跳回到它的位置。在TBookmark或更新的TBookmarkStr窗体中可以使用书签来实现。该方法将在“使用书签”一节中讨论。
- 使用Locate方法确定符合给定条件的数据集记录的位置。这甚至可以在关闭并重新打开数据集后执行，因为我们是在一种逻辑层面（而不是物理层面）上进行工作的。该方法将在下一节中进行介绍。

数据表格列的总和

目前，在我们的范例中，用户可以查看数据库表格的当前内容，并手动编辑数据或插入新记录。现在，让我们看看如何通过程序代码来改变一些数据。该范例的设计思想很简单，我们使用的Employee数据表格有一个Salary字段。公司的经理可以浏览数据表格并改变某个员工的薪金。但对于公司来说，薪金的总额是多少呢？并且如果经理想为每个员工增加（或减少）10%的薪水，又将如何操作呢？

在这段程序中，使用按钮来计算当前薪水的总和并且可以改变它们，同时这段程序也展示了对于标准的数据集操作的操作列表。总计操作允许我们计算所有员工薪水的总和。简

单地说，我们需要扫描数据表格，为每个记录读取cdsSalary字段的值：

```
var
    Total: Double;
begin
    Total := 0;
    cds.First;
    while not cds.EOF do
    begin
        Total := Total + cdsSalary.Value;
        cds.Next;
    end;
    MessageDlg ('Sum of new salaries is ' +
        Format ('%m', [Total]), mtInformation, [mbOk], 0);
end
```

这一段代码运行后，输出如图13.11所示，但是它也存在一些问题。问题之一就是记录的指针被移动到最后一个记录，因此先前在表格中的位置就丢失了。另一个问题是在操作的过程中，用户界面被刷新了很多次。

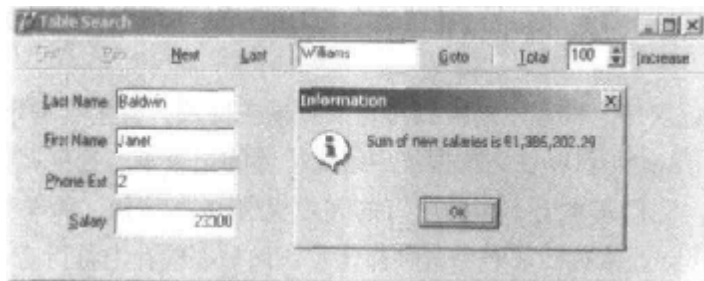


图13.11 Total程序的输出，显示了员工的总薪金值

使用书签

为了避免上面所述两个问题的发生，我们需要禁止更新并存储数据表格中记录指针的当前位置，最后对它进行恢复。可以使用表格书签达到该目的，这是存储数据库表格中记录位置的特殊变量。Delphi的传统做法是声明一个TBookmark数据类型的变量，并在从数据表格中获得当前位置时初始化它：

```
var
    Bookmark: TBookmark;
begin
    Bookmark := cds.GetBookmark;
```

在ActionTotalExecute方法的最后，我们可以恢复这个位置并且使用下面两条语句删除这个书签（在一个finally块中，保证指针内存被明确释放）：

```
cds.GotoBookmark (Bookmark);
cds.FreeBookmark (Bookmark);
```

还有一种更简单的方法，亦即使用TDataSet类的Bookmark属性，它引用了一个看起来可以自动删除的书签（从技术上讲，这可以被实现为一个不透明的字符串，这是一种用于字符串生存期管理的结构对象，但它不是一个字符串，所以不能查看其内部情况）。我们对上面的代码进行改动如下：

```
var
    Bookmark: TBookmarkStr;
begin
    Bookmark := cds.Bookmark;
    ...
    cds.Bookmark := Bookmark;
```

为了避免程序的另一副作用（当例程浏览数据时，我们可以看到记录的滚动），我们需临时关闭与数据表格相连接的控件，以避免该影响。数据表格有一个DisableControls方法，可以在while循环启动前调用；还有一个EnableControls对象方法，可以在循环结束、记录指针恢复后调用。

提示：在长时间操作期间关闭与数据表格连接的数据敏感控件，不但可以改善用户界面的效果（因为输出不会不断地改变），而且会大大加快程序运行的速度。实际上，更新用户界面所花费的时间远远多于执行计算需要的时间。作为测试，读者可以注释掉Total范例中的DisableControls与EnableControls方法，看一看速度差异。

最后，在读取表格数据时还可能面临一些出错的危险，特别是在程序通过网络访问一台服务器并从中读取数据时。当检索数据时，如果有问题发生，则引起一个异常，控件会保持关闭状态，而且程序不能恢复正常状态。所以我们应该使用一个try/finally块。实际上，如果想使程序百分之百地防止错误，应该使用两个嵌套的try/finally块。包括该变化以及上面讨论的两个变化，下面是最终的代码：

```
procedure TSearchForm.ActionTotalExecute(Sender: TObject);
var
    Bookmark: TBookmarkStr;
    Total: Double;
begin
    Bookmark := Cds.Bookmark;
    try
        cds.DisableControls;
        Total := 0;
        try
            cds.First;
            while not cds.EOF do
            begin
                Total := Total + cdsSalary.Value;
                cds.Next;
            end;
        finally
            cds.EnableControls;
        end;
    end;
```

```

finally
    cds.Bookmark := Bookmark;
end;
MessageDlg ( Sum of new salaries is ' +
    Format ('%m', [Total]), mtInformation, [mbOK], 0);
end;

```

说明：我们编写这段代码是为了向读者显示如何循环浏览数据表格的内容，但要注意，还有一种基于SQL查询的方法可以返回字段值的和。当使用SQL服务器时，SQL调用会在计算总和方面显示出很大的速度优势，因为我们不需要将每个字段的所有数据从服务器移到客户端。服务器只是向客户端发送最终结果。当使用ClientDataSet的时候，还有一个更好的办法，因为对一系列数据的汇总是汇聚提供的一个特性，我们将在本章结尾部分进行讨论。这里我们只是介绍一个通用的解决方案，它可以用于任何数据集。

编辑数据表格列

增加动作的代码与我们刚看到的代码相似。ActionIncreaseExcute方法也用于扫描数据表格，计算薪金的总额，就像前面的方法一样。尽管只多了两行语句，但有一个主要的区别。当增加薪水时，实际改变的是数据表格中的数据。这两行关键语句包含在while循环中：

```

while not cds.EOF do
begin
    cds.Edit;
    cdsSalary.Value := Round (cdsSalary.Value * SpinEdit1.Value) / 100;
    Total := Total + cdsSalary.Value;
    cds.Next;
end;

```

第一行语句将数据表格设置为编辑模式，以便对字段的改变立即产生影响。第二行语句用于计算新的薪水，计算过程是，先用旧薪水值乘以SpinEdit组件的值（默认值是105），再除100。这是一个百分之五的增长率，尽管此值将被四舍五入。使用该程序可以对薪金进行任何幅度的改动，而操作只需单击一个按钮。

警告：注意，每当执行while循环时，数据表格进入编辑模式。这是因为在一个数据集中，编辑操作一次只能针对一个记录。我们必须调用Post或移到另一个记录上来完成编辑操作，如上面的代码所示。这时，如果我们想改变另一个记录，必须再次进入编辑模式。

自定义数据库网格

与其他大多数数据敏感控件（它们一般不具有为数众多的属性）不同，DBGrid控件有很多选项，并且功能非常强。下面几节将介绍使用DBGrid控件可以执行的高级操作。第一个范例解释如何在网格中绘图，第二个范例解释如何使用网格的多重选择特性。

绘制DBGrid

想要定制网格输出的原因很多。首先，好的范例可以突出特殊字段或记录。另外，还可以为通常不在网格中显示的字段，如BLOB、图形以及备注字段提供输出。

为了完全定制DBGrid控件的绘图，我们必须将其DefaultDrawing属性设置为False，并处理其OnDrawColumnCell事件。事实上，如果我们将DefaultDrawing的值设置为True，网格将在调用该方法之前显示默认的输出。这样，所需要做的就是向网格的默认输出添加一些内容，除非决定重新绘制它，但这会花费额外的时间并会造成闪烁现象。

另一种方法是调用网格的DefaultDrawColumnCell方法，可能是在改变当前字体或限定输入矩形之后。在上一种情况中，我们可以在一个单元中提供额外的绘图，让网格使用标准输出填充剩余的区域。这就是我们在DrawData范例中要做的事情。

该范例中的DBGrid控件（与Borland经典的Biolife数据表格相连）有以下属性：

```
object DBGrid1: TDBGrid
  Align = alClient
  DataSource = DataSource1
  DefaultDrawing = False
  OnDrawColumnCell = DBGrid1DrawColumnCell
end
```

OnDrawColumnCell事件处理程序对网格的每个单元都需要调用一次并且有多个参数，包括对应着单元的矩形、我们必须绘制的列的索引、列本身（带有字段、其对齐方式与其他子属性）以及单元的状态，那么怎样将指定单元的颜色变成红色呢？只需在特殊情况中改变它即可：

```
procedure TForm1.DBGrid1DrawColumnCell(Sender: TObject;
  const Rect: TRect; DataCol: Integer; Column: TColumn;
  State: TGridDrawState);
begin
  // red font color if length > 100
  if (Column.Field = cdsLengthcm) and (cdsLengthcm.AsInteger > 100) then
    DBGrid1.Canvas.Font.Color := clRed;

  // default drawing
  DBGrid1.DefaultDrawDataCell (Rect, Column.Field, State);
end;
```

下一步是绘出这个备注和图像字段。对于这个备注，我们可以实现这个备注字段的OnGetText和OnSetText事件。如果它的OnSetText事件不是nil，网格将允许编辑一个备注字段。下面的代码是这两个事件的处理程序。我们使用Trim来去掉文本尾部不打印的字符，以使文本在编辑时更加清晰：

```
procedure TForm1.cdsNotesGetText(Sender: TField;
  var Text: String; DisplayText: Boolean);
begin
  Text := Trim (Sender.AsString);
end;

procedure TForm1.cdsNotesSetText(Sender: TField; const Text: String);
begin
  Sender.AsString := Text;
end;
```

对于这个图像，最简单的办法是创建一个临时的TBitmap对象，将这个图形字段赋给它，并且将图像绘制在网格的Canvas里。还有一种办法，就是把图形字段的Visible属性设置为False而将其从网格中删除，并且将图像添加的鱼的名称上，使用下面的OnDrawColumnCell事件处理程序代码：

```
var
  Picture: TPicture;
  OutRect: TRect;
  PictWidth: Integer;
begin
  // default output rectangle
  OutRect := Rect;

  if Column.Field = cdsCommon_Name then
  begin
    // draw the image
    Picture := TPicture.Create;
    try
      Picture.Assign(cdsGraphic);
      PictWidth := (Rect.Bottom - Rect.Top) * 2;
      OutRect.Right := Rect.Left + PictWidth;
      DBGrid1.Canvas.StretchDraw (OutRect, Picture.Graphic);
    finally
      Picture.Free;
    end;
    // reset output rectangle, leaving space for the graphic
    OutRect := Rect;
    OutRect.Left := OutRect.Left + PictWidth;
  end;

  // red font color if length > 100 (omitted see above)

  // default drawing
  DBGrid1.DefaultDrawDataCell (OutRect, Column.Field, State);
```

