北京科海培训中心

新概念 Delphi 6.0 教程

杨　宇　编著

北京科海集团公司　　出品

2001.10

内 容 提 要

Delphi 6.0 是 Borland 公司最新推出的一套无论是界面还是功能都近乎完美的应用程序开发工具。与以前的 Delphi 版本相比，Delphi 6.0 使用更简便，效率也更高。

本书共分 17 章，详细介绍了 Delphi 开发过程中必须了解的重要知识点。通过本书的学习，读者不仅可以快速领悟 Delphi 6.0 的编程要旨，还能够掌握许多新技术。

本书内容丰富、通俗易懂，叙述深入浅出。另外，本书精心配置的多媒体教学光盘，附有书中所有的操作实例，对操作过程进行了详细地讲解，便于读者理解和学习。

本书可作为初、中级 Delphi 用户的自学教程和参考用书，也可作为初级 Delphi 用户的培训教材。

书　　名：新概念 Delphi 6.0 教程
作　　者：杨　宇
责任编辑：高卫平
出　　品：北京科海集团公司
印 刷 者：北京门头沟胶印厂
发　　行：新华书店总店北京科技发行所
开　　本：787×1092　1/16　　印张：31　　字数：723 千字
版　　次：2001 年 10 月第 1 版　　2001 年 10 月第 1 次印刷
印　　数：0001~5000
盘　　号：ISBN 7-89998-017-8
定　　价：45.00 元（1 张多媒体光盘）

前　言

学习计算机高级语言的目的，就是使用最快最好的语言工具来开发高效的应用程序，基于这一点，本书向广大读者真诚推荐 Delphi 6.0 这一最新应用程序开发工具。

自 Windows 问世以来，其漂亮的 GUI 界面、多线程的并发机制、高效的内存管理和丰富的多媒体服务，吸引了无数的用户向 Windows 靠拢，同时，基于 Windows 的程序设计方法也在不断完善。VB 的出现，使开发 Windows 应用程序变得简单方便，但是由于 Basic 自身的缺陷，很多工作都难以实现。虽然 Borland C++比 VC++中使用的 C 语言功能强大得多，可是它所依赖的 OWL 和 MFC 类库却非常复杂，使用起来相当不易。此时 Borland 公司的快速开发工具 Delphi 则应运而生。Delphi 以其具有高效、舒适、方便的开发环境迅速风靡全球。可以说，Delphi 是 Windows 中最完美的开发工具之一。

Delphi 是一套非常优秀的编程工具，尽管它在竞争激烈的编程工具中属于后起之秀，但自从它诞生以来，已逐渐赢得了众多程序员的青睐。值得一提的是，除了 Delphi 自身为开发者提供的组件外，Delphi 爱好者为用户提供了大量的第三方组件，这些组件成为 Delphi 开发的最为宝贵的资源。和其他编程工具相比，在这一点上，Delphi 实现了真正意义上的开发资源共享。

从程序员角度看，Delphi 6.0 是一套无与伦比的快速、高效、美观的程序开发工具；而从 Delphi 6.0 自身的角度看，它又是一个编译神速、构架坚固的开发利器。如果用户在此之前曾经使用过别的开发工具的话，将会对 Delphi 6.0 的这些特性赞叹不已。

另外，随着新一代开放型操作系统 Linux 的诞生和蓬勃发展，我们越来越感到需要一种跨平台的开发工具，以适应今后多种平台共存形势的需要。Borland Kylix，作为惟一基于 Linux 的组件化、全编译、纯 OOP 的快速开发工具，成了当前最具技术潜力和市场潜力的工具。而 Delphi 6.0 和 Kylix 几乎如出一辙的开发界面和开发方式，使得现在学习 Delphi 6.0 具有了更特殊的意义。

本书从实际开发的角度出发，将 Delphi 6.0 的使用和作者多年的开发实践经验紧密结合起来，力图帮助读者以最快的速度进入 Delphi 6.0 的世界，掌握其理论知识，提高 Delphi 开发技术水平。

本书提供了大量实用性源代码，在看本书时，可以一边学习理论知识，一边上机实践程序，也就是把阅读文字和运行程序结合起来，最终使读者利用学到的知识创建自己的程序。

由于本人水平有限，书中难免有错误和疏漏之处，敬请读者批评指正。

编者

2001 年 10 月

光盘使用说明

为了提高读者学习 Delphi 6.0 的效率，更好地发挥出计算机的学习功能，我们特意在此书的基础上制作了一张多媒体教学光盘。在光盘中，我们对书中的知识点和使用技巧进行了归纳和总结，并且还使用了播放动画的形式对书中的实际操作过程进行了全真实环境的演示，再配以真人发音进行讲解。使读者能身临其境，快速地学会使用 Delphi 6.0。

运行环境：

- Pentium 166以上的处理器
- VGA显卡
- 光盘驱动器
- Windows 98/NT/2000/XP操作系统
- IE 4.0及以上版本浏览器
- 800×600分辨率
- 16位真彩色以上显示模式

操作方法：

- 一般情况下，将本光盘放到光驱中后，光盘就会自动运行。
- 如果本光盘没有自动运行，请双击光盘根目录下的Play.exe文件即可启动运行。
- 如果要查看使用说明，请双击光盘根目录下的Help.htm文件即可启动运行。

光盘内容：

Play.exe：运行主文件
help.htm：帮助主文件
Data：多媒体光盘数据文件夹
Example for Book：书中程序源文件文件夹
Example for CD：多媒体光盘演示程序源文件文件夹
Component：书中用到的第三方组件文件夹。

目　录

第 1 章　Delphi 6.0 入门 1
1.1 Delphi 的历史回顾 1
1.2 Delphi 6.0 的新特性 2
1.3 Delphi 6.0 的菜单 5
1.3.1 主菜单 5
1.3.2 File 菜单 5
1.3.3 Edit 菜单 7
1.3.4 Search 菜单 9
1.3.5 View 菜单 10
1.3.6 Project 菜单 18
1.3.7 Run 菜单 22
1.4 Delphi 6.0 的工具栏 25
1.5 组件选项板 26
1.6 Delphi 6.0 的主编辑器 27
1.7 Delphi 6.0 中的快捷方式 30
1.8 本章小结 31

第 2 章　Object Pascal 语言 32
2.1 Object Pascal 程序框架 32
2.1.1 工程主程序框架 32
2.1.2 程序单元框架和语法 33
2.1.3 程序单元引用方式与 Uses 子句 35
2.2 Object Pascal 语法元素 36
2.3 注释 36
2.4 变量 37
2.5 常量 38
2.6 运算符 39
2.6.1 赋值运算符 39
2.6.2 比较运算符 39
2.6.3 逻辑运算符 40
2.6.4 算术运算符 40
2.6.5 按位运算符 41
2.7 Object Pascal 数据类型 41
2.7.1 类型的比较 41

2.7.2 字符和字符串 42
2.8 用户自定义类型 43
2.8.1 数组 43
2.8.2 动态数组 44
2.8.3 记录 45
2.8.4 集合 45
2.8.5 对象 47
2.8.6 指针 47
2.9 条件语句 48
2.9.1 If 语句 48
2.9.2 Case 语句 49
2.10 循环结构 50
2.10.1 For 循环 50
2.10.2 While 循环 50
2.10.3 Repeat…Until 循环 51
2.10.4 Break 语句 51
2.10.5 Continue 语句 52
2.11 过程和函数 52
2.12 包 53
2.13 面向对象编程 54
2.14 Delphi 对象 55
2.14.1 声明和创建实例 55
2.14.2 析构 56
2.15 方法 56
2.15.1 重载 57
2.15.2 特性 57
2.15.3 代码可见性 58
2.16 接口 59
2.17 异常处理 63
2.17.1 异常类 65
2.17.2 触发异常 67
2.18 运行期类型信息 67
2.19 本章小结 68

第 3 章 Delphi 应用程序框架和设计 69
3.1 Delphi 环境和项目的体系结构 69
3.2 构成 Delphi 项目的文件 69
3.2.1 项目文件 69
3.2.2 单元文件 70

3.2.3 Form 文件 70
3.2.4 资源文件 72
3.2.5 项目选项和桌面设置文件 72
3.2.6 包文件 73
3.3 Delphi 项目管理 73
3.3.1 一个项目一个目录 73
3.3.2 代码中被共享的单元 73
3.3.3 多项目管理 74
3.4 项目选项设置 75
3.5 Delphi 项目的框架类 80
3.5.1 TForm 80
3.5.2 有模式的 Form 81
3.5.3 无模式的 Form 81
3.5.4 管理 Form 的图标和边框 81
3.5.5 Tapplication 类型 83
3.5.6 TApplication 的特性 83
3.5.7 TApplication 的方法 84
3.5.8 TApplication 的事件 85
3.5.9 TScreen 类 85
3.6 MDI 应用程序 86
3.6.1 创建 MDI 应用程序 86
3.6.2 使用菜单 87
3.6.3 隐藏一个子 Form 88
3.7 公共体系结构 88
3.7.1 应用程序的体系结构 88
3.7.2 Delphi 固有的体系结构 89
3.7.3 体系结构的示例 89
3.8 程序窗体设计 90
3.8.1 显示程序的启动界面 90
3.8.2 限制窗体的大小 91
3.8.3 实现窗体拖动 91
3.8.4 Form 生成的顺序 91
3.8.5 停靠窗口 92
3.9 本章小结 95

第 4 章　代码标准规则 96
4.1 源代码格式规则 96
4.1.1 缩进 96
4.1.2 边距 96

4.2 Object Pascal 97
4.2.1 括号 97
4.2.2 过程和函数 97
4.2.3 例程中的形参 97
4.2.4 变量 98
4.2.5 类型 99
4.2.6 构造类型 99
4.2.7 语句 100
4.2.8 结构化异常处理 101
4.2.9 类 102
4.3 文件 103
4.3.1 项目文件 103
4.3.2 Form 文件 103
4.3.3 数据模块文件 103
4.3.4 单元文件 103
4.4 Form 和数据模块 104
4.4.1 Form 104
4.4.2 数据模块 105
4.5 包 105
4.5.1 运行期包与设计期包 105
4.5.2 文件命名标准 105
4.6 组件 105
4.6.1 自定义组件 105
4.6.2 组件实例的命名规则 106
4.6.3 组件的前缀 106
4.7 本章小结 111

第 5 章 使用 Delphi 6.0 的组件 112
5.1 Standard 组件组 112
5.1.1 TFrame 113
5.1.2 TMainMenu 和 TPopMenu 菜单 115
5.1.3 TLabel、TEdit 和 TlabeledEdit 组件 118
5.1.4 TCheckBox 和 TradioButton 组件 118
5.1.5 TListBox 和 TcomboBox 组件 119
5.1.6 Tpanel 组件 120
5.1.7 TactionList 组件 121
5.2 Additional 组件组 121
5.2.1 TmaskEdit 组件 122
5.2.2 Timage 组件 122

5.2.3 Tsplitter 组件 123
5.2.4 TapplicationEvents 组件 123
5.2.5 TActionManager 组件 124
5.3 Win32 组件组 126
5.3.1 TpageControl 组件 126
5.3.2 TimageList 组件 129
5.3.3 TstatusBar 组件 131
5.3.4 TMonthCalendar 和 TdateTimePicker 组件 131
5.3.5 TComboBoxEx 组件 132
5.4 Dialogs 组件组 133
5.5 本章小结 135

第 6 章 VCL 组件基础 136
6.1 VCL 应用框架 136
6.2 组件简介 137
6.3 组件的种类 138
6.4 组件的结构 139
6.4.1 组件的特性 139
6.4.2 组件的事件 140
6.4.3 组件的拥有关系 141
6.4.4 组件的父子关系 142
6.5 组件的继承关系 142
6.5.1 TPersistent 类 143
6.5.2 TComponent 组件 143
6.5.3 TControl 类 145
6.5.4 TGraphicControl 类 145
6.5.5 TWinControl 类 145
6.5.6 TCustomControl 类 146
6.5.7 VCL 助手类 146
6.6 运行期类型信息 150
6.6.1 TypInfo.pas 单元 151
6.6.2 获取 RTTI 153
6.6.3 检查特性 157
6.7 本章小结 158

第 7 章 编写自己的组件 159
7.1 组件设计基础 159
7.1.1 编写组件的时机 159
7.1.2 编写组件的步骤 159
7.1.3 确定组件的祖先类 160

7.1.4 创建组件单元 161
7.1.5 加入新的特性 162
7.1.6 向组件中加入事件 171
7.1.7 加入自定义的方法 173
7.1.8 构造和析构 174
7.1.9 注册组件 175
7.1.10 组件图标 176
7.2 定制组件 176
7.3 复合组件 177
7.4 组件包 181
7.4.1 使用包的好处 182
7.4.2 包的类型 182
7.4.3 包文件 182
7.4.4 安装包 183
7.4.5 设计包 184
7.4.6 维护包的版本 185
7.5 本章小结 185

第 8 章 异常处理 187
8.1 异常理论 187
8.1.1 错误处理方法 187
8.1.2 Try…Finally 块 188
8.1.3 Try…Except 块 190
8.1.4 混合使用资源保护和异常处理 192
8.1.5 异常处理的必要性 194
8.2 异常类 195
8.3 异常的实例 197
8.3.1 一个异常的实例 197
8.3.2 找到异常的地址 200
8.4 引发异常 200
8.4.1 引发 VCL 异常类 200
8.4.2 创建和引发异常 201
8.4.3 再次引发异常 203
8.5 高级异常处理技术 204
8.5.1 事件驱动环境下的异常 204
8.5.2 哑异常 205
8.5.3 应用对象的错误处理 205
8.6 处理数据库异常 206
8.6.1 EDatebaseError 和 EDBEngineError 异常 206

8.6.2 OnPostError()、OnEditError()和 OnDeleteError()事件 207
8.6.3 错误常量 207
8.6.4 自定义数据库服务器异常 209
8.7 本章小结 210

第 9 章 动态链接库 211
9.1 DLL 简介 211
9.2 静态链接和动态链接 212
9.3 使用 DLL 的必要性 212
9.3.1 共享代码、资源和数据 213
9.3.2 隐藏实现的细节 213
9.3.3 自定义控件 213
9.4 创建和使用 DLL 213
9.4.1 创建 DLL 213
9.4.2 定义接口单元 215
9.5 在动态库中显示 Form 215
9.5.1 显示模式 Form 216
9.5.2 显示无模式 Form 217
9.6 DLL 的入口和出口函数 218
9.7 本章小结 221

第 10 章 数据库开发 222
10.1 配置 ODBC 数据源 222
10.2 Borland 数据库引擎 225
10.2.1 BDE 管理器 225
10.2.2 Databases 选项卡 228
10.2.3 Configuration 选项卡 229
10.2.4 分发 BDE 230
10.3 数据库应用程序体系结构 230
10.3.1 数据集 230
10.3.2 BDE 数据访问组件 230
10.3.3 数据库的连接 235
10.4 数据库应用程序实例 236
10.4.1 打开和关闭数据集 236
10.4.2 浏览数据集 236
10.4.3 一个实例 237
10.4.4 对数据集的操作 240
10.4.5 TField 类型 241
10.4.6 字段编辑器 241
10.4.7 计算字段和查找字段 244

10.4.8 过滤器 246
10.4.9 主从表 247
10.5 数据模块 248
10.6 SQL 语句 248
10.6.1 SQL 语句语法 248
10.6.2 动态 SQL 250
10.7 连接数据库 252
10.8 dbExpress 255
10.9 本章小结 258

第 11 章 COM 基础 259
11.1 COM 基础 259
11.1.1 COM（组件对象模型） 259
11.1.2 COM 的问题和未来 261
11.1.3 COM、OLE 和 ActiveX 的异同 261
11.1.4 COM 技术中的术语 261
11.1.5 COM 的线程模式 262
11.2 接口 262
11.2.1 接口简介 262
11.2.2 声明接口类型 264
11.2.3 实现接口 265
11.2.4 使用接口的原因 270
11.2.5 接口的维护和更新 271
11.2.6 理解 IUnknown 272
11.2.7 IDispatch、双重接口和 DispInterface 279
11.2.8 HResult 类型 280
11.2.9 虚方法表 280
11.3 COM 对象和类工厂 280
11.3.1 TComObject 和 TComObjectFactory 281
11.3.2 进程内 COM 服务器 281
11.3.3 进程外 COM 服务器 282
11.4 DCOM（分布式 COM） 282
11.5 COM Automation 283
11.5.1 创建 Automation 服务器 284
11.5.2 创建 Automation 控制器 287
11.6 TOleContainer 289
11.6.1 一个简单的程序示例 289
11.6.2 OLE 对象的操作方法 291
11.7 本章小结 291

第 12 章 DCOM 292
12.1 COM 和分布式体系结构 292
12.1.1 DCOM 简介 292
12.1.2 DCOM 的系统设置 293
12.2 DCOM 服务器和客户程序 293
12.2.1 创建 DCOM 服务器 293
12.2.2 理解 Safecall 300
12.2.3 创建 DCOM 客户程序 302
12.2.4 深入 DCOM 305
12.3 本章小结 307
第 13 章 分布式编程 308
13.1 MIDAS 多层应用 308
13.1.1 MIDAS 的概念 308
13.1.2 MIDAS 的核心技术 309
13.1.3 简单理解 MIDAS 313
13.1.4 MTS、COM/DCOM、CORBA、OLEnterprise 313
13.1.5 MIDAS 的应用和未来 315
13.2 分布式应用程序基础 315
13.2.1 DataSnap 组件组 315
13.2.2 建立 3 层 MIDAS 结构 317
13.2.3 创建 MIDAS 服务器 318
13.2.4 理解服务器 326
13.2.5 创建和理解 MIDAS 客户程序 326
13.2.6 远程访问服务器 328
13.3 建立一对多应用程序 329
13.3.1 创建步骤 329
13.3.2 刷新和更新数据 330
13.3.3 公文包模式 331
13.3.4 PacketRecords 331
13.4 错误处理 333
13.5 服务器端和客户端逻辑 334
13.6 本章小结 336
第 14 章 创建 ActiveX 控件 337
14.1 创建 ActiveX 控件的原因 337
14.2 创建 ActiveX 控件 337
14.2.1 ActiveX 控件向导 338
14.2.2 ActiveX 控件示例 339
14.2.3 ActiveX 框架 349

14.3 ActiveX 控件在 Web 上的应用 351
14.3.1 与 Web 浏览器交互 351
14.3.2 Web 分发 351
14.4 本章小结 354

第 15 章 文件处理 355
15.1 文件的输入/输出 355
15.1.1 文本文件 355
15.1.2 处理有类型文件 358
15.1.3 处理无类型文件 366
15.2 TTextRec 和 TFileRec 结构 368
15.3 驱动器和目录 369
15.3.1 获得驱动器列表 370
15.3.2 获得驱动器信息 371
15.3.3 获取 Windows 目录 373
15.3.4 获取系统目录 374
15.3.5 获取当前目录 374
15.3.6 从目录中查找文件 375
15.3.7 复制和删除目录 378
15.4 内存映射文件 380
15.4.1 内存映射文件的作用 380
15.4.2 创建内存映射文件 381
15.5 本章小结 387

第 16 章 图像编程 388
16.1 GDI 和 TCanvas 类 388
16.1.1 理解 GDI 388
16.1.2 画笔 TPen 389
16.1.3 TCanvas.Pixels 特性 391
16.1.4 TCanvas.Brush 特性 391
16.1.5 Font 字体 392
16.1.6 TCanvas.CopyMode 特性 393
16.2 TCanvas 的方法 396
16.2.1 TCanvas 画线 396
16.2.2 TCanvas 画几何形状 396
16.2.3 TCanvas 输出文字 397
16.2.4 定制图形 401
16.2.5 设备描述表 403
16.3 坐标系统和映射模式 404
16.3.1 设备坐标系 404

16.3.2 逻辑坐标系 404
16.3.3 屏幕坐标系 404
16.3.4 Form 坐标系 405
16.3.5 坐标映射 405
16.3.6 设置映射模式 407
16.3.7 设置窗口/视区范围 407
16.4 高级字体 408
16.4.1 Win32 字体类型 409
16.4.2 基本字体元素 409
16.4.3 GDI 字体分类 410
16.4.4 显示不同字体 411
16.5 本章小结 411
第 17 章 多媒体编程 412
17.1 一个简单的媒体播放器 412
17.2 播放 WAV 文件 413
17.3 播放 AVI 文件 413
17.4 设备支持 415
17.5 音频 CD 播放器 416
17.5.1 编写 CD 播放器程序 416
17.5.2 获取 CD 播放信息 417
17.6 本章小结 420
附录 1 Delphi 函数方法参考手册 421
附录 2 Win32 API 函数库 429
附录 3 Delphi 网络资源 476

第 1 章　Delphi 6.0 入门

作为本书的第一章，本章将介绍使用 Delphi 6.0 以及 Delphi 6.0 的 IDE（集成开发环境）与其他语言和开发环境进行 Windows 和 NT 编程的区别。本章的内容主要是为应用 Delphi 6.0 编程的初学者编写的。如果读者过去使用过 Delphi 的早期版本，并且有一定的编程经验，本章开始将做一个简单的回顾。

今天，Delphi 已经是一个 Microsoft Windows、Windows NT 以及 Linux 下的 RAD（Rapid Application Development，快速应用程序开发环境）和强大的数据库开发工具，它综合了可视化开发环境的易用性，32 位优化编辑技术的快速、强大以及数据库引擎的可伸缩性等特点。当然，这些技术都不是 Delphi 所特有的，但 Delphi 是惟一把这些主流技术无缝集成在一起的开发工具。

Delphi 是一个可视化的开发工具，用户只要从组件选项板上选择一个组件，然后把它放到 Form（窗体）上，就可以构成一个应用程序了。最让人惊叹的是，当用户把组件放到 Form 上的时候，Delphi 会自动为用户生成相应的代码。和其他开发工具相比，Delphi 是真正面向对象的，用户可以方便地扩展组件的功能，并集成到 IDE 中，以满足自己的需要。例如，如果用户发现一个组件的功能在一个方面很强，而在另一个方面却不尽人意，这时就可以在这个组件的基础上派生出一个新的组件；然后加入自己的代码来加强不如人意的方面，完善这个组件；最后，用户可以将新生成的组件注册进 Delphi 的组件选项板，以便另外的应用程序使用，甚至，用户可以将该组件发布到相应的网站上，供广大的 Delphi 开发者下载使用。在一定程度上，Delphi 实现了真正意义上的资源开发共享。

1.1　Delphi 的历史回顾

Delphi 是一个 Pascal 编译器。自从 16 年前，第一个 Turbo Pascal 编译器诞生以来，Pascal 编译器一直在不断地演变，直至发展成今天的 Delphi 6.0。过去，Turbo Pascal 具有稳定、优雅以及编译速度快等特点，今天的 Delphi 6.0 也不例外，它综合了数十种编译器的经验和最新的 32 位优化编译技术。用户在使用 Delphi 6.0 时将会发现，和以前的版本相比，它的功能更强大，而且仍然非常稳定、优雅和快捷。

下面就对 Delphi 的发展作一下历史回顾。

- Delphi 1

在 DOS 年代，程序员只有两种选择：一种是编写复杂但编译效率极高的汇编语言，另一种是容易使用但速度极慢的 BASIC 语言。Turbo Pascal 以其结构化语言的简练和编译能力的强大，综合了两者的优势。

到了 Windows 3.1 年代，程序员仍然面临着两种选择：一种是强大但很难使用的 C++

语言，另一种是容易使用但语言有很大局限性的 VB。Delphi 1 提供了一种完全不同的 Windows 程序开发方法：可视化的开发环境、优化的源代码编译器、可伸缩的数据库访问引擎。正是 Delphi 1，为程序开发界奠定了 RAD 的概念。

- Delphi 2

Delphi 2 具有 Delphi 1 所具有的一切优势，只不过转移到了 32 位 Windows 和 NT 平台上。另外，Delphi 2 还增加了许多 Delphi 1 没有的功能，例如，32 位的编译器能够生成速度更快的应用程序，改进了字符串处理，支持 OLE，支持 Form 的可视化继承，并且，Delphi 2 与 16 位的 Delphi 1 是完全兼容的。

- Delphi 3

从 Delphi 2 到 Delphi 3 是一个质的飞跃。从 Delphi 1 到 Delphi 2，开发者主要考虑把 Delphi 从 16 位平台升级到 32 位平台上，同时又要保留对 16 位版本的兼容。同时，为了满足用户的要求，Delphi 2 增强了数据库和 Client/Server 的功能。到了 Delphi 3，它已经基本能够为 Windows 开发者提供一套完整的解决问题的方案，使本来非常复杂的 COM、ActiveX、WWW、“瘦”客户、多层数据库体系结构等技术变得非常容易使用。而且，Delphi3 新增加的 Code Insight 技术，为开发者提供了一条方便学习而且能有效避免代码出错的途径。

- Delphi 4

Delphi 4 使 Delphi 变得更加容易使用。新增加的 Module Explorer 技术使程序员能够以一致的图形界面浏览和编辑代码。“代码导航器”功能和“类自动生成”功能使程序员把精力更多的集中在应用程序本身，而不是花在输入代码上。IDE 重新做了设计，支持浮动和停靠，使开发环境更加个性化。Delphi 4 在 MIDAS、DCOM 和 CORBA 等方面的技术使 Delphi 的应用范围扩展到了企业。

- Delphi 5

为了更加有效地利用数据库开发领域最新的技术标准，Delphi 5 提供了一系列全新的 ADOExpress 技术。该项技术利用了 Microsoft 的 ADO 和 OLE DB 技术，为开发者提供了快速存取数据的方法，以提供给终端用户更好的商业选择。ADOExpress 被 Delphi 5 打包在它的 VCL 组件选项板中。

在 IDE（集成开发环境）方面，Delphi 5 继续增强了这个环境的易用性。全新的树状结构的 DataModule Designer 和 Data Diagram 视图能够帮助程序员充分理解应用程序中的数据。To do list 是一个非常有效的依据时间表来保证项目开发进度的工具。

另外，分布式开发的远程调试、多进程和跨进程的调试、断点提示、对组和活动点进行快速的导航、用于低级调试的 FPU 视图等众多的新功能和工具使 Delphi 5 更加具有活力和竞争力。

1.2 Delphi 6.0 的新特性

本小节的内容将介绍 Delphi 6.0 在各个方面的全新的特性。如果读者已经非常熟悉

Delphi 6.0 以前的版本的话，在这里将会看到 Delphi 6.0 在哪些方面做了改进；如果读者刚开始接触 Delphi，可以先跳过本节，阅读后面的内容。

Delphi 6.0 与以前所有的版本相比，无论是在开发效率方面，还是在开发环境易用性方面都有了很大的提高。今天，当 Microsoft 正全力开发.Net 虚拟环境，而 Borland 本身也在向 Linux 转移重心、全力开发 Kylix 的时候，Delphi 6.0 的出现正好满足了目前仍然占绝对多数的 Windows 程序员的开发需要。Delphi 6.0 在传统的开发能力方面不但持续地在进步，继续让程序员享有一个更具开发效率的环境，而且在最新的技术方面，例如 XML/XSL、COM、DCOM、COM+、SOAP、Web Service 等方面也提供了非常好的支持，使程序员能够以最快的速度使用这些新技术。

另外，Delphi 6.0 在核心执行和编译器方面也有了很大的改善，它不但支持更强的 Windows 应用程序，同时支持非常平稳的向 Linux 平台的移植。在全新的可视化组件 FrameWork CLX 的帮助下，传统的 Windows 程序员完全可以在 Windows 环境下快速的开发基于 Linux 平台运行的应用程序。

同以往 Delphi 每次更新版本一样，Delphi 6.0 在 IDE 方面仍然做了改进，它帮助程序员更加高效地进行开发，下面就介绍这些新的功能。

- Code Complete、Class Complete 和 Interface Complete

Code Complete 的功能在 Delphi 5 中就已经有了，它非常受程序员的欢迎，因为它可以大幅度地减少程序员的代码输入量。熟练使用该功能，一个 Delphi 保留字或者组件特性只需要输入前面几个字符，其余就可以由 Delphi 自动补全。因为 Delphi 中的代码很大一部分都是保留字和组件的特性名称，所以 Code Complete 功能确实能够节省很多的代码输入量，并且能够大大减少输入错误的几率。

Delphi 6.0 中的 Code Complete、Class Complete 和 Interface Complete 的功能在以前版本的基础上，增加了几个新的特性。首先，Delphi 6.0 的程序员可以自由地调整 Code Complete 框的大小，而且可以用不同的颜色来表示不同的对象，例如函数、方法或者变量等用不同颜色表示，默认情况下，方法用浅绿色来表示，而函数则用蓝色来表示。

另外，由 Code Complete 完成了方法或者函数代码的输入后，Delphi 6.0 能够自动在方法或者函数后面输入一对括号“( )”，并且光标停止在这对括号之间，同时还显示出这个方法或者函数需要的所有参数的提示框。

- Object TreeView

Delphi 6.0 的另一个显著的变化是在主界面上的 Object TreeView 窗口。这个全新的窗口中除了可以显示窗体上的所有组件以外，还能够以树的关系来表达这些组件之间的拥有者关系。当程序员在 Object TreeView 窗口中选中一个组件以后，这个组件马上就会在窗体上被选中，并且 Object Inspector 也会出现该组件，以供程序员来设置特性和处理事件。

在 1.3 小节的 Delphi 6.0 菜单介绍中将会详细介绍该窗口。

- 增强的代码编辑窗口

Delphi 6.0 最强大的 IDE 功能应该要算新的可定制的代码编辑窗口了。这个编辑器不但可以用来查看和编辑 Object Pascal 的程序代码，而且，当程序员正在编辑的是 Web 应用

