Borland Delphi 程序设计

蒋方帅　编著

清 华 大 学 出 版 社

（京）新登字 158 号

内 容 简 介

本书全面介绍了 Delphi 6.0 面向对象的编程方法。主要内容包括：Delphi 6.0 简介；Object Pascal 语法；面向对象编程基础；控件、工程和应用程序；Delphi 的基本控件；工具控件和图形控件；菜单、工具栏和标准动作；Delphi 中的绘图；开发数据库应用程序；控件设计；系统信息管理；控件设计高级技术等。

本书适用于学习 Delphi 编程的初、中级用户，初级用户可以通过学习本书前 3 章的基本内容快速入门，而中级用户则可以通过本书后 8 章的内容获得许多编程经验和高级技巧。本书非常适合作为 Delphi 6.0 的社会培训班教材，上课用其中的基础部分，在学生上完课后通过本书还有较大的发展空间。

图书在版编目 (CIP) 数据

Borland Delphi 程序设计/蒋方帅编著. —北京：清华大学出版社，2002.4
ISBN 7-302-05298-0

Ⅰ. B... Ⅱ. 蒋... Ⅲ. Delphi 语言–程序设计–高等学校–教材 Ⅳ. TP312

中国版本图书馆 CIP 数据核字 (2002) 第 013182 号

出 版 者：清华大学出版社（北京清华大学学研大厦，邮编 100084）
http://www.tup.tsinghua.edu.cn
责任编辑：田在儒
印 刷 者：北京清华园胶印厂
发 行 者：新华书店总店北京发行所
开 本：787×960 1/16 印张: 30.75 字数: 690 千字
版 次：2002 年 4 月第 1 版 2002 年 4 月第 1 次印刷
书 号：ISBN 7- 302 - 05298 - 0/TP・3115
印 数：0001~5000
定 价：43.00 元

前　言

1. 关于 Delphi

在 DOS 年代，程序员可以选择的开发工具是非常有限的。随着 Windows 平台的推出，这种情况有所改善，人们可以使用 C++语言或者 Visual Basic 语言。但是它们要么功能虽然强大但是使用非常困难，要么使用虽然简单但是语言本身具有重大的局限。因此人们迫切需要一种能够集两者优点而摒弃两者缺点的开发工具——这便是 Delphi 1.0 产生的历史背景。在古希腊的神话里，有一个智慧女神，她的名字叫 Delphi，也许这就是 Borland 公司把这个强大的开发工具命名为 Delphi 的最初意愿吧。事实上，Delphi 名副其实，它是第一个综合了可视化开发环境、优化的源代码编译器和可伸缩的数据库访问引擎的 Windows 开发工具。

随着 Windows 平台的不断升级，Delphi 也完成了自己从 1.0 到 6.0 的阶梯迈进。在 Delphi 2.0 中，Delphi 实现了从 16 位平台向 32 位平台的转移，初步形成了 RAD（Rapid Application Development）的概念。随着软件技术的进步，COM、ActiveX、WWW、多层数据库应用程序等复杂技术越来越多地进入到程序员的日常开发工作中，Borland 公司适时地推出了 Delphi 3.0，为上述复杂技术的应用提供了一套完整的解决方案。为了方便程序员们编写程序，在 Delphi 3.0 中，还加入了 Code Insight 功能。Delphi 4.0 的推出首先是为了使 Delphi 的使用更加简单，为此 Borland 加入了代码导航和类自动完成功能，同时 MIDAS、DCOM、CORBA 等技术的应用更使得 Delphi 成为企业级的开发平台。之后不久，Borland 很快便推出了 Delphi 5.0。Delphi 5.0 在数据库的开发、集成环境的改善、VCL 的增强和 COM 服务器的扩展上都取得了长足的进步，引入了 ADO 数据集，并能够有效控制 Microsoft Excel、Microsoft Word 等 COM 服务器。

Delphi 的每次更新，都给程序员们带来了巨大的惊喜，人们在使用 Delphi 5.0 的同时，又在盼望着 Delphi 6.0 的出现。在事隔两年之后，Borland 终于推出了人们翘首以待的 Delphi 6.0 版本。

2. 关于本书

我们编写本书的目的是为了能够帮助那些初级的和中级的用户能够进一步了解 Delphi，并能更熟练地应用 Delphi 来开发各种应用程序。

Delphi 是一个涵盖范围非常广的软件开发工具，使用它需要的不仅仅是 Delphi 的基本知识以及 Object Pascal 语言的基本知识，还需要用户具有所要开发的软件相关领域内的诸多知识，比如你要开发一个大型的数据库应用程序，首先应该去学习关于大型数据库服务器的诸

多知识。所以，试图用一本书来介绍 Delphi 程序开发中可能涉及的所有问题的想法是不切实际的。但是本书仍然提供了尽可能多的 Delphi 程序开发的参考知识，使得读者在阅读本书后，基本上可以开发中等偏上难度的应用程序。比如，尽管在 IT 业中，数据库的存在形式和管理形式是多种多样的，但是读者在阅读了本书的第 9 章之后，参考一定的在线帮助，便可以编写各种类型的数据库应用程序，不管是简单的单层数据库应用程序，还是多层的数据库应用程序。

除了上文提到的专业知识之外，在 Delphi 本身的程序开发上也有许多的技巧和注意事项。在本书的写作过程中，使用了大量的“ 说明 ”等字样来提示读者此时需要注意的要点或者需要学习的技巧。虽然这些内容对读者跟随本书进行程序设计没有任何影响，但是对读者程序设计技术和思想的进步是十分重要的。

为了帮助读者能够更为形象地学习 Delphi，同时也为了使本书脱离空泛说教的格局，在书中提供了大量的程序代码，例如在第 8 章中编写的绘图程序，在第 10 章和第 11 章中设计的控件。读者会发现，只要对它们稍作修改，便可以成为和专业程序相媲美的自己的软件。学习程序设计的一个重要手段之一便是能够读懂别人的代码，并能够修改别人的代码为己所用。

3. 本书导读

本书按照由浅入深的顺序分为 11 章来尽可能全面地介绍 Delphi 6.0 程序设计各个方面的知识。下面分别介绍各章的主要内容，并说明它们的适用对象。

第 1 章 Delphi 6.0 简介

本章首先介绍了关于安装 Delphi 6.0 的详细步骤和注意事项，以及如何进行相应的安装配置。然后对 Delphi 6.0 的集成开发环境进行了简要的介绍，重点介绍了集成界面中各个组成部分的名称、功能和特点，并顺带介绍了 Delphi 6.0 在集成环境上的新功能，让读者对 Delphi 的开发环境有一个基本的概念，为后面的程序设计打下了一定的基础。

第 2 章 Object Pascal 语法

Object Pascal 语言是 Delphi 程序设计的基础，掌握它是进行 Delphi 程序设计的前提。在本章中，通过对 Object Pascal 语言中的特殊字符、数据类型、变量和常量、语句、过程和函数等的介绍，基本上涵盖了 Object Pascal 语言的所有方面。对于熟悉 Object Pascal 语言的读者来说，本章内容可以略过，也可以在需要的时候再来查阅。

第 3 章 Delphi 6.0 面向对象编程基础

在本章中，首先通过建立一个简单的实例程序，介绍了利用 Delphi 开发应用程序的基本步骤，对 Delphi 6.0 的强大功能进行的了解。然后利用两节的内容对面向对象的编程思想、技术以及它们与 Delphi 的关系进行了介绍。需要强调的是，面向对象的程序设计是 Delphi 程序设计的语法核心。最后，根据我们以及前人的经验，向读者介绍了编写好的应用程序所需的一些注意事项。

第 4 章　控件、工程和应用程序

在本章中，对应用程序和 Delphi 中的工程、工程和控件、窗体和控件以及控件和属性的关系进行了介绍，使读者了解到，我们要开发的应用程序，就是 Delphi 中的工程；一般来说，工程是由多个窗口组成的，而每个窗口又是通过控件来完成界面组成和功能实现的；为了定义控件的功能和行为特点，需要处理各个控件的属性和事件。同时在本章中，还介绍了 Delphi 6.0 中关于工程管理和工程设置的相关操作。

第 5 章　Delphi 的基本控件

基本控件是设计应用程序界面的基础，掌握它们非常重要。其中窗体虽然不是控件，但实际上它几乎具有所有常规控件的全部属性，所以在本章中，花大力气介绍了窗体对象的各个重要属性和方法。在掌握了它们之后，便可以比较容易地去学习其他控件的相应内容了。然后根据各个基本控件的功能，将控件分成文本控件、按钮控件、选项控件、列表控件和容器控件 5 类基本控件进行介绍。

第 6 章　工具控件和图形控件

本章所介绍的控件从本质上来说仍然属于基本控件的范围，但是它们的功能也具有一定的特殊性。在工具控件中，介绍了滚动条、过程条和文件系统等几组控件的使用方法；在图形控件中，介绍了图像控件、形状控件和图像列表控件的使用方法；在图表控件中，介绍了功能强大的 TChart 控件。

第 7 章　菜单、工具栏和标准动作

在常规应用程序的界面设计中，菜单和工具栏是非常重要的部分。在本章中，对它们进行了详细的介绍。其中包括基本的菜单设计、菜单的合并、工具栏的各种设计方法、菜单工具栏的一体化。为了设计出浮动的工具栏，在本章的内容中，还介绍了 Delphi 中对象和窗体的停靠操作的实现。在 Windows 的程序设计中，有一些内容几乎已经成为标准的操作，比如打开文件，这些内容在 Delphi 6.0 中已经被定义成一些标准的动作，本章通过一节的内容介绍了 Delphi 中的标准动作的使用。另外，本章还介绍了应用程序本身的一些事件的处理方式。通过上面的 7 章内容，特别是对示例程序的学习，读者已经可以设计各种常规的应用程序了。

第 8 章　Delphi 中的绘图

绘图是各种 Windows 软件开发工具的重要功能。在本章中，介绍了 Delphi 中的各种可以用于绘图或者可以影响绘图的对象、方法和属性。通过一个比较完整的、复杂的示例程序，不仅演示了如何应用 Delphi 中的绘图技术，更为建立一个复杂应用程序提供了范例，同时也说明，程序设计是一门非常富有技巧的技术，绝不是仅仅掌握诸多的语法、规则就可以做好的。

第 9 章　开发数据库应用程序

数据库应用程序的开发从 Delphi 2.0 开始便成为了 Delphi 的强项。通过本章对基本数据库控件和数据库访问控件的详细介绍，对其他相关控件的扼要介绍，读者基本上可以了解关于数据库应用程序开发的全面内容了。这是因为其他相关的数据库控件基本上都是从基本的

数据库控件继承来的，它们在很多方面具有共性。同时，本章还介绍了主从式数据库应用程序和多层数据库应用程序的开发。

第 10 章　控件设计

自定义控件的开发是 Delphi 具有强大的生命力的重要保证，使得用户可以通过自己编写的或者别人编写的控件获得各种各样的功能，这些特殊功能也许是 Delphi 没有提供的。正如有的文章所说，网上的 Delphi 控件多如牛毛。在本章中，不仅介绍了创建控件的基本过程（包括继承父类、添加属性、添加方法、添加事件、编写属性编辑器），还介绍了怎样来封装自己的控件并安装到 Delphi 中。最后，通过一个具体的控件实例演示了 Delphi 控件设计的基本技术。

第 11 章　控件设计高级技术

在本章中，通过编写一些可视控件和一些非可视控件，介绍了在 Delphi 中进行控件开发的一些高级技术，并特别提到了对话框控件和 ActiveX 控件的设计方法。

4. 本书的约定

为了方便读者使用本书，在编写本书的时候，作者使用了一些统一的约定，从而保持风格统一，这些约定主要包括：

- 本书所有的中文菜单项都以【】括起，而多级菜单用“|”隔开，例如，file | New。
- 本书中在一些需要特别强调或者需要展示技巧的地方多使用“注意”、“说明”等字样来标记特殊的内容，以辅助读者的学习。
- 在每章的结尾都有一个“本章小结”，对本章的主要内容进行概括总结，或强调其中的重点内容。

5. 联系作者

如果读者在学习和工作中遇到什么问题，可以访问作者的个人网站和与本书作者清华大学的蒋方帅博士联系，网址是 http://wokftp.dhs.org。

6. 致谢

本书由蒋方帅博士执笔，还有很多朋友在本书预读、程序调试、测试、校对等方面做了大量的工作，这些人有：陈河南、贺军、贺民、龚亚萍、徐江、唐永久、孟丽艳、孟韬、姜言铭、曾冬松、李志云、李志伟、戴军、陈伊文、王雷、马新、王巧红、李文、李晓春、伍模、孟秦、陶涛、杨颖、周里文、萧展业、彭少熙、李倪、李和平等人，在此对各位朋友付出的辛苦表示感谢！

作　者

2002.1.1

目　　录

第 1 章　Delphi 6.0 简介

本章首先介绍 Delphi 6.0 的安装过程，然后介绍 Delphi 6.0 的基本环境、软件界面和新增功能。以便读者在熟悉了 Delphi 6.0 的开发环境之后，可以轻松地在后面的各章中学习 Delphi 6.0 的编程。

本章主要内容有：

- Delphi 6.0 的安装
- Delphi 6.0 的界面概述

1.1　安装 Delphi 6.0

Delphi 6.0 是一个可以用来开发其他应用软件的软件。首先需要把它安装到计算机上才能完成我们希望它完成的工作。Delphi 6.0 软件的安装过程和一般应用程序的安装过程十分类似，一般需要经过以下几个步骤。

(1) 启动安装程序。
(2) 输入软件序列号。
(3) 设置要安装的组件。
(4) 指定安装目录。
(5) 复制需要的文件。
(6) 重新启动计算机。

1.1.1　Delphi 6.0 的最低系统配置

在安装一个软件之前，应该了解一些关于该软件的基本情况，比如这个软件的正常运行需要什么样的软件、硬件环境，要不要安装其他的辅助软件。同时如果软件包含了很多组件，那么我们应该去了解需要安装哪些组件。

Delphi 6.0 分两个版本，企业版和个人版。对于企业版，它的最低配置如下：

- CPU　Intel Pentium 166MHz 或者更高的 CPU（建议使用 P Ⅱ 400MHz CPU）。
- 操作系统　Microsoft Windows 2000，Windows Me，Windows 98，或 Windows NT 4.0（安装了 Service Pack 5）或者更新版本的操作系统。
- 内存　64MB（建议使用 128MB 的内存）。

- **硬盘空间** 115MB 的磁盘空间（最小安装），
  350MB 的磁盘空间（完全安装）。
- CD-ROM 驱动器。
- 标准 VGA 或者更高分辨率的显示器。
- 鼠标或其他输入设备。

对于 Delphi 6.0 的个人版，则需要以下最低配置：

- CPU Intel Pentium 166MHz 或者更高的 CPU（建议使用 P II 400MHz CPU）。
- **操作系统** Microsoft Windows 2000，Windows Me，Windows 98，或 Windows NT 4.0（安装了 Service Pack 5）或者更新版本的操作系统。
- **内存** 64MB（建议使用 128MB 的内存）。
- **硬盘空间** 105MB 的磁盘空间（最小安装），
  260MB 的磁盘空间（完全安装）。
- CD-ROM 驱动器。
- 标准 VGA 或者更高分辨率的显示器。
- 鼠标或其他输入设备。

在本书中所使用的计算机的配置如下：

- Intel 赛杨 600MHz CPU。
- Microsoft Windows XP 简体中文版
- 256MB 内存。
- 46GB 硬盘。

如果你的计算机的配置达不到上面提到的最低要求，请升级你的计算机。否则很难流畅地运行 Delphi 6.0。

1.1.2 安装 Delphi 6.0

下面就可以开始安装 Delphi 6.0 了。把 Delphi 6.0 的光盘放到 CD-ROM 中，此时一般会自动运行安装程序，并显示如图 1.1 所示的初始安装画面。

说明：

如果不能显示上面的安装画面，则可以在 Windows 的资源管理器中打开 CD-ROM，并双击根目录下的 Install.exe 文件。

下面的安装画面显示的是 Borland 提供的一个用来安装 Delphi 6.0 及其组件的综合程序。在这个对话框中，包含了 6 个安装选项，包括 Delphi、InstallShield Express Custom Edition for Delphi 等。可以根据自己的需要来选择安装某一组件。在这里我们主要介绍 Delphi 6 的安装。

(1) 单击图 1.1 中的第一个选项，也就是 Delphi 6，此时会显示如图 1.2 所示的对话框。

图 1.1　Delphi 6.0 的初始安装画面

图 1.2　安装程序的初始窗口

说明：

如果此时希望取消 Delphi 6.0 的安装，那么可以在这个对话框中单击 Cancel 按钮。事实上，在后面介绍的安装步骤中几乎都可以通过单击 Cancel 来取消 Delphi 的安装。

(2) **现在来到了 Delphi 6.0 安装的第一个画面，如图 1.3 所示。在这个对话框中不需要进行任何特殊的操作，只要单击 Next 按钮就可以了。**

(3) **在单击 Next 按钮之后，会显示如图 1.4 所示的对话框。在这个窗口中，要求输入你购买的 Delphi 6.0 软件的序列号。这个序列号可以在你的 Delphi 6.0 的光盘包装盒上或者其中的说明书中找到。**

(4) **在输入了正确的序列号之后，单击 Next 按钮，此时会显示如图 1.5 所示的对话框。这是一个软件许可证，或者叫协议书，当你在安装这个软件的时候同时表明你必须要遵守该协议书中的内容。选择 I accept the terms in license agreement，然后单击 Next 按钮，此时会显示另外一个安装信息对话框，在这个对话框中，显示了关于 Delphi 6.0 安装以及安装后可能会遇到的很多问题。对于那些对 Delphi 不是很熟悉的人来说，应该仔细的阅读这部分内容。在了解了这些问题之后，单击对话框中的 Next 按钮，此时会显示如图 1.6 所示的对话框。**

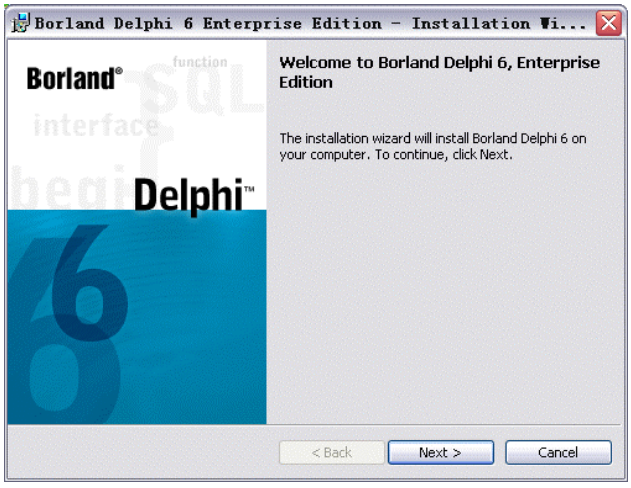

图 1.3　安装向导初始画面

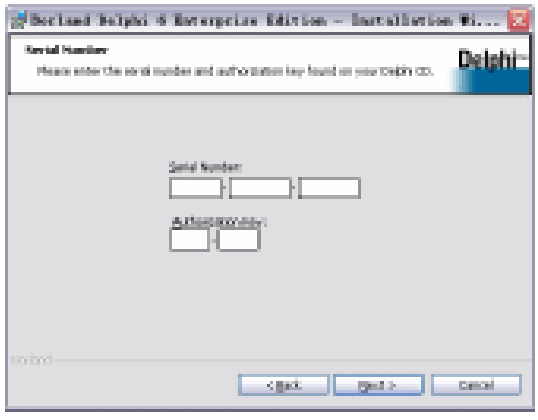

图 1.4　输入序列号的窗口

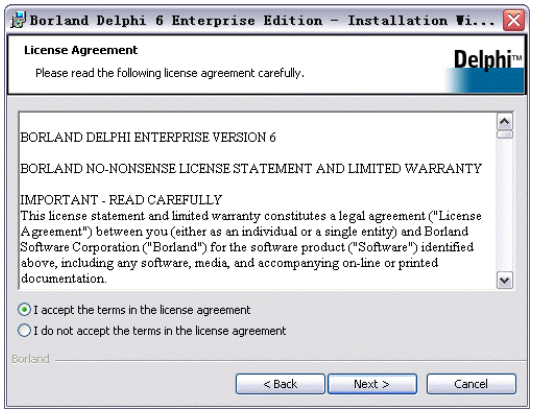

图 1.5　软件许可证

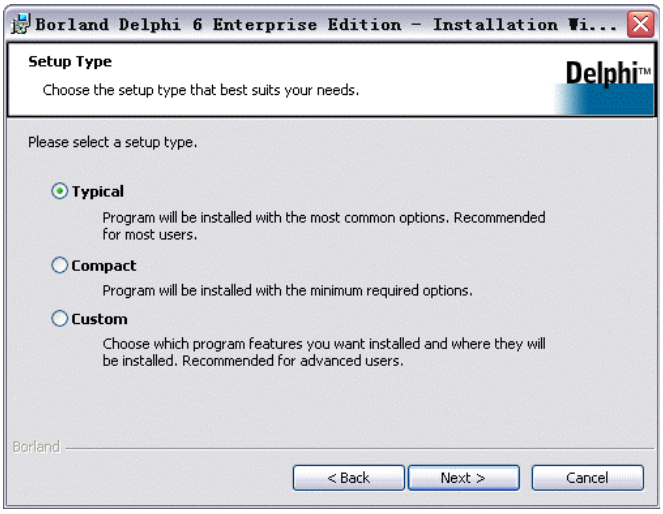

图 1.6　安装类型窗口

说明：

在安装软件的时候，通常会有如图 1.5 所示的这样一个协议书窗口。这时候我们没有其他的选择，如果要使用该软件，就必须同意协议书中的内容。

(5) **在图 1.6 所示的对话框中，要求我们确定希望的安装类型。这里有三个选项，依次是：**

- Typical(**典型安装**)　**就是按照 Borland 设计的大多数用户可能会采用的安装模式。**
- Compact(**紧缩安装**)　**也叫最小安装，它只安装运行 Delphi 6.0 所需要的最基本的组件。利用这种安装模式，可以节省磁盘空间。**
- Custom(**自定义安装**)　**对于高级用户来说，往往喜欢这种自定义模式的安装过程。因为可以根据自己的需要来确定安装或者不安装哪些内容。**

为了说明安装过程的各个细节，我们这里采用“自定义”安装的模式。选择 Custom，然后单击 Next 按钮，此时便进入 Delphi 6.0 安装向导的另外一个窗口，如图 1.7 所示。

(6) **在这个对话框中，列出了 Delphi 6.0 所包含的所有组件，在这里可以根据自己的需要来进行选择。双击对话框中的任何一个项目，可以展开或者收缩它所包含的子项目（如果它包含子项目的话）。单击任何一个项目左边的图标，会显示一个如图 1.8 所示的菜单。在这个菜单中提供了四个选项：**

- Install On Local Hard Drive(**在本地磁盘上安装该项目**)
- Install Entire Feature On Local Hard Drive(**在本地磁盘上完全安装该项目——包含子项目**)
- This feature will be installed to run from CD(**该项目将从光盘上运行**)
- This feature, and all subfeatures, will be installed to run from the CD(**该项目以及它的子项目将在光盘上运行**)

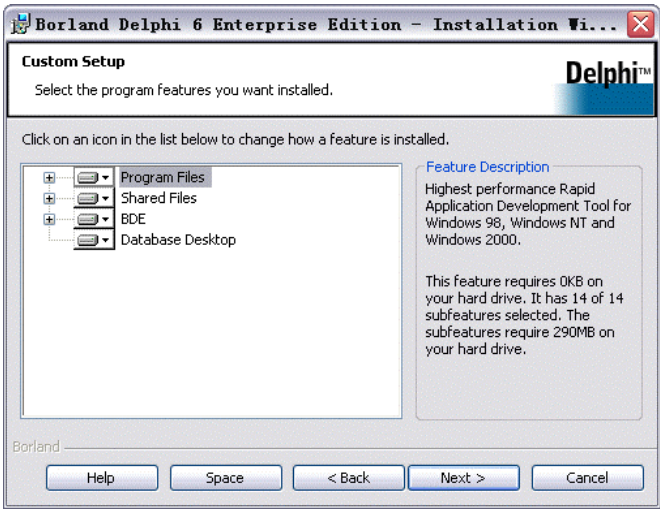

图 1.7　设置安装选项

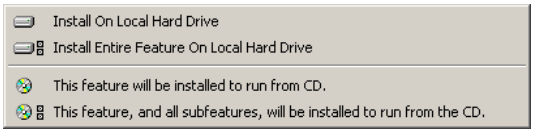

图 1.8　安装选项设置菜单

(7) **在根据自己的需要设置完了所有的项目之后，单击** Next **按钮，**Delphi 6.0 **安装向导会要求你确定要安装的数据库引擎，如图** 1.9 **所示。**

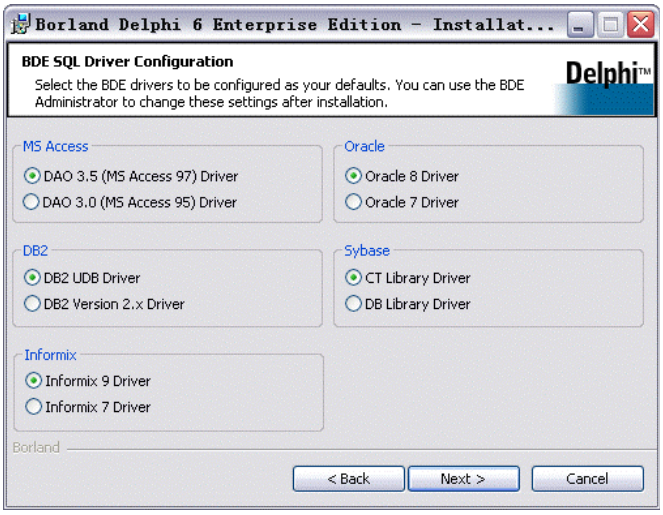

图 1.9　确定要安装的数据库引擎

(8) **在这个对话框中，提供了五种数据引擎的不同版本，通常来说我们应当选择高版本**

的驱动程序（一些特殊的情况除外）。

说明：

这是配置 BDE 数据库引擎的安装对话框。在这里可以看到，Delphi 6.0 中使用的是比 Delphi 5.0 中使用的更高的数据库引擎版本。

(9) 如果希望使用 Delphi 安装向导默认的数据库引擎，可以单击 Next 按钮，此时会显示如图 1.10 所示的对话框。

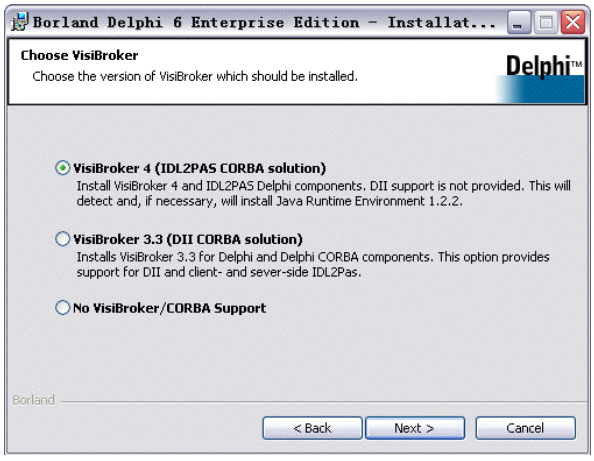

图 1.10　选择 VisiBroker 的安装版本

在以前的 Delphi 中，VisiBroker 通常是一个可选的组件，可以选择安装，也可以不安装。但是在 Delphi 6.0 中，它成了一个必须安装的程序。这里我们选择 VisiBroker 4。

(10) 单击 Next 按钮，此时会显示如图 1.11 所示的对话框。这要根据你计算机上安装的 Office 程序的版本而定。

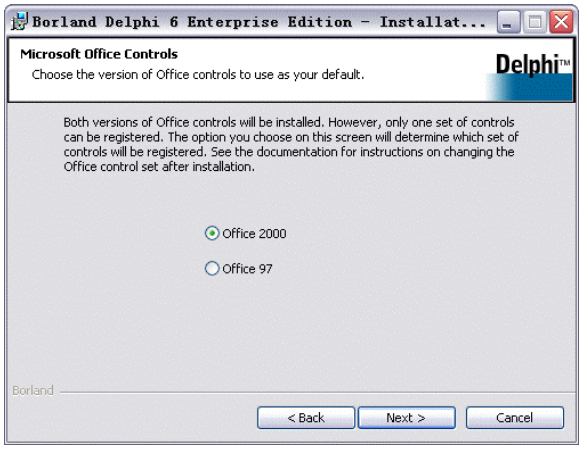

图 1.11　选择需要安装的 Office 组件

说明：

从 Delphi 5 开始，在 Delphi 中包含了一组关于 Microsoft Office 系列软件的组件。通过它们，我们可以自动化文档编辑、可以控制相应的 Office 程序的启动、编辑、关闭等。

(11) **在选择了合适的版本之后，单击 Next 按钮。此时会显示如图 1.12 所示的对话框。**

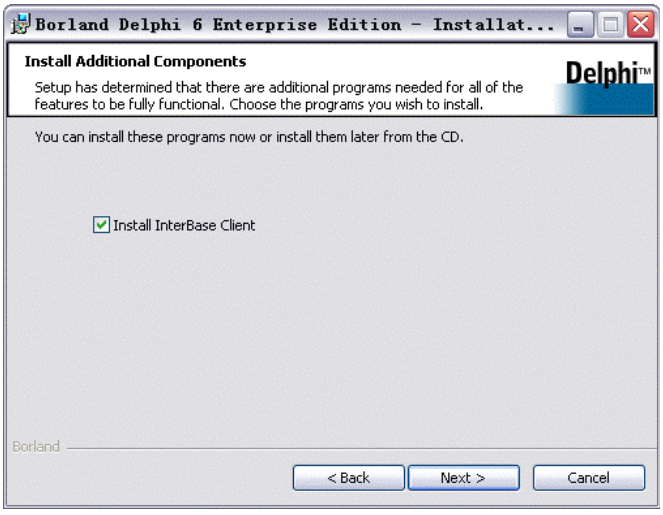

图 1.12　确定是否安装 Delphi 6.0 的附加组件

(12) **在这里可以选择是否安装 Delphi 6.0 的 Interbase Client 组件，这个程序可以很好地帮助你开发数据库应用程序。如果希望安装这个软件，可以选择该复选框，然后单击 Next，此时会显示如图 1.13 所示的对话框。**

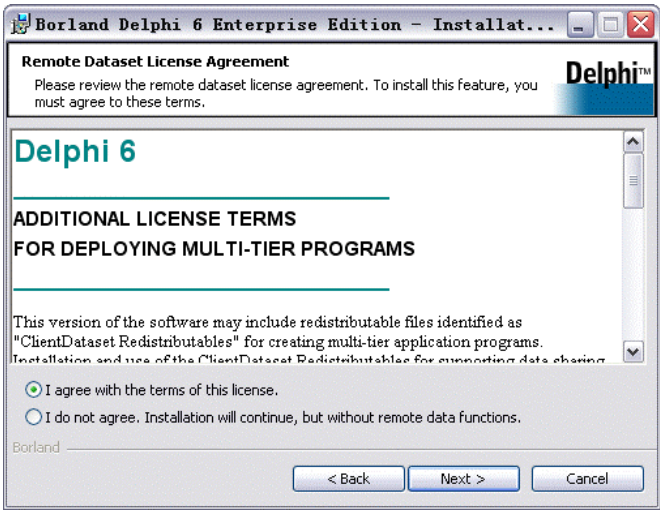

图 1.13　关于发布多层应用程序的协议书

(13) 在这个对话框中，Borland 的安装程序提供了它们关于用户在开发和发布多层应用程序的时候应该遵循的一些条款。在浏览了协议书中的内容之后，单击 Next 按钮，此时将进入下一个对话框，如图 1.14 所示。

说明：

现在版权的保护越来越受到人们的重视，所以我们在安装和使用软件的时候，也会遇到越来越多的许可协议。

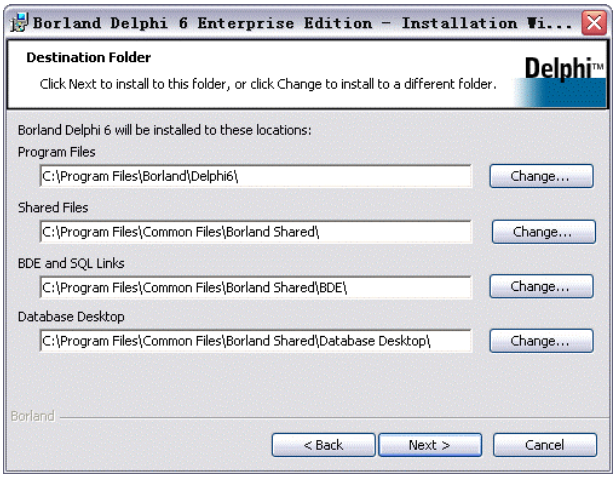

图 1.14　确定 Delphi 6.0 及其组件的安装位置

(14) 在图 1.14 所示的对话框中，显示了各个组件的安装位置，你可以根据自己的磁盘空间情况来确定安装位置。如果希望改变某个组件的安装位置，则可以单击它右边的 Change（改变）按钮，此时会显示如图 1.15 所示的对话框。

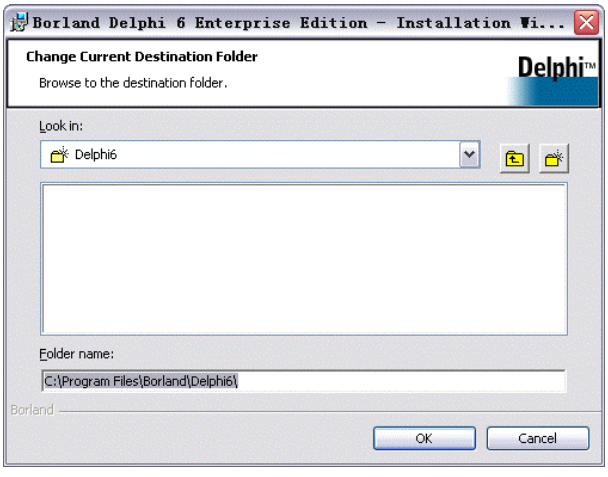

图 1.15　改变安装位置对话框

说明：

在这个对话框中，可以直接输入安装位置，比如“ D:\Delphi 6\BDE **”，也可以通过上面的浏览功能找到你需要的位置。**

(15) **在改变了相应的安装位置之后，单击** Ok **按钮，此时将回到图** 1.14 **所示的对话框。单击“** Next **”按钮，此时会显示如图** 1.16 **所示的对话框。**

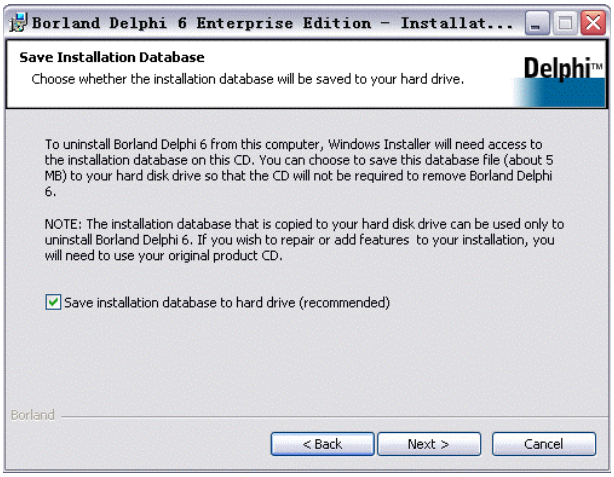

图 1.16 **把安装数据库保存到硬盘上**

(16) **在图** 1.16 **所示的对话框中提到是否保存安装数据库，它是用来卸载** Delphi 6.0 **的。这里我们选中该复选框，然后单击** Next **按钮，此时会显示一个对话框，对我们的安装选项进行确认，如果你希望修改安装配置，那么可以单击对话框中的** Back **按钮，重新进行配置；如果认为无需修改，可以单击对话框中的** Install **按钮，此时将开始安装** Delphi 6.0**，如图** 1.17 **所示。**

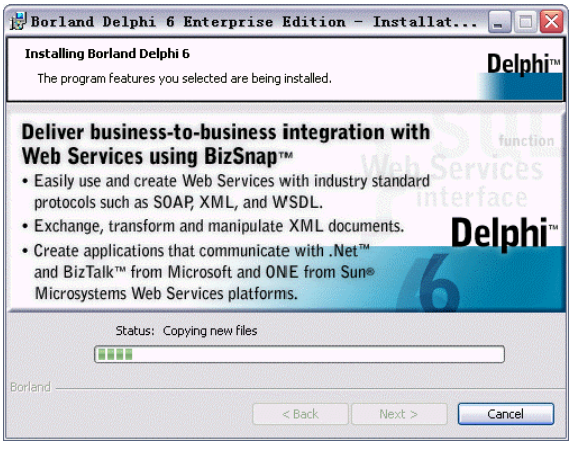

图 1.17 **开始安装** Delphi 6.0

在安装结束后，Delphi 6.0 安装向导还会根据你前面选择的组件情况进行其他的安装，比如 VisiBroker 4.0 和 InterBase Client。它们的安装过程同普通应用程序的安装过程类似，这里就不详细介绍了。需要的读者可以根据自己的情况参照安装向导说明进行操作。

在安装结束后，Delphi 6.0 的安装向导会提示你重新启动计算机。在重新启动了计算机之后，就可以运行 Delphi 6.0 了。

1.2　Delphi 6.0 界面概述

在完成 Delphi 的安装之后，就可以运行它了。选择【开始】|【程序】| Borland Delphi 6，然后选择 Delphi 6，便会运行该应用程序。Delphi 6.0 的集成界面如图 1.18 所示。

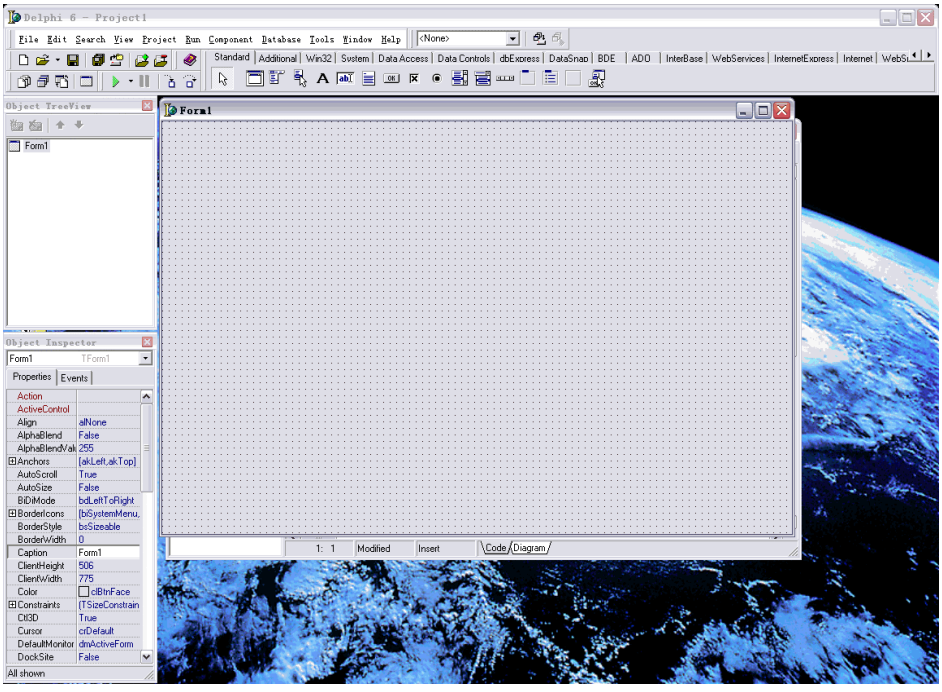

图 1.18　Delphi 6.0 的集成界面

可以看出，Delphi 6.0 共有 5 个窗口，上面有一个主窗口，在下面的工作区间里面有 4 个窗口。

- ❖ **主窗口**　在默认情况下，位于屏幕的顶部，上面包含了 Delphi 6.0 的菜单、工具栏和控件选项板。
- ❖ **Form 窗口**　该窗口占有大部分屏幕空间，是创建应用程序界面的基本空间。我们可以在这个窗口上放置各种控件，并设置它们的属性、安排它们的位置，从而完成

应用程序的界面设计。

- ❖ 对象浏览器　在默认情况下位于屏幕的左上方，在这个窗口中，会显示出当前活动 Form 窗口中包含的所有的对象。
- ❖ 对象属性浏览器　该窗口位于对象浏览器的下方，其中显示了当前选中对象的所有属性或者可以利用的事件。如果没有选中任何对象，那么显示的是当前 Form 窗口所代表的 Form 属性。
- ❖ 代码编辑器　该窗口在默认情况下隐藏在 Form 窗口的下面，是用来编辑 Delphi 代码的，通过这些代码可以实现应用程序中的功能。

1.2.1 主窗口

Delphi 的主窗口分为两个部分，第一部分是功能区，另一部分是控件面板区。

主窗口的一个重要的功能就是选择对应的组件，并把它放置在 Form 窗口上。这部分功能集中在控件面板区上，如图 1.19 所示。

图 1.19　控件面板

在这个控件面板上，分门别类地列出了 Delphi 的很多控件。可以说利用 Delphi 的控件面板所提供的控件，几乎可以设计各种形式的应用程序。例如，可以单击它们上面的任何一个，并在 Form 窗口上绘制一个矩形，这个矩形的大小代表了控件的大小，当你松开鼠标按键的时候，便把一个控件放到了 Form 窗口中。

说明：

在 Delphi 6.0 中，新增加了一些控件的种类以及新的控件，我们将在后面的内容中介绍这些控件的用法。

控件面板是一个控件的容器，当我们需要使用第三方开发的控件或者自己开发的控件的时候，需要先把它们安装到控件面板上。

功能区是用来完成关于 Delphi 工程的建立、打开、保存、编译、运行以及 Delphi 自身集成环境的设置等操作的。它是 Delphi 程序的代表，当关闭主窗口的时候，也就关闭了 Delphi。在功能区中，包含了菜单和工具栏，菜单包含的功能，我们将在后面使用到它们的时候分别介绍，下面我们来熟悉一下 Delphi 6.0 的工具栏，如图 1.20 所示。

图 1.20　Delphi 6.0 的工具栏

利用工具栏可以方便地实现 Delphi 工程的创建、打开、保存和编译。也就是说 Delphi 6.0

的工具栏上集成了在利用 Delphi 进行程序开发时最常用的功能。在该工具栏上按照从左到右，从上到下的顺序包含了以下按钮。

1. New（新建）按钮

单击该按钮，会显示 Delphi 6.0 的新建对话框，如图 1.21 所示。在该对话框中列出了 Delphi 可以建立的各种项目的种类以及各个项目。选择相应图标，并单击 OK 按钮，便建立了一个新的项目。

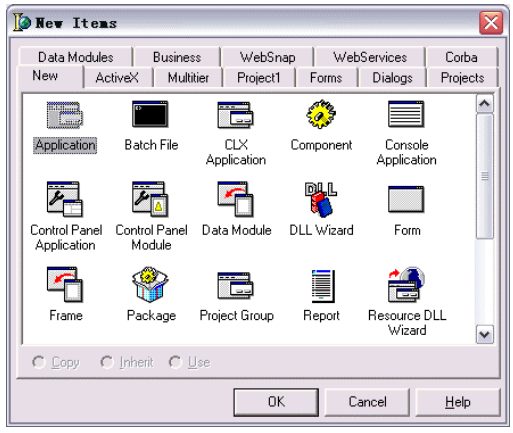

图 1.21　New Items 对话框

2. Open（打开）按钮

单击该按钮，将显示一个如图 1.22 所示的对话框，从中可以选择要打开的 Delphi 文件，包括 Delphi 的工程文件、单元文件、包文件等。你可以在【文件类型】列表框中选择需要的文件类型，然后找到需要的文件，并单击【打开】，该文件便在 Delphi 的代码编辑器中打开了。

说明：

在 Open 按钮的右边有一个箭头，单击该箭头，可以显示最近打开的各个文件。利用这个功能，可以快速地打开你最近打开的各个文件。这个功能和菜单中的 Reopen 功能十分类似。

3. Save（保存）按钮

单击该按钮，将保存当前编辑的文件。如果是第一次保存这个文件，Delphi 会显示一个 Save as 对话框，也就是我们熟悉的【另存为】对话框，让你指定文件保存的路径和名称。

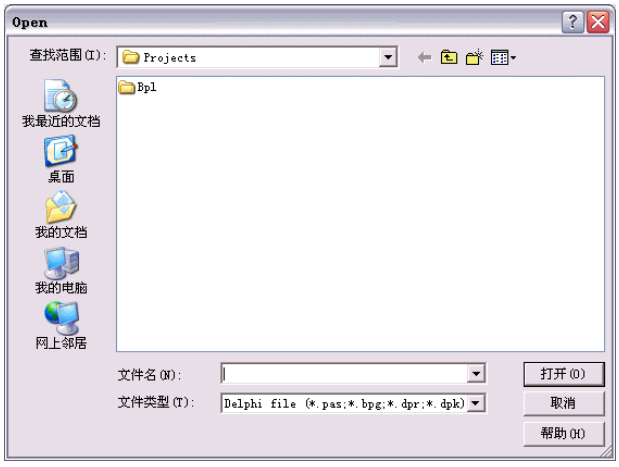

图 1.22 Open 对话框

4. Save all（保存所有文件）按钮

单击该按钮，将保存当前打开的所有文件。对于每个文件的保存和 Save 按钮类似。

5. Open Project（打开工程）按钮

单击该按钮，可以打开硬盘上的某个工程文件，它的作用和 Open 按钮类似，不同的是，Open 按钮可以打开各种 Delphi 文件，而该按钮主要是打开工程文件。

6. Add to Project（加入工程）按钮

在设计应用程序的时候，也许希望把设计好的一个单元文件加入到当前的工程中，比如设计了一个具有自己风格的【关于】窗口，那么我们可以使用这个按钮将该窗口加入到当前工程中。单击它会显示一个如图 1.23 所示的对话框，选择其中的单元文件，并单击【打开】，便把该文件加入到了当前的工程中。

说明：

“关于”窗口几乎在所有的应用程序中都有，而且成为一个应用程序的标志。所以开发应用程序的时候，也许已经设计好了这样的一个模板似的窗口，那么当需要的时候，只要把该窗口对应的单元文件加入到工程中就可以了。

7. Remove file from Project（从工程中删除文件）按钮

这个按钮是和上面按钮相对应的，它的功能是把某个文件从当前工程中删除。当单击该按钮的时候，会显示如图 1.24 所示对话框。该对话框列出了当前工程中包含的文件，选中某

个文件，单击 OK 按钮，该文件便从当前工程中删除了。

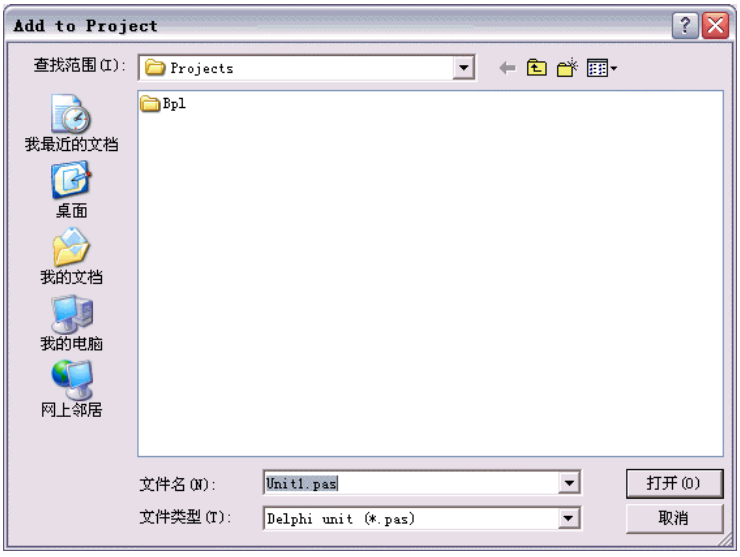

图 1.23　Add to Project 对话框

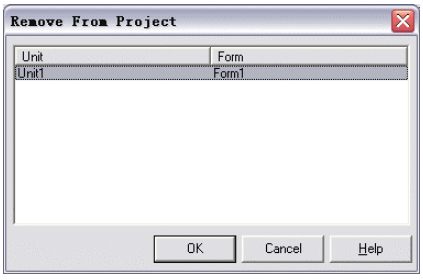

图 1.24　Remove from Project 对话框

说明：

所谓把文件从当前工程中删除，仅是删除了工程对该文件的拥有关系，而不是真正意义上删除了该文件。从当前工程中把某个文件删除了之后，该文件中所定义的对象、方法、数据等对于当前工程来说就是不可访问的了，除非专门在程序中指明对该文件的引用。

8. Help（帮助）按钮

单击它将会显示 Delphi 的帮助。

9. View Unit（查看单元文件）按钮

这是工具栏第二行第一个按钮。在设计应用程序时，可能会使用多个单元文件，有时候

可能需要查看某个文件的内容。当然可以利用 Open 按钮来实现，但是我们更可以利用这个 View Unit 按钮。单击该按钮，显示如图 1.25 所示的对话框。在这个对话框中，列出了当前工程中包含的所有单元文件，也包括当前工程的工程文件。选择其中的某个文件，并单击 OK 按钮，便在代码编辑器中打开了该文件。

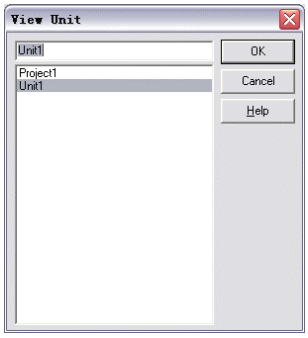

图 1.25 View Unit 对话框

说明：

一个工程也有一个工程文件，工程文件是一个标准的 Delphi 单元文件，我们也可以编辑它。

10. View Form（查看窗口）按钮

这个按钮的功能和查看单元文件按钮的功能类似，不过查看的对象是当前工程中的窗口。

11. Toggle Form/Unit（窗口和代码切换）按钮

在编辑一个窗口上的控件时，可能需要切换到代码编辑器窗口来编辑对应的代码，或者在编辑完代码之后，需要切换到该代码对应的 Form 窗口来重新组织控件的布局，这时可以利用该按钮。

12. Run（运行）按钮

单击该按钮可以运行当前编辑的工程。当单击该按钮时，如果需要，Delphi 将先编译当前工程，生成一个可执行文件，然后运行它。

13. Pause（暂停）

只有在运行某个工程的时候，该按钮才可用。单击该按钮，可以暂停运行当前程序。

14. Trace Into（单步运行）按钮

该按钮是一个程序调试按钮，当单击该按钮时，Delphi 将控制应用程序按照代码一行一行地运行，当到达某一使用了方法或者函数的行时，单击该按钮将进入到这一方法或者函数定义的代码中，并单行执行。

15. Trace Over（追踪）按钮

该按钮的功能和 Trace Into 类似，不同的是当到达某一使用了方法或者函数的行的时候，单击该按钮，将把该行中的函数或者方法当做一个完整的代码一次执行，也就是说它不会进入该函数或者方法的定义代码中。

主窗口是 Delphi 的主要操作区域，利用上面介绍的工具栏中按钮，基本上可以满足 Delphi 编程时的需要。当涉及到其他内容时，我们会进行相应的介绍。

1.2.2　Form 窗口

Form 窗口是主要的设计空间之一，在这个窗口上，将定义我们所设计的应用程序的界面。Delphi 是一个所见即所得的可视化设计软件，也就是说在该窗口上设计的内容，将同样地显示在我们的应用程序中。

需要说明的是，一般来说，一个单元文件对应一个（至多对应一个）Form 窗口，而一个工程中可以包含许多 Form 窗口。我们需要对每个 Form 窗口进行设计，并设计对应的单元文件。

我们在 Form 窗口上做的最多的工作是放置控件，调整控件的位置，以及调整窗口本身的位置和大小等。通常的操作顺序是在 Form 窗口上放置一个控件，调整它的大小和位置，然后到对象属性浏览器中设置对应的属性，然后在代码编辑器中编辑对应的代码。

说明：

在 Delphi 中，几乎可以这样说，程序就是窗口，窗口就是程序。因为几乎所有的程序都会包含一个对应的窗口，虽然这个窗口的显示形式有很多。特别是在一个程序中，必定会包含一个主窗口，这个主窗口更是和程序等价的。在运行应用程序的时候，如果关闭了主窗口，也就是关闭了应用程序。这个特点和 Delphi 6.0 本身的集成环境很类似。

1.2.3　代码编辑器

代码编辑器是 Delphi 包含的智能编辑器。可以在这里快速地设计程序代码。

代码编辑器包括三个部分，如图 1.26 所示。

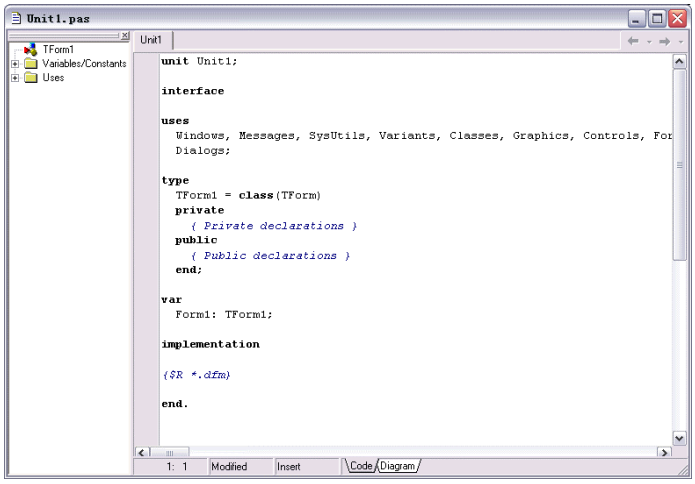

图 1.26　代码编辑器

这个窗口的主要部分是用来编辑代码的，实际上这就是一个功能比较齐全，适合于编辑 Delphi 代码的文本编辑器。在代码编辑器中，Delphi 有一个十分重要的功能，就是著名的 Code Insight，我们通常称之为属性帮助器。它的使用非常简单，例如要使用 Form 的宽度属性，那么通常在代码编辑器中这么书写：

```
Form1.Width := 150;
```

在这里需要输入这个属性 Width，但是在 Delphi 中，可以输入 Form1.，然后稍微停顿一下，此时代码编辑器会显示一个窗口，里面包含了 Form1 的所有属性或者方法，此时再输入字母 W，会自动移动到 Width 属性，如图 1.27 所示。只要按下空格键，该属性便自动输入到代码编辑器中。

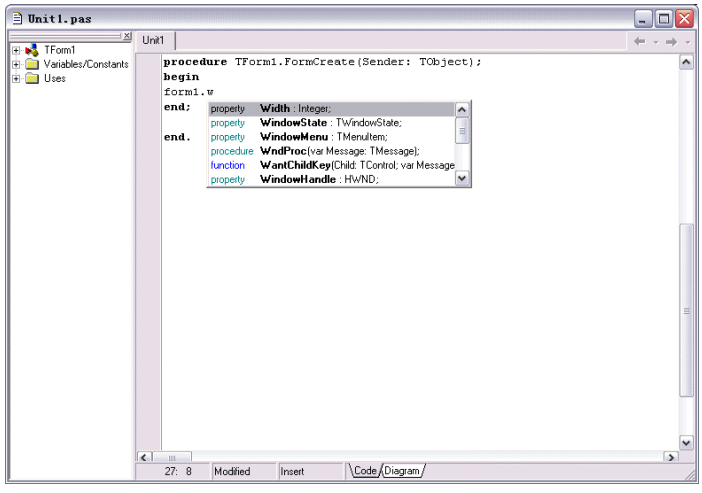

图 1.27　代码编辑器中的 Code Insight 功能

说明：

在按下属性或者方法的首字母后，如果没有显示需要的属性或者方法，可以单击 Code Insight 窗口中的滚动条或者利用上下键来寻找自己需要的属性和方法。

这个功能也适用于其他所有的控件或者组件。也许我们这里举的例子还不够复杂，对于一些名称特别长的属性，这个功能非常方便。在 Delphi 中，每个控件或者对象都有非常多的属性，有一些属性的名称还比较长，那么在编写代码的时候要记住这些属性或者方法的名称是很困难的，实际上我们也不需要记住它们，只要大体上知道它们的功能，以及它们的第一个字母就可以了。

在默认情况下，代码编辑器底部是一个信息窗口，主要是显示当前工程在编译的时候的错误、警告或者其他信息。比如在利用 Search（搜索）菜单中的 Find（查找）功能的时候，查找的结果便显示在该信息窗口中。

代码编辑器的左边是一个代码浏览器，通过这个窗口，可以迅速地找到需要的变量、方法、对象或者引用的模块，展开的代码浏览器如图 1.28 所示。

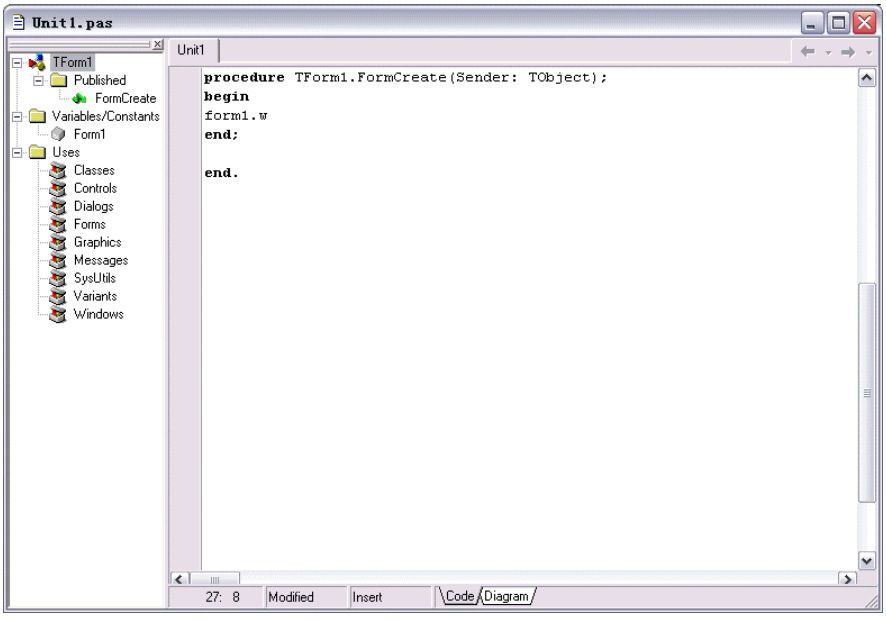

图 1.28　左边的是代码浏览器

在代码编辑器底部有两个选项卡，一个是 Code（代码窗口），就是上面介绍的代码编辑器的主要窗口；另一个是 Diagram，这是 Delphi 6.0 中新增加的一个视图，我们通常称之为图表视图。这是为了更方便我们在设计一些相关控件之间的关系时使用的。典型的例子就是数据库访问控件，我们总是通过将它们的一些相关属性联系在一起，从而建立起它们的相对关系。现在 Delphi 提供了一个图表视图，利用它，就像绘图一样，很容易地建立起它们

的关系。

下面首先介绍一些它的基本特点，然后通过一个示例来演示它的用途。该图表视图如图 1.29 所示。

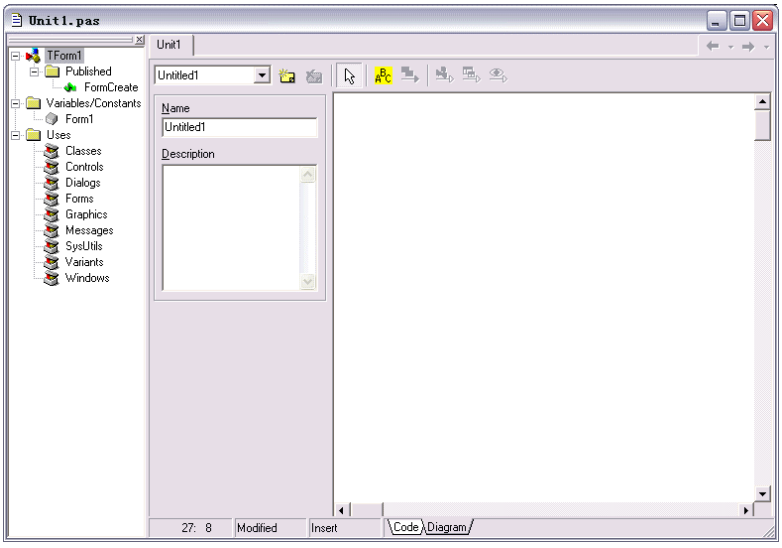

图 1.29 Delphi 6.0 的 Diagram 视图

在 Delphi 6.0 中，有一个对象浏览器，它用树的形式把窗体上的控件显示出来。关于这个对象浏览器将在后面的内容进行详细介绍。图表视图页的作用，就是提供一个可视化工具来组织这些控件。也可以为这些控件添加注释，创建或者打印文档。

为了让控件显示在图表视图上，可以将其从对象浏览器中拖过来。简单地拖动控件到图表视图，控件将会垂直排列。按住 Shift 键拖动，控件将会水平排列。也可以通过拖拽操作重新排列图表视图上的控件。如你在 Form 窗口以及我们后面要介绍的对象属性浏览器中配置了相关控件属性，当控件被放置在图表视图上时，自动会有一些线段连接它们，用箭头来表示它们之间的关系。当然，也可以自己添加这些连接符号。

在 Delphi 6.0 中，控件之间的关系有五种：

- 暗指 (箭头)
- 属性 (带有指向对象箭头的线段)
- 主从 (两端带有不对称“鼓状”图标的线段)
- 查看 (末端带有“眼睛”图标的线段)
- 父子 (带有空心箭头的线段，指向父对象)

这些关系分别对应图表视图右上方的五个按钮。也就是说，在设计的时候，可以通过这些按钮，为控件指定上述关系。

在下面，我们通过一个简单的示例来演示图表视图是如何工作的。

(1) 首先运行 Delphi，此时 Delphi 会自动新建一个应用程序，该程序包含一个 Form 窗口。

(2) 单击主窗口上的控件面板上的标签控件，然后在窗口上单击，或者在窗口上按下鼠标并拖动画出一个矩形，此时会在 Form 窗口上添加一个标签控件，即 Label 控件。

(3) 在对象属性浏览器中，把 Label 控件的 Caption 属性修改为&Input the Text。

提示：

“ & ”字符在 Delphi 中是一个特殊字符，该字符后面的字符将被定义成该控件的热键。比如在这里，I 键便被定义成 Label 控件的热键。当用户按下 Alt+I 组合键时，程序的焦点便转移到标签控件上。

(4) 利用同样的方法在 Form 窗口上添加一个文本框控件，此时的 Form 窗口如图 1.30 所示。

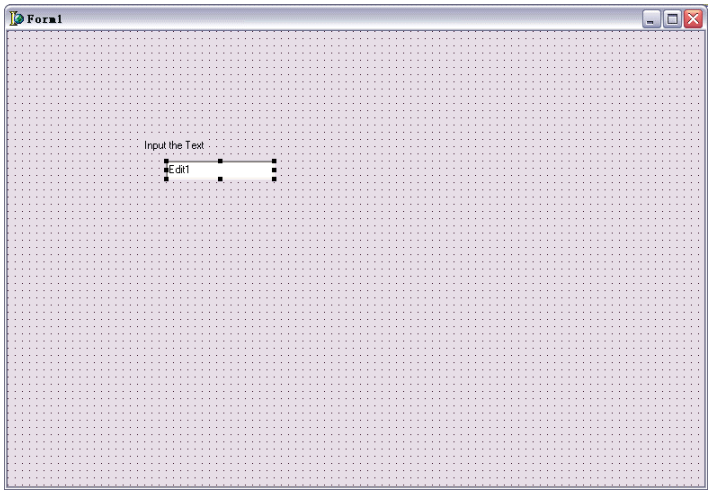

图 1.30　添加了标签和文本框的 Form 窗口

(5) 然后利用工具栏上的“窗口和代码切换”按钮切换到代码编辑器，并单击底部的 Diagram 选项卡，切换到图表视图，把对象浏览器中的 Edit1 和 Label1 控件拖动到图表视图中。

(6) 单击 Property Connector 按钮。这个按钮可以利用把鼠标移动到窗口中的各个图标上来寻找 Delphi 提供的提示来查找，把鼠标放置在 Label1 上，并拖动鼠标，连接到 Edit1 上。此时的图表视图如图 1.31 所示。

(7) 下面来看看上面建立的控件关系有什么作用。单击主窗口上 Run 按钮，运行当前应用程序，如图 1.32 所示。按下 Alt+I 组合键，此时会发现程序的焦点自动移动到文本框中。

如果利用 Delphi 中的对象属性浏览器来查看 Label1 控件属性，你会发现，它的 FocusControl 属性已经变成了 Edit1。这就是图表视图的作用。也许在这个例子中看起来作用还没有那么大，当要使用多个控件相互关联的时候，你会发现该功能实在是 Delphi 6.0 的一

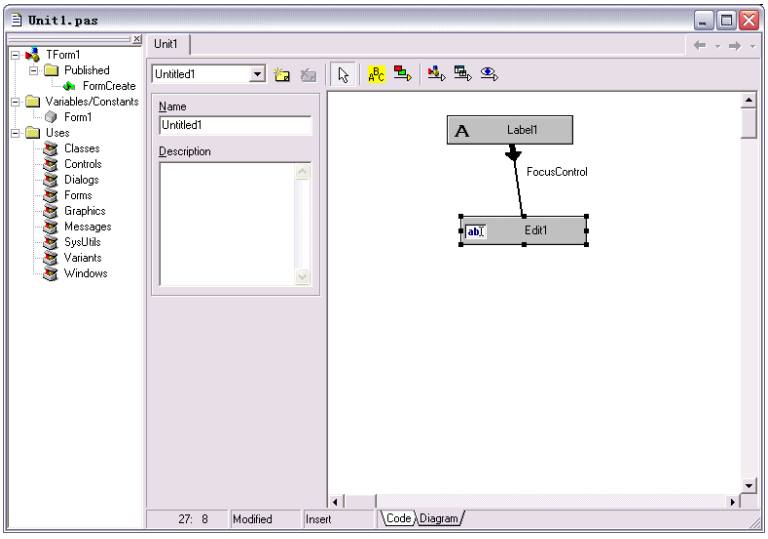

图 1.31　放置在图表视图中的控件关系图

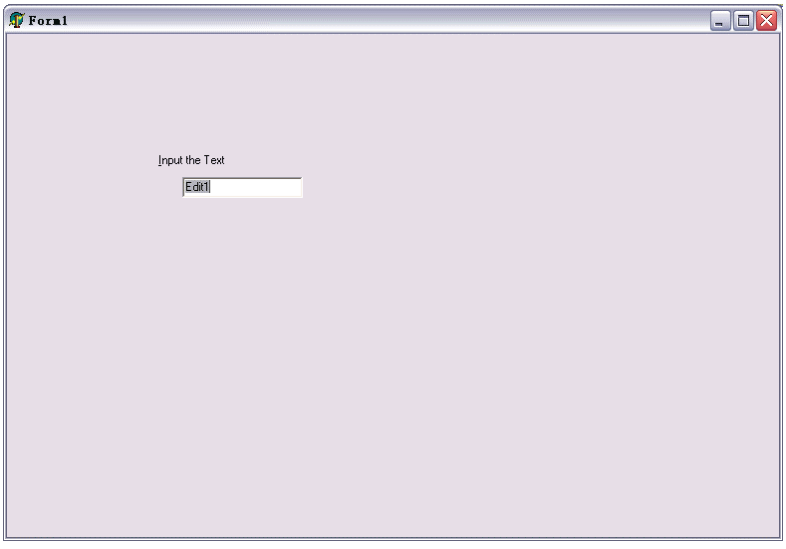

图 1.32　Alt+I 组合键将输入焦点移动到文本框中

个重要改进。

1.2.4　对象浏览器

Delphi 6.0 **的对象浏览器如图** 1.33 **所示，这是** Delphi 6.0 **新增加的一个窗口。**

在这个窗口中，可以进行控件和对象的复制、删除、新建等操作，但是笔者认为这个窗口更多是为了图表视图建立的。

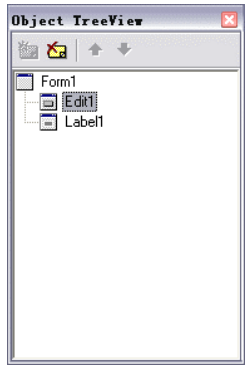

图 1.33　对象浏览器

1.2.5　对象属性浏览器

Delphi 一直包含这样一个窗口，让用户用来编辑控件或者对象的属性以及事件。该窗口如图 1.34 所示。

通过该窗口，可以完成下面三项任务。

- 设置放在 Form 窗口上控件的属性。
- 帮助创建对象或者控件的事件处理程序或者在不同的事件处理程序中切换。
- 过滤可以看到的属性或者事件。

例如在上面的例子中，要改变 Label1 控件的 Caption 属性，那么可以在这个窗口中浏览，找到 Label1 控件的 Caption 属性，并在右边一栏中单击，此时就可以在这个框中输入 Caption 属性的内容了。

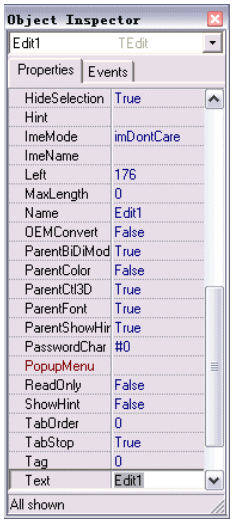

图 1.34　对象属性浏览器窗口

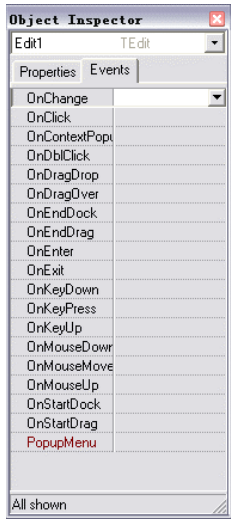

图 1.35　对象属性浏览器中的事件选项卡

对象属性浏览器窗口具有两个选项卡，一个是 Property，它具有我们上面介绍的功能；另一个是 Events，也就是用来管理对象或者控件的事件的，单击该选项卡，对象浏览器窗口如图 1.35 所示。

在某个事件上双击，可以切换到代码编辑器，并自动生成该事件处理程序的框架，如图 1.36 所示。我们需要做的就是在这个框架中填入对应的代码。

对象属性浏览器的另外一个功能是过滤我们所看到的属性或者事件，这个功能通过该窗口上的快捷菜单来实现。

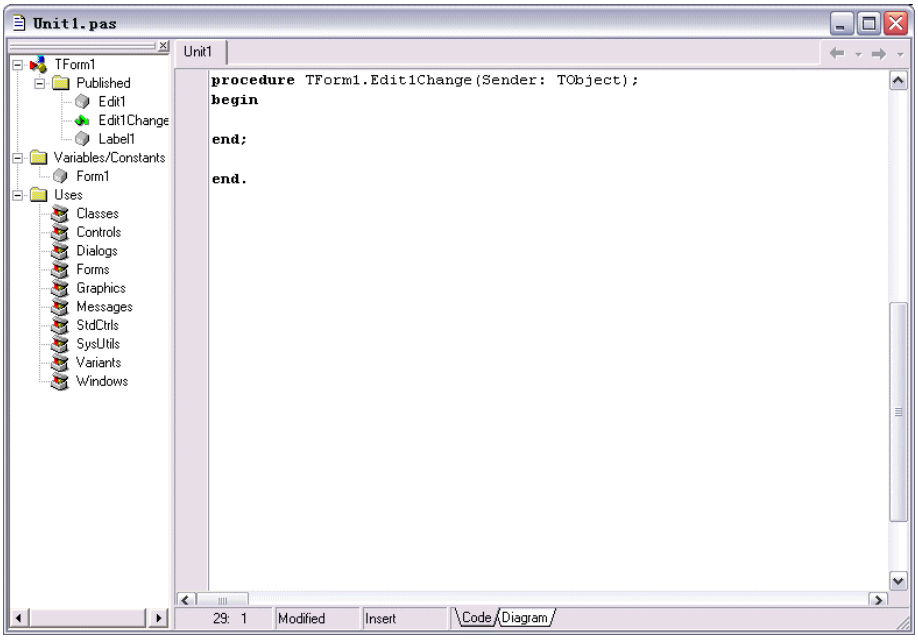

图 1.36　双击 Label1 控件的 OnClick 事件后的代码编辑器窗口

下面介绍一个 Delphi 6.0 中的对象属性浏览器同以往版本不同的地方。从上面的图 1.34 和图 1.35 中都看到了有红色的项目，例如在图 1.34 中，可以看到 FocusControl 属性就是红色的，并且属于有附加属性的属性。单击该属性左边的加号，可以展开该属性，此时会发现 FocusControl 所对应的 Edit1 控件的属性也显示在了该对象浏览器中，如图 1.37 所示。

这是以前的 Delphi 中不能实现的。在以前的 Delphi 中，在对象属性浏览器中，在某个时刻只能显示某个对象或者控件的属性或者事件。这是一个非常重要的改进。由于这个功能，就可以一次设置好相关的各个控件的属性，不需要在多个控件之间来回切换了。

说明：

对象属性浏览器中切换控件有两个方法。一个是通过该窗口顶部的下拉列表，从中可以选择需要的控件或者对象；另一个方法是在 Form 窗口中单击要设置属性的对象或者控件。

从图 1.37 中，可以看到，嵌套对象的属性是用绿色表示的。如果对这里的颜色搭配不满意，可以在对象属性浏览器中右击鼠标，从弹出的快捷菜单中选择 Properties，此时将显示如图 1.38 所示的对话框。

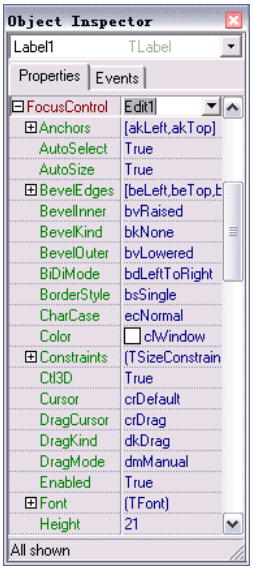

图 1.37　对象属性浏览器中的嵌套

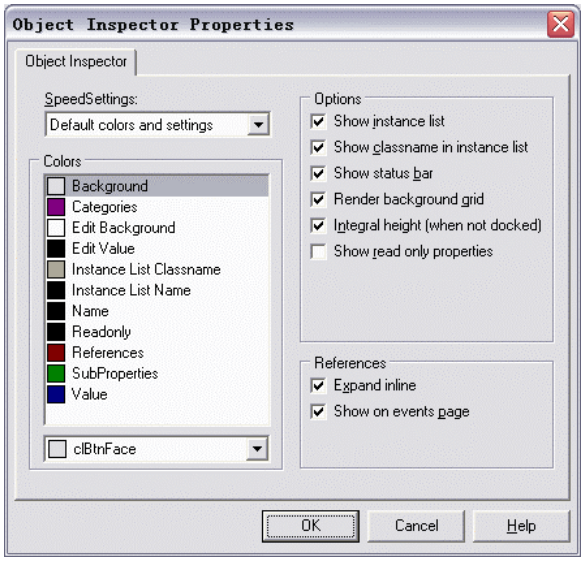

图 1.38　对象属性浏览器的属性

通过这个窗口，可以定义对象属性浏览器中的一些颜色搭配以及其他的一些属性。

说明：

实际上，Delphi 提供了相当强大的集成环境自定义功能。像对象属性浏览器中的自定义窗口，在 Delphi 6.0 中几乎每个窗口中都具有。特别是代码编辑器中，也可以通过 Delphi 6.0 的 Tools（工具）菜单中的 Environments Options（环境选项）和 Editor Options（编辑器选项）等命令来进行更为详尽的设置。但是我们不提倡用户自己去定义这些内容，除非认为自己特别需要。因为我们学习 Delphi 的目的不是研究它的环境，而是利用它来进行编程，开发出自己所希望的应用程序。

在对象属性浏览器窗口的 Events 选项卡中，同样存在着嵌套的内容，比如在图 1.35 所示的图中，可以通过单击红色项目左边的加号，展开对应的内容，这里就是 FocusControl（Edit1）对象的事件。

通过上面的内容，我们对 Delphi 6.0 的集成环境有了一定的了解。但是还不能说自己已经熟悉了 Delphi 6.0 的集成环境，因为这个集成环境包含了太多的内容。如果要面面俱到地介绍它所有内容，我们相信编写一本专门的书籍也是完全可能的。

1.3 本 章 小 结

在这一章中，我们详细地介绍了 Delphi 6.0 的安装过程，虽然它并不复杂。这么做的目的是帮助那些没有安装过 Delphi 的人，因为它毕竟比普通软件的安装具有更多选项。对于那些非常熟悉 Delphi 安装的人，这一部分内容完全可以跳过。

然后介绍了 Delphi 6.0 集成环境各个组成部分的主要功能。通过这些介绍，你对操作 Delphi 6.0 已经具有一定的基础了。这样你在阅读后面的内容，当遇到我们要进行某项操作的时候，你就知道应该在哪个部分进行怎样的操作了。

但是对于一个没有接触过 Delphi 的人来说，到目前为止，他还不知道什么是 Delphi，我们要用它来干什么？我们怎样来实现这个目标？即使对于那些对 Delphi 有一定的了解的人来说，如果要问什么是 Delphi，你是怎么理解 Delphi 中的程序结构的？也许他也不会有明确的认识。那么请你阅读本书的下一章。

第 2 章　Object Pascal 语法

在开始介绍使用 Delphi 进行编程之前，先介绍一下 Object Pascal 语言的基本语法。它们是编程的基本工具，如果不知道其中的基本规则，那么，在介绍编程的时候，读者可能无法了解我们所写的语句。

但是 Object Pascal 语言是一种比较复杂的语言，用一章的内容无法把它们全面详细地介绍完毕，所以，我们只是对它进行综述，选择其中最基本的和最关键的内容进行介绍。关于其他内容，在后面使用到的时候，再进行专门的介绍。

在本章将主要介绍：

- 特殊字符
- 数据类型
- 变量和常量
- 语句、过程和函数

2.1　Delphi 中的特殊字符

在 Delphi 中，存在一些特殊的字符，它们作为 Object Pascal 语言的保留字符出现在程序中。这样的字符有很多，我们通常把它们分成注释、保留字以及其他一些特殊字符。

2.1.1　注释

几乎在所有的高级编程语言中，都有用于添加注释的语句。而注释可以增加程序的可读性，所以一个优秀的程序员应当养成注释的习惯。需要注意的是，注释不是可以执行的代码，Delphi 在编译程序的时候，将自动跳过注释行的语句。

在 Object Pascal 语言中，注释用一对大括号{}或一对（**）括起用来作注释文字，如果注释只有一行，也可以用//表示注释开始，这一点和 C 语言中的写法十分类似。例如在下面的例子中，就使用了这三种形式的注释：

```
TForm1 = class(TForm)
private
   （＊ Private declarations ＊）
public
  Flag:integer;//用来描述程序标志
```

```
    { Public declarations }
end;
```

注意：

注释的内容中不能包含“}”或“*)”等字符，因为编译器把它们作为注释结束符号，注释也不能嵌套。

但是有一种特殊情况，如果“{”或“(*”后紧跟着一个$符号（不能有空格），编译器把它作为编译指令，而不是注释。可以观察一下 Delphi 当前单元文件，其中肯定存在下面的语句：

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

2.1.2 保留字

几乎在所有的编程语言中，都存在保留字。通常来说，所谓保留字，就是某种编程语言中具有特殊和固定意义的单词，如 For、And、While、With 等。对于这些单词，Object Pascal 语言中不允许用作其他用途，比如把它们说明为变量。

在 Delphi 6 中，代码编辑器加粗显示保留字，以区别于其他语法成分。在表 2.1 中，列出了 Delphi 中的绝大多数保留字。读者可以大体上浏览一下它们的样子，以免在编程的时候出现使用保留字的错误。

表 2.1 Delphi 中的保留字

And	Exports	Mod	Str
Array	File	Nil	String
As	Finalization	Not	Stringresource
Asm	Finally	Object	Then
Begin	For	Of	Threadvar
Case	Function	Or	To
Class	Goto	Out	Try
Const	If	Packed	Type
Constructor	Implementation	Procedure	Unit
Destructor	In	Program	Until
Dispinterface	Inherited	Property	Uses
Div	Initialization	Raise	Var
Do	Inline	Record	While
Downto	Interface	Repeat	With
Else	Is	Resourcestring	Xor
End	Label	Set	
Except	Library	Shl	

2.1.3　其他特殊字符

在下面的内容中，将介绍一些 Delphi 中的特殊字符，虽然它们也有自己的名称，但是作为使用者来说，更关心的是不能在程序中用它们来作为变量名称或者其他元素名称。所以我们统统称它们为特殊字符。

其中的一类字符被称为指示字。所谓指示字，通常用在声明变量、过程、函数和输出例程时指定附加的属性。总的来说，它们与保留字是十分类似的，有一个重要的区别，那就是，它们可以重新定义，虽然我们不建议用户这么做。表 2.2 列出了 Delphi 中绝大多数指示字。

表 2.2　Delphi 中的指示字

Absolute	External	Pascal	Safecall
Abstract	Far	Private	Stdcall
Assembler	Forward	Protected	Stored
Automated	Index	Public	Virtual
Cdecl	Message	Published	Write
Default	Name	Read	Writeonly
Dispid	Near	Readonly	
Dynamic	Nodefault	Register	
Export	Override	Resident	

除了上面提到的指示字之外，还有一类字符，通常用来作为运算符号。在表 2.3 中列出了 Delphi 中的几种运算符。我们在研究运算符的时候，要注意它们的优先级的问题，也就是说，当这些运算符集中在一起的时候，哪些会最先执行，哪些最后执行。在表 2.3 中，按照它们的优先级从高到低的顺序进行了排列。

表 2.3　Delphi 中的运算符

级别	运算符
1	. ^
2	@ Not
3	* / Div Mod As And Shl Shr
4	+ – Or Xor
5	= <> < > <= >= In Is

下面介绍一下它们的含义。

1. 对象运算符

“ . ”符号的主要作用就是表示它后面跟的元素（可能是数据，也可能是方法）是隶属于

它前面那个对象的。比如在程序中可能会出现这样的语句：

```
form1.Caption := 'Trying';
```

其中的 Form1 就是一个对象，它后面的 Caption 是该对象的一个属性。再如下面的语句：

```
form1.ShowModal;
```

“.”后面的 ShowModal 是隶属于 Form1 对象的一个方法。

2. 指针动态变量运算符

在 Delphi 中，用于指针运算的字符有两个，一个是“^”，一个是“@”。其中“^”用于获得指针变量的动态变量 ，例如，假设 Pointer 是一个指针类型的变量，Pointer^就是这个指针变量的动态变量，代表指针所指对象的值。

@运算符用于获得操作数的地址，相当于返回一个指向该操作数的指针，操作数可以是变量、过程、函数或方法。如果@运算符的操作数是变量，那么@运算符返回的是指向这个变量的指针，该指针的类型由编译指令{$T}来决定。如果是在{$T}状态编译的，指针的类型就是 Pointer，也就是无类型指针，这种指针类型与任何其他指针类型兼容。如果是在{$T+}状态下编译的，返回的指针只与指向相同变量类型的指针兼容。如果@运算符的操作数是过程或函数的数值参数，那么@运算符返回的是形参在栈中的位置。

3. 位运算符

位运算符的操作数只能是整数，结果也是整数。它们的含义如表 2.4 所示。

表 2.4 Delphi 中的位运算符

运算符	说明
Not	按位非，也就是把操作数的每一位取反，1 变成 0，0 变成 1
And	按位与，当两个位都为 1 时，结果的这一位为 1
Or	按位或，当两个位中有一个是 1 时，结果的这一位为 1
Xor	按位异或，当两个位相异时，结果的这一位为 1
Shl	左移，相当于 E1 乘以 2 的 E2 次方，其中 E1 为左边的操作数，E2 为右边的操作数
Shr	右移，相当于 E1 除以 2 的 E2 次方的整数部分

4. 算术运算符

算术运算符是在程序计算中最经常使用的字符，它们主要用于进行加、减、乘、除、取模等运算，其含义如表 2.5 所示。

表 2.5　Delphi 中的算术运算符

运算符	说明
+	两数相加
–	两数相减
*	两数相乘
/	两数相除，不管操作是整数还是实数，操作结果都是实数
Div	两整数相除，结果是商的整数部分
Mod	取模，两个操作数必须是整数，操作结果是两数相除的余数

5. 逻辑运算符

逻辑运算的操作数是逻辑表达式，结果是逻辑值。可以这么认为，逻辑运算符构成了一个复合的布尔表达式。

Object Pascal 语言中有一个编译开关{$B}，默认是{$B-}状态，表示当复合的布尔表达式的值已经很明显的时候，就不再计算每一个操作数的值。当其中一个操作数包含有函数调用时，这种方式能节约时间和代码长度。

Delphi 中的逻辑运算符的含义如表 2.6 所示。

表 2.6　Delphi 中的逻辑运算符

运算符	说明
Not	逻辑非，把 True 变成 False，把 False 变成 True
And	逻辑与，相当于汉语中的“并且”的意思
Or	逻辑或，相当于汉语的“或者”的意思
Xor	逻辑异或，当两边的操作数一个为 True 另一个为 False 时结果为 True，当两边的操作数都为 True 或都为 False 时结果为 False

6. 类型识别运算符

类型识别运算符 Is 用于判断某对象是否属于指定的类，其结果是一个逻辑值。当对象属于指定的类或该类的派生类时，Is 的结果就是 True，反之就是 False。Is 运算符实际上也是一种关系运算符，因此 Is 运算符通常用在 If 语句中。例如在下面的例子中就使用了一个 Is 运算符：

```
procedure TForm1.FormCreate(Sender: TObject);
begin
if Sender is TForm then
        ...
end;
```

上面例子的含义是说，如果该方法中包含的参数 Sender 代表的是一个 TForm 类，那么将执行下面的操作，否则将不进行该段程序的操作。

7. 类型强制转换符

在编程的过程中，经常会使用到类型强制转换符。比如，可能会使用下面的语句：

```
(Sender as Tform).Color := clBlue;
```

其中的 As 就是类型强制转换符。需要注意的是，如果把两种不兼容的类型进行强制转换，可能会引发异常。

8. 集合运算符

集合运算符+、－、*用于对集合类型的数进行操作，下面举例说明这些运算符的用法，假设有两个集合 A 和 B，它们的运算关系及其含义如表 2.7 所示。

表 2.7 Delphi 中的集合运算符

运算关系	含义
A＋B	其结果是属于集合 A 或集合 B 元素组成的集合（并集）
A－B	其结果是集合 A 中不属于集合 B 的元素组成的集合
A*B	其结果是同时属于集合 A 和集合 B 元素组成的集合（交集）
In	用于判断某个元素是否在集合中，其结果是布尔值

9. 字符串运算符

这里介绍一个常用的字符串运算符，就是“ + ”。两个字符串相加，就是把第二个字符串连接到第一个字符串后面，其结果仍然是一个字符串。如果两个操作数都是短字符串，相加后的结果如果超过了 255 个字符，超过的部分将被截掉。如果两个操作数中有一个是长字符串，结果也是长字符串。

说明：

事实上，在程序中使用更多的是一些字符串处理函数，在 Delphi 中我们更多地把它们称为方法。比如字符串的相互复制、相互比较等等。这些内容将在后面的章节中进行详细的介绍。

2.2 Delphi 中的数据类型

在任何编程语言中，数据类型都是应该掌握的一个重要的语法内容。实际上，从编程的

角度来说，要处理的不过两个问题：一个是程序中的数据，另一个是如何按照我们的目的组织这些数据。Object Pascal 支持的数据类型非常丰富，大致分为 6 大类：简单类型（包括有序类型和实数类型）、字符串类型（包括短字符串、长字符串和宽字符串）、构造类型（包括数组、文件、类、类引用、记录和集合）、指针类型、过程类型和可变类型。

2.2.1　简单类型

简单类型包括有序类型和实数类型。简单类型的数据可以进行比较，但是不相容的类型之间的比较是不允许的，例如一个整型数不能同一个字符比较。

有序类型的特点就是它的值总对应着一个整数序数，其值域是有限的。

Object Pascal 中共有 14 种预定义的有序类型：

Integer，Cardinal，Shortint，Smallint，Longint，Int64，Byte，Word，Longword，Boolean，ByteBool，WordBool，LongBool，Char。

还有 2 个用户自定义的有序类型：

Enumerated (枚举类型)，Subrange (子界类型)。

Object Pascal 允许把一个整型表达式强制转换成某种有序类型的数。有序类型中的整型 Integer 实际上是所有有序号整型数的统称。Cardinal 实际上是所有的无符号整型数的统称。把一个变量声明为 Integer 或 Cardinal，其字长和值域取决于 CPU 和操作系统，也就是说，在不同的机器上或不同的操作系统下，可能是不同的。例如，在 16 位的操作系统下，Cardinal 的值域是 0~65535，而在 32 位的操作系统下，Cardinal 的值域是 0~2147483647。建议读者尽量使用 Integer 和 Cardinal 来声明变量。

布尔类型主要用于关系运算和条件语句中，Object Pascal 中有四种布尔类型，如表 2.8 所示。

表 2.8　Delphi 中的布尔类型

类型	字长	值域
Boolean	1 个字长	只能是两个数：0 或 1
ByteBool	1 个字长	8 位的有符号整数
WordBool	2 个字长	16 位的有符号整数
LongBool	4 个字长	32 位的有符号整数

其中最常用的是 Boolean 类型，而 ByteBool、WordBool 和 LongBool 类型是为了与其他语言如 C/C++和 Windows 环境兼容，因为 Windows 的 API 在返回一个布尔值时，其值可能是一个两字节的符号整数，如果要把返回值赋给 Boolean 类型的数据，编译器认为类型不匹配，如果进行类型强制转换，又可能使返回值的有效数据被截断。

为了与 VisualBasic 的逻辑上兼容， Delphi 的 ByteBool、WordBool 和 LongBool 为 True 时其值是相当于－1 而不是 1，为 False 时其值仍然是 0。不过，对于 Boolean 类型来说，为 True 时其值仍然是 1，为 False 时其值仍然是 0。

Char 实际上是字符类型的统称，就像整型中的 Integer 一样，其精确类型为 Ansi Char 和 Wide Char 两种。Ansi Char，占一个字节，字符集包括扩展的 ANSI 字符。Wide Char，占二个字节，字符集包括 Unicode 字符，头 256 个字符对应于 ANSI 字符。

如果把一个变量声明为 Char ，通常就是 Ansi Char ，但在某些 CPU 或操作系统下有可能是 Wide Char。因此如果程序中必须用到字符的字节数，最好先用 SizeOf 函数取得，此函数是少数能用类型标识作为参数的函数之一。例如这样写：

```
SizeOf(Wide Char);
```

枚举类型和子界类型不同于标准的简单类型。标准的简单类型如整型和实型等，其字长和值域以及所能参加的运算都已由 Object Pascal 语言预先定义好,用户不必再进行类型描述，而枚举类型和子界类型则是由用户自定义，同样是枚举类型，如果类型描述不同，所表现出来的属性也不同。

所谓枚举类型，就是用一组数量有限的标识符来表示一组连续的整数常数，使用枚类型能够更清晰表示出现实世界。例如，一年中有 12 个月，如果程序中使用 0，1，2，3，4，5，6，7，8，9，10，11 来表示一年中的每个月，虽然从程序实现上来说是完全可以的，但是程序的可读性就比较差了。

当然，对于有些编程人员来说，也习惯于用常量来表示。这是完全可以的，但是在说明这些常量的时候可能步骤较为复杂一些。

在下面的例子中，声明了一个关于一年中月份的枚举类型：

```
Type MonthOfYear = (January, February, March, April, May, June, July, August, Septemper, October, November, December);
```

Object Pascal 规定，第一个标识符的值为 0，第二个标识符的值为 1，以此类推。

当然，为了简化程序，可以把类型声明和变量声明合二为一，例如：

```
Var MyDays: =(Sun, Mon, Tue,Wed, Thu, Fri, Sat);
```

枚举类型的优点是显而易见的，在许多情况下使用枚举类型会大大增强程序的可读性。但是，它也存在一定的缺点。例如，如果在编程的时候需要使用的常量数量比较大的时候，使用枚举类型也是比较麻烦的，需要逐个输入这些常量的名称。Delphi 另一种数据类型——子界类型则解决了这个问题。子界类型的特点在于它的值总是在一个区间内（把值全部罗列出来可能有困难）。

声明一个子界类型的语法是这样的：

```
subrange type DataName =  第一个常量..最后一个常量
```

其下界常数和上界常数的类型必须是同一种有序类型，如整型、布尔型、字符型，还可以是枚举型，但不能是实型，这两个常数的类型也称为子界的宿主类型。

例如在下面的程序中就演示了如何声明一个子界类型：

```
Type Letters = 'A'..'Z';
```

实型不是有序类型，因为它的值域是无限的，实型的数据也叫实数，类似于 C/C++中的浮点数。实数可以用科学计数法表示，例如：

- 7E –2 表示是 7 乘以 10 的负 2 次方，其中 7 称为尾数部分，–2 称为指数部分；
- 12.25E+6 或 12.25E6 表示是 12.25 乘以 10 的 6 次方。

使用科学计数法要注意指数必须为整数，可以为正也可以为负。即使尾数为 1，也不能省略，如写成 E–2 就是错误的。

2.2.2　字符串类型

与 Integer 类型一样，String 类型实际上是所有字符串类型的统称。Object Pascal 语言中确切的字符串类型有三种：ShortString、AnsiString 和 WideString，分别表示短字符串、长字符串和宽字符串。这三种类型的字符串可以相互赋值，或混合出现在一个表达式中，Delphi 会自动进行必要的转换。

1. ShortString

所谓短字符串，也叫 Pascal 风格的字符串，是指长度最大不超过 255 个字符的字符序列。保留这种数据类型的目的主要是为了与 Pascal 的早期版本兼容。

声明短字符串的语法如下：

```
var VarName: string[Number];
```

当编译开关{$H}的状态为{$H-}时，用保留字 String 声明的是一个短字符串。不过，不管编译开关{$H}在什么状态，用 ShortString 声明的总量总是短字符串类型。从语法可以看出，如果要用 String 保留字来声明短字符串，保留字 String 后必须有一对方括号，方括号内是一个无符号整数，此整数就是字符串的最大长度。程序示例如下：

```
Var myStr: string[125];
```

上例中，声明了一个最大长度为 125 的字符串变量 myStr。

当把一个字符串赋给一个短字符串类型的变量时，超过其最大长度的部分被截掉。短字符串中的每一个字符可以通过字符串变量名加字符索引号来访问。

注意：

短字符串类型的索引尽管是从 0 开始的，但索引号为 0 处的字节代表的是字符串的实际

长度，也就是说调用 Ord（S）[0]）相当于调用 Length(S)。真正的字符串是从索引号为 1 处开始的。

由于短字符串的长度是动态变化的，可以使用 Low 和 High 函数取得字符串中字符的最大序号的最小序号，Low 的返回值当然是 0，High 的返回值就是字符串的实际长度。

2. AnsiString

长字符串是用预定义的标识符 AnsiString 或把编译开关{$H}的状态改为{$H+}后用保留字 String 来声明的。长字符串的最大长度几乎是无限制的，理论上可达到 2GB，实际上受计算机的内存限制。长字符串类型的变量实际上是一个 32 位的指针，指向为长字符串动态分配的一块区域。如果长字符串是空的，指针的值就是 NIL。对长字符串类型的变量赋值，实际上只是复制一个 32 位的指针，而不是长字符串的全部字符。

多个变量可以同时引用同一个长字符串，内存中不会重复开辟多个区域。另外，长字符串的引用记数机制能保证：当修改其中一个变量的值，不会影响其他变量的值，虽然这些变量原先引用的是同一个字符串。

长字符串中的每一个字符也可以通过字符串变量名加字符索引号来访问。需要注意的是，长字符串的索引是从 1 开始的。

要取得长字符串的实际长度，只能调用 Length 函数。

长字符串还有个特点，就是编译器会自动在长字符串的最后一个字符后添加一个 NULL 字符（#0），表示字符串到此结束，但 NULL 字符本身并不是长字符串的一部分。

长字符串类型与字符指针类型（PChar）有点相似，可以很方便地把一个长字符串类型强制转换成 PChar 类型，语法是 PChar(S)，其中 S 是长字符串表达式。这样长字符串就转换为一个指针，指向长字符串的第一个字符，即使长字符串是空的。

长字符串还可以强制转换成一个无类型的指针，语法是 Pointer(S)，其中 S 是长字符串表达式，转换后的结果是指向第一个字符的指针，与用 PChar 强制转换不同的是，如果长字符串是空的，转换后的结果就是 NIL。

不管是用 PChar 还是用 Pointer 转换获得的指针都有一个有效范围的。Object Pascal 规定，如果被转换的是长字符串表达式，指针只在进行转换的语句内是合法的。如果被转换的是长字符串变量，指针在变量重新赋值前是合法的。

反过来，还可以把 PChar 类型的数强制转换为长字符串，语法是 String(P)，其中 P 是 PChar 类型的表达式。转换为长字符串后，就可以进行诸如两字符串合并等操作，例如：

```
S:=String(P1)+String(P1);
```

长字符串可以与 PChar 类型的数混合出现在一个表达式中，编译器将自动把 PChar 类型强制转换为长字符串类型。PChar 类型的数可以赋值给长字符串类型。

3. WideString

WideString 数据类型是从 Delphi 3 才开始出现的，用于表达多字节字符组成的字符串。有了宽字符串类型，Delphi 就能够处理国际字符集如 UNICODE 等。对于 ActiveX 应用程序来说，字符串必须是宽字符串。

宽字符串与长字符串非常类似，它们在内存中实际上都是用一个动态分配的数组来存储字符。不同的是，对于长字符串来说，数组元素的类型是 AnsiChar，而对于宽字符串来说，数组元素的类型是 WideChar。

2.2.3 数组

数组也是在编程中经常使用到的一个数据元素。所谓数组，就是由一些同类型的元素按一定顺序组成的序列。数组中的每一个元素都可以通过数组名加一序号来存取。

在 Object Pascal 语言中，要声明一个数组类型，其语法如下：

```
Type ArrayName = array [BeginNum..EndNum] of DataType;
```

例如可以声明如下一个数组类型：

```
Type MyArray=array [1..100] of Double;
```

表示数组类型的标识符是 MyArray，下标类型是整型，共有 100 个元素，元素下标从 1 到 100，每一个元素类型是 Double。

除了上面讲的一维数组，还有多维数组。多维数组是指数组中的元素本身还是个数组，其一般形式为：

Type 数组标识符=Array[下标类型 1]of Array[下标类型 2]of 元素类型

也可以把上述形式简写成下面的形式：

Type 数组标识符=Array[下标类型 1，下标类型 2]of 元素类型

现在可以声明这么一个两维数组变量：

```
var array1:array [1..5,1..8] of Integer；
```

表示数组有 5 个元素，每个元素又是由 8 个元素组成的数组。

假设有两个数组 array1 和 array2，其变量声明如下：

```
Var array1, array2:MyArray；
```

2.2.4 集合

集合本来是数学中的概念，Object Pascal 中引入集合的概念，是为了把一组相关的对象作为一个整体来参加运算，其中每一个对象称作集合的元素。

声明一个集合类型的语法如下：

```
Type SetName = Set of 有序类型；
```

其中 Set 和 of 是保留字，of 后面的类型可以是任何有序类型如整型、布尔型、字符型、枚举型和子界型，但不能是实型或其他的构造类型。

一个集合类型的值域实际上就是它的基类型值域的一个子集，也可能是空集，而一个集合类型变量的值域又是基类型值域的子集或空集。

2.2.5 指针类型

指针类型在任何语言中都是比较难理解也是比较灵活的一种数据类型，指针通常是它所指变量的内存地址。声明指针类型的语法如下：

```
Type DataName = ^BasicType;
```

程序示例如下：

```
Type BytePtr = ^Byte;
     WordPtr = ^Word;
```

上例中，声明了两个指针类型，一个是 BytePtr，指向 Byte 类型的数据，另一个是 WordPtr，指向 Word 类型的数据。

Object Pascal 不要求基类型一定是在前面已声明的，也可以只是一个标识符，然后，在同一个声明块内声明基类型。

2.2.6 过程类型

把一个变量声明为过程类型，可以把一个过程或函数作为一个整体赋给这个变量或者把这个变量作为参数传递给其他过程或函数。声明一个过程类型的语法同声明一个过程或函数的首部语法相似，不同的是在保留字 Procedure 或 Function 后不需要有过程或函数标识符。例如在下面的程序中，声明了三种不同情况的过程类型：

```
Type
     Proc1 = Procedure;
     Proc2 = Procedure (var x, y: Integer);
     Proc3 = Function ( X: Double): Double;
```

上列中，声明了 3 个过程类型，第 1 个是不带任何参数的过程，第 2 个是带有两个参数的过程，第 3 个是带一个 Double 类型的参数并返回 Double 类型值的函数。

同声明一个过程或函数一样，在声明一个过程类型时可以指定一种调用约定方式，默认就是 Register 方式。

2.2.7　可变类型

可变类型的特点是它的确切类型在运行期是可变的，也就是说它可以表示某些其他类型的数据（Variant 类型自身除外）。Variant 类型主要用于当数据的类型在编译期间是不确定的或是运行期间可能变化的情况。

尽管可变类型提供了许多编程上的灵活性，但要占用更多的内存空间和运行时间。

Object Pascal 用 Variant 来声明可变类型的变量，例如：

```
var x1, x2, x3: Variant;
```

可变类型的变量在被声明后总是被初始化为一个特殊的值 Unassigned（不管它是全局的还是局部的还是构造类型的成员），表示该变量还没有赋值。如果变量的值为 NULL，表示该变量的值是未知的或已丢失。

可以对可变类型的变量进行赋值，赋值号右边可以是一般的数据类型如整型、实型、字符串型和布尔型的值，也可以是日期和时间值，还可以是 OLE 自动化对象以及长度和维数可变的数组。

可变类型的变量可以与整型、实型、字符型、字符串型和布尔型的值混合出现在一个表达式中，编译器将自动进行必要的类型转换。

可变类型的高级用法是构造可变数组，从表面上看，数组元素的类型是一致的，都是 Variant 类型，但实际上由于可变类型的变量可以用其他类型的数值来赋值，因此，可变数组也像记录类型一样，可以有不同类型的元素。

2.3　变量和常量

2.3.1　变量

在 Object Pascal 语言中，声明变量的语法如下：

```
var Identifier List: Type;
```

变量是通过保留字 Var 声明的，Var 后面可以同时声明多个变量，如果几个变量的类型相同，可以合并在一起，彼此之间用逗号隔开。

程序示例如下：

```
var
  X, Y, Z: Double;
  I, J, K: Integer;
```

```
Digit: 0..9;
Flag: Boolean;
```

在 Delphi 中声明变量的时候需要注意变量的作用范围问题。实际上，在声明变量、常量、函数、方法等的时候都要注意这个问题。对于变量来说，存在全局变量和局部变量之分。所谓全局变量，就是在所有过程和函数之外声明的变量。全局变量又分为两大类，一是整个程序能访问的公共变量，必须在单元 Interface 部分声明；另一类是只限于某个单元访问的公共变量，必须在该单元 Implementation 部分声明。对于全局变量，可以在声明的同时赋一个初始值，如果没有赋初始值，这个全局变量的初始值就是 0。所谓局部变量，就是在某个过程或函数内部声明的变量，其作用只限于声明所在的块内。例如在下面的例子中，我们同时声明了一个全局变量和一个局部变量：

```
implementation
var RecNum:integer;     //全局变量
{$R *.dfm}
procedure TForm1.FormCreate(Sender: TObject);
var i:integer;       //局部变量
begin
end;
```

上例中，i 只是在过程 FormCreate 内有定义，而 RecNum 变量则在整个程序中有定义。对于局部变量而言，不能在声明时赋初始值，在明确地给它们赋值之前，它们的值是不确定的。

2.3.2 类型常量

类型常量最主要用于在声明变量时给变量赋初始值。类型常量与变量有些相似，不同的是类型常量是用 Const 声明的，而变量却是用 Var 声明的，常量的值在运行期间是不改变的，而变量的值是可以改变的。类型常量大致分为 4 大类，分别是简单类型的常量、指针类型的常量、过程类型的常量、构造类型的常量。

简单类型的常量是指常量的值为简单类型（有序类和实型）的类型常量，例如：

```
const Max: Integer = 100;
```

2.4 语　　句

通过前面的介绍，我们已经掌握了 Object Pascal 语言的数据类型和变量以及常量等，但是如果现在就进入 Delphi 的编程，还缺少一些把它们组织起来的语句。在这一节内容中，将

介绍把数据元素组织起来的语句。

2.4.1　声明语句

在 Delphi 中，语句通常分为两大类，一类是声明语句，另一类是执行语句。所谓声明语句，就是用来说明某些标识符的属性或者属于什么性质的元素。而执行语句则是用来按照程序员的意图进行运算、操作等的语句。

首先介绍声明语句。在 Delphi 中，几乎所有的元素（除了 Delphi 自身定义的全局元素之外）都需要先声明才能使用。常用的声明语句有八类，它们的语法以及示例见表 2.9。

表 2.9　Delphi 中的声明语句

名称	对应的概念	语法及示例	说明
标号	所谓标号（Label），就是用一个整型常数或标识符来标识程序的某个执行语句	Label 标号列表; 示例： Label 15,25,35	其中 Label 是保留字。以后可用 Goto 语句跳转到这个语句，用作标号的整数常数必须在 0 到 9999 之间
常量	常量的特点在于它的值在运行期间是不能修改的，对常量赋值是非法的	Const 常量标识符＝常量值 示例： Const Name='John'	Object Pascal 还允许等号右边是常数表达式，常数表达式的值是由编译器在编译的时候计算的，而不是在运行期由程序本身计算的
资源字符串	资源字符串是一种特殊的字符串类型的常量，编译器能够把它作为资源加到可执行程序、动态链接库或包文件中	ResourceString 资源字符串标识符＝常量	资源字符串可以用来对 AnsiString 类型的类型常量或变量赋值
类型	Object Pascal 允许用户声明自己的类型，这就要用到类型声明语句	Type 类型标识符＝类型； 示例： Type MyArray=Array[1..9] of integer	类型也有作用域问题，所有的过程或函数声明的类型可以在整个程序中使用，在某个过程或函数中声明的类型只能在这个过程或函数中使用
变量	变量与常量的区别在于，变量在运行期间其值是可以改变的	Var 变量标识符列表: 数据类型 示例：Var I,j,k:integer	使用变量一定要注意作用域问题
过程	Object Pascal 提供了大量预先定义好的过程，也可以定义用户自己的过程	procedure 过程标识符（参数 1, 参数 2, …）	如果有多个参数，参数之间用逗号隔开
函数	函数与过程最大的区别是函数具有返回值	funciton 过程标识符（参数 1, 参数 2, …）返回值类型	
输出项	输出项声明语句（保留字是 Exports）用于在 DLLs 中列出要输出的过程和函数	Exports 函数名称列表	关于这个声明语句的用法将在后面介绍创建 Dll 的时候进行详细的介绍

在 Delphi 中，声明语句虽然有很多变化，但是基本格式就是上面所提到的那些。而执行语句则不一样，它们有许许多多的变化。

下面要介绍的内容也只是它们的一些基本的构成元素或者是基本语句。利用它们可以写出千变万化的程序。当然，还有一些语句我们在这里没有介绍。随着对 Delphi 了解的深入就会知道，希望一本书把 Delphi 中可能出现的所有语句都介绍清楚是不可能的，也是不必要的。

2.4.2 赋值语句

赋值语句是变量处理中的最基本语句，一般来说，在对一个变量进行处理之前，要做的工作通常是先赋一个值给它。赋值语句是最简单也是最常见的语句，其语法如下：

变量:=值；

在赋值语句中，“值”可以是一个和变量类型相符的具体值，也可以是一个计算式。在下面的程序中，既有直接赋值的语句，也有把计算式的值赋予某个变量的语句：

```
VSplitter.Left := APanel.Width + VSplitter.Width;
Source.DockRect := ARect;
```

说明：

赋值号是:=，而不是=，初学者尤其要注意。=是关系运算符，用于判断等号两边的数是否相等。

2.4.3 Goto 语句

用过 Basic 或者 Fortran 编程语言的人可能对 Goto 语句非常熟悉，Delphi 中也存在这样的语句，虽然我们很少用它。原因是过多地使用 Goto 语句，甚至对结构化程序设计都是有害的，更不用说对面向对象的程序设计了。Goto 语句可能会使得程序跳来跳去，都可能使作者在一段时间之后无法读懂自己的程序，更重要的是，它可能在某个地方埋下了隐患，虽然当时没有发现。但是当程序出现问题的时候，就很难找到程序中不合理的地方。

Goto 语句用于把程序的执行点从当前位置无条件地转移到另一个语句，其语法如下：

Goto 标号。

程序示例如下：

```
procedure Cal;
var i,j, Count: Integer;
label OutFlag;
begin
  Count := 100;
  for i := 1 to Count do
```

```
            if i>=90 then
               goto OutFlag;
      ...
    Exit;
     OutFlag:
       ...
end;
```

上面的程序通过一个 Goto 语句跳出了循环结构，但是我们不建议读者使用这样的语句。关于如何跳出循环结构，Delphi 中提供了更为合理的方法。

2.4.4　复合语句

所谓复合语句，并不像前面介绍的赋值语句和 Goto 语句一样是某句程序，而是由一批语句组成的。它们通常作为某个条件或者某个函数或者某个过程中的一段语句。这些语句用保留字 Begin 和 End 括起来，Begin 是复合语句的开始，End 是结束。

复合语句可以嵌套，也就是说 Begin 和 End 之间还可以嵌套下一层 Begin 和 End，但要注意配套，程序示例如下：

```
procedure TMainForm.BottomDockPanelDockOver(Sender: TObject;
   Source: TDragDockObject; X, Y: Integer; State: TDragState;
   var Accept: Boolean);
var
   ARect: TRect;
begin
   Accept := Source.Control is TDockableForm;
   if Accept then
   begin
      //Modify the DockRect to preview dock area.
      ARect.TopLeft := BottomDockPanel.ClientToScreen(
         Point(0, -Self.ClientHeight div 3));
      ARect.BottomRight := BottomDockPanel.ClientToScreen(
         Point(BottomDockPanel.Width, BottomDockPanel.Height));
      Source.DockRect := ARect;
   end;
end;
```

说明：

使用复合语句要注意：Begin 后不需要分号，即使加了分号，编译器也把它看作一条空语句，而 End 语句后必须有分号。还有一个需要说明的问题是，Begin 和 End 之间只能是执行语句，不能有声明语句。

2.4.5 条件语句

条件语句用于判断某个条件是否满足，根据满足与否来控制程序的执行流程。简单地说，条件语句就是常说的“如果（条件），那么我们将（动作）”。

在 Object Pascal 中，有两种条件语句，一是 If 语句，另一个是 Case 语句。

对于单个条件的判断可以使用下面的语法：

```
if 条件表达式 then
begin
    语句
end;
```

上面程序的意思是如果条件表达式满足，那么将执行下面的复合语句。如果需要对条件表达式不满足的情况也进行处理，可以使用下面的语法：

```
if 条件表达式 then
begin
    语句 1
end
else
begin
    语句 2
end;
```

但是上面的语句在有些情况下也不能把所有情况表达清楚，例如需要在某个变量等于 3 时去完成某些情况，在小于 3 时去进行其他的动作，在大于 3 时进行另外的操作，那么此时可以使用下面的语法：

```
if 条件表达式 1 then
begin
    语句 1
end
else if 条件表达式 2 then
begin
    语句 2
end
else
begin
    语句 3；
end;
```

说明：

读者请注意 Else 语句前面的 End，它的后面没有我们通常在每一句后面使用的“ ;”。

程序示例如下：

```
if EP.IsWhiteSpace then
begin
  if EP.Character = #9 then
  begin
    C := EP.Column;
    EP.Delete(1);
    if C <> EP.Column then
      SpaceAdjust := C - EP.Column;
  end;
  EP.MoveCursor(mmSkipRight or mmSkipWhite);
end
else if EP.IsWordCharacter then
begin
  EP.MoveCursor(mmSkipRight or mmSkipWord);
  EP.MoveCursor(mmSkipRight or mmSkipWhite);
end
else if EP.IsSpecialCharacter then
begin
  EP.Delete(1);
  EP.MoveCursor(mmSkipRight or mmSkipWhite);
end
else
  EP.Delete(1);
```

If **语句可以嵌套，语法如下：**

```
if 条件表达式 1 then
  if 条件表达式 2 then
    语句 1
  else
    语句 2
```

Object Pascal **语言规定，**Else **总是与它靠近的没有** Else **部分的** If…Then **语句配套，上例中如果想让** Else **与第一个** If…Then **语句配套，程序就要这么写：**

```
if 条件表达式 1 then
begin
  if 条件表达式 2 then
    语句 1
```

```
end
  else
     语句 2
```

需要特别注意的是，在 If 语句中，必须知道每一个 Else 语句是和哪个 If 语句配对的。否则程序可能会不按照设想的过程去执行，虽然它们在语法上是没有错误的。

应该说，使用 If 语句基本上已经能够满足我们在条件判断上的需要，但是有时候比较麻烦，例如下面的语句：

```
if   RadioGroup1.ItemIndex =0 then
  Begin { screen proportional }
        Chart1.PrintResolution:= 0;
        Chart2.PrintResolution:= 0;
        Chart3.PrintResolution:= 0;
        Chart4.PrintResolution:= 0;
  End;
 Else if RadioGroup1.ItemIndex = 1 then
 Begin { thin lines and small fonts }
        Chart1.PrintResolution:= -100;
        Chart2.PrintResolution:= -100;
        Chart3.PrintResolution:= -100;
        Chart4.PrintResolution:= -100;
 End;
 Else if RadioGroup1.ItemIndex = 2 then
 begin
     ...
 end;
Else if RadioGroup1.ItemIndex = 3 then
 begin
     ...
 end;
 Else if RadioGroup1.ItemIndex = 4 then
begin
     ...
 end;
 Else
 Begin
     ...
End;
```

在上面的程序中，只有 6 个条件判断，如果有更多的条件判断，比如 255 个，这是有可能的，那么上面的 If 语句不知道要写多少个 If … Then 了。Delphi 中提供了一个更为简便的

方法来处理上面的情况，那就是 Case 语句。Case 语句的语法如下：

```
case selectorExpression of
  caseList1: statement1;
  ...
  caseListn: statementn;
end;

case selectorExpression of
  caseList1: statement1;
  ...
  caseListn: statementn;
else
  statement;
end;
```

例如上面提到的 If 语句可以利用下面的程序来实现：

```
Case RadioGroup1.ItemIndex of
  0: Begin { screen proportional }
       Chart1.PrintResolution:= 0;
       Chart2.PrintResolution:= 0;
       Chart3.PrintResolution:= 0;
       Chart4.PrintResolution:= 0;
     End;
  1: Begin { thin lines and small fonts }
       Chart1.PrintResolution:= -100;
       Chart2.PrintResolution:= -100;
       Chart3.PrintResolution:= -100;
       Chart4.PrintResolution:= -100;
     End;
  2:begin
     ...
     end;
  3:begin
     ...
     end;
  4:begin
     ...
     end;
  Else
     Begin
       ...
```

```
    end;
end;
```

Case 语句允许多个常量作为一种情况，彼此之间用逗号隔开，当多个常量是连续的值时，还可以用..符号。程序示例如下：

```
Case Flag of
1,2:begin
      ...
      end;
3..9:begin
      ...
      end;
end;
```

上面的程序中的第一个条件判断是 Flag 为 1 或者 2 的时候，将进行一种操作，第二个条件判断是 Flag 为 3 到 9 中的某个数的时候，将进行另外的操作。

Case 语句可以嵌套，但要注意每个 Case 语句必须有一个 End 配套。

在列出各种情况时，最好按照从小到大的顺序排列，这样能使编译器优化，否则每次都要计算一遍。

2.4.6 循环语句

循环语句和条件语句一样，是程序设计中最常用的简单语句之一。它主要用于重复执行一段语句，直至退出循环的条件满足为止。Object Pascal 语言提供了 3 种循环语句，其中 For 语句适用于循环次数确定的情况；While 语句适用于循环次数未知，需要判断条件满足与否来决定是否循环，有可能一次循环都没有；Repeat 语句适用于至少要循环一次，然后再判断条件是否满足来决定是否继续循环的情况。在循环中，可以使用两个标准的过程 Break 和 Continue 来控制循环流程。

For 语句的语法如下：

```
for counter := 初始值 to 终止值 do
begin
      语句
end;
```

上面的语句可以从某个初始值循环到终止值，直到循环变量 Counter 大于终止值为止。也就是说，在循环的过程中，循环变量每循环一次就加 1，直到大于终止值。如果需要循环变量递减的循环语句，可以使用下面的语法：

```
for counter := 初始值 downto 终止值 do
begin
```

```
    语句
end;
```

这个语法与前面介绍的不同的地方是使用了 Downto 保留字，就是说，循环变量将从初始值开始，逐次递减，直到循环变量的值小于等于终止值。

从上面的两个语法可以看出，For 语句首先赋一个初始值给循环变量，然后把它同终止值比较，当循环变量满足循环条件的时候，继续进行循环，当条件变得不满足的时候，循环便结束。在循环的过程中，循环变量将自动进行相应的操作，加 1 或者减 1。程序示例如下：

```
for I := 0 to FIDList.Count-1 do
begin
  DisposePIDL(ShellItem(I).ID);
  Dispose(ShellItem(I));
end;
FIDList.Clear;
```

与 If 语句和 Case 语句一样，For 语句也可以嵌套，但每个 For 语句应使用不同的循环变量，程序示例如下：

```
for I := 1 to 10 do
  for J := 1 to 10 do
  begin
    X := 0;
    for K := 1 to 10 do
      语句；
  end;
```

正常退出循环后，控制变量的值并不是像想象的那样是终止值加 1 或减 1，除非在循环后重新赋值，否则控制变量没有意义。有一个特殊情况是，当 Goto 语句提前跳出循环时，控制变量的当时值可以被引用。

正如前面介绍过的，For 语句通常用于知道一定循环次数的循环语句。如果不知道循环次数，可以使用 While 语句或者 Repeat 语句。

While 语句的语法如下：

```
while 条件表达式 do
begin
 语句；
end;
```

While 语句首先计算条件表达式的值，当值为 True 时就执行一次 Do 后面的语句，否则一次也不执行。程序示例如下：

```
while CurModule <> nil do
```

```
begin
  if CurModule.Instance = HInstance then
  begin
    if CurModule.ResInstance <> CurModule.Instance then
      FreeLibrary(CurModule.ResInstance);
    CurModule.ResInstance := NewInstance;
    Result := NewInstance;
    Exit;
  end;
  CurModule := CurModule.Next;
end;
```

While 语句的特点是先判断条件，这样有可能一次循环都不执行。While 语句的循环体中必须要有影响条件表达式的值操作，这样循环才有可能退出，否则将引起死循环。

如果不知道循环语句可能需要多少次数，但是希望先执行一次，然后根据结果来判断是否继续执行循环语句，可以使用 Repeat 语句，这样的情况的典型范例是在硬盘上的某个目录中搜索文件的示例。通常是先搜索一次，如果找到了对应的文件，那么将继续搜索，如果没有找到，那么循环就应该结束了。

说明：

关于搜索文件的示例程序，在后面的内容中将会有详细的介绍。

Repeat 语句的语法如下：

```
repeat
begin
      语句；
until 条件表达式
```

Repeat 语句首先执行一遍循环体中的语句，然后判断条件表达式的值，为 False 时就重复执行一遍循环体，为 True 时就退出循环，程序示例如下：

```
repeat
if (I = FIDList.Count-1) then
     if Wrap then I := 0 else Exit;
       Found := Pos(UpperCase(FindString),
                    UpperCase(ShellItem(I)^.DisplayName)) = 1;
       Inc(I);
 until Found or (I = StartIndex);
```

Repeat 语句的特点是至少循环一次。

For 和 While 语句的循环体可以是复合语句，但必须用 Begin 和 End 括起来。而 Repeat 语句在保留字 Repeat 和 Unitl 之间是可以有多条语句，但不需要用 Begin 和 End 括起来。

与 While 语句一样，在 Repeat 语句循环体中必须有影响条件表达式的操作，否则会引起死循环。

通过上面的内容，我们已经掌握了 Delphi 中的三种关于循环语句的用法，通常情况下，已经能够应付这方面的需要了。但是仍然存在一些特殊情况需要进行处理。比如在一个循环模块中，当程序执行到某个结果的时候，我们希望终止该循环，去执行其他的程序，也就是可能需要按照意愿跳出循环，此时可以使用 Break 语句。

Break 语句用于提前终止循环，程序示例如下：

```
for j := 0 to TS.Count-1 do
  if TS.Strings[j] = Name then
  begin
    Result := j;
      Break;
  end;
```

当满足条件的时候退出循环是 Break 语句的特点。但是还有一种特殊的情况，那就是在循环中，有一些情况希望执行其他语句，或者不执行主循环体中的语句，而其他的大多数情况需要执行主循环体中的语句，此时可以使用 Continue 语句。

Continue 语句用于不等整个循环体执行完就重新判断循环条件是否满足,程序示例如下：

```
For K:=-10 to 10 do
begin
    if K=0 then continue;
    X[K] := X[k]/k;
End;
```

在上面的例子中，需要从–10 到 10 进行循环处理，但是在处理的过程中，对于 0 的情况不能用主循环体中的语句来处理，所以，此时可以用一个 Continue 语句来处理。Continue 语句只是把循环变量加 1，然后继续进行循环。当然也可以在 Continue 语句前面加上其他的处理语句，并把这些语句用 begin 和 end 括起来，作为复合语句使用。

2.5　过程和函数

到目前为止，我们已经学会了 Delphi 中的基本数据类型和基本语句，利用它们现在可以做些什么了。下面介绍如何声明过程和函数，并利用上面介绍的内容编写一个过程或者函数。

说明：

包括上面介绍的内容，数据类型和基本语句，以及本节所要介绍的过程和函数，都是结构化程序设计的内容。实际上在掌握任何编程语言的时候都要涉及到这些方面，那就是如何

声明数据，如何处理数据。一个很好地掌握了结构化程序设计的人，可以很快地掌握面向对象的程序设计，因为面向对象的程序设计是建立在结构化程序设计的基础上的。另外，面向对象的程序设计的另外一个重要内容是程序中的对象和类。对于结构化程序设计员来说，这是一种全新的设计思想。关于 Object Pascal 语言中对象和类的问题，我们将在后面的内容中进行详细的介绍。

过程和函数十分类似，它们的根本区别是有没有返回值。过程是一段更为复杂的复合语句，把它们区别出来是为了程序设计的方便，或者为了重复调用；而函数虽然同样是一段更为复杂的复合语句，但是它的目的是通过一段语句得到某个结果，并把这个结果返回给调用它的程序，以便于根据结果进行不同的处理。

Delphi 中有两类过程和函数，一类是 Delphi 预先定义好的过程和函数，这些用户只要调用就可以了；另一类是用户自己定义的过程和函数，它们的定义规则稍微复杂一些。

说明：

在 Delphi 中，第一类过程和函数的数量是相当庞大的，除了多看关于这些方面的书籍之外，需要阅读更多的示例程序。很多时候，需要自己费尽千辛万苦去编写的过程和函数，而实际上 Delphi 已经定义好了。所以建议读者尽量使用 Delphi 中预先定义的过程和函数。

从语法上来说，使用 Delphi 预先定义好的过程和函数的方法是十分简单的，只要知道 Delphi 的这个过程或函数是在哪个单元文件中定义的，比如 ComCtrl，只要把它加入到程序中的 Uses 语句中就可以调用这个过程和函数了。所以重点是介绍如何定义和调用我们自己的过程和函数。

2.5.1 过程的声明、定义及调用

在 Delphi 中，一个过程，（包括 Delphi 中的预先定义的过程），都包含这样的内容，先声明这个过程，然后在相应的代码中定义这个过程，最后在自己的程序中调用这个过程。

声明过程的语法如下：

```
procedure MyProc (参数列表);
```

其中 Procedure 是保留字，过程标识符可以是任何合法的标识符，以后要用这个标识符调用这个过程。一个过程可以有参数，如果有多个参数，参数之间用分号隔开，也可以没参数，对于后者，过程标识符后直接跟分号。声明过程的程序示例如下：

```
procedure SetForecolor(value:Tcolor);
```

声明了一个过程后，就应当在单元的 Implementation 部分定义这个过程，程序示例如下（过程可以有自己的局部变量）：

```
procedure TGraphNum.SetForecolor(value:Tcolor);
```

```
begin
    if Fforecolor<>value then
    begin
        Fforecolor:=value;
        invalidate();
    end;
end;
```

过程经声明和定义后，就可以在程序中调用。在程序中可以像下面一样来调用一个过程：

```
SetForecolor(clred);
```

实参可以是常量、变量或表达式，但实参的个数和类型必须与形参完全匹配，缺少一个参数、多出一个参数或者参数类型不匹配都是错误的。

说明：

在过程声明和定义中，多个参数是用分号隔开的，但在调用时是用逗号隔开的。

2.5.2　函数的声明、定义及调用

声明函数的基本方法和上面介绍的声明过程是十分相似的，但是有一个重要的区别，那就是函数需要确定返回值的类型。声明函数的语法如下：

```
Function MyFunc（参数列表）：返回值类型；
```

例如在下面的例子中，声明了一个返回双精度值的函数。

```
function getg11(i,j:integer):double;
```

说明：

函数可以有参数，也可以没有参数，但必须指定返回类型，Object Pascal 的函数可以返回除了文件类型以外的任何数据类型。

定义函数的过程就是数据处理的过程，需要注意的是必须返回一个值。在下面的示例程序中，定义了上面声明的函数：

定义函数的程序示例如下：

```
function    getg11 (i,j:integer):double;
var s1,s2:double;
begin
    s1:=(deltx[i,j+1,2]-deltx[i,j-1,2])/(2*jjj[i,j]);
    s1:=-s1*(deltx[i+1,j,2]-deltx[i-1,j,2])/(2*jjj[i,j]);
    s2:=-(deltx[i,j+1,1]-deltx[i,j-1,1])/(2*jjj[i,j]);
    s2:=s2*(deltx[i+1,j,1]-deltx[i-1,j,1])/(2*jjj[i,j]);
```

```
    Getg11:=s1+s2;
end;
```

程序中的 DeltX 是我们定义的一个全局数组。在程序的结尾用一个表达式把结果赋予了函数名，这代表着表达式的结果将作为函数的返回值带入调用该函数的语句中。

注意：

在过程的定义中，过程的执行体可以是空的，但函数的执行体至少要有一个语句，那就是返回一个数值。

调用函数与调用过程有些相似，但也有不同。调用过程的语句是个独立的语句，称之为过程语句，例如可以在程序中这么写：

```
for j:=1 to nj+1 do
begin
        g11[i,j]:=getg11(i,j);
        g12[i,j]:=getg12(i,j);
        g22[i,j]:=getg22(i,j);
end;
```