正如我们在代码中看到的，程序在一个网格左边单元的小矩形中显示了这个图像，然后在激活默认绘图之前更改了矩形的输出。我们可以在图13.12中看到输出的效果。

Species No.	Category	Common Name	Species Name	Length (cm)	Length (in)
90829	Triggerfish	Clown Triggerfish	Ballistoides conspicillum	50	19.685
90830	Triggerfish	Red Emperor	Lutjanus sebae	50	23.622
90850	Wrasse	Giant Moon Wrasse	Cheilinus undulatus	229	90.157
90878	Angelfish	Blue Angelfish	Pomacanthus nasurus	38	11.811
90880	Cod	Lunartail Rockcod	Varicorhinus leuiscus	80	31.49
90899	Scorpionfish	Firefish	Pterois volitans	38	14.968
90108	Butterflyfish	Ornate Butterflyfish	Chaetodon ornatissimus	19	7.4803
90118	Shark	Swirl Shark	Cephaloscyllium ventriosum	182	40.157
90129	Ray	Bat Ray	Myliobatis californica	56	22.047
90138	Eel	California Moray	Gymnothorax mordax	158	58.955
90148	Cod	Lingcod	Ophiodon elongatus	158	58.955

图13.12 DrawData程序显示了一个网格，其中包括一个备注字段的文本内容以及各种Borland鱼

允许多重选择的网格

第二个，也是最后一个定制DBGrid控件的范例与多重选择有关。我们可以设立DBGrid，使用户可以选择多行（也就是说，多个记录）。这是非常容易的，因为我们所需做的只是切换网格Options属性的dgMultiSelect元素。一旦选择了该选项，就可以按住Ctrl键并用鼠标单击，选择网格的多行，效果如图13.13所示。



图13.13 MltGrid范例具有一个DBGrid控件，允许选择多个行

因为数据库表格只能有一个活动的记录，那么选项在网格中存储了什么信息呢？网格只为所选项保存了一个书签列表，该列表可以在SelectedRows属性中找到，它是TBookmarkList类型的。除了使用Count属性访问列表中的对象个数外，我们还可以获取每个带有Items属性（默认的数组属性）的书签。列表的每一项都是TBookmarkStr类型的，它代表了一个书签指针，我们可以将它指派给数据表格的Bookmark属性。

说明：TBookmarkStr是一种为便于使用而引入的字符串类型，但它的数据应该被看做是“不透明的”与易变的。对于我们发现的数据，如果是书签的值，则不应该依赖任何特殊的结构，而且不应该使用太长的数据，或将它存储到一个独立的文件中。书签数据将随着数据库驱动程序与索引配置而变化，当在数据集添加新行或删除某行时，它将会变得无用。

为了总结一下以上步骤，下面列出MltGrid范例的代码，该程序可以通过单击按钮激活，从而将所选记录的Name字段移到列表框中：

```
procedure TForm1.Button1Click(Sender: TObject);
var
  I: Integer;
  BookmarkList: TBookmarkList;
  Bookmark: TBookmarkStr;
begin
  // store the current position
  Bookmark := cds.Bookmark;
  try
    // empty the list box
    ListBox1.Items.Clear;
```



```

// get the selected rows of the grid
BookmarkList := DbGrid1.SelectedRows;
for I := 0 to BookmarkList.Count - 1 do
begin
    // for each, move the table to that record
    cds.Bookmark := BookmarkList[I];
    // add the name field to the listbox
    ListBox1.Items.Add (cds.FieldByName ('Name').AsString);
end;
finally
    // go back to the initial record
    cds.Bookmark := Bookmark;
end;
end;

```

向网格拖动记录

另一个有趣的技术是使用网格进行拖动。从一个网格中拖出记录并不特别困难，因为我们知道哪一个当前记录，以及用户选择的列。向一个网格拖动记录，显然对于编程是困难的。我们在第2章中提到过“盗用受保护的数据”，这就是我们打算用于实现向一个网格拖动记录的技术。

名为DragToGrid的范例有一个与country数据集相连的网格，一个可以为字段输入新值的编辑框，和一个拖到网格单元上修改有关字段的标签。真正的问题是如何确定该字段。代码只有不多的几行，如下所示，但确实很难理解，因此我们将针对代码进行一些介绍：

```

type
    TDBGHack = class (TDbGrid)
    end;

procedure TFormDrag.DBGrid1DragDrop(Sender, Source: TObject; X, Y: Integer);
var
    gc: TGridCoord;
begin
    gc := TDBGHack (DbGrid1).MouseCoord (x, y);
    if (gc.y > 0) and (gc.x > 0) then
    begin
        DbGrid1.DataSource.DataSet.MoveBy (gc.y - TDBGHack(DbGrid1).Row);
        DbGrid1.DataSource.DataSet.Edit;
        DbGrid1.Columns.Items [gc.X - 1].Field.AsString := EditDrag.Text;
    end;
    DbGrid1.SetFocus;
end;

```

第一个操作决定了鼠标在哪个单元上被释放。从鼠标的X和Y坐标开始，我们可以调用受保护的MouseCoord方法，以访问这个单元对应的列与行。除非这个拖曳的目标为第一行（通常是标题行）或者第一列（通常是指示符列），否则程序会利用所请求行（gc.y）与

当前活动行（网格受保护的Row属性）之间的差移到当前行上。下一步是将数据集置于编辑模式，读取目标列的字段（Columns.Items[gc.X-1].Field），并改变它的文本。

带有标准控件的数据库应用程序

尽管基于数据敏感控件编写Delphi应用程序通常比较快，但并不一定总需要这样。当我们在数据库应用程序的用户界面上需要非常精确的控件时，可以定制数据从字段对象向可视控件的传送。笔者个人认为，这只有在非常特殊的情况时才需要，因为我们可以通过设置属性与处理字段对象的事件来定制数据敏感控件。然而，不使用数据敏感控件有助于读者理解Delphi的默认行为，它也会帮助我们介绍一些与数据库有关的事件。

如果应用程序的开发不基于数据敏感控件，可以使用两种不同的方法：在代码中模拟标准的Delphi行为（特殊情况中可以把它分出去），或使用经过更多定制的方法。我们将在NonAware范例中演示第一种技术，在SendToDb范例中演示第二种技术。

模拟Delphi的数据敏感控件

如果我们想建立一个不使用数据敏感控件但工作方式却与标准Delphi应用程序相似的应用程序，可以由数据敏感控件自动执行的操作编写事件处理程序。基本上，当用户改变可视控件的内容时，需要将数据集放置在编辑模式中；当用户从控件中退出时，需要更新数据集的字段对象，并将焦点移到另一个元素中。

提示：该方法对于将非数据敏感控件集成到标准应用程序中是很方便的。

NonAware范例中还有一个按钮列表，对应着DBNavigator控件中的一些按钮并连接到五个自定义操作上。我们对该范例不能使用标准数据集操作，因为它们会自动连接与当前控件相关的数据源，该机制将会失败，原因是使用了非数据敏感的编辑框。通常情况下，我们还可以将一个数据源与每一个操作的DataSource属性相接。但是在这个特定的情况下，我们并没有在范例中使用数据源。

该程序有一些事件处理程序，我们还没有将它们用于过去使用数据敏感控件的应用程序中。首先，我们必须通过处理数据集组件的OnAfterScroll事件，在可视控件中显示当前记录的数据（如图13.14所示）：

```
procedure TForm1.cdsAfterScroll(DataSet: TDataSet);
begin
    EditName.Text := cdsName.AsString;
    EditCapital.Text := cdsCapital.AsString;
    ComboContinent.Text := cdsContinent.AsString;
    EditArea.Text := cdsArea.AsString;
    EditPopulation.Text := cdsPopulation.AsString;
end;
```

这个控件的OnStateChange事件处理器在一个状态栏控件中显示表格的状态。当用户开始在编辑框中输入或者下拉组合框列表的时候，应用程序都将这个表格设定为编辑模式：

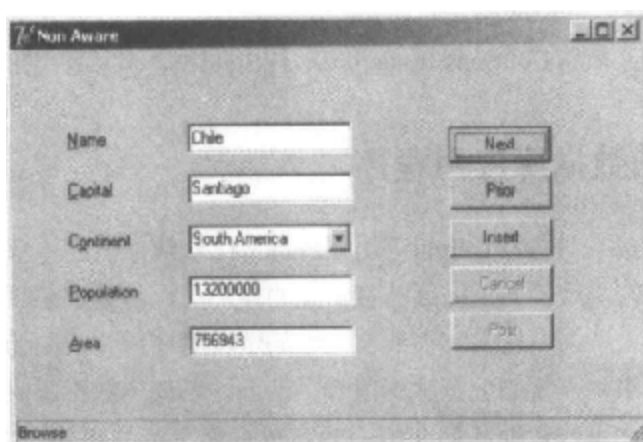


图13.14 使用Browse模式时NonAware范例的输出。每一次记录发生变化时，该程序要靠人工取回数据

```

procedure TForm1.EditKeyPress(Sender: TObject; var Key: Char);
begin
    if not (cds.State in [dsEdit, dsInsert]) then
        cds.Edit;
    end;

```

该对象方法与五个组件的OnKeyPress事件相连，而且与组合框的OnDropDown事件处理程序相似。当用户离开一个可视控件时，OnExit事件的处理程序会将数据复制到相应的字段中，如下所示：

```

procedure TForm1.EditCapitalExit(Sender: TObject);
begin
    if (cds.State in [dsEdit, dsInsert]) then
        cdsCapital.AsString := EditCapital.Text;
    end;

```

只有当数据表格处于编辑模式时才会发生该操作，也就是说，只有当用户在该控件或另一个控件中输入文本时，才会出现这种情况。这真的不能令人满意，因为即使编辑框中的文本没有改变，额外的操作也会执行，但执行的速度非常快，甚至可能用户注意不到。对于第一个编辑框，我们在复制之前检查了文本；如果编辑框是空的，则引起一个异常：

```

procedure TForm1.EditNameExit(Sender: TObject);
begin
    if (cds.State in [dsEdit, dsInsert]) then
        if EditName.Text <> '' then
            cdsName.AsString := EditName.Text
        else
            begin
                EditName.SetFocus;
                raise Exception.Create ('Undefined Country');
            end;
    end;
end;

```

另外一个测试字段值的办法是处理数据集的BeforePost事件。请记住，在这个范例中，post操作不是被一个特定按钮处理的，而是当用户移动到或者添加一个新记录的时候自动发生的：

```
procedure TForm1.cdsBeforePost(DataSet: TDataSet);
begin
    if cdsArea.Value < 100 then
        raise Exception.Create ('Area too small');
end;
```

在这些情况中，还有一种引起异常的方法，即设置一个默认值。然而，如果字段具有默认值，那么最好向前设立它，这样用户可以看到将被发往数据库的值。为了实现该方法，可以处理数据集的AfterInsert事件，并且在建立新记录之后便立刻触发它（也可以使用OnNewRecord事件）：

```
procedure TForm1.cdsAfterInsert(DataSet: TDataSet);
begin
    cdsContinent.Value := 'Asia';
end;
```