程序，那么就可以直接在编辑器中查看 HTML 脚本程序，而且可以预览 Web 应用程序的执行结果界面，甚至可以查看由 Web 应用程序产生的 HTML 代码。

- 新一代数据引擎——dbExpress

为了让 Windows 上的 Delphi 6.0 与 Linux 上的 Kylix 一样具有相同的数据存取引擎，Borland 特意开发了一套全新的数据引擎——dbExpress。它是一组存取各种不同类型数据库的原生数据驱动程序，集成在一组 VCL 组件中。由于 dbExpress 是以跨平台的考虑开发的，因此 Delphi 与 Kylix 都可以使用它，而且在 Windows 下完全可以利用它们来开发 Linux 下的数据库应用程序。

使用 dbExpress 省去了安装 BDE 或者各种数据驱动的繁琐工序，使应用程序更加简洁干净，更加重要的是，dbExpress 在性能上几乎可以与传统的 BDE 相媲美，在某些项目中甚至比 BDE 表现得更好。

BDE 在存取数据的性能方面几乎由其内部机制限制住了，无论怎样改善外部配置，存取速度上都不会有很大的提高，而 dbExpress 则不同，Borland 在开发 dbExpress 时赋予这种机制很大的发挥空间和潜力，在对其不作任何调试的情况下，它已经达到了 BDE 的性能，如果知道如何微调它的话，用户会发现它几乎能够以比 BDE 快 2~3 倍的速度来处理数据。

在数据库方面，Delphi 6.0 仍然保留了最新的 BDE/IDAPI 机制，而且已经修正了几个 Bug，并且加入了新的 DB2 的数据驱动。此外，Delphi 6.0 中还包含了以前版本中的 ADOExpress，并且修正了它的几个 Bug。其中最严重的一个 Bug 就是：在 Delphi 5 中，ADOExpress 组件存取主从表数据的时候，从表经常不能实时地刷新数据，也就是从表数据与主表不能对应。

- 更好的 COM 功能

Delphi 6.0 在支持 COM/MTS/COM+方面做了加强工作，除了在 Type Library 编辑器上增加了功能以外，还增加了 COM 精灵工具。利用 COM 精灵工具，当 Delphi 6.0 的程序员需要制作特定的 COM 界面方法的时候，再也不需要像以前那样要辛苦地声明制作 COM 界面，COM 精灵可以让程序员自由地选择要制作的 COM 界面，然后帮助程序员自动产生所有的界面声明以及界面中的方法代码。使用 COM 精灵制作 COM 界面不但更方便，而且在定义声明界面时不会出错。

除了一般的 COM 组件以外，Delphi 6.0 在支援 MTS/COM+方面也以全新的 Transaction Object 和 Transactional Data Module 取代了以前的 MTS Object 和 MTS Data Module，让程序员可以更方便地开发 Microsoft DNA 架构的应用程序。

- 开发 Internet 应用程序的 WebSnap 组件

能够让程序员开发 Internet/Intranet 应用系统的新架构和组件应该是 Delphi 6.0 中最重要的新增功能之一，这个新的功能称为 WebSnap，Delphi 6.0 的 WebSnap 允许程序员使用组件和可视化方法快速开发复杂的 Internet/Intranet 应用程序。WebSnap 的特点是运行程序员在这些组件中直接加入的脚本语言的能力，并且能够直接在 Delphi 的 IDE 中编写脚本语言，预览 Web 应用程序的显示界面以及 Web 应用程序自动产生的 HTML 代码。此外，开发 Web 应用程序经常需要使用很多服务，例如图形处理、上传文件服务、Session 服务、搜索服务、以及登录和注销服务等，WebSnap 已经提供了相应的组件供程序员直接使用。

- MIDAS 的改进版本——DataSnap

Delphi 6.0 中，MIDAS 改名为 DataSnap。DataSnap 不但强化了 MIDAS 原有的功能，还加入了许多新的组件，以供程序员使用它们开发出更加强大的应用程序。而且，DataSnap 改善了 MIDAS 的执行效率，能够使应用程序更快速。

DataSnap 还提供了以前 MIDAS 所没有的功能，例如直接以 XML 的形式表现 MIDAS 的 Data Packet。

- 对 XML/XSL 的支持

Delphi 6.0 的另外一个重要改进是对 XML/XSL 的支持。增加了许多新的 VCL 组件，使程序员不必使用低级的 COM 界面或者复杂的 API 函数就可以轻易处理 XML/XSL 文件，甚至，MIDAS 的资料可以直接输出成 XML 格式，或是把 XML 的资料直接输入成 MIDAS 的 Data Packet。

1.3　Delphi 6.0 的菜单

1.3.1　主菜单

图 1.1 显示了 Delphi 6.0 的主菜单，读者可以看到，它主要由 3 部分组成：菜单栏、工具栏和组件列表。其中工具栏中的按钮都能够在菜单中找到对应的命令与之相对应。下面就开始依次介绍比较常用而且重要的命令和按钮。

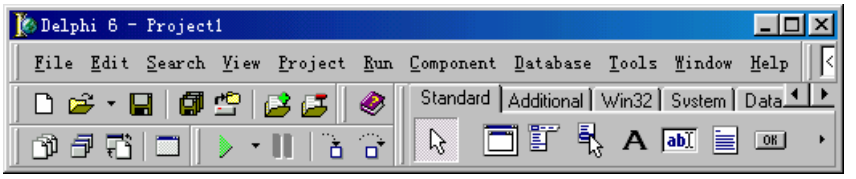

图 1.1　Delphi 6.0 主菜单

1.3.2　File 菜单

File 菜单中列出了与文件相关的各种命令，如图 1.2 所示。

File | New 菜单是一个级联菜单，它提供了创建普通应用程序、单元、窗体等各种组件的方法。其中 Application 命令用于创建一个 Windows 应用程序，CLX Application 命令用于创建一个能够运行在 Linux 操作系统下的应用程序，Data Module 命令用于创建一个数据模板，Form 命令用于创建一个新的窗体，Unit 命令用于创建一个新的单元，Frame 命令用于创建一个新的 TFrame 单元。这些只是比较常用的单元，其实 Delphi 能够创建的单元非常多，它们都列在了由 Other 命令打开的 New Items 对话框中，如图 1.3 所示。

读者可以看到，New Items 对话框含有许多标签，例如 Data Modules、ActiveX、Multitier、Corba、WebSnap 等，它们包含了 Delphi 应用程序开发中会用到的所有组件。

注意：Other 命令在工具栏中有一个相应的快捷按钮，即左数第一个按钮，也就是 Windows 中常用作新建功能的按钮。

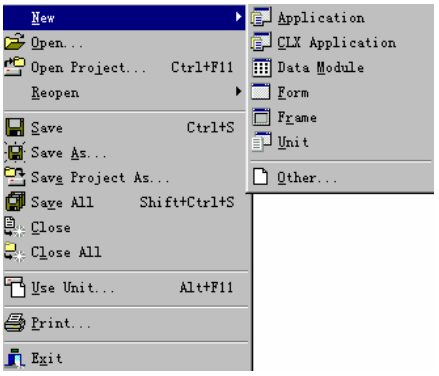

图 1.2 File 菜单

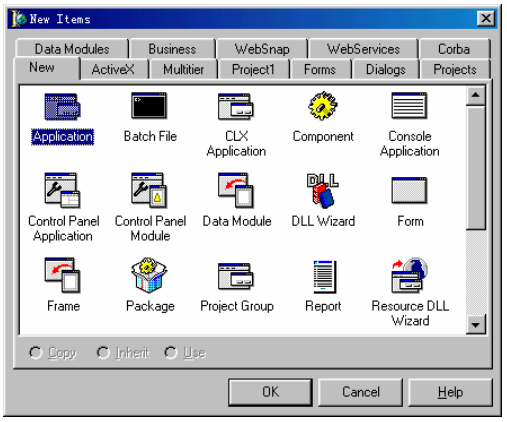

图 1.3 New Items 对话框

File 主菜单下的 Reopen 子菜单也是一个级联菜单，其中列出了最近打开过的 Delphi 工程、单元和窗体等。子菜单分上下两个部分，分隔线以上为最近打开过的项目，以下为最近打开过的单元或者窗体文件等。工具栏中的 Open 按钮提供了打开 Delphi 文件的功能，它旁边的下拉箭头可以将快速打开 Reopen 子菜单。

Use Unit 命令用来打开一个单元列表，如图 1.4 所示。

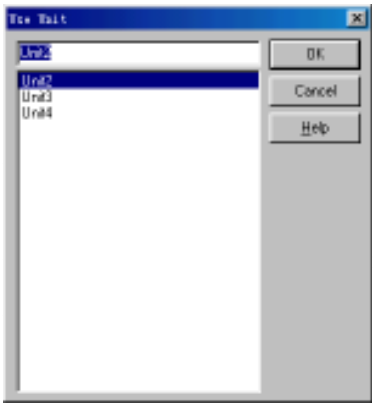

图 1.4 Use Unit 对话框

选中一个单元后，单击 OK 按钮，系统将所选中的单元自动加入到当前单元的 Uses 子句中去，用户同样可以通过自己手动修改 Uses 子句加入一个单元。这两种方式的作用是相同的，都是向当前单元中加入另一个单元。实际上，用户会发现，Use Unit 对话框中只会列出当前单元中没有通过 Uses 子句引用过的单元。例如，如果选择 Unit2 的话（见图 1.4），将在当前单元 Unit1 中看到如下所示的程序段：

```
unit Unit1;
interface
uses
    Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
    Dialogs;
type
    TForm1 = class(TForm)
    private
        { Private declarations }
    public
        { Public declarations }
    end;
var
    Form1: TForm1;
implementation
uses Unit2;   //引入 Unit2 单元
{1R *.dfm}
end.
```

读者注意到了，Unit2 加到了 implementation 后的 Uses 子句中，而不是 Interface 后的 Uses 子句中。原因是因为 Interface 后的 Uses 子句中通常用于引用 Delphi 系统中自带的各种单元，或者组件单元，而 implementation 后的 Uses 子句只是用于引用当前项目中的单元。

1.3.3　Edit 菜单

Delphi 6.0 中的 Edit 菜单主要提供了在程序的编辑过程中会用到的各种工具和功能。如图 1.5 所示，它分为 4 个部分，最上面的部分包含两套命令：Undelete 和 Redo，或 Undo 和 Redo，它们分别适用于不同的操作之后。如果程序员在窗体上删除了一个组件，那么这部分命令将是 Undelete 和 Redo，运行 Undelete 命令将恢复刚刚删除了的组件；如果程序员只是对程序代码进行了一些常规的编辑，例如删除一行代码，那么这时候，Edit 菜单中的第一个部分将是 Undo 和 Redo 两个命令，其中 Undo 命令用于恢复删除了的代码行，而 Redo 命令则重新执行刚刚删除的代码行的操作。Delphi 中用到 Undo 和 Redo 的操作很多，甚至包括代码编辑中光标的移动以及代码行的选取等。所以，学会利用这几个命令，能够帮助用户很轻松地进行代码的编辑工作。

Edit 菜单中的第二部分命令主要针对于程序代码。其中包括了常规的文本的拷贝、剪切、粘贴以及全选等命令。它们同样存在于代码编辑页中的右键弹出菜单中。

图 1.5 Edit 菜单

Edit 菜单的第三部分包括很多命令，它们主要针对于窗体组件的编辑和设计。其中包括组件的网格对齐、位置对齐、组件尺寸大小、组件叠放顺序等。同样，在窗体或者组件的右键弹出菜单中同样具有这部分命令。

Edit 菜单的第三部分中 Align 命令用于整齐地摆放组件，单击它将打开一个对话框，如图 1.6 所示。在这里，用户可以从水平和垂直两个方向上来对齐组件，或者均匀分布组件，在 1.3.5 小节中介绍的 View 下拉菜单中的 Alignment Palette 命令同样可以实现以上的功能。

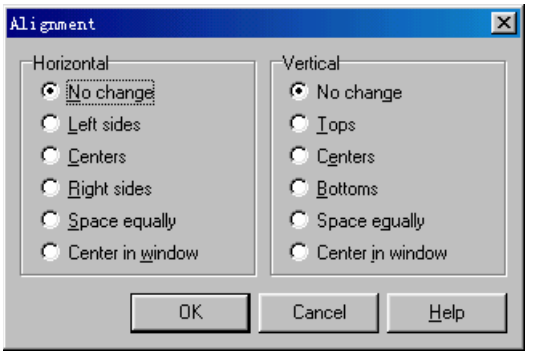

图 1.6 Alignment 对话框

Tab Order 命令用于设置可获得焦点的组件在窗体上获得焦点的顺序，也就是在窗体上通过 Tab 键切换组件的顺序。图 1.7 中列出了当前窗体上的所有可控组件，那些没有焦点的组件，例如 TLabel 等就没有出现在这个列表中。调整窗体上可控组件的顺序以后，通过查看这些组件的 TabOrder 特性，可以得知：Button1 的 TabOrder 为 0，Edit1 为 1，CheckBox1

为 2，RadioButton1 为 3，这正是它们依次获得焦点的顺序。用户也可以直接编辑组件的 TabOrder 特性来设置 Tab 顺序，但是那样做不如通过 Tab Order 对话框设置来得直观和有效。

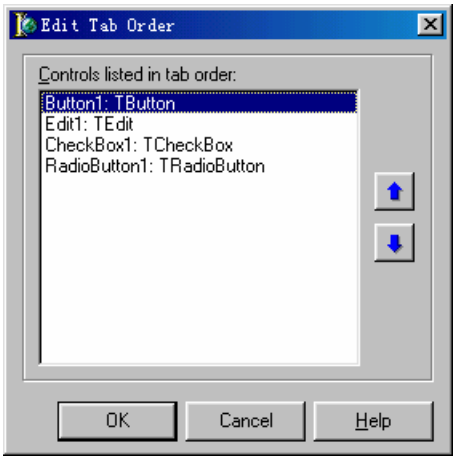

图 1.7　编辑组件的 Tab 顺序

1.3.4　Search 菜单

Search 菜单提供了一些方便程序代码编辑的功能，如图 1.8 所示。

Find 命令用于在当前正在编辑的代码中查找指定的字符串。

Find in Files 命令除了提供了 Find 命令提供的功能以外，还支持从当前工程中所有的单元中，或者指定的文件和目录中查找指定的字符串的功能。它适用于用户不确定要查找的字符串在哪个单元中的情况。

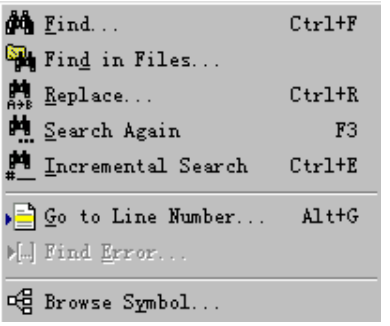

图 1.8　Search 菜单

Replace 命令提供了在查找字符串的基础上，对指定的字符串进行替换的功能。

Search Again 命令应用于通过 Find 命令查找到第一个指定目标后，继续在后面的文本中查找该目标。

Incremental Search 命令提供了一种更简洁的查找字符串的方法。单击该命令后，用户在代码编辑器底部的状态条中会看到“Searching for:”的字样，然后继续输入需要查找的字符串，这时代码编辑器中的光标将自动定位到匹配的字符处，如图 1.9 所示。输入了 ad 字

符串后，光标定位到了 Radio 上，并且将反复选择目标字符串 ad。如果输入的字符串在当前单元中没有任何匹配，那么就无法在状态条中输入它。

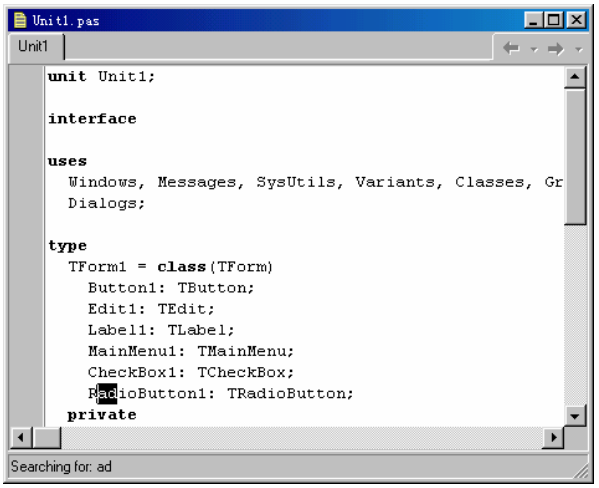

图 1.9 Incremental Search 查找字符串

1.3.5 View 菜单

View 下拉菜单在 Delphi 编程中的应用比较重要。View 菜单包含了很多 Delphi 开发环境中的各种管理工具和命令，它们在开发中起到了很重要的辅助作用，下面就一一介绍这些工具和命令，如图 1.10 所示。

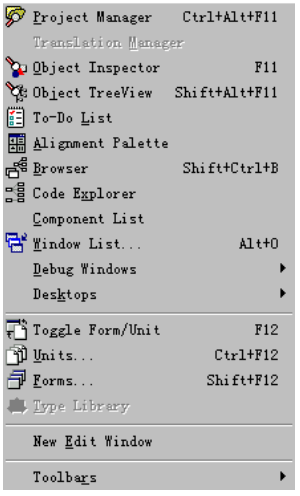

图 1.10 View 菜单

1．Project Manager（工程管理器）

工程管理器是一个非常有效的工程管理工具，其中列出了一个 Project Group（工程组）中所有工程，以及每个工程中的单元和窗体，如图 1.11 所示。它能够帮助用户快速地打开

一个工程中的某个单元或者窗体。当一个工程组中包含了多个工程的时候，工程管理器的作用就更加重要，它能够进行快速的切换，而不必重新打开工程。

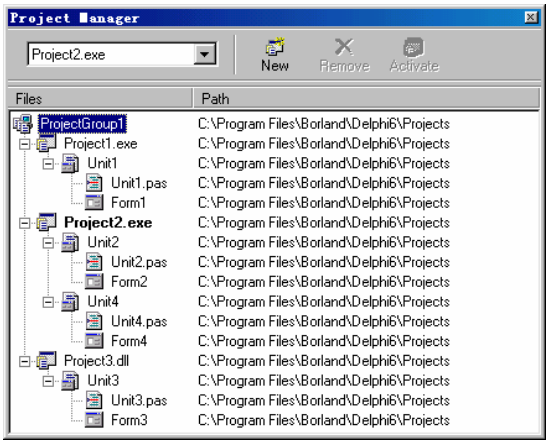

图 1.11　Project Manager

工程组 ProjectGroup1 中包含了两个 EXE 工程和一个动态库工程（见图 1.11）：Project1.exe、Project2.exe 和 Project3.dll，用户从这个管理器中能够非常清晰地看到每个工程所包含的单元和窗体。双击其中的一个工程，Delphi 将自动切换到这个工程，同时会看到这个工程节点变为了粗体，例如图中的 Project2.exe 节点。双击工程管理器中的一个单元或者窗体，系统将打开这个单元或者窗体，同时当前的工程也自动切换到该单元或者窗体所属的工程。

2．Object Inspector（对象监视器）

对象监视器是 Delphi 中的主要开发工具，它用来进行各种组件的属性设置以及事件句柄的处理，它分为两部分：Properties（属性）设置和 Events（事件）设置，如图 1.12 所示为一个 TForm 实例的对象监视器设置，左面为属性选项卡，右面为事件选项卡。

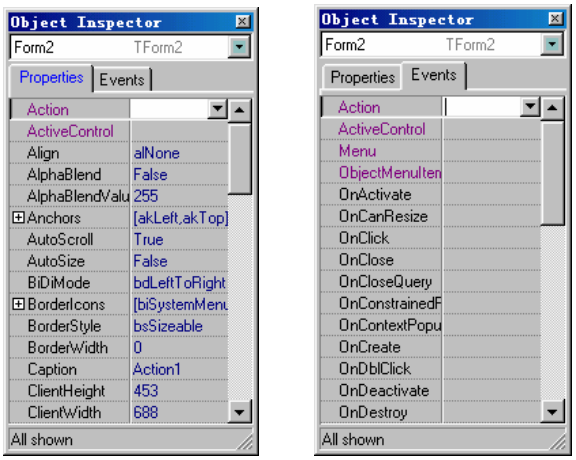

图 1.12　对象监视器的属性选项卡和事件选项卡

属性是指组件对象的各种特征，它可以在运行组件时或设计组件时被修改。

位于对象监视器顶部的组合框显示当前选中的组件，图中的 Form2 TForm2（见图 1.12），它表示当前组件为 Form2，组件是 TForm2 类型。在对象监视器中可以查看并修改窗体上所有组件的属性、定义以及查看这些组件的事件。属性选项卡可以按字母顺序或者分类显示当前组件的属性，系统默认为按字母顺序显示。事件选项卡则显示了该组件的所有事件。某些属性的旁边带有“+”号，“+”号表明该属性还有子属性，这样的属性可以看成是提纲式属性。当单击“+”使之变为“-”号时，表示该属性已显示了所有的子属性。

在属性选项卡中，不同类型的属性通过不同的显示形式来表示，例如字符串类型的属性用一个普通的编辑框来表示，布尔类型的属性是一个下拉列表框，其中只有 True 和 False 两个选项，枚举类型的属性也是一个下拉列表框，其中列出了所有的可选项，集合类型的属性项的左边同样有一个“+”符号，单击这个“+”号，可以展开一系列的可选项，每一个可选项都以布尔类型表示，设置为 True 即表示集合中包含了此项。更加详细的有关属性类型的内容将在第 7 章中介绍。

属性选项卡中有一种比较特殊的属性项，它是对另外一个组件的引用，称之为引用属性，例如 Form 组件中的 Action、Menu 属性，或者 ActionList 组件中的 ImageList 属性。通过下拉列表框选择一个引用组件之后，会发现在属性项左边会出现一个“+”号，就像集合类型的属性项一样，单击“+”号，对象监视器中将展开所引用的组件的各个属性。这种机制使用户不必切换到当前组件就可以方便的设置一个被引用的组件的属性。如图 1.13 所示，Form 组件引用了 Action 属性。

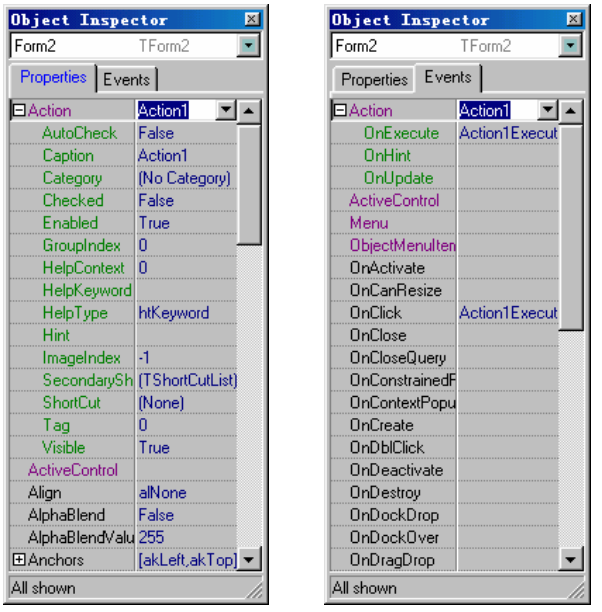

图 1.13 Form 组件中的引用属性 Action

在对象监视器中，用户会看到有些属性项用了不同的颜色标识，每种颜色代表不同的属性种类。最普通的属性项默认用了黑色，而引用属性项用了一种 clPurple 色，引用属性

组件的子属性则默认用了绿色。Delphi 允许用户自己定义这些颜色。具体方法是：在对象监视器上右击，在弹出的快捷菜单中执行 Properties 命令，这时将打开一个对象监视器属性设置对话框，如图 1.14 所示。

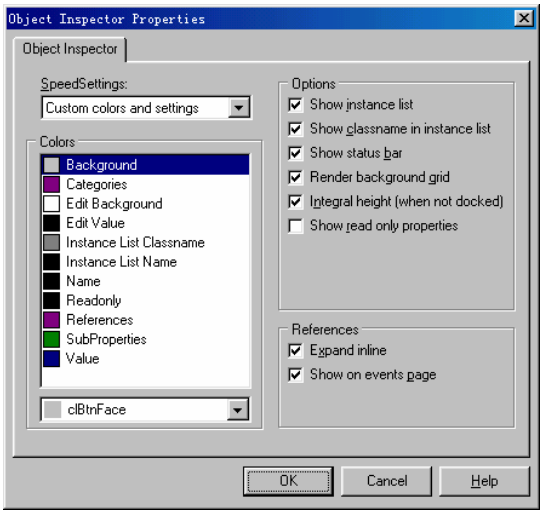

图 1.14　Object Inspector Properties 对话框

在这个对话框中，系统为颜色设置部分提供了一组快速颜色风格设置，其中包括系统默认颜色设置、Delphi 5 风格颜色、Visual Studio 风格颜色以及用户自定义颜色。在这里，用户可以设置很多颜色，例如对象监视器的背景色、属性值颜色、编辑框颜色、引用属性颜色、引用属性组件的子属性颜色等。除了颜色设置以外，在这里还可以设置对象监视器的另外一些选项，例如是否显示状态条、网格等。最后一项设置是针对引用属性的，包括两个设置：Expand Inline 和 Show on events page，前者表示引用属性组件的子属性是否可展开，后者表示在 Events 事件页中是否同时显示引用属性组件的子事件。实际上，上面讨论的引用属性的设置都是基于系统的默认设置，即以上提到的两个设置都选中的情况。

3．Object TreeView（对象树）

对象树是一种将窗体上的组件以树的形式进行表示的方法，如图 1.15 所示，它不仅列出了窗体上的所有组件，还能够以树的形式体现这些组件之间的从属关系。例如，当前窗体上包括了 TPanel、TButton、TTable、TDataSource 等各种组件，它们之间具有父子和关联关系，如 TPanel 是两个 TButton 组件、TCheckBox 和 TRaidoButton 组件的父组件，而 TTable 和 TDataSource 之间具有关联关系，这些关系在对象树上都有很明确的表示。

在普通窗体上，因为这类组件并没有直观的名称标识，当窗体上的组件非常多的时候，尤其是有很多相同的组件，例如 TTable 和 TDataSource 的时候，用户会发现找到一个组件有时候很困难，而通过对象树就可以很轻松地定位一个组件，因为对象树上的当前选中的组件同时就是窗体上当前选中的组件。用户甚至可以像在 Windows 的资源浏览器中定位一个文件一样，在对象树上直接快速地输入要找的组件的名称，对象树就可以马上定位到相应名称的组件上。例如在对象树上快速地输入 list（见图 1.15），可以看到当前树上的当前节点定位到了 ListBox1 上，而在窗体上，当前组件也相应地定位在了 ListBox1 组件上。

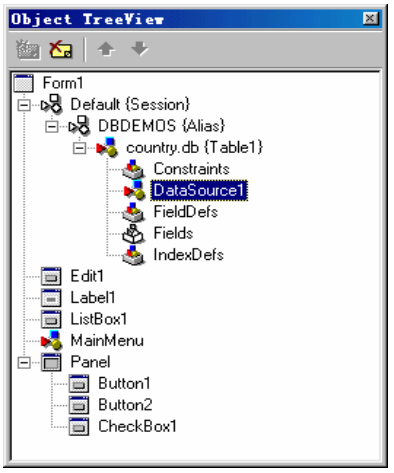

图 1.15　对象树

通过删除对象树上的节点，用户还可以从树上直接删除一些组件，而不必到窗体上找到这个组件后再删除。

如果用户已经非常熟悉了 Delphi 6.0 以前的版本，那么可能会很不习惯 Object TreeView 窗口的出现。默认情况下，它占据了开发界面中比较大的区域，使 Object Inspector 的有效区域小了很多，但是笔者仍然建议用户使用这个窗口，而不要总是把它关闭掉，慢慢地就会发现这个窗口的作用不仅大而且好用。

4．Alignment Palette

Alignment Palette 主要应用于将多个组件按照各种形式规则地摆放到窗体上。这些规则包括左对齐、右对齐、居中、均匀间隔等，如图 1.16 所示，其中每一个按钮对应于一种摆放规则。单击其中一个按钮不放，就会看到规则摆放后的样子，如图 1.17 所示就是单击左对齐规则按钮不放时的样子。其余所有的按钮都具有这种效果。

图 1.16　Alignment Palette

图 1.17　按左对齐规则摆放后的样子

在 Alignment Palette 的帮助下，用户可以很轻松的对齐多个组件，而不必一个一个地摆放，从而节省了代码编写的时间和精力。

5．Code Explorer

Code Explorer 以树的形式列出了当前单元中声明的所有类型以及这些类型中的公共成员和私有成员，另外还包括了单元中定义的变量、常量以及所引用的单元等。如图 1.18 所示，Unit1 单元中声明了一个 TForm1 类、一个 TMyType 类，定义了 Form1 和 i 两个变（常）量，并且引用了多个单元。

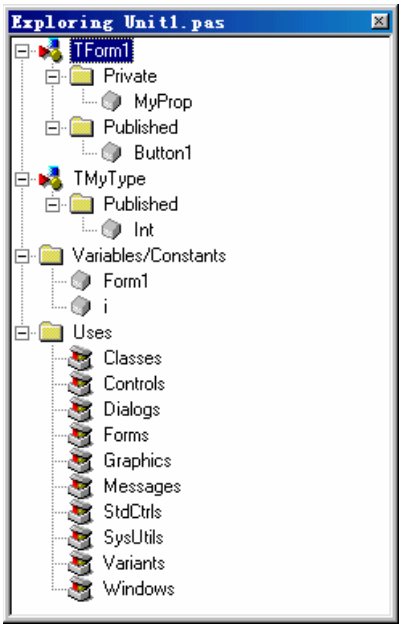

图 1.18　Code Explorer

6．Component List（组件列表）

组件列表中包括了所有 VCL 组件选项板中的组件，用户可以通过组件名称来定位到一个组件，并且可以通过 Add to form 按钮直接将组件加到当前的窗体上去，如图 1.19 所示。组件是按照字母顺序排列在组件列表中的。组件列表适用于用户知道一个组件的名称，但不确定它在组件选项板上的哪一页中的情况。如果一页一页的查找它会非常费时，使用组件列表，通过名称快速定位到该组件，然后绕过组件选项板直接向窗体上添加组件。