在上面的程序中，利用一个循环语句，计算了很多点上的 Getg11 的值。同时还调用了其他的函数。

函数调用也可以直接作为操作数，如下列：

```
if Getg11(0,0)>25 then
begin
    ...
end;
```

与调用过程一样，调用函数时，实参的个数和类型必须与形参完全匹配，否则调用将是非法的。

注意：

在函数中，只要遇到对函数名赋值的语句，函数就返回，也就是说，对函数的赋值是函数执行的最后一个语句。

除了上面介绍的把最后的结果赋值给函数名之外，Delphi 还自动在函数内部隐含了一个局部变量，这就是 Result 变量，这个变量的类型与函数的返回值类型相同，对 Result 的赋值相当于对函数名赋值，也就是说，相当于返回一个数值。通常来说，在定义函数的时候，更多的是使用 Result 变量。程序示例如下：

```
function   TForm1.getg22 (i,j:integer):double;
var s1,s2:double;
```

```
begin
    s1:=(deltx[i+1,j,2]-deltx[i-1,j,2])/(2*jjj[i,j]);
    s2:=(deltx[i+1,j,1]-deltx[i-1,j,1])/(2*jjj[i,j]);
    result:=s1*s1+s2*s2;
end;
```

如果 Result 变量仅仅是为了返回一个数值，那它也没有什么意思，Result 变量的关键在于它能出现在赋值语句的右边，而如果函数名出现在赋值语句的右边，则意味着有递归调用。对 Result 变量赋值不一定是从函数返回，这一点与对函数名赋值也不同，程序示例如下：

```
function   TForm1.getg22 (i,j:integer):double;
var s1,s2:double;
begin
    s1:=(deltx[i+1,j,2]-deltx[i-1,j,2])/(2*jjj[i,j]);
    s2:=(deltx[i+1,j,1]-deltx[i-1,j,1])/(2*jjj[i,j]);
    result:=s1*s1+s2*s2;
result:=result+103.15;
end;
```

2.5.3　过程和函数的调用方式

所谓调用方式，是指参数的传递方法。Object Pascal 语言共有 5 种调用约定方式，分别是 Register，Pascal，Cdecl，Stdcall，Safecall。如果没有明确地指定调用约定方式，那就是采用默认的 Register 方式。各种方式的说明见表 2.10。

表 2.10　过程和函数调用方式说明

名称	说明
Register 方式	Register 方式尽量采用寄存器来传递参数，传递次序从左到右，最多可用到 3 个 CPU 的寄存器，如果参数多于 3 个，剩下的就通过栈传递，使用寄存器传递参数速度最快。由于 CPU 的部分寄存器被专用于参数传递，因此程序本身不要再去存取这几个寄存器，这几个寄存器是 EBX，ESI，EDI 以及 EBP
Pascal 方式	Pascal 方式采用栈来传递参数，传递次序从左到右，这种方式适用于调用动态链接库中的例程，而这些例程可能是用其他语言编写的
Cdecl 方式	Cdecl 方式采用 C/C++的调用约定，参数从右到左依次传递到栈中。这种方式适用于调用动态链接方式输出的例程，而这些例程是用 C/C++等语言编写的
Stdcall 方式	Stdcall 方式采用 Windows 的标准调用约定来传递参数，传递次序从右到左，这种方式适用于调用 Windows 的 API
Safecall 方式	Safecall 方式传递参数的次序是从右到左，适合于专用 OLE 对象中的方法

当过程或函数返回时，参数应当从栈中清除。上述 5 种调用约定方式中，除了 Cdecl 外，

参数都是自动清除的，对于 Cdecl 方式来说，调用者必须自己清除栈。

2.5.4 过程或函数中变量的作用域问题

用 Object Pascal 语言编程，要特别注意作用域问题。

在单元的 Interface 部分声明的变量适用于整个程序，在单元的 Implementation 部分声明的变量适用于本单元，上述两个部分声明的变量都叫全局变量。

在过程或函数的定义部分声明的变量只适用于这个过程或函数，称为局部变量。

注意：

如果过程或函数的参数是数值参数，这些参数也是这个过程或函数的局部变量，局部变量只在它的作用域有意义，编译器不允许在作用域之外引用局部变量。

局部变量被放在过程或函数的栈中，如果局部变量太多，注意不要使栈溢出，如果发生栈溢出，将触发 EStackOverflow 异常，Object Pascal 提供了两个编译指令：$MINSTACKSIZE 和$MAXSTACKSIZE 分别用于指定最小和最大的栈空间。默认最小栈空间是 16KB，最大的栈空间是 1MB。

如果全局变量与局部变量同名怎么办？Object Pascal 规定，在过程或函数外，是全局变量起作用，而在过程或函数内部是局部变量起作用。当然尽量不要让全局变量与局部变量同名。

2.5.5 指示字

在声明过程和函数时，可以附加三种指示字：Assembler、External、Forward。

1. Assembler

这个指示字表示过程或函数是用嵌入式汇编语言编写的。用这个指示字后，编译器对参数的处理做了一些优化：

- 对于某些数值参数如字符串类型的参数或者字节长度不是 1、2、4 的参数，编译器将把它当做变量参数。
- 对于函数来说，没有隐含的 Result 变量，只有返回字符串的函数可以引用这个变量。
- 对于没有参数或没有局部变量的过程或函数来说，编译器不生成栈框架，因为这需要耗费时间。

注意：

因为函数必须返回一个值，因此用汇编语言写的函数也必须返回一个值，返回的规则是这样的：

如果返回类型是有序类型，Byte 类型的数从 AL 寄存器返回，Word 类型的数从 AX 寄存器返回，Double 类型的数从 EAX 寄存器返回。

如果返回类型是实型，返回值可以从浮点协处理器的栈首寄存器 ST（0）读取。其中，如果返回类型是 Currency，返回值就是 ST（0）寄存器的值乘以 10000。

如果返回类型是字符串、方法指针或可变类型，编译器将通过一个 32 位的指针来传递一个变量，用这个变量来返回函数的结果。

如果返回类型是无类型指针、类、类引用以及全局过程指针，通过 EAX 寄存器返回。

如果返回类型是数组、记录和集合，当数据只有一个字节时，通过 LA 寄存器返回；当数据有两个字节时，通过 AX 寄存器返回；如果数据有四个字节，通过 EAX 返回。否则，必须另外指定一个变量参数来返回函数结果。

2. External

这个指示字适用于从 DLLs 中引入过程或函数，表明过程或函数是外部的。

3. Forward

一般情况下，当调用一个过程或函数时，被调用的过程或函数必须是在前面已定义的，不过使用 Forward 指示字可以让编译器向前查找过程或函数的定义，程序示例如下：

```
procedure MyProc(Param1, Param2:Integer); Forward;
procedure TForm1.FormCreate(Sender:TObject);
begin
    MyProc(1,2);
end;
procedure MyProc;
begin
    MessageBeep(0);
end;
```

上例中，在调用 MyProc 过程之前，该过程还没有定义。由于声明这个过程时加了 Forward 指示字，编译器向前查找该过程的定义。

读者可能已发现，在过程 MyProc 的定义中省略了参数，只写了过程名，这在使用 Forward 指示字的情况下是允许的。

2.5.6　参数类型

Object Pascal 语言允许 6 种类型的参数（不是指参数的数据类型），分别是数值参数、常数参数、变量参数、无类型的参数、外部参数和开放数组参数。

在调用过程或函数时，实参与形参在个数上必须相同，在数据类型上必须赋值相容（无类型参数除外）。

1. 数值参数

数值参数是最常见的参数。当调用过程或函数时，程序把实参值复制一个副本，传递给数值参数，数值参数对过程或函数而言是局部变量，对数值参数的修改不会直接改变实参的值。

注意：

作为实参传递数值参数的值不能是文件类型或包含文件类型成员的构造函数类型。

2. 常量参数

常量参数与数值参数的区别在于，常量参数是只读的，也就是说不允许在过程或函数中改变常量参数的值，对常量参数赋值是非法的，并且不允许把常量参数再作为实参传递给另一个过程或函数。

注意：

作为实参传递给数值参数的值不能是文件类型或包含文件类型成员的构造函数类型。

3. 变量参数

变量参数有点像 C++中的引用，参数传递的不是实参的副本，而是实参的地址。这样当参数在过程或函数中被改变时，实参的值也相应改变，在过程或函数中对形参的引用实际上就是对实参自身的引用。

注意：

当形参是变量参数时，实参也必须是一个变量而不能是一个表达式。

另外，实参与形参的数据类型必须完全一致而不仅仅是赋值相容。

当形参是变量参数时，实参可以是文件类型或包含文件类型成员的构造函数类型。

当参数是构造函数类型时，使用变量参数具有优势，因为如果使用数值参数，在调用时必须分配足够大的内存空间来存放实参的副本，当参数比较庞大时，将花费大量的空间和时间，而使用变量参数时，传递的仅仅是实参的地址，其效率是显而易见的。

4. 无类型参数

所谓无类型参数，就是在过程或函数声明和定义时不指定参数的数据类型。

如果参数没有指定数据类型，就必须在前面加：Var、Out 或 Const，加 Var 或 Out 表示

参数的值是可以改变的，加 Const 表示参数是只读的。

当参数是无类型参数时，实参可以是任意数据类型的变量或表达式，使用无类型参数可以使编程更加灵活，但也给编译器带来麻烦，因为它没有对参数进行合法检查。

正因为参数是无类型的，因此在使用前一定要先进行类型强制转换，把它转换成某种特定的数据类型。

如果函数的参数是无类型参数，并且是可以修改的，在使用这两个参数时，先要用 TByte 这个类型标识符对它们进行强制类型转换。

5. 外部参数

外部参数非常类似于变量参数，当过程或函数调用结束时，通过外部参数可以回传某些信息。如果调用失败的话，外部参数就被设为 Ni1。

注意：

一般情况下，不需要用到外部参数，因为变量参数完全可以起到回传信息的作用。仅在声明 COM 对象的方法时可能需要用到 Out 类型的参数。

6. 开放数组参数

所谓开放数组参数，就是参数是个数组，但数组的长度都是不确定的。

如果参数 A 就是一个开放数组参数，它的基类型是 Double，长度是不确定的，同时它又是一个常量参数。

7. 过程或函数本身作为参数

在 Object Pascal 中，允许把过程或函数本身作为参数来传递，因为过程或函数也可以看作是一种数据类型。

注意：

不能把标准的过程或函数作为参数，只能把用户自己定义的过程或函数作为参数传递。

2.6　本 章 小 结

这里我们再总结一下掌握一门编程语言，特别是面向对象的编程语言的基本思路，在上面的内容中虽然曾经提到了这个方面的内容，但是仍然有必要进行说明。对于面向对象的编程语言一般分为三个方面的内容：

- 数据类型：这里包括数据类型的种类，各自的特点，如何用合适的变量来代表它们。

- 语句：和数据类型构成一门语句的两个基本元素。在了解了这些语句之后，就可以编写一些简单的程序了。
- 对象：数据和语句构成了对象，但是对象又不是简单的数据和语句的组合。我们需要了解对象的属性和方法，在面向对象的程序设计中，处理最多的是对象，而在对象之间以及对象内部主要是数据的处理。

在一章中介绍 Object Pascal 语言的结构是一件十分艰难的事情，但是我们终于完成了。也许你在阅读了本章之后对 Object Pascal 中的一些语法问题仍然不清楚，没有关系，我们将在后面的内容中在使用到它们的时候再进行详细的介绍。

第 3 章　Delphi 6.0 面向对象编程基础

Delphi 最大的特点是一个面向对象的编程，所以在本章中，将通过一些简单的例子，向读者介绍关于 Delphi 编程中的面向对象问题。对象、属性、方法是 Delphi 编程处理的核心。

另外 Delphi 6.0 是一个可视化编程工具，它具有可视化编程的一切优点，遵循着 WYSIWYG（所见即所得）的编程原则，使用起来编程简便快捷。

但是必须承认，要使用 Delphi 6.0 编写出专业的、高级的、功能强大的应用程序也并不是一件非常容易的事情。必须遵循一些编程的原则，才能使我们的程序具有良好的可通用性、可移植性和可维护性。

在本章中，将介绍以下内容

- 创建第一个应用程序
- 了解面向对象的编程思想
- 编写好的程序的注意事项

3.1　创建第一个应用程序

虽然我们已经花了大量的精力来了解 Delphi 6.0 集成开发环境的使用方法，但是，到目前为止，还没有介绍一个称得上是程序的东西。所以，你对如何使用 Delphi 6.0 编程可能还没有一点明确的概念。在这一节中，就以创建一个最简单的应用程序为例，向大家介绍在 Delphi 6.0 中创建一个应用程序的基本步骤。

3.1.1　新建一个工程

一般来说，Delphi 把建立一个应用程序（或者 DLL、AcitveX 控件以及其他 Delphi 认可的对象）所需要的所有文件的综合叫做一个工程。其中有一个文件专门记录了工程所需要的信息，这个文件通常就叫做工程文件，也有的书籍上把它称为一个项目。

换句话说，如果要创建一个新的应用程序，必须首先创建一个新的工程。单击 File（文件）菜单中的 New（新建），并选择 Application（应用程序），或者 New Application（新建应用程序），就会创建一个新的工程，如图 3.1 所示。

说明：

实际上，每次启动 Delphi 6.0 的时候，都会自动创建一个工程。这个工程和用上面的方

法创建的工程是一模一样的。

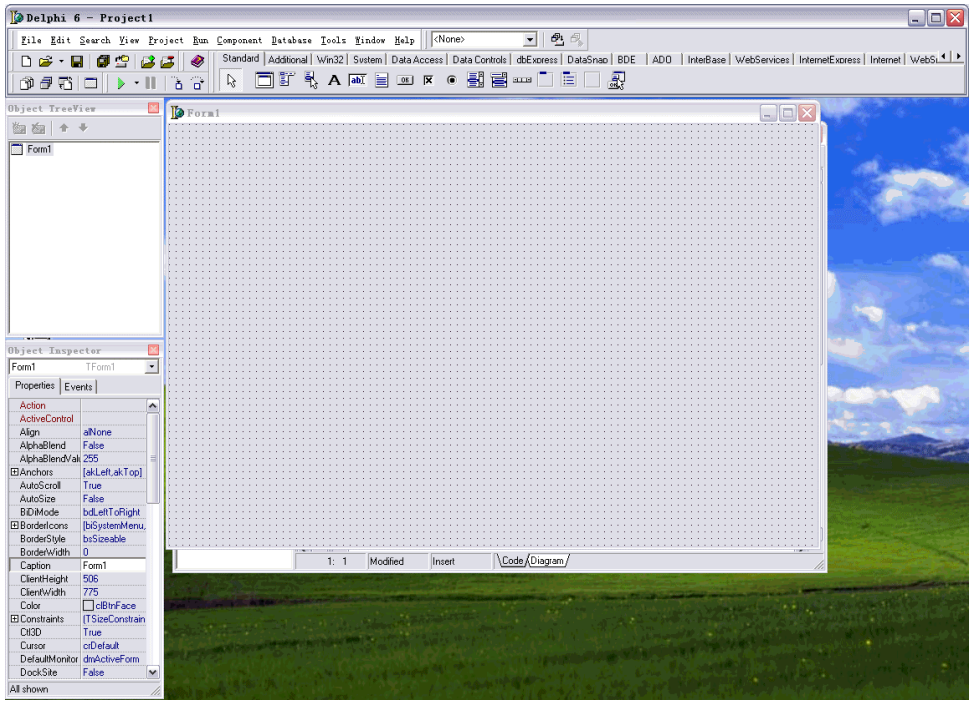

图 3.1　新建的一个工程

单击 View（视图）菜单中的 Project Manager（工程管理器）命令，可以显示如图 3.2 所示的一个窗口。在这个窗口中，显示了当前工程的结构。

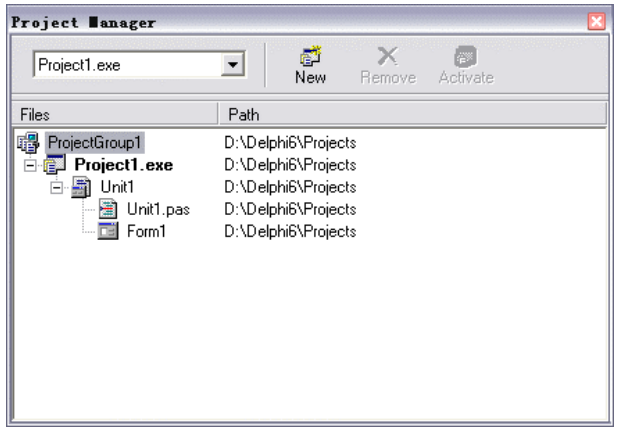

图 3.2　Project Manager 窗口

1. 单元的结构

从图中可以看出，这个工程包括一个 Form，一个 Unit（单元）。单击工具栏上的 Toggle Form/Unit 按钮，可以切换到代码编辑器。在代码编辑器中，Delphi 6.0 已经自动为程序添加了一段代码。这些代码声明了一个类，这个类代表的就是工程中的窗体。代码如图 3.3 所示。

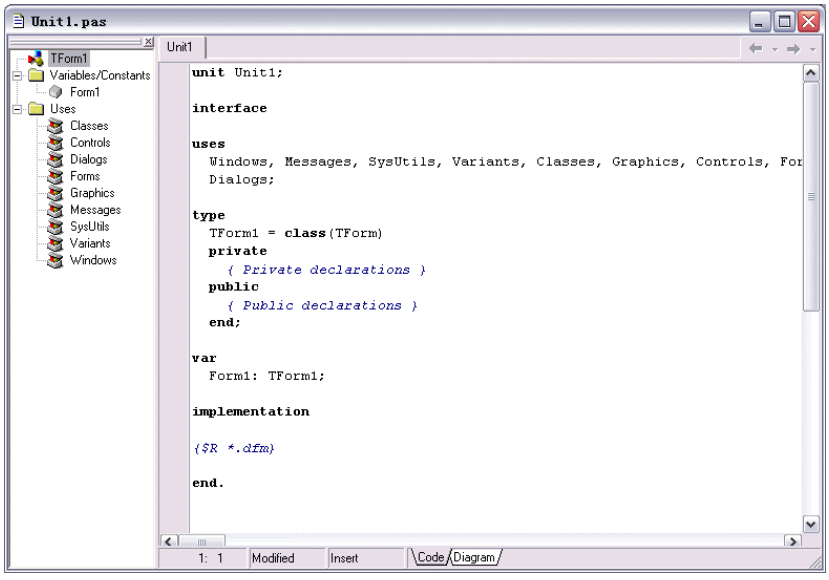

图 3.3　Delphi 6.0 自动生成的代码

不要小看这段代码的意义。虽然它是 Delphi 自动生成的，而且形式上也极其简单，但是它却代表了 Delphi 中最为典型的程序单元结构。下面分析一下这段代码的结构。

从图中可以看出，一个典型的程序单元主要包括三个部分：Unit（单元头）、Interface（接口）和 Implementation（实现）。实际上一个完整的程序单元还可能包含其他两个部分：初始化部分和终止部分。不过这两个部分是可选的。

第一个部分是 Unit。这个部分比较简单，只有一个语句，那就是“ Unit Unit1; ”。这里的 Unit1 指的是这个单元的名称，在默认的情况下是 Unit+数字。如果在一个单元中需要引用其他单元中的代码的时候，就需要使用这个单元名称了。当我们把某个单元保存成其他名字的时候，这个单元名称会随之而改变。通常来说，不需要直接改变这个单元头的名称，因为这样作可能会引起多方面的问题，特别是在从其他的程序单元中引用了这个单元的时候。需要注意的是，“ Unit Unit1; ”是一个完整的 Pascal 语句，其中 Unit 是保留字，所以在这一句的结尾必须写上“ ; ”。

第二个部分是 Interface 部分，它的主要作用是用来声明引用的单元、常量、类、变量、

过程和函数等。例如，在这里 Uses 语句代表的就是在当前这个单元中所引用的其他单元的单元名称。紧接着声明了一个类。在类中，可以声明类的公共、私有、保护的属性、元素和方法。接口部分从 Interface 保留字开始，到 Implementation 保留字结束。

在接口部分，过程和函数只需要写出它们的首部，它们的具体定义是在下面的 Implementation 部分实现的。接口部分又可以分成几个部分，其中大多数部分都是可选的。它们分别是：单元引用部分、声明部分、类声明部分、变量声明部分、过程和函数声明部分。

单元引用部分用于列出该单元要引用的标准单元和其他单元。单元引用的概念类似于 C 语言中的 Include 语句，用于把外部的已经声明过的常量、类、变量、过程和函数引入到本单元中，使得我们可以像使用本单元中的常量、类、变量、过程和函数一样使用这些元素。

说明：

实际上，在一个单元中引用其他单元元素和引用本单元元素还是有区别的，在后面的章节中将会详细地介绍如何引用其他单元中元素的问题。

在上面的例子中，Uses 部分引用了 Windows、Messages、SysUtils、Classes、Graphics、Controls、Forms、Dialogs 等单元，这样就可以直接引用这些单元中的例程，而不需要其他的操作，例如可以直接调用下面的过程：

```
MessageBeep（0）；
```

说明：

虽然 Uses 部分在 Interface 部分是可选的，但是，如果在接口部分包含了 Uses 部分的话，这个部分必须紧跟在 Interface 保留字的下一行。

在上面的例子中，引用的都是标准单元，而且其中的大部分单元都是几乎所有的单元都会引用的标准单元，因此，Delphi 会自动把这些单元加入到 Uses 部分。不过，对于一些不经常使用的标准单元来说，如果需要引用其中的代码，就必须手动加上对该单元的引用。

在接口部分还存在着其他一些声明部分，关于这些部分的语法规则请参见本书后续章节。

下面我们再来看一下程序的实现部分。实现部分分为两大部分，一部分是声明部分，包括单元引用、常量、类、变量、过程和函数，这一点和接口部分十分类似。所不同的是，如果在其他单元中引用了该单元的话，该单元中的接口部分声明对引用它的单元来说是可见的，也就是说，可以在另外一个单元中引用该单元接口部分的常量、类、变量、过程和函数，但是却不能使用实现部分的常量、类、变量、过程和函数。另外的一个不同点是，在实现部分的过程和函数声明不需要遵循先声明后定义的规则，可以直接写出函数或者过程的定义。

实现部分的另一部分是在接口部分声明的过程、函数定义。

2. 工程文件的结构

在前面曾经提到，Delphi 的一个工程是由它的工程文件来定义的。实际上，这个工程文

件的结构也是非常典型的，虽然在绝大多数情况下不需要改变其中的内容，但是有一些特殊的情况，如果对这个文件做一些改动的话，可以取得非常好的效果。例如，如果希望自己的程序能够有一个启动画面时，就需要改动这个文件。所以，了解一下工程文件的结构还是非常必要的。

如果要查看当前工程的工程文件，可以选择工具栏上的 Units 按钮，或者选择 View 菜单中的 Units 命令，此时会出现如图 3.4 所示的对话框。

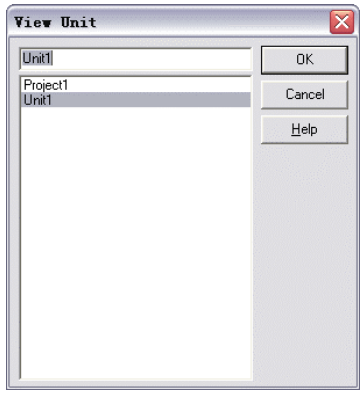

图 3.4　View Unit 对话框

在图中，和工程同名的单元就是工程文件。Delphi 在默认的情况下创建的第一个新工程叫做 Project1。选择该文件，并单击 OK 按钮，或者双击该文件，便可以在代码编辑器中打开这个文件，如图 3.5 所示。

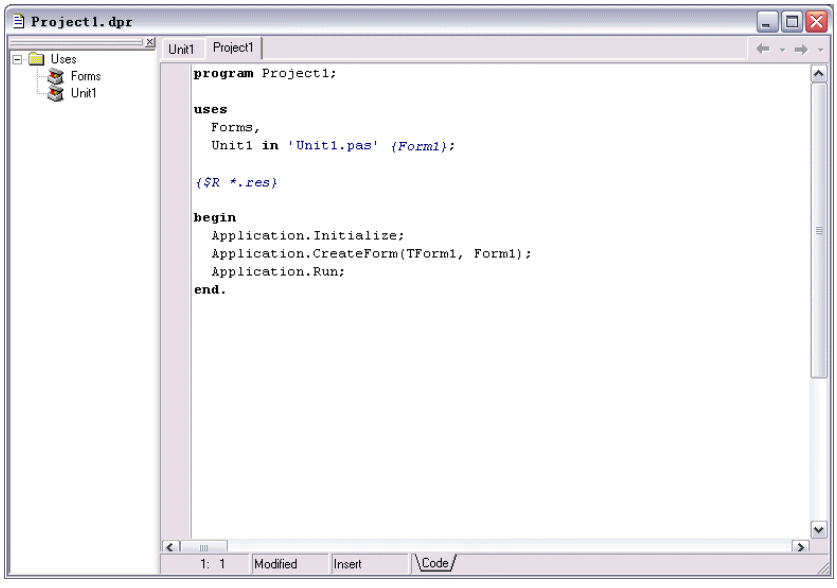

图 3.5　代码编辑器中的工程文件

从图中可以看出，工程文件是由四个部分组成的。

- 程序头：这个部分和单元文件中的单元头的作用是一样的，只不过使用的保留字是 Program，而不是 Unit。
- 引用部分：这里的引用部分和单元文件中的作用相同。但是它包含了一个该工程所包含的单元文件 Unit1，并且指定了这个文件的名称。
- 编译指令：在图中，只有一个编译指令，就是{$R *.$res}，注意编译指令也是可选的。
- 执行部分：这个部分也叫做初始化部分，是由保留字 Begin-End 对包括起来的一段代码。代码中的第一句，进行应用程序的初始化工作；第二句创建工程中包含的窗体（程序的主窗体），第三句开始运行应用程序。

实际上，即使是像上面一样简单的工程，Delphi 所包含的文件也不止上面提到的这两个。但是和编程工作密切相关的也就是这两个文件。对于其他的文件，有的是关于这个工程所必须包含的一些属性和 Delphi 集成开发环境的一些选项记录，有的是关于 Delphi 编译这个工程的指令记录。还有一个文件，Unit1.dfm 文件，它的结构比较简单。等到我们介绍了 Pascal 语言的语法和面向对象的编程思想之后，看懂这个文件就是一件很简单的事情了。它里面无非是记载了一些该 Form 在 Delphi 6.0 设计环境中的一些属性。

到目前为止至少应该知道，一个标准的 Delphi 应用程序至少应该包含一个工程文件和一个定义相关的类的单元文件。

说明：

如果看一下 Delphi 6.0 的菜单，会发现有很多可以新建的项目。据 Inprise 公司自己介绍，现在的 Delphi 6.0 是跨平台的。跨平台的意思不是指我们设计的程序编译后可以在任何程序上运行，比如 Windows 下的程序到 Linux 下运行，而是指在新的操作系统下，只要重新编译一下该程序就可以运行，几乎不需要任何修改工作。

3.1.2 向窗体上添加控件

虽然从上面的介绍中可以看出，当前的工程中已经包含了许多信息，但是它还仅仅是一个空白的工程，因为它除了提供了一个窗口之外，没有提供其他任何有用的东西。

要使一个工程成为利用的程序，需要做两个方面的工作：一是要向窗体（Form）上添加一些内容，使得程序运行时窗体上不会空空如也；二是要在单元文件中加入一定的代码，使得程序能够执行一定的动作。

现在先来完成第一个方面的工作，那就是向窗体上添加一些内容。

单击控件选项板中的 Standard 选项卡，会显示一些标准控件。单击上面的 Button 按钮  ，然后在窗体上按下鼠标，同时拖动鼠标，当所画出代表控件大小的矩形满足要求时，松开鼠

标按钮，此时我们已经为窗体添加了一个 Button 控件，如图 3.6 所示。

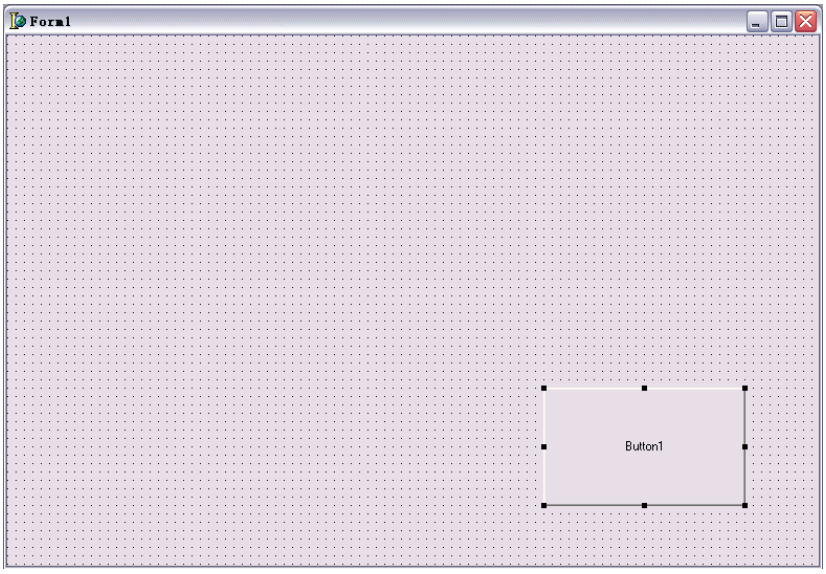

图 3.6　加入了 Button 控件的窗体

还需要设置控件的一些属性，使控件能够显示出所希望的外观。先单击这个控件，此时可以看到，在属性编辑器列表框中出现了这个控件的名称，如图 3.7 所示。

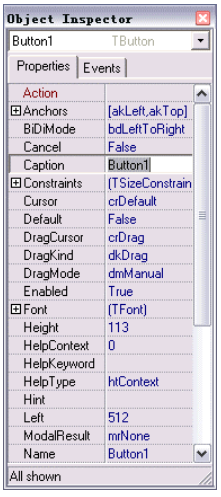

图 3.7　属性编辑器

首先，需要给这个控件取一个具有个性的名字。单击属性编辑器的滚动条，找到 Name 属性，然后把插入点移动到这个属性右边的框中，输入一个名字，比如我们在这里输入“ BtnSayHello ”。这个按钮上显示的文字还不够专业，我们希望能够修改这些文字。那么找到 Caption 属性，然后输入“ Say Hello! ”。

现在已经完成了这个控件的外观设置了。

3.1.3　添加事件处理程序

不说你可能已经猜出来了，这个按钮的作用是当单击它的时候，它能够显示“ Hello! ”文本。那么下面我们就让这个控件能够响应单击事件，并且显示这样的文本。

这个时候需要为这个按钮控件添加事件处理程序。实际上一个事件处理程序和前面介绍的过程完全一样，不过说法不同而已。我们完全可以用我们对过程的理解来理解 Delphi 中的事件处理程序。

Delphi 事件处理程序中一般都包含了一些预定的参数，最常使用的是 Sendor。这些参数

对很好地捕捉一些事件是非常有帮助的，特别是在捕捉鼠标事件和键盘事件的时候。在后面的内容中，我们会涉及这个方面的介绍。

单击属性编辑器的 Events 选项卡，如图 3.8 所示。

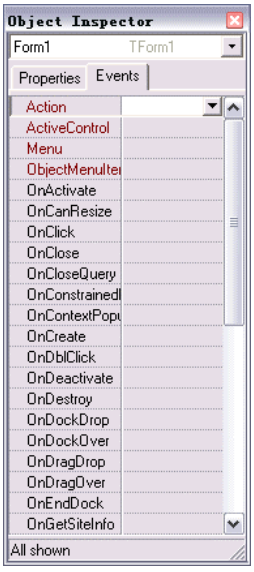

图 3.8　属性编辑器的 Events 选项卡

然后双击 OnClick 右边的文本框。此时会显示代码编辑器，并且自动建立了一个过程的框架，输入下面的代码：

```
showmessage('Hello!');
```

输入了程序代码的代码编辑器如图 3.9 所示。

从上面的图中可以看到，事件处理程序和我们说的过程还不完全类似，它们在定义部分有一个 TForm1 作为前引，这是因为这个过程是在 TForm1 这个类中说明，也就是说，该过程是属于 TForm1 的。如果像下面一样来说明一个过程，那么就不能像上面一样具有一个 TForm1 的前引：

```
unit Unit1;
interface
uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
  Dialogs;
type
  TForm1 = class(TForm)
    BtnSayHello: TButton;
  private
```

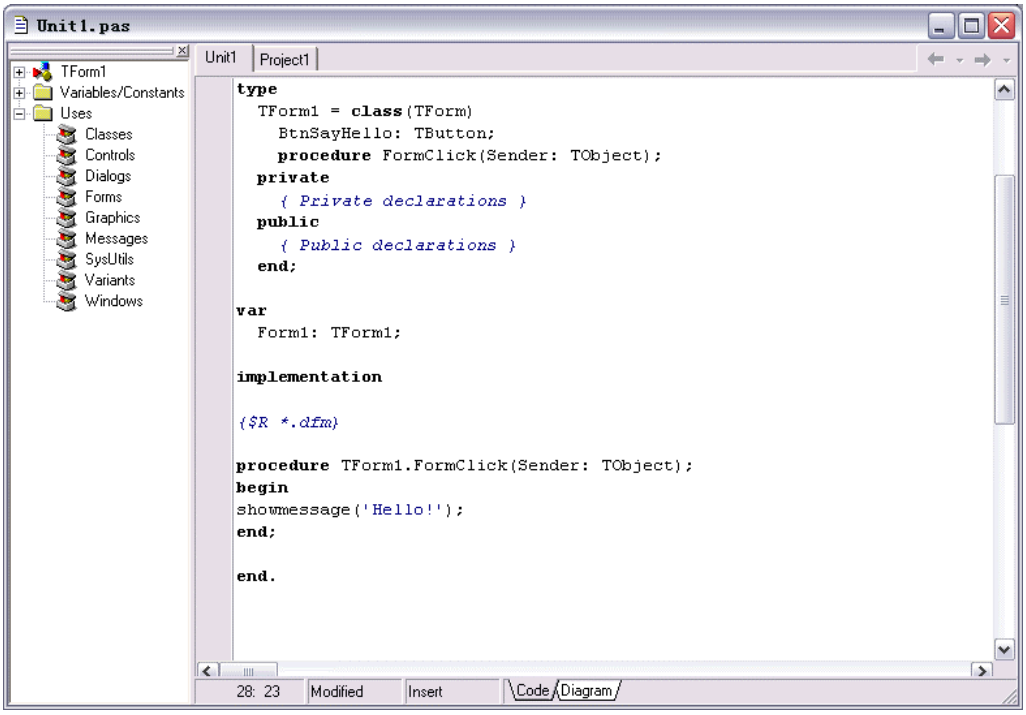

图 3.9　响应单击事件的程序代码

```
    { Private declarations }
  public
    { Public declarations }
  end;
var
  Form1: TForm1;
procedure FormClick(Sender: TObject);
implementation

{$R *.dfm}
procedure FormClick(Sender: TObject);
begin
showmessage('Hello!');
end;
end.
```

3.1.4　运行应用程序

现在，已经完成了全部的工作，可以运行应用程序了。单击工具栏上的 Run 按钮  **，**

或者选择 Run 菜单中的 Run 命令，便可以运行编辑好的应用程序。如果在输入代码的时候出现了错误，那么在这里会进行提示。

当出现了我们设计的包含一个按钮的窗体之后，单击其上的 Run 按钮，程序的运行结果如图 3.10 所示。

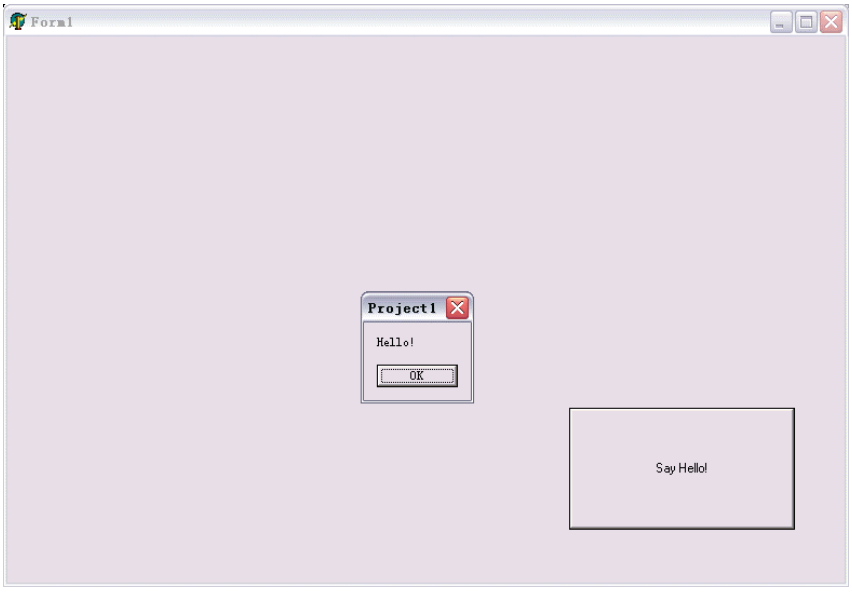

图 3.10　第一个应用程序的运行结果

通过上面的过程，我们建立了一个极其简单的应用程序，但是即使是从上面这样一个简单的程序中，也仍然可以看出，在 Delphi 6.0 中，编写一个应用程序基本包括下面四个主要步骤：

(1) 建立一个新的工程，或者打开已经存在的工程。

(2) 向 Form 上添加适当的控件，并修改控件的属性，使得控件能够表现出我们希望的外观。

(3) 添加控件或者对象的事件处理程序，或者添加一些通用的过程或函数。

(4) 编译并运行工程。

3.2　面向对象编程思想

3.2.1　简述

面向对象的程序设计思想（Object-Oriented Programming，简称为 OOP）是 Delphi 诞生

的基础。OOP 的主要目的是要创建可以重用的代码，具备更好地模拟现实世界环境的能力，这使它被公认为是自上而下编程的优胜者。它通过给程序中加入扩展语句，把函数“封装”进 Windows 编程所必需的“对象”中。面向对象的编程语言使得复杂的工作条理清晰、编写容易。可以说，面向对象编程思想的出现带来了软件开发技术的一场革命，使人们从结构化的编程思想走到了面向对象的编程思想上。但是，要考虑到，我们所说的对象并不与传统的程序设计和编程方法兼容。如果在设计程序的时候，只是部分采用了面向对象的编程思想，可能反而会使情形变得糟糕起来。所以，除非整个开发环境都是面向对象的，否则对象产生的好处还没有带来的麻烦多。而 Delphi 是完全面向对象的，这就使得 Delphi 成为一种触手可及的促进软件代码重用的优秀开发工具，从而具有强大的吸引力。

一些早期的具有 OOP 性能的程序语言，如 C++、Pascal、Smalltalk 等，虽然具有面向对象的特征，但不能轻松地描述出可视化对象，与用户交互的能力比较差，程序员仍然需要编写大量的代码。Delphi 的推出，填补了这项空白。我们可以不必自己辛苦地建立各种对象，只要在提供的程序框架中加入完成功能的代码，其余的部分都可以交给 Delphi 去做。要生成漂亮的界面和结构良好的程序丝毫不必绞尽脑汁，Delphi 将帮助我们轻松地完成。Delphi 允许在一个具有真正 OOP 扩展的可视化编程环境中，使用它的 Object Pascal 语言。这种革命性的组合，使得可视化编程与面向对象的开发框架紧密地结合起来。

在上面的介绍以及前面的介绍中，我们多次提到了对象、属性、方法。也许你仍然对什么是对象、属性、方法以及它们之间的相互关系感到疑惑。但是由于我们在这里的主要目的是介绍 Delphi 6.0 的编程，希望能够和广大使用或者学习 Delphi 的人们探讨 Delphi 编程中的各种问题，所以不可能花费太多的精力用来介绍面向对象的编程思想。但是我们也希望通过这一节的介绍，能够使读者从总体的角度对面向对象的编程思想有一定的了解。这对正确地理解 Delphi 编程中一些规则和方法还是十分有好处的。

3.2.2　基本机制

面向对象的基本机制是对象、消息、类、实例以及继承。所有的真正的面向对象系统都具备这些基本机制，虽然对机制的实现方式可能不完全相同。在这一小节中，我们希望能够为读者提供关于这些机制的比较简要而准确的定义。

1. 对象、消息

在我们以前的结构化编程思想中，一个典型的程序是由过程和数据组成的。而一个面向对象的程序则只由含过程与数据的对象组成。换句话说，对象是包含数据以及对数据进行操作的方法模块。因此，在对象中有着通常语言中的数据（如数组、数、串和记录），以及对这些数据进行操作的函数、过程和事件。所以，对象是有着特殊属性及过程的实体。

与传统系统中的被动数据不同，对象具有行动的能力。例如我们在上面的程序中使用了

一个 Button 控件，这里说的控件实际上就是一类特殊的对象。它能够响应我们的单击事件，不仅在外观上给我们以被按下的感觉，而且可以根据我们的定义显示一个信息对话框。对象的行动能力还表现为它的自我描述功能。例如当我们改变 Button 控件的一个属性（Captions），它会对自己进行相应的描述（把 Button 控件上的文字换成了我们需要的文字）。

那么是什么使得对象具有了行动的能力呢？应该说是组成对象的属性和方法。可以说对象的属性就是包含在对象中的一类特殊数据（应该强调的是属性是数据），它代表了对象的某类特征。比如在上面的例子中，Button 控件的 Caption 属性代表的是显示在控件上的文字。那么改变这个属性也就改变了控件上的文字。另外一个问题就是，当我们改变了这个属性的时候，是什么使得控件上的标题发生了变化？在这个时候要提到对象的方法了。是对象的方法，把对象在外观上进行了刷新，所以说方法是对象操作数据的主要工具。

回顾上面的例子，可以看到，当我们用鼠标单击按钮的时候，按钮会先呈现按下的状态，然后恢复原来的凸起状态，这代表了控件的一种典型的行为方式。简单来说，就是鼠标单击控件的时候，相当于向控件发送了一个消息（Message）。这个消息启动了控件的对应方法，使得控件表现出上面的动态行为。

2. 类、实例

在上面我们提到的是对象本身的一些表现以及其中的一些特性。但是，从编程的角度来说，如何来定义和描述一个对象呢？这里需要使用到类和子类的概念。类是一组几乎相同的对象的描述，是由概括了一组对象共同性质的方法和数据（包括属性）组成的。从一组对象中抽象出公共方法和数据并将它们保存在一类中是面向对象功能的核心。对类的定义意味着将可用代码放在公共区中，而不是一遍又一遍地对之加以表示。换句话说，类是创建对象的蓝图、模板。当我们用类来创建一个对象的时候，实际上是生成了类的一个实例。根据我们的需要把这个实例的对应数据赋予相应的值，便创建了一个我们需要的、在一定程度具有自己个性的对象了。

类可以具有子类，可以为一组子类概括公共元素。子类可以直接从父类中继承一些方法和属性。通过使用子类，面向对象的程序员就可以将应用程序描述成一组通用或抽象模块。

3. 封装

对对象最基本的理解是把数据和代码组合在同一个结构中，这就是对象的封装特性。将对象的数据域封闭在对象的内部，使得外部程序必须而且只能使用正确的方法才能对要读写的数据域进行访问。封装性意味着数据和代码一起出现在同一结构中，如果需要的话，可以在数据周围砌上“围墙”，只有用对象类的方法才能在“围墙”上打开缺口。

4. **继承**

继承是面向对象编程思想中的一个重要的概念，它是自动地共享各个类、子类中的方法和数据的有效机制。作为过程式系统中没有的一个有力机制，继承使得我们可以通过从父类中继承公共方法和数据，而通过添加新的数据和方法定义成一个新的类。其后，又可以通过把新创建的子类作为父类，来创建它的新的子类。

从上面的描述中可以看出，面向对象的编程中，类的创建是沿着树形的结构发展下来的。正是通过继承的机制，实现了编写的代码重用，不仅节省了重复编码的时间，而且可以逐渐地完善类，直到创建出满足我们需要的各种类。

5. **多态性**

多态性是在对象体系中把设想和实现分开的手段。如果说继承性是系统的布局手段，多态性就是其功能实现的方法。多态性意味着某种概括的动作可以由特定的方式来实现，这取决于执行该动作的对象。多态性允许以类似的方式处理类体系中类似的对象。根据特定的任务，一个应用程序被分解成许多对象，多态性把高级设计处理的设想如新对象的创建、对象在屏幕上的重显、程序运行的其他抽象描述等，留给知道该如何完美地处理它们的对象去实现。

3.3　Delphi 中的面向对象编程

我们知道，Delphi 是基于面向对象编程的先进开发环境。面向对象的程序设计（OOP）是结构化语言的自然延伸。OOP 的先进编程方法，会产生一个清晰而又容易扩展及维护的程序。一旦我们为自己的程序建立了一个对象，我们自己和其他的程序员可以在其他的程序中使用这个对象，完全不必重新编制繁复的代码。对象的重复使用可以大大地节省开发时间，切实地提高工作效率。

3.3.1　通过 Delphi 实例了解 Delphi 对象

让我们结合 Delphi 的实例讨论对象的概念。

当我们要建立一个新工程时，Delphi 将显示一个窗体作为设计的基础，这一点我们在上面的示例中已经看到了。在代码编辑器中，Delphi 将这个窗体说明为一个新的对象类型，并同时在与窗体相关联的库单元中生成了创建这个新窗体对象的程序代码。

```
unit Unit1;
interface
uses  SysUtils, Windows, Messages, Classes, Graphics, Controls, Forms, Dialogs;
```

```
type
    TForm1 = class(TForm)    {窗体的类型说明开始}
    private
    { Private declarations }
    public
    { Public declarations }
    end;                {窗体的类型说明结束}
var
    Form1: TForm1;   {说明一个窗体变量}
implementation
{$R *.DFM}
end.
```

新的窗体类型是 TForm1**，它是从** TForm **继承下来的一个对象。它具有对象的特征：含有属性或方法。由于没有给窗体加入任何部件，所以它只有从** TForm **类中继承的属性和方法，在窗体对象的类型说明中，我们是看不到任何属性、方法和说明的。**Form1 **称为** TForm1 **类型的实例。我们可以说明多个对象类型的实例，例如在多文档界面**(MDI)**中管理多个子窗口时就要进行这样的说明。每一个实例都有自己的说明，但所有的实例却共用相同的代码。**

例如在上面的示例中，向窗体中加入了一个按钮部件，并对这个按钮建立了一个 OnClick **事件处理过程。这个时候，如果再来查看** Unit1 **的源代码的话，将会发现** TForm1 **的类型说明部分如图** 3.11 **所示。**

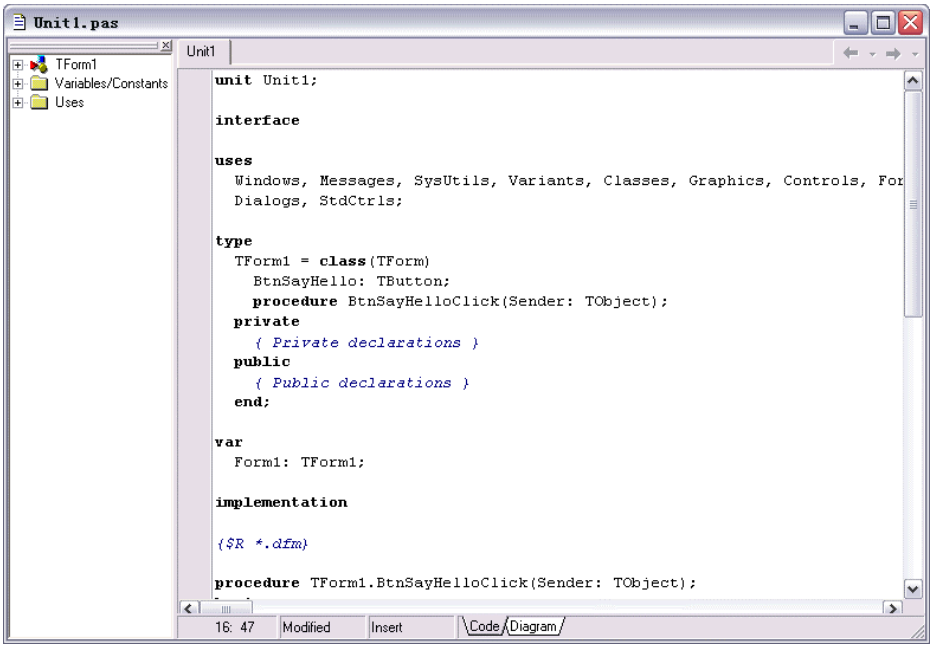

图 3.11 **加入了控件后的对象声明**

现在 TForm1 对象有了一个名为 BtnSayHello 的数据，它是我们在窗体中加入的按钮。TButton 是一个对象类型，BtnSayHello 是 TButton 的一个实例。它被 TForm1 对象所包含，作为它的数据域。每当在窗体中加入一个控件时，控件的名称就会作为 TFom1 的数据域加入到类型说明中来。在 Delphi 中，我们所编写的事件处理过程都是窗体对象的方法。每当我们建立一个事件处理过程，就会在窗体的对象类型中说明一个方法。

当使用属性编辑器来改变对象（控件）的名称时，这个名称的改变会反映到程序中。例如，在属性编辑器中将 Form1 的 Name 属性命名为 ColorBox，会发现在类型说明部分，前文的 TForm1 改为：

```
TColorBox=class(TForm);
```

并且在变量说明部分，会说明 ColorBox 为 TColorBox 类型的变量，由 Delphi 自动产生的事件处理过程名称会自动改为 TColorBox.Button1Click；但我们自行编写的实现部分的代码却不会被自动修改。因此，如果在改变 Name 属性前编写了程序，那么必须改变事件处理过程中的对象名称。所以，原先的 Form1.Color 要改为 ColorBox.Color。

3.3.2　继承数据和方法

前面的 TForm1 类型是很简单的，因为它只含有域 BtnSayHello 和方法 BtnSayHelloClick。对于一个只包含一个域和方法的对象来讲，我们并没有看到显式的支持程序。在窗体上单击鼠标或用属性编辑器上端的 Object Selector 选中 Form1 对象，按 F1 查阅它的在线帮助，会在 Properties 和 Method 中找到它继承到的全部属性和方法。这些是在 TForm 类型中说明的，TForm1 是 TForm 的子类，直接继承了它所有的数据域、方法、属性和事件。例如窗体的颜色属性 Color 就是在 TForm 中说明的。当我们在工程中加入一个新窗体时，就等于加入了一个基本模型。通过不断地在窗体中加入部件，就自行定义了一个新的窗体。要自定义任何对象，都将从已经存在的对象中继承域和方法，建立一个该种对象的子类。

一个比较特殊的对象是从一个范围较广或较一般的对象中继承下来的，这个范围较广或较一般的对象是这个特别对象的祖先，这个特别对象则称为祖先的后代。一个对象只能有一个直接的祖先，但是它可以有许多后代。TForm 是 TForm1 类型的祖先，所有的窗体对象都是 TForm 的后代。

图 3.12 是 Delphi 的 VCL（Visual Component Library）结构简图。

在这个结构中所有的组件都是对象。组件类型 TComponent 从 TObject 类型中继承数据和程序代码，并具有额外的可以用作特殊用途的属性、方法、事件，所以控件可以直接和用户打交道，记录它的状态并存贮到文件中等等。控件类型 TControl 从 TComponent 中继承而来，又增加了新的功能，如它可以显示一个对象。在上图中，虽然 TCheckBox 不是直接由 TObject 继承来的，但是它仍然有任何对象所拥有的属性，因为在 VCL 结构中，TCheckBox 终究还是从 TObject 中继承了所有功能的特殊对象，但它还有些自行定义的独到功能，如可以选择

记录状态等。

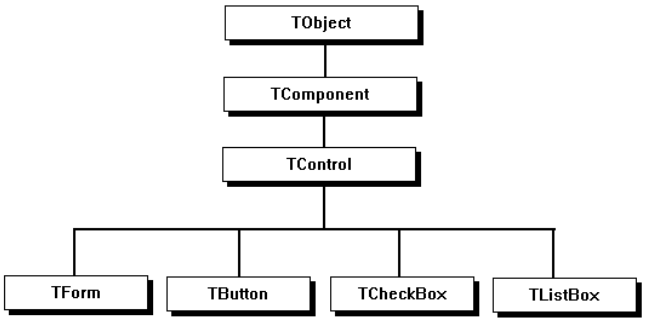

图 3.12 简化的 Delphi VCL 结构图

3.3.3 对象的范围

1. 关于对象的范围

一个对象的范围决定了它的数据域、属性值、方法的活动范围和访问范围。在一个对象的说明部分说明的数据域、属性值、方法都只是在这个对象的范围中，而且只有这个对象和它的后代才能拥有它们。虽然这些方法的实际程序代码可能是在这个对象之外的程序库单元中，但这些方法仍然在这个对象的范围内，因为它们是在这个对象的说明部分中说明的。

当我们在一个对象的事件处理过程中编写程序代码来访问这个对象的属性值、方法或域时，不需要在这些标识符之前加上这个对象变量的名称。例如，如果在一个新窗体上加入一个按钮和一个编辑框，并为这个按钮编写 OnClick 事件处理过程：

```
procedure TForm1.Button1Click(Sender:Tobject);
begin
   Color :=clFuchsia;
   Edit1.Color :=clLime;
end;
```

其中的第一行语句是为整个窗体 Form1 着色。我们也可以编写如下：

```
Form1.Color :=clFuchsia;
```

但可以不必加上 Form1.，因为 Button1Click 方法是在 TForm1 对象的范围里。当我们在一个对象的范围中时，可以省略所有这个对象中的属性值、方法、域之前的对象标识符。但是当编写第二个语句改变编辑框的底色时，因为此时想访问的是 TEdit1 对象的 Color 属性，而不是 TForm1 类型的，所以需要通过在属性前面加上编辑框的名称来指明 Color 属性值的范围。如果不指明，Delphi 会像第一个语句一样，将窗体的颜色变成绿色。因为 Edit1 部件

是在窗体中的，它是窗体的一个数据域，所以我们同样不必指明其从属关系。

如果 Edit1 是在其他窗体中，那么需要在编辑框之前加上这个窗体对象的名称。例如，如果 Edit1 是在 Form2 之中，那它是 Form2 说明的一个数据域，并位于 Form2 的范围中，那么需要将第二句改为：

```
Form2.Edit1.Color := clLime;
```

而且需要把 Unit2 加入到 Unit1 的 uses 子句中。

一个对象的范围扩展到这个对象的所有后代。TForm 的所有属性值、方法和事件都在 TForm1 的范围中，因为 TForm1 是 TForm 的后代。我们的应用程序不能说明和祖先的数据域重名的类型、变量等。如果 Delphi 显示了一个标识符被重复定义的信息，就有可能是一个数据域和其祖先对象（例如 TForm）的一个数据域有了相同的名称。可以尝试改变这个标识符的名称。

2. 重载一个方法

可以重载（Override）一个方法。通过在后代对象中说明一个与祖先对象重名的方法，就可以重载一个方法。如果想使这个方法在后代对象中做和祖先对象中一样的工作但是使用不同的方式时，就可以重载这个方法。Delphi 不推荐我们经常重载方法，除非想建立一个新的部件。重载一个方法，Delphi 编译器不会给出错误或警告提示信息。

3.3.4　公有域和私有域

当使用 Delphi 的环境来建立应用程序时，可以在一个 TForm 的后代对象中加入数据域和方法，也可以通过直接修改对象类型说明的方法来为一个对象加上域和方法，而不是把一个部件加入窗体或事件处理过程中。

可以在对象的 Public 或 Private 部分加入新的数据域和方法。Public 和 Private 是 Object Pascal 的保留字。当我们在工程中加入新的窗体时，Delphi 开始建立这个新窗体对象。每一个新的对象都包含 Public 和 Private 指示，以便在代码中加入数据域和方法。在 Public 部分中说明其他库单元中对象的方法也可以访问的数据域或方法。在 Private 部分的说明有访问的限制。如果在 Private 中说明域和方法，那么它在说明这个对象的库单元外是不透明的，而且不能被访问。Private 中可以说明只能被本库单元方法访问的数据域和本库单元对象访问的方法。过程或函数的程序代码可以放在库单元的 Implementation 部分。

3.3.5　访问对象的域和方法

当想要改变一个窗体对象的一个域的某个属性，或是调用它的一个方法时，必须在这个属性名称或调用方法之前加上这个对象的名称。例如，如果窗体上有一个编辑框部件，而我

们需要在运行中改变它的 Text 属性，需要编写下列的代码：

```
Edit1.Text := 'Welcome to Delphi';
```

同样，清除编辑框部件中选中的文本，可以调用 TEdit 部件的相应方法：

```
Edit1.ClearSelection;
```

如果想改变一个窗体对象中一个对象域的多个属性或调用多个方法时，使用 with 语句可以简化程序。with 语句在对象中可以和在记录中一样方便地使用。下面的事件处理过程在响应 OnClick 事件时，会对一个列表框做多个调整：

```
procedure TForm1.Button1Click(Sender:TObject);
begin
      ListBox1.Clear;
      ListBox1.MultiSelect :=True;
      ListBox1.Item.Add('One');
      ListBox1.Item.Add('Two');
      ListBox1.Item.Add('Three');
      ListBox1.Sorted :=Ture;
      ListBox1.FontStyle :=[fsBold];
      ListBox1.Font.Color :=clPurple;
      ListBox1.Font.Name :='Times New Roman';
      ListBox1.ScaleBy(125,100);
end;
```

如果使用了 With 语句，则程序如下：

```
procedure TForm1.Button1Click(Sender:TObject);
begin
  with (ListBox1) do
  begin
      Clear;
      MultiSelect :=True;
      Item.Add('One');
      Item.Add('Two');
      Item.Add('Three');
      Sorted :=Ture;
      FontStyle :=[fsBold];
      Font.Color :=clPurple;
      Font.Name :='Times New Roman';
      ScaleBy(125,100);
  end;
end;
```

使用 with 语句，不必在每一个属性或方法前加上 ListBox1 标识符，在 With 语句之内，所有的属性或调用方法对于 ListBox 这个对象而言都是在它的范围内的。

3.3.6　对象变量的赋值

如果两个变量类型相同或兼容，则可以把其中一个对象变量赋给另一个对象变量。例如，对象 TForm1 和 TForm2 都是从 TForm 继承下来的类型，而且 Form1 和 Form2 已被说明过，那么可以把 Form1 赋给 Form2:

```
Form2 :=Form1;
```

只要赋值的对象变量是被赋值对象变量的祖先类型，就可以将一个对象变量赋给另一个对象变量。例如，下面是一个 TDataForm 的类型说明，在变量说明部分一共说明了两个变量：AForm 和 DataForm。

```
  type
TDataForm = class(TForm)
  Button1:TButton;
  Edit1:TEdit;
  DataGrid1:TDataGrid;
  Database1:TDatabase;
  TableSet1:TTableSet;
  VisibleSession1:TVisibleSession;
private
  {私有域说明}
public
  {公有域说明}
end;
var
  AForm:TForm;
  DataForm:TDataForm;
```

因为 TDataForm 是 TForm 类型的后代，所以 Dataform 是 AForm 的后代，因此下面的赋值语句是合法的：

```
AForm :=DataForm;
```

这一点在 Delphi 中是极为重要的。让我们来看一下应用程序调用事件处理过程的过程，下面是一个按钮部件的 OnClick 事件处理过程：

```
procedure TForm1.Button1Click(Sender:TObject);
begin
end;
```

在图 3.12 中可以看到 TObject 类在 Delphi 的 Visual Component Library 的顶部，这就意味着所有的 Delphi 对象都是 TObject 的后代。因为 Sender 是 TObject 类型，所以任何对象都可以赋值给它。虽然我们没有看见赋值的程序代码，但事实上发生事件的部件或控制部件已经赋给 Sender 了，这就是说 Sender 的值是响应发生事件的部件或控制部件的。

可以使用保留字 is 来测试 Sender 以便找到调用这个事件处理过程的部件或控制部件的类型。

3.4 如何编写一个好的程序

在下面的章节中，将讨论如何创建有效的程序。读者将会发现这一部分与本书其余章节的主题不同，它的技术性不是特别强。但是，我们认为这部分内容对读者，特别是正在学习 Delphi 编程的人来说是十分重要的。

首先，我们需要明确一下作为一个程序员需要注意的三件事：

- 在编写程序之前先进行良好的规划，也就是说，在开始编写程序之前，应该对我们要设计的程序有一个建设性的设计方案。在编写程序的过程中，可以重复改进这个设计方案。
- 弄清楚 OOP 的基本概念，特别是封装，接口设计，以及信息隐藏。
- 尽量使用控件。

简而言之，这三个重要事项就是规划、OOP 和组件。我们将在下面的内容中分别讨论这几个问题。

3.4.1 书写尽可能简单的代码

“简单性是这个世界上最难获得的东西；它是经验的最终界限，也是天才的最终努力目标。”、“简单的就是完美的”，这些名言都说明，“简单”在工作生活中的重要作用。

这个观点可能和许多人的思想相矛盾，但是，我们要说的不是只编写简单的应用程序，这样做的话就没有必要来研究编程的思想了。我们所强调的是尽一切所能让程序尽可能的简单、易懂。我们在刚开始学着编写一些稍微复杂的程序时，总是会编写出长篇累牍的程序，以致于后来需要阅读并修改这些程序的时候，连我们自己也不知道自己所编写的程序的关键技术在什么地方了。

编程是一个创造性的工作，许多的人如痴如醉地沉迷于开发工作是因为他们从中体会到创作的快乐。那么，在这个类似于艺术的创造性工作中，最基本的原则就是简单性。在编写程序的过程中，几乎所做的每一步都是简单的。说到底，编程是一个非常简单的职业。之所以看起来困难，是因为它需要把许多非常简单的过程组合起来。两个简单的事情要比一件简

单的事情稍微复杂一点。二十件简单的事情就明显要比一件简单的事情复杂多了。对于大多数人来说，把一万件简单的事情放在一起就很复杂了。

在一个大型的、复杂的程序中，成百上千个独立的片段结合在一起组成一个整体。如果看得足够透彻的话，即使是一个简单的 Delphi 窗体也是由上千个复杂的片段组成的。作为程序员，目标是要合理地协调所有这些不同的片段，使它们能够和谐地一起工作。所有的程序都应该努力遵循简单性的原则。而且，编程的工作也应该简单，并正朝着简单的方向发展。可以说，我们现在面临的编程的复杂性，比几年前要简单多了。

需要再次强调的是，我们说的简单，不是以牺牲我们要编写的应用程序的功能为代价的。不能通过减少应用程序功能的方法来使我们的应用程序简单起来，而是需要通过合理地运用目前的编程技术，使得我们的程序简单易懂。

要编写尽可能简单的应用程序可不是一件容易的事情，它需要我们花费相当大的精力来研究如何才能实现简单化的目的。这就需要在开始编写程序之前进行很好地规划，拿出最合理的设计方案，充分利用面向对象编程的精髓——代码的重用，在实现同样的功能的前提下，使我们的程序尽可能显得简单一些。

3.4.2　编写适当的测试程序

虽然我们一再强调要在设计程序之前进行很好的规划，要有一个比较合理的设计方案，但是，我们不能想象，通过开始的这些规划就能够找到设计所需要的程序的最好办法，肯定要经历一个边编写边修改方案的过程。

有时，如果不专门编写一些测试代码，我们就无法把设计过程进行下去。在这样的情况下，应该进行一些实验，来测试我们的方案在理论上的有效性。这个过程是不断循环的，直到我们成功地结束了编写任务为止。

这个过程最后应该像一个螺旋那样最终结束。从底端的一个大圈开始，然后不断重复编码和设计工作，从而使工作的圈子越来越小。每个循环都应该比上一个更小，直到最终达到顶端。在顶端，圈子可能已经小到了一个优美的点，于是，就确定了程序的正确实现方法。

3.4.3　合理使用 OOP

在前面我们一再强调，要尽可能编写简单的代码。尽量使我们的代码建立在一个清楚、简单的体系结构的基础上。但是，问题是如何才能做到简单呢？答案就是使用 OOP 技术。

但是我们也要注意到，合理地使用 OOP 技术不是一件简单的事情。如果使用得不好，不仅不能起到简化代码的作用，可能恰恰相反，会使得代码晦涩难懂。所以说，OOP 是一把双刃剑。

下面看一下这两行代码：

```
printf ("Foo");
WriteLn ( 'Foo' );
```

虽然还没有介绍 Pascal 语言的语法，但是，从这两句代码上，你肯定可以猜出它们的作用是什么。这就是优秀代码的特征：即使一个不是程序员的人也能够明白它们要做什么，编程世界中的所有人都明白它们是如何工作的，以及它们为什么要这样工作。

然后再来看一下下面的类的声明：

```
TBird = class(TObject)
Private
    FName:string;
    Fweight:integer;
    FColor:TColor;
      Public
          Procedure Fly;
          Procedure Walk:
      End;
```

从代码中可以看出，我们的目的是声明一个“鸟”类，它能够反映鸟的一些基本特征，能够完成鸟的一些基本动作。从上面的声明中，我们体会到的是简单、清楚、易于理解，并且非常强大。它把一系列特殊的特征联系到一起，放在一个易于使用的抽象的整体中。

要想弄清楚如何和对象一起工作，必须了解面向对象编程思想的封装和继承，最好还能了解多态性。但最重要的事情是要学会如何正确地组织对象。面向对象的程序的主要任务是发现和整理对象。当我们开始编写代码时，主要目标是找出工程中的对象，并把它们表现出来。如果能够在工程中找到对象，那么通常就能够在很短的时间里完成一个通用性很好的程序。

在程序中发现对象并不是一件多么困难的事情。例如，在 Delphi 中，很明显需要用对象来包装基本 Windows 组件，例如 TEdit 和 TButton。而一些核心的 VCL 类就不是那么明显了，例如 TApplication、TControl、TWinControl、TDataSet 和 TPersistent。

3.4.4 简短的方法

为了使事情简单，应该养成编写简短的方法的习惯。并非所有方法都应该是短小的，但大多数应该尽量简短。

一个方法中也有可能包含多个需要独立出来的方法，这一点同样重要。努力找出隐藏在代码中的方法，就像努力找出代码中隐藏着的对象一样。

一个方法太大的最明显的特征表现为这个方法中包含有一些会被两个不同的方法调用的逻辑。例如，我们经常要写一些用来将路径指到应用程序根目录的代码。有时需要在程序的几个不同地方重复使用这个逻辑。在每个需要这些信息的方法中重新编写这一代码是非常

繁琐的。最简单的做法是把这个逻辑从这些方法中分离出来，然后把它封装在一个单独的方法中，然后可以从不同的地方调用这个方法。这意味着每个过去包含有这段逻辑的方法都会变小，因为它现在可以把这个任务指派给另一个方法去完成。

例如，我们可能在编程的过程中编写了下面的两个过程：

```
procedure TForm1.MethodOne;
var
    Path:string;
begin
    Path := ExtractFilePath(ParamStr(0));
    if Path[length(Result)]<>'\' then
        Path := Path + '\';
    … //Code omitted here that manipulates the path…
end;
procedure TForm1.MethodTwo;
var
    Path:String;
begin
    result := ExtractFilePath(Paramstr(0));
    if result[length(Result)] <> '\' then
        Result := Result + '\';
    …// Code omitted here that manipulates the path…
end;
```

可以看出，这两个过程都有重复的代码需要分离到一个单独的方法中，如下所示：

```
function TForm1.GetStartDir:string;
begin
    Result := ExtractFilePath(ParamStr(0));
    if Result[length(Result)] <> '\' then
        Result := Result + '\';
end;
```

最终我们把这个方法移到了一个包含有一系列例程的单元中，多个程序重复使用了这些例程。

如果在编程过程中动一点脑筋，可能会很容易地找出像 GetStartDir 这样的方法来。总的来说，建立易于理解和调试的简短方法是一个好主意。虽然有时冗长的方法也是必需的或是合理的，但它们很可能成为无数程序错误的主要根源。

3.4.5　变量、函数以及过程的命名

在编写代码的过程中，不可避免地要声明许多的变量、函数和过程。那么如何为它们起

一个合理的名字就成为我们在这里要讨论的话题。

我们认为最好给变量、函数以及过程起一些清楚的、易于理解的名字。使用缩写和一般的、无特征的名字可能会给我们带来许多麻烦。

在本书的例子中，经常可以看到，我们给对象起了像 TMyGrid 或 TMyEdit 这样的一般性的名字。但是，在复杂的程序中，这种命名方式也是不可取的。要根据变量、对象、函数、过程的目的和用途，为它们起一个合乎逻辑的名称。同时，我们也希望能够从它的名称中看出这是一个什么类型的控件。例如，TMyGrid 可能会变成 TDailyEarningsGrid 或 TMilesTraveledGrid。虽然键入长一点的名字可能要多花一点工夫，但它有助于我们对程序的理解和后期对程序的维护。

Delphi 程序员倾向于支持特定的命名协定。例如，对象中的私有数据通常加上一个字母 F 作为前缀，就像 FWidth 或 FHeight。这个字母表明该变量是对象中的一个私有数据。在程序中声明的类型应该使用一个字母 T 作为前缀，它代表 type。例如，关键 Delphi 对象具有像 TObject, TComponent 和 TEdit 这样的名字。像 Integers 和 Words 这样的一些古老类型在这个惯例建立之前就已经存在了，但最近 5 到 10 年来创建的大多数类型都遵循这个惯例，因此本书使用的所有程序中都遵循这个惯例。

3.4.6 创建控件

在 Delphi 中，我们应该大力推广的，用于创立简单性好、通用性强的应用程序的主要方法可能就是创建控件了。这是因为使用控件具有以下优点：

- 根据它的特性，我们所建立的每个控件都是独立的，独立于任何其他对象。这样，可以很容易地测试对象并重复使用它。
- 建立的每个控件必须能够通过属性编辑器进行操作。这种限制迫使我们为每个对象建立简单、易用的接口。
- 控件可以帮助我们设计程序。这从上面的示例程序中可以看出来，使用 Tbutton 控件使得我们创建一个像上面那样的界面是多么的容易。

3.5 本 章 小 结

在本章中，试图通过一个简单的例子，让读者了解在 Delphi 中面向对象编程的含义，当然就用一章的内容来介绍这个问题是无法把它全面地解释清楚的。但是最少可以建立一些基本的概念，然后在后面的过程中将会获得更深刻的体会。

第 4 章　控件、工程和应用程序

在正式编写 Delphi 程序之前，应该了解一下 Delphi 关于控件、工程和应用程序的相关设置选项。这些设置非常重要，它们直接关系到最终结果的样子。在设计程序的时候，我们通常会在窗体上添加很多控件，特别是其中的一些可能会显示成一组的形式，那么如何处理它们的相对位置将会影响程序的外观。当然，由于 Delphi 6.0 是一个所见即所得的设计环境，所以可以通过鼠标来调整它们，但是有的时候，这样做是比较麻烦的，特别是在控件比较多的情况下。Delphi 提供了一些处理这个问题的工具，利用它们可以帮我们非常方便地安排控件。

工程是我们的程序，是我们在 Delphi 中要处理的一个基本的问题。而应用程序是工程的编译结果。关于它们的许多选项会直接影响程序的许多属性，比如应用程序的图标，等等。

在本章中，将主要介绍三个方面的内容：

- 窗体上的控件位置
- 对象的属性和事件
- Delphi 的工程管理

4.1　窗体上的控件

4.1.1　在窗体上放置控件

在前面的例子中已经介绍了如何在一个窗体上添加一个控件。这个过程是比较简单的，只要单击控件选项板上所需要的选项卡，然后单击需要的控件，这个时候，可以选择三种方式把这个控件放置到窗体上。

(1) 在窗体上的适当位置按下鼠标左键，拖动鼠标，画出一个代表控件大小的矩形，然后松开鼠标。这样，就在窗体上放置了一个对应的控件。但是，这个方法只适用于那些可视控件，比如 TButton。对于一些在运行期不可见的控件来说，这样做不会起到任何作用，Delphi 仍然会生成一个默认大小的控件（比如 Timer 控件）。

(2) 在窗体上单击鼠标左键，Delphi 会以默认的大小创建这个控件，控件的位置定位在单击鼠标的地方。

(3) 双击控件选项板上的控件，这样就会以默认的大小在窗体的中央位置创建这样的一个控件。

在把控件放置到窗体上之后，我们就可以利用两种方法来调整它的大小和位置。一种方法是利用鼠标或者键盘的操作，直接改变控件的位置或者大小。这个操作方法在介绍窗体设计器的时候已经作了介绍。另一个方法是修改这个控件的位置和大小属性。几乎每个控件都包含四个和位置、大小相关的属性，它们是：Left、Top、Width 和 Height。它们都是整数型的数据，可以直接在属性编辑器上修改这些属性，然后把插入点移动到其他属性或者窗口中。此时，窗体设计器上的控件就会根据这些属性进行自动刷新。

4.1.2 对齐多个控件

我们在使用 Delphi 进行编程的时候，在大多数情况下不会只使用一个控件。而且，在有些情况下，还需要使用多个同类的控件。比如，要建立一个登录窗口，如图 4.1 所示。

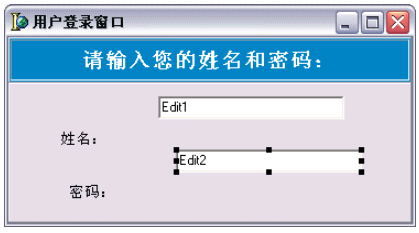

图 4.1 使用多个同类控件的窗体

在图 4.1 的窗体中，我们放入了两个 TPanel 控件、两个 TLabel 控件和两个 TEdit 控件。并对它们的属性进行了相应的设置。关于如何设置它们的属性的方法，请读者参考本书后面的内容。从图中可以看出，这些控件的位置是需要调整的。我们当然可以利用调整单个控件的方法来调整各个控件，使得它们能够对齐在相应的位置，但是这未免太麻烦了。在这里也有两种方法可供选择：

(1) 选择需要设置相对关系的所有控件，然后利用属性编辑器，修改它们共同的位置和大小属性。

(2) 利用 Delphi 在 Edit（编辑）菜单和 View（视图）菜单中提供的对齐工具。

例如对于上面的窗口，通过利用 Delphi 的 Alignment Palette（对齐面板）窗口，可以把它们安排成如图 4.2 所示的效果。

图 4.2 在安排了相对位置之后的登录窗口

另外，还可以像复制文本那样，把一个或者多个控件复制到剪贴板上，然后粘贴到多个窗体上。在创建同类风格的窗口或者布局的时候，这个方法是十分有效的。

4.1.3　容器、父控件和子控件

在讲述窗体和控件、控件和控件的关系时，经常会提到容器、父控件和子控件。在介绍这些关系之前，应该首先解释一下这些名词的含义。当我们把一个控件或者对象用来作为放置其他控件或者对象的场所时，就把这个控件或者对象称为容器。如果这个容器是一个控件，通常我们也把这个控件称为父控件，而把放置在其中的控件称为子控件。

所以，我们在 Delphi 中使用的 Form（窗体）（从本质上来说是一个控件）实际上是用来放置其他控件的一个容器。可以这样说，如果不能在窗体上放置任何控件的话，这个窗体基本上就没有任何作用。

在许多情况下，需要把窗体上的控件进行分组，也就是说，不要把它们一股脑地、同等地放置在一个窗体上。可以通过使用父控件来达到这个目的。通过父控件来把窗体上的控件进行分组是一个相当不错的主意。在 Delphi 中，作为容器的对象或者控件具有以下的一些特性：

- 当我们在一个容器中放置控件的时候，子控件的位置坐标是相对于容器而不是窗体来设定的。也就是说，当我们移动容器的时候，可以同时移动上面的子控件。比如，窗体作为最高级别的容器，当我们移动它的时候，它上面的控件也随之发生了移动。
- 对于容器来说，子控件成为它的一个部分，这使得我们可以通过改变容器的一些属性来影响子控件的行为。比如，对于上面介绍的窗体来说，如果修改子控件所在的 TPanel 控件的 Font 属性，就可以影响它上面的所有控件的 Font 属性，如图 4.3 所示。

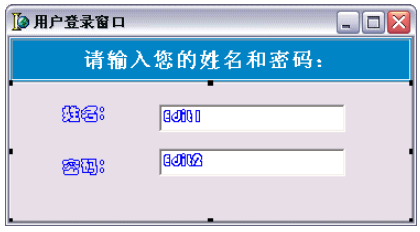

图 4.3　修改父容器的属性直接影响上面的控件属性

说明：

几乎所有的可视控件都有四个 Parent 属性，它们分别是：ParentBiDiMode、ParentColor、ParentFont、ParentShowHint。它们都是 Boolean 型的属性。第一个属性代表控件是否具有和其容器（也就是父控件）相同的 BiDiMode 属性，这个属性决定了如何在窗体上显示文本和垂直滚动条；第二个属性代表是否采用和父控件相同的颜色方案，比如背景的颜色；第三个

属性表示是否采用和父控件相同的字体；第四个属性表示是否和父控件采用同样的提示显示功能。

- 当我们删除、复制作为容器的控件时，同时也删除、复制了容器所包含的控件。这一点需要特别小心。

4.2 Delphi 工程中的窗体

到目前为止，还没有介绍过一个工程中出现多个窗体的问题。但是在实际的程序开发过程中，使用多个窗体的情况是屡见不鲜的。回顾一下我们在使用 Windows 应用程序的时候，所遇到的应用程序有多少是只有一个窗口的呢？这个数量和具有多个窗口的应用程序比起来，简直是沧海一粟。

4.2.1 向工程中加入新的窗体

如果要向工程中加入新的窗体，则可以使用 Delphi 的 New Form（新建窗体）命令，该命令位于 File（文件）菜单上。

如果在添加窗体的时候，希望加入的不是一个空白窗体，而是具有一定内容的窗体，则可以使用 Delphi 的对象库来达到这个目的。

选择 Tools(工具)菜单中的 Repository(库)命令,会打开如图 4.4 所示的 Object Repository（对象库）对话框。

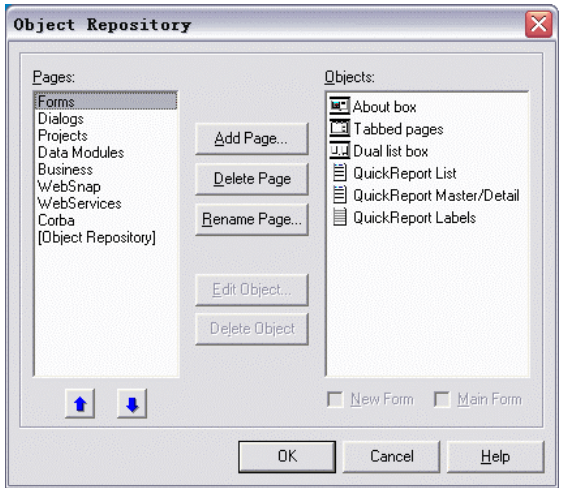

图 4.4 Object Repository 对话框

在这个对话框的左边是一些可以提供的对象类别，而右边就是每个类中提供的对象。可

以通过这个对话框，选定我们在选择 New Form 命令的时候，添加的窗体形式。例如可以选择 Forms，这时会在右边的列表框中显示出它所包含的对象。从中选择一个，然后选中下面的 New Form 复选框，表示创建新的窗体的时候，就以这个对象为模板添加新的窗体。如果选中 Main Form（主窗体）复选框，则表示用当前选定的对象作为一个新工程的主窗体。

比如，选定 Dual List Box（双列表框），然后选中 New Form 复选框。单击 OK 按钮，返回 Delphi 的集成开发环境。如果此时再单击 New Form 命令，将会添加一个如图 4.5 所示的窗体，而不再是一个空白窗体。

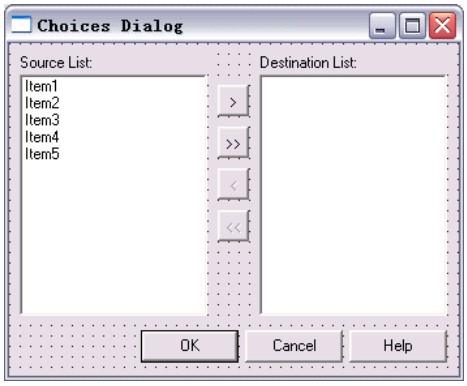

图 4.5　新创建的非空窗体

如果希望取消这样的模板功能，则可以重新调出图 4.4 所示的对话框，然后找到我们需要的对象，例如在这里就是 Forms 中的 Dual List Box，然后取消其中的 New Form 复选框就可以了。

说明：

Delphi 的每个工程都有一个，并且只有一个主窗体。所谓主窗体，就是在默认的情况下，程序运行时推出的第一个窗体。从某种意义上来说，主窗体就是应用程序，应用程序就是主窗体。主窗体的关闭意味着应用程序的终止。

利用同样的方法，可以设定选择 New Application（新建应用程序）命令时创建的不是一个空白的工程。

在这个对话框上，还有一些可以用来操作左边的类别和右边的对象的按钮，它们的用途可以从它们的按钮名称上猜测出来。

一个工程中存在着多个窗体，意味着必定存在着多个程序单元。虽然一个程序单元不一定对应着一个窗体，但是一个窗体肯定至少对应着一个程序单元。可以通过前面介绍的 Units 和 Forms 命令以及工具栏上的对应按钮在它们之间来回切换。也可以通过工程管理器 Project Manager，直接在各个窗体和单元之间导航。

4.2.2 从一个窗体调用另一个窗体

在第 3 章中，简单地介绍了面向对象编程的基本思想和 Delphi 中的面向对象的编程。关于 Delphi 中的面向对象的编程，我们在后面的内容中还会进行专门的介绍，这也是一个值得我们用一章的篇幅来介绍的 Delphi 中的重要主题。这些都是后话。从目前来说，我们已经掌握的是，在 Delphi 中，可以说什么都是对象，包括我们的窗体（Form）。所以，我们在处理窗体之间的关系时，必须从对象的思想出发，才能更好地理解那样做的原因。

既然一个窗体就是一个对象，那么当然可以在一个窗体中包含另外的一个窗体对象。

当我们在工程中加入一个新的窗体时，Delphi 自动把新的窗体对应的单元名称加到工程文件的 Uses 语句中，但是并不会自动加入到工程的其他单元中。如果要在其他单元中引用这个窗体，则必须把这个窗体对应单元文件包含在引用它的单元的 Uses 语句中。

例如，在一个包含两个窗体的工程中，如果要在 Form1 中引用 Form2 的话，则可以像下面一样编写代码：

```
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
unit2;
```

当加入了对 Form2 的引用之后，就可以像使用其他对象一样在程序中使用 Form2 对象了。例如可以在 Form1 的 Create 事件中加入下面的代码：

```
procedure TForm1.FormCreate(Sender: TObject);
begin
      form2.Show;
end;
```

但是，也许会存在下面所描述的一种情况：我们在 Form1 中引用了 Form2，而在编程中又需要在 Form2 中引用 Form1。那么这就会引起一个交叉引用的问题。如果出现了这种情况，当你试图运行应用程序的时候，Delphi 会在 Message 窗口中显示一个错误信息。

如何才能避免这种情况呢？我们可以把对工程中其他单元的引用不放在 Interface 部分的 Uses 语句中，而是在程序的 implementation 中加入一个 Uses 语句，然后把需要引用的单元放置在这个语句中。

另外，也可以不用自己去添加对工程中其他单元的引用，而是直接使用工程中包含的 Form 对象。也就是说，我们在没有向 Uses 语句中加入对对应的单元引用之前，就在相关的程序中使用了其他的 Form 对象。然后编译该工程，Delphi 会显示一个如图 4.6 所示的提示框，询问是否把相应的单元名称放置到对应的 Uses 语句中。

单击 Yes，Delphi 会自动在单元文件的 Implementation 中的 Uses 语句部分加入这些引用。然后重新编译该工程，便可以正常运行了。

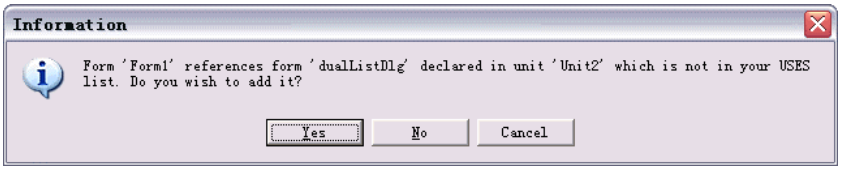

图 4.6　信息提示框

还有一个处理这种引用的方法：单击 File 菜单中的 Use Unit（引用单元），会显示如图 4.7 所示的对话框，然后从中选择需要引用的单元名称，并单击 OK。这样 Delphi 就会把这个单元加入到 Implementation 的 Uses 语句中。

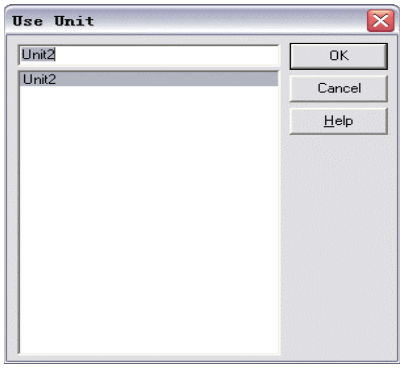

图 4.7　Use Unit 对话框

4.2.3　与其他工程共享窗体

在前面介绍添加一个新的窗体的时候，我们提到了使用 Delphi 的对象库，也就是 Delphi 的 Object Repository。这是和其他程序共享窗体的一个很常用的方法。我们已经掌握了如何使用 Delphi 的对象库来创建非空的窗体或者工程。那么现在我们要介绍的问题是怎样才能把自己设计的窗体或者其他的对象添加到对象库中，以便我们自己和其他程序员在以后可以共享我们设计的这个窗体。

例如对于上面设计的用户登录窗口，可以在窗体上右击鼠标，并从弹出的快捷菜单中选择 Add to Repository（添加到库中）命令，会出现如图 4.8 所示的对话框。

在这个对话框中，左边的 Forms 列表框中列出了当前工程中的所有窗体，从这些窗体中选择一个需要的窗体，比如在这里就选择我们刚才设计的 UserLogOn 窗体。然后按照下面的步骤进行操作：

(1) 在 Title 框中输入这个窗体出现在对象库中的名称，例如可以输入“用户登录窗口”。

(2) 把插入点移动到 Description 框中，然后输入下面的内容：“这是一个用来登录用户的名称和密码的窗口”。

(3) 然后单击 Pages 框右边的箭头，并从下拉列表中选择一个类，在这里当然选择 Forms。

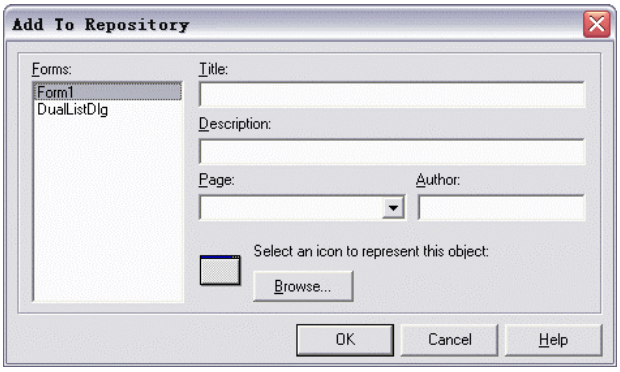

图 4.8 Add to Repository 对话框

(4) 在 Author 部分输入你自己的名字，例如这里输入“ JM ”。

(5) 单击对话框上的 Browse 按钮，来为我们的窗体指定一个图标。在完成了所有的操作之后，单击 OK，便把我们设计的窗体作为一个对象保存到了 Delphi 的对象库中。

(6) 单击 Tools 菜单中的 Repository 命令，调出 Delphi 的 Object Repository 对话框，然后选择它的“ Forms ”，此时在右边的列表框中已经列出了我们的窗体的名称，如图 4.9 所示。

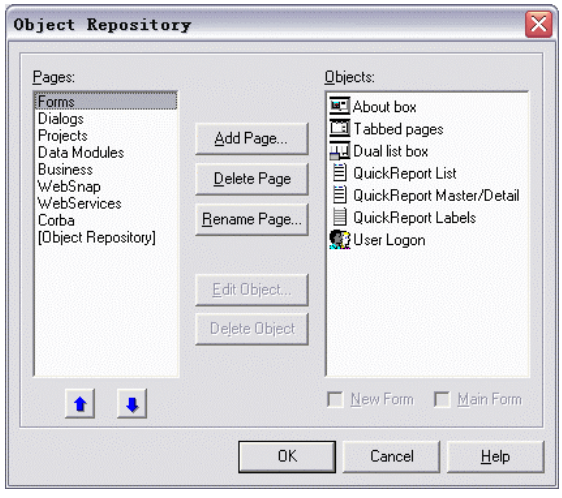

图 4.9 Object Repository 对话框中的新窗体

(7) 单击 File 菜单中的 New 命令，在出现了 New Items 对话框之后，单击 Forms 选项卡，此时的对话框如图 4.10 所示。

从上面的窗口中，我们可以看到，当我们选择新建一个窗体的时候，可以选择三种方式。

- 第一种方式是 Copy，顾名思义，就是把我们的对象库中的模板代码直接复制过来，创建一个形式上相同，但是已经和原来的窗体没有任何关联的窗体。
- 第二种方式是 Inherit，这种方式是把新建的窗体作为原来模板窗体的一个子类，使新窗体继承了原来窗体的所有数据域和方法，当然我们可以在这个新的窗体中添加

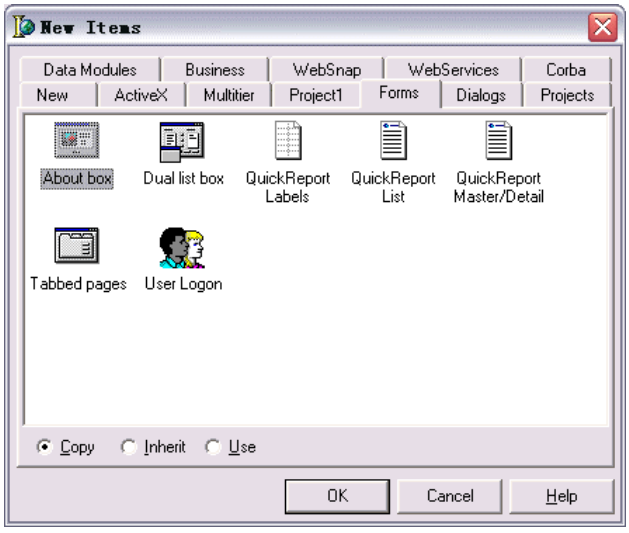

图 4.10　New Items 对话框中的新窗体

各种控件、方法和数据域。

❖　第三种方式是 Uses 方式，这种方式将直接使用我们设计的窗体和对应的单元代码。

对于后面的两种方式来说，存在着一个需要注意的问题是，在多个工程共享一个窗体时，如果在一个工程中对这个窗体进行了修改，就会影响到其他的共享这个窗体的应用程序，甚至会使共享这个窗体的应用程序不能运行。

4.2.4　使用 Form 模板和向导

Delphi 提供了一些预先设计好的模板，包括 Form 模板和其他对象的模板。在这里我们主要介绍 Form 模板。

使用模板的好处是在开发应用程序的时候有了一个参考和起点，使得我们可以不用再从最底层的编程做起，从而大大加快了开发应用程序的速度。这种模板的出现正是 Delphi 可视化编程的体现。

实际上，我们已经介绍了如何使用 Delphi 的 Form 模板。在上面介绍和其他工程共享 Form 的时候，我们以自己创建的窗体模板为例，介绍了使用 Form 模板的三种方式。在图 4.10 所示的对话框中，也列出了其他一些模板，它们的使用方法是类似的。

在 Delphi 中，也提供了一些 Form 向导，主要是 New Items 对话框中的 Business 选项卡中的 Database Form Wizard，如图 4.11 所示。

使用这个向导，可以快速迅捷地创建一个数据库访问的窗口。

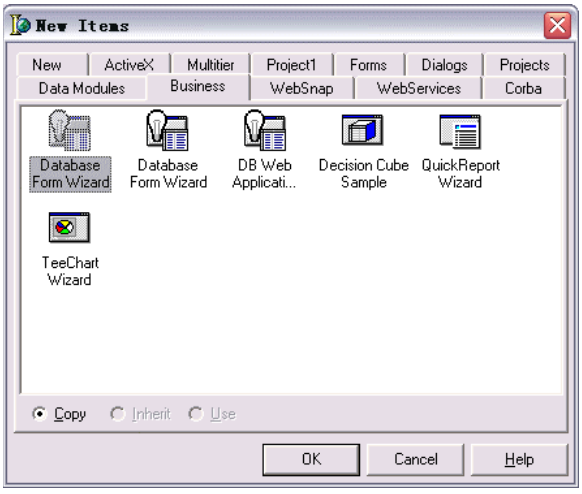

图 4.11 New Items 对话框

4.3 对象的属性和事件

使用过 Delphi 的人都会有这样的感觉，Delphi 实在是太奇妙了，它不仅提供了功能上十分强大的对象，而且允许我们在设计期就可以控制对象的外观和行为，而不用编写任何代码。在许多情况下，我们只要在设计期把各个对象的属性设置好，一个应用程序基本上就完成了。当然，有时也需要在运行期修改程序中的对象属性。尽管这样做需要一些编程工作，但是这仍然是十分简单的，修改对象属性的代码通常不过是几条赋值语句而已。可视化的编程思想在这里表现得淋漓尽致。

另外，利用 Delphi 编写应用程序的另一个重要的工作是处理响应对象的事件以及处理对象的事件。这些事件的来源有可能是来自系统的内部，也可能是来自用户的操作。事件使得用户能够与操作系统进行交互，使得用户能够驱动应用程序的运行。Delphi 为响应事件做了最大程度的简化，使得我们可以通过 Delphi 的属性编辑器方便地产生、定位和修改事件句柄，代码编辑器对用户的操作会做出同步的反应。我们也可以通过属性编辑器把一个已经存在的事件句柄同一个对象的事件句柄相联系。

总之，事件和属性是 Delphi 面向对象编程思想的最大体现。从而使我们在用 Delphi 编程的时候涉及最多的工作中可以充分体会 Delphi 的面向对象编程思想。

4.3.1 在设计期间修改对象的属性

当我们向工程的窗体上添加一个对象的时候，Delphi 会自动用这个对象的默认属性值显

示这个对象。我们可以通过属性编辑器来修改这些属性的值，对于可视对象来说，这些修改会立即在屏幕上反应出来，而不用编译和运行应用程序。这正是可视化编程的基本特征。

要修改一个对象的属性，首先选中这个对象，然后单击 Delphi 的属性编辑器的 Properties 选项卡，从中找到需要的属性，然后在其右边的框中直接输入相应的属性值，或者利用属性编辑器提供的其他方式选择需要的属性值。

属性编辑器是一个动态的工具，它总是显示当前选择的对象的属性，当我们在任何 Form 上选择新的对象时，它立刻就会显示这个新选定对象的属性。

说明：

在这里，我们一再使用对象这个名词，而不是控件或者窗体，这是因为在 Delphi 中，一般不把 Form 理解为一个控件，虽然它是从 Tcontrol 继承而来的一个对象。而实际上，我们是可以把一个 Form 作为一个特殊的控件来使用的。所以，我们介绍的绝大多数处理控件的方法同样适用于 Form 对象。

如果同时选择了多个对象，那么属性编辑器显示的就是这些对象的公共属性。如果这些公共属性的值是相同的，这个值就显示在属性编辑器的对应属性框中；如果这些对象中对某个公共属性的赋值是不相同的，则属性编辑器中的对应属性框中就不显示任何内容。当然，可以通过对这个空白的属性框赋值，从而强制这些对象具有同样的属性，但是在这样做的时候要十分小心，比如如果我们把一些对象的位置属性 Left 和 Top 中的一个赋予了同一个值，那么可能会取得在某个方向上对齐的效果；但是如果把它们都赋予了同一个值，可能会使得所有的这些对象都重叠到了一起。这可能不是我们希望的结果。

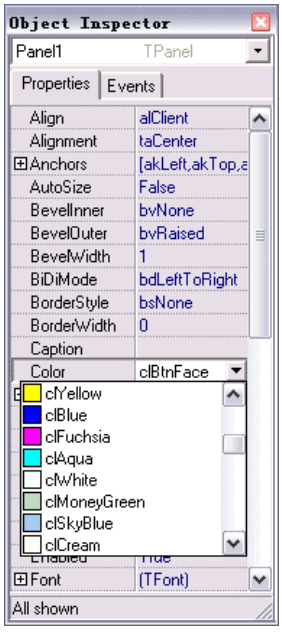

图 4.12　下拉列表型的属性设置

属性编辑器针对不同的属性设计了不同的编辑器，总的来说，可以归纳为以下几类。

- 直接输入型：这一类的属性值可以在属性编辑器中直接输入，绝大多数的属性都是这样的。
- 下拉列表型：Delphi 预先定义了一个下拉列表，可以从中选择我们需要的类型，例如在设置对象的颜色的时候，就可以使用这样的一个下拉列表，如图 4.12 所示。
- 对话框型：当单击这个属性旁边的一个按钮的时候，会弹出一个对话框，让我们对这个属性进行详细设置，例如大多数对象的字体属性就是这样进行设置的，如图 4.13 所示。

也可以定义自己的属性编辑器，在后面介绍到控件编程的时候，我们会详细介绍这个问题。

图 4.13　对话框型的属性设置

4.3.2　在运行期间修改对象的属性

我们也可以在运行期间修改对象的一些属性，使得对象根据用户的输入表现出相应的响应，或者表现出一种动态的效果。

例如，在设计一个类似于 Word 的 MDI 应用程序时，需要根据打开的文档名字修改主窗体的标题。在这里可以加入下面的代码：

```
MainForm.Caption:=Document[ActiveDocIndex].FileName;
```

另外，我们在前面第 3 章中介绍的第一个例子中，使用了一个 Tbutton 控件，并把它的 Caption 属性修改为了“ Say Hello! ”。我们也可以在程序中修改它的值，使它能够按照我们的需要显示自己的标题。例如，可以使用下面的代码：

```
BtnSayHello.Caption:='Hello!';
```

观察这两个在运行期修改对象的属性的例子，可以看出，在运行期修改对象的属性的时候，一般采用下面的格式：

```
Object.Property := ValueOfProperty;
```

说明：

需要说明的是，并不是所有的属性都是可以在运行期进行赋值的，个别对象的个别属性是只读属性。顾名思义，只读属性是不能进行赋值操作的，不管是在运行期还是在设计期。但是，从对属性的操作范围上来说，使用在运行期修改对象属性的方法可以比在设计期修改属性的方法具有更多的可以修改的属性。也就是说，在 Delphi 中，许多对象的许多属性由于各种原因，并没有显示在 Delphi 的属性编辑器中。

4.3.3　对象的事件

对象的属性是用来控制对象的外观和一些常规行为的。在设置属性的时候通常不需要进行编写程序的工作。如果希望在运行期改变对象的属性，可能需要进行一些赋值语句的编写，但是这仍然是十分简单的。当我们开始处理对象事件的时候，就要涉及到编写大量的代码了。

虽然在一些特殊的情况下，我们也可能需要编写独立的程序模块，但是从普通的应用角度来说，绝大部分编程工作都是在对象的事件中完成的。

在 Delphi 中编程的时候，经常使用事件句柄这样的术语。所谓对象的事件句柄，实际上就是一个 Delphi 例程。在下面的介绍中，你完全可以把事件句柄理解为在其他高级编程语言中经常使用的例程（又称为子函数）。

可以使用属性编辑器为窗体或者其他对象生成新的事件句柄。步骤如下：

(1) 首先选择窗体 Form 或者窗体上的控件，然后单击属性编辑器上的 Events 选项卡，此时的属性编辑器如图 4.14 所示。

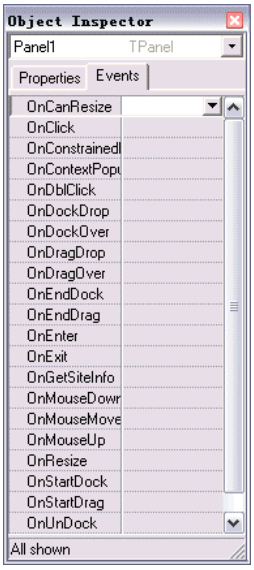

图 4.14　属性编辑器上的 Events 选项卡

(2) 在这个选项卡上列出了所选定对象能够响应的所有事件。这些事件的名称代表了该事件发生的条件，当然这只是一个简要的代表方式，要理解一个对象所能响应的事件发生的先后顺序以及具体条件，还要专门查阅该对象的说明文档。我们在后面的章节中会对该问题进行简要的介绍。

(3) 选定要处理的事件，然后双击该事件或者按下 Ctrl+Enter 组合键，便可以生成事件句柄。Delphi 自动在代码编辑器中生成该事件句柄的框架，包括默认的方法名称和参数以及 Begin 和 End 保留字，并自动把光标移动到它们之间，用户需要做的就是在这里输入所需要的代码了，如图 4.15 所示。

对于许多对象来说，它们都具有自己默认的事件句柄，这个默认的事件句柄往往是该对象最经常处理的事件。如果要生成对象的默认事件句柄，可以不必按照上面的步骤进行操作，而是双击对应的对象，Delphi 也会自动生成这个默认的事件句柄。但是要注意的是，并不是所有的对象都具有自己的默认的事件句柄。有些对象在双击的时候不会生成事件句柄，而是显示该对象的默认属性编辑器。

在编写 Delphi 程序的时候，经常会遇到需要多个对象进行同样的操作。例如，我们可能希望用户在窗体上单击和在旁边的按钮上单击时程序进行同样的操作。那么此时可以不必重复编写 Form 和 Button 的 Click 事件，而是在编写了 Button 的 Click 事件之后，在 Form 上重用事件句柄就可以了。

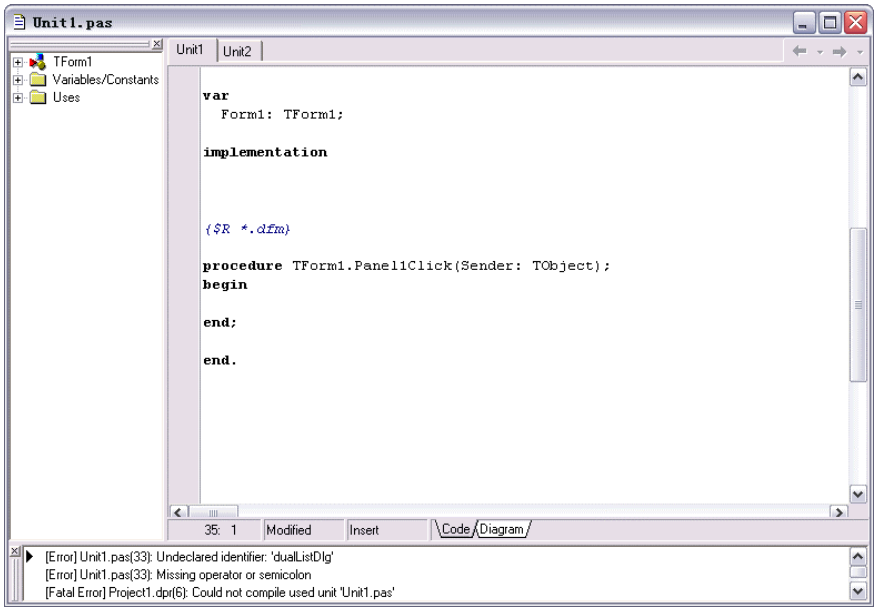

图 4.15 Delphi 自动生成的事件句柄

在编写了一个事件句柄之后，可以选中需要重用事件句柄的对象，然后单击属性编辑器上的 Events 选项卡，然后单击某个事件句柄旁边的箭头，从下拉列表中选择已经编写好的一个事件句柄就可以了，如图 4.16 所示。

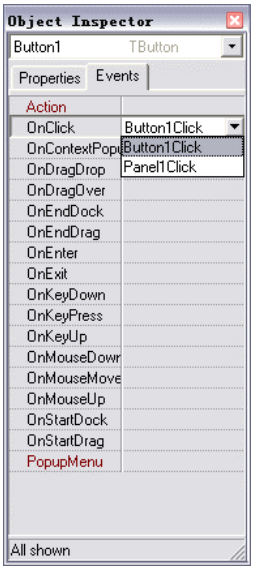

图 4.16 重用事件句柄

当按照上面的步骤进行操作之后，不要错误地认为在代码中具有多个事件句柄，而是多

个事件调用同一个事件句柄而已。

说明：

在让多个事件共享一个事件句柄的时候，必须要注意，如果在编程的过程中由于某种原因修改了这个事件句柄中的代码，那么所有重用该事件句柄的对象在响应相应的事件的时候都会受到影响。

如果我们在编程的过程中需要重新定位一个已经存在的事件句柄，只要在属性编辑器上双击需要的事件，就可以把光标定位到代码编辑器中的对应的事件句柄中。

要删除一个事件句柄，可以删除在单元文件中这个事件句柄的声明和代码模块。当重新编译 Delphi 的工程的时候，Delphi 会提示你删除相应的对象中关于这个事件句柄的调用。

4.4　Delphi 的工程管理

在 Delphi 中，正在开发中的一个应用程序或者动态链接库称为一个工程，并且提供了完善的工程管理功能。在这一节中，我们将对 Delphi 中的工程的创建、管理、编译等方面的内容进行介绍。

4.4.1　工程概述

从我们在第 3 章中建立的简单的应用程序就可以看出，在利用 Delphi 创建应用程序的时候，Delphi 建立的文件有许多个。也就是说，在 Delphi 中，一个应用程序所包含的文件不会只有一个，通常是由多个文件共同组成一个工程，然后经过编译，才能创建出一个应用程序。

所谓工程，从其内容上来看，可以理解为构成应用程序或动态链接库的一些文件的集合。其中有些文件是在设计的时候建立的，有些文件则是在编译期间由编译器生成的。这些文件可以通过不同的文件后缀名来区分，具体来说，一个 Delphi 工程通常包含以下的文件。

- 工程文件：该文件的扩展名为 DPR，每个工程文件只有一个工程文件。从文法的角度来看，工程文件与一般的单元文件并没有太大的区别。工程文件一般是由 Delphi 自动维护的。在有些特殊的情况下，需要对这个文件进行改动，从而实现我们的特殊目标。
- Form 文件：扩展名为 DFM，是由 Form 以及 Form 上的控件组成的二进制文件。一个工程可以具有多个 Form 文件，每个 DFM 文件都对应一个扩展名为 PAS 的单元文件。
- 单元文件：扩展名为 PAS，这是 Object Pascal 语言的源代码文件。一般来说，一个 Form 文件对应一个单元文件，但是一个单元文件不一定对应于一个 Form 文件。

- DCU 文件：这是编译后的单元文件，是由编译器生成的二进制文件。

4.4.2 关于工程的基本操作

关于工程的基本操作包括工程的创建、保存、打开、运行、关闭等。

要创建一个新的工程，可以单击 File 菜单上的 New Application 或者 New 命令。如使用前者的时候，Delphi 会根据当前的默认设置创建一个新的工程；如果使用后者，则会弹出 New Items 对话框，在这个对话框中提供了许多的模板，可以利用这些模板建立各式各样的 Delphi 工程。

建立了工程之后，在编辑的过程中，通常需要多次保存该工程。使用 File 菜单上的 Save Project As 命令可以把当前工程保存为我们希望的名称。在保存了工程之后，就可以使用工具栏上的 Save All 命令来保存整个工程。

说明：

在保存工程的时候，同时会要求保存工程中所包含的文件。在命名这些文件的时候，注意不要使用同样的名称，这个名称指的是不包含扩展名的文件名称，否则 Delphi 会显示一条错误信息。

利用 File 菜单中的 Open 或者 Open Project 命令,可以打开已经存在的 Delphi 工程。

要运行一个工程，可以使用 Run 菜单中的 Run 命令，或者单击工具栏上的 Run 按钮。

要关闭当前的工程，可以使用 File 菜单中的 Close All 命令。

4.4.3 Project 菜单

在 Delphi 中，管理工程的直接工具就是 Project 菜单。使用这个菜单中的各个命令可以实现关于 Delphi 的工程的几乎所有的管理工作。该菜单如图 4.17 所示。

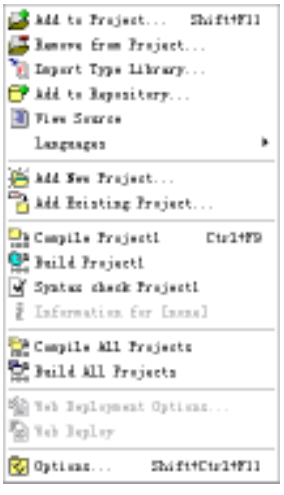

图 4.17 Project 菜单

从图中可以看出，Delphi 把它的关于工程的管理命令分成了 5 组，每一组都有自己的处理功能。下面我们就简要地介绍各个命令的用法。

在第一组命令中，主要的操作对象是工程中的各个组成文件，它们包括以下命令：

- Add to Project（向工程中添加文件）命令执行时会显示如图 4.18 所示的 Add to Project 对话框。通过这个对话框，你可以找到自己需要加入到当前工程中的文件，然后单击“打开”按钮，便可以把该文件加入到自己的工程中。我们在长期编程的过程中，肯定编写了一些完整的代码模块，例如编写了一个用于硬件输

入输出的模块，在编写其他应用程序时通常会重复使用这个模块。在这种情况下，使用该命令便可以达到重复利用代码模块的目的。

- Remove From Project（从当前工程中删除选定的文件）命令的作用和上一个命令正好相反。单击该命令会显示如图 4.19 所示的 Remove From Project 对话框。该对话框中列出了当前工程中包含的单元文件和对应的窗体。选中需要删除的文件，然后单击 OK，便可以从当前工程中删除选中的文件。

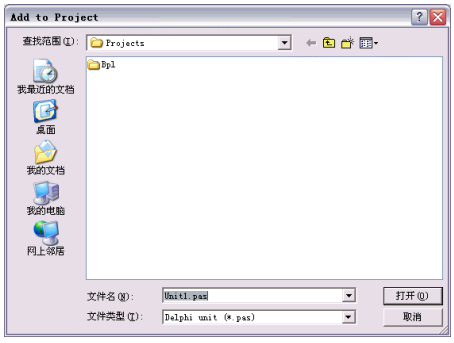

图 4.18　Add to Project 对话框

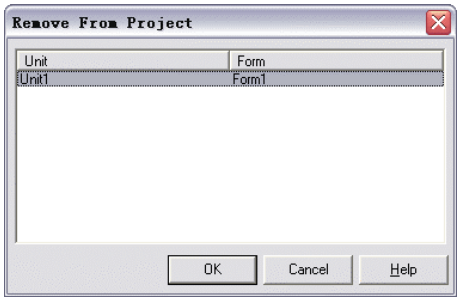

图 4.19　Remove From Project 对话框

- Add to Repository（添加到库中）命令的使用和我们前面介绍的窗体编辑器上的弹出菜单中的同名命令十分类似。
- View Code（查看代码）命令可以显示当前工程文件的代码。
- Language（语言）命令中的子命令可以用来为工程定义语言种类。
- Add New Project（添加新工程）命令向 Delphi 中加入一个新的工程。
- Add Existing Project（添加已存在工程）命令向 Delphi 中加入一个已经存在的工程。
- Compile XXX（编译 XXX）命令会编译所有自上次编译以来修改过的文件，并且为每一个单元生成一个扩展名为 DCU 的文件。如果某个单元的 Interface 部分发生了改变，所有引用该单元的单元文件也会被编译；如果编译器找不到某个单元文件，该单元就不进行编译。当所有的文件编译完成之后，Delphi 就会编译工程文件并生成可执行文件或动态链接库。
- Build XXX（重建 XXX）命令会重建当前的工程，也就是说会重新编译当前工程的所有文件，并生成新的可执行文件或动态链接库。
- Syntax Check（语法检查）命令用于检查程序的语法错误。这个命令和 Compile 命令十分类似，也是对程序进行编译，但要比 Compile 命令快得多，因为它只检查语法，不产生目标文件，也不会链接目标文件以生成可执行文件或动态链接库。这个命令主要用于程序的开发初期。如果检查出语法错误，Delphi 自动把代码编辑器推到前端，并把有错误的那一行突出显示在编辑器的第一行，同时在下面的状态窗口中显示错误信息，在错误信息上按下 F1 键可以看到该错误信息的详细解释。

- Compile All Projects（编译所有工程）和 Build All Projects（重建所有工程）命令用于处理同时开发多个工程的情况。这些命令的作用和上面介绍的处理单个工程的作用相同。
- Web Deploy Options（Web 配置选项）和 Web Deploy（Web 配置）命令。在开发用于 Web 客户端的 ActiveX 控件时需要使用。

上面，我们已经介绍了 Project 菜单中的大部分命令，还有两个命令，由于它们的重要性，所以在下面进行单独介绍。

Project 菜单中的 Import Type Library（导入类型库）是在管理工程的过程中的一个十分重要的命令。单击该命令会显示如图 4.20 所示的 Import Type Library 对话框。

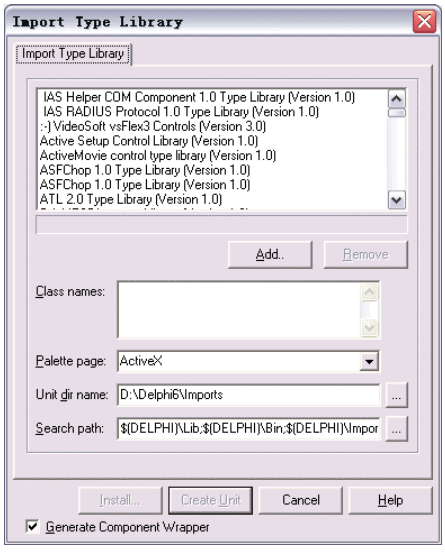

图 4.20 Import Type Library 对话框

这个对话框的作用是十分强大的。在任何一个计算机中，肯定会安装了一些除了 Delphi 以外的其他软件，而这些软件的作者在创建这些软件的时候也会创建一个通用的模块或者控件。在安装这些软件的时候，安装程序会在当前计算机系统中注册这些模块或者控件。在 Delphi 中，利用 Import Type Library 对话框可以利用这些模块或者控件，从而不仅大大拓展了 Delphi 编程中的可用资源，而且也为我们编写的代码重用提供了一种适用范围十分广泛的方法。在这个对话框中，上面的列表中显示了在当前的计算机系统中注册的控件或者模块。

单击对话框上的 Add 按钮，会显示如图 4.21 所示的 Open Type Library（打开类型库）对话框。选中这个对话框出现的类型库，例如 OCX、DLL 等等，然后选择“打开”按钮，便可以把该类型库添加到 Delphi 的类型库中。

单击 Import Type Library 对话框中的 Remove 按钮可以删除当前选中的类型库。

当选中某个类型库的时候，这个类型库中所包含的可以重建的类便会显示在 Class Names

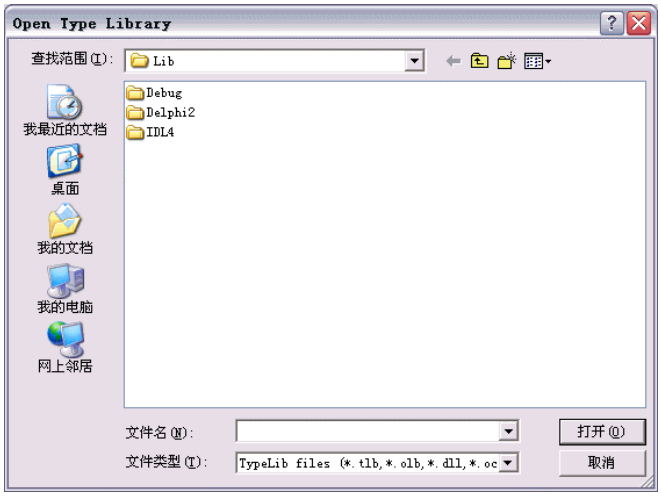

图 4.21　Open Type Library 对话框

框中。下面的三个框分别显示了重建的类所在的控件面板的名称以及包含要重建的类的单元文件和参考库的搜索路径。

在对话框底部的 Install 按钮可以把当前选中的类型库中的类所定义的对象安装到指定的控件面板中。单击 Create Unit 可以创建当前选中的类型库中所定义的对象的单元文件，并把该文件添加到当前的工程中。这两种方法都可以使该类型库中的对象成为当前工程中的可用部分，所不同的是前者可以在任何工程中使用这里安装的对象，而后者则只能在当前工程中使用该对象。

在 Project 菜单中的 Options 命令是关于工程管理的最为复杂的操作，我们将把它作为单独的一节进行介绍。

4.5　工程的设置选项

在创建编辑 Delphi 的工程的时候，可以进行许多选项的设置，从而创建不同的工程或者为工程指定特殊的引用资源、编译方法，等等。单击 Project 命令中的 Options 命令会显示如图 4.22 所示的 Project Options（工程选项）对话框。

4.5.1　指定主窗体

所谓主窗体是指程序运行时最先推出的窗体，每个应用程序都必须有一个主窗体，而且只能有一个主窗体，关闭主窗体意味着关闭应用程序。

Delphi 能自动给每个加入到应用程序中的 Form 一个包含需要的名字，例如 Form1、Form2

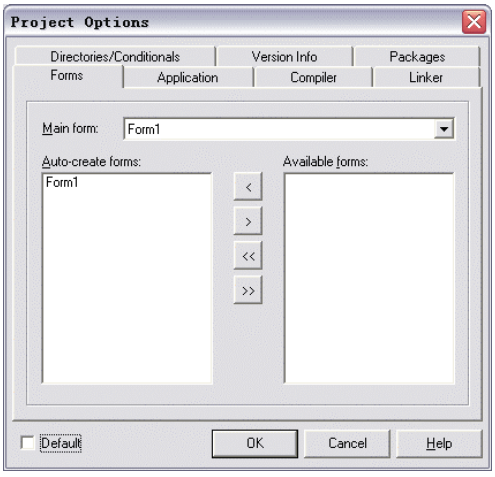

图 4.22 Project Options 对话框

等等。其中主窗体就是工程文件中 CreateForm 语句列表的第一项，如图 4.23 所示，其他 Form（如果存在的话）将按照加入的顺序依次排列。这样，第一个生成的 Form 总是应用程序的主窗体。不过，可以把工程中的任何一个窗体指定为工程的主窗体。

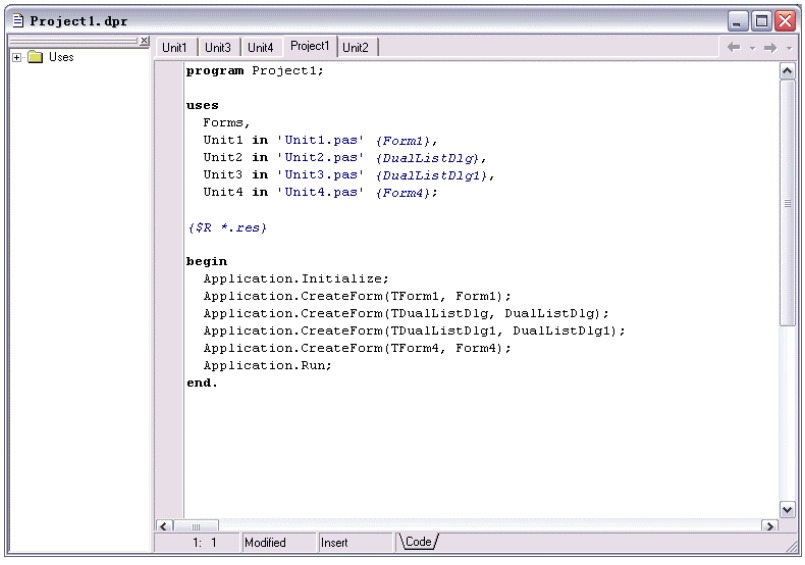

图 4.23 工程文件中主窗体的创建

说明：

在使用了多个窗体的应用程序中要注意，在退出应用程序的时候一定要关闭主窗体。举例来说，如果在主窗体中调用了另外一个窗体，并隐藏了主窗体，那么在关闭后来显示窗体的时候，必须显示主窗体，否则用户可能以为已经关闭了应用程序，而实际上应用程序仍然

在运行，仍然占用着计算机的资源，如果多次这样运行一个应用程序，可能会导致计算机系统的崩溃。

那么如何指定工程的主窗体呢？可以使用图 4.22 所示的 Project Options 对话框来指定工程的主窗体。

在对话框的 Forms 选项卡中，第一个框中显示的就是当前工程的默认主窗体。单击该下拉列表框右边的箭头，可以显示该工程中所有的可用窗体，从中选择需要的窗体，单击 OK 就可以把该窗体设置为当前工程的主窗体。设置主窗体的工程就是如此简单。

在这个对话框的底部，还有另外的两个列表。利用它们，可以指定哪些窗体可以在应用程序初始化的时候就创建，而哪些窗体需要程序员在程序运行的过程中创建。对于需要程序员在运行过程中创建的窗体，工程文件中将取消对它们的创建。

4.5.2　设置应用程序的选项

利用 Project Options 对话框，我们还可以设置由工程生成的可执行文件的一些选项，例如程序的标题、帮助文件、图标，等等。

单击该对话框上的 Application 选项卡，此时的对话框如图 4.24 所示。

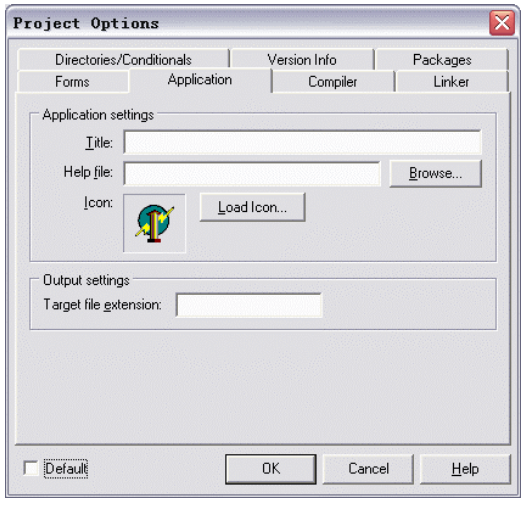

图 4.24　Project Options 对话框的 Application 选项卡

在这个选项卡中，主要有四个需要设置的选项。第一个是应用程序的标题，这个标题是当应用程序最小化的时候在 Windows 9.x 的任务栏上显示的应用程序的标题。这个标题的长度不能超过 255 个字符。第二个选项可以指定应用程序的帮助文件，如果不能确定帮助文件的准确名称，可以单击旁边的 Browse 按钮来定位该帮助文件。这个文件名称和路径将传递给 WinHelp 函数。运行应用程序并按下 F1 键的时候，会打开这个帮助文件。

第三个选项是指定应用程序的图标。Delphi 为应用程序提供了默认的图标，已经显示在了对话框中。单击旁边的 LoadIcon 按钮，可以显示一个如图 4.25 所示的 Applicaton Icon 对话框。在这个对话框中定位需要的 ICO 图标文件，然后单击“打开”按钮，便可以把选定的图标指定给应用程序。

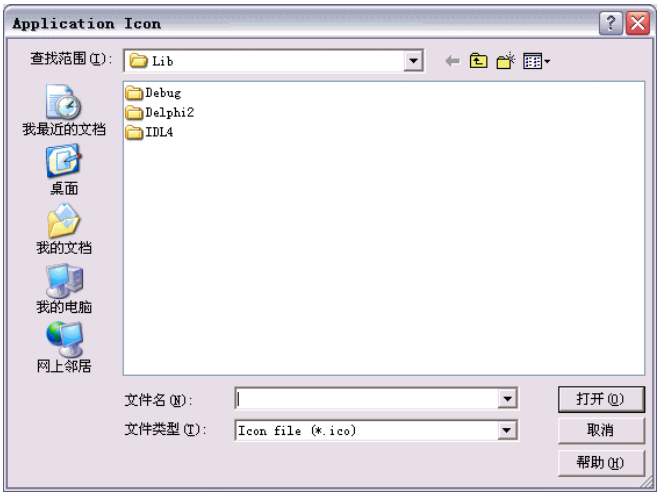

图 4.25 Application Icon 对话框

Application 选项卡中的第四个选项是应用程序的输出设置，在这里可以指定输出文件的后缀名。在当前工程是用来创建 ActiveX 控件或者 DLL 的时候，可以指定这个后缀名为 OCX 或者 DLL。

这个选项卡的底部还有一个复选框，选中该复选框表示当前选项卡中的设置将作为 Delphi 工程的默认设置，当以后再创建新的工程时，将使用当前的设置。

在这个对话框中还有另外几个选项卡，它们的设置方法和设置 Delphi 的集成环境中的对应选项类似，这里就不再详细介绍了。

4.6 本 章 小 结

“工欲善其事，必先利其器”，我们在开始编程之前，首先应该熟练的掌握 Delphi 中关于窗体、控件、工程和应用程序的相关操作是十分重要的。在后面的内容中我们的绝大多数工作都和它们有关，那时就不再进行详细的介绍了，而只是告诉你应该操作什么，比如我们可能会简单地说“打开某个工程”，而怎么打开这个工程，我们在本章中已经介绍过了。

第 5 章　Delphi 的基本控件

从本章开始将介绍如何利用 Delphi 6.0 进行编程。在编程的时候，首先要设计程序的界面，需要构思在界面中放置哪些控件，如何安排它们的布局，利用它们的哪些属性来实现我们的目的。

在本章中，将对 Delphi 中的一些基本的控件进行介绍。所谓的基本控件，我们指的是 Delphi 控件选项板的 Standard 选项卡和 Additional 选项卡上的控件。

在学习了上面的关于 Delphi 的基础知识之后，我们终于可以开始介绍真正的程序设计部分了。Delphi 是开发 Windows 应用程序的最快捷的工具之一，使用 Delphi 可以轻松地制作出程序的图形化界面，包括命令按钮、文本处理控件、列表框图形处理控件，等等。通过使用这些技术可以轻松愉快地创建出具有专业水平的应用程序界面，使得程序的设计成为一件富有吸引力的事情。

在本章将主要介绍：

- 用于处理文本的控件
- 各类命令按钮
- 选项按钮和复选框
- 各类列表框
- 容器控件

5.1　窗　　体

在 Delphi 中，几乎所有的控件或者对象都要放置在一个容器上，就像我们的家具也必须有一个放置的地点一样，而在这里我们要介绍的窗体可以说是能容放所有控件和对象的一个大容器，而且是其他容器的父容器。我们必须把控件、对象放置到窗体上，才能构建我们的应用程序。从某种意义上来说，在 Delphi 中，窗体就是应用程序，应用程序就是窗体，特别是对于程序的主窗体来说。在 Windows 程序中，几乎没有什么程序不使用窗体，因为不提供窗体的应用程序，除了按照设计者预先设计好的流程进行之外，它不能完成任何事情，特别是和用户的交互。