向数据库发送请求

如果决定在处理时不按照标准Delphi数据敏感控件相同的顺序进行编辑操作，那么可以进一步定制应用程序的用户界面。这给了我们极大的自由，尽管可能出现一些副作用（如处理并发的限制，我们将在第14章讨论该话题）。

在新范例中，我们用另一个组合框替换了第一个编辑框，用两个定制按钮替换了所有与数据表格操作相关的按钮（对应着DBNavigator按钮），用于从数据库获得数据并向数据库发送更新的数据。为了强调该范例的不同之处，我们甚至删除了DataSource组件。

与相应按钮相连的GetData方法只获取一个字段，该字段对应着在第一个组合框中指定的记录：

```
procedure TForm1.GetData;
begin
    cds.Locate ('Name', ComboName.Text, [loCaseInsensitive]);
    ComboName.Text := cdsName.AsString;
    EditCapital.Text := cdsCapital.AsString;
    ComboContinent.Text := cdsContinent.AsString;
    EditArea.Text := cdsArea.AsString;
    EditPopulation.Text := cdsPopulation.AsString;
end;
```

无论何时用户单击该按钮，在组合框中选择一个表项，或者在组合框中按Enter键，这个方法都会被调用：

```
procedure TForm1.ComboNameClick(Sender: TObject);
begin
    GetData;
end;
```

```

procedure TForm1.ComboNameKeyPress(Sender: TObject; var Key: Char);
begin
    if Key = #13 then
        GetData;
end;

```

为了使这个范例工作得更加平滑,在开始的时候,我们使用表格中的国家名称对组合框进行了填充:

```

procedure TForm1.FormCreate(Sender: TObject);
begin
    // fill the list of names
    cds.Open;
    while not cds.Eof do
    begin
        ComboName.Items.Add (cdsName.AsString);
        cds.Next;
    end;
end;

```

通过这个方法,组合框成为了一种记录的选择器,正如我们在图13.15中看到的那样。由于采用了这种方法,我们就可在程序中不使用定位按钮了。

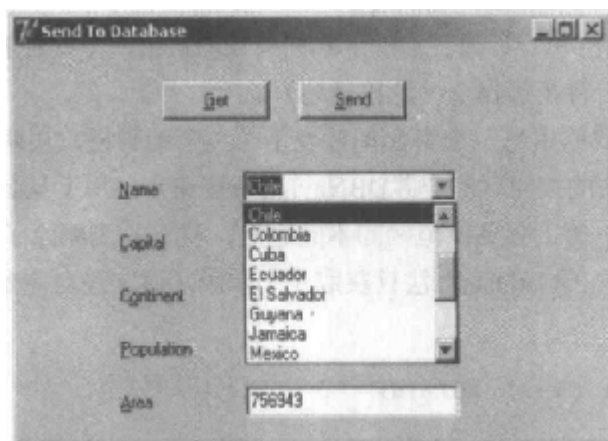


图13.15 在SendToDb范例中,使用一个组合框来选择希望察看记录

用户还可以更改控件的值并且单击Send按钮。根据操作类型是更新或者发送,相应地执行不同的代码。可以根据名称做出决定,虽然在这段代码中,一个错误的名字不能够再被改变:

```

procedure TForm1.SendData;
begin
    // raise an exception if there is no name
    if ComboName.Text = '' then
        raise Exception.Create ('Insert the name');
    // check if the record is already in the table
    if cds.Locate ('Name', ComboName.Text, [loCaseInsensitive]) then

```

```

begin
    // modify found record
    cds.Edit;
    cdsCapital.AsString := EditCapital.Text;
    cdsContinent.AsString := ComboContinent.Text;
    cdsArea.AsString := EditArea.Text;
    cdsPopulation.AsString := EditPopulation.Text;
    cds.Post;
end
else
begin
    // insert new record
    cds.InsertRecord ([ComboName.Text, EditCapital.Text,
        ComboContinent.Text, EditArea.Text, EditPopulation.Text]);
    // add to list
    ComboName.Items.Add (ComboName.Text)
end;

```

在向数据表格发送数据之前，我们可以对数值进行任何形式的确认测试。在这种情况下，处理数据库组件的事件是没有意义的，因为当执行更新或插入操作时，我们有功能完备的控件可以使用。

分组与合计

我们已经看到，**ClientDataSet**能够拥有与存储在文件中的顺序不同的索引。一旦我们定义了一个索引，就可以使用它进行数据分组。在实际中，一个分组定义为一系列递归的记录（根据索引），对应的索引字段的值不会改变。例如，如果我们使用州的名称作为索引，这个州里所有的地址都可以归到这个分组中。

分组

CdsCalcs示例有**ClientDataSet**组件，该组件从我们熟悉的**Country**数据集中抽取它的数据通过为这个索引指定一个分组的级别，即可获得组以及一个索引（**Index**）的定义：

```

object ClientDataSet1: TClientDataSet
    IndexDefs = <
        item
            Name = 'ClientDataSet1Index1'
            Fields = 'Continent'
            GroupingLevel = 1
        end>
    IndexName = 'ClientDataSet1Index1'

```

当一个组处于激活状态时，通过在**DBGrid**中显示这个分组的结构，我们可以使用户明确地认识它，如图13.16所示。所有我们需要做的是处理分组字段（范例中的**Continent**字段）的**OnGetText**事件，并且显示第一个分组中的记录的文本：

```

procedure TForm1.ClientDataSet1ContinentGetText(Sender: TField;
  var Text: String; DisplayText: Boolean);
begin
  if gbFirst in ClientDataSet1.GetGroupState (1) then
    Text := Sender.AsString
  else
    Text := '';
end;

```

Continent	Name	Capital	Area	Population
North America	Mexico	Mexico City	1,967,190	88,600,000
	Nicaragua	Managua	139,000	3,900,000
	El Salvador	San Salvador	20,865	5,300,000
	Cuba	Havana	114,524	10,600,000
	Jamaica	Kingston	11,424	2,500,000
	United States of America	Washington	9,363,130	249,200,000
	Canada	Ottawa	9,976,147	26,500,000
South America	Paraguay	Asuncion	406,576	4,660,000
	Uruguay	Montevideo	176,140	3,002,000
	Venezuela	Caracas	912,047	19,700,000
	Peru	Lima	1,285,215	21,600,000
	Argentina	Buenos Aires	2,777,815	32,300,000
	Guyana	Georgetown	214,569	800,000
	Ecuador	Quito	455,902	10,600,000
	Colombia	Bogota	1,138,307	33,000,000
	Chile	Santiago	756,543	13,200,000
	Brazil	Brasilia	8,511,196	150,400,000

TotalArea aggregate: 17,733,885 Get Aggregates Area: 17,733,885 Population: 663,162,003 Number: 11

图13.16 CdsCalcs范例通过一小段代码进行了演示，我们可以用DBGrid控件显示ClientDataSet定义的分组

定义合计

ClientDataSet组件另一个功能强大的特性是支持合计。合计是一个基于多个记录的计算值，如整个数据表格或一组记录（使用我们刚才讨论过的分组逻辑来定义）中某个字段的和值或平均值。合计是可维护的，也就是说，如果有一个记录发生改变，会立刻重新计算合计值。例如，当用户在发货清单条目中输入时，发货单的总和会自动被重新计算出来。

说明：合计是维持递增的，而不是每当有一个值改动时就重新计算所有的值。合计的更新利用了ClientDataSet追踪的增量。例如，当字段发生改变时，为了更新和值，ClientDataSet会从合计中读取旧值，并加上新值。这只需要两次计算，即使在该合计组中有上千行。因此，合计更新是瞬时的。

有两种方法定义合计：使用ClientDataSet（是一个集合）的Aggregates属性，或使用Fields编辑器定义合计字段。在这两种情况下，需要定义合计表达式，赋给它一个名称，并将它与一个索引和一个分組级别（除非想将它应用于整个数据表格）连接。下面是CdsCalcs范例的Aggregates集合：

```

object ClientDataSet1: TClientDataSet
  Aggregates = <
    item

```

```

    Active = True
    AggregateName = 'Count'
    Expression = 'COUNT (NAME)'
    GroupingLevel = 1
    IndexName = 'ClientDataSet1Index1'
    Visible = False
end
item
    Active = True
    AggregateName = 'TotalPopulation'
    Expression = 'SUM (POPULATION)'
    Visible = False
end>
AggregatesActive = True

```

注意，在上面最后一行代码中，除了激活每个想使用的特定合计之外，我们还必须为合计激活支持。解除合计是重要的，因为合计太多会减慢程序的执行速度。

我们曾提到的另一种方法是使用Fields编辑器，在其快捷菜单中选择New Field命令，并选择Aggregate选项（只有在一个ClientDataSet中可以与InternalCalc选项一起使用）。下面是一个合计字段的定义：

```

object ClientDataSet1: TClientDataSet
  object ClientDataSet1TotalArea: TAggregateField
    FieldName = 'TotalArea'
    ReadOnly = True
    Visible = True
    Active = True
    DisplayFormat = '###,###,###'
    Expression = 'SUM(AREA)'
    GroupingLevel = 1
    IndexName = 'ClientDataSet1Index1'
  end
end

```

合计字段在Fields编辑器中被显示为独立的一组，如图13.17所示。与普通合计相比，使用合计字段的优点是，我们可以定义显示格式，并将字段直接与数据敏感控件相连，如CdsCalcs范例中的DBEdit。因为合计与一个组相连，所以只要选择了另一组的记录，输出就会被自动更新。而且，如果改变数据，合计值也会立刻显示新值。

为了使用普通合计，我们必须编写一小段代码，如下面的例子所示，请注意这个合计中的Value是一个变量：

```

procedure TForm1.Button1Click(Sender: TObject);
begin
  Label1.Caption :=
    'Area: ' + ClientDataSet1TotalArea.DisplayText + #13'Population: '
    + FormatFloat ('###,###,###', ClientDataSet1.Aggregates [1].Value) +
    #13'Number: ' + IntToStr (ClientDataSet1.Aggregates [0].Value);
end;

```




图13.17 ClientDataSet字段编辑器的底部显示合计字段

主/详结构

通常，我们需要将数据表格联系起来，这些表格具有一种一对多的关系。这意味着，对于主表格的某个记录来说，在二级表格中有很多从记录。典型范例是一张发票与发票项目间的关系；另一个例子是顾客列表与每个顾客的定单。

这在数据库编程中是经常遇到的情况，而且Delphi通过主/详结构显式支持它。TDataSet类拥有一个通用的DataSource属性，可以建立主数据源。这个属性被用在一个与当前的主数据集记录连接的详细数据集之中，同时它结合了MasterFields属性。

主/详结构与ClientDataSets

MastDet范例使用了客户和订单的范例数据集。我们为每一个数据集添加了一个数据源组件，对于二级数据集，我们为连接到第一个数据集的数据源附加了DataSource属性。最后，我们使用MasterFields属性的特殊编辑器，将二级表与主表中的一个字段相关联。为此，我们使用了一个数据模块来完成上述动作，就像前面的“数据访问组件的数据模块”一节中讨论的那样。