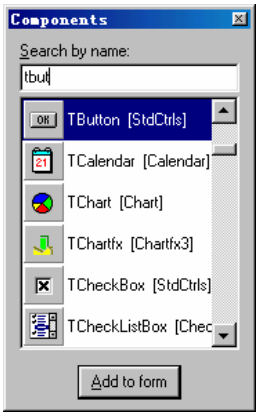

图 1.19　Component List 窗口

7．Window List（窗口列表）

窗口列表中列出了 Delphi 中当前所有的已打开的可视窗口，选中一个窗口可以马上切换到该窗口，如图 1.20 所示。

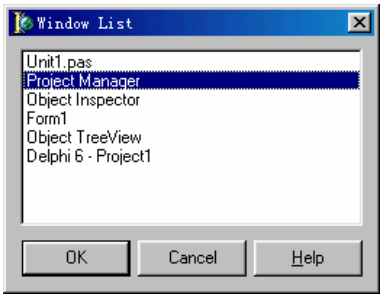

图 1.20 Window List

需要注意的是，Window List 中只会列当前已经打开了的窗口，如果图中的 Project Manager 窗口已经关闭了，那么重新打开 Window List 后，其中将不会有 Project Manager 项。另外，Window List 中还包括了 Delphi 6.0 的主窗口本身，即图中的 Delphi 6.0 – Project1 项，这是 Window List 与 Delphi 6.0 中的 Window 菜单的不同之处。

8．Debug Windows | BreakPoints 命令

BreakPoints 命令用于打开一个当前项目中所有断点的列表，其中包括了断点所在的单元、行号、状态以及断点运行过的次数等信息，如图 1.21 所示。

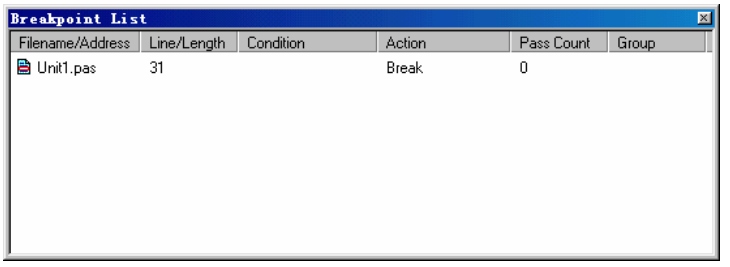

图 1.21 BreakPoint List 断点列表

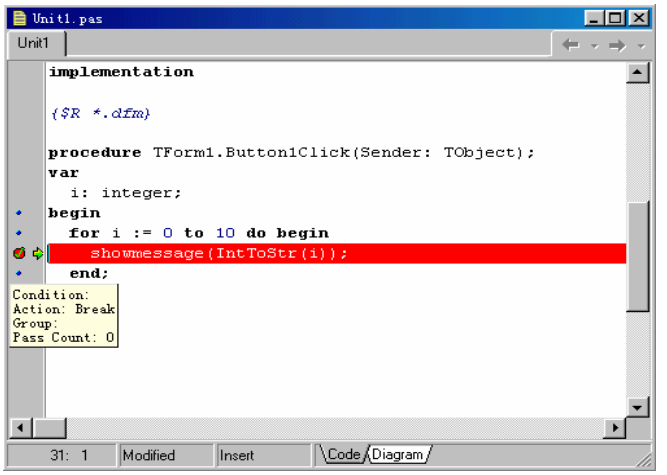

图 1.22 单元中的断点提示框

实际上，断点列表中所列断点的信息，通过在代码单元中的断点上的鼠标提示同样可以获得。例如图 1.22 中，将鼠标指针放在断点的红色小点上停留一会儿，就会出现一个提示框（黄色框），里面显示了当前断点的各种信息，也就是上面的断点列表中信息 Condition、Action、Group、Pass Count 等。

9．Desktops 命令

执行 Desktops 命令后，弹出一个级联菜单，其中包含有几个子命令。其中 Save Desktop 命令用于打开一个 Delphi 保存桌面环境方案的对话框，如图 1.23 所示，它能够将当前 Delphi 中的各个窗口的分布、位置、大小等信息全部保存起来。

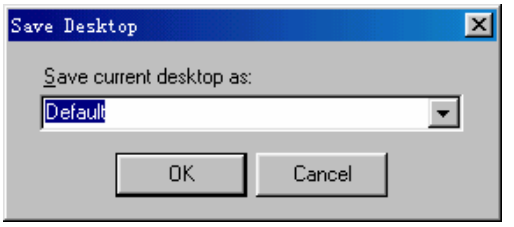

图 1.23　Save Desktop 对话框

Select Debug Desktop 命令用于打开一个类似图 1.23 的选择桌面设置方案的对话框，如图 1.24 所示。

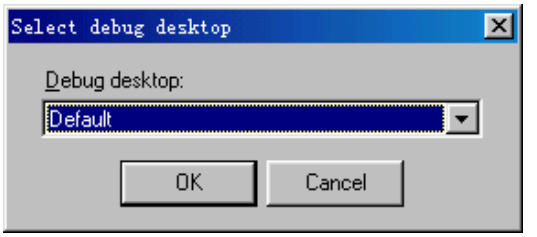

图 1.24　Select debug desktop 对话框

Delphi 6.0 中的 Desktop 工具栏（默认在主菜单右侧）中，同时提供了以上两个命令功能的工具按钮，这两个按钮分别具有和菜单相同的图标。另外还有一组复选框，其中列出了所有的桌面设置方案，通过它可以快速地转换 Delphi 桌面设置方案。

10．Toggle Form/Unit 命令

Toggle Form/Unit 命令的作用是在窗体和单元之间进行相互切换，即由当前的窗体切换到它的代码单元上，如果一个单元具有对应的窗体，那么同样可以通过该命令切换到窗体上。Delphi 的 View 工具栏中提供了与该命令对应的工具按钮，它们具有相同的图标。

11．Units 命令

Units 命令的功能是打开一个单元列表，其中列出了当前工程中所有的单元，包括主工程单元 dpr 和普通单元 pas 等，如图 1.25 所示。选择其中一个单元后，单击 OK 按钮可以将系统切换到该单元上。同样，View 工具栏中提供了与该命令对应的工具按钮。

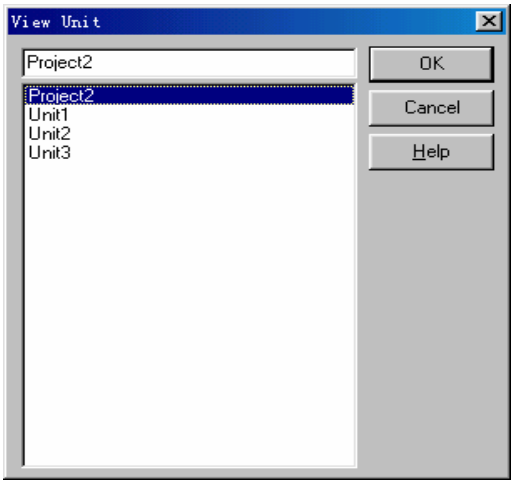

图 1.25 单元列表

12．Forms 命令

与 Units 命令相似，Forms 命令用于打开一个窗体的列表，如图 1.26 所示。同样，View 工具栏中提供了与该命令对应的工具按钮。

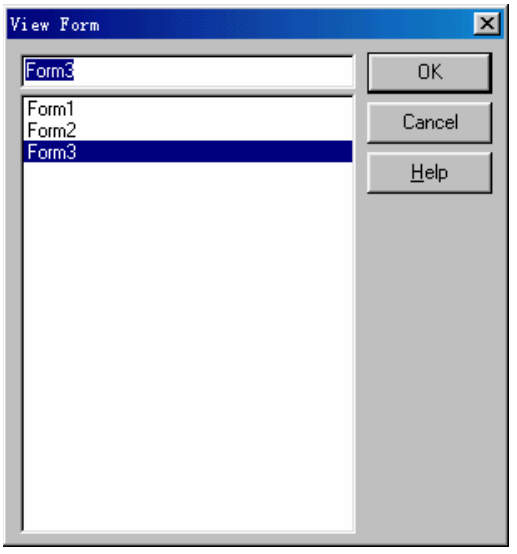

图 1.26 窗体列表

1.3.6 Project 菜单

Project 菜单是一个非常重要的菜单，其中包含了若干用于 Delphi 工程选项设置、代码语法分析、代码编译、版本信息维护等各方面非常重要的工具和命令。在学习 Delphi 编程之前，读者非常有必要了解这个菜单的内容。Project 菜单如图 1.27 所示。

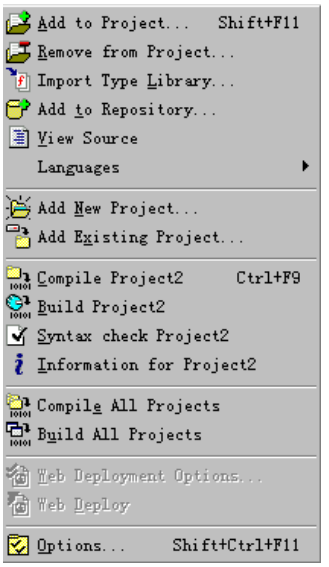

图 1.27　Project 菜单

下面就逐一介绍 Project 菜单中的命令。

1．Add to Project（向工程中添加单元）

Add to Project 菜单的功能是向当前的工程中添加一个保存的单元。单元添加进工程以后，展开 Project Manager 中该工程的节点，就会看到新添加的单元也列在了下面，或者通过 View 工具栏中 Units 按钮打开当前工程的单元列表，也可以看到该工程。用户也可以打开当前工程的 dpr 文件，在 Uses 子句中找到新加的单元名以及该单元所处的路径。例如下面的 Project1 工程中通过 Add to Project 命令添加了一个 NewAdded.pas 单元。

```
program Project1;

uses
    Forms,
    Unit1 in 'Unit1.pas' {Form1},
    NewAdded in '..\..\..\..\WINDOWS\Desktop\NewAdded.pas' {Form2};
```

该命令在 Standard 工具栏中有一个对应的工具按钮，它们具有相同的图标，另外，在 Project Manager 的窗口中，右击一个工程节点，在弹出的快捷菜单中的 Add 命令同样具有这个功能。

2．Remove from Project（从工程中删除单元）

与 Add to Project 命令相对应，Remove from Project 命令提供了将一个单元从工程中删除的功能，用它可以打开一个对话框，其中列出了当前工程中的所有单元，如图 1.28 所示，从中选择一个单元，单击 OK 按钮后，系统就会将该单元从工程中删除。

注意：Remove from Project 并不是删除单元文件，而只是从工程中去掉了单元的连接，该单元的文件仍然存在于磁盘上，用户仍然可以通过 Add to Project 命令重

新加入该单元。

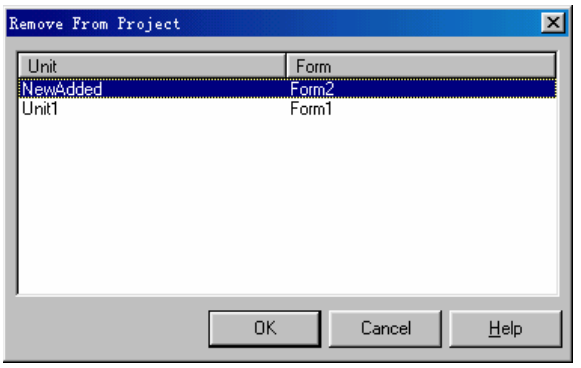

图 1.28 Remove from Project 对话框

3．Import Type Library（导入类型库）

该命令的应用将在第 11 章中介绍，这里笔者只简单介绍它的作用，使读者能够对该菜单有个初步的了解。执行 Import Type Library 命令将打开一个导入类型库的对话框，如图 1.29 所示。

在这个对话框中，首先列出了当前系统中所有注册的类型库，可以看到有许多这样的类型库，其实这说明了一个问题，就是现在越来越多的应用程序开始使用与 COM 相关的技术。通过单击 Add 按钮，可以向系统注册并加入用户自己的类型库对象，它可以是 DLL、TLB 或者 OLB 等类型库文件。如何生成自己的类型库，以及如何选择相应的文件添加类型库将在第 11 章详细介绍。

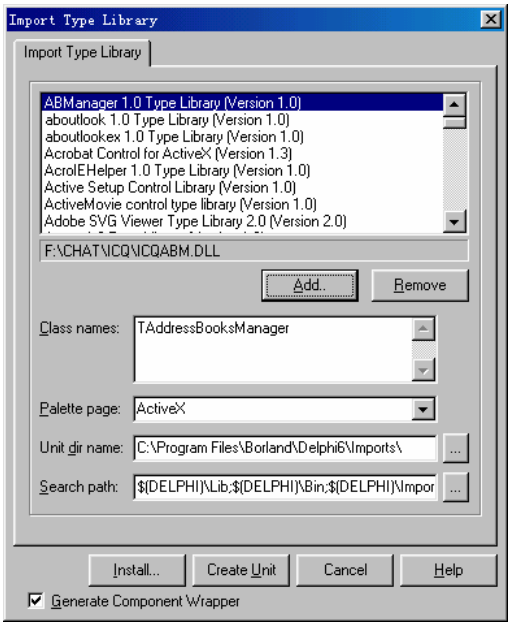

图 1.29 Import Type Library 对话框

导入类型库功能的关键是将一个类型库以 VCL 的形式加入到组件选项板上。单击 Install 按钮将执行该功能，之后对该类型库，用户就可以像使用一个普通的 VCL 组件一样轻松地调用、使用该类型库提供的接口。类型库组件默认情况下将会加到组件选项板上的 ActiveX 选项卡中。Create Unit 按钮的功能是为当前选择的类型库创建一个接口定义单元，其中将列出该类型库接口中提供的函数、方法和属性。

提示：并不是所有的类型库都可以加到 VCL 组件选项板上，只有那些具有可创建的 CoClass 的类型库才可以，用户可以通过 Class names 文本框中是否有指定的类名来做判断，也就是说，只要 Class names 文本框中有类名，那么该类型库就可以加到组件选项板上。

4．Add New Project 命令和 Add Existing Project 命令

这两个命令的作用是向当前工程组 ProjectGroup 中加入一个新的工程或者已经存在的工程。在 Project Manager 窗口中，右击工程组节点，在弹出的快捷菜单中同样存在这两个命令。

5．Compile 命令和 Build 命令

Complie 命令的功能是编译当前的工程。如果程序中没有任何语法错误，那么将产生一个可执行文件（EXE 或者 DLL）；如果存在语法错误，Delphi 将停止编译，并且将光标自动停在第一个语法错误的地方，在 Messages 对话框中同时显示出所有的 Error（错误）和 Warning（警告），或者提示 Hint，如图 1.30 所示。

图 1.30　Messages 对话框

Build 命令的功能与 Compile 命令很相似，也是编译工程、产生可执行文件。它们之间的不同之处在于：如果程序已经编译过了，再次执行 Compile 命令，它将仅仅编译代码更改的地方，然后更新可执行文件中与更改的代码相关的地方，而 Build 命令无论在什么时候，它都将编译全部代码、重新产生全新的可执行文件。因此，Build 命令能够比较全面地检查代码的各个角落、找出所有的问题，并且在 Messages 中显示出错误、警告或者提示。

6．Information for 命令

Information for 命令能够显示出当前工程的编译信息以及编译状态等，如图 1.31 所示是该命令打开一个对话框，可以看到其中包括了工程编译过的代码行数、代码文件大小、数据大小、初始堆栈大小、可执行文件大小、所使用的包以及工程现在的编译状态等信息。

7．Compile All Projects 命令和 Build All Projects 命令

这两个命令主要应用于工程组中，对工程组中的所有工程进行编译和 Build。

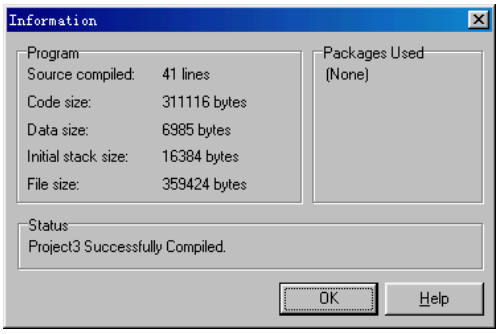

图 1.31 Information 对话框

8．Options 命令

Options 命令用于将打开一个项目选项设置的对话框，其中包括有关当前工程的大量设置信息，例如当前工程的编译路径、查找路径、编译设置、工程版本信息、窗体创建时机等。

有关 Project Options 对话框的内容，将在 3.4 一节中详细讲解。

1.3.7 Run 菜单

Run 菜单中包含的都是运行或者调试 Delphi 程序所需的命令和工具，如图 1.32 所示。

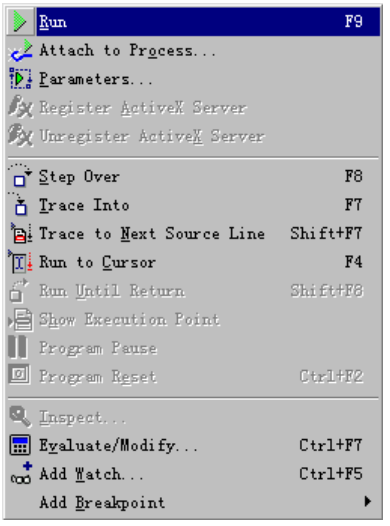

图 1.32 Run 菜单

1．Run 命令

Run 命令用于运行程序，在这种状态下，程序将逐行运行代码，直到断点处或者程序运行结束。

2．Parameters 命令

Parameters 命令主要应用于调试 DLL 应用程序的情况，用它可以打开一个对话框，如

图 1.33 所示。

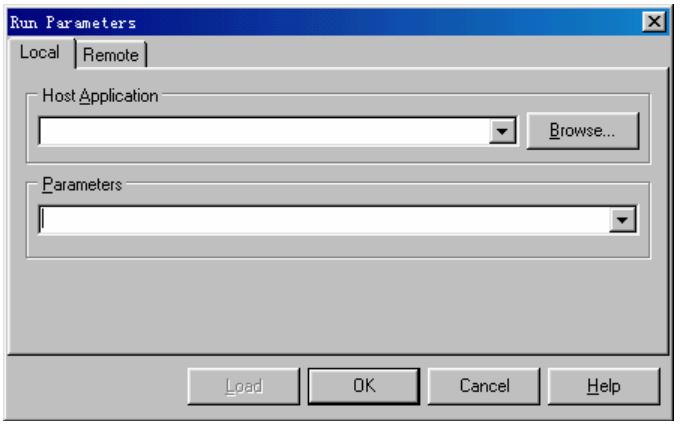

图 1.33　Run Parameters 对话框

其中可以指定一个宿主程序，也就是调用当前 DLL 程序中的函数的主程序。因为 DLL 程序不能直接运行，所以为了调试程序中的代码，必须由一个普通应用程序来启动 DLL 程序。从 Run Parameters 对话框中指定一个宿主应用程序以后，单击 OK 按钮确定，然后单击 Run 命令，系统首先将运行这个宿主程序，通过它来调用 DLL 程序中的函数，这时候用户就可以像调试普通的应用程序一样调试 DLL 程序了。当然，宿主程序中必须有一个方法调用了 DLL 程序中的一个函数或者方法，才能使 Delphi 进入 DLL 程序中相应的代码中。

3．Register ActiveX Server 命令和 Unregister ActiveX Server 命令

这两个命令的功能是向系统注册和反注册一个 ActiveX 服务器，前提是当前编译的工程必须是 ActiveX 服务器程序，例如 ActiveX Library 程序。实际上，它相当于编译好一个 ActiveX 服务器程序以后，用系统提供的 RegSvr32.exe 程序来注册一个未注册的服务器，或者反注册一个已注册的服务器。

4．Step Over 命令和 Trace Into 命令

Step Over 命令的功能是逐行运行程序中的代码行，Trace Into 命令则是使当前运行点进入一个函数中，逐行运行函数中的每一行代码。

结合这两个命令，用户可以任意控制调试代码。可以用 Step Over 命令来整体运行一个调用函数；当发现一个函数出现了错误，则可以用 Trace Into 命令进入这个函数，逐行运行来定位错误。

这两个命令是程序调试的主要工具，一定要学会熟练使用。

5．Program Reset 命令

Program Reset 命令用来结束正在运行或者调试的程序，回到程序编辑状态。有时候在运行一个程序的时候，发现了一个错误，或者进入了意外的死循环状态中，这时，程序已经没有必要继续运行下去了，那么用户就可以使用 Program Reset 命令来快速地结束本次程序运行。而且，该命令还能够使程序的当前光标停止在终止程序运行时的地方，以便查找

错误的地方。

6．Evaluate/Modify 命令

Evaluate/Modify 命令是 Delphi 自带的一个调试程序的非常有用的工具，它能够在程序运行期间监视和修改一个变量或者表达式的值，如图 1.34 所示。

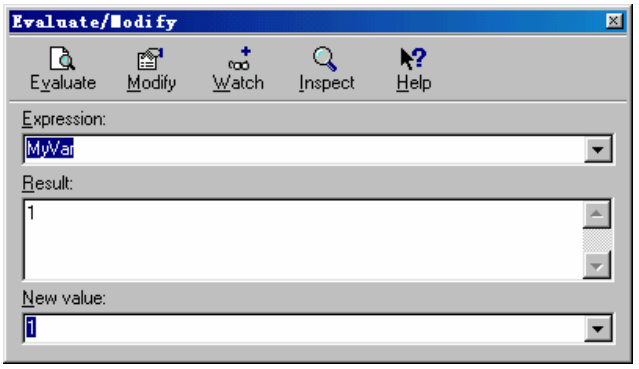

图 1.34　Evaluate/Modify 对话框

使用这个工具，用户可以轻松监视到程序指定部分的运行情况。如果仅仅是想查看一个简单变量的赋值情况，那么用户也可以通过用鼠标置于当前变量上，停留片刻，Delphi 编辑器将弹出一个提示框，其中能够显示当前变量的值，如图 1.35 所示。

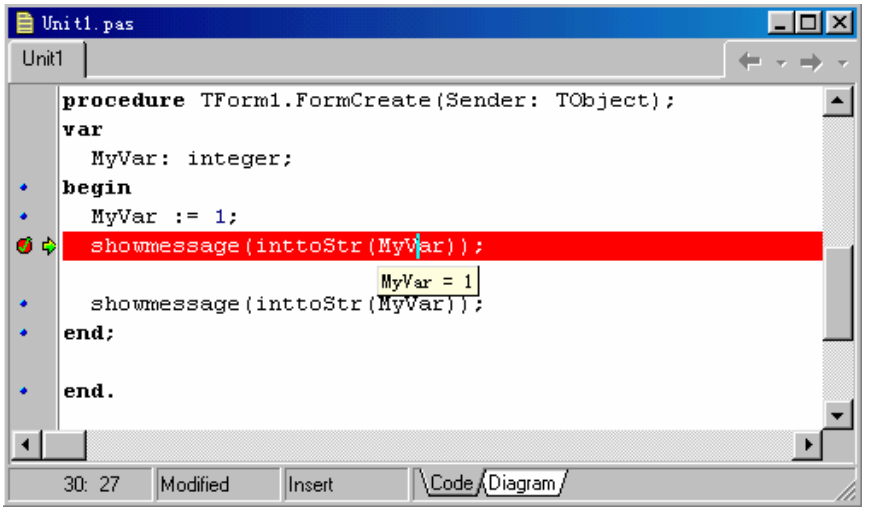

图 1.35　系统变量提示

如果想在程序运行期间，手动地改变其中一个变量的值，只需要在 Evaluate/Modify 对话框中的 New Value 文本框中输入一个新的合法的值，回车之后，当前监视的变量将被赋予一个新的值。当运行完第一个 showmessage 函数的时候，见图 1.34 中的程序。通过 Evaluate/Modify 命令来修改其中的 MyVar 变量值，设为“2”，然后让程序继续运行第二个 showmessage 函数，会发现信息框中将显示“2”。

1.4　Delphi 6.0 的工具栏

Delphi 6.0 中提供了 Standard、View、Debug、Desktop、Internet 和 Component Palette 等工具栏，每个工具栏中包含了很常用的一系列工具按钮，它们分别对应着某个命令的快捷方式。这些按钮继承了 Microsoft 的 Office 系列产品中工具栏上工具按钮的风格。当鼠标移动到这些按钮上时它们看上去向上突起了一点，而在此之前它们看上去一直是平的。而且，每个工具按钮都提供了 Hint 提示，就是当鼠标在按钮上停留一定时间后，会有一个悬浮着的带有文字的小框出现，这种界面小技巧现在已成为 Windows 程序的一种标准，它能够很好地解释工具栏上的工具按钮具有什么功能。当然，由于 Delphi 本身就是用 Delphi 创建的，因此，利用 Delphi 完全可以创建出所有的这些功能和效果。

和 Delphi 6.0 的其他方面一样，工具栏同样可以灵活订制。如果想要改变某一工具栏的位置，只要用鼠标将它拖到新的位置就可以了。还可以决定要显示哪几组工具栏，或者哪些工具按钮。方法很简单：首先右击任一工具栏，在弹出的菜单中将会列出所有的工具栏，只要选中相应的工具栏，Delphi 中就会出现这个工具栏，同样用户也可以隐藏掉某个工具栏。或者，还可以执行弹出菜单中的 Customize 命令，将出现一个自定义的对话框，如图 1.36 所示，在 Toolbars 选项卡上，将要显示的工具栏旁边的复选框打上勾就可以了。

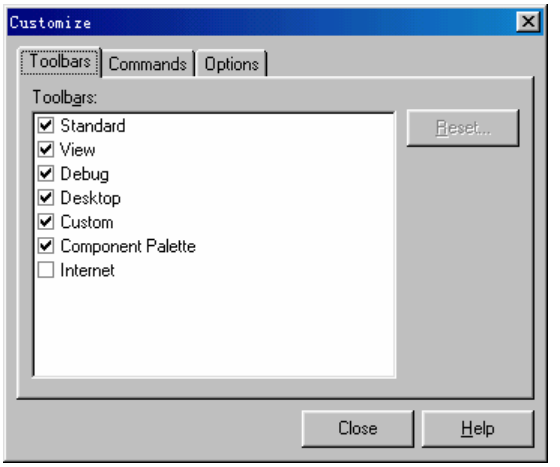

图 1.36　Customize 对话框的 Toolbars 选项卡

Customize 对话框的 Commands 选项卡中分组列出了 Delphi 系统中很多的命令，其中包括了所有的菜单命令，如图 1.37 所示。从中可以选择一个命令加到任意一个工具栏上。具体步骤非常简单：首先选择其中任意一个命令，然后拖拉到一个工具栏上，松开鼠标左键以后该命令就自动加进了工具栏中。同样，如果想从工具栏中删除一个已经存在的按钮，只需要把该按钮拖拉到 Customize 对话框上，松开鼠标左键后，该按钮就从工具栏上消失了。

其实这种工具栏的操作方式在 Delphi 6.0 开发的应用程序中完全可以实现，而且有很多非常优秀的第三方组件，例如比较著名的 DevExpress 公司的一系列组件，可以更方便、更美观地实现它。

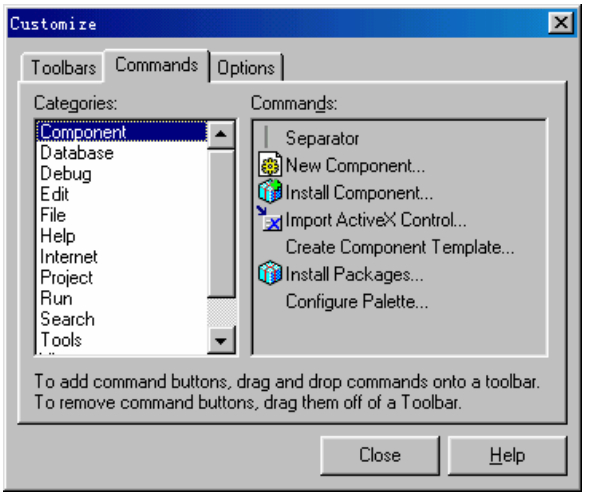

图 1.37 Customize 对话框的 Commands 选项卡

在 Customize 对话框中的 Options 选项卡中，用户可以设置与工具按钮相联系的显示提示，以及设置显示提示的同时是否显示该按钮的快捷键。默认情况下，系统会设置显示提示，并且同时显示快捷键，如图 1.38 所示。这样，将鼠标指针放在一个工具按钮上停留一会儿，将显示出该按钮的提示框，如果该按钮具有快捷键设置，则该快捷键也会同时显示在提示框中。

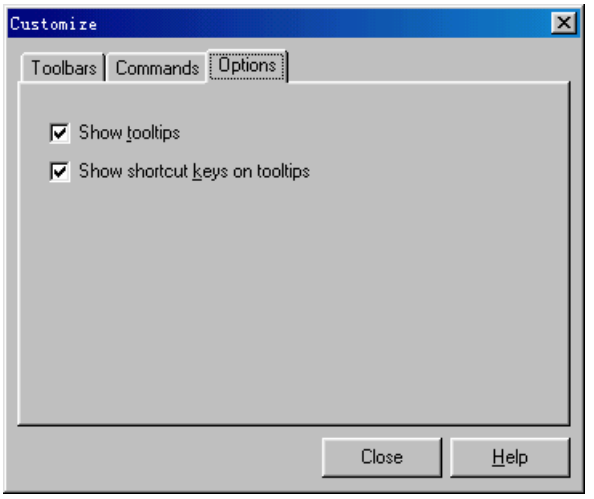

图 1.38 Customize 对话框的 Options 选项卡

1.5 组件选项板

Delphi 6.0 的主界面除了上面介绍过的主菜单以及几个工具栏以外，Component Palette 组件选项板还占据着重要的位置，如图 1.39 所示。

图 1.39　组件选项板

组件选项板中按选项卡分组列出了各种组件，每一选项卡基本上是按照组件的功能来分类的，选项卡能够显示出当前选项卡中组件的种类，例如 Data Access 选项卡中放置的基本是用于数据访问的组件，而 Data Controls 选项卡中放的则是用于数据控制，即数据感知组件。