同时，窗体，也就是 Tform 类，还不是一个简单的容器，它本身就是一个 Delphi 的对象，而且和我们要介绍的 Delphi 控件的性质十分相似。它也具有自己的属性、方法，也可以像普通的可视控件一样，可以调整它的尺寸，等等。

如果在 Delphi 中观察一下所有控件的属性编辑器，就会发现，TForm 以及其他的对象，都具有一些共同的属性。实际上它们也具有一些共同的方法。根据在第 3 章中介绍的 Delphi 中的对象的继承关系，我们知道，Delphi 中的所有对象都来自同一个祖先，并且，对于可视控件，它们有着非常近的血缘关系，所以有些属性和方法是所有这些控件或者对象都具有的。在这里，我们通过介绍 Tform，同时也对这些共同的属性进行一些介绍，这样在后面介绍其他控件的时候，只要介绍它们的特殊属性或者关键属性就可以了。

在设计窗体的时候，我们首先关注的是如何控制它的外观。可以通过修改窗体的属性来改变它的外观。下面就对这些属性分别介绍。

5.1.1 改变窗体的标题

首先可以改变窗体的一些基本属性，比如窗体的标题、颜色等。如果要改变窗体的属性，那么可以使用 TForm 的 Caption 属性。在属性编辑器中，选择 Caption 属性，然后输入我们需要的文字。比如这里可以输入“示例窗口一”。

说明：

在 Delphi 中，有一些用来作为标题或者文本的属性，比如 TForm、TLabel、TPanel、TButton 等控件的 Caption 属性，又如 TEdit 控件的 Text 属性。Caption 和 Text 属性都是 String 类型的。我们可以把自己希望的任何字符串赋给它们。需要注意的是，它们都是 String 类型的。如果要把其他类型的值赋给它们，不要忘记进行类型转换。

5.1.2 改变窗体的颜色

如果我们希望把窗体的颜色换成我们希望的颜色，而不是通用的灰色，那么我们可以选择 Color 属性，然后选择该属性旁边的箭头按钮，此时会显示如图 5.1 所示的下拉列表。

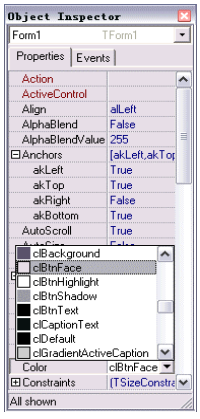

图 5.1 Delphi 中 Color 属性的下拉列表

图 5.2 Delphi 中的标准颜色选择框

在 Delphi 中，所有的和颜色相关的属性赋值都是和这里 TForm 的 Color 属性赋值相同的。可以通过这个下拉列表选择自己喜欢的颜色，这里显示了一些颜色名称，这是 Delphi 中预先定义的颜色常量。当然，如果在这些颜色中找不到自己喜欢的颜色，那么可以在 Color 属性右边的框中双击鼠标，此时会显示一个如图 5.2 所示的标准的颜色选择对话框。

如果你希望自己定义颜色，那么可以单击对话框中的“规定自定义颜色”按钮，此时对话框会显示它的另外部分，此时的对话框如图 5.3 所示。

图 5.3　可以定义任何颜色的颜色选择对话框

这个对话框读者肯定非常熟悉，因为我们在很多的应用程序中都会看到这样的对话框。在这里我们选择了一个浅黄的颜色。此时你会发现选择的颜色出现在对话框左边的“自定义颜色”部分，选中这个颜色，然后选择“确定”按钮，此时会发现窗体的颜色改成了浅黄色。如果观察一下 Delphi 属性编辑器中 TForm 的 Color 属性现在的值，会发现它是一个 16 进制的数值，例如“$00CCF4F7”。通常来说，这个数值只有后六位有用，每两位一组，可以分为三组，它们分别代表红、绿、蓝三种基本的颜色。我们知道，计算机中的任何颜色可以用这三个基本色来合成。通过后面的一个例子就会发现可以如何来利用这三种颜色调和成任意一种颜色。

说明：

在 Delphi 中，颜色是一种数据类型，称为 TColor。实际上它是一个整型的数值。也就是说，可以把任何整型的数值赋给它，它的范围是从–$7FFFFFFF-1 到$7FFFFFFF。

5.1.3　改变窗体的标题栏

在 Delphi 中改变窗体的尺寸是非常容易的。如果不在乎窗体的具体尺寸，只是希望能把它调整到眼睛看起来很舒服的大小，那么可以在 Delphi 的设计环境中，直接用鼠标拖动窗体的边框，直到对它满意为止。如果希望能够精确地控制窗体的位置、宽度和高度，那么可以在属性编辑器中直接修改窗体的 Left、Top、Width 和 Height 四个属性。

下面我们来观察一下修改后的窗体，如图 5.4 所示。

图 5.4　修改了颜色、标题、位置属性的窗体

也许有的读者会问，这个窗体是空的，有什么可观察的。那是因为我们对窗体的注重点不同。比如这个窗体首先具有一个标题栏，在这个标题栏上有标题、窗体的图标和三个系统按钮。在另外的三边都具有边框，而且这个窗体是规则的常规窗口。它没有最大化，也许没有出现在屏幕的中央位置，而是停放在我们拖动它所在的位置上。

那么上面我们提到的这些属性可不可以改变呢？在 Delphi 中可以很容易地改变它们。

首先来看一下关于窗体标题栏的操作。标题栏上的标题当然是可以修改的，甚至可以不包含任何文字。这个问题我们在上面已经介绍过了。可以看到窗口的左上角有一个图标，这是窗口的标志。目前是 Delphi 默认的图标。在设计应用程序的时候，我们希望程序具有自己的图标。关于应用程序的图标的问题我们在前面的内容中也已经进行了介绍，可以通过 Project 菜单中的 Option 命令来完成。但是如果程序有很多窗口，那么你可能希望每个窗口都具有自己的图标，此时我们可以按照下面的步骤来修改窗体的图标。

(1) 在属性编辑器中找到 TForm 的 Icon 属性，单击它右边的按钮，此时会显示如图 5.5 所示的对话框。

说明：

图 5.5 所示的是 Delphi 中的标准的载入图片对话框。我们在后面介绍的很多控件都具有一个图像属性，典型的如 TImage 控件。在修改控件的这个属性的时候，我们通常会使用这个对话框。

(2) 单击对话框中的 Load 按钮，此时会显示如图 5.6 所示的对话框。

(3) 在这个对话框中浏览，直到找到需要的图标文件，然后选择该文件，并单击对话框中的“打开”按钮，此时会回到图 5.5 所示的对话框，并把所选择的图片载入了对话框中。

如果对图片满意，就可以单击对话框中的 OK 按钮。如果不满意，可以选择 Clear 按钮，清除该图片，然后重新载入新的图片。

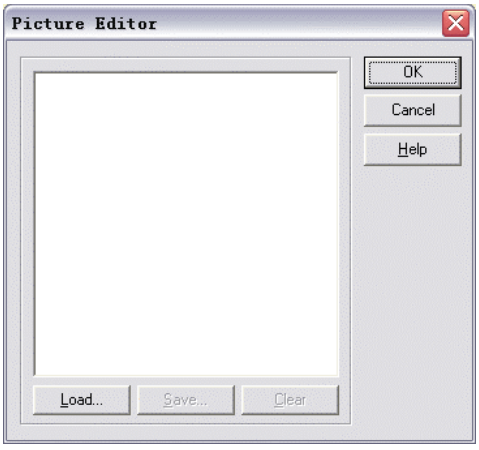

图 5.5　载入图片对话框

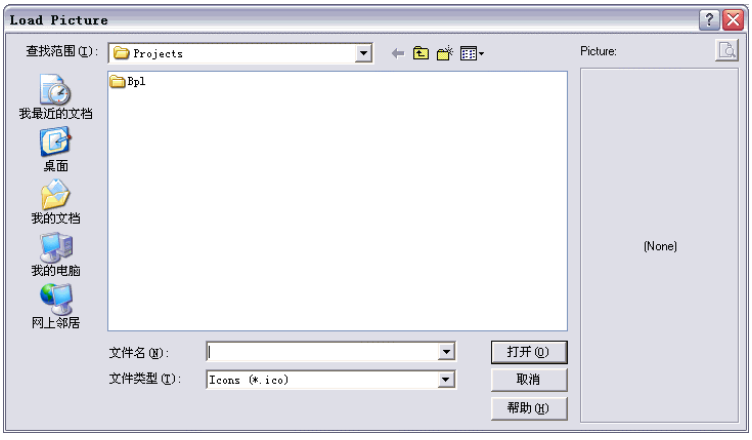

图 5.6　载入图片对话框

(4) 此时你会发现，所选择的图标已经出现在的窗体的标题栏中，运行该应用程序，并单击窗口的图标，此时的程序运行结果如图 5.7 所示。

在上面的图中，有一个菜单，我们称为系统菜单，这个大家肯定都不陌生。实际上它们在几乎所有的 Windows 程序中都会存在。

下面来修改出现在标题栏上的系统按钮。一般来说，在 Windows 应用程序中，窗口上都会有几个关于窗口的基本操作的按钮，我们称它们为系统按钮。

在 Delphi 的 TForm 中，有一个 BorderIcons 属性组，它下面有四个属性，如表 5.1 所示。

图 5.7 修改图标后的窗体

表 5.1 TForm 的 BorderIcons 属性

属性值	说明
biSystemMenu	窗体是否具有系统菜单，True 表示有，False 表示没有
biMinimize	窗体是否显示最小化按钮
biMaximize	窗体是否显示最大化按钮
biHelp	这个属性是和 BorderStyle 属性结合使用的,如果 BorderStyle 属性的值为 bsDialog、biMinimize 或者 biMaximize，那么在窗体的系统按钮的位置会出现一个“？”按钮，这是一个帮助按钮，在其他情况下，设置这个属性没有任何作用

5.1.4 改变窗体的边框

在上面的表格中，我们提到了 BorderStyle 属性，这个属性可以决定窗体的边框的形式。比如通常使用的 bsSizeable，那么此时的窗口是标准的、可以改变大小的边框类型。这个属性共有六个可以选择的值，如表 5.2 所示。

表 5.2 窗体的 BorderStyle 属性

属性的值	说明
bsDialog	不能改变大小，是标准的对话框边框
bsSingle	不能改变大小，其边框的样子在不同的操作系统下有不同的显示形式，这取决于操作系统对该类型边框的定义
bsNone	不能改变大小，无边框，没有标题栏
bsSizeable	标准的边框类型
bsToolWindow	和 bsSingle 相似，但是标题栏是小标题栏
BsSizeToolWin	和 bsSizeable 相似，但是标题栏是小标题栏

下面我们看看它们通常用在什么样的用途上。

- 当选择 Tform 的 BorderStyle 属性为 bsDialog 的时候，该窗口只有一个“关闭”系统按钮。其他的和普通窗口没有什么区别，除了不能改变大小。当我们需要自定义一个对话框时，可以使用这种类型的边框类型。
- 当选择 TForm 的 BorderStyle 属性的值为 bsSingle 的时候，窗口和普通的标准窗口看不到任何区别，除了不能改变大小。这样的窗口通常在开发那种只提供三种窗口大小（最大化、最小化或者设计好的大小）或者更少选项的时候使用。
- bsNone 属性值是在开发一些特殊界面应用程序时常使用的。可以把一个窗口的所有边框取消掉，然后利用我们自己的编程来实现个性化的窗口。例如在如图 5.8 所示的 CD-Player 程序中，我们就采用了这样的窗口形式。在这个程序中，我们利用自己制作的图片和图片按钮来替代了标准的 Delphi 窗口的标题栏。虽然这个过程稍微有些复杂，但是它可以实现具有个性特点的窗口。
- bsToolWindow 属性值和 bsSizeToolWin 是非常相似的，通常用它们来作为一个工具窗口，例如在 Delphi 中就经常看到这样的窗口。但是它们的应用是十分广泛的。例如在开发一些不是像我们常见的应用程序那样具有比较复杂界面的程序时，就可以考虑使用它们；例如要开发一个聊天软件，那么窗口就没有必要特别大，或者没有必要非得具有上面的那些系统按钮，等等。典型窗口如图 5.9 所示。

图 5.8　CD-Player 程序

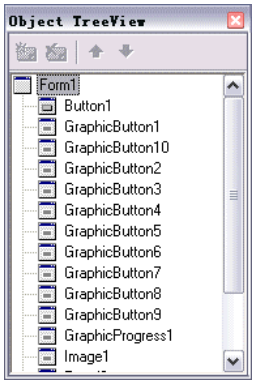

图 5.9　典型的具有 bsSizeToolWin 边框的窗口

5.1.5　窗体的状态

在一般情况下，当我们显示应用程序中窗口时，窗口通常显示在默认的位置，也就是我们在设计这个窗体的时候它的位置。但是有的时候希望它能显示在特定的位置，比如显示在屏幕的中央；或者，窗体显示时通常是在正常状态，而有时候我们需要它在显示的时候就处于最大化状态。诸如这样的情况，在 Delphi 的程序设计中是常见的。

首先看一下关于窗体位置的属性 Position。通过修改这个属性，可以使窗体显示在我们希望的位置。这个属性的各个取值和对应的含义如表 5.3 所示。

表 5.3 Tform 的 Position 属性

属性值	说明
poDesigned	窗体出现在我们设计时候的位置，并保持设计时候的宽度和高度
poDefault	窗体根据系统确定的高度和宽度出现在系统的默认位置。每次显示该窗口时，窗体都会向右下方移动。不管系统的屏幕分辨率是多少，窗体右边框总不会超过屏幕的右边界，而窗体的下边框总是在屏幕下边界的上方。当窗体移动到屏幕某个边界时，就从屏幕的左上角开始显示
poDefaultPosOnly	窗体将按照在设计时确定的尺寸显示，但是显示位置却由操作系统来确定。每次运行应用程序时，窗体将向屏幕的右下方移动，当窗体不能完整显示在屏幕上时，窗体将显示在屏幕的左上角
poDefaultSizeOnly	窗体将按照设计时的位置显示在屏幕上，但是它的尺寸却由操作系统来确定
poScreenCenter	窗体将根据设计时的尺寸显示，但是位置将位于屏幕的中央
poDesktopCenter	窗体将根据设计时的尺寸显示，但是位置将位于桌面的中央
poMainFormCenter	窗体将根据设计时的尺寸显示，但位置将位于应用程序的主窗口中央。这个属性值只适用于那些不是主窗体的窗口。如果把该属性值应用于主窗口，那么该值的作用将和 poScreenCenter 相同
poOwnerFormCenter	窗体将根据设计时的尺寸显示，但是位置将位于窗体拥有者的中央。拥有者是根据 TForm 的 Owner 属性来确定的。如果 Owner 属性是空的，也就是说，没有制定窗体的所有者，那么该属性值将和 poMainFormCenter 相同

TForm 还有一个属性，可以确定窗体显示时的状态，是正常、最大化还是最小化。这个属性就是 WindowState，该属性具有下面三个属性值，分别对应窗体的三种状态。

- wsNormal： 窗体将显示为我们设计时的状态。
- wsMinimized：窗体将显示为最小化状态。
- wsMaximized：窗体将显示为最大化状态。

提到窗体的最大化，这里我们要说一下关于窗体最大化的限制问题。并不是窗体在最大化时一定会显示为覆盖全屏幕的窗口。在 Delphi 中，它的最大化范围受 Constraints 属性的限制。这个属性实际上也是一个对象，它是从 Tpersistent 对象继承下来的。这个对象一般不会直接存在，通常是作为其他控件的附属属性，比如在这里的 TForm 对象中就包含了该属性。几乎在所有的可视控件中，几乎都存在着这个对象。它包括四个属性，那就是 MaxHeight、MaxWidth、MinHeight 和 MinWidth。它们分别对应于所属控件的最大高度、最大宽度、最小高度和最小宽度。例如在图 5.10 所示的窗口中，我们把 MaxHeight 属性设置为 480，而把 MaxWidth 属性设置成 640，那么此时当单击窗口上的最大化按钮的时候，窗口并没有覆盖这个可用空间，而是显示为一个 640×480 的窗口。

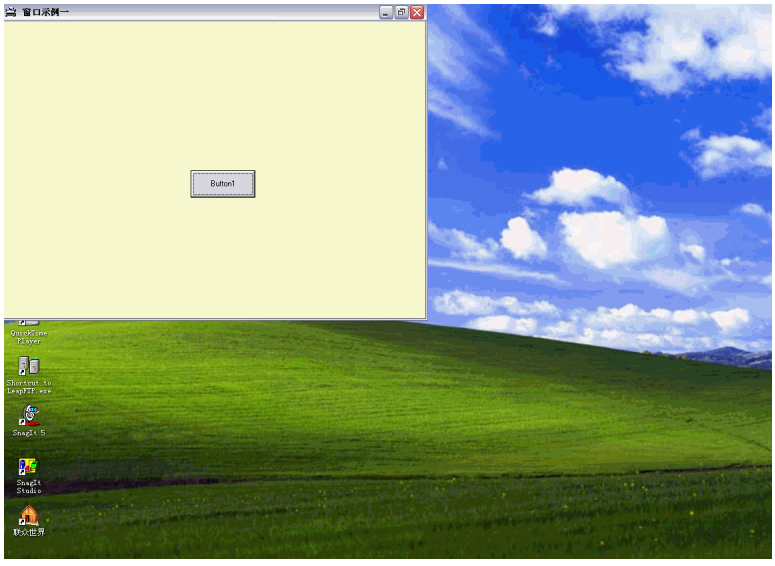

图 5.10　设置了 Constraints 属性的最大化窗体

从图上窗口的系统按钮可以看出，此时的窗体已经处于最大状态。这个功能可以帮助我们解决一个重要的问题，那就是有的时候，我们希望能够开发一个程序，它能够覆盖所有的画面，包括 Windows 系统的工具栏（包含“开始”菜单）。那么可以在程序运行的时候获取屏幕的当前分辨率，然后把这些属性赋给 Constraints 的各个属性，然后把窗体设置成最大化，那么便覆盖了屏幕上的所有内容。这种情况在设计一个工程程序或者安装程序的时候经常会使用到。

5.1.6　窗体的透明和半透明

窗体的透明与半透明是 Delphi 6.0 中的 TForm 对象新具有的一个功能。通过这些功能，可以开发更加具有个性的窗体。

在 Delphi 6.0 的 TForm 对象中，增加了两个属性：TransparentColor 和 TransparentColorValue。可以说这两个属性的出现为我们提供了重要的功能。比如，以前我们在创建不规则窗体时需要自己调用 Windows 的 API，现在就非常简单了，我们只要载入一个图片，并把 TransparentColorValue 的值设置成图片的背景颜色，然后把 TransparentColor 设置成 True，那么想要什么样式的窗口，便可以生成什么样的窗口了。这是一个多么了不起的改变！

例如在图 5.11 所示的窗口中，利用一个 Delphi 中的 Tshape 控件，把窗体的一部分作成了透明的。

这似乎还不能说明透明窗体的功能。比如要制作一个特殊界面的应用程序，那么可以首先制作一幅图片，然后把它载入到一个 Timage 图片中，然后把上面的两个属性分别设置成

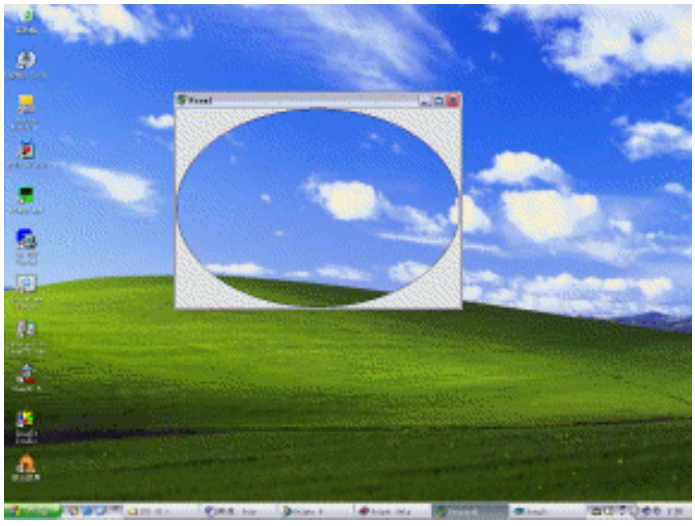

图 5.11　透明窗体

下面的值。

- TransparentColor：True
- TransparentColorValue：图片背景颜色

例如，在图 5.12 所示的 Delphi 的窗体设计器中，设计了一个载入了图片的窗体，当程序运行的时候将如图 5.13 所示。

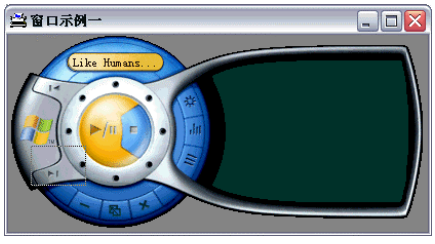

图 5.12　Delphi 的窗体设计器

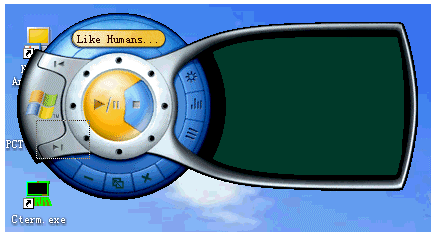

图 5.13　运行后的应用程序

说明：

上面的原始图片是从 Windows XP 操作系统的 Media Player 应用程序界面上取下来的。用户当然可以把自己设计的图片放置到上面程序的 Timage 对象中，然后按照我们介绍的内容设置窗体的相应属性，那么剩下的内容就是依次实现界面上的各个部分功能了。

在本小节开始的时候，我们还提到了半透明。这是由 Delphi 6.0 的 TForm 对象的另外两个新属性提供的，它们是：AlphaBlend 和 AlphaBlendValue，前一个决定了是否采用窗体透明选项，后一个确定了透明度。它们的设置和上面我们介绍的两个透明的属性相似，前一个是一个 Boolean 型的值，后一个是一个整型值，只不过后一个的最大值为 255。例如我们在一

个新程序的窗体上把上面的两个属性分别设置为 True 和 125，此时运行该应用程序，会看到如图 5.14 所示的程序界面。

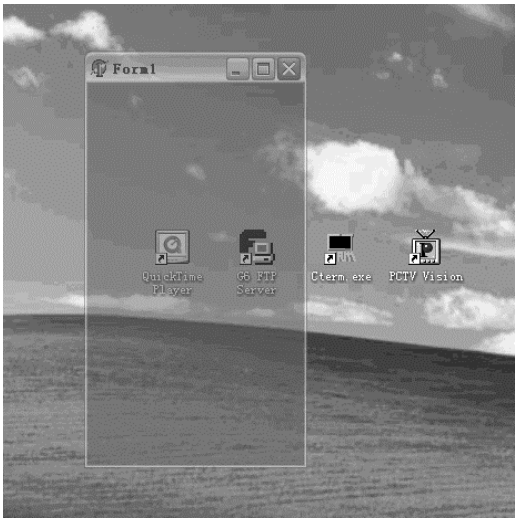

图 5.14　半透明的窗体

从上面的简单的例子我们可以想到，如果希望用户在运行我们的应用程序时，该程序界面能够淡淡地出现在用户的面前，那么就可以把窗体的 AlphaBlendValue 设置成 0，把 AlphaBlend 设置成 True，然后在窗体上放置一个 Ttimer 控件，并把它的 Interval 属性设置成 100。

说明：

关于 Timer 控件的用法我们将在后面进行详细的介绍。读者在这里可以把它理解为一个计时器，它可以每隔一定的时间执行一段代码。

然后我们可以利用 Ttimer 控件的一个 OnTimer 事件完成上面的功能。
整个程序代码如下所示。

```
unit Unit1;

interface

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

type
  TForm1 = class(TForm)
    Timer1: TTimer;
```

```
    Button1: TButton;
    procedure Timer1Timer(Sender: TObject);
    procedure FormShow(Sender: TObject);
    procedure Button1Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation

{$R *.dfm}

procedure TForm1.Timer1Timer(Sender: TObject);
var i:integer;
begin
i:=Form1.AlphaBlendValue;
if i < 255 then
    i := i+15;
if i >255 then
begin
    i := 255;
    Timer1.Enabled := False;
end;
Form1.AlphaBlendValue := i;
end;

procedure TForm1.FormShow(Sender: TObject);
begin
Timer1.Enabled := True;
end;

procedure TForm1.Button1Click(Sender: TObject);
begin
...
end;

end.
```

在上面的程序中使用了两个事件，一个是 TForm 对象的 OnShow 事件，在这个事件中我们启动了 TTimer 控件，也就是启动了计时器控件，然后在 TTimer 控件的 OnTimer 事件中把 TForm 对象的 AlphaBlendValue 值累加，直到 255 为止。读者可以利用上面的代码自己来观看窗体淡出的效果。

5.1.7　窗体的其他属性概述

从窗体的属性编辑器上可以看到 TForm 还有很多其他的属性，如果在窗体上放置一些其他控件，会发现许多控件和 TForm 一样，它们有很多共同的属性。这里我们把其中一些属性介绍一下，见表 5.4。

表 5.4　窗体 TForm 的其他属性

名称	说明
AutoScroll	当某个窗体对象的大小不能显示它上面的控件时，是否自动显示滚动条
AutoSize	根据自己所包含控件的情况自动确定本身的大小
BiDiMode	控件可以根据程序在本地运行时调整自己的外观，以确定是从左到右还是从右到左显示内容
BorderWidth	控件的边框的宽度。对于控件来说，这个值会直接影响外观。对于 TForm 来说，它影响的是 ClientWidth 和 ClientHeight 两个属性的值
ClientHeight	该属性确定了对象的可用空间高度。对窗体和一个窗口类控件来说， Height 和 ClientHeight 并不是一致的。这在窗体中反映尤为明显
ClientWidth	该属性确定了对象的可用空间宽度
Ctrl3D	是否按照 3D 格式显示对象
Cursor	指针类型。指的是当前对象或者控件的鼠标指针的样式。我们在需要表示某个控件具有特殊作用时，往往通过修改鼠标在该控件上时的指针形状来表示
Font	字体属性。在 Delphi 中，字体属性实际上也是一个对象，它们的操作方式相同。我们在后面介绍控件的时候还会介绍该属性的设置
KeyPreview	确定窗体是否在控件接受键盘事件之前对键盘事件进行处理
OldCreateOrder	指定 OnCreate 和 OnDestroy 事件的发生时间。当 OldCreateOrder 为 False 的时候（默认值），OnCreate 时间发生在所有的构造器完成之后，而 OnDestroy 事件则发生在所有的析构器被调用之前
PixelsPerInch	代表了设计窗体的系统上的字体的比例
PrintScale	代表了打印窗体的比例变化。它有三个可选值，分别代表按照窗体的比例大小打印、按照窗体的实际大小打印、按照纸张的大小缩放窗体
Scaled	确定窗体是否根据 PixelsPerInch 属性的值来调整大小，包括子控件的大小
Tag	这是 Delphi 的对象和控件中的一个附加属性，我们通常用它来存储一些附加信息
Visible	确定对象或者控件是否可见

通过上面的表格，我们介绍了许多 Delphi 中 TForm 以及其他控件所具有的一个共同的

属性。也许你会发现还有一些其他的属性我们没有介绍，因为它们都涉及一些需要进行专门介绍的主题。在后面的程序中会陆续进行介绍。

5.1.8 使用事件处理程序

面向对象编程的中心是对象，包括它的属性、方法和事件。事实上，在面向对象的编程语言中，事件是推动程序执行的重要因素。在 Delphi 中编程时，很多时候是在处理对象的事件处理程序。下面介绍一下 TForm 的一些事件。

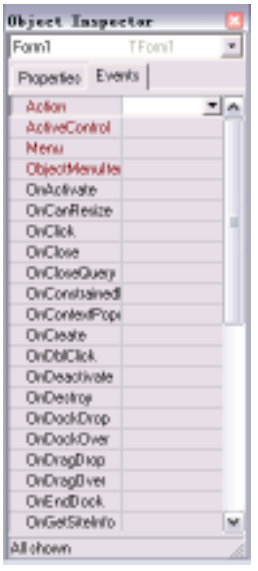

图 5.15 TForm 对象的事件

在掌握一个对象的事件的时候，特别需要注意的有以下两个方面。

- 事件发生的时刻。也就是说我们应该知道这个事件是在什么情况下发生的。比如 TForm 的 OnCreate 事件是发生在 OnShow 事件之前的。如果希望在程序创建窗体的时候进行一些相应的操作，那么应该在 OnCreate 事件中进行，而不应该在 OnShow 事件中，因为此时窗体的创建过程已经完成了。
- 事件处理程序能够提供给我们的参数。这些参数是事件在发生的时候的系统参数。比如在 OnMouseMove 事件中，能从它的参数中获得当前鼠标的位置。

单击属性编辑器上的 Events 选项卡，此时会看到 Delphi 中的一个 TForm 对象的所有事件，如图 5.15 所示。

如果要处理其中的某个事件，可以双击其中的某个项目，此时 Delphi 会自动为我们生成该事件的处理程序框架，我们只要在里面填入代码就可以了。

说明：

在 Events 选项卡以及 Properties 选项卡中，我们都会看到一些用红色表示的项目。它们有的是 Delphi 6.0 中新增加的项目，有的是原来有但是现在重新安排的项目。这些项目都十分重要。我们在介绍到相应的控件或者编程内容的时候会详细地介绍它们的用法。

例如在下面的程序中，我们编写了 Form1 的 OnCreate 事件处理程序，在这个程序中，对窗体的大小和颜色进行了重新设置。

```
procedure TForm1.FormCreate(Sender: TObject);
begin
    left:=50;
    Top:=50;
    Width:=600;
```

```
    Height:=400;
    Color:=clRed;
end;
```

在上面的程序中，可以看到事件处理程序的参数是 Sender，它是 Tobject 类型。在程序中它代表了调用这个事件处理程序的对象，比如这个就是 Form1。实际上一个事件处理程序可以被多个对象调用，只要这些对象对应的事件处理程序的说明相同。所以在这种情况下我们就需要根据 Sender 参数来判断到底是哪个对象调用了该程序。例如在下面的例子中，就为多个 TLabel 对象声明了一个事件处理程序，并根据 Sender 参数来判断是哪个对象调用了该处理程序。

```
procedure TForm1.Label4Click(Sender: TObject);
begin
    Label1.Caption := 'Not Me';
    Label2.Caption := 'Not Me';
    Label3.Caption := 'Not Me';
    Label4.Caption := 'Not Me';
    if Sender = Label1 then
      Label1.Caption := 'Clicked Me';
    if Sender = Label2 then
      Label2.Caption := 'Clicked Me';
    if Sender = Label3 then
      Label3.Caption := 'Clicked Me';
    if Sender = Label4 then
      Label4.Caption := 'Clicked Me';
end;
```

上面的程序非常简单，只是把被单击对象的标题设置成一个字符串。可以想像，在一个比较大型的应用程序中，经常会遇到这样的情况，那就是有许多同类控件，比如有很多 TLabel，它们要执行相同或者相似的操作，那么就没有必要为其中的每个控件的每个事件处理程序编写代码，只要编写一个和上面类似的代码，然后指定给每个对象对应的事件就可以了。

在上面的程序中，只要把里面涉及的四个 TLabel 对象的 OnClick 事件都指定为该处理程序，那么此时运行该应用程序，并在任意一个 Label 上单击鼠标，运行结果的窗口如图 5.16 所示。

在下面的内容中，介绍了 Delphi 的 TForm 对象的事件发生情况以及传递的参数。

- OnActivate：传递的参数为（Sender:Tobject），该事件在窗体获得焦点的时候发生。需要注意的是，用户在不同的应用程序之间切换时触发的是应用程序的 OnActivate 事件。

图 5.16　多个对象调用一个事件处理程序

- OnCanResize：传递的参数是（Sender: TObject; var NewWidth, NewHeight: Integer; var Resize: Boolean），利用该事件可以处理控件重新定义大小的方式。在这个事件处理程序中，可以修改 NewWidth、NewHeight 属性来指定控件新的大小，或者把 Resize 指定为 False 从而拒绝调整控件的大小。需要注意的是，这个事件发生在控件的尺寸进行调整之前。如果在程序中没有定义 OnCanResize 事件处理程序，或者该事件处理程序中指定允许进行尺寸调整，那么下一步将是引发 OnConstrainedResize 事件。
- OnClick：传递的参数是（Sender:Tobject），利用该事件可以处理用户在单击某个控件或者对象时程序的响应。OnClick 事件通常发生在用户在某个控件或者对象上按下并释放了鼠标左键的时候。在以下几种情况下会引发该事件。
  - 用户在一个 Grid、List 或者 ComboBox 控件中选择一个项目时。
  - 当一个 Button 或者 CheckBox 控件在焦点状态下时，按下空格键。
  - 当前活动窗口具有一个默认按钮（由窗体的 Default 属性指定），按下 Enter 键时。
  - 当前活动窗口具有一个取消按钮（由窗体的 Cancel 属性指定），按下 Esc 键时。
  - 用户按下一个按钮或者选择快捷键时。
  - 单选按钮的 Checked 属性被设置成 True 时。
  - 单击菜单项目时。
  - 对于一个窗体来说，OnClick 事件发生在用户单击窗体上的一个空白区间或者一个禁用的控件时。
- OnClose：传递的参数是(Sender: TObject; var Action: TCloseAction)，利用该事件可以在窗体关闭的时候处理一些特殊的过程。OnClose 事件指定了窗口在要关闭时要运行什么样的处理程序。关闭事件发生在程序调用 Close 方法关闭窗口或者用户单击关闭按钮时。在这个事件处理程序的参数中有一个 TcloseAction 类型的参数，它

就是我们要指定的处理程序参数。这是一个十分重要的参数，它有以下几个选项。

- caNone：该值将不允许窗口的关闭，所以任何事情都不会发生。
- caHide：此时窗口不会关闭，只是隐藏起来，应用程序仍然在运行。
- caFree：窗口将会关闭，并且窗口所占用的内存将全部释放。
- caMinimize：窗口最小化，而不是关闭。对于 MDI 子窗口来说，这是默认的操作。

❖ OnCloseQuery：传递的参数是(Sender: TObject; var CanClose: Boolean)。利用这个事件可以指定在什么条件下窗口可以关闭。在该事件处理程序的参数中，有一个 Boolean 类型的 CanClose 变量，它决定了一个窗口是否可以关闭。它的默认值是 True。 我们可以利用这个事件询问用户是否要立即关闭当前窗口。例如我们在很多文档编辑软件中，经常会遇到这样的情况，在关闭程序之前，询问用户是否保存当前文档。

❖ OnConstrainedResize：传递的参数是(Sender: TObject; var MinWidth, MinHeight, MaxWidth, MaxHeight: Integer)。利用该事件可以在程序试图改变控件大小时，来调整控件的尺寸范围。在这个事件的参数中有四个参数变量，它们分别对应控件或者对象中的 Constraints 属性的四个值。这个事件处理程序可以在控件应用新的宽度和高度之前调整这些值。一旦这些值应用到控件中，控件的高度和宽度也会相应改变。在该事件之后，将随即发生 OnResize 事件来进行最后的调整和响应。需要注意的是，该事件处理程序是在 OnCanResize 事件之后发生的。

❖ OnContextPopup：传递的参数是(Sender: TObject; MousePos: TPoint, var Handled: Boolean)。该事件发生在用户使用鼠标或者键盘来要求弹出一个菜单的时候，是由系统消息 WM_CONTEXTMENU 引发的。而该消息是用户在右击鼠标或者按下 Shift+F10 组合键要求弹出快捷菜单时产生的。这个事件在处理弹出菜单时是十分有用的，特别是在窗口没有弹出菜单（是由 PopupMenu 属性确定），或者指定弹出菜单的 AutoPopup 属性没有设置成 True 的情况下。另外，在控件关联弹出菜单的 AutoPopup 属性为 True 时，该事件也可以用来重载自动的上下文菜单。在这种情况下，如果事件处理程序要显示自己的菜单，应该把 Handled 参数设置成 True，来禁止默认的上下文菜单显示。处理程序中的 MousePos 参数指明了鼠标的位置，注意，这里采用的是在客户坐标系里面。如果事件不是由一个鼠标事件引发的，那么 MousePos 的值为（–1, –1）。

❖ OnCreate：该事件处理程序传递的参数为(Sender: TObject)。利用该事件处理程序可以在创建窗体时进行特殊的处理。我们可以应用这个事件处理程序，也可以重载窗体的构造器，但是我们不能同时做这两件事情。当一个窗体被创建并显示时，将依次发生下面的事件：OnCreate、OnShow、OnActivate 和 OnPaint

❖ OnDblClick：该事件处理程序传递的参数为(Sender: TObject)。该事件在双击对象或者控件时发生。

- OnDeActivate：该事件处理程序传递的参数为(Sender: TObject)。该事件发生在窗体失去焦点时。利用该事件可以在一个应用程序的焦点从一个窗口移动到另一个窗口时进行特殊的程序处理。如果焦点是从一个应用程序移动到另外一个应用程序，那么不会发生该事件。
- OnDestroy：该事件处理程序传递的参数为(Sender: TObject)。该事件发生在窗体被析构时，也就是窗体被释放的时候。在这个事件中，我们应该释放所有在 OnCreate 事件中创建的对象。
- OnHelp：该事件处理程序传递的参数为(Command: Word; Data: Integer; var CallHelp: Boolean)。它发生在用户要求帮助时。HelpContext 和 HelpJump 方法会自动引发该事件。如果希望禁止操作系统的帮助程序启动，那么可以把 CallHelp 设置成 False。需要注意的是，这是一个函数事件处理程序，它是有返回值的。如果处理成功，则返回 True，否则返回 False。
- OnHide：该事件处理程序传递的参数为(Sender: TObject)。该事件发生在窗体隐藏时。
- OnPaint：该事件处理程序传递的参数为(Sender: TObject)。该事件发生在窗体重绘时。窗体任何特殊的绘制都应该在这个事件中完成。需要注意的是，该事件发生在窗体上的控件重绘之前。
- OnResize：该事件处理程序传递的参数为(Sender: TObject)。该事件发生在对象或者控件的尺寸重新调整之后。需要注意的是，如果要对窗体的尺寸进行限制，那么应该在前面介绍的几个事件中完成，而不应该在这个事件中。因为在这个事件中关于尺寸的操作会引起界面的闪烁。
- OnShow：该事件处理程序传递的参数为(Sender: TObject)。该事件发生在对象或者控件显示时。

通过上面的内容，我们把窗体甚至许多控件的一些共同事件都进行了说明。读者可能也注意到有几个事件没有进行介绍。这几个事件一部分是和鼠标、键盘有关，一部分和对象的停靠有关，我们将在后面的内容中进行详细的介绍。

说明：

在上面的事件中，我们经常提到一个名词：焦点。所谓焦点就是输入点，焦点存在于不同的应用程序之间、同一个程序的不同窗口之间，同一个窗口的不同控件之间。OnActivate 和 OnDeActivate 事件就是和焦点相关的事件。

5.2 用于处理文本的控件

处理文本无疑是编写任何应用程序时都要遇到的一个问题，也是在高级程序开发过程中

的一个基本技术。虽然它听起来是如此简单，但是事实上，要真正全面地掌握这个技术却需要足够的耐心和细心。

文本的处理包括文本的显示、文本的输入输出，文本字体的变化，以及根据文本的变化程序需要进行相应的操作。另外，一个并不经常使用的技术就是控件中文本的具体处理，例如对文本的格式及各个字符的处理。下面我们就结合各个控件的使用介绍这些方面的问题。

5.2.1　标签控件

标签控件（Label）的主要用途是用来标注其他的控件，或者对程序操作过程中的一些问题进行说明。要制作一个标签，可以使用控件选项板上 Standard 选项卡中的 Tlabel 控件。这个控件在选项卡上的位置如图 5.17 所示（凸起的控件即标签控件）。

图 5.17　Standard 选项卡上的标签控件

TLabel 控件是少数非窗口类控件之一，它是从 TCustomLabel 类继承下来的，而 TCustomLabel 则是从 TGraphicControl 类继承下来的。可以利用上面介绍 TForm 对象时获得的关于属性和事件的认识来了解关于 Tlabel 控件以及其他控件的属性和事件。在 TLabel 对象中，使用最频繁、也是最关键的属性有以下几个。

- Caption：TLabel 控件的标题。
- AutoSize：是否根据标题的内容自动调整控件的大小。
- Color：控件的颜色。注意这个颜色是控件的颜色，而不是字体的颜色。
- FocusControl：焦点控件。当 TLabel 控件获得焦点时，输入焦点自动移动到由该属性指定的控件中，通常是个 TEdit 控件。
- Enabled：启用或禁用。确定 TLabel 控件是否禁用。
- Font：设置控件中标题的字体格式，包括字体、大小等。
- Layout：标题的布局，指的是标题在控件中是上对齐、中对齐还是下对齐。
- Transparent：确定控件是否透明。
- WordWrap：确定当标题的内容长度超过了控件的宽度时，是否绕行显示。

下面我们创建一个使用标签控件的应用程序，以演示上面的这些属性对控件具有何种影响。

(1) 首先单击 File 菜单中的 New Application 建立一个新的应用程序。这是一个空的应用程序，它的窗体设计器上没有任何内容。

(2) 然后在窗体上放置一个 TPanel 控件，并按照表 5.5 来设置它的属性。

表 5.5 Panel1 控件的属性

属性	设置值
Color	clTeal
Align	alRight
Caption	空

说明：

在上面的控件中，我们使用了 Align **属性，这个属性用来确定控件，特别是具有窗口句柄的控件的对齐问题。**

(3) **然后在** Panel1 **控件上放置一个** TLabel **控件，并按照下面的表格设置它的属性。**

表 5.6 Label1 控件的属性

属性	设置值
Color	ClSilver
Width	330
Height	240
Autosize	False
Caption	wok wok wok wok wok wok wok wok wok wok wok wok wok wok wok wok wok wok wok wok wok

(4) **下面在窗体上放置一些可以让用户修改** Label1 **控件属性的控件。这些控件包括三个** TEdit **控件，两个** TLabel **控件，四个** TCheckBox **控件，两个** TButton **和一个** TComboBox **控件。然后按照表** 5.7 **的内容修改它们的属性值。**

表 5.7 窗体上控件及其属性的设置

控件	属性	设置值
Edit1	Name	EdCaption
Edit2	Name	EdWidth
	Text	330
Edit3	Name	EdHeight
	Text	240
Label2	Caption	&Width
	FocusControl	EdWidth
Label3	Caption	&Height
	FocusControl	EdHeight
CheckBox1	Name	CHKAutosize
	Caption	Autosize
CheckBox2	Name	CHK Enabled
	Caption	Enabled

续表

控件	属性	设置值
CheckBox3	Name	CHKTransparent
	Caption	Transparent
CheckBox4	Name	CHKWordwrap
	Caption	Wordwrap
Button1	Name	BtnColor
	Caption	Color…
Button2	Name	BtnFont
	Caption	Font…
ComboBox1	Name	CmbLayout
	Items	tlTop，tlCenter，tlBottom
	ItemIndex	0

(5) 在上面的表格中，在设置两个 TLabel 控件的 Caption 属性时，我们使用了“ & ”符号,这在 Delphi 中是一个特殊的符号；另外，也使用了 FocusControl 属性。“ FocusControl ”属性通常和“ ShowAccelChar ”配合使用。从字面上理解这两个属性的话，它们的意思是“焦点控件”和“显示加速键”。加速键通常称为快捷键，用字母加下划线来表示。观察我们使用的应用程序，它们的菜单标题上一般都显示了对应的快捷键。在 Delphi 程序设计中通常用“ & ”符号加上需要的字母来定义快捷键。ShowAccelChar 属性就是用来控制在 TLabel 的文本中是否显示快捷键。如果这个属性的值为“ True ”(这也是默认值)，那么“ & ”符号后面定义的字符就是该 TLabel 的快捷键，否则“ & ”符号就作为普通符号进行设置。

(6) 在使用 ComboBox1 控件的时候，我们需要为它的 Items 属性增加项目，在属性编辑器上单击该属性旁边的按钮，此时会显示如图 5.18 所示的对话框。可以在该对话框中依次输入要输入的项目，每个项目后面要按 Enter 键。

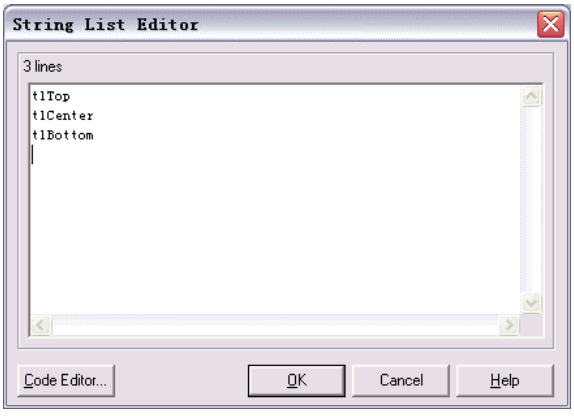

图 5.18　CmbLayout 控件的 Items 属性编辑器

(7) 单击控件面板上最右边的箭头按钮，找到 Dialogs 选项卡，然后选择上面的 FontDialog 和 ColorDialog 控件，并放置到窗体上。

接下来的工作就是要为每个控件编写事件处理程序，用来完成我们希望的功能了。下面列出的是程序的完整代码。

```
unit Unit1;

interface

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

type
  TForm1 = class(TForm)
    Panel1: TPanel;
    EdCaption: TEdit;
    CHKAutoSize: TCheckBox;
    CHKEnabled: TCheckBox;
    CHKTransparent: TCheckBox;
    ColorDlg: TColorDialog;
    BtnColor: TButton;
    FontDlg: TFontDialog;
    BtnFont: TButton;
    CmbLayout: TComboBox;
    CHKWordwrap: TCheckBox;
    Label1: TLabel;
    EdWidth: TEdit;
    EdHeight: TEdit;
    Label2: TLabel;
    Label3: TLabel;
    procedure EdCaptionChange(Sender: TObject);
    procedure CHKAutoSizeClick(Sender: TObject);
    procedure CHKEnabledClick(Sender: TObject);
    procedure CHKTransparentClick(Sender: TObject);
    procedure CHKWordwrapClick(Sender: TObject);
    procedure BtnColorClick(Sender: TObject);
    procedure BtnFontClick(Sender: TObject);
    procedure CmbLayoutChange(Sender: TObject);
    procedure EdWidthKeyPress(Sender: TObject; var Key: Char);
    procedure EdHeightKeyPress(Sender: TObject; var Key: Char);
  private
```

```
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation

{$R *.dfm}

procedure TForm1.EdCaptionChange(Sender: TObject);
begin
    Label1.Caption := EdCaption.Text;
end;

procedure TForm1.CHKAutoSizeClick(Sender: TObject);
begin
    Label1.AutoSize := CHKAutoSize.Checked;
    if not Label1.AutoSize then
    begin
        Label1.Width := 330;
        label1.Height := 240;
    end;
end;

procedure TForm1.CHKEnabledClick(Sender: TObject);
begin
    Label1.Enabled := CHkEnabled.Checked;
end;

procedure TForm1.CHKTransparentClick(Sender: TObject);
begin
    Label1.Transparent := CHKTransparent.Checked;
end;

procedure TForm1.CHKWordwrapClick(Sender: TObject);
begin
Label1.WordWrap := CHKWordwrap.Checked;
end;
```

```
procedure TForm1.BtnColorClick(Sender: TObject);
begin
    if ColorDlg.Execute then
       Label1.Color := ColorDlg.Color;
end;

procedure TForm1.BtnFontClick(Sender: TObject);
begin
    if FontDlg.Execute then
       Label1.Font := FontDlg.Font;
end;

procedure TForm1.CmbLayoutChange(Sender: TObject);
begin
  case Cmblayout.ItemIndex of
  0:
       Label1.Layout :=   tlTop;
  1:
       Label1.Layout := tlCenter;
  2:
       Label1.Layout := tlBottom;
  end;
end;

procedure TForm1.EdWidthKeyPress(Sender: TObject; var Key: Char);
begin
if Key=chr(13) then
  Label1.Width := Strtoint(edWidth.Text);
end;

procedure TForm1.EdHeightKeyPress(Sender: TObject; var Key: Char);
begin
if Key=chr(13) then
  Label1.Height := strtoint(EdHeight.Text);
end;


end.
```

运行该应用程序，并试着改变上面的对应选项，此时就会发现右边的 Label1 **控件的外观也相应随之改变。**

单击上面程序中的 Color **窗口，此时会显示如图** 5.19 **所示的对话框。**

图 5.19　程序中显示的标准颜色选取对话框

可以为 Label1 控件选择一个颜色，比如这里选择了黄色。

单击程序中的 Font 按钮，此时会显示如图 5.20 所示的标准字体对话框。可以在这个对话框中修改 Label1 控件的字体格式，包括字体的型号、大小、类型、颜色，等等。

图 5.20　程序中显示的标准字体对话框

修改后的程序运行结果如图 5.21 所示。

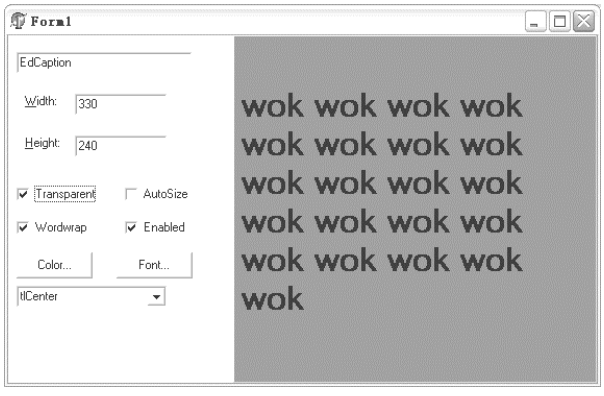

图 5.21　程序的运行结果

5.2.2 文本框控件

TLabel 控件用来显示单行文本是十分方便的，但是它有一些缺点。由于它是非窗口类的控件，所以它不能接受输入，也不能响应 Change 事件，也就是说我们无法在 TLabel 控件中输入文本，而且更无法根据文本的变化进行相应的程序操作。如果要达到这样的目的，使用文本框控件（TEdit）将是首选。

从处理单行文本的角度来说，文本框控件可以说是应用最为广泛、功能最为强大的控件了。它不仅可以像 TLabel 控件一样进行文本显示，还可以让用户输入文本，并根据文本的变化进行相应的程序操作。另外还可以对其中的文本进行复杂的处理工作。下面我们就来介绍如何使用文本框软件完成这样的功能。

TEdit 控件的主要用途是建立一个标准的 Windows 文本框，文本框的作用是让用户输入文字。当用户在文本框中键入字符时，TEdit 控件的 Text 属性总是显示文本框中的最新内容，并且将触发 OnChange 事件，表示文本框中的内容发生了变化。

TEdit 控件的最常见用法是用来获得用户输入的一个字符串。

TEdit 控件的 Text 属性是一个字符串属性，它包含了该控件显示的值。我们曾经说过，TEdit 和 TLabel 相比的一个重要优势是可以响应 OnChange 事件，也就是说，它会根据文本的变化进行相应的处理。例如在下面的过程中，我们将建立一个用来演示如何使用 TEdit 的这一功能的应用程序。

(1) 首先新建一个应用程序，然后把 Form 的字体设置成 12 号字体。这样可以使后面添加的所有控件在默认情况下都采用这个字号设置。

(2) 然后在 Form 上放置一个 TLabel 控件，把它的 Name 属性修改为“ LbInput ”，Caption 属性修改为“ 输入一个字母 ”。

(3) 在 LbInput 控件的旁边放置一个 TEdit 控件。把该控件的 Name 属性修改为‘ EdInput ”。

(4) 选中 EdInput，找到该控件的 CharCase 属性，从它提供的列表中选择 ecUpperCase。这个属性是控制 TEdit 控件文本中的字母转换形式的。可以选择 ecNormal，这就是正常状态的属性值。此时输入什么样的字母就是什么样的字母。如果选择 ecUpperCase，那么不管输入的是小写字母还是大写字母，统一变成大写字母。相反，ecLowerCase 属性值将把所有输入的字母全部变成小写字母。

(5) 找到 MaxLength 属性，把这个属性值修改为“ 1 ”。这个属性值是用来控制 TEdit 中容纳的字符串长度的。如果这个值是“ 0 ”，那么字符串的长度没有限制，否则就限制在它所代表的范围之内。

(6) 找到它的 Text 属性，把它修改为“ A ”。

(7) 然后在 Form 上再放置一个 TLabel 控件，把它的 Name 属性修改为“ LbShow ”。再把它的 Color 属性值修改为“ ClBlack ”，即黑色。字体的颜色设置为绿色。

(8) 此时双击 EdInput，或者到属性编辑器的 Events 选项卡，然后双击 OnChange 事件。此时会切换到代码编辑器中。然后输入下面的代码：

```
LbShow.Caption := EdInput.Text ;
```

(9) 把 Form1 的 Caption 属性修改为“TEdit 控件的 OnChange 事件”之后，运行应用程序。程序的运行结果如图 5.22 所示。

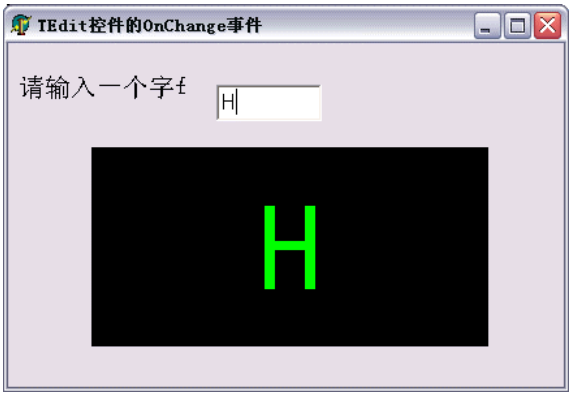

图 5.22　TEdit 控件的 OnChange 事件示例

以上只是如何使用 OnChange 事件的一个示例。利用 TEdit 的这个功能可以实现许多看似复杂的功能。

除了上面的用法之外，TEdit 控件还有一些其他的重要属性和用法。实际上 TEdit 控件的一些属性和 TLabel 控件的许多属性都是相同或者类似的。除了一些具有自己的特色的属性和用法之外，其他属性就不再介绍了，读者可以参考前面的内容来了解这些属性的用法。这里主要介绍 TEdit 特有的或者用法不同的属性和方法。我们在后面介绍其他控件的用法时也是如此，对于已经介绍过的用法相同的属性和方法就不再重复了。

AutoSelect 属性是一个布尔型值。如果这个属性设置为“True”，则当输入焦点移动到文本框时，文本框的文字自动被选中（用加亮表示），此时按下任何一个键，文本框中原文本将会被替换掉。需要说明的是，这个功能有时可能正是用户所需要的，而有时可能会引起混乱。如果不需要这样的功能，只要把这个属性设置成“False”就可以了。

TEdit 控件也有一个 AutoSize 属性，但是这个属性和 TLabel 控件的同名属性却不尽相同。TLabel 控件的 AutoSize 属性会在高度和宽度两个方面进行自动调整，而 TEdit 的 AutoSize 属性则只是控制在高度方向上进行调整。也就是说，如果把 TEdit 控件的 AutoSize 属性设置为“True”，那么 TEdit 控件只是自动调整自己的高度来适应字体的大小，而在宽度上没有变化。如果设置为“False”，那么 TEdit 将保持当前的固定大小，而不管能否容下当前设置的字体大小。

在上面的演示示例中，我们看到的都是 TEdit 默认的边框情况。事实上可以通过

BorderStyle 属性来修改文本框的边框类型。默认值是“ bsSingle ”，表示边框是单线，如果把它设置为“ bsNone ”，表示没有边框。如果把这个属性和“ Color ”属性相配合，可以设计出 TEdit 的多种表现形式。

TEdit 控件有一个 HideSelection 属性，如果把这个属性设置为“ True ”，那么当把输入焦点从该 TEdit 控件上移走时，原来文本框中选中的内容不再表现为选中状态（即不再加亮显示）。

OEMConvert 属性可以设置是否要把键入的 Ansi 字符转换成 OEM 字符，默认值为“ False ”，只有在输入文件名的时候才把这个属性改为“ True ”。

TEdit 也可以用来让用户输入密码。为了防止别人看到密码的内容，需要对密码（也就是 TEdit 的 Text 值）进行隐藏。这时可以使用 TEdit 的 PasswordChar，这个属性的默认值是“ #0 ”，这不是表示 0，而是 Ansi 字符中的 Ansi 码为 0 的那个字符。此时将正常显示 TEdit 的文本。如果这个属性设置成了其他的任何值，那么将用这个值来替代输入的字符。当然这个属性只是改变了 TEdit 的显示形式，并没有改变 TEdit 的 Text 属性本身，也就是说，我们仍然可以使用自己所熟悉的方法来存取 TEdit 的 Text 值。

在有的情况下，TEdit 控件只是作为显示文本的控件来使用，此时的 TEdit 控件和 TLabel 控件十分类似。那么可以使用 ReadOnly 属性来禁止在该文本框中输入字符串。在默认的情况下，该属性的值是“ False ”，当把它设置成“ True ”的时候，便不能在文本框中输入字符串了。这个属性在使用时存在一个问题，就是虽然不能输入内容，但是光标仍然可以在这个控件中来回移动，给人的错觉好像是可以输入文本似的。所以还应该把 Enable 属性设置为“ False ”。

TEdit 也有一些方法，使用它们可以非常容易地在运行期改变 TEdit 的内容。例如 Clear 方法，这个方法可以清除文本框中的内容。例如下面的语句：

```
Edit1.Clear;
```

实际上我们更习惯于用下面的代码来替代上面的代码：

```
Edit1.Text:=’’;
```

5.2.3 静态文本框控件

Delphi 还提供了一个控件，称为静态文本框控件（TStaticText）。所谓静态文本，非常类似于标签，它是不可编辑的。与 TLabel 控件不同的是，TStaticText 控件是从 TCustomStaticText 类继承来的，而 TCustomStaticText 类则是从 TWinControl 类继承来的。也就是说，TStaticText 控件是窗口类控件，它可以有窗口句柄，这是它和 TLabel 控件的最大区别。在其他方面，两者几乎是一模一样的。如果需要使用具有窗口句柄显示文本的控件时，可以选用静态文本框控件。

由于其他用法和 TLabel 控件完全相同，这里就不再赘述了。

5.2.4　格式化文本框控件

在 Additional 选项卡上有一个 TMaskEdit 控件，又称为格式化文本框控件。从名称上就可以看出，这个控件和 TEdit 控件相类似，它的作用也是建立一个用于用户输入的文本框。所不同的是 TEdit 控件允许用户输入任意字符，而 TMaskEdit 控件则可以对用户输入的文本进行格式限制，用户只能按照指定的格式输入。这种按照格式输入的文本框在输入诸如时间、日期、电话号码等具有通用格式的文本时是非常有用的。

TMaskEdit 控件和 TEdit 控件十分类似，主要的区别体现在 EditMask 属性上。这个属性用来指定用户输入时必须遵循的格式，如果用户输入的字符不符合格式要求，这个字符将被过滤掉，也就是说文本框将拒绝接受该字符。

在设计期间，要修改 EditMask 属性，可以在属性编辑器中单击该属性旁边的省略号按钮，Delphi 将显示如图 5.23 所示的 Input Mask Editor 对话框。

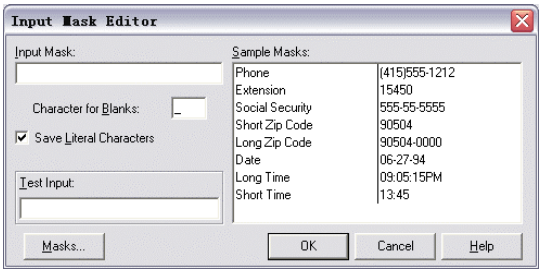

图 5.23　Input Mask Editor 对话框

在这个对话框中，可以在 Input Mask 框中用 Delphi 提供的格式化符号定义一个格式。也可以在 Sample Masks 列表框中选择一个预设的格式。Sample Masks 列表框中列出了一些常用的格式，左边是格式的名称，右边是格式的具体形式。另外还可以单击 Masks 按钮来打开包含有格式的文件（这些文件的后缀名为 DEM）。

如果在对话框中选中 Save Literal Characters 复选框，Text 属性将和 EditText 属性一样，都是文本框中格式化后的内容。如果清除该复选框，Text 属性中的值将只是文本框中实际键入的字符，不带格式。EditText 属性总是带有格式的。

Character for Blanks 框用于指定代表空白的字符，默认的是“_”。也可以指定其他字符来表示空白。

上面已经提到了，TMaskEdit 控件除了 Text 属性之外还有一个 EditText 属性。这两个属性的差别在上面也做了简单的陈述。下面我们就用一个应用程序来具体地比较它们的区别。

(1) 新建一个应用程序，然后把 Form 的字体修改为 12 号字体，设定控件在默认的情况下能够采用这样的字体大小。

(2) 然后在 Form 上放置一个 TMaskEdit 控件、两个 TEdit 控件以及三个 TLabel 控件。

(3) 把 TmaskEdit 控件的 EditMask 属性设置为预设格式中的 Date。并清除 Input Mask Editor 对话框中的 Save Literal Characters 复选框。

(4) 然后按照图 5.24 所示修改另外三个 TLabel 控件的 Caption 属性。

(5) 双击程序中的 TMaskEdit 控件，然后在它的 OnChange 事件中键入下面的代码：

```
procedure TForm2.MaskEdit1Change(Sender: TObject);
begin
    edit1.text:=Maskedit1.text;
    edit2.text:=maskedit1.edittext;
end;
```

(6) 运行应用程序，然后在 TMaskEdit 控件中按照指定的格式输入字符，然后观察下面的两个文本框中的文本区别，如图 5.25 所示。

图 5.24 修改应用程序中其他对象的属性

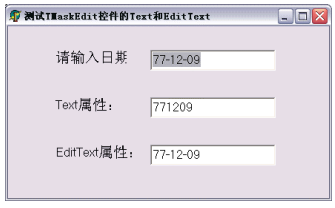

图 5.25 Text 属性和 EditText 属性的区别

从程序运行的结果来看，EditText 属性是经过格式化的字符串。

5.2.5 备注控件

在上面的介绍中，虽然 TLabel 控件可以显示多行文本，但是从本质上来说，它显示的仍然是单行文本，只不过用多行进行显示罢了，而且操作过程非常不方便。Delphi 提供了一个可以用来显示多行文本的控件——TMemo 控件。

TMemo 控件是从 TEdit 控件继承下来的，所以它的许多属性和 TEdit 控件大体相同。这里主要介绍它的一些与 TEdit 控件不同的属性和方法。

和 TEdit 控件相比，TMemo 控件的一个重要的属性是 Lines 属性。TMemo 控件和 TEdit 控件一样也有一个 Text 属性。虽然在编辑文本时也可以使用 TMemo 控件的 Text 属性来访问出现在 TMemo 控件中的文本，但是这样访问的是控件中的所有文本。如果使用 Lines 属性来访问控件的文本，功能就要强大许多。

首先观察一下这两个属性的区别，例如，在一个 Form 上放置一个 TMemo 控件，然后放置两个 TButton 控件，并在两个 TButton 控件的 OnClick 事件中输入下面的代码。

```
procedure TForm1.BtnTextClick(Sender: TObject);
```

```
begin
    memo1.Text := '在这个示例中我们使用了 Text 属性';
end;

procedure TForm1.BtnLinesClick(Sender: TObject);
begin
    Memo1.Lines[0]:='在这个示例中我们使用了 Lines 属性';
end;
```

运行应用程序可以发现，当赋值的字符串比 TMemo 的一行所能容纳的字符少的时候，上面的两个过程是一样的。如果赋值的字符串比较长，前一个过程会改变所有的文本，而后一个过程只是改变文本的第一行。如果赋予的值超过了第一行所能容纳的范围，其余的部分将绕行到其他行，而原来第二行以及以后的文本将附加到这些字符的后面。

从这个例子中，不难想象在运行期要部分改变或者读取 TMemo 控件的文本，最好使用 Lines 属性。

另外，由于 Lines 属性是一个 TStrings 对象，所以可以使用 TStrings 的一些方法来处理 Lines。例如，如果要从一个文本文件中载入文本，可以使用下面的方法。

```
procedure TForm1.Button1Click(Sender: TObject);
begin
    memo1.lines.LoadFromFile('c:\autoexec.bat');
end;
```

运行该程序，结果如图 5.26 所示。

图 5.26　从文件中载入文本

由于 TMemo 控件是适用于处理多行文本的，因而可能存在这样的情况：要处理的文本非常长，可能完全超出了从屏幕上能够看到的范围。为了处理这种情况，Delphi 为 TMemo 提供了一个 ScrollBars 属性。使用这个属性可以决定是否为 TMemo 添加滚动条，以及添加哪些滚动条。它有四个可以选择的值。

- ssNone：　不添加任何滚动条（默认设置）。

- ssHorizontal：加上水平滚动条。
- ssVertical： 加上垂直滚动条。
- ssBoth： 加上水平滚动条和垂直滚动条。

当我们在 TMemo 控件中编辑文本时，难免会使用 Enter 键和 Tab 键。我们都知道，可以在窗体上设置一个默认按钮，那么在按下 Enter 键的时候，相当于按下了该默认按钮。当我们在 TMemo 中换行时也需要按下 Enter 键，那么 Delphi 如何区分这些情况呢？Delphi 为 TMemo 提供了一个 WantEnter 属性。如果把这个属性设置成“ True ”，则当按下 Enter 键时，就是换行；如果设置为“ False ”，那么当按下 Enter 键时就是按下了默认按钮，在这种情况下，如果需要换行，必须使用 Ctrl+Enter 组合键。

同样的道理，当我们在程序中按下 Tab 键时，程序的焦点将按照 TabOrder 属性规定的顺序在各个控件之间来回切换。而我们在编辑文本时，可能也会使用 Tab 键来产生制表位。那么 Delphi 就为 TMemo 提供了一个 WantTab 属性，它的作用和 WantEnter 类似，只不过作用的键是 Tab 而已。

5.3 使用命令按钮

实际上，在前面的介绍中，我们已经多次地使用了命令按钮中的 TButton 控件。从前面的程序中可以看出，这些控件是用户向应用程序发命令的主要形式之一。但是，在 Delphi 中，还为用户提供了其他几种形式的命令按钮。由于它们的存在，使得创建丰富多彩的应用程序图形化界面成为可能。

5.3.1 按钮控件

按钮控件（TButton）同 TEdit 控件一样，是 Delphi 中应用范围最为广泛的控件之一。它的主要作用是建立一个标准的 Windows 命令按钮。按钮的使用比较简单，当用户单击按钮时，将触发它的 OnClick 事件。可以把按钮设置成默认按钮，当用户按下 Enter 键时就相当于按下了这个按钮。也可以把一个按钮设置成取消按钮，用户按下 Eec 键时就相当于单击这个按钮并触发了 OnClick 事件。

下面我们就来详细地介绍如何使用 TButton 控件。如果要把一个 TButton 控件设置成窗体上的默认按钮，则可以把该控件的 Default 属性设置成“ True ”。如果在一个窗体上有多个按钮被设置成了默认按钮，那么当按下 Enter 键时，相当于按下其 TabOrdre 值最小的按钮。如果要把一个按钮设置成取消按钮，那么可以把该控件的 Cancel 属性设置成“ True ”。同样，如果多个按钮被同时设置成了取消按钮，那么当按下 Enter 键时，相当于按下其 TabOrdre 值最小的按钮。

说明：

如果在按下 Enter 键之前，把焦点移动到某个按钮上，那么在按下 Enter 键时，响应单击的按钮就是具有焦点的这个按钮，而不是上面所介绍的 TabOrder 属性值最小的那个按钮。

Delphi 为把 TButton 方便地应用于模式对话框，为 TButton 提供了 ModalResult 属性。在默认情况下这个属性的值是“ mrNone ”。如果把它设置成其他值，按下这个按钮将自动关闭对话框而无需响应按钮的 OnClick 事件。利用 ShowModal 函数将返回 ModalResult 属性的值。ModalResult 属性有如表 5.8 中所示的预设值可以使用。

表 5.8　ModalResult 属性的预设值

预设值	返回值
mrNone：	返回 0
mrOk：	返回 idOk
mrCancel：	返回 idCancel
mrAbort：	返回 idAbort
mrRetry：	返回 idRetry
mrIgnore：	返回 idIgnore
预设值	返回值
mrYes：	返回 idYes
mrNo：	返回 idNo
mrAll：	返回 idNo+1
mrNotoAll	返回 idNo+2
mrYestoAll	返回 idNo+3

例如在下面所创建的应用程序中，我们就利用了几个设置了 ModalResult 属性的 TButton。

(1) 首先建立一个新的应用程序，然后把 Form1 的字体大小设置成 12 号字。

(2) 在上面放置一个按钮和一个文本框，把它们的 Name 属性分别修改成‘ BtnShowForm ”和“ EdResult ”。

(3) 单击工具栏上的 New Form 按钮，为应用程序加入一个新的窗体，把它的 Caption 属性修改为“ ModalResult 属性测试对话框 ”。然后把它的 Name 属性修改为“ FrmDialog ”。在属性编辑器上找到 FrmDialog 窗体的 BorderStyle 属性，然后修改为“ bsDialog ”。

(4) 在 FrmDialog 上放置一个 TLabel 控件和两个 TButton 控件。把它们的 Name 属性修改为“ LbNote ”、“ BtnOk ”和“ BtnCancel ”，然后把 BtnOk 按钮的 ModalResult 属性设置为“ mrOk ”，而把 BtnCancel 的 ModalResult 属性设置成“ mrCancel ”。把 LbNote 的 Caption 属性设置为“ 在这个作为对话框使用的窗体中，使用了两个设置了 ModalResult 属性的按钮 ”。然后修改两个按钮的 Caption 属性。

(5) 返回到代码编辑器中的 EmpButton.pas，然后在 implementation 的后面输入下面的代码，加入对第二个窗体的引用：

```
uses EmpButtonM;
```

然后在 BtnShowForm **的** OnClick **事件中输入如下的代码：**

```
procedure TFrmButton.BtnShowFormClick(Sender: TObject);
begin
EdResult.text:=inttostr(FrmDialog.ShowModal);
end;
```

(6) **在重新保存了所有的文件之后运行应用程序，然后单击“显示对话框”按钮，将会显示作为对话框的第二个窗体，如图** 5.27 **所示。单击该窗体上的任何一个按钮，便会关闭该对话框，并在主窗体的文本框中显示一个数字。**

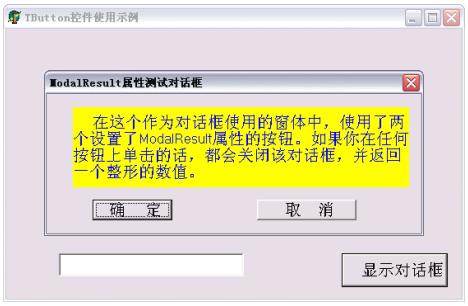

图 5.27 TButton **控件使用示例**

5.3.2 位图按钮

在许多情况下，使用 TButton **控件无疑是十分方便的，而且它也可以完成我们所需要的功能。但是，如果你是一个对程序的界面要求比较苛刻的人，希望建立更为丰富多彩的图形化界面，那么在这一小节中介绍的位图按钮以及下一小节中要介绍的快捷按钮，可能是更好的选择。**

位图按钮（TBitBtn**），顾名思义，就是可以在按钮上使用位图的按钮。显然，如果我们的按钮不仅能够显示标题内容，而且可以显示代表一定含义的位图的话，那么我们的应用程序的界面将变得十分友好了。**

下面我们通过一个应用程序的例子来说明位图按钮控件的典型用法。

(1) **首先新建一个应用程序，然后把** Form1 **的字体改成** 12 **号字。并把该** Form **的名称修改为“** FrmBitBtn **”。**

(2) **单击控件选项板上的** Additional **选项卡，在窗体上连续放置** 4 **个** TBitBtn**，并把它们的名称一次修改为“** BitBtnDown **”，“** BitBtnRight **”，“** BitBtnUp **”，“** BitBtnLeft **”。**

(3) **单击窗体上的** BitBtnDown **位图按钮，然后在属性编辑器上找到它的** Glyph **属性，然后单击该属性旁边的省略号按钮，此时会出现如图** 5.28 **所示的** Picture Editor **对话框。**

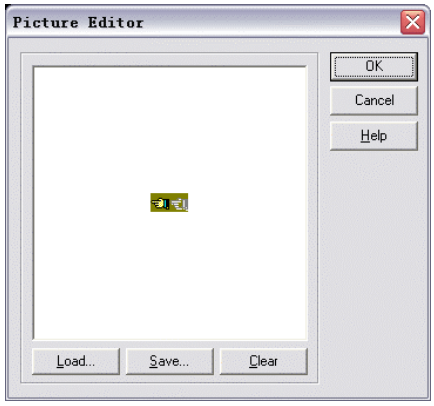

图 5.28　Picture Editor 对话框

(4) 单击该对话框上的 Load 按钮，在出现的对话框中选择一个图片文件时，会在对话框的右边框内出现这个图片的预览。

(5) 为四个位图按钮依次载入如图 5.29 所示的四个图形。

图 5.29　四个位图按钮的 Glyph 属性中载入的图片

注意：

在 Delphi 中，使用位图按钮时，用于按钮的图片可以分成两部分，一部分是按钮正常状态时的图片，另一部分则是按钮在 Enabled 属性设置成“ False ”时使用的图片。例如，在上面的图片中，每个图片又分成两部分，左边的部分是按钮正常状态下的形状，而右边的图片则是按钮在禁用的情况下的形状。

(6) 然后修改位图按钮的 Caption 属性。这四个位图按钮的 Caption 属性依次为“ 向下 ”、“ 向右 ”、“ 向上 ”和“ 向左 ”。

(7) TBitBtn 控件还有一个属性 Layout，它可以控制位图按钮上的位图和文字的相对位置。观察该属性的预设值，可以发现它有四个选择，如表 5.9 所示。把该应用程序中的这四个按钮都设置成“ blGlyphTop ”。

表 5.9　Glyph 属性的值

属性值	图的位置
blGlyphLeft	图像出现在标题的左边
blGlyphRight	图像出现在标题的右边
blGlyphTop	图像出现在标题的上方
blGlyphBottom	图像出现在标题的下方

(8) 在窗体上放置一个 TPanel 控件，然后把它的 Align 属性设置成‘ clBottom ”，BevelInner

属性设置成“ bvLowered ”，BevelOuter 属性设置成“ bvRaised ”，BorderWidth 设置成 3，Capiton 属性设置成空。

(9) 在 Panel1 控件上放置另外一个 TPanel 控件 Panel2，然后把它的 Align 属性设置成“ clClient ”，BevelInner 属性设置成“ bvNone ”，BevelOuter 属性设置成“ bvNone ”，Capiton 属性设置成空，Color 属性设置成“ clBlack ”。

(10) 然后在 Panel2 上放置一个 TShape 控件，把它的 Shape 属性设置成“ stCircle ”，把它的 Brush 属性的 Color 属性设置成“ clRed ”。

(11) 然后在四个位图按钮的 OnClick 事件中输入下面的代码：

```
procedure TFrmBitBtn.BitBtnDownClick(Sender: TObject);
begin
if shape1.Top + Shape1.Height+5 < Panel2.ClientHeight then
   shape1.Top:=Shape1.Top +5;
end;
procedure TFrmBitBtn.BitBtnRightClick(Sender: TObject);
begin
if shape1.Left + Shape1.Width+5 < Panel2.ClientWidth then
   shape1.Left:=Shape1.Left +5;
end;
procedure TFrmBitBtn.BitBtnUpClick(Sender: TObject);
begin
if shape1.Top >5 then
   shape1.Top:=Shape1.Top -5;
end;
procedure TFrmBitBtn.BitBtnLeftClick(Sender: TObject);
begin
if shape1.Left >5 then
   shape1.Left:=Shape1.Left -5;
end;
```

(12) 运行应用程序，结果如图 5.30 所示。单击位图按钮，红色的圆将作相应的移动。

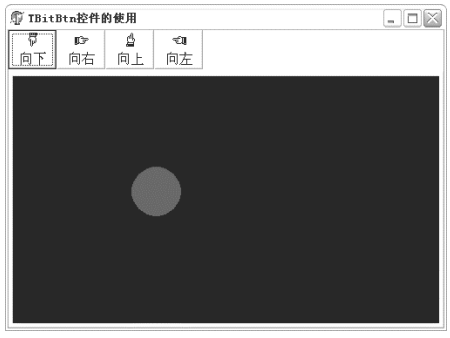

图 5.30　TBitBtn 控件的典型用法

(13) 在上面的应用程序中，如果禁用其中的某个按钮，程序将如图 5.31 所示。

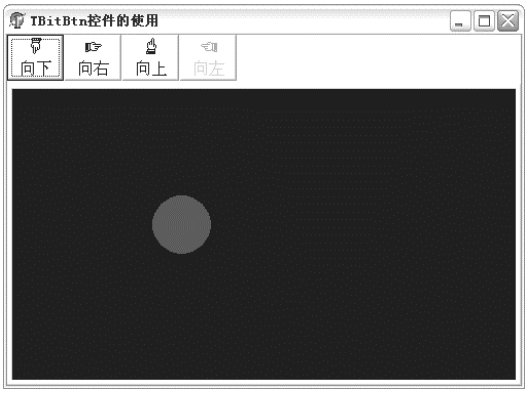

图 5.31　禁用的按钮将使用 Glyph 属性中图片的第二部分

在一般情况下，不需要设置控件上的文字和位图的位置。但是，如果需要，可以通过两个属性来完成：Margin 和 Spacing。前一个属性用来设置图像与按钮边界之间的距离（以像素为单位），默认值是–1，表示图像和文字总是居中，设置 0 则表示紧挨着按钮的边界。后一个属性用于设置按钮上的图像和文字之间的距离，默认值是 4。如果把这个属性设置成–1，则文字将显示在图像和按钮边界的中间。

在使用位图按钮时，许多情况下需要我们自己准备按钮上出现的图片。但是，如果要设置的按钮是一些标准的按钮，例如 Close、Ignore 等按钮，Delphi 则已经在位图按钮中预先定义了许多按钮的图像和文字，这是通过 TBitBtn 控件的 Kind 属性来完成的。Kind 属性有 11 个值，其中 bkCustom 属性是根据用户自己的图片来进行定义的。而其他的 10 个值都是位图按钮的预设按钮，如表 5.10 所示。

表 5.10　Kind 属性的预设值和对应的按钮形状

Kind 属性的值	按钮的形状	说明
bkOK	OK	自动成为默认按钮，并把 ModalResult 属性设置成 mrOK
bkCancel	Cancel	自动成为取消按钮，并把 ModalResult 属性设置成 mrCancel
bkYes	Yes	自动成为默认按钮，并把 ModalResult 属性设置成 mrYes
bkNo	No	自动成为取消按钮，并把 ModalResult 属性设置成 mrNo
bkHelp	Help	单击此按钮将显示帮助，帮助文件由 TApplication 对象的 HelpFile 属性指定，上下文编号由 HelpContext 属性指定
bkClose	Close	自动成为默认的关闭按钮，并把 ModalResult 属性设置成 mrClose

续表

Kind 属性的值	按钮的形状	说明
bkAbort		自动成为默认的中断按钮，并把 ModalResult 属性设置成 mrAbort
bkRetry		自动成为默认的重试按钮，并把 ModalResult 属性设置成 mrRetry
bkIgnore		自动成为默认的忽略按钮，并把 ModalResult 属性设置成 mrIgnore
bkAll		自动成为默认的全选按钮，并把 ModalResult 属性设置成 mrAll

5.3.3 快捷按钮

快捷按钮（TSpeedButton）和位图按钮十分类似，按钮上也可以显示图像，在使用方法上也十分类似。可以说，完全可以使用 TSpeedButton 控件来替代 TBitBtn 控件。但是 TSpeedButton 控件和位图控件还是有所区别的。总的来说，快捷按钮和位图按钮的区别如下。

- 快捷按钮的默认尺寸总是 25×25，当然可以改变尺寸。
- 快捷按钮一般只显示图像，不显示文字，因为快捷按钮的尺寸太小。
- 快捷按钮可以保持在按下的位置，而命令按钮和位图按钮则不能保持在按下的位置。

根据这个特点，快捷按钮可以模拟一组单选按钮，被选择的按钮可以保持在按下位置，未被选择的按钮在弹起位置。

我们已经提到，快捷按钮的使用和位图按钮的使用非常相似，包括 Glyph 和 Layout 属性。但是快捷按钮可以成组使用，也就是说可以把几个快捷按钮作为一组进行操作，这经常适用于只能从多个选项中选择一个的情况。那么如何把快捷按钮成组使用呢？这要通过快捷按钮的 GroupIndex 属性。如果该属性值为 0，那么每个快捷按钮就是一个独立的按钮；如果该属性值不为零，那么属性值相同的快捷按钮为一组。也就是说，我们可以在一个容器（这个容器可能是窗体或者其他控件）放置多个快捷按钮，并把它们分成多个组。

例如在图 5.32 所示的应用程序中，就把窗体上的快捷按钮分成了两组。

TSpeedButton 控件的属性中，绝大部分都和 TBitBtn 控件相同。除了上面我们介绍的比较重要的 GroupIndex 属性之外，还有一些比较重要的属性，我们将在下面的内容中介绍它们的使用方法。

一般来说，几个快捷按钮模拟单选按钮的时候，其中总是有一个按钮在按下状态，但是也可以是所有的按钮都在弹起状态，如果把 AllowAllUp 属性设置为“True/”，那么表示允许所有的按钮都在弹起状态；反之，如果把该属性设置成“False”，那么必须有一个按钮处于按下状态。

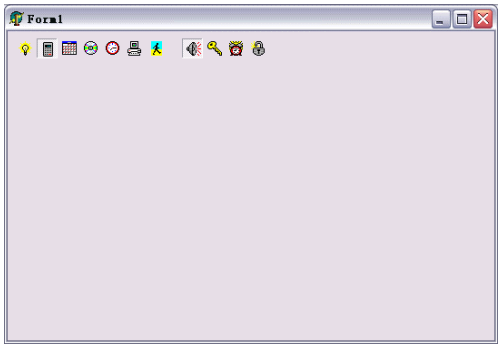

图 5.32　TSpeedButton 控件的典型用法

需要读者注意的是，改变一个快捷按钮的 AllowAllUp 属性，将同时改变它所在的组的所有快捷按钮的属性。

如果一个组只有一个快捷按钮组成，可以把这个快捷按钮的 AllowAllUp 属性设置成 True，这样快捷按钮就可以在按下和弹起的状态间切换，相当于一个复选框。

还有一个属性称为 Down 属性。这个属性要和 AllowAllUp 属性配合使用。在设置为一组的快捷按钮中，可能在有的时候个别的快捷按钮在默认的情况下要处在选中状态，那么可以把 Down 属性设置成“ True ”。当然这个属性还可以有其他的用法，比如，对于独立的快捷按钮，我们也可以在运行期设置这个 Down 属性，改变按钮的弹起或者按下状态。

和位图按钮不同的是，快捷按钮还有一个 Flat 属性。如果把这个属性设置成“ True ”，那么按钮在未被选中的情况下将没有边框，在选择状态下才会有边框。

5.4　选项按钮和复选框

选项按钮和复选框是在应用程序中经常使用的控件，它们的主要目的是让用户选择一些选项，这些选项可能与应用程序的进行方向有关，也可能与应用程序的进行条件有关。

5.4.1　选项按钮

选项按钮，许多人又称之为单选按钮，主要用于多选一的选项设置。也就是说，选项按钮所表示的选项是互斥的，并且通常成组使用，一组选项中只能有一个选项被选择。如果选择了其中的一个选项，同一组中的其他选项将自动处于非选择状态。

这个控件只有两个属性是比较特殊的，一是 Alignment 属性，这个属性用来设置选项按钮的文字和小圆圈的位置关系，在默认的情况下是文本显示在小圆圈的左边。但是也可以使得文本显示在小圆圈的右边。另一个属性是 Checked 属性，这个属性用来表示当前的选项按钮是否被选中。我们既可以利用这个属性来读取选项按钮的状态，也可以利用这个属性来设

置选项按钮的状态。

这里我们需要强调的是选项按钮的分组情况。选项按钮和我们前面介绍的快捷按钮不同，它不能在同一个容器上分成许多组。在一个容器上所有的选项按钮都作为一组来使用。所以，对处于同一个容器上的选项按钮从意义上来说必须是互斥的，我们无法使得同一个容器上的选项按钮成为不同的组。

5.4.2 复选框

实际上复选框的使用和选项按钮的使用也十分类似，所不同的是复选框适用于多选多的情况。也就是说，我们可以从一组复选框中选择多个选项，而不必像选项按钮那样只能选择一个。

实际上这个控件的使用方法和选项按钮的使用方法十分类似，所不同的只是在使用这个控件的 OnClick 事件时要加上对复选框的状态判断。对于选项按钮来说，不需要判断这个控件的状态，因为如果发生单击这个控件的事件，这个控件的状态必定是选中状态。但是对于复选框来说，单击它会使它的状态发生转换，例如如果当前的状态是选中状态，那么单击该控件会使控件处于非选中状态。所以，对于需要根据复选框的状态来进行的应用程序来说，需要对其状态进行判断。

当然它也有一些属性和选项按钮是不同的，例如 AllowGrayed 属性，如果把这个属性设置成“ True ”，那么复选框就可以处于选中但便变灰的状态。还有一个属性是 State 属性。这个属性可以返回或者设置复选框的状态。

- Unchecked：表示复选框处于未选中状态。
- cbChecked：表示复选框处于选中状态。
- cbGrayed： 复选框处于选中但变灰的状态。

5.5 各类列表框的使用

在 Windows 应用程序中，列表框的使用是非常普通的。列表框的形式是多种多样的，我们会根据需要出现在列表框中的条目多少，以及界面上的空间的大小来确定需要的列表框形式。在 Delphi 中，提供了五种列表框，它们分别是 ListBox，ComboBox、CheckListBox、ColorBox 和 ComboBoxEx。

5.5.1 列表框控件

列表框的使用是比较复杂的，它具有多个各类形式的列表框控件所共有的属性和方法，所以对它进行比较详细的介绍是非常有必要的，这样可以做到举一反三，使我们在掌握后面

要介绍的其他类型列表框时容易许多。

下面首先介绍一些属性和方法的含义和用法，然后再通过一个实际的应用程序来演示如何在应用程序中使用这些属性和方法来操作列表框。

首先要介绍的属性是 MultiSelect 属性。因为在实际的应用程序中，经常需要使用到进行多选的情况。在列表框控件中就是用 MultiSelect 属性来控制进行单选还是多选的。如果把这个属性设置成“ True ”，表示允许同时选择多项。例如可以通过下面的语句来设置列表框控件的 MultiSelect 属性。

```
LstBox.MultiSelect := ChkMulti.Checked ;
```

列表框控件对如何进行多选也提供了一个属性——ExtendedSelect 属性。这个属性用于设置是否允许使用 Shift 键和 Ctrl 键同时选择多个选项，前提是 MultiSelect 属性必须设置成“ True ”。如果该属性设置成了“ True ”，表示如果用户按下并保持 Shift 键可以选择多个连续的项，按下并保持 Ctrl 键可以选择多个选项（这些选项不一定连续）。

那么在多选的情况下，我们怎样知道在应用程序中选择了多少项呢？这可以使用列表框控件的 SelCount 属性，这是一个只读属性，也就是说，我们不能对它进行赋值。利用这个属性，我们可以确切地知道用户选择了多少条目。显然，提供这个属性对操作列表框控件来说是非常重要的。

上面介绍的内容都是和列表框中的条目多选有关的，还有一个内容是关于列表框中的条目添减的。实际上，列表框控件的关于条目的添加和删减方法只有一个，就是 Clear 方法。利用这个方法可以清空列表框中的所有内容。但是可以通过 TStrings 对象的一些方法来实现列表框中的条目添减。所以下面介绍的内容主要是 TStrings 对象的内容，这部分内容适用于任何包含 TStrings 属性的控件。

如果要为列表框添加一个项目，可以使用两个方法，一个是 Add 方法，一个是 Insert 方法。例如下面的语句将在列表框的末尾添加一个新的项目。

```
LstBox.Items.Add(EdItem.Text);
```

Add 方法会在列表框的最后一条后面加上一个条目。Add 是一个函数，它返回的是列表框中新添加的条目索引号。TStrings 对象还有一个 Append 方法，这个方法和 Add 方法一样的，除了它是一个过程之外；也就是说，它不能返回数值。

如果希望在某个位置插入一个条目，可以使用 Insert 方法。例如可以使用下面的语句在列表框控件中加入一个条目。

```
LstBox.Items.Insert ( LstBox.ItemIndex, EdItem.text);
```

如果希望删除某个位置的条目，可以使用 Delete 方法，这个方法将删除指定索引号处的条目。例如在下面的语句中，将删除用户在文本框中指定的条目。

```
LstBox.Items.Delete (strtoint(EdBIndex.Text));
```

如果希望把某个位置的条目移动到另外一个指定的位置，可以使用 Move 方法。例如在下面的语句中，将会把指定的条目从一个位置移动到另外一个指定位置。

```
LstBox.Items.Move (strtoint(EdBIndex.text),strtoint(EdEindex.Text));
```

例如在下面的应用程序中，我们演示了如何处理列表框的条目和一些选项。关于如何建立这个应用程序的步骤在这里就不再详细地介绍了，程序代码如下所示。

```
procedure TFrmList.BtnAddClick( Sender: TObject );
begin
if EdItem.Text <> '' then
    LstBox.Items.Add( EdItem.Text );
end;
procedure TFrmList.BtnInsertClick(Sender: TObject);
begin
    if EdItem.Text <> '' then
        LstBox.Items.Insert(LstBox.ItemIndex,EdItem.text);
end;
procedure TFrmList.BtnFindClick(Sender: TObject);
var i:integer;
begin
  for i:=0 to LstBox.Items.Count-1 do
      if comparestr(EdItem.Text,LstBox.Items[i]) = 0 then
      begin
          LstBox.ItemIndex :=i;
          LstBoxClick(Self);
          Break;
      end;
end;
procedure TFrmList.LstBoxClick(Sender: TObject);
var i:integer;
     s:string;
begin
  s:='';
  EdSelCount.Text := inttostr(LstBox.SelCount);
  for i:=0 to LstBox.Items.Count-1 do
  begin
      if LstBox.Selected [i] = True then
          s := s+'1'
      else
          s := s+'0';
  end;
  EdSelected.Text := s;
```

```
    EdHeight.Text := inttostr (LstBox.ItemHeight);
end;
procedure TFrmList.BtnSelectClick(Sender: TObject);
begin
if EdBIndex.Text <> '' then
begin
    LstBox.ItemIndex := strtoint(EdBIndex.Text);
    LstBoxClick(Self);
end;
end;
procedure TFrmList.BtnDeleteClick(Sender: TObject);
begin
if EDBIndex.Text <> '' then
    LstBox.Items.Delete (strtoint(EdBIndex.Text));
end;
procedure TFrmList.BtnMoveClick(Sender: TObject);
begin
if (EdBIndex.Text <>'') and (EdEIndex.Text <> '') then
    LstBox.Items.Move (strtoint(EdBIndex.text),strtoint(EdEindex.Text));
end;
procedure TFrmList.ChkMultiClick(Sender: TObject);
begin
LstBox.MultiSelect := ChkMulti.Checked ;
end;
procedure TFrmList.ChkSortedClick(Sender: TObject);
begin
LstBox.Sorted := ChkSorted.Checked ;
end;
procedure TFrmList.ChkExtendClick(Sender: TObject);
begin
LstBox.ExtendedSelect := ChkExtend.Checked ;
end;
procedure TFrmList.ChkIntHClick(Sender: TObject);
begin
if ChkIntH.Checked = True then
begin
    LstBox.Style := lbOwnerDrawFixed;
    LstBox.IntegralHeight := True;
end
else
    LstBox.Style := lbStandard;
end;
```

程序的运行结果如图 5.33 所示。

图 5.33 处理列表框的条目

从上面的程序运行结果以及程序代码中可以看出，根据用户选择的条目多少，Selected 文本框中的数字将作相应的变化。读者可以参考这里的程序对如何处理多选的情况有一定的了解。

到此为止，我们似乎已经完成了关于处理列表框控件的全部内容，但是，观察一下在本节开始时提供的对话框中的列表框就可以发现，我们还没有介绍如何创建在类似于这个示例中展示的带有图形的列表框问题。下面就将介绍这方面的内容。

在创建这样的列表框时，需要使用到 TListBox 控件的 Canvas 属性，这个属性返回列表框的画布。画布，顾名思义，就是用来在控件或者其他对象上进行图形操作的底版，实际上它是不可见的，许多对象或者控件都具有这个属性，它是一个对象属性，在后面的处理图形的内容中，我们将详细地介绍关于画布、画笔和画刷的属性和方法。

在下面的程序中，需要使用 Canvas 对象的 BrushCopy 方法，这是一个过程，声明如下。

```
procedure BrushCopy(const Dest: TRect; Bitmap: TBitmap; const Source: TRect; Color: TColor);
```

这里有两个 TRect 参数，代表了两个矩形的区域，一个是源区域，一个是目标区域。Bitmap 参数代表了要处理的位图对象，其中的 Color 是一个颜色参数，它的作用是指定位图中需要用画布中的画刷的颜色来替代的颜色。

当我们要创建具有图片的列表框时，需要使用列表框控件的 OnDrawItem 事件，这个事件发生在 Style 属性设置为 lbOwnerDrawFixed 或者 lbOwnerDrawVariable 的情况下，并且是在列表框需要画出条目的时候。

例如在下面的应用程序中，先在一个 TImageList 控件中载入了许多图片，然后利用下面的程序，把该控件中的各个图片添加到列表框中，代码如下所示。

```
procedure TFrmList.LstBoxDrawItem (Control: TWinControl; Index: Integer;
                                   Rect: TRect; State: TOwnerDrawState);
var BitMap:TBitmap;
    Offset:integer;
begin
     With TListBox(Control).Canvas Do
     Begin
          FillRect(Rect);
          Offset:=2;
          Bitmap := TBitMap.Create;
          ImgLst.GetBitmap(Index,BitMap);
          if Bitmap <> nil then
          begin
          BrushCopy(Bounds(Rect.left+2,Rect.Top,BitMap.Width,Bitmap. Height),
               bitmap,Bounds(0,0,Bitmap.Width,Bitmap.height),clred); Offset:=Bitmap.width+6;
          End;
          TextOUt(Rect.Left + Offset, Rect.top,TListBox (Control). Items [index]);
     end;
end;
```

在这个程序中使用了列表框控件的 OnDrawItem 事件，这个事件对应的方法中的各个参数的含义如下。

- **Control 参数在这里代表的就是列表框控件。**
- **Index 参数是需要画出的条目索引号。**
- **Rect 参数是条目在列表框的画布中的位置，是用一个矩形的区域表示的，矩形的高度由 ItemHeight 属性或者由处理 OnMessureItem 事件的句柄指定的。**
- **State 参数表示条目的状态。**

上面的程序的运行结果如图 5.34 所示。

图 5.34　创建包含图片的列表框

虽然上面的图片是随意选择的，可能所代表的意义和旁边的文字文不对题，但是，毕竟演示了如何创建这样的列表框控件，在实际的程序设计过程中，完全可以根据需要对要显示在列表框中的图形进行仔细的筛选。

5.5.2 组合框控件

组合框控件也是在 Windows 编程中经常会使用到的一个控件，可以把它看作是折叠起来的列表框控件。与列表框控件相比，组合框可以节省屏幕上的空间，因为组合框在同一时刻可以只显示其中的一项，通常是用户选择的一项。除了让用户从下拉列表中选择之外，组合框还可以让用户直接输入文字。

可以说，组合框是列表框控件的升级，它几乎包含了列表框控件的所有属性和方法，而且这些名称相同的属性和方法的用法和列表框控件也完全一样。当然，也可以利用上面介绍的方法来建立包含图片的组合框，建立方法和列表框是类似的，这里就不再详细介绍了。但是，列表框控件中的关于多选的属性和方法，组合框却没有继承下来。

下面首先通过一个应用程序来看看如何在程序中使用组合框。读者观察一下可以发现这个程序就是上面一小节中介绍列表框控件时介绍的应用程序的翻版。可以看到，在使用组合框的各个条目时的效果几乎和列表框一样。

从图 5-35 中可以看出，组合框所占用的屏幕空间比起列表框控件来就小多了。这正是组合框控件的一个突出的优点。

另外，组合框还有一些列表框控件所没有的属性，下面就对这些属性进行简要的介绍。

从图 5-35 中可以看出，当我们单击组合框旁边的箭头时，会出现下拉列表，那么这个列表的容量是多少呢？也就是说，在默认的情况下，这个列表一次显示多少条目呢？这是由组合框控件的 DropDownCount 属性来确定的。这个属性决定了在不添加滚动条的情况下可以显示的条目个数，默认值为 8，表示用户下拉组合框的时候，如果条目的个数超过了 8 个就会添加上滚动条。如果实际的条目数没有 DropDownCount 属性指定的值多，下拉的组合框会自动缩小。所以我们可以在程序中指定这个数目来改变组合框的显示情况，例如可以使用下面的语句。

```
CmbItem.DropDownCount := strtoint(edNumber.Text);
```

在图 5.35 中显示了不同的 DropDownCount 属性值的情况下组合框的表现情况。

组合框区别于列表框的一个特点是组合框可以输入内容，当然这只是在 Style 属性设置为“csDropDown”和“csSimple”的情况下。在组合框的编辑框中输入文本时，组合框在很多方面和我们前面介绍的文本框非常类似。例如也有一个 MaxLength 属性，如果这个属性为零，那么可以输入的字符串的长度不受限制；如果是一个非零的值，那么用户可以输入的字符串长度就被这个属性规定好了。在上面的程序中，我们也编写了可以设置这个属性的程序，只要用户在“可输入的文本长度”框中输入一个整数，那么这个文本框的 OnChange 事件就

会把这个值传递给组合框的 MaxLength 属性。程序如下所示。

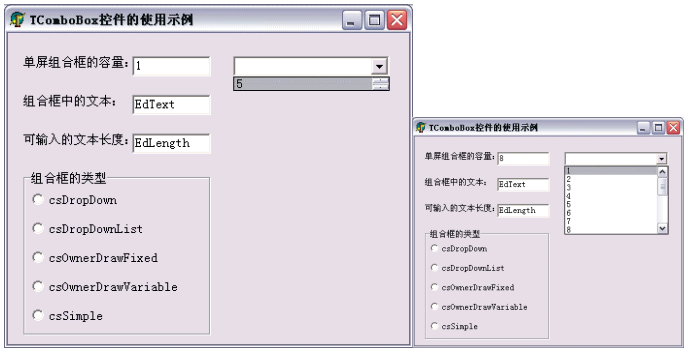

图 5.35　DropDownCount 属性和组合框的下拉列表

```
procedure TFrmComboBox.EdLengthChange(Sender: TObject);
begin
CmbItem.MaxLength := strtoint(edLength.text);
end;
```

同样，组合框也有一个 Style 属性，这个属性可以设置组合框的风格类型，它可以取 5 个预设的值。这些值的名称和含义如表 5.11 所示。

表 5.11　组合框的 Style 属性的预设值

属性值	说明
csDropDown	标准的组合框，由编辑框和下拉列表组成，列表中的每一项是字符串，并且高度一致
csDropDownList	编辑框不能进行编辑，除此之外和 csDropDown 预设值完全相同
csOwnerDrawFixed	类似于列表框控件的 lbOwnerDrawFixed
csOwnerDrawVariable	类似于列表框控件的 lbOwnerDrawVariable
csSimple	编辑框和组合框的下拉列表同时显示出来，用户可以编辑编辑框中的文字

在下面的程序中，我们通过一个选项按钮组控件来控制一个组合框的类型，代码如下所示。

```
procedure TFrmComboBox.RgStyleClick(Sender: TObject);
begin
case RgStyle.ItemIndex of
    0: CmbItem.Style := csDropDown;
    1: CmbItem.Style := csDropDownList;
    2: CmbItem.Style := csOwnerDrawFixed;
    3: CmbItem.Style := csOwnerDrawVariable;
    4: CmbItem.Style := csSimple;
end;
```

```
CmbItem.Height := 250;
end;
```

这段程序根据我们选择的组合框的类型来显示组合框，其中 csSimple 值情况下的组合框如图 5.36 所示。

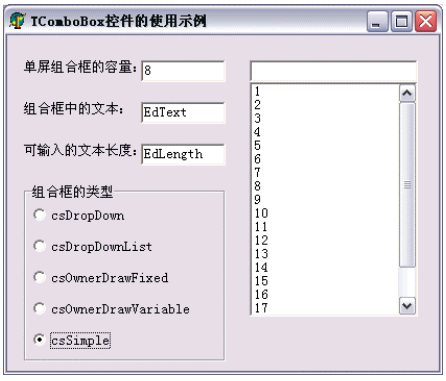

图 5.36　组合框示例

5.5.3　复选列表框控件

复选列表框控件可以说是复选框控件和列表框控件的组合，是条目为复选框的列表框控件。这类控件在许多 Windows 应用程序的安装程序中经常使用。

复选列表框控件首先是一个列表框，所以它具有普通的列表框所具有的属性，例如 Items 属性、Style 属性，等等，这些属性和 TListBox 控件是一样的。同时它又是包含复选框的，所以它还具有复选框的一些属性，例如 Checked 属性、State 属性和 ItemEnabled 属性，只不过这些属性都是数组。下面我们通过一个应用程序来演示如何根据这些属性进行操作。

首先建立一个新的应用程序，然后在上面放置一个 TCheckListBox 控件，并为这个控件输入一些条目。下面我们就来处理它的几个属性。由于这些属性都是数组，而且 Checked 属性和 ItemEnabled 属性都是 Boolean 型的，而 State 是 TCheckBoxState 型的，所以无法直接观察它们的情况。下面就对这些属性进行一些处理。

对于 Checked 属性和 ItemEnabled 属性来说，可以用一个字符串来表示它的值，例如，我们用“T”来表示 True，用“F”来表示 False。那么把这些字符串按照 Checked 属性和 ItemEnabled 属性的值的顺序进行组合，便可以生成一个表示该数组属性的字符串。代码如下所示。

```
Function GetArrNum(P:array of boolean):string;
Function TFrmControl.GetArrNum (p:array of Boolean):string;
var i:integer;
    s:string;
```

```
begin
    s:='';
    for i:=Low(p) to High(p) do
    begin
        case P[i] of
            True: s:=s+'T';
            False: s:=s+'F';
        end;
    end;
    Result:=s;
end;
```

然后我们来分析一下 State 属性。在 Delphi 中，该属性是这样定义的：

```
type TCheckBoxState = (cbUnchecked, cbChecked, cbGrayed);
property State[Index: Integer]: TCheckBoxState;
```

它代表了复选框的三种状态：未选中、选中和变灰。这个数组属性的处理自然不同于上面 Boolean 类型的数组属性，但是处理思路是一样的。我们可以在程序中用字符“U”来代表 cbUnchecked，用字符“C”来代表 cbChecked，用字符“G”来代表 cbGrayed，那么就可以用下面的代码来获取 State 属性的状态。

```
Function GetChkNum(p:array of TCheckBoxState):string;
Function TFrmControl.GetChkNum (p:array of TCheckBoxState):string;
var i:integer;
    s:string;
begin
    s:='';
    for i:= 0 to High(p) -Low(p) do
    begin
        Case p[i] of
            cbUnchecked: s:=s+'U';
            cbChecked:    s:=s+'C';
            cbGrayed:     s:=s+'G';
        end;
    end;
    Result:=s;
end;
```

然后在复选列表框控件的 OnClick 事件中输入下面的代码：

```
procedure TFrmControl.ChkLstClick(Sender: TObject);
var p:array of boolean;
    s:array of TCheckBoxState;
```

```
    i:integer;
begin
    setlength(p,chklst.items.count);
    for i:=0 to chklst.items.count-1 do
      p[i]:=chklst.checked[i];
    EdCheck.Text := GetArrNum( p );
    for i:=0 to chklst.items.count-1 do
      p[i]:=chklst.ItemEnabled[i];
    EdItemEnabled.Text := GetArrNum( p );
    setlength(s,chklst.items.count);
    for i:=0 to chklst.items.count-1 do
      s[i]:=chklst.state[i];
    edstate.Text := GetchkNum( s );
end;
```

在这段程序中，EdCheck、EdItemEnabled **和** EdState **是三个** TEdit **控件，它们用来显示我们所获取的上面介绍的三个属性的状态。**

运行这个应用程序，结果如图 5.37 **所示。**

从图中可以清楚地看出，变灰和选中是两种状态。当然由于在这里并没有改变复选列表框中的各个条目的 Enable **属性，所以** ItemEnabled **属性都是** True**。**

下面我们再对这个程序进行改进，使它能够对复选列表框控件的这些属性进行设置。首先从文本框中取得一些字符串，然后按照和上面变换相反的规则把这些字符串进行反变换，并对复选列表框进行设置。请参见下面的代码。

图 5.37　**复选列表框控件的三个数组属性的状态**

```
procedure TFrmControl.GetChecked (p:string);
var i:integer;
    L:integer;
begin
    L:=Length(p);
    if L > chklst.Items.Count then
       L:= ChkLst.Items.Count;
```

```
    p:=UpperCase(p);
    for i:=1 to L do
    begin
        case p[i] of
            'T': ChkLst.Checked [i-1]:=True;
            'F': ChkLst.Checked [i-1]:=False;
        end;
    end;
end;
procedure TFrmControl.GetItemE (p:string);
var i:integer;
    L:integer;
begin
    L:=Length(p);
    p:=Uppercase(p);
    if L > chklst.Items.Count then
      L:= ChkLst.Items.Count;

    for i:=1 to L do
    begin
        case p[i] of
            'T': ChkLst.ItemEnabled [i-1]:=True;
            'F': ChkLst.ItemEnabled [i-1]:=False;
        end;
    end;
end;
procedure TFrmControl.GetState (p:string);
var i:integer;
    L:integer;
begin
    L:=Length(p);
    if L > chklst.Items.Count then
       L:= ChkLst.Items.Count;
    p:=Uppercase(p);
    for i:=1 to L do
    begin
        case p[i] of
            'U': ChkLst.state [i-1]:=cbUnchecked;
            'C': chklst.state [i-1]:=cbChecked;
            'G': ChkLst.State [i-1]:= cbGrayed;
        end;
    end;
```

```
end;
```

然后在三个文本框的 OnChange 事件中的加入下面的代码。

```
procedure TFrmControl.EdCheckChange(Sender: TObject);
begin
    GetChecked(EdCheck.Text);
end;
procedure TFrmControl.EdItemEnabledChange(Sender: TObject);
begin
    GetItemE(EdItemEnabled.Text);
end;
procedure TFrmControl.EdStateChange(Sender: TObject);
begin
    GetState(EdState.Text);
end;
```

运行该应用程序，然后在第二个文本框中输入“TFFTTFF”，在第一个文本框中输入“TFFTTFF”，在第三个文本框中输入“GUCUGCC”。此时的程序运行结果如图 5.38 所示。

图 5.38　设置复选对话框控件的属性

5.5.4　ColorBox 控件

ColorBox 控件是 Delphi 6.0 中新增加的一个控件，可以让读者很容易地选择需要的颜色。在上面的例子中，我们已经使用到了 Delphi 中的 ColorDialog 控件，它也是用户选取颜色的一个很好的工具。这里介绍的 ColorBox 控件比 ColorDialog 更为简洁。

ColorBox 控件和 Delphi 6.0 中的颜色属性的赋值方式非常相似。在图 5.39 所示的窗口中，我们使用了一个 ColorBox 控件。

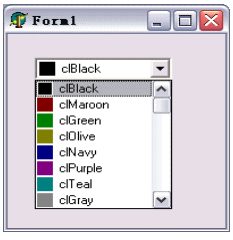

图 5.39　默认情况下的 ColorBox 控件

在这个控件中，显示了和 Delphi 中的颜色属性赋值时一样的一些预先定义好的颜色值。所以从某种意义上说，该控件实际上

是 ComboBox 的一种特殊的应用形式。如果我们希望使用 Windows 的标准颜色选取对话框来自定义颜色的话，可以使用该控件的 Style 属性。该属性有 6 个 Boolean 类型的分量，通过它们，可以定义 ColorBox 控件的各种外观和行为方式。

- cbStandardColors：此时的 ColorBox 控件中只显示最基本的 16 种颜色，它们都是常量。
- cbExtendedColors：此时的 ColorBox 控件中将包含一些扩展的颜色值，例如 clMoneyGreen、clSkyBlue、clCream 和 clMedGray。
- cbSystemColors：此时的 ColorBox 控件中将包含 Windows 操作系统控制面板中规定的系统颜色，例如窗口的颜色，等等。
- cbIncludeNone：此时的 ColorBox 控件中将包含 clNone 值，该选项只在包含 cbSystemColors 的情况下有效。
- cbIncludeDefault：此时的 ColorBox 控件中将包含 clDefault 值，该选项只在包含 cbSystemColors 的情况下有效。
- cbCustomColor：此时的 ColorBox 控件的第一个选项是 Custom，也就是说当用户选择第一个项目的时候，程序会显示 Windows 的标准颜色选择对话框，用户可以在其中选择任意的颜色。
- cbPrettyNames：此时的 ColorBox 控件中显示的不是上面介绍的 clRed 等颜色值，而是颜色的名称，比如对于 clRed 来说就是 Red。

利用该控件选择的颜色存储在 ColorBox 控件的 Selected 属性中，例如，可以用下面的语句来访问该颜色。

```
Shape1.Brush.Color:=ColorBox1.Selected;
```

如果要访问 ColorBox 中的所有可能的颜色值，可以使用 ColorBox 控件的 Colors 属性。

5.5.5　ComboBoxEx 控件

同 ColorBox 控件一样，ComboBoxEx 控件也是 Delphi 6.0 中新增加的控件。利用这个控件，我们可以实现更复杂和更个性化的列表框。比如，如果列表框中的项目希望能够呈现一定的层次，也就是说包含一些子项目，那么此时可以选择这个控件。

这个控件和我们前面介绍的 ComboBox 控件十分类似。可以用前面介绍的 ComboBox 控件的属性和方法来处理该控件。和 ComboBox 控件相比，我们需要关注的是以下两个属性。

- Images：这个属性是一个 TImageList 类型的属性，用来指定存储了列表框中的项目的图片。在很多控件中都会包含这个类型的属性。此时一般需要在窗体上添加一个 TImageList 控件，该控件位于 Delphi 的控件面板上的 Win32 选项卡上。
- ItemsEx：这个属性在作用上和 ComboBox 控件的 Items 是相同的，只是定义的方式不同。在 ComboBoxEx 控件中，所包含的每个项目也是一个对象，具有自己的属性

和方法，而不像 ComboBox 中的项目一样通常是单纯的字符串。

在下面的过程中，我们通过一个简单的例子来介绍如何使用 ComboBoxEx 控件。

(1) 新建一个应用程序。然后在窗体上放置一个 ComboBoxEx 控件和一个 TimageList 控件。把 ComboBoxEx 控件的 Images 属性设置成这里加入的 TimageList 控件（默认情况下应该是 ImageList1）。

(2) 双击 Images 控件，此时会出现如图 5.40 所示的对话框。

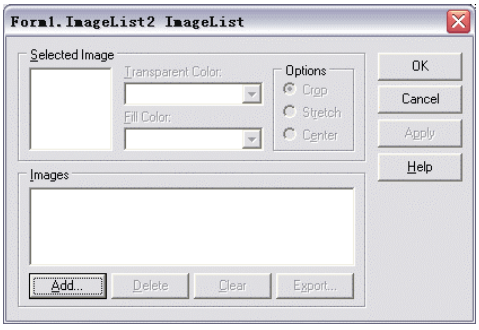

图 5.40 ImageList 对话框

(3) 单击该对话框中的 Add 按钮，此时会显示一个载入图片的对话框，如图 5.41 所示，在此你可以选择一些位图文件或者图标文件。需要注意的是，这些图片的尺寸应该相同，否则，将这些图片载入到 TImages 控件中时，Delphi 会自动将它们分割。

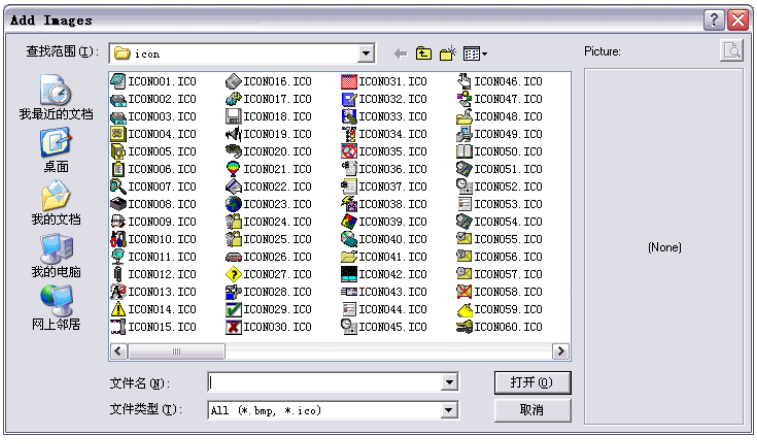

图 5.41 添加图片对话框

(4) 在属性编辑器中，找到 ItemsEx 属性，单击它旁边的按钮，此时会显示如图 5.42 所示的对话框。

(5) 图 5.42 所显示的对话框是 Delphi 中经常使用的一个为控件添加项目的对话框。在这个对话框上有四个按钮。

- 第一个按钮用来添加新的项目，在这里添加一个 ComboBoxEx 控件的项目。
- 第二个按钮用来删除已经存在的项目。
- 第三个按钮和第四个按钮的作用是调整这些项目的顺序。

图 5.42　添加新项目对话框

(6) 单击第一个按钮，此时会添加一个新的项目，此时你会发现对象浏览器中的项目发生了变化，如图 5.43 所示。

(7) 此时的属性编辑器也相应地发生了变化，显示了该对象的属性，如图 5.44 所示。

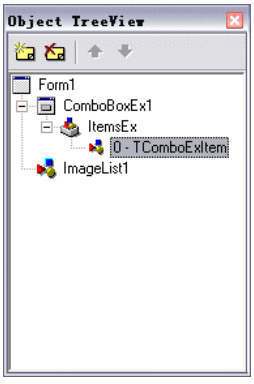

图 5.43　对象浏览器

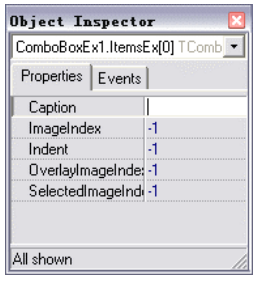

图 5.44　属性编辑器中的对象属性

(8) 从这个属性编辑器中可以看到，该对象具有五个属性，它们的含义如下。

- Caption：标题属性，这个属性和 TLabel 等控件的对应属性是相同的。
- ImageIndex：代表了显示在该项目上的 ImageList1 控件中的图片，–1 表示没有图片，0 表示第一个图片。
- Indent：缩进的层次，–1 表示没有层次，0 表示缩进的第一层，依次类推。

图 5.45　ComboBoxEx 控件示例

- OverLayImageIndex：该属性和 ImageIndex 的含义类似，不过代表的是当焦点在上面移动时显示的图片。
- SelectedImageIndex：该属性和 ImageIndex 的含义类似，不过代表的是选定某个项目时项目的图片。

(9) 按照上面的顺序，我们添加了如图 5.45 所示的一些项目。

5.6 容 器 控 件

上面介绍的控件，特别是文本处理控件、按钮控件和选项控件，当我们在一个窗体上安排它们时通常会根据它们的内容不同，把它们分成几个部分。在把窗体分成几个部分时，通常会使用到在这一节中要介绍的容器控件。在 Delphi 中，提供了多个可以用作容器的控件，它们包括 TGroupBox、TRadioGroup、Tpanel，等等。

5.6.1 TGroupBox 控件

TGroupBox 控件的作用就是作为一个容器控件，在这个容器控件中放置其他控件。通过这些控件可以把一个窗体错落有致地分隔成许多代表一定功能的部分。这个控件的使用比较简单，它对我们来说比较常用的控件只有两个，一个是 Caption 属性，修改这个属性可以比较好地反映这个容器中的内容；另一个是 Color 属性，它的使用方法我们也已经比较熟悉了，通过修改这个属性，可以用颜色区别开不同的部分。

在图 5.46 所示的程序中，我们就使用了两个这样的控件。

图 5.46 TGroupBox 控件示例

当然也可以把 TGroupBox 控件的 Caption 属性设置为空，那么这个控件就变成了一个方框。

5.6.2　TRadioGroup 控件

这个控件可以说是 TGroupBox 控件和 RadioButton 控件的组合，它惟一的功能就是作为前面介绍的一个 RadioButton 控件组，只不过在处理这些 RadioButton 控件时非常类似于一个 RadioButton 控件的列表框。

观察这个控件的属性编辑器，可以发现一个 Items 属性，单击这个属性旁边的省略号按钮，此时也会出现一个编辑对话框。可以像编辑列表框控件中的 Items 属性一样来编辑这些项目，同样也可以在程序中像访问列表框控件中的 Items 属性一样访问 TRadioGroup 控件的 Items 属性。

在程序中，当我们单击某个选项按钮的时候，这个选项按钮就会被选中，TRadioGroup 控件的 ItemIndex 属性就变成所单击的这个选项按钮的索引号。所以在程序中可以根据 ItemIndex 属性来进行操作，示例程序如下所示。

```
procedure TFrmComboBox.RgStyleClick(Sender: TObject);
begin
case RgStyle.ItemIndex of
    0: CmbItem.Style := csDropDown;
    1: CmbItem.Style := csDropDownList;
    2: CmbItem.Style := csOwnerDrawFixed;
    3: CmbItem.Style := csOwnerDrawVariable;
    4: CmbItem.Style := csSimple;
end;
CmbItem.Height := 250;
end;
```

5.6.3　TPanel 控件

TPanel 控件是 Delphi 中最常用的容器控件之一，它可以作为许多控件的父控件。由于 TPanel 控件的主要目的是作为一个容器控件，所以 Delphi 为改善它的外观提供了多种属性。下面我们就来介绍一下如何使用这些属性来控制 TPanel 控件的外观。

TPanel 控件有五个比较重要的属性，它们对 TPanel 控件的外观具有重要的影响。第一个是 BevelOuter 属性，它控制着 TPanel 控件的外部斜角形态。同样有一个 BevelInner 属性，它控制着 TPanel 控件的内部斜角形态。它们都是 TPanelBevel 型属性，具有四个预设值，如下所示。

- bvNone：没有斜角。
- bvLowered：嵌入的斜角。
- bvRaised：升起的斜角。

❖ bvSpace：在 TPanel 控件中这个属性值和 bvRaised 具有相同的效果。

我们要介绍的第三个属性是 BevelWidth，从字面上来理解就是斜角的宽度。改变它可以修改斜角的宽度。第四个属性是 BorderWidth，即边框的宽度。需要注意的是，这里的边框宽度并不是我们要介绍的 Border 的宽度，而是内外斜角之间的距离。设置 TPanel 控件的 BorderStyle 属性可以控制控件是否显示边框。

通过下面的一段程序，我们演示了如何在运行期控制 TPanel 控件的一些属性，并演示了 TPanel 各种斜角的组合，以及在不同的斜角宽度的情况下的状态。

```
procedure TForm1.FormCreate(Sender: TObject);
begin
combobox1.ItemIndex := 0;
ComboBox2.ItemIndex := 0;
end;

procedure TForm1.ComboBox1Change(Sender: TObject);
begin
case ComboBox1.ItemIndex of
    0: panel1.BevelOuter := bvLowered;
    1: panel1.BevelOuter := bvNone;
    2: panel1.BevelOuter := bvRaised;
    3: panel1.BevelOuter := bvSpace;
end;
end;

procedure TForm1.ComboBox2Change(Sender: TObject);
begin
case ComboBox2.ItemIndex of
    0: panel1.BevelInner := bvLowered;
    1: panel1.BevelInner := bvNone;
    2: panel1.BevelInner := bvRaised;
    3: panel1.BevelInner := bvSpace;
end;
end;

procedure TForm1.Edit1Change(Sender: TObject);
begin
panel1.BevelWidth := strtoint(edit1.text);
end;

procedure TForm1.CheckBox1Click(Sender: TObject);
begin
```

```
case CheckBox1.Checked of
    true: panel1.BorderStyle := bsSingle;
    false: panel1.BorderStyle := bsNone;
end;
end;

procedure TForm1.Edit2Change(Sender: TObject);
begin
panel1.BorderWidth := strtoint(edit2.text);
end;
```

程序运行的结果如图 5.47 所示。

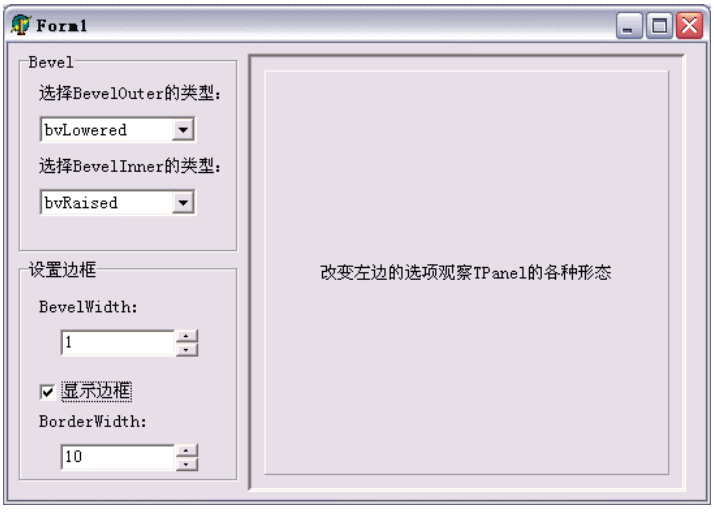

图 5.47　TPanel 控件的各种形态

5.6.4　TScrollBox 控件

TScrollBox 控件为我们提供了一个浏览大的对象的工具，它是一个典型的容器控件，目的就是把用户无法完整看到的一些内容利用滚动条翻转开来。例如我们在介绍 TImage 控件时，可能会存在一些图片比较大，无法在一个窗口中看到它全部画面的问题，虽然我们可以使用前面介绍的 Stretch 属性来把它缩放到一个指定的 TImage 控件中，但是，对于许多图片来说，这会造成失真，而且有的人不喜欢这样做。那么这时可以使用 TScrollBox 控件。

在 TScrollBox 控件的使用过程中，主要是关于它的两个滚动条的设置，例如设置 AutoScroll 属性，TScrollBox 控件会根据所包含的控件自动显示滚动条。它还有两组关于滚动条的属性，其中 Tracking 属性是一个比较重要的属性，如果把它设置成“ True ”，那么当我们拖动滚动条上的滑块时，TScrollBox 控件中的内容也会随之滚动。

5.7 本 章 小 结

在本章的内容中，我们介绍了 Delphi 中的窗体和一些基本的控件用法，特别是窗体的用法，介绍得非常详细。目的是通过这样一个大家经常使用的对象，把 Delphi 中的许多对象或者控件的共同点介绍出来。实际上，从后面我们介绍其他控件的过程中可以发现，窗体和很多控件是十分相似的。

在掌握控件和对象时，读者需要关注的重点是它的关键属性、关键方法和关键事件。这些是我们最常使用的。

Delphi 为我们提供了丰富的控件资源，我们应该充分利用它们。它们就像是积木，我们的主要工作首先是把它们合理地利用起来，搭建起好的程序界面。

第 6 章　工具控件和图形控件

在设计应用程序图形界面的过程中，需要使用的控件是很多的。我们在上面一章中介绍的只是其中的一部分，在这一章中将继续介绍如何设计应用程序的图形界面。在上一章中，主要是针对一些控件的属性和方法进行专门的介绍。在这一章中，在介绍新控件用法的同时，还会结合前面介绍的那些控件的用法，对这些控件的使用方法作进一步的介绍。

上面介绍的控件是构成图形界面的基本元素，在这一章中，除了介绍一些关于图像处理的基本控件之外，主要是介绍如何使用其他一些容器控件、工具控件以及其他的综合控件。

在这一章中将主要介绍：

- 工具控件
- 图形控件
- 图表控件

6.1　工 具 控 件

我们首先来介绍如何使用 Delphi 中的工具控件，这些控件的主要用途是在程序中提供某种数据、显示某种数据。例如使用滚动条控件（TScrollBar）可以获得一个形象地变化的整形数，我们对程序稍作处理便可以获得其他类型的数据；使用过程条控件（TProgressBar）可以形象地演示一个过程进行的进度；使用文件系统处理控件可以获得计算机系统的文件系统。这样类似的控件还有 TTrackBar、TUpDown。

6.1.1　滚动条控件

在 Windows 程序中，滚动条控件是非常常见的控件。在 Delphi 中，许多控件或者对象的滚动条都是自动添加上的。例如前面介绍的列表框控件，当其中的条目超出了显示的边界时，会自动添加上滚动条，并且当用户操作滚动条时列表框会自动滚动，并不需要我们对此进行任何的编程。

如果希望自己来操作一些数据的变化，那么可以使用 TScrollBar 控件。这个控件可以方便形象地用来获得一些整数。当然也可以获得一些其他类型的数据。

滚动条有两类，一是垂直滚动条，一是水平滚动条。在 Delphi 中，可以使用 TScrollBar 控件的 Kind 属性来控制滚动条控件的种类。一般来说，我们不需要在运行期改变这个属性，

这主要是在设计期完成的。当然也可以在运行期改变这个属性，例如利用下面的程序语句，就可以随时改变该控件的类型：

```
Scrollbar1.Kind:=sbHorizontal;
Scrollbar2.Kind:=sbVertical;
```

我们说过，使用 TScrollBar 控件可以获得一个在一定范围中变化的整数。那么首先要界定这个范围。滚动条控件有 Min、Max 属性，它们的含义是明显的，使用 Min 可以指定滚动条控件的最小值，使用 Max 可以指定滚动条控件的最大值。

那么当我们单击滚动条上的箭头按钮或者移动滑块时，滚动条的值（用属性 Position 来表示）会发生相应的变化。Position 是一个整数，它在 Min 属性值和 Max 属性值之间。我们可以在设计期设置该属性来指定滚动条中小滑块的初始位置，在运行期修改该属性可以使滚动条发生滚动。

该控件有两个控制滚动条滚动的属性：SmallChange 属性和 LargeChange 属性。第一个属性用来控制用户单击滚动条的箭头按钮滚动的步长，第二个属性用来控制用户在滚动条的滑槽上单击的时候滚动的步长。