下面是一个数据模块的完整列表（但是不包括无关的位置属性），它被用于MastDet程序中：

```
object DataModule1: TDataModule1
  OnCreate = DataModule1Create
  object dsCust: TDataSource
    DataSet = cdsCustomers
  end
  object dsOrd: TDataSource
    DataSet = cdsOrders
  end
  object cdsOrders: TClientDataSet
    FileName = 'orders.cds'
    IndexFieldNames = 'CustNo'
    MasterFields = 'CustNo'
    MasterSource = dsCust
  end
end
```

```

object cdsCustomers: TClientDataSet
  FileName = 'customer.cds'
end
end
end

```

从图13.18中, 我们可以看到MastDet程序的主窗体的运行时范例。我们在窗体的上半部分将数据敏感控件与主表格进行关联, 另外我们在窗体的下半部分放置了一个网格, 连接到详细表格中。通过这个方式, 对于每个主记录来说, 我们可以立即看到相连的详细记录, 在这种情况下, 所有订单都是由当前客户提交的。每当我们选择一个新的客户, 窗体下面的网格中将会只显示与这个客户相关的订单信息。

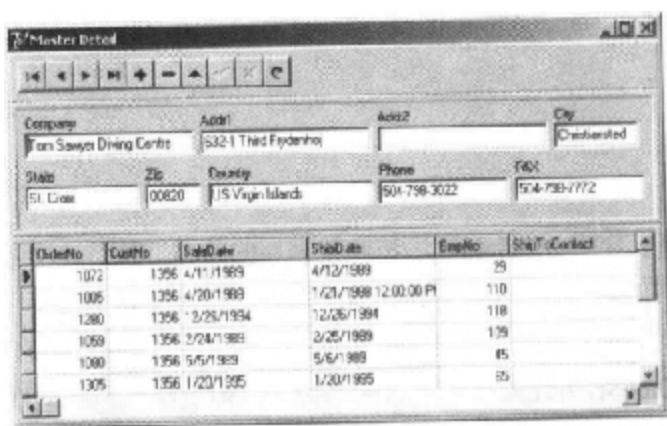


图13.18 运行时的MastDet范例

处理数据库错误

数据库编程的另一个特性是用定制方式处理数据库错误。当然, 每当出现数据库错误时, 可以让Delphi显示一条异常消息, 但我们还想改正这些错误, 或获得更详细的信息。基本上有三种方法用于处理数据库相关的错误:

- 将有危险的数据库操作放在一个try/except块中, 如对Query的Open方法的调用, 或是对数据集的Post方法的调用。但是, 若操作是由于使用数据敏感控件而生成的, 这便不可能。
- 为全局Application对象的OnException事件安装一个处理程序。
- 处理与错误相关的数据集的特殊事件, 如OnPostError、OnEditError、OnDeleteError与OnUpdateError。

在Delphi中的大部分异常类只会简单地发送一条错误消息; 而使用数据库异常, 可以看到一个错误列表, 显示了错误代码, 以及连接的SQL服务器本身的错误代码和消息。ClientDataSet只在它的异常类中加入了一个错误代码, EDBClient。我们接下来将介绍如何处理它, 这也为大家提供了一个处理其他类似情况的指导方法。

笔者使用该信息建立了一个简单的数据库程序, 在Memo组件中显示错误的细节(当用户单击程序按钮时, 错误会自动产生)。为了处理所有的错误, DBError范例为全局对象Application的OnException事件安装了一个处理程序。如果这个数据库是一个EDBClient, 那么该处理程序会将一些日志信息记录在一个备注(Memo)中, 其中包含这个数据库的详细

错误信息:

```
procedure TForm1.ApplicationError (Sender: TObject; E: Exception);
begin
  if E is EDBClient then
  begin
    Memol.Lines.Add('Error: ' + (E.Message));
    Memol.Lines.Add('  Error Code: ' +
      IntToStr(EDBClient (E).ErrorCode));
  end
  else
    Memol.Lines.Add('Generic Error: ' + (E.Message));
end;
```

小结

在本章中，我们介绍了Delphi程序访问数据库的范例，还讨论了基本的数据敏感组件以及基于标准控件的数据库应用程序开发。我们研究了TDataSet类以及字段对象的内部结构，并讨论了很多由所有数据集共享以及供所有数据库应用程序使用的事件与属性。即使大多数范例均使用ClientDataSet组件进行本地数据的访问，但该组件也常常作为（在客户机/服务器体系结构中）一台SQL服务器或者（在三层体系结构中）一个远程应用程序中的数据通道。我们还讨论了计算字段、查寻字段、DBGrid控件的定制，以及很多更高级的技术。

我们没有深入探讨数据库与数据访问，因为它们要根据我们实际使用的数据库引擎与服务器的类型来定。下一章将开始讨论这一专题，届时会详细讨论如何使用Borland提供的dbExpress库进行客户机/服务器的开发。我们还将介绍InterBase以及IBX组件，并给出一些实际应用中的元素。

接下来的章节将会继续介绍关于数据库部分的Delphi编程，我们将讨论ADO连接性与组件、Borland的三层体系结构（数据快照——DataSnap，即过去的MIDAS）、数据敏感控件的开发、自定义数据集组件以及报表技术。

第14章 使用dbExpress的客户机/服务器编程

在前面的章节中，我们介绍了Delphi对数据库编程的支持，并在大多数范例中使用了本地文件（特别是使用ClientDataSet组件或者MyBase）而没有涉及任何专用的数据库技术。本章将会介绍SQL服务器数据库的使用，重点讨论使用BDE和新的dbExpress技术对客户机/服务器进行开发。仅仅一章的篇幅不足以详细讨论如此复杂的问题，所以我们只是从Delphi开发人员的角度讨论有关技术，并介绍一些技巧与提示。

我们要在例子中使用InterBase，因为这是Borland的关系数据库管理系统（RDBMS，Relational DataBase Management System），或者说是SQL服务器，已经包含在Delphi的企业级版本中，并且是一个免费并开放源代码的服务器（虽然不是在其所有的版本中都包括，我们下面会做详细的介绍）。我们将从Delphi自身的角度出发来讨论InterBase，而不过多地深入它的内部体系结构。因为我们所介绍的内容中，有很多可以用于其他的SQL服务器，所以即使今后不打算使用InterBase，大家也会发现这些内容对自己很有帮助。

本章主要包括以下内容：

- 客户机/服务器简介
- 数据库设计的元素
- InterBase简介
- 服务器端编程：视图、存储过程与触发器
- dbExpress库
- 使用ClientDataSet组件进行缓存
- InterBase Express (IBX) 组件

客户机/服务器的体系结构

上一章中介绍的数据库应用程序使用了原始的组件来访问存储在本地计算机上的文件中的数据，并且将全部文件装入内存。这是一个极端的解决办法。更传统的办法是逐个记录地读取文件，这样多个应用程序可以同时访问它，但前提是使用了写同步机制。

当数据位于一个远程服务器的时候，将整个表格放到内存中进行处理无论对于时间还是网络带宽的消耗都是很大的，而且通常是一种浪费。

作为一个范例，我们可以考虑使用与EMPLOYEE一样的数据表格（Delphi携带的InterBase范例数据库的一部分），向它添加上千个记录，并将它放置在一台作为文件服务器的网络计算机上。如果我们想了解公司职员的最大薪金，可以打开Table组件（EmpTable），连接数据库表格并运行下列代码：

```
EmpTable.Open;  
EmpTable.First;  
MaxSalary := 0;
```

```
while not EmpTable.Eof do
begin
  if EmpTable.FieldName ('Salary').AsCurrency > MaxSalary then
    MaxSalary := EmpTable.FieldName ('Salary').AsCurrency;
  EmpTable.Next;
end;
```

该方法的作用是将（很大的）数据表格中的所有数据从网络计算机中移到本地机器上，该操作可能会花费几分钟的时间。在这种情况下，一个比较好的办法是让SQL服务器直接计算出我们需要的结果，然后我们使用下列SQL代码只取回这个结果信息：

```
select Max(Salary) from Employee
```

说明：上面的两段代码都属于GetMax范例，它包括了一些代码为两种方法计时。在小EMPLOYEE数据表格上使用Table组件花费的时间大约是使用查询的十倍，即使InterBase服务器安装在运行这个程序的客户机上也是如此。

如果我们想将大量数据存储在中央计算机上，并避免为了要进行处理而向客户端移动数据，则唯一的解决方法是让中央计算机处理数据并向客户机发回有限的信息。这是客户机/服务器编程的基础。

通常，我们会在服务器（RDBMS）上使用一个已有的程序并编写与之相连的定制客户应用程序。然而，有时可能甚至想同时编写定制的客户与定制的服务器，如在三级应用程序中那样。Delphi支持这种类型的应用程序——即分布式中间层应用服务（MIDAS）结构，现在已经被称为DataSnap。我们将在第16章“多层DataSnap应用程序”中进行介绍。

将数据从本地文件向SQL服务器数据库引擎移动通常是出于性能方面的考虑，同时也是为了允许移动大量数据。让我们回到前一个范例，在客户机/服务器环境中，用于选择最高薪金的查询将由RDBMS进行计算，它将只向客户发回最终结果，仅仅是一个数字而已。使用功能较强的服务器（如多处理器的Sun工作站），计算结果所需的时间可以减少到最小。

然而，选择客户机/服务器结构还有其他一些原因：

- 帮助我们管理大量的数据，因为我们不希望将数百兆的内容存储在一个本地文件中。
- 支持多个用户同时数据访问。SQL服务器数据库通常使用优化锁定（optimistic locking）技术，这是允许多个用户在同一个人数据上共同工作并且延迟一致性控制直到用户发回更新信息的方法。
- 提供数据的完整性、事务控制、安全、访问控制、备份支持以及其他功能。
- 支持可编程性：在服务器上运行部分代码（存储过程、触发器、表格视图和其他技术）的能力，因此减少了网络的流量以及客户端计算机的工作量。

现在，我们可以讨论用于客户机/服务器编程的特殊技术了。注意，该技术基本的目的是在客户端与服务器之间适当地分配工作量，并减小传递信息所需的网络带宽。

该方法的基础是好的数据库设计，包括数据库结构与相应的数据确认与限制，或者商业规则。在服务器上执行数据确认是重要的，因为数据库的完整是任何程序的主要目的之一。然而，客户端也应该包括数据确认，以改善用户界面并使数据的输入与处理对用户更友好。当我们可以防止错误输入时，让用户输入无效数据并从服务器中接收到一条错误消息是没有意义的。

数据库设计的元素

尽管本书是介绍Delphi程序而不是介绍数据库的,但我觉得有必要介绍几个好的(以及现代的)数据库设计的元素。原因很简单:如果我们的数据库设计不正确或令人费解,我们就可能必须写出很复杂的SQL语句和服务端代码,或者需要写出大量的Delphi代码来访问我们的数据,甚至与TDataSet类的设计相抵触。

实体与关系

经典的关系数据库设计方法基于实体-关系(E-R)模式,要求我们为需要在数据库上展示每个实体都创建一个表格,为每一个数据元素字段加上另一个字段,该字段用于反映到另一个实体(或表格)的一对一或一对多关系。相反,对于多对多关系式,我们需要一个独立的表格。