而且，Delphi 的组件选项板是完全开放的，这意味着用户不仅可以向某一选项卡上添加用户自己的组件，而且还可以创建任意一类组件组选项卡，其中放置用户自己编写的组件。事实上，大量的第三方组件都是用户自己创建特定名称的组件选项板选项卡，一个 Delphi 编程高手的组件选项板上常常包括大量的自己或者别人编写的组件，这些组件往往占据着几十个选项卡。在后面的第 7 章“编写自己的组件”一章中将讲解如何创建编写用户自己的组件，以及如何把自己的组件加入到组件选项板上。

如果一组件选项卡上的可用空间被组件占满后，在组件选项板的右边将出现一个右箭头，用它可以滚动并显示隐藏着的组件，当原先显示着的组件隐藏后，在组件面板的左边将出现一个左箭头用以回滚。

1.6　Delphi 6.0 的主编辑器

Delphi 6.0 中最重要的编辑器是占据重要位置的代码编辑器和窗体编辑器，程序员在这里可以完成全部的窗体设置和程序代码编辑。

代码编辑器用来供程序员查看和编辑 Object Pascal 程序代码。强大的 Delphi 代码编辑器能够自动地完成 Object Pascal 大部分的程序框架代码，这个功能大大节省了程序员书写框架代码的时间，从而可以把精力集中到功能代码的编写上。

当打开一个全新的工程的时候，会看到代码编辑器中已经有了几行代码，这些代码就是一个 Object Pascal 程序的主框架。对这部分主框架程序代码，除非用户已经非常清楚它的结构和作用，否则建议不要轻易改动，因为错误的主框架代码有可能破坏整个程序。

```
unit Unit1;
interface
uses
    Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
    Dialogs;
type
    TForm1 = class(TForm)
    private
        { Private declarations }
    public
```

```
        { Public declarations }
    end;
var
    Form1: TForm1;
implementation
{1R *.dfm}
end.
```

Delphi 中的代码编辑器和窗体编辑器结合得非常紧密，而且是自动化的。在窗体编辑器上的改动不需要程序员动手就能够自动反映到代码编辑器上，例如在窗体上添加一个组件，代码编辑器中将会自己产生一行该组件的声明语句，再比如，在窗体上双击一个按钮，Delphi 将自动在代码编辑器中产生一个按钮默认的 Click 单击事件的框架代码，并且光标停在填充程序内容的地方，如图 1.40 所示就是产生按钮单击事件的框架代码。

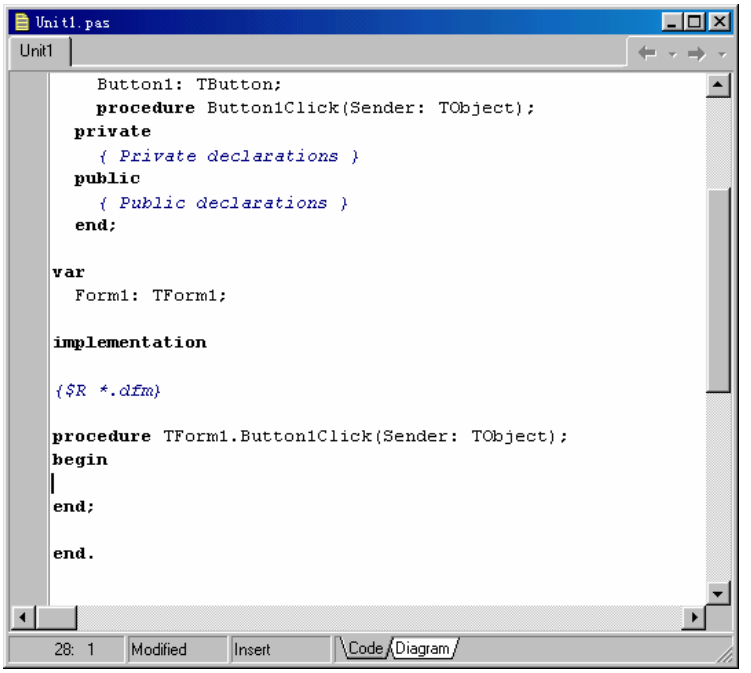

图 1.40　代码编辑器自动产生的框架代码

事实上，代码编辑器并不仅仅是一个简单的查看和编辑代码的工具，它还包含很多隐含的功能。

- 自动提示

将鼠标放到任何一个类型名称、方法或者函数名称上提留片刻，代码编辑器中将出现一个提示框，其中显示了该类型或函数方法的声明，以及在哪个单元中声明的等信息。

- 自动定位

按 Ctrl 键，然后单击任何一个类型名称、方法或者函数名称，代码编辑器将自动把用户带到该类型或者函数方法声明定义的地方，如果是在另外一个单元中，该单元将同时被打开。

- **函数声明和函数内容之间的定位**

在一个单元的开头部分会有所有该单元中函数的声明部分，但是具体函数的内容部分是在单元后面的部分。将光标停止在一个函数声明部分，然后按下 Shift + Ctrl +上/下方向键组合键，代码编辑器将自动定位到该函数的内容部分，同样，从函数的内容部分使用同样的方法可以定位到函数的声明部分。

- **自动生成函数框架**

前面已经讲到，代码编辑器能够自动产生按钮单击事件的框架代码，实际上，对于那些自己声明的函数，用户只要写好了函数声明，将光标停在函数的声明代码行上，然后按下 Ctrl + Shift + C 键就可以让编辑器自动产生函数的框架代码，而且如果前面已经声明了多个函数，这个组合键能够同时产生所有这些声明函数的框架代码。

- **标签**

在代码编辑器中按 Shift+ Ctrl +数字键组合键，将会在当前代码行上出现一个绿色的标签符号，其中显示着组合键中的数字，如图 1.41 所示。在当前单元的任何地方，按下 Ctrl + 数字键组合键，光标就可以自动回到标有该数字的标签处。要取消一个标签，只需要再次按下 Shift + Ctrl +数字键组合键即可。

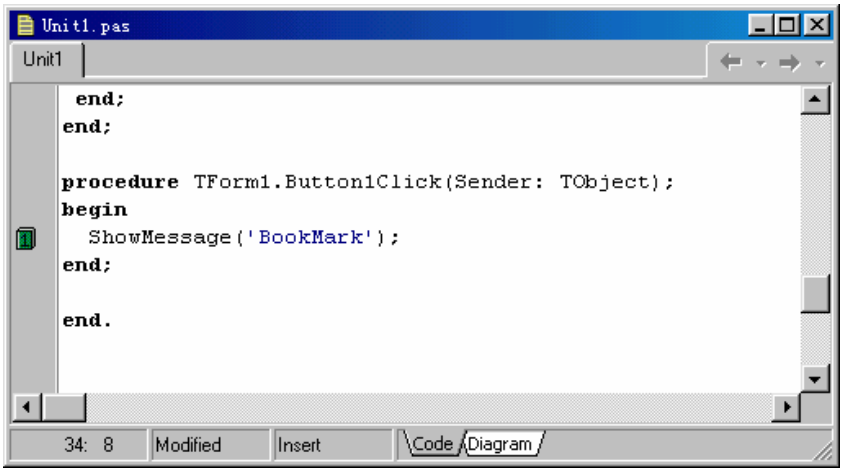

图 1.41　在代码上做标签

随着对 Delphi 开发环境的不断熟悉，还会发现它更多的强大功能。事实上，在代码编辑器上右击后出现的快捷菜单中，包含了大量有用的功能，该菜单如图 1.42 所示。

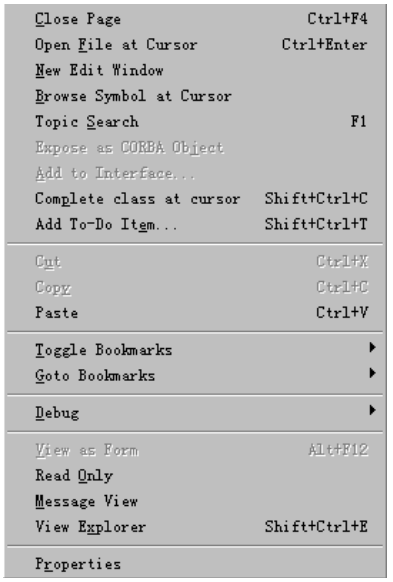

图 1.42　**代码编辑器中的快捷菜单**

1.7　Delphi 6.0 中的快捷方式

随着对 Delphi 6.0 **的不断熟悉和越来越多的使用，用户会不断学习到一些使用** Delphi 6.0 **工具的技巧，其中对各种命令（按钮或者菜单）的快捷方式就是一个非常有用的技巧，它能够大大地提高使用** Delphi 6.0 **的效率。而且，能够最大程度地使用户脱离开鼠标，因为用户会发现不断地使用鼠标去打开菜单执行一个命令是一件非常繁琐的工作，而且很容易使用户的手感到疲劳。**Delphi 6.0 **中提供的快捷方式，如果能够灵活使用，基本可以不使用鼠标。下面就介绍一些经常会使用的快捷方式，其实在每一个菜单的命令上都可以找到这些快捷方式。**

代码编辑期间：

- F12：**切换代码编辑器和窗体编辑器，即** Toggle Form/Unit **命令。**
- F11：**显示对象监视器。**
- Ctrl + F12：**显示单元列表，从中选择一个单元打开。**
- Shift + F12：**显示窗体列表，从中选择一个窗体打开。**
- Alt + F11：**向当前单元中引用另外一个单元，即加入** Uses **单元中。**
- Ctrl + F：**查找指定字符串。**
- Ctrl + E：**快速查找字符，即** Incremental Search**。**
- F3：**继续向后查找。**
- Alt + A：**全选当前所有代码。**
- Ctrl + Z：**撤消上次操作。**

代码运行调试期间：

- F9：**运行程序。**
- Ctrl + F9：**编译程序。**
- F8：**逐行运行程序。**
- F7：**运行代码进入当前函数。**
- F5：**代码编辑器中，在光标所在行设置或者取消一个断点。**
- Ctrl + F7：**监视/更改一个变量或者表达式的值。**
- Ctrl + F2：**终止程序的运行。**

1.8　本 章 小 结

本章重点介绍了 Delphi 6.0 **的操作界面，包括菜单、工具栏、组件选项板，以及代码编辑器、对象监视器等，通过这些介绍，刚刚接触** Delphi **的读者将对它有一个基本的认识，而对** Delphi **以前版本比较熟悉的读者也会对** Delphi 6.0 **众多的新特性有一个了解。**

本章中所介绍的内容仅仅为读者起到一个帮助了解的作用，而对 Delphi 6.0 **开发环境的掌握程度最终还是要取决于对它的使用和总结。**

第 2 章　Object Pascal 语言

本章的重点主要介绍 Delphi 所使用的 Object Pascal 程序设计语言。首先介绍 Object Pascal 语言的基础，例如语法规则；然后介绍 Object Pascal 语言的高级技术，例如类和异常处理；最后，本章将介绍一些重要的概念，例如变量、类型、运算符、循环、条件、异常和对象。即使读者已经有 Object Pascal 语言的编程经验，仍会发现这一章很有用，因为这里讲述了 Pascal 语法的内在本质。

大多数使用 Delphi 开发软件的程序设计人员都是在 Delphi 集成开发环境——IDE 中编写和编译程序，Delphi 可以自动处理源程序文件设置上的细节，也可以替程序的组织框架加上一些限制，而这些限制严格来说并不属于 Object Pascal 程序设计语言所定义的规范。本章假设是在 Delphi 的集成开发环境下，并且使用 VCL 来开发应用程序。

2.1　Object Pascal 程序框架

Object Pascal 是一种高级的程序语言，它支持结构化与面向对象两种程序设计方法。用 Object Pascal 编写的程序可读性强、编译速度快，它会将应用程序分解成多个程序单元，以便达到模块化的程序设计。

2.1.1　工程主程序框架

一般来说，在开发过程中会把一个程序拆开成数个源程序 Module（模块），这些程序模块在 Object Pascal 当中叫做 Unit（程序单元）。每个程序单元有自己的文件，各自独立编译，编译后的程序单元链接在一起之后，就成为一个应用程序。

传统的 Pascal 程序设计方式，是把所有的源程序文件以 pas 为扩展名存储起来，包括主程序也一样。但在 Delphi 中，主程序存放在工程文件即 DPR 文件中，每个应用程序都是由一个工程文件、一个或多个程序单元文件组成。

一个 Delphi Object Pascal 程序包括：一个程序头、一个 Uses 子句、一个包含 Declaration（声明）与 Statement（语句）在内的 Block（程序区）块。程序头部分标明了这个程序的名字，Uses 子句列出的是这个程序用到的各个程序单元，而程序区块则包含了声明部分，以及程序所要执行的那些语句。严格来讲，一个工程中也可以不引用任何程序单元，不过，即使如此，每个工程还是会自动引用 System 这个程序单元。在编译工程文件的时候，对每个引用到的程序单元，编译程序需要这些单元的源程序文件或者编译好的 DCU 文件。

以下示例是一个程序的工程文件：

```
program Project1;
uses
    Forms,
```

```
        Unit1 in 'Unit1.pas' {Form1};

{2R *.RES}
begin
        Application.Initialize;
        Application.CreateForm(TForm1, Form1);
        Application.Run;
end.
```

程序的第 1 行就是所谓的单元头；第 2 行~第 4 行是 Uses 子句；第 5 行是一个编译命令，它会把这个工程的资源文件与程序链接在一起；第 6 行~第 10 行是程序区块，要执行的代号都写在这里；最后，就和每个源程序文件一样，这个工程文件也要以一个句号结束。事实上，这个程序实例是一个比较典型的工程文件，工程文件通常都很简单，因为大部分的程序逻辑都是写在它引用的那些程序单元文件里。工程文件一般都是由 Delphi 自动生成和维护的，几乎不需要手工编辑它们。

程序头指明了这个程序的名字，它包含了保留字 Program，接着是一个合法的 Identifier（标识符）作为工程名字，然后跟着一个分号。在 Delphi 中，这个标识符必须和工程文件的主文件名相同，事实上，当用户通过 Delphi 来保存工程时，所保存的 DPR 文件（工程文件）将自动地确认为程序头中的程序名。

在 Uses 子句中所列出的程序单元，会链接到应用程序当中。这些程序单元里头也可能会有它们自己的 Uses 子句。

所谓程序区块，指的是在运行时刻，程序所执行的一组简单的语句，或者是一段结构化的语句。在大部分的 Delphi 工程文件中，程序区块是由一个夹在 Begin 和 End 这两个保留字中的 Compound Statement（复合语句）所构成。复合语句里的各个语句不过是去调用一堆 Application 对象的 Method（方法）。在这个程序区块中也可以包含常数、数据变量、变量、过程和函数声明等，不过这些声明必须写在程序区块的程序语句部分之前。

2.1.2　程序单元框架和语法

程序单元是构成 Pascal 程序的独立的代码模块，它由数据类型、常数、变量以及函数和过程所构成，其中的函数和过程可以被主程序调用。每个程序单元都定义在自己的 PAS 文件（程序单元文件）中。程序单元文件是由程序单元头开始，接着是 Interface、Implementation、Initialization 和 Finalization 部分，其中后面两个部分可以不用定义。程序单元的框架如下所示：

```
unit Unit1;

interface

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

```
uses Unit2;
initialization

finalization

end.
```

1．程序单元头

程序单元头部分指定好了一个程序单元的名字。程序单元头包含保留字 Unit，接着是一个合乎语法规则的标识符，然后跟着一个分号。在同一个工程中的程序单元名字必须是独一无二的，即使分别放在不同的目录中，程序单元的名字也不能相同。

2．Interface 部分

程序单元的 Interface 部分从保留字 Interface 开始、一直到 Implementation 部分的开头为止。Interface 部分用来声明常数、数据类型、变量、过程、函数，这些声明是要让其他程序或者程序单元能够直接使用这些数据或函数。过程或者函数在 Interface 部分只是声明其头部分，它们的主体写在后面的 Implementation 部分当中。如果要在 Interface 部分声明新的类，这个类中的所有成员必须一并在这一部分声明。

3．Implementation 部分

Implementation 部分是用来定义在 Interface 部分里面声明的过程和函数，这些函数和过程可以以任意的顺序定义和调用。比较特别的是，在 Implementation 部分里定义这些公开声明在 Interface 部分里的函数和过程的时候，它们的参数部分可以省略不写，但是如果要把参数写出来，就必须与在 Interface 部分中声明的一模一样。

除了定义公共函数和过程以外，Implementation 部分也可以声明常数、数据类型、变量、过程和函数。但是这些项目都是这个程序单元 Private（私有）的，也就是说，其他的程序和程序单元不能访问它们。

4．Initialization 部分

Initialization 部分根据实际情况可有可无。它包含了一些语句，在程序一开始的时候，这些语句会以它们写出来的顺序执行。所以，如果在程序中有一些需要设置初值的数据结构的话，就可以把初始化的这些程序代码写在 Initialization 部分中。

在一个程序或程序单元中所用到的各个程序单元中，它们的 Initialization 部分执行的先后关系，是以它们在 Uses 子句中出现的顺序决定的。

5．Finalization 部分

在程序单元的框架中，Finalization 部分也是可有可无的，不过特别的是，它只能出现在 Initialization 中。Finalization 部分包含的是在程序要结束的时候所要执行的语句，可以利用 Finalization 部分来释放那些在 Initialization 部分分配的资源。

各个程序单元中 Finalization 部分执行的先后顺序，刚好与 Initialization 部分执行的顺序相反。一旦程序单元中的 Initialization 部分的代码已经执行，那么在应用程序结束的时候，Finalization 部分中的代码就一定会被执行到。

2.1.3　程序单元引用方式与 Uses 子句

Uses 子句中所列出的，是这个 Uses 子句所在的程序、程序库或者程序单元要引用到的程序单元。Uses 子句可以用在很多地方，包括：

- 一个应用程序或者程序库的工程文件
- 一个程序单元的 Implementation 部分
- 一个程序单元的 Interface 部分

大部分的工程文件中都会有 Uses 子句，大部分程序单元的 Interface 部分也都会有它，同样的，程序单元的 Implementation 部分也可以有它自己的 Uses 子句。

每个 Delphi 应用程序都会自动引用 System 这个程序单元，可是它却不能包括在 Uses 子句中，否则将会引发重复定义的错误。除此以外，其他属于标准程序库里的程序单元，例如 SysUtils，如果需要引入的话，就必须要清楚地写在 Uses 子句中。在大多数情况下，在 Delphi 生成以及维护一个源程序文件的时候，它会自动把要用到的程序单元加入 Uses 子句中。

下面讨论 Uses 子句的一些语法要求。Uses 子句是由保留字 Uses 后跟一个或多个以逗号隔开的程序单元名、再加上一个分号所构成的。例如：

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

在应用程序或程序库的 Uses 子句中，一个程序单元名后面还可以写保留字 in 和一个用小括号括起来的源程序文件的文件名，这个文件名中的路径可有可无。例如：

```
uses
    Forms,
    Unit1 in '..\Source\Unit1.pas' {Form1},
    Unit2 in 'D:\Unit2.pas' {Form2};
```

在需要指明程序单元的源程序文件时，才需要在程序单元名后写出 in 和文件名。因为 Delphi 的集成开发环境本来就默认程序单元名会与其相应的源程序文件名一致，基本上并没有写出 in 和文件名的必要。只有在源程序文件的存放位置不是很明确的时候，才有必要写出 in 和文件名。

在程序单元的 Uses 子句中，不能使用 in 来告诉编译程序该到什么地方去找所需的源程序文件。每一个程序单元都必须存放在编译程序的搜索路径、Delphi 的程序库搜索路径中，或者是与使用它的程序单元存放在相同的目录下。而且，程序单元名必须与它们相应的源程序文件的文件名一致。

注意：有时候，可能会出现这样一种情况，即单元 A 引用了单元 B，而单元 B 又引用了单元 A，这称为循环引用。如果出现了循环引用，说明程序的结构设计有缺陷，应当尽量避免循环引用。一个较好的解决方案是，把单元 A 和单元 B 都要用到的代码放到第三个单元中去。不过，正如大多数情况一样，有时候可能无法

避免循环引用。这种情况下，必须把其中一个 Uses 子句移到 Implementation 部分中，而把另一个留在 Interface 部分。这样做通常能解决这个问题。

2.2 Object Pascal 语法元素

Object Pascal 所使用的 ASCII 字符集，包含了英文字母 A~Z、a~z，数字 0~9，其他字符。空格字符（即 ASCII32）及控制字符（即 ASCII0~ASCII31，包含 ASCII13 换行字符）称为 Blanks（空白）。

语法元素中最基础的单元称为 Token（标记），包含了 Expression（表达式）、Declarations（声明）及 Statements（语句）。语句描述了程序实际能够被执行的算法动作；表达式是出现在语句中的语法单元，用来代表一个数值；声明则定义了能够运用于表达式及语句中的 Identifier（标识符，例如变量名和函数名），并在适当时机为该标识符分配适当的内存。

简而言之，程序是由一连串标记和分隔符组成的，标记是指组成程序中最小的单元且有意义的文字，分隔符则是一个空白或是注释。严格来说，在两个分隔符之间并不一定要放置一个分隔符，例如下面的代码：

```
I:=20;J:=100;
```

是完全符合语法规则的。但考虑到程序编写的约定和可读性，应写为：

```
I := 20;
J := 100;
```

关于代码的编写格式规范将在第 4 章详细介绍。

标记可分为特殊符号、标识符、保留字、伪指令、数字、标号和字符串。字符串常数中只有分割符号被视为标记的一部分，两个相邻的标识符、保留字、数字及标号之间都必须有一个或多个分隔符号。

2.3 注　释

作为起点，用户应当掌握怎样在 Pascal 代码中加注释。

Object Pascal 语言支持 3 种形式的注释：花括号注释、圆括号和星号注释、C++风格的双斜杠注释。下面分别是这 3 种形式的注释示例：

- {…} ——使用花括号注释。
- (*…*) ——使用圆括号和星号注释。
- //——使用双斜杠注释。

前两种注释方法将前后括号内的内容全部注释起来，而第三种注释方法只注释双斜杠至行末的代码。

注意：同形式的注释是不能嵌套使用的，但不同形式的注释可以嵌套使用。即使

这样，笔者仍然不赞同嵌套使用注释。

2.4 变　量

有的用户过去可能习惯于在程序代码的任意一个地方声明变量，但是，在 Delphi 中声明变量是有所限制的。Object Pascal 要求用户必须在开始一个过程、函数或程序前声明所有的变量，如下所示：

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

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

var
    Form1: TForm1;
    I: integer;    //在这里定义全局变量
    J: string;

implementation
{2R *.DFM}
procedure TForm1.Button1Click(Sender: TObject);
var
    I: integer;    //在这里定义局部变量
    J: string;
begin
    {do something...}
end;

end.
```

Object Pascal 语言不分大小写，这一点不同于 C++语言，但对于一个复合的单词来说，适当地运用大小写可以使它更加醒目，例如下面的这种写法就不太好：

```
procedure getcountryname；
```

但如果改成下面的写法就清楚多了：

```
Procedure GetCountryName；
```

Object Pascal **语言的这种结构化风格的好处很明显，改善了代码的可读性和可维护性，最大限度地减少了出错的可能性。**

Object Pascal **运行几个相同类型的变量应在同一行中声明，如下所示：**

```
VarName1，VarName2：String；
```

从 Delphi 2 **开始，就允许在声明全局变量的时候对变量赋初值，例如：**

```
Var
    I: integer = 10;
    S: string = 'hello';
    D: Double = 3.14;
```

注意：只有全局变量才能在声明的同时赋初值，而在过程或者函数内部声明的局部变量则不允许这种操作。对于全局变量，即使没有对全局变量赋初值，Delphi **也会自动对全局变量赋初值。对于** Integer **类型的全局变量来说，初值是** 0**；对于** Real **类型的全局变量来说，初值是** 0.0**；对于指针类来说，初值是** nil**；对于** String **类型来说，初值是空字符串。因此，除非要赋的初值与默认值不同，否则不需要显式地赋初值。**

2.5 常　量

C **语言中用关键字** const **来声明常量，在** Object Pascal **中也是使用** const **来声明常量，但语法稍有不同，下面是** C **语言中声明常量的示例，如下所示：**

```
Const float PI = 3.14;
Const int I = 10;
Const char * MyString = "hello";
```

C **语言与** Object Pascal **语言声明常量的语法的差别在于：**Object Pascal **语言不需要专门指定常量的数据类型，因为编译器会根据常量的值自动判断常量的类型并分配内存，或者对整型常量，运行时编译器将跟踪这个值，但不会分配内存空间。**

当然，声明常量时也可以显式地指定常量的类型，但语法与 C **语言的语法不同。下面是典型的示例：**

```
const
    D: Double = 3.14;
    I: intger = 10;
```

```
S: string = 'hello';
```

在声明常量和变量时，Object Pascal 语言允许在等号的右边出现一些编译器函数，例如 Ord()、Chr()、Trunc()、Round()、Hight()、Low()和 SizeOf()。例如，下面的代码是合法的，如下所示：

```
type
    A = array[1..10] of Integer;]
Const
    W: Word = SizeOf(Byte);
Var
    I: integer = 8;
    J: SmallInt = Ord('a');
    L: LongInt = Trunc(3.14159);
    X: shortInt = Round(2.718);
    B1: Byte = Hight(A);
    B2: Byte = Low(A);
    C: Char = Chr(46);
```

如果程序试图修改常量的值，则编译器会发生警告，因为常量的值是只读的，不能修改。为了优化，编译器把常量放在应用程序的代码页而不是数据页。

2.6　运 算 符

运算符是代码中各种类型的数据进行计算的符号。例如，有的运算符可以进行加、减、乘、除运算，有的运算符可以访问一个数组的某个元素的地址。这一节将详细介绍各种 Pascal 运算符以及 Pascal 运算符与 C、VB 运算符的区别。

2.6.1　赋值运算符

如果用户是一个 Pascal 新手，Delphi 的赋值运算符可能是最不习惯的事情之一。要给一个变量赋值，应当使用“ := ”运算符，而不是像 C 或者 VB 中使用“ = ”运算符。Pascal 程序员通常称“ := ”为获得运算符或者赋值运算符。

2.6.2　比较运算符

如果用户曾经用 VB 编过程序，可能会对 Delphi 的比较运算符感到很舒服，因为这两种语言的比较运算符是很相近的。事实上，大多数编程语言的比较运算符都是差不多的。

Object Pascal 使用“ = ”运算符来对两个表达式或者数值进行比较，在 C 语言中相当于“ = = ”运算符。一个 C 语言的比较表达式通常这么写，如下所示：

```
If (x == y)
```

而在 Object Pascal 的比较表达式通常这么写，如下所示：

```
if x = y
```

注意：在 Object Pascal 语言中，“ := ”运算符用于对变量进行赋值，而“ = ”运算符用于对两个变量或数值进行是否相等的比较。

Delphi 的不等运算符是“ < > ”，相当于 C 语言中的“ != ”运算符。例如，要判断哪个表达式是否不等，代码应该这么写，如下所示：

```
If x < > y then DoSomethingsdf
```

2.6.3 逻辑运算符

Pascal 使用单词 And 和 Or 作为逻辑与和逻辑或的运算符，分别相当于 C 语言中的“ && ”和“ || ”运算符。例如：

```
If (Condition1) and (Condition2) then
    DoSomething;
While (Condition1) or (Condition2) do
    DoSomething;
```

Pascal 语言中的逻辑非运算符是 Not，用于检查一个条件是否为假，相当于 C 语言中的“ ! ”运算符，例如：

```
If not (Condition) then
    DoSomething; //如果条件 condition 为假，则 DoSomething…
```

表 2.1 是一个简明的对照表，列出了 Pascal 运算符与 C、VB 运算符的对应关系。

表 2.1 赋值、比较和逻辑运算符

运算符	Pascal 语言	C 语言	Visual Basic
赋值	:=	=	=
比较	=	= =	=
不等	<>	!=	<>
小于	<	<	<
大于	>	>	>
小于等于	<=	<=	<=
大于等于	>=	>=	>=
逻辑与	And	&&	And
逻辑或	Or	‖	Or
逻辑非	Not	!	Not

2.6.4 算术运算符

用户应该对大部分的 Object Pascal 算术运算符不感到陌生，因为 Pascal 的算术运算符和 C、C++、VB 的算术运算符差不多。表 2.2 列出了 Pascal 的算术运算符和 C、VB 算术运算符的对比关系。

表 2.2　算术运算符

运算符	Pascal 语言	C 语言	Visual Basic
加	+	+	+
减	-	-	-
乘	*	*	*
浮点整除	/	/	/
整数除	Div	/	/
取模	Mod	%	Mod

由上表可以看出（见表 2.2），Pascal 与其他语言的算术运算符的主要区别在于：Pascal 提供了分别针对浮点数和整数的除法运算符，其中，Div 运算符会自动截掉余数取整数。

注意：不同类型的表达式要使用不同的除法运算符，如果把 Div 用于两个浮点数相除，编译器将报告错误。同样，如果把“ / ”用于两个整数相除，编译器也会报错。其他语言往往不区分浮点数相除和整数相除，而是尽量进行浮点数相除，然后根据需要把结果进行转换。从程序性能看，后者的开销较大，而 Pascal 的 Div 运算符要快一些。

2.6.5　按位运算符

通过按位运算符可以修改一个变量的单独的一位。可以把一个数向左或者向右移位，或者进行按位与、或、取反、异或等操作，移位运算符 Shl 和 Shr 分别相当于 C 语言中的“ << ”和“ >> ”运算符。表 2.3 列出了这些按位运算符。