该控件有两个主要的事件，一个是 OnChange 事件，这个事件发生在滚动条控件的 Position 属性发生改变时。如果需要根据 Position 属性进行操作，那么可以在这个事件中进行处理。另外一个重要的事件是 OnScroll 事件。这个事件发生在用户操作滚动条时，比如可能是按下了两端的箭头，也可能是单击滚动条内或键盘上的 PgUp 和 PgDn 键。这个事件的声明如下：

```
type   TScrollCode = (scLineUp, scLineDown, scPageUp, scPageDown, scPosition, scTrack, scTop,
scBottom, scEndScroll);
  TScrollEvent = procedure(Sender: TObject; ScrollCode: TScrollCode; var ScrollPos: Integer) of
object;
property OnScroll: TScrollEvent;
```

其中 ScrollPos 参数返回滚动条中小滑块的位置，这个位置还没有赋予滚动条控件，ScrollCode 参数返回滚动条的状态，可能值如表 6.1 所示。

表 6.1　ScrollCode 参数的预设值

参数值	说明
scLineUp	用户按下滚动条的左箭头或上箭头，或者按下键盘上的向上方向键
scLineDown	用户按下滚动条的右箭头或下箭头，或者按下键盘上的向下方向键
scPageUp	用户单击滚动条内的滑块左边或上边的区域，或者按下 PgUp 键
scPageDown	用户单击滚动条内的滑块右边或下边的区域，或者按下 PgDn 键
scPositon	用户在滚动条内拖曳滑块移动并且已经释放
scTrack	用户正在拖曳滑块
scTop	用户把滑块移动到滚动条的上端或左端
scBottom	用户把滑块移动到滚动条的下端或右端
scEndScroll	用户操作滚动条后释放了鼠标或按键

下面我们就建立一个应用程序来演示如何使用滚动条控件。

(1) 首先建立一个新的应用程序，然后设置 Form1 的标题、名称和字体。

(2) 在窗体上放置三个滚动条控件，把它们的 LargeChange 属性设置成 5，Max，Min 属性值为 255、0，然后把这三个滚动条控件的名称命名为 ScbRed，ScbGreen 和 ScbBlue。

(3) 在窗体上放置三个文本框，用来显示三个单纯颜色的值。

(4) 然后在 ScbRed 控件的 OnChange 事件中输入下面的代码：

```
procedure TFrmControl.ScbRedChange(Sender: TObject);
var r,g,b,c:longint;
begin
    r:=ScbRed.Position;
    g:=ScbGreen.Position ;
    b:=ScbBlue.Position ;
    EdRed.Text := inttostr(r);
    EdGreen.Text := inttostr(g);
    EdBlue.Text := inttostr(b);
    c:=b shl 16;
    ShpBlue.Brush.Color := c;
    c:=c+(g shl 8);
    ShpGreen.Brush.Color := (g shl 8);
    c:=c+r;
    ShpRed.Brush.Color := r;
    ShpColor.Brush.Color := c;
end;
```

通常来说，我们经常把颜色理解成三种颜色的合成，这三种颜色是红、绿和蓝。颜色的原始类型是长整型。例如纯红色是$000000FF，纯蓝色是$0000FF00，纯绿色是$00FF0000。黑色是$00000000，白色是$00FFFFFF。利用这三种颜色的不同值，我们可以组合出各种颜色。在上面的程序中，通过进行移位和求和运算可以组合出各式各样的颜色。

(5) 然后把 ScbRed 控件的 OnChange 事件句柄赋予 ScbGreen 和 ScbBlue 控件的 OnChange 事件句柄。

(6) 运行应用程序，改变三个滚动条，此时的应用程序运行结果如图 6.1 所示。

在 Delphi 中，类似的控件还有两个，它们是 TTrackBar 控件和 TUpDown 控件，这两个控件的用法和 TScrollBar 控件类似，都有 Min、Max 属性和 Position 属性。我们利用 Min、Max 属性来界定整数变化的范围，利用 Position 属性来获取当前控件所代表的值。需要说明的是 TUpDown 控件，它有一个 Associate 属性。通过这个属性可以和一个控件相联，通常来说是和一个文本框控件相联，例如在下面的窗口中我们就把 TUpDown 控件和一个文本框控件相联，并没有进行任何编程的工作，它就可以改变数字了，如图 6.2 所示。

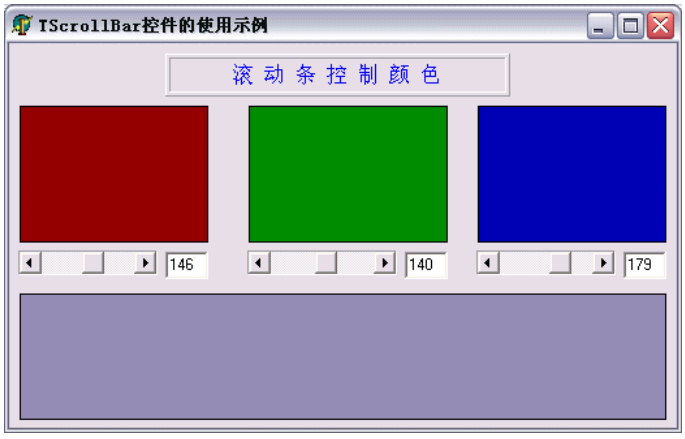

图 6.1　使用滚动条改变颜色

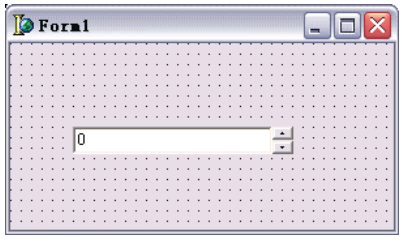

图 6.2　TUpDown 控件和 TEdit 控件相联

6.1.2　过程条控件

过程条控件（TProgressBar）经常用来显示一个过程进行的过程。特别是在许多应用程序的安装程序中，经常会看到这样的控件。在 Windows 系统的资源管理器程序中，当进行文件的复制和删除时出现的对话框中也会使用该控件。

实际上这个控件在许多方面和 TScrollBar 控件十分类似，它也有 Min、Max 和 Position 属性，它们所代表的含义和 TScrollBar 控件中的对应属性是相同的。我们也是用 Min、Max 属性来界定该控件变化的范围，然后用 Position 属性来获取或者设置当前的位置。除此之外，我们还利用另外几个属性来控制这个控件的行为。

该控件在默认的情况下是根据系统控制面板中规定的系统颜色来显示的。Step 属性定义了该控件变化的步长，在默认的情况下是 10。我们也可以通过另外一个属性，使得该控件连续变化，这就是 Smooth 属性，改变这个属性的值可以使该控件在连续变化和块状变化之间切换。例如在如图 6.3 所示的程序中，演示了两个同时进行的过程变化情况。在这个程序中，可以通过改变两个文本框中的内容改变控件的 Max 属性和 Step 属性。

在编程的过程中，使用过程条控件已经可以满足绝大部分的需要了。但是 Delphi 还提供

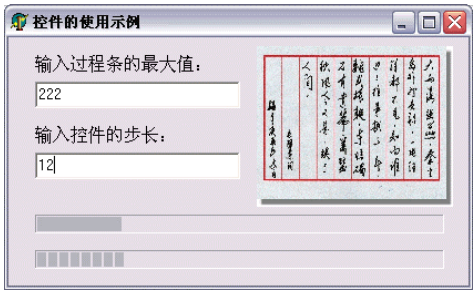

图 6.3　过程条控件的使用

了另外一个控件，它就是 TGauge 控件，这个控件的基本用途和过程条控件是完全相同的，所不同的是该控件具有多种不同的表现形式，例如饼图等。

这个控件的使用方法也和过程条控件类似，所不同的是该控件没有 Position 属性，而是有一个 Progress 属性，这个属性的含义和用法同 Position 是相同的。在图 6.4 所示的应用程序中，演示了该控件的大部分形式。

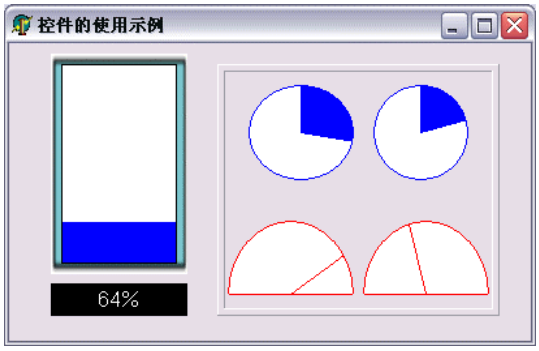

图 6.4　TGauge 控件的形式

程序代码如下：

```
procedure TFrmControl.Timer1Timer(Sender: TObject);
begin
    N1:=(N1+1) mod 100;
    if N1 mod 2= 0 then
    begin
        N2:=(N2+1) mod 100;
        if N2 mod 2 = 1 then
        begin
            N3:=(N3+1) mod 100;
            if N3 mod 2=1 then
            begin
                N4:=(N4+1) mod 100;
                if N4 mod 2= 1 then
```

```
                N5:=(N5+1) mod 100;
            end;
        end;
    end;
    Gauge6.Progress := N1;
    GauGe2.Progress := N5;
    Gauge4.Progress := N2;
    Gauge3.Progress := N3;
    Gauge5.Progress := N4;
    Gauge1.Progress := N5;
end;
```

6.1.3 文件系统控件

由于在后面的介绍中，我们经常会使用到文件系统来定位一些文件或者文件夹。同时这些控件也是我们在处理各种文档应用程序中经常使用到的。所以我们在这里先介绍一下关于文件系统的这些控件的使用方法。

这是一组控件，位于 Delphi **的** Win31 **选项卡上，包括** TFileListBox、TDirectoryListBox、TDriveComboBox **和** TFilterComboBox **控件。**

下面我们就来介绍如何利用这些控件获取文件的名称及其路径。

我们先来看一下 TDriveComboBox **控件。这是一个特殊的组合框控件，它能够列出在应用程序运行时计算机上所有的驱动器，包括软盘驱动器、硬盘驱动器和光盘驱动器。这个控件主要有三个属性，它们是** DriveList、Drive **和** TextCase。DriveList **属性可以和一个** TDirectoryListBox **控件相联，**TDriveComboBox **控件会自动把用户所选择的驱动器的信息传递给** DriveList **属性所指定的** TDirectoryListBox **控件。**Drive **属性是一个** Char **型属性，它代表的是当前选中的驱动器盘符。**TextCase **属性是用来控制显示的文本大小写的。**

TDirectoryListBox **控件是一个文件夹列表框控件，这个控件可以显示出指定驱动器上的文件夹列表。该控件的属性主要有四个，即** Drive、Directory、FileList **和** DirLabel **属性。**Drive **属性是一个** Char **型属性，它是该控件所显示的文件夹所在的驱动器盘符。**Directory **属性是一个** String **型属性，代表的是当前打开的文件夹。**FileList **属性可以和** TFileListBox **控件相联，使得** TDirectoryListBox **的** Directory **信息能够自动地传递给** TFileListBox **控件。**DirLabel **属性是一个** TLabel **型属性，通过它和一个** TLabel **控件相联，可以把当前的目录信息直接输入到该** TLabel **控件的** Caption **属性中。**

TFileListBox **控件和** TDirectory **控件十分类似，只不过它是一个文件的列表框控件。它具有** 7 **个主要的属性：**Directory、Drive、FileEdit、FileName、FileType、Mask、ShowGlyphs。Directory **属性代表的是列表框中显示的文件所在的文件夹。**Drive **是** Directory **所代表的文件夹所在的驱动器的盘符。**FileEdit **属性的作用和** TDirectory **控件中的** DirLabel **控件相似，可**

以和一个 TEdit 控件相联并把选中的文件的名称信息传递给该文本框。FileName 属性是一个 String 属性，这个属性代表了选中的文件的名称，包括路径。FileType 属性限定了显示在列表框中的文件的属性，定义如下：

```
TFileAttr = (ftReadOnly, ftHidden, ftSystem, ftVolumeID, ftDirectory, ftArchive, ftNormal);
TFileType = set of TFileAttr;
property FileType: TFileType;
```

Mask 属性实际上是一个文件过滤器，例如我们可以使用通用字符串（例如“*.*”表示所有文件，“*.bmp”表示所有的位图文件）来过滤显示在列表框中的文件。ShowGlyphs 属性是一个 Boolean 型属性，把它设置成 True，可以在列表框中显示一些代表普通文件和可执行文件的图标。

我们最后要介绍的控件是 TFilterComboBox 控件，它是一个过滤器组合框控件，也就是说，这个组合框中集合了我们在程序中可能需要的各种文件过滤器，只要在程序运行时进行选择就可以了。这种情况经常可以在许多应用程序的打开文件对话框中看到。这个控件主要有三个属性，FileList、Filter、Mask。FileList 属性和 TDirectory 控件中的同名属性的作用相同，我们可以为它指定一个 TFileListBox 控件，TFilterComboBox 控件会通过 FileList 属性把用户选定的过滤器信息自动传递给该 TFileListBox 控件。Filter 属性是 TFilterListBox 控件中所包含的所有过滤器所组成的文本。通过这个属性我们既可以获得也可以设置控件中的过滤器，例如可以使用下面的程序：

```
FilterComboBox1.Filter := 'All files (*.*)|*.*| Pascal files (*.pas)|*.pas';
```

但是我们建议读者在设计期通过如图 6.5 所示的 Filter Editor 对话框为 Filter 属性进行赋值。

图 6.5　Filter Editor 对话框

例如在上面的程序中，就输入了四个过滤器。Mask 属性是一个只读属性，通过这个属性可以获得当前用户选定的过滤器。

通过上面的介绍，你一定已经掌握了如何建立一个在计算机中定位某个或者某些文件的

应用程序了。我们可以在窗体上放置一个 TDriveComboBox 控件、TDirectoryListBox 控件、TFileListBox 控件和 TFilterListBox 控件，然后把 TDriveComboBox 控件的 DirList 属性设置成这里的 TDirectoryListBox 控件，把 TDirectoryListBox 控件的 FileList 属性设置成这里的 TFileListBox 控件，并把 TFilterListBox 控件的 FileList 属性设置成这里的 TFileListBox 控件。这样便建立了一个定位某个或某些文件的应用程序。如果愿意，还可以在窗体上放置一个 TLabel 控件和一个 TEdit 控件，并把它们指定给 TDirectoryListBox 控件的 DirLabel 属性和 TFileListBox 控件的 FileEdit 属性，此时的程序运行结果如图 6.6 所示。

图 6.6　使用文件系统控件

当然我们也可以在程序中把这些控件关联起来，代码如下：

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  FileListBox1.FileEdit := Edit1;
  FilterComboBox1.FileList := FileListBox1;
  DirectoryListBox1.FileList := FileListBox1;
  DirectoryListBox1.DirLabel := Label1;
  DriveComboBox1.DirList := DirectoryListBox1;
end;
```

从上面的程序中可以看出，这里介绍的这些控件的表现形式都是 Win31 下的控件形式。虽然它的功能是值得称道的，但是从形式上来说如果要用它们建立一个类似于 Windows 系统的资源管理器应用程序还是不够的。

在以前的时候，我们通常用 Windows 的 API 函数来完成这个任务，但是现在不同了，Delphi 6.0 提供了四个特殊的控件：ShellComboBox、ShellTreeView、ShellListView 和 ShellChangeNotifier，专门用来处理关于建立资源管理器或者其他的和文件系统相关的应用程

序。这是四个全新的 Delphi 控件，虽然网上已经出现了类似的控件，但是在 Delphi 的帮助文件里面却找不到这些控件的帮助。因此，如果要了解它们的属性以及方法，恐怕最好的方法就是直接阅读源代码了。

第一个我们需要介绍的控件是 ShellComboBox 控件，这个控件相当于 TDriveComboBox 控件，但是要比它漂亮得多，虽然在大多数情况下我们并不经常使用这个控件。我们并不需要修改这个控件的太多属性，因为在默认的情况下已经很完美了。但是我们仍然需要关注它的以下几个属性。

- 首先我们会发现它具有一些 ComboBox 控件具有的属性，比如 DropDownCount、Color 等等。在设置这些属性时和 ComboBox 控件是一样的。
- Images 属性：这个属性和前面介绍过的一样，主要是用来存储该控件需要的图片。但是在这个控件中，我们通常不需要使用这个属性。因为如果希望制作普通的下拉列表，那么用 ComboBox 控件或者 ComboBoxEx 控件就足够了，没有必要再使用这个控件。
- Root 属性：该属性是这类控件基本上都具有的一个属性，它代表了要展示的文件系统的起点。单击该属性旁边的按钮，会显示如图 6.7 所示的对话框。这个属性分为两个类型，一类是使用标准的路径名称，比如桌面、网上邻居等，这些内容包含在 Use Standard Folder 的下拉列表中，单击该下拉列表，会显示如图 6.8 所示的内容。用户可以通过选择其中的内容来选择合适的路径起点。它们的含义都是我们所熟悉的、在资源管理器中经常看到的一些内容。也可以使用绝对路径名称，此时需要使用 Use Path 下面的内容。可以在这个文本框中直接输入需要的内容，也可以通过旁边的按钮寻找一个路径。

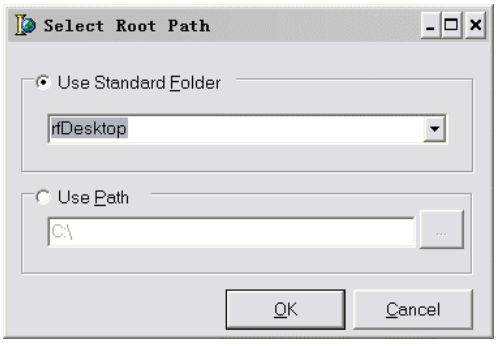

图 6.7　Root 属性设置对话框

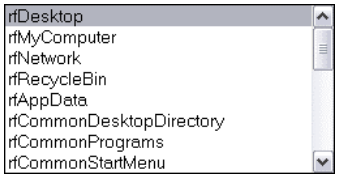

图 6.8　标准的路径名称

- ShellListView 属性 这个属性代表和 ShellComboBox 控件关联的 ShellListView 控件。可以在窗体上放置一个 ShellListView 控件，然后把它指定给该属性，那么两个控件就自动关联起来了。
- ShellTreeView 属性：这个属性代表和 ShellComboBox 控件关联的 ShellTreeView 控

件。可以在窗体上放置一个 ShellTreeView 控件，然后把它指定该属性，那么这两个控件就自动关联起来了。

- ❖ UseShellImages 属性：是否使用系统的默认文件系统图标，如果要开发具有自己格式的资源管理器程序或者其他和文件系统相关的程序，可以把该属性设置成 False，然后在窗体上添加一个 TImages 控件，用其中的图片来代替系统默认图标。

下面我们来介绍 ShellTreeView 控件的用法。这个控件的很多属性和 ShellComboBox 控件十分相似。这个控件的作用是自动生成一个类似于 Windows 系统资源管理器中的目录树结构。在该控件中，我们需要关注以下几个属性。

- ❖ AutoContextMenu 属性：若把这个属性设置成 True，则控件在用户右击鼠标的时候自动显示 Windows 操作系统中的关联菜单。对于本书所采用的系统，当用户在控件中右击鼠标的时候，将会显示和资源管理器中一样的快捷菜单，如图 6.9 所示。当用户选择其中的某个命令的时候，系统会自动运行该命令。如果把该属性设置成 False，那么将会关闭系统的关联菜单。
- ❖ AutoRefresh 属性：如果把该属性设置成 True，那么当用户修改了文件系统中的某些内容时，比如删除了一个文件或者添加了一个文件，那么这些改变将自动显示在 ShellTreeView 控件中；如果关闭该属性，那么这些修改将不反应在控件中。
- ❖ ChangDelay 属性：该属性决定了在改变 ShellTreeView 的根节点时的延迟时间，该时间是以毫秒为单位的。

图 6.9 ShellTreeView 控件的关联菜单

- ❖ Indent 属性：该属性决定了缩进的最大层数。
- ❖ ObjectType 属性：该属性决定了显示在控件中的内容，它包含以下三个属性。
- otFolders：当该属性为真时，文件夹将显示在该控件中。
- otNonFolders：当该属性为真时，一些非文件夹的内容，比如文件，也显示在该控件中。
- otHidden：当该属性为真时，具有隐藏属性的文件夹和文件等也显示在该控件中。
- ❖ RightClickSelect 属性：如果该属性为真，那么当用户在某个项目上右击时也会选中该项目。否则，只有单击时才会选中项目。
- ❖ Root 属性：该属性的含义和设置同 ShellComboBox 控件相同。如果需要在运行期改变该属性，可以使用下面的程序代码。

```
shelltreeview1.Root:='c:\';
shelltreeview1.Root:= 'rfMyComputer';
```

说明：

注意，在相互关联的 ShellComboBox **控件、**ShellListView **控件和** ShellTreeView **控件中，修改某个控件的** Root **属性，将自动改变其他控件的** Root **属性。**

- ShellComboBox **属性：该属性代表了和** ShellTreeView **控件相关联的** ShellComboBox **控件。**
- ShellListView **属性：该属性代表了和** ShellTreeView **控件相关联的** ShellListView **控件。**
- ShowButtons **属性：当该属性为** True **时，**ShellTreeView **控件将显示目录树旁边的按钮，否则将不显示这些按钮。显示该按钮和不显示该按钮的** ShellTreeView **控件如图** 6.10 **所示。**

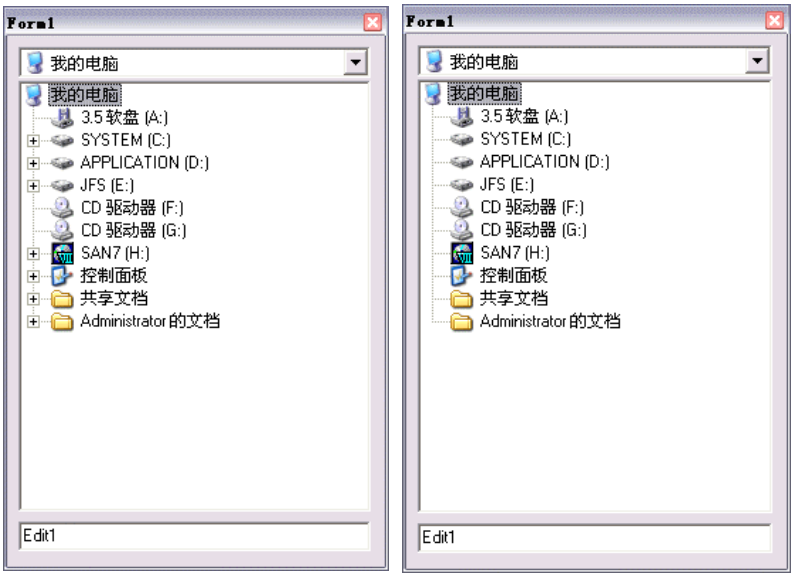

图 6.10　ShowButtons **属性的** True **值和** False **值的区别**

- ShowLines **属性：如果该属性值是** True**，那么** ShellTreeView **控件中的项目之间将用虚线连接，反之则没有任何线。**
- ShowRoot **属性：如果该属性的值是** True**，那么** ShellTreeView **控件中的根节点之前将显示按钮，否则将不显示按钮。**
- UseShellImages **属性：该属性和** ShellComboBox **控件中的同名属性相同。**

还有一个控件是 ShellListView **控件。这个控件相当于** Windows **系统中资源管理器右边的内容列表部分。看一下属性编辑器中的属性就会发现，它具有很多和** ShellTreeView **相同的属性。关于这些属性，可以参考上面所介绍的内容。下面我们将介绍一些它自己特有的或者非常关键的属性。**

- ❖ AutoNavigate 属性：如果该属性为 True，那么用户可以在该控件中浏览。当双击某个文件夹的时候，会自动进入到该文件夹。
- ❖ ViewStyle 属性：决定了列表框中的显示类型，有：vsIcon，vsList，vsReport 和 vsSmallIcon 四个类型。它们分别对应于资源管理器中的四种状态。
- ❖ IconOptions 属性：该属性主要是在 ViewStyle 属性为 vsIcon 或者 vsSmallIcon 的时候的显示状态，该属性包含三个属性。
- Arrangement 属性：如果该属性为 iaTop，那么图标将采用上对齐；如果该属性为 iaLeft，那么图标将采用左对齐。它们的样子如图 6.11 所示。

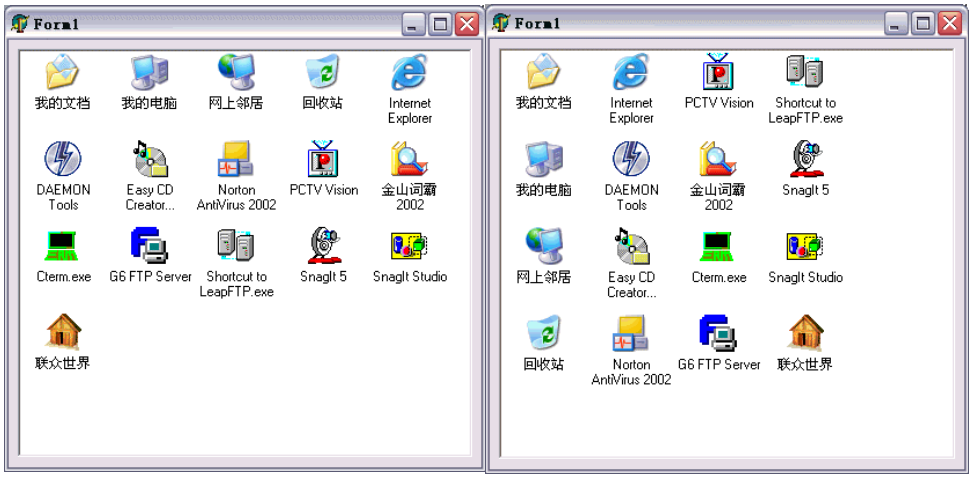

图 6.11 iaTop 和 iaLeft 属性值代表的不同形式

- AutoArrange 属性：该属性为 True 时，控件中的图标将自动排列。
- WrapText 属性：该属性为 True 时，在图标的标题文本超过规定的宽度时会自动绕行，否则将在一行中显示。
- ❖ MultiSelect 属性：该属性为 True 时，控件中的项目可以多选，否则只能单选。
- ❖ RowSelect 属性：该属性为 True 时，可以选择一行。
- ❖ ShellComboBox 属性：该属性指定了和该控件关联的 ShellComboBox 控件。
- ❖ ShowColumnHeader 属性：是否显示列的标题。
- ❖ Sorted 属性：当该属性为 True 的时候，控件中的内容将自动排序。

下面需要关注的是我们在选择其中的项目时如何获得它的路径以及名称。因为在这样的文件系统处理程序中，这才是我们最需要的。

如果禁止了多选，那么可以使用下面的代码取得项目的路径及其名称：

```
Path:=shelllistview1.selectedfolder.PathName;
```

对于 ShellTreeView 控件和 ShellComboBox 控件，也可以利用上面的属性来取得路径。

如果是在多选的情况下，可以利用下面的语句来获得项目的路径及其名称：

```
if ShellListView1.SelCount <=0 then
    exit;
memo1.Clear;
for i:=0 to ShellListview1.Items.Count-1 do
begin
    if ShellListView1.Items[i].Selected then
        memo1.Lines.Add(ShellListView1.Folders[i].PathName);
end;
end;
```

在上面的过程中，我们利用了每个项目的 Selected 属性来判断是否选择了该项目。当我们判定一个项目被选择时，可以利用下面的语句来判断项目是否是一个文件夹：

```
if   ShellListView1.Folders[i].IsFolder then
```

6.2　图 形 控 件

在 Delphi 中用于处理图形的控件主要有 TImage 控件、TShape 控件、TImageList 控件和 TChart 控件。这些控件分别具有自己的用途，TImage 控件主要是用来显示存储在文件中的图形；TShape 控件是用来显示预定的一些几何形状；TImageList 控件主要是用来存储一系列的图片，以便于在其他的控件或者对象中进行使用；TChart 控件的主要用途是用来生成各式各样的统计图表，例如饼图或者棒图。

6.2.1　图像控件

在图形化的程序中，图像的使用是比较频繁的。显然使用图像可以大大丰富应用程序的界面效果，但是我们要注意，并不是在应用程序中使用越多的图像，应用程序的效果就越好。必须根据实际的需要确定使用多少以及使用什么样的图像，而不要盲目地添加诸多的图像，以免画蛇添足。在 Delphi 中，TImage 控件可以用来显示已经存储在文件中的图形。首先我们先建立一个使用图像的应用程序。这个应用程序可以浏览用户选择的图形文件，有点像图像浏览器。

(1) 首先建立一个类似于上面我们在介绍文件控件的时候建立的应用程序，只要调整一下程序中的各个文件控件的位置就可以了。

(2) 在窗体上放置一个 TImage 控件，把名称修改成“ImgShow”，并把 Center 属性修改成“True”。Center 属性为 True 的时候将把图片显示在 TImage 控件的中央。

我们的目的是当用户在我们所提供的文件控件中选定一个文件时，如果这个文件是一个图形文件，那么就显示这个图形文件中所包含的图形，所以需要使用 TImage 控件的 Picture 属性的 LoadFromFile 方法。这个方法将从文件中读入图形。该方法的声明如下：

```
procedure LoadFromFile(const FileName: string);
```

这个方法只有一个字符串参数，利用前面介绍的 ShellListView 控件的 SelectedFolder.pathname 属性正好可以获得一个文件的完整路径名称，所以使用这个方法可以实现上面的目的。

但是另外一个问题是，图形可以有许多的文件格式，例如大家熟悉的位图文件（BMP 文件），还有图标文件（ICO 文件）以及其他的一些文件格式，那么如何来辨别这些文件格式呢？在 Delphi 中主要是利用图形文件的扩展名来区分不同的文件格式。例如“.bmp”代表图形文件。所以我们就不必为图形的格式而操心，只要指定正确的文件名就可以了。如果一个图形文件采用了某个扩展名但是实际上是另外一种格式的话，在调用 LoadFromFile 方法的时候会引发一个异常。

当然我们也可以分辨出在 TImage 控件中显示的是什么图形格式，这需要利用 TPicture 对象的属性。在图 6.12 中显示了该对象的属性，这是 Delphi 的帮助窗口。

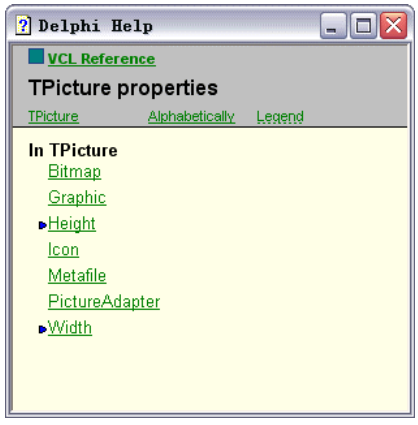

图 6.12　TPicture 对象的属性

从图中可以看出，TPicture 对象在默认的情况下，支持位图文件、图标文件和元文件三种格式。这三种格式的文件用户不需要进行任何操作就可以直接进行显示。对于其他的文件格式则不能直接进行显示。

(3) 然后在 ShellListView1 控件的 OnClick 事件中输入下面的代码：

```
procedure TFrmControl.ShellListView1Click(Sender: TObject);
var
  FileExt: string[4];
begin
```

```
if ShellListView1.SelCount <=0 then
    exit;
FileExt :=AnsiUpperCase (ExtractFileExt(ShellListView1.selectedfolder.PathName));
if (FileExt = '.BMP') or (FileExt = '.ICO') or (FileExt = '.WMF') or (FileExt = '.EMF') then
begin
  ImgShow.Picture.LoadFromFile(ShellListView1.selectedfolder.PathName);
  if (ImgShow.Picture.Width > ImgShow.Width)
        or (ImgShow.Picture.Height>ImgShow.Height) then
    ImgShow.Stretch := True
  else
      ImgShow.Stretch :=false;
end;
If FileExt = '.JPG' then
begin
      try
        ImgShow.Picture.LoadFromFile(ShellListView1.selectedfolder.PathName);
      except
        on EInvalidGraphic do
          ImgShow.Picture.Graphic := nil;
      end;
      SetJPEGOptions(self);
  end;
end;
```

在这段程序中，首先提取文件的扩展名，为了便于操作，把所有的扩展名都变成大写字符，AnsiUpperCase **方法就是用于这个目的的。然后如果是图形文件，那么就显示该图形。**

(4) **运行应用程序，运行结果如图** 6.13 **所示。**

在上面的程序中，使用了 TImage **控件的** Stretch **属性。这个属性将根据图片的尺寸进行调整，在不改变** TImage **控件尺寸的情况下把图片进行缩放，以充满整个** TImage **控件。但是设置** Stretch **属性带来的另外一个问题是对于尺寸小于** TImage **控件的图片来说可能会使得显示效果变得粗糙，由于图片较小，所以使得显示效果比较差。对于这个问题，上面程序的解决办法是根据图片的尺寸来确定是否设置** Stretch **属性。下面的程序代码就实现了这一功能：**

```
if (ImgShow.Picture.Width > ImgShow.Width)
 or (ImgShow.Picture.Height>ImgShow.Height) then
      ImgShow.Stretch := True
  else
      ImgShow.Stretch :=false;
```

在编程中会经常使用到 TImage **控件的其他一些属性，例如** AutoSize **属性，如果把这个属性设置成“** True **”，那么** TImage **控件会根据用户选择的图片的大小自动调整自身的大小。**Transparent **属性可以决定** TImage **控件是否透明。**

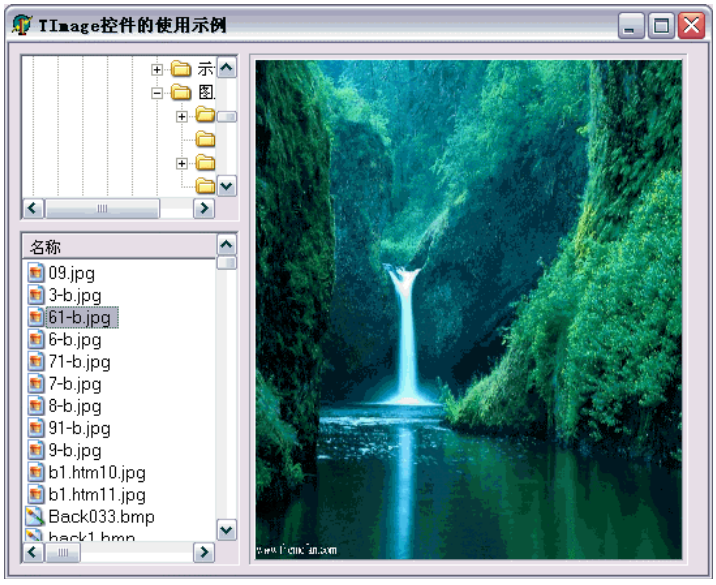

图 6.13　图像浏览器

在上面的程序中，我们使用的都是文件，也就是说从文件中载入图片。实际上还可以从其他资源中载入图片。例如使用资源文件或者使用剪贴板。

6.2.2　形状控件

形状控件（TShape）用于显示一些常见的几何形状。这个控件是比较简单的，它主要有三个属性，Brush、Pen 和 Shape 属性。第一个和第二个属性我们将在后面介绍绘图的时候将进行专门的介绍。Shape 属性中提供了一些预设的几何形状。通过设置这个属性可以改变控件的形状。

6.2.3　图像列表控件

图像列表控件的使用比较简单，我们在前面介绍列表框控件的时候已经使用了这个控件。这个控件的主要作用是提供一系列的统一尺寸图片，以便成组地使用在控件中的成组项目。在使用这些图片时只要利用一个索引号就可以了。通过如图 6.14 所示的 ImageList Editor 对话框，可以把一些图片添加到图像列表控件中。在图像列表控件上右击鼠标，然后从弹出的快捷菜单中选择 ImageList Editor 便可以调出该对话框。

关于如何把图像列表控件的索引号和其他控件的项目相关联的方法将会在介绍对应的控件时进行详细介绍，这里就不再赘述了。

在 Delphi 中，也提供了在运行期对图像列表控件中的图片进行处理的方法，例如 Add、

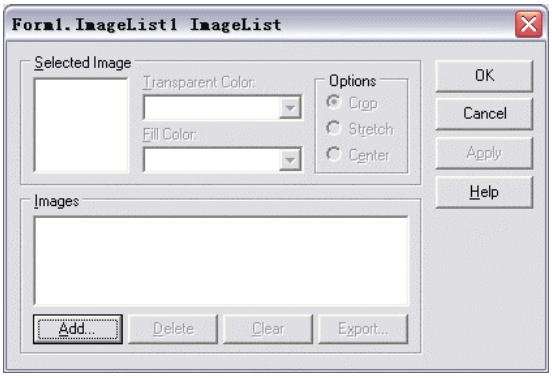

图 6.14　ImageList Editor 对话框

Delete 等。但是我们认为这远远没有改变引用图像列表控件中图片的控件索引号来得方便，也就是说，可以在设计时就把可能需要的所有图片都添加到图像列表控件中，然后在运行时，如果需要改变某个控件引用的图片，改变它的索引号就可以了。

6.3　图 表 控 件

图表控件（Tchart）是 Delphi 中一个相当复杂的控件，它位于 Delphi 的控件选项板的 Additional 选项卡上，如图 6.15 所示。

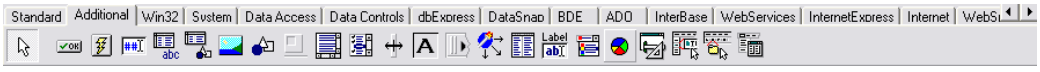

图 6.15　Delphi 的图表控件

这个控件的功能是十分强大的，它可以用来制作统计图、饼图、棒图等常见图表，另外它还具有许多其他的强大功能。同时它又具有十分复杂的属性和方法，要想像上面我们介绍其他控件一样来介绍这个控件的使用方法是十分不容易的，所以在下面的讲解过程中，我们将在介绍一部分内容之后就建立一个示例程序，通过一个一个的程序，力图使读者能够掌握该控件的使用方法。

6.3.1　使用不同类型的 Series

TChart 控件的用途是绘制一个一个的图表。在 Delphi 中，一个图表就是一个 Series，所以在 TChart 控件中，要处理的主要对象是 Series，其他的功能都是为了增强 Series 的功能和效果而设置的。下面就来介绍如何处理 Series。

(1) 首先建立一个新的应用程序，然后在窗口上放置一个 TChart 控件，并把它命名为“ChtShow”。

(2) 右击 TChart 控件，并从弹出的快捷菜单中选择 Edit Chart 命令，会出现如图 6.16 所示的 Editing Chart 对话框。

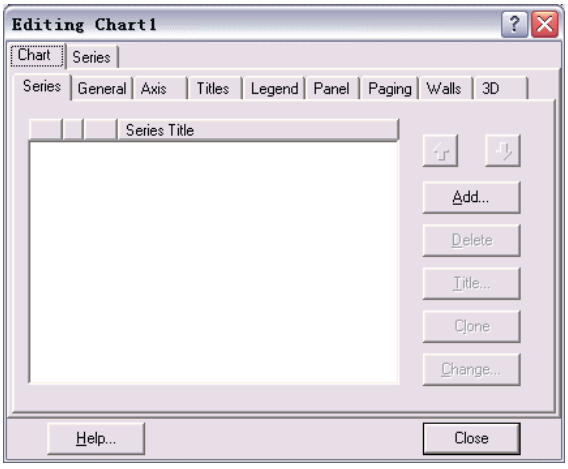

图 6.16 Editing Chart 对话框

(3) 这个对话框中有多个选项卡，利用这个对话框可以进行完整的 TChart 控件的属性设置。我们先来观察第一个选项卡中的选项的设置。在第一个选项卡中，可以处理出现在 TChart 控件中的 Seires 对象的添加和删除等操作。单击对话框中的 Add 按钮，会出现如图 6.17 所示的 TeeChart Gallery 对话框。

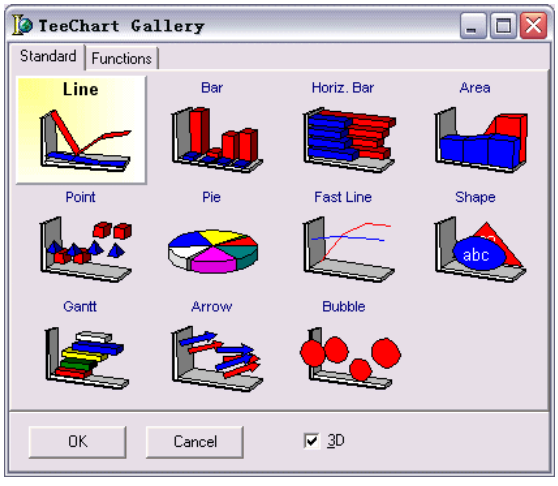

图 6.17 TeeChart Gallery 对话框

(4) 这个对话框的主要用途是在设计期为 TChart 控件添加各种形式的 Series。在图中列出了 11 种 Series，它们又都具有两种形式：3D 形式和平面形式。选中对话框底部的 3D 复选框，就会添加 3D 形式的 Series，反之则添加平面形式的 Series。通过这样的方法添加的 Series

是一个个独立的 Series 对象，也就是说它们相互之间并没有任何相关的联系。双击其中的一个图标便可以为 TChart 控件添加一个 Series，例如在这里单击 Line 图标，此时便为 TChart 控件添加了一个线型的 Series，并返回到如图 6.18 所示的对话框。

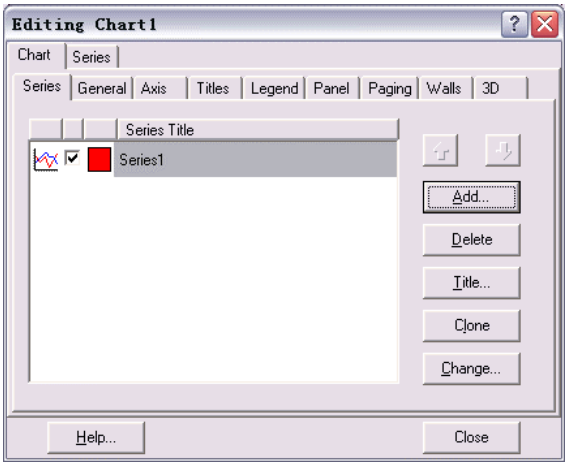

图 6.18　添加了一个 Series 的 Editing Chart 对话框

(5) 在这个对话框的列表中，显示了新加入的这个 Series 的一些属性，例如是否激活，也就是说是否在默认的情况下对它进行显示。如果希望取消默认的显示功能，可以清除列表框中的复选框。单击复选框旁边的颜色按钮，还可以改变它的颜色。

(6) 如果在加入了一个 Series 之后又不希望它存在，可以使用这个对话框上的 Delete 按钮删除这个 Series。利用 Title 按钮可以修改 Series 的标题。利用 Clone 按钮，可以复制一个和先前 Series 相同类型的 Series。利用 Change 按钮可以修改当前 Series 的一些类型。

(7) 利用同样的方法为 TChart 控件添加各种类型的 Series，每个 Series 各添加一个。然后在窗体上放置一个 TComboBox 控件，再把不同的 Series 的类型名称输入到组合框中。并在组合框的 OnChange 事件中加入下面的代码：

```
procedure TFrmControl.ShowSeries (index:integer);
var i:integer;
begin
    for i:=0 to (chtshow.Serieslist.Count-1) do
        chtshow.series[i].active:=false;
    chtshow.Series[index].active:=true;
end;
procedure TFrmControl.ComboBox1Change(Sender: TObject);
begin
showseries(combobox1.itemindex);
end;
```

上面的代码的作用是当用户在组合框中选择某种类型的 Series 的时候，在 TChart 中就显示对应的 Series。

(8) 但是需要注意的是，就目前来说，在 TChart 控件的 11 个 Series 中，我们并没有添加任何值。即使有，也是 Delphi 自动添加的一些随机数，所以为了确保每个 Series 中都有我们加入的值，可以在窗口的 OnCreate 事件中加入下面的代码：

```
procedure TFrmControl.FormCreate(Sender: TObject);
var i:integer;
begin
for i:=0 to chtshow.SeriesList.count-1 do
      chtshow.Series[i].FillSampleValues(6);
Combobox1.ItemIndex :=0;
showseries(0);
end;
```

这段代码的意义是首先调用 FillSampleValues 方法，用指定个数的随机数填充 Series。这个方法的参数就是随机数的个数。

(9) 运行上面的应用程序，并从组合框中选择对应的 Series，便可以显示不同类型的 Series，如图 6.19 所示。

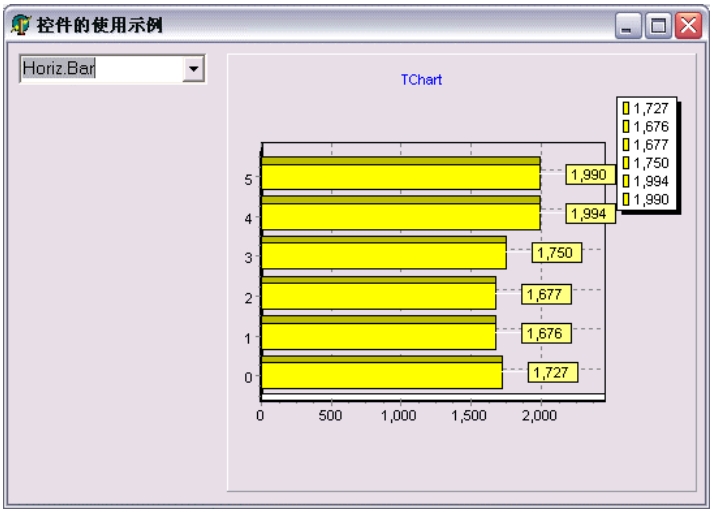

图 6.19 TChart 控件的 Series 示例

6.3.2 Series 的 Function

我们在前面已经说过，通过上面的方法添加的 Series 相互都是独立的。在实际的应用中，经常会遇到一列数据是根据其他几列数据计算得来的情况。通过上面的图 6.17 所示的对话框

中的 Functions 选项卡，可以添加根据其他 Series 的值来确定本身值的 Series。该选项卡如图 6.20 所示。

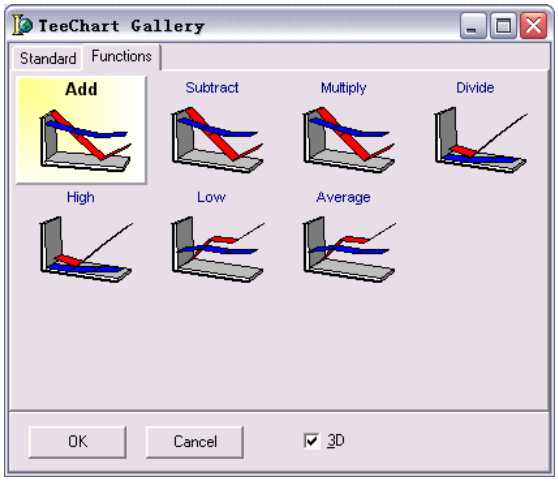

图 6.20　TeeChart Gallery 对话框的 Funtions 选项卡

双击对话框中的某个图标，便可以为 TChart 控件加入一个 TeeFunction。随之会出现如图 6.21 所示的 Editing Cht Show 对话框的 Series 选项卡。

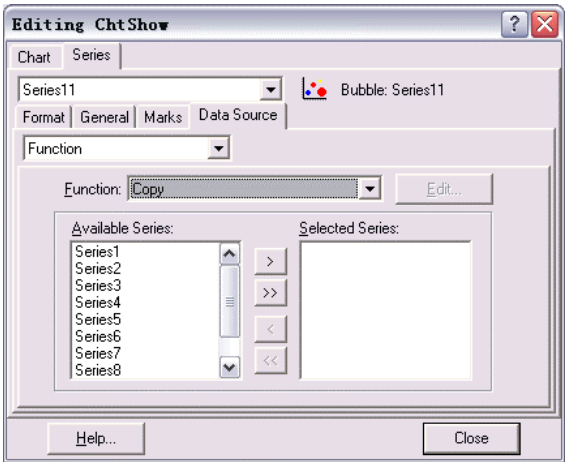

图 6.21　Editing Cht Show 对话框

在这个对话框中，可以对这个 Series 的数据源进行设置。在上面的组合框中，可以选择数据源的类型，在这里是 Function，另外的两个选项是 No Data 和 Random Values。No Data 就是说该 Series 在初始的情况下不添加任何数据，而 Random Values 将用一些随机数作为 Series 的数据。在这里比较复杂的是 Function。在第二个组合框中，可以选择 Function 的类型。在 Delphi 中，提供了 7 种 Function 类型。在选择了一种 Function 类型之后，可以在下面

的列表框中选择需要作为根据的 Series，通常可以选择两个或者两个以上的 Series。当所选择的 Series 多于两个时，它们之间是连续操作的关系。举例来说，如果选择的是 Subtract，那么选择的多个 Series 之间是连续减的关系。

例如对于上面的程序，我们可以选择 Series1 和 Series2。此时的应用程序将如图 6.22 所示。

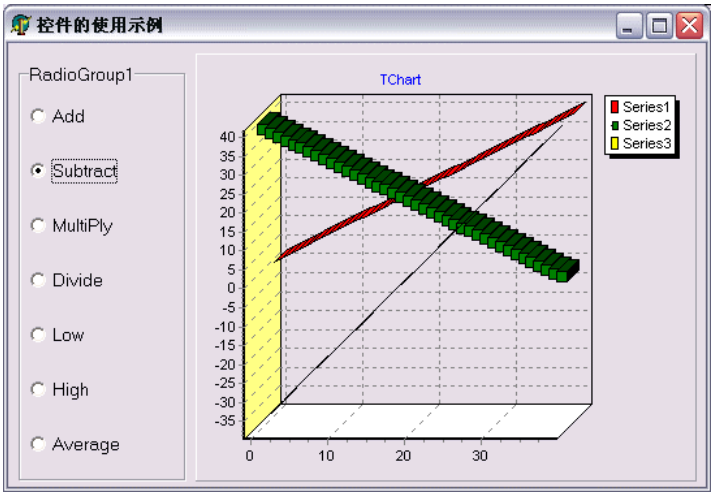

图 6.22 设定了 Function 的 Series

从图中可以看出，Series3 是 Series1 减去 Series2 的结果。

当然，可以在程序运行时指定 Function 的类型。例如，可以利用下面的代码来指定 Series3 需要根据的两个 Series：

```
With series3.DataSources do
 begin
  Clear;
  Add( Series1 );
  Add( Series2 );
 end;
```

在上面的程序中，首先清除 Series3 的 DataSource，然后加入 Series1 和 Series2。在指定了 Series3 的 Function 的数据源之后，就可以指定 Series3 的 Function 类型了。例如这里在程序中加入了一个选项按钮组框，关于这个控件的用法将在本章的后续内容中进行详细介绍。在输入了各个 Function 类型的名称之后，在该控件的 OnClick 事件中加入下面的代码：

```
procedure TFrmControl1.RadioGroup1Click(Sender: TObject);
begin
case RadioGroup1.ItemIndex of
    0: series3.SetFunction(TAddTeeFunction.Create(self));
```

```
    1: series3.SetFunction(TSubtractTeeFunction.Create(self));
    2: series3.SetFunction(TMultiplyTeeFunction.Create(self));
    3: series3.SetFunction(TDivideTeeFunction.Create(self));
    4: series3.SetFunction(TLowTeeFunction.Create(self));
    5: series3.SetFunction(THighTeeFunction.Create(self));
    6: series3.SetFunction(TAverageTeeFunction.Create(self));
end;
end;
```

在上面的代码中，使用了 Series **的** SetFunction **方法，这个方法声明如下：**

```
procedure SetFunction(AFunction:TTeeFunction); virtual;
```

其参数 AFunction **就是一个** Function **类型。上面的程序中列出了** Delphi **提供的** 7 **种** Function **类型。如果要删除一个** Function **类型，可以使用下面的语句：**

```
series1.SetFunction(Nil)；
```

在程序中，为了加强效果的对比，按照两条曲线的形式对 Series1 **和** Series2 **进行了赋值，赋值代码如下：**

```
for i:=1 to 40 do
 begin
   series1.Add(i);
   series2.Add(40-i);
 end;
```

运行上面的程序，运行结果如图 6.23 **所示。**

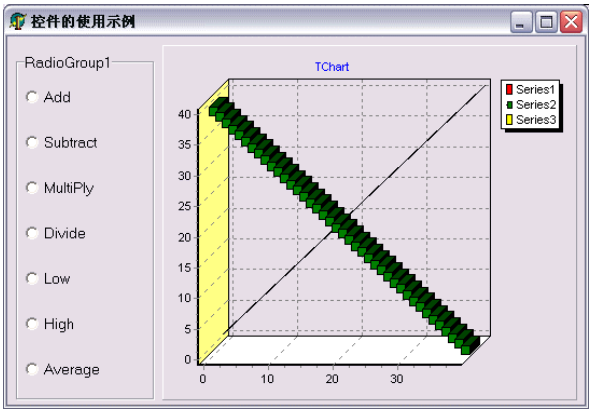

图 6.23　**程序运行的默认界面**

在程序中单击不同的选项按钮，观察 Series3 **的变化。**

(1) **选择** Add **单选按钮，此时的程序如图** 6.24 **所示。**

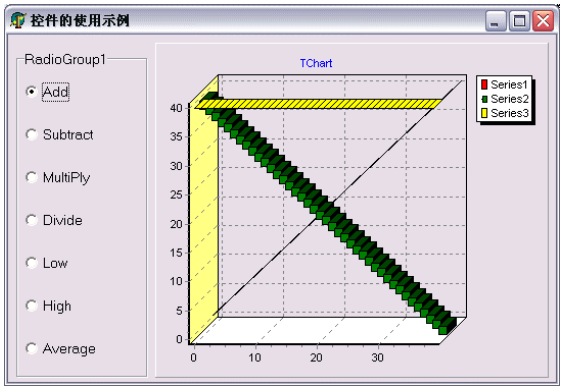

图 6.24 Add 类型

(2) 选择 MultiPly 单选按钮，此时的程序如图 6.25 所示。

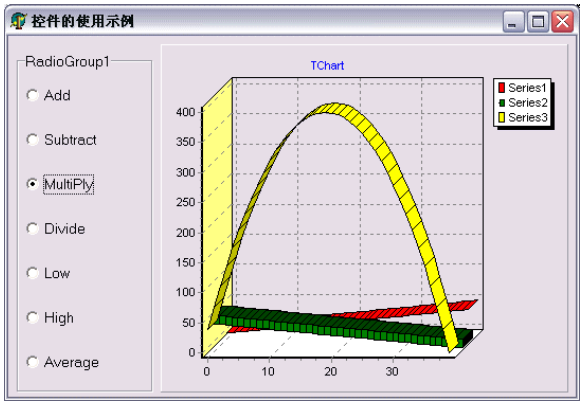

图 6.25 MultiPly 类型

(3) 选择 Divide 单选按钮，此时的程序如图 6.26 所示。

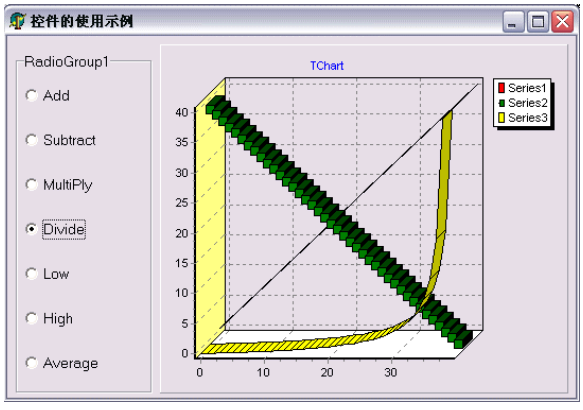

图 6.26 Divide 类型

(4) 选择 Low 单选按钮，此时的程序如图 6.27 所示。

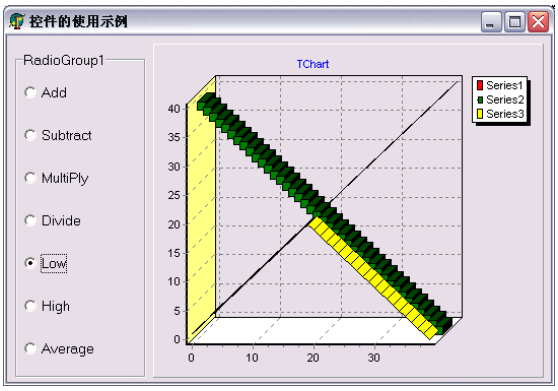

图 6.27　Low 类型

(5) 选择 High 单选按钮，此时的程序如图 6.28 所示。

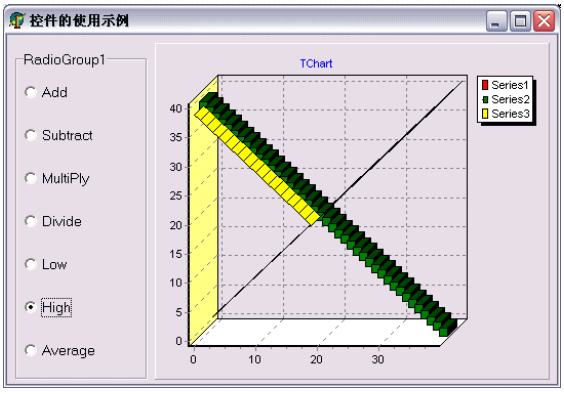

图 6.28　High 类型

(6) 选择 Average 单选按钮，此时的程序如图 6.29 所示。

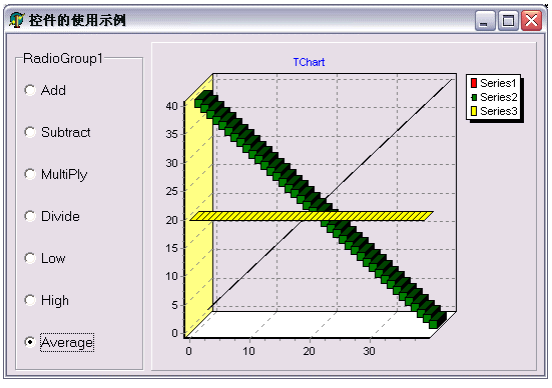

图 6.29　Average 类型

6.3.3 TChart 控件的选项

在上面的程序中，我们已经知道了如何在 TChart 控件中添加和处理 Series。但是在 Delphi 中，为了适应更多的情况，为 TChart 控件提供了许多比较复杂的选项，这里就将对它们进行介绍。

在上面的程序中，主要使用的是对话框的 Series 选项卡。这个选项卡是包含于第一级选项卡 Chart 之下的。在这里还有许多其他的选项卡，它们控制着 TChart 控件的表现形式。在这里我们主要以前面的三个选项卡为例，介绍它们的使用以及在程序中如何控制这些对应的属性。关于其他选项的设置读者可以参考这些选项卡。

该对话框的第二个选项卡如图 6.30 所示。

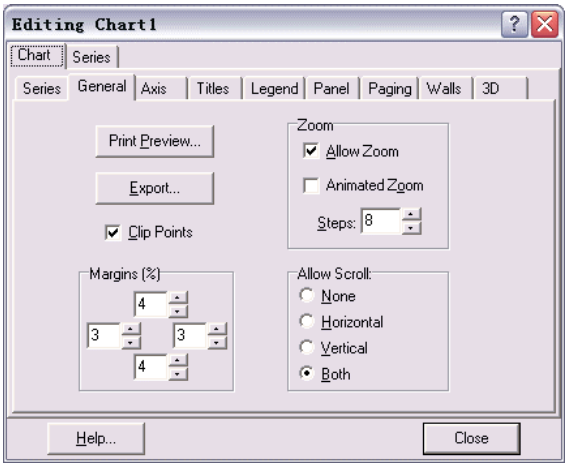

图 6.30 General 选项卡

这个选项卡中的选项主要控制 TChart 控件的一些常规属性。单击该对话框中的 Print Preview 按钮，会出现如图 6.31 所示的 TeeChart Print Preview 窗口。

在这个窗口中，可以设置关于 TChart 控件的打印功能选项，例如打印的边界，等等。这个功能在运行时需要通过 TChart 控件的 Print 方法来实现。例如可以使用下面的代码实现 TChart 控件的打印功能：

```
Chart1.Print;
```

在上面窗口中设置的四个边界的距离可以通过 MarginLeft 等四个属性来完成。

现在我们回到图 6.30 所示的对话框。单击 Export 按钮，会出现如图 6.32 所示的 Export TeeChart 对话框。

在这个对话框中，可以把当前的 TChart 控件保存成三种文件格式，或者复制到剪贴板上。实际上这样的功能在设计期并没有多大的实际用途，如果在运行期能够实现这样的功能，程

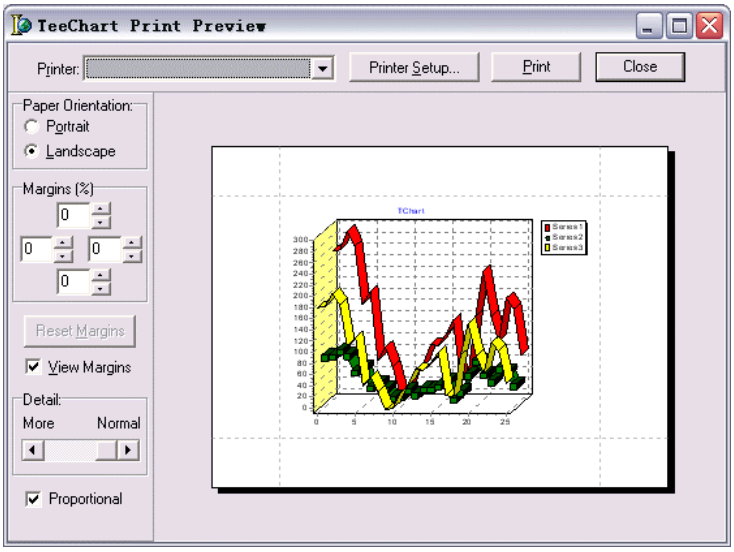

图 6.31　TeeChart Print Preview 窗口

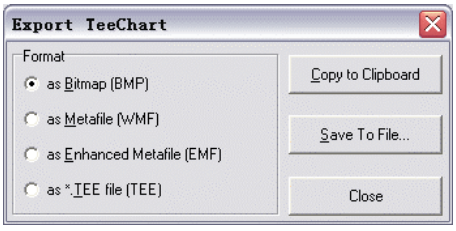

图 6.32　Export TeeChart 对话框

序的用途就非常大了。在后面的内容中，读者会看到我们在应用程序中也能够实现这样的功能。

在这个对话框中的 ClipPoints 属性用来控制 Series 中的值超过所指定的最大值或者低于最小值时是否显示这些值。如果把这个属性设置成“ True ”，那么当 Series 中的值超出了所规定的范围的时候，就会自动把这一部分图像切除。在默认的情况下这个属性的值是“ True ”。

在这个对话框中，还有关于缩放的设置。如果选中 AllowZoom，那么 TChart 控件将支持自动缩放。也就是说当用户在 TChart 控件上用鼠标拖出一个矩形时，TChart 控件将自动进行缩小或者放大。例如如果我们在 TChart 控件上用鼠标从左上角向右下角画出一个矩形，则 TChart 将自动进行放大，相反则进行缩小。如果选中了 AnimatedZoom 复选框，那么上面介绍的缩放过程将采用一定序列的逐步缩放来完成缩放过程。这个序列的步数是由 AnimatedZoomSteps 属性来指定的，这个属性的值显示在 Step 框中。当然我们可以修改这个属性。

在这个对话框中，还可以设置 TChart 控件的四个边界情况和滚动条的配置。

下面的主要任务是在运行期能够实现上面的大部分功能。程序代码如下所示：

```
procedure TFrmControl2.FormCreate(Sender: TObject);
begin
series1.FillSampleValues (10);
end;

procedure TFrmControl2.CheckBox1Click(Sender: TObject);
begin
chart1.AllowZoom := checkbox1.Checked ;
end;

procedure TFrmControl2.CheckBox2Click(Sender: TObject);
begin
chart1.AnimatedZoom := checkbox2.Checked ;
chart1.AnimatedZoomSteps := strtoint(edit1.text);
end;

procedure TFrmControl2.CheckBox3Click(Sender: TObject);
begin
chart1.ClipPoints := Checkbox3.Checked ;
end;

procedure TFrmControl2.Button1Click(Sender: TObject);
begin
chart1.SaveToBitmapFile ('e:\temp\try.bmp');
end;

procedure TFrmControl2.Button2Click(Sender: TObject);
begin
chart1.SaveToMetafile ('E:\temp\try.wmf ');
end;
```

上面列出了各个控件事件中的代码，它们大部分都比较简单，这里就不再进行说明了。程序的运行结果如图 6.33 所示。

单击程序中的两个按钮，根据上面的程序，将会保存两个图形文件，这两个图形文件如图 6.34 所示。

作为一个图表控件，通常具有自己的坐标线。这个坐标线可以出现在四个位置：左、右、上、下。那么如何控制它的一些行为呢？这就要使用到图 6.35 所示的 Axis 选项卡。

这里列出的 5 个选项卡是 5 个 TChartAxis 对象，我们在下面的内容只介绍关于 BottomAxis，也就是底部坐标线的设置情况，其他 4 个坐标线的设置可以依此类推。选中 Show Axis 复选框，会显示所设置的坐标线，如果清除这个复选框，那么将会隐藏所有的坐标线。在下面还有一个 Visible 复选框，这个复选框设置的是 Axis 框中选中的坐标线的可见情况。

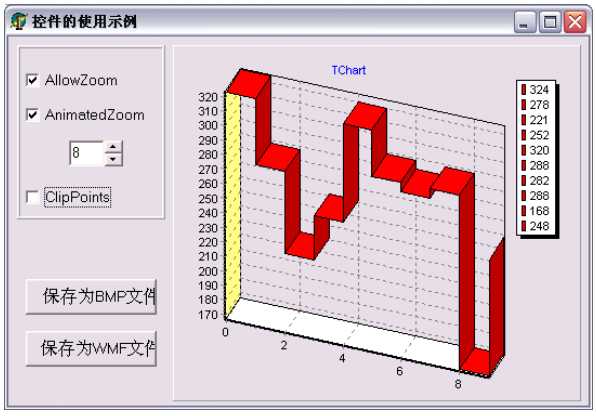

图 6.33　设置常规属性的应用程序

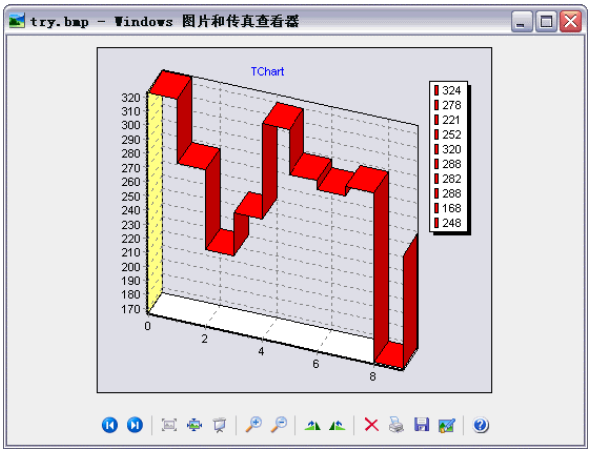

图 6.34　生成的两个图形文件

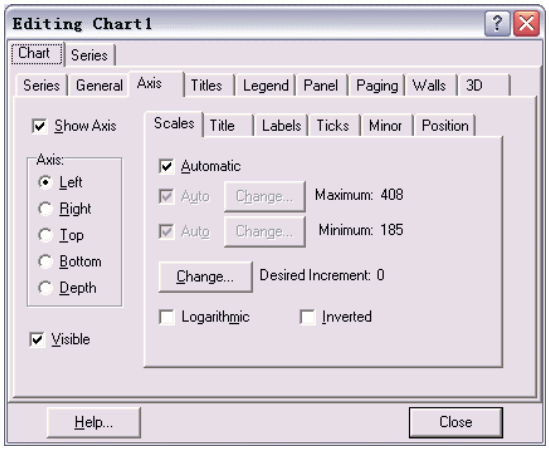

图 6.35　Axis 选项卡

在这个选项卡中又有 6 个选项卡，这 6 个选项卡设置的是在 Axis 框中选中的坐标线选项。

第 1 个选项卡是关于坐标线范围的。这个选项卡中有 5 个复选框。选中第一个复选框 AutoMatic，会根据 Series 中的数值的情况自动调整坐标线的最大值和最小值。另外通过下面的 Max 和 Min 复选框，可以单击旁边的按钮来设置对应的最大值或者最小值。单击 Min 复选框下面的按钮可以设置坐标线上的标注距离。如果所选的坐标线是数值型的，那么这个属性应该是一个数值；如果这个选中的坐标线是日期或者时间型的，那么这个属性应该是一个日期或者时间。在下面的内容中，我们将会介绍如何在程序中改变这个属性。第 4 个复选框是 Logarithmic，如果选中这个复选框，那么对应的坐标线会按照对数进行变化。第 5 个复选框是 Inverted，利用这个复选框可以把当前选中坐标线的最大值和最小值的显示顺序进行互换。

第 2 个选项卡可以设置坐标线上的标签格式和内容。这个选项卡的使用是比较简单的。可以在文本框中改变标签的内容、字体和显示角度。

第 3 个选项卡的使用也是比较简单的，它的主要作用是用来控制坐标线上坐标值的显示情况。需要说明的是 Multi-Line 复选框的含义。当我们选中这个复选框时，如果要显示的标注比较大，则可以把它进行多行显示。还有一个选项是 MinSeparation，这个选项设定的是标注在坐标线上的值之间的最小距离。也就是说，当要标注的值和其他的标注之间的距离小于这个数时，就不显示了。

第 4 个选项卡可以设置出现在坐标线上的短线长度等格式和 TChart 控件中的网格线格式。主要是长度和颜色以及线型的设置。

第 5 个选项卡和上一个选项卡十分类似，用来控制坐标线上两个短线之间的更为短小的线的格式。

第 6 个选项卡如图 6.36 所示。

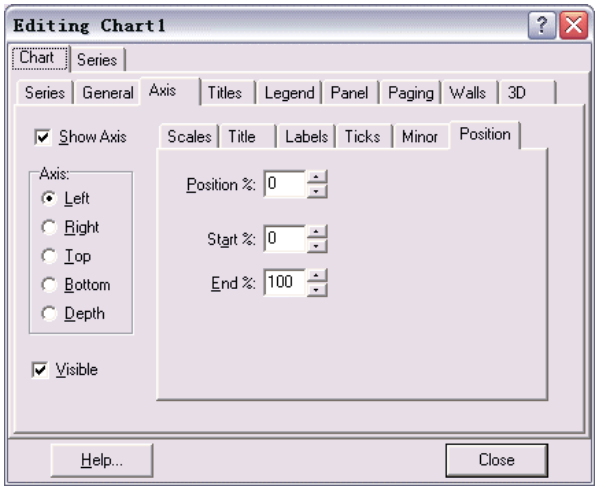

图 6.36　Position 选项卡

这个选项卡控制的是坐标线的位置。这个选项卡中只有三个选项，后面的两个选项可以设置坐标线的起始位置。主要的选项是第一个选项 Position，改变它可以修改坐标线的位置。

到此为止，已经介绍了如何在设计期设置 TChart 控件的坐标线。下面我们就来介绍如何在运行期设置这些属性。程序代码如下：

```
procedure TFrmControl3.FormCreate(Sender: TObject);
begin
series1.FillSampleValues(40);
end;
procedure TFrmControl3.Button1Click(Sender: TObject);
begin
if Timer1.Enabled = False then
begin
    Button1.Caption := '停止移动';
    Timer1.Enabled := True;
end
else
begin
    Button1.Caption := '移动坐标';
    Timer1.Enabled := False;
end;
end;
procedure TFrmControl3.Timer1Timer(Sender: TObject);
begin
     Pos := (Pos+2) mod 100;
     chart1.BottomAxis.PositionPercent := pos;
     chart1.LeftAxis.PositionPercent := pos;
end;
procedure TFrmControl3.CheckBox1Click(Sender: TObject);
begin
chart1.LeftAxis.Logarithmic := checkbox1.Checked ;
end;
procedure TFrmControl3.CheckBox2Click(Sender: TObject);
begin
chart1.LeftAxis.Inverted := checkbox2.Checked;
end;
procedure TFrmControl3.CheckBox3Click(Sender: TObject);
begin
chart1.LeftAxis.Visible := checkbox3.Checked ;
end;
```

上面的程序并不复杂，需要说明的是在 Timer 控件的 OnTimer 事件中，通过对一个变量

进行操作，来控制坐标线的移动。程序运行结果如图 6.37 所示。

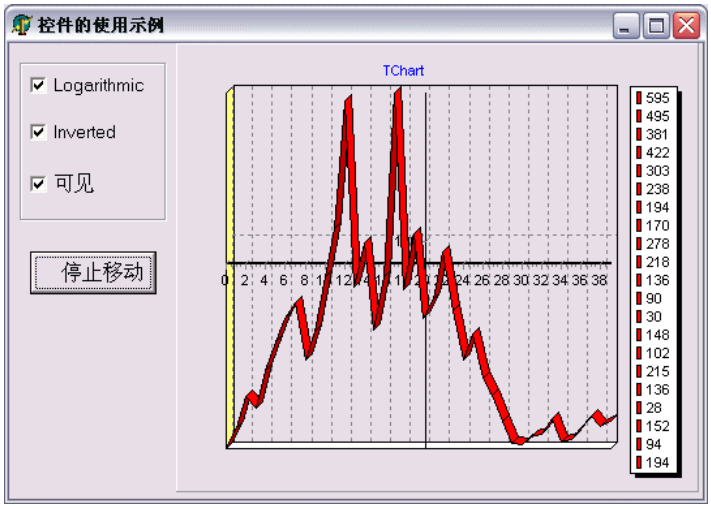

图 6.37　在程序中控制坐标线

上面的内容中比较详细地介绍了 TChart 选项对话框中的前三个选项卡中的内容设置。下面简单地介绍一下其他几个选项卡中的内容。

在 Titles 选项卡中，可以设置 TChart 控件的标题和注脚格式。利用 Visible 属性可以控制标题和注脚的可见性。

在 Legend 选项卡中，是关于设置 TChart 控件的图例选项。利用这个选项卡中的选项设置图例的位置和字体，以及显示在图例中的内容。

由于 TChart 控件的基本框架控件是 TPanel 控件，所以在 Editing Chart 对话框中也提供了一个 Panel 选项卡。

Paging 选项卡可以控制翻页，Walls 选项卡可以设置 TChart 的墙的格式，3D 选项卡中可以设置 TChart 是否进行 3D 显示，或者设置关于 3D 的一些格式。

关于在运行期如何控制这些属性，建议读者仔细研究 Delphi 6.0 提供的一个演示程序：TeeChart.dpr。这个程序中几乎使用到了 TChart 控件的全部常见属性和方法。由于程序比较长，包含的窗体也比较多，这里就不详细介绍了。

6.3.4　在运行期修改 Series 的数据

在上面的程序中，我们通常是用 FillSampleValue 方法填充 Series 的数据。在大多数情况下，需要在运行时修改 Series 的数据。下面就对如何处理这个问题进行详细的介绍。

如果在程序中只是需要对单个的 Series 进行数值改变，则可以使用各个 Series 对象。例如在下面的程序中，当我们利用上面的方法为 TChart 控件添加一个 Series 对象时，可以在

TForm 的说明部分看到这个对象的说明，如图 6.38 所示。

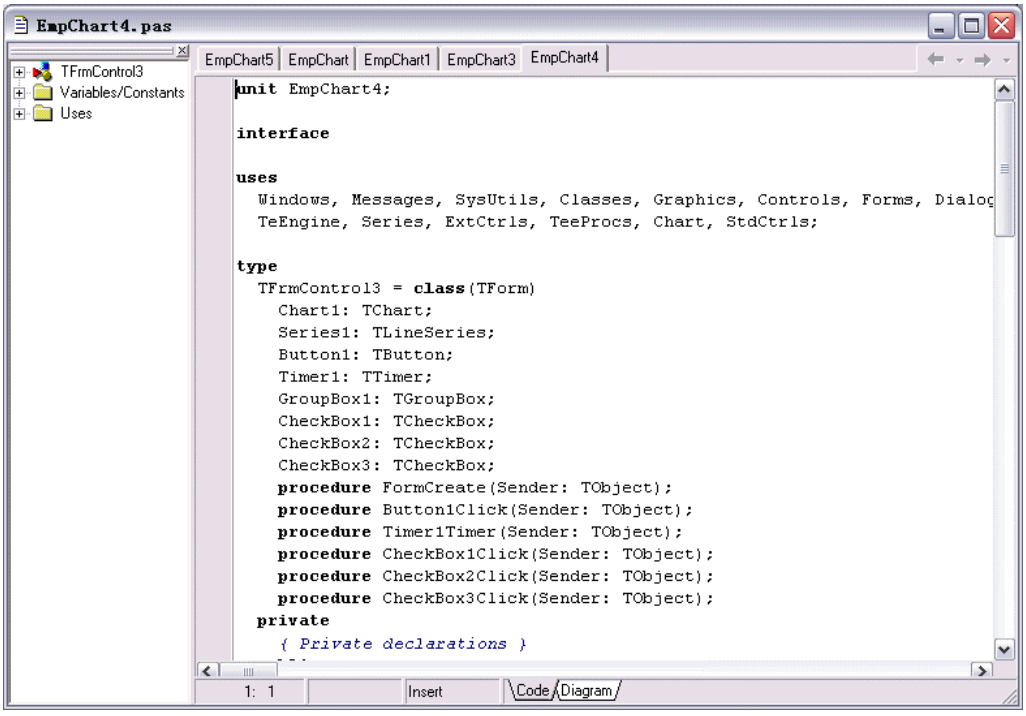

图 6.38　TForm 说明中的 Series

例如，可以使用下面的程序代码在 Series1 中添加数据。

```
With Series1 do
Begin
   Add( 40, 'Pencil' , clRed ) ;
   Add( 60, 'Paper', clBlue ) ;
   Add( 30, 'Ribbon', clGreen ) ;
end;
```

如果需要对 TChart 控件中的所有的 Series 进行操作，则可以使用 TChart 控件的 SeriesList 控件，例如可以使用下面的代码：

```
procedure TFrmControl4.FormCreate(Sender: TObject);
var i:integer;
begin
    for i:=0 to chart1.SeriesList.Count-1 do
        chart1.SeriesList[i].Add(i*i,format('%f',[i*i]),0);
end;
```

对于饼图来说，还可以利用下面的方法使 Series 进行旋转，这可以取得一定的动画效果，

在许多程序中使用这一技术会大大增强程序的图形效果。

```
series1.Rotate(5);
```

这个方法的参数是一个角度，例如在上面的语句中，每次旋转 5 度。

6.4 本 章 小 结

本章介绍了一些工具控件、图形控件和图标控件的使用方法。在上面介绍的控件中，尤其要关注文件系统控件和图表控件的使用。在使用文件系统控件时，要注意相互的关联关系以及文件路径的取得。在使用图表控件时，要注意每条曲线的数据更新和控制。

我们曾经介绍过，使用 Delphi 编程就像是搭积木，每个控件就是一个积木块，那么能够搭出什么样的应用程序，关键就看编程者本身对编程思想的掌握了。

第 7 章　菜单、工具栏和标准动作

涉及到应用程序时，除了利用前面所介绍的控件以及其方法，编写一个个的功能外，还需要做的另外一个工作就是把这些功能向用户提供出来。TButton 等控件的 OnClick 事件当然是提供这些功能的一个好方法，但是在 Windows 操作系统中，人们更习惯用菜单和工具栏的形式来使用应用程序。实际上从 Delphi 5.0 开始，Delphi 中的菜单设计功能已经很完善了，但是现在的 Delphi 6.0 又对菜单、工具栏的设计提供了更多的补充。利用它们我们设计一个具有标准菜单栏和工具栏的应用程序界面简直不费吹灰之力。

Delphi 现在开始把动作（Action）作为一个对象来处理。所谓动作就是集成了一段代码，可以完成一个独立的功能的方法或者函数。Delphi 除了提供了许多标准的动作之外，还允许我们定义自己的动作列表。利用它们，我们可以很容易地实现文档编辑类应用程序的开发。

在布置前面介绍的这些控件时，通常需要把它们按照功能或者涉及的方面分成几个组，然后用一些框架或者其他的控件把它们分开。这些用于包含控件的控件就是通常称为容器控件的控件。除了作为控件容器基本功能之外，这些控件还具有其他一些功能，例如窗口的停靠等。

在本章中，将主要介绍以下内容：

- 菜单
- 工具栏
- 停靠窗口
- 动作列表
- 应用程序事件对象

7.1　菜　　单

菜单是 Windows 应用程序中最常见的组件之一，是用户用来对应用程序进行操作的主要途径之一。

在 Delphi 中，菜单分为两类，一是主菜单，就是我们常说的下拉菜单；另一类是弹出式菜单，也就是我们常说的快捷菜单。本章将主要介绍这些菜单的创建和使用。

7.1.1 使用菜单设计器

主菜单为应用程序提供了一个下拉菜单，它是由控件选项板上的 TMenu 控件完成的。如果要为应用程序添加主菜单，则应该首先在 Form 上放置一个 TMenu 控件。在控件选项板上双击 TMenu 控件或者先单击该控件，然后在 Form 上右击鼠标。

但是上面的操作并不能为程序添加任何菜单项目。如果要为菜单添加项目，必须使用 Delphi 提供的菜单设计器。这个设计器也是一个所见即所得的设计窗口。要调出这个菜单设计器，可以双击 Form 上的 TMenu 控件，或者在属性编辑器上单击 Items 旁边的省略号按钮，该菜单设计器如图 7.1 所示。

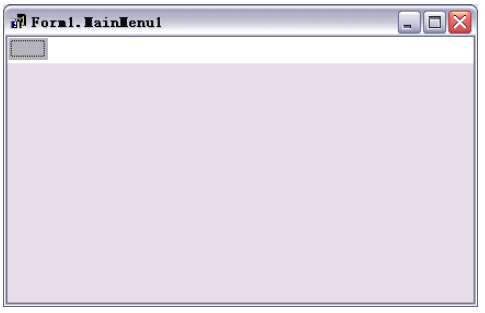

图 7.1 菜单设计器

利用这个菜单设计器可以定义主菜单的各个菜单以及每个菜单的子菜单。下面我们就演示如何创建一个菜单。

打开菜单设计器，可以发现属性编辑器中也有了相应的变化，如图 7.2 所示。

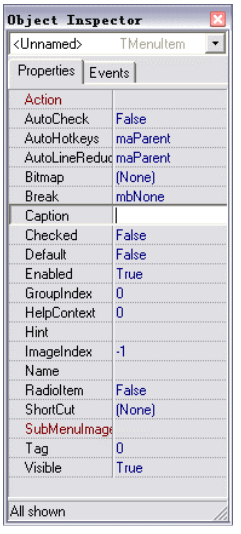

图 7.2 打开菜单设计器后的属性编辑器

我们需要说明的是，在菜单中的每一项都是一个 TMenuItem 对象，所以它具有自己的属性。关于它们的属性设置我们将在后面介绍，这里主要先使用它的 Caption 属性和 Name 属性。

当第一次打开菜单设计器时，在菜单设计器中有一个蓝色的区域，这个区域就是即将创建的第一个 TMenuItem 显示的区域。在属性编辑器中找到 Caption 属性，然后输入需要的菜单的标题，例如可以输入“ &File ”，这里使用的“ & ”字符是用来定义快捷字符的。也就是说，当程序运行时，只要按下 Alt+F 键就相当于单击了 File 菜单。这里输入的标题就是程序在运行时显示的菜单名称，而 TMenuItem 对象的名称是由 Name 属性来指定的。然后修改 Name 属性，例如这里可以命

名为“ File1 ”。

注意：

在定义菜单的名称时，一方面要尽量反映菜单的意义，另外还要注意不要违反我们前面介绍的命名规则。

这里创建的 File 菜单是一级菜单，标示区域会自动移动到下一个一级菜单，可以利用同样的方法定义该菜单。然后单击 File 菜单，此时会在该菜单的下面出现二级菜单，如图 7.3 所示。

图 7.3　设计二级菜单

同样修改二级子菜单的 Caption 属性和 Name 属性。利用同样的方法可以创建各个菜单的二级菜单。

有的时候，需要为二级子菜单创建子菜单。利用这个菜单设计器也可以为各级菜单设计子菜单。首先选中要创建子菜单的菜单项目，然后右击鼠标，并从弹出的快捷菜单中选择 Create Submenu。在图 7.4 中，便设计了一个包含了子菜单的菜单。

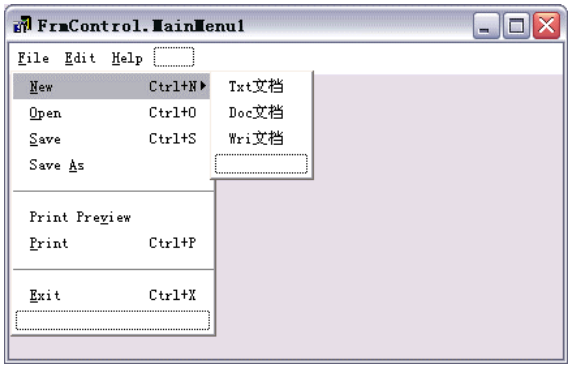

图 7.4　包含子菜单的菜单

在许多程序的菜单中，都有一些灰色的横线把一个菜单分成几个部分，这样用户可以比较容易地找到自己需要的命令。这在菜单中包含有许多命令的时候尤其有用。我们也可以为

自己的菜单添加这样的分隔符。例如对于上面的菜单，我们要在 Print Preview 菜单项目的前面和 Exit 菜单项目的前面添加一个分隔符。实际上这个操作过程并不复杂，只要在这些项目前面插入一个菜单项目（这可以利用在菜单项目上的快捷菜单实现），然后把这个项目的标题修改为“-”，那么这个菜单项目就变成了一个分隔符，从这里也可以看出，分隔符也是一个菜单项目，只不过它不能响应单击等事件罢了，如图 7.4 所示。

也可以利用 TMenuItem 的 Break 属性，这个属性可以控制一个菜单中的各个菜单项是否排列成多列。这个属性有三个值，含义分别如下。

- mbNone：这是该属性的默认值，也是我们最熟悉的菜单形式，不会把一个菜单中的项目分成两列或者多列。
- mbBarBreak：菜单会在设置成该值的菜单项前面分成两列，并且在两列之间有一个分隔的竖条把两列分隔开来。
- mbBreak：这个值只是把菜单在设置成该值的菜单项前面分成两列，这两列菜单只是用一段空白分隔开来，

例如在图 7.5 所示的菜单中，我们就设计了一个包含了上面所讲的后两种属性值的菜单。

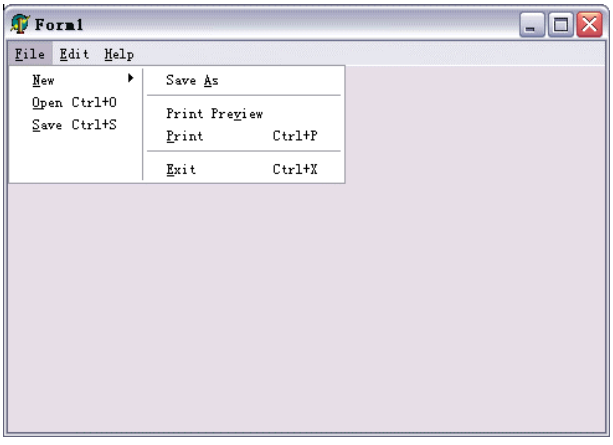

图 7.5　把菜单分成多列

使用 TMenuItem 对象的 Checked 属性可以在用户单击菜单项目时使菜单项的左边出现一个对号标记或者清除这个对号标记。RadioItem 具有类似的功能。

利用 TMenuItem 对象的 ShortCut 属性，可以指定菜单项的快捷键。单击该属性并从下拉列表中选择需要的快捷键。在图 7.5 所示的菜单中就指定了快捷键。

Delphi 也为菜单的设计提供了一些模板。在菜单设计器中右击鼠标，此时会显示一个快捷菜单。在这个菜单的后半部分包含的就是关于模板操作的一些命令。

利用这些命令，既可以把设计好的菜单存储为一个菜单模板，也可以利用已经定义好的菜单模板方便快捷地设计菜单。选择 Insert form Template 命令，会出现如图 7.6 所示的 Insert Template 对话框。

图 7.6　Insert Template 对话框

从图中选择需要的模板，然后单击 OK 按钮，便可以插入一个菜单了。如果对菜单设计器上的菜单排列位置不满意，可以使用鼠标把菜单拖到合适的位置上。

7.1.2　在菜单上使用图形

从 Delphi 的菜单中可以看到，在菜单项目的旁边都有一个图形。利用 TMenu 和 TMenuItem 的一些属性也可以创建带有图标的菜单。首先我们可以使用前面介绍过的 TImageList 控件来为菜单提供图形。

例如，我们在一个 TImageList 控件中载入下面的图形，如图 7.7 所示。

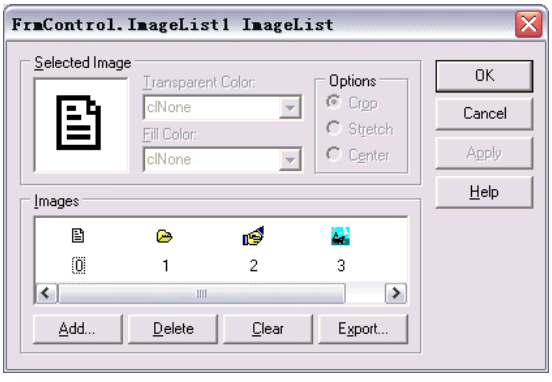

图 7.7　TImageList 控件中的图形

然后可以把这个 TImageList 控件指定给菜单的某个项目的 SubMenuImage 属性。那么这个菜单项目的子菜单就可以使用该 TImageList 控件中的图形了。在使用这些图形的时候，需要指定子菜单项目的 ImageIndex 属性。这个属性的默认值是“–1”，表示不使用图形。如果指定一个非负数，那么这个菜单项目将引用 SubMenuImage 属性中指定的 TImageList 控件中索引号为 ImageIndex 的图形。例如在图 7.8 所示的菜单中，我们便使用了图形，其中也有几个菜单项目没有使用图形。

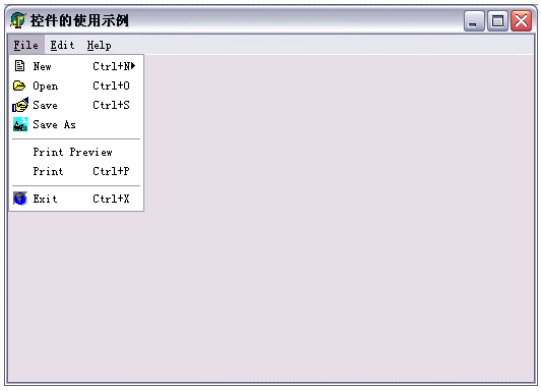

图 7.8 使用图形的菜单

当然也可以使用菜单项目的 Bitmap 属性来为个别的菜单项目指定图形。但是，在指定这个属性的时候，如果没有使用上面介绍的 SubMenuImage 属性和 ImageIndex 属性，则这个图形是有效的。如果指定了上面的两个属性，那么不管在这个 Bitmap 属性中指定了什么样的图形都是无效的，Delphi 首先按照上面的两个属性来引用菜单项目的图形。

7.1.3 合并菜单

如果一个应用程序中有多个窗体，而在这多个窗体中又有多个窗体具有菜单。那么当这些窗体同时显示时就可能需要把窗体上的菜单合并到一起。这就是我们在这里要介绍的合并菜单。

对于 MDI 应用程序来说，子窗体上的菜单是自动和主窗体上的菜单进行合并的，所以我们就不用对它进行详细介绍了。对于 SDI 应用程序来说，需要对 TMenu 控件进行必要的操作，才能实现菜单的合并功能。

下面首先介绍实现自动合并的方法，然后介绍在程序中控制两个窗体上的菜单的合并情况。

新建一个窗体，然后在窗体上新建一个菜单，如图 7.9 所示。

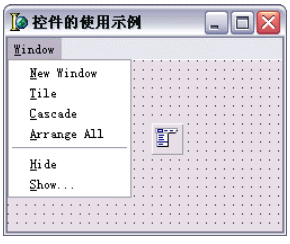

图 7.9 包含菜单的又一个窗体

选中这个窗体上的菜单控件，然后把属性编辑器中的 AutoMerge 属性设置成 True。这样在显示这个窗体时便会把这个窗体的菜单和上一个窗体的菜单合并起来。

但是我们还必须设置各个一级菜单项目的 GroupIndex 属性。对于上面的操作来说，如果没有对该属性进行设置，那么进行菜单合并的时候，不是把出现的菜单合并起来，而是用第二个窗体的菜单替换掉了第一个窗体的菜单。利用 GroupIndex 属性还可以控制菜单合并的时候菜单的顺序。例如我们把第一个窗体中的 File 和 Edit 菜单的 GroupIndex 属性设置成“0”，而把 Help 菜单的 GroupIndex 属性设置成“2”，然后把第二个窗体中的 Window 菜单的 GroupIndex 属

性设置成“ 1 ”，那么当我们显示第二个窗体时，菜单就会按照 GroupIndex 属性的值，从小到大进行合并，如图 7.10 所示。

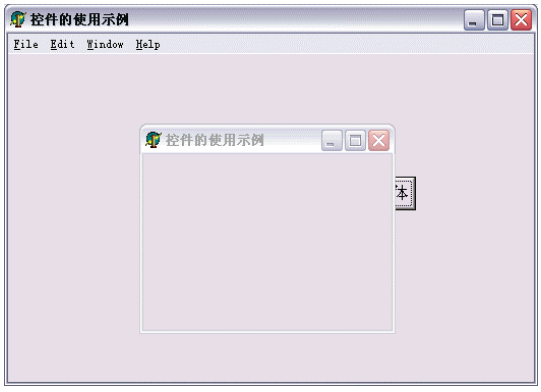

图 7.10　合并的菜单

在处理合并菜单时要注意，尽量不要使用重名的菜单，这样可能会给用户带来误解。

在运行期我们也可以合并菜单或者解除菜单的合并。这仍然需要设置菜单的 GroupIndex 属性，然后调用下面的两个方法：

```
procedure Unmerge(Menu: TMainMenu);
procedure Merge(Menu: TMainMenu);
```

例如在下面的程序中，通过两个按钮，便可以控制上面的示例中介绍的两个窗体上的菜单的合并与否。

```
procedure TFrmControl.Button2Click(Sender: TObject);
begin
    MainMenu1.Merge(FrmControl1.MainMenu1);
end;

procedure TFrmControl.Button3Click(Sender: TObject);
begin
    MainMenu1.Unmerge (frmControl1.mainmenu1);
end;
```

在上面的程序中，第二个窗体显示与否并不影响程序的运行。但是这个窗体必须是已经创建的，对于这里的程序来说，这第二个窗体必须是自动创建的。从上面的工程文件的代码中可以看出，应用程序在运行之前已经创建了这两个窗体，否则上面的程序就会出错。

7.1.4　响应菜单的命令

在设计了上面的形式上的菜单之后，必须要添加菜单的事件。否则上面的菜单不能完成

任何事情。为菜单项目添加事件的方法和为普通控件添加事件并没有任何不同。单击菜单中的某个项目就可以进入到这个项目的默认事件框架中。然后根据自己的需要编写程序就可以了。

例如在下面的程序中，我们设计了一个用来改变主窗口区域中的 TShape 控件的形状、颜色等格式的应用程序，这些格式的改变都是通过菜单命令来完成的。程序代码如下：

```
procedure TMenuForm.DisplayPanelResize(Sender: TObject);
begin
  DemoShape.Top := SHAPE_OFFSET;
  DemoShape.Left := DemoShape.Top;
  DemoShape.Height := DisplayPanel.Height - 2 * SHAPE_OFFSET;
  DemoShape.Width := DisplayPanel.Width - 2 * SHAPE_OFFSET;
end;
procedure TMenuForm.Exit1Click(Sender: TObject);
begin
  Close;
end;
function TMenuForm.RandomColor;
var
  red, green, blue: Byte;
begin
  red := Random(255);
  green := Random(255);
  blue := Random(255);
  Result := red or (green shl 8) or (blue shl 16);
end;
procedure TMenuForm.Randomize1Click(Sender: TObject);
begin
  DemoShape.Pen.Color := RandomColor;
end;
procedure TMenuForm.Select1Click(Sender: TObject);
begin
  try
    SolidColorDialog.Color := DemoShape.Pen.Color;
    if SolidColorDialog.Execute then
      DemoShape.Pen.Color := SolidColorDialog.Color;
  except
    ShowMessage('Color selection dialog failed to load');
  end;
end;
procedure TMenuForm.Randomize2Click(Sender: TObject);
begin
  DemoShape.Brush.Color := RandomColor;
```

```
end;
procedure TMenuForm.Select2Click(Sender: TObject);
begin
  AnyColorDialog.Color := DemoShape.Brush.Color;
  try
    if AnyColorDialog.Execute then
      DemoShape.Brush.Color := AnyColorDialog.Color;
  except
    ShowMessage('Color selection dialog failed to load.');
  end;
end;
procedure ToggleCheck(Sender: TObject);
var
  Item: TMenuItem;
begin
  Item := Sender as TMenuItem;
  Item.Checked := not Item.Checked;
end;
procedure SetCheck(Sender: TObject);
var
  Item: TMenuItem;
begin
  Item := Sender as TMenuItem;
  Item.Checked := True;
end;
procedure TMenuForm.AlterShape(shape: TShapeType; roundable: Boolean);
begin
  Self.Roundable := roundable;
  DemoShape.Shape := shape;
end;
procedure TMenuForm.Circle1Click(Sender: TObject);
begin
  SetCheck(Sender);
  AlterShape(stCircle, False);
end;
procedure TMenuForm.Ellipse1Click(Sender: TObject);
begin
  SetCheck(Sender);
  AlterShape(stEllipse, False);
end;
procedure TMenuForm.Rectangle1Click(Sender: TObject);
begin
```

```
  SetCheck(Sender);
  if RoundedShape1.Checked then
    AlterShape(stRoundRect, True)
  else
    AlterShape(stRectangle, True);
end;
procedure TMenuForm.Square1Click(Sender: TObject);
begin
  SetCheck(Sender);
  if RoundedShape1.Checked then
    AlterShape(stRoundSquare, True)
  else
    AlterShape(stSquare, True);
end;
procedure TMenuForm.ThickOutline1Click(Sender: TObject);
begin
  ToggleCheck(Sender);
  DemoShape.Pen.Width := 11 - DemoShape.Pen.Width;
end;
procedure TMenuForm.RoundedShape1Click(Sender: TObject);
begin
  ToggleCheck(Sender);
  case DemoShape.Shape of
    stRectangle:     DemoShape.Shape := stRoundRect;
    stRoundRect:     DemoShape.Shape := stRectangle;
    stSquare:        DemoShape.Shape := stRoundSquare;
    stRoundSquare: DemoShape.Shape := stSquare;
  end;
end;
procedure TMenuForm.About1Click(Sender: TObject);
var
  AboutBox: TAboutBox;
begin
  AboutBox := TAboutBox.Create(Self);
  try
    AboutBox.ShowModal;
  finally
    AboutBox.Free;
  end;
end;
procedure TMenuForm.Random1Click(Sender: TObject);
var
```

```
    newshape: TShapeType;
begin
    repeat newshape := TShapeType(Random(6)) until
        newshape <> DemoShape.Shape;
    case newshape of
        stEllipse: Ellipse1Click(Ellipse1);
        stCircle: Circle1Click(Circle1);
        stRectangle, stRoundRect:
            begin
                RoundedShape1.Checked := newshape = stRoundRect;
                Rectangle1Click(Rectangle1);
            end;
        stSquare, stRoundSquare:
            begin
                RoundedShape1.Checked := newshape = stRoundSquare;
                Square1Click(Square1);
            end;
        end;
end;
procedure TMenuForm.RandomizeColors1Click(Sender: TObject);
begin
    DemoShape.Brush.Color := RandomColor;
    DemoShape.Pen.Color := RandomColor;
end;
procedure TMenuForm.InvertColors1Click(Sender: TObject);
var
    i : Integer;
begin
    i := Integer(DemoShape.Brush.Color) xor $FFFFFF;
    DemoShape.Brush.Color := TColor(i);
end;
```

程序的运行结果如图 7.11 所示。利用菜单可以修改 TShape 的形状、颜色等格式。

7.1.5　在运行期控制菜单

在设计期控制菜单的选项是比较容易的，这正是 Delphi 所见即所得功能的功劳，但是，关于菜单在运行期的控制则具有一定的复杂性。首先这会涉及许多 TMenu 和 TMenuItem 对象的一些方法，包括菜单项目的添加、删除、隐藏和禁用等。

例如下面的程序就演示了如何在运行时为一个根本没有菜单的窗体添加菜单、子菜单和其他的一些格式。

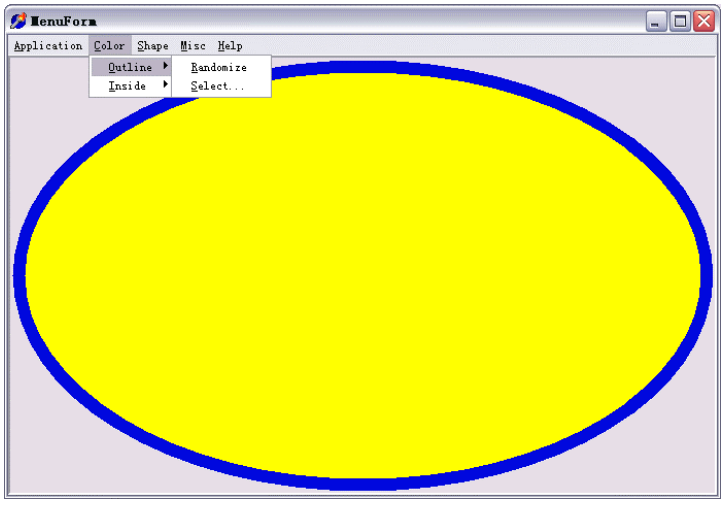

图 7.11　**响应菜单的命令**

```
procedure TForm1.Button1Click(Sender: TObject);
begin
   MyMainMenu:= TMainMenu.Create(Self); //创建一个菜单对象
   Button4.Enabled := true;
   ShowMessage('MainMenu created but no items' + #13+
                        'are added so it does not show.');
   Button1.Enabled := False;
end;

procedure TForm1.Button4Click(Sender: TObject);
var
   MyItem: array[0..2] of TMenuItem;
   i: Integer;
begin
   for i := 0 to 2 do begin
      MyItem[i] := TMenuItem.Create(Self);//创建菜单项目
      MyItem[i].Caption := 'New item ' + IntToStr(i);//设置菜单项目的标题
      MyMainMenu.Items.Add(MyItem[i]);
      end;
   Button4.Enabled := False;
   Button5.Enabled := true;
end;
procedure TForm1.Button5Click(Sender: TObject);
var
   i: Integer;
begin
```

```
    for i := 0 to 3 do begin
      MySubItems[i] := TMenuItem.Create(Self);//创建子菜单
      MySubItems[i].Caption := 'New item ' + IntToStr(i);
      MyMainMenu.Items[0].Add(MySubItems[i]);
    end;
    Button6.Enabled := true;
    Button8.Enabled := true;
    Button5.Enabled := False;
end;
procedure TForm1.Button6Click(Sender: TObject);
begin
    MySubItems[3].Break := mbBarBreak;//设置 Break 属性
end;
procedure TForm1.Button8Click(Sender: TObject);
begin
    MySubItems[2].Caption :='-';//设置分隔符
end;
```

程序的运行结果如图 7.12 **所示。**

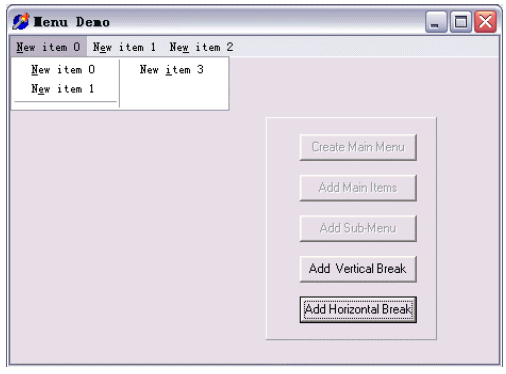

图 7.12　**在运行期创建菜单及菜单项目**

7.1.6　快捷菜单

一个界面友好的应用程序应当支持鼠标右键和快捷菜单，大家可以从 Windows **以及其他应用程序的使用中体会到快捷菜单的好处。在这一节中我们将介绍如何建立快捷菜单，怎样弹出快捷菜单，怎样有选择地显示菜单项。**

要建立快捷菜单，首先要把 TPopupMenu **控件添加到** Form **上，然后修改** Form **的** PopupMenu **属性指定** TPopupMenu **的控件名称。**

和 TMainMenu **控件一样，**TPopupMenu **控件也提供了菜单设计器，用于在设计期间建立**

菜单的结构，不过，由于快捷菜单的菜单项往往是动态变化的，因此，更多的工作是在运行期间操作菜单。

与 TMainMenu 控件一样，TPopupMenu 控件也是从 TMenu 对象继承下来的。快捷菜单的每个项目都是一个 TMenuItem 对象，我们仍然可以通过 Items 属性来访问菜单中的每一个项目。

TPopupMenu 控件有一个 Alignment 属性，这个属性同其他控件的 Alignment 属性不同，其他控件的 Alignment 属性的作用是控制本身在容器中的位置的，而 TPopupMenu 控件的 Alignment 属性控制的是当用户按下鼠标右键时快捷菜单的显示位置。在默认的情况下，快捷菜单会出现在鼠标的右下方，也就是说鼠标出现在快捷菜单的左上方。

TPopupMenu 控件还有一个 AutoPopup 属性，如果把这个属性设置成“ True ”，那么当用户在指定了 PopupMenu 属性的对象上右击鼠标时，会自动弹出快捷菜单。否则需要调用 Popup 方法才能显示出快捷菜单。

前面曾经提到，在实际的应用程序中，通常会根据用户右击鼠标时所单击的对象不同显示不同的快捷菜单。那么如何辨认单击的对象呢？这需要使用 TPopupMenu 的 PopupComponent 属性。通过这个属性，我们可以知道用户是在什么控件上右击鼠标，那么就可以根据需要进行不同的快捷菜单的显示。同样，如果在程序中显式地调用 Popup 方法来弹出快捷菜单，可以在弹出快捷菜单之前把 PopupComponent 属性设置成需要的控件。

7.2 工 具 栏

工具栏的发展日益复杂，从最开始的仅仅是一个个简单按钮的罗列，到现在的丰富多彩的工具栏，可以说工具栏的使用已经大有替代菜单的势头了。

在 Delphi 中，开始使用多种方法创建工具栏。但是通常来说，一般不会使用一种控件来建立工具栏，因为这样建立的工具栏具有一定的局限性，对于已经熟悉了现在的工具栏风格的用户来说，单调的工具栏可能会影响程序的效果。下面我们就介绍几种常用的创建工具栏方法。

7.2.1 使用 TPanel 和 TSpeedButton 控件创建工具栏

许多人都习惯于使用 TPanel 和 TSpeedButton 控件来创建工具栏。首先把一个 TPanel 控件放置的到窗体上，然后把它的 Align 属性设置成“ alTop ”，那么工具栏将自动移动到 Form 剩余空间的底部（所谓剩余空间是说可能在上面已经存在着一个菜单），其水平尺寸将随着 Form 的变化而自动变化，并总是保持充满整个 Form。然后逐个地把快捷按钮（TSpeedButton）控件添加到 TPanle 控件上，如图 7.13 所示。

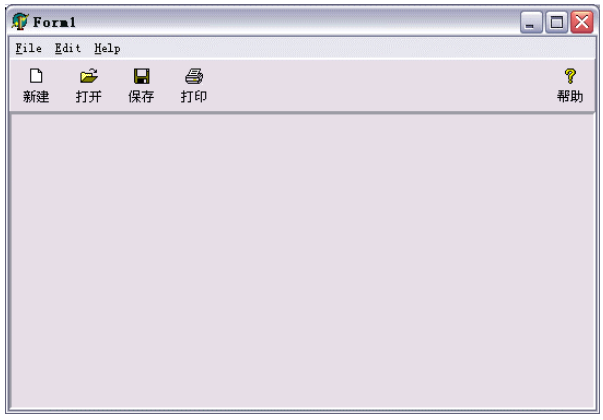

图 7.13　利用 TPanel 和 TSpeedButton 控件创建的工具栏

要使得工具栏中的按钮响应对应的命令，则可以逐个地设置每个快捷按钮的 OnClick 事件。

如果要建立多个工具栏，则可以把多个 TPanel 控件添加到 Form 上，并把它们的 Align 属性都设置为“ alTop ”，这样工具栏将在窗体上从上到下依次排列。

7.2.2　使用 TToolBar 和 TCoolBar 控件创建工具栏

从 Delphi 3.0 开始，Delphi 就提供了一个 TToolBar 控件，利用它创建工具栏十分方便，而且还有许多其他功能。但是现在人们在创建工具栏时一般不会只用 TToolBar 控件，而是要结合其他的一些容器控件，以便使工具栏具有更为强大的功能。经常使用的容器控件有 TControlBar 控件和 TCoolBar 控件，这两个控件十分类似。

下面我们首先介绍如何使用 TToolBar 控件。这个控件的使用十分简单，首先在窗体或者其他容器中放置一个 TToolBar 控件，然后在该控件上右击鼠标，此时会弹出一个快捷菜单，从这个快捷菜单中选择 New Button 或者 New Separator 命令，Delphi 就会在工具栏上添加一个按钮或者一个分隔符。按钮添加到工具栏上之后，就可以利用鼠标来改变它的位置、尺寸，或者指定它的标签。

TToolBar 控件是一个窗口类控件，它不仅可以放置按钮，而且可以作为其他控件的父控件。也就是说，我们可以把其他的一些控件放置到 TToolBar 控件上。例如在图 7.14 所示的应用程序中，就包含了一个组合框控件和一个文本框控件。

TToolBar 控件可以自动维护工具栏上的按钮尺寸，当改变其中的一个按钮尺寸时，TToolBar 控件将自动调整其他按钮的尺寸，以保证所有按钮的尺寸都是一样的。工具栏上的按钮还可以引用来自 TImageList 控件中的图形列表。首先要用一个 TImageList 控件建立一个图形列表，然后设置 TToolBar 控件的 Images 属性，把它指定到这个图形列表中，图形列表中的所有图片将自动应用到 TToolBar 控件的按钮上。每个按钮都具有不同的索引号，除非手

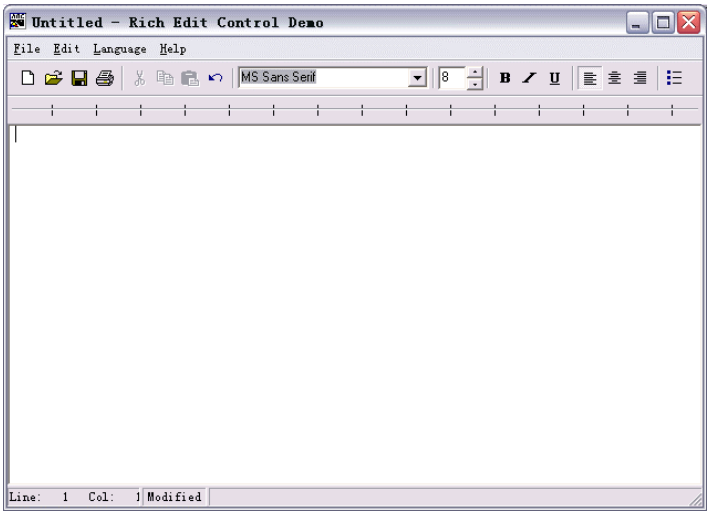

图 7.14 TToolBar 控件中包含其他的控件

动去改变这个索引号。在默认的情况下，所有的按钮都是凸起的，不过也可以像现在的许多应用程序一样，把这些按钮设置成平坦的，只有当用户把鼠标移动到该按钮上时该按钮才会凸起。如果要这样做，只要把 Flat 属性设置成“ True ”就可以了。

现在许多应用程序的工具栏，特别是 Microsoft Internet Explorer 的工具栏，平时按钮上的图像都是黑白的，当我们把鼠标移动到工具栏的按钮上时，按钮上的图像就变成彩色的了。这是一个比较吸引人的功能，利用 TToolBar 控件也可以实现这个功能。我们只要新建一个 TImageList 控件，然后把当鼠标移动到工具栏按钮上时要显示的图像载入到这个图形列表控件中，然后把它指定给 TToolBar 控件的 HotImages 属性就可以了。

TToolBar 控件还有一个比较独特的功能，就是工具栏可以自动绕回。也就是说，当 Form 的水平尺寸小于工具栏的水平尺寸时，工具栏可以自动绕回，在一行上显示不下的按钮将自动移动到下一行中。要实现这样的功能，只要把 Wrappable 属性设置成“ True ”就可以了。

在操作 TToolBar 控件时还有其他一些比较琐碎的属性和方法，下面也一并给予介绍。

如果将 TToolBar 控件的 AutoSize 属性设置成“ True ”，工具栏将自动调整自身的高度以适应工具栏上的按钮和控件。

BorderWidth 属性和一般控件的 BorderWidth 属性不同，它控制的是工具栏的内侧和工具栏按钮之间的距离。如果这个值大于 0，则表示按钮并不是紧贴着工具栏的。

TToolBar 控件的 ButtonCount 属性可以返回工具栏上的按钮数目，也就是 TToolBar 控件的 Buttons 数组属性中的元素个数。通过 Buttons 属性可以访问工具栏上的所有按钮，每个按钮都有自己惟一的索引号，这些按钮的 Parent 属性就是工具栏，它们的高度和宽度都是相同的。

ButtonHeight 属性可以用来获得或者设置工具栏上的按钮高度。如果按钮的 Style 属性被

设置成了 tbsSeparator 或者 tbsDivider，该属性的设置边框功能就无效了。

除了上面提到的 Images 和 HotImages 属性之外，TToolBar 控件还有一个图形列表对象属性：DisabledImages 属性。利用这个属性可以为工具栏上的按钮指定按钮处于禁用状态时按钮上显示的图形。如果没有指定这个属性，那么 Delphi 就用 Images 属性中指定的图片变灰来表示禁用的状态。

EdgeBorders 属性是一组 Boolean 属性，它共有四个，分别控制着 TToolBar 控件的四个边框的显示与否。EdgeINner 和 EdgeOuter 属性与 TPanel 控件的 BevelInner 和 BevelOuter 属性类似，可以使工具栏具有 3D 效果。Indent 属性控制着工具栏上的第一个按钮或者控件与工具栏的左边框距离。List 属性类似于 TSpeedButton 控件的 Layout 属性，控制着工具栏按钮上的文字和图像之间的位置关系，当其为 True 时图像显示在按钮的中央而文字显示在图像的下方，为 False 时图像在左边而文字在右边。

利用 RowCount 属性可以获得当前工具栏被分成了几行。使用 ShowCaption 属性可以确定是否显示按钮上的标签。

和 TMainMenu 控件类似的是，TToolBar 控件上的每一个按钮都是一个对象（TToolButton），该对象也具有自己的属性和方法，通过对这些属性和方法的控制，可以分别控制工具栏上的各个按钮。下面对 TToolButton 对象的各个属性进行介绍。

TToolButton 对象的一个比较重要的属性是 Style 属性，它确定了按钮的风格。Style 属性共有五个预设值，它们的名称和含义如下。

- tbsButton：普通的快捷按钮，上面可以显示图片和文字。
- tbsCheck：该按钮可以保持在按下的位置，通常是把几个按钮分成一组，这时的这组按钮具有单选按钮的作用。
- tbsDropDown：按钮的左边将显示一个向下箭头，单击这个箭头可以弹出一个菜单。这样的工具栏按钮在许多应用程序中都可以看到。
- tbsSeparator：分隔符，可以把工具栏上的按钮分成几个组。
- tbsDivider：类似于 tbsSeparator，形象稍有不同。

上面提到的 tbsCheck 类型的按钮通常是成组使用的。在把工具栏上的按钮分组时，首先要确保分成同一组的按钮必须是相邻的，而且不同组的按钮之间必须用分隔符或者普通按钮隔开，然后把同一组的按钮类型设置成 tbsCheck，并且需要设置这些按钮的 Grouped 属性。在设置了一组按钮之后，还可以设置工具栏按钮的 AllowAllUp 属性。如果这个属性为 True，那么该工具栏上同一组的按钮可以在某个时刻都处于弹起状态，否则就必须有一个按钮处于按下的状态。在修改一个按钮的 AllowAllUp 属性时会同时修改同一组中其他按钮的 AllowAllUp 属性。当按钮处于按下状态时，按钮的 Down 属性就是 True，通过这个 Down 属性我们可以控制或者读取按钮的状态。

上面也提到了工具栏上的按钮可以设置成 tbsDropdown 类型。当处于这个类型的时候，需要为该按钮指定 DropdownMenu 属性。也就是说需要指定当按下这个按钮旁边的箭头时弹

出哪个菜单。这个菜单必须是一个 TPopupMenu，并且最好把这个 TPopupMenu 的 AutoPopup 属性指定为“ True ”，这样当单击箭头的时候会自动弹出这个菜单，否则必须响应按钮的 OnClick 事件菜单弹出快捷菜单。

上面我们也提到了通常不会只使用 TToolBar 控件来建立工具栏，而是把它放置在一个容器控件中。这里我们就先介绍一个建立工具栏常用的容器控件：TCoolBar 控件。

TCoolBar 控件似乎就是专门为工具栏创建的，虽然它可以用于其他的用途，但是人们更习惯于把它作为 TToolBar 控件的容器。在介绍 TCoolBar 控件时，必须同时介绍另外一个对象 TCoolBand，它们之间的关系就像 TToolBar 控件和 TToolButton 对象之间的关系一样。

在设计期把一个 TCoolBar 控件放置到 Form 上（TCoolBar 控件位于 Win32 选项卡上），然后用鼠标右击该控件，或者单击属性编辑器上 Bands 属性旁边的省略号按钮，此时会显示如图 7.15 所示的 Editing Bands 对话框。

图 7.15 Editing Bands 对话框

在这个对话框中，共有四个按钮，第一个按钮可以增加一个段（Band），第二个按钮可以删除一个段，另外两个按钮可以调整段的位置。

在该对话框中选择一个段，就可以在属性编辑器中编辑它的属性，关于这些属性的含义我们将在后面的内容中进行详细介绍。

段本身也是一个容器对象，可以把一个控件放置在段上。一种办法是先把控件放置到窗体上，然后把段的 Control 属性设置为该控件；还有一种办法，就是直接把一个控件放置到 TCoolBar 控件上，Delphi 将自动创建一个段，它的 Control 属性将自动指向这个控件。

也可以在运行期创建段，例如下面的代码就演示了如何创建段：

```
var MyBand: TCoolBand;
CoolBar1.Images := Imagelist1;
MyBand := CoolBar1.Bands.Add;
(MyBand As TCoolBand).Control := Edit1;
```

下面介绍一下 TCoolBar 控件的属性。

Align 属性和其他控件的 Align 属性类似，也是控制着 TCoolBar 控件在窗体上的位置。所不同的是，当把 Align 属性设置成“ alLeft ”、“ alRight ”时，Delphi 将自动把 Vertical 属性设置成“ True ”；当把 Align 属性设置成“ alTop ”、“ alClient ”和“ alBottom ”时，Delphi 会自动把 Vertical 属性设置成 False。Vertical 属性决定了 TCoolBar 控件是否进行垂直显示。需要注意的是，我们改变对齐方式会影响 Vertical 属性的值，但是反过来设置 Vertical 属性的值却不会影响 Align 属性的值。

AutoSize 属性为 True 时，TCoolBar 将根据工具栏上的段的数量和尺寸自动调整自己的尺寸。如果 Vertical 属性为 True，将自动调整工具栏的宽度；反之则自动调整该栏的高度。

Bands 属性是工具栏上所有段的集合，通过这个属性可以访问每一个段，同时 Bands 也

是一个 TTCoolBands 对象，利用这个对象的方法可以在运行期控制段。

Bitmap 属性可以指定一个位图，作为背景图案显示在工具栏上。在默认的情况下这个位图从工具栏的左上角开始显示，并覆盖整个段的客户区。

若将FixedOrder 属性设置成“ True ”，用户仍然可以在运行时拖动段，但是不会改变段在 Bands 属性中的索引号。

若将 FixedSize 属性（在 Vertical 属性设置为 False 的情况下）设置为“ False ”，段的高度将随段的窗口类控件的高度自动调整，如果段上没有窗口类控件，段的高度就随段的文字的高度自动调整。如果把 FixedSize 属性设置成“ True ”，则所有段的高度都是一致的。

TCoolBar 控件也有一个 Images 属性，用于为 TCoolBar 控件上的每一个段指定图像。

ShowText 属性是一个 Boolean 型属性，当设置成“ True ”时将在段上显示一个字符串，字符串的值是段的 Text 属性。如果为 False，那么就只能显示控件。

下面我们再介绍一下 TCoolBand 对象的一些属性和方法。TCoolBand 对象也有一个 Bitmap 属性，它用于指定一个位图作为墙纸显示在段上。利用 Color 属性可以设置段的背景颜色。Control 属性的用法我们在前面已经提到，利用它可以指定一个窗口类控件，并把该控件显示在段上。FixedBackground 属性在 ParentBitmap 属性为 True 的情况下段会把它所在的 TCoolBar 控件的背景图案作为自己的背景图案。如果将该属性设置成“ True ”，那么所有的段都会显示一个背景图案；如果设置成“ False ”，那么每个段将会根据自己的 Bitmap 属性的设定来显示背景图案了。Text 属性设定的是显示在段中的文字。

在介绍了制作工具栏的 TCoolBar 控件和 TToolBar 控件的一些属性之后，就可以制作所需要的工具栏了。我们先介绍一个如何在工具栏上添加其他控件的工具栏，然后制作一个使用 HotImages 属性的类似于 IE 的工具栏。

首先在窗体上放置一个 TCoolBar 控件，然后在 TCoolBar 控件上放置两个 TToolBar 控件。再在窗体上放置两个 TImageList 控件，并为这两个控件载入如图 7.16 和图 7.17 所示的图片。

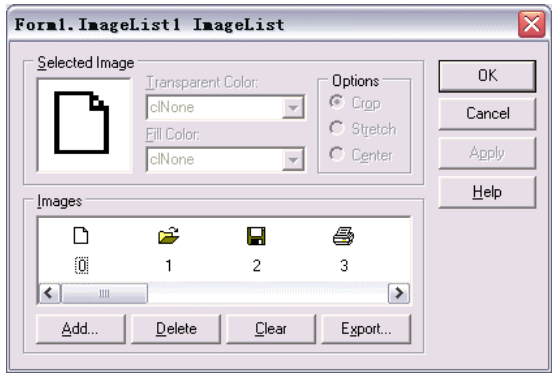

图 7.16　ImageList1 控件中的图片

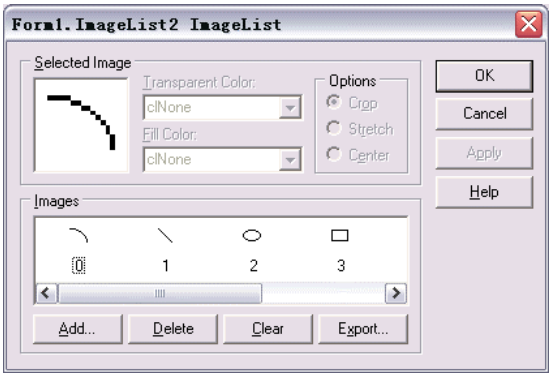

图 7.17 ImageList2 控件中的图片

然后把两个 TToolBar 控件的 Images 属性分别指定为位图 ImageList1 和 ImageList2，在 TToolBar 控件上创建所需要的按钮，并把这两个工具栏的 Flat 属性设置成“ True ”。

从图中可以看出，在第二个工具栏上我们创建了一个类型为 tbsDropdown 的按钮。下面我们将介绍如何为它创建用于指定颜色的弹出菜单。

在第一个工具栏上放置一个 TComboBox 控件，我们在程序中希望用这个控件来显示计算机系统可以使用的字体名称。把这个控件的 Style 属性设置成“ csDropDownList ”，这样做的目的是防止用户在这个组合框中输入文字。把这个组合框的名称修改为“ FontName ”之后，我们再来介绍如何获得计算机系统所安装的字体名称的方法。

可以通过一个 API 函数来获取计算机系统中所安装的字体，这里要使用的 API 函数为 EnumFonts。它可以提取计算机系统所安装的所有字体的信息，并把字体信息传递给我们指定的一个回调函数。通过这个回调函数来安装需要处理的字体信息。这个函数的声明如下：

```
int EnumFonts(
  HDC hdc,
                //设备句柄，我们在这里需要的是计算机设备的句柄。
  LPCTSTR lpFaceName,
                // 指向字体名称的指针。
  FONTENUMPROC lpFontFunc,
                // 指向回调函数的指针。
  LPARAM lParam
                // 用来存储获取的信息的变量
);
```

首先要编写一个用来处理提取的字体信息的回调函数，代码如下：

```
function EnumFontsProc(var LogFont: TLogFont; var TextMetric: TTextMetric;
   FontType: Integer; Data: Pointer): Integer; stdcall;
begin
   TStrings(Data).Add(LogFont.lfFaceName);
```

```
  Result := 1;
end;
```

对于这里要建立的应用程序来说，此处的 Data 参数就是用来存储字体名称的 TComboBox 控件的 Items 属性。然后建立一个提取字体信息的过程，如下所示：

```
procedure TForm1.GetFontNames;
var
  DC: HDC;
begin
  DC := GetDC(0);
  EnumFonts(DC, nil, @EnumFontsProc, Pointer(FontName.Items));
  ReleaseDC(0, DC);
  FontName.Sorted := True;
end;
```

最后在窗体的 OnCreate 事件中输入下面的代码：

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  GetFontNames;
end;
```

此时程序的运行结果如图 7.18 所示。

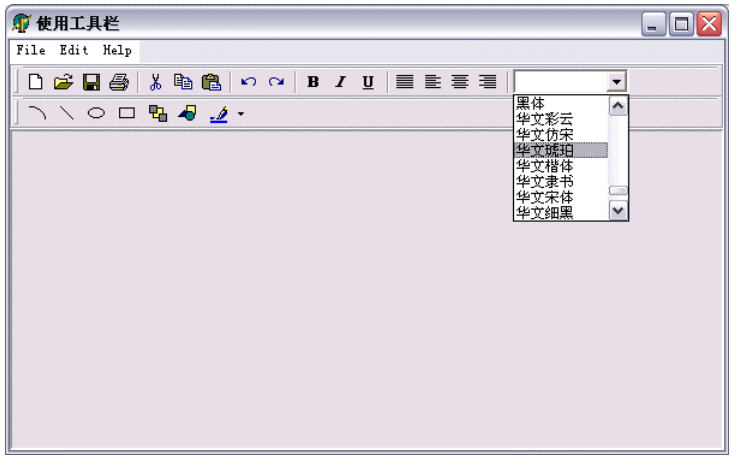

图 7.18　使用组合框的工具栏

下面再来介绍如何为 tbsDropDown 类型的按钮指定一个快捷菜单。首先按照前面介绍的方法创建一个快捷菜单，并为它指定一个 TImageList 控件，然后把类型为 tbsDropDown 的按钮的 DropDownMenu 属性指定给 PopupMenu，此时的程序如图 7.19 所示。