假设一个表格用一对一关系展示某大学的课表。对于每个相关数据元素(名称、描述以及在哪个教室上课等等)它都有一个字段加上单独一个说明教师的字段。事实上,教师的数据不应被储存在课程数据内,而是在一个独立表格中,因为它有可能从别处被引用。

每门课程的时间表以不同方式包括一个每天都可能不相同的不明确的小时数,因此在描述课程的相同表格内它们不能被添加。相反,该信息必须被放置在一个独立的表格中,该表格中包含所有时间表,以及引用该时间表的字段。在一个一对多关系中,如这种情况,时间表中有很多记录都指向课程表中的同一条记录。

一种更复杂的情况是要求储存哪个学生上哪节课。在课表中,学生不能被直接显示,因为他们的号码不固定,并且班级不能由于该原因而储存在学生的数据中。在相似的多对多关系中,惟一的方法是有一个额外的表格显示关系,列出对学生和课程的引用。

标准化规则

经典的设计原则包括一系列所谓的标准化规则。这些规则的目标是避免在数据库中复制数据(不仅是为了节省空间,主要是为了避免加进不适合的数据)。例如,我们不必在每个定单中重复所有客户的详细信息,而可以引用一个独立的客户实体。这种方法可以节省内存,而且当客户的信息改变时(例如地址的改变),这个客户的所有定单都会反映出新的数据。与这个客户相关的其他表格也可能会自动更新。

标准化规则暗示为共同的重复值使用代码。例如,如果我们有几个不同的出货选择,则我们在定单表格中将不为这些选择使用基于描述的字符串,而是使用短数据代码,映射到单独的查询表格中的描述信息上。

使用前面介绍的规则时应该很谨慎,以避免为每个查询加入大量的表格。我们可以使用一些非标准化格式(在定单查询中使用短的出货说明)或客户程序来提供说明,但这样会导致错误的数据库设计。最后的选项只在我们使用单个开发环境(就是说Delphi)访问该数据库时才切实可行。

从主关键字到OID

在关系型数据库中，记录不是通过物理位置来确定的（如在Paradox和其他本地数据库中那样），而只被记录内的数据本身所确定。通常，我们不需要通过所有字段的数据来确定一个记录，而只需数据的一个子集形成的主关键字。如果字段是主关键字的一部分，并且必须确定一个单个记录，那么对于表格中的每个可能的记录来说，它们的值必须不同。

说明：技术上，许多数据库服务器为表格添加内部记录标识符，但这只用于内部优化和对关系数据库的本地设计进行处理。这些内部标识符在不同的SQL服务器上工作方式不同，并且在版本间有所改变，因此最好不要过分依赖它们。

关系理论的早期版本描述了逻辑键的使用，这意味着挑选一个或多个描述实体的记录没有任何风险。但说起来容易做起来难。例如，公司名字不是惟一的，甚至公司名和它的地址也不能够保证是惟一的。而且，如果一个公司改变它的名字（就如同Borland公司一样）或它的地址，并且我们在其他表格中引用了它，那么我们就必须也改变所有那些有风险的相关引用。

出于这个原因，也出于对效率的考虑（使用字符串进行引用意味着在常用来放置引用的二级表格中使用大量空间），逻辑键已经被物理键或代理键所淘汰：

物理键 一个单个字段，能以惟一的方式确定一个元素。例如，美国的每个公民都有一个社会保障号（SSN），但几乎每个国家都使用纳税ID或其他政府指定的编号来鉴别每个人。同样的例子在公司也存在。尽管这些ID码被保证是惟一的，但随着地区的不同它们也有所不同（当一个公司向国外销售货物的时候会产生问题），甚至在同一个地区内也会发生变更（由于新的税法）。它们有时效率低下，因为它们太长（例如，在意大利，使用16个代码、字母和数字来识别每位居民）。

代理键 它以客户代码定单编号等形式来识别每个记录。这些代理键通常用于数据库设计。然而，在许多情况，它们是一些逻辑标识符，带有客户端代码。

警告：当这些代理键带有某些含义并必须遵循指定原则时，情况就会变的很复杂。例如，公司发票号是惟一并连续的，在号码顺序上没有漏洞。这种情况用程序来处理十分复杂，特别是当我们发送断数据时只有数据库能决定这些惟一而又连续的的代码的时候。同时，在我们把数据发送到数据库之前，我们需要识别记录。否则，我们将无法再次获得该数据。怎样解决该情况的实际例子在第15章“使用ADO”中介绍。

OID介绍

使用代理键的一个扩展是惟一标识符的使用，也叫做对象标识符（OID）。OID是一个数值或有顺序的数值和数字的字符串，添加到用于表示实体的每个表格的每个记录中（有时，添加到表示关系的表格记录中）。与客户机代码、发票号、SSN或购买顺序号不同，OID是随机产生的，没有任何顺序原则并且对于最终用户不可视。这意味着我们仍能使用代理键和OID，但所有对表格的外部引用都要基于OID。

该方法的发起人建议的另一个解决办法（是支持对象关系映射理论的一部分）是，在系统范围内使用惟一的标识符。如果我们有一个客户公司的表格和一个员工的表格，为什么要为这样多样的数据使用惟一的ID呢？原因是，如果我们这样做了，就能够在不复制员

工信息到客户表格的情况下向员工推销货物，只需在订单和发票中引用员工即可。订单可以某人的OID来放置，该OID能引用许多不同表格。

使用OID和对象关系映射是Delphi应用程序设计的高级元素，我们建议在着手中等或大型Delphi项目前研究该主题，因为这有关利益问题（对该方法进行研究并且建立一些基础支持代码也需要先期投入）。

外部键和引用的完整性

标识记录的键在其他表格中可以用做外部键，例如，显示以前讨论的关系的多种类型。所有SQL服务器有能力确定这些外部引用，因此我们不能引用一个在其他表格中不存在的记录。当我们创建表格时，这些引用完整性限制是明确的。

除了不允许添加引用到不存在的记录之外，我们通常还要防止删除有外部引用的记录。一些SQL服务器更进一步：当我们删除一条记录时，不是简单地执行否定操作，而是在其他表格中删除所有引用它的记录。

更多限制

除了主关键字的惟一性和引用的限制之外，我们还能使用数据库在数据上实施更有效的规则。可以使用特定的栏（例如指向一个税务ID或者订单号码）来包括惟一值。也能利用多个栏的值的惟一性——例如，我们在同一时间不能在同一教室同时上两种课。

通常，可以在表格上定义简单规则来实施限制，而复杂的规则通常意味着需要执行被触发器（例如，每次数据发生变化或者接收到新数据）激活的存储过程来完成。

此外，还有更适当的数据库设计，但在本小节介绍的内容能够给读者提供一个初步的认识，或着说是一个有益的知识更新。

说明：关于SQL的数据定义语言与数据处理语言的更多信息，参见附录C介绍的电子图书中的“SQL基础”部分。

单向光标

在本地数据库中，数据表格是连续文件，其顺序是物理顺序或由索引定义的顺序。SQL服务器却正相反，它对数据的逻辑集合进行操作，与物理顺序无关。关系型数据库服务器根据关系模型（基于集合理论的数学模型）来处理数据。

目前讨论的重点是，数据表格的记录（有时称做元组）不是通过位置来识别的，而是惟一地通过主要关键字（基于一个或多个字段）来识别的。一旦我们获得了一个记录集，服务器就会为每个记录添加对下一个记录的索引，这使得从一个记录移到下一个记录变得很快捷，但从一个记录移到前一个记录却非常慢。因此，通常说RDBMS使用了一种单向光标。将这样的数据表格或查询连接到DBGrid控件中是不实际的，因为它使得反向浏览网格的速度缓慢得让人难以忍受。

某些数据库引擎将已经取回的数据保存在缓存值中，以支持完全的双方向定位。在Delphi的体系结构中，这个功能可以由ClientDataSet组件或者其他缓存数据集来完成。我们将在稍后讨论dbExpress和SQLDataset的时候详细介绍这个过程。

说明：使用DBGrid浏览整个数据表格的情况在本地程序中是常见的，但在客户机/服务器环境中通常应避免这样做。比较好的做法是过滤出部分记录以及感兴趣的字段。当我们需要查看一个名称列表时，可以从字母A开始，然后是B，以此类推，或要求用户输入名称的首字母。

请注意，如果反向移动会导致问题的话，那么跳到数据表格的最后一个记录可能会导致更坏的结果。通常，这个操作表示取回所有记录！相似的情况还可以应用于数据集的RecordCount属性。计算记录的数量通常意味着要将它们全部移到客户端，这就是为什么DBGrid的垂直滚动条的滚动块可在本地数据表格中使用，而对于远程数据表格不能使用的原因。如果我们需要了解记录的数量，可以运行一个单独的查询，让服务器（而不是客户机）计算它。例如，如果我们对那些薪金字段超过50 000的记录感兴趣的话，可以查看一下EMPLOYEE数据表格中有多少这样的记录：

```
select count(*)  
from Employee  
where Salary > 50000
```

提示：使用SQL指令count(*)对于计算由查询返回的记录数目是很方便的。除了使用“*”通配符之外，我们还可以使用一个指定字段的名称，如count(First_Name)等，此外也可以与distinct或all组合，来计算字段值与众不同的记录或所有具有非空值的记录。

InterBase简介

尽管市场份额有限，但InterBase仍是一个功能很强的RDBMS。在本节中，我们将介绍InterBase的主要特征，但不会过多地讨论细节问题。事实上，这是一本关于Delphi编程的书。遗憾的是，目前很少有关于InterBase的书出版，可以找到的资料一方面来自产品的附带文档，一方面来自几个Web站点，如www.borland.com/interbase以及www.ibphoenix.com站点。

InterBase从一开始就具有现代的和稳定的结构。它的原作者Jim Starkey创造了一种结构，用于处理并行操作和事务，而不用在表格部分施加物理锁，这一点甚至其他一些著名的数据库服务器今天也很难做到。InterBase结构叫做多代结构（MGA），它同时处理多个用户对相同数据的访问，使用户们可以修改记录而不影响其他同时在数据库中操作的用户。

该方法自然地映射到可重复读取（Repeatable Read）事务隔离模式，在该模式下，用户在一个事务内保持相同数据，而不管其他用户所完成和提交的更改。从技术上来讲，服务器要通过为每个打开的事务维护被访记录的不同版本来实现此目的。尽管该方法会导致大量的内存消耗，但它会避免表格的物理锁定，并使系统在面临崩溃的情况下显得更强壮。MGA也推出了一个清晰的编程模式Repeatable Read，但其他著名的服务器在支持它的时候也会造成性能的巨大下降。

除了位于InterBase核心的MGA以外，该服务器还具有许多其他技术优势：

- 配置要求有限，它使InterBase成为可以直接在客户计算机上（包括笔记本）运行的理想的候选。InterBase最小安装所需的磁盘空间小于10MB，它的内存要求也很低。
- 在处理大量数据时有良好的性能。
- 在不同的平台上均可使用（包括32位的Windows、Solaris和Linux），具有完全兼容版本，它使服务器可从小系统升级到大系统而不会造成太大的变化。