表 2.3　按位运算符

运算符	Pascal 语言	C 语言	Visual Basic
与	And	&	And
或	Or	|	Or
取反	Not	~	Not
异或	Xor	^	Xor
左移	Shl	<<	没有定义
右移	Shr	>>	没有定义

2.7　Object Pascal 数据类型

Object Pascal 中的数据类型定义是非常严谨的，因此，在函数或过程中传递的参数的类型必须与形参的类型严格一致。Pascal 编译器绝对不允许不同类型的指针进行转换。

2.7.1　类型的比较

表 2.4 列出了各种数据类型的比较。

表 2.4 数据类型的比较

位数	Pascal 语言	C 语言	Visual Basic
8 位有符号整数	ShortInt	char	
8 位无符号整数	Byte	BYTE, unsigned short	
16 位有符号整数	SmallInt	short	Short
16 位无符号整数	Word	short	
32 位有符号整数	Integer, Longint	int, long	Integer
32 位无符号整数	Cardinal, LongWord	unsigned long	
64 位有符号整数	Int64	_int64	
4 字节浮点数	Single	float	Single
6 字节浮点数	Real48		
8 字节浮点数	Double	double	Double
10 字节浮点数	Extended	long double	
64 字节 Variant	variant, OleVariant, TvarData	VARIANT, Variant, Ole Variant	
64 位货币值	Currency		Currency
1 字节字符	Char	char	
2 字节字符	WideChar	WCHAR	
固定字节长度的字符串	ShortString		
动态字符串	AnsiString	AnsiString	$
以 Null 结束的字符串	Pchar	char	
以 Null 结束的宽字符串	PwideChar	LPCWSTR	
动态 2 字节字符串	WideString	WideString	
1 字节布尔值	Boolean, ByteBool	任何 1 字节数	
2 字节布尔值	WordBool	任何 2 字节数	Boolean
4 字节布尔值	Bool, LongBool	BOOL	

Delphi 的基本类型与 C 和 VB 差不多。上表（见表 2.4）列出了 Object Pascal 的基本数据类型以及 C/C++、VB 中的基本数据类型。这个表很重要，因为当用户调用非 Delphi 编写的动态链接库或者目标文件时，需要知道确切的数据类型。

2.7.2 字符和字符串

Delphi 中有以下 3 种类型的字符：

- AnsiChar：这是标准的 1 字节的 ANSI 字符。
- WideChar：这是 2 字节的 Unicode 字符。
- Char：等同于 AnsiChar。

由于字符不一定是一个字节，所以不能在程序中限定字符的长度，而要使用 SizeOf() 函数动态地获得字符的长度。

字符串可以看作是一组字符。每一种语言的字符串类型的存储和使用方法也是不同的。Pascal 中存在以下几种字符串类型：

- AnsiString：这是 Object Pascal 的默认字符串类型。该字符串是由 AnsiChar 字符构成的，其长度几乎没有限制。它与以 null 结束的字符串兼容。
- ShortString：保留这个数据类型的目的是为了与 Delphi 1 兼容。字符串最多不超过 256 个字符。
- WideString：功能上类似于 AnsiString，但它是由 WideChar 字符构成的。
- PChar：指向以 null 结束的 Char 字符串的指针，类似于 C 语言中的 char 或 lpstr。
- PAnsiChar：指向以 null 结束的 AnsiChar 字符串的指针。
- PWideChar：指向以 null 结束的 WideChar 字符串的指针。

默认情况下，如果按照下面的代码来声明一个字符串，编译器将把它当作是 AnsiString，如下所示：

```
var
    S：String // 编译器将认为 S 为 AnsiString 类型
```

2.8　用户自定义类型

整数、字符串和浮点数往往不足以表达现实世界中复杂的数据类型。一些情况下，用户必须创建自己的数据类型来更好地表达问题。

2.8.1　数组

Object Pascal 允许用户创建除了文件以外的任何类型的数组。例如，下面的代码声明了一个由 10 个整数组成的数组，如下所示：

```
Var
    MyArray: array[0..9] of Integer;
```

与 C 语言不同的是，Delphi 中的数组元素的序号可以从任意一个数开始，例如，可以声明一个序号从“12”开始的 3 个元素的数组，如下所示：

```
var
    MyArray: Array[12..14] of Integer;
```

Object Pascal 的编译器提供了两个内建的函数 Hight()和 Low()，可以分别返回一个数组中的最大序号和最小序号。为了避免在循环中出现下标超出边界的错误，除非要特意地从某个序号开始或结束，建议用户采用这两个函数来控制循环的开始和结束。

提供：对于字符数组来说，下标最好从“0”开始，0 基准的字符数组可以传递给

PChar 类型的变量，这是编译器特许的。

要定义多维数组，只要在序号范围中多定义几个范围即可，例如：

```
var
    MyArray: Array[0..9, 0..9] of Integer;
```

它定义了一个二维数组，共有 10*10 个元素，要访问其中的一个元素，可以参考下面的示例：

```
I := MyArray[9, 9];
```

2.8.2 动态数组

从 Delphi 4 开始增加了动态数组的功能。所谓动态数组，是指数组的维数和长度在编译期是不确定的，而是在运行期通过程序来设定的。例如：

```
Var
    A: Array of String; //  字符串数组 A 的长度和维数都没有确定
```

运行期在使用动态数组之前，必须调用 SetLength()来为数组分配内存，如下所示：

```
Begin
    SetLength(A, 100); //为数组 A 分配了 100 个元素的空间
End;
```

分配了空间之后，该数组就可以像一般的数组一样访问了。

注意：动态数组的序号总是从“0”开始的。

为动态数组分配了内存之后，在程序终止之前，必须释放它所占用的内存，只能用 nil 对数组赋值，如下所示：

```
A := nil; //  释放数组 A
```

要特别注意，两个动态数组在相互赋值的时候，复制的仅仅是对数据的引用，而不是数据本身，这一点与一般的数组不同。例如：

```
Var
    A1, A2: Array of Integer;
Begin
    SetLength(A1, 4);
    A2 := A1;
    A1[0] := 1;
    A2[0] := 10;
End;
```

执行完上面的代码以后，A1[0]将等于 10，而不是 1。这是因为赋值语句 A2 := A1 并没有在内存中为 A2 新开辟一个数组，而是将 A2 指向了 A1 引用的数组位置。这样，事实

上，A1 和 A2 指向了相同的内存位置，任何对 A2 的修改都会影响到 A1，同样，A1 的变化也会影响 A2。如果想将 A1 的数据复制给 A2，而不是引用，应当调用 Copy()函数：

```
A2 := Copy(A1);
```

执行它以后，A2 将成为一个独立的数组，其中的数据和 A1 中的数据相同，但是如果再修改 A2 中数据，将不会影响到 A1 的数据。Copy()函数可以指定从第几个元素开始复制，复制几个元素，例如：

```
A2 := Copy(A1, 1, 2);
```

它的功能是将 A1 数组中从序号“1”开始的两个元素赋值给 A2 数组。

上面讨论的动态数组都是一维的，对于多维的动态数组可以这样处理：

```
Var
    A: array of array of Integer; // 定义二维的动态整数数组
Begin
    SetLength(A, 10, 10); // 数组 A 为 10*10 的二维整数数组
End;
```

2.8.3　记录

Record（记录）也是 Object Pascal 语言中的一种用户自定义类型，相当于 C 语言中的 struct。下面是在 Delphi 中定义记录的方法，如下所示：

```
Type
    MyRec = record
        I: integer;
        S: String;
End;
```

上面的记录 MyRec 中有两个域：一个整数域，一个字符串域。使用该记录时，可以通过一个小圆点“.”来访问记录的域，例如：

```
Var
    N: MyRec
Begin
    N.I := 100;
    N.S := ‘hello’;
End;
```

2.8.4　集合

集合是 Object Pascal 语言惟一一种在 C 语言、C++、Basic 语言中找不到的数据类型。集合能够把有序数、字符、枚举收集起来。要声明一个集合类型，需要用到关键字 set of，后跟一个有序类型或子界给出值的范围。例如：

```
Type
    TCharSet = set of char;

    TEnum = (Mon, Tue, Wed, Thu, Fri, Sat, Sun);
    TEnumSet = set of TEnum;

    TSubRangeSet = set of 1..10;
    TAlphaSet = set of 'a'..'z';
```

注意：一个集合最多只能包含 256 个元素。另外，只有有序类型才能作为数组中的元素类型。

下面的代码是非法的：

```
type
    TintSet = set of Integer;   // 整型的元素太多了，超出了 256 个
                                // 但 Char 合法，因为 Char 元素在#0-#255 范围内
    TStrSet = set of String;    // String 不是有序类型
```

注意：集合是以位来存储它的元素的，这非常有利于提高访问速度和内存使用效率。如果集合的基类型的元素不多于 32 个，则集合将把它们存储在 CPU 的寄存器中，此时的效率更高。如果元素多于 32 个，集合将把它们存储在内存中。所以，为了获得最优的性能，集合基类型的元素最多不要超过 32 个。

Object Pascal **提供了几种用于集合的运算符。可以使用这些运算符判断集合与集合之间的关系、可以在集合中增删元素，以及求集合的交集等。**

1．关系运算

用 in 这个运算符可以判断一个给定的元素是否属于某个集合。例如：

```
if 'S' in CharSet then      //CharSet 是一个字符集合
    DoSomething…
```

2．增删元素

使用“+”和“–”运算符，或者调用 Include()和 Exclude()过程在一个集合中增加或删除元素。例如：

```
Include(CharSet, 'a');          //在字符集合中增加元素'a'
CharSet := CharSet + ['b'];     //在集合中增加元素'b'
ExClude(CharSet, 'a');          //从集合中删除元素'a'
CharSet := CharSet – ['b']      //从集合中删除具有'b'
```

注意：如果要在一个集合中增删单个元素，最好用 Include()和 Exclude()，而不要用“+”和“–”运算符，因为 Include()和 Exclude()只有一条机器指令，而“+”和“–”运算符有 13+6n 条技巧指令（n 为集合的位数）。

3．交集

使用“*”运算符可以计算两个集合的交集。例如：可以判断一个元素或者多个元素是否属于一个集合，如下所示：

```
if ['a', 'b', 'c'] * CharSet = ['a', 'b', 'c'] then    //如果'a', 'b', 'c'同时存在集合中
    DoSomething…
```

2.8.5　对象

对象可以当作记录，只不过对象中还包含了过程和函数。例如，下面的示例声明了一个对象：

```
Type
    TMyObject = class(TParentObject);
        MyVar: Integer;
        Procedure Myproc;
    End;
```

尽管 Delphi 的对象与 C++对象不同，但声明对象的语法却差不多。下面是 C++中声明一个对象的示例：

```
class TMyObject: public TParentObject
{
    int MyVar;
    void MyProc();
};
```

注意：Object Pascal 的对象与 C++的对象在内存中的布局不同，因此，不能在 Delphi 中直接使用 C++中定义的对象，反过来也是一样。

2.8.6　指针

指针是一种指示内存位置的数据类型。Pascal 的指针类型为 Pointer，有时也叫做无类型指针，因为它只指示内存的地址，而不管所指的数据类型是什么。

有类型的指针是用“^”和关键字 Pointer 在程序的 Type 部分声明的。对于有类型的指针来说，编译器能够精确地跟踪指针所指的数据类型。下面是典型的声明指针的示例：

```
Type
    PInt = ^Integer;              //PInt 是指向 Integer 的指针
    MyRec = record
        S: String;
        R: Real;
    End;
    PMyRec = ^MyRec;              //PMyRec 是指向 MyRec 类型的指针
```

注意：指针属于高级编程技巧，对于编写 Delphi 应用程序来说并不是必需的。随

着用户编程经验的增加，指针将成为一个很有价值的工具。

指针的变量只存储内存地址，而为指针所指的数据分配空间则是程序员的事情。表 2.5 列出了所有的内存分配和释放的函数。

表 2.5 分配和释放内存的函数

分配内存	释放内存
AllocMem()	FreeMem()
GlobalAlloc()	GlobalFree()
GetMem()	FreeMem()
New()	Dispose()
StrAlloc()	StrDispose()
StrNew()	StrDispose()
VirtualAlloc()	VirtualFree()

要访问一个指针所指的数据，可以在指针变量名称后面跟“^”运算符。下面的示例说明了指针的用法：

```
Procedure UsePointer;
Var
    Rec: PMyRec;
Begin
    New(Rec);          //为指针分配内存
Rec^.S := 'hello';
Rec^.R := 3.14;
Dispose(Rec);      //释放内存
End;
```

上例中使用了 New()函数来为指针分配内存，该函数能够指定分配的长度。由于编译器知道空间的确切大小，因此，调用 New()函数能够分配正确的字节数，这比 GetMem()和 AllocMem()函数更加安全和方便。

2.9 条 件 语 句

这里将介绍 Pascal 语言中的 If 语句和 Case 语句，并与 C 语言中的相应语句进行比较。

2.9.1 If 语句

If 语句用于判断某个条件是否满足，以决定是否要执行后面的代码。为了比较，下面列出了 If 语句在 Pascal、C 中的语法。

在 Pascal 中如下所示：

```
if x = 4 then y := x;
```

在 C 中如下所示：

```
if (x = = 4) then y = x;
```

在 Pascal 中，如果在一条 If 语句中要判断多个条件，则需要把这几个条件用括号括起来，例如：

```
if (x = 4) and (y = 5) then…
```

下面的写法将导致错误：

```
if x = 4 and y = 5 then…
```

对于多条执行语句，需要把它们用 Begin 和 End 关键字包含起来，就像 C 语言中的“{”和“}”。例如：

```
if x = 4 then begin
        DoSomething…
DoOtherThing…
…
end;
```

也可以用 If...Else 结构来组织多重条件，如下所示：

```
if x = 100 then
        SomeThing…
Else if x = 200 then
        SomeOtherThing…
Else begin
        ElseThing…
        …
End;
```

2.9.2　Case 语句

Pascal 中的 Case 语句非常类似于 C 和 C++中的 switch 语句。Case 语句的优势是：可以从多个可能的条件中选择一个条件，而不必用过多的 If…Else 结构。下面的示例演示了 Case 语句的用法：

```
Var
        I: integer;
Case I of
        0:   DoSomething1…
        1:   DoSomething2…
        2:   begin
                DoSomething3…;
                DoSomething4…;
```

```
            End;
    Else DoElseThing…
End;
```

注意：Case **语句的选择因子必须是有序类型，例如整型，字符等，而不是有序的类型例如字符串是不允许作选择因子的。**

上面的代码如果用 C **语言中的** switch **语句来改写，如下所示：**

```
switch (I)
{
    case 0:   DoSomething1…break;
    case 1:   DoSomething2…break;
    case 2:   begin
                DoSomething3…;
                DoSomething4…;
                break;
                  End;
    Default: DoElseThing…
}
```

2.10 循 环 结 构

循环语句能重复进行某个动作。Pascal **的循环结构非常类似于其他语言中的循环结构，所以，这里不再花太多的时间来教用户怎样使用循环。下面将描述** Pascal **中的几种不同的循环结构。**

2.10.1 For 循环

For **循环适合于重复次数确定的情况。下面的代码是：在一个循环中把控制变量加上另一个变量，共循环** 10 **次。**

```
Var
    I, X: integer;
Begin
    X := 0;
    For I := 1 to 10 do
        Inc(X, I);
End;
```

2.10.2 While 循环

While **循环适合于需要先判断条件是否为真，再决定是否循环的情况。一个典型的示例是，在对文件进行操纵时，只要没有遇到文件尾就一直循环。下面的代码演示了从文件中每次读取一行，然后把它显示在屏幕上。**

```
Var
    F: TextFile;
    S: string;
Begin
    AssignFile(f, 'C:\example.txt');
    Reset(f);
    While not EOF(f) do begin
        Readln(f, S);
        Writeln(S);
    End;
CloseFile(f);
End;
```

Pascal 中的 while 循环基本上与 C 语言和 VB 相同。

2.10.3　Repeat…Until 循环

Repeat…Until 循环与 While 循环相似，但考虑问题的角度不同。对于 Repeat…Until 循环来说，在某个条件为真之前，循环将一直进行。与 While 循环不同的是，Repeat…Until 循环的代码将至少被执行一次，因为条件是在循环之后才判断的。Repeat…Until 语句与 C 语言中的 do…while 语句基本相同。

下面是一个循环语句的示例：它不断把一个计数器加 1，直到它大于 100 为止。

```
Var
    X: integer;
Begin
    X := 1;
    Repeat
        Inc(X);
    Until x>100;
End;
```

2.10.4　Break 语句

在 While 循环、For 循环或者 Repeat 循环内调用 Break 语句将导致程序的流程立即跳到循环尾。当某种条件满足时，用户可能要立即退出循环，此时就要调用 Break 语句。Pascal 的 Break 语句的作用非常类似于 C 的 break 语句和 VB 中的 Exit()语句。下面的代码演示了在 5 次循环之后调用 Break 语句中断循环：

```
Var
    I: integer;
Begin
    For I := 1 to 100 do
    Begin
        MessageBeep(0);   // 计算机将发出‘嗒’的声音
```

```
            If I = 5 then break;
        End;
End;
```

2.10.5 Continue 语句

Continue 语句的作用是使程序跳出当前的循环，然后进入下一次的循环。下面的代码演示了 Continue 语句的用法。

```
Var
    I: integer;
Begin
    For I := 1 to 100 do
    Begin
        Writeln(I, 'Jump out…');
        If I = 10 then continue;
        Writeln(I, 'After Jump');
    End;
End;
```

2.11 过程和函数

作为一个程序员，可能对过程和函数已经非常熟悉了。Procedure（过程）是一段相对独立的代码，它能够在被调用时执行某种任务，然后返回到调用它的地方。Function（函数）和过程类似，不同的是函数能够而且必须返回一个值。

如果用户已经对 C 或者 C++比较熟悉，Pascal 的过程实际上相当于 C 或 C++的 void function（无返回值函数），而 Pascal 的函数相当于 C 中的有返回值函数。下面的示例列出了过程和函数的程序：

```
procedure MyProcedure(I: integer);    //过程
begin
    if I > 10 then writeln('Larger than 10');
end;

function MyProcedure(I: integer): Boolean;    //函数
begin
    if I > 10 then Result := True
    else Result := False;    //如果参数 I 大于 10，函数返回真。
end;
```

Pascal 允许通过值或者引用来传递参数给过程和函数。参数的数据类型可以是任何标准的数据类型，也可以是用户自定义的类型，甚至开发数组。参数也可以作为常量来传递，这时，参数的值在过程和函数中就不会变化。

- **值参数**

把参数以值的形式传递是默认的传递方式。当一个参数以值的形式传递时，意味着参数的值被复制了一个副本，过程和函数实际上是工作于这个副本的。例如下面的代码：

```
procedure work(S: string);
```

当用户调用这个过程并且传递了一个字符串给它的时候，字符串将被复制了一个副本，Work()过程将工作于这个副本，对这个副本的任何修改不会影响本来的变量。

- **引用参数**

Pascal 允许通过引用来传递参数给过程或函数。通过引用来传递的参数也叫做变量参数。当过程和函数收到了传递过来的变量以后，可以修改变量的值。要把一个变量以引用的形式传递，声明过程或者函数的时候需要用关键字 var 加在参数名称的前面，如下所示：

```
Procedure Work(var S: string)
Begin
    S := 'New';
End;
```

由于加了 var 这个关键字，所以这里传递的是变量的地址而不是副本，参数将可以被过程修改。上面的代码执行后，变量“S”的值将被修改为“New”。

- **常数参数**

如果用户不希望传递给过程或者函数的参数被修改，可以使用关键字 const 来声明一个参数。const 关键字不仅可以防止参数被修改，而且对于字符或者记录类型的参数，还能优化代码。

2.12　包

package（包）的技术能够把应用程序的部分代码放到一个单独的模块中，从而使这段代码能够被其他应用程序共享。如果打算利用过去 Delphi 1 或 Delphi 2 中的代码，将看到包的好处，因为用户不需要修改已有的代码。

包可以看作是若干个单元集中在一起以类似于 DLL 的形式存储的 DPL（Delphi Package Library），应用程序可以在运行期而不是编译期链接包中的单元。由于被链接的单元仍然在 DPL 文件中，而不是在 EXE 或 DLL 中，因此，EXE 或 DLL 的长度可以非常小。用户可以创建和使用以下 4 种包：

- **运行期包**

这种类型的包中包含了要在运行期被应用程序使用的单元。当应用程序运行时，需要找到它所要使用的运行期包，否则无法正确执行。Delphi 的 VCL30、VCL40、VCL50 和 VCL60.DPL 包就是典型的运行期包。

- **设计期包**

这种类型的包中包含了组件、特性和组件编辑器、专家等需要的单元。用户可以使用

Component | Install package 命令把一个设计期包安装到组件库中。Delphi 的 DCL*.dpl 文件就是这种类型的包。

- 既是运行期又是设计期包

这种类型的包既可以作为运行期包，也可以作为设计期包。创建这种类型的包能够使应用程序的开发和分发变得简单，但这种类型的包的效率不是很高，因为它必须携带大量的信息。

- 既非运行期又非设计期包

这种类型的包很少见，通常用于被其他包引用，而不是直接被应用程序引用。

要在应用程序中使用包的技术非常容易，只要在 Project | Options | Packages 对话框中单击 Build with Runtime Package 复选框即可。以后当用户编译和运行应用程序时，应用程序会动态链接运行期包的单元，而不是把包的单元静态的链接到 EXE 或者 DLL 中。这将使应用程序变得更加精简。

包通常是用包编辑器创建的。要启动包编辑器，可以执行 File | New | Package 命令。包编辑器会生成一个 DPK（包的源文件），并把此编辑成包。DPK 文件的语法相当简单，其格式如下所示：

```
Package MyPackage

Require Package1, Package2…;
Contains Unit1, Unit2…;
End;
```

列在 Require 子句中的包是这个包需要调用的其他包，列在 Contains 子句中的单元是这个包所包含的单元。在 Contains 字节中列出的单元将被编译进这个包中。注意，这里列出的单元不能同时被列在 Requires 子句中的包所包含。另外，除非它们已经被列在了 Requires 子句中的包所包含，由这些单元所引用的单元也会间接地包含到包中。

2.13 面向对象编程

下面将介绍关于 OOP（Object Oriented Programming，面向对象编程）的一些基本原则，这是 Object Pascal 语言的基础。这里，笔者不会讨论面向对象的编程方法到底有什么优点和缺点，因为已经有很多书已经讨论过了。

OOP 的精髓就是对象（一种既包含数据又包含代码的实体）。尽管 OOP 并不能使代码更加容易编写，但它能够使代码容易维护。通过把数据和代码封装在一起，能使定位和修复错误的工作得以简化，并最大限度地减少对其他对象地影响。一般来说，一门面向对象的编程语言至少要实现下列 3 个 OOP 概念：

- 封装性：把相关的数据和方法封装在一起，同时隐藏实现的细节。封装的好处是有利于程序的模块化，并且把自己的代码与其他的代码分开。

- 继承性：是指一个新的对象能够从祖先对象中获取有的成员和行为。这样，要创建诸如 VCL 的多层次对象，用户可以首先创建一个通用的对象，然后派生出新的对象，使派生对象的功能更加专业。继承性的好处是，可以充分共享已有的代码。
- 多态性：从字面上看，多态性是指“很多形态”，调用一个对象变量的方法时，实际被调用的代码与变量中对象的实例有关。

在进一步讨论对象的概念之前，读者必须要明白以下 3 个术语：

- Field（字段）：是一个对象的数据。对象的字段类似于 Pascal 中的记录的域。字段也可以称作对象的数据成员。
- Method（方法）：专指对象中的过程和函数。在 C 中也叫成员函数。
- Property（特性）：是外部代码访问对象中的数据和代码的窗口。特性隐藏了一个对象具体实现的细节。

Delphi 是一个真正面向对象的环境。这意味着用户既可以从头开始创建一个新的对象，也可以继承一个已有的对象并派生出一个新的对象。这其中包含所有的 Delphi 对象，而不管对象是可视的还是不可视的或者甚至是设计时的 Form 对象。

2.14　Delphi 对象

前面说过对象是同时包含数据和代码的实体。Delphi 的对象具有面向对象编程所具有的强大优势，全面支持继承、封装和多态。

2.14.1　声明和创建实例

在使用一个对象之前，必须在程序或单元的 Type 部分，使用关键字 class 来声明这个对象：

```
Type
    TMyObject = class;
```

声明了一种对象类型之后，通常还要在 Var 部分声明它的一个实例：

```
var
    MyObject: TMyObject;
```

在 Object Pascal 中，要创建一个对象的实例，可以通过调用它的构造函数。构造的作用是创建对象的实例、分配内存并且对字段进行初始化。Object Pascal 的对象至少要有一个构造函数 Create()，尽管一个对象还可以有其他构造。对于不同类型的对象来说，Create()可以带不同数量的参数。这里主要讨论不带参数的 Create()。

与 C++不同的是，Object Pascal 对象的构造不是自动调用的，程序员必须自己调用构造。下面是一个对象的构造函数：

```
MyObject := TMyObject.Create();
```

可以看到，调用函数是由类型，而不是像其他方法那样由实例来调用的。通过调用构造来创建对象的实例，这就是对象的实例化。

提示：通过调用构造函数创建了对象的一个实例以后，编译器保证对象的每一个字段都已经初始化了，也就是说所有的数字都初始化为 0，所有的指针都初始化为 nil，所有的字符串都初始化为空字符串。

2.14.2 析构

当对象使用完毕以后，应当及时地释放它，通常会调用对象的 Free()方法来删除对象。Free()方法首先检查对象实例是否为 nil，如果不是的话，就会调用对象的析构函数 Destroy()。析构和构造的作用恰恰相反，它释放先前分配的内存，并做一些清除工作，以保证从内存中彻底清除对象。

与调用 Create()不同的是，释放对象必须调用 Free()，而不是 Destroy()，因为 Free()会检查对象的实例是否为 nil，这样就会避免对一个空对象进行操作。

注意：对于动态创建的对象，您必须自己调用析构来删除它，但如果一个对象被其他对象所拥有，那么在拥有对象释放的时候，会自动释放该对象。比如，直接放在窗体上的组件对象，在窗体创建的时候，会自动创建它所拥有的对象，在释放窗体的时候，会自动释放它所拥有的对象。

2.15 方 法

方法专指对象中的过程和函数，它使对象具有某种行为而不仅仅是数据。对于一个对象来说，重要的两个方法是构造和析构。刚才已经讨论过这两个函数了，用户可以在对象中创建一些自定义的方法来实现不同的任务。

要创建一个方法分为两步，首先要在声明对象时声明一个方法，然后定义这个方法。下面的代码演示了怎样声明和定义方法：

```
type
    TMyObject: class
        Height: integer;
        Procedure GetHeight;
End;
…
procedure TMyObject.GetHeight;
begin
    Height := 100;
end;
```

提示：对象中的方法可以直接访问对象中的字段，如上面的示例中，GetHeight 访问了字段 Height。

2.15.1　重载

重载方法体现了 Object Pascal **的** OOP **概念之一，即多态性。通过重载，用户可以使用同一个方法在不同的派生类之间表现出不同的行为。要使方法能够被重载，只有在派生类的声明中用** override **来标识方法。例如：**

```
TMyForm = class(TFrom)
        Procedure Create: override;
        …
End;
```

与普通的过程和函数一样，方法也可以重载，这样，一个类中就有多个名称相同但参数不同的方法。要重载一个方法，必须用 overload **指示字，但第一个方法不必用这个指示字。下面的代码声明了一个类及** 3 **个重载的方法：**

```
Type
        TSomeClass = class
                Procedure AMethod(I: integer);overload;
                Procedure AMethod(S: string);overload;
                Procedure AMethod(D: Double);overload;
        End;
```

2.15.2　特性

特性可以被看作特殊的字段，使用户能够修改对象的数据和执行代码。对于组件来说，公开的特性会显示在 Object Inspector **中。下面的示例中声明了一个类和一个特性：**

```
TMyObject = class
Private
        SomeVar: integer;
        Procedure SetValue(AValue: Integer);
Public
        Property Value: Integer read SomeValue write SetSomeValue;
End;

Procedure TMyObject.SetSomeValue(AValue: Integer);
Begin
        If SomeValue <> AValue then
                SomeValue := AValue;
End;
```

TMyObject **是一个对象，包含下列内容：一个整型字段** SomeValue**、一个方法** SetSomeValue **和一个特性** Value**。方法** SetSomeValue **的惟一功能就是设置** SomeValue **字段的值。**Value **本身并不包含数据，只是提供了访问字段** SomeValue **特性的途径。当程序需要知道** Value **特性的值时，它就读** SomeValue **字段的值。特性的意义有两个，首先，通过**

一个简单的变量就使外部代码可以访问对象的数据，而不需要知道对象的实现细节；其次，在派生类中可以重载诸如 SetSomeValue 的方法，以实现多态性。

2.15.3 代码可见性

Object Pascal 允许对字段和方法设置不同的可见性，这需要用到几个指示字，例如 Protected、private、public、published 和 automated。关于这些指示字的用法请参考下列代码：

```
TSomeObject＝class
Private
        APrivateVariable：Integer；
        AnotherPrivate Variable：Booleam；
Proteced
        Procedure AProtectedProcedure；
        Function ProtectMe：Byte；
Public
        ConstructorAPublicContructor；
        DestructorAPublicKiller；
Published
        Property AProperty read APrivate Variable write APrivate Variable；
End；
```

在每个指示字下面，用户可以根据需要声明任意多的字段和方法。请注意上述代码的书写风格。这些指示字的含义是：

- private 私有的成员只能在对象实现的单元中访问。使用这个指示字的目的是隐藏实现的细节，防止敏感的成员被意外修改。
- proteced 保护的成员可以被派生对象访问。既隐藏了实现的细节，又对派生对象提供了最大程度的灵活性。
- public 公共的成员可以在程序的任何一个地方被访问。对象的构造和析构应当是公共的。
- published 公开的成员具有 RTTI（运行期类型信息），使用户的应用程序的其他部分能得到在对象的 published 部分的信息。Object Inspector 就是通过 RTTI 来建立特性列表的。
- automated 这个指示字其实已不用了，保留这个指示字的目的是为了与 Delphi 2 的代码兼容。