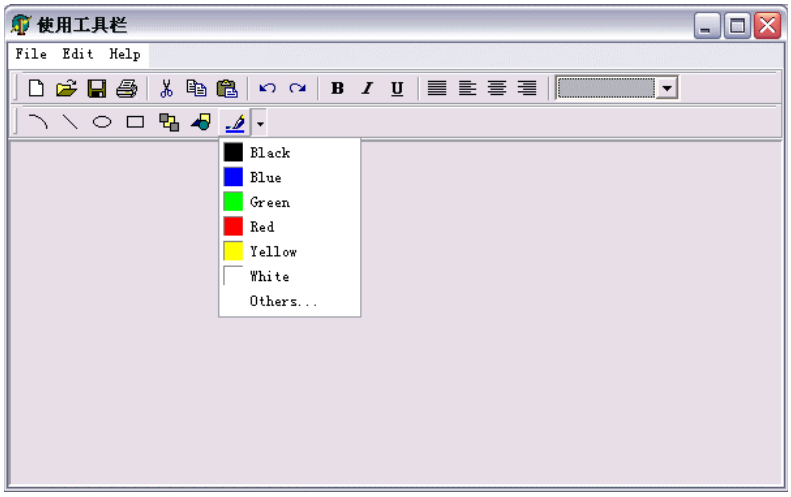

图 7.19 使用快捷菜单的工具栏按钮

7.2.3 利用 TControlBar 控件和 TToolBar 控件创建浮动工具栏

浮动工具栏是许多应用程序工具栏都具有的功能。利用 TControlBar 控件可以很容易地实现这样的浮动功能。在下面的示例中，将不仅介绍如何创建浮动的工具栏，而且还会介绍如何创建浮动的菜单栏。

首先在窗体上放置一个 TControlBar 控件，然后在该控件上放置三个工具栏控件，我们就利用这三个工具栏控件来制作可以浮动的工具栏和菜单栏。把 TControlBar 控件的尺寸调整合适之后把它的 AutoSize 属性设置成“ False ”。利用前面介绍的方法在该工具栏上添加按钮和其他对象。

在窗体上放置一个 TPopupMenu 控件，然后利用菜单编辑器为这个控件创建如图 7.20 所示的菜单。

图中有三个包含子菜单的菜单项，它们都是从模板中直接插入的。在第三个工具栏上创建三个按钮，再把该工具栏的 ShowCaption 属性设置成“ True ”，然后把每个按钮的 MenuItem 属性分别设置成上面设计的“ File1 ”、“ Edit1 ”和“ Help1 ”菜单项。

下面的任务是使这三个工具栏具有浮动功能。为了实现工具栏的浮动功能，必须设置这三个工具栏控件的两个属性：DragKind 和 DragMode。第一个属性要设置成“ dkDock ”，意思是设置成停靠模式；第二个属性要设置成“ dmAutomatic ”，即自动停靠模式，使用这种模式可以避免进行编程的麻烦。在后面介绍停靠这部分内容时，还将专门介绍关于停靠方面的知识。

把一个工具栏拖动成一个浮动的窗口时，我们可能有时会关闭这个浮动窗口，如果没有重新显示这个窗口的选项，则这些选项就无法重新显示，除非关闭应用程序然后重新启动应用程序。所以，必须设置关于如何显示或者关闭这些选项的命令。

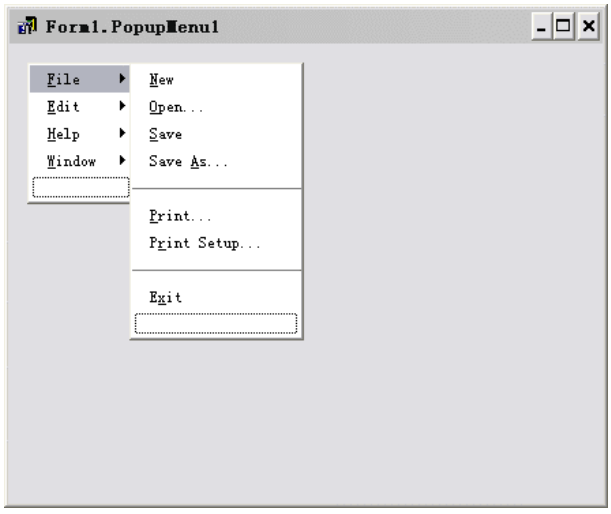

图 7.20　制作的 TPopupMenu 菜单

在窗体上放置另一个 TPopupMenu 控件，然后按照图 7.21 所示的内容设置它的菜单。

图 7.21　设计用于显示工具栏的快捷菜单

在设计完了这样的菜单之后，不要着急关闭这个菜单设计器。在菜单上右击，然后从弹出的快捷菜单中选择 Save As Template，把该菜单保存成一个模板。然后重新打开上面加入的 PopupMenu1 控件的菜单设计器，或者在菜单设计器中右击并从弹出的快捷菜单中选择 Select，然后选择需要的菜单。在适当的位置右击，并选择 Insert From Template，选择上面刚刚保存的模板。这样便创建了一个用于显示工具栏的菜单。现在我们的任务是为它们添加代码，如下所示：

```
procedure TForm1.NewWindow1Click(Sender: TObject);
begin
toolbar3.Visible := not Toolbar3.Visible ;
newWindow1.Checked := toolbar3.Visible;
end;
procedure TForm1.Tile1Click(Sender: TObject);
begin
toolbar1.Visible := not toolbar1.Visible ;
tile1.Checked := toolbar1.Visible;
end;
procedure TForm1.Cascade1Click(Sender: TObject);
begin
toolbar2.Visible := not toolbar2.Visible ;
```

```
cascade1.Checked := toolbar2.Visible;
end;
```

然后在属性编辑器中，把 PopupMenu1 中新插入的菜单的 OnClick 指定成上面对应的事件处理句柄。

选中 TControlBar 控件，然后把它的 PopupMenu 属性设置成“ PopupMenu2 ”，这就是我们后来创建的第二个快捷菜单。

在调整 TControlBar 控件上的工具栏时，由于尺寸的变化可能会引起工具栏的一些变形，例如有的按钮可能被隐藏掉了，解决这个问题的一个简单办法是使用这个工具栏的 OnResize 事件，这个事件发生在工具栏的尺寸要发生改变的时候。为此编写如下所示的代码：

```
procedure TForm1.ToolBar3Resize(Sender: TObject);
begin
toolbar3.Width := controlbar1.ClientWidth;
end;
procedure TForm1.ToolBar2Resize(Sender: TObject);
begin
toolbar2.Width :=142;
end;
procedure TForm1.ToolBar1Resize(Sender: TObject);
begin
toolbar1.Width := 400;
end;
```

此时便完成了关于可浮动的工具栏和菜单的全部设计工作，运行应用程序，结果如图 7.22 所示。

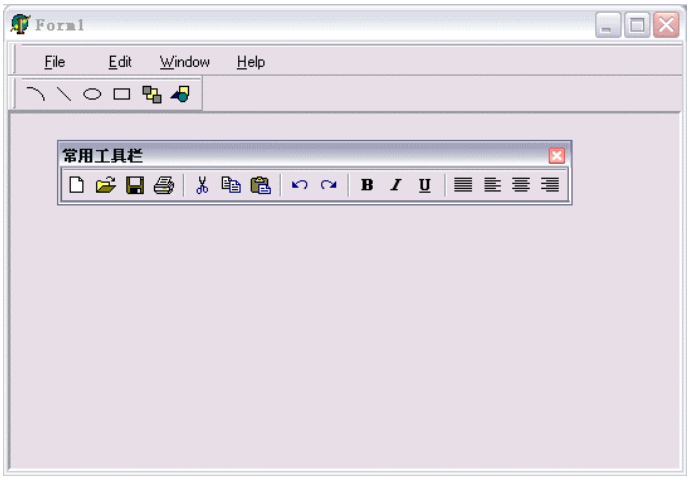

图 7.22　浮动的工具栏和菜单

7.3　停 靠 窗 口

上面介绍的浮动工具栏使用了关于停靠（Dock）的一些属性。在 Delphi 中，自从 Delphi 4.0 开始，就为大多数控件提供了关于设置停靠功能的属性。停靠功能就是在窗体里或者在其他控件顶部停放控件的能力。

停靠的实现很容易。下面介绍了如何将一个标准窗体、TPanel 或 TControlBar 变成一个可以停放其他控件空间中的基本步骤。

(1) 把它的 DockSite 属性设置为“True”。

(2) 如果使用像 TPanel 或是从 Additional 页面新建的 TControlBar 组件这样的控件，则可以把它放置在窗体的边上，直到用户需要把什么东西停放在它上面时再让它出来。要想做到这一点，可以把控件的 AutoSize 属性设置为“True”。

现在，当用户把一个组件拖到窗体的边上时，面板或控件栏就会突然出现，作为停放一个浮动的控件空间。对于用户来说，所做的动作好像只是把控件停靠在窗体的边缘上罢了；用户并不需要知道组件实际上是放置到了一个 TPanel 或 TControlbar 上面。

把一个控件变成可停靠的只需要下面的简单两步。

(1) 把它的 DragKind 属性设为“dkDock”。

(2) 把它的 DragMode 属性设为“dmAutomatic”。

7.3.1　在窗体中停靠控件

在下面将介绍一个名为 SimpleDock 的程序，该程序中演示了一个简单的停靠范例。这个程序在主窗体的两边各有两个 TPanel，每个 TPanel 的 DockSite 属性都设置为“True”。在窗体的底部是一个 TControlBar，它的 AutoSize 属性被设置为“True”。在默认的情况下，TControlBar 的 DockSite 属性应该被设置为“True”。在窗体中间有六个 TShape 控件，它们的 DragKind 和 DragMode 属性都已经为停放设置好了，并且每一个都被设定成明亮的颜色。主窗体的 DockSite 属性也被设置为“True”。

运行程序之后，可以把 TShape 控件拖放到面板、控件栏上，或是拖回到主窗体中。实验一下这个过程，试一试这些控件的各种可能的组合，如图 7.23 所示。

在 Delphi 中，并不是所有的控件都具有 UseDockManager 属性，这个属性只和部分控件有关，例如前面介绍的 TPanel。当这个属性被设置为“True”时，它会导致放到容器控件上的控件变得和容器控件的形状或容器控件上当前可以利用的空间相一致。例如，如果在 SimpleDock 程序中，把一个 TShape 控件放到了一个 TPanel 上，那么停放管理器（Dock manager）就会自动让该控件占满 TPanel 上的可用空间。如果再把第二个控件放到这个 Tpanel

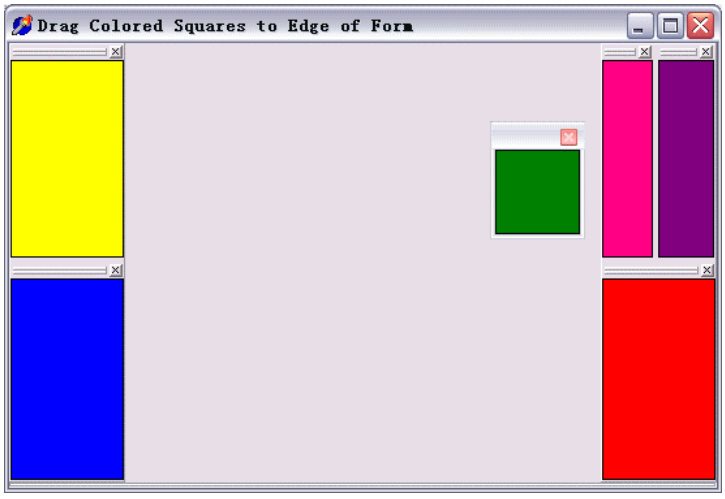

图 7.23 SimpleDock 程序的示例

上，那么它将占据面板的一半空间，而原先的那个控件则占据另一半可用空间。为了实际看到这个过程，可以打开 SimpleDock 程序中一个面板的 UseDockManager 属性，而关闭另一个面板的 UseDockManager 属性。

注意：

在使用 TPanel 等控件的停靠功能时，必须要注意，每个 TPanel 控件都可以脱离窗体而成为一个浮动的窗口，因此必须在程序中有能够显示它们的命令，因为在程序中，用户可能会由于某种原因把这个浮动窗口关闭掉。

7.3.2 在窗体中停靠窗体

下面将通过另外一个程序来介绍如何在窗体中停靠窗体。在下面的内容中，将建立一个带有三个窗体的应用程序。第一个窗体上放置了一个 TPageControl 控件，它的 DockSite 属性被设置为“True”。也就是说将用它来作为停靠的容器。

这个程序还有另外两个窗体，这两个窗体的 dkKind 和 dkMode 属性都被设置成使它们可停靠的状态。在运行期间，可以同时启动所有这三个窗体，并把后面两个停放到第一个窗体的 TPageControl 控件中去。

基本的属性设计就是这样的。为了演示停靠后的窗体功能是否受到影响，这里还在后两个窗体中放置和生成了其他的一些控件。下面我们来分析一下这三个窗体中的代码。先看第一个窗体的代码，如下所示：

```
unit Main;
interface
```

```
uses
  Windows, Messages, SysUtils,
  Classes, Graphics, Controls,
  Forms, Dialogs, StdCtrls,
  StdActns, ActnList, Menus,
  ComCtrls, ToolWin, ExtCtrls,
  Gauges;

type
  TMyGauge = class(TGauge)
  private
    FTimer: TTimer;
  public
    constructor Create(AOwner: TComponent); override;
    destructor Destroy; override;
    procedure Timer1Timer(Sender: TObject);
  end;

  TForm1 = class(TForm)
    PageControl1: TPageControl;
    MainMenu1: TMainMenu;
    File1: TMenuItem;
    ShowForms1: TMenuItem;
    N1: TMenuItem;
    Exit1: TMenuItem;
    Help1: TMenuItem;
    Panel1: TPanel;
    N2: TMenuItem;
    procedure Exit1Click(Sender: TObject);
    procedure ShowForms1Click(Sender: TObject);
    procedure Help1Click(Sender: TObject);
    procedure N2Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation
```

```
uses
  DockWindow1, DockWindow2;

{$R *.DFM}

procedure TForm1.Exit1Click(Sender: TObject);
begin
  Close;
end;

procedure TForm1.ShowForms1Click(Sender: TObject);
begin
  Form2.Show;
end;

{ TMyGauge }

constructor TMyGauge.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);
  FTimer := TTimer.Create(Self);
  FTimer.OnTimer := Timer1Timer;
  FTimer.Interval := 10;
  Width := 100;
  Height := 100;
  Kind := gkPie;
end;

destructor TMyGauge.Destroy;
begin
  FTimer.Enabled := False;
  FTimer.Free;
  inherited Destroy;
end;

procedure TMyGauge.Timer1Timer(Sender: TObject);
var
  i: Integer;
begin
  Progress := Progress + 1;
  if Progress >= 100 then begin
```

```
      Progress := 0;
      for i := 0 to MAX do begin
        BackColor := RGB(Random(255), Random(255), 12);
        ForeColor := RGB(Random(255), Random(255), 12);
      end;
    end;
end;

procedure TForm1.Help1Click(Sender: TObject);
const
  S = '单击菜单中的显示窗体命令，可以分别显示两个窗体。在显示了它们之后拖动它们，观察它们的停靠情况';
begin
  ShowMessage(S);
end;

procedure TForm1.N2Click(Sender: TObject);
begin
  Form3.Show;
end;

end.
```

在这段代码中，关于窗体的代码没有什么需要介绍的，只是一些普通的用来显示窗口和退出应用程序的代码，它们是比较简单的。我们重点来介绍一下这段代码中的 TMyGauge **控件代码。**

关于 TGauge **控件的用法在前面已经做了介绍，但是在这里我们希望创建一个能够自动变化的** TGauge **控件，并事先控制这个控件的一些行为。参考类的编程可以得出，在我们声明的** TMyGauge **控件的构造函数和析构函数中的内容主要是调用原来的构造函数和析构函数，然后对控件的一些基本属性进行设置。请注意这两个函数中语句的顺序，不能把这些顺序颠倒。其中比较重要的是新建了一个** TTimer **控件，并把一个事件句柄赋给** TTimer **控件的** OnTimer **事件。这里使用的目的虽然比较简单，但是它提供了一种很有用的方法，即通过这种形式的组合，可以方便地把** Delphi **中已有的控件综合利用，以实现更为强大的功能。在后面的内容中，还会利用这个方法建立更为复杂的应用程序。**

第二个窗体中的代码引用了第一个窗体的代码，如下所示：

```
unit DockWindow1;

interface

uses
```

```
  Windows, Messages, SysUtils,
  Classes, Graphics, Controls,
  Forms, Dialogs, StdCtrls,Main;

const
  MAX = 3;

type
  TForm2 = class(TForm)
    procedure FormCreate(Sender: TObject);
    procedure FormDestroy(Sender: TObject);
  private
    FGauge: array [0..MAX] of TMyGauge;
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form2: TForm2;

implementation

{$R *.DFM}

procedure TForm2.FormCreate(Sender: TObject);
var
  i: Integer;
begin
  Randomize;
  for i := 0 to MAX do begin
    FGauge[i] := TMyGauge.Create(Self);
    FGauge[i].Parent := Self;
    FGauge[i].Visible := True;
    FGauge[i].Left := Round((Width / 4.5) * (i + 0.2));
    FGauge[i].Top := 100;
    FGauge[i].Color := Color;
    FGauge[i].backColor := RGB(Random(255), Random(255), Random(255));
    FGauge[i].ForeColor := RGB(Random(255), Random(255), Random(255));
    FGauge[i].BorderStyle := bsNone;
  end;
end;
```

```
procedure TForm2.FormDestroy(Sender: TObject);
var
  i: Integer;
begin
  for i := 0 to MAX do
    FGauge[i].Free;
end;

end.
```

这段代码的重要作用是演示了在 Delphi 中如何使用控件数组。在 Delphi 中使用控件数组的时候必须注意：如果在窗体的 OnCreate 事件中创建了这些控件数组，那么在窗体的 OnDestroy 事件中最后再释放这些控件，否则可能会引起系统的崩溃。使用控件数组是 Delphi 中处理多个相同类型控件的重要技术，对于一些可以通过它们的索引号进行重复操作的控件来说，使用控件数组可以大大方便我们的编程。

第三个窗体代码就比较简单了，只要列出它上面的两个选项按钮的 OnClick 事件就可以了，如下所示：

```
procedure TForm3.RadioButton1Click(Sender: TObject);
begin
    shape2.Shape := stRoundRect;
end;

procedure TForm3.RadioButton2Click(Sender: TObject);
begin
    shape2.Shape := stRectangle;
end;
```

图 7.24　窗体的停靠功能

下面就可以运行应用程序来观察窗体的停靠功能了。当启动这个程序时，会看到一个窗体，窗体的主体是一个空的 TPageControl 控件。使用主菜单可以显示项目中另外两个窗体。依次把它们拖放到 TPageControl 控件上，并停放好。等做好以后，就可以在二级窗体之间进行页面切换了。程序运行结果如图 7.24 所示。

7.4 动 作 列 表

在上面介绍工具栏的时候，曾经遇到这样的情况，就是定义了一个菜单的各个菜单项的 OnClick 事件，然后把这些事件句柄赋给另外一些菜单项目的 OnClick 事件。利用前面介绍的方法当然也可以实现所需要的功能，但是从程序阅读和组织上来说是十分不方便的，为此 Delphi 提供了一个 TActionList 对象。

TActionList 对象是从 Delphi 4.0 才开始出现的一个对象，它位于控件选项板的 Standard 选项卡上。这个对象在应用程序中提供了一个单独的空间，在这里可以对一个或者一个以上的方法进行管理。也就是说，通过这个对象，事先定义了一组方法，然后在应用程序中，只要把这些方法指定给对应的事件句柄就可以了。但是要注意的是，把某个方法指定给一个事件句柄时，必须注意它们的类型和参数相匹配。

7.4.1 使用动作列表

在下面的程序中，我们考虑建立一个图像浏览器，即一个可以通过一个 TImage 控件显示一个位图的简单程序。在这个程序中会发生如下三个动作。

- 可以打开一个文件并显示它。
- 可以决定是否需要拉伸图像。
- 可以关闭程序。

下面我们就开始创建这个程序，首先把一个 TActionList 对象放到主窗体上去。此时双击 TActionList 对象，会显示如图 7.25 所示的 Editing ActionList 对话框。

单击 ActionList 对话框左上角的图标，给这个列表添加三个动作。该图标的提示是 New Action。如果不想使用鼠标，只需按下 Insert 键就可以创建一个新的动作。

创造的默认动作可以根据上图对话框中所选择的值而改变。单击对话框顶部图标旁边的黑色箭头，会看到一个具有两个选项的菜单。第一个选择是 New Action（新建动作），第二个则是 New Standard Action（新建标准动作）。选择后面这个选项，进入一个如图 7.26 所示的 Standard Action 对话框。进入这个对话框的热键是 Ctrl + Insert。

当进入 Standard Action 对话框之后，将看到一系列 Delphi 中预先定义的动作列表。现在，请选择列表中的第一个项目，它应该叫作 TAction。稍后，在这一部分中，还会讨论像 TEditCut

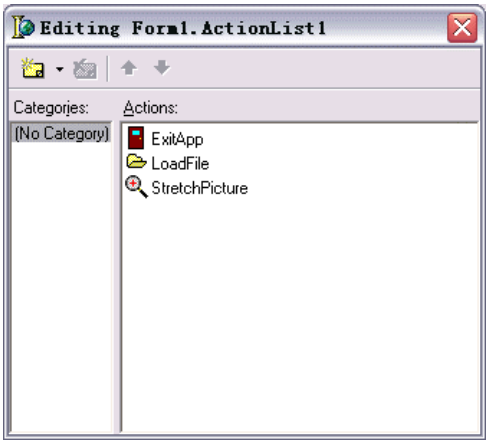

图 7.25　Editing ActionList 对话框

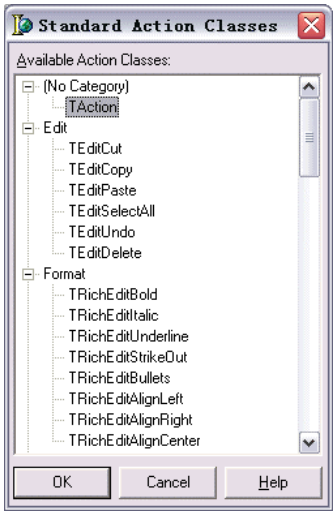

图 7.26　Standard Action 对话框

和 TEditPaste 这样的标准动作。

在默认情况下，创建的动作应该叫做 Action1、Action2、Action3。如果偶尔用别的名字建立了其他一些动作，只要选择对话框上面两个图标中的第二个图标就可以把它们删除掉。给这些动作改名，让它们的 Name 属性和 Caption 属性如表 7.1 所示。

表 7.1　TActionList 中的动作

Name（名称）	Caption（标题）
ExitApp	Exit
LoadFile	Open
StretchPicture	Stretch Picture

选中一个动作，然后在属性编辑器上单击 Events 选项卡，给每个动作指定一个事件。例如，与 ExitApp 动作关联的事件应该如下所示：

```
procedure TForm1 . ExitAppExecute ( Sender : Tobject ) ;
begin
      Close ;
end ;
```

下面是与 LoadFile 和 StretchPicture 相关联的事件：

```
procedure TForm1.LoadFileExecute(Sender: TObject);
begin
  if OpenDialog1.Execute then
    Image1.Picture.LoadFromFile(OpenDialog1.FileName);
```

```
    Caption := OpenDialog1.FileName + format( '%d X %d', [image1.picture.width,
  image1.picture.height]);
  end;
  procedure TForm1.StretchPictureExecute(Sender: TObject);
  begin
    Image1.Stretch := not Image1.Stretch;
  end;
```

然后从 Win32 选项卡拖下一个 TImageList 控件，并把它关联到 TActionList 控件上。向图像列表中添加几张图画。

现在把建立的每个动作分别关联到 ImageList 中的图像上。通过改变 Object Inspector 中关联到每个动作的 ImageIndex 属性值，就可以达到这个目的。如果把 ImageIndex 设置为 0，那么就会得到图像列表中的第一个图像，如果把它设置为 1，那么将会得到第二个图像，依此类推。

到这个阶段，已经建立了一个具有一系列动作的应用程序。这些动作中的每一个都与 TActionList 组件中的一个动作相关联。在某种意义上说，这个程序已经可以算是完成了。惟一欠缺的是缺少一条允许用户调用每个动作的途径。

为了完成这个程序，在主窗体上再放上一个 TMenu 控件。向 TMenu 控件中加入新的菜单项目。不必填写菜单标题，也不必修改每一个菜单项目的值，只要把菜单项目关联到 TActionList 中相应的动作上就行了。具体地说，新建一个叫做 Options 的菜单项目，将来它会出现在程序的顶端。在这个菜单项目的下面，添加三个或更多的菜单项目。

当选择每个项目时，不要给它一个标题或事件；而是应该把 TMenuItem 的 Action 属性与所创建的一个定制动作相关联。此时将会发现菜单项目的 Caption 和 OnClick 事件将被自动地设置为选中动作的 Caption 和 OnExecute 属性。

而且，如果把 TImageList 和菜单通过 Images 属性链接起来，那么在运行时，和动作关联的图标也将会出现在菜单上。

再另外给这个程序添加一些像 TToolBar 或 TButton 这样的控件。现在可以把 TToolBar 上的每一个按钮与不同的动作相关联，它的属性和标题将被自动填写好。

这样我们便完成了这个应用程序的制作过程。下面运行这个应用程序，结果如图 7.27 所示。

可以使用这个程序的菜单或工具栏来选择一个图片，并显示它，既可以按它正常的分辨率进行显示，也可以对它拉伸之后进行显示。

7.4.2 使用标准动作

上面我们已经提到，标准动作（Standard Actions）是一些预定义的动作，它们用来给程序自动添加一些特定的功能。下面我们就通过一个程序来了解它们是如何工作的。

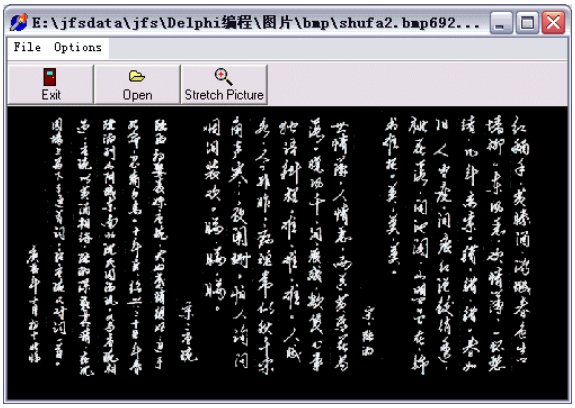

图 7.27　TActionList 对象的应用示例

我们要建立的应用程序有三个菜单项目和三个工具栏按钮，它们分别被标记为“ Copy ”, “ Cut ”和“ Paste ”。单击这些项目，它们就会完成期望的动作。

在程序的主窗体中有一个 RichEdit 控件。运行时，选中该控件中的文本，然后单击工具栏上的一个或几个按钮，看看怎样能够在这些控件中复制、剪切和粘贴文本。

如果不知道可以使用 Standard Action，也许需要使用剪贴板以及许多控件的 Copy to Clipord 等方法。虽然这样做并不困难，但是如果把用上面的方法建立的应用程序和我们下面要建立的应用程序进行一下对比，你会发现，利用下面的方法可以更加容易地创建一个具有上述功能的应用程序。

首先，新建一个程序，在主窗体中放上一个 TMenu、TToolbar、TImgeList 和一个 TActionList 控件。利用上面介绍的方法单击 Editing ActionList 对话框中的第一个按钮的旁边的箭头，然后选择 New Standard Action，此时会调出了 Standard Action 对话框。在这个对话框中，我们将会看到三个项目，分别叫做 TEditCut、TEditPaste 和 TEditCopy。选中这三个项目之后单击 OK 按钮。现在再向 TImgeList 中添加一些图像，并将图像列表和 TActionList、TMenu 以及 TToolbar 关联起来。在工具栏上添加三个按钮，每个按钮和一个不同的动作关联。为程序的菜单建立三个并列的菜单项目。现在可以编译代码并运行程序了，结果如图 7.28 所示。

试验后发现上面的这个应用程序和使用 RichEdit 控件关于剪贴板方法时的效果相同，而从程序上来说却简单多了。

TActionList 对象可以把特定的命令集中单独放在一起，即使那个命令的接口散布在一个或多个窗体上也没有关系。当我们对程序进行维护时，只需简单地单击 TActionList 组件，就会得到一个独立的、集中的、关于用户能够选择的所有命令的参考。它可以帮助用户建立具有合理顺序的接口，易于理解的图标，热键以及标签的程序。

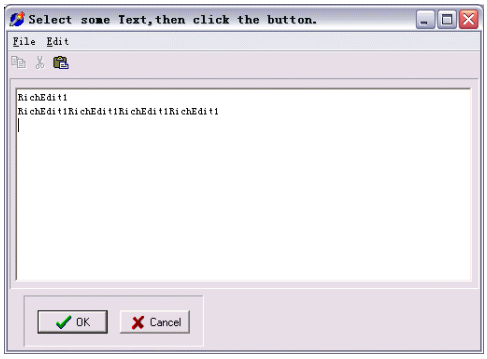

图 7.28 使用 Standard Action 程序示例

7.5 应用程序事件对象

我们在前面也介绍过，一个应用程序对 Delphi 来说也是一个对象，例如在一个工程文件中，经常会见到如下的代码：

```
program AppEvents;

uses
  Forms,
  main in 'main.pas' {MainForm};

{$R *.RES}

begin
  Application.Initialize;
  Application.CreateForm(TMainForm, MainForm);
  Application.Run;
end.
```

从上面的程序中，可以看到，Application 是 Delphi 中的一个特殊的对象，可以通过这个对象对应用程序的一些信息进行处理，但是我们现在希望应用程序就像 Delphi 中的其他控件一样具有可以自动响应的事件。由于上面提到的这个原因，在 Delphi 6.0 中，出现了一个新的控件：TApplicationEvents 控件。

该控件可以响应作为全局对象的 Application 对象的事件。当我们向窗体上添加一个 TApplicationEvents 对象时，通过这个对象就可以响应应用程序的事件了。虽然这里介绍的 TApplicationEvents 控件是响应作为全局对象的 Application 对象事件的，但是可以在每个窗体上都放置一个 TApplicationEvents 控件。当应用程序的某个事件获得响应时，工程中所有

TApplicationEvents 控件中对应的事件都会得到响应。如果要改变不同 TApplicationEvents 响应事件的顺序，可以改变 TApplicationEvents 对象的 Activate 方法。为了阻止其他的 TApplicationEvents 对象响应应用程序的事件，可以使用 CancelDispatch。

在使用 TApplicationEvents 控件时，比较重要的是要理解它各个事件的含义以及它们在什么情况下发生。

TApplicationEvents 控件的第一个事件是 OnActionExecute，它发生在调用一个执行动作时，此时这个动作中的内容还没有执行。当用户在操作应用程序时涉及了动作列表中的一个动作，但是应用程序中并没有一个 OnExecute 事件处理句柄来处理它时，如果一个动作列表包含了一个没有在 OnExecute 事件句柄中处理的动作，那么 TApplicationEvents 对象将捕获这个事件并用自己的 OnActionExecute 事件句柄来处理它。该事件句柄的 Handled 参数在默认情况下将返回 False。如果句柄处理了该事件，它将把 Handled 参数设置成 True，从而可以用其他的句柄处理该动作。当事件句柄在退出时把该参数设置成了 False，那么该事件将先被其他 TApplicationEvents 对象进行处理，然后再响应动作的 OnExecute 事件。如果该动作一直没有被处理而其他所有的句柄都没有处理该动作，则将会调用激活的控件的 ExecuteAction 方法。

注意：

在一个 OnActionExecute 事件句柄中调用 CancelDispatch 方法可以防止让其他 TApplicationEvents 对象发生该事件。

TApplicationEvents 控件的第二个事件是 OnActionUpdate。该事件发生在调用一个动作的 Update 方法并且它的动作列表还没有处理该动作之前。如果动作列表没有在 OnUpdate 事件句柄中处理这个事件，使用 OnActionUpdate 事件句柄可以在应用程序空闲时更新动作的属性。OnActionUpdate 事件发生在 OnIdle 事件之后。如果包含某个动作的动作列表没有在 OnUpdate 事件句柄中更新它，则该动作将转向 TApplicationEvents 对象的 OnActionUpdate 事件。TApplicationEvents 控件将接管这个事件并用自己的 OnActionUpdate 事件句柄来处理这个事件。在默认的情况下，事件句柄的 Handled 参数将会被设置成 False。如果事件句柄处理了该事件，则会自动把 Handled 参数设置成 True，从而可以结束动作的处理过程。如果事件处理句柄在退出时把 Handled 设置成 False，该事件会先进入其他 TApplicationEvents 对象并发生 OnUpdate 事件。如果该动作没有进行更新，将会调用激活的控件的 UpdateAction 方法，以便更新该动作。最后，如果所有的其他句柄没有处理这个动作，则激活的 Form 的 UpdateAction 方法将会处理这个事件。

OnActivate 事件发生在应用程序被激活时。我们在操作 Windows 系统时，经常会运行许多应用程序，当从一个应用程序切换到另一个应用程序时，另一个应用程序便被激活了，此时就会发生该应用程序的 OnActivate 事件。编写这个事件句柄可以在应用程序被激活时执行特殊的处理过程。除了上面提到的激活情况之外，应用程序被初始化时，也就是第一次运行

时也会被激活。

OnDeactivate 事件的含义正好和 OnActivate 事件相反，当从应用程序 A 切换到应用程序 B 时，应用程序 A 会发生 OnDeactivate 事件。可以在这样的一个事件句柄中编写任何代码，以便在应用程序失去焦点时执行。

OnException 事件的情况比较特殊，它发生在应用程序发生一个异常的时候。该事件的声明如下：

```
type TExceptionEvent = procedure (Sender: TObject; E: Exception) of object;
property OnException: TExceptionEvent;
```

当出现了程序代码中没有处理的异常时，可以使用 OnException 来改变异常处理程序的默认表现。需要注意的是，OnException 事件会处理发生在消息处理过程中的异常。在应用程序的 Run 方法执行之前或者之后发生的异常不会引发 OnException 事件。在这个事件句柄的参数中，Sender 参数就是引发异常的那个对象，而 E 就是 Exception 对象。

OnHelp 事件的声明如下：

```
type THelpEvent = function (Command: Word; Data: Longint; var CallHelp: Boolean):
                    Boolean of object;
property OnHelp: THelpEvent;
```

这个事件发生在应用程序收到一个帮助要求时。当用户要求帮助时，利用 OnHelp 事件句柄可以执行一些特殊的处理过程。HelpContext 属性和 HelpJump 方法会自动触发 OnHelp 事件。在事件之后会把 CallHelp 设置成 True，这样可以使应用程序调用 WinHelp。如果把 CallHelp 参数设置成了 False，则可以使应用程序不会调用 WinHelp。应用程序的所有帮助方法都会引发 OnHelp 事件。只有 OnHelp 的 CallHelp 参数设置成 True 或者没有 OnHelp 事件处理句柄的时候，应用程序才会调用 WinHelp。如果事件句柄成功调用，将返回 True，否则将返回 False。

OnHint 事件发生在把鼠标指针移动到一个会显示提示条的控件或者菜单项目上面的时候。也就是说，当鼠标指针在一个 Hint 属性非空的控件或者菜单项目上停留时，就会引发 OnHint 事件。该事件的一个常见用法就是把该控件或者菜单项目的 Hint 属性的值显示在状态栏上。

OnIdle 事件发生在应用程序空闲时，该事件句柄的声明如下所示：

```
type TIdleEvent = procedure (Sender: TObject; var Done: Boolean) of object;
property OnIdle: TIdleEvent;
```

当一个应用程序不再处理任何代码时，这个应用程序就处于空闲状态。例如，当一个应用程序等待用户的输入时，这个应用程序就处于空闲状态。TIdleEvent 有一个 Boolean 型参数 Done，它在默认的情况下是 True。当 Done 参数是 True 的时候，在 OnIdle 返回时将调用 Windows API 函数 WaitMessage。当 Done 参数值是 False 的时候，应用程序在空闲时不会接

受其他应用程序的控制。

当应用程序进入空闲状态时，只会调用一次 OnIdle，而不是只要应用程序处于空闲状态就不断地发生 OnIdle 事件。除非把 Done 参数设置成“ False ”，否则它不会连续地执行。如果在应用程序中把 Done 参数设置成 False，会占用大量的 CPU 时钟，这会大大降低系统的整体性能。

OnMessage 事件发生在应用程序接收到一个 Windows 消息的时候，该事件句柄的声明如下：

```
type TMessageEvent = procedure (var Msg: TMsg; var Handled: Boolean) of object;
property OnMessage: TMessageEvent;
```

OnMessage 可以捕获传递到应用程序中所有窗口的任何 Windows 消息。OnMessage 事件发生在应用程序接收到一个 Windows 消息的时候。需要注意的是，OnMessage 只会接收那些添加到消息序列中的消息，而不是那些直接用 Windows API 函数 SendMessage 处理的消息。OnMessage 事件句柄使得我们可以在应用程序中响应消息，而不是在 TApplicatino 对象的其他事件中处理。如果应用程序没有一个用于接收消息的特定的句柄，那么消息会指派给它所约定的窗口，并由 Windows 来处理这个消息。

在这个事件句柄中，Msg 参数代表了 Windows 的消息，而 Handled 参数可以指明事件句柄是否响应了消息。如果在事件句柄中已经完全处理了该消息，则可以把 Handled 参数设置成“ True ”，那么后面的程序就不能响应该消息了。

OnMinimize 事件发生在窗口进行最小化的时候。当应用程序进行最小化的时候，可以利用 OnMinimize 事件进行特殊的处理。有以下三种情况可引发该事件。

- 应用程序被最小化。
- 用户最小化了主窗口。
- 调用了 Minimize 方法。

OnRestore 事件发生在最小化的应用程序恢复到自己的正常尺寸时。可以说这个事件是上一个事件的反事件，它们情况非常类似。

OnShortCut 事件发生在用户按下键盘上的一个按键时，该事件发生在应用程序中的窗体和控件的 OnKeyDown 事件之前。

```
TShortCutEvent = procedure (var Msg: TWMKey; var Handled: Boolean) of object;
property OnShortCut: TShortCutEvent;
```

从上面的简述就可以看出，可以利用 OnShortCut 事件在窗体和其他控件处理这些按键之前进行处理，从而可以指定为应用程序的快捷按键。当用户按下一个按键时，应用程序可以把它指定为快捷键，从而可以阻止其他的标准键盘按键处理程序的运行（例如 OnKeyDown、OnKeyPress 和 OnKeyUp）。在 Delphi 中，内置的快捷键处理程序会为菜单和动作的快捷键处理需要的键盘按键，利用 OnShortCut 可以指定其他的快捷键。如果该事件处理句柄响应了键

盘按键，可以把 Handled 参数设置成“ True ”，这可以防止这个按键仍然被传递给菜单或者其他动作来进行处理。当然，这样做也会阻止其他的标准键盘处理事件来响应该键盘按键。

我们要介绍的 TApplicationEvents 对象的最后一个事件是 OnShowHint，该事件的声明如下：

```
type TShowHintEvent = procedure (var HintStr: string; var CanShow: Boolean;var HintInfo:
THintInfo) of object;
property OnShowHint: TShowHintEvent;
```

可以想象，这个事件发生在应用程序显示提示条的时候。实际上这个事件发生在即将要显示提示条，但是还没有显示的时候。如果要改变提示条的内容，则可以在这里直接改变 HintStr 参数的值，那么显示在提示条中的内容就会变成在这里所设置的内容，而不是在设计应用程序时指定的内容。使用 CanShow 参数可以允许或者禁止显示提示条。如果该参数设置成“ True ”，就会显示提示条；否则将禁止提示条的显示。

HintInfo 参数是一个记录结构，它包含了关于提示条窗口的外观和行为信息，改变它的内容可以自定义提示条显示的方式。

7.6 菜单、工具栏一体化工具

在 Delphi 6.0 中，新增加了三个关于菜单、工具栏设计的控件，它们是：ActionManager、ActionMainMenuBar 和 ActionToolBar。它们的使用是联系在一起的。通过它们可以更方便地建立起程序的界面，特别是菜单和工具栏。

一般来说，用户不会单独使用其中的某个控件。所以先介绍一下这些控件是如何在一起发挥作用的。

(1) 首先，新建一个应用程序，然后在窗体上放置一个 ActionManager 控件和一个 ActionMainMenuBar 控件，以及一个 TImageList 控件并载入需要的图片。

(2) 然后在对象浏览器中展开 ActionManager 控件，此时会看到该控件还包含了其他两个对象，如图 7.29 所示。

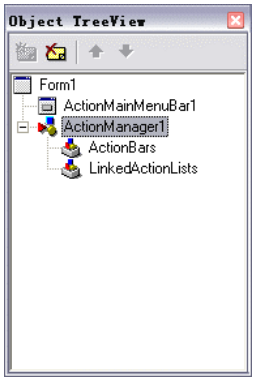

图 7.29 对象浏览器中的 ActionManager 控件

(3) 在 ActionBars 上右击鼠标，此时会显示一个快捷菜单，选择上面的 Add Item，此时会添加一个 TActionBarItem 对象，然后在属性编辑器中把 ActionBar 属性指定为上面添加的 ActionMainMenuBar1 控件。

(4) 双击 ActionManager 控件，此时会显示如图 7.30 所示的对话框。

(5) 在中间的两个框中右击鼠标，此时会显示如图 7.31 所示的快捷菜单。

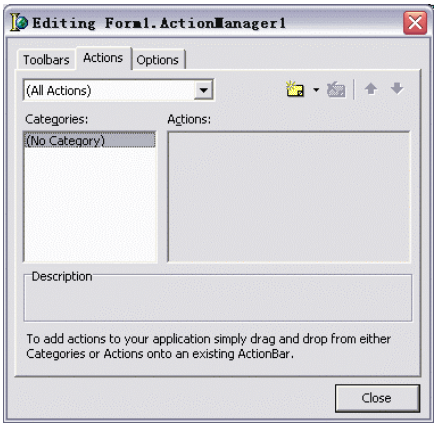

图 7.30　ActionManager 控件的编辑窗口

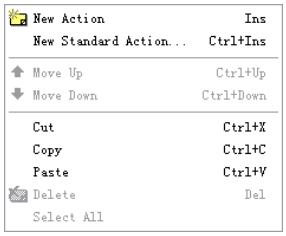

图 7.31　关于 Action 的快捷菜单

(6) 选择 New Action，此时便添加了一个 Action，注意 Action 也是一个对象，可以在属性编辑器中设置它的属性，比如 Name 等。这里主要是指定每个 Action 所对应的 TImageList 控件中的图片，也就是 ImageIndex 属性的值。然后双击对话框中的该 Action，便进入代码编辑器，此时就可以编写该 Action 对应的代码了。

(7) 选择快捷菜单中的 New Standard Acton，此时便显示如图 7.32 所示的标准动作对话框。选择需要的动作，把它们加入到 ActionManager 控件中。

(8) 单击对象浏览器中的 0-ActionBar-ActionMainMenuBar，把该项目展开，此时的对象浏览器如图 7.33 所示，注意其中也包含了我们在上面加入的动作。

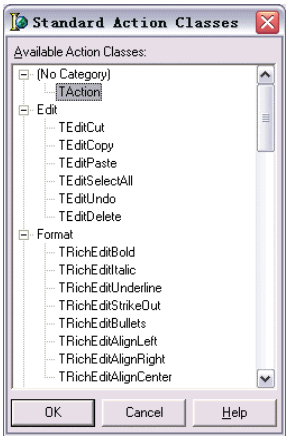

图 7.32　标准动作对话框

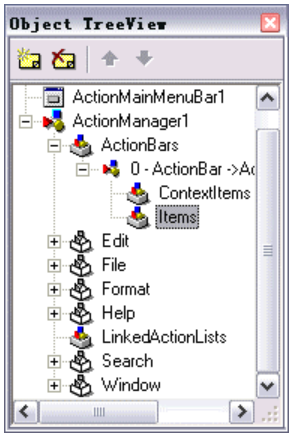

图 7.33　加入了动作的对象浏览器

(9) 在如图所示的 Items 中，右击鼠标，此时也会显示一个快捷菜单，选择快捷菜单中的 Add Item，便会为它添加一个新的项目。在属性编辑器中把该项目的 Caption 属性修改为“&File”，然后重复上面的操作，直到把所有需要的项目添加完毕。此时把 Caption 属性为

“&File”的项目再展开，会看到它也包含一个 Items 对象，在该对象上右击鼠标，并选择 Add Item，此时便增加了一个新的项目。在属性编辑器中把该项目的 Action 属性指定为前面添加的 FileNew。

(10) 重复上面的过程，直到把 File 菜单中需要的所有项目添加完毕。此时的对象浏览器如图 7.34 所示。

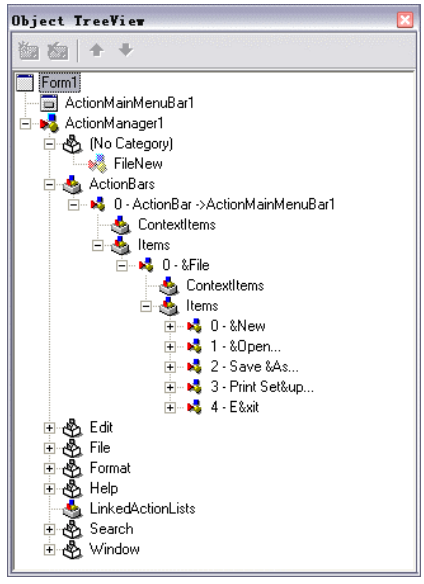

图 7.34 对象浏览器

(11) 重复上面的(9)、(10)两步，直到把所有的需要的菜单项目都添加上为止。运行应用程序，结果如图 7.35 所示。

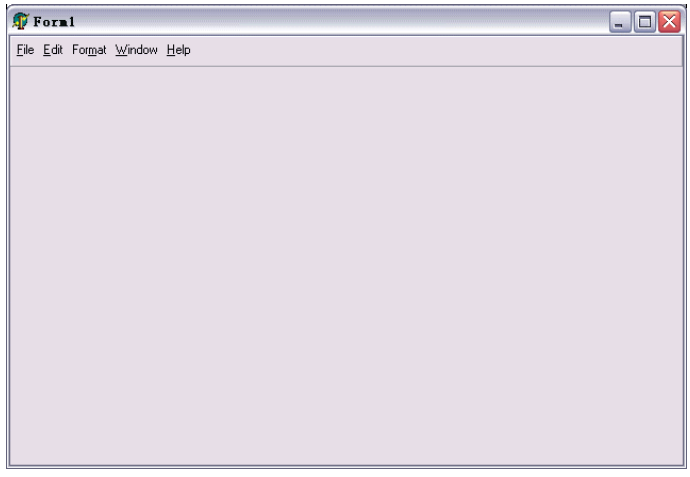

图 7.35 使用 ActionManager 生成的菜单

(12) 在此双击 ActionManager 控件，显示它的编辑对话框，然后单击对话框上的 Toolbars 选项卡，此时的对话框如图 7.36 所示。

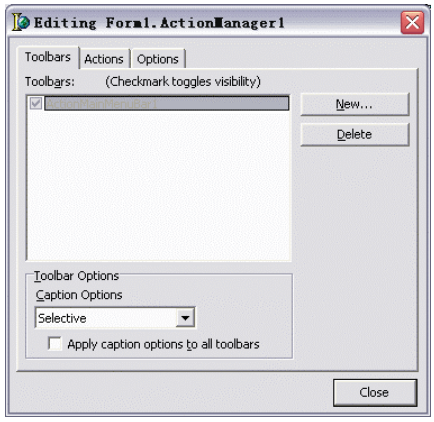

图 7.36　ActionManager 控件中的 Toolbars

(13) 单击对话框中的 New 按钮，添加两个 Toolbar，然后单击 Close，这时会发现在窗体上创建了两个工具栏。仍然是在对象浏览器中，展开其中的一个工具栏对应的控件，然后在它下面的 Items 中添加需要的项目，就像为 ActionMainMenuBar 添加项目一样。每添加一个项目，要确定是否显示该项目的标题，也就是确定 ShowCaption 属性是否为 True。

(14) 在添加完了所有需要的项目之后，再在窗体上放置一个 TRichEdit 控件。此时需要编写两行程序，代码如下所示：

```
procedure TForm1.FileOpen1Accept(Sender: TObject);
begin
  RichEdit1.Lines.LoadFromFile(FileOpen1.Dialog.FileName);
end;

procedure TForm1.FileSaveAs1Accept(Sender: TObject);
begin
  RichEdit1.Lines.SaveToFile(FileSaveAs1.Dialog.FileName);
end;

procedure TForm1.FileNewExecute(Sender: TObject);
begin
  RichEdit1.Clear;
end;
```

(15) 在上面的代码中，使用了标准事件的 OnAccept 事件和自定义事件的 OnExeCute 事件。通过这三个事件，可以打开、保存或者新建文档。运行该应用程序，结果如图 7.37 所示。

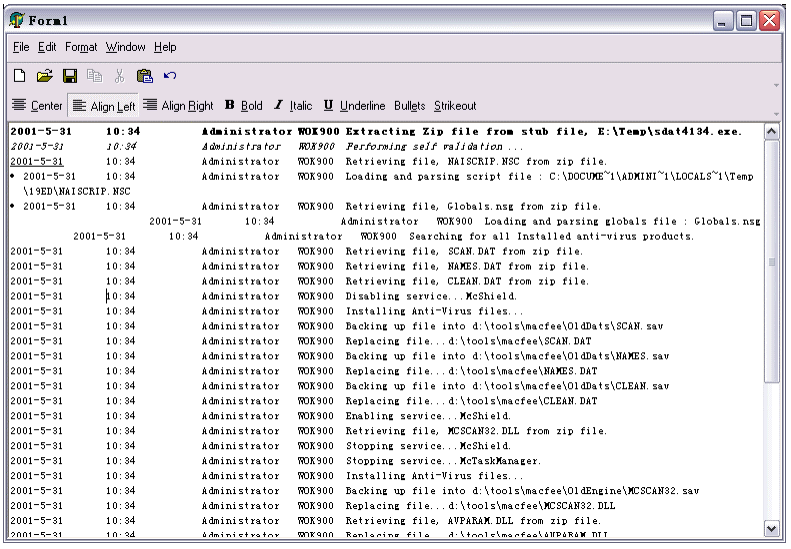

图 7.37 载入了文档的应用程序

仅仅通过三行代码，就完成了上面这样一个文档编辑器，这不能不说是 Delphi 的神奇。现在，利用 Delphi 编写常规的应用程序已经变得非常非常容易了。而且，如果使用一下该应用程序，你会发现，它的菜单能够使用 Microsoft Word 里面的菜单记忆功能，也就是只显示常用的菜单，隐藏其他不常用的菜单功能。

7.7 本 章 小 结

本章介绍了如何在 Delphi 中使用菜单和工具栏。现在我们可以用多种方式快速地创建自己所需要的菜单或者工具栏，特别是对于那些属于 Delphi 预先定义的标准动作的菜单或者工具栏。在本章中，动作和事件的概念尤其重要。

第 8 章　Delphi 中的绘图

如果在应用程序中适当使用一些图像方面的内容，毫无疑问可以使这些应用程序更加具有吸引力。Delphi 提供了很多在应用程序中引入图像的方法。比如，如果我们要在程序中加入图像，那么可以使用图像控件把预先设计好的图片加入到程序中，也可以在程序运行的时候动态地绘制它们。

在本章的内容中，主要介绍如何绘制图像，将使用到 Delphi 中的 TCanvas 对象。Delphi 在 TCanvas 对象中很好地封装了绘图功能，利用它几乎可以完成所有的绘图任务。

在这一章中，主要讲述可视组件库（VCL）图形类，将重点阐述以下内容：

- TCanvas
- Brushes
- Pens
- Fonts
- 示例程序 WokPaint

8.1　图像编程概述

8.1.1　图像编程中的 Canvas 对象

在 Delphi 中，关于图像控件以及对象的定义在 Graphics 单元中，在该单元中封装了 Windows GDI（Graphics Device InterFace，图像设备接口），这方便我们向应用程序中添加图像对象，并对它们进行处理。

需要说明的是，如果希望在应用程序中绘制一个图形，需要在某个对象的 Canvas 对象上进行，而不是直接在这个对象上进行绘制。在很多情况下，Canvas 将是要绘图的那个对象的一个属性。形象地说，Canvas 就是一个画布，我们可以在上面绘制任何图形、图像。它提供了很多画图的工具、设置颜色的方式。完全可以像理解画家的画夹一样来理解 Canvas 对象。

使用 Canvas 对象有一个很大的优点，就是不管是在屏幕上、打印机上还是一个位图上绘制图形时，都可以使用同样的方法，而不要根据输出设备的不同而修改方法。

说明：

Canvas 属性只能在运行期访问，也就是说，我们不得不编写代码来完成工作。

图像在程序中显示的方式取决于所使用的 Canvas 属性的对象类型，这里所说的对象是拥有该 Canvas 属性的那个对象。如果直接在一个控件的 Canvas 属性上进行绘图，那么绘制的结果将直接显示在屏幕上。但是，如果在一个隐含的对象的 Canvas 属性上作图，那么直到用某种方法把结果显示出来，我们才能看到自己的绘图结果。比如，如果在程序中，对一个 TBitmap 对象的 Canvas 属性进行绘图，那么在把这个对象的图片复制给一个相应的图像控件（如 Image）之前，我们是无法看到自己的绘图结果的。

在处理图像的时候，经常会使用到两个名词：Draw 和 Paint，这在图像处理和控件的开发中是十分重要的。但是这里要特别注意它们之间的区别。

- Draw：它的功能是创建一个特定的对象，比如在程序中，我们可能会在一个指定的区域绘制一个矩形、圆形或者就是一条直线。那么此时使用的是 Draw 方面的方法，比如 Line 等等，对应的事件有 OnDraw 等。
- Paint：它的功能是绘制对象的外观，比如在显示一个窗体时，程序需要把一个窗体绘制到屏幕上，此时对应的事件有 OnPaint 等。

8.1.2 屏幕的刷新

在图像编程中的另一个重要问题是屏幕的刷新问题。在特定情况下，是由操作系统来确定屏幕上的对象是否需要刷新的，这个时候，操作系统会生成 WM_PAINT 消息，从而引发 OnPaint 事件。如果在程序中定义了某个对象的 OnPaint 事件处理程序，那么当我们在程序中调用 Refresh 方法的时候，程序将调用该事件处理程序。例如，当在窗体上放置了一些图像控件，或者直接在窗体上绘制图形时，可以在窗体的 OnResize 事件中调用 Refresh 方法来重新绘制窗体上的对象。

例如，如果要在窗体上放置一个 TImage 控件，并利用这个控件的 Canvas 属性来绘制图形时，可以试试下面的程序：

```
procedure TForm1.Timer1Timer(Sender: TObject);
begin
    xx:=xx+5;
    yy:=random(Image1.Height);
    Image1.Canvas.Pen.Color :=ClBlack;
    Image1.Canvas.LineTo(xx,yy);
end;

procedure TForm1.Button1Click(Sender: TObject);
begin
Timer1.Enabled := not Timer1.Enabled;
end;
```

在上面的程序中，我们在窗体上放置了一个 TTimer 控件，并把该控件的时间间隔设置成 100 毫秒。然后运行该程序，并在不同的窗体之间进行切换。注意，在切换的过程中，要有一个窗口能够覆盖这个程序的窗口。此时你会发现 Windows 系统会自动刷新整个屏幕，也就是说，当我们切换到这个程序的窗口时，我们在 Image1 控件上绘制的图形仍然存在。如果把 Image1 控件的 Transparent 属性设置成“ True ”，那么此时再运行这个程序，就会发现，屏幕晃动得厉害。这是因为每当这个程序在 Image1 控件中画一条直线时，该控件都会自动刷新。

如果把上面的 Image1 控件换成一个 PaintBox 控件，仍然用类似的代码，比如下面的代码：

```
procedure TForm1.Timer1Timer(Sender: TObject);
begin
    xx:=xx+5;
    yy:=random(Paintbox1.Height);
    PaintBox1.Canvas.Pen.Color :=ClBlack;
    PaintBox1.Canvas.LineTo(xx,yy);
end;

procedure TForm1.Button1Click(Sender: TObject);
begin
Timer1.Enabled := not Timer1.Enabled;
end;
```

再运行该程序，并在不同的程序之间切换，此时你会发现，我们绘制的图像不能在屏幕上保持，如图 8.1 所示。

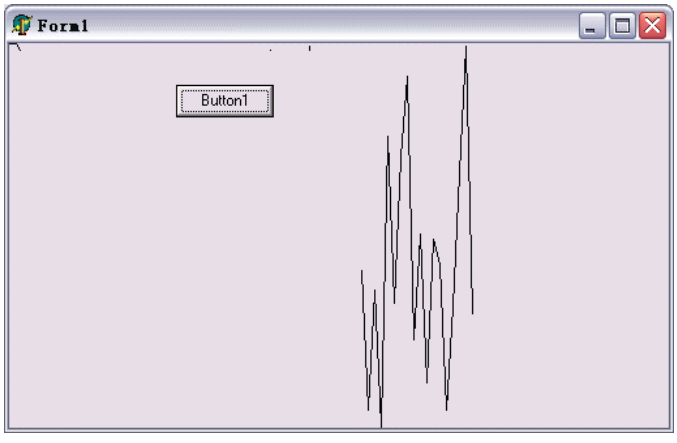

图 8.1　在切换后窗体上曾被覆盖的部分图像没有保留下来

所以，如果需要使用 PaintBox 控件来绘制图形，则必须在这个控件的 OnPaint 事件中对

图像的绘制进行处理，它上面的内容才会一直显示，否则，当切换窗口时，原来的内容有可能会消失掉其中的一部分。

8.1.3 图像对象的类型

在 Delphi 中，提供了很多的图像对象。这些对象一般都具有在 Canvas 对象上作图的方法，也可以把某些预先绘制好的图像载入到这些对象中，或者把这些对象中的图像保存到文件中去（见表 8.1）。

表 8.1 Delphi 中的图像对象

对象	说明
Picture	可以用来存储图像。如果要为它添加另外的文件格式，则可以使用它的 Register 属性。通常用这个控件来处理即将在一个图像控件上显示的图片文件
Bitmap	这是 Delphi 中非常有用的一个图像对象，我们可以用它来创建、处理图像，并可以把图像存储到硬盘上的文件中。在程序中，复制一个 Bitmap 对象是非常快的，因为这个过程只是复制句柄，而不是复制对象
Clipboard	剪贴板对象，代表了其中的图像和文字。使用该对象，可以获得需要的数据。它的处理包括计算、打开或者关闭 Clipboard 对象，以及管理和处理其中的对象
Icon	图标对象
Metafile	元图像对象。该对象和 Bitmap 对象不同，Bitmap 对象包含的是实际的像素点，而它则包含了绘制一个图像所需要的所有操作记录。这种图像格式基本不会失真，而且需要很少的内存，特别是对于打印机来说。但是元图像的显示没有位图快

8.2 使用 TCanvas

TCanvas 对象是像 Tform、TGraphicControl 及其后代类的一种标准属性。它提供一个接口，可以在此接口上进行绘图以及重绘对象和控件。

要想学会跑步，就得先学会走路，因此本章的重点在于讲解 TCanvas 类的基础。TCanvas 对象包括了一定数量的与图形有关的类。可以通过以下属性访问它们：

- Brush
- CopyMode
- Font
- Pen
- PenPos
- Pixels
- ClipRect

8.2.1　Brush 对象概述

在下面的内容中，我们会介绍 TBrush 对象。主要是介绍它的四种最重要的方法或属性：

- Color
- Style
- Bitmap
- Assign

Brush 是一种由应用程序用以在多边形、椭圆形的内部，或者在一个窗口或窗口化的控件背景上涂色的图形工具。例如我们可以把属性编辑器中的 Form 的颜色设置为 clWhite。这种设置实际上是将该窗体的图形刷颜色变成白色，从而可以使整个窗体的背景颜色变成白色。

当使用画布或者画布上的对象和形状时，需要指定用什么颜色来涂绘或“充满”该画布或形状。这需要用一个 TBrush 对象来完成这项工作。在一般情况下，没有必要自己创建一个图形刷。图形刷的默认颜色是白色：clWhite。例如，一个窗体通常将 clBtnFace 作为它的默认颜色。

可以通过 Brush 的 Color 属性来设置 Brush 对象的颜色，如下所示：

```
Canvas.Brush.Color := clRed;
```

我们要谈到的 TBrush 的下一个属性是 Style（样式）。样式是指用图形刷的什么模式来涂绘。表 8.2 中列出了可用的预定义样式，可以将它们赋值给画布的图形刷。

表 8.2　TBrush 样式类型

名称	说明
bsSolid	单色
bsCross	相交叉的水平线和垂直线
bsClear	透明
bsDiagCross	相交叉的双向对角线
bsBDiagonal	向后的对角线
bsHorizonal	水平线
bsFDiagonal	向前的对角线
bsVertical	竖直线

要设置图形刷的样式，可以使用下面的代码：

```
Canvas.Brush.Style := bsCross;
```

在下面的程序代码中，利用一个 RadioGroup 控件来控制程序中的 TImage 对象的图形刷样式。

```
unit BrushStyle;

interface

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

type
  TForm1 = class(TForm)
    RadioGroup1: TRadioGroup;
    Panel1: TPanel;
    Image1: TImage;
    procedure RadioGroup1Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation

{$R *.DFM}

procedure TForm1.RadioGroup1Click(Sender: TObject);
begin
Image1.Canvas.Pen.Color := clbtnface;
Image1.Canvas.Brush.Color := clbtnface;
Image1.Canvas.Rectangle (0,0,panel1.ClientWidth,Panel1.ClientHeight);

  case RadioGroup1.ItemIndex of
      0:
          image1.Canvas.Brush.Style := bsSolid;
      1:
          image1.Canvas.Brush.Style := bsCross;
      2:
          Image1.Canvas.Brush.Style := bsClear;
      3:
          Image1.Canvas.Brush.Style := bsDiagCross;
```

```
        4:
            Image1.Canvas.Brush.Style := bsBDiagonal;
        5:
            Image1.Canvas.Brush.Style := bsHorizontal;
        6:
            Image1.Canvas.Brush.Style := bsFDiagonal;
        7:
            Image1.Canvas.Brush.Style := bsVertical;
    end;
Image1.Canvas.Pen.Color := clYellow;
Image1.Canvas.Brush.Color := clBlue;
Image1.Canvas.Rectangle (10,10,panel1.ClientWidth-20,Panel1.ClientHeight-20);
end;
end.
```

程序运行结果如图 8.2 所示。

图 8.2　TBrush 对象的 Style 属性示例

除了这些预定义的模式以外，也可以创建自己定制的模式，并把它贮存于位图格式中，或者使用预先存在的位图来定义自己的模式。这些定制的位图通过图形刷 Bitmap 的属性加载。如果使用定制模式，它将超越已经定义的任何样式。因此，当用完定制模式后，必须自己清除该模式。

下面的代码段说明了如何把图形刷设置成位图。首先，需要声明一个 TBitmap 类型的变量：

```
Var
MyCustomBrush: TBitmap;
```

然后，需要创建这种类型的一个实例，把一个位图加载在这个实例中，然后给 Brush 对象赋值：

```
MyCustomBrush := TBitmap.Create;
MyCustomBrush.LoadFromFile('MyPattern.bmp');
```

```
Canvas.Brush.Bitmap := MyCustomBrush;
```

当用完一个位图时，请不要忘记将这个属性重新设置为 NIL，并且释放这个位图：

```
Canvas.Berush.Bitmap := NIL;
MyCustomBrush.Free;
```

例如，我们只要把上面的程序稍加修改，便可以获得这样的一个图形，需要注意的是，这里使用的是一个已经存在的位图文件。

```
unit BrushStyle;

interface

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

type
  TForm1 = class(TForm)
    RadioGroup1: TRadioGroup;
          … //同上。
  private
    MyPatten:TBitmap;//声明一个位图对象
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation

{$R *.DFM}

procedure TForm1.RadioGroup1Click(Sender: TObject);
begin
Image1.Canvas.Pen.Color := clbtnface;
Image1.Canvas.Brush.Color := clbtnface;
Image1.Canvas.Rectangle (0,0,panel1.ClientWidth,Panel1.ClientHeight);
Image1.Canvas.Brush.Bitmap := nil;
//清除以前的图形格式。
Image1.Canvas.Pen.Color := clYellow;
```

```
Image1.Canvas.Brush.Color := clBlue;
   case RadioGroup1.ItemIndex of
       0:
           image1.Canvas.Brush.Style := bsSolid;
… //同上。
       8:
           Image1.Canvas.Brush.Bitmap := Mypatten;//设置自定义模式。
   end;
Image1.Canvas.Rectangle (10,10,panel1.ClientWidth-20,Panel1.ClientHeight-20);
end;

procedure TForm1.FormCreate(Sender: TObject);
begin
    Mypatten := TBitmap.Create;
    MyPatten.LoadFromFile('mypatten.bmp');//载入我们需要的位图。
end;

procedure TForm1.FormDestroy(Sender: TObject);
begin
Mypatten.Free;//释放位图对象。
end;
end.
```

程序的运行结果如图 8.3 所示。这个位图是利用 Delphi 提供的 Image Editor 来设计的。当然，如果你精于设计这样的图形，那么肯定可以设计出更丰富多彩的填充风格。

图 8.3　自定义 Style 模式

利用 Brush 的 Bitmap 属性，还可以用一个图片来填充窗体的背景，例如可以使用下面的代码来完成这样的功能：

```
procedure TForm1.FormPaint(Sender: TObject);
var
Bitmap: TBitmap;
begin
Bitmap := TBitmap.Create;
try
Bitmap.LoadFromFile('E:\Temp\WallPaper1.bmp');
Form1.Canvas.Brush.Bitmap := Bitmap;
Form1.Canvas.FillRect(Rect(0,0,bitmap.Width,bitmap.height));
finally
Form1.Canvas.Brush.Bitmap := nil;
Bitmap.Free;
end;
end;
```

注意，这里把这部分代码放在了窗体的 OnPaint **事件处理程序中，其原因在前面已经做了介绍，程序的运行结果如图** 8.4 **所示。**

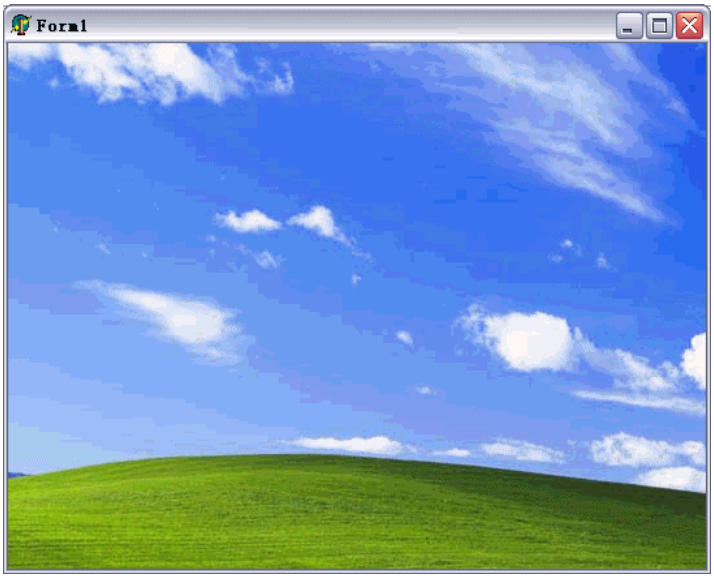

图 8.4　**使用** Brush **的** Bitmap **属性**

要讲的最后一项是 Brush **的** Assign **方法。这个方法相当方便，它允许将一个** Brush **的内容复制到另一个** Brush **对象上去。可以创建几个定制的图形刷，通过在图形刷之间的转换来同时使用它们。下面的例子将当前的图形刷复制到了** myBrush **中。在这个例子中，可以认为** myBrush **已经在** Var **部分声明过了。**

```
myBrush := TBrush.Create;
myBrush.Assign(Image1.Canvas.Brush);
```

现在 myBursh 具有了当前图形刷的所有特征。

说明：

Assign 方法可以适用于很多对象间的赋值。在把一个对象赋值给另一个相容的对象时，通常使用该方法。

8.2.2　Pen 对象概述

TPen 对象规定了画布用来画线及图形轮廓的画笔种类。画笔有几种属性和方法是与图形刷相同的，例如画笔和图形刷的 Color 和 Style 属性以及 Assign 方法是有相同的功能。在本节中，我们将讨论下面几种不同的属性：

- Style
- Width
- Mode

画笔和图形刷的差别之一在于画笔具有不同的样式类型。表 8.3 中列出了画笔可以设定的不同的画笔样式。

表 8.3　画笔样式类型

名称	说明
psSolid	实线
psDash	短划线
psDot	点线
psDashDot	包含短划和点的线
psDashDotDot	包含一条短划和两个点的线
psClear	透明的线
psInsideFrame	实线，但当宽度大于一个像素时使用一种 diThered 颜色

表 8.3 中的画笔样式，除了 psSolid 以外，都只在画笔的 Width 属性设置为一个像素时有效。如果宽度大于 1，将忽略这个属性。

Width（宽度）属性允许以像素为单位改变运行期的 Pen 对象的宽度。例如可以使用下面的代码来改变画笔的宽度：

```
Canvas.Pen.Width := 5;
```

Mode（方式）属性指的是画笔的颜色如何与画布的颜色相互作用。当在画布上画出一条线时，需要考虑下面三个因素：

- 画笔的颜色
- 目标的颜色
- 画笔的方式

画笔方式是一种光栅运算（Raster Operation，ROP）。表 8.4 描述了不同的可用画笔方式。

表 8.4 画笔方式

名称	说明
pmBlack	总是黑色
pmWhite	总是白色
pmNop	不变
pmNot	与画布背景颜色相反
pmCopy	在 Color 属性中指定的画笔颜色
pmNotCopy	与画笔颜色相反
pmMengePenNot	画笔色与画布背景的相反色组合
pmMaskPenNot	画笔色与画布背景的相反色的共有色组合
pmMergeNotPen	画布背景色与画笔的相反色组合
pmMaskNotPen	画布背景色与画笔的相反色的共有色组合
pmMerge	画笔色与画布背景色的组合
pmNotMerge	与 pmMerge 相反的颜色
pmMark	画笔色与画布背景的相反色的共有色组合
pmNotMask	与 pmMask 相反的颜色
pmXor	画笔色或画布背景色，但不是二者都有组合
pmNOtXor	与 pmXor 相反的颜色

下面的代码表示的是 pmMergePenNot：

```
procedure TForm1.FormClick(Sender: TObject);
begin
    Canvas.Pen.Mode := pmMergePenNot;
    Canvas.Pen.Color := clRed;
    Canvas.Pen.Width := 5;
    Canvas.RoundRect(10, 10, 300, 300, 50, 50);
    canvas.RoundRect (50, 50, 350, 350, 50,50);
end;
```

由于上面的程序代码是位于 TForm1 的 OnClick 事件中，所以当单击窗体的时候，会发现程序会在窗体上绘制出不同的两个图形，如图 8.5 所示。

正如所看到的，线条交叉的结果是画笔画出了画笔色及界面色的合并算法的相反颜色。记住，一旦画了什么，它就成为绘画界面的一部分。这就意味着在线条交叉或者重叠的地方，用一支红色的画笔在红色的界面上绘画，而红色与红色合并（或者“和”）的结果是红色。现在得到了这个合并的相反色，可以从交叉点上看到这个结果。

注意：

众所周知，学习画笔方式最简单的方法就是使用它们，也可以先用一个二进制计算器来

计算所想要的结果，然后运用这种方式。

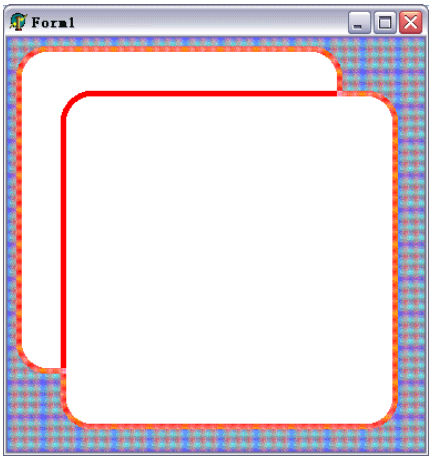

图 8.5　第一次单击窗体所绘出的图形

8.2.3　Font 对象概述

和画笔和图形刷一样，字体是另一种运用在画布上的绘画形式。字体是在屏幕上展示文本的方式。字体在 VCL 中用 TFont 对象来表示，并通过 Font 属性供使用。下面是几个经常使用的字体属性：

- Height
- Size
- Name
- Style

字体的 Height 只是不包括出现在字体顶端的内部的字体高度。Height 以像素为单位测量。Delphi 用下面的公式决定 Height 属性的值：

```
Font.Heiht := -Font.Size * Font. PixelsPerInch div 72
```

因此，每当确定了 Height 属性是正值，字体的 Size 属性就就变成了负数。相反的，如果把 Size 属性定为正值，那么字体的 Height 就变成负数。一个正的 Height 包括 internal leading，而负的则不包括。

Name 属性定义了所使用的字体，例如 Arial，Times New Roman。可以使用系统中已经安装的任何一种字体，也可以创建自己的定制字体。其 Name 属性是 AnsiString。在 Delphi 中，可以这样设置 Name 属性：

```
Canvas.Font.Name := 'Arial';
```

TFont 对象也有一个 Style 属性，该属性是一个集合。通过 TFontStyles 数据类型，可以访问这个属性，预定的值的名称如下：

- FsBold
- FsItalic
- FsUnderline
- fsStockOut

例如我们可以用下面的程序来设置 Font 对象的 Style 属性：

```
Canvas.Font.Style := [fsBold， fsUnderline];
```

8.2.4 PenPos 属性

PenPos 属性是一个非常简单的属性。PenPos 就是画布或绘画平面上活动画笔的当前位置的位置标志符。画笔位置是以点的结构贮存的，可以通过下面的方法访问：

```
Canvas.PenPos.x;
Canvas.PenPos.y;
```

可以读出当前画笔的位置，如下列代码所示：

```
my.x := Canvas.PenPos.x;
my.y := Canvas.PenPos.y;
```

这个属性是一条用 LineTo 方法划出的线的起点。也可以修改 PenPos，如下列代码所示：

```
Canvas.PenPos.x := my.x;
Canvas.PenPos.y := my.y;
```

设置 PenPos 属性等同于调用 MoveTo 方法。特别是在我们需要连续地处理一些点的时候，使用这个属性是非常重要的。

8.2.5 CopyMode 属性

CopyMode 属性与画笔的 Mode 属性相似。这个属性决定了一个图像是如何从一幅画布复制到另一幅画布上去的。设置 CopyMode 可以影响将图形图像画到画布上的方式。可以用 CopyMode 属性来创作出许多不同的效果，包括像合并图像和将不同 CopyMode 的多个图像相结合，从而使位图的部分区域变得透明这样的特殊效果。表 8.5 中列出了可以使用的 CopyMode 属性的所有可用的值。

表 8.5　CopyMode

CopyMode	描述
cmBlackness	将画布上的目的矩形涂成黑色
cmDstInvert	将画布上的图像翻转而忽略源图像
cmMergeCopy	使用布尔“和”（AND）算符将画布上的图像与源位图结合起来
cmMergePaint	使用布尔“或”（OR）算符将翻转的源位图与画布上的图像结合起来
cmNotSrcCopy	将翻转的源位图复制在画布上
cmNotSrcErase	使用布尔“或”（OR）算符将画布上的图像与源位图结合起来并将此结果翻转
cmPatCopy	将源模式复制到画布上
cmPatInvert	使用布尔“异或”（XOR）算符将源模式与画布上的图像结合起来
cmPatPaint	使用布尔“或”（OR）算符将翻转的源位图与源模式相结合，再使用布尔“或”（OR）算符将此运算的结果与画布上的图像结合起来
cmSrcAnd	使用布尔“和”（AND）算符将画布上的图像与源位图结合起来
cmSrcCopy	将源位图复制到画布上
cmSrcErase	将画布上的图像翻转并用布尔“和”（AND）算符将此结果与源位图结合起来
cmSrcInvert	用布尔“异或”（XOR）算符将画布上的图像与源位图结合起来
cmSrcPaint	用布尔“或”（OR）算符将画布上的图像与源位图结合起来
cmWhiteness	将画布上的目的矩形涂成矩形

8.2.6　Pixels 属性

Pixels 属性使可以读出一个特定位置的像素颜色，也可以改变这个像素的颜色。例如下面的程序将会改变对应像素的颜色：

```
Canvas.Pixels[5][25] := clBlue;
```

该属性是 Canvas 对象中的一个非常重要的属性，利用它，我们可以实现许多特殊的功能。例如下面的程序就可以实现图像的淡出效果，即图像从没有到模糊到清晰的过程：

```
var ImageSave:TImage;
    i,j,k,l:longint;
begin
    ImageSave := TImage.Create(self);
    ImageSave.Picture.LoadFromFile('E:\TEMP\Tsinghua.bmp');
    for i:=0 to ImageSave.Picture.Width-1 do
      for j:=0 to ImageSave.Picture.Height-1 do
      begin
          ImageShow.Canvas.Pixels[i,j] := 0;
      end;

    for l:= 0 to 15 do
```

```
begin
for i:=0 to ImageSave.Picture.Width-1 do
   for j:=0 to ImageSave.Picture.Height-1 do
   begin
      if ImageShow.Canvas.Pixels[i,j] < ImageSave.Canvas.Pixels[i,j] then
      begin
       ImageShow.Canvas.Pixels[i,j]
              := ImageShow.Canvas.Pixels[i,j]+16*256+16*256*256+16;
       if ImageShow.Canvas.Pixels[i,j] > ImageSave.Canvas.Pixels [i,j] then
           ImageShow.Canvas.Pixels[i,j] := ImageSave.Canvas.Pixels [i,j];
      end;
     end;
   Application.ProcessMessages;
end;
ImageSave.Free;
```

8.3 TCanvas 的方法

以前接触过图形编程的读者，肯定会对许多绘图函数都比较熟悉。实际上，在 Delphi 中，也为 TCanvas 对象定义了一些类似的方法（见表 8.6）。由于没有必要对每个方法的参数以及使用方法都进行详细的介绍，因此如果希望深入地理解它们，最好的办法就是进行应用。

表 8.6 TCanvas 方法

函数调用	描述
Arc(X1， Y1， X2， Y2， X3， Y3， X4， Y4)	提供一段弧
Chord(X1， Y1， X2， Y2， X3， Y3， X4， Y4)	提供一段弧，其两端点间有连线
CopyRect(Dest，Canvas，Source)	将另一个画布上的部分图像转移到 Tcanvas 对象的图像上
Draw(X， Y， Graphic)	在画布上画图形图像
Ellipse(X1，Y1，X2，Y2，)	提供一个椭圆
FillRect(Rect)	填充一个矩形
FloodFill(X，Y，Color，FillStyle)	填充一个闭合区域
FrameRect(Rect)	画出矩形的边界
LineTo(x，y)	绘制直线
MoveTo(x，y)	移动画笔位置
Pie(X1， Y1， X2， Y2， X3， Y3， X4， Y4)	提供一个馅饼形区域
Polygon(Points:array of TPoints)	提供一个多边形
PolyLine(Points: array of TPoints)	连接画布上的一系列点
Rectangle(X1，Y1，X2，Y2);	提供一个矩形

续表

函数调用	描述
RoundRect(X1， Y1， X2， Y2， X3， Y3)	提供一个四角为圆弧的矩形
StretchDraw(Rect，Graphic)	配合画布上的图形
TextHeight(constText: String)	决定一个字符串在图像中占据的高度
TextOut(X，Y，Text)	在画布上写一个字符串
TextRect(Rect，X，Y，Text)	在一个有限矩形空间中写字符串
TextWidth(Text)	决定一个字符串在图像中占据的长度

8.4　制作 WokPaint 程序

到目前为止，我们已经介绍了如何使用 TCanvas 对象的许多属性和方法，但是还只是比较简单地介绍。下面将通过一个稍微复杂一些的应用程序来演示如何在应用程序中使用 TCanvas 对象的属性和方法。

我们的目的是制作一个基本具备 Windows 下面的 MsPaint 程序功能的软件。这个软件我们平时会经常用到，相信所有的计算机用户都已经对如何使用这个程序非常熟悉了。通过上面的介绍，我们知道 TCanvas 对象有很多基本方法和 MsPaint 中的功能很相似，那么大家就会认为利用它制作一个作图的程序应该不是很难。实际上，从这个程序的编制就可以看出，即使非常简单的程序，如果要把它做得尽量完善，也是需要花费很多精力的。下面我们就开始这项工作。

8.4.1　确定工作控件

首先需要确定的是使用 TImage 控件还是使用 TPaintBox 控件，请读者用下面的代码作一个试验。

```
var i:integer;
begin
      for i:=0 to 100 do
      begin
        Control1.Canvas.MoveTo(5, y+I);
        Control1.Canvas.LineTo(300,y+i);
       Sleep(100);
       Application.ProcessMessages();
      End;
End;
```

把上面程序中的 Control1 改成用来作试验的控件的名称，比如如果用 TImage 控件来作

试验，就把它改成 Image1（在默认的情况下）。在作试验的过程中，你要注意屏幕的刷新情况，看看是否有闪烁；另外还可以在不同的程序之间来回切换，看看程序上的图形是否有保留。

通过上面的试验，我们会发现 TImage 控件和 TPaintBox 控件都具有自己的优点和缺点。

- TImage 控件：优点是能够保持在上面作的图形，但是在连续作图时，屏幕闪烁得相当厉害。
- TPaintBox 控件：优点是在连续作图时屏幕不会闪烁，但是在我们切换窗口之后，这个控件的图形都消失了。

所以，单独使用上面的任何一个控件都不能达到完美的目的。于是我们试图把两者的优点结合起来，采用了下面的思路：

(1) 用 TPaintBox 控件来进行绘图，用 Timage 控件来保存图片。

(2) 当窗口被覆盖或者其他原因可能会导致 TPaintBox 控件上的一部分图片不可见时，用 TImage 控件中保存的图片来重新覆盖 TPaintBox 控件的 Canvas 对象。

于是我们需要使用两个控件，一个称为 CurBitmap，是一个 TImage 控件；一个是 ImgPicture，是一个 TpaintBox 控件。它们的安排如下。

(1) 在窗体上放置一个 TPanel 控件，用来作为这两个控件的容器。在 TPanel 控件上放置一个 TImage 控件，并把它的 Visible 属性设置成‘ False ”。因为这个控件要用来保存 ImgPicture 控件上的图形，所以它的尺寸不能比 ImgPicture 控件小。因此在程序中使用了 TImage 控件的 Contraints 属性，将它的 MinWidth 和 MinHeight 分别设置为 ImgPicture 控件的宽度和高度。

(2) 在 Tpanel 控件上放置一个 TScrollBox 控件，因为要作的图片不见得一定比应用程序的窗口小，所以使用这个控件来滚动该图片。然后在该控件上放置一个 TPaintBox 控件，也就是上面所说的 ImgPicture 控件。

为了实现上面所说的功能，需要编写一些代码，用来保存图像和恢复图像。我们使用下面的过程来保存图像：

```
procedure TForm1.KeepPic;
begin
with Curbitmap do
begin
    Canvas.CopyMode := cmSrcCopy;
    Canvas.CopyRect(ImgPicture.Canvas.ClipRect ,
                    ImgPicture.Canvas,
                    ImgPicture.Canvas.ClipRect );
end;
end;
```

它的基本功能就是利用 CopyRect 方法，把 ImgPicture 控件上的图像复制到 CurBitmap 控件中。然后编写 ImgPicture 控件的 OnPaint 事件，代码如下：

```
procedure TForm1.ImgPicturePaint(Sender: TObject);
begin
ImgPicture.Canvas.CopyMode := cmSrcCopy;
ImgPicture.Canvas.CopyRect(curbitmap.Canvas.ClipRect,
                           CurBitmap.Canvas,
                           curBitmap.Canvas.ClipRect);
end;
```

通过上面的两个过程以及在适当时候对它们的调用，我们已经完全可以解决上面提到的屏幕闪烁和图像保留问题了。

8.4.2　程序界面的设计

有了前面介绍的大量控件的使用基础之后，普通的程序界面设计就不是什么很大的问题了。我们主要是模仿 MsPaint 应用程序的界面，构造了如图 8.6 所示的界面。

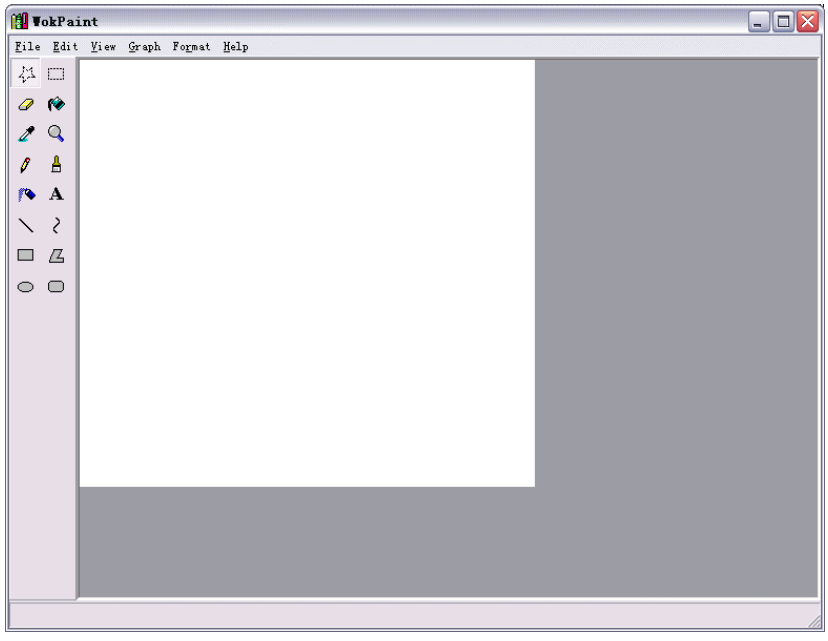

图 8.6　WokPaint 程序的主窗口

在这个窗口上有一个菜单栏、一个工具箱，一个状态条和工作空间。在设计左边工具箱的时候，使用了一个 TPanel 控件和一个 TToolBar 控件相结合的形式，把 TToolbar 控件放置到 TPanel 上，并把 TToolBar 控件的 Align 属性设置成 alClient。

说明：

如果是为了商业目的而开发应用程序，则需要自己去设计很多在程序中可能会使用到的

图像和图标。

在这里，我们把工具栏中的所有按钮都设置成一组，这样的结果就使得在某个时刻只能按下其中的一个按钮。这里会使用到 ToolButton 控件的 Down、Grouped 和 Style 属性。

8.4.3 程序编写的基本思路

在开始编写程序之前，应该考虑一下程序编写的基本思路，虽然这个思路不可能在一开始就是成熟的，在后面的过程中还会多次对它进行修改。在图 8.6 所示的程序中，工具箱中共有 16 个按钮，它们代表了 16 种功能，这个程序的目的就是要实现这些功能以及一些辅助功能。

通过分析，我们把程序功能分成三类：

- 当我们在 ImgPicture 控件上按下鼠标的时候，意味着这个功能马上就要实现。比如填充工具，它就会填充所有和单击点的颜色相同的点，直到遇到不同的颜色为止。
- 当我们在 ImgPicture 控件上移动鼠标的时候，该类功能同时实现。比如橡皮擦，它将擦除所有经过的点。
- 当我们在 ImgPicture 控件上松开鼠标的时候，该类功能才能实现。比如矩形工具，当我们松开鼠标的时候，才最后确定了矩形的范围，从而才能画出该矩形。

那么下面我们来具体分析这三类功能的实现方法。

对于第一类功能，在思路上并不复杂，只要处理 ImgPicture 控件的 OnMouseDown 事件就可以了，可以使用下面的程序框架：

```
procedure TForm1.ImgPictureMouseDown(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
    DownDrawShape(X,Y);
end;
```

程序中的参数（X,Y）代表了当前鼠标的位置，这个位置是在 Sender 对象的坐标系上的。Button 参数则代表了当前鼠标按下的键。Shift 参数代表是否同时按下了键盘上的一些功能键，比如 Ctrl 键。

对于第二类功能，我们不仅要处理 OnMouseDown 事件，而且需要处理 OnMouseMove 事件。它们通常可以采用下面的程序框架：

```
procedure TForm1.ImgPictureMouseDown(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  DownDrawShape(X,Y);
  Drawing := True;
  ImgPicture.Canvas.MoveTo(X, Y);
```

```
    Origin := Point(X, Y);
    MovePT := Point(x,y);
end;

procedure TForm1.ImgPictureMouseMove(Sender: TObject; Shift: TShiftState; X, Y: Integer);
begin
if Drawing then
begin
    MovePt := Point(X, Y);
    MoveDrawShape([Origin,MovePt]);
end;
end;
```

上面的程序在 OnMouseDown **中先绘制需要绘制的部分，然后记录鼠标的起始点。在** OnMouseMove **事件中，要记录鼠标的移动点，并根据起始点和移动点绘制此时需要绘制的图形。它们是相关联着的。**

第三类功能的思路要稍微复杂一些，我们通常需要处理三个事件，它们一般采用下面的程序框架：

```
procedure TForm1.ImgPictureMouseDown(Sender: TObject; Button: TMouseButton;
    Shift: TShiftState; X, Y: Integer);
begin
    Drawing := True;
    ImgPicture.Canvas.MoveTo(X, Y);
    Origin := Point(X, Y);
    MovePT := Point(x,y);
end;

procedure TForm1.ImgPictureMouseMove(Sender: TObject; Shift: TShiftState; X, Y: Integer);
begin
if Drawing then
begin
    DrawFrame([Origin, MovePt]);
    MovePt := Point(X, Y);
    DrawFrame([Origin, MovePt]);
end;
end;

procedure TForm1.ImgPictureMouseUp(Sender: TObject; Button: TMouseButton;
    Shift: TShiftState; X, Y: Integer);
begin
    DrawFrame([Origin,Point(x,y)]);
```

```
    DrawShape([Origin, Point(X, Y)], pmCopy);{ draw the final shape }
    UpDrawShape(X,Y);
    KeepPic;
  end;
```

上面的程序的思路是在 OnMouseDown 事件中设置一个标志 Drawing，说明现在进入了绘图状态，记录鼠标的起始点。当鼠标移动时，开始绘制图形的轮廓，并记录当前的移动点。在松开鼠标时，绘制图形完毕，并把当前新绘制的图形保存到 CurBitmap 控件中去。

从思路和结构上说，我们的程序基本上已经比较完善了，但是还需要一些用来记录操作和选项的中间变量。首先，我们声明了一个用来记录当前进行的绘图操作的类型，称为 DrawingTool，定义如下：

```
type
  TDrawingTool = (dtLine, dtRectangle, dtEllipse, dtRoundRect,
                  dtPolyLine,dtArc,dtChord,dtSelect,dtDrop,
                  dtFill,dtZoom,dtPen,dtBrush,dtPentong,dtText,
                  dtMove,dtErace);
```

同时在主窗口的声明部分，声明了一个 DrawingTool 变量，同时还声明了其他一些变量，如下所示：

```
private
  Drawing: Boolean;                 //记录当前是否处于绘图状态。
  Origin, MovePt: TPoint;               //记录鼠标的起始点和移动点。
  DrawingTool: TDrawingTool;        //记录当前进行的功能类型。
  NonSelected:boolean;              //记录当前是否进行了选择。
  SelectRect:TRect;                     //记录当前选择的区域。
  { Private declarations }
public
  BrushStyle:TBrushStyle;           //记录当前 Brush 的样式。
  BrushColor:TColor;                //记录当前的 Brush 的颜色。
  BrushBitmap:Tbitmap;              //记录在自定义填充格式情况下的图片。

  PenStyle:TPenStyle;               //记录当前 Pen 的样式。
  PenColor:TColor;                  //记录当前 Pen 的颜色。
  PenWid:Integer;                   //记录当前 Pen 的宽度。
  PenMode:TPenMode;                 //记录当前 Pen 的模式。

  BLineStyle,BLineWid:integer;      //记录绘图刷工具的类型和宽度。

  { Public declarations }
end;
```

然后，统一编写了一个用于工具箱中的按钮的 OnClick 事件，代码如下所示：

```
procedure TForm1.BtnClick(Sender: Tobject);
var Selected:boolean;
begin
   Drawing:=False;
   Selected:=Not NonSelected;
   NonSelected:=True;
   if (Sender as TToolButton).Tag <=0 then
      exit;
   case (Sender as TToolButton).Tag of
      1:
         DrawingTool:=dtSelect;
      2:
         DrawingTool:=dtErace;
      3:
         DrawingTool:=dtFill;
      4:
         DrawingTool:=dtDrop;
      5:
         DrawingTool:=dtZoom;
      6:
         DrawingTool:=dtPen;
      7:
         DrawingTool:=dtBrush;
      8:
         DrawingTool:=dtPenTong;
      9:
         DrawingTool:=dtText;
      10:
         DrawingTool:=dtLine;
      11:
         DrawingTool:=dtArc;
      12:
         DrawingTool:=dtRectangle;
      13:
         DrawingTool:=dtPolyLine;
      14:
         DrawingTool:=dtEllipse;
      15:
         DrawingTool:=dtRoundRect;
      16:
        begin
```

```
            DrawingTool:=dtMove;
            if Selected then
                NonSelected:=False;
          end;
    end;
    if Selected and NonSelected then
    begin
      With ImgPicture.Canvas do
      begin
        Pen.Style :=psDashDot;
        Pen.Mode := pmNotXor;
        Pen.Width:=1;
        Brush.Style := bsClear;

        Rectangle(SelectRect.Left,SelectRect.Top,SelectRect.Right,SelectRect.Bottom);
      end;
    end;
end;
```

上面的程序很好地演示了如何利用 Sender 参数为一组控件统一编写事件处理程序。通过这个程序，我们可以集中地处理关于 DrawingTool 变量的事情。虽然从代码的行数上也许节省不了多少，但是它使得我们可以有一个集中的思路来考虑该变量的事情。

8.4.4 直线、矩形、椭圆和圆角矩形的绘制

这是绘图程序中最简单的四个功能，下面就来看看如何实现它们。它们都是属于我们上面分析的第三类功能。所以我们需要使用 ImgPicture 控件的四个鼠标事件。基本结构我们在上面已经介绍了。下面我们讨论一下在各个事件中，我们都完成了哪些事情。

在 OnMouseDown 事件中，我们完全采用了上面介绍的程序结构，没有什么变化。在 OnMouseMove 事件中，关键是画出我们要绘制的图形轮廓，并能随着鼠标的移动进行更新。这个功能是由 DrawFrame 过程完成的。这个过程的定义如下：

```
procedure TForm1.DrawFrame(Points: array of TPoint);
var R:TRect;
     Ps:Array [1..4] of TPoint;
begin
with ImgPicture.Canvas do
begin
  Brush.Style := bsClear;
  Pen.Mode := pmNotXor;
  Pen.Width := 1;
```

```
    Pen.Style :=psDashDot;
    case DrawingTool of
      dtLine:
      begin
        MoveTo(Points[0].x, Points[0].Y);
        LineTo(points[1].X, points[1].Y);
      end;
      dtRectangle:
        Rectangle(Points[0].X, Points[0].Y, points[1].X, points[1].Y);
      dtEllipse:
        Ellipse(Points[0].X, Points[0].Y, points[1].X, points[1].Y);
      dtRoundRect:
        RoundRect(Points[0].X, Points[0].Y, points[1].X, points[1].Y,
                  (Points[0].X - Points[1].X) div 2,
                  (points[0].Y - points[1].Y) div 2);
    ... //还有其他的处理程序。
    end;
  end;
end;
```

这个过程的关键是采用了 Pen 对象的 pmNotXor 模式。在这种模式下，在同一个位置的两次绘画将会相互抵消，从而恢复没有进行绘制之前的状态。利用上面的这个过程和前面介绍的两个鼠标事件，当我们在程序中拖动鼠标的时候，结果将会如图 8.7 所示。

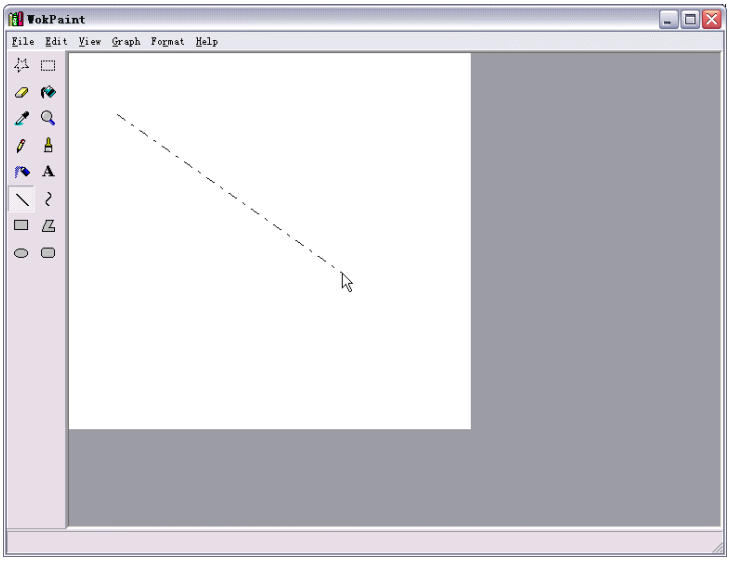

图 8.7　DrawFrame 过程在程序中画出的图形轮廓

为了区别最终的结果和中间的轮廓，在上面的过程中，当绘制轮廓的时候，采用了 Pen

对象的 psDashDot 类型。

然后需要在 OnMouseUp 事件中绘制最后的图形，这里采用了一个称为 DrawShape 的过程，该过程的定义如下：

```
procedure TForm1.DrawShape(Points: array of TPoint; PMode: TPenMode);
var R:TRect;
    Ps:Array [1..4] of TPoint;
    bit:TImage;
begin
with ImgPicture.Canvas do
begin
  Pen.Width := PenWid;
  Pen.Style := PenStyle;
  Pen.Color := PenColor;
  Pen.Mode := PenMode;
  case DrawingTool of
    dtLine:
    begin
      MoveTo(Points[0].x, Points[0].Y);
      LineTo(points[1].X, points[1].Y);
    end;
    dtRectangle:
    begin
      if BrushBitmap.Empty then
      begin
        Brush.Color := BrushColor;
        brush.Style := BrushStyle;
      end
      else
        Brush.Bitmap := BrushBitmap;
      Rectangle(Points[0].X, Points[0].Y, points[1].X, points[1].Y);
    end;
    dtEllipse:
    begin
      if BrushBitmap.Empty then
      begin
        Brush.Color := BrushColor;
        brush.Style := BrushStyle;
      end
      else
        Brush.Bitmap := BrushBitmap;
      Ellipse(Points[0].X, Points[0].Y, points[1].X, points[1].Y);
```

```
        end;
        dtRoundRect:
        begin
          if BrushBitmap.Empty then
          begin
            Brush.Color := BrushColor;
            brush.Style := BrushStyle;
          end
          else
            Brush.Bitmap := BrushBitmap;
          RoundRect(Points[0].X, Points[0].Y, points[1].X, points[1].Y,
                    (Points[0].X - Points[1].X) div 2,
                    (points[0].Y - points[1].Y) div 2);
        end;
...     //还有其他处理程序。
      end;
    end;
    end;
```

这段程序的基本思路和上面的 DrawFrame 基本相似，不同的是完全采用了变量 BrushColor 等中的值来作为 Brush 和 Pen 对象的对应属性，所以画出的也就是我们最后希望的结果。这两个过程中比较重要的是各个坐标点的计算，在绘图程序中，这是十分重要的，在后面的程序中读者会有更深的体会。

通过上面的两个过程，我们已经实现了 WokPaint 程序中的四个基本功能，此时绘制的图形如图 8.8 所示。

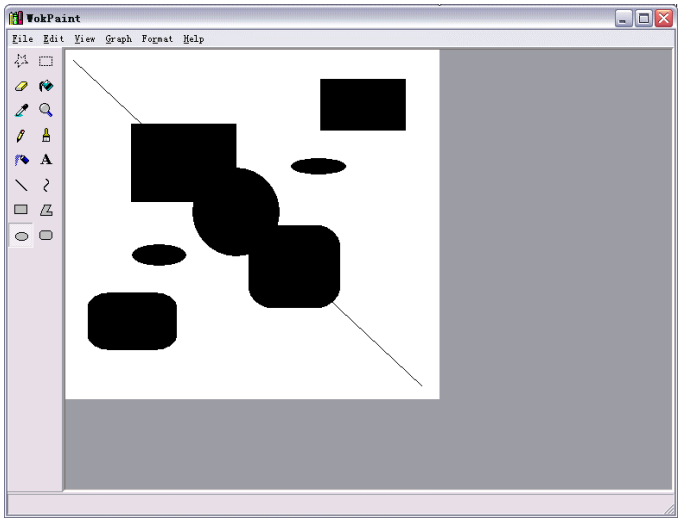

图 8.8　WokPaint 程序中绘制的简单图形

8.4.5 填充模式和画笔模式的设置

在前面曾经提到，我们使用了几个变量来存储当前采用的 Brush 和 Pen 对象的模式。在我们的程序中应该能够对它们进行设置。关于它们的设置是通过 Format 菜单中的两个命令进行的。

下面我们先来看看关于画笔模式的设置。编写下面的菜单命令：

```
procedure TForm1.PenStyle1Click(Sender: TObject);
begin
if Form3.ShowModal = mrOK then
begin
   PenColor:=Form3.PenColor;
   PenStyle:=Form3.PenStyle;
   PenMode:=Form3.PenMode;
   PenWid:=Form3.PenWid;
end;
```

从程序中看，这个过程还是比较简单的，重要的是我们利用 Form 对象的 ShowModal 方法进行了模式化显示。如果只是普通地显示一个窗体，可以使用下面的语句：

```
Form3.show;
```

这两种显示方式的最大区别就是模式化显示 ShowModal 是有返回值的，而且不关闭被显示的窗口就不能回到程序的主窗口；而后者则不是如此。

前面提到过，Pen 对象的关键属性主要是：Color、Width、Mode 和 Style，所以在这里设计了一个窗口；也就是上面的 Form3 来处理这些属性，该窗口如图 8.9 所示。

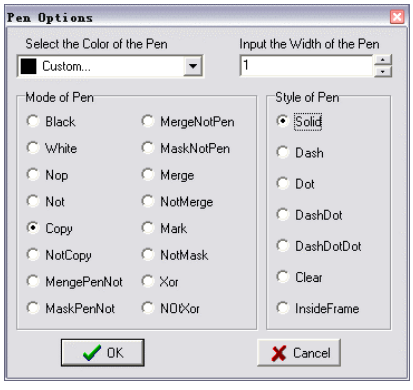

图 8.9 WokPaint 程序中的画笔选项窗口

在这个窗口上我们使用了一个 ColorBox 控件、一个 LabelEdit 控件和两个 RadioGroup 控件来处理 Pen 对象的四个属性。该窗口中的代码如下所示：

```
unit Unit3; //用于处理 Pen 对象的选项的单元。

interface

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

type
  TForm3 = class(TForm)
    RadioGroup1: TRadioGroup;
    RadioGroup2: TRadioGroup;
    LabeledEdit1: TLabeledEdit;
    UpDown1: TUpDown;
    ColorBox1: TColorBox;
    Label1: TLabel;
    BitBtn1: TBitBtn;
    BitBtn2: TBitBtn;
    procedure FormShow(Sender: TObject);
    procedure ColorBox1Change(Sender: TObject);
    procedure RadioGroup2Click(Sender: TObject);
    procedure RadioGroup1Click(Sender: TObject);
    procedure LabeledEdit1Change(Sender: TObject);
  private
    { Private declarations }
  public
    PenColor:Tcolor;
    PenStyle:TPenStyle;
    PenMode:TPenMode;
    PenWid:integer;
    { Public declarations }
  end;

var
  Form3: TForm3;

implementation

uses Unit1;

{$R *.dfm}
```

```
procedure TForm3.FormShow(Sender: TObject);
begin
PenColor:=Form1.PenColor;
PenStyle:=Form1.PenStyle;
PenMode:=Form1.PenMode;
PenWid:=Form1.PenWid;
ColorBox1.Selected := PenColor;
Updown1.Position := PenWid;
RadioGroup1.ItemIndex := integer(PenStyle);
RadioGroup2.ItemIndex := integer(PenMode);
end;

procedure TForm3.ColorBox1Change(Sender: TObject);
begin
PenColor:=ColorBox1.Selected;
end;

procedure TForm3.RadioGroup2Click(Sender: TObject);
begin
PenMode:=TPenMode(RadioGroup2.ItemIndex);
end;

procedure TForm3.RadioGroup1Click(Sender: TObject);
begin
PenStyle:=TPenStyle(RadioGroup1.ItemIndex);
end;

procedure TForm3.LabeledEdit1Change(Sender: TObject);
begin
  PenWid:=UpDown1.Position;
end;


end.
```

这段程序的主要编写思路就是在窗口显示时取得主窗口中相关的值，然后在窗口这里显示出来，在用户改变窗口中的相应设置时，把这些设置保存下来。

需要注意的是，在获取主窗口相关值时使用的是 OnShow 事件，而不是 OnCreate 事件，这是由它们的发生顺序决定的。OnCreate 事件发生在窗口创建时，在默认的情况下是发生在程序刚开始运行时；而 OnShow 事件则发生在窗口每次进行显示的时候。在程序中不会只调用一次该窗口，所以我们选用 OnShow 事件。关于 Form 对象的事件发生顺序，请参考前面介绍窗体对象时候所讲述的内容。

说明：

我们这里定义了相同的一套变量，主要是为了增加程序的可读性。

下面的问题是关于 Brush 对象的相关设置的完成。同样，我们也定义了类似的一个窗口，如图 8.10 所示。

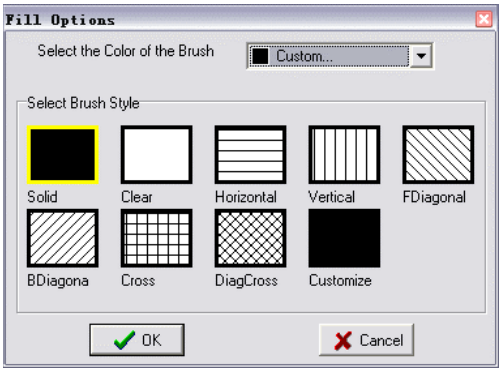

图 8.10　Brush 对象的设置选项

这个程序的代码要稍微复杂一些，有以下两个原因。

- 我们在窗口中加入了一些 TImage 控件，用来代表每种填充类型的示例。因而，在对类型进行选择的时候就不如使用 RadioGroup 控件那么顺利。
- 在 Brush 的 Style 的定义中，还有一个自定义模式的问题。从下面的代码中可以看出，由于自定义模式的存在，我们不得不进行了更多的编程工作。

当用户选择其中的某个模式时，该模式的图形应该相应地变化，特别是当用户选择自定义模式时，我们需要显示一个 OpenDialog 控件，即打开文件对话框。该控件位于 Delphi 控件面板中的 Dialogs 选项卡中。让用户选择需要的图片文件，并用新的模式来绘制它的示例图片。整个单元的完整代码如下所示：

```
unit Unit2; //用于处理 Brush 对象选项的单元。

interface

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

type
  TForm2 = class(TForm)
    GroupBox1: TGroupBox;
    Image1: TImage;
     ...
```

```
    Image9: TImage;
    Label1: TLabel;
     ...
    Label10: TLabel;
    ColorBox1: TColorBox;
    BitBtn1: TBitBtn;
    BitBtn2: TBitBtn;
    OpenDialog1: TOpenDialog;
    procedure FormCreate(Sender: TObject);
    procedure FormDestroy(Sender: TObject);
    procedure Image1Click(Sender: TObject);
     ...
    procedure Image9Click(Sender: TObject);
    procedure FormShow(Sender: TObject);
    procedure ColorBox1Change(Sender: TObject);
  private
    procedure InitStyle(Img:TImage;Pattern:TBrushStyle;flag:boolean);
    procedure InitImage();
    { Private declarations }
  public
    BrushStyle:TBrushStyle;
    BrushColor:TColor;
    BrushBitmap:TBitmap;

    { Public declarations }
  end;

var
  Form2: TForm2;

implementation

uses Unit1;

{$R *.dfm}

{ TForm2 }

procedure TForm2.InitStyle(Img: TImage;Pattern:TBrushStyle;flag:boolean);
begin
  if Flag then
      Img.Canvas.Pen.Color := clYellow
```

```
    else
        Img.Canvas.Pen.Color := clblack;
    Img.Canvas.Pen.Width := 4;
    Img.Canvas.Brush.Color := Form1.BrushColor;
    Img.Canvas.Brush.Style := Pattern;
    Img.Canvas.Rectangle(1,1,Img.Width,Img.Height);
end;

procedure TForm2.FormCreate(Sender: TObject);
begin
    BrushBitmap:=TBitmap.Create;
end;

procedure TForm2.FormDestroy(Sender: TObject);
begin
BrushBitmap.Free;
end;

procedure TForm2.Image1Click(Sender: TObject);
begin
BrushStyle:=bsSolid;
BrushBitmap.FreeImage;
InitImage;
end;

procedure TForm2.Image2Click(Sender: TObject);
begin
BrushStyle:=bsClear;
BrushBitmap.FreeImage;
InitImage;
end;

procedure TForm2.Image3Click(Sender: TObject);
begin
BrushStyle:=bsHorizontal;
BrushBitmap.FreeImage;
InitImage;
end;

procedure TForm2.Image4Click(Sender: TObject);
begin
BrushStyle:=bsVertical;
```

```
BrushBitmap.FreeImage;
InitImage;
end;

procedure TForm2.Image5Click(Sender: TObject);
begin
BrushStyle:=bsFDiagonal;
BrushBitmap.FreeImage;
InitImage;
end;

procedure TForm2.Image6Click(Sender: TObject);
begin
BrushStyle:=bsBDiagonal;
BrushBitmap.FreeImage;
InitImage;
end;

procedure TForm2.Image7Click(Sender: TObject);
begin
BrushStyle:=bsCross;
BrushBitmap.FreeImage;
InitImage;
end;

procedure TForm2.Image8Click(Sender: TObject);
begin
BrushStyle:=bsDiagCross;
BrushBitmap.FreeImage;
InitImage;
end;

procedure TForm2.InitImage;
var i:integer;
begin
  if not BrushBitmap.Empty then
  begin
      Image9.Canvas.Brush.Bitmap := nil;
      Image9.Canvas.Brush.bitmap:=BrushBitmap;
      Image9.Canvas.Pen.Color := clYellow;
      Image9.Canvas.Pen.Width := 4;
      Image9.Canvas.Rectangle(1,1,Image9.Width,Image9.Height);
```

```
    end
    else
    begin
        Image9.Canvas.Brush.bitmap:=BrushBitMap;
        initStyle(Image9,TBrushStyle(0),False);
    end;
    Image9.Refresh;
    for i:=0 to 7 do
    begin
        if ((TBrushStyle(i)=BrushStyle) and (BrushBitmap.Empty)) then
            InitStyle((GroupBox1.Controls[i] as TImage),TBrushStyle(i),True)
        else
            InitStyle((GroupBox1.Controls[i] as TImage),TBrushStyle(i),False);
    end;
end;

procedure TForm2.FormShow(Sender: TObject);
begin
    BrushStyle:=Form1.BrushStyle;
    BrushColor:=Form1.BrushColor;
    BrushBitmap.Assign(Form1.BrushBitmap);
    InitImage;
    ColorBox1.Selected := BrushColor;
end;

procedure TForm2.Image9Click(Sender: TObject);
begin
    if OpenDialog1.Execute then
    begin
        BrushBitmap.LoadFromFile(OpenDialog1.FileName);
        InitImage;
    end;
end;

procedure TForm2.ColorBox1Change(Sender: TObject);
begin
BrushColor:=ColorBox1.Selected;
end;

end.
```

在参考上面的程序时需要注意以下三个问题。

- 一个是关于类型的转换，在程序中，把整型转换成了 TBrushStyle 类型。我们在程序中经常会遇到这样的情况。
- 对于容器控件的子控件的引用。对于放置了大量同类控件的容器控件，可以使用它的 Controls 属性来访问放置在它上面的每个控件。这样的好处是可以用同一的代码来处理关于这些控件的所有问题。
- 利用 TBitmap 对象，而 FreeImage 方法可以释放存储在里面的图像。

在下面的窗口中载入了一个自定义类型的 Brush。

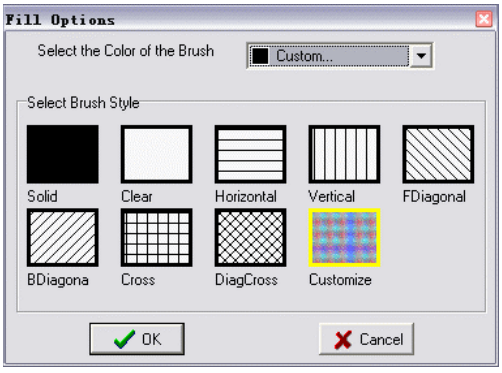

图 8.11　自定义 Brush 类型

在完成了上面的 Brush 和 Pen 的定义之后，我们就可以用 WokPaint 程序的几个简单的功能绘制出更复杂的图片，如图 8.12 所示。

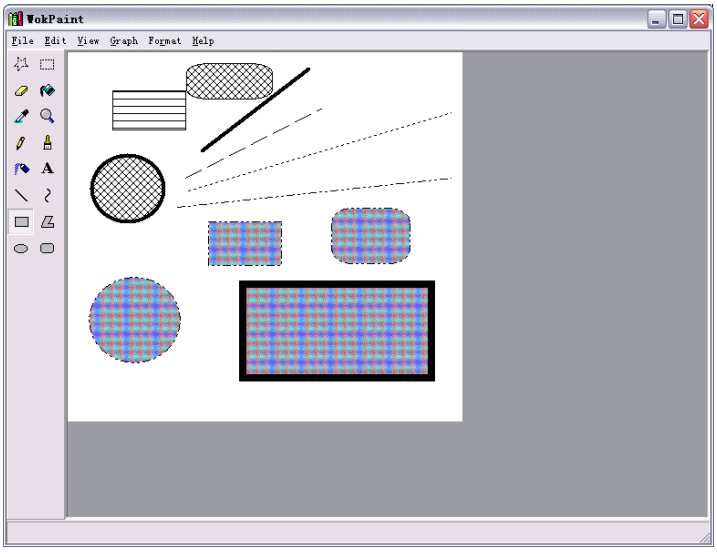

图 8.12　WokPaint 程序中 Pen 和 Brush 对象的不同格式举例

8.4.6　多点连线、弧线和填充

在 WokPaint 程序中，还有几个简单的功能，我们可以比较容易地实现它们：多点连线、弧线和填充功能。

多点连线功能看起来使用的是 Canvas 对象 PolyLine 方法，其实我们在程序中并没有这样做，因为如果使用 PolyLine 方法，需要在获得所有的点之后才能进行绘图。而用户可能更喜欢随着鼠标的单击，这个图形就绘制出来了。我们在 OnMouseDown 事件中完成了该图形的绘制，调用的是 DownDrawShape 过程。有关的代码如下：

```
procedure TForm1.DownDrawShape(x, y: integer);
begin
With ImgPicture.Canvas do
begin
case DrawingTool of
   dtPolyLine:
   begin
      If Drawing then
      begin
         Pen.Mode := PenMode;
         Pen.Style := PenStyle;
         Pen.Color := PenColor;
         Pen.Width := PenWid;
         Brush.Style := bsClear;
         MoveTo(Origin.X,OriGin.Y);
         LineTo(X,Y);
      end;
   end;
...
end;
end;
```

在用户每单击一次鼠标时，我们就把新的点和前面的点连接起来，需要注意的是，和实现 Line 功能一样，我们在调用 LineTo 方法的时候，一定要调用 MoveTo 方法来移动 Canvas 的光标位置。在图 8.13 所示的图形中，我们绘制了几种不同线型的 PolyLine。

弧线功能调用的是 Canvas 对象的 Arc 方法，这个功能的实现和 Line 等功能的实现很类似，相关的代码如下：

```
procedure TForm1.DrawFrame(Points: array of TPoint);
//绘制轮廓边框。
var R:TRect;
```

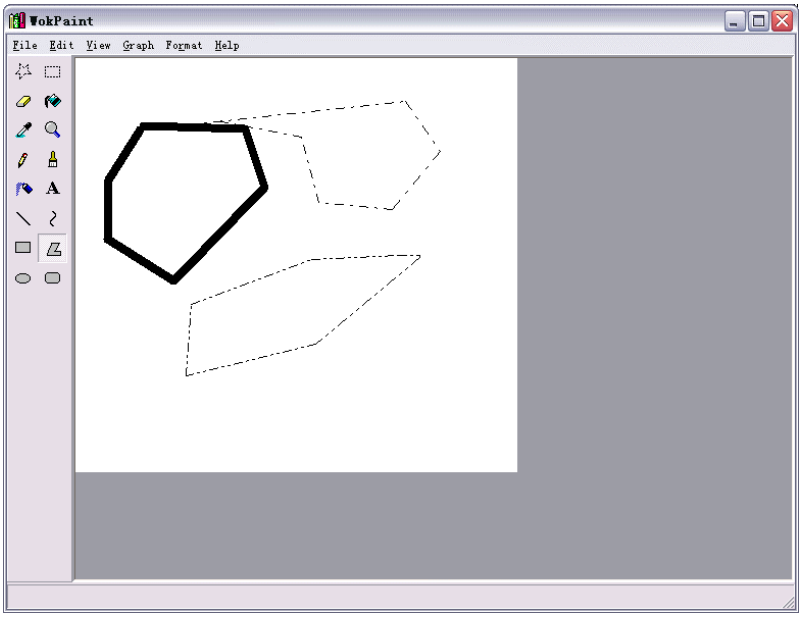

图 8.13　WokPaint **程序绘制的** PolyLine

```
    Ps:Array [1..4] of TPoint;
begin
with ImgPicture.Canvas do
begin
  Brush.Style := bsClear;
  Pen.Mode := pmNotXor;
  Pen.Width := 1;
  Pen.Style :=psDashDot;
  case DrawingTool of
    dtArc:
    begin
      R:=Rect(Points[0].X,Points[0].Y,Points[1].x,Points[1].Y);
      Arc(R.Left, R.Top, R.Right, R.Bottom, R.right, R.bottom,
          R.left,R.Top);
    end;
...
  end;
end;
end;
procedure TForm1.DrawShape(Points: array of TPoint; PMode: TPenMode);
var R:TRect;
    Ps:Array [1..4] of TPoint;
    bit:TImage;
```

```
begin
with ImgPicture.Canvas do
begin
   Pen.Width := PenWid;
   Pen.Style := PenStyle;
   Pen.Color := PenColor;
   Pen.Mode := PenMode;
   case DrawingTool of
      dtArc:
      begin
         R:=Rect(Points[0].X,Points[0].Y,Points[1].x,Points[1].Y);
         Arc(R.Left, R.Top, R.Right, R.Bottom, R.right, R.bottom,
             R.left,R.Top);
      end;
...
   end;
end;
end;
```

程序的运行结果如图 8.14 所示。

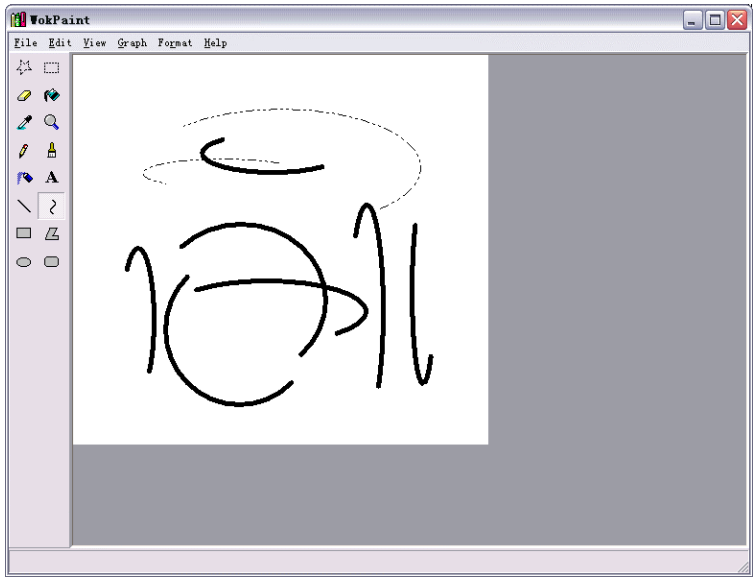

图 8.14　WokPaint 程序绘制的弧线

关于填充的问题，主要是调用 Canvas 对象的 FloodFill 方法，该方法的声明如下：

```
type TFillStyle = (fsSurface, fsBorder);
procedure FloodFill(X, Y: Integer; Color: TColor; FillStyle: TFillStyle);
```

在这个方法的参数中，X，Y 代表的是开始填充的起始点，这个点的颜色非常重要。FillStyle 参数则决定了填充的方式。

- fsSurface 方式：所有和填充点相同颜色的区域将被用 Color 参数指定的颜色所填充，直到遇到不同于 X、Y 坐标指定点的颜色为止。
- fsBorder 方式：从当前点开始，填充所有和 Color 颜色不同的颜色点，直到遇到 Color 参数指定的颜色为止。

显然，在这里我们应该采用 fsSurface 方式。

填充功能是在 OnMouseDown 事件中完成的，相关的代码如下：

```
case DrawingTool of
  dtFill:
  begin
    if BrushBitmap.Empty then
    begin
      Brush.Color := BrushColor;
      brush.Style := BrushStyle;
    end
    else
      Brush.Bitmap := BrushBitmap;
    FloodFill(X,Y,Pixels[X,Y],fsSurface);
  end;
```

通过上面的方法，我们不但可以用 Brush 的各种类型来填充区域，也可以用我们前面介绍过的自定义方式填充。例如在图 8.15 所示的窗口中，我们就用自定义方式填充了利用多点连线功能画出的一个多边形区域。

8.4.7 图像刷、画笔和橡皮擦

这三个功能之所以放在一起介绍，是因为它们都牵扯到一个自定义线型的问题。事实上我们可以根据自己的需要画出任何一种想要的线条。其中的决窍就是不按照绘制普通线条的方法进行绘制。也就是说，我们画出一系列的形状，再将它们连接起来，使之看起来像线条一样，从而创造出自定义的线型。例如，画出一条像是用书法用笔画出来的线条实际上十分简单，可以使用画布的 Polygon 方法，画出所谓的平行四边形。

当用这样的方法绘制一个封闭图形时，该方法会用当前的 Brush 填充形状的内部。所以，我们在绘制自己定制的线条时，可以把 Pen 和 Brush 设置成同一种颜色，那么得到的图形就成为这里的自定义线型。

所以，可以这么说，自定义线型，实际上不是线型，只是一些多边形的组合。所以，在定义自己线型的时候，最关键的是这些多边形的计算。

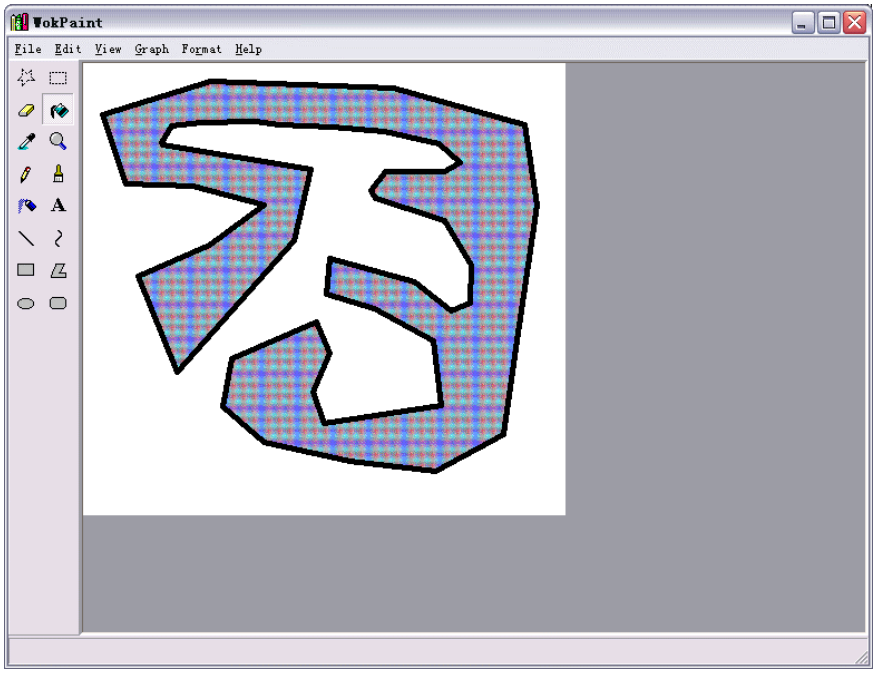

图 8.15　利用自定义类型填充图片

在本程序中图像刷具有以下四种类型：

- 下平行四边形
- 上平行四边形
- 矩形
- 圆形

我们专门定义了一个窗口用来设置关于这些类型以及它们类型的相关选项，这个窗口比较简单，如图 8.16 所示。

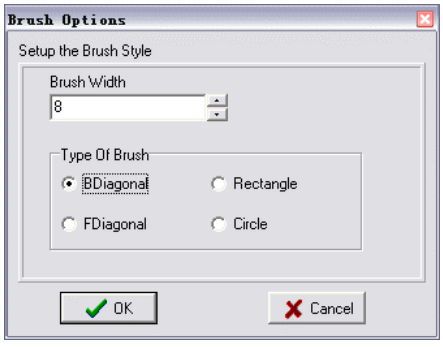

图 8.16　自定义线型的设置选项

说明：

由于我们定义这些线型主要是为了实现图像刷的功能，而图像刷的英文是 Brush，所以也许我们这里的命令不尽合理，但是从用户的角度来讲，能够理解就可以了。

这个窗口中的程序非常简单，代码如下：

```
procedure TForm4.FormShow(Sender: TObject);
begin
BLineStyle:=Form1.BlineStyle;
BLineWid:=Form1.BLineWid;
RadioGroup1.ItemIndex := BLineStyle;
UpDown1.Position := BLineWid;
end;

procedure TForm4.LabeledEdit1Change(Sender: TObject);
begin
BLineWid:=UpDown1.Position;
end;

procedure TForm4.RadioGroup1Click(Sender: TObject);
begin
BLineStyle:=RadioGroup1.ItemIndex;
end;
```