- 具有良好的使用记录，InterBase已使用了15年且极少出现问题。
- 是一种非常近似于ANSI SQL标准的兼容语言。
- 具有高级的编程功能，包括位置触发器、可选存储过程、可更新视图、异常、事件、发生器等。
- 安装和管理简单，很少给管理员带来问题。

InterBase简史

Jim Starkey是为他自己的公司Groton Database Systems编写InterBase的（因此.gds扩展名仍用于InterBase文件）。该公司后来被Ashton-Tate买下，随后又被Borland收购。Borland公司直接负责InterBase，然后又创建了InterBase子公司，该子公司后来又被重新吸收回母公司。

从Delphi 1开始，InterBase的评估版本就随着开发工具一同销售，在开发者之间传播数据库服务程序。尽管它没有大块的RDBMS市场（它被一批厂商所占有），但InterBase仍被几个相关组织选中，从Ericsson到美国国防部，从股票交易所到家庭银行系统。

最近的事件包括声明将InterBase 6作为开放式源数据库（1999年11月），对公众发布源代码的有效版本（2000年7月），Borland发布官方认可的InterBase 6版本（2001年3月）。在这些事件中，有些独立的公司声明可以在开放源数据库之上提供咨询和支持服务。尽管同以前的InterBase开发者和管理者（已离开公司）的接触并没有达成协议，但集团已决定根据支持InterBase用户的计划，创建IBPhoenix（www.ibphoenix.com）。

同时，InterBase专家组成的独立集体启动了Firebird开放源项目来进一步扩展InterBase。这个项目使用SourceForge主机，地址为：sourceforge.net/projects/firebird/。在早些时间前，SourceForge也曾经为Borland的开放源代码项目提供主机，但不久以后，该公司宣布它只继续支持本公司的专有版本，而放弃它在开放源代码方面所做的努力。因此，现在的情况很明确。即如果你需要一个带有传统许可证的版本（需要花费的代价相当于竞争对手的SQL服务器价格的零头），那么就去找Borland；但是如果你喜欢开放源代码的彻底的免费模式，那么就跟随Firebird计划（当然，要从IBPhoenix购买专业支持服务）。

使用IBConsole

在InterBase的老版本中，我们能利用两种主要工具与程序进行直接的互操作：服务器管理器应用程序（用于管理本地和远程服务器）和Windows交互式SQL（WISQL）。版本6中包含更强大的前端应用程序，被称为IBConsole。这是一个羽翼丰满的Windows程序（是为Delphi建立的），允许我们管理、配置、测试和查询InterBase服务器，无论是在本地还是从远程。

对于管理InterBase服务器和它们的数据库来说，IBConsole是一个简单而且完整的系统。我们能使用它详细查看数据库的结构，修改它，查询数据（它有益于开发我们想要嵌入程序中的查询），备份和恢复数据库以及完成所有其他管理任务。

如我们在图14.1中看到的，IBConsole允许我们方便地管理列在单个配置树上的多个服务器和数据库。可以请求关于数据库的全面信息并列出它的实体（例如表格、域、储存过程、触发器等），并且详细访问每一个。也能创建新数据库并配置它们，备份文件，更新定义，

检查正在进行什么操作以及谁正在被连接等等。

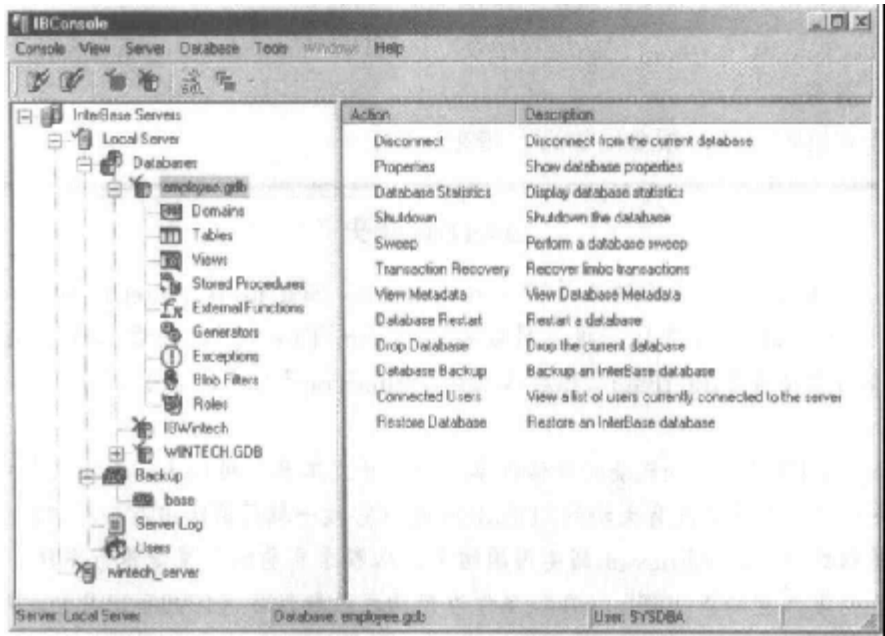


图14.1 IBConsole允许我们从一台计算机上管理放在多个服务器上的InterBase数据库

IBConsole应用程序允许我们打开多个窗口来详细查看信息，例如用图14.2描述的表格窗口。在该窗口中，我们能够看到每个表格关键属性的列表（栏、触发器、约束和索引），能够看到未加工的元数据（表格的SQL定义），能够访问许可，还能够查看实际数据，修改它并且研究表格的相关性。对于我们定义在数据库的其他实体也可以使用类似窗口进行管理和操作。

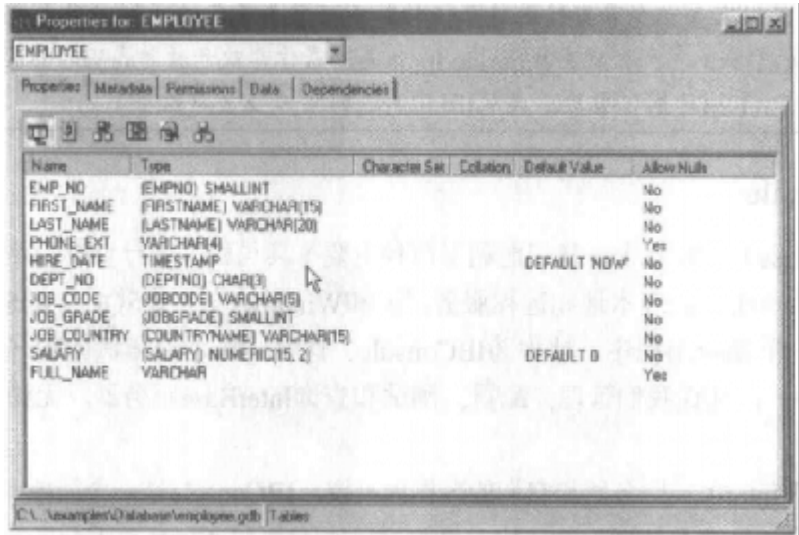


图14.2 IBConsole可以打开不同的窗口来显示每一个表项的详细内容，本图里，是一个表格

此外，IBConsole嵌入了一个原始的Windows交互式SQL应用程序的改进版本（如图14.3所示）。我们能直接在窗口的上半部分输入SQL语句（遗憾的是，不能从这个工具得到帮助信息），然后执行SQL查询。作为结果我们能看到数据，也能看到访问数据库所使用的方案

(专家能根据这个信息决定查询的有效性) 和对服务器执行的实际操作的统计。

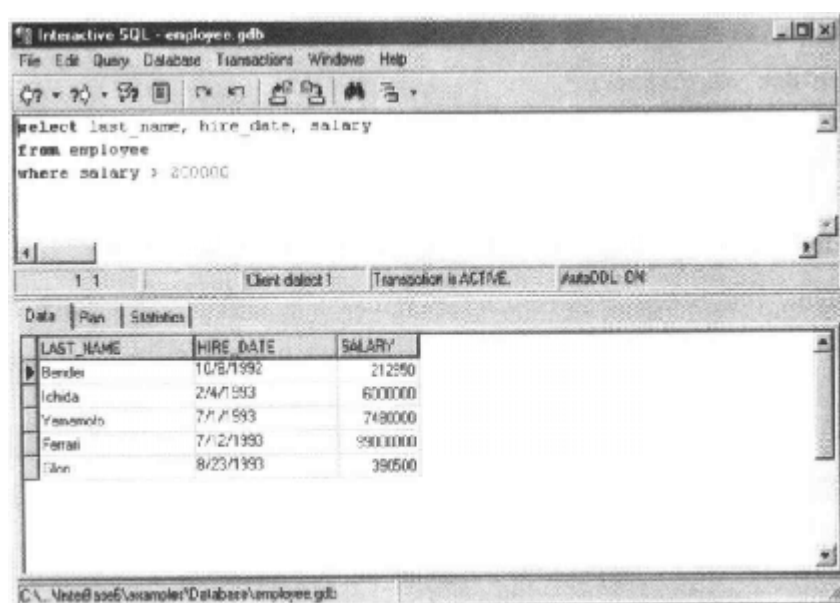


图14.3 IBConsole的交互式SQL窗口允许我们预先试验将要放入Delphi程序中的查询语句

以上是一个关于IBConsole最小限度的描述，它是一个相当强大的工具，并且是除了命令行工具以外，惟一的和服务器一起被Borland包含的工具。IBConsole在它的同类中并不是最完备的工具。有些第三方的InterBase管理应用程序的确非常强大，尽管它们不完全稳定或者用户界面不十分友好。一些InterBase工具是共享软件程序，而其他都是免费的。InterBase Workbench (www.upscene.com) 和IB_WISQL (InterBase Objects的一部分, www.ibobjects.com) 是其中的两个例子。

InterBase服务器端编程

本章开始我们曾经强调过，客户机/服务器编程的目的之一，也是它的问题之一，是如何分配所涉及的计算机之间的工作量。当从客户端激活SQL语句时，大部分的工作就落在了服务器上。然而，我们应该尽量使用select语句来返回大型的结果集合，以避免网络拥塞。

除了接受数据定义语言 (DDL) 与数据处理语言 (DML) 的请求外，大部分RDBMS服务器允许我们使用标准SQL语句，加上它们自己的服务器专用扩展特性 (通常是不可移植的)，直接在服务器上建立例程。这些例程有两种典型的形式，存储过程与触发器。

存储过程

存储过程类似于一个Delphi中的全局函数，必须由客户端明确地调用。存储过程通常用于定义数据维护的例程，编排不同情况下所需操作的顺序或者处理复杂的select语句。

与Delphi过程一样，存储过程可以有一个或多个类型参数。但它与Delphi过程不一样的是，它有不只一个返回值。除了返回值外，存储过程还可以返回结果集合——内部select语句的结果或者用户自定义语句的结果。

下面是为InterBase编写的一个存储过程，它用于接收输入的日期并计算当日雇员中的最高薪金：

```
create procedure MaxSalOfTheDay (ofday date)
returns (maxsal decimal(8,2)) as
begin
    select max(salary)
    from employee
    where hiredate = :ofday
    into :maxsal;
and
```