下面的代码实际上是前面提到的 TMyObject，只不过这里加上了一些指示字：

```
TMyObject＝class
Private
        SomeValue：Integer；
        Procedure SetSome Value（AValue：Integer）；
Published
```

```
        Property Value：Integer read Some Value write SetSome Value；
End；

Procedure TMyObject.SetSome Value(AValue: Integer);
Begin
        If Some Value <>AValue then
            Some Value:=AValue;
End;
```

现在，只要是对象的用户就无法直接修改 SomeValue **字段的值，只能通过** Value **特性间接地修改对象的数据。**

2.16　接　口

对于 Object Pascal **语言来说，最有意义的改进就是从** Delphi 3 **开始支持** Interface**（接口）。接口中定义了一些过程和函数，用于与一个对象交互。定义接口实际上就是实现接口。一个类可以实现几个接口，这样，一个客户可以从几个方面来控制对象。**

之所以取名叫接口，是因为它定义了对象与客户之间通讯的界面。接口类似于 C++**中的纯虚类。凡是支持接口的类必须实现接口中的过程和函数。**

本章将介绍接口的语法。

- **声明接口**

就像所有的类都是从 TObject **继承下来的一样，所有的接口也都是起源于一个叫** IUnknown **的接口。在** System **单元中，**IUnknown **接口是这样声明的，如下所示：**

```
Type
    IUnknown=interface
        ['{00000000-0000-0000-C000-000000000046}']
        function Query Interface(const IID;TGUID;out Obj):Integer;stdcall;
        function_AddRef:Integer;stdcall;
        function_Release:Integer;stdcall;
    end;
```

正如读者所看到的那样，声明一个接口的语法非常类似于声明一个类。主要的区别在于：接口可以附带一个 GUID**（全局惟一标识符）。**IUnknown **接口声明循环** Microsoft **的** COM**（构件对象模型）规范。**

如果用户已经掌握了怎样创建一个类，要声明一个自定义的接口也是非常简单的。下面的代码声明了一个叫 IFoo **的接口，其中有一个方法叫** FIO**：**

```
type
    IFoo=interface
        ['{2137BF60-AA33-11D0-A9BF-9A4537A42701}']
        function F1:Integer;
```

```
    end;
```

提示：在 Delphi 的 IDE 中按 Shift＋Ctrl＋G 键，可以自动创建一个新的 GUID。

下面的代码声明了另一个叫 IBar 的接口，它是从 IFoo 接口继承而来的，如下所示：

```
type
    IBar=interface(IFoo)
        ['{2137BF61-AA33-11D0-A9BF-9A4537A42701}']
        function F2:integer;
    end;
```

- **实现接口**

下面的代码演示了用一个叫 TFooBar 的类实现 IFoo 和 IBar 接口，如下所示：

```
type
    TYooBar=class(TIntefacedObject,IFoo,IBar)
        Function F1:Integer;
        Function F2:Integer;
    End;

Function TFooBar.F1:Integer;
Begin
    Result:=0;
End;

Function TFooBar.F2:Integer;
Begin
    Result:=0;
End;
```

注意：一个类可以实现多个接口，只要在声明这个类时依次列出要实现的接口即可。编译器通过名称把接口中的方法与实现接口的类中的方法对应起来。如果一个类只是声明要实现某个接口，但并没有具体实现这个接口的方法，编译时将会出错。

如果一个类要实现多个接口，而这些接口中包含同名的方法，必须把同名的方法另取一个别名。请看下面的程序示例：

```
type
    IFoo=interface
        ['{2137BF60-AA33-11D0-A9BF-9A4537A42701}']
        function F1:Integer;
      end;

    IBar=interface
```

```
        ['{2137BF61-AA33-11D0-A9BF-9A4537A42701}']
        function F1:Integer;
    end;

    TFooBar=class(TIterfacedObject,IFoo,IBar)
 //aliased methods
        function IFoo.F1=FooF1;
        function IBar.F1=BarF1;
        //interface methods
        function FooF1:Integer;
        function BarF1:Integer;
    end;
function TFooBar.FooF1:Integer;
begin
    Result:=0;
End;

Function TFooBar:BarF1:Integer;
Begin
    Result:=0;
End;
```

- Implements 指示字

Implements 指示字是 Delphi 4 新增加的，它的作用是委托另一个类或接口来实现接口的某个方法。这个技术有时候也称为委托实现。关于 Implements 指示字的用法可参见下面的代码，如下所示：

```
type
    TSomeClass=class(TInterfacedObject,IFoo)
        //stuff
        function GetFoo:TFoo;
        property Foo:TFoo read GetFoo imlements IFoo;
        //stuff
    end;
```

用了 Implements 指示字后，编译器遇到实现 IFoo 接口的方法时，就去查找 Foo 特性。这个特性的类型必须是一个类并且包含 IFoo 接口中的方法，或者是一个接口，其类型是 IFoo 或 IFoo 派生接口。Implements 指示字后面可以列出几个接口，彼此用逗号隔开。

Implements 指示字对开发有两个关键的好处：首先，它允许以无冲突的方式进行接口聚合。Aggregation（聚合）是 COM 中的概念，其作用是把多个类合在一起共同完成一个任务。其次，除非现在确实需要，它能够延后占用由于实现接口所需要的资源。例如，假设实现一个接口需要分配一个 1MB 的位图，但这个接口很少被用到，因此，为了避免浪费

资源，用户可能平时并不想实现这个接口，用了 Implements 指示字后，用户可以只在特性被访问的时候才创建一个类来实现接口。

- 使用接口

关于在应用程序中使用接口类型的变量要遵循一些规则，其中最重要的规则是，接口属于生存期管理的类型。这意味着接口总是被初始化为 nil，具有引用计数机制。当获得一个接口时，引用计数会自动加 1。当接口的变量超出其作用域时，它们会自动释放或被赋值为 nil。下面的代码演示了一个接口类型的变量的生存期管理：

```
var
    I：ISomeInterface；
Begin
    // I 被初始化为 nil
    I := FunctionReturningAnInterface;// 引用计数加 1
I.SomeFunc;
// 引用计数减 1 。 如果减到 0，I 就被释放
end；
```

另一个独特的规则是，一个接口的变量与实现这个接口的类是赋值相容的。例如，下面的代码是合法的：

```
procedure Test（FB：TFooBar）
var
    F：IFoo；
Begin
    F := FB;  //合法，因为 FB 支持 IFoo
    …
```

最后，类型强制转换运算符 As 可以把一个接口类型的变量转换为另一种接口。请看下面的代码：

```
var
    FB：TFooBar；
    F：IFoo；
    B：IBar；
Begin
    FB := TFooBar.Create
    F := FB；                  // 合法，因为 FB 支持 IFoo
    B := F as IBar；           // 把 F 转换为 IBar
    …
```

如果请求的接口不被支持，就会触发异常。

2.17　异 常 处 理

结构化的异常处理是一种处理错误的手段，使应用程序能够从致命错误中很好地恢复。在 Delphi 1 中，异常处理是由 Object Pascal 语言实现的。从 Delphi 2 开始，异常处理成为 Win32 API 的一部分。用 Object Pascal 语言来处理异常比较简单，因为异常中包含了错误的位置和本质信息，这使得异常的使用就好像一个普通的类一样。

Delphi 包含了一些预定义的公共程序错误异常，例如内存不足、被零除、数字上溢和下溢以及文件输入输出错误。用户可以定义自己的异常类，以适应应用程序的需要。

下面的程序示范了怎样在文件输入输出过程中进行异常处理，如下所示：

```
Program FileIO;

Uses Classws, Dialogs;

{2APPTYPE CONSOLE}

var
        F:TextFile;
        S:string;
Begin
        AssignFile(F, ? OO.TXT*)
        Try
                Reset(F);
                Try
                    ReadLn(F,S);
                Finally
                    CloseFile(F);
                End;
        Expect
      On EInOutError do
                    ShowMessage('Error Accessing File!');
        End;
End.
```

这里，不管是不是发生了异常，内层的 Try...Finally 块用于确保文件总是能关闭。这段代码实际上意味着“ Hey，程序，请执行 Try 与 Finally 之间的代码。如果执行完毕或出现异常，就执行 Finally 与 End 之间的代码，如果确实有异常发生了，就跳到外层的异常处理块 ”。这样，即使出现异常，文件总是能关闭，并且异常总是能得到处理。

注意：因为 Finally 后面的语句不能以发生异常为前提，所以，在 Try...Finally 块中，Finally 后面的语句不管有没有异常总是执行的。另外，由于 Finally 后面的语

句并没有处理异常，因此，异常被传递到下一层的异常处理块中。

外层的 Try...Except 块用于处理程序中发生的异常。当文件被关闭后，Except 块显示一个信息，告诉用户发生了 I/O 错误。

这种异常处理机制比传统的错误处理方式优越，它使得错误检测代码从错误纠正代码中分离出来。这是一件好事情，它会使程序更可读，使用户能专注于其他代码部分。

为什么在 Try…Finally 块中不能捕捉特定种类的异常呢？当代码中使用 Try...Finally 块的时候，意味着程序并不关心是否会发生异常，而只是想最终总是能执行某项任务。Finally 块最适合于释放前分配的资源（例如文件或 Windows 资源），因为它总是能执行的，即使发生了错误。不过，很多情况下，用户可以需要针对特定的异常作特定的处理，这时候就要用到 Try...Except 块来捕捉特定的异常。下面的程序就是这样的示例：

```
Program HandleIt;

{2APPTYPE CONSOLE}

var
    R1,R2:Double;
Begin
    While True do Try
        Write (‘Enter a read number :’);
        ReadLn (R1);
        Write (‘Enter another read number :’);
        ReadLn (R2);
        WriteIn (‘I will now divide the first number by the second …’);
        WriteIn (‘The answer is :’, (R1 / R2):5:2);
    Except
On EZeroDivide do
        WriteIn (‘You cannot divide by zero!’);
        On EInOutError do
        WriteIn (‘That is not a valid number !’);
    End;
End.
```

尽管在 Try...Except 块中可以捕捉特定的异常，用户也可以从 Try...Except...Else 结构中捕捉其他异常。请看下面的代码：

```
try
    Statements
Except
    On ESomeException do Something；
Else
    {进行一些默认的异常处理}
end；
```

注意：当使用 Try...Except...Else **结构的时候，用户应当明白** Else **部分会捕捉所有的异常，包括用户没有预料到的异常，例如内存不足或其他运行期库异常。因此，使用** Else **部分时要分外小心，能不用则不用。当用户陷入不合格的异常处理句柄中时，应当一直触发这个异常。**

因为在 Except **部分没有区分特定的异常，其实，下面的代码也能够达到类似于** Try...Except...Else **结构的效果。**

```
Try
    语句
except
    处理异常        //效果与 try..except..else 结构几乎相同
end；
```

2.17.1　异常类

异常是一种特殊的对象。这些对象在异常发生的时候会被实例化，在异常被处理后会自动删除。异常对象的基叫 Exception**，它是这样声明的：**

```
type
    Exception=class (TObject)
    Private
        Fmessage: string;
    Public
        Constructor Create (const Msg:string):
        Constructor CreateFmt (const Msg:string；const Args:array of const);
        Constructor CreateRes(Ident:Integer);
        Constructor CreateResFmt(Ident:Integer;const Args:array of const);
        Constructor CreateHelp(const Msg:string;AHelpContext:Integer);
        Constructor CreateFmtHelp(const Msg:string;const Args:array of const;
        AhelpContext:Integer);
        Constructor CreateResHelp(Ident:Integer;AHelpContext:Integer);
        Constructor CreateResFmtHelp(Ident:Integer;const Args:array of const;
        AHelpContext:Integer);
        Property HelpContest:Integer read FHelpContext write FHelpContext;
        Property Message:string read FMessage write FMessage;
    End;
```

Exception **对象中很重要的是** Message **特性，它是一个字符串。**Message **特性提供了有关异常的信息或解释。由** Message **特性提供的信息取决于异常的类型。**

注意：如果要定义自己的异常，务必要使它继承一个已有的异常对象，例如 Exception **或它的派生类,这是为了使通用的异常处理句柄能够捕捉用户自定义的异常。**

当用户在 Except 块中处理一个特定的异常时，可能会捕捉到该异常的派生异常。例如 EMathError 是所有与数学有关的异常。请看下面的代码：

```
try
    Statements
Except
    On EMathError do        // 将捕捉 EMathError 及其派生异常
        HandleException
End；
```

凡是最终都没有处理的异常将被传递给 Delphi 运行期库中的默认句柄，并在此处得到处理。这个默认的句柄将打开一个消息框，告诉用户一个异常发生了。顺便说一下，第 3 章“ Delphi 应用程序框架和设计 ”将给出程序实例来演示怎样重载默认的异常处理句柄。

处理异常的时候，用户可能需要访问异常对象的实例，以便获得更多的有关异常的信息。要访问异常对象的实例有两种方式：一是在 on ESomeException 结构中使用可选的标识符，二是使用 ExceptObject()函数。

可以在 on ESomeException 结构中插入一个可选的标识符，使这个标识符映射当前触发的异常对象的实例。有关的语法可以参考下面的代码：

```
try
    Something
Except
    On E:ESomeException do
        ShowMessage(E.Message);
End；
```

在上述代码中，标识符“ E ”代表当前对象的实例。它的类型和它触发的异常类型一样。

也可以使用 ExceptObject()函数返回当前异常对象的实例。不过，ExceptObject()函数的返回类型是 TObject，必须把它强制转换为需要的异常对象。下面的代码演示了 ExceptObject() 函数的用法：

```
try
    Something
Except
    On ESomeException do
        ShowMessage (ESomeException(ExceptObject).Message);
End;
```

如果当前没有异常，ExceptObject()函数将返回 nil。

触发一个异常的文法类似于创建一个对象实例。例如，要触发一个叫 EBadStuff 的用户自定义的异常，可以参考下面的代码：

```
Raise EBadStuff.Create('Some bad stuff happened.');
```

2.17.2　触发异常

当要针对 Try...Except **块中的一个语句进行特殊的处理并且还要使异常能够传递给外层的默认处理句柄时，需要重新触发异常。下面的程序演示了怎样触发异常：**

```
try                    //这是外层
    { statements }
    { statements }
    { statements }
    try                //这是内层
        { 某些需要特殊处理的语句 }
    except
        on ESomeException do
        begin
            { 对某些语句进行特殊处理 }
            raise；// 再次触发异常
        end；
    end；
except
    // 这里用于进行默认的处理
    on ESomeException do Something；
end；
```

2.18　运行期类型信息

RTTI（运行期类型信息）是一种语言特征，它使应用程序能够在运行期获得一个对象的信息。RTTI 是使 Delphi 的组件能够融合到 IDE 中的关键。

由于对象都是从 TObject 继承下来的，因此，对象都包含一个指向它们的 RTTI 的指针，以及几个内建的方法，调用这些方法能够获取 RTTI 中的有用信息。表 2.6 列出了 TObject 中的一些方法，这些方法用于从 RTTI 中获取信息。

表 2.6　TObject 中的一些方法

方法	返回类型	返回值
ClassName()	String	对象的类名
ClassType()	Tclass	对象的类型
InheritsFrom()	Bollean	判断对象是否继承于一个指定的类
ClassParent()	Tclass	对象的祖先类型
InstanceSize()	Word	对象实例的长度（字节）
ClassInfo()	Pointer	指向 RTTI 的指针

Object Pascal 提供了两种运算符：Is 和 As，用于通过 RTTI 对对象进行比较和类型强制转换。

用运算符 As 进行类型强制转换比较安全，它能够把一个对象强制转换为相容的类型。如果转换不成功，会触发一个异常。假设一个过程要能够传递任何类型的对象，这个过程应当这样声明，如下所示：

```
Procedure Foo（AnObject：TObject）；
```

如果要在这个过程中对 AnObject 对象进行处理，用户可以把它转换为一种需要的类型。假设实际传递的是 TEdit 组件，并且要修改编辑框中的文本，可以参考下面的代码，如下所示：

```
（Foo as TEdit）.Text:=‘Hello World’;
```

可以用布尔比较运算符 Is 来检查两个对象是否相容。Is 运算符可以把一个未知的对象与一个已知的类型或实例进行比较，以判断这个未知对象的类型。例如，可以在强制类型转换之前检查 AnObject 是否与 TEdit 相容，如下所示：

```
If（Foo is TEdit）then
    TEdit（Foo）.Text:=‘Hello World’；
```

这里并没有使用 As 运算符来进行类型强制转换。这是因为使用 RTTI 多少有点开销，而且前面已经判断了 Foo 就是 TEdit，所以可以通过在程序第 2 行进行指针转换来进行类型强制转换。

2.19 本 章 小 结

本章的内容相当丰富，介绍了 Object Pascal 语言的基本语法，包括变量、运算符、函数、过程、类型、构造和风格。通过本章的学习，想必读者已经清晰地领会了 OOP、对象、特性、方法、TObject、接口、异常处理和 RTTI 的概念。

现在，读者对面向对象的 Object Pascal 语言已经有了大致的了解，下一步可以转到更高级的领域，例如 Win32 API 和 VCL 中。

第 3 章　Delphi 应用程序框架和设计

本章将介绍 Delphi 的项目管理和体系结构。这里读者将了解到如何正确地使用 Form，以及如何操纵它们的行为和可视化特征。本章将讨论 Delphi 应用程序的启动、初始化过程、Form 重用、继承以及增强的用户界面等技术。

3.1　Delphi 环境和项目的体系结构

要正确建立和管理 Delphi 的项目，至少需要掌握两个要素：一是要创建项目的开发环境的细节，二是构成 Delphi 项目的体系结构。本章将讲述 Delphi IDE 的特点，以帮助用户更有效地管理项目，另外还有解释 Delphi 应用程序的体系结构，这不但使用户能够充分发挥开发环境的特点，而且使用户能够正确使用固有的体系结构。

在开始学习本章之前，用户应当尽可能熟练掌握 Delphi 的开发环境，用户应当把 Delphi 的菜单和对话框都打开一遍，如果遇到不太理解的选项、设置或动作，应当打开 Delphi 的在线帮助，从中找到答案。

> 提示：Delphi 提供的在线帮助无疑是最有价值和最快的寻求帮助的途径，学会在数千屏的帮助主题中浏览是非常重要的。Delphi 的在线帮助包含了如何使用 Delphi、Win32 API 的细节以及复杂的 Win32 结构等众多的内容。

3.2　构成 Delphi 项目的文件

一个 Delphi 项目由若干个相关的文件构成。这些文件中有些是在设计期被创建的，有些是在编译期形成的。要有效地管理 Delphi 项目，用户应当知道每一个文件的用途。Delphi 的文档和帮助介绍了项目中的文件。现在就来回顾一下这些文档。

3.2.1　项目文件

项目文件是在设计期被创建的，它的扩展名是 dpr，这个文件也就是主程序源文件。项目文件是主 Form 以及其他 Form 实例化的地方。一般情况下除非要在程序启动之前进行一些初始化工作，用户不需要编辑项目文件。下面的代码就是一个典型的项目文件，如下所示：

```
program Project1;

uses
```

```
        Forms,
        Unit1 in 'Unit1.pas' {Form1};

{3R *.RES}
begin
        Application.Initialize;
        Application.CreateForm(TForm1, Form1);
        Application.Run;
end.
```

Pascal 程序员可以把项目文件作为一个标准的 Pascal 源文件。项目文件的 Uses 子句列出了项目中的所有单元。“{3R *.RES}”这行代码用于引用项目的资源文件。它告诉编译器去链接一个资源文件，这个资源文件的名称与项目文件相同，但扩展名是 RES。项目的资源文件中包含了图标和版本信息。

Begin…End 之间的语句是应用程序要执行的主代码。在上面这个示例中，主要是创建主 Form 即 Form1。当 Application.Run()这条语句执行时，Form1 就作为主 Form 显示出来了。

3.2.2 单元文件

单元文件就是 Pascal 源文件，它的扩展名是 PAS。有 3 种类型的单元文件：Form/数据模块的单元文件、组件的单元文件和通用的单元文件。

- Form /数据模块单元文件：由 Delphi 自动生成。每个 Form 或者数据模块都有一个对应的单元文件，两个 Form 不可能共用一个单元文件。
- 组件的单元文件：它在创建一个新的组件的时候生成。
- 通用的单元文件：由程序员创建，用于声明类型、变量、过程和类等。

在第 4 章中将详细介绍这些单元文件的细节。

3.2.3 Form 文件

Form 文件存储了 Form 的二进制信息。当创建一个 Form 的时候，Delphi 将同时创建一个 Form 文件(*.dfm)一个和 Pascal 单元文件（*.pas）。如果用户打开一个 Form 的单元文件，将会出现以下的代码：

```
{3R *.DFM}
```

这一行代码告诉编译器去链接一个 Form 文件到项目中，这个 Form 文件的名称和单元文件的名称相同，但扩展名为.dfm。

一般不需要直接编辑 Form 文件，用户可以用代码编辑器打开一个 Form 文件，这样就会看到文本形式的 Form 文件并且能够编辑它。要以文本的形式打开一个 Form 文件，可以执行 File | Open 命令，或者右击 Form 设计器，在弹出的快捷菜单中单击 View As Text 命令，图 3.1 后的代码就是一个 Form 的文本化文件，而 Form 本身的显示如图 3.1 所示。

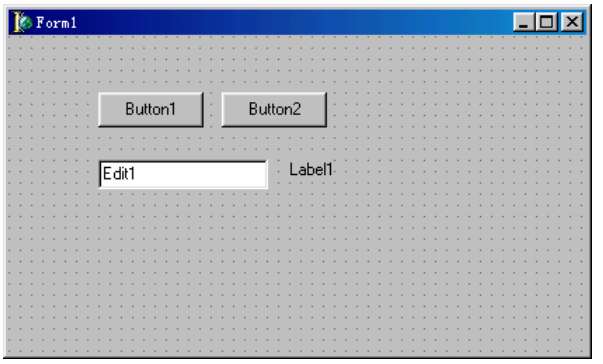

图 3.1　一个 Form 实例

```
object Form1: TForm1
    Left = 192
    Top = 107
    Width = 420
    Height = 253
    Caption = 'Form1'
    Color = clBtnFace
    Font.Charset = DEFAULT_CHARSET
    Font.Color = clWindowText
    Font.Height = -11
    Font.Name = 'MS Sans Serif'
    Font.Style = []
    OldCreateOrder = False
    PixelsPerInch = 96
    TextHeight = 13
    object Label1: TLabel
        Left = 200
        Top = 88
        Width = 32
        Height = 13
        Caption = 'Label1'
    end
    object Button1: TButton
        Left = 64
        Top = 40
        Width = 75
        Height = 25
        Caption = 'Button1'
        TabOrder = 0
    end
```

```
        object Button2: TButton
            Left = 152
            Top = 40
            Width = 75
            Height = 25
            Caption = 'Button2'
            TabOrder = 1
        end
        object Edit1: TEdit
            Left = 64
            Top = 88
            Width = 121
            Height = 21
            TabOrder = 2
            Text = 'Edit1'
        end
end
```

用户可以通过编辑这个文件来修改 Form **的设置。例如，把其中的** Button1 **的属性改写为** TLable**，**Button1 **将由按钮更改为标签。**

注意：通过文本形式修改 Form **有可能导致错误。例如，**TButton **原来有一个** TabOrder **特性，如果把** TButton **改为** TLabel**，由于** TLabel **没有** TabOrder **特性，这就导致了错误。不过，用户用不着手工去更正它，因为当保存这个** Form **时，**Delphi **会自动更正它。**

3.2.4 资源文件

资源文件中包含了二进制数据，也称为资源。这些资源将链接到应用程序的可执行文件中，资源文件是 Delphi **自动创建的，其中包含应用程序的图标、应用程序的版本信息以及其他信息。用户可以创建一个单独的资源文件，然后把它链接到项目中。**

注意：不要编辑 Delphi **自动生成的资源文件。如果那样做的话，下次编译项目时所作的修改可能丢失。**

3.2.5 项目选项和桌面设置文件

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

桌面选项文件（*.dsk**）存储了** Tools | Environment Options **和** Editor Options **命令所设置的桌面选项。桌面选项不同与项目选项，项目选项是与具体项目有关的，而桌面选项适用于** Delphi **环境。**

注意：错误的*.dof **和***.dsk **文件有可能导致不可预测的错误。如果真的出现这种**

错误，应当把*.dof 和*.dsk 文件都删除掉，然后保存项目或者退出 Delphi，这时候，这两个文件将重新生成，项目和 IDE 会恢复为默认设置。

3.2.6　包文件

包类似于动态链接库，包的代码可以被几个应用程序共享。不过，包是 Delphi 特有的，用于共享组件、类、数据和代码。把组件放到包中，而不是直接链接到应用程序中，可以大大地减小应用程序的长度。包的资源文件扩展名是.dpk（Delphi Package）。

3.3　Delphi 项目管理

通过良好的组织和代码重用，可以使开发过程得以优化。下面的章节将介绍一些有关项目管理的建议。

3.3.1　一个项目一个目录

建议用户最好把一个项目中的文件与另一个项目中的文件分开，这样可以防止一个项目的文件覆盖另一个项目的文件。

最好事先规划好一个项目中文件的命名约定，这些约定将在第 4 章中详细介绍。

3.3.2　代码中被共享的单元

如果一些程序需要被其他应用程序共享，最好把它们放到一个单独的单元中。通常的做法是：在磁盘中建立一个目录，并把需要共享的单元放到这个目录中。以后，如果一个项目要共享其中的某个单元，只要把那个单元的名称加到 Uses 子句中即可。

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

注意：通过使用项目管理器，可以把位于其他目录中的单元加到当前项目中，这种情况下，Delphi 会自动添加有关路径到 Search Path 文本框中。

一个单元可以专门用来声明全局标识符。一个项目往往由若干个单元文件组成，包括 Form 的单元文件、组件的单元文件和通用的单元文件。如果需要声明一个所有单元都可以访问的变量，可以创建一个专门用来声明全局标识符的单元。

首先，执行 File | New 命令，然后双击 New Items 中的 Unit 按钮来创建一个单元，给这个单元命名并保存，名称最好能够体现这个单元是用来声明全局标识符的，例如 Globals 或者 ProjectGlob 等。在这个单元的 Interface 部分声明变量、类型等，为了使这些变量和类型能够被所有的单元访问，需要把该单元名称加到需要访问的单元的 Uses 子句中去。

Delphi 在每一个 Form 的单元文件中会声明 Form 的一个全局实例变量，通过该变量可以访问到 Form 上的组件，只要把该 Form 的单元加到 Uses 子句中就可以了，例如：

```
unit Unit1;

interface
```

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

type
    TForm1 = class(TForm)
    private
        { Private declarations }
    public
        { Public declarations }
    end;

var
    Form1: TForm1;

implementation

uses Unit2;