在主窗口中，定义了 BLineStyle 和 BLineWid 两个变量来记录自定义线型的选项。这里主要是对它们的操作。

下平行四边形和上平行四边形类型的自定义线型的实现要容易一些，可以在 OnMouseMove 事件调用的 MoveDrawShape 过程中调用下面的过程：

```
procedure TForm1.DownDrawBrush(Points: array of TPoint;
                               Co:TColor;Style,Wid:integer);
var Ps:Array [1..4] of TPoint;
    Pt:Array [1..6] of TPoint;
    F1,F2,d:integer;
begin
With ImgPicture.Canvas do
begin
  Pen.Color:=Co;
  Brush.Color := Co;
  Pen.Width :=1;
  Pen.Style := psSolid;
  if Drawing then
  begin
```

```
        Case Style of
        0:
          begin
            ps[1] := Points[0];
            ps[2] := Points[1];
            ps[3] := Point(Ps[2].x + Wid,Ps[2].y + Wid);
            ps[4] := Point(Ps[1].x + Wid,Ps[1].y + Wid);
            Polygon(ps);
            Origin:=Points[1];
          end;
        1:
          begin
            ps[1] := Points[0];
            ps[2] := Points[1];
            ps[3] := Point(Ps[2].x - Wid,Ps[2].y + Wid);
            ps[4] := Point(Ps[1].x - Wid,Ps[1].y + Wid);
            Polygon(ps);
            Origin:=Points[1];
          end;
...
    end;
end;
end;
```

从上面的程序看出，对于这种平行四边形的线型，我们只要把鼠标的起始点和移动点坐标进行平移，作为四边形的另外两个点来绘图就可以了。需要注意的是，每次绘制一个平行四边形之后，便把当前的移动点赋给鼠标起始点，只有这样才能随着我们的鼠标的移动，不断地画出曲线。

MoveDrawShape **过程中的调用如下：**

```
case DrawingTool of
dtBrush:
    DownDrawBrush(points,brush.Color,BLineStyle,BLineWid);
```

说明：

我们在定义 DownDrawBrush **过程的时候，采用了一些参数来指定线型、线宽和颜色等，主要是为了能够利用这个过程完成其他的任务。**

在图 8.17 **所示的** WokPaint **程序中，在左边绘制了一些上平行四边形线型，右边绘制了一些下平行四边形线型。**

利用这两种线型，我们可以绘制出类似于美术钢笔画出的线条图形。

图 8.17 WokPaint 程序中的两种不同线型

对于圆形的线型，实现起来比较简单，主要是在 OnMouseDown 事件中画第一个圆，在 OnMouseUp 事件中画最后一个圆，中间的部分，也就是 OnMouseMove 事件中采用画线的方法就可以了。

矩形线型的实现也复杂一些，它的基本思路仍然是在 OnMouseDown 事件中画第一个矩形，在 OnMouseUp 事件中画最后一个矩形，关键的部分是中间的图形实现，此时的移动有八个可能的移动方向，如图 8.18 所示。

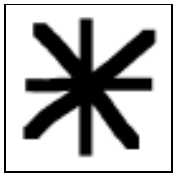

图 8.18 自定义线条类型的可能方向

在利用 PolyGon 方法的时候，一般需要计算 6 个点的坐标值。在上面的程序中，我们总结出了一套用于该种情况的计算方法，把计算减少到了两种情况。程序运行结果表明，这套算法还是不错的。

说明：

这里我们没有直接采用 Rectangle 方法，读者可以自己试用一下。事实证明，当使用 Rectangle 方法的时候，特别是在鼠标移动速度比较快的情况下，很难画出连续的图形。

下面是关于在中间部分，也就是在 OnMouseMove 事件中的绘图代码，其中包含了计算方法：

```
procedure TForm1.DownDrawBrush(Points: array of TPoint;
                               Co:TColor;Style,Wid:integer);
```

```
var Ps:Array [1..4] of TPoint;
    Pt:Array [1..6] of TPoint;
    F1,F2,d:integer;
begin
With ImgPicture.Canvas do
begin
  Pen.Color:=Co;
  Brush.Color := Co;
  Pen.Width :=1;
  Pen.Style := psSolid;
  if Drawing then
  begin
    Case Style of
     ...
    2:
      Begin
        d:=Wid div 2;
        F1:=sign(Points[1].X-Points[0].X);
        F2:=Sign(Points[1].Y-Points[0].Y);
        if F1=0 then
        begin
          ps[1]:=Point(Points[0].X-F2*d,Points[0].Y-F2*d);
          Ps[2]:=Point(Points[0].X+F2*d,ps[1].Y);
          Ps[3]:=Point(Ps[2].X,Points[1].Y+F2*d);
          ps[4]:=Point(Ps[1].X,Ps[3].Y);
          Polygon(ps);
          Origin:=Points[1];
        end
        else if F2=0 then
        begin
          ps[1]:=Point(Points[0].X-F1*d,Points[0].Y-F1*d);
          Ps[2]:=Point(Points[1].X+F1*d,ps[1].Y);
          Ps[3]:=Point(Ps[2].X,Points[1].Y+F1*d);
          ps[4]:=Point(Ps[1].X,Ps[3].Y);
          Polygon(ps);
          Origin:=Points[1];
        end
        else
        begin
          pt[1]:=Point(Points[0].X+F1*d,Points[0].Y-F2*d);
          Pt[2]:=Point(Points[0].X-F1*d,Points[0].y-F2*d);
          Pt[3]:=Point(Points[0].X-F1*d,Points[0].Y+F2*d);
```

```
                pt[4]:=Point(Points[1].x-F1*d,Points[1].Y+F2*d);
                pt[5]:=Point(Points[1].X+F1*d,Points[1].Y+F2*d);
                pt[6]:=Point(Points[1].X+F1*d,Points[1].Y-F2*d);
                Polygon(pt);
                Origin:=Points[1];
              end;
            end;
            3:
            Begin
              Pen.Width := Wid;
              MoveTo(Points[0].x,Points[0].Y);
              Lineto(Points[1].X,Points[1].Y);
              Origin:=Points[1];
            end;
            end;
      end;
  end;
end;
```

这里利用前后移动的 X，Y 的值确定移动方向，并利用 Sign 函数计算中间的坐标点，此时，程序的运行结果如图 8.19 所示。

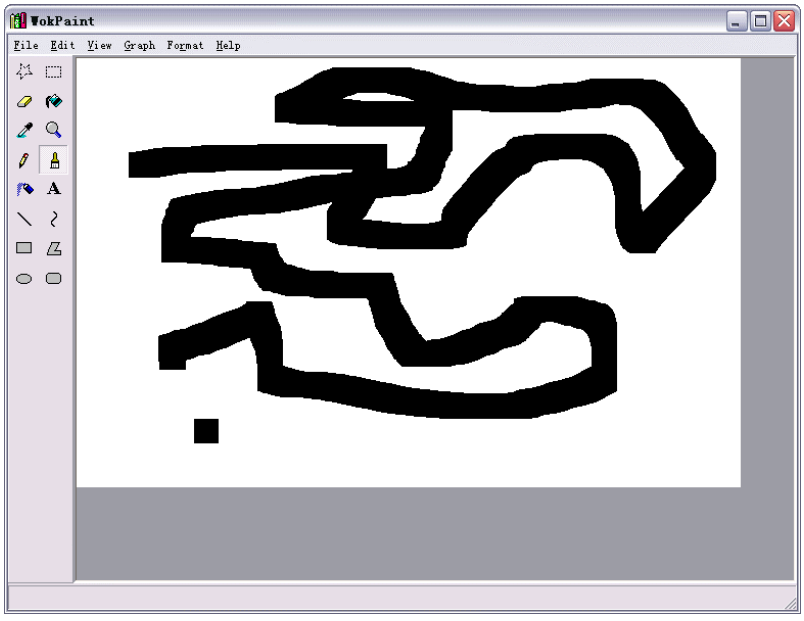

图 8.19　矩形的自定义线型

橡皮擦工具的实现就是利用了上面的矩形线型，所不同的是它总是采用背景的颜色来绘

制图形，这里使用的是白色。它的过程和绘制图像刷是一样的。

铅笔工具分为两种情况，如果铅笔的线宽小于等于 1，那么可以直接使用 Line 方法实现；如果宽度大于 1，那么仍然可以利用上面的矩形线型实现。

下面列出了实现上面两个功能的 MoveDrawShape 中的代码：

```
case DrawingTool of
dtBrush:
    DownDrawBrush(points,brush.Color,BLineStyle,BLineWid);
dtPen:
begin
  if PenWid>1 then
    DownDrawBrush(Points,Pen.Color,2,PenWid)
  else
  begin
    MoveTO(Points[0].X,Points[0].Y);
    Lineto(Points[1].X,Points[1].Y);
  end;
  Origin:=Points[1];
end;
dtErace:
  DownDrawBrush(Points,clWhite,2,EraceWid);
end;
```

在图 8.20 中，演示了这两种功能的运行情况。

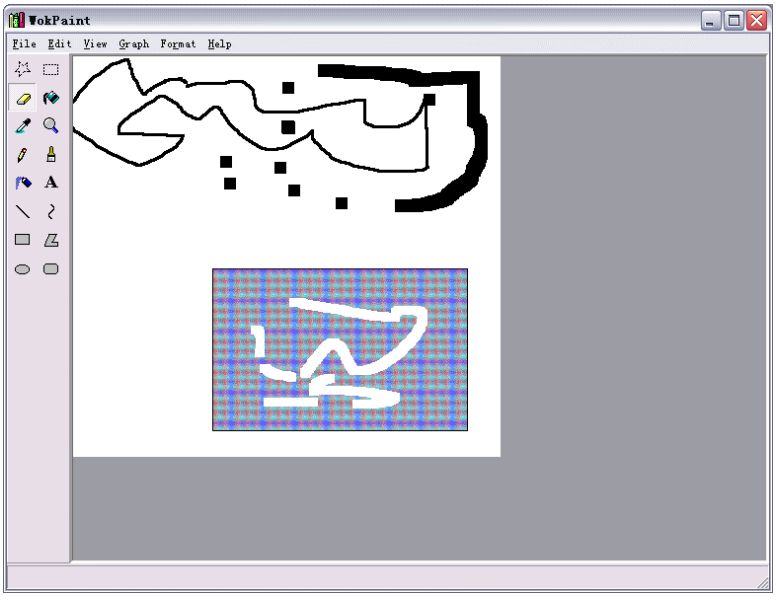

图 8.20　铅笔工具和橡皮擦工具的实现

8.4.8 图形中的文本

处理图像中的文本也是 WokPaint 程序的一个基本功能，它需要使用 Canvas 对象的处理文本方法。我们是在 OnMouseUp 事件中完成该功能的，使用了下面的代码：

```
dtText:
  if Form5.ShowModal = mrOK then
  begin
    ImgPicturePaint(self);
    ImgPicture.Canvas.Font.Assign(Form5.fontdialog1.Font);
    ImgPicture.Canvas.Brush.Style := bsClear;
    si:=ImgPicture.Canvas.TextExtent(Form5.Edit1.Text);
    ImgPicture.Canvas.TextRect(Rect(X,Y,x+si.cx,Y+si.cy ),x,Y,Form5.Edit1.Text);
  end
  else
    ImgPicturePaint(Self);
end;
```

通过一个窗口来获得需要的文本以及字体的格式，该窗口如图 8.21 所示。

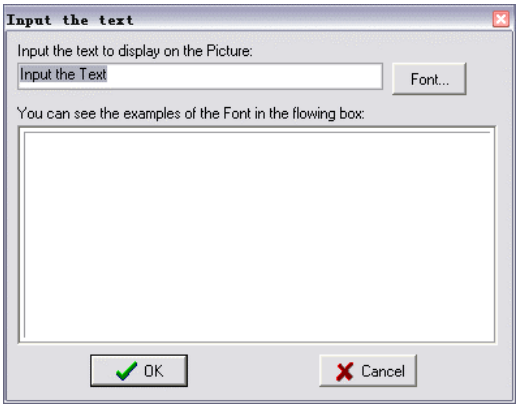

图 8.21 输入文本的窗口

在这个窗口中，可以在文本框中输入需要的文本，此时根据字体格式的设置，在下面的框中会显示这些文本的示例。如果需要设置字体的格式，可以单击对话框中的 Font 按钮，此时会显示一个标准的字体对话框。这里使用了 FontDialog 控件。

这个窗口中的代码比较简单，只是定义了下面的一些过程和事件：

```
procedure TForm5.BitBtn1Click(Sender: TObject);
begin
    if FontDialog1.Execute then
```

```
        PaintBox1.Canvas.Font.Assign(FontDialog1.Font);
    Edit1Change(sender);
End;

procedure TForm5.Edit1Change(Sender: TObject);
begin
    PaintBox1.Canvas.FillRect(PaintBox1.ClientRect);
    PaintBox1.Canvas.TextOut(20,40,Edit1.Text);
end;

procedure TForm5.FormShow(Sender: TObject);
begin
  Edit1Change(sender);
end;
```

在输出文本时，把 Brush **对象的模式设置成了** bsClear**，这样做的目的是为了在输出文本时不会出现挡住后面图形的情况。通过上面的代码，就可以实现在图片上输入文本的功能了，如图** 8.22 **所示。**

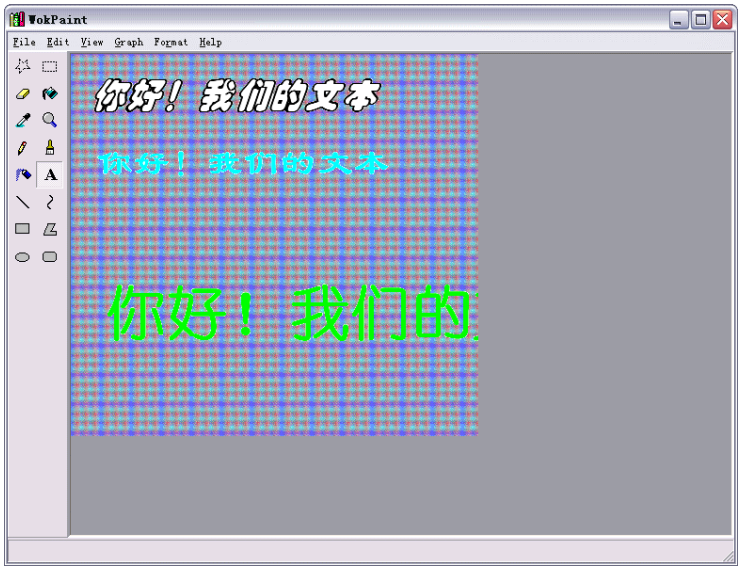

图 8.22　**在图片上的文本的输出**

8.4.9　选择和移动

选择和移动是图片编辑程序中的一个重要功能。选择功能类似于我们前面介绍的矩形工具，但是所不同的是选择的轮廓最终是要擦除的，所以最后不能用 pmCopy **方式进行绘制，**

而仍然使用 pmNotXor。那么当我们对选定的这部分内容进行操作时，要注意清除选择的轮廓方框，于是我们编写了下面一个过程，用来完成这个功能：

```
procedure TForm1.ClearSelectFrame;
begin
  if not NonSelected then
    With ImgPicture.Canvas do
    begin
      Pen.Style :=psDashDot;
      Pen.Mode := pmNotXor;
      Pen.Width:=1;
      Brush.Style := bsClear;

      Rectangle(SelectRect.Left,SelectRect.Top,SelectRect.Right,SelectRect.Bottom);
    end;
end;
```

和矩形工具不同的是，在 OnMouseDown 事件中，如果当前的状态是进行选择，那么首先需要清除原来的选择框，我们使用了下面的代码：

```
dtSelect:
  begin
   if (not NonSelected) then
   begin
      Pen.Style :=psDashDot;
      Pen.Mode := pmNotXor;
      Pen.Width:=1;
      Brush.Style := bsClear;
      Rectangle(SelectRect.Left,SelectRect.Top,SelectRect.Right,SelectRect.Bottom);
      NonSelected:=True;
      SelectRect.Left := SelectRect.Right;
      SelectRect.Top :=SelectRect.Bottom;
   end;
  end;
```

其他的，选择工具和矩形工具的实现就没有什么差别了。

在程序中，用 SelectRect 来存储选择的区域，那么就可以利用 Canvas 对象的 CopyRect 方法，把这个区域移动到需要的位置。我们用下面的代码实现了移动功能：

```
    dtMove:
    begin
    if Not NonSelected then
    begin
```

```
//清除选定的区域；
        R:=Rect(SelectRect.Left +Points[1].X-Points[0].x,
                SelectRect.Top+Points[1].Y-Points[0].Y,
                SelectRect.Right+ points[1].X-Points[0].X,
                SelectRect.Bottom+Points[1].Y-Points[0].Y);
        bit:=Timage.Create(Self);
        bit.Width := ImgPicture.Width;
        bit.Height := imgPicture.Height;
        bit.Canvas.CopyMode:=cmSrcCopy;
        bit.Canvas.CopyRect(R,ImgPicture.Canvas,SelectRect);
        CopyMode:=cmWhiteness;
        CopyRect(SelectRect,ImgPicture.canvas,SelectRect);
        CopyMode:=cmSrcCopy;
        NonSelected:=True;
        copyRect(R,bit.Canvas,R);
        bit.Free;
      end;
      end;
```

在上面的程序中，用一个临时的对象来作为缓存，首先把选定的内容复制到缓存中，然后用 cmWhiteness 的复制模式把原来选定的区域清除，最后再把缓存中的内容移动到指定的区域中。例如在图 8.23 所示的 WokPaint 程序中，我们把一个选定的图像从一个位置移动到了另外一个位置。

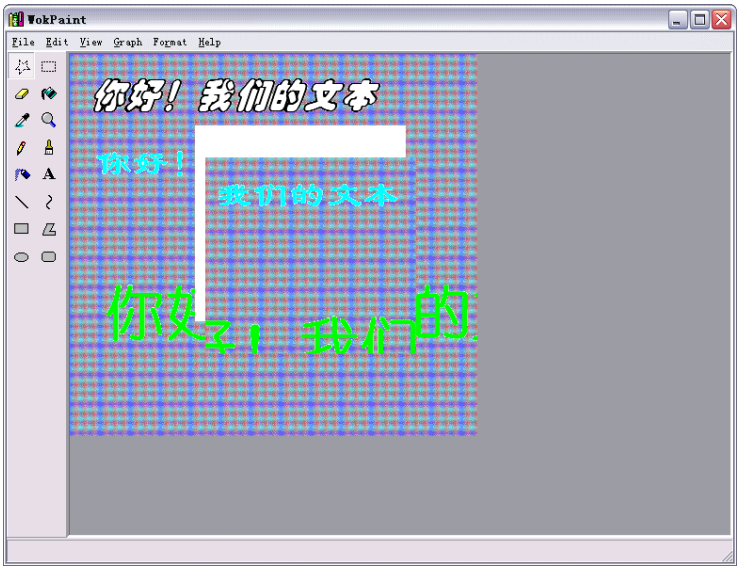

图 8.23　图像的选择和移动

8.4.10 使用剪贴板对象

在这一节中，要介绍如何实现 Edit 菜单中的 Cut、Copy、Paste 和 Delete 功能，其中要涉及到剪贴板对象的使用。通过剪贴板，我们可以在程序中剪切、复制或者粘贴图片，也可以把图片复制到其他应用程序中去。

如果要在程序中使用剪贴板，我们必须在程序的 Uses 语句中加上 Clipbrd，引用 Delphi 中定义剪贴板对象的单元。

首先介绍把图片复制到剪贴板中去的功能。可以使用 Assign 方法把要复制的图片赋给 ClipBoard 对象，语句如下：

```
procedure TForm1.Copy1Click(Sender: TObject);
begin
Clipboard.Assign(Image.Picture)
end.
```

在程序中，需要注意的是，所要复制的不是 ImgPicture 控件的所有图片，而是图片中我们选定的那部分，所以，我们编写了下面的程序，也就是菜单命令中的 Copy 的响应事件：

```
procedure TForm1.Copy1Click(Sender: TObject);
var bit:TImage;
begin
bit := TImage.Create(Self);
if not NonSelected then
begin
    bit.Width:=SelectRect.Right-SelectRect.Left;
    bit.Height := SelectRect.Bottom-SelectRect.Top;
    ClearSelectFrame;
    bit.Canvas.CopyRect(bit.Canvas.ClipRect,
                        ImgPicture.Canvas,SelectRect);
    Clipboard.Assign(bit.Picture);
    ClearSelectFrame;
end;
bit.Free;
end;
```

在这个程序中，仍然用一个 TImage 对象作为处理图片的缓存。

如果要删除所选定的图片，可以采用下面的方法：

```
procedure TForm1.Delete1Click(Sender: TObject);
begin
if not NonSelected then
```

```
    with ImgPicture.Canvas do
    begin
      CopyMode := cmWhiteness;
      CopyRect(SelectRect, ImgPicture.Canvas, SelectRect);
      CopyMode := cmSrcCopy;
    end;
  NonSelected:=true;
  end;
```

我们用 cmWhiteness **的复制模式覆盖了选定的区域。**

要实现剪切功能，也就是菜单中的 Cut **功能，可以把上面的** Copy **和** Delete **功能组合起来，可以利用下面的程序：**

```
procedure TForm1.Cut1Click(Sender: TObject);
begin
Copy1Click(Sender);
Delete1Click(Sender);
end;
```

把图片复制或者剪切到剪贴板对象上的时候，所有的 Windows **程序就都可以使用它了。可以在** Word **文档或者** PhotoShop **等图像处理软件中用** Ctrl+V **组合键来把所复制的图片粘贴到文档中去。**

现在我们来实现 WokPaint **程序中的** Ctrl+V **组合键的功能，也就是** Paste **功能。此时需要使用** ClipBoard **对象的** HasFormat **函数，用来判断剪贴板中是否存在所需要类型的对象，这里需要的是图片。可以使用下面的代码：**

```
procedure TForm1.Paste1Click(Sender: TObject);
var bit:TImage;
begin
bit := TImage.Create(Self);
if Clipboard.HasFormat(CF_BITMAP) then
begin
    bit.AutoSize := True;
    bit.Picture.Bitmap.Assign(Clipboard);
    ImgPicture.Canvas.CopyRect(bit.Canvas.ClipRect,
                               Bit.Canvas ,bit.Canvas.ClipRect);
end;
bit.Free;
end;
```

在程序中，给 HasFormat **函数传递的是** CF_BITMAP **参数，代表位图对象，也就是我们常说的** BMP**。除了** CF_BITMAP **之外，**HasFormat **中还具有以下一些可能的值。**

- CF_TEXT：文本。
- CF_METAFILEPICT：Windows 元文件图片，也就是 WMF 或者 EMF 格式的图片。
- CF_PICTURE：具有 TPicture 兼容类型的对象。
- CF_COMPONENT：其他的任何固定对象。

在下面的图 8.24 中，显示了复制并粘贴的图片。

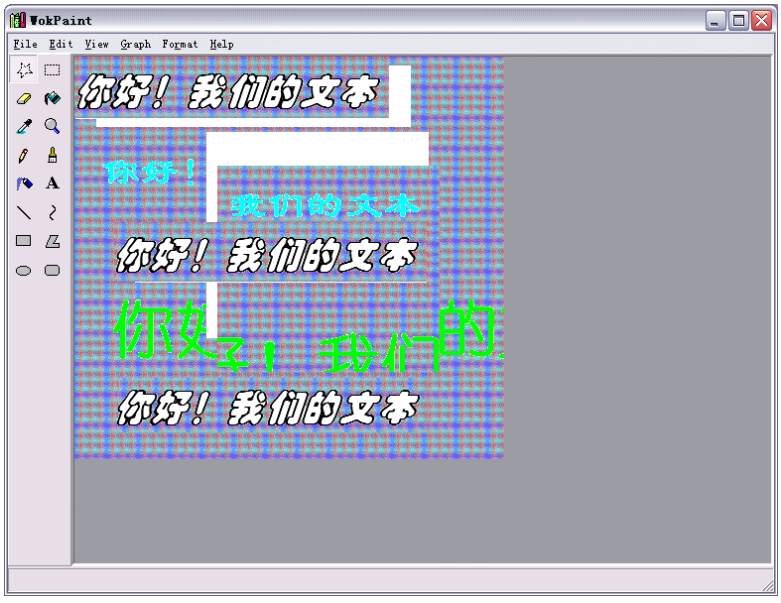

图 8.24　图像的剪切、复制和粘贴

在上面的窗口中，首先把选定的区域剪切到了剪贴板中，然后在图片上复制了三份。当然也可以把图片通过剪贴板复制到其他的应用程序中去。对于文本及其他格式对象的剪切、复制和粘贴，基本上是一样的，读者可以参考我们这里的程序。

8.4.11　收尾工作

到这里，我们基本上已经完成了 WokPaint 程序的所有基本工作，剩下的事情是完成图像尺寸的修改，以及图片文件的打开、新建、保存等操作。

对于图像尺寸的修改，利用一个如图 8.25 所示的窗口来完成。

在这个窗口中，左边是图片的尺寸，右边是图片的缩略图。程序的代码如下所示：

```
procedure TForm6.FormShow(Sender: TObject);
begin
Updown1.Position := Form1.ImgPicture.Width;
Updown2.Position := Form1.ImgPicture.Height;
```

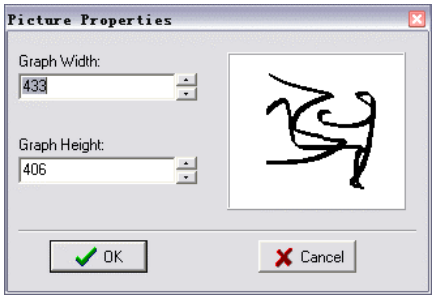

图 8.25　图像尺寸属性

```
Image1.Canvas.CopyRect(Image1.Canvas.ClipRect,
                       Form1.CurBitmap.Canvas,
                       Form1.ImgPicture.Canvas.ClipRect);
end;
```

在主程序中，使用了下面的代码来处理修改后的尺寸：

```
if Form6.ShowModal = mrOK then
begin
    ImgPicture.Width := strtoint(Form6.LabeledEdit1.Text);
    ImgPicture.Height := Strtoint(form6.LabeledEdit2.Text);
    curBitmap.Constraints.MinHeight := ImgPicture.Height;
    curBitmap.Constraints.MinWidth := ImgPicture.Width;
end;
```

在这个程序中，需要注意的是，当你改变图像的尺寸时，应该注意修改 CurBitmap 对象的 Constraints 属性。

关于图片文件的打开、新建和保存功能，主要利用了 TImage 控件的 LoadFromFile、SaveToFile 等方法。从这里可以看出我们在程序中使用 ImgPicture 和 CurBitmap 两个控件来处理图像的好处了，你会注意到，TPaintBox 控件是没有上面的方法的。当然，也可以通过自己编写适当的程序来完成相同的功能，但是毕竟麻烦了一些。

关于文件新建、打开、保存的代码如下所示：

```
procedure TForm1.New1Click(Sender: TObject);
begin
ClearGraph1Click(sender);
end;

procedure TForm1.Open1Click(Sender: TObject);
begin
if OpenDialog1.Execute then
begin
```

```
    CurBitmap.Picture.LoadFromFile(OpenDialog1.FileName);
    ImgPicture.Width :=Curbitmap.Picture.Width;
    ImgPicture.Height := Curbitmap.Picture.Height;
    CurBitmap.Constraints.MinHeight :=ImgPicture.Height;
    CurBitmap.Constraints.MinWidth := ImgPicture.Width;
    ImgPicturePaint(Sender);
end;
end;

procedure TForm1.Save1Click(Sender: TObject);
begin
if SaveDialog1.Execute then
begin
    CurBitmap.Picture.SaveToFile(SaveDialog1.FileName);
end;
end;
```

在实现这些功能的时候，我们将 OpenDialog 和 SaveDialog 两个对话框控件进行了结合使用。它们的使用比较简单，在本程序中的使用方法基本上已经能够满足用户的各种需要了。

除了这些之外，还有一些其他的细节没有介绍，它们的实现都比较简单。在这里因为篇幅的关系，没有列出完整的程序代码，请读者自行设计。

8.5 本 章 小 结

本章介绍了在 Delphi 中绘图的一些基本知识，主要是 TCanvas 对象及其附属对象和属性方法的利用。我们还可以设计更多的关于图像处理方面的内容，虽然这些内容要更复杂一些，但是基本方法一般都不会超出上面介绍的内容。

在本章中，还介绍了一个完整的程序示例——WokPaint 程序的编制，基本上实现了 Windows 操作系统中画笔程序的所有功能。从这个程序的编写过程可以看出，在编程的时候，最重要的两个事情是：一是对要编写的程序功能以及实现这些功能的控件或者对象的属性和方法具有很好的认识，二是要掌握一定的算法。

第 9 章　开发数据库应用程序

数据库应用程序的开发显然是 Delphi 中一个非常重要的问题。这一点从 Delphi 的控件面板上有关数据库的控件所占的比例就可以看出来。数据库的应用现在越来越广泛，它提供了一种存储信息的结构，通过这种结构，就可以有很多用户通过各种应用程序来访问存储在其中的信息。

Delphi 支持关系式数据库应用程序。在开发数据库应用程序时，通常要分为两个部分的设计，一个部分是用户界面的设计，它关系到向用户提供数据的形式，程序的吸引力等方面；另一个部分是关于数据模块的处理，在处理数据模块时，会涉及到众多控件的使用方法。

在现代的数据库应用程序中，除了常用的两层数据库结构的应用程序之外，现在开发多层的数据库应用程序也是一个非常重要的内容。在本章中我们还将针对如何开发多层的数据库应用程序进行讨论。

在本章中将主要介绍：

- 数据库应用程序涉及概述
- 数据应用程序的界面设计
- 数据集控件以及其他数据控件的使用
- 主从式数据库的使用
- 多层数据库应用程序

9.1　数据库应用程序设计概述

数据库应用程序使用户可以使用存储在数据库中的信息，而数据库提供了存储信息的结构，并可以在不同的应用程序之间共享这些信息。Delphi 提供了对关系式数据库应用程序的支持，所谓关系式数据库，就是把信息存储在一个一个的表中，这些表包含行（也就是记录）和列（也就是字段）。这些表可以通过简单的操作进行处理。在设计一个数据库应用程序时，必须理解数据在数据库中是怎样存储的。基于这种存储结构，就可以设计出自己的应用程序界面，通过这些界面，用户可以浏览、输入或者修改数据库中的信息。

在本节内容中，将讨论设计数据库应用程序的一些基本概念和数据库应用程序的用户界面的一些基本结构。

9.1.1 数据库概述

Delphi 提供了很多关于数据库的访问和数据库信息显示方面的控件和对象。它们根据数据访问的机制，分成以下种类。

- 在 Delphi 控件面板上的 BDE 选项卡中包含了使用 Borland 数据库引擎（BDE）的控件。BDE 定义了大量关于数据库的 API 函数，在所有的数据库访问机制中，BDE 支持的范围最广，支持的功能最多，可以说，它是处理 Paradox 或者 dBase 中数据的最好方法。但是，它也是发布起来最麻烦的方式。
- 控件面板上的 ADO 选项卡中包含了使用 Data Objects (ADO)通过 OLEDB 访问数据库信息的控件和对象。需要说明的是，ADO 是微软公司标准，在连接到不同的数据库服务器时，可以使用很多的 ADO 驱动程序。使用基于 ADO 机制的控件可以使我们的应用程序能够集成到基于 ADO 的环境中，比如利用基于 ADO 的应用程序服务器。
- dbExpress 选项卡中的控件使用 dbExpress 来访问数据库中的信息。它能够迅速地访问数据库中的信息，该类控件还支持跨平台的开发。但是 dbExpress 数据库控件支持的数据处理功能比较少。
- InterBase 选项卡中包含了可以直接访问 InterBase 数据库而不通过另外的引擎层的控件。
- Data Access 选项卡中包含了可以用来处理任何数据访问机制的数据库信息控件。

在设计一个数据库应用程序时，必须确定应该使用哪些控件。在支持的功能上、发布的难易上和可用的驱动程序方面，每种数据库访问机制都各有优缺点。另外，在选择使用某种数据访问机制时，也必须选择一种数据库服务器。在此之前，应该仔细考虑各种类型的数据库服务器的优点和缺点。有一点可以肯定的是，所有类型的数据库都是把信息存储在表中，并且大多数服务器都支持数据库安全性、事务、引用完整性、存储过程和触发器等功能。

下面我们来看一下基本的数据库类型。关系式数据库服务器根据它们存储信息的方式和多个用户同时访问其中的数据的方式，可以分成很多种类。Delphi 支持两类关系式数据库服务器。

- 远程数据库服务器。该类数据库服务器一般不在程序运行的机器上。在有的情况下，来自一个远程数据库服务器的数据也许不是存储在某个计算机上，而是分布在几个服务器上。尽管远程服务器存储信息的方式不同，但是它们都提供了一种通用的逻辑接口，就是我们常说的结构化查询语言（SQL）。由于通常是用 SQL 来访问它们的数据，所以它们又经常被称为 SQL 服务器。常用的 SQL 服务器包括 InterBase、Oracle、Sybase、Informix，以及微软公司 SQL 服务器以及 DB2。
- 本地数据库。该类数据库存在于计算机上或者局域网中。它们通常提供了一些访问

其中数据的私有 API 函数。当在几个用户之间共享这些数据时，它们使用的是基于文件的锁定机制。正是因为这个原因，有时候又把它们称为基于文件的数据库。常用的基于文件的数据库包括 Paradox、dBase、Foxpro 和 Access。

由于应用程序和数据库使用同一套文件系统，所以使用本地数据库的数据库应用程序通常被称为单层程序。同样道理，由于应用程序和数据库在不同的操作系统上，所以使用远程数据库服务器的数据库程序称为两层或者多层数据库程序。

在选择要使用的数据库类型时，需要考虑许多因素，例如这些数据可能早就存在于某个数据库中。如果要创建应用程序使用的数据库的表，需要考虑以下几个问题。

- 会有多少用户共享我们的数据表？远程数据库服务器本身就是设计来让多个用户同时访问的，所以它们通过称为事务的机制提供了对多用户的支持。有些本地数据库也支持事务，例如本地的 InterBase，但是大多数本地数据库由于是基于文件的锁定机制，所以很多时候根本就不支持多用户访问。
- 数据表中将存储多少数据？毫无疑问，远程数据库可以比本地数据库存储更多的信息。有些远程数据库服务器是为数据的海量存储设计的，有些远程数据库服务器则是为了其他的标准设计的，比如快速更新。
- 对于你使用的数据库，你最关注它的什么性能？比如是速度还是稳定性？本地数据库通常要比远程数据库服务器快得多，因为它们就和数据库应用程序位于同一个计算机上。而不同的远程数据库服务器是为了不同的功能进行了优化的，所以在选择远程数据库时，也需要考虑它们的性能。
- 对于数据库的管理员来说，他需要哪些功能支持？一般来说，本地数据库比远程数据库服务器需要的管理支持要少。一般来说，操作本地数据库比操作远程数据库服务器要容易得多。

需要考虑的另外一个问题是数据库的安全性问题。数据库中通常存储了一些非常重要的信息，这些信息可能不能让别人随便浏览，所以就需要能够保护这些信息的工具。几乎所有的数据库都提供了保护方案，但是保护方案的层次不同。比如，有些数据库，诸如 Paradox 和 dBase，只是提供了在数据表和数据字段层次上的数据保护。当用户试图访问这些受保护的数据表时，程序会要求用户输入密码。一旦用户被确认输入了正确的密码，就能够看到他们拥有权限的那些字段了。大多数 SQL 服务器在用户需要使用该数据库服务器的时候要求用户输入用户名和密码，然后根据用户名和密码来确定他们可以使用哪些数据表。

所以，在设计数据库应用程序时，必须考虑应该提供什么样的身份确认。通常来说，应用程序不会显示明确的用户登录窗口，而是在程序本身进行登录。在这种情况下，必须十分小心，因为如果赋予了程序过多的权限，那么用户可能会损坏我们的数据库。如果要明确地要求用户提供密码，那么必须考虑在什么时候提出这个要求。如果程序可以登录到几个受保护的系统或者数据库，那么有可能需要用户提供多个密码。在这种情况下，可以用一个数据表来存储所有可能的密码，并把它们分组，然后让用户提供一个密码来确定他们适用于哪组

密码。从而通过这个数据表，程序就能够自动提供多个密码，而不是让用户不断地输入密码。在多层应用程序中，可以使用 HTTP、CORBA 和 COM+来控制对中层的访问，并让中层来处理登录到数据库的问题。

在上面的过程中，我们多次提到了事务。所谓事务，就是在数据库中的一个或者多个数据表在更新之前，可以对它们成功执行的一组行为。如果组中的任何一个行为失败了，那么所有的行为将被取消。通过事务，可以确保以下几个方面。

- 某个事务中的所有的更新要么完全进行，要么中断、恢复到事务开始前的状态，这一性能称之为可分性。
- 一个事务是对系统状态的一个成功转换，可以保持系统状态的稳定性，这一性能称之为一致性。
- 同时发生的事务不会看到相互的中间结果或者未被允许的结果，所以程序会处于不同的状态，这一性能称之为隔离性。
- 提交对记录的更新时可以避免出错，包括通信错误、处理错误以及服务器系统的错误，这一性能称之为稳定性。

因此事务可以使数据库命令在执行的过程中保护数据，使之不会因为某些原因受到破坏，比如硬件上的原因。事务性记录也使得能够在磁盘介质发生错误的时候恢复数据库。同时，事务也是构成 SQL 服务器多用户并发控制的基础。当每个用户都是通过事务处理数据库时，其中一个用户的命令不会打断其他用户事务中的命令。相反，SQL 服务器统一安排所有的事务，让它们或者整体成功，或者整体失败。

所有的关系式数据库都有特定的功能用来存储和处理数据。同时数据库也经常提供其他的一些数据库专用的功能来确保数据库中的数据表之间的一致关系。这些功能包括以下内容。

- 引用完整性：引用完整性提供了一种机制，用来防止数据表之间的主从关系被破坏。当用户试图删除一个主表，从而会导致有些表中的记录失去了详细细节的时候，引用完整性功能将会禁止这个删除操作，或者自动删除那些无效的记录。
- 存储过程：存储过程是声明和存放于 SQL 服务器上的一系列 SQL 语句，它们通常用来在服务器上执行数据库相关的任务，有时会返回一组记录。
- 触发器：触发器是用来自动对特定的命令做出反应的 SQL 语句。

9.1.2 数据库应用程序的结构体系

通常来说，数据库应用程序是由用户界面、代表了数据库信息的控件和与数据库信息相连接或者相互之间互相连接的控件组成的。如何组织这些内容便是数据库应用程序的结构体系。从上面的分析中可以看出，数据库应用程序至少要包含三个内容：

- 用户界面

- 数据库信息控件
- 和数据库相连的控件以及相互关联的控件

在数据库应用程序中，有很多不同的方法来组织这些控件，其中的大多数控件都遵从图 9.1 所示的结构。

图 9.1　常用的数据库应用程序结构

说明：

在上面的图形中，UI 代表用户界面（User Interface）。

下面就按照上面图示的结构来介绍各个部分的特点。首先介绍的是用户界面部分的特点。

把窗体上的用户界面部分和应用程序的其他部分完全地分隔开来是一个很好的想法。这样做有很多好处。通过分离用户界面和代表数据库信息的控件，大大提高了程序设计的灵活性。

- 当我们对程序中管理数据库信息的部分进行修改的时候，不需要重新编写用户界面。
- 当我们修改程序中的用户界面时，不需要改变程序中处理数据库的部分。
- 使我们可以设计出能够在多个应用程序中使用的窗体，从而提供了一致的用户界面。

在数据模块部分，如果把用户界面设计到它们自己的窗体中，那么可以使用一个 Data Module 来放置和数据库相关的控件，包括和数据库相连的控件以及和自己的应用程序的用户界面相连的控件。就像设计用户界面一样，我们也可以在多个应用程序之间共享自己的 Data Module。

在数据模块部分中，又分为三个部分，Data Source（数据源）、DataSet（数据集）和 Connection to Data（数据连接）。

- 数据源：数据源就是用户界面和代表来自数据库信息的数据集通道。如果窗体上的数据处理控件可以使用一个数据源，那么这些控件的显示将是同步的，比如当用户在记录中浏览的时候，这些控件中的所有相应内容都发生了改变。
- 数据集：数据集是数据库应用程序的核心。这些控件代表了来自数据库的一系列记录。这些记录可能来自一个数据表，也可能来自多个数据表的每个记录中的部分字段，也可能来自多个数据表的组合。通过使用数据集，应用程序可以用来缓存对数

据库中实际数据表的重新构造。当数据库改变时，可能需要改变数据集控件定义自己数据的方式，但是该数据库应用程序的其他部分不需要进行任何修改就可以继续使用。

- 数据连接：不同类型的数据集使用不同的连接机制，这里介绍了四种基本的连接机制，在有的情况下，这些机制可以结合起来使用。
  - 直接连接到数据库服务器。这样的大多数数据集都是 TCustomConnection 对象的派生对象。
  - 使用磁盘上的一个指定的文件。客户端数据集具有处理磁盘上指定文件的能力，因为数据集本身就知道如何来读写文件。
  - 连接到其他数据集。客户端数据集可以处理由其他数据集提供的数据。TDataSetProvider 对象可以作为客户端数据集和源数据集的中介。这个对象可以和客户端数据集一样存在于同一个 Data Module 中，也可以成为在其他计算机上运行的应用程序的一部分。如果该对象是一个应用程序服务器的一部分，就需要使用特定的 TCustomConnection 派生对象来连接到应用程序服务器上。
  - 从关系式数据库系统的 DataSpace 对象中获得数据。在使用基于 ADO 应用程序服务器开发的多层数据库应用程序中，ADO 数据集可以用一个 TRDSConnection 控件来处理数据。

9.2 设计数据库应用程序界面

在开始介绍数据库应用程序的数据集等部分内容之前，我们应该先来了解一下数据库应用程序的用户界面问题，也就是使用数据控件的问题。数据控件位于 Delphi 控件面板的 Data Control 选项卡中。这些控件的功能是显示数据库中记录的字段数据，如果数据集允许，也可以让用户编辑这些数据，并把修改更新到数据库中。通过在程序上放置这些数据控件，就可以构建起数据库应用程序的用户界面，从而也就使得用户可以访问数据库的信息。

9.2.1 数据控件的通用功能

在开发数据库应用程序的用户界面时，经常会使用到下面的一些功能。

- 把数据控件和一个数据集关联起来。
- 编辑和更新数据。
- 启用或者禁用数据显示。
- 刷新数据显示。
- 使用鼠标、键盘和 Timer 控件的 Timer 事件。

数据控件使用户可以显示和编辑相关联的数据集中的当前记录字段。表 9.1 中总结了数据控件的基本功能。

表 9.1　Delphi 中的数据控件

数据控件名称	说明
TDBGrid	用表格的形式显示数据源中的信息，表中的列代表指定的数据源中的列，表中的行代表对应的记录
TDBNavigator	在数据集的记录之间导航，可以更新记录、删除记录、取消对记录的编辑以及刷新数据显示
TDBText	就像 Label 控件一样显示数据集中的一个字段的数据
TDBEdit	在一个文本框中显示数据集中的一个字段的数据
TDBMemo	在一个类似于 Tmemo 控件的框中显示数据集中的 Memo 或者 Blob 类型的字段数据
TDBImage	在一个图形框中显示数据字段中的图像
TDBListBox	显示用来更新当前记录中指定字段的项目列表
TDBComboBox	显示用来更新当前记录中指定字段的项目列表，可以直接输入要修改的值
TDBCheckBox	显示一个用来代表一个 Boolean 类型字段的复选框
TDBRadioGroup	显示一系列关于字段的单选按钮
TDBLookupListBox	显示根据某个字段的值在其他数据集中找到的项目列表
TDBLookupComboBox	显示根据某个字段的值在其他数据集中找到的项目列表，也可以直接输入要查找的值
TDBCtrlGrid	在一个表格中显示一系列和记录对应的数据控件
TDBRichEdit	在一个 RichEdit 控件中显示某个字段的格式化数据

数据控件在设计期就能够响应数据。在设计应用程序时，当把数据控件和一个打开的数据集相关联时，就会立即看到数据控件中显示了数据集中的数据。此时，可以不用编译和运行应用程序就能通过字段列表来确认数据是否正确显示。在运行时，数据控件将显示数据集中的数据，如果可能，数据控件还可以编辑数据。

下面来介绍如何关联数据控件和数据集。数据控件通过一个数据源对象（TDataSource）连接到数据集上。数据源对象就像数据控件和包含数据的数据集通道。每个数据控件都必须关联到一个数据源对象上才能显示和处理数据。类似地，一个数据集控件也必须关联到一个数据控件上才能显示和处理所包含的数据。

通过下面的步骤，就可以把数据控件和数据集关联起来。

(1) 首先在一个窗体上或者在一个 Data Module 上放置一个数据集控件，并适当地设置它的属性。

(2) 在同一个窗体或者 Data Module 上放置一个数据源控件，在属性编辑器中，把它的 DataSet 属性设置成上面放置的数据集控件。

(3) 在一个窗体上放置一个数据集控件。注意，这里的窗体最好和上面的窗体不同。

(4) 在属性编辑器中，把 DataSource 属性设置成上面放置的数据源控件。

(5) 单击 DataField 属性旁边的按钮，会显示当前数据源控件中的所有字段，从中选择一个需要的字段。这个步骤在 TDBGrid、TDBCtrlGrid 和 TDBNavigator 控件上没有作用，因为它们访问的是全部字段，所以不需要进行选择。

(6) 把数据集控件的 Active 属性设置成“ True ”，此时数据集中的数据将会显示在数据控件中。

利用上面的过程，便可以在设计期实现数据控件和数据集控件的关联，也可以在运行时设置这些关联。例如，利用下面的程序就可以修改 CustSource 数据控件的 DataSet 属性：

```
with CustSource do begin
if   (DataSet = Customers) then
     DataSet := Orders
else
     DataSet := Customers;
end;
```

另外，也可以通过 DataSet 属性的变化来使不同窗体中的数据控件能够同步。例如，通过使用下面的代码，就可以使窗口 2 中的数据控件和窗口 1 中的数据控件显示相同的内容。

```
procedure TForm2.FormCreate (Sender : TObject);
begin
     DataSource1.Dataset := Form1.Table1;
end;
```

可以通过数据源控件的 Enabled 属性来启用或者禁用数据源。当 Enabled 属性为 True 的时候，数据源控件连接到了数据集控件上。如果要断开某个数据源和数据集的连接，则可以把 Enabled 属性设置成“ False ”。断开数据源控件和数据集控件的连接之后，所有和该数据源控件关联的数据控件将全部变成空的，直到恢复这个连接为止。但是我们强烈建议读者使用数据集控件的 DisableControls 和 EnableControls 方法来处理这方面的工作，因为它们可以影响所有关联的数据源控件。例如，可以使用下面的代码：

```
with CustTable do
begin
   DisableControls;
   try
      First;
      While not Eof do
      begin
        { Process each record here }
        Next;
      end;
   finally
```

```
      EnableControls;
    end;
  end;
```

下面来分析一下数据源控件的事件响应。因为数据源提供了数据控件和数据集之间的连接，所以它会掌握两者之间发生的所有通信。例如数据响应控件就能够自动反应数据集中的变化。但是，如果用户界面用的是不能自动响应数据的控件，那么应该使用数据源控件来提供同样的响应。

要介绍的第一个事件是 OnUpdateData 事件，该事件发生在当前记录要被更新的时候，例如，在程序中调用 Post 方法时，便会引发该事件，但是需要注意的是，这个事件发生在数据库服务器或者本地的数据库真正更新之前。

OnStateChange 事件发生在数据集控件的状态发生变化的时候。当这个事件发生时，可以检测数据集控件的 State 属性以确定它的当前状态。例如，可以使用下面的事件处理程序来启用或者禁用相应的界面控件、菜单或者按钮：

```
procedure Form1.DataSource1.StateChange(Sender: TObject);
begin
    CustTableEditBtn.Enabled := (CustTable.State = dsBrowse);
    CustTableCancelBtn.Enabled := CustTable.State in
                                    [dsInsert, dsEdit, dsSetKey];
    CustTableActivateBtn.Enabled := CustTable.State in [dsInactive];
end;
```

编辑和更新数据是数据控件的一个重要功能。所有的数据控件，除了 TDBNavigator 控件之外，都能显示数据库中的字段的数据。而且还可以利用这些控件来编辑和更新数据，当然这需要关联的数据集允许。

一个数据集控件必须在 dsEdit 状态下，才能运行编辑它的数据。如果数据源控件的 AutoEdit 属性是 True，数据控件会在用户试图编辑记录的数据时，把数据集控件置于 dsEdit 状态。如果 AutoEdit 的值为 False，就需要使用其他的方法来把数据集控件置于编辑状态。其中的一个方法就是使用带有 Edit 按钮的 TDBNavigator 控件。它可以让用户明确地把数据集控件置于编辑模式。

如果一个数据集控件的 CanModify 属性为 True，那么一个数据控件就会只把修改的内容刷新到数据集控件中去。但是对于单向性的数据集来说，CanModify 属性总是 False。有些数据集控件具有 ReadOnly 属性，利用这个属性可以指定 CanModify 属性是否为 True。

即使数据集控件的 CanModify 属性的值为 True，连接数据集和数据控件的数据源控件的 Enabled 属性也必须为 True，数据控件才能把结果更新到数据库的表中。

最后，需要考虑是否允许用户在数据库应用程序中进行输入。数据控件的 ReadOnly 属性可以确定用户是否可以编辑显示在控件上的数据。如果该属性为 False（默认值），那么用

户可以编辑数据。当数据集控件的 CanMofify 属性为 False 的时候，数据控件的 ReadOnly 属性将为 True。对除了 TDBGrid 之外的所有数据控件来说，当修改一个字段，在输入焦点离开该控件的时候，修改会复制到对应的数据集控件中。如果在输入焦点离开数据控件之前按下了 Esc 键，那么数据控件将放弃修改，控件中的值仍然保持原来的状态。在 TDBGrid 控件的不同记录中移动时，修改便提交到数据集控件中。同样利用 Esc 键可以在移动之前取消修改。

在提交对一个记录的修改时，Delphi 会自动检查所有和对应数据集相关联的数据响应控件，以保证它们的一致性。如果在更新记录时出错，Delphi 将引发一个异常，不会对记录进行任何修改。

如果在程序中需要刷新当前的数据显示，那么可以使用数据集控件的 Refresh 方法。Refresh 方法将清空本地的缓存，并重新从一个打开的数据集中获取数据。Refresh 有时也会导致一些不可预料的结果。比如，如果一个用户在查看一个记录时，另外的一个用户删除了该记录，那么当调用 Refresh 方法时刚才查看的记录就消失了。同样，如果有的用户在调用 Refresh 之前已经对我们正在查看的记录的某个字段进行了修改，那么当我们调用 Refresh 方法的时候，我们也会发现内容发生了改变。

说明：

在程序中，可以通过数据控件的 Enabled 属性来启用或者进行一个数据控件。当该属性为 False 时，数据源控件就不会从这个数据控件中获取任何信息了，同时数据控件的鼠标、键盘等事件也就无法响应了。数据控件中仍然显示着在禁用之前的数据，但是这些数据不会再变化。

9.2.2 利用数据控件显示单个记录

在上面的内容中，我们简单介绍了数据控件的一些基本功能，相信通过这些内容，读者基本上已经可以使用数据控件了。同时每个数据控件还具有自己的一些特殊属性，一般来说，可以在常规控件中找到和数据控件对应的控件，比如 TDBEdit 控件就和常规控件中的 TEdit 控件相对应，它们的很多属性都是相同的。所以在使用数据控件时可以参考这些常规控件的用法。

数据控件可以分为显示单行记录的控件、显示多行记录的控件和浏览处理记录的控件。

在许多应用程序中，可能每次只需要提供单个记录的信息。例如，在一个输入订单的应用程序中，只需要显示关于单个订单的信息，而不需要显示其他订单的信息。这一信息可能来自订单数据集中的某个记录。一般来说，显示单个记录的应用程序是比较简单和容易理解的，因为它上面的所有数据库信息都是关于同一个记录的。在这种用户界面中的数据响应控件中显示了数据库记录的一个字段。Delphi 中提供了大量的关于显示不同形式字段的数据控

件，要使用哪个控件取决于数据的类型。下面通过一个简单的应用程序，演示一下显示单个记录的数据控件的各种用法。

(1) 首先，建立一个新的应用程序，然后选择 File 菜单中的 New，并从子菜单中选择 Data Module，此时会建立一个如图 9.2 所示的模块。

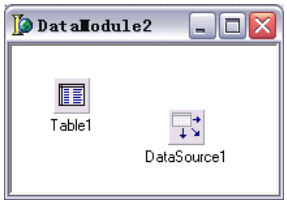

图 9.2　数据模块窗口

(2) 在这个窗口中放置一个 TTable 控件和一个 TDataSource 控件，把它们的名称修改为“TbExample”和“DsExample”，把 DataModule2 的 Name 属性修改为“dmExample”。

(3) 把 tbExample 控件的 DataBaseName 属性设置成“DBDEMOS”，这是 Delphi 提供的一个演示数据库。在后面的很多程序中，都将使用这样的演示数据库来介绍我们的程序处理。把 TableName 属性设置为“biolife.db”。此时可以把 tbExample 控件的 Active 属性设置成“True”，相当于打开了我们的数据库。

(4) 把 dsExample 控件的 DataSet 属性设置成“tbExample”。到此为止，就完成了这个程序中的数据模块的设计。

(5) 在 Form1 窗口中的 Uses 语句部分，加入“dmExample”。保存所有的窗口和单元。

(6) 下面我们来设计程序的用户界面部分。选择 Data Controls 选项卡中的 TDBText 控件，在窗口的适当位置放置这样的一个控件。该控件和常规控件中的 TLabel 控件非常相似，是个只读的文本显示控件。如果希望能在窗口上显示不能被修改的数据，那么这个控件是非常有用的。像上面介绍的一样，把该控件的 DataSource 属性设置成“DmExample.dsExample”，只要单击该属性旁边的按钮，便会自动显示可以使用的数据源控件。把该控件的 DataField 属性设置成“Common_Name”。此时会看到窗口中的该控件已经显示出了数据集中的第一个记录的名为 Name 的字段值。

(7) 在窗口上放置一个 TDBEdit 控件。该控件和常规控件中的 TEdit 控件十分类似。关于它的非数据属性，读者可以参考前面介绍的 TEdit 的内容。和 TDBText 类似，也设置它的 DataSource 属性和 DataField 属性，这里把 DataField 属性设置成“Species Name”。

(8) 在窗口上放置一个 TDBMemo 控件。该控件和常规控件中的 TMemo 控件类似。仍然是设置 DataSource 和 DataField 两个属性。这里把 DataField 属性设置成“Notes”。

(9) 在窗口上放置一个 TDBImage 控件，该控件和常规控件中的 TImage 控件类似。仍然需要设置它的 DataSource 属性和 DataField 属性。这里把 DataField 属性设置成“Graphic”。该控件有一个 AutoDisplay 属性，如果该属性为“True”，那么控件将自动显示连接到的数据；如果该属性为 False，控件将显示字段的名称，只有在该控件中双击鼠标，才会显示数据中的图像。由于该控件要显示大量的数据（图像的数据往往是比较大的），所以在运行的时候它的显示可能会比较慢一些。该控件像 TImage 控件一样，支持使用 CutToClipboard、CopyToClipboard 和 PasteFromClipboard 方法来编辑其中的图像。如果把前面的 dsExample 的

AutoEdit 属性设置成“True”，那们可以把修改后的该控件中的数据存储到数据库中。也可以利用一个 OpenDialog 对话框控件，把本地计算机上的一个图像文件传送到数据库中。

(10) 在窗口上放置一个 TDBListBox 控件，这是一个用于修改数据库数值的控件。可以在这个控件中利用 Items 属性来输入一些值。那么在运行的时候，如果当前记录中的该字段的值和该控件中的某个项目相同，那么该项目将使用加亮显示。当用户在这个控件中选择一个项目时，数据集中对应的字段值将会被修改成所选择的项目。在这里把它的 DataField 属性设置成“Species No”，并输入了三个预定的数据。

(11) 在窗口上放置一个 TDBComboBox 控件。它和 TDBListBox 控件类似，也是一个用于数据库数值修改的控件。也可以利用该控件的 Items 属性输入一些值。如果当前记录中该字段的值和该控件中的某个项目相同，那么该项目将使用加亮显示，并显示在该控件中。当用户在这个控件中选择一个项目的时候，数据集中对应的字段值将会被修改成所选择的项目。

(12) TDBLookupListBox 控件的表现和 TDBListBox 控件相似，不同的是它显示的项目不是我们输入的，而是来自另外一个数据集。所以需要设置它的 ListSource、ListField 和 KeyField 属性。

(13) 同样地，TDBLookupComboBox 控件和表现和 TDBComboBox 类似，不同的是它显示的项目不是我们输入的，而是来自另外一个数据集。所以也需要设置它的 ListSource、ListField 和 KeyField 属性。

(14) TDBCheckBox 在外观上具有和 TCheckBox 控件相似的特征，但是在使用的时候具有一些自己的特性。首先它可以像我们理解的那样，用来显示数据库中的 Boolean 型字段的值。但是它的功能还不止如此，该控件具有一个 ValueChecked 属性和一个 ValueUnchecked 属性，它们都是字符串类型的。实际上，该控件将查看我们指定的 DataSource 中当前记录的 DataField 字段中的值，如果该值和 ValueChecked 属性中的值相同，那么该控件的 Checked 属性将为 True。如果当前记录的指定字段的值和 ValueUnchecked 属性中的值相同，那么此时控件的 Checked 属性将为 False。如果当前记录的指定字段中的值既不和 ValueChecked 中的值相同，也不和 ValueUnchecked 中的值相同，则该控件的 Enabled 属性将为 False，也就是说程序将会禁用该控件。例如在这里，把该控件的字段指定为 Species No，同时把 ValueChecked 属性设置为“90020”，把 ValueUnchecked 属性设置成“90030”。

(15) TDBRadioGroup 控件和 TRadioGroup 控件有些类似，但是它也具有自己的一些属性。在这个控件中，提供了一定数量的某个字段的可能值，用户可以通过在其中选择某个单选按钮来设置数据集中的字段值。和 TRadioGroup 控件一样，该控件具有一个 Items 属性，是一个字符串列表。如果当前记录中指定字段的值和 Items 属性中的某个字符串相同，那么该单选按钮将被选中。例如，如果在 Items 中有三个字符串：“Red”、“Yellow”和“Blue”，并且当前记录中的指定字段的值为 Blue，那么该控件中的第三个单选按钮将被选中。另外，该控件还具有一个 Values 属性，它也是一个字符串列表。如果当前记录中指定字段的值和

Items 中的各个字符串都不匹配，而和 Values 中的某个字段匹配，那么该控件中也会有一个单选按钮被选中。例如，对于上面的例子，如果 Values 属性中包含“Megenta”、“Yellow”和“Cyan”，并且当前记录的指定字段的值为“Megenta”，那么控件中的第一个按钮将被选中。此时，如果用户选择控件中的第三个按钮，“Cyan”将被传送到数据库中。如果 Values 属性值是空的，那么当用户选择控件中的一个单选按钮的时候，该项目对应的 Items 中的值将被传送到数据库中。

前面已经介绍过，对于显示单个记录的数据控件是比较容易理解的。在上面的过程中，基本上介绍了 Data Controls 选项卡中的所有显示单个记录的数据控件，它们基本上都有一个常规控件与之类似。所以关于这些控件的其他的没有介绍到的属性和事件，读者可以参考与之类似的常规控件的属性和事件来处理。

9.2.3　记录的导航和处理

上面介绍的是单个记录的显示问题，通过上面的控件，可以很容易地把数据集控件中当前记录的各个字段值显示出来。那么接下来的问题是，如何来改变当前的记录呢？如果能改变数据库的当前记录，那么上面单个记录显示的控件就没有任何用途。此时需要使用 Delphi 的数据控件中的 TDBNavitator 控件。

TDBNavitator 控件为用户提供了在数据集中浏览记录和处理记录的一个简单的控制。该工具包含了一系列的按钮，通过这些按钮，用户可以在数据集中前后移动，或者直接到数据集的第一个记录、最后一个记录，也可以插入一个新的记录、更新当前的记录、刷新数据的修改或者取消数据的修改、删除一个记录和刷新记录的显示等。在图 9.3 中显示了一个 TDBNavitator 控件的默认状态。

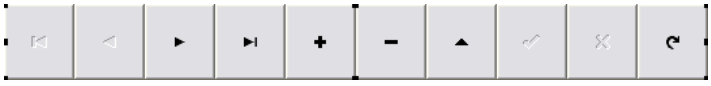

图 9.3　TDBNavitator 控件的默认状态

在表 9.2 中，按照从左到右的顺序列举了一个 TDBNavitator 控件可以具有的按钮的名称和功能。

表 9.2　TDBnavitator 控件的按钮

按钮	功能
First	调用数据集控件的 First 方法，当前记录变成数据集控件中的第一个记录
Prior	调用数据集控件的 Prior 方法，当前记录变成数据集控件中的当前记录的前一个记录
Next	调用数据集控件的 Next 方法，当前记录变成数据集控件中的当前记录的下一个记录
Last	调用数据集控件的 Last 方法，当前记录变成数据集控件中的最后一个记录
Insert	调用数据集控件的 Insert 方法，在当前记录前面插入一个新的记录，并把数据集控件置于 Insert 状态

续表

按钮	功能
Delete	删除当前记录。如果 ConfirmDelet 属性为 True，那么在删除之前会提示用户是否确定要删除该记录
Edit	把数据集控件置于 Edit 状态，以便用户能够修改当前记录中的数据
Post	把对当前记录的修改写入数据库中
Cancel	取消对当前记录的编辑，并把数据集控件置于 Browse 状态
Refresh	清除数据控件的显示缓存，然后从实际的数据集控件中重新载入该缓存。在其他应用程序有可能改变了实际的数据库中的数据时，这个按钮非常有用

在窗体上放置一个 TDBNavigator 控件时，它的所有的按钮都会显示出来。如果不加修改，那么此时用户就可以使用 TDBNavigator 控件的所有功能，甚至删除数据库中的记录等。所以应该对显示哪些按钮进行限制。

TDBnavigator 控件有一个 VisibleButtons 属性，通过它可以确定显示与隐藏哪些按钮。如果要在设计期改变这个属性的值，可以使用属性编辑器，如图 9.4 所示。

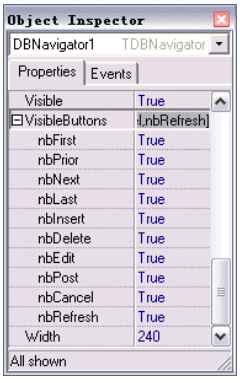

图 9.4 属性编辑器中的 VisibleButtons 属性

如果要显示某个按钮，比如 Edit 按钮，可以把属性编辑器中的 nbEdit 设置成 True，如果要隐藏该按钮，则可以把该属性设置成 False。

说明：

当隐藏或者显示某些按钮的时候，TDBNavigator 控件的大小并不改变，它上面的按钮则根据控件的情况自动进行调整，如果希望改变这些按钮的大小，需要调整 TDBNavigator 控件的大小。

在程序运行时，也可以根据用户的操作或者应用程序的状态来改变 TDBNavigator 控件上的按钮。例如，可以使用下面的程序来改变这些按钮：

```
procedure TForm1.CustomerCompanyEnter(Sender :TObject);
begin
```

```
if Sender = CustomerCompany then
begin
    DBNavigatorAll.DataSource := CustomerCompany.DataSource;
    DBNavigatorAll.VisibleButtons := [nbFirst,nbPrior,nbNext,nbLast];
end
else
begin
    DBNavigatorAll.DataSource := OrderNum.DataSource;
    DBNavigatorAll.VisibleButtons := DBNavigatorAll.VisibleButtons
                                     + [nbInsert,nbDelete,nbEdit,
                                     nbPost,nbCancel,nbRefresh];
end;
end;
```

为了在程序运行的时候能够给予用户一定的提示，可以设置 TDBNavigator 控件的 ShowHint 和 Hint 属性。这个操作和所有的常规控件一样。比如可以把 TDBNavigator 控件的 ShowHint 属性设置成“True”，然后按照下面的表格设置各个按钮的 Hint。

表 9.3　TDBNavigator 控件的各个按钮的 Hint

按钮	Hint
First	第一个记录
Prior	前一个记录
Next	下一个记录
Last	最后一个记录
Insert	插入一个新记录
Delete	删除当前记录
Edit	修改当前记录
Post	写入数据库
Cancel	取消对当前记录的编辑
Refresh	刷新数据

和其他数据控件一样，TDBNavigator 控件的 DataSource 属性指定了连接到数据集控件的数据源控件。在程序运行的时候改变 TDBNavigator 控件的 DataSource 属性可以使一个 TDBNavigator 控件能够为多个数据集控件进行导航。例如可以使用下面的代码来改变 TDBNavigator 控件的导航对象：

```
procedure TForm1.CustomerCompanyEnter(Sender :TObject);
begin
if Sender = CustomerCompany then
DBNavigatorAll.DataSource := CustomerCompany.DataSource
else
DBNavigatorAll.DataSource := OrderNum.DataSource;
```

```
end;
```

为了演示 TDBNavigator 控件的运行状态，这里制作了如图 9.5 所示的一个窗口。

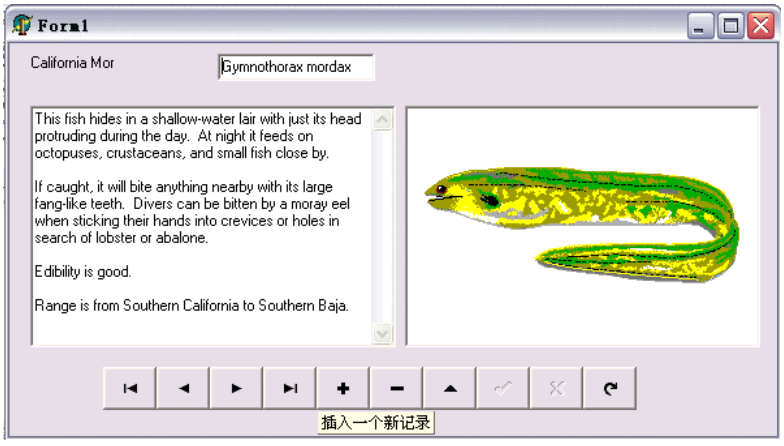

图 9.5 程序中 TDBNavigator 控件的使用演示

9.2.4 显示多个记录

在很多情况下，需要在一个窗体上显示多个记录，此时需要使用能够同时显示多个记录的数据控件。最典型的多记录数据控件就是 TDBGrid 控件。TDBGrid 控件提供了多字段、多记录的显示方式，使得应用程序更有吸引力。

在使用 TDBGrid 控件的时候，有三个因素会影响该控件中记录的显示形式。

- 用 Columns 编辑器生成的 TDBGrid 控件的固定 Column 对象的存在。固定 Column 对象为 TDBGrid 控件和数据显示形式的设置提供了更大的灵活性。
- 显示在 TDBGrid 的数据集控件中的固定字段对象的创建。
- 数据集控件中关于网格显示的 ADT 和数组字段的 ObjectView 属性。

下面先来介绍一些关于 TDBGrid 控件的一些基本属性。

TDBGrid 控件有一个 Columns 属性，它是一个 TDBGridColumns 类型的属性。该类型的属性是 TColunm 对象的集合，这些 TColumn 对象就代表了 TDBGrid 对象中的各个列。可以使用 Column 对象编辑器来设置各个列的属性。TDBGrid 控件的 Columns 属性有一个 State 属性，该属性指明了 TDBGrid 控件中是否存在固定的 Column 对象。这个属性只有在程序运行的时候才能访问。在默认的情况下，State 属性的值为 csDefault，这就意味着在 TDBGrid 中不存在固定的 Column 对象。在这种情况下，TDBGrid 控件中的数据显示就完全取决于 TDBGrid 的数据集控件中的字段属性。此时 TDBGrid 控件的 Columns 对象将动态地自动创建，每个 Column 对象将和数据集控件中的一个字段相关联。任何对字段属性的修改都会立即反应在 TDBGrid 控件中。

如果可能会在程序运行时打开一些数据表并查看其中的内容，那么这种动态创建 Column 对象的方法是十分有用的。这是因为我们并不知道要打开的数据库结构、类型等，也许是先打开一个 Paradox 的数据表，然后又打开了一个 SQL 的查询。

当然在设计期和运行期都可以改变 TDBGrid 控件列的外观。我们在这样做的时候，实际上是在修改显示在列中对应字段对象的属性。所以对 TDBGrid 控件中列的修改存在一个生命期的问题。例如，如果改变了一个列的宽度，那么实际上修改的是和这个列关联的字段的 DisplayWidth 属性。如果这个字段是一个固定字段，那么即使我们的程序在某个时刻关闭了该数据集控件，这个 DisplayWidth 属性仍然是保留的，所以再次打开该数据集控件的时候，所设置的宽度仍然有效。但是如果列对应的字段是程序动态生成的，那么当数据集控件关闭的时候，这些字段也就消失了，所以关于它们的设置自然就不存在了。

这里将编写一个程序用来显示各种数据库文件，当然这里使用的是本地的数据库。此处主要是使用了一个按钮的 OnClick 事件，代码如下：

```
procedure TForm1.Button1Click(Sender: TObject);
begin
if OpenDialog1.Execute then
begin
    dmgrid.tbExample.Active := false;
    dmgrid.tbExample.DatabaseName := ExtractFileDir(OpenDialog1.FileName);
    dmgrid.tbExample.TableName := Extractfilename(OpenDialog1.FileName);
    dmgrid.tbExample.Active := Active;
end;
end;
```

在上面的程序中，也演示了如何直接把路径名称和文件名称赋给 Table 控件。程序的运行结果如图 9.6 所示。

在上面的程序中，没有对 TDBGrid 控件进行任何的设置，除了赋给它必须的 DataSource 属性之外，所以 TDBGrid 控件的表现也就是该控件的默认表现。

我们当然希望能够按照自己的需要改变这个控件的显示内容，这里我们称之为自定义网格显示。自定义网格显示具有以下优点。

- ❖ 通过网格显示的变化，可以用多个网格显示同一个数据集中的内容。
- ❖ 通过网格显示的变化，可以在程序中允许用户改变网格的外观，但是不会影响数据集控件中的字段。

但是需要注意的是，自定义网格显示最好应用于那些在设计期就知道其结构的数据集。对于那些需要在程序运行中来动态地确定结构的数据集控件，最好不要使用自定义网格显示。在自定义网格显示时，处理最多的就是网格控件中的固定列。

下面将通过一个程序来介绍如何在自己的数据库应用程序中自定义网格显示。

(1) 同前面介绍的程序一样，先建立一个新的应用程序，并建立一个 Data Module 模块，

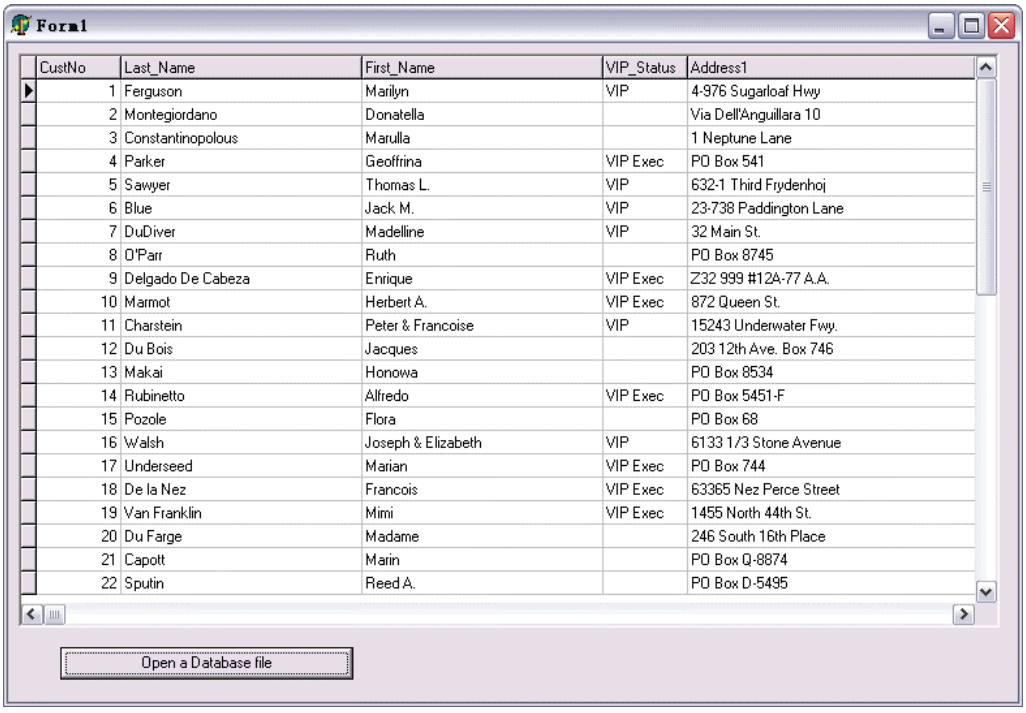

图 9.6　用 TDBGrid 控件显示不同的数据表中的内容

然后在模块上放置一个数据集控件和一个数据源控件：tbExam **和** dsExam**。**

(2) **选择** Database **菜单中的** Explorer**，此时会调用** Delphi **中的数据库浏览程序，如图** 9.7 **所示。**

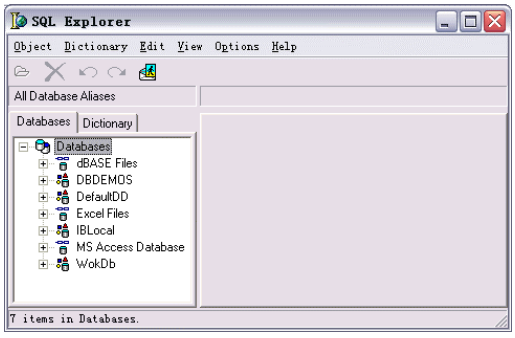

图 9.7　Delphi 的数据库浏览程序

(3) **在左边的框中右击鼠标，并从弹出的快捷菜单中选择** New**，此时会显示一个对话框，要求你选择要建立的数据库的类型，如图** 9.8 **所示。**

(4) **选择所需要的类型，然后单击** OK **按钮，便在左边的框中添加了一个新的数据库别名。修改它的名字后，选中它，在它右边的框中选择** Path**，这里可以指向计算机上的某个目录。**

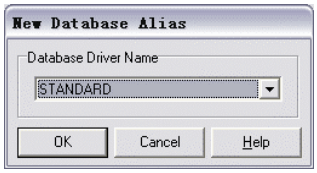

图 9.8　建立新的数据库对话框

(5) 然后选择 Object 菜单中的 Apply 菜单，保存工作。选择 Tools 菜单中的 Database Destop，此时会启动 Delphi 附带的 Database Desktop 程序。

(6) 在该程序中建立一个如图 9.9 所示的 Table。

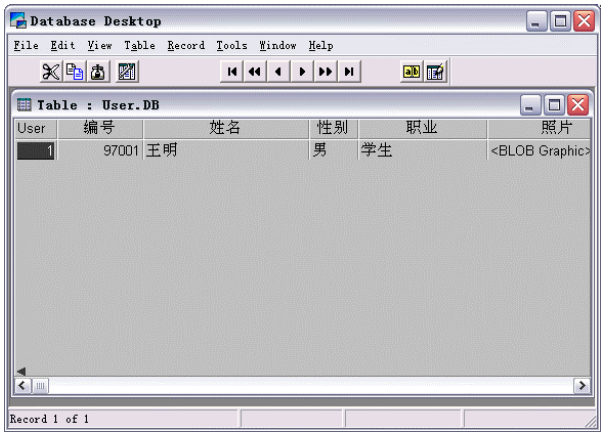

图 9.9　Database Destop 程序

(7) 把 Data Module 单元中的 tbExam 的 DatabaseName 属性设置成这里新建的“WokData”，然后把 TableName 属性设置成新创建的“User.DB”。并把 Active 属性设置成“True。”

现在，我们已经完成了示例程序的准备工作，下面的任务是创建这个数据库应用程序的界面了。在本示例程序中，将介绍如何来自定义 TDBGrid 控件的显示形式。

(8) 把窗口切换到应用程序的主窗口，也就是 Form1 窗口中，修改窗口的 Caption 属性，并在窗口的定义单元中的 Uses 语句中加入 DmGrid（Data Module 模块的名称）。

(9) 在窗口上放置一个 TDBGrid 控件，一个 TPanel 控件，把 TDBGrid 控件的 Align 属性设置成“alClient”，把 TPanel 的 Align 属性设置成“alLeft”。在 TPanel 上放置一个 TImage 控件，并把 Align 属性设置成“alClient”，把 Stretch 属性设置成“True”。并载入一个图片。

(10) 在窗体上放置一个 TImage 控件，载入一个图像，把 TImage 控件的 AutoSize 属性设置成“True”，让控件根据图片的大小来确定自己的尺寸。然后在窗体上放置一个 Tlabel 控件，把它的 Autosize 属性设置成“False”，并调整它的大小，使它能够和 TImage 控件完全重合。把 TLabel 控件的 Transparent 属性设置成“True”。

(11) 选中叠放一起的 TImage 控件和 TLabel 控件，然后选择 Component 菜单中的 Create

Component Template 命令，此时会显示一个对话框，如图 9.10 所示。

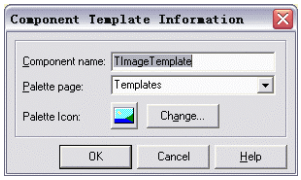

图 9.10　创建控件模板

(12) 在这里，可以修改这个模板的名称或者修改它们在控件面板上的图标。设置好选项之后，单击 OK 按钮，便在 Delphi 中创建了一个控件模板。在控件面板中找到该模板，然后在窗体上单击，便可以同时在窗口上放置多个已经设置好大多数属性的控件。

说明：

通过控件模板的功能，可以把很多控件组合成我们需要的样式，那么程序设计就变得更加简单了。

(13) 修改各个 TLabel 控件的标题，并设置 TDBGrid 控件的 Color 属性（网格中的单元格的颜色）、FixedColor 属性（固定单元格的颜色）、Font 属性（网格中的单元格的字体格式）和 TitleFont 属性（固定单元格中的字体格式）。此时我们创建了一个如图 9.11 所示的应用程序界面。

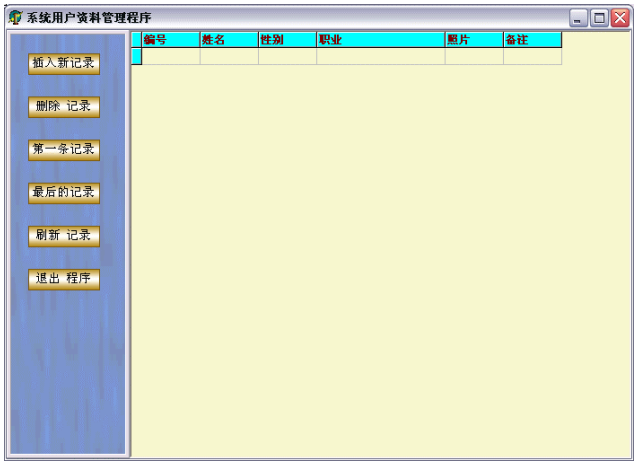

图 9.11　示例程序的界面

把 TDBGrid 控件的 DataSource 属性设置成 dmGrid 数据模块中的“ dsExam ”。从界面中可以看出，我们在左边放置的 TLabel 控件和 TImage 控件是为了作为按钮使用，所以必须处理它的一些鼠标事件。下面列出了程序中关于这些控件的鼠标事件处理程序，在这些代码中，又一次使用了事件处理过程中的 Sender 参数，使得我们编写的这几个事件处理程序可以用在这里的所有 TLabel 控件上。

```
procedure TForm1.Label1MouseDown(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
```

```
(sender as Tlabel).font.Color := clBlue;
end;

procedure TForm1.Label1MouseMove(Sender: TObject;
                                 Shift: TShiftState; X,
  Y: Integer);
begin
(sender as Tlabel).font.Color := clpurple;
end;

procedure TForm1.Label1MouseUp(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
(sender as Tlabel).font.Color := clpurple;

end;

procedure TForm1.Label1MouseLeave(Sender: TObject);
begin
(sender as Tlabel).font.Color := clBlack;
end;
```

设计这个应用程序的目的是能够管理一个用户数据库。所以这里着重介绍一些这些数据库管理功能。左边按钮中的功能就不再仔细介绍了，它们分别调用了数据集控件中的方法，其功能和 TDBNavigator **控件中的按钮是一样的。我们的工作重点是通过自定义** TDBGrid **控件的行为来完成输入记录的功能。要处理的数据表有** 6 **个字段，这里分别采用三种形式来实现在** TDBGrid **控件中的编辑功能。对于前两个字段，编号和姓名，我们就直接使用** TDBGrid **控件的编辑功能，这个功能可以通过设置** TDBGrid **控件的** Options **属性来完成，如表** 9.4 **所示。**

表 9.4　TDBGrid 控件的 Options 属性

值	说明
dgEditing	用户可以在网格中编辑数据。如果 Options 属性中 dgRowSelect 为 True，那么 dgEditing 将被忽略
dgAlwaysShowEditor	控件总是处于编辑状态。也就是说，用户选择某个单元格时自动进入编辑状态。如果 dgEditing 为 True，或者 dgRowSelect 为 False，则该属性值将不起任何作用
dgTitles	是否显示网格的列的标题
dgIndicator	是否在第一列中显示一个小的箭头指明当前记录
dgColumnResize	每个列是否能够改变大小
dgColLines	在列与列之间是否画线

续表

值	说明
dgRowLines	在行与行之间是否画线
dgTabs	用户能够使用 Tab 键或者 Shift+Tab 键来在网格中移动
dgRowSelect	用户可以选定整行，就像选定某个单元格一样
dgAlwaysShowSelection	选定的单元格总是高亮显示，即使在控件失去焦点时也是一样
dgConfirmDelete	当用户在网格中使用 Ctrl+Delete 键来删除一行时，会显示一个提示框，让用户进行确认
dgCancelOnExit	当用户退出插入记录的时候，不向数据集控件提交任何更新信息

这个方式的实现比较简单。现在来讨论性别和职业两个字段的编辑方式。在属性编辑器中，单击 Columns 属性旁边的箭头，此时会显示一个如图 9.12 所示的列编辑器。

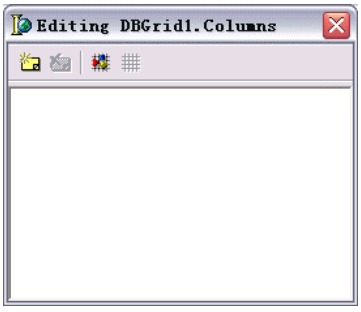

图 9.12　列编辑器

在这个编辑器中，可以编辑 TDBGrid 控件的固定列。固定列是一定会在网格中显示的。所以如果数据集的结构是未知的，将不得不进行大量的关于数据集结构的编程处理，这也就是我们建议自定义网格显示最好用在已知结构数据集上的原因。

在这个对话框中有四个按钮，可以新建、删除列，也可以把与 TDBGrid 控件关联的数据集控件中的所有字段都作为固定列添加进来，还可以恢复默认的值。此外，在这里还可以调整各个字段的显示顺序，比如可以将数据集中后面的字段放在前面显示，而把前面的字段放在后面显示。

利用固定列的另外一个重要的优势是，一旦定义了 TDBGrid 控件的固定列，程序将不再动态地创建和数据集中字段关联的列，而是只显示固定的列。所以我们可以有选择地显示数据集中的部分内容。

下面继续介绍示例程序。我们希望能够在 TDBGrid 的单元格中实现下拉列表的功能，也就是说，对于某些字段，它们的可能值是一定的，此时可以提供一个列表，让用户在输入信息时直接从这个列表中进行选择。下面就来实现这个功能。

在这里选择了 Add All Field 按钮，把数据集中的所有字段都添加到固定列中。然后，选中其中的一个固定列，比如对应于职业字段的列，然后在属性编辑器中找到 PKlist 属性，单击它旁边的按钮，此时会进入类似于 TListBox 控件的 Items 属性编辑器窗口。然后在窗口中输入这个字段可能的值。对于性别字段来讲就只有“男”和“女”两个项目，而职业可能有很多。

在输入了相应的信息之后，我们可以运行这个应用程序，此时程序如图 9.13 所示。从图中可以看出，在职业对应的列中，当进入其中的一个单元格时，它的旁边会显示一个按钮，单击该按钮，便出现了一个下拉列表，我们可以从中选择一个需要的项目，此时项目便出现在单元格中。

图 9.13　使用下拉列表形式的显示方式

在 Column 对象的属性编辑器中，有一个 DropDownCount 属性，这个属性和 TComboBox 控件的对应属性类似，通过修改这个属性，可以改变下拉列表中每屏显示的项目的个数。

在这个应用程序中，另外两个字段具有一定的特殊性，它们一个是图像字段，一个是备注字段，都是 BLOB 字段。所以实现它们的编辑功能需要另外的窗口。此时可以利用 TDBGrid 控件的 Column 对象来改变 TDBGrid 控件的显示形式，使得它在单元格中能够显示一个“ ... ”按钮，当用户单击这个按钮时，便进入了该字段的编辑窗口。

仍然是在列编辑器中，选择图片字段对应的列，然后在属性编辑器中把 ButtonType 属性修改为“ cbsEllipsis ”。此时运行程序，如果选择图片列中的某个单元格，此时的单元格会如图 9.14 所示。

图 9.14　显示了小按钮的单元格

然后编写 TDBGrid 控件的 OnEditButtonClick 事件，我们编写了下面的代码。

说明：

注意该事件发生在 TDBGrid 控件中的小按钮被单击时。比如在这个程序中，有两个字段会在单元格中显示该按钮，但是不管单击哪个按钮，它们将引发同一个事件。所以我们在程序处理的时候要注意这一点。

```
procedure TForm1.DBGrid1EditButtonClick(Sender: TObject);
begin
dmGrid.tbExam.Edit;
if DbGrid1.SelectedField.FieldName = '照片' then
begin
```

```
    form3.Caption := DbGrid1.SelectedField.FieldName;
    if not DbGrid1.SelectedField.IsNull then
        form3.Image1.Picture.Assign(dbGrid1.SelectedField);
    if form3.showmodal = mrOk then
    begin
        DbGrid1.Selectedfield.Assign(form3.Image1.Picture.Graphic);
    end;
end
else
begin
    form4.Caption := DbGrid1.SelectedField.FieldName;
    form4.ShowModal;
end;
dmgrid.tbExam.Post;
end;
```

在程序的开始，用 Edit 方法使数据集处于编辑状态，在程序的结尾我们使用了 Post 方法，把修改后的数据更新到数据集中。

在程序中，我们用和列关联的字段名称来判断所选择的是哪个列。这里使用了两种获得选定字段值的方法。在照片列中，使用的是把字段的值赋给设置窗口中变量的方式。我们设计了一个窗口用来处理该类字段，该窗口如图 9.15 所示。

图 9.15　处理照片字段的窗口

这个窗口的代码如下：

```
procedure TForm3.BitBtn1Click(Sender: TObject);
begin
if OpenDialog1.Execute then
    Image1.Picture.LoadFromFile(OpenDialog1.FileName);
end;
procedure TForm3.FormPaint(Sender: TObject);
```

```
var bit:TBitmap;
begin
    bit:=TBitmap.Create;
    bit.LoadFromFile('Blank.bmp');
    canvas.Brush.Bitmap := bit;
    canvas.Rectangle(canvas.ClipRect);
    bit.Free;
end;
```

这里用一个自定义的填充方式绘制了窗体的背景。

在定义处理备注字段的窗口的时候，在窗口中放置了一个 TDBMemo 控件，然后把它的 DataSource 属性设置成了 dmGrid 数据模块中的 dsExam，并指定它的关联字段为备注。此时我们的修改便直接作用于数据集控件上，该窗口如图 9.16 所示。

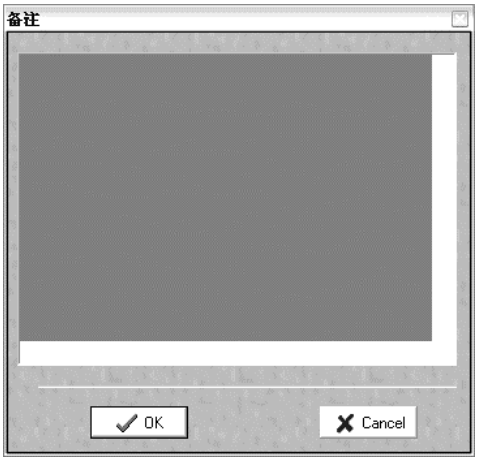

图 9.16　处理备注字段的窗口

该窗口的代码非常简单，只有上一个窗口的 OnPaint 事件中的代码。

对于照片和备注字段，我们在设置后进行了提交，所以所做的修改便直接保存到了数据库中。下面我们来看看另外四个字段的更新问题。

程序中使用了 TDBGrid 控件的 OnColExit 事件来处理字段的更新。该事件发生在输入焦点将要离开某个单元格的时候。此时 TDBGrid 控件的 SelectedField 还没有被赋予新的值，所以仍然可以用这个属性来判断当前选择的是哪个字段。随后将发生 OnColEnter，此时的 SelectedField 已经被赋予新的值了。这里的代码非常简单，我们只是调用了数据集控件的 Edit 方法和 Post 方法。

```
procedure TForm1.DBGrid1ColExit(Sender: TObject);
begin
dmgrid.tbexam.edit;
dmgrid.tbExam.Post;
```

```
end;
```

在我们的程序中，当用户改变了某个单元格的内容时，如果用户进入了其他单元格，修改便提交到了数据集控件中。

从上面的程序中可以看出，TDBGrid 已经相当完美，通过自定义它的显示形式，我们可以很容易地实现各种各样的显示方式。但是如果从数据编辑的角度来说已经是很完美了，特别是下拉列表形式的使用，但是从显示上来说还有不足的地方，比如控件中的照片字段就无法在网格中显示。由于 TDBGrid 控件的单元格高度是固定的，所以我们几乎没有办法来显示图片。因为虽然可以通过自己定义的方式来在单元格中显示图形，但是由于高度问题，图片就不得不改变大小。如果要自定义单元格中的内容的显示方式，可以使用控件的 OnDrawColumnCell 事件。例如在下面的事件处理程序中，我们对图像类型的字段进行了特殊的处理：

```
procedure TForm1.DBGrid2DrawColumnCell(Sender: TObject; const Rect: TRect;
   DataCol: Integer; Column: TColumn; State: TGridDrawState);
var r:TRect;
      pic:TPicture;
begin
    r:=Rect;
    Pic:=TPicture.Create;
    if Column.FieldName = 'Graphic' then
    begin
       Pic.Assign(Column.Field);
       r.Right := r.Left + Pic.Width;
       r.Bottom := r.Top + Pic.Height;
       dbgrid2.Canvas.StretchDraw(rect,Pic.Graphic);
    end
    else
       dbgrid2.DefaultDrawDataCell(Rect,Column.Field,state);
    Pic.Free;
end;
```

对于一个使用 DBDEMOS 数据库中的 biolife.db 数据表的数据集控件来说，和它关联的 TDBGrid 控件通过上面的程序，将如图 9.17 所示。

当然，我们实现了显示图片的功能，但是效果并不好，图像都明显变形了。作为一个带有图像的数据库信息显示工具，TDBGrid 控件不能很好地完成任务。当然，我们可以从 TGrid 控件中派生的网格控件来解决这个问题。但是对于文本和图像的位置安排将是一个让人十分

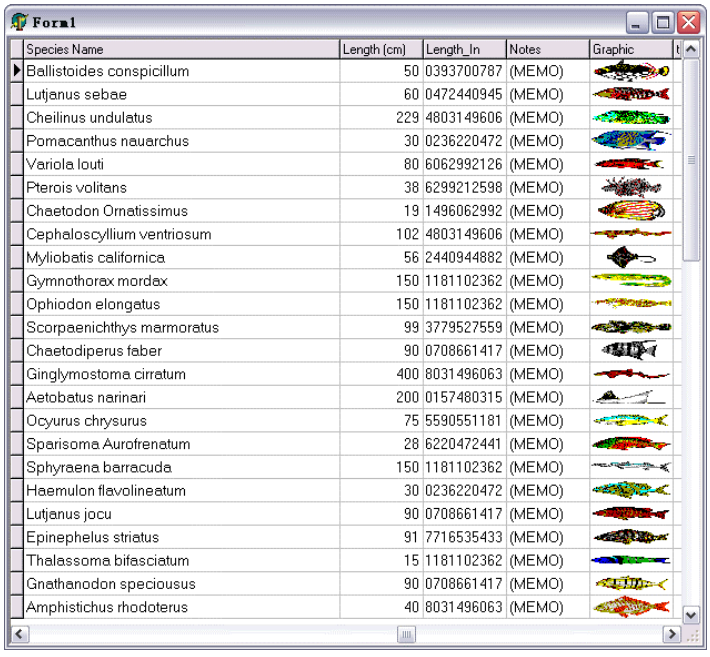

图 9.17　在 TDBGrid 控件中显示图像

头痛的事情。因为如果把显示图像的单元格扩大，那么显示文本的单元格也随之发生了变化，而用这么大的单元格来显示文本显然有些浪费空间了。

此时可以使用数据控件中的 TDBCtrlGrid 控件来实现这个目标。在窗体上放置一个 TDBCtrlGrid 控件，它的默认状态如图 9.18 所示。

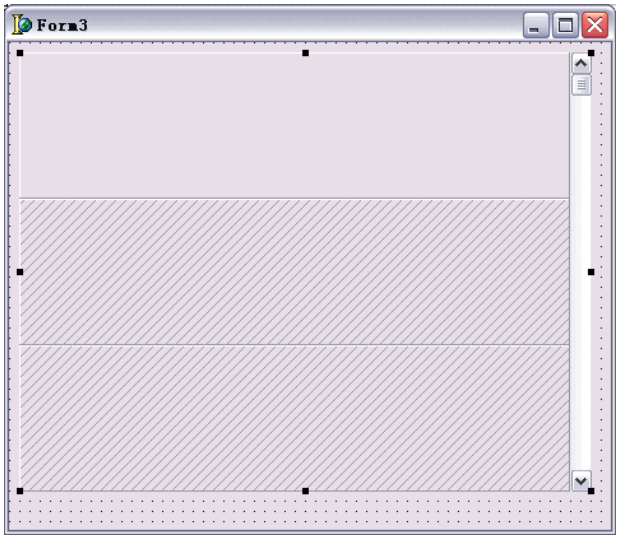

图 9.18　放置在窗体上的 TDBCtrlGrid 控件

这个控件的主要功能就是把 Panel 控件作为自己的子控件，运行用户在 Panel 上放置任何数据控件来访问数据集中的数据。当程序运行时，该控件将以我们设计的这个 Panel 为模板，生成显示数据集中所有记录的 Panel。这个 Panel 在设计状态下就是图 9.18 中最上面的一个。我们可以在这个 Panel 上放置任何控件，所有的控件都会在运行时被复制到每个 Panel 中。如果上面的控件和数据集中的某个字段相关联，控件还会把数据集中的数据自动填充到对应的控件中。

下面介绍一下 TDBCtrlGrid 控件的一些关键的属性和方法。

- AllowDelete 属性：如果该属性为 True，那么当用户按下 Ctrl+Del 键时，将会删除当前的记录。当然此时的数据集控件必须处于编辑状态。
- AllowInsert 属性：如果该属性为 True，那么当用户按下 Ctrl+Insert 键时，将会插入一条记录。
- ColCount 属性：确定了控件中的列数，在默认的情况下是 1。
- Color 属性：可以指定该控件中的颜色。
- DataSource 属性：同其他数据控件一样，该属性是用来确定与之关联的数据集的。
- Orientation 属性：决定了控件滚动的方向，如果该属性为 goVertical，控件将按照纵向来组织记录，并显示一个垂直滚动条；如果该属性为 goHorizontal，则按照横向来组织记录，并显示一个水平滚动条。
- 三个关于 Panel 的属性：PanelBorder 属性确定了控件中的 Panel 的边框的类型，PanelHeight 属性确定了控件中 Panel 的高度；PanelWidth 属性确定了控件中 Panel 的宽度。
- RowCount 属性：和 ColCount 属性相对应，确定了控件上每次显示的记录条数。
- SelectedColor 属性：指定了用户在选定一个记录的时候该 Panel 的颜色。
- ShowFocus 属性：如果该属性为 True，那么当前选定的记录所对应的 Panel 将显示一个虚框，表示焦点在该记录上。

TDBCtrlGrid 控件具有两个重要的方法，一个是 KeyDown，一个是 DoKey，这两个方法通常是配对使用的。

KeyDown 方法的声明如下：

```
procedure KeyDown(var Key: Word; Shift: TShiftState); override;
```

Shift 参数代表了当前的按键状态，这个参数可以从 OnKeyDown 事件中获得。在该方法的参数中，Key 是要返回的值。KeyDown 的功能就是根据 Shift 的值得到当前的 Key 的值。它们之间的对应关系如表 9.5 所示。

表 9.5　按键和 Key 值的转换关系

按键（Shift）	Key 的值	按键（Shift）	Key 的值
LeftArrow	gkLeft	Return	gkEditMode
RightArrow	gkRight	F2	gkEditMode
UpArrow	gkUp	Insert	gkAppend
DownArrow	gkDown	Ctrl+Insert	gkInsert
PageUp	gkPageUp	Ctrl+Delete	gkDelete
PageDown	gkPageDown	Escape	gkCancel
Home	gkHome	All other keys	gkNull
End	gkEnd		

DoKey 方法的声明如下：

```
procedure DoKey(Key: TDBCtrlGridKey);
```

该方法根据 Key 参数代表的含义，进行指定的操作。Key 参数的可能值与要进行的操作见表 9.6。

表 9.6　TDBCtrlGridKey 类型

值	操作
gkNull	不进行任何操作
gkEditMode	锁定 EditMode 属性
gkPriorTab	移动到上一个 Panel
gkNextTab	移动到下一个 Panel
gkLeft	向左移动一个 Panel
gkRight	向右移动一个 Panel
gkUp	向上移动一个 Panel
gkDown	向下移动一个 Panel
gkScrollUp	向上移动一个 Panel
gkScrollDown	向下移动一个 Panel
gkPageUp	向上移动 ColCount * RowCount 个记录
gkPageDown	向下移动 ColCount * RowCount 个记录
gkHome	移动到第一个记录
gkEnd	移动到最后一个记录
gkInsert	在当前记录前插入一个新的记录，并把 EditMode 设置成 True
gkAppend	在数据集的末尾插入一个记录，并把 EditMode 设置成 True
gkDelete	从数据集中删除当前记录，并把 EditMode 设置成 False
gkCancel	取消提交给数据集的修改，并把 EditMode 设置成 False

这个控件的使用并不复杂，我们在图 9.19 所示的窗口中设置了一个 TDBCtrlGrid 控件用来显示 bioslife.db 中的数据。中间的颜色稍深，表示是当前记录。

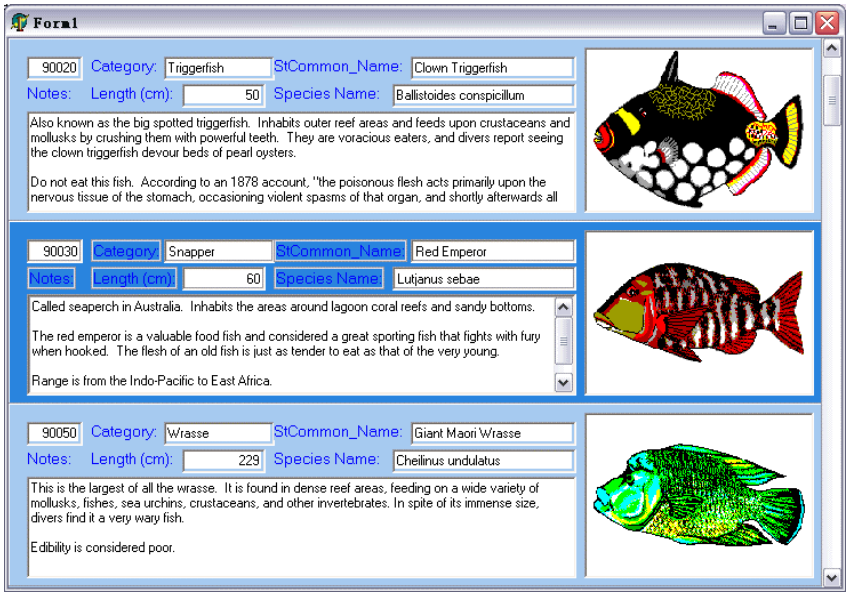

图 9.19　使用 TDBCtrlGrid 控件显示多个记录

在上面的程序中，除了数据控件之外，还使用了五个常规控件，它们是 Additional 选项卡中的 Static Text 控件。

9.3　使用数据集控件

在上一节的内容中，介绍了设计数据库应用程序界面方面的有关问题。在 Delphi 中还有这个方面的其他一些功能，比如在使用 Decision Cub 选项卡上的控件时，会发现它上面的内容也和设计用户界面有关，当然也有数据集控件。它们的使用和这里介绍的数据控件很相似，所以就不详细解释了。在上面的程序中，我们使用了 TTable 控件，它是使用 BDE 引擎的数据集控件，我们并没有详细地介绍该控件的用法，只是设置它的两个属性，使我们能够在设计用户界面时使用而已。至于怎样连接到数据库，我们并没有详细地说明。在本节的内容中，将详细介绍连接到数据库方面的问题。

我们已经知道，数据库有很多种类，它们都有各自的特点。所以 Delphi 为每种类型的数据库都设计了相应的数据集控件，分别位于 BDE、ADO 等选项卡中。也许这样的控件使我们觉得无从下手，但是没有关系，实际上它们的基本用法都是相同的。如果你清楚地掌握了其中一类数据集控件的操作方法，那么只要再了解一些关于其他数据库的特点，使用其他种类的数据集控件就易如反掌了。所以，在本部分的内容中，将以基于 BDE 的数据集控件为主，来介绍如何使用 Delphi 中的数据集控件。完成了对数据集控件的介绍之后，我们还会举一个关于 ADO 数据集的例子，读者可以看出，从使用方法的角度来说，它们的区别并不大。

9.3.1　数据库连接概述

大多数数据集控件都能直接连接到数据库服务器上。一旦连接上了，数据集控件将自动和服务器进行通信。打开一个数据集时，数据集从服务器上获取数据；提交记录的时候，数据集把这些数据送到服务器。

1. 连接控件的访问机制

多个数据集控件可以使用同一个连接控件，也可以使用自己的连接。每种数据集控件都使用自己类型的连接控件来连接到数据库服务器，这些控件都是用来处理一种数据访问机制的。

表 9.7 列出了这些数据访问机制和相关的连接控件。

表 9.7　数据库连接控件

数据访问机制	连接控件
The Borland Database Engine (BDE)	Tdatabase
ActiveX Data Objects (ADO)	TADOConnection
DbExpress	TSQLConnection
InterBase Express	TIBDatabase

不论使用什么样的数据访问机制，都可以明确地创建用来管理连接、处理与数据库服务器通信的连接控件。对于基于 BDE 和基于 ADO 的数据集来说，也可以通过数据集的属性来描述数据库连接，并让数据集控件产生一个明确的连接。对于支持 BDE 的数据集来说，可以利用数据集控件的 DatabaseName 来连接到数据库。对于基于 ADO 的数据集来说，可以使用 ConnectionString 属性来连接到数据库，在这些情况下，一般不需要直接创建连接控件。对客户端/服务器应用程序来说，因为它们有很多的用户，对数据库连接就有很多不同的要求，此时就应该创建连接控件来满足应用程序的需要。明确的连接控件使我们具有更多的控制能力。通常来说，我们在完成下面几类任务时需要使用连接控件。

- 自定义数据库服务器登录支持。
- 控制事务，并指定事务的隔离层次。
- 在不使用数据集控件的情况下执行 SQL 命令。
- 对连接到同一个数据库的所有打开的数据集进行操作。

另外，如果程序中有多个数据集控件使用了同一个服务器，那么使用连接控件就更容易一些。这是因为只需要在一个位置指定到服务器的连接就可以了。

2. 连接、登录和断开数据库服务器

在连接到一个数据库服务器之前，我们的应用程序必须提供一些描述数据库服务器的关键信息。尽管不同的连接控件封装了不同的属性来指定服务器，但是，一般来说，它们都提供了一种让我们指定服务的方式，并提供了用来控制连接的一系列参数。当然，对于每种服务器来说，这些参数可能都是不一样的。一旦我们确认了要连接的服务器和连接参数，就可以利用一个连接控件打开一个连接。此时连接控件会根据与数据库服务器的连接情况产生一些事件，让我们在这些事件中编写代码来处理这些情况。

可以使用连接控件的 Open 方法或者 Connected 属性来连接到数据库。在设置 Connected 属性时，连接控件会产生一个 BeforeConnect 事件，在该事件中，可以执行任何初始化方面的工作。通常来说，我们会在这个事件中处理连接参数。当连接成功时，连接控件会产生一个 AfterConnect 事件，在这里可以编写那些需要在数据库一打开就要执行的操作。连接成功之后，只要有一个打开的数据集控件在使用这个连接，那么这个连接就不会中断。当没有任何使用该连接的数据集控件打开时，连接控件将断开连接。有些连接控件具有一个 KeepConnection 属性，把这个属性设置成“True”，那么即使所有与该连接相关联的数据集控件都关闭了，连接也仍然保持着。

和连接数据库服务器的方式类似，断开一个数据库服务器，也有两种方式，把 Connected 属性设置成“False”或者调用 Close 方法。当 Connected 事件为 False 的时候，将产生一个 BdforeDisconnect 事件，最后将产生 AfterDisonnect 事件。

图 9.20 数据库登录窗口

大多数数据库服务器都包含了一个安全功能来防止未被授权的访问。一般来说，服务器在提供数据之前都会要求提供一个用户名称和一个密码。在设计期，如果我们要登录一个服务器，此时会显示一个标准的登录对话框，提示我们输入用户名和密码，该对话框如图 9.20 所示。

在程序运行的时候，我们可以选择以下三种方式来处理用户登录的问题。

- 使用默认的登录窗口来处理。此时可以把连接控件的 LoginPrompt 属性设置成“True”，并把 DBLogDlg 加入到程序的 Uses 语句中。此时应用程序会显示标准的登录窗口。
- 自己提供登录的相关信息。比如，如果是使用 BDE、dbExpress 和 Interbase Express 数据集，用户名和密码可以通过控件的 Params 属性来访问。而对于 ADO 数据集控件来说，用户名和密码可以包含在 ConnectionString 属性中。如果我们提供了用户名和密码，那么应该把 LoginPrompt 属性设置成“False”。例如在下面的例子中，我们在一个 SQL 连接控件的 BdforeConnect 事件处理程序中提供了需要的用户名和密码。

```
procedure TForm1.SQLConnectionBeforeConnect(Sender: TObject);
begin
  with Sender as TSQLConnection do
  begin
    if LoginPrompt = False then
    begin
    Params.Values['User_Name'] := 'SYSDBA';
    Params.Values['Password'] := Decrypt(Params.Values['Password']);
  end;
  end;
end;
```

- **提供自己的登录事件处理。当我们要求登录的时候，如果需要用户名和密码，连接控件会产生一个事件，可以在这个事件中完成对用户名和密码的处理。对于不同的控件可能会产生不同的事件，比如** TDataBase、TSQLConnection **和** TIBDatabase **控件来说，会产生** OnLogin **事件。**TADOConnecton **控件会产生** OnWillConnect **事件。它们虽然名称不同，但基本功能是类似的。**

3. **管理事务**

我们前面提到过，事务就是一组操作，这些操作必须都成功，否则该事务将失败，并且其中的所有操作都将被取消。

对于 TADOConnection **控件来说，我们可以使用** BeginTrans **方法来开始一个事务，例如下面的语句：**

```
Level := ADOConnection1.BeginTrans;
```

该方法返回的是开始事务的嵌套层次。在服务器开始该事务之后，ADO **连接会产生一个** OnBeginTransComplete **事件。**

对于 TDatabase **控件来说，我们可以使用** StartTransaction **方法。**TDataBase **不支持嵌套或者交叉的事务。如果调用该方法时，另一个事务正在进行，将会引发一个异常。为了避免这个问题，在调用** StartTransaction **方法之前，应该检查** inTransaction **属性。例如可以使用下面的代码来开始一个事务：**

```
if not Database1.InTransaction then
  Database1.StartTransaction;
```

TSQLConnection **控件也使用** StartTransaction **方法，但是** TSQLConnection **控件的该方法能够提供更多的控制。需要说明的是，**TDatabase **和** TSQLConnection **的该方法都不会产生事件。**TIBDatabase **控件提供了更多的控制选项。**

在开始一个事务之后，这个事务当然会在适当的时候结束。可以使用下面的代码来结束

一个 TDatabase 的事务：

```
database1.Commit;
```

对 TSQLConnection 控件来说，需要使用下面的代码来结束一个事务：

```
SQLConnection1.Commit(TD);
```

对 TIBDatabase 控件来说，需要使用下面的方法：

```
IBDatabase1.DefaultTransaction.Commit;
```

对 TADOConnection 来说，可以使用 CommitTrans 方法：

```
ADOConnection1.CommitTrans;
```

对于 ADO 连接来说，如果成功地结束了一个事务，将会引发 OnCommitTransComplete 事件。

如果希望结束一个没有完成的事务，可以使用下面的代码：

```
database1.Rollback;
SQLConnection1.Rollback(TD);
IBDatabase1.DefaultTransaction.Rollback;
ADOConnection1.RollbackTrans;
```

如果成功地执行了结束事务的事件，对于 ADO 连接控件来说，会引发一个 OnRoolbackTransComplete 事件。

在开始一个事务时，我们不能确定数据库服务器上是否有其他的事务也在执行。事实上很可能存在一些同时进行的事务，特别是对于大型的远程数据库服务器来说。所以需要指定一个事务的隔离层次。隔离层次决定了同时处理一个数据表的多个事务之间的相互影响。有三种可能的事务隔离层次。

- DirtyRead：如果事务的隔离层次是 DirtyRead，那么事务会看到其他事务带来的变化，即使它们还没有完成。没有完成的事务不是稳定的，因为其他程序可能会在任何时候取消这个事务。这种隔离层次是最低的，对很多数据库服务器来说，不提供这种隔离层次的支持，例如 Oracle、Sybase，MS SQL 和 InterBase。
- ReadCommitted：如果事务的隔离层次是 ReadCommitted，那么只有其他事务完成后的变化对该事务来说才是可见的。尽管这个设置可以让事务不会看到没有完成的事务，但是事务仍然有可能会得到数据库状态的不同结果——如果在事务处理数据的过程中，另外一个事务完成了对数据的修改。除了由 BDE 管理的本地事务之外，这种隔离层次对所有的事务来说都是可以使用的。
- RepeatableRead：如果事务的隔离层次是 RepeatableRead，事务将会得到关于数据库数据的一致状态。事务只能看到数据的一个印象，它不能看到后来的其他事务对数

据进行的修改，即使这些事务完成了。这是事务隔离的最高层次。

4. 获得数据库的信息

我们可以通过一些方法来获得关于数据库的信息，下面列出了它们中的一部分。

- 列举可用的数据表：

```
MyDBConnection.GetTableNames(ListBox1.Items, False);
```

- 列举数据表中的字段：

```
MyDBConnection.GetFieldNames('Employee', ListBox1.Items);
```

- 列举可用的存储过程：

```
MyDBConnection.GetProcedureNames(ListBox1.Items);
```

- 列举可用的索引：

```
SQLConnection1.GetIndexNames('Employee', ListBox1.Items);
```

- 列出存储过程的参数：

```
SQLConnection1.GetProcedureParams('GetInterestRate', List1);
```

9.3.2　关联数据集控件和数据库

上面的内容从一般的角度介绍了关于数据库连接的一些基本情况，如果你在阅读了上面的内容之后对数据库的连接仍然没有明确的概念，没有关系，虽然通过上面的内容，你可能还不了解每种数据库连接的具体方式，但是你已经建立起了关于数据库连接的很多基本概念，此时再来了解具体的数据库连接方式就比较容易了。

我们的数据集控件只有在连接到数据库或者会话（Session）的时候才能接受到数据，这是显然的。所以在使用一个数据集控件来搜集数据之前，应该先把这个数据集控件和数据库或者会话关联起来。

数据集控件代表了到指定数据库服务器的连接，而数据库实际上等同于一个 BDE 驱动程序、使用该驱动程序的特定数据库服务器和连接到数据库服务器的一系列参数。每一种数据库都可以用 TDatabase 控件来代表。我们可以把数据集控件和一个 TDatabase 控件关联起来，也可以简单地指定数据库服务器的名称，然后让 Delphi 自己生成一个隐含的数据库控件。对于大多数应用程序来说，建议用户使用直接创建的 TDatabase 控件，因为通过数据库控件我们可以更好地控制连接的建立、用户的登录以及创建和使用事务。要把一个支持 BDE 的数据集控件和一个数据库关联起来，可以使用数据集控件的 DatabaseName 属性。该属性是一个字符串，根据要使用的数据库类型的不同，包含了不同的信息。

- 如果使用的是直接创建的 TDatabase 控件，那么 DatabaseName 属性的值就等于 TDatabase 控件的 DatabaseName 属性的值。
- 如果使用的是隐含的数据库控件，并且数据库具有 BDE 别名，那么可以把数据库的 BDE 别名作为 DatabaseName 属性的值。在介绍使用数据控件的例子中，我们已经介绍了如何利用 Delphi 的附带工具来创建 BDE 别名。
- 如果要使用一个隐含数据库控件的 Paradox 或者 dBase 数据库，也可以把存储数据库的数据表的位置（文件路径）指定给 DatabaseName 属性。

会话提供了对一个应用程序中的一组数据库连接的全局化管理。当我们在应用程序中加入一个支持 BDE 的数据集控件时，该应用程序便自动创建了一个会话控件，称为 Session。当我们继续向应用程序中添加数据库控件和数据集控件时，这些数据库控件和数据集控件便自动和默认的会话关联起来。这个会话还控制着用户对受保护的 Parados 文件的访问，同时它也指定了在一个局域网中共享 Paradox 文件的目录位置。可以通过会话对象的属性、事件和方法来控制访问 Paradox 文件的数据库连接。由于该应用程序中有一个默认的会话，所以，可以通过这个默认的会话来控制应用程序中的所有的数据库连接。另外，我们也可以在设计应用程序时加入其他的会话，或者在应用程序运行时动态地创建对话。

要把我们的数据集控件和一个明确创建的会话对象相关联，则可以使用数据集控件的 SessionName 属性。如果没有在程序中创建明确的会话对象，就没有必要指定这个属性的值。不管我们使用的是默认的会话对象还是明确创建的会话对象，都可以用数据集控件的 DBSession 属性来访问和数据集控件关联的会话对象。

上面介绍的是关于数据集控件的一般情况，下面我们针对支持 BDE 的几个数据集控件来介绍如何关联和使用这些数据集控件。

在 Delphi 中，支持 BDE 的数据集控件有三类：TTable、TQuery 和 TStoredProc。下面将分别介绍它们的使用情况。

1. 使用 TTable 控件

在介绍利用数据控件来创建数据库应用程序用户界面时，我们使用的一直都是 TTable 控件，但是并没有详细介绍它的使用方法。

TTable 封装了一个指定数据库表中的全部结构和数据。它实现了 TDataSet 对象的所有基本功能和所有基于数据表的数据集的特殊功能。TTable 控件必须和一个数据库或者会话相关联才能发挥作用。例如，在上面介绍数据库应用程序用户界面时，我们通常把 TTable 控件的 DatabaseName 属性设置成 BDE 中的一个别名，然后把 TableName 属性设置成该别名代表的数据库中的某个数据表。

如果应用程序的访问以 Paradox、dBase、FoxPro 或者文本文件作为数据表，那么 BDE 会用 TableType 属性来确定数据表的类型。如果 TTable 控件代表了一个数据库服务器上的基于 SQL 的数据表，就不必使用 TableType 属性。在默认情况下，该属性是 ttDefault，此时 TTable

根据文件的扩展名来确定数据表的类型，比如.db 的文件代表 Paradox 类型的数据表。

像上面介绍的那样,在设置好了 TTable 控件的 DatabaseName、TableName 和 SessionName 属性之后，我们可以用 Active 属性或者 Open 方法打开数据表。

在打开数据表之后，如果希望控制对数据表的读写访问，可以使用 ReadOnly 属性。如果该属性为 True，那么只能从 TTable 控件中读取数据，而不能把数据提交到该数据表中。如果 TTable 控件连接的是 Paradox、dBase 或者 Foxpro 数据表时，可以通过 Exclusive 属性来确实是否让其他的应用程序也同时访问我们的数据表。如果该属性为 True，那么，当我们的程序打开该数据集时，其他程序就不能访问对应的数据表文件了，我们称之为独占模式。例如下面的语句就以独占模式打开一个数据表：

```
CustomersTable.Exclusive := True;
CustomersTable.Active := True;
```

2. 使用 TQuery 控件

在使用 TQuery 控件时也要指定一个 DatabaseName 属性，它的用法和 TTable 控件中的 DatabaseName 属性相同，我们也可以不指定 DatabaseName 属性，而是在 SQL 属性中的查询语言中来指定要查询的数据库名称。和 TTable 控件不同的是，TQuery 控件不使用 TableName 属性来指定数据表，而是使用 SQL 属性来指定一组 SQL 查询语句。在使用 TQuery 控件的时候一般采用下面的步骤。

(1) 利用 BDE 管理工具或者 SQL explorer 程序创建要访问的各个数据库的别名。

(2) 不用指定 TQuery 控件的 DatabaseName 属性，而在 SQL 语句中指定要使用的数据库名称。

(3) 在 SQL 属性中输入要执行的 SQL 语句。需要注意的是，要在每个数据表名称之前加上 BDE 别名。

(4) 在 Params 属性中指定查询的参数。

(5) 在第一次执行查询之前调用 Prepare 方法。

(6) 调用 Open 方法或者 ExecSQL 方法，执行查询并打开数据库。

例如,如果定义了两个数据库别名：WokData 和 WokCustom，那么我们可以在 SQL 属性中输入下面的查询语句：

```
SELECT Customer.CustNo, Orders.OrderNo
FROM "WokCustom:CUSTOMER"
JOIN ":WokData:ORDERS"
ON (Customer.CustNo = Orders.CustNo)
WHERE (Customer.CustNo = 1503)
```

如果在程序中使用 TDatabase 控件连接到数据库，那么可以在 TDatabase 控件中指定它

的 DatabaseName 属性，该属性必须不能和我们系统中已经能够访问的其他数据库别名相同。然后可以像利用 BDE 中的数据库别名的方法一样来查询数据库。

如果希望查询到的结果能够在数据控件中进行编辑，则应该把 TQuery 控件的 RequestLive 属性设置成“True”。在默认的情况下该属性为 False，那么此时查询结果是只读的。

当然也有办法来更新 TQuery 控件返回的只读数据集。例如，如果所有的更新都是针对一个数据表，则可以在 OnGetTableName 事件中处理需要进行的更新。但是我们更推荐使用 BDE 中的另外一个控件来处理 TQuery 控件的只读的返回结果，它就是 TUpdateSQL 控件。可以按照下面的步骤进行操作。

(1) 在窗体上或者数据模块中放置一个 TUpdateSQL 控件，然后把 TQuery 控件的 UpdateObject 属性设置成该 TUpdateSQL 控件。

(2) 在 TUpdateSQL 控件中修改 ModifySQL、InsertSQL 和 DeleteSQL 属性，它们都是执行对应更新操作的 SQL 语句。

说明：

关于使用 TUpdateSQL 等更新控件来更新数据库的内容，我们将在后面有详细的介绍。TUpdateSQL 控件的使用通常和 BDE 的缓存更新相关联。

3. 使用 TStoredProc

TStoredProc 控件代表了一个存储过程。它实现了 TDataSet 的所有基本功能，还提供了适用于存储过程类型的数据集的所有特殊功能。一般来说，可以通过下面的步骤来访问存储在一个服务器上的存储过程。

(1) 在窗体上或者数据模块上放置一个 TStoredProc 控件，并修改它的 Name 属性。

(2) 指定定义了存储过程的数据库服务器。对于 TStoredProc 来说，可以用 TDatabase 或者 BDE 别名指定该数据库服务器。

(3) 指定要执行的存储过程名称，也就是指定 StoredProcName 属性的值。

(4) 如果存储过程返回的是可以用在数据控件中的指针，那么可以在数据模块上放置一个数据源控件，并把它的 DataSet 属性设置成 TStoredProc 控件，然后我们就可以通过这个数据源控件，把得到的结果显示在数据控件中，比如 TDBGrid 控件中。

(5) 在必要的情况下提供存储过程的输入参数。

(6) 执行存储过程。对于返回指针的存储过程来说，可以使用 Open 方法或者 Active 属性。对于那些没有任何返回值或者只返回输出参数的存储过程来说，可以调用 ExecProc 方法。如果需要多次执行某个存储过程，则可以在第一次调用 ExecProc 方法之前调用 Prepare 方法。

(7) 处理存储过程的执行结果。

在上面介绍的三种数据集控件中，都多次提到了使用 BDE 的数据库别名和 TDatabase 控

件或者 TSession 控件。关于使用 BDE 数据库别名的方法在前面已经多次介绍了，这里来介绍一下 TDatabase 控件和 TSession 控件的使用方法。

在 Delphi 中，可以先用一个 TDatabase 控件来连接到数据库，然后让 TTable 等数据集控件通过该数据库控件访问数据库。所有的 TDatabase 控件都必须和一个 BDE 会话相关联，这个关联的会话由 TDatabase 控件的 SessionName 属性来指定。在数据模块中创建一个 TDatabase 控件时，SessionName 的值为 Default。需要说明的是 Default 是应用程序的默认会话。当向应用程序中加入任何 BDE 类的控件时，该会话便自动创建。当然我们在程序中可能会需要多个会话，所以也可以在窗体或者数据模块中添加 TSession 控件。

下面的窗口代码演示了如何把一个 TDatabase 控件和一个 TSession 控件相关联。

```
object Session1: TSession
  Active = True
  SessionName = 'try'
  Left = 192
  Top = 208
end
object Database1: TDatabase
  AliasName = 'DBDEMOS'
  DatabaseName = 'dbtry'
  LoginPrompt = False
  SessionName = 'try'
  Left = 104
  Top = 240
end
```

9.3.3　处理数据集中的记录

1. 记录的导航

实际上关于数据集中的导航问题在本章的第二节中已经介绍了其中的一部分。在介绍如何使用数据控件构建数据库应用程序的用户界面时，我们提到了 TDBNavigator 控件，它是一个在数据集中导航记录的控件，调用的就是数据集控件的导航方法。

在本节中，要介绍的第一个内容便是关于数据集控件记录导航的方法和属性。在 Delphi 的数据集控件中，一般都具有表 9.8 所示的几个基本的导航方法。

表 9.8　数据集控件中的导航方法

方法	功能
First	数据集控件的第一个记录
Last	数据集控件的最后一个记录

续表

方法	功能
Next	数据集控件的下一个记录
Prior	数据集控件的前一个记录
MoveBy	根据其参数向前或者向后移动一定数目的记录

调用它们非常简单，比如要到达数据集控件的第一个记录，可以使用下面的语句：

```
dataset1.First
```

Moveby 方法的说明如下：

```
function MoveBy(Distance: Integer): Integer;
```

通过该方法的参数，可以指定移动的方向和数目，比如要向前移动两个记录，可以使用下面的方法：

```
dataset1.Moveby(-2)；
```

数据集控件还有两个用来表明当前记录位置的属性：EOF 和 BOF。如果当前的记录指针位于第一个记录，那么 BOF 为 True 如果记录指针位于最后一个记录，那么 EOF 属性为 True。在程序中，可以利用这两个属性来判断记录指针是否位于数据库的开始和结尾。

在数据集控件中对记录进行导航的另外一个方法是使用书签（BookMark）。下面用一个简单的示例程序介绍如何处理数据集中的书签。

(1) 新建一个应用程序，添加一个 Data Module，然后就像我们在介绍 TDBCtrlGrid 控件的使用时一样，添加相应的数据控件和常规控件，并配置它们的数据库相关属性。

(2) 在窗口上添加一个 TListBox 控件、一个 TPanel 控件、一个 TImage 控件和两个我们以前创建的 TImage 和 TLabel 控件的组合，并进行相应的设置。

(3) 为了存储在程序中创建的书签，我们声明了下面的数据类型和变量：

```
type
  BookMarkRec=record
        BkString:String[10];
        bookmark:TBookMark;
  end;
  BkMkList=array of BookMarkRec;
BkList:BkMkList;
```

说明：

在这个例子中，读者也可以学习到如何在程序中使用动态数组。

(4) 显然我们的目的是把自己创建的各个书签存储到 ListBox 控件中，所以编写了下面一个过程，用来把 bkList 数组中的书签显示在 ListBox 控件中。

```
procedure TForm1.RefreshList;
var i:integer;
begin
    Listbox1.Clear;
    for i:=0 to high(bklist) do
        Listbox1.Items.Add(bklist[i].BkString)
end;
```

(5) **当用户单击程序中的一个图像按钮时，将在数据集控件的当前记录位置创建一个书签，程序如下所示。我们在这个过程中使用** SetLength **方法来设置动态数组的长度，用数据集控件的** GetBookmark **方法创建对当前记录位置的一个书签。在创建书签时，还利用数据集控件的** CompareBookMarks **方法验证当前要创建的书签是否和我们已经创建的书签重复。**

```
procedure TForm1.Button1Click(Sender: TObject);
var bkmk:TBookmark;
      i:integer;
      j:integer;
begin
bkmk:=dmExam.tbExam.GetBookmark;
i:=high(BkList)+1;
for j:=0 to i-1 do
      if dmExam.tbExam.CompareBookmarks(bkmk,bklist[j].bookmark)=0 then
      begin
          Showmessage('已经在这个位置创建了 BookMark。');
          exit;
      end;
setlength(bkList,i+1);
bklist[i].BkString := 'BookMark'+inttostr(i);
bklist[i].bookmark := bkmk;
listbox1.Items.Add(bklist[i].BkString)
end;
```

(6) **当用户单击程序中的另外一个按钮时，将删除列表框中的当前书签，此时调用了数据集控件的** FreeBookMark **方法，并利用** SetLength **方法释放该部分内容在** bkLisk **数组中的内存。在删除一个书签之前，调用数据集控件的** BookMarkValid **方法来检查当前少删除的书签是否是一个有效的书签，代码如下：**

```
procedure TForm1.Button2Click(Sender: TObject);
var i,j:integer;
begin
if ListBox1.SelCount = 0 then
    exit;
i:=Listbox1.ItemIndex;
```

```
if dmexam.tbExam.BookmarkValid(bklist[i].bookmark) then
    dmExam.tbExam.FreeBookmark(bklist[i].bookmark);
for j:=i to high(bklist)-1 do
begin
    bklist[j].BkString := bklist[j+1].BkString;
    bklist[j].bookmark := bkList[j+1].bookmark ;
end;
setlength(bklist,High(bklist));
RefreshList;
end;
```