请注意into子句的用法，它会通知服务器在maxsal返回值中存储select语句的结果。为了修改或删除存储过程，可以使用alter procedure与drop procedure命令。

对于这个存储过程，读者可能会问，与从客户端执行相似的查询相比，其优点是什么？这两种方法的区别不在于我们获得的结果，而在于它们的速度。当建立存储过程时，它会在服务器上快速地被编译，而且服务器会同时确定它用于访问数据的策略。相比而言，客户端查询则是每次向服务器申请时都要编译的查询（尽管服务器可以高速缓存它并避免重复编译两个相同的请求）。因此，存储过程可以代替非常复杂的查询，条件是不要太频繁地改动它！

对于Delphi来说，可以使用下面的SQL代码来激活一个存储过程：

```
select *
from MaxSalOfTheDay ('01/01/2003')
```

触发器（以及生成器）

触发器多少有些像Delphi事件，当给定事件发生时会自动激活。触发器可以拥有特定代码或调用存储过程。在这两种情况中，操作都是在服务器上完成的。触发器可用于保持数据的一致性，通过更复杂的（限定检测所无法做到的）方法检测新数据，并自动实现一些输入操作的附加效果（例如，当改动当前薪金时建立过去薪金改动的记录）。

触发器可以通过三个基本的数据更新操作来激活：即insert, update与delete。当我们建立触发器时，需要指定该触发器应该在这三种操作之前还是之后激活。

作为触发器的例子，我们使用生成器在数据表格中按顺序建立惟一索引。很多数据表格将惟一索引用作主关键字。与Paradox以及其他本地数据库不同，InterBase没有AutoInc字段。因为多个客户不能生成惟一的标识符，所以可以依靠服务器来完成。几乎所有的SQL服务器都提供了一个计数器，可以请求一个新的ID，将它用于数据表格。InterBase称这些自动计数器为生成器（Generator），而Oracle称它们为序列（Sequence）。下面是InterBase代码的范例：

```
create generator cust_no_gen;
...
gen_id (cust_no_gen, 1);
```

然后，gen_id函数会提取生成器的新的惟一值作为第一个参数传递，第二个参数指定增加的量（在本例中为1）。

在这里，我们可以向数据表格添加一个触发器，即一个数据表格事件的自动处理程序。触发器与Table组件的事件处理程序相似，但我们要在SQL中编写它并在服务器上执行它，而不是在客户端。下面是一个例子：

```
create trigger set_cust_no for customers
before insert position 0 as
begin
    new.cust_no = gen_id (cust_no_gen, 1);
end
```

该触发端是为Customer数据表格定义的，每当插入新记录时都会激活它。new符号表示我们要插入的新记录，position选项指定多个与相同事件相连的触发器的执行顺序。具有最低值的触发器将首先执行。

在一个触发器中，我们还可以编写更新其他数据表格的DML语句，但注意，更新会导致重复激活触发器，出现无穷递归。在更新后可以调用alter trigger语句或drop trigger语句修改或关闭触发器。

触发器会自动被指定事件激活。如果我们必须在数据库上使用批处理操作做很多改动的话，使用触发器会减慢进程速度。如果输入数据已经完成了一致性检测，那么我们可以临时关闭触发器。这些批处理操作通常被编与在存储过程中，但存储过程通常不能执行DDL语句，也就是用于重新激活或者关闭触发器的语句。在这种情况下，我们可以定义一个视图（View），这个视图是基于select * from table命令的，这样，就给这个表格建立一个别名（alias）。接下来，使用存储过程在表格上执行批处理，同时将触发器应用到视图上，当然，客户程序也需要使用该视图。

dbExpress库

目前，在Delphi中用于访问SQL服务器数据库的主流方法是由dbExpress提供的。我们在第13章“Delphi的数据库体系结构”中提到过，这不仅是可能的，而且是当然的主流方案。首次在Delphi 6以及Kylix中引进的dbExpress库，允许访问不同的服务器（InterBase、Oracle、DB2、MySQL、Informix和目前的Microsoft SQL Server）。我们在第13章中已经简单地介绍了dbExpress，并且与其他的解决方案做了比较。因此现在让我们就技术问题集中讨论一下。

说明：包含支持微软SQL Server的驱动程序是Delphi 7对于dbExpress最重要的更新。它不是通过与供应商的库连接来实现的（与其他dbExpress驱动程序类似），而是连接到用于SQL Server的微软OLEDB供应器（我们将在第15章中详细讨论）。

使用单向光标工作

dbExpress的格言是“获取但不缓冲”。该库和BDE或ADO间的关键不同之处在于，dbExpress只能执行SQL查询并且在单向光标上取得结果。在“单向”数据库访问中，我们能从一个记录移动到另一个记录，但不能返回到数据集前面的记录中（除非我们重新打开这个查询并且重新获取所有的记录，而这将会是一个慢得出奇的操作）。这是因为这个库不能把它取得的数据存储在本地缓存中，而只能把它们从数据库服务器传递到调用程序上。

使用单向光标更像是一种限制，而事实正是如此。除了导航定位问题外，我们不能将一个数据库网格连接到数据集上。那么，一个单向数据集的好处在哪里呢？

- 我们能使用一个单向数据集进行报告。在打印的报告中，以及在HTML页面或XML转换中，我们能一个一个记录地移动，产生输出。通常，这种情况下，不需要返回到过去的记录，也不需要数据和用户的交互。单向数据集对于网络和多级结构是最好的选择。
- 我们能在本地缓存中使用单向数据集，如ClientDataSet组件提供的缓存。在这点上，我们能够将可视组件连接到内存区数据集，并运用所有标准技术进行操作，包括可视网格的使用。我们能在内存缓存区中定位和编辑数据，也能比BDE或ADO更好地控制它。

请注意，很重要的一点是，在这些情况下，避免使用数据库引擎的缓存能够节省时间和内存。该库不必为缓存使用额外的内存，也不需要浪费时间存储数据，复制信息。在过去几年中，许多程序员从基于BDE的缓存更新转移到ClientDataSet组件，它对于管理数据的内容和更新保存在内存中的信息提供了更大的灵活性。然而，在BDE或（ADO）顶部使用ClientDataSet使我们有可能拥有两个独立缓存，从而浪费大量内存。

使用ClientDataSet组件的另一个优点是它的缓存支持编辑操作，并且在该缓存存储的更新能通过DatasetProvider组件被应用到最初的数据库服务器。该组件能产生适当的SQL的update语句，采用比BDE更灵活的方式（虽然ADO在这个方面更加强大）。通常，提供者为了更新也使用数据集，但不太可能直接使用dbExpress数据集组件。

平台与数据库

与用于Delphi（BDE和ADO）的所有其他数据库引擎（它们只能用于Windows）相比，dbExpress库的一个关键特性在于它可以同时应用于Windows和Linux。但是，请注意，一些数据库专用组件，如InterBase Express也可以在多平台上使用。

当我们使用dbExpress时，得到的是一个通用的框架，该框架独立于我们计划使用的实际SQL数据库。dbExpress带有MySQL、InterBase、Oracle和IBM DB2的驱动程序。

说明：为dbExpress体系结构写定制驱动程序是可能的。在dbExpress草案规范页中（Borland联合网络站点出版）有详细说明。在编写本书时，该文档位于<http://community.borland.com/article/0,1410,22495,00.html>。我们能发现第三方的驱动程序。例如，有一个免费的驱动程序（在Kylx也能获得），它连接着dbExpress和ODBC。所有的文章列表可以在下面的连接中找到：<http://community.borland.com/article/0,1410,28371,00.html>。

驱动器版本问题以及嵌入单元

从技术上说，dbExpress驱动程序是一些不同的DLL，我们需要在程序中进行部署。这是在Delphi 6中的情况，到Delphi 7也是保持同样的状况。问题是，DLL的名称是不能改变的。因此，如果在一台装有Delphi 6的dbExpress驱动程序的机器上安装一个Delphi 7应用程序，那么这个程序还是能够运行起来的，并且能打开一个到服务器的连接，然后在取回数据的时候发生错误。这时，我们将看到这个错误：“SQL Error: Error mapping failed。”可见，如果在dbExpress的版本上存在不兼容问题，那么这不是一个好的提示。

为了验证这个问题，让我们来查看DLL文件是否带有任何版本信息：在Delphi 6的驱动程序中丢失了。为了使我们的应用程序更加强壮，可以在代码中提供一个类似的检查，使用

相关的Windows API来访问版本信息:

```
function GetDriverVersion (strDriverName: string): Integer;
var
  nInfoSize, nDetSize: DWord;
  pVInfo, pDetail: Pointer;
begin
  // the default, in case there is no version information
  Result := 6;

  // read version information
  nInfoSize := GetFileVersionInfoSize (pChar(strDriverName), nDetSize);
  if nInfoSize > 0 then
    begin
      GetMem (pVInfo, nInfoSize);
      try
        GetFileVersionInfo (pChar(strDriverName), 0, nInfoSize, pVInfo);
        VerQueryValue (pVInfo, '\\', pDetail, nDetSize);
        Result := HiWord (TVSFixedFileInfo(pDetail^).dwFileVersionMS);
      finally
        FreeMem (pVInfo);
      end;
    end;
  end;
```

这个代码片段是从DbxMulti范例中提取的, 我们稍后将要讨论。程序使用它可在版本不兼容的情况下产生一个异常:

```
if GetDriverVersion ('dbexpint.dll') <> 7 then
  raise Exception.Create (
    'Incompatible version of the dbExpress driver "dbexpint.dll" found');
```

如果我们将Delphi 6的bin文件夹中的驱动程序放到应用文件夹中, 将会发生错误。我们必须修改这个额外的对于驱动程序更新版本的安全检查, 但是这个步骤应该帮助我们在第一时间避免dbExpress安装上的错误。

我们还可以使用另外一个办法: 将dbExpress驱动程序的代码静态连接到应用程序。为了完成这个任务, 应在程序中包含一个给定的单元 (诸如dbexpint.dcu或者dbexpora.dcu), 使用一个uses指令将其列出。

警告: 连同这些单元中的一个, 我们需要包含MidasLib单元并且连接MIDAS.DLL的代码到应用程序中。如果这个操作失败了, Delphi 7的链接程序将会停止显示内部错误, 因为这是一个没有什么意义的信息。注意, 嵌入的dbExpress驱动程序使用国际字符集。

dbExpress组件

用做dbExpress库接口的VCL组件包含一组数据集组件和几个辅助组件。为了将这些组件同其他数据库访问系列区分开来, 应给它们加以前缀字母SQL, 表明用于访问RDBMS服务器。

这些组件包括一个数据库连接组件，几个数据集组件（一个普通组件，三个特殊组件用于表格、查询和存储过程，以及一个封装有ClientDataSet的组件）和一个监控工具。

SQLConnection组件

TSQLConnection类继承自TCustomConnection组件，并处理数据库连接，像它的兄弟类一样（Database、ADOConnection和IBConnection组件）。

提示：与其他组件系列不一样，在dbExpress中，连接是强制的。在数据集的每个组件中，我们不能直接确定使用哪个数据库，但只能引用一个SQLConnection。