{3R *.DFM}

end.
```

这样，在 Unit1 的 Implementation 部分就可以引用 Form2 中的组件了。

3.3.3 多项目管理

用工程组在 Delphi 的一个实例中管理相关工程是很有效的。不过，创建一个 Delphi 应用程序并不一定必须要创建工程组。例如，打开的工程不属于任何工程组，Delphi 会生成一个新的工程组（缺省名为 ProjectGroup1）。用户可以保存这个新的工程组，但这并不是必要的。

一个产品往往由几个项目组成，这些项目之间可能相互依赖，例如要被其他应用程序调用的动态库 DLL，DLL 本身也可能是一个项目。

Delphi 中引入了一个非常有用的工具——项目管理器，来对多项目进行管理，在第 1 章“Delphi 6.0 入门”一章中已经介绍过了这个工具，它允许把若干个项目集中在一起，组成一个项目组。项目管理器是用来管理项目组文件的惟一途径，可以通过 View | Project Manager 命令打开这个管理器。

如图 3.2 中所示，项目组 ProjectGroup1 中包含了两个项目：Project1.exe 和 Project2.exe。在一个项目组中，每个项目的文件最好放在一个单独的目录中，凡是需要共享的单元、Form 最好放在一个公共的目录中，以便被其他所有的项目共享。例如图 3.2 下面是一个典型的目录结构：

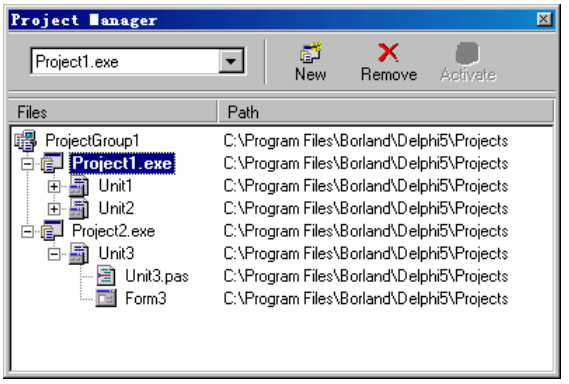

图 3.2　项目管理器

\MyProjectGroup

\MyProjectGroup\Project1

\MyProjectGroup\Project2

\MyProjectGroup\CommonFiles

从上面的目录结构可以看出，有两个专门的目录各自存放一个项目，分别是 Project1 和 Project2，这两个项目可能需要共享一些单元和 Form，它们放在 CommonFiles 目录中。

把一个项目组规划好是至关重要的，尤其是在一个团队的开发环境中。

3.4　项目选项设置

一个应用程序具有很多选项设置，Delphi 提供了一个 Project Options 对话框，在这里可以设置项目的各种选项。这非常重要，建议一定要认真读下面的文字。

在工程管理器中，右击需要的工程，从快捷菜单中执行 Options 命令可以打开 Project Options 对话框，也可以用 Project | Option 命令打开这个对话框。在这个对话框中可以修改工程的多种特征。这是一个标签式对话框，包括以下选项卡：Forms（窗体）、Application（应用程序）、Complier（编译器）、Linker（连接器）、Directories/Conditionals（目录/条件）、Version Information（版本信息）和 Packages（包）等。下面将逐个讨论。

Project Options 对话框左下角有一个 Default 检查框，用它可以把当前的设置保存为缺省设置，此后所有创建的工程都可以使用这个缺省设置。

- Forms 选项卡

对话框的 Forms 选项卡用来确定程序处理窗体的方式。这一选项卡的界面如图 3.3 所示。

在这个选项卡中，可以指定应用程序的主窗体。左边的列表框用来指定自动生成的窗体，右边的列表框显示工程的可用窗体。列表中的窗体在装载程序时会自动创建。如果用户事先知道程序会用到某一个窗体，最好让它自动创建。其他的窗体不会由 Delphi 自动创建，必须在程序中自己来创建它们。对于那些不是每次程序运行都会用到的窗体，应该采取这种处理方式。使用手工创建的窗体可以减少启动应用程序花费的时间并且减少应用程

序占用的内存。要用到窗体时，再用程序显式创建。例如应用程序中有一个很少运行的 Setup 对话框，就不必让应用自动创建这个窗体，因为程序运行时不会每次都用到这个窗体，只是在需要改变 Setup 设置时才让它出现。

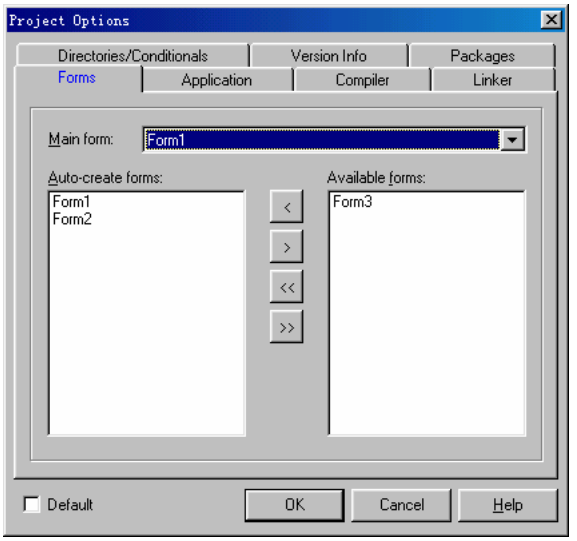

图 3.3 Project Options 对话框的 Forms 选项卡

- Application 选项卡

对话框中的 Application 选项卡用来设置应用程序的各种属性，如图 3.4 所示。

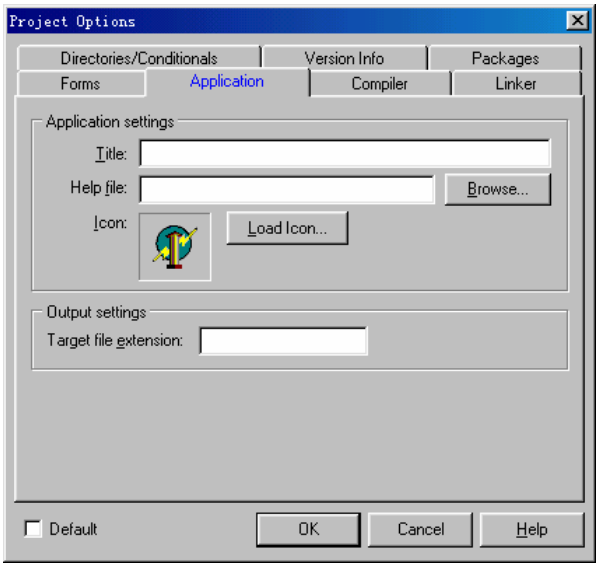

图 3.4 Project Options 对话框的 Application 选项卡

其中，Title 文本框用来命名应用程序的标题，它是运行时显示在 Windows 任务栏上的文字。Help File 文本框指定程序的帮助文件，Icon 选项用以指定程序图标。注意，应用程

序被编译之前，图标文件一直保存在资源文件中（即 RES 文件），如果这个资源文件被删除，应用程序将使用缺省图标，但是，一旦程序被编译，图标将被捆绑在可执行文件中，这也体现 Delphi 创建独立于 Windows 应用的能力。Output Settings 部分中的 Target file extension 文本框用以定义目标可执行文件的扩展名。可以为目标可执行文件设置扩展名。例如，如果是为了生成一个 ActiveX 应用程序，可以指定标准的文件扩展名为 ocx。一般而言如果不是创建 ActiveX 应用程序，就不必设置此项。

- Complier 选项卡

Project Options 对话框的 Compiler 选项卡用以设置程序编译器选项，如图 3.5 所示。

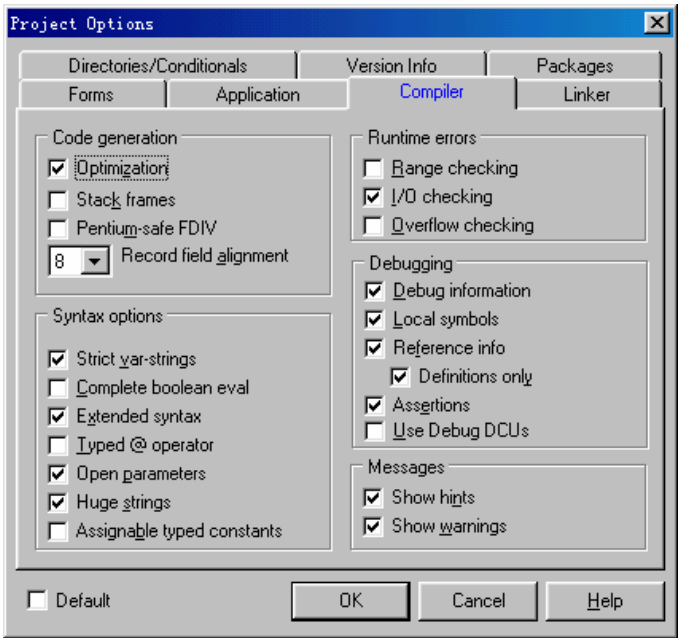

图 3.5　Project Options 对话框的 Compiler 选项卡

用户在这里可以设置当前工程的编译器选项。编译器的设置属于工程级而不属于环境级。也就是说，不同的工程可以有不同的编译器设置，而它们的全局设置可以相同。

在这个对话框中还可以设置编译器的优化器。如果选择 Optimization 复选框，则生成的 EXE 文件是压缩的，并且能够最有效地装载。

- Linker 选项卡

Project Options 对话框的 Linker 选项卡用来设置与内存和连接有关的选项，如图 3.6 所示。

这些选项包括指定用于调试的 Map file（映射文件）；Linker Output（连接器输出）也是在这里定义的(例如标准的 Delphi DCU 格式文件、二进制 C 或 C＋＋对象文件)；Memory Sizes 选项区用来定义编译后的可执行文件的栈的最大和最小尺寸以及堆的基地址；Generate console application 复选框告诉连接器在 EXE 程序文件里设置一个标记，标明它是控制台程序；EXE Description 文本框用以对 EXE 文件做一个描述性的说明，通常用于版权

信息；最后，如果用户使用远程调试，可以单击 Include remote debug symbols 复选框。

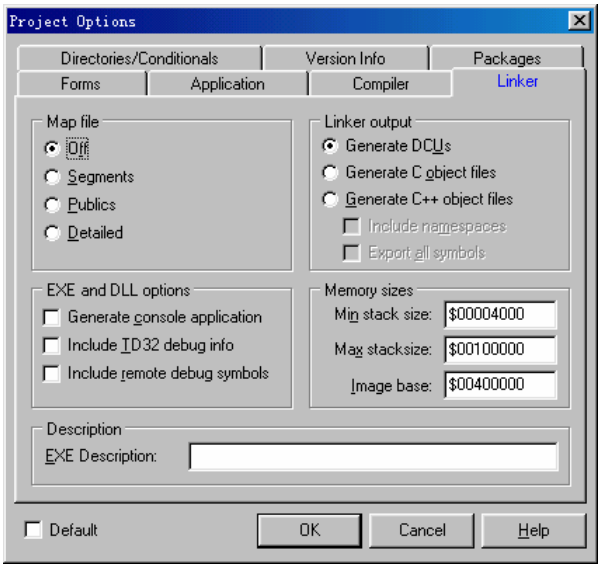

图 3.6 Project Options 对话框的 Linker 选项卡

- Dircetories/conditionals 选项卡

Project Options 的 Directories/Conditionals 选项卡用于指定路径和条件，如图 3.7 所示。

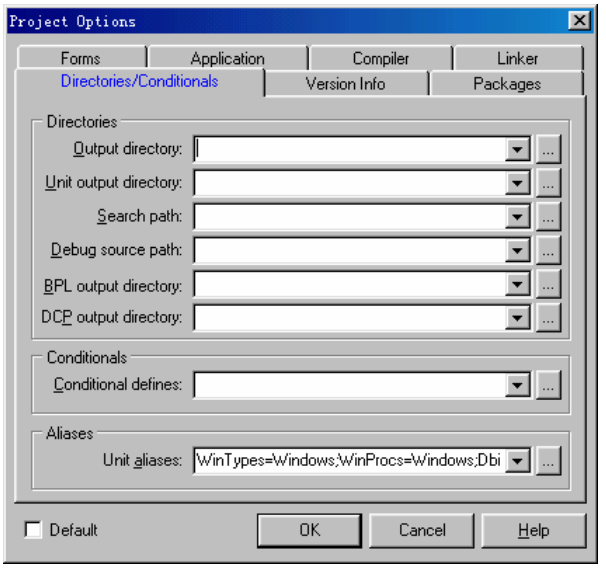

图 3.7 Project Options 对话框的 Directories/Conditionals 选项卡

在这个选项卡中，用户可以指定由系统产生的所有工程文件（编译过的单元与生成的可执行文件）的输出路径。在某些情况下，用户可能想把源文件与系统生成的文件放在不同的目录下。Unit output directory 选项为 DCU 文件指定了一个单独的路径。就个人经验，

笔者常常将工程源代码放在一个 Source 目录中，同级创建一个 Output 目录，其中用来放置 DCU 文件，也就是 Unit output directory 项。另外，Output directory 项一般设置为“ ..\ ”，它表示生成的 EXE 文件将保存在上一层目录中。

Search path 选项允许用户将单元放在与工程不同的目录中。例如，如果用户要将一些通用的库函数放在硬盘中的另一个位置，就可以在这个选项中指定库函数所在的目录。在这个选项中可以设置多条搜索路径（不超过 127 个字符）并用分号将它们隔开。Delphi 在项目编译过程中，将会到这些路径中进行搜索，查找必要的文件。

Debug source path 选项用于调试器，缺省状态下调试器将搜索由编译器指定的路径。用户也可以在这里设置一条路径来包含调试过程中用到的文件。

BPL output directory 和 DCP output directory 可以设置编译成包后的文件的输出路径。

用户可以用 Conditional defines 选项列出在条件编译指令中所引用的符号。Unit aliases 选项主要是为了向后兼容。用这个选项可以为那些可能改变了名字或合并了的单元定义一个别名。

- Version Information 选项卡

用户可以很方便地在 Version Info 选项卡中设置应用程序的版本信息，如图 3.8 所示。这些版本信息可以通过右击生成的 EXE 文件，在弹出的快捷菜单中执行“ 属性 ”命令来获得。

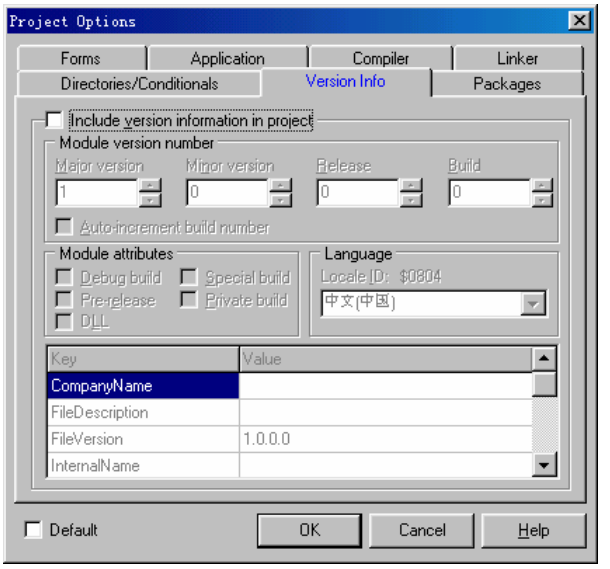

图 3.8　Project Options 对话框的 Version Info 选项卡

- Packages 选项卡

Project Options 对话框的最后一个选项卡是 Packages 选项卡，如图 3.9 所示。

使用包可以控制如何分发用户的应用程序。可以用包将一些通用的代码放在 DLL 中并与应用程序一同安装，这样就可以有多个应用可以访问这些代码。关于包的更详细的信息请查阅第 7 章相关内容。

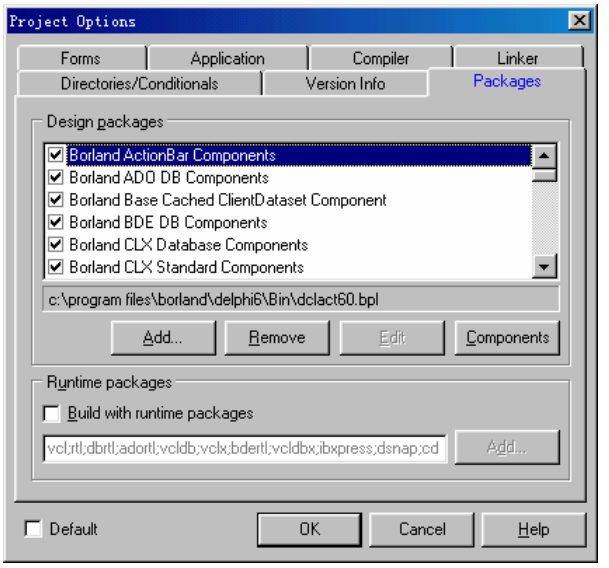

图 3.9 Project Options 对话框的 Packages 选项卡

注意：在 Compiler、Liner、Directories/Conditionals **这** 3 **个选项卡中的设置将保存在**.DOF **文件中。尽管可以直接修改这些文件，但是建议不要这样做。如果该文件被删除，**Delphi **将采用缺省设置。**

3.5 Delphi 项目的框架类

多数 Delphi **的应用程序至少需要一个** TForm **的实例，但** Delphi VCL **应用程序只能有一个** TApplication **实例和一个** TScreen **实例。**TForm、TApplication、TScreen **这** 3 **个类在** Delphi **中扮演着重要的角色。本节将重点介绍它们。**

3.5.1 TForm

TForm **是** Delphi **应用程序的焦点。多数情况下，整个应用程序是围绕着主** Form **转的。用户可以让** Delphi **自动为用户创建** Form **的实例，这样，用户就不必管什么时候创建、什么时候释放** Form **的实例了。当然，也可以在运行期动态地创建** Form **的实例，但这样的话，用户必须自己来释放这些实例。**

注意：Delphi **可以创建不需要任何** Form **的应用程序，例如** COM **服务器、服务等。**

Form **有两种类型，一种是** ModalForm**（有模式的** Form**），一种是无模式的。具体使用哪一种** Form**，取决于用户是否希望能够同时与这个** Form **和其他** Form **交互。**

为了帮助读者尽快学会灵活设计和使用窗体，在 3.8 **小节中将详细介绍窗体设计的一些基础知识和技巧。**

3.5.2　有模式的 Form

当一个有模式的 Form 打开以后，用户将无法与应用程序的其他部分交互，除非用户关闭了这个 Form。有模式的 Form 通常是对话框，事实上，大多数情况下应当使用有模式的 Form。要显示一个有模式的 Form，只要调用 ShowModal()函数就可以了。

ShowModal()函数的返回值将成为 Form 的 ModalResult 特性值。默认情况下，ModalResult 特性的值为 mrNone，相当于“0”，如果 ModalResult 特性被赋予其他的非零值，则 Form 将被关闭。用户可以在运行期对 Form 的 ModalResult 特性进行赋值：

```
ModalForm.ModalResult := 100;
ModalForm 将被关闭。
```

3.5.3　无模式的 Form

要打开一个无模式的 Form，可以调用 Show()方法。无模式的 Form 与有模式的 Form 的区别在于：用户可以在无模式的 Form 和其他 Form 之间切换。这样，用户就可以同时工作于一个应用程序的几个部分。

无模式的 Form 允许用户与应用程序的其他部分交互，这样，用户照常可以使用菜单的命令，或者创建其他实例，因此，用户需要考虑这些实例的创建和释放问题。

当用户通过 Close()方法关闭一个 Form 的实例的时候，除非关闭了主 Form，即应用程序，Form 并没有真正从内存中释放，它仍然存在于内存中。如果希望当用户关闭 Form 的时候就从内存中释放它，必须处理 Form 的 OnClose 事件，并且把 Action 参数设为 caFree，这样 VCL 就会释放这个 Form。

3.5.4　管理 Form 的图标和边框

TForm 有一个 BorderIcons 特性，它是一个集合，可以包含以下元素中的一个或多个：biSystemMenu、biMinimize、biMaximize 和 biHelp。分别对应系统菜单按钮、最小化按钮、最大化按钮和帮助按钮，但是，除非通过设置 BorderStyle 特性来隐藏它，Form 上总是有关闭按钮的。

BorderStyle 特性包含以下值中的一个，分别代表的含义见表 3.1。

表 3.1　BorderStyle 特性

BorderStyle 特性	描述
BsDialog	不能重设大小，只有关闭按钮
BsNone	没有边框，不能重设大小，没有任何按钮
BsSingle	不能重设大小，但可以有所有的按钮
BsSizeable	有边框，可以重设大小
BsSizeToolWin	可以重设大小，只有关闭按钮和标题栏
BsToolWindow	不能重设大小，只有关闭按钮和标题栏

注意：在设计期修改 BorderStyle 和 BorderIcons 特性并不能立即反映出来，这些变化要到运行期才能够看到效果。其实 TForm 的大部分特性都是这样的，这是因

为在设计期修改 Form 的外观没有多大意义，而且可能影响到 Form 的设计。例如，把 Form 的 Visible 特性设置为 False，如果 Form 就不再显示了，那么用户将无法操纵 Form 上的组件。

用户可以注意到了，上面的所有选项中（见表 3.1），没有一个选项能够创建一个没有标题但可以重设大小的 Form，要实现这种特殊的 Form，需要重载 Form 的 CreateParams() 方法，然后设置相应的风格。另外，当 Form 的标题栏不显示的时候，如何拖动 Form 呢？这就需要捕捉 Form 上的 WM_NCHITTEST 消息。下面的示例同时实现了上面的要求，即：没有标题栏，但可以重新设置 Form 的大小，并且能够拖动 Form 来移动位置。

```
unit Unit1;
interface

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

type
    TForm1 = class(TForm)
    private
        { Private declarations }
    public
        { Public declarations }
        procedure CreateParams(var Params: TCreateParams); override;
        procedure DragForm(var Msg: TWMNCHitTest); message WM_NCHITTEST;
    end;

var
    Form1: TForm1;

implementation

{3R *.DFM}

{ TForm1 }

procedure TForm1.CreateParams(var Params: TCreateParams);
begin
    // 设置 Form 的风格
    inherited CreateParams(Params);
    Params.Style := WS_THICKFRAME or WS_POPUP or WS_BORDER;
end;

procedure TForm1.DragForm(var Msg: TWMNCHitTest);
```

```
begin
    // 使鼠标可以拖动 Form 本身移动
    Msg.Result := HTCAPTION;
end;

end.
```

3.5.5　TApplication 类型

任何一个基于 Form 的 Delphi 程序都含有一个全局变量 Application，它是一个 TApplication 类型。TApplication 封装了一些特性和方法，使应用程序能够正确地在 Windows 环境下运行。这些方法中，有的用于建立窗口类定义，有的用于创建应用程序的主窗口、激活应用程序、处理消息、上下文敏感的帮助以及 VCL 的异常处理。

注意：只有基于 Form 的 Delphi 应用程序才有 Application 大小，而其他像控制台程序和服务程序就没有 Application 大小。

一般情况下，用户并不需要关心 TApplication 在背后到底做了些什么。不过，在一些情况下，用户还是需要弄清楚 TApplication 的详细情况。

由于 TApplication 并不在 Object Inspector 中出现，所以不能在设计期对它进行修改，但可以通过 Project | Options 命令，打开 Application 选项卡，设置一些关于 TApplication 的属性。大部分情况下，用户只能对 TApplication 的实例——Application 进行操纵，也就是说，用户只能在运行期设置 Application 的特性、方法和事件。

3.5.6　TApplication 的特性

TApplication 具有几个特性，用户可以在运行期访问它们。下面将讨论这些特性以及怎样通过它们改变 Application 的默认行为。这些特性在 Delphi 的在线帮助中也有叙述。

1．TApplication.ExeName 特性

ExeName 特性能够返回应用程序的路径和文件名。由于这个特性在运行期是只读的，所以用户无法修改它，只能读取它。

有时候需要得到应用程序的文件名、路径或者扩展名等单独的信息，用户可以使用下面的函数来完成这些功能：ExtractFileName()函数可以从 ExeName 中取出文件名，ExtractFilePath（）函数可以取出路径，ExtractFileExt()函数可以取出扩展名。

2．TApplication.Handle 特性

Handle 特性是一个 HWND（Win32 API 的窗口句柄）类型。在调用某些 Win32 API 的时候，很多情况下会要求传递 Handle 参数，即要求传递应用程序的窗口句柄。

3．TApplication.Icon 和 Title 特性

Icon 特性用于设置当应用程序最小化的时候代表应用程序的图标。

在 Windows 系统的任务栏上，显示在图标右边或者下面的文字可以通过 Title 特性进行设置。

4．TApplication 的其他特性

Active 是一个只读的特性，它的值表明应用程序是否激活和具有输入焦点。Component-Count 特性表明应用程序所包含的组件数。对于那些没有拥有者的组件来说，ComponentIndex 特性总是“-1”。Components 特性是一个数组，它的元素就是那些属于 Application 的组件。TApplication.Owner 特性总是 nil，因为 TApplication 不能被哪个组件所拥有。

3.5.7 TApplication 的方法

1．TApplication.CreateForm()方法

CreateForm()方法的声明如下所示：

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

这个方法用于创建一个 Form 的实例，InstanceClass 参数用于指定这个 Form 的类，创建的实例由 Reference 参数返回。

如果 Form 出现在 Project Options 对话框的 Auto-create forms 列表中，则 Delphi 将自动创建该 Form，并在适当时候自动释放它。当然，用户可以在程序的任何一个地方调用 CreateForm()方法来创建一个 Form 的实例。CreateForm()方法相当于 Form 本身的 Create()方法，但 CreateForm()方法会检查 MainForm 特性是否为 nil，如果是的话，CreateForm()方法会把新创建的 Form 作为主窗体。一般情况下，不要调用 CreateForm()方法，而要调用 Form 本身的 Create()方法。

2．TApplication.HandleException()方法

HandleException()方法用于处理应用程序中出现的异常并显示有关信息。信息将显示在一个由 VCL 定义的标准的异常信息框中。如果希望用自己的方式来显示异常信息，可以创建相应 TApplication.OnException()事件。

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

HelpCommand()、HelpContext()和 HelpJump()方法分别用于操纵由 WinHelp.exe 程序提供的帮助系统。其中，HelpCommand()方法用于执行一条 WinHelp 宏命令和帮助文件中定义的宏；HelpContext()方法用于打开一个帮助主题，主题的编号由 Context 参数传递；HelpJump()方法类似于 HelpContext()方法，但它的 JumpID 参数需要传递一个字符串。

4．TApplication.ProcessMessages()方法

ProcessMessage()方法用于从 Windows 消息队列中检索并处理消息，当程序正在执行一个很长的循环，而系统又需要在这个期间来处理其他代码，例如相应“取消”按钮等操作，这时候就要用到 ProcessMessages()方法。相反，HandleMessages()方法如果发现没有消息，它就会使应用程序处于空闲状态，而 ProcessMessages()方法则不会使应用程序处于空闲状态。

5．TApplication.Run()方法

Delphi 会自动调用 Run()方法，而不需要用户自己去调用它，但用户需要知道这个方法到底做了些什么工作。首先，Run()方法建立了一个退出过程，以保证应用程序在退出时会

释放所有的组件，然后，它就建立一个循环来处理消息，直到应用程序终止。

6．TApplication.ShowException()方法

ShowException 方法需要传递一个异常类作为参数，它就显示有关该异常的信息。后面的“重载应用程序的异常处理”将详细介绍 ShowException()方法。

7．TApplication 的其他方法

Create()和 Destroy()方法用于创建和删除 TApplication 的实例。不过，这两个方法是 Delphi 内部调用的，应用程序本身不应当调用它。MessageBox()方法用于打开一个 Windows 消息框。Minimize()方法用于把应用程序的主窗口最小化。Restore()方法用于把应用程序的主窗口恢复为最大化或者最小化之前的大小。Terminate()方法用于终止应用程序的执行。与 Halt()方法不同的是，Terminate()方法会隐含调用 PostQuitMessage()方法来检查要处理的消息。

> 注意：调用 Terminate()方法可以终止应用程序的执行。Terminate()方法会调用 Windows 的 PostQuitMessage()函数向应用程序的消息队列中发一个消息。VCL 据此释放应用程序创建的所有对象。但是，调用 Terminate()方法并不是会使应用程序马上终止，而是当应用程序检索到 WM_QUIT 消息时才会真正终止。而 Halt()方法则会使应用程序的执行马上终止，但它不会释放先前创建的对象，也不会返回到调用 Halt()方法的地方。

3.5.8　TApplication 的事件

TApplication 有一些事件，用户可以建立处理这些事件的句柄。与这些在组件选项板上的组件不同的是，TApplication 的事件在 Object Inspector 上是找不到的。要建立处理某个事件的句柄，用户必须先声明一个作为句柄的方法，然后在运行期动态地把该方法赋值给某个事件特性。表 3.2 中列出了 TApplication 的所有事件。

表 3.2　TApplication 的事件

事件	描述
OnActive()	当应用程序被激活时将触发这个事件，相应的有 OnDeactive 事件
OnException()	当一个未处理的异常发生时，将触发这个事件
OnHelp()	当用户请求帮助的时候将触发这个事件。例如，用户按下 F1 键，或者程序调用 HelpCommand()、HelpContext()或者 HelpJump()函数
OnMessage()	当应用程序接到一个消息时将触发这个事件。特别注意的是，所有的消息都会触发这个事件，因此，这个事件可能会造成应用程序执行的瓶颈
OnHint()	当鼠标指向某个控件时将触发这个事件，这样就可以显示提示信息
OnIdle()	当应用程序进入空闲状态时将触发这个事件。处于空闲状态以后，需要收到一个消息才能把应用程序唤醒

3.5.9　TScreen 类

TScreen 封装了有关内屏幕的消息。TScreen 既不能作为组件加到 Form 上，也不能在

运行期动态地创建它。Delphi 会自动创建一个 TScreen 类型的全局变量——Screen。TScreen 的有些特性很有用，这些特性见表 3.3。

表 3.3 TScreen 的特性

特性	含义
ActiveControl	这是一个只读的特性，它表明屏幕上哪个控件当前具有焦点。当焦点从一个控件切换到另一个控件时，ActiveControl 将在失去焦点的那个控件发生 OnExit 事件之前切换为新的控件
ActiveForm	表明屏幕上哪个 Form 具有焦点。不管是同一个应用程序内还是不同的应用程序之间切换，都会使这个特性发生变化
Curor	这个特性用于设置应用程序的光标形状。它的默认值是 crDefault。每个控件有自己的 Cursor 特性，可以单独修改。不过，如果 TScreen 的 Cursor 特性设为其他值，则所有控件的光标形状都会更着修改，除非 Screen.Cursor 又恢复为 crDefault。
Cursors	这是一个列表，列出了屏幕所支持的各种光标形状
FormCount	表示应用程序中 Form 的个数
DataModuleCount	表示应用程序中数据模块的个数
DataModules	这是一个数组，其中的元素就是应用程序的数据模块
Forms	这是一个数组，它的元素就是应用程序中的 Form
Fonts	这是一个列表，列出了屏幕所支持的各种字体
Height	表示屏幕的高度（以像素为单位）
PixelsPerInch	表示 System 字体的相对缩放比例
Width	这是屏幕的宽度（以像素为单位）

3.6 MDI 应用程序

MDI 也称为多文档界面，它是从 Windows 2.0 下的 Microsoft Excel 电子表格程序开始引入的，这样，Excel 的用户就可以同时操作几张表格。在多个 Form 上同时处理事件看起来有些困难，在传统的 Windows 编程中，程序员需要知道 Windows MDI Client 类、MDI 数据结构和 MDI 特有的函数和消息。而在 Delphi 编程中，创建 MDI 应用程序是非常简单的。本节将向读者介绍编写 MDI 应用程序的基础知识，在此基础上，读者就可以轻松地掌握更高级的技术。

3.6.1 创建 MDI 应用程序

要创建 MDI 应用程序，用户必须熟悉 Form 的两种样式：fsMDIForm 和 fsMDIChild，并且要熟悉 MDI 的编程方法。下面就介绍一些关于 MDI 的基本概念。

一个 MDI 应用程序所包含的窗口有：

- 框架窗口：应用程序的主窗口。这个窗口有标题、菜单条和系统菜单。右上角有最小化、最大化和关闭按钮。框架窗口中的空白区域就是所谓的客户区，也是一个实际的子窗口。

- 客户窗口：MDI 程序的窗口管理器，用于处理与 MDI 有关的命令并管理子窗口。当用户创建框架窗口的时候，VCL 会自动创建客户窗口。
- MDI 子窗口：实际的文档，例如文本文件、表格、位图和其他文档。与框架窗口相似，子窗口也有标题、系统菜单、最小化、最大化和关闭按钮，可能还有一个帮助按钮。子窗口的菜单和框架窗口的菜单是有关系的，当子窗口最大化的时候，它的菜单可以融合到主窗口的菜单中。子窗口不能移出客户区。

Delphi 不要求用户熟悉 MDI 的窗口消息，客户窗口会负责管理 MDI 的功能，例如排列子窗口等。例如，要层叠子窗口，传统的方法是调用 Win32 API 函数 SendMessage()来发送 WM_MDICASCADE 消息给客户窗口。但在 Delphi 中，用户只要调用 Cascade()方法就可以了。

MDI 应用程序中的主窗口的 FormStyle 特性应为 fsMDIForm，子窗口的 FormStyle 特性应为 fsMDIChild，当程序中创建一个子窗口的实例以后，该子窗口将自动出现在主窗口的客户区中。要注意的是，在子窗口的 FormClose 事件中，应当把 Action 参数设置为 caFree，这样就会保证当程序关闭时自动释放该子窗口的实例。也就是说，当调用子窗口的 Close()方法时，MDI 子窗口并不会自动关闭，用户必须在处理 OnClose 事件的句柄中决定如何处理这些子窗口。处理 OnClose 事件的句柄需要传递一个变量参数，该参数（即 TCloseAction）可以是以下 4 种中的一种：

- caNone　　什么都不做
- caHide　　隐藏但不释放
- caFree　　释放
- caMinimize　　最小化（默认）

当一个子窗口被激活后，将触发 OnActivate()事件，在这里，用户必须进行一些操作，其中包括调用 MainForm.SetToolBar()事件，这个方法使工具栏出现在主窗口上，而不是子窗口上。这样，当不同类型的子 Form 激活的时候，主 Form 上可以出现不同的工具栏，对应于相应的子窗口。

主窗口 MainForm 中有一个特性 MDIChildren，它是一个数组，其中的元素是应用程序中所有活动的子窗口。MDICount 特性是当前活动的子窗口的数量。

3.6.2　使用菜单

在 MDI 应用程序中使用菜单并不会比其他类型的应用程序中使用菜单更困难，只是稍有不同。下面将介绍如何通过“菜单合并”的技术使子 Form 能够共享主 Form 的菜单栏。

放在主 Form 上的菜单的每一个命令都有一个 GroupIndex 特性，从 Object Inspector 中可以看到它的值。这里，GroupIndex 特性的作用体现在合并菜单上，这意味着，当主 Form 打开一个子 Form 时，子 Form 的菜单就合并到主 Form 的菜单中了。GroupIndex 特性决定了菜单中命令的替换顺序。

如果子 Form 菜单中的某个命令的 GroupIndex 特性与主 Form 菜单的某个命令的 GroupIndex 值相同，那么子 Form 菜单的这个命令将替代主 Form 菜单上的命令，其他命令

将按照他们 GroupIndex 值的顺序排列。

对于 MDI 应用程序来说，菜单合并是完全自动的，用户只要正确地设置了命令的 GroupIndex 特性值，当一个子 Form 激活的时候，菜单就能正确地合并。

在 MDI 应用程序中，会有一个 Window 菜单，这里罗列着所有活动的子窗口。要把打开的子窗口列在 Window 菜单下，用户只需要把 TMainMenu.WindowMenu 特性设为某个命令。注意，用户只能选择菜单栏上可见的菜单。

3.6.3 隐藏一个子 Form

如果用户试图用下面的语句来隐藏 MDI 程序的一个子 Form，Delphi 将返回一个错误，如下所示：