(7) **当用户在程序中的列表框中选择某个书签时，该程序将跳到该书签对应的数据集控件中的记录。此时调用了数据集控件的** GotoBookMark **方法，代码如下所示：**

```
procedure TForm1.ListBox1Click(Sender: TObject);
var i:integer;
begin
if ListBox1.SelCount = 0 then
    exit;
i:=Listbox1.ItemIndex;
if dmexam.tbExam.BookmarkValid(bklist[i].bookmark) then
    dmExam.tbExam.GotoBookmark(BkList[i].bookmark);
end;
```

(8) **通过上面的过程，我们便建立了一个利用书签在数据集控件中导航的应用程序，同时也提供了对书签的管理功能。程序的运行结果如图** 9.21 **所示。**

图 9.21　**书签使用示例程序**

当我们在数据集中导航时，上面介绍的方法已经能够满足很多种情况下的需要了。但是有一种情况利用上面的这些方法就不好解决。比如，要处理一个包含有大量数据的数据库，

例如有上万条。现在希望找到其中的字段 Species No 的值为 90030 的记录，那么如果利用上面的这些方法，就需要不断地翻页，同时还得留神不要漏过了要查找的内容，这样的程序显然是不能令人满意的。要解决这个问题当然有很多办法，这里先介绍利用数据集控件的 Locate 和 Lookup 方法来处理这一类的问题。

Locate 方法的声明如下：

```
function Locate(const KeyFields: String; const KeyValues:
                Variant; Options: TLocateOptions): Boolean;
```

这个方法的功能是按照 Options 参数指定的搜索方式，在数据集控件中查找字段 KeyFields 的值等于 KeyValues 的记录。如果 Locate 方法找到了匹配的记录，那么将把数据库指针移动到该记录上，也就是说该记录将变成当前记录，并返回 True；否则，将返回 False，并不影响当前记录的位置。在这个方法中的 Options 参数具有两个可选值。

- loCaseInsensitive：Keyfields 和 KeyValue 都必须严格匹配，也就是完全相同。
- loPartialKeyKey：部分匹配，指的是要搜索的字段名称可能是数据集中的字段的名称的一部分。例如，如果把 KeyFields 参数指定为“No”，那么“Species No”和“OrderNo”都有可能匹配。

Lookup 方法的声明如下：

```
function Lookup (const KeyFields: String; const KeyValues: Variant;
                 const ResultFields: String): Variant;
```

Lookup 方法的功能是在数据集中查找字段 KeyFields 的值为 KeyVlues 的记录，并把 ResultFields 参数中指定的字段值返回。这个方法的第二个参数是 Variant 类型，所以，如果要指定多个字段进行查找，应该使用 VarArrayOf 方法把多个指定值进行转换。它和 Locate 方法的不同点是，它不会把当前记录的指针移动到找到的记录上。

为了演示这个两个方法的使用以及它们的区别，这里编写了一个简单的示例程序。在这个程序中，首先利用数据集控件的 GetFieldName 方法，获得了要操作的数据集控件的字段名称，并填充到一个下拉列表中，代码如下：

```
procedure TForm1.FormShow(Sender: TObject);
begin
    dmExam.tbExam.GetFieldNames(ComboBox1.Items);
end;
```

然后我们编写了一个利用 Locate 方法来定位的事件，代码如下：

```
procedure TForm1.Button1Click(Sender: TObject);
begin
    if (ComboBox1.Text ='') or (Edit6.Text = '') then
        exit;
```

```
    dmExam.tbExam.Locate(ComboBox1.Text,Edit6.Text,[loPartialKey]);
end;
```

此时运行该应用程序，结果将如图 9.22 所示。

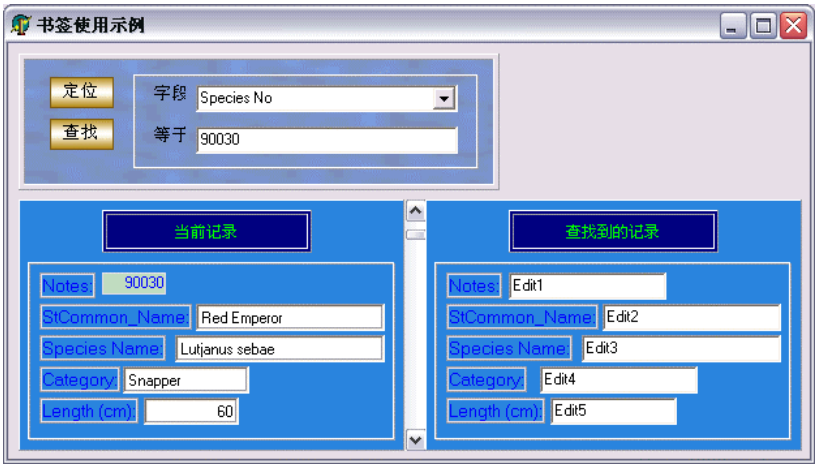

图 9.22　定位记录

从程序的运行结果上可以看出，已经找到了我们要查找的记录，并且数据集的当前记录也变成了该记录。

然后我们又编写了一个关于使用 Lookup 方法的事件，代码如下：

```
procedure TForm1.Button2Click(Sender: TObject);
var Re:variant;
     StrField:string;
begin
    if (ComboBox1.Text =”) or (Edit6.Text = ”) then
        exit;
    StrField:=’Species No;Common_Name;Species Name;Category;Length (cm)’;
    Re:=dmExam.tbExam.LookUP(ComboBox1.Text,Edit6.Text,strfield);
    if not VarIsEmpty(Re) then
    begin
        Edit1.Text := Re[0];
        Edit2.Text := Re[2];
        Edit3.Text := Re[1];
        Edit4.Text := Re[3];
        Edit5.Text := Re[4];
    end;
end;
```

这段程序首先利用 Lookup 方法查找符合条件的记录，然后把返回的结果显示在一些文

本控件中，运行结果如图 9.23 所示。

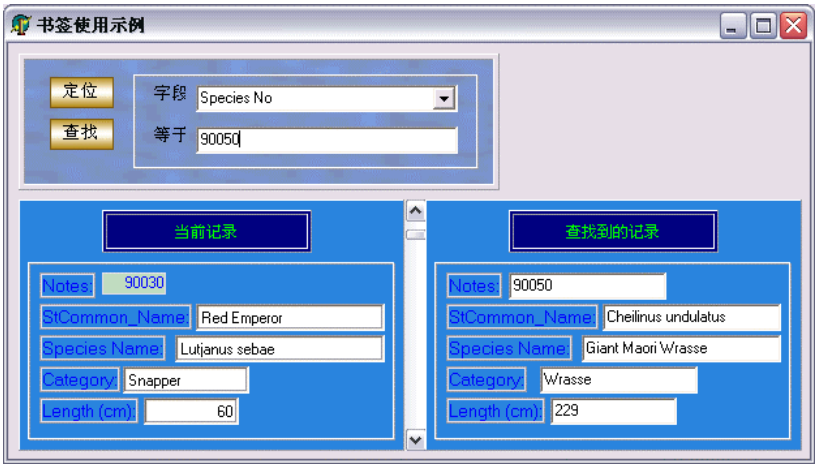

图 9.23　Lookup 方法示例

从程序的运行结果上可以看出，我们找到了需要的结果，并显示在右边的文本框中，但是当前记录并没有变化，这可以从左边的 DBCtrlGrid 控件上看出来。

2. 记录的过滤

在处理数据库时，特别是在处理具有成千上万条记录的数据库时，可能不需要显示所有的数据。那么可以利用数据集控件中关于记录过滤的一些功能来过滤掉不符合条件的记录。

此时可以使用数据集控件的 Filter 和 Filtered 两个属性。首先，把一定的限制条件输入到 Filter 属性中，然后把 Filtered 属性设置成“ True ”。这样数据集控件便对数据进行了过滤。例如，可以使用下面的语句对数据集进行过滤：

```
Salary>30000
```

也可以在程序运行的时候设置这两个属性。例如下面的语句：

```
table1.Filter:='State=True';
table1.Filtered:=True;
```

在设置数据集控件的 Filter 属性时，可以使用下面的符号：

<、>、>=、<=、=、<>、AND、NOT、OR、+、-、*、/

那么利用这些符号，就可以创建各种过滤条件的组合来过滤数据集中的数据了。例如在图 9.33 所示的应用程序中，对 employee.db 数据表进行了“ EmpNo >30 and Salary>30000 ”的过滤设置。

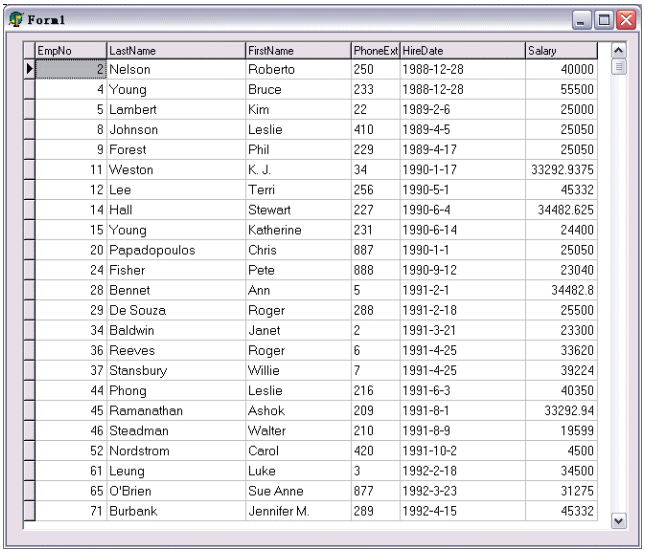

图 9.33　使用 Filter 属性过滤数据

上面的方法的优点是简单方便，而且可以在程序运行时根据用户的输入来确定过滤条件。只要向用户提供一些编辑条件的工具，然后把用户定义的过滤条件字符串赋给 Filter 属性，然后再把 Filtered 属性设置成“True”就可以了。但是它的缺点是能定义的条件是有限而且简单的。如果希望进行复杂的过滤，可以使用数据集控件的 OnFilterRecord 事件。这个事件发生在数据集控件收到一个记录的时候。所以可以在这个事件的处理程序中对是否接受这个数据进行处理。例如，可以使用下面的代码：

```
procedure TDmGrid.tbExampleFilterRecord(DataSet: TDataSet;
  var Accept: Boolean);
begin
Accept:=(Dataset['Salary']>30000) and (Dataset['EmpNo']>30);
end;
```

利用上面的代码同样可以实现利用 Filter 属性所实现的过滤。这里只是举个简单的例子，从这个例子中可以看到，通过 DataSet 参数，可以访问当前数据集控件中当前记录的所有字段。所以，可以完成任何对这些字段的值的过滤。

需要说明的是，对数据集控件进行了过滤之后，就不能利用前面的介绍的 First 等方法来在数据集中导航了，因为这时导航的仍然是数据集对应的数据表中的实际记录。此时应该使用下面的方法。

- FindFirst：**第一个记录。**
- FindLast：**最后一个记录。**
- FindNext：**下一个记录。**
- FindPrior：**前一个记录。**

如果没有对数据集控件进行过滤，那么这些方法和上面介绍的对应方法的作用是相同的。

3. 修改记录

关于修改记录的内容在上面的一个示例中已经涉及到了，比如 Edit、Post 等方法。在这里要详细介绍它们的使用方法。

在一个数据集控件中修改记录一般要有以下步骤。

(1) 使用 Edit 方法使数据集控件处于 dsEdit 状态。当数据集控件刚进入 dsEdit 状态时会发生 OnBeforeEdit 事件；当退出该状态时会发生 OnAfterEdit 事件。可以在程序中利用这两个事件进行特殊的处理，比如向用户提示是否确认要修改数据集控件中的数据。

(2) 插入或者追加记录，然后对记录中的各个字段进行赋值。比如，可以使用下面的代码：

```
table1.insert;
table1.fieldbyname('Name').asString:='John';
```

(3) 提交对记录的修改。此时可以使用数据集控件的 Post 方法。如果希望取消对数据的修改，那么可以不使用 Post 方法，而是使用 Cancel 方法，该方法将取消前面对当前记录的各个字段的修改。

如果要删除当前记录，可以使用 Delete 方法。例如下面的语句将删除数据集控件中的当前记录。

```
Table1.Delete;
```

也可以利用数据集的一些方法，把整个记录作为处理对象进行赋值或者修改。这样的方法包括：AppendRecord、InsertRecord 和 SetFields。它们的参数都是一个数组。例如，可以使用下面的方法在数据集控件中修改当前记录的值：

```
table1.edit;
table1.Setfields([nil,nil,nil,345,765]);
Table1.Post;
```

在这个示例程序中，Nil 参数表示将保留原来的值。

9.3.4　使用索引

索引决定了记录在数据表中的显示顺序。在默认情况下，记录将根据主索引或者默认的索引来进行升序排列。这是系统的默认操作，不要我们的应用程序进行任何干预。如果需要使用一个不同的索引，则必须指定另外的一个索引或者一系列字段进行索引。使用索引可以使数据在数据表中按照不同的顺序进行显示。在基于 SQL 的数据表中，索引是由 Order By

语句产生的。在其他的数据表中，是由显示记录的数据访问机制决定的。所以下面的内容中，将主要针对数据表类型的数据集控件来介绍如何使用索引，典型的数据表类型的数据集控件就是 BDE 的 TTable 控件。

1. 索引使用概述

对于所有数据表类型的数据集，我们的应用程序可以读取关于服务器定义的索引信息，此时需要使用 GetIndexName 方法，该方法的声明如下：

```
procedure GetIndexNames(List: TStrings);
```

在这个方法中，通过 List 参数获得服务器定义的索引情况。需要说明的是，对于 Paradox 的数据表，主索引是没有名称的，因此也不会被这个方法所获得。也就是说，我们无法获得主索引的名称。但是仍然可以在程序中把当前数据表的索引指定为主索引，只要把 IndexName 属性设置成空就可以了。

要获得当前索引的字段信息，可以使用 IndexFieldCount 和 IndexFields 属性，下面的代码演示了如何使用它们：

```
var
I: Integer;
ListOfIndexFields: array[0 to 20} of string;
begin
with CustomersTable do
begin
for I := 0 to IndexFieldCount - 1 do
ListOfIndexFields[I] := IndexFields[I].FieldName;
end;
end;
```

如果要指定一个索引，可以在 IndexName 属性中指定。在设计期，可以使用属性编辑器旁边的按钮来选择可用的索引；在运行期，可以使用上面的 GetIndexNames 方法获得可用索引的列表，然后把其中的一个索引指定给数据表类型的数据集控件。

也可以利用 IndexFieldNames 属性来创建自己的索引。创建的方法也很简单，只要把上面获得的 IndexFields 中的字段名排列起来就可以了。注意，各个字段名之间要用分号隔开。

2. 利用索引搜索记录

在前面的内容中，介绍了可以通过 Locate 方法和 Lookup 方法来搜索记录。在这里要介绍的是利用索引来搜索记录。通过索引进行搜索，在性能上要比 Locate 和 Lookup 方法改进很多。

在利用索引进行搜索时，经常会使用到表 9.9 中的方法。

表 9.9　基于索引的搜索方法

方法	说明
EditKey	保留当前搜索字段的缓存，并把数据集置于 dsSetKey 状态
FindKey	是 SetKey 方法和 GotoKey 方法的组合
FindNearest	是 SetKey 方法和 GotoNearest 方法的组合
GotoKey	查找符合标准的第一个记录，并把指针移动到该记录上
GotoNearest	搜索基于字符串的字段，并把光标移动到第一个最相近的记录上
SetKey	清除当前搜索字段的缓存，把数据表置于 dsSetKey 状态

在利用这些方法进行搜索的时候，一般要遵循下面的顺序：

(1) 在数据集控件中指定索引，使用 IndexName 属性。该索引既是数据集排序的顺序，也是用来搜索的索引。

(2) 打开数据集。

(3) 调用 SetKey 方法，使数据集处于 dsSetKey 状态。

(4) 指定索引中要使用的字段值，这里一般要使用 Fields 属性。需要说明的是，只能指定参与索引的字段，否则将会引发一个异常。

(5) 调用 GotoKey 方法或者 GotoNearest 方法进行搜索。

例如，可以使用下面的代码进行对指定字段的搜索：

```
procedure TForm1.Button1Click(Sender: TObject);
begin
Table1.SetKey;
Table1.FieldByName(ComboBox1.Text).AsString := Edit1.Text;
if not Table1.GotoKey then
ShowMessage('Record not found');
end;
```

在图 9.34 中显示了搜索的结果，可以看出，当前记录移动到了搜索到的第一个结果上。

如果所使用的索引中包含了不止一个字段，那么可以对其中的部分字段进行搜索，此时要使用到 KeyFieldCount 属性。一般来说，可以结合 KeyFieldCount 属性和 EditKey 方法进行连续的搜索。比如，对于上面的程序，要搜索的是 Country 字段等于 U.S.A.的记录。如果希望能在搜索结果中找到一个 City 字段为 Chicago 的记录，此时要保证 City 字段也出现在索引字段中，那么可以使用下面的代码进行搜索：

```
procedure TForm1.Button2Click(Sender: TObject);
begin
Table1.EditKey;
table1.KeyFieldCount := 2;
Table1['City'] := Edit2.Text;
Table1.GotoNearest;
end;
```

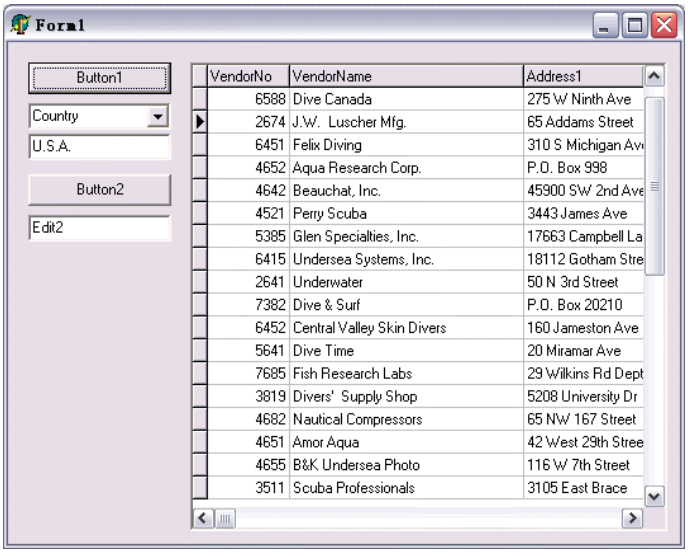

图 9.34　**利用索引进行搜索**

在程序中，如果单击 Button2 **按钮，将会把记录移动到** Edit2 **控件中指定的文本记录上。如果在修改了** Edit2 **的文本之后，再次单击该按钮，将会继续进行搜索。**

3. **设置数据集的范围**

我们知道，可以利用数据集控件的 Filter **属性来进行数据集的过滤，也就是只显示部分记录。利用索引，也可以实现对数据集范围的限制。它们是有区别的。利用索引设置的是范围，利用** Filter **属性进行的是过滤。范围是处于指定的值之间连续的经过索引的记录；而使用** Filter **属性，则是把不符合条件的内容过滤，所以它们很可能是不连续的。通常来说，使用过滤器具有更大的灵活性，但是如果数据库非常庞大，并且已经指定了相应的索引，此时使用索引将更有效。**

可以利用数据集控件的 SetRangeStart **和** SetRangeEnd **方法来设置范围的开始和结束，用** ApplyRange **方法把设置的范围应用到数据集上，或者用** CancelRange **方法取消设定的范围。例如在下面的程序中，对** VendorNo **字段进行了索引，并设置了范围。**

```
procedure TForm1.Button3Click(Sender: TObject);
begin
table1.SetRangeStart;
table1.FieldByName('VendorNo').AsString := Edit1.Text;
table1.SetRangeEnd;
table1.FieldByName('VendorNo').AsString := Edit2.Text;
table1.ApplyRange;
end;
```

程序的运行结果如图 9.35 所示。

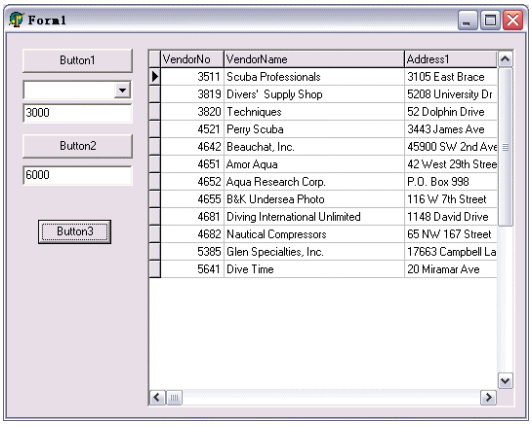

图 9.35　设置数据集控件的范围

9.3.5　处理数据集控件中的字段

字段在 Delphi 中也是一个对象，称之为 TField。同其他对象一样，它也具有自己的属性、方法和事件。在本节的内容中，将介绍如何处理数据集中的字段对象，同时也介绍如何利用它们的方法和属性来改变其显示和编辑的形式。

1. 动态字段和固定字段

在 Delphi 中，字段又分成固定字段和动态字段。关于动态字段的问题，这里就不再介绍了，因为在上面的内容中，我们几乎一直在使用动态字段。下面就来介绍关于使用固定字段的问题。

说明：

在 Delphi 中，固定字段和动态字段是互斥的，也就是说，如果在数据集中使用了固定字段，那么程序中将不再显示动态字段。如果要恢复到字段状态，需要把所有的固定字段删除。

在数据模块中加入一个 TDataSet 控件，比如 TTable 控件。然后像前面一样指定它的数据库名称和数据表名称。然后双击该数据集控件，此时会显示一个如图 9.36 所示的对话框，我们称之为字段编辑器。

在这个对话框中右击鼠标，此时会显示一个快捷菜单，这里要使用其中的两项：第一项和第三项。如果选择第三项 Add All Fields，此时会把数据库的数据表中所包含的所有字段都添加到固定字段中；如果选择菜单中的第一项 Add Fields，此时会显示如图 9.37 所示的添加字段对话框。

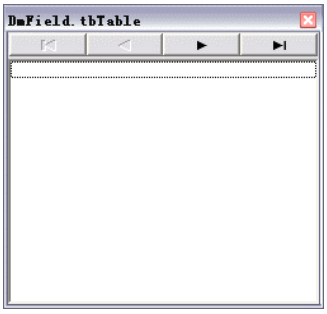

图 9.36 字段编辑器

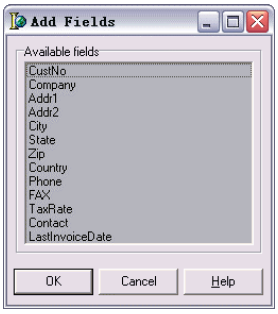

图 9.37 添加字段对话框

在这个对话框中显示了当前数据集控件中的所有字段。可以选择其中的所有内容，也可以选择全部内容，然后单击 OK 按钮，便在数据集控件中添加了相应的固定字段。比如在这里选择了如图 9.38 所示的固定字段。

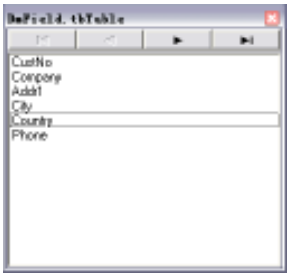

图 9.38 我们的示例程序中添加的固定字段

此时如果在窗体上放置一个数据控件，比如 TDBGrid，然后在数据模块中放置一个 TDataSource 控件，进行一定的关联后，把 TTable 控件的 Active 属性设置成“ True ”，那么此时的窗口将如图 9.39 所示。

如果在字段编辑器中调整各个字段的顺序，那么 DBGrid 控件中的显示顺序也会相应改变。所以，利用固定字段也是调整网格中显示的字段顺序的重要工具。

CustNo	Company	Addr1	City	Country	Phone
1221	Kauai Dive Shoppe	4-976 Sugarloaf Hwy	Kapaa Kauai	US	808-555-0269
1231	Unisco	PO Box Z-547	Freeport	Bahamas	809-555-3915
1351	Sight Diver	1 Neptune Lane	Kato Paphos	Cyprus	357-6-876708
1354	Cayman Divers World Unlimited	PO Box 541	Grand Cayman	British West Indies	011-5-697044
1356	Tom Sawyer Diving Centre	632-1 Third Frydenhoj	Christiansted	US Virgin Islands	504-798-3022
1380	Blue Jack Aqua Center	23-738 Paddington Lane	Waipahu	US	401-609-7623
1384	VIP Divers Club	32 Main St.	Christiansted	US Virgin Islands	809-453-5976
1510	Ocean Paradise	PO Box 8745	Kailua-Kona	US	808-555-8231
1513	Fantastique Aquatica	Z32 999 #12A-77 A.A.	Bogota	Columbia	057-1-773434
1551	Marmot Divers Club	872 Queen St.	Kitchener	Canada	416-698-0399
1560	The Depth Charge	15243 Underwater Fwy.	Marathon	US	800-555-3798
1563	Blue Sports	203 12th Ave. Box 746	Giribaldi	US	610-772-6704
1624	Makai SCUBA Club	PO Box 8534	Kailua-Kona	US	317-649-9098
1645	Action Club	PO Box 5451-F	Sarasota	US	813-870-0239
1651	Jamaica SCUBA Centre	PO Box 68	Negril	West Indies	011-3-697043
1680	Island Finders	6133 1/3 Stone Avenue	St Simons Isle	US	713-423-5675
1984	Adventure Undersea	PO Box 744	Belize City	Belize	011-34-09054
2118	Blue Sports Club	63365 Nez Perce Street	Largo	US	612-897-0342
2135	Frank's Divers Supply	1455 North 44th St.	Eugene	US	503-555-2778
2156	Davy Jones' Locker	246 South 16th Place	Vancouver	Canada	803-509-0112
2163	SCUBA Heaven	PO Box Q-8874	Nassau	Bahamas	011-32-09485
2165	Shangri-La Sports Center	PO Box D-5495	Freeport	Bahamas	011-32-08574
2315	Divers of Corfu, Inc.	Marmoset Place 54	Ayios Matthaios	Greece	30-661-88364
2354	Kirk Enterprises	42 Aqua Lane	Houston	US	713-556-6437
2975	George Bean & Co.	#73 King Salmon Way	Lugoff	US	803-438-2771

图 9.39 设置了固定字段后的网格显示

2. 格式化字段的编辑和显示格式

在字段编辑器中选中一个固定字段，此时属性编辑器中的内容也相应地发生了变化，显示了固定字段的属性或者事件。利用这些属性和事件，可以实现对字段编辑格式和显示格式的设置。

对于日期、时间、字符串的字段来说，它们有一个 EditMask 属性，单击该属性旁边的按钮，此时会显示如图 9.40 所示的 Input Mask Editor 对话框。

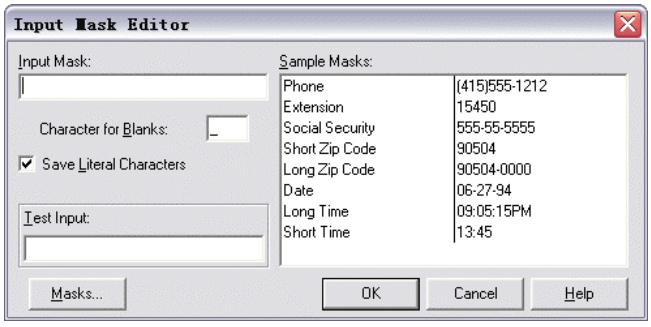

图 9.40　Input Mask Editor 对话框

在这个对话框中列出了很多预先定义好的格式，我们可以在里面选择自己需要的格式，也可以在 Input Mask 文本框中自己编辑对应的格式。

对于数字等字段的属性来说，它们具有两个和显示编辑相关的属性，一个是 EditFormat 属性，该属性决定了当用户在数据控件中进入编辑状态的时候的输入格式；一个是 DisplayFormat 属性，通过该属性，可以指定该字段的显示格式。例如对于上面的例子，我们对 CustNo 字段的这两个属性进行了设置：

```
EditFormat:='00000_00000';
DisplayFormat:='0000000.000';
DisplayWidth:=45;
```

在上面的代码中，还使用了 DisplayWidth 属性，该属性决定了在网格中的单元格宽度，是以字符为单位的。此时字段的显示如图 9.41 所示。

还可以利用 DefaultExpress 属性来指定字段的默认值。

3. 访问字段的值

如果在程序中需要使用当前记录某个字段的值，可以使用字段的一些关于类型转换的属性。比如，如果要把一个数字类型的字段赋值给一个字符串，可以使用下面的代码：

```
edit1.text:=Table1.Fields[0].asString;
```

在 Delphi 中，这样的转换属性有很多，它们以及与之相关联的字段类型见表 9.10。

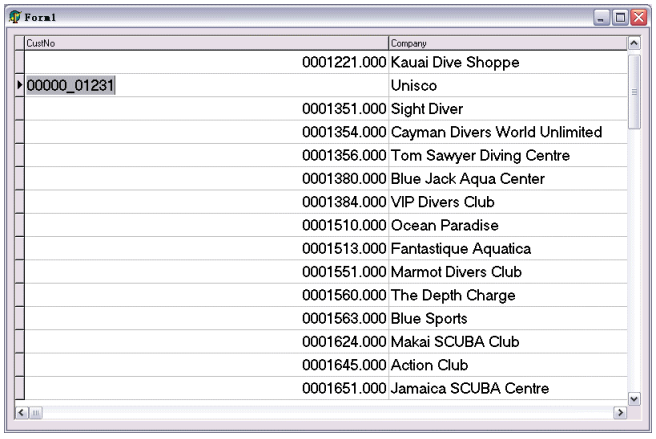

图 9.41　格式化的显示和编辑

表 9.10　字段类型和类型转换

	AsVariant	AsString	AsInteger	AsFloat AsCurrency AsBCD	AsDateTime AsSQLTimeStamp	AsBoolean
TStringField	yes	NA	yes	yes	yes	yes
TWideStringField	yes	yes	yes	yes	yes	yes
TIntegerField	yes	yes	NA	yes		
TSmallIntField	yes	yes	yes	yes		
TWordField	yes	yes	yes	yes		
TLargeintField	yes	yes	yes	yes		
TFloatField	yes	yes	yes	yes		
TCurrencyField	yes	yes	yes	yes		
TBCDField	yes	yes	yes	yes		
TFMTBCDField	yes	yes	yes	yes		
TDateTimeField	yes	yes	yes	yes		
TDateField	yes	yes	yes	yes		
TTimeField	yes	yes	yes	yes		
TSQLTimeStampField	yes	yes	yes	yes		
TBooleanField	yes	yes				
TBytesField	yes	yes				
TVarBytesField	yes	yes				
TBlobField	yes	yes				
TMemoField	yes	yes				
TGraphicField	yes	yes				
TVariantField	NA	yes	yes	yes	yes	yes
TAggregateField	yes	yes				

在了解了字段的值的类型转换之后，可以使用三种方式来访问数据集控件中的字段值：

```
Customers.FieldValues['CustNo'] := Edit2.Text;
Edit1.Text := CustTable.Fields[6].AsString;
Customers.FieldByName('CustNo').AsString := Edit2.Text;
```

第一种方法使用的是 FieldValues 属性，该属性的每个子值都是 Variant 类型的，所以可以把内容合理的任何类型的值赋给数据集控件的字段，程序将自己进行转换。

第二种方法使用的是 Fields 属性，这是一个数组，可以通过指定数组索引的方式来对字段进行操作。

第三种方法使用的是 FieldByName 属性，如果在程序中无法确切地知道某个字段的索引，那么使用该方法将是很好的选择。

4. 处理特殊的固定字段

在 Delphi 中，还提供了一些特殊的固定字段。在上面介绍添加固定字段的时候，所使用的是数据集所关联的数据表中的字段。也可以添加一些特殊的字段。在字段编辑器中，右击鼠标从弹出的快捷菜单中选择第二项 New Field，此时会显示如图 9.42 所示的对话框。

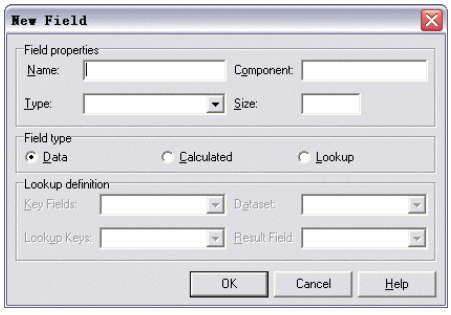

图 9.42　添加特殊字段的对话框

在这个对话框中，可以加入几种特殊的字段。其中最简单的是数据字段。可以利用下面的过程来加入一个数据字段。

(1) 在数据模块中把数据集控件的 Active 属性设置成“ False ”，并双击该数据集控件，比如 TTable 控件，显示出字段编辑器。然后把需要用来创建固定数据字段的对应字段删除，比如在我们的例子中，删除的是 CustNo 字段。最后显示如图 9.42 所示的对话框。

(2) 在 Field Properties 中输入数据字段的相关信息，注意，这里的名称必须和要使用的字段名称相同。比如这里输入的是 CustNo，然后在 Type 中指定新的字段类型，我们在这里选择了 Currency。如果需要，在 Size 框中指定该字段的长度。

(3) 在确认 FieldType 中选择的是 Data 之后，单击 OK 按钮，此时便在数据集中创建了一个新的字段。此时再把数据集控件的 Active 属性设置成“ True ”，此时数据控件将如图 9.43

所示，注意其中 CustNo 字段的变化。

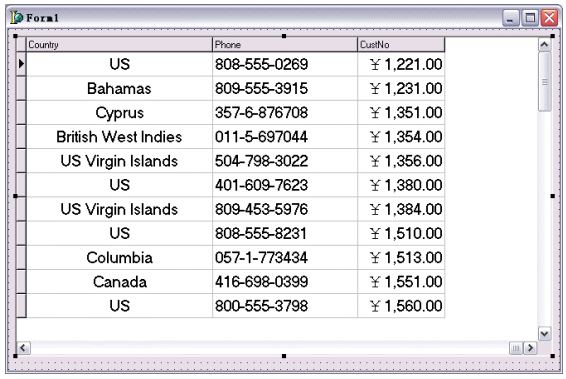

图 9.43　数据集中的新字段

还可以在图 9.42 所示的对话框中添加计算字段。添加计算字段的过程除了不需要删除一个现存的字段之外，和添加数据字段的过程是一样的。例如这里加入了一个名为 Address 的计算字段。

在添加了这个计算字段后，需要处理数据集控件的 OnCalFields 事件来处理这个计算字段。比如在这里使用了下面的代码：

```
procedure TDmField.tbTableCalcFields(DataSet: TDataSet);
begin
tbTableAddress.Value := tbTableCountry.Value +
                        '-' +tbTableCity.Value +'-'+tbTableAddr1.Value;
end;
```

程序的运行结果如图 9.44 所示，从结果中可以看出新添加的字段值是其他三个字段值的和。

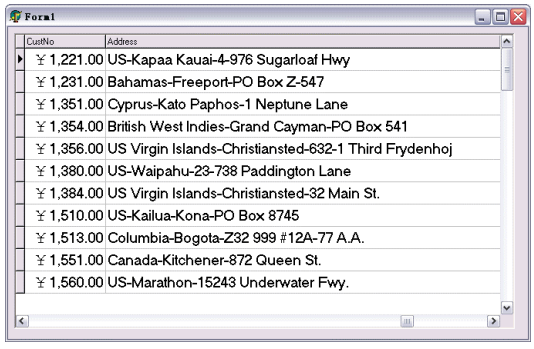

图 9.44　计算字段的值

还可以在图 9.42 所示的对话框中添加查找字段，这个字段类型在 Delphi 中称为 Lookup 字段。这种类型的字段的创建要稍微复杂一些。下面来详细的介绍如何在数据集控件中创建查找字段。

(1) 在数据模块上放置一个数据集控件，比如一个 TTable，这个控件将在查找字段中使用。然后设置该数据集控件的相关数据库属性，并把控件的 Active 属性设置成“True”。

(2) 双击我们要创建查找字段数据集控件，这里是 tbTable 控件，调出字段编辑器，然后利用右键快捷菜单调出图 9.42 所示的对话框。

(3) 指定要添加的字段的名称、数据类型和长度。然后在 Field type 中选择 Lookup，此时下面的几个下拉列表也可以使用了。

(4) 首先选择 Key Fields 中的字段，这个下拉列表中列出了 tbTable 数据集控件的各个字段。选择其中的一个，比如，选择 Phone，然后在其右边的下拉列表中选择要查找的数据集，这里是 Table1。此时下面的两个下拉列表也可以使用了。然后选择 Result Field 中的一个字段 FirstName 和 Lookup Keys 中的一个字段 PhoneExt，这两个下拉列表中显示的都是要查询的数据集的字段。

(5) 单击 OK 按钮，便为 tbTable 控件添加了一个新的固定字段。这里各个字段的意思是：从数据集 Table1 中查找 Lookup Keys 中的字段，把该字段对应的值赋给新建的查找字段，并把数据集 Table1 中的同一记录中的 Result Field 字段中的值赋给 tbTable 数据集中的 Key Fields 字段。

程序的运行结果如图 9.45 所示，从图中可以看出，当我们选择了 Lookup 字段中的一个值的时候，Phone 中的值也相应的发生了变化，并且值就是第二个数据集的 FirstName 中的值。

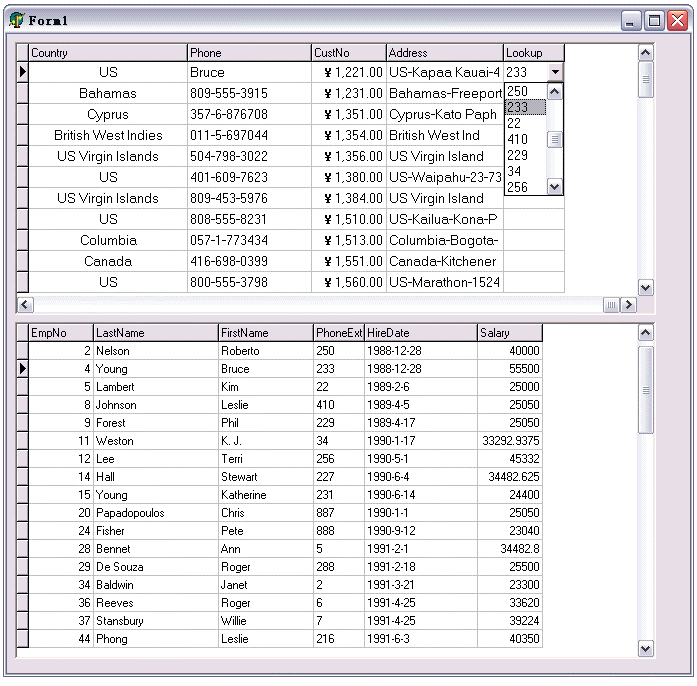

图 9.45　使用查找字段

9.4 主从式关系数据库的使用

主从式关系数据库也是数据库应用程序中应用十分广泛的类型。在主从式关系数据库中，有一个数据表作为主表，可以有几个数据表作为从表。当主表的当前记录发生变化时，所有从表中的内容都会发生相应的变化。

在 Delphi 中，如果要建立主从式关系数据库应用程序，可以使用 Table 类型的控件或者使用 Table 类型的控件和 Query 类型的控件组合。

9.4.1 使用 Table 类型的数据集控件

下面通过一个简单的例子来说明如何利用两个 TTable 建立主从式关系数据库应用程序。在 Delphi 中，TTable 控件具有两个和建立主从式关系数据库应用程序相关的属性：MasterSource 和 MasterFields，通过设置它们可以建立起数据集控件之间的主从关系。

说明：

几乎所有的 Table 类型的数据集控件都支持建立主从式关系数据库应用程序。在设计主从式关系数据库应用程序时，可以尽量地选择使用 Table 类型的数据集控件。

(1) 新建一个应用程序，然后建立一个数据模块。在数据模块上放置两个 TTable 控件，然后把它们的属性分别设置成要使用的两个数据表。

- 第一个 Table 的 Name 属性：tbMaster。
- 第一个 Table 的 DatabaseName 属性：DBDEMOS。
- 第一个 Table 的 TableName 属性：Customer。
- 第二个 Table 的 Name 属性：tbDetail。
- 第二个 Table 的 DatabaseName 属性：DBDEMOS。
- 第二个 Table 的 TableName 属性：Orders。
- 第二个 Table 的 IndexName 属性：ByCustNo。

说明：

如果没有对应的索引字段，请用数据库表编辑工具设置对应的索引字段。

(2) 在数据模块上放置两个 TDataSource 控件，然后设置它们的属性。

- 第一个 DataSource 的 Name 属性：dsMaster。
- 第一个 DataSource 的 DataSet 属性：tbMaster。
- 第二个 DataSource 的 Name 属性：dsDetail。
- 第二个 DataSource 的 DataSet 属性：tbDetail。

(3) 把数据模块的名称属性改为“DmMD”，并保存该数据模块，这里我们使用的是默认的名称 Unit2。

(4) 切换到用户界面窗口，也就是 Form1，在 Uses 语句中加入 Unit2。

(5) 在窗口上放置两个 TDBGrid 控件，分别把上面的两个 TDataSource 控件设置给这两个 TDBGrid 控件的 DataSource 属性。

(6) 切换到数据模块窗口，把 tbDetail 控件的 MasterSource 设置成“dsMaster”，并把 tbMaster 的 Active 属性设置成“True”。然后在 tbDetail 的属性编辑器中，选择 MasterFields 属性旁边的按钮，此时会显示如图 9.46 所示的对话框。

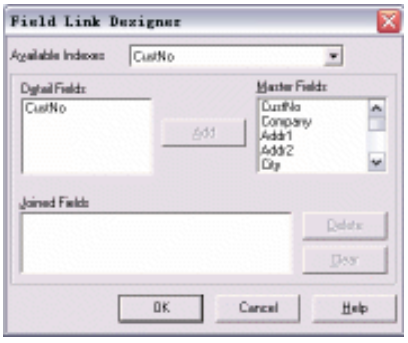

图 9.46　Field Link Designer 对话框

(7) 在这个对话框中显示了把两个数据集关联起来的可用字段。选中左边的 CustNo 和右边的 CustNo，并单击中间的 Add 按钮，此时便在 Joined Fields 框中出现了一个 CustNo→CustNo 的连接。

(8) 单击对话框中的 OK 按钮，便完成了两个字段之间的关联。

(9) 在数据模块窗口中，把两个 TTable 控件的 Active 属性都设置成“True”。

(10) 保存应用程序，并运行它，此时的程序将如图 9.47 所示。选择上面网格中的一个记录，你会注意到在下面的表格中显示了和该记录相对应的记录。观察它们的 CustNo 字段的值，它们是相同的。

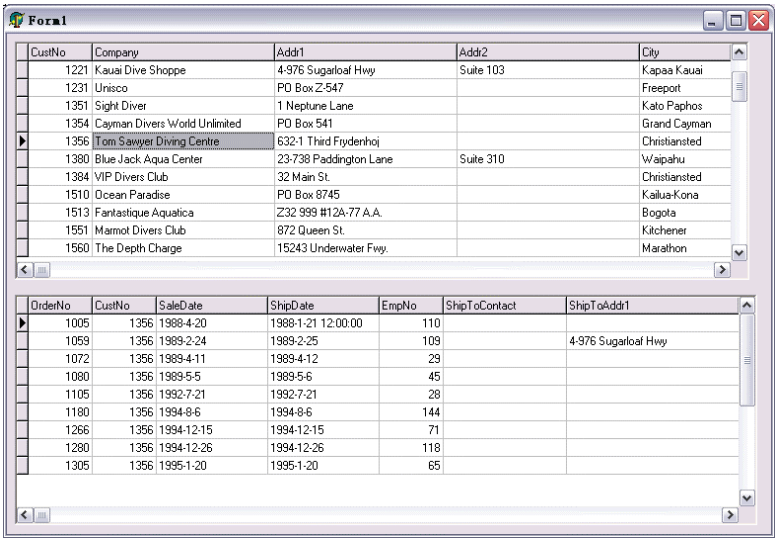

图 9.47　一个简单的主从式关系数据库应用程序

说明：

在这里使用了两个 TTable 控件，事实上可以使用很多这样的控件。

9.4.2 使用 Qurey 类型的数据集控件

如果使用一个 Query 类型的数据集控件来作为主从式关系数据库应用程序的从数据集，则必须在查询语句中使用参数。这些参数引用了主数据集中当前字段的值。由于主数据集控件上的数据在运行时是会变化的，所以必须在主数据集变化之后，重新设置从数据集的参数。通过 Query 类型数据集的 DataSource 属性，可以很好地解决这个问题。下面将通过一个例子，利用 Query 类型的数据集控件建立等同于上面的 Table 类型数据集控件的主从式关系数据库应用程序。

(1) 新建一个应用程序，然后在窗体上放置一个 Table 控件，然后设置它的属性。

- name 属性：tbMaster。
- DatabaseName 属性：DBDEMOS。
- TableName 属性：Customer。

(2) 在窗体上放置一个 TQuery 控件，然后设置它的属性。

- Name 属性：qrDetail。
- SQL 属性：

```
SELECT *
FROM Orders
WHERE CustNo = :CustNo
```

(3) 在属性编辑器中找到 qrDetail 控件的 Params 属性，单击它旁边的按钮，此时会显示一个参数编辑对话框，如图 9.48 所示。

(4) 在这个对话框中建立一个名为 CustNo 的参数，注意这个参数必须和 tbMaster 控件中的索引字段相同。在默认的情况下，DBDEMOS 中的 Customer 数据表的索引字段是 CustNO。

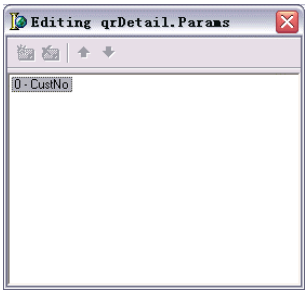

图 9.48 参数编辑对话框

(5) 在窗口上放置两个 TDataSource 控件，然后设置它们的属性。

- 第一个 DataSource 的 Name 属性：dsMaster。
- 第一个 DataSource 的 DataSet 属性：tbMaster。
- 第二个 DataSource 的 Name 属性：dsDetail。
- 第二个 DataSource 的 DataSet 属性：qrDetail。

(6) 选中窗口中的 qrDetail 控件，然后把它的 DataSource 控件设置成 dsMaster。

(7) 在窗口上放置两个 TDBGrid 控件，然后设置它们的属性，并把它们的 DataSource 属

性分别设置成上面的两个 TDataSource 控件。

(8) 把 tbMaster 和 qrDetail 控件的 Acitve 属性设置成“ True ”。

(9) 保存应用程序，并运行它，结果如图 9.49 所示。注意观察，该程序的功能和上面使用两个 Table 类型数据集控件时创建的程序是一样的。

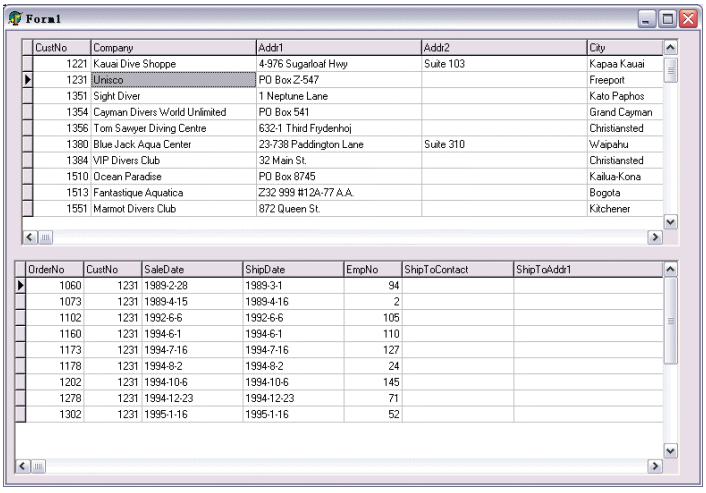

图 9.49　使用 Query 类型的数据集控件建立的主从式关系数据库应用程序

9.5　多层数据库应用程序

9.5.1　多层数据库应用程序概述

1. 多层数据库应用程序的结构

在前面的过程中，介绍的基本上都是两层的数据库应用程序。所谓两层的数据库应用程序，就是应用程序和数据库本身都是位于同一台计算机上的数据库程序。在本节中，将介绍如何来建立多层的数据库应用程序。一个多层的客户端/服务器端应用程序可以分成逻辑单元和调用层，它们运行在相互关联的不同计算机上。多层的应用程序通过一个局域网甚至是 Internet 来共享数据和相互通信。多层数据库应用程序的最简单形式，也就是我们常说的三层模式，通常包括以下三个部分。

- 客户端应用程序：提供了用户计算机上的用户界面。
- 应用程序服务器：在一个所有客户端都能访问的网络的中心计算机上，为客户提供数据。
- 远程数据库服务器：提供了关系式数据库管理系统。

在这种三层的模式中，应用程序服务器负责管理客户端和远程数据库服务器之间的数据流。通常在设计这样的多层数据库应用程序时，需要设计应用程序服务器和客户端程序，如果需要，当然也可以建立自己的数据库后端程序，但是由于远程数据库管理系统已经非常完善，所以通常不需要这样的工作。

在更为复杂的多层应用程序中，通常会在客户和远程数据库服务器之间设计一些附加的服务。例如,有时候需要一个安全服务来处理 Internet 事务，或者一个桥服务来处理不同平台间的数据共享，等等。

多层数据库模式打破了原来的数据库应用程序模式，把原来的数据库应用程序分成几个部分，客户端主要是用来处理数据的显示以及和用户的交互，在理想情况下，它不知道后台的数据是如何存储和维护的。应用程序服务器，也就是中层，负责处理来自多个用户的要求和更新，它主要用来处理数据集定义的所有细节以及和数据库服务器交互的问题。

和原来的单层数据库应用程序相比，多层数据库应用程序具有以下优势。

- 在一个中层中封装商务逻辑。不同的客户应用程序都访问相同的中层，这使得我们能够避免在每个客户应用程序中复制自己的商务规则。
- 瘦客户端。通过利用中层的处理能力，客户端应用程序可以写得很小，而且它们更容易发布，因为它们不再去关注数据库连接软件的安装、配置和维护。
- 分布式数据处理。因为负荷的平衡和冗余系统的存在，把一个应用程序的工作分布在几个机器上可以改进应用程序的性能。比如，在一个服务器出现问题时，另一台服务器可以迅速接管。
- 增加了系统的安全性。通过多层的数据库应用程序，可以把重要的数据访问功能隔离出来。这在安全性上提供了更大的灵活性和可配置性。中层可以限制重要资料的入口，使得我们可以更容易地控制用户对数据的访问。如果使用的是 HTTP、CORBA 或者 COM+，那么还可以利用它们提供的安全模式。

可以利用 Delphi 中 DataSnap 和 DataAccess 选项卡上的控件来建立多层的数据库应用程序。此时还需要使用一个远程的数据模块。它们都是基于 Provider 控件的功能来把数据封装成可以传输的数据包，并处理得到的用于更新的数据包。

2. 多层数据库应用程序的工作过程

在创建多层数据库应用程序之前，我们来介绍一下在一个三层应用程序中事件的发生顺序，这对理解和建立多层数据库应用程序是很有帮助的。

(1) 某个用户启动了客户端应用程序。该客户端连接到指定的应用程序服务器上。如果应用程序服务器没有启动，连接将自动启动应用程序服务器。客户端从应用程序服务器上获得一个 IAPPServer 接口。

(2) 客户端从应用程序服务器上获取数据。客户端可能会在一次请求中获取全部数据，也可能通过会话分批次获得数据。

(3) 应用程序服务器从数据库服务器上获取数据，然后封装成包，并把数据包返回给客户端。可以在这里发送附加的信息。

(4) 客户端把数据包解码，然后显示给用户。

(5) 用户在客户端进行交互，数据进行更新（包括添加记录、删除记录和修改记录等等）。这些修改被客户端存储在一个修改日志中。

(6) 最后客户端把更新应用到应用程序服务器上。为了应用这些更新，客户端把修改日志封装成包并发送给服务器。

(7) 应用程序服务器把收到的数据包解码，并把更新提交给数据库服务器。如果一个记录不能更新（例如另外的一个应用程序正在更新这个记录），应用程序服务器或者使客户端对当前数据的修改服从正在进行的修改，或者把这些不能更新的记录保存起来。

(8) 在应用程序服务器完成了上面的解析过程之后，它把任何不能更新的记录返回给客户端以便于进行进一步的处理。

(9) 客户端接受不能解析的记录。

(10) 客户端从服务器上刷新数据。

3. 客户端应用程序的结构

对于终端用户来说，多层应用程序的客户端使用起来和前面介绍的两层数据库应用程序没有什么不同。用户通过标准的数据控件显示来自 TclientDataSet 控件的数据；TclientDataSet 从 Provider 控件上获取数据或者向该控件应用更新；它从一个连接控件上得到这个连接接口；连接控件建立了从客户端到应用程序服务器的连接；对于不同的连接协议，通常需要使用不同的连接控件（见表 9.11）。

表 9.11　用于连接的不同的控件

控件	协议
TDCOMConnection	DCOM
TsocketConnection	Windows Sockets (TCP/IP)
TwebConnection	HTTP
TSOAPConnection	SOAP (HTTP and XML)
TcorbaConnection	CORBA (IIOP)

4. 应用程序服务器的结构

当设置并运行一个应用程序服务器时，它不会建立任何和客户端应用程序的连接。相反，连接是由客户端应用程序来建立和维护的。客户端应用程序利用连接控件建立一个到应用程序服务器的连接。所有的这些都是自动发生的，不需要为此编写任何程序。

应用程序服务器是建立在一个远程数据模块基础上的，远程数据模块是一个用来支持

IappServer 接口的特殊数据模块。客户端应用程序使用 IappServer 接口和应用程序服务器上的 Providers 对象通信。

在 Delphi 中共有四种远程数据模块。

- TremoteDataModule：这是一个双接口的自动服务器。如果客户端要使用 DCOM、HTTP、Sockets 或者 OLE 来进行连接，则可以使用这种远程数据模块。有一种情况例外，如果要用 MTS 来安装应用程序，可以不使用该种远程数据模块。
- TMTSDataModule：这是一个双向的自动服务器。如果要创建的应用程序服务器是用于安装在 MTS 或者 COM+上的 Active Library（DLL），则可以使用这种类型的远程数据模块。可以在 DCOM、HTTP、Sockets 或者 OLE 上使用 MTS 远程模块。
- TcorbaDataModule：这是一个 CORBA 服务器。利用这种远程数据模块可以向 CORBA 客户提供远程数据。
- TsoapDataModule：这种数据模块可以在一个 Web Service 应用程序中作为 IAppServer 的派生类来提供数据。

如果应用程序服务器是用于 MTS 或 COM+上，远程数据模块应该包含处理应用程序服务器启动和关闭的事件。

可以在远程数据模块上放置一些非可视控件。在远程数据模块上，可以放置下面一些控件。

- 如果远程模块的目的是从一个数据库服务器上获取信息，那么它必须包含一个用来代表来自数据库服务器的记录的数据集控件。如果需要，还应该在远程数据模块上放置一些关于该数据集控件连接到数据库服务器所需要的连接控件，比如会话控件和数据库控件（Session 和 Database）。对于远程数据模块上的每个数据集控件，都应该为它提供一个 Provider 控件。
- 如果远程数据模块是向一个 XML 文档提供数据，那么必须在远程数据模块上放置一个 XML Provider 控件。XML Provider 的功能和数据集的 Provider 类似，除了它是从一个 XML 文档中获取数据并向 XML 文档中更新数据，而不是数据库服务器中这一点之外。

根据多层数据库应用程序的结构，一般采用下面的设计顺序。

(1) 创建一个应用程序服务器。

(2) 注册或者安装应用程序服务器。

(3) 创建客户端应用程序。

这里介绍的创建顺序是十分重要的。在创建客户端应用程序之前，应该先创建应用程序服务器并运行它。按照这样的顺序创建多层数据库应用程序，可以在设计客户端应用程序时，连接到应用程序服务器。

9.5.2　服务器端编程

从本质上说，创建多层数据库应用程序的过程和创建普通数据库应用程序的过程没有根本的区别。它们最大的区别是多层数据库应用程序服务器使用的是远程数据模块。

可以按照下面的步骤创建一个应用程序服务器。

(1) 新建一个应用程序。如果使用 SOAP 作为传输协议，则应该创建一个 Web Service 应用程序，对于其他的应用程序服务器，只要建立一个新的应用程序就可以了。

(2) 在当前的工程中加入一个新的远程数据模块。从主菜单中选择 File | New | Other，此时会显示一个如图 9.50 所示的对话框。选择其中的 Multier 选项卡，其中包含了我们要加入的远程数据模块。

(3) 在其中选择一个需要的数据模块，比如，这里使用 Remote Data Module。然后单击对话框中的 OK 按钮，此时会显示如图 9.51 所示的对话框。

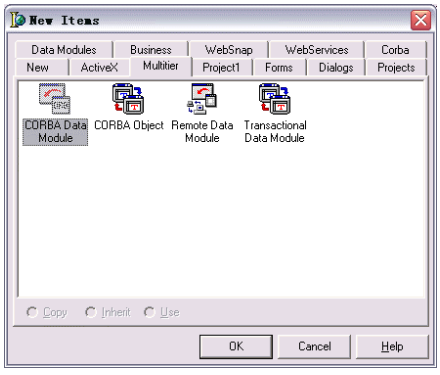

图 9.50　New Items 对话框

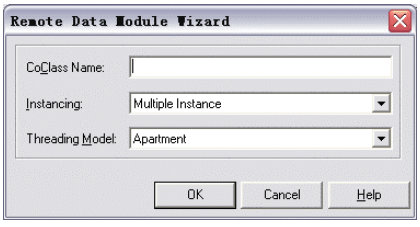

图 9.51　远程数据模块向导

(4) 根据不同的数据模块，向导的内容会有所区别，但是前三种都是类似的，第四种远程数据模块 Transactional Data Module 的向导窗口有所不同。下面主要针对 TRemoteDataModule 远程数据模块进行介绍。在上面的对话框中，首先必须为远程数据库模块提供一个类名称。这也是该类接口的命名基础。例如，如果指定了一个类名称为 MyDataServer，那么向导将创建一个声明为 TMyDataServer 的新单元，和一个 TRemoteDataModule 派生类。在远程数据模块向导的对话框中还要指定线程化的模式。可以选择 Single-threaded、Apartment-threaded、Free-threaded、Both 或者 Neutral。

- Singled-threaded：将同时只能接受一个客户要求。我们需要为同时发生的客户请求提供处理方法。
- Apartment-threaded：将确保远程数据模块的一个实例处理一个客户请求。在编写这类的数据模块时，必须要注意全局变量和全局对象之间的冲突。如果要使用基于 BDE 的数据集控件，建议使用这种模式。需要注意的是，此时需要一个把

AutoSestionName 属性设置成 True 的 Session 对象来处理线程问题。

- Free-threaded：此时的应用程序可以用几个线程处理多个同时的客户请求。此时需要关注的是线程的安全性。因为多个客户可能会同时访问我们的远程数据模块。如果使用的是 ADO 数据集，建议采用这种处理模式。
- Both：此时的应用程序和 Free-threaded 模式是相同的，除了一种情况之外——所有的回调都是连续的。
- Neutral：远程数据模块可以同时处理多个线程，就像 Free-threaded 模式，但是会保证两个线程不能同时访问同一个方法。

(5) 如果我们要创建一个 EXE 文件，必须指定要使用的实例类型。在这里可以使用三种模式。

- Internal：此种模式仅应用于客户端的代码也是处理数据过程的一部分情况。
- Single：每个客户连接运行它自己的实例。这些实例是每个客户程序专用的。
- Multiple：应用程序的一个实例将处理所有为客户应用程序创建的远程数据模块。每一个远程数据模块是每个客户连接专用的，但是它们使用同样的处理空间。

(6) 在这个例子中，指定为 Multiple Instance 和 Apartment-threaded 模式。然后在类名称中输入 MyFirstServer。单击对话框中的 OK 按钮，此时便在当前的应用程序中添加了一个远程数据模块。

(7) 在远程数据模块中放置需要的数据集控件和数据库控件以及相关的其他控件。例如在我们的程序中，放入了一个 TTable 控件和一个 TSession 控件。并把控件的相应属性进行了设置，以便它们连接到需要的数据库信息。下面是该控件属性在远程数据模块中的代码：

```
object MyFirstServer: TMyFirstServer
  OldCreateOrder = False
  object Table1: TTable
    DatabaseName = 'DBDEMOS'
    SessionName = 'Session1_1'
    TableName = 'customer.db'
    Left = 88
    Top = 24
  end
  object Session1: TSession
    Active = True
    AutoSessionName = True
    Left = 48
    Top = 24
  end
end
```

(8) 在远程数据模块窗口上为每一个数据集控件放置一个 TDataSetProvider 控件。然后把

它的 DataSet 属性设置成“Table1”，也就是上面所添加的数据集控件。

(9) 编写应用程序服务器的代码来完成相应的事件处理，共享商务规则，共享数据和处理安全性等问题。

(10) 保存、编译并注册或者安装应用程序服务器。如果我们的服务器应用程序不是使用的 DCOM 或者 SOAP，那么必须安装服务器程序。

到这里为止，我们已经创建了一个服务器程序。下面来讨论一下关于服务器程序的注册问题。

在客户应用程序能够定位我们的应用程序服务器和调用它之前，我们必须把自己的应用程序服务器注册到系统中，或者安装到系统中。但是不同类型的应用程序服务器的注册方式是不同的。在 Delphi 中，自动生成了它们的注册代码，如下所示。下面的代码中也展示了到目前为止我们建立的应用程序服务器的所有代码：

```
unit SrvDataMod;

{$WARN SYMBOL_PLATFORM OFF}

interface

uses
  Windows, Messages, SysUtils, Classes, ComServ, ComObj, VCLCom, DataBkr,
  DBClient, Project1_TLB, StdVcl, DBTables, DB, Provider;

type
  TMyFirstServer = class(TRemoteDataModule, IMyFirstServer)
    Table1: TTable;
    Session1: TSession;
    DataSetProvider1: TDataSetProvider;
  private
    { Private declarations }
  protected
    class procedure UpdateRegistry(Register: Boolean; const ClassID, ProgID: string); override;
    procedure MyInterface; safecall;
  public
    { Public declarations }
  end;

implementation

{$R *.DFM}
```

```
class procedure TMyFirstServer.UpdateRegistry(Register: Boolean; const ClassID, ProgID: string);
begin
  if Register then
  begin
    inherited UpdateRegistry(Register, ClassID, ProgID);
    EnableSocketTransport(ClassID);
    EnableWebTransport(ClassID);
  end else
  begin
    DisableSocketTransport(ClassID);
    DisableWebTransport(ClassID);
    inherited UpdateRegistry(Register, ClassID, ProgID);
  end;
end;

initialization
  TComponentFactory.Create(ComServer, TMyFirstServer,
    Class_MyFirstServer, ciMultiInstance, tmApartment);
end.
```

在目前的应用程序中，我们的窗口也就是 Form1 **没有起到任何作用。可以在窗口上放置一些控件来标志服务器的运行以及运行状态。**

运行一次这个的应用程序，我们的应用程序服务器便自动在系统中注册了。

9.5.3 客户端编程

创建客户端应用程序的过程和创建普通的应用程序没有什么区别，除了需要使用几个特殊的控件之外。

(1) **新建一个应用程序，根据需要，可以向程序中添加一个数据模块。当然也可以把需要的数据库相关的控件都放置在窗体上。**

(2) **在窗体上放置一个合适的连接控件。我们在介绍客户端应用程序结构时介绍了这些控件。在示例程序中，使用的是** TDCOMConnection **控件。**

(3) **如果你的应用程序服务器位于另外一台机器上，那么可以在** ComputerName **属性中输入该计算机的名称。由于我们是在同一台计算机上进行演示，所以这里可以空着这个属性。**

(4) **在** ServerName **属性旁边的按钮上单击，属性编辑器会列出你所指定计算机上可用应用程序服务器所能提供的所有远程服务名称。在这里是上面的** Project1.Remote**。**

(5) **当选择了** ServerName **属性之后，**ServerGUID **属性将被自动填写。**

(6) **放置一个** TclientDataSet **控件，把它的** RemoteServer **属性设置成我们需要的连接控件。这里就是上面放置的** TDCOMConnection **控件。**

(7) 为每个 TclientDataSet 控件设置它们的 Provider 属性。当选择该属性的时候，连接控件的 Servername 属性管理服务器应用程序将会启动,同时显示其中可以使用的 Provider 控件。选择其中所需要的一个。在我们的程序中，只有一个可以使用 Provider 控件。

(8) 放置一个 TdataSource 控件，然后把它的 DataSet 属性设置成这里的 TclientDataSet 控件。

(9) 接下来的工作和前面介绍的处理普通数据库应用程序的方法就完全一样了。比如可以在窗体上放置一些数据控件来显示这些窗体。在我们的示例程序中，放置了一个 TDBGrid 控件，用来显示获得的数据。

说明：

从上面的过程中可以看出，需要使用的控件主要是用于连接的连接控件和一个用于提供数据的数据集控件。这两个控件和普通的数据库应用程序中使用的控件有所区别。但是它们也有相似的地方。比如 TclientDataSet 控件的用法，就完全可以利用对 TTable 等数据集控件的用法来使用它们的方法和属性。

在设计客户端程序的时候，把 TclientDataSet 控件的 Active 属性设置成“ True ”，便可以看到来自数据库应用程序服务器的数据，如图 9.52 所示。

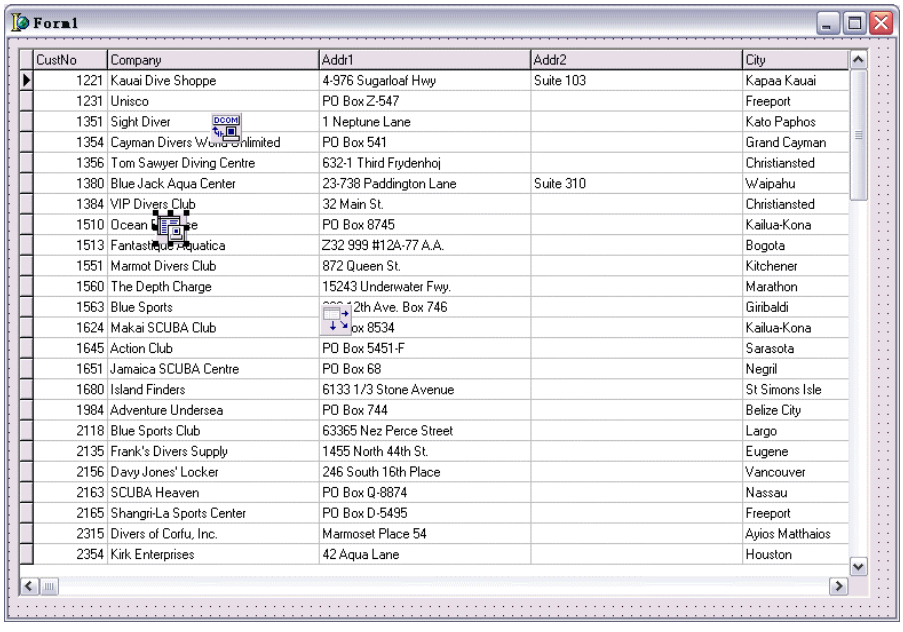

图 9.52　客户端在设计期获得服务器上的数据

在上面的过程中，我们通过一个简单的例子，介绍了如何建立多层的数据库应用程序。虽然例子比较简单，但是读者从这个过程中可以看出建立一个多层数据库应用程序的基本思路和基本方法。

9.6 本 章 小 结

本章介绍了开发数据库应用程序的相关内容，包括数据库应用程序的结构、各个部分的实现，以及 Delphi 中的常用数据库控件、对象的用法。此外还介绍了 Delphi 中的主从式关系数据库应用程序的开发以及多层数据库应用程序的开发。

在开发普通的数据库应用程序时，我们关注的是数据库的类型，以及对数据库进行操作的方法。在开发多层数据库应用程序时，我们关注的是连接方式、使用的协议等内容。

第 10 章　控 件 设 计

在前面的内容中，介绍了很多关于 Delphi 的程序设计知识。读者可能已经发现我们的编程总是和控件有关。Delphi 面向对象编程的体现之一便是使用控件。

在 Delphi 的面向对象编程中，控件可以说是其中的一个至关重要的技术，也是面向对象编程思想的具体体现之一。控件的存在，使得我们不必去关心任何控件内部的代码是怎样完成的，只要熟悉它们的各个属性和方法的功能就可以了。显然这对代码的封装是十分有用的。从前面两部分的介绍中，我们已经了解到，Delphi 提供了丰富的控件，这些控件正是构成应用程序的主要成分之一。

但是我们的目标还不仅仅是使用 Delphi 提供的或者是别人提供的控件，我们希望创建自己的控件，以实现自己的特殊目的。可以说，自定义控件技术是面向对象编程思想的最好的体现之一，它使得代码的重用性获得了极大的提高。

我们可以借助从已有的控件（包括 Delphi 预先提供的控件或者我们自己新创建的控件）派生新的控件。这样做的好处是只要完成少许工作便可以完成整个控件的编制工作。例如在前面的示例程序中，我们很快就创建了四五个控件。当然也可以从一个抽象的对象开始创建一个全新的控件。

在 Delphi 中，对控件的特殊属性以及对一些特殊的控件，都提供了它们自己的编辑器，比如 TChart 控件的编辑器。在本章中也将介绍如何创建自己定义的属性编辑器和控件编辑器。

在本章中，将主要介绍一些关于创建自定义控件的技术，主要包括：

- 控件概述
- 创建派生控件
- 创建包
- 属性编辑器
- 控件编辑器

10.1　控 件 概 述

控件的开发和使用是现代编程的重要特点，几乎所有的编程都会涉及控件的使用和创建。一些原来在这个方面所作努力比较少的公司也正在把目光投向这个重要的技术。但是在目前的所有开发环境中，Delphi 是其中最具竞争力的开发工具之一。

相比而言，Delphi 控件有以下三个非常大的优势。

- 它们是用 Delphi 语言建立的本地控件。这意味着可以在标准的 Delphi 程序内部来编写、调试和测试控件。总之，学习发掘和编写自己的 Delphi 控件比试图去理解和编写自己的 ActiveX 控件要容易得多。使用向导可以使创建 ActiveX 控件变得相对简单，但是真正理解这项技术是极为困难的。
- Delphi 控件是完全面向对象的，这意味着很容易就可以通过创建派生对象来改变和提高现存的控件功能。
- 它们轻巧便捷，可以直接链接到执行程序中去。然而 ActiveX 控件就显得笨重、缓慢，它们不能直接连接到程序中去而必须单独来处理。
- 尽管大家都知道，VBX 是十分重要的，ActiveX 控件的发展趋势也是令人十分看好的，另外，COM 也将会成为一个极其吸引人的结构。但是，在学习了这一章之后，读者就会发现，在 Delphi 中创建控件和上面这些技术比较起来是多么的容易，而且可以生成轻便、易用的包。可以创建几乎能完成所有事情的 Delphi 控件，包括从串行通讯到数据库连接到多媒体等。这一性能使得 Delphi 相对其他可视化工具来说具有很大的优势，使我们不必去面对那些比较抽象的其他开发语言，例如 C++。

Delphi 控件是一种灵活的工具，任何知道 OOP（面向对象）和 Delphi 语言的人都可以很容易地创建自己的控件。在本书后面要介绍的内容中，读者会学会创建控件所需的所有必备知识的详细解释，从 Delphi 自身的描述，到其语言的描述，到 OOP 实现的概述。在这一基础上，可以很容易地开始创建自己的控件。

在 Delphi 中，创建一个控件一般包括下面 6 个步骤。

(1) 确定一个父类。
(2) 创建一个单元文件。
(3) 在新的控件中加入属性、方法和事件。
(4) 测试该控件。
(5) 在 Delphi 环境中注册该控件。
(6) 为该控件提供一个帮助文件。

在这一节中，将对前 5 个步骤进行一个简略的讨论，这将使读者在阅读下面的内容时相对容易一些。第 6 个步骤会涉及帮助文件的编写工作，因为这不属于 Delphi 编程的范围内，所以我们就不再介绍了。

10.1.1 确定一个父类

继承是面向对象编程思想的几个重要方面之一。利用继承可以很好地实现代码的重用，为我们逐步编写复杂的代码提供了极大的方便。被继承的类就是这里要介绍的父类。

确定一个父类通常来说不是一件困难的事情，除非你的目的是从头开始创建一个控件。

就通常情况而言，在打算创建控件时，一般对控件起码具有的一些功能应该有一个基本的认识。例如，如果要创建一个用特定的文本格式显示或者接收字符的控件，我们可以想到应该从 TEdit 控件继承；要创建一个显示特别复杂的图形形状的控件，应该从 TShape 控件继承。再例如，Delphi 的 TChart 控件就是从 TPanel 控件继承来的，从前面介绍的许多该控件的属性中就可以看到许多 TPanel 控件的影子。

确定一个合适的父类是十分重要的。这不仅可以省时省力，而且在扩展自己的控件功能时也能提供极大的方便。我们无法通过这么简单的一个小节就能告诉你所有选择父类的情况，这只能靠你自己根据需要，在全面掌握 Delphi 已经提供的控件基础上对此进行选择。

10.1.2　创建一个单元文件

创建一个单元文件的工作是比较简单的。可以完全从一个空的单元文件开始，然后根据自己的需要输入代码。要创建这样的一个单元文件可以单击 File | New，然后从弹出的对话框中选择 Unit。这样便创建了一个空的单元文件。

但是，从创建控件的角度来说，完全没有必要这样做。我们可以利用 Delphi 提供的模板进行操作。选择 Components | New Component，便会显示一个如图 10.1 所示的对话框。利用这个对话框可以方便容易地创建一个单元文件。

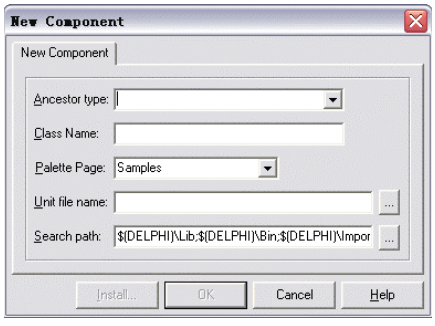

图 10.1　Delphi 的 New Component 对话框

在这个对话框中，可以指定要创建控件的父类、控件的类名称和控件的单元文件。在指定了这些信息之后，单击 OK 按钮，便可以创建一个单元文件。在这个单元文件中，已经建立了控件的基本框架。比如，如果要创建一个基于 TCustomImage 控件的新控件，那么此时生成的代码将如下所示：

```
unit jfsLabel;

interface

uses
  Windows, Messages, SysUtils, Classes, Controls, StdCtrls;
```

```
type
  TjfsLabel = class(TCustomLabel)
  private
    { Private declarations }
  protected
    { Protected declarations }
  public
    { Public declarations }
  published
    { Published declarations }
  end;

procedure Register;

implementation

procedure Register;
begin
  RegisterComponents('WokControl', [TjfsLabel]);
end;

end.
```

10.1.3 加入控件的属性

为控件加入属性是创建控件的工作中极为重要的一个步骤。我们总是通过控件的属性和方法来控制控件的行为的，在更多的情况下，是通过控件的属性来控制控件的。

控件的属性也是数据，所以 Delphi 的 Object Pascal 语言中关于数据类型的各种规定仍然适用于控件的属性。属性数据类型的不同，决定了将来它出现在属性编辑器中的形式和编辑方法也不同。Delphi 中控件的属性可以分成下面的几个类型。

- 简单类型：包括数字、字符和字符串，可以直接在属性编辑器中输入这些属性的值。
- 枚举类型：可以通过在属性编辑器中反复地单击对应的属性来选择希望的属性值，或者直接单击属性编辑器中该属性的下拉列表框，并从中选择需要的属性值。
- 集合类型：可以在属性编辑器中展开该集合的属性，集合中的每个元素就好像一个 Boolean 型的属性，设置为 True 便表示集合中包含该元素。
- 对象类型：要编辑对象类型的属性往往需要打开一个特殊的编辑器。例如在设置 TListBox 控件的 Items 属性时就可以使用 Delphi 的字符串编辑器。不过，如果对象本身的属性也是公开的，那么也可以在属性编辑器中展开这个对象类型的属性，然

后分别设置每一个子属性。

- 数组类型：数组类型是一种特殊的属性类型，必须为它设计专门的属性编辑器，Delphi 的属性编辑器不能编辑这类属性。

1. 简单类型

在定义控件的属性时，经常会使用控件的内部字段。控件的内部字段实际上就是控件的一些内部数据，它们一般被声明成私有的，以防止用户从外部对它进行访问。如果用户要访问这些字段，就必须通过控件的属性，所们通常也把属性称为控件内部字段的访问器。

我们来看一下下面这个控件的一个属性声明：

```
TCustomEdit = class(TWinControl)
private
  FMaxLength: Integer;
  procedure SetMaxLength(Value: Integer);
protected
  property MaxLength: Integer read FMaxLength write SetMaxLength default 0;
end;
```

这段代码节录自 Delphi 的 stdctrls.pas。MaxLength 属性就是 FMaxLength 字段的访问器。从这段代码中，我们也可以了解如何声明一个属性的语法。属性的定义包含了属性的名称、属性的类型、读和写的声明以及一个可选的默认值定义。

在声明了这样的属性之后，我们需要完成属性的访问方法，如果要读取控件的属性，我们在这里指定从 FMaxLength 内部字段直接读取。如果要设置该字段的值，需要使用 SetMaxLength 方法。在 stdctrls.pas 中，该方法的定义如下：

```
procedure TCustomEdit.SetMaxLength(Value: Integer);
begin
  if FMaxLength <> Value then
  begin
    FMaxLength := Value;
    if HandleAllocated then DoSetMaxLength(Value);
  end;
end;
```

这个方法首先检查目前控件的该属性值是否和要赋予的值相同，不同则进行赋值，否则忽略赋值。这是在编写属性访问方法时经常使用的方法。

2. 枚举类型

典型的枚举类型属性是许多控件都具有的 Align 属性。要在我们自己的控件中加入枚举类型，需要首先声明自己的枚举类型：

```
Type
  Tjfsshadowstyle=(shdNone,shdWithDigit,shdAll);
```

然后需要声明一个内部字段来代表每一个枚举值。在下面的程序中，声明了一个枚举型属性：

```
type
  Tjfsshadowstyle=(shdNone,shdWithDigit,shdAll);
  Tjfsnum = class(TGraphicControl)

  private
    FShadowStyle:TjfsshadowStyle;
    procedure setshadowStyle(value:TjfsshadowStyle);
        { Private declarations }
  published
    property ShadowStyle:TjfsshadowStyle read FshadowStyle
                              write setshadowStyle default shdNone;
end;
```

如果把这个控件安装到 Delphi 的集成开发环境中，此时的属性编辑器将如图 10.2 所示。

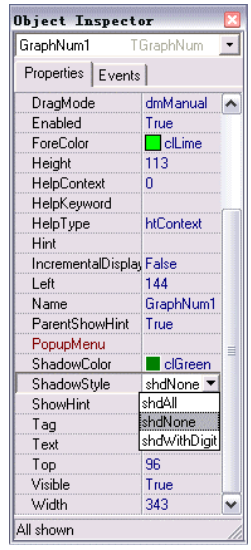

图 10.2　属性编辑器中的枚举型属性

3. 集合类型

属性编辑器中的集合类型属性可以进行展开，然后可以设置这个集合中的每一个子属性。要为一个控件加入集合类型的属性，需要先声明一个集合类型。

```
unit WokTest;

interface

uses
  Windows, Messages, SysUtils, Classes, Controls;

type
  TWokSetP = (WokP1,WokP2,WokP3,WokP4);
  TWokSetPs = Set of TWokSetP;
  TWokTest = class(TCustomControl)
  private
    FOptions: TWokSetPs;
    { Private declarations }
  protected
    { Protected declarations }
  public
```

```
    { Public declarations }
  published
    property Options: TWokSetPs Read Foptions Write Foptions;
    { Published declarations }
  end;

procedure Register;

implementation

procedure Register;
begin
  RegisterComponents('WokControl', [TWokTest]);
end;

end.
```

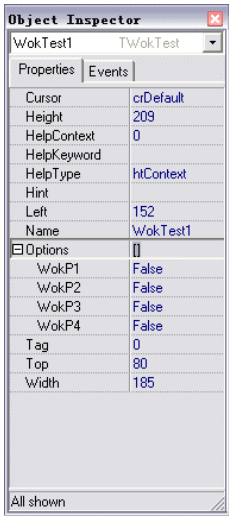

图 10.3　属性编辑器中的集合类型属性

在上面的程序中演示了如何在一个控件中加入集合类型的属性，如果把这样的一个控件加入到 Delphi 的集成开发环境中，该控件的属性编辑器将如图 10.3 所示。

4. **对象类型**

一个控件的属性可以是一个对象，甚至是另外一个控件。例如许多控件中都有 Canvas 对象。如果一个属性是对象，那么它就可以在属性编辑器中展开，这有点像上面介绍的集合类型属性。

如果要把一个对象作为另外一个控件的属性，则这个对象必须是从 TPersistent 派生出来的，当然经过多级派生也是可以的，这样对象的公开特性才会显示在属性编辑器中。

下面我们为自己的控件加入一个对象类型的属性。在代码中，首先声明了一个对象类型，并定义了它的方法，然后在要创建的控件中加入一个私有的 TMyObject 类型字段，但是需要特别注意的是，需要在控件的 Create 方法中为这个字段创建一个实例。我们目前面临的问题是需要自己来管理内存中这个对象的实例。所以在创建控件时，不仅要重载控件的 Create 方法，还要重载控件的 Destroy 方法。在 Destroy 方法中，要释放我们所创建的对象实例。下面列出了完整地加入对象类型属性的代码，注意，在下面的程序中还加入了一个 TLabel 类型的对象属性。

说明：

在控件中加入对象类型的属性时，可能经常会遇到 Delphi 提示你没有定义该对象的情况。这是因为在你程序的 Uses 语句中没有包含定义了该对象的单元。通过 Delphi 的在线帮助找到该单元，并添加在 Uses 语句中便可以解决这个问题。

```
unit WokTest;

interface

uses
  Windows, Messages, SysUtils, Classes, Controls;

type
  TWokSetP = (WokP1,WokP2,WokP3,WokP4);
  TWokSetPs = Set of TWokSetP;

  TMyObject = class(TPersistent)
  private
    FMyP1:integer;
    FMyP2:string;
  public
    procedure Assign(Source:TPersistent);override;
  published
    property MyP1: integer read FMyP1 write FMyp1;
    property MyP2: string read FMyP2 write FMyP2;
  end;

  TWokTest = class(TCustomControl)
  private
     FOptions: TWokSetPs;
     FLabel:TLabel;
     FObject:TMyObject;

     procedure SetLabel(value:TLabel);
     procedure setmyobject(value:TMyObject);
    { Private declarations }
  protected
    { Protected declarations }
  public
    constructor    Create(Aowner:TComponent);override;
    Destructor Destroy; Override;
    { Public declarations }
  published
     property Options: TJfsSetPs Read Foptions Write Foptions;
     property MyLabel: TLabel Read FLabel Write SetLabel;
     property MyObject: TMyObject Read FObject write SetMyObject;
    { Published declarations }
```

```
    end;

procedure Register;

implementation

procedure Register;
begin
   RegisterComponents('WokControl', [TWokTest]);
end;
constructor TWokTest.Create (Aowner:Tcomponent);
begin
   inherited Create(AOwner);
   FLabel := TLabel.create(Self);
   FObject := TMyObject.Create;
end;
destructor TWokTest.Destroy;
begin
   FLabel.free;
   FObject.Free;
   inherited Destroy;
end;
procedure TWokTest.SetLabel(value:TLabel);
begin
   if Assigned(Value) then
       FLabel := value;
end;
procedure TWokTest.setmyobject(value:TMyObject);
begin
   if Assigned(Value) then
       FObject.assign(value);
end;

procedure TMyObject.Assign(Source:TPersistent);
begin
    if Source is TMyObject then
    begin
        FMyP1 := TMyObject(Source).MyP1;
        FMyP2 := TMyObject(Source).MyP2;
        inherited assign(Source);
    end;
end;
```

end.

加入了对象类型的控件在 Delphi 属性编辑器中的样子如图 10.4 所示。

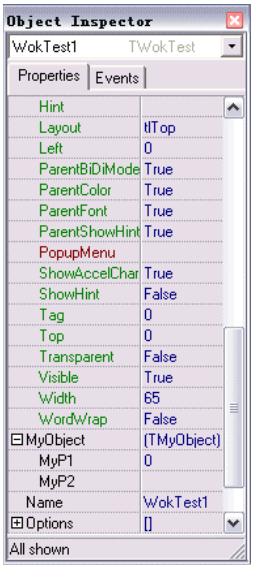

图 10.4 属性编辑器中对象类型的属性

5. 数组类型

关于数组类型的属性处理要稍微复杂一些。因为如果不进行特殊的处理，就不能像其他类型属性一样在属性编辑器中直接访问它。典型的数组类型属性有 TMemo 控件的 Lines 属性，等等。如果希望在属性编辑器中也能够访问这些数组属性，需要为该属性创建专门的属性编辑器，例如常用的字符串属性编辑器。关于设计属性编辑器的问题我们留在下一章中进行介绍，现在的任务是介绍如何在一个控件中加入数组属性。

下面我们来看一个新的控件，这个控件的代码如下所示：

```
unit planets;

interface

uses
  Classes, SysUtils;

type

  TddgPlanets = class(TComponent)
  private
    // 数组属性访问方法
    function GetPlanetName(const AIndex: Integer): String;
    function GetPlanetPosition(const APlanetName: String): Integer;
  public
    property PlanetName[const AIndex: Integer]: String read GetPlanetName; default;
    property PlanetPosition[const APlantetName: String]: Integer read GetPlanetPosition;
  end;

implementation

const
  PlanetNames: array[1..9] of String[7] =
   ('Mercury', 'Venus', 'Earth', 'Mars', 'Jupiter', 'Saturn',
    'Uranus', 'Neptune', 'Pluto');
```

```
function TddgPlanets.GetPlanetName(const AIndex: Integer): String;
begin
  if (AIndex < 0) or (AIndex > 9) then
    raise Exception.Create('Wrong Planet number, enter a number 1-9')
  else
    Result := PlanetNames[AIndex];
end;

function TddgPlanets.GetPlanetPosition(const APlanetName: String): Integer;
var
  i: integer;
begin
  Result := 0;
  i := 0;
  repeat
    inc(i);
  until (i = 10) or (CompareStr(UpperCase(APlanetName),
      UpperCase(PlanetNames[i])) = 0);

  if i <> 10 then
    Result := i;
end;

end.
```

上面的代码演示了两个数组属性，一个是以整数为下标，而另外一个则以字符串为下标。需要注意的是，这些属性是由读访问方法返回的，而不是由内部字段返回。

总结数组属性的添加，可以得出下面的几个结论或者说是经验。

- **数组型的属性必须带有读写访问方法，而不能是内部的字段。**
- **如果是多重数组，则访问方法也必须有相应个数的参数。**

10.1.4　加入控件的方法

为控件加入方法和我们为普通的类加入方法没有什么区别，只不过这里要特别注意这些方法的作用范围，如果你不希望用户从外部调用这些方法，则应该把它声明在 private **或者** protected **中。**

10.1.5　加入控件的事件

需要说明的是，控件的事件也是一类属性，它表示的是当某个动作发生时，就会执行预

先指定的代码。首先，我们来研究一下事件是如何产生的。一个事件有可能产生于操作系统、用户的操作，甚至是应用程序本身。Delphi 的事件机制就是把事件和特定的代码联系起来，当事件发生时，就会调用相应的代码。处理事件的代码实际上就是一个方法，我们称之为事件句柄。例如，当用户单击鼠标时，一个 WM_MOUSEDOWN 消息就被发送到 Win32 系统中。Win32 把这个消息传递给相应的控件，这样这个控件就可以响应这个消息。首先这个控件检查相应的事件属性是否指向了一段代码，即事件句柄。如果是，就执行这个事件句柄。

从这里可以看出，Delphi 事件机制的关键是消息机制。如何发送正确的消息，将是自定义事件的关键。

OnClick 事件是一个标准事件。OnClick 事件或者其他事件都有一个事件调度方法。事件调度方法通常是在控件的 protected 部分声明的，它的作用是检查事件属性是否指向了一段代码，即事件句柄。对于 OnClick 事件来说，它的事件调度方法就是 Click。OnClick 事件属性和 Click 方法都是在 TControl 中声明的，代码如下：

```
TControl = class(TComponent)
private
  FOnClick: TNotifyEvent;
protected
  procedure Click; dynamic;
  property OnClick: TNotifyEvent read FOnClick
write FOnClick stored IsOnClickStored;
end;
```

其中 Click 方法是这样实现的：

```
procedure TControl.Click;
begin
  if Assigned(FOnClick) and (Action <> nil)
       and (@FOnClick <> @Action.OnExecute) then
    FOnClick(Self)
  else if not (csDesigning in ComponentState) and (ActionLink <> nil) then
    ActionLink.Execute
  else if Assigned(FOnClick) then
    FOnClick(Self);
end;
```

事件属性实际上就是一个方法指针。方法指针也是有类型的，例如上面的 FOnClick 的类型就是 TNotifyEvent。该类型是这样声明的：

```
TNotifyEvent = procedure(Sender: TObject) of object;
```

作为一个控件的创作者，我们需要自己声明事件属性，并自己创建事件调度方法。而对于控件的使用者来说，只要创建事件句柄就可以了。事件调度方法会自动检查用户是否建立

了事件句柄。

在声明一个事件属性之前，需要仔细考虑一下，是否真的需要这样一个特殊的事件。最好我们先熟悉一下 Delphi 中的已有事件。在很多情况下，自定义的控件往往是从一个已有的控件中继承事件，也就是说，自定义的控件已经继承了一些事件，而且这些事件基本上已经能够满足我们绝大多数的需要了。只有在一些特殊的情况下，我们才会想到定义自己的事件。

下面创建一个新的控件，用来演示如何定义自己的事件。假设我们有一个控件，希望它能够每隔 30 秒就触发一次事件。也就是说，事件在一分钟中会发生两次。当然，对于这样的功能，可以利用 TTimer 控件的 OnTimer 事件来完成。但是，如果我们专门定义一个事件，那么用户就不用处理 OnTimer 事件，只要处理这个新定义控件的新事件就可以了。

下面列出了创建这个控件所需要的代码：

```
unit halfmin;

interface

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

type
  TTimeEvent = procedure(Sender: TObject; TheTime: TDateTime) of object;
  TddgHalfMinute = class(TComponent)
  private
    FTimer: TTimer;
    FOnHalfMinute: TTimeEvent;
    FOldSecond, FSecond: Word;
    procedure FTimerTimer(Sender: TObject);
  protected
    procedure DoHalfMinute(TheTime: TDateTime); dynamic;
  public
    constructor Create(AOwner: TComponent); override;
    destructor Destroy; override;
  published
    property OnHalfMinute: TTimeEvent read FOnHalfMinute
write FOnHalfMinute;
  end;

implementation

constructor TddgHalfMinute.Create(AOwner: TComponent);
begin
```

```
    inherited Create(AOwner);
    if not (csDesigning in ComponentState) then
    begin
      FTimer := TTimer.Create(self);
      FTimer.Enabled := True;
      FTimer.Interval := 500;
      FTimer.OnTimer := FTimerTimer;
    end;
end;

destructor TddgHalfMinute.Destroy;
begin
  FTimer.Free;
  inherited Destroy;
end;

procedure TddgHalfMinute.FTimerTimer(Sender: TObject);
var
  DT: TDateTime;
  Temp: Word;
begin
  DT := Now;
  FOldSecond := FSecond;
  DecodeTime(DT, Temp, Temp, FSecond, Temp);
  if FSecond <> FOldSecond then
    if ((FSecond = 30) or (FSecond = 0)) then
      DoHalfMinute(DT)
end;

procedure TddgHalfMinute.DoHalfMinute(TheTime: TDateTime);
begin
  if Assigned(FOnHalfMinute) then
    FOnHalfMinute(Self, TheTime);
end;

end.
```

在建立一个自定义事件的时候，一定要考虑事件句柄要有哪些参数，换句话说，就是要提供哪些信息。请读者观察这个控件中的事件类型定义中的参数。

在创建控件时，还有其他的一些技术，我们希望在后面的实例中能够更为形象具体地介绍它们。在本章后面的内容中，将集中力量介绍如何利用已有的比较成熟的控件来创建新控件。虽然有的人也许会说，这样创建的控件是比较简单的，但是，如果仔细研究一下就会发

现，实际上，几乎我们要创建的所有控件都是从已有的控件派生出来的。我们没有必要从零开始创建一个控件。而且利用已有控件以及控件的组合，可以创建出适合我们需要的绝大多数控件。

10.2　创建派生控件

上面我们提到了，从现有的控件中创建派生控件是十分重要的，它使我们不必从零开始创建一个控件。可以先继承原有控件中对我们有帮助的属性和方法，然后添加我们希望存在的或者屏蔽掉我们不希望存在的属性和方法。

在这一部分中，我们将会了解到如何创建一系列的从 TEdit、TPanel、TLable 派生来的自定义控件。对标准 TEdit、TLable 控件所做的改变包括：改变它们的颜色和它们的字体颜色、名称、大小以及风格。这部分的目的是为了显示如何创建一组自定义控件，可以将它们放在控件选项板里作为特殊的控件来使用，或者用来定义属于特殊部门或公司的一套应用程序的外观。

10.2.1　创建简单的控件

下面我们先从一个简单的例子开始，力图让读者能够对创建控件的过程有一个初步的认识。这样做可以使读者比较容易地、由浅入深地掌握相关的技术。如果一开始就从一个复杂的例子开始，那么读者很可能会不太适应和理解所要介绍的内容。

下面先给出要创建的控件代码，请读者把代码浏览一遍，并试着去理解和掌握它的基本结构，然后再来介绍创建这样的控件所需要的各方面知识。

```
unit JFSSmallEdit;

interface

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

type
  TJFSSmallEdit = class(TEdit)
  public
    constructor Create(AOwner: TComponent); override;
  end;
```

```
    procedure Register;

implementation

constructor TCCSmallEdit.Create(AOwner: TComponent);
begin
   inherited Create(AOwner);
   Color := clBlue;
   Font.Color := clYellow;
   Font.Name := '宋体';
   Font.Size := 12;
   Font.Style := [fsBold];
end;

procedure Register;
begin
   RegisterComponents('CustomControl', [TJFSSmallEdit]);
end;

end.
```

这个控件的内容是十分简单的，它仅仅是创建了一个 TEdit 控件，然后对它的颜色和字体属性进行了设置。在创建了控件之后，我们一定希望能够先亲眼观察一下自己的控件外观和表现，然后再根据自己的需要来进一步修改这个控件。那么，我们可以在不进行其他操作的情况下，在新建的应用程序的 Uses 语句中加入该控件的单元文件名，例如在这里是 JFSSmallEdit。然后就可以在程序中声明一个我们创建的控件类型变量，并调用该控件的 Create 方法创建该控件的一个实例。

例如在下面的程序中，就创建了这样的一个实例。

```
var ss:TJFSSmallEdit;
procedure TForm1.FormCreate(Sender: TObject);
begin
     SS := TJFSSmallEdit.Create(Self);
     ss.parent := Self;
     ss.Left := 100;
     ss.Top := 100;
     ss.Text := '成功创建！';
end;
```

运行这个应用程序，可以看到，程序在创建窗体的同时也创建了一个新的控件。

10.2.2　注册控件

Delphi 中的控件绝大多数都放置在 Delphi 的控件选项板上，这使得控件的使用非常方便。只要单击控件选项板上对应的选项卡中的对应控件，然后在窗体上放置这样的一个控件，控件的创建工作便完成了。如果我们自己创建的控件也能够具有这样的能力，显然是一件十分诱人的事情。当然，我们是可以做到这一点的，但是在把一个自定义控件放置到控件选项板之前，需要注册这个控件。

观察上面的程序，可以发现，程序中具有下面的一个方法：

```
procedure Register;
begin
  RegisterComponents('CustomControl', [TJFSSmallEdit]);
end;
```

注册一个类可以使得当该单元被编译到 Delphi 的控件库中时，Delphi 的控件选项板能够知道系统中又添加了新的控件，并把它显示在对应的选项卡上。注册过程对与该单元一起编译的程序没有任何影响。除非读者的程序主动调用了该单元中的注册过程（实际上我们没有任何必要进行这样的操作），注册过程的代码甚至永远不会出现在我们的可执行程序中。

像通常做的那样，创建一个目录并将上面的单元文件和工程文件保存在里面。但是需要提到的是，我们创建的控件的单元文件最好不要和普通的工程文件与单元文件混合在一起，否则时间长了可能会造成混乱，而且不利于将来对控件进行进一步的处理。更为重要的是，定义控件单元文件中的代码将作为 Delphi 代码的一部分来起作用，这样，读者可以只需要一个通向所有该类文件的单一目录。如果把需要的所有自定义控件放置到一个或者几个特定的文件夹中，无疑会使得我们在创建应用程序时比较容易查找需要的代码，而且整个程序所使用到的文件也有一个清楚的框架。

另外的一个问题是，建议你在把控件添加到控件选项板上时，要注意，不要把控件随意地添加到任何选项卡中，这样可能会造成极大的混乱。也就是说尽量不要采用 Delphi 提供的默认选项卡，因为我们可以在这里指定自己的选项卡。例如在上面的程序中，我们指定了一个 CustomControl 选项卡。最好把自己定义的控件按照功能分成几个类，然后把它们分别放置到各自独立的选项卡上。

10.2.3　改变控件的默认行为

上面建立控件的目的是给出一个 TEdit 类型的新的默认行为，以便于它以某种颜色和字体出现。为了达到这个目的，需要重载 TEdit 对象的 Create 方法，并且改变内部的字体。为了声明该方法，在你的类声明中写下如下的声明：

```
TJFSSmallEdit = class(TEdit)
public
  constructor Create(AOwner: TComponent); override;
end;
```

TJFSSmallEdit 的 Create 方法被声明为 public。因为在编程时，可能会在运行期动态地生成一些 TJFSSmallEdit 类的实例，如果不把这个方法声明成 public，那么可能导致在程序的外部就无法访问该方法，也就不能创建对象了。

Create 向 TJFSSmallEdit 类型传递了一个参数，该参数是一个封装了需要派生其他控件的最小功能的基类。特别是，放置控件的任何一个窗体通常就是控件的所有者。但是，所有者不一定必须是一个窗体。例如，它可能是一个 TPanel。实际上，任何的 VCL 控件都有内嵌功能，这使得它可以非常合适地扮演控件所有者的角色。最后，使用 override 指令表明这是一个读者想要重新定义的虚方法。

Create 方法的实现是很简单的：

```
constructor TJFSSmallEdit.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);
  Color := clBlue;
  Font.Color := clYellow;
  Font.Name := '宋体';
  Font.Size := 12;
  Font.Style := [fsBold];
end;
```

代码首先调用了父类的 Create，将 AOwner 变量传递给它。用户将会把一个控件拖到一个窗体中，该窗体将成为控件的所有者。在这种情况下，AOwner 是指向窗体的变量。VCL 用它来初始化 Owner 的属性，该属性是拥有控件的一个对象。

控件的所有者负责在最后来释放该控件。这个过程在所有者自身将被破坏的时候发生。如果你自己处理一个控件，要保证把它设置为 nil，否则控件的所有者在试图释放它的时候，可能要有出异常的危险。然而，大多数的情况下，你不必担心释放控件，可以让所有者来做这项工作。

10.2.4 测试控件

在把一个新定义的控件添加到控件选项板之前，应该对它进行测试。例如在上面的程序中,我们在运行期创建了这样的一个控件，并对该控件的许多属性进行了设置。这些代码位于窗体的 OnCreate 事件句柄中，如下所示：

```
var ss:TJFSSmallEdit;
```

```
procedure TForm1.FormCreate(Sender: TObject);
begin
    SS := TJFSSmallEdit.Create(Self);
    ss.parent := Self;
    ss.Left := 100;
    ss.Top := 100;
    ss.Text := '成功创建！';
end;
```

这段代码创建控件并且将它在主窗体中显示出来。当然，Self 是 TForm1 从它自己内部的方法中引用的。新控件的所有者是 Form1，当程序结束时，它负责处理该控件。就像我们以前所说的那样，该过程是自动地进行的。读者永远不用担心显示在窗体中的一个可视控件的处理问题。

控件的父类也是 Form1。当 Windows 试图确定如何显示窗体在屏幕上时，它会使用 Parent 变量。如果读者在窗体中放置一个面板并在面板中放置一个按钮，那么按钮的所有者是窗体，但父类是面板。所有关系决定了控件是什么时候和如何被撤销的，而父类关系决定了控件在哪里和如何被显示。所有关系是 Delphi 的一个基本的内容，但是父类关系是 Windows 主要考虑的问题。

我们想强调的是，动态创建控件的过程是十分有用的，特别是当你测试自己控件的时候。通过这样的测试，可以防止你向控件选项板中加入不能使用的或者存在着重大缺陷的控件。否则，可能会使得整个 Delphi 环境变得不稳定。

10.3　创　建　包

当你把一个新定义的控件进行了测试之后，如果对测试结果满意，则需要进行的下一个步骤就是把这个控件放置到控件选项板上。在完成这个步骤的过程中，需要使用一个称为包的项目。

10.3.1　Delphi 中的包

包是一个包括一个或更多的控件、对象或功能的特殊种类的 DLL（动态连接库）。它的主要优点是允许读者将控件封装在一个库中并与多个程序共享。从 Object Pascal 工作组中出现的包是为了建立一种在 DLL 中储存对象的好方法。在不断发展的过程中，它们开始承担一种更大和更复杂的角色。

如果想将一个控件放置在控件选项板中，必须创建一个包。所有在控件选项板中显示的控件都必须储存在包中。在控件选项板中储存的控件可以直接和我们的程序相连。就因为控

件选项板使用包，所以这意味着在可执行程序里不一定非得使用它们。注意 Delphi 的 LIB 目录中的 DCU 文件。这些 DCU 文件包括可以直接连接到我们程序里的控件。

你可以创建包括一系列功能的包。例如，如果创建了一套多媒体控件，将它们放入自己的包中是非常明智的。聪明的程序员可以创建只包括他们自己的程序所使用的控件的小的包。而且，可以仅简单地通过创建一个包的新版本来提高特定程序的性能。

可以浏览现存的已安装的 Delphi 包的内容。为了做到这一点，请从 Delphi 菜单中选择 Components | Install Packages。此时会出现如图 10.5 所示的 Project Options 对话框。

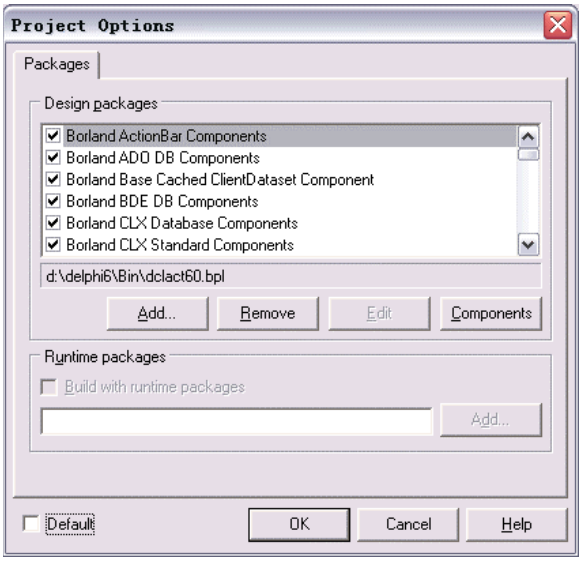

图 10.5 Project Options 对话框

当你感兴趣的一个包加亮显示时，选择 Components 按钮。该包的控件列表将会在一个对话框中显示，如图 10.6 所示。

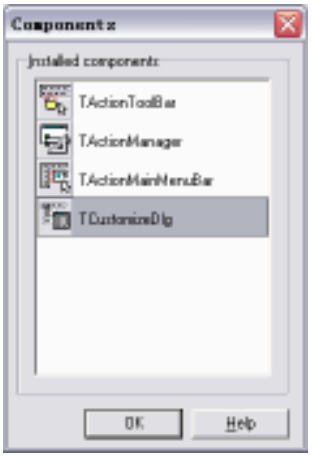

图 10.6 包中的控件

也可以使用 Install Packages 菜单选项从 Delphi 中增加或删除存在的包。也可以通过去掉一个包的名字左边复选框中的选中标记来使一个特定的包暂时失效。

10.3.2 创建自己的包

上面介绍的主要是 Delphi 中已经存在的或者是别人已经设计好的包。下面我们就来介绍如何创建自己的包。

可以通过选择 Component | Install Component，然后选择 Into New Component 选项卡来创建包，此时的对话框如

图 10.7 所示。

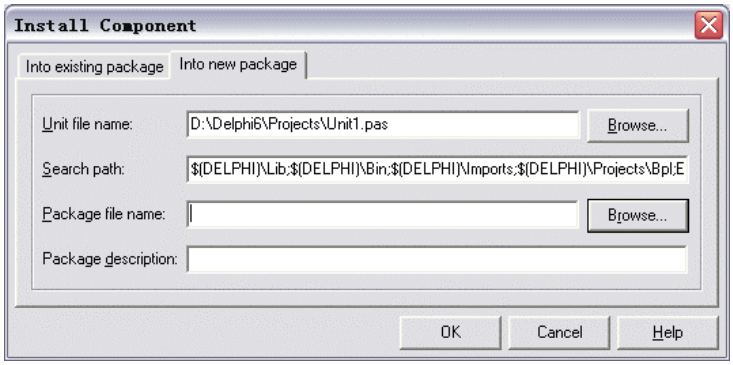

图 10.7　Install Component 对话框

现在用户就可以浏览硬盘来寻找 JfsSmallEdit.pas 文件，并且允许 Delphi 在后台自动创建包。例如在这里直接在 Pakage File name 左边的框中输入包文件所在的路径和文件名称。

也可以利用 Delphi 的新建功能来创建新的包。从 Delphi 菜单上选择 File | New | Others，并从弹出的对话框中选择 Package 项目。这会激活 Package Editor 并创建一个称为 Package1.dpk 的新文件，如图 10.8 所示。

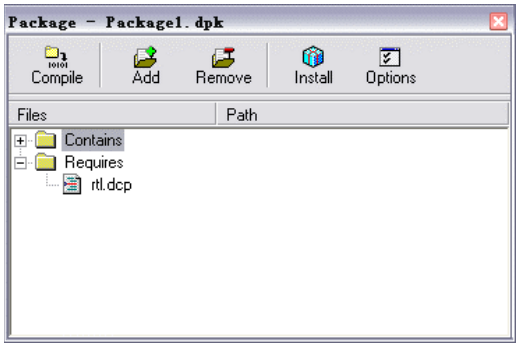

图 10.8　Package Editor 窗口

把这个文件保存在实例程序的 Common 路径中。如果想查看包的源文件，右击 Package Editor 的 Contains，并且从菜单中选择 View Source，一般来说我们没有必要这么做。

为了向包中增加一个单元，可以自己手动编辑 uses 子句，或者也可以单击 Package Editor 顶部的 Add 按钮。如果单击了 Add 按钮，就立刻可以浏览想要包括在包中的文件，此时会显示如图 10.9 所示的 Add 对话框。

利用这个对话框中的第一个选项卡，可以加入我们已经创建的控件，例如在前面创建的简单控件。利用后两个选项卡可以迅速地创建新的控件框架，这些步骤和创建新的控件的步骤十分类似，所不同的是现在创建的控件直接加入到了包文件中。

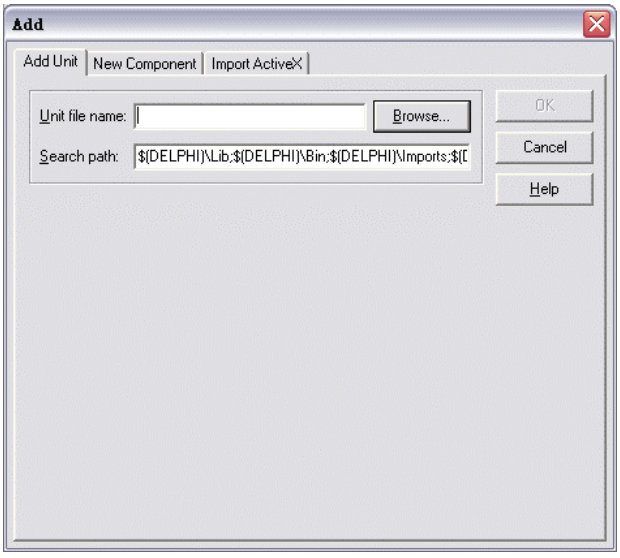

图 10.9 Add 对话框

当我们已经创建了一个包并且将包含自己的控件的单元加入到其中之后，就可以简单地通过单击 Package Editor 顶部的 Compile 和 Install 按钮在控件选项板中安装自己的控件了。在安装成功后会显示一个对话框，上面显示了生成的包的位置和注册的控件的类。单击了 OK 按钮之后，此时的控件面板上便会增加一个我们指定名称的选项卡。

在这里，我们添加的控件直接继承了 Delphi 中的 TEdit 控件的图标。在后面的内容中，还将介绍如何为自己的控件添加符合控件功能的图标。

可以向包文件中加入任意多的单元。如果想要编辑一个现存的 DPK 文件，那么只需要选择 File | Open 菜单来浏览具有.dpk 扩展名的文件。

可以在打开另一个工程的同时让 Package Editor 处于打开状态。如果 JFSSmallEdit.pas 文件和当前的工程相连，那么任何控件上的改变都会在第一次编译工程时反映出来。然而，这些改变不会在设计态时在控件中反映出来，除非重建包含该控件的包。例如，如果给一个控件增加了一个新发行的属性，那么立即可以在程序的代码中使用它，但是它不会在属性编辑器中显示出来，除非重建包含该控件的包。

在创建并安装了一个控件之后，用户就可以创建一个新的程序并将自己的控件放在主窗体中。这时我们将发现自己创建了和预先安装在 Delphi 中的控件功能完全类似的控件。

10.4 属性编辑器和控件编辑器

在前面曾经提到过，对于一些特殊的属性，可以专门为它们创建属性编辑器。属性编辑

器也是一类对象，只不过它们比较特殊而已。下面就开始介绍如何创建属性编辑器。

在介绍属性编辑器之前，我们要先建立一个基本的控件，然后编写它的属性编辑器和控件编辑器。到目前为止，我们还没有创建一个真正像样的控件。下面就结合介绍属性编辑器和控件编辑器，来介绍一个基本上可以用于实际的编程用途的控件——TJfsNum 控件。

10.4.1　创建一个 TJfsNum 控件

在开始创建一个控件之前，如果要创建的控件无法从现存的控件中继承时，我们应该选择从哪些控件派生而来？下面先解答这个问题。

通常来说，新的控件可以从 4 个抽象类继承出来。术语“抽象”有特殊的技术含义，但这里用它指的是任何满足如下条件的对象：只有当这个对象存在时才能创建它的继承类。简单地说，下述 4 个对象拥有所有的控件都需要提供的功能。

- TwinControl 和 TcustomControl 基本类可以用来产生标准的 Windows 控件，比如像 Tedit 等。这两个类的继承类存在于它们自己的窗口内，可以接受输入焦点而且拥有标准的 Windows 句柄。TListBox，TTabbedNoteBook，TNoteBook 和 TPanel 都是这种控件。大部分这类的控件都从 TCustomControl 继承而来，而后者正是 TWinControl 的继承类。此两类对象的差别在于 TCustomControl 有 Paint 的方法而 TWinControl 没有。如果想要自己画出新控件的显示外观，就必须从 TCustomControl 类继承；如果对象自己知道如何描绘自己，就可以从 TWinControl 继承。
- TgraphicControl 类用于不需要接收输入焦点，不包含其他控件并且不需要句柄的控件。这些控件直接在父控件上显示，这样可以节省 Windows 资源。不需要句柄则可以减少 Windows 管理的经常开支，这意味着更快地显示更新。简单地说，TGraphicControl 类存在于它们的父窗口内而不是像 TwinControl 继承类一样拥有自己的窗口。它们使用父窗口的句柄和设备联系。这些类仍然有 Handle 和 Canvas 属性可供使用，但这些属性都属于它们的父窗口。TLabel 和 TShape 就是这一类的控件。
- 利用 TComponent 可以创建不可见控件。如果想建立像下面这些工具：TTable、TQuery、TOpenDialog 或 TTimer 设备，就应该从这里开始。这些控件存在于控件面板上，但它们是预先形成用户可以通过代码获取的内部功能，而不是在运行时显示给用户。在本章后面将要提到的 TJFSFileIterater 控件就是这一类的控件。像 TOpenDialog 从技术上而言也是不可见控件。尽管它会弹出对话框，但这个控件本身是不可见的。在本章后面的 TJFSPickDirDlg 例子中将给出一个创建此类控件的例子。

下面两条规则可用来判断在何处创建 TWinControl 或 TGraphicControl 继承类。

- 用户需要直接和一个可见控件进行交互作用时可以创建 TWinControl 或

TCustomControl 继承类。

❖ 用户不需要直接和可视控件进行交互作用时可以创建 TGraphicControl 继承类。

在这些控件上单击鼠标或期望用键盘输入字符都不会有明显的结果。这些控件从来不会接受焦点。现在在窗体上放置 TEdit 控件。它将对鼠标单击有所反应，接受焦点，并且也可以在里面输入字符。TEdit 控件是 TWinControl 的继承类，而 TShape 则是 TGraphicControl 的继承类。

当无法确定该从 TWinControl 还是 TCustomControl 继承类时，可以从 TCustomControl 开始。它有一个真正的 Paint 方法和一些其他的功能，在创建自己的控件时是大有用处的。如果想将现存的 Windows 控件打包存进 VCL（可视对象库）中，则应该从 TWinControl 开始。

大多数从 TWinControl 继承的 Delphi 控件都是创建中间的定制对象。TEdit 类的等级组织如下：

TwinControl
TcustomEdit
Tedit

TListBox 类的等级组织如下：

TwinControl
TcustomListBox
TlistBox

每个都有中间的 TCustomXXX 控件。

TCustomEdit 类和 TEdit 类之间的差别在于 TCustomEdit 不公开显示其属性而 TEdit 则显示其属性。TCustomEdit 提供一个创建自己的 TEdit 控件的机会，这个控件具有我们希望在属性编辑器中看到的属性。

下面是出现在 CONTROLS.PAS 或 CONTROLS.INT 中的对 TGraphicControl 和 TCustomControl 的说明：

```
TGraphicControl = class(TControl)
private
  FCanvas: TCanvas;
  procedure WMPaint(var Message: TWMPaint); message WM_PAINT;
protected
  procedure Paint; virtual;
  property Canvas: TCanvas read FCanvas;
public
  constructor Create(AOwner: TComponent); override;
  destructor Destroy; override;
end;
```

```
TCustomControl = class(TWinControl)
private
  FCanvas: TCanvas;
  procedure WMPaint(var Message: TWMPaint); message WM_PAINT;
protected
  procedure Paint; virtual;
  procedure PaintWindow(DC: HDC); override;
  property Canvas: TCanvas read FCanvas;
public
  constructor Create(AOwner: TComponent); override;
  destructor Destroy; override;
end;
```

这些对象相当简单。仅仅是在其父控件上加入了绘画功能。如果往回再走一步到 TControl 或 TWinControl 层，这将是一个非常巨大的对象。例如，对 TWinControl 的说明就有近 200 行（而且提醒一句，这还仅仅只是类型说明，而不是实现）。

控件创建者通常直接利用 VCL 资源进行工作，而不是使用在线帮助或文档。对于简单的工作而言，可以不需使用资源轻松地创建自己的控件。然而对于大的工程，则必须取得资源代码。所有的 Delphi 版本中在其文档目录下都带有 INT（中断例行子程序）文件，这些文件也很有帮助，但是没有什么可以代替资源的作用。