连接组件使用在drivers.ini和connections.ini文件上可获得的信息，它们是dbExpress的两个配置文件，这些文件默认存储在Common Files\Borland Shared\DBExpress目录下。第一个文件drivers.ini列出了可用的dbExpress驱动程序，用于对每一个数据库提供支持。对于每个驱动程序，有一套默认连接参数。例如，其InterBase部分如下所示：

```
[Interbase]
GetDriverFunc=getSQLDriverINTERBASE
LibraryName=dbexpint.dll
VendorLib=GDS32.DLL
BlobSize=-1
CommitRetain=False
Database=database.gdb
Password=masterkey
RoleName=RoleName
ServerCharSet=ASCII
SQLDialect=1
Interbase TransIsolation=ReadCommitted
User_Name=sysdba
WaitOnLocks=True
```

这些参数指出了dbExpress驱动程序DLL（LibraryName的值）、要使用的登录函数（GetDriverFunc）、厂商客户库和一些依赖数据库的专用参数。如果我们读取整个drivers.ini文件，将看到参数是数据库专用的。我不得不说，这些参数中的一些在驱动程序级（数据库连接到这里）没起多大作用，但列表包括所有可用参数，不管它的实际使用如何。

connections.ini文件提供数据库专用的描述信息。该列表将各项设置与一个名称进行关联，我们能为每个数据库驱动程序输入多个详细连接，用以描述我们想连接的物理数据库。作为示例，这里给出默认IBLocal定义的一部分：

```
[IBLocal]
BlobSize=-1
CommitRetain=False
Database=C:\Program Files\Common Files\Borland Shared\Data\employee.gdb
DriverName=Interbase
Password=masterkey
RoleName=RoleName
ServerCharSet=ASCII
```

```
SQLDialect=1
Interbase TransIsolation=ReadCommitted
User_Name=sysdba
WaitOnLocks=True
```

比较两个列表，我们可以看到，这是参数的一个子集。当我们创建一个新连接时，系统将从驱动程序拷贝默认参数，我们能为特殊连接编辑它们——例如，提供一个适当的数据库名字。每个连接对于它的关键属性都涉及到驱动程序，像DriverName属性描述的那样。还需要注意的是，这里引用的数据库是我编辑的结果，与我将所有范例中所用的设置有关。

需要注意的一件重要的事情是这些初始化文件只在设计时使用。事实上，当我们在设计时挑选一个驱动程序或一个连接时，这些文件的值便被拷贝到SQLConnection组件的对应属性上，如下例所示：

```
object SQLConnection1: TSQLConnection
    ConnectionName = 'IBLocal'
    DriverName = 'Interbase'
    GetDriverFunc = 'getSQLDriverINTERBASE'
    LibraryName = 'dbexpint.dll'
    LoginPrompt = False
    Params.Strings = (
        'BlobSize=-1'
        'CommitRetain=False'
        'Database=C:\Program Files\Common Files\Borland Shared\Data\employee.gdb'
        'DriverName=Interbase'
        'Password=masterkey'
        'RoleName=RoleName'
        'ServerCharSet=ASCII'
        'SQLDialect=1'
        'Interbase TransIsolation=ReadCommitted'
        'User_Name=sysdba'
        'WaitOnLocks=True')
    VendorLib = 'GDS32.DLL'
end
```

在运行时，我们的程序将依赖属性拥有所有请求信息，因此我们不需要随着程序部署两个配置文件。理论上，如果我们想在运行时改变DriverName或ConnectionName属性，将需要这些文件。然而，如果需要连接程序到新数据库，我们就可以直接设置相关的属性。

当我们添加一个新SQLConnection组件到应用程序时，能以不同方式进行。譬如，建立一个驱动程序，使用可用于DriverName属性的一系列值，然后通过为ConnectionName属性挑选一个可用值来选择一个预定义连接。依据我们已挑选的驱动程序，第二个列表被过滤。另外一种方法是直接从选择ConnectionName属性开始，这时它包括全部列表。

除了保持已有的连接外，我们还可以通过双击SQLConnection组件并启动dbExpress Connection Editor来定义一个新连接（如图14.4所示）。左边的编辑器列出了所有用于指定驱动程序或所有驱动程序的预定义连接，右边的网格用于编辑连接属性。我们能使用工具

栏按钮来添加、删除、重命名和测试连接并打开只读dbExpress Drivers Settings窗口, 如图14.4所示。

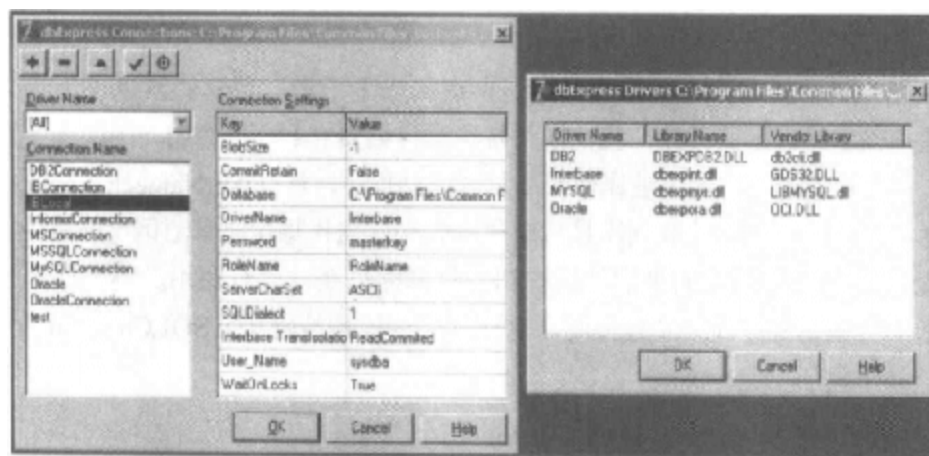


图14.4 带有dbExpress驱动程序设置对话框的dbExpress Connection编辑器

除了编辑预定义连接设置外, dbExpress连接编辑器允许我们为SQLConnection组件挑选一个连接, 这是通过OK按钮来完成的。注意, 事实上, 如果我们改变一些设置, 数据会被立即写到配置文件上: 单击Cancel按钮也不能撤销我们的编辑。

如果想定义对数据库的访问, 编辑连接属性是值得推荐的方法。这样, 当我们从其他应用程序或同一应用程序中的另一个连接来访问相同数据库时, 所需要做的只是挑选连接。然而, 因为该操作要复制连接数据, 因此更新连接不会自动在其他引用同名连接的SQLConnection组件中刷新值: 我们不得不重挑选其他组件引用的连接。

对于SQLConnection组件来说, 真正重要的是它的属性值。驱动程序和厂商库被列在属性中, 我们能在设计时自由改变它, 而数据库和其他指定数据库的连接设置由Params属性说明。这是一个包括数据库名字, 用户名和密码等信息的字符串列表。实践中, 我们能通过建立驱动程序来建立SQLConnection组件, 然后直接在Params属性上分配数据库名, 忘记所有预定义连接。我不赞成将其作为最好的选择, 但它是一种可能, 预定义连接是很方便的, 但当数据改变时, 我们不得不更新每个SQLConnection组件。

说到底, 我们不得不承认还有另外一种选择。通过设置LoadParamsOnConnect属性来说明当每次打开连接时, 我们想从初始化文件更新组件的参数。这时, 若我们在设计时或运行时打开连接, 对预定义的修改将被重新加载。在设计时, 这提供了一个方便的技术(它与重新选择这个连接的效果相同), 但在运行时使用它意味着需要使用connections.ini文件, 这到底是一个好主意还是一个麻烦的方法, 还需要取决于我们的实际环境。

SQLConnection组件的惟一不涉及到驱动程序和数据库设置的属性是LoginPrompt。设置它为False, 允许我们在组件设置中提供一个密码, 在设计时和运行时跳过登录请求。这对开发是方便的, 但它能降低系统的安全性。当然, 这也是我们想用于自动连接的选项, 例如在一个网站服务器上。

dbExpress数据集组件

dbExoress组件系列提供四种不同的数据集组件：一般的数据集、表格、查询和存储过程。后三个组件用于兼容等效的BDE组件并拥有类似的命名属性。如果不必非得使用现有的代码，那么我更趋向于使用一般的SQLDataSet组件，该组件能用于执行查询，也能用来访问表格或存储过程。

首先需要注意，所有这些数据集均继承自一个新的特殊的基类TCustomSQLDataSet。该类和它的父类显示单向数据集，拥有前面已提及的关键特征。实际上，这意味着浏览操作被限制调用First和Next；而Prior、Last、Locate、书签和所有其他定位功能都无法使用。

说明：从技术上讲，一些移动操作要调用CheckBiDirectional内部函数并最终产生一个异常。

CheckBiDirectional引用TDataSet类的公共IsUnidirectional属性，我们可以在自己的代码中使用它，以禁用在单向数据集中非法的操作。

除了具有有限的导航（navigational）能力外，这些数据集没有编辑支持功能，因此无法获得很多对其他数据集来说很常见的方法和事件。例如不具备AfterEdit或BeforePost事件。

像前面已提到的，在dbExpress的四个组件中，基本的一个是TSQLDataSet。它可用于检索数据集和执行一条命令。通过调用Open方法（或将Active属性设置为True）和ExecSQL方法，即可激活这两种操作。

SQLDataSet组件能取回整个表格的内容，也能使用SQL查询或存储过程来读取数据集或发出命令。CommandType属性决定着三个访问模式中的一个。可能的值是ctQuery、ctStoredProc和ctTable，它们决定了CommandText属性的值（以及对象检验器中的相关属性编辑器的操作）。对于一个表格或存储过程，CommandText属性给出的是数据库相关元素的名字，而编辑器提供的是一个有可能值的下拉列表。对于一个查询，CommandText属性存储的是SQL命令的文本，而编辑器在建立SQL查询中提供一点帮助。如图14.5所示。

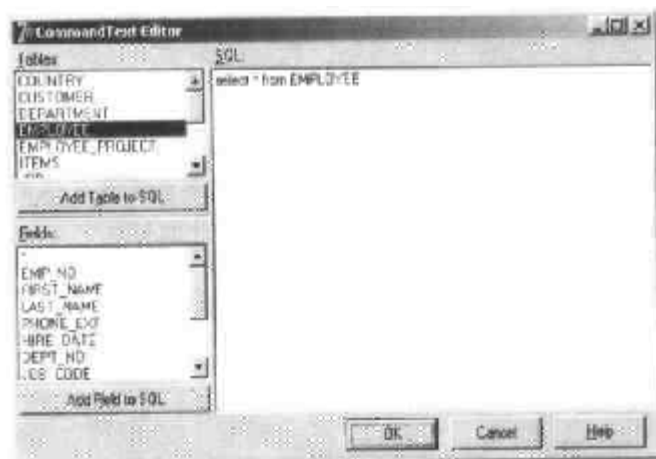


图14.5 SQLDataSet用于查询的CommandText编辑器

当我们使用表格时，组件能为我们产生SQL查询，因为dbExpress只指向SQL数据库。产生的查询将包括表格的所有字段，而如果我们指定SortFieldNames属性，它将包括一个sortby指令。