下面我们首先创建一个从 TGraphicControl 继承的控件，为上面介绍的内容作一个完整的例证。

我们的目的是创建一个可以显示数字的控件，类似于电子时钟上显示的数字形式。首先它要能够显示数字，其次还要能够显示一些在表示数值时经常使用的字符。然后我们还能够改变它的颜色、大小，并可以显示数字的阴影。程序的代码如下：

```
unit jfsnum;

interface

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

type
  Tjfsshadowstyle=(shdNone,shdWithDigit,shdAll);
  Tjfsnum = class(TGraphicControl)
  private
      FText:string;
      Fforecolor:Tcolor;
      FShadowColor:Tcolor;
```

```
        FShadowStyle:TjfsshadowStyle;
        FAutoSize:Boolean;
        FMaxLen:Integer;

        FOnChange:TNotifyEvent;

        Fgridwidth:integer;
        FDotPos:integer;

        NumWid:integer;
        NumHeight:integer;
        LineWid:integer;
        LineHeight:integer;

        procedure SetText(value:string);
        procedure SetForecolor(value:Tcolor);
        procedure setshadowcolor(value:Tcolor);
        procedure setshadowStyle(value:TjfsshadowStyle);
        procedure SetAutoSize(value:boolean);
        procedure SetMaxLen(Value:integer);

        Function MyLen(str:string):integer;
            { Private declarations }
    protected

        procedure Change; dynamic;
      { Protected declarations }

    public
        procedure Paint;override;
        procedure ReSize;override;
        constructor Create(Aowner:TComponent);override;

      { Public declarations }
    published

        property Text:string read FText write SetText;
        property ForeColor:tcolor read Fforecolor write SetForecolor;
        property ShadowColor:Tcolor read FShadowcolor write setshadowcolor;
        property ShadowStyle:TjfsshadowStyle read FshadowStyle
write setshadowStyle;
        property OnChange:TNotifyEvent read FOnChange write FOnChange;
```

```
        property AutoSize: Boolean read FAutoSize Write SetAutoSize;
        property MaxLen: integer read FMaxLen write SetMaxlen;
    end;
procedure Register;

implementation

procedure Register;
begin
   RegisterComponents('WokControl', [Tjfsnum]);
end;
constructor Tjfsnum.Create(Aowner:TComponent);
begin
     inherited Create(AOwner);
     FText:='0';
     Fdotpos:=0;
     Fforecolor:=cllime;
     Fshadowcolor:=clgreen;
     FshadowStyle:=shdNone;
     Height := 100;
end;

procedure Tjfsnum.SetForecolor(value:Tcolor);
begin
      if Fforecolor<>value then
      begin
          Fforecolor:=value;
          invalidate();
      end;
end;

procedure Tjfsnum.Setshadowcolor(value:Tcolor);
begin
      if Fshadowcolor<>value then
      begin
          Fshadowcolor:=value;
          invalidate();
      end;
end;
procedure Tjfsnum.SetshadowStyle(value:Tjfsshadowstyle);
begin
      if FshadowStyle<>value then
```

```
      begin
          FshadowStyle:=value;
          invalidate();
      end;
end;

procedure Tjfsnum.SetText(value:string);
begin
    if FText<>value then
    begin
        FText:=value;
        invalidate();
        change;
    end;
end;

procedure Tjfsnum.ReSize;

begin
      inherited resize;
      invalidate();
end;
procedure Tjfsnum.Paint;
var i,ii,pos:integer;
var xx,yy,maxl:integer;
var iii:integer;
begin

      NumHeight := height;
      numwid:= trunc(numheight / 2);
      linewid:= trunc(numheight / 11);
      if linewid<1 then
          linewid:=1;
      Fgridwidth:=4*linewid+numwid;
      lineheight:= trunc(numheight / 2-1);

    iii:=trunc(width / fgridwidth);
    maxl := Mylen(FText);
    if Maxlen > 0 then
        if maxl > maxlen then
            Maxl := MaxLen;
    if AutoSize then
```

```
      Width := FgridWidth * Maxl + 6;
  yy:= 0;
  xx:=width;

inherited paint;
   canvas.Pen.color:=clblack;
   canvas.brush.color:=clblack;
case Fshadowstyle of
   shdNone:
       iii:=0;
   shdWithDigit:
       if length(FText)<iii then
           iii:=length(FText);
   shdAll:
           iii:=iii;
end;
canvas.Pen.color:=FShadowColor;
canvas.brush.color:=FShadowColor;
for pos:=1 to iii do
with canvas do
begin
   for ii:=0 to linewid do {top}
   begin
       moveto (xx-pos*fgridwidth+ii+1,yy+ii);
       lineto(xx-pos*fgridwidth+numwid-ii-1,yy+ii)
   end;
   for ii:=0 to linewid do     {right top}
   begin
       moveto (xx-pos*fgridwidth+numwid-ii,trunc(yy+lineheight / 10+ii));
       lineto(xx-pos*fgridwidth+numwid-ii,
                       trunc(yy+19*lineheight / 20-ii));
   end;
   for ii:=0 to linewid do     {bottom}
   begin
       moveto(xx-pos*fgridwidth+ii+1,yy+2*lineheight -ii);
       lineto(xx-pos*fgridwidth+numwid-ii-1,yy+2*lineheight-ii);
   end;
   for ii:=0 to linewid do    {rigth bottom}
   begin
       moveto(xx-pos*fgridwidth+numwid-ii,
                          trunc(yy+21*lineheight / 20+ii));
       lineto(xx-pos*fgridwidth+numwid-ii,
```

```
                        trunc(yy+19*lineheight / 10-ii));
    end;
    for ii:=0 to trunc(linewid / 2) do {center}
    begin
        moveto (xx-pos*fgridwidth+ii+1,yy+ii+lineheight );
        lineto (xx-pos*fgridwidth-ii-2+numwid,yy+ii+lineheight);
        moveto (xx-pos*fgridwidth+ii+1,yy-ii+lineheight );
        lineto (xx-pos*fgridwidth-ii-2+numwid,yy-ii+lineheight );
    end;
    for ii:=0 to linewid do       {left top}
    begin
        moveto (xx-pos*fgridwidth+ii,trunc(yy+ii+lineheight / 10));
        lineto(xx-pos*fgridwidth+ii,trunc(yy+19*lineheight / 20-ii));
    end;
    for ii:=0 to linewid do        {left bottom}
    begin
        moveto(xx-pos*fgridwidth+ii,trunc(yy+21*lineheight / 20+ii));
        lineto(xx-pos*fgridwidth+ii,trunc(yy+19*lineheight / 10-ii));
    end;
end;
canvas.pen.color:=Fforecolor;
canvas.Brush.color:=Fforecolor;
pos:=Maxl;

i:=1;
if (Maxlen < Mylen(Ftext)) and (MaxLen > 0) then
    i:=i+Mylen(Ftext)-Maxlen;
while pos >= 1 do
begin
if (xx-pos*Fgridwidth)<0 then
begin
          pos:=pos-1;
          i:=i+1;
          continue;
end;
case ord(FText[i]) of
    48:
        with canvas do
        begin
            for ii:=0 to linewid do {top}
            begin
                moveto (xx-pos*fgridwidth+ii+1,yy+ii);
```

```
            lineto(xx-pos*fgridwidth+numwid-ii-1,yy+ii)
        end;
        for ii:=0 to linewid do     {right top}
        begin
          moveto (xx-pos*fgridwidth+numwid-ii,trunc(yy+lineheight / 10+ii));
         lineto(xx-pos*fgridwidth+numwid-ii,trunc(yy+19*lineheight/20-ii));
        end;
        for ii:=0 to linewid do    {rigth bottom}
        begin
        moveto(xx-pos*fgridwidth+numwid-ii,
                    trunc(yy+21*lineheight / 20+ii));
        lineto(xx-pos*fgridwidth+numwid-ii,
                    trunc(yy+19*lineheight / 10-ii));
        end;
        for ii:=0 to linewid do     {bottom}
        begin
            moveto(xx-pos*fgridwidth+ii+1,yy+2*lineheight-ii);
            lineto(xx-pos*fgridwidth+numwid-ii-1,yy+2*lineheight -ii);
        end;
        for ii:=0 to linewid do       {left top}
        begin
            moveto (xx-pos*fgridwidth+ii,trunc(yy+ii+lineheight / 10));
            lineto(xx-pos*fgridwidth+ii,trunc(yy+19*lineheight / 20-ii));
        end;
        for ii:=0 to linewid do        {left bottom}
        begin
            moveto(xx-pos*fgridwidth+ii,trunc(yy+21*lineheight / 20+ii));
            lineto(xx-pos*fgridwidth+ii,trunc(yy+19*lineheight / 10-ii));
        end;
        pos:=pos-1;
    end;
49:
    with canvas do
    begin
      for ii:=0 to linewid do     {right top}
      begin
        moveto(xx-pos*fgridwidth+numwid-ii,trunc(yy+lineheight / 10+ii));
       lineto(xx-pos*fgridwidth+numwid-ii,
                    trunc(yy+19*lineheight/ 10-ii));
       end;
       for ii:=0 to linewid do    {rigth bottom}
       begin
```

```
            moveto(xx-pos*fgridwidth+numwid-ii,
                        trunc(yy+21*lineheight/ 20+ii));
            lineto(xx-pos*fgridwidth+numwid-ii,trunc(yy+19*lineheight/10-ii));
        end;
        pos:=pos-1;
    end;
50:
    with canvas do
    begin
        for ii:=0 to linewid do {top}
        begin
            moveto (xx-pos*fgridwidth+ii+1,yy+ii);
            lineto(xx-pos*fgridwidth+numwid-ii-1,yy+ii)
        end;
        for ii:=0 to linewid do     {right top}
        begin
          moveto (xx-pos*fgridwidth+numwid-ii,trunc(yy+lineheight / 10+ii));
          lineto(xx-pos*fgridwidth+numwid-ii,
                    Trunc(yy+19*lineheight / 20-ii));
        end;
        for ii:=0 to linewid do     {bottom}
        begin
            moveto(xx-pos*fgridwidth+ii+1,yy+2*lineheight -ii);
            lineto(xx-pos*fgridwidth+numwid-ii-1,yy+2*lineheight-ii);
        end;
        for ii:=0 to linewid do         {left bottom}
        begin
            moveto(xx-pos*fgridwidth+ii,trunc(yy+21*lineheight / 20+ii));
            lineto(xx-pos*fgridwidth+ii,trunc(yy+19*lineheight / 10-ii));
        end;
        for ii:=0 to Trunc(linewid / 2) do    {center}
        begin
            moveto (xx-pos*fgridwidth+ii+1,yy+ii+lineheight );
            lineto (xx-pos*fgridwidth-ii-2+numwid,yy+ii+lineheight);
            moveto (xx-pos*fgridwidth+ii+1,yy-ii+lineheight );
            lineto (xx-pos*fgridwidth-ii-2+numwid,yy-ii+lineheight );
        end;
        pos:=pos-1;
    end;

51:
    with canvas do
```

```
    begin
      for ii:=0 to linewid do {top}
      begin
          moveto (xx-pos*fgridwidth+ii+1,yy+ii);
          lineto(xx-pos*fgridwidth+numwid-ii-1,yy+ii)
      end;
      for ii:=0 to linewid do     {right top}
      begin
       moveto (xx-pos*fgridwidth+numwid-ii,trunc(yy+lineheight / 10+ii));
       lineto(xx-pos*fgridwidth+numwid-ii,
            Trunc(yy+19*lineheight / 20-ii));
      end;
      for ii:=0 to linewid do     {bottom}
      begin
          moveto(xx-pos*fgridwidth+ii+1,yy+2*lineheight -ii);
          lineto(xx-pos*fgridwidth+numwid-ii-1,yy+2*lineheight-ii);
      end;
      for ii:=0 to linewid do    {rigth bottom}
      begin
       moveto(xx-pos*fgridwidth+numwid-ii,
                  trunc(yy+21*lineheight / 20+ii));
       lineto(xx-pos*fgridwidth+numwid-ii,
                  trunc(yy+19*lineheight / 10-ii));
      end;
      for ii:=0 to trunc(linewid / 2) do {center}
      begin
          moveto (xx-pos*fgridwidth+ii+1,yy+ii+lineheight );
          lineto (xx-pos*fgridwidth-ii-2+numwid,yy+ii+lineheight);
          moveto (xx-pos*fgridwidth+ii+1,yy-ii+lineheight );
          lineto (xx-pos*fgridwidth-ii-2+numwid,yy-ii+lineheight );
      end;
      pos:=pos-1;
    end;

52:
    with canvas do
    begin
      for ii:=0 to linewid do     {right top}
      begin
       moveto (xx-pos*fgridwidth+numwid-ii,trunc(yy+lineheight / 10+ii));
       lineto(xx-pos*fgridwidth+numwid-ii,
                  trunc(yy+19*lineheight / 20-ii));
```

```
        end;
        for ii:=0 to Trunc(linewid / 2) do {center}
        begin
            moveto (xx-pos*fgridwidth+ii+1,yy+ii+lineheight );
            lineto (xx-pos*fgridwidth-ii-2+numwid,yy+ii+lineheight);
            moveto (xx-pos*fgridwidth+ii+1,yy-ii+lineheight );
            lineto (xx-pos*fgridwidth-ii-2+numwid,yy-ii+lineheight );
        end;
        for ii:=0 to linewid do       {left top}
        begin
            moveto (xx-pos*fgridwidth+ii,trunc(yy+ii+lineheight / 10));
            lineto(xx-pos*fgridwidth+ii,trunc(yy+19*lineheight / 20-ii));
        end;
        for ii:=0 to linewid do    {rigth bottom}
        begin
          moveto(xx-pos*fgridwidth+numwid-ii,
                      trunc(yy+21*lineheight / 20+ii));
          lineto(xx-pos*fgridwidth+numwid-ii,
                      trunc(yy+19*lineheight / 10-ii));
        end;
        pos:=pos-1;
    end;

53:
    with canvas do
    begin
        for ii:=0 to linewid do {top}
        begin
            moveto (xx-pos*fgridwidth+ii+1,yy+ii);
            lineto(xx-pos*fgridwidth+numwid-ii-1,yy+ii)
        end;
        for ii:=0 to linewid do     {bottom}
        begin
            moveto(xx-pos*fgridwidth+ii+1,yy+2*lineheight -ii);
            lineto(xx-pos*fgridwidth+numwid-ii-1,yy+2*lineheight-ii);
        end;
        for ii:=0 to linewid do    {rigth bottom}
        begin
          moveto(xx-pos*fgridwidth+numwid-ii,
                      trunc(yy+21*lineheight / 20+ii));
          lineto(xx-pos*fgridwidth+numwid-ii,
                      trunc(yy+19*lineheight / 10-ii));
```

```
        end;
        for ii:=0 to trunc(linewid / 2) do {center}
        begin
            moveto (xx-pos*fgridwidth+ii+1,yy+ii+lineheight );
            lineto (xx-pos*fgridwidth-ii-2+numwid,yy+ii+lineheight);
            moveto (xx-pos*fgridwidth+ii+1,yy-ii+lineheight );
            lineto (xx-pos*fgridwidth-ii-2+numwid,yy-ii+lineheight );
        end;
        for ii:=0 to linewid do       {left top}
        begin
            moveto (xx-pos*fgridwidth+ii,trunc(yy+ii+lineheight / 10));
            lineto(xx-pos*fgridwidth+ii,trunc(yy+19*lineheight / 20-ii));
        end;
        pos:=pos-1;
    end;

54:
    with canvas do
    begin
        for ii:=0 to linewid do {top}
        begin
            moveto (xx-pos*fgridwidth+ii+1,yy+ii);
            lineto(xx-pos*fgridwidth+numwid-ii-1,yy+ii)
        end;
        for ii:=0 to linewid do     {bottom}
        begin
            moveto(xx-pos*fgridwidth+ii+1,yy+2*lineheight -ii);
            lineto(xx-pos*fgridwidth+numwid-ii-1,yy+2*lineheight-ii);
        end;
        for ii:=0 to Trunc(linewid / 2) do {center}
        begin
            moveto (xx-pos*fgridwidth+ii+1,yy+ii+lineheight );
            lineto (xx-pos*fgridwidth-ii-2+numwid,yy+ii+lineheight);
            moveto (xx-pos*fgridwidth+ii+1,yy-ii+lineheight );
            lineto (xx-pos*fgridwidth-ii-2+numwid,yy-ii+lineheight );
        end;
        for ii:=0 to linewid do       {left top}
        begin
            moveto (xx-pos*fgridwidth+ii,trunc(yy+ii+lineheight / 10));
            lineto(xx-pos*fgridwidth+ii,trunc(yy+19*lineheight / 20-ii));
        end;
        for ii:=0 to linewid do        {left bottom}
```

```
        begin
            moveto(xx-pos*fgridwidth+ii,trunc(yy+21*lineheight / 20+ii));
            lineto(xx-pos*fgridwidth+ii,trunc(yy+19*lineheight / 10-ii));
        end;
        for ii:=0 to linewid do    {rigth bottom}
        begin
          moveto(xx-pos*fgridwidth+numwid-ii,
                    trunc(yy+21*lineheight / 20+ii));
          lineto(xx-pos*fgridwidth+numwid-ii,
                    trunc(yy+19*lineheight / 10-ii));
        end;
        pos:=pos-1;
    end;

55:
    with canvas do
    begin
        for ii:=0 to linewid do {top}
        begin
            moveto (xx-pos*fgridwidth+ii+1,yy+ii);
            lineto(xx-pos*fgridwidth+numwid-ii-1,yy+ii)
        end;
        for ii:=0 to linewid do     {right top}
        begin
          moveto (xx-pos*fgridwidth+numwid-ii,trunc(yy+lineheight / 10+ii));
          lineto(xx-pos*fgridwidth+numwid-ii,
                    trunc(yy+19*lineheight / 20-ii));
        end;
        for ii:=0 to linewid do    {rigth bottom}
        begin
          moveto(xx-pos*fgridwidth+numwid-ii,
                    trunc(yy+21*lineheight / 20+ii));
          lineto(xx-pos*fgridwidth+numwid-ii,
                    trunc(yy+19*lineheight / 10-ii));
        end;
        pos:=pos-1;
    end;

56:
    with canvas do
    begin
        for ii:=0 to linewid do {top}
```

```
    begin
        moveto (xx-pos*fgridwidth+ii+1,yy+ii);
        lineto(xx-pos*fgridwidth+numwid-ii-1,yy+ii)
    end;
    for ii:=0 to linewid do     {right top}
    begin
      moveto (xx-pos*fgridwidth+numwid-ii,trunc(yy+lineheight / 10+ii));
      lineto(xx-pos*fgridwidth+numwid-ii,
                trunc(yy+19*lineheight / 20-ii));
    end;
    for ii:=0 to linewid do     {bottom}
    begin
        moveto(xx-pos*fgridwidth+ii+1,yy+2*lineheight -ii);
        lineto(xx-pos*fgridwidth+numwid-ii-1,yy+2*lineheight-ii);
    end;
    for ii:=0 to linewid do   {rigth bottom}
    begin
      moveto(xx-pos*fgridwidth+numwid-ii,
                trunc(yy+21*lineheight / 20+ii));
      lineto(xx-pos*fgridwidth+numwid-ii,
                trunc(yy+19*lineheight / 10-ii));
    end;
    for ii:=0 to trunc(linewid / 2) do {center}
    begin
        moveto (xx-pos*fgridwidth+ii+1,yy+ii+lineheight );
        lineto (xx-pos*fgridwidth-ii-2+numwid,yy+ii+lineheight);
        moveto (xx-pos*fgridwidth+ii+1,yy-ii+lineheight );
        lineto (xx-pos*fgridwidth-ii-2+numwid,yy-ii+lineheight );
    end;
    for ii:=0 to linewid do       {left top}
    begin
        moveto (xx-pos*fgridwidth+ii,trunc(yy+ii+lineheight / 10));
        lineto(xx-pos*fgridwidth+ii,trunc(yy+19*lineheight / 20-ii));
    end;
    for ii:=0 to linewid do        {left bottom}
    begin
        moveto(xx-pos*fgridwidth+ii,trunc(yy+21*lineheight / 20+ii));
        lineto(xx-pos*fgridwidth+ii,trunc(yy+19*lineheight / 10-ii));
    end;
    pos:=pos-1;
end;
```

```
57:
    with canvas do
    begin
        for ii:=0 to linewid do {top}
        begin
            moveto (xx-pos*fgridwidth+ii+1,yy+ii);
            lineto(xx-pos*fgridwidth+numwid-ii-1,yy+ii)
        end;
        for ii:=0 to linewid do     {right top}
        begin
          moveto (xx-pos*fgridwidth+numwid-ii,trunc(yy+lineheight / 10+ii));
          lineto(xx-pos*fgridwidth+numwid-ii,
                    trunc(yy+19*lineheight / 20-ii));
        end;
        for ii:=0 to linewid do     {bottom}
        begin
            moveto(xx-pos*fgridwidth+ii+1,yy+2*lineheight -ii);
            lineto(xx-pos*fgridwidth+numwid-ii-1,yy+2*lineheight-ii);
        end;
        for ii:=0 to linewid do    {rigth bottom}
        begin
          moveto(xx-pos*fgridwidth+numwid-ii,
                    trunc(yy+21*lineheight / 20+ii));
          lineto(xx-pos*fgridwidth+numwid-ii,
                    trunc(yy+19*lineheight / 10-ii));
        end;
        for ii:=0 to trunc(linewid / 2) do {center}
        begin
            moveto (xx-pos*fgridwidth+ii+1,yy+ii+lineheight );
            lineto (xx-pos*fgridwidth-ii-2+numwid,yy+ii+lineheight);
            moveto (xx-pos*fgridwidth+ii+1,yy-ii+lineheight );
            lineto (xx-pos*fgridwidth-ii-2+numwid,yy-ii+lineheight );
        end;
        for ii:=0 to linewid do       {left top}
        begin
            moveto (xx-pos*fgridwidth+ii,trunc(yy+ii+lineheight / 10));
            lineto(xx-pos*fgridwidth+ii,trunc(yy+19*lineheight / 20-ii));
        end;
        pos:=pos-1;
    end;

ord('.'):
```

```
        canvas.RoundRect(trunc(xx+numwid+2*linewid
                               / 3-(pos+1)*Fgridwidth),
                         trunc(yy+numheight-7*linewid / 3),
                         trunc(xx+numwid+7*linewid / 3-(pos+1)*Fgridwidth),
                         trunc(yy+numheight-2*linewid / 3),
                         trunc(linewid / 2),trunc(linewid / 2));

ord('-'):
    with canvas do
    begin
        for ii:=0 to trunc(linewid / 2) do {center}
        begin
            moveto (xx-pos*fgridwidth+ii+1,yy+ii+lineheight );
            lineto (xx-pos*fgridwidth-ii-2+numwid,yy+ii+lineheight);
            moveto (xx-pos*fgridwidth+ii+1,yy-ii+lineheight );
            lineto (xx-pos*fgridwidth-ii-2+numwid,yy-ii+lineheight );
        end;
            pos:=pos-1;
    end;
ord('E'),ord('e'):
    with canvas do
    begin
        for ii:=0 to trunc(linewid / 2) do {center}
        begin
            moveto (xx-pos*fgridwidth+ii+1,yy+ii+lineheight );
            lineto (xx-pos*fgridwidth-ii-2+numwid,yy+ii+lineheight);
            moveto (xx-pos*fgridwidth+ii+1,yy-ii+lineheight );
            lineto (xx-pos*fgridwidth-ii-2+numwid,yy-ii+lineheight );
        end;
        for ii:=0 to linewid do {top}
        begin
            moveto (xx-pos*fgridwidth+ii+1,yy+ii);
            lineto(xx-pos*fgridwidth+numwid-ii-1,yy+ii)
        end;
        for ii:=0 to linewid do     {bottom}
        begin
            moveto(xx-pos*fgridwidth+ii+1,yy+2*lineheight -ii);
            lineto(xx-pos*fgridwidth+numwid-ii-1,yy+2*lineheight-ii);
        end;
        for ii:=0 to linewid do      {left top}
        begin
            moveto (xx-pos*fgridwidth+ii,trunc(yy+ii+lineheight / 10));
```

```
                lineto(xx-pos*fgridwidth+ii,trunc(yy+19*lineheight / 20-ii));
            end;
            for ii:=0 to linewid do        {left bottom}
            begin
                moveto(xx-pos*fgridwidth+ii,trunc(yy+21*lineheight / 20+ii));
                lineto(xx-pos*fgridwidth+ii,trunc(yy+19*lineheight / 10-ii));
            end;
            pos:=pos-1;
        end;
      ord(':'):
        with canvas do
        begin
            RoundRect(xx+numwid+linewid-(pos+1)*Fgridwidth,
                      yy+numheight-3*linewid,
                      xx+numwid+3*linewid-(pos+1)*Fgridwidth,
                      yy+numheight-5*linewid,
                      trunc(linewid / 2),trunc(linewid / 2));
            RoundRect(xx+numwid+linewid-(pos+1)*Fgridwidth,
                      yy+3*linewid,
                      xx+numwid+3*linewid-(pos+1)*Fgridwidth,
                      yy+5*linewid,
                      trunc(linewid / 2),trunc(linewid / 2));
        end;

     end;
     i:=i+1;
  end;
end;
procedure Tjfsnum.Change;
begin
    inherited changed;
    if Assigned(FOnChange) then FOnChange(Self);
end;
procedure Tjfsnum.SetAutoSize(value: boolean);
begin
    if value <> FAutoSize then
    begin
        FAutoSize := value;
        invalidate();
    end;
end;
```

```
procedure Tjfsnum.SetMaxLen(Value: integer);
begin
    if value <> FMaxlen then
    begin
        FMaxLen := value;
        invalidate();
    end;
end;

function Tjfsnum.MyLen(str: string): integer;
var i,j:integer;
begin
     j:=0;
     for i:=0 to length(str) -1 do
         if (str[i] <> '.') and (str[i] <> ':') then
             j:=j+1;
     result := j;
end;

end.
```

这段代码可以说比较长，但是，如果仔细研究一下这段程序，就会发现这些代码并不难于理解。这段程序中有许多用于计算位置的代码，利用它们可以知道应该在什么地方画什么样的内容。关于这些具体的代码我们就不再详细介绍了，稍加研究就可以发现其中的奥妙。

但是，这段代码还是有几个比较有特色的地方需要进行详细说明。

首先要说明的是，在修改一些控制控件的外观属性时，在定义的内部字段访问方法中，不仅定义了字段的赋值，而且调用了原始的重画函数，使得能够重新绘制控件的外观。例如在设置前景颜色的时候，采用了下面的代码：

```
procedure Tjfsnum.SetForecolor(value:Tcolor);
begin
     if Fforecolor<>value then
     begin
         Fforecolor:=value;
         invalidate();
     end;
end;
```

注意：

在定义写方法的时候，进行判断是非常必要的。如果不进行判断便进行赋值，有可能会造成死循环。

要说明的第二个问题是，我们也为这个控件定义了一个 OnChange 事件。很简单，由于这个控件的目的是用来显示数字，那么在实际的应用过程中，有可能需要根据其中显示的数字的变化进行一定的操作。但是，由于我们继承的是 TGraphicControl，所以它不具有句柄，而且这个抽象类没有 OnChange 事件。那么怎样来处理这个问题呢？由于所谓的变化只是在这个控件中显示的文本进行变化时存在，所以，我们在定义属性 Text 的写方法时调用了事件调度函数，如下所示：

```
procedure Tjfsnum.SetText(value:string);
var i,ii:integer;
var dd:pchar;
begin
    if FText<>value then
    begin
        FdotPos:=0;
        ii:=length(value);
        dd:=pchar(value);
        for i:=1 to ii do
        begin
          if dd[i-1]='.' then
                Fdotpos:=ii-i;
        end;
        FText:=value;
        invalidate();
        change;
       end;
end;
```

这样，当这个控件中的数字发生了变化时，也会响应我们定义的 OnChange 事件。显然这是一个投机取巧的办法，但是这样做十分有效，使得一个 TGraphicControl 也能响应 OnChange 事件了。而且这个事件的形式和普通的我们所熟悉的 OnChange 事件是完全相同的。

在测试成功之后，便可以把这个控件添加到 Delphi 开发环境中，在开始演示它的使用之前，我们还想顺便介绍一下如何为我们自己创建的控件制作图标。这里所谓的图标是指显示在 Delphi 的控件选项板上的对应控件的图标。

与控件相联系的图标放置在控件选项板中，它们定义在一个具有.dcr 扩展名的文件中。如果用户没有提供该文件，Delphi 使用与对象的父类相联系的图标。如果在控件父类或父类以上的类中的任何地方都没有图标，那么就会使用默认的图标。例如在图 10.10 所显示的控件选项板中便显示了一个默认的图标。

图 10.10　控件选项板上默认的控件图标

注意：

DCR 文件是将扩展名从.res 改变为.dcr 的 Windows 资源文件。该资源文件包括与用户的控件有相同名字的位图资源。例如，TColor 控件在 DCR 文件中的位图资源会有一个名为 TCOLOR 的 ID 号。该资源有一个 28×28 像素（或者更小）的位图，并且该位图可以在 Image Editor 中加以编辑。用户需要做的工作就是将该 DCR 文件放置在与用户的控件相同的目录下。使用 Image Editor 可以来研究与 Delphi 一起出售的 DCR 文件，它们存储在\DELPHI\LIB 目录下。

为了将我们自己的位图与一个特定的控件联系起来，我们必须完成如下的步骤。

(1) 打开 Image Editor，并选择 New | Component Resource File，如图 10.11 所示。

图 10.11　Image Editor 应用程序的界面

(2) 选择 Resource 菜单中的 New，并从子菜单中选择 Bitmap，此时会出现一个如图 10.12 所示的对话框。

(3) 将 Colors 设置为 256 色，因为现在该技术已经可以在几乎所有的 Windows 系统中使用。将 Size 设置成 28×28，或者一个更小的，例如 24×24 的大小。单击 OK 按钮。

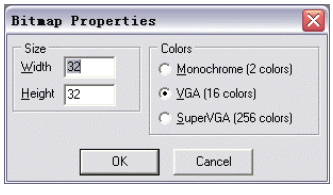

图 10.12　添加一个新位图

(4) 此时会在应用程序的一个窗口中显示包含了一个位图。把这个位图的名称修改为我们使用的控件类名称，例如在这里是 TJFSNUM，注意这里全是大写。

(5) 然后双击该名称，便进入位图编辑界面。使用编辑器的缩放功能放大位图以便于我们可以看得更清楚。现在请画一些图形或者图片。例如我们创作了一个如图 10.13 所示的

图标。

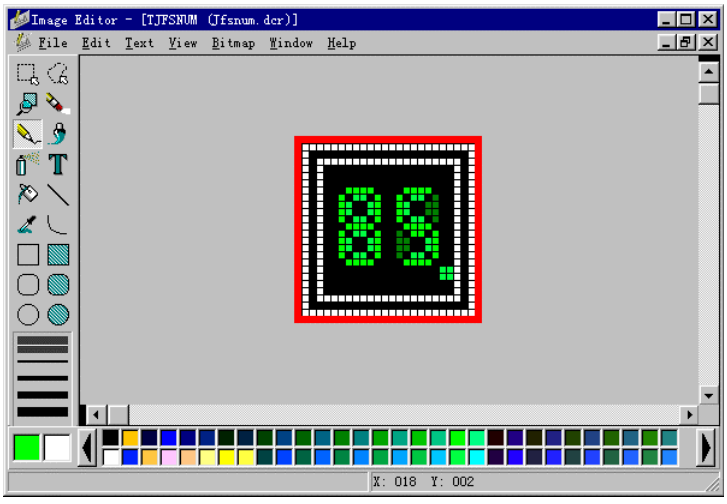

图 10.13 创建的新图标

(6) 关闭 Image Editor 窗口并以 JfsNum.DCR 的名字存储文件。

如果你不喜欢使用 Image Editor，也可以在 Windows 的画笔应用程序或者其他的图形编辑器里创建一幅 28×28 的位图并且像如下这样来创建 RC 文件：

```
TJFSNUM BITMAP JfsNum.bmp
```

将文件以 JfsNum.RC 的名字保存。从命令行来运行 Resource Compiler（资源编译器）：

```
brc - r JfsNum.rc
```

所得到的文件称为 JfsNum.RES。将该文件的名字改变为 JfsNum.DCR。

注意要把这个 DCR 文件放置在和控件的单元文件同一个位置上。然后把这个文件加入到我们的包文件中，进行编译之后，便把这个图标应用到了控件选项板上。例如在图 10.14 所示的控件选项板上便显示了我们新创建的图标。

图 10.14 显示在控件选项板上的新图标

下面我们就建立一个测试程序来测试这个控件，包括控件的外观和 OnChange 事件，程序的代码如下所示：

```
procedure TForm1.jfsnum1Change(Sender: TObject);
begin
```

```
Jfsnum2.Text := Inttostr(strtoint(jfsnum2.Text) + 1 mod 10000);

end;

procedure TForm1.jfsnum3Change(Sender: TObject);
begin
JfsNum1.Text := floattostr(strtofloat(Jfsnum1.Text) + 3.1415926 / 100);
end;

procedure TForm1.Timer1Timer(Sender: TObject);
begin
Jfsnum3.Text := Inttostr(strtoint(jfsnum3.Text) + 1 mod 10);
end;

end.
```

在上面的应用程序中，使用了一个 Timer 控件来控制其中一个控件的行为，然后利用这个控件的 OnChange 事件来影响其他两个控件的数字。程序的运行结果如图 10.15 所示。

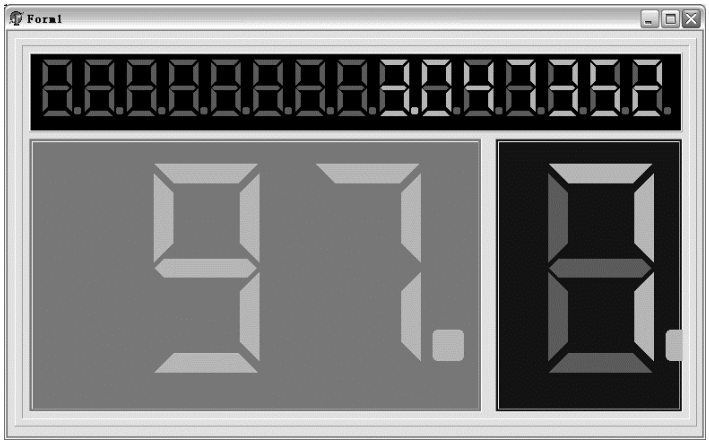

图 10.15　新创建的数字控件的测试程序

从上面的程序中，我们也可以看出，如果把该控件的 Text 属性声明成 Double 型的属性可能会更方便。但是，如果是为了用这个控件来显示所有的英文字符，那么这样的声明显然是不合适的。

10.4.2　Delphi 中的五类 API 工具函数

用户可以使用五类主要的 API 工具函数，这些 API 使用户能够编写可以与 Delphi 的集成开发环境直接相连的代码。举例来说，用户可以将用户的工具连接到包中，也可以用同样

的方法将它们与控件相连。表 10.1 列出了 API 工具函数以及定义它们的 Delphi 的源程序文件。

表 10.1 Delphi API 工具函数

API 工具函数	描述/文件
专业	使用户能够编写自己的专业程序
	EXPINTF.PAS
	VIRTINTF.PAS
	TOOLINTF.PAS (列举控件页和安装的控件，并且向工程增加模块，等等)
版本控件	使用户能够编写自己的版本控件系统或者与第三方系统相连
	VCSINTF.PAS
	VIRTINTF.PAS
	TOOLINTF.PAS（打开或关闭编辑器中的文件）
控件编辑器	创建设计时与控件相连的对话框（例如，DataSet Designer 就是一个控件编辑器）
	DSGNINTF.PAS
属性编辑器	创建能够在属性编辑器中使用的编辑器
	DSGNINTF.PAS
编辑器接口	允许第三方使用 Delphi 的编辑器
	EDITINTF.PAS
	FILEINTF.PAS

字母缩写 INTF 是单词 interface（接口）的缩写。API 工具函数是在用户自己的代码和 Delphi 开发者代码之间的一个接口。

绝大多数的人没有使用过这些 API 工具函数。然而，对于少数的开发者来说它是非常重要的，并且它的存在意味着每个人都可以购买或者下载扩展 IDE 功能的工具。

10.4.3 属性编辑器

用于创建属性编辑器的 API 工具函数可能是最常用的通向 IDE 核心的接口。当你第一次使用 Delphi 并且开始熟悉属性编辑器的时候，你一定会以为它是一个静态的元素，从来都不会改变。然而，你可以通过给它增加新的属性编辑器来改变属性编辑器的功能。

就像前面所说的那样，属性编辑器控制着属性编辑器中属性页右边所发生的事情。例如，当用户点一下 TEdit 的 Color 属性时，用户可以从一个下拉式列表中，或从一个普通的对话框中，或者通过键入一个新的数值来选择一种新的颜色。在所有这三种情况下，用户都是在使用一个属性编辑器。

如果我们想要创建一个新的属性编辑器，应该创建一个 TPropertyEditor 的派生类，该类在 VCLEditors.PAS 中加以声明。下面的代码就是对与 TJFSColorClock 控件相联系的属性编辑器的声明：

```
TJFSColorNameProperty = class(TColorProperty)
public
  function GetAttributes: TPropertyAttributes; override;
  procedure Edit; override;
end;
```

VCLEditors 单元开发者非常仔细地编写了该单元的文档，在这一方面它是十分不同寻常的。例如，下面选录的代码描述了在 TPropertyEditor 的派生类中所调用两种方法。

- Edit：在属性编辑器上的省略号按钮按下或者该属性被双击的时候调用。例如，这可以调出一个允许用比文本方式更容易理解的方式来编辑控件的对话框（例如 Font 属性）。
- GetAttributes：返回在属性编辑器中使用的信息，以便能够显示适当的工具。

我们不必要把所有可以使用的方法都列举出来，因为最常用的也就是上面的两个方法。但是我们必须提到，这些条目是 Delphi 开发者所编写的，它们大量地记录了与集成开发环境核心处的核心代码相联系的接口。对于高级程序员来说，我们还是建议多去读一读这些文档，对开发商业用的软件是非常有帮助的。下面的代码是 Edit 和 GetAttributes 的声明，还有 TPropertyEditor 中其他一些关键的功能：

```
TPropertyEditor = class
  public
  destructor Destroy; override;
  procedure Activate; virtual;
  function AllEqual: Boolean; virtual;
  function AutoFill: Boolean; virtual;
  procedure Edit; virtual;
  function GetAttributes: TPropertyAttributes; virtual;
  function GetComponent(Index: Integer): TPersistent;
  function GetEditLimit: Integer; virtual;
  function GetName: string; virtual;
  procedure GetProperties(Proc: TGetPropEditProc); virtual;
  function GetPropType: PTypeInfo;
  function GetValue: string; virtual;
  procedure GetValues(Proc: TGetStrProc); virtual;
  procedure Initialize; virtual;
  procedure Revert;
  procedure SetValue(const Value: string); virtual;
  function ValueAvailable: Boolean;
  property Designer: IFormDesigner read FDesigner;
  property PrivateDirectory: string read GetPrivateDirectory;
  property PropCount: Integer read FPropCount;
```

```
    property Value: string read GetValue write SetValue;
end;
```

所有这些方法在 VCLEditors.PAS 文件中都做了详细的记录。如果你想要学习更多的关于如何创建复杂的属性编辑器的知识，就应该仔细地研究这个文件。

TPropertyEditor 中的 Edit 方法是我们想要覆盖的一个方法，以便于能够改变属性编辑器实际上编辑数据的方式。通常，我们会按照下面的结构来编写这样的一个方法：

```
procedure TColorNameProperty . Edit ;
var
      S ： String ;
begin
          S ： = ' ' ;
          InputQuery (' New Color ' , S ) ;
          SetValue ( S ) ;
end ;
```

在这种情况下，创建了一个替代 TColorDialog 的类，TColorDialog 在我们单击一下属性编辑器中的椭圆形图标的时候会弹出来。当然，我们可以用一个简单的对话框来替换 Windows 中特有的普通对话框，以便于输入一个字符串，例如“ clBlue ”，或者“ clGreen ”。然而，关键的问题是我们正在学习如何创建自己的属性编辑器。

在一个更复杂的例子中，用户可以打开一个窗体，该窗体允许用户对一个属性做更多的改变。这里我们创建了一个具有一个 TColorGrid 类、两个 BitBtn 的对话框，该窗体如图 10.10 所示。

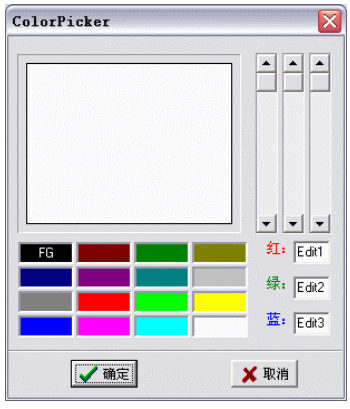

图 10.10　用于设置 TJFSNum 及其派生类的颜色属性的属性编辑器

该窗体对应的单元文件如下所示：

```
unit ColorPicker1;

interface
```

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

type
  TColorPicker = class(TForm)
    BitBtn1: TBitBtn;
    BitBtn2: TBitBtn;
    Bevel2: TBevel;
    ScrRed: TScrollBar;
    ScrGreen: TScrollBar;
    ScrBlue: TScrollBar;
    Edit1: TEdit;
    Edit2: TEdit;
    Edit3: TEdit;
    ColorGrid1: TColorGrid;
    Bevel1: TBevel;
    Shape1: TShape;
    Label1: TLabel;
    Label2: TLabel;
    Label3: TLabel;

    procedure BitBtn1Click(Sender: TObject);
    procedure ColorGrid1Change(Sender: TObject);
    procedure ScrRedChange(Sender: TObject);
    procedure ScrGreenChange(Sender: TObject);
    procedure ScrBlueChange(Sender: TObject);
  private
    FColorChoice: TColor;
    { Private declarations }
  public
    property ColorChoice: TColor read FColorChoice;
  end;

var
  ColorPicker: TColorPicker;

implementation
```

```
{$R *.DFM}

procedure TColorPicker.BitBtn1Click(Sender: TObject);
begin
  FColorChoice := shape1.Brush.color;//ColorGrid1.ForegroundColor;
end;

procedure TColorPicker.ColorGrid1Change(Sender: TObject);
begin
  Shape1.Brush.Color := ColorGrid1.ForegroundColor;
end;

procedure TColorPicker.ScrRedChange(Sender: TObject);
begin
    Shape1.Brush.Color := RGB(ScrRed.Position, ScrGreen.Position, ScrBlue.Position);
    Edit1.Text := inttostr(Scrred.Position);
end;

procedure TColorPicker.ScrGreenChange(Sender: TObject);
begin
    Shape1.Brush.Color := RGB(ScrRed.Position, ScrGreen.Position, ScrBlue.Position);
    Edit2.Text := inttostr(ScrGreen.Position);
end;

procedure TColorPicker.ScrBlueChange(Sender: TObject);
begin
    Shape1.Brush.Color := RGB(ScrRed.Position, ScrGreen.Position, ScrBlue.Position);
    Edit3.Text := inttostr(ScrBlue.Position);
end;

end.
```

这个对话框可能没有我们常见的颜色对话框那么花哨，但是它却是一个关于如何创建自己的定制控件编辑器的很好的例子。

生成该对话框的 TColorProperty .Edit **方法如下所示：**

```
procedure TJFSColorNameProperty.Edit;
var
  S: String;
  ColorPicker: TColorPicker;
begin
  S := '';
  ColorPicker := TColorPicker.Create(nil);
```

```
    ColorPicker.ShowModal;
    S := ColorToString(ColorPicker.ColorChoice);
    ColorPicker.Free;
    SetValue(S);
end;
```

代码创建了一个 TColorPicker **对话框的实例，将它显示给用户，然后使用用户所选择的颜色。**

该处理程序末尾所调用的 SetValue **函数是另一个** TPropertyEditor **方法。为了创建一个灵活的、多态的层次，**SetValue **方法要求将所编辑的值转化成字符串。所以，对于字符串、整数、浮点数，甚至是** TColor **属性所用的** SetValue **方法的声明是相同的。最基本的是，**SetValue **方法必须将字符串转化回整数、浮点数、**TColor **对象，或者任何用户所有的东西。在这种特殊的情况下，**Graphics.PAS VCL **单元中的** ColorToString **和** StringToColor **方法有助于简化这一工作。**

使用 GetAttributes **方法是一种定义与** TColorNameProperty **相联系的，用户所需要的属性编辑器类型的方法：**

```
function TJFSColorNameProperty.GetAttributes;
begin
    Result := [paMultiSelect, paValueList, paDialog];
end;
```

即使用户选择了多于一个的该类型的控件，一个具有 paMultiSelect **标志的属性编辑器仍然可以保持激活状态。例如，用户可以选择** 10 **个** Edit **控件，并且一次就改变它们所有的字体。**Delphi **之所以允许用户这样做，是因为** TEdits **有它们自己的** paMultiSelect **标志设置。**paValueList **标志决定了当用户点一下编辑器最右边的箭头按钮时，属性编辑器的下拉式列表中是一个列举类型还是一个设置类型。该功能是内嵌在** Delphi **内部的，我们只是需要设置该标志以便于用户的属性编辑器可以支持它。**

最后，paDialog **声明了属性编辑器弹出一个对话框。因为以前所展示的** Edit **功能或者使用** InputQuery **对话框，或者使用** TColorPicker **对话框，我们决定设置该标志。最终，**paDialog **标志只是保证省略号按钮显示在属性编辑器的右边。**

注意：

当用户在一个单一的控件里既选择 paDialog **标志，又选择** paValueList **标志时，属性编辑器按钮总是以组合下拉式列表按钮的形式来显示。换句话说，对话框的按钮是模糊的，虽然所有的功能都是起作用的。例如，请看** TForm **或者** TEdit **类中的** Color **属性。**

10.4.4 注册自定义属性编辑器

在编写了上面所介绍的这样一个属性编辑器之后，如果要把它使用在 Delphi 的集成开发环境中的话，必须把它注册到 Delphi 中。

在注册属性编辑器的时候，可以使用下面的代码：

```
procedure Register;
begin
…
  RegisterPropertyEditor(TypeInfo(TColor),
              TJFSNum, 'ForeColor',
              TJFSColorNameProperty);
  RegisterPropertyEditor(TypeInfo(TColor),
              TJFSNum, 'ShadowColor',
              TJFSColorNameProperty);
end;
```

RegisterPropertyEditor 的声明如下所示：

```
procedure RegisterPropertyEditor(PropertyType:PTypeInfo;
                    ComponentClass:TClass;
                    Const PropertyName:string;
                    EditorClass:TPropertyEditorClass);
```

下面描述了各种参数。

- PropertyType：第一个传递到该函数中去的参数，它说明了被编辑器所处理的数据的类型。在这种情况下，它是 TColor。Delphi 使用该信息作为一系列复选框列表中的第一项，该列表决定了哪些属性应该与该编辑器相联系。
- ComponentClass：第二个参数，它进一步限制了哪个控件可以使用该编辑器。在这种情况下，我们将范围限制在 TJFSClock 及其派生类上。如果编写的是 TComponent 而不是 TJFSColck，或者如果将该参数设置成 nil ，TColor 类型所有的属性都会使用该编辑器。所以，我们可以为字体或颜色创建一个新的编辑器，将它安装在一个客户系统中，这样它就会与该类型的所有属性一起工作。换句话说，为了给它编写一个编辑器，我们没有必要事先创建一个控件。
- PropertyName：第三个参数，它将范围限制在了该字符串所传递的名字上。如果该字符串是空的，编辑器就会为前两个参数所传递的所有属性所使用。
- EditorClass：该参数定义了与在前三个参数中所定义的编辑器的类。

10.4.5　控件编辑器

在介绍了如何创建属性编辑器之后，我们很容易就可以理解怎样创建控件编辑器了。就像属性编辑器是 TPropertyEditor 的派生类一样，这些工具是 TComponentEditor 的派生类：

```
TJFSControlEditor = class(TComponentEditor)
public
  procedure Edit;Override;
end;
```

当然，这个例子是可能的控件编辑器中最简单的情况，但它是我们开始学习这些有用工具的起点。

我们在这里编辑了一个如图 10.11 所示的定制窗体，利用这个窗体，可以同时为时钟控件的三个颜色属性进行赋值。

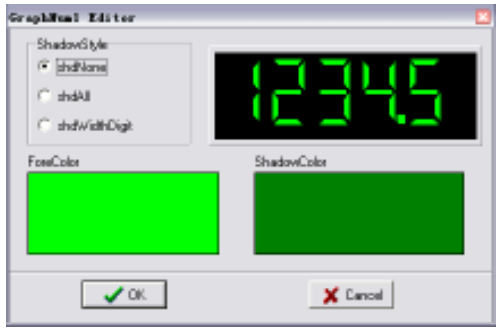

图 10.11　TJfsNum 控件的控件编辑器

这个窗体对应的单元文件如下，其中包含了如何在这个窗体中处理颜色：

```
unit GraphNumEditor;

interface

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

type
  TFrmCtrlEditor = class(TForm)
    Shape1: TShape;
    Shape2: TShape;
    RadioGroup1: TRadioGroup;
```

```
    Bevel1: TBevel;
    Label1: TLabel;
    Label2: TLabel;
    BitBtn1: TBitBtn;
    BitBtn2: TBitBtn;
    ColorDialog1: TColorDialog;
    Image1: TImage;
    procedure Shape2MouseUp(Sender: TObject; Button: TMouseButton;
      Shift: TShiftState; X, Y: Integer);
    procedure Shape1ContextPopup(Sender: TObject; MousePos: TPoint;
      var Handled: Boolean);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  FrmCtrlEditor: TFrmCtrlEditor;

implementation

{$R *.dfm}

procedure TFrmCtrlEditor.Shape2MouseUp(Sender:
                                       TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
if ColorDialog1.Execute then
begin
   Shape2.Brush.Color := ColorDialog1.Color;
end;
end;

procedure TFrmCtrlEditor.Shape1ContextPopup(Sender: TObject;
  MousePos: TPoint; var Handled: Boolean);
begin
if ColorDialog1.Execute then
begin
   Shape1.Brush.Color := ColorDialog1.Color;
end;
```

```
end;

end.
```

用户可以通过在 Delphi 的集成开发环境中双击窗体上的一个控件来进入它的控件编辑器。当用户双击控件的时候，将会执行下面的代码：

```
procedure TJFSControlEditor.Edit;
var
  MyFrm: TFrmCtrlEditor;
begin
  MyFrm := TFrmCtrlEditor.Create(nil);
  MyFrm.Shape2.Brush.Color := TGraphNum(Component).ForeColor;
  MyFrm.Shape1.Brush.Color := TGraphNum(Component).ShadowColor;
  MyFrm.RadioGroup1.ItemIndex :=
                          integer(TGraphNum(Component).ShadowStyle);
  MyFrm.Caption:=TGraphNum(Component).name+' Editor';
  Myfrm.ShowModal;
  TGraphNum(Component).SetForecolor(MyFrm.Shape2.Brush.Color);
  TGraphNum(Component).setshadowcolor(MyFrm.Shape1.Brush.Color);
TGraphNum(Component).setshadowStyle(TJfsshadowstyle(
                                    MyFrm.Radiogroup1.ItemIndex));
  MyFrm.Free;
end;
```

这段代码首先创建了上面设计的窗体对象的一个实例，然后它对用户所要编辑的所有字段都分配了一个默认值，最后，它将对话框显示出来并随后获得用户的输入。

和使用属性编辑器的时候类似，一个控件编辑器也必须要注册到 Delphi 的集成开发环境中才能达到我们的最终目的：简化控件的设计操作。所以还需要对这个控件编辑器进行注册：

```
procedure Register ;
begin
      ...
     RegisterComponentEditor ( TJFSNum , TJFSControlEditor) ;
      ...
end ;
```

下面是对该处理过程的声明：

```
procedure RegisterComponentEditor
      ( ComponentClass ： TComponentClass ;
      ComponentEditor ： TComponentEditorClass ) ;
```

第一个参数指定了与编辑器相联系的类，第二个参数指定了编辑器的类。

10.5 本 章 小 结

本章介绍了在 Delphi 中创建控件的基本知识，包括继承父类、添加属性、添加事件，以及使用属性编辑器和控件编辑器。它们都是创建控件时必须掌握的一些基本知识。在掌握了它们之后，我们就可以创建许多简单的控件了。

但是控件的创建不仅仅是掌握这些基本内容，还需要掌握和要创建的控件的功能相关的许多其他控件。比如，如果要创建一个图形控件，那需要掌握关于在 Delphi 中图形处理的技术；如果要创建一个用于网络中的控件，那么需要掌握相关的网络协议的技术。而且在本章的内容中，我们也仅仅举了一个简单的例子来演示了创建控件的过程和基本技术，在下一章的内容中，将给出更多的关于控件设计的例子，帮助读者了解在创建控件的过程中需要掌握的内容。

包是创建控件中的一个重要的内容。我们总是通过包来封装控件以及它们的属性编辑器、控件编辑器等内容。通过它，我们的控件才可以显示在 Delphi 的控件面板上，我们才能像使用 Delphi 自定义的控件一样使用自己的控件。

第 11 章　控件设计高级技术

在设计控件时，可以创建可视控件和不可视控件。这两种控件在创建的基本技术上是相同的。但是对于可视控件来说，我们的主要重心在它的外观和功能的实现上，比如可能要控制它的外观，使它按照我们需要的方式进行显示。对于不可视控件，我们通常关注它的功能实现。创建不可视控件有时候是为了封装一些特殊的功能，有时候是为了下一步创建可视控件时将它作为可视控件的父类。

可视控件的处理不仅仅会使用到我们在上面一章中讨论到的技术，而且可能会使用到很多关于图像处理，甚至是关于 Windows 本身机制的技术，比如 Windows 的消息机制。在本章的内容中，将通过一个控件的例子来介绍如何创建复杂的控件。

不可视控件的开发同可视控件相比在调试上具有一定的难度。因为必须结合其他的控件把不可视控件的功能发挥出来才能看到控件的功能。如果不可视控件具有一些不易发现的 Bug，那么可能会导致将来的程序崩溃。

ActiveX 控件也是控件的一种类型，比我们开发的适用于 Delphi 中的控件具有更广泛的适用性。我们可以把自己开发的 Delphi 的控件转换成 ActiveX 控件，并应用到其他的应用程序中，比如可以在 C++、Visual Basic 甚至是 Microsoft Word 等 Office 程序中应用。

本章将主要介绍：

- 可视控件
- 不可视控件
- ActiveX 控件

11.1　可视控件的开发

可视控件的开发涉及的技术是多方面的，我们很难在一节的内容中把所有可能需要的技术都一一进行介绍。在本部分的内容中，将通过一个图像按钮控件的例子来介绍在可视控件的开发中经常会使用到的技术。

图 11.1　SpeedButton 按钮的两个状态

在 Delphi 中提供了一个 SpeedButton 控件，它的功能我们在前面的内容中已经作了介绍。图 11.1 显示了该按钮的两个状态。

我们要开发一个和这样的控件类似的控件，这是许多多媒体应用程序或者其他独特风格

的程序界面中经常会使用到的一种控件，它可以根据以下四种状态进行显示。

- 鼠标不在控件上：控件的正常状态。
- 鼠标在控件上：当鼠标移动到控件上的时候。
- 鼠标按下：表示鼠标被按下。
- 控件禁用：Enabled 属性为 False。

在图 11.2 所示的窗口中显示了该控件的四种状态。为了夸张地显示我们的控件的效果，这里使用了一个夸张的图片。上面两个分别是正常状态和鼠标在控件上时的状态，下面两个则分别是鼠标单击和控件禁用的状态。

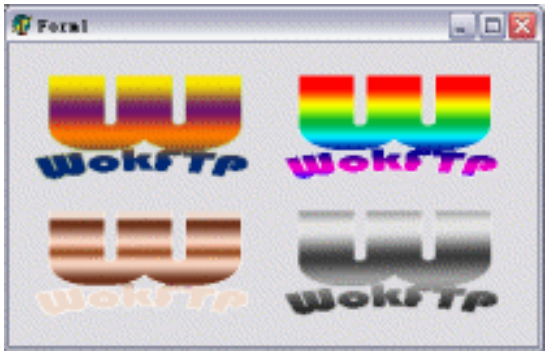

图 11.2　新控件的四种状态

下面介绍如何创建这样的一个图像按钮控件。同创建普通的可视控件相比较，在创建这样的控件时，需要解决如下两个问题。

- 控件中图像的处理：包括图像的显示和控件状态之间的关系；图像的存储方式；图像的显示状态，比如是否透明（Transparent 属性），是否进行拉伸（Stretch 属性）。
- 鼠标事件的处理：该控件的目的是根据鼠标以及控件本身的状态，把不同的图片显示到控件中，所以需要确定在什么时刻发生了什么样的鼠标事件。

11.1.1　处理控件中的图像

由于要处理一个图像控件，而且我们的控件不见得将来一定会显示成矩形的形状，比如很多多媒体播放器中的按钮通常都是不规则的形状，所以我们决定从 TGraphicControl 对象来派生要创建的控件。

在该控件中定义了下面的一些属性：

```
public
  property Canvas: TCanvas read GetCanvas;
published
  property Picture: TPicture read FPicture write SetPicture;
  property Transparent: Boolean read FTransparent
```

```
                write SetTransparent default False;
    property OnProgress: TProgressEvent read FOnProgress write FOnProgress;
    property Index:integer read FIndex write SetIndex;
    property KeepDown:boolean read FKeepDown write SetKeepDown;
    property PicNum:integer read FPicNum write SetPicNum;
    property Stretch: Boolean read FStretch write SetStretch default False;
```

所有的这些属性基本上都有一个私有的数据字段和它相关联。从它们的说明中也可以看出来。下面介绍一下这些属性的功能。

- Canvas **属性：这是一个只读属性，是用来作图和进行图像处理的** TCanvas **对象。**
- Picture **属性：是一个** TPicture **对象，用来存储显示在控件中的图片。**
- Transparent **属性：是一个** Boolean **属性，用来确定是否进行透明显示。**
- OnProgress **属性：是一个事件，用来处理图形变化时候的事件处理。实际上在我们的程序中，把一个事件处理程序赋给了** FPicture **对象的** OnProgress **事件。在这个事件处理程序中，处理控件的** OnProgress **事件。**
- Index **属性：显示了当前显示的是哪个状态的图片。**
- KeepDown **属性：如果该属性为** True**，那么当用户单击该控件的时候，控件将保持按下状态。**
- PicNum **属性：**Picture **属性中包含的图像数目。我们在这个控件中把需要的多个图片放在一个图片中，并利用** PicNum **属性来指定该控件中包含的图片数目。**
- Stretch **属性：该属性类似于** TImage **控件的** Stretch **属性，用于把图片充满控件的整个区域。**

下面来看一下控件的创建方法，也就是 Create **方法：**

```
constructor TGraphBtn.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);
  ControlStyle := ControlStyle + [csReplicatable];
  FPicture := TPicture.Create;
  FPicture.OnChange := PictureChanged;
  FPicture.OnProgress := Progress;
  OnMouseDown:=MouseDown;
  OnMouseUp:=MouseUp;
  Height := 25;
  Width := 25;
  KeepDown:=False;
  Downed:=False;
  FIndex:=0;
  Stretch:=True;
```

```
    PicNum:=4;
end;
```

在这个程序中，用显示调用继承的创建方法来创建对象，然后对 FPicture 数据字段进行了初始化，最后对控件的默认属性进行设置。在上面的程序中，使用了 TControl 对象的 ControlStyle 属性，该属性确定了控件的类型。这里使用了 csReplicatable，在这种情况下，该控件可以利用 PaintTo 方法把一个图片绘制到 Canvas 对象上。

下面来看一下图像的处理。在程序中，根据用户指定的 PicNum 属性的值把 Picture 对象中包含的图像进行分隔显示，那么需要注意的的问题就只有以下两个。

- 由 Index 属性来确定当前要显示的图像。
- 计算要显示的图像在 Picture 属性中的位置。

对于指定的 Index，通过下面一个方法来计算目标位置：

```
function TGraphBtn.DestRect: TRect;
var Rec,Rec1:TRect;
begin
  if not Enabled then
    Index:=3;
  Rec.Top := 0;
  if Stretch then
  begin
    Rec.Left :=-Index*Width;
    Rec.Bottom := Height;
    Rec.Right :=(PicNum-Index)*Width;
  end
  else
  begin
    Rec.Left :=-Index*Picture.Width Div PicNum;
    Rec.Bottom := Picture.Height;
    Rec.Right :=(PicNum-Index)*Picture.Width div PicNum;
  end;
     Result := Rec;
end;
```

在这个方法中，分了三种情况：一种是控件为禁用状态的时候，另一种是 Stretch 属性为 True 的时候，以及其他情况。

在计算了要绘制的目标区域之后，可以使用控件的 Paint 方法来绘制控件，Paint 方法的代码如下所示：

```
procedure TGraphBtn.Paint;
var
  Save: Boolean;
begin
```

```
    if csDesigning in ComponentState then
      with inherited Canvas do
      begin
        Pen.Style := psDash;
        Brush.Style := bsClear;
        Rectangle(0, 0, Width, Height);
      end;
    Save := FDrawing;
    FDrawing := True;
    try
      with inherited Canvas do
        StretchDraw(DestRect, Picture.Graphic);
    finally
      FDrawing := Save;
    end;
  end;
```

我们处理了 Picture **属性中的** FPicture **对象的** OnChange **事件。在** OnChange **事件中写入了下面的代码：**

```
procedure TGraphBtn.PictureChanged(Sender: TObject);
var
  G: TGraphic;
begin
  if AutoSize and (Picture.Width > 0) and (Picture.Height > 0) then
    SetBounds(Left, Top, Picture.Width, Picture.Height);
  G := Picture.Graphic;
  if G <> nil then
  begin
    if not ((G is TMetaFile) or (G is TIcon)) then
      G.Transparent := FTransparent;
    if (not G.Transparent) and (Stretch or (G.Width >= Width)
      and (G.Height >= Height)) then
      ControlStyle := ControlStyle + [csOpaque]
    else
      ControlStyle := ControlStyle - [csOpaque];
    if DoPaletteChange and FDrawing then Update;
  end
  else ControlStyle := ControlStyle - [csOpaque];
  if not FDrawing then Invalidate;
end;
```

在这里需要说明的是控件的 AutoSize、Transparent **属性的实现方法，这里采用的方法和**

Delphi 中的 TImage 控件的方法类似。这里我们仍然用 ControlStyle 属性来处理图像的绘制类型。

11.1.2 处理控件中的事件

在 Delphi 的控件设计中，有时候需要处理一些特殊的事件，比如在这里的程序中，需要实现鼠标离开控件时的控件状态。

当鼠标离开控件之后，实际上已经很难再用常规的方法来获得鼠标是否还在控件上，以及什么时候鼠标离开了控件。所以此时必须使用 Windows 的消息。这些消息有一部分定义在 Controls 单元中，有一部分定义在 Messages 单元中，还有一部分定义在 Windows 单元中。这里要使用 Controls 单元中定义的两个消息，下面列出了这个单元中的部分消息的定义：

```
CM_CTL3DCHANGED              = CM_BASE + 16;
CM_PARENTCTL3DCHANGED        = CM_BASE + 17;
CM_TEXTCHANGED               = CM_BASE + 18;
CM_MOUSEENTER                = CM_BASE + 19;
CM_MOUSELEAVE                = CM_BASE + 20;
CM_MENUCHANGED               = CM_BASE + 21;
CM_APPKEYDOWN                = CM_BASE + 22;
CM_APPSYSCOMMAND             = CM_BASE + 23;
CM_BUTTONPRESSED             = CM_BASE + 24;
CM_SHOWINGCHANGED            = CM_BASE + 25;
CM_ENTER                     = CM_BASE + 26;
CM_EXIT                      = CM_BASE + 27;
```

这里要使用其中的两个消息，CM_MOUSELEAVE 和 CM_MOUSEENTER。我们声明了下面两个方法：

```
procedure WmMouseMove(var Msg:TMessage);message Cm_MouseEnter;
Procedure WmMouseLeave(var Msg:TMessage);message Cm_MouseLeave;
```

这里演示了如何来使用处理 Windows 消息的技术。我们在程序中是这样定义的：

```
procedure TGraphBtn.WmMouseLeave(var Msg: TMessage);
begin
    if not Enabled then
        exit;
    if FIndex<>0 then
    begin
        FIndex:=0;
        invalidate;
    end;
```

```
end;

procedure TGraphBtn.WmMouseMove(var Msg: TMessage);
begin
    if not Enabled then
        exit;
    if FIndex<>1 then
    Begin
        FIndex:=1;
        invalidate;
    end;
end;
```

这里用重新定义控件的 OnMouseDown 和 OnMouseUp 的方法来定义鼠标在控件上和离开控件时的情况。

11.1.3　控件的完整代码

上面简要地选择了其中我们认为比较关键的几个问题进行了介绍，下面列出了控件的完整代码，供读者参考。

```
unit GraphBtn;

interface

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

resourcestring
  SImageCanvasNeedsBitmap = 'Can only modify an image if it contains a bitmap';

type
  TGraphBtn = class(TGraphicControl)
  private
    FPicture: TPicture;
    FIndex: integer;
    FPicNum: integer;
    FOnProgress: TProgressEvent;
    FStretch: Boolean;
    FCenter: Boolean;
    FIncrementalDisplay: Boolean;
    FTransparent: Boolean;
```

```
    FDrawing: Boolean;
    FKeepDown:Boolean;
    Downed:Boolean;

    procedure SetPicNum(Value:integer);
    function GetCanvas: TCanvas;
    procedure PictureChanged(Sender: TObject);
    procedure WmMouseMove(var Msg:TMessage);message Cm_MouseEnter;
    Procedure WmMouseLeave(var Msg:TMessage);message Cm_MouseLeave;
    procedure SetKeepDown(Value:boolean);
    procedure SetIndex(Value:integer);
    procedure SetCenter(Value: Boolean);
    procedure ShowPic(Pic:TPicture);
    procedure SetPicture(Value: TPicture);
    procedure SetStretch(Value: Boolean);
    procedure SetTransparent(Value: Boolean);

    property Center: Boolean read FCenter write SetCenter default False;
    procedure MouseUp(Sender: TObject; Button: TMouseButton;
      Shift: TShiftState; X, Y: Integer);
    procedure MouseDown(Sender: TObject; Button: TMouseButton;
      Shift: TShiftState; X, Y: Integer);

  protected
    function CanAutoSize(var NewWidth, NewHeight: Integer): Boolean; override;
    function DestRect: TRect;
    function DoPaletteChange: Boolean;
    function GetPalette: HPALETTE; override;
    procedure Paint; override;
    procedure Progress(Sender: TObject; Stage: TProgressStage;
      PercentDone: Byte; RedrawNow: Boolean; const R: TRect; const Msg: string); dynamic;
  public
    constructor Create(AOwner: TComponent); override;
    destructor Destroy; override;
    property Canvas: TCanvas read GetCanvas;
  published
    property Align;
    property Anchors;
    property AutoSize;
    property Constraints;
    property DragCursor;
    property DragKind;
```

```
    property DragMode;
    property Enabled;
    property IncrementalDisplay: Boolean read FIncrementalDisplay write FIncrementalDisplay
    default False;
    property ParentShowHint;
    property Picture: TPicture read FPicture write SetPicture;
    property PopupMenu;
    property ShowHint;
    property Transparent: Boolean read FTransparent write SetTransparent default False;
    property Visible;
    property OnClick;
    property OnContextPopup;
    property OnDblClick;
    property OnDragDrop;
    property OnDragOver;
    property OnEndDock;
    property OnEndDrag;
    property OnProgress: TProgressEvent read FOnProgress write FOnProgress;
    property OnStartDock;
    property OnStartDrag;
    property Index:integer read FIndex write SetIndex;
    property KeepDown:boolean read FKeepDown write SetKeepDown;
    property PicNum:integer read FPicNum write SetPicNum;
    property Stretch: Boolean read FStretch write SetStretch default False;
  end;

procedure Register;

implementation

procedure Register;
begin
  RegisterComponents('JfsControl', [TGraphBtn]);
end;
constructor TGraphBtn.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);
  ControlStyle := ControlStyle + [csReplicatable];
  FPicture := TPicture.Create;
  FPicture.OnChange := PictureChanged;
  FPicture.OnProgress := Progress;
  OnMouseDown:=MouseDown;
```

```
  OnMouseUp:=MouseUp;
  Height := 25;
  Width := 25;
  KeepDown:=False;
  Downed:=False;
  FIndex:=0;
  Stretch:=True;
  PicNum:=4;
end;

destructor TGraphBtn.Destroy;
begin
  FPicture.Free;
  inherited Destroy;
end;

function TGraphBtn.GetPalette: HPALETTE;
begin
  if FPicture.Graphic <> nil then
       Result := FPicture.Graphic.Palette;
end;

function TGraphBtn.DestRect: TRect;
var Rec,Rec1:TRect;
begin
  if not Enabled then
    Index:=3;

  Rec.Top := 0;
  if Stretch then
  begin
    Rec.Left := –Index*Width;
    Rec.Bottom := Height;
    Rec.Right :=(PicNum-Index)*Width;
  end
  else
  begin
    Rec.Left := –Index*Picture.Width Div PicNum;
    Rec.Bottom := Picture.Height;
    Rec.Right :=(PicNum-Index)*Picture.Width div PicNum;
  end;
     Result := Rec;
```

```
end;

procedure TGraphBtn.Paint;
var
  Save: Boolean;
begin
  if csDesigning in ComponentState then
    with inherited Canvas do
    begin
      Pen.Style := psDash;
      Brush.Style := bsClear;
      Rectangle(0, 0, Width, Height);
    end;
  Save := FDrawing;
  FDrawing := True;
  try
    with inherited Canvas do
      StretchDraw(DestRect, Picture.Graphic);
  finally
    FDrawing := Save;
  end;
end;

function TGraphBtn.DoPaletteChange: Boolean;
var
  ParentForm: TCustomForm;
  Tmp: TGraphic;
begin
  Result := False;
  Tmp := Picture.Graphic;
  if Visible and (not (csLoading in ComponentState)) and (Tmp <> nil) and
    (Tmp.PaletteModified) then
  begin
    if (Tmp.Palette = 0) then
      Tmp.PaletteModified := False
    else
    begin
      ParentForm := GetParentForm(Self);
      if Assigned(ParentForm) and ParentForm.Active and Parentform.HandleAllocated then
      begin
        if FDrawing then
          ParentForm.Perform(wm_QueryNewPalette, 0, 0)
```

```
            else
              PostMessage(ParentForm.Handle, wm_QueryNewPalette, 0, 0);
            Result := True;
            Tmp.PaletteModified := False;
          end;
        end;
      end;
    end;

    procedure TGraphBtn.Progress(Sender: TObject; Stage: TProgressStage;
      PercentDone: Byte; RedrawNow: Boolean; const R: TRect; const Msg: string);
    begin
      if FIncrementalDisplay and RedrawNow then
      begin
        if DoPaletteChange then Update
        else Paint;
      end;
      if Assigned(FOnProgress) then FOnProgress(Sender, Stage, PercentDone, RedrawNow, R, Msg);
    end;

    function TGraphBtn.GetCanvas: TCanvas;
    var
      Bitmap: TBitmap;
    begin
      if Picture.Graphic = nil then
      begin
        Bitmap := TBitmap.Create;
        try
          Bitmap.Width := Width;
          Bitmap.Height := Height;
          Picture.Graphic := Bitmap;
        finally
          Bitmap.Free;
        end;
      end;
      if Picture.Graphic is TBitmap then
        Result := TBitmap(Picture.Graphic).Canvas
      else
        raise EInvalidOperation.Create(SImageCanvasNeedsBitmap);
    end;

    procedure TGraphBtn.SetCenter(Value: Boolean);
```

```
begin
  if FCenter <> Value then
  begin
    FCenter := Value;
    PictureChanged(Self);
  end;
end;

procedure TGraphBtn.SetPicture(Value: TPicture);
begin
  FPicture.Assign(Value);
  Width:=FPicture.Width Div PicNum;
  Height:=FPicture.Height;
end;

procedure TGraphBtn.SetStretch(Value: Boolean);
begin
  if Value <> FStretch then
  begin
    FStretch := Value;
    PictureChanged(Self);
  end;
end;

procedure TGraphBtn.SetTransparent(Value: Boolean);
begin
  if Value <> FTransparent then
  begin
    FTransparent := Value;
    PictureChanged(Self);
  end;
end;

procedure TGraphBtn.PictureChanged(Sender: TObject);
var
  G: TGraphic;
begin
  if AutoSize and (Picture.Width > 0) and (Picture.Height > 0) then
    SetBounds(Left, Top, Picture.Width, Picture.Height);
  G := Picture.Graphic;
  if G <> nil then
  begin
```

```
      if not ((G is TMetaFile) or (G is TIcon)) then
        G.Transparent := FTransparent;
      if (not G.Transparent) and (Stretch or (G.Width >= Width)
        and (G.Height >= Height)) then
        ControlStyle := ControlStyle + [csOpaque]
      else
        ControlStyle := ControlStyle - [csOpaque];
      if DoPaletteChange and FDrawing then Update;
    end
    else ControlStyle := ControlStyle - [csOpaque];
    if not FDrawing then Invalidate;
end;

function TGraphBtn.CanAutoSize(var NewWidth, NewHeight: Integer): Boolean;
begin
    Result := True;
    if not (csDesigning in ComponentState) or (Picture.Width > 0) and
      (Picture.Height > 0) then
    begin
      if Align in [alNone, alLeft, alRight] then
        NewWidth := Picture.Width Div PicNum;
      if Align in [alNone, alTop, alBottom] then
        NewHeight := Picture.Height;
    end;
end;
procedure TGraphBtn.MouseUp(Sender: TObject; Button: TMouseButton;
    Shift: TShiftState; X, Y: Integer);
begin
    if FIndex<>1 then
    begin
       FIndex:=1;
       Invalidate;
    end;
end;

procedure TGraphBtn.ShowPic(Pic: TPicture);
var
    Save: Boolean;
begin
    if csDesigning in ComponentState then
      with inherited Canvas do
      begin
```

```
            Pen.Style := psDash;
            Brush.Style := bsClear;
            Rectangle(0, 0, Width, Height);
        end;
    Save := FDrawing;
    FDrawing := True;
    try
        with inherited Canvas do
            StretchDraw(DestRect, nil);
    finally
        FDrawing := Save;
    end;

    try
        with inherited Canvas do
            StretchDraw(DestRect, Pic.Graphic);
    finally
        FDrawing := Save;
    end;
end;
procedure TGraphBtn.SetIndex(Value:integer);
begin
    if FIndex=11 then
    begin
        FIndex:=0;
        invalidate;
    end;
    if FIndex<>Value then
    begin
        FIndex:=Value;
        invalidate;
    end;
end;
procedure TGraphBtn.SetKeepDown(Value: boolean);
begin
    if Value<>FKeepDown then
        FKeepDown:=Value;
end;
procedure TGraphBtn.SetPicNum(Value: integer);
begin
    if FPicNum <> Value then
    begin
```

```
        FPicNum:=Value;
        invalidate;
    end;
end;

procedure TGraphBtn.WmMouseLeave(var Msg: TMessage);
begin
    if not Enabled then
        exit;
    if FIndex<>0 then
    begin
        FIndex:=0;
        invalidate;
    end;

end;

procedure TGraphBtn.MouseDown(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
    if FIndex<>2 then
    begin
        FIndex:=2;
        Invalidate;
    end;
end;

procedure TGraphBtn.WmMouseMove(var Msg: TMessage);
begin
    if not Enabled then
        exit;
    if FIndex<>1 then
    Begin
        FIndex:=1;
        invalidate;
    end;
end;

end.
```

11.2　不可视控件

11.2.1　不可视控件概述

虽然上面提到，在设计不可视控件时所使用的技术基本上和设计可视控件的情况类似，例如可以为控件定义属性和事件，也可以为控件定义属性编辑器和控件编辑器，但是由于不可视控件具有不可视的特点，所以对于许多人来说可能仍然不太容易掌握这些技术的运用。在本节以及后面的内容中，将主要针对一些具体的不可视控件示例来介绍如何创建一个不可视控件。

对于不可视控件，通常是从一个顶层的对象或者一些不可视控件继承来的。所以在创建一个不可视控件的时候，不要试图从一个可视控件中派生这样的不可视控件。对于必须使用已经设计好的一些可视控件的功能时，可以用前面所介绍的利用控件的组合来创建新控件的办法来创建这样的不可视控件。

在本章的例子中，将主要创建一些和系统文件或者文件夹相关的控件。例如，我们会创建一个称为 TJfsFileSearch 的控件，利用这个控件，可以在一个文件夹中进行搜索，通过这样一个控件，不仅可以掌握创建不可视控件的技术，而且又回顾了关于系统文件操作的技术。

我们为自创建的 TJfsFileSearch 控件建立了两个示例程序，利用这两个示例程序可以充分演示如何使用自己创建的不可视控件。第一个是一个称为 Test1 的相对较简单的程序，第二个是一个复杂一些的程序，称为 Test2。

Test1 程序允许用户浏览文件夹来搜索文件然后在编辑器中查看它们。Test2 程序允许用户比较两个文件夹树来看看它们是否相同。如果它们是不同的，就会显示文件和文件夹的列表，并且通过使用这些列表，用户可以对两个文件夹进行操作以达到使它们相同的目的。

这两个程序都使用了 TJfsFileSearch 控件。这样，它们生动地展示了控件重用的概念。用户只编写了一次 TJfsFileSearch 控件，但是可以在多个程序里面重用它们。

11.2.2　创建基类

在创建控件的时候，特别是在创建以后可能需要不断完善的控件时，应该把控件分层放置，以便从不同的父类派生，在不同的环境下，建立不同的对象。这个想法是面向对象设计的关键，不要希望一次就把一个控件将来可能用到的所有功能都设计全面，“宁缺毋滥”这个词语用在这里可能再合适不过了，因为在创建这样的需要不断完善的控件时，最好的办法就是先创建一个顶层的控件，这个控件的功能不要求完善，但是它的每个功能应该是以后所有从这个控件派生出来的相关控件都可能使用到的。以后，当我们又有了新的想法之后，就

可以从这个控件派生出其他的控件。否则，如果想获得它的一个新的功能，但是又想去掉它的一些原有功能时，就不得不重新编写这个对象。

在这个方面，Delphi 的开发人员可以说做得非常出色。例如，如果他们没有建立 TBDEDataSet 控件，而是代之建立了一个叫做 TTable 的控件，他们可能就不得不重复 TQuery 控件的相同功能，这种重复是一种浪费。相反，他们采用了聪明的做法，建立了一个叫做 TDataSet 的控件，它的功能只包括通用的功能，具体属性则需要由 TQuery 和 TTable 去定义。这样，TQuery 和 TTable 就都可以使用 TBDEDataSet 的功能了，而不需要为每个对象重新编写相同的功能。

我们已经介绍了要创建这样的一个控件的目的，但是在创建 TJfsFileSearch 控件之前，应该创建一个包含了能够为实现 TJfsFileSearch 控件的功能而服务的其他对象。我们把这些对象放置在了一个称为 AllDirs 的单元文件中。这个单元文件中具有内建的堆栈，并有一个叫做 TRunDirs 的对象，该对象知道如何遍历子文件夹。需要说明的是，TRunDirs 是 TComponent 的派生对象。不可见控件经常是直接从 TComponent 派生出来的。根据定义，不可见控件是不可能从 TCustomControl、TGraphicControl 或者 TWinControl 派生出来的，因为它们都是可见的控件。

这个单元文件的代码如下所示：

```
unit AllDirs;

interface

{$H+}

uses
  Classes, Controls, SysUtils;

type
  DirStr = string;
  PathStr = string;
  NameStr = string;
  ExtStr = string;

  TStack = class;
  TShortStack = class;

  TStackAry = array[1..1000] of PString;
  TStacksAry = array[1..1000] of TShortStack;

  TStack = class(TObject)
```

```
    First,
    Last: Word;
    constructor Create;
    procedure InitCount;
    function IsEmpty: Boolean;
    function Count: Integer;
end;

TBigStack = class(TStack)
    Stacks: TStacksAry;
    destructor Destroy; override;
    procedure Push(P: TShortStack);
    function Pop: TShortStack;
    function PopValue(var Num: Integer): String;
end;

TShortStack = class(TStack)
    StackAry: TStackAry;
    destructor Destroy; override;
    procedure Push(S: String);
    function Pop: String;
    function GetMoreDirs(DirAndWildCard: String): Integer;
    procedure Show;
end;

TFoundFileEvent = procedure(FileName: string; SR: TSearchRec) of Object;
TFoundDirEvent = procedure(DirName: string) of Object;

TRunDirs = class(TComponent)
private
    FOnFoundFile: TFoundFileEvent;
    FOnProcessDir: TFoundDirEvent;
    FFileMask: String; //was Str12
    FCurDir: DirStr;
    FBigStack: TBigStack;
    FShortStack: TShortStack;
protected
    procedure PushStack;
    procedure ProcessName(FName: String; SR: TSearchRec); virtual;
    procedure ProcessDir(Start: String); virtual;
public
    constructor Create(Owner: TComponent); override;
```

```
    destructor Destroy; override;
    function Run(Start: PathStr; StartingDirectory: String): String;
  published
    property OnFoundFile: TFoundFileEvent
      read FOnFoundFile write FOnFoundFile;
    property OnProcessDir: TFoundDirEvent
      read FOnProcessDir write FOnProcessDir;
    property CurDir: DirStr read FCurDir;
  end;

implementation

{$IfDef Debug}
var
  F: Text;
{$EndIf Debug}

function Shorten(S: string; Cut: Integer): string;
begin
  SetLength(S, Length(S) - Cut);
  Shorten := S;
end;

constructor TStack.Create;
begin
  inherited Create;
  InitCount;
end;

procedure TStack.InitCount;
begin
  First := 1;
  Last := 0;
end;

function TStack.IsEmpty: Boolean;
var
  OutCome: Boolean;
begin
  OutCome := First > Last;
  IsEmpty := OutCome
end;
```

```
function TStack.Count: Integer;
begin
  Count := Last - First;
end;

destructor TBigStack.Destroy;
var
  i: Integer;
begin
  for i := First to Last do
    Stacks[i].Destroy;
  inherited Destroy;
end;

procedure TBigStack.Push(P: TShortStack);
begin
  Inc(Last);
  Stacks[Last] := P;
end;

function TBigStack.Pop: TShortStack;
begin
  Result := nil;
end;

function TBigStack.PopValue(var Num: Integer): String;
begin
  Num := 0;
  if IsEmpty then begin
    PopValue := '–1';
    Num := –1;
    Exit;
  end;
  while Stacks[Last].IsEmpty do begin
    Inc(Num);
    Stacks[Last].Destroy;
    Dec(Last);
    if IsEmpty then begin
      PopValue := '–1';
      Num := –1;
      Exit;
```

```
      end;
    end;
    if Last = 0 then begin
      PopValue := '–1';
      Exit;
    end;
    PopValue := Stacks[Last].Pop;
end;

destructor TShortStack.Destroy;
var
    i: Integer;
begin
    if not IsEmpty then
      for i := First to Last    do
        DisposeStr(StackAry[i]);
    inherited Destroy;
end;

procedure TShortStack.Show;
var
    i: Integer;
begin
    for i := First to Last do begin
      {$IfDef Debug}
      WriteLn(F, StackAry[i]^);
      {$EndIf}
      WriteLn(StackAry[i]^);
    end;
    {$IfDef Debug}
    WriteLn(F, '===============');
    {$EndIf}
end;

procedure TShortStack.Push(S: String);
begin
    if (S <> '.') and (S <> '..') then begin
      Inc(Last);
      StackAry[Last] := NewStr(S);
    end;
end;
```

```
function TShortStack.Pop: String;
var
   S: PString;
   Temp: ShortString;
begin
   S := StackAry[First];
   if S <> nil then begin
      Temp := S^;
      DisposeStr(StackAry[First]);
      Inc(First);
      Pop := Temp;
   end
   else begin
      WriteLn('Error TShortStack.Pop');
      Halt;
   end;
end;

function TShortStack.GetMoreDirs(DirAndWildCard: String): Integer;
var
   SR: SysUtils.TSearchRec;
   Total: Integer;
begin
   Total := 0;
   if FindFirst(DirAndWildCard, faDirectory + faReadOnly, SR) = 0 then
      repeat
         if (SR.Attr and faDirectory = faDirectory) then begin
            Push(SR.Name);
            Inc(Total);
         end;
      until FindNext(SR) <> 0;
   FindClose(SR);
   GetMoreDirs := Total;
end;

constructor TRunDirs.Create(Owner: TComponent);
begin
   inherited Create(Owner);
   {$IfDef Debug}
   Assign(F, 'DirLists.dat');
   ReWrite(F);
   {$EndIf}
```

```
  FShortStack := TShortStack.Create;
  FBigStack := TBigStack.Create;
end;

destructor TRunDirs.Destroy;
begin
  FShortStack.Free;
  FBigStack.Free;
  {$IfDef Debug}
  Close(F);
  {$EndIf}
  inherited Destroy;
end;

procedure TRunDirs.PushStack;
begin
  FBigStack.Push(FShortStack);
  FShortStack := TShortStack.Create;
end;

procedure SplitDirName(Path: PathStr; var Dir: DirStr; var WName: String);
begin
  Dir := ExtractFilePath(Path);
  WName := ExtractFileName(Path);
end;

function RemoveDir(Start: String; NumDirs: Integer): String;
var
  i, j: Integer;
  CurDir: DirStr;
  FileMask: string;
begin
  SplitDirName(Start, CurDir, FileMask);
  i := Length(CurDir);
  for j := 1 to NumDirs + 1 do begin
    if CurDir[i] = '\' then   begin
      CurDir := Shorten(CurDir, 1);
      Dec(i);
    end;
    while CurDir[i] <> '\' do begin
      CurDir := Shorten(CurDir, 1);
      Dec(i);
```

```
      end;
    end;
    RemoveDir := CurDir;
end;

procedure TRunDirs.ProcessName(FName: String; SR: SysUtils.TSearchRec);
begin
    if Assigned(FOnFoundFile) then
      FOnFoundFile(FName, SR);
end;

procedure TRunDirs.ProcessDir(Start: String);
var
    SR: SysUtils.TSearchRec;
    DoClose: Boolean;
begin
    DoClose := False;
    if Assigned(FOnProcessDir) then FOnProcessDir(FCurDir);
    if FindFirst(Start, faArchive, SR) = 0 then begin
      DoClose := True;
      repeat
        ProcessName(UpperCase(FCurDir) + SR.Name, SR);
      until FindNext(SR) <> 0;
    end;
    if DoClose then
      FindClose(SR);
end;

function TRunDirs.Run(Start: PathStr; StartingDirectory: string): string;
const
    DirMask = '*.*';
var
    Finished: Boolean;
    NewDir, StartedAt: string;
    NumDirs: Integer;
    OutCome: Integer;
    SaveDir: string;
begin
    GetDir(0, SaveDir);
    try
      ChDir(StartingDirectory);
    except
```

```
      raise Exception.Create('Directory does not exist: ' +
        StartingDirectory);
    end;
    Start := ExpandFileName(Start);
    FCurDir := ''; FFileMask := '';
    Finished := False;
    StartedAt := Start;
    SplitDirName(Start, FCurDir, FFileMask);
    Start := FCurDir + DirMask;
    while not Finished do begin
      FCurDir := ExtractFilePath(Start);
      ProcessDir(FCurDir + FFileMask);
      OutCome := FShortStack.GetMoreDirs(Start);
      if OutCome > 2 then begin
        PushStack;
        Start := FCurDir + FBigStack.PopValue(NumDirs) + '\' + DirMask
      end else begin
        NewDir := FBigStack.PopValue(NumDirs);
        FCurDir := RemoveDir(Start, NumDirs);
        Start := FCurDir + NewDir + '\' + DirMask;
        if (Start = StartedAt) or (NewDir = '-1') then Finished := True;
      end;
    end;
    ChDir(SaveDir);
  end;

  end.
```

下面，让我们研究一下这个单元文件中的几个关键点。AllDirs **是这个特定操作的主程序，它知道怎样搜索文件夹，怎样找出每个文件夹中的所有文件以及怎样通知用户找到了新的文件夹和文件。**FileIter **单元增加了把文件和文件夹列表保存在** TStringList **对象中去的功能。当然，也可以使用** SaveToFile **命令把这些列表写到磁盘上去。**

搜索文件夹的任务可以使用一种简单的递归算法来完成。不过，递归这项技术通常很慢、很耗时，同时还要占用大量堆栈空间。因此，AllDirs **建立了自己的堆栈并把它找到的文件夹推进到这些堆栈中。**

下面的对象都是在 AllDirs **单元中建立的。**

- TStack **对象是一个抽象类，它提供了一些可以处理所有堆栈类的基本功能。永远也不必用到这种类型的实际对象。**
- TShortStack **对象可以处理一个长达** 1000 **个** long strings **的数组。它具有存储和删除这些项目的全部逻辑。它把它们存放在一个数组中，这个数组只占用** 4000 **字节的**

内存。每个 long string 占 4 个字节，乘以 1000，就是 4000 字节。对于这类程序来说，这个数量有点过大了，因为不太可能遇到一个 1000 重嵌套的文件夹。

- ❖ TBigStack 对象为 TShortStack 对象创建栈。一个文件夹所拥有的子文件夹的信息可以存放在 TShortStack 对象中。但是如果一个文件夹有多个子文件夹，它的子文件夹又有多个子文件夹，那么就需要用到 TBigStack 对象了。
- ❖ TRunDirs 对象里到处都是一系列 FindFirst 和 FindNext 调用。它使用这些 Delphi 函数在文件夹中查找文件。然后它就把它找到的文件夹推进 TShortStack 对象和 TBigStack 对象中去。

用经典的比喻来说，这里实现的 FIFO（先入先出）堆栈和 LIFO（后入先出）堆栈就像是厨房柜子里的一叠盘子。可以放下一个盘子，然后再把另一个盘子放到它的上面。当需要盘子时，就从顶端拿走一个，或者是从底部抽出一个，这就要看究竟使用的是 FIFO 堆栈还是 LIFO 堆栈了。把一个盘子放到堆栈的顶部叫做把对象入栈（pushing），而拿走一个盘子则叫做将对象出栈（popping）。

下面我们来分析，如何来搜索文件夹并处理堆栈。这里的技术核心是调用 FindFirst、FindNext 和 FindClose，它们可以搜索文件夹、查找特定的文件。

使用 FindFirst、FindNext 和 FindClose 就好像是在 DOS 提示符下，在某个文件夹中使用 DIR 命令一样。FindFirst 找到文件夹中的第一个文件，FindNext 找到剩下的文件。当结束这个过程时，则应该调用 FindClose。

这些调用运行指定文件夹和文件通配符，就像在 DOS 提示符下使用下面这种命令一样：

```
Dir C : \ aut*.bat
```

很显然，这个命令将显示所有以 aut 开头，以.bat 结尾的文件。在典型的情况下，这个命令将找到 AUTOEXEC.BAT 文件，也许还有其他一个或两个别的文件。

调用 FindFirst 时，要传递三个参数：

```
function FindFirst(const Path:string;Attr:Word;var F:TSearchRec):integer;
```

第一个参数包括想要查找的文件的路径和文件通配符。例如，可以在这个参数中传递：

```
'c : \ delphi32 \ source \ vcl \ * . pas '
```

或

```
' c : \ program files \ borland \ delphi 2.0 '
```

第二个参数则列出想看的文件类型。这些文件类型都是我们比较熟悉的，它们在不同的开发系统中的定义大同小异。

文件属性见表 11.1。

表 11.1 文件属性

faReadOnly	$01	只读文件
faHidden	$02	隐藏文件
faSysFile	$04	系统文件
faVolumeID	$08	卷标 ID 文件
faDirectory	$10	文件夹文件
faArchive	$20	文档文件
faAnyFile	$3F	任何文件

在多数情况下，应该给这个参数传递 faArchive。不过，如果想查看文件夹，则应该使用 faDirectory。传递 faDirectory 的结果是文件夹也会被包括在普通文件列表中，而不是仅限于文件才能进入列表。如果愿意，可以使用 OR（或方法）将几个不同的 faXXX 常量连在一起。

最后一个参数是一个 TSearchRec 类型的变量，它的声明如下：

```
TSearchRec = record
    Fill: array [1..21] of Byte;
    Attr: byte;
    Time: Longint;
    Size: Longint;
    Name: string[12];
end;
```

TSearchRec 中最重要的值是 Name 域，如果调用成功，该字段将确定找到的文件的名称。如果 FindFirst 找到了文件，它就返回零；而如果调用失败，则返回一个非零值。

FindNext 与 FindFirst 很相似，只不过这时只需传递一个 TSearchRec 类型的变量就可以了，因为 FindNext 会采用与 FindFirst 相同的文件通配符和文件属性。和 FindFirst 一样，如果 FindNext 一切顺利，它会返回零；而如果它找不到文件，则会返回一个非零值。在顺序完成 FindFirst 或者 FindNext 后，应该调用 FindClose。

知道了这些知识以后，可以用下面这样简单的方法来调用 FindFirst、FindNext 和 FindClose：

```
var
  SR: TSearchRec;
begin
  if FindFirst(Start, faArchive, SR) = 0 then begin
    repeat
      //DoSomething(SR.Name);
    until FindNext(SR) <> 0;
  end;
  FindClose(SR);
```

当 TRunDirs 准备好处理新文件夹时，它把它的名字传递给叫做 ProcessDir 的方法：

```
procedure TRunDirs.ProcessDir(Start: String);
var
  SR: SysUtils.TSearchRec;
  DoClose: Boolean;
begin
  DoClose := False;
  if Assigned(FOnProcessDir) then FOnProcessDir(FCurDir);
  if FindFirst(Start, faArchive, SR) = 0 then begin
    DoClose := True;
    repeat
      ProcessName(UpperCase(FCurDir) + SR.Name, SR);
    until FindNext(SR) <> 0;
  end;
  if DoClose then
    FindClose(SR);
end;
```

ProcessDir 在文件夹的所有文件之间切换，并把找到的每个文件都传递给 ProcessName 的方法：

```
procedure TRunDirs.ProcessName(FName: String; SR: SysUtils.TSearchRec);
begin
  if Assigned(FOnFoundFile) then
    FOnFoundFile(FName, SR);
end;
```

ProcessDir 和 ProcessName 方法都是虚方法。因此可以创建一个 TrunDirs 的派生类，重载其中的某一个方法，并按照自己喜欢的方式做出相应的响应。

创建一个 TrunDirs 的派生类是一个非常简单的操作，但通过授权模型对事件处理程序做出反应会更加简单。换句话说，可以建立一个 TrunDirs（或者是 TJfsFileSearch）的派生类，然后重载 ProcessName 方法。这样做可以很容易地访问到被处理的每个名字。但是，还可以通过一个更简单的方法达到这个目的。具体地说，可以建立一个事件处理程序，并在每次找到文件时调用这个事件。

要想创建一个像 OnXXX 这样的事件处理程序，必须首先定义一个指向方法的指针。建立的这个方法指针将指向在事件发生时应该调用的方法。每一个特定类型的方法处理程序都会有一个标志。例如，OnClick 事件总是得到应该叫做 Sender 的参数，该参数是 TObject 类型的。

参数为 Sender/Tobject 类型的例程，叫做 TNotifyEvent，是像下面这样声明的：

```
TNotifyEvent = procedure ( Sender : TObject ) of object ;
```

这句代码只是一个方法指针的声明。其中令人困惑的是“ of object ”语句。如果把它去掉，可以很容易地说出这里正在进行什么。“ of object ”只不过是要告诉编译器这是一个指向方法的指针，而不是指向函数或进程的指针。

从 TNotifyEvent 方法指针声明到一个如下的事件很容易理解：

```
procedure TForm1.Button1Click ( Sender : TObject ) ;
```

Button1Click 方法就是一个 TNotifyEvent 类型的方法的实例。一个 OnClick 事件所做的全部工作就是提供一个与存放在对象中的方法指针相匹配的方法实例。

可以像下面这样为 OnFoundDir 事件声明方法指针：

```
TFoundDirEvent = procedure ( DirName : string ) of Object ;
```

这个指针指向的方法只有一个字符串参数。当单击 Events 页面的 OnFoundDir 事件时，就会创建出这种类型的方法。当单击一个按钮的 OnClick 事件时也会发生几乎同样的事，只不过方法类型的句柄不同而已。更具体地说，这时的句柄是定义在 AllDirs 单元中的，而不是随 Delphi 发送的那些单元中。

为了定义自己的事件，我们也声明了下面的事件类型：

```
TFoundFileEvent = procedure ( FileName : string ;
          SR : SysUtils . TSearchRec ) of Object ;
```

OnFoundFile 事件是 Test1 程序中实际用到的一个事件。但是我们只是集中讨论了 OnFoundDir 事件，这是因为 OnFoundDir 事件只有一个参数，因而比较容易理解。

可以像下面这样声明一个指向这种类型的对象的指针：

```
FOnProcessDir : TFoundDirEvent ;
```

现在 TRunDirs 有了一个内部变量，这个变量可以被设成等于正确类型的方法。不论发生了哪种特定事件，TRunDirs 对象都能使用这个变量调用指定的方法来处理该事件：

```
if Assigned ( FOnProcessDir ) then FOnProcessDir ( FCurDir ) ;
```

这句代码来自 ProcessDir 方法的主体。它首先检查 FOnProcessDir 是否被设成了 nil，如果不是 nil，那意味着已经指定了一个方法来处理这个事件，然后就调用这个方法。

事件句柄只是一种包含有指向函数的指针的属性，而不是其他什么别的类型的数据。下面是关于 OnProcessDir 事件的声明：

```
property OnProcessDir : TFoundDirEvent
    read FOnProcessDir
    write FOnProcessDir ;
```

可以看到这个属性被声明为 TFoundDirEvent 类型，而不是其他别的什么更简单的类型，例如字符串、整数或者集合。这个属性是作为一个接口，为以前提到过的 FOnProcessDir 变

量服务的。TFoundDirEvent 隐藏在私有部分，其他对象是看不到的：

```
private
   FonProcessDir : TFoundDirEvent ;
```

可以看到，FOnProcessDir 就是一个方法指针。它就是另一种 4 字节的指针，只不过它所指向的变量恰巧是一个方法指针，或者更具体地说，是一个事件句柄。

事件句柄很有吸引力，这是因为可以很容易地从属性编辑器那里访问到它们。双击事件句柄的属性编辑器，与该事件相关的方法就会立刻被插入到代码中。简而言之，事件句柄是代码生成器的一种温和形式，它所生成的代码可以为希望定义的任何种类的方法进行声明。

通过上面的代码，我们建立了一些用来提供最基本搜索文件夹功能的对象和其他必要的功能。但是，从程序代码中可以看出，我们并没有注册这样的对象，因为这样的对象的功能实在是太简单了，我们不希望任何用户在自己的应用程序中直接使用这样的对象。若要使用这样的对象的功能，则建议按照我们的做法，派生其他的控件。

11.2.3　创建 TJfsFileSearch 控件

现在我们可以创建 TJfsFileSearch 控件了。该 TJfsFileSearch 对象派生自 TRunDirs，但增加了列表管理能力。换句话说，TRunDirs 知道如何遍历子文件夹，也知道如何调用 OnFoundFile 和 OnFoundDir 事件。不过，可能还希望给 TRunDirs 的功能中再添加上管理列表的功能。因此这里加入了 TJfsFileSearch 控件作为额外功能的第二层。

下面是这个控件的代码：

```
unit JfsFS;

interface

{$H+}

uses
  SysUtils, Windows,
  Messages, Classes, Graphics,
  Controls, Forms, AllDirs;

type
  TJfsFileSearch = class(TRunDirs)
  private
    FFileList: TStringList;
    FDirList: TStringList;
    FUseFileList: Boolean;
```

```
    FUseDirList: Boolean;
    procedure SetFileList(UseList: Boolean);
    procedure SetDirList(UseList: Boolean);
  protected
    procedure ProcessName(FName: String; SR: TSearchRec); override;
    procedure ProcessDir(Start: String); override;
  public
    destructor Destroy; override;
    property FileList: TStringList read FFileList;
    property DirList: TStringList read FDirList;
  published
    property UseFileList: Boolean read FUseFileList write SetFileList;
    property UseDirList: Boolean read FUseDirList write SetDirList;
    property OnFoundFile;
    property OnProcessDir;
  end;

procedure Register;

implementation

procedure Register;
begin
  RegisterComponents('CustomControl', [TJfsFileSearch]);
end;

destructor TJfsFileSearch.Destroy;
begin
  FFileList.Free;
  FDirList.Free;
  inherited Destroy;
end;

procedure TJfsFileSearch.SetFileList(UseList: Boolean);
begin
  FUseFileList := UseList;
  if FUseFileList then
    FFileList := TStringList.Create
  else
    FFileList.Free;
end;
```

```
procedure TJfsFileSearch.SetDirList(UseList: Boolean);
begin
  FUseDirList := UseList;
  if FUseDirList then
    FDirList := TStringList.Create
  else
    FDirList.Free;
end;

procedure TJfsFileSearch.ProcessName(FName: String; SR: TSearchRec);
begin
  inherited ProcessName(FName, SR);
  if FUseFileList then FFileList.Add(FName);
end;

procedure TJfsFileSearch.ProcessDir(Start: string);
begin
  inherited ProcessDir(Start);
  if FUseDirList then FDirList.Add(CurDir);
end;

end.
```

由于我们使用了代码重用，所以这段代码看起来就简练多了。从这里就可以充分体会到代码重用的好处。

11.2.4　测试程序

接着开始使用上面创建的 TJfsFileSearch 控件。下面是我们建立的第一个应用程序：

```
unit Main;

interface

uses
  WinTypes, WinProcs, Classes,
  Graphics, Forms, Controls,
  AllDirs, StdCtrls, FileCtrl,
  Dialogs, SysUtils, ExtCtrls, JfsFS;

type
  TForm1 = class(TForm)
```

```
    BStartSearch: TButton;
    ListBox1: TListBox;
    MaskEdit: TEdit;
    Label1: TLabel;
    Label2: TLabel;
    JfsFileSearch1: TJfsFileSearch;
    DirectoryListBox1: TDirectoryListBox;
    DriveComboBox1: TDriveComboBox;
    Label3: TLabel;
    Bevel1: TBevel;
    Bevel2: TBevel;
    procedure BStartSearchClick(Sender: TObject);
    procedure ListBox1DblClick(Sender: TObject);
    procedure JfsFileSearch1FoundFile(FileName: string; SR: TSearchRec);
  end;

var
  Form1: TForm1;

implementation

{$R *.DFM}

procedure TForm1.BStartSearchClick(Sender: TObject);
var
  RootDirectory, Mask, SaveCaption: String;
begin
  SaveCaption := Label3.Caption;
  JfsFileSearch1.FileList.Clear;
  ListBox1.Clear;

  Mask := MaskEdit.Text;
  RootDirectory := DirectoryListBox1.Directory;

  JfsFileSearch1.Run(Mask, RootDirectory);
  ListBox1.Items := JfsFileSearch1.FileList;
  Label3.Caption := SaveCaption;
end;

procedure TForm1.JfsFileSearch1FoundFile(FileName:
                        string; SR: TSearchRec);
begin
```

```
    Label3.Caption := FileName;
    Label3.UpDate;
end;

procedure TForm1.ListBox1DblClick(Sender: TObject);
var
    S: string;
begin
    S := 'Write ' + ListBox1.Items.Strings[ListBox1.ItemIndex];
    WinExec(PChar(S), sw_ShowNormal);
end;

end.
```

Test1 程序使用 TJfsFileSearch 控件来搜索文件夹。一旦找到一个新的文件夹或文件，TJfsFileSearch 控件就向程序发送事件。事件包括新文件夹或文件的名称，以及关于找到的文件的尺寸和日期的信息。可以随心所欲地对这些事件做出反应。Test1 程序不对找到文件夹的事件做出反应，而是对有关找到文件的事件做出反应。

例如：当找到一个具有正确后缀名的文件时，Test1 程序将做出这样的反应：

```
procedure TForm1.JfsFileSearch1FoundFile (FileName: string;
                                          SR: TSearchRec);
begin
    Label3.Caption := FileName;
    Label3.UpDate;
end;
```

可以看到，代码所做的工作不过是把文件名显示给用户。对 Update 的调用强迫标签显示文件，而不必等到处理器时钟周期的到来。

TJfsFileSearch 控件的内置功能之一就是维护它找到的文件列表。为了得到这个功能，必须将 UseFileList 属性设置为“True”。当搜索完所有文件夹后，就会得到一个想要的文件列表。这个列表是存放在 TStringList 对象中的，因此，可以把它指定到一个列表框的 Items 属性。摘自 BStartSearchClick 方法的片断如下所示：

```
JfsFileSearch1.Run ( Mask , RootDirectory ) ;
ListBox1.Items : = JfsFileSearch1. FileList ;
```

其中第一行传递希望搜索的文件通配符。例如，通常可能用 Start 参数传递*.pas。第二个参数 RunDir，指定搜索的起始文件夹。通过调用 Run，就开始了处理过程。每次找到一个满足要求的文件，就把它传递给 JfsFileSearch1FoundFile 事件处理程序并显示出来。运行结

束后，将能够访问像前面显示过的那样一个列表。

下面是一个调用 FileIterator.Run 的更加具体的代码范例：

```
JfsFileSearch1.Run ( ' * . pas ' , ' c : \ Source ' ) ;
```

这个例子将找到在 c: \ Sourece 文件夹中，或是在其子文件夹中的所有带有. pas 后缀的文件。

请注意，JfsFileSearch1FoundFile 方法是一个事件处理程序。使用控件的用户可以通过这个方法创建此处理程序，即在 TJfsFileSearch 的属性编辑器的 Events（事件）页面上单击一下 OnFoundFile 事件。TJfsFileSearch 既有找到文件的事件，也有找到文件夹的事件。Test1 程序只选择了找到文件的事件进行处理，它并不保存找到的文件夹列表。不过，它会遍历完在 Run 方法中用第二个参数传递的文件夹下的所有子文件夹。

当 TJfsFileSearch 控件结束了一轮运行之后，可以查看列表框中找到的文件。如果双击其中的一个文件，它就将被加载到 Windows Write 或 WordPad 程序中去。

```
procedure TForm1.ListBox1DblClick(Sender: TObject);
var
  S: string;
begin
  S := 'Write ' + ListBox1.Items.Strings[ListBox1.ItemIndex];
  WinExec(PChar(S), sw_ShowNormal);
end;
```

WinExec 进程启动一个可执行程序，然后再把控制权还给程序。其第一个参数用来传递文件的名称，以及想要传递的命令参数。第二个参数告诉 WinExec 想要以何种状态启动程序。例如，sw_ShowMinimized 将使程序在最小化的状态下启动。

现在人们已经不再习惯使用 WinExec 了。不过，它仍然是 WIN32 的一部分。Microsoft 正式指定用来实现这一功能的函数叫做 CreateProcess，不过它不太好用。下面给出了使用 CreateProcess 的 WinExec 版本，它叫做 WinExec2。

```
procedure WinExec2(ProgramToStart: string; Params: string; Show: Integer);
var
  StartupInfo: TStartupInfo;
  ProcessInfo: TProcessInformation;
begin
  if (Params <> '') and (Params[1] <> ' ') then
    Params := ' ' + Params;
  FillChar(StartupInfo, SizeOf(TStartupInfo), 0);
  StartupInfo.cb := SizeOf(TStartupInfo);
  StartupInfo.dwFlags := STARTF_USESHOWWINDOW;
  StartupInfo.wShowWindow := Show;
```

```
  if not (CreateProcess(PChar(ProgramToStart), PChar(Params), nil,
    nil, False, NORMAL_PRIORITY_CLASS, nil, nil,
    StartupInfo, ProcessInfo)) then
    RaiseLastWin32Error;
end;
procedure TForm1.Button1Click(Sender: TObejct);
begin
  WinExec2('c:\windows\notepad.exe', 'c:\autoexec.bat',SW_SHOWNORMAL);
end;
```

Button1Click 方法显示了应该如何调用这个函数。

我们建立的应用程序可以在硬盘的子文件夹里漫游以搜索一个特定名字文件或者搜索具有通配符的文件。例如，用户可以搜索*.PAS 或 m*.PAS，或者是 ole2.PAS。它将把与用户传递给它的通配符相匹配的所有文件都显示在一个列表框里面，如图 11.3 所示。然后用户就可以双击列表框里的任何一个文件，这样它就会显示在 WordPad 中。从那里，用户可以浏览文件，寻找特定的入口，或者做任何用户想要做的事情。

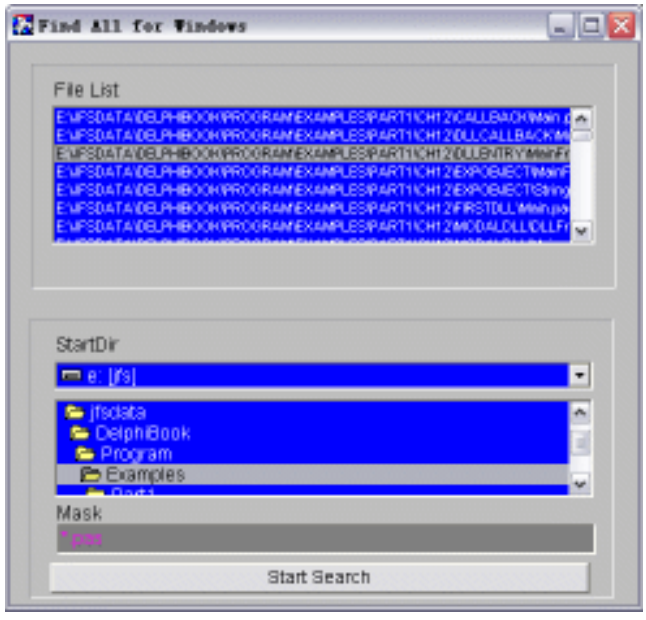

图 11.3　Test1 应用程序的运行界面

就像 TTable 和 TQuery 一样，TJfsFileSearch 是一个不可视的控件。正因为这样，在程序运行时，它不能在用户的窗体中出现，它只是在设计的时候才是可见的。TJfsFileSearch 控件也是很重要的，这是因为它向我们展示了如何创建一个定制的事件处理程序。

当创建了 TJfsFileSearch 之后，它可以被多个程序重用。这可以使得一个相对复杂的任务变得简单。用户只需要将对象拖到一个窗体上并将它插入到用户的程序中。结果是可视化

的工具将被用来处理诸如在文件夹中漫游这样的相对较复杂的细节问题。这是 OOP 和控件的成功，它能够制造一种工具，这种工具可以采用表面上看起来是不可捉摸的概念并且将这一概念真实地展示在用户的窗体上。

作为一个程序员所需要做的全部工作其实只是编写最后三个清单中的代码，其中包含了三个简短的方法。在 JfsFs 单元和 AllDirs 单元中的代码都被包装到一个控件中去了，可以把这个控件拖放到窗体上。这里的关键一点在于 Test1 程序的编写是如此容易，特别是如果考虑到其中的功能继承的话。

是否应该把对象变成控件并不是非常明显的。例如，TStringList 对象就没有任何相关的控件，也不能通过可见的工具操纵。所以在创建一个不可视控件之前，一定要考虑清楚，是否真的有必要创建一个不可视控件。例如在本程序中，定义了 TJfsFileSearch 控件，然后把它加入到 Delphi 的开发环境中，这样做会带来双重好处。

- 在使用对象之前，可能需要做出几个选择。特别是需要决定是否要把找到的文件夹和文件的列表保存到 TStringList 里的内存中去。如果让程序员通过单击属性来决定这些事情，将很难达到给对象提供简捷的、易于使用的接口的目的。
- TJfsFileSearch 对象有两个特性可以通过 Event 选项卡进行访问。特别是每当找到一个新文件或文件夹时，客户事件处理程序都会得到通知。不过，手工构造事件处理程序可能很令人困扰，特别是如果不知道该向涉及的函数传递哪些参数的话。如果把控件放到控件选项板上，程序员就不需要知道怎样处理事件了。需要做的工作仅仅是在 Event 选项卡上很快地单击一下，事件处理程序就会自动建立完成了。

建立控件还有一个好处，那就是它可以迫使或是诱使程序员去为对象设计简单的接口。当把一个对象放到控件选项板上之后，我们总是希望能够保证用户在短短的几秒中里就能把它同他的程序挂起钩来。因此我们会强烈地倾向于创造简单的、易于使用的接口。如果我们不把控件放到控件选项板上，就会发现很容易产生这样一种倾向，用一个复杂接口敷衍了事就算了，而这样的一个接口会让我们以及其他使用这个控件的人不得不编写很多行代码。按照我们的看法，好的控件不仅应该是没有漏洞的，而且它们还应该非常易于使用。

11.3 创建对话框控件

我们曾经提到，在 Delphi 的不可视控件中，典型的代表是对话框控件。那么现在我们也来创建一个对话框控件。在本章中介绍的控件和程序总是离不开文件和文件夹的处理，所以在这个控件中，仍然为它添加一些处理文件夹的功能。具体来说，这个控件会弹出一个定制的 Delphi 窗体。TOpenDialog 和 TColorDialog 是与它类似的控件，只不过它们弹出的是普通对话框，而这个控件弹出的则是 Delphi 对话框。

在这个例子中使用的控件叫做 TJFSPickDirDlg。它是一个非常简单的 TComponent 子类，

它允许用户选择一个文件夹，如图 11.4 所示。

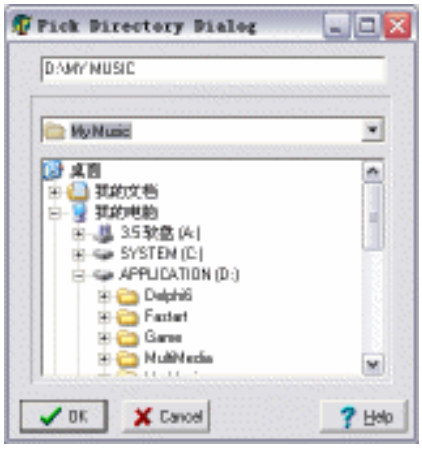

图 11.4　我们要创建的对话框控件

对话框的显示通常需要由程序来完成，Delphi 附带的控件中没有哪个能够让用户选择文件夹，只是提供了一些关于文件和文件夹处理的控件。

为了帮助理解这个控件，让我们在这个控件和我们熟悉的其他一些控件之间进行一下对比：

- TOpenDialog 让用户选择一个文件
- TColorDialog 让用户选择一种颜色
- TJFSPickDirDlg 让用户选择一个文件夹

实际上，这个控件所做的工作不过是建立一个 Delphi 窗体实例，把它显示在用户面前，并在窗体中继续用户的动作。像这样的一个控件并不难建立，但它的功能非常强大。本质上，它可以把 Delphi 的所有资源都放在一个窗体中，然后把这个窗体包在一个控件中。很显然，TJFSPickDirDlg 只使用了 Delphi 资源中的微不足道的一点点。

这个控件的代码如下所示：

```
unit JfsDialog;

interface

uses
  Windows, Messages, SysUtils,
  Classes, Graphics, Controls,
  Forms, Dialogs, PickDirectory2;

type
  TJFSPickDirDlg = class(TComponent)
```

```
  private
    { Private declarations }
    FPickDialog: TPickDirectory;
    function GetDirectory: string;
    procedure SetDirectory(const Value: string);
  protected
    { Protected declarations }
  public
    constructor Create(AOwner: TComponent); override;
    destructor Destroy; override;
    function Execute: Boolean;
  published
    property Directory: string read GetDirectory write SetDirectory;
    { Published declarations }
  end;

procedure Register;

implementation

procedure Register;
begin
  RegisterComponents('CustomControl', [TJFSPickDirDlg]);
end;

constructor TJFSPickDirDlg.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);
  FPickDialog := TPickDirectory.Create(AOwner);
end;

destructor TJFSPickDirDlg.Destroy;
begin
  inherited Destroy;
end;

function TJFSPickDirDlg.Execute: Boolean;
begin
  Result := False;
  if FPickDialog.ShowModal = mrOk then
    Result := True;
end;
```

```
function TJFSPickDirDlg.GetDirectory: string;
begin
  Result := FPickDialog.Directory;
end;

procedure TJFSPickDirDlg.SetDirectory(const Value: string);
begin
  FPickDialog.Directory := Value;
end;

initialization
  RegisterClass(TPickDirectory);
end.
```

实际上在这个控件中并没有关于如何选择文件夹的操作，这部分操作位于另外一个单元文件中，该文件实际上就是我们在前面看到的那个窗体。该单元文件的代码如下所示：

```
unit PickDir;

interface

uses
  Windows, SysUtils, Classes,
  Graphics, Forms, Controls,
  StdCtrls, Buttons, ExtCtrls,
  FileCtrl, ComCtrls, ShellCtrls;

type
  TPickDirectory = class(TForm)
    HelpBtn: TBitBtn;
    OkBtn: TBitBtn;
    CancelBtn: TBitBtn;
    Bevel1: TBevel;
    Edit1: TEdit;
    ShellComboBox1: TShellComboBox;
    ShellTreeView1: TShellTreeView;
    procedure HelpBtnClick(Sender: TObject);
    procedure OKBtnClick(Sender: TObject);
    procedure Edit1KeyDown(Sender: TObject; var Key: Word;
      Shift: TShiftState);
    procedure FormShow(Sender: TObject);
    procedure ShellComboBox1Click(Sender: TObject);
```

```
    procedure ShellTreeView1Click(Sender: TObject);
  private
    FDirectory: string;
    procedure SetDirectory(const Value: string);
  public
    function EditDirs(Dir1, Dir2: string): Boolean;
    property Directory: string read FDirectory write SetDirectory;
  end;

var
  PickDirectory: TPickDirectory;

implementation

uses
  Dialogs;

{$R *.DFM}

function TPickDirectory.EditDirs(Dir1, Dir2: string): Boolean;
begin
  if ShowModal <> mrOk then
    Result := False
  else begin
    FDirectory := Edit1.Text;
    Result := True;
  end;
end;

procedure TPickDirectory.HelpBtnClick(Sender: TObject);
begin
  Application.HelpContext(HelpContext);
end;

procedure TPickDirectory.OKBtnClick(Sender: TObject);
begin
  FDirectory := Edit1.Text;
  ModalResult := mrOk;
end;

procedure TPickDirectory.Edit1KeyDown(Sender: TObject; var Key: Word;
  Shift: TShiftState);
```

```
begin
  if Key = Ord(#13) then
    ShellComboBox1.Path := Edit1.Text;
end;

procedure TPickDirectory.FormShow(Sender: TObject);
begin
  Edit1.Text := ShellComboBox1.Path;
end;

procedure TPickDirectory.SetDirectory(const Value: string);
begin
  FDirectory := Value;
end;

procedure TPickDirectory.ShellComboBox1Click(Sender: TObject);
begin
Edit1.Text := ShellComboBox1.Path;
end;

procedure TPickDirectory.ShellTreeView1Click(Sender: TObject);
begin
  FDirectory := UpperCase(ShellTreeview1.SelectedFolder.PathName);
  Edit1.Text := FDirectory;
end;

end.
```

如果在把这个控件放进控件选项板之前想对它进行一下测试，可以建立一个简单的 Delphi **项目，在它的主窗体上放置一个按钮，对按下按钮的情况做出如下反应：**

```
procedure TForm1.Button1Click(Sender: TObject);
begin
PickDlg := TJFSPickDirDlg.Create(self);
if PickDlg.execute then begin
   showmessage(Pickdlg.directory);
end;
PickDlg.free;
end;
```

这些代码首先建立一个 TJFSPickDirDlag **实例。然后调用该控件的** Execute **方法向用户显示窗体。当用户选择完文件夹以后，控件的内存将被清除。**

控件中的关键方法是用于创建 Delphi **窗体实例，向用户显示窗体，并暗中摧毁窗体的那**

些方法。下面是建立窗体实例的例程：

```
constructor TJFSPickDirDlg.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);
  FPickDialog := TPickDirectory.Create(AOwner);
end;
```

可以看到，这段代码所做的全部工作只不过是调用 Delphi 窗体的 Create 方法。它把控件的属主作为窗体的属主用作参数。这样做了以后，控件和它的窗体就会在适当的时候被自动摧毁。

我们为这个窗体建立了析构函数，这只是因为在第一次编写控件的代码时，我们不小心释放了窗体。释放窗体是一个错误，因为窗体的属主稍后还会要破坏窗体，而我们是不希望同一个析构函数调用两次的。特别是，这样做通常会引发冲突。但是，我们发现很容易就会忘掉这个事实，因此这里特意明显地加入了析构函数，以便提醒不要破坏窗体：

```
destructor TJFSPickDirDlg.Destroy;
begin
  inherited Destroy;
end;
```

当然，如果愿意，也可以在这里破坏掉窗体，但是如果这么做，还需要将指向它的变量也设置为 nil：

```
FPickDialog . Free ;
FPickDialog : = nil ;
```

如果检查这个变量，Delphi 就会知道已经破坏了这块内存，于是就不会再试图重新进行这项工作了。在大多数情况下，像这样编写代码不会带来任何好处。不过，如果需要沿着这条道路编写代码，那么了解一下这个处理过程也不错。

剩下的另外一个重要的方法就是实际弹出 Delphi 窗体的方法：

```
function TJFSPickDirDlg.Execute: Boolean;
begin
  Result := False;
  if FPickDialog.ShowModal = mrOk then
    Result := True;
end;
```

在本质上，这段代码仅仅是在 Delphi 窗体上调用了 ShowModal 而已。这个布尔函数同时也负责对用户是选择了 OK 按钮还是 Cancel 按钮做出反馈。

在设计 Execute 方法时，我们有意模仿了 TOpenDialog 的动作。之所以这样做是因为开发人员对它都很熟悉。设计良好的控件有一条原则，那就是应该尽可能地利用用户对现有控

件的知识。人们在测试控件时，总是希望能很快地掌握和使用它。如果他们觉得它诡异或奇特，则可能会随手就把它扔到一旁，而去找自己能更快理解的控件。所以我们建议让控件界面尽可能熟悉和简单，尽可能地学习和遵循传统习惯。把创新冲动留到控件的特性设置或控件的功能实现上去吧。

下面再简要介绍一下 OkBtnClick 方法。当用户选择一个文件夹的时候，他们就会选择这样一个方法。

```
procedure TPickDirectory.OKBtnClick(Sender: TObject);
begin
  FDirectory := Edit1.Text;
  ModalResult := mrOk;
end;
```

这段简单的代码将窗体的字符串变量 FDirectory 设置为用户选定的文件夹名。每当用户在 TDirectoryListBox 控件中选择一个文件夹时，Edit1 文本域都会得到这个值：

```
procedure TPickDirectory . DirectoryListBox1Change ( Sender : TObject ) ;
begin
  Edit1 . Text : = DirectoryListBox1. Directory ;
end ;
```

然后，TJFSPickDirDlg 就可以通过窗体的 Directory 属性来得到这个值。

总而言之，最后这个例子非常简单。我们在这里使用了这个例子是因为它展示了很重要的一点，那就是可以把完整的一个窗体包在一个控件当中。

11.4　ActiveX 控件

在结束本章以及本部分内容之前，我们打算简单地说几句关于建立 ActiveX 控件的问题。关于这个题目，可繁可简，而且是一个不容易处理的问题。如果愿意，针对这个主题我们可以用一本书来展开。但是在这里我们只是尽量简要地给出创建这种控件的一些基本知识。本章的重点是创建 Delphi 的控件。

在 Delphi 中建立 ActiveX 控件极为直接。可以按照下面的步骤创建 ActiveX 控件。

(1) 首先选择 File | Close All，把 Delphi 中打开的所有活动的项目关掉。

(2) 从 Delphi 菜单中选择 File | New | Other，然后从弹出的对话框中选择 ActiveX 选项卡，此时的对话框如图 11.5 所示。

(3) 选择 ActiveX Control。此时会看到如图 11.6 所示的 ActiveX 控件向导。

(4) 在向导对话框的顶端，选择 TCustomShellTreeView 控件作为放置控件的基础 VCL 类。系统将会自动地为用户填写好一个默认名称、实现单元以及项目单元。例如在我们的例子中

全部采用默认的名称。

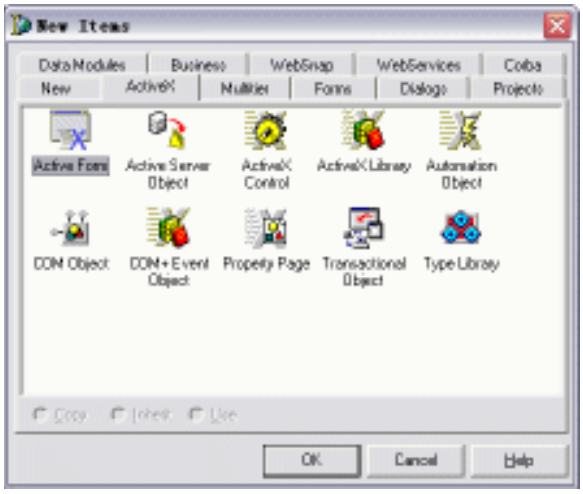

图 11.5 Delphi 中的 New Items 对话框

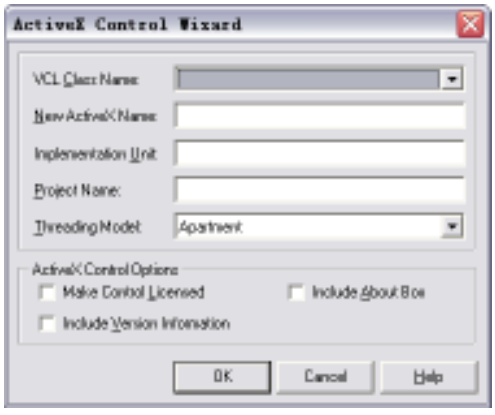

图 11.6 ActiveX 控件向导对话框

(5) 还可以给该控件添加设计时间许可证、版本信息和一个 About 对话框。如果想添加这些项目，只要选中相应的复选框就行了。如果选中了时间许可证复选框，那么将会生成一个许可证文件，例如在这里可能是 CustomShellTreeViewImpl1.lic。它是 ActiveX 控件的许可证文件，它必须存在，否则控件就无法正常工作。

现在可以在 Delphi 中编译控件并进行测试了。要想编译控件，请选择 Project | Build。要想注册控件，请选择 Ru | Register ActiveX Sever。

要想把控件安装到 Delphi 环境中，请选择 Component | Import ActiveX 控件。这时将会出现一个 Import ActiveX 对话框，如图 11.7 所示。

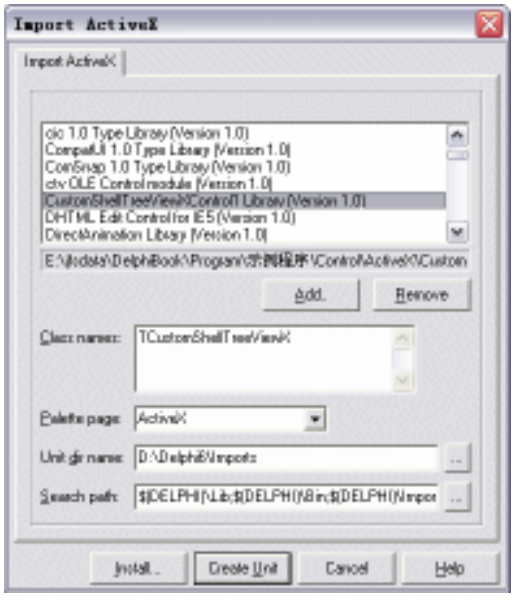

图 11.7　Import ActiveX 对话框

在对话框中列表框的上部将会出现该 ActiveX 控件。如果没有也没有关系，可以单击对话框上的 Add 按钮，找到刚刚创建的 OCX 文件，然后使用 Install 按钮选择使用这个控件。

如果愿意，可以把它放在我们一直使用的 CustomControl 包中，或者也可以浏览 Delphi 6 \ Lib 文件夹，把它放在 dclusr.dpk 中。DclUsr 是一个给“用户”存放控件的通用包。当然以后我们也可以利用 Components 菜单中的 Configure Palette 命令来改变一个控件所在的选项卡。在本例中，使用我们习惯的 CustomControl 选项卡就行了。

完成这些工作以后，在控件选项板的 CustomControl 选项卡上将出现一个新的控件。启动一个新工程，然后把这个控件放在上面，使用方法和使用其他控件一样，如图 11.8 所示。

这个控件还可以用于其他的一些程序中，例如 Word 、Excel、Microsoft Visual C++，或者还可以是 Visual Basic。例如在 Word 2002 中，可以按照下面的步骤在文档中使用我们创建的 ActiveX 控件。

(1) 从 Word 2002 的菜单中选择【工具】|【宏】|【Visual Basic 编辑器】。

(2) 然后在 Visual Basic 菜单中选择【视图】|【工程资源管理器】。在工程资源管理器中找到相应的项目（可以是任意一篇文档）。

(3) 在它上面右击，选择【插入】|【添加用户窗体】。于是将出现一个窗体和控件工具箱。

(4) 在工具箱上右击，并选择附加控件选项。然后就会看到一个对话框，可以在这个对话框中选择我们创建的 AcitveX 控件。该控件前的复选框上将出现复选标记，如图 11.9 所示。

(5) 单击【确定】按钮，我们设计的时钟控件将出现在工具箱中，可以把它放置在 VB 窗体中。使用该控件的方法就和在 Delphi 中所做的一样。选择【视图】|【属性窗口】调出

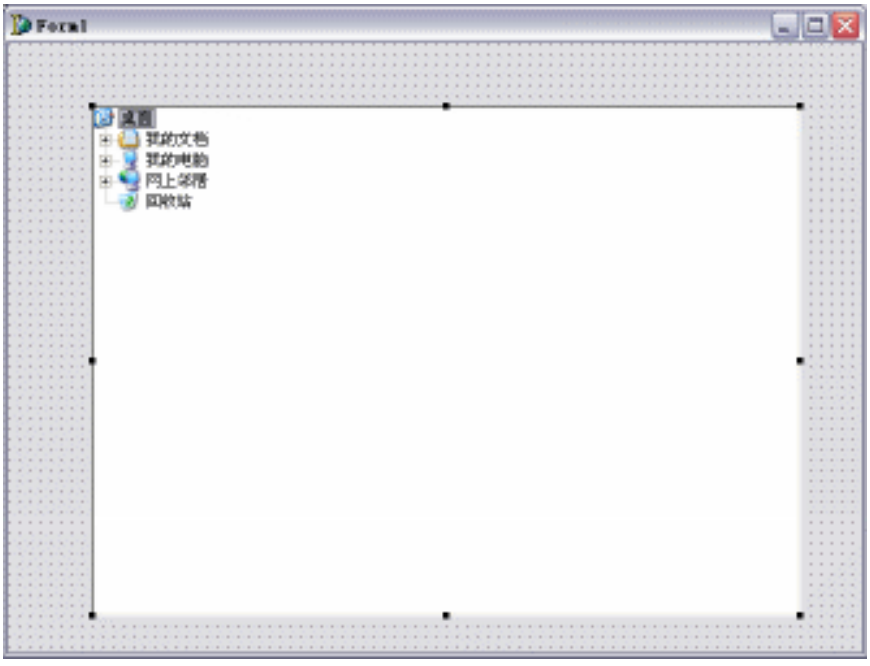

图 11.8　在工程中使用我们自己创建的 ActiveX 控件

图 11.9　Word 的 Visual Basic 编辑器中的【附加控件】对话框

VB 中相当于 Delphi 中的属性编辑器那样的一个工具【属性窗口】。可以使用这个工具来为控件设置属性，甚至处理与控件关联的事件，如图 11.10 所示，在这个窗口中，同样显示了在前面定义的控件属性。

选择【文件】|【关闭并返回 Microsoft Word】。在 Word 中，选择【视图】|【工具栏】、【控件工具箱】，可以调出一个和 VB 中一样的工具箱。选择一个按钮控件并把它放在文档上面。双击这个按钮，建立一个 VB 事件处理程序。

编辑这段事件处理程序，把它修改成下面这样：

```
Private Sub CommandButton1_Click ( )
    UserForm1 . Show
End sub
```

现在可以关闭 VB 了。选择控件工具箱左上部的图标就可以切换进或切换出设计模式。如果需要，则可以选择 View | Page Layout，这样可以看到那个按钮。现在单击这个按钮，一个带有标签编辑控件的窗体将会显示出来，如图 11.11 所示。

图 11.10　VB 编辑器中的时钟控件的属性

在 Delphi 中制造工具，然后在 Word、VB ，甚至 IE 中使用这些工具的能力大大扩展了计算机的能力。开发人员可以越来越容易地编写有关操作系统所有方面以及操作系统支持的关键工具程序。随着我们在 Windows 上获得越来越完善的控件，以后将能够创造出强大的交互系统，它们可以自由地利用系统上可用工具的最好功能。

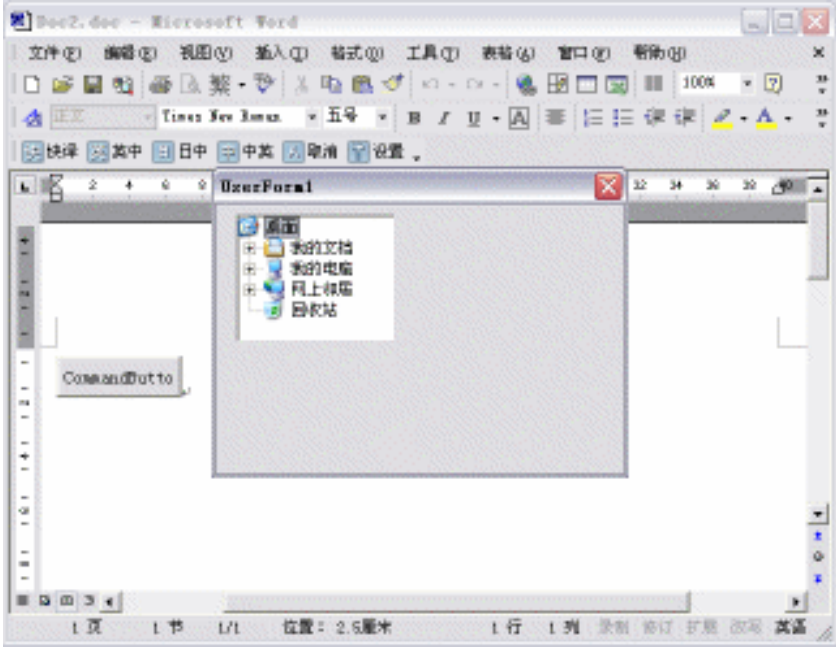

图 11.11　带有一个按钮的 Word 文档和按下这个按钮显示的含有 JfsClockX 控件的对话框

虽然在上面介绍的过程是比较简单的，但是这正代表了目前最常使用的 ActiveX 控件的开发过程，它主要分为以下几步。

(1) 在 Delphi 中建立控件。

(2) 使用向导把它转变成 ActiveX 控件。

(3) 把控件引入到选择的工具中。

如果想更进一步学习，不花上几个月的时间，起码也得花上几周的时间来深入挖掘 ActiveX 的细节。这不是本书的目的，所以我们就不做进一步的讨论了。

11.5 本 章 小 结

在本章的内容中，通过几个控件的例子，介绍了关于 Delphi 的控件设计中可能会使用的技术，同时也为读者提供了几个控件的示例。

在创建控件的过程中，需要的技术是多方面的，我们不可能在一章的内容中介绍全面。事实上在 Delphi 编程中可能使用到的技术，在控件设计中几乎都可以使用到。

控件的开发是一项比较热门的技术，可以在网上找到许多关于控件开发的站点，需要的读者可以到网站上进行搜索。

沟　　通

亲爱的读者朋友：

谢谢您选择本书。

作为编辑的职责驱使我们去尽力制作一些能给读者带来更多帮助的书籍。虽然，我们需要探索的路会很长，但请您相信，我们一直在努力。

计算机业的发展，其主要成就之一就是给了人们更多沟通的空间。

既然我们结缘于本书，那么，您大可不必吝于挥动您的钢笔或敲击您的键盘，将您的想法和看法，传递给我们。

也许，我们的进步和您的意外收获均将得益于此。

联系地址： 北京海淀清华大学出版社计算机编辑室（100084）

传真： 010—62771155　Email: tianzr@tup.tsinghua.edu.cn

Borland Delphi 程序设计

个人资料：

姓名：_____性别：1□男 2□女　　出生年月（或年龄）：_____

职业：_____文化程度：_____通讯地址：_____

电话（或寻呼）：_____传真：_____电子信箱（E-mail）：_____

您是如何得知本书的：

□别人推荐　□书店　□出版社图书目录

□杂志、报纸等的介绍（请指明）_____

□其他（请指明）_____

您从何处购得本书：

□书店　□电脑商店 □软件销售处

□邮购　□商场　□其他_____

影响您购买本书的因素（可复选）：

□封面封底　□装帧设计　□价格

□内容提要、前言或目录　□书评广告

□出版社名声　□作者名声　□责任编辑

□其他_____

您对本书封面设计的满意度：

□很满意　□比较满意　□一般　□较不满意

□不满意　□改进建议_____

您对本书印刷质量的满意度：

□很满意　□比较满意　□一般　□较不满意

□不满意　□改进建议_____

您对本书的总体满意度：

从文字角度　□很满意　□比较满意

□一般　□较不满意　□不满意

从技术角度　□很满意　□比较满意

□一般　□较不满意　□不满意

本书最令您满意的是：

_____

_____

您希望本书在哪些方面进行改进：

_____

_____

您是否希望本书配光盘：

□希望　□不希望　□无所谓

您对使用中文版软件或外文版软件介意吗？更喜欢用哪一种版本？

□介意　□无所谓；　□中文版　□外文版

您对图书所用软件版本是否很介意？是否要求用最新版本？

□是，要求是最新版本　□无所谓

□不，因为硬件或软件跟不上要求

您更喜欢阅读哪些类型和层次的计算机书籍？

□入门类　□提高类　□技巧类

□实例类　□精通类　□综合类

您的其他要求：

_____

_____