```
ChildForm.Hint;
```

之所以会返回错误是因为，以这种方式隐藏子 Form 是不允许的。Windows 在 MDI 上的实现存在缺陷，隐藏子 Form 会破坏子窗口的层叠次序。除非用户在使用这个技术时非常小心，否则，试图隐藏子 Form 会给应用程序带来混乱。

有时候又确实需要隐藏子 Form，这里有两种方法可以实现它，但使用起来要非常小心。

一个方法是不画出子 Form。用户可以调用 LockWindowUpdate()函数来禁止画 MDI 子窗口。如果用户要创建一个子 Form 但又不想显示它，这个时候，就可以使用上面的技术来实现它。要注意的是，任何时刻，只能有一个窗口被锁定。如果将“0”传递给 LockWindowUpdate()函数，可以画出先前的窗口。

隐藏子 Form 的另一个办法是调用 Win32 API 函数 ShowWindow()。通过传递 SW_HIDE 参数就可以隐藏 Form，以后要恢复这个 Form 的时候，用户必须调用 SetWindowPos()函数。这个技术适用于已经创建或者已经显示给用户的窗口。

> 注意：这里不得不提醒读者，Windows MDI 框架是存在缺陷的，所以，建议大家不要继续在 MDI 模式下开发应用程序。如果必须要继续使用 MDI 的话，用户一定要注意它是有缺陷的。

3.7 公共体系结构

过去，用户不得不花费很大的精力用于创建应用程序的体系结构，而现在用户可以轻松多了。问题是，很多的开发者往往急于写代码而忽略了应用程序的结构，这使得一个项目往往以失败而告终。

3.7.1 应用程序的体系结构

下面列出了一些用户应当考虑的问题：

- 体系结构支持代码重用吗？
- 应用程序中模块、对象能够本地化吗？
- 修改体系结构容易吗？

- 用户界面和后端可以本地化吗？
- 特性结构支持团队开发吗？

上面这几个问题其实只是开发过程中要考虑的一部分问题。

3.7.2　Delphi 固有的体系结构

读者可能经常听到这句话，作为一个 Delphi 开发者，没必要是一个组件编写者。尽管这句话是正确的，但下面这句话也是正确的，即如果用户是一个组件编写者，就一定是一个更优秀的 Delphi 开发者。

这是因为，组件编写者更加清楚地知道对象模式和 Delphi 应用程序的体系结构，这意味着组件编写者能更好地发挥它们的优势。事实上，读者可能已经听说过了，Delphi 本身就是用 Delphi 编写的。Delphi 本身就是一个运用体系结构的示例。

即使用户并不想编写一个组件，但掌握体系结构还是有好处的。

3.7.3　体系结构的示例

为了证明 Form 继承以及对象库的强大，下面将定义一个公共的应用程序体系结构。主旨是代码重用性、修改的灵活性、一致性和易于团队开发。

Form 继承，更准确地说是框架，它们的典型应用是在数据库应用程序中。Form 应当对数据库的操作具有感知能力。Form 上还应当包含一些公共组件，例如工具栏和状态栏，以便对数据库表进行操作。另外，这些 Form 还应当提供事件，以便跟踪 Form 的变化。

应用程序的框架应当允许团队开发。越来越多的应用程序已经不是一个人或者两个人就可以开发的了，而且，应用程序的开发往往有很强的时间要求，这就要求一个团队同时进行一个应用程序的开发，每个成员各自工作于应用程序的一部分，每个部分之间不允许重复和覆盖。一个良好的应用程序体系结构，可以使一个团队很好地协作开发，缩短开发周期，而且良好的模块划分更加利于后期的修改和维护。

一个数据库 Form 的框架可以分为 3 个层次，见表 3.4。

表 3.4　数据库 Form 框架

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

1．TchildForm（子 Form）

TChildForm 是那些能成为其他窗口的子窗口的基类。

TChildForm 支持团队开发，每个成员可以工作于应用程序的一部分。同时，TChildForm 也实现了漂亮的用户界面，用户可以在应用程序内打开一个 Form，作为一个单独的实体。

2．TDBModeForm（数据库基础模式 Form）

TDBModeForm 是从 TChildForm 继承下来的，它能够感知数据库的状态。在 TDBModeForm 中还提供了一些事件，以跟踪数据库状态的变化。

3．TDBNavStatForm（数据库导航/状态 Form）

TDBNavStatForm 携带了框架的许多功能。它包含了数据库导航栏和状态栏，能够随数据库的状态而发生变化。例如，当用户使数据库进入浏览状态，导航栏上的提交和取消按钮就被禁止，而当进入插入状态和编辑状态的时候，这两个按钮才有效。状态栏上将显示数据库的状态。

这里没有列出上述 3 种 Form 的实现代码，但它们很好地表达出了 Form 框架的层次。具有很好的代码重用性、修改的灵活性、一致性和易于团队开发性。

3.8 程序窗体设计

要使用户自己开发的应用程序有一个美观友好的界面，程序的窗体设计起着至关重要的作用，这其中包括主窗体以及其他的对话框子窗体。有时候，仅仅利用 Delphi 提供出来的 TForm 的各种特性和方法是远远不够的，用户还需要大量的 Win32 API 函数的帮助和 Windows 消息处理。

3.8.1 显示程序的启动界面

如果需要的话，任何一个规范的应用程序，都要在启动前显示一个启动界面，用来显示一些有关该应用程序以及开发商的信息。启动界面的另外一个重要的作用是填充系统启动时初始化所占用的一段时间空白。

在启动程序之前显示启动画面的方法有很多，但有些是调用计时器来延时的，这种方法往往会延长程序的启动时间，并不实用，下面介绍一种被广泛采用的方法。

新建一个启动界面窗体，假设为 dlg_Cover，设置好其 BorderIcons、BorderStyle、FormStyle、Position 等特性（其中 BorderIcons 特性设为空、BorderStyle 特性设为 bsNone，这样将不显示窗体边框），在窗体上加入图片框、文本框，设置要显示的内容信息，执行 Project | Options 命令，把 dlg_Cover 从 Auto-create forms 里转到 Available forms 中。打开工程单元，其内容如下所示：

```
begin
    Application.Initialize;
    Application.Title := '演示程序';
    dlg_Cover := Tdlg_Cover.Create(Application);
    dlg_Cover.Show;   //显示启动界面
    dlg_Cover.Update;   //刷新启动界面
    Application.CreateForm(Tfm_Main, fm_Main); // 创建程序主窗体，或者初始化程序
    dlg_Cover.Free;   // 释放启动界面窗体
    Application.Run;
end.
```

3.8.2　限制窗体的大小

读者也许已经注意到了，Delphi 最上面的窗口在极大时只占屏幕的一小部分，实现这种功能并不困难，只需要截获 Windows 系统消息 WM_GETMINMAXINFO，稍加处理就可以了。

首先，在 FORM 私有声明部分加上以下一行声明：

```
procedure WMGetMinMaxInfo(var Message :TWMGetMinMaxInfo);
  message WM_GETMINMAXINFO;
```

然后，在声明部分加上以下几行消息处理：

```
procedure TForm1.WMGetMinMaxInfo(var Message: TWMGetMinMaxInfo);
begin
with Message.MinMaxInfo^ do
        begin
                ptMaxSize.X := 200; //  最大化时宽度
                ptMaxSize.Y := 200; //  最大化时高度
                ptMaxPosition.X := 99; //  最大化时左上角横坐标
                ptMaxPosition.Y := 99; //  最大化时左上角纵坐标
        end;
Message.Result := 0; //  告诉 Windows 你改变了  minmaxinfo
inherited;
end;
```

3.8.3　实现窗体拖动

为了实现通过 Form 的 Client 区实现窗体拖动这个功能，需要“欺骗”一下系统，因为当鼠标点在 Caption 标题区时，可以拖动当前窗体，因此，当点在 Client 客户区时，只要让系统以为是点在 Caption 标题区，那么就可以实现拖动的功能了。

在窗体中加入以下事件：

```
procedure WMNCHitTest(var M: TWMNCHitTest); message wm_NCHitTest;
```

该事件的实现如下：

```
procedure TForm1.WMNCHitTest(var M: TWMNCHitTest);
begin
        inherited; //  调用默认的事件处理程序
        if M.Result = htClient then
                M.Result := htCaption; //是在 Client 区让 Windows 认为是在 Caption 区
        end;
```

3.8.4　Form 生成的顺序

对于普通的 Form，各事件的发生次序如下所示：

（1）OnCreate
（2）OnShow
（3）在屏幕上看到这个窗体
（4）OnActivate
（5）OnPaint

如果是 MDI 窗体，并且 MDIChild 的第一个子窗体是在程序启动时就出现在 MDIForm 中的话，各事件的发生次序如下所示：

（1）主窗体的 OnCreate
（2）子窗体的 OnCreate
（3）子窗体的 OnShow
（4）子窗体的 OnActivate
（5）主窗体的 OnShow
（6）在屏幕上看到主窗体及第一个子窗体
（7）主窗体的 OnPaint

很多情况下，有些事件往往希望在创建并看到窗体显示在屏幕上以后马上执行，那么可以将该事件写在 OnPaint 事件句柄中，不过，OnPaint 是一个经常发生的事件，每当显示的窗体图像刷新就会触发该事件，所以需要一个变量来控制，不要每次窗体重画时都触发事件，代码如下所示：

```
procedure TForm1.FormPaint(Sender: TObject);
begin
    if FirstPaint then begin
        DoSomething . . .
        FirstPaint := False; // 关闭控制变量
    end;
end;
```

3.8.5 停靠窗口

从 Delphi 4 开始，Delphi 增加了一个很明显的增强功能，就是在窗体中或者在其他控件顶部停靠窗口的能力，当然在 Delphi 5 和 Delphi 6.0 中都支持该功能。

停靠窗口实现很容易。下面介绍如何将一个标准窗体、面板 Panel 或者 TControlBar 变成一个可以停放其他窗口的空间。

首先，在窗体上放置一个 Panel，设置它的 DockSite 特性为 True；然后，用户可以把 Panel 藏在窗体的边上，直到用户需要把什么东西停放在它上面时再让它显示出来。要想做到这一点，需要把 Panel 的 AutoSize 的特性设置为 True。这样，当用户把一个控件或者其他窗体拖到窗体的边上的时候，Panel 就会突然出现，作为停放的空间。对于用户来说，他所做的动作好像只是把控件停靠在窗体的边缘上，而永远也不必知道控件实际上是放在一个 Panel 上。

要把一个控件或者窗体变成可停靠的只需要下面简单的两步：

（1）把 DragKind 特性设置为 dkDock。

（2）把 DragMode 特性设置为 dmAutomatic。

下面来看两个停靠的示例。

第一个示例是在一个窗体中停靠控件：在程序的主窗体上放置两个 TPanel，其中左边的 Panel 设置其 DockSite 特性为 True，右边的 Panel 不作任何设置。另外放置两个 TShape 控件在窗体上，对它们做相同的设置：DragKind 设为 dkDock，DragMode 设为 dmAutomatic，这样，两个 TShape 都是可停靠的了。程序运行以后，将一个 Shape 拖动到左边的 Panel 上，而另一个 Shape 拖放到右边的 Panel 上，就看到程序的界面如图 3.10 所示：

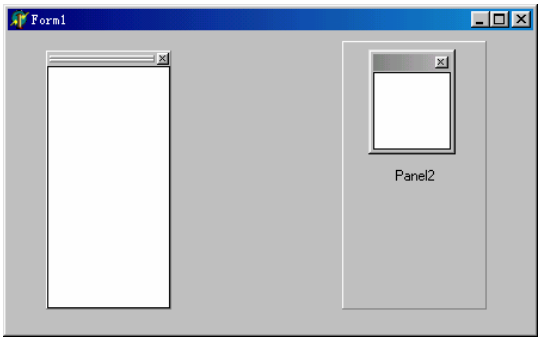

图 3.10　控件停靠程序

TPanel 控件有一个 UseDockManager 特性，如果该特性为 True 的话，它的作用是将停靠在上面的控件的尺寸大小扩充为 Panel 的大小，即充满全部 Panel 空间，就像图 3.10 中，左边的 Panel 一样，如果有多个停靠控件同时停靠在 Panel 上，那么它们就会各自占用一部分空间，如图 3.11 中左边的 Panel 所示，Delphi 6.0 的主界面很多地方都具有这种停靠功能。相反，如果 UseDockManager 特性为 False，那么程序将不会改变停靠控件的大小，而保持原有尺寸，如图 3.11 中右边的 Panel 所示，操作结果如图 3.11 所示。

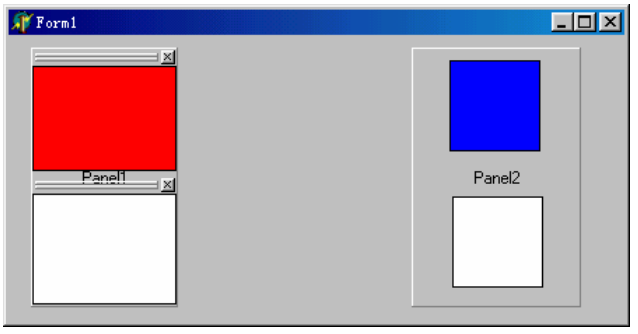

图 3.11　UseDockManager 特性的作用

第二个示例是在一个窗体上停靠另外的窗体。程序中需要创建 3 个 Form，主 Form 上放置两个 Panel，将它们设置为可停靠控件，即 DockSite := True；另外两个 Form 设置为停靠控件，即 DragKind := dkDock，并且 DragMode := dmAutomatic。程序中需要设置两个按钮，分别用来显示出 Form2 和 Form3，运行以后，单击一个按钮显示 Form2，然后将它拖

放到其中一个 Panel 上，单击另外一个按钮显示出 Form3，同样可以拖放到另一个 Panel 上，当然也可以拖放到相同的 Panel 上，这时它们将分割这个 Panel，各自占用一半的空间，以下两图分别显示出了这两种情况。为了明显起见，这里的 Form2 和 Form3 分别标识了不同的颜色，如图 3.12 和图 3.13 所示。

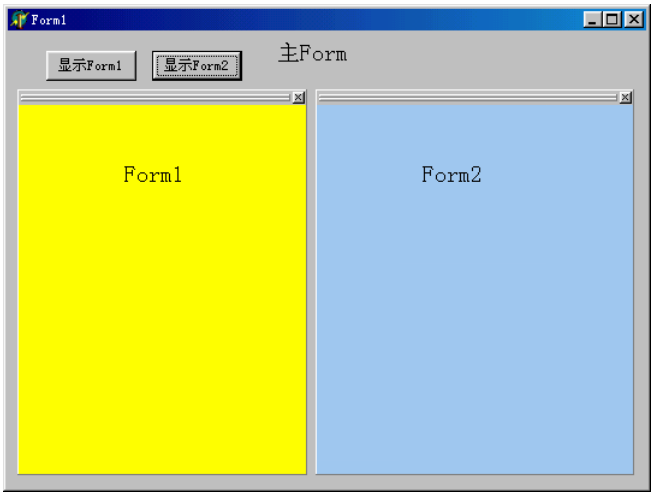

图 3.12　Form 停靠到不同 Panel 上

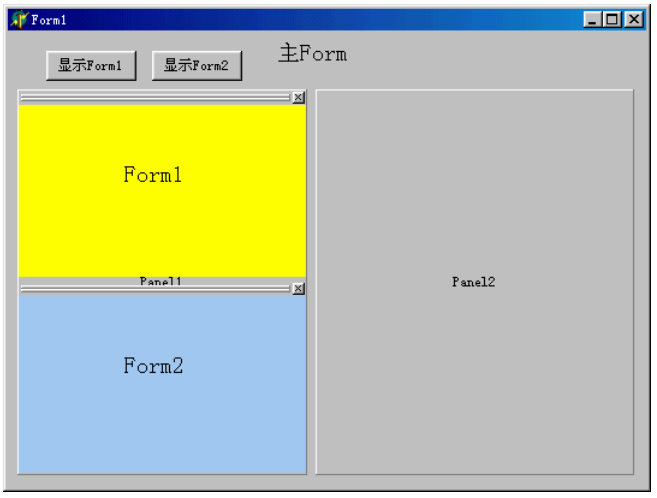

图 3.13　Form 停靠到相同的 Panel 上

到目前为止，读者对于如何使用可停靠窗口已经有了一些认识，当然这些都是最基本的，在此基础上可以做一些进一步的试验，例如创建可以容纳多个控件的面板 Panel，并且把这些面板拖放到 Delphi 的 Component Palette（组件选项板）上，或是拖放到其他的面板活动窗体上。

3.9　本 章 小 结

本章重点讨论了组成 Delphi 应用程序项目的 3 大要素：TForm、TApplication 和 TScreen，以及项目管理技术和体系结构。这些都是在一个 Delphi 程序中起着框架作用的类。另外还涉及到了组成 Delphi 项目所需的各种文件类型，例如项目文件、单元文件、窗体文件等。

这里还介绍了如何在 Delphi 中创建 MDI 应用程序，读者也学到了一些关于 MDI 的高级技术。有了这些基础，用户就能够创建更具专业外观的 MDI 程序。在本章的最后还介绍了程序窗体设计的一些必要技巧和知识。

本章的最后，介绍了停靠控件的技术，利用该技术，用户可以编写出像 Delphi 6.0 的 IDE 一样的停靠效果。

第 4 章　代码标准规则

本章将讨论使用 Delphi 的编码标准，主要是为开发组提供一个方法，使他们在编程时有一致的格式可以遵循。这样，开发组中每个开发人员编写的代码都能够被其他的开发人员所理解。这要求大家使用一致的代码样式。

本章中所描述的代码标准，用户可以自由地使用和修改，以满足不同的需要。但是，并不建议在这些标准上花太多的时间。之所以介绍这些标准，是因为当新的开发人员加到开发组中来的时候，他们可能已经对 Borland 的标准很熟悉，这样就能够很快地融入到开发组的工作中来。

本章不包含用户界面标准。用户界面标准是独立于其他标准的，并且同样是非常重要的。读者也许注意到了，所有的 Microsoft 的 Office 套件的用户界面非常类似，这就是 Microsoft 的用户界面标准。可以到 Microsoft Developers Network 和其他的地方去查一下有关信息。

4.1　源代码格式规则

有很多的 Delphi Expert（专家工具）提供了自动规范代码格式的功能，用户可以通过它来定制自己的格式规范。这样的工具有：DelForExp（Souce Code Formatter），可以在下面的网站上找到它：

http://www.slm.wau.nl/wkao/delforexp.html

4.1.1　缩进

缩进就是每级间有两个空格。不要在源代码中保存制表符，这是因为，制表符的宽度随着用户设置和代码管理实用程序不同而不同（打印、文档、版本控制）。

可以通过 Tools | Editor Property 命令打开 Editor Property 对话框，也可以在代码编辑框中右击，在弹出的快捷菜单中单击 Property 命令来打开这个对话框。这里可以通过设置 Use tab Character、Smart tab、Optimal fill 和 Tab Stops 来设定 tab 字符是否有效以及 tab 字符的步长。

4.1.2　边距

边距设置为 80 个字符。源代码一般不会因写一个单词而超出边距，但长度超出一行的语句应该用逗号或者换行符换行。换行后，应缩进两个字符。

4.2　Object Pascal

4.2.1　括号

在左括号与下一个字符之间没有空格。同样，右括号和前一个字符之间也没有字符。如下所示：

```
MyFunction( APara );   // 错误
MyFunction(APara);   // 正确
```

不要在组件中包含多余的括号。在源代码中，括号只有在确定要使用的时候才能使用。下面的示例演示了正确和不正确的用法，如下所示：

```
if (I = 2) then    // 这里括号是不需要的
if (I = 2) and (J = 3) then    //这里括号是必需的
```

4.2.2　过程和函数

过程和函数也称为例程，它们的名称应当以大写字母开始，并且要大小写交错使用，以增加可读性。如下所示：

```
procedure thisismyfunction;    // 全部用小写可读性很差
procedure ThisIsMyFunction; // 大小写交错使用增强可读性
```

例程名应当有意义。表示一个动作的例程最好在名称前面加上表示动作的动词作为前缀。如下所示：

```
procedure WriteFile;
```

设置输入参数值的例程名应用 Set **为其前缀，如下所示：**

```
procedure SetUserName;
```

获取数据的例程名应以 Get **为前缀，如下所示：**

```
function GetUserName: string;
```

4.2.3　例程中的形参

1．格式

只要有可能，同一类型的参数应归并到一起，如下所示：

```
procedure FunName(Para1, Para2: Integer; Para3, Para4: string);
```

2．命名

所有形参的名称都应表达出它的用途。如果合适的话，形参的名称最好以字母 A 开头，如下所示：

```
procedure SomeProc(AUserName: string; AUserAge: Integer);
```

3．参数顺序

参数的顺序主要是考虑寄存器的调用规则。

最常用的参数应当作为第一个参数，也就是说，按参数的使用频率依次设置参数。

另外，输入参数要位于输出参数的前面。范围大的参数应当放在范围小的参数之前。

上面的规则只是对于一般的参数设置，有些则例外，例如在事件处理句柄中，TObject 类型的 Sender 参数往往是放在第一个要传递的参数位置上。

4．参数常量

要使记录、数组、短字符串或者接口类型的参数不能被例程所修改，则应当把形参标以 Const。这样，编译器将以最有效的方式生成代码，保证传递的参数不可变。

如果希望其他类型的参数不被例程所修改，也可以标上 Const，这样做对编译效率没有什么影响，但对于例程的调用者来说，可以带来更多的信息。

5．命名冲突

当两个单元中含有相同的名称的例程时，在调用这个例程名称的时候，实际上被调用的是在 Uses 子句中最后出现的那个单元中的例程。为了避免这种情况，可在方法名称前面加上单元名，这样就可以确定所调用的例程是在哪个单元中了。

4.2.4 变量

1．变量的命名和格式

变量的名称应当能够表达出它的用途。

循环控制变量常常是单个的字母，如 I，J 或者 K，也可以是更加有确定意义的名称，例如 ItemIndex。布尔变量的名称必须能够清楚地表达出 True 值和 False 值的意义。

2．局部变量

局部变量用于例程内部，遵循其他变量的命名规则。

如果需要的话，应当在例程的入口立即初始化变量。有些类型的变量会自动初始化，例如：局部的 AnsiString 类型变量会自动初始化为空字符串，局部的接口和调度接口类型的变量自动被初始化为 nil，局部的 Variant 和 OleVariant 类型的变量自动被初始化为 Unassigned。

3．全局变量

一般不鼓励使用全局变量。不过，有时候需要用到，即使如此，也应当把全局变量限制在需要的环境中。例如，一个全局变量可能只在单元的 Implementation 部分是全局的。

所有的全局变量在声明时都将自动进行零初始化，用户也可以直接初始化为一个值，但没有必要初始化为 0、nil 等空值。零初始化的全局变量在编译后的可执行文件中不占用

任何空间。零初始化的数据保存在虚拟的数据段中，而虚拟数据段只在应用程序启动的时候才分配空间。

4.2.5　类型

1．大小写规则

类型标识符是保留字，应当全部小写。Win32 API 类型常常全部大写，并且遵循诸如 Windows.pas 或者其他 API 单元中关于特定类型名的规则。对于其他变量名，第一个字母应大写，其他字母则大小写交错。例如：

```
var
    MyString: string;       // 保留字
    WindowsHandle: HWND;    // Win32 API 类型
```

2．浮点型

不鼓励使用 Real 类型，因为它只是为了与老的 Pascal 代码兼容而保留的。通常情况下，对于浮点数应当使用 Double 型，Double 类型可被处理器优化，并且它是 IEEE 定义的标准的数据格式。当需要比 Double 型提供的范围更大的数据的时候，可以使用 Extend。Extend 是 Intel 专用的类型，Java 不支持，当浮点变量的物理字节数很重要时（可能使用其他语言编写的 DLL），则应当使用 Single 型。

3．枚举型

枚举类型名称代表枚举的用途。名称前面要加“T”字符作为前缀，表示这是个数据类型。枚举型的标识符列表的前缀应包含 2～3 个小写字符，来彼此关联。例如：

```
TSongType = (stRock, stClassical, stCountry, stAlternative, stHeavyMetal, stRB);
```

除非为了给变量一个更加特殊的名称，枚举类型的变量实例的名称与类型相同，但没有前缀“T”。

4．Variant 和 OleVariant 类型

一般情况下不建议使用 Variant 和 OleVariant 类型，但是，在 COM 和数据库应用程序中，当数据类型只有在运行期才能确定类型的时候，这两个类型对编程是有用的。当进行诸如 Automation、ActiveX 控件的 COM 编程时，应当使用 OleVariant，而对于非 COM 编程，则应当使用 Variant。这是因为，Variant 能够有效地保存 Delphi 的原生字符串，而 OleVariant 则只能够将所有字符串转换为 OLE 字符串（即 WideString 字符串），并且没有引用计数功能。

4.2.6　构造类型

Object Pascal 语言与 Pascal 语言一样，除了支持基本的简单类型外还支持构造类型，在 Delphi 中用户主要能建立以下几种数据类型。

1．数组类型

数组类型名应表达出数组的用途。类型名必须加字母“T”为前缀。如果要声明一个指

向数组类型的指针，则必须加字母“P”为前缀，并且在类型声明之前声明。例如：

```
type
    PMyArray = ^TMyArray;
    TMyArray = array [1..100] of Integer;
```

实际上，数组类型的变量实例和类型名称相同，但没有“T”前缀。

2．记录类型

记录类型名应表达出记录的用途。类型名必须加字母“T”为前缀。如果要声明一个指向记录类型的指针，则必须加字母“P”为前缀，并且在类型声明之前声明。例如：

```
type
    PEmplyee = ^TEmployee;
    TEmployee = record
        EmplyeeName: string;
        EmplyeeRate: Double;
End;
```

4.2.7 语句

1．If 语句

在 If / Then / Else 语句中，最有可能执行的情况应放在 Then 子句中，不太可能的情况放在 Else 子句中。

为了避免出现许多 If 语句，可以使用 Case 语句代替。如果多于 5 级，不要使用 If 语句。不要在 If 语句中使用多余的括号。

如果在 If 语句中有多个条件，应按照计算的复杂程度从右到左排。这样，可以使代码充分利用编译器的短路估算逻辑，以最快的速度得出逻辑值。例如，如果 Condition1 比 Condition2 快、Condition2 比 Condition3 快，则 If 语句应以如下方式构造：

```
if Condition1 and Condition2 and Condition3 then
```

2．Case 语句

Case 语句中每种情况的常量应当按数字或者字母的顺序排列。

每个情况的动作语句应当简短并且通常不超过 4～5 行代码。如果动作太复杂，应将代码单独放在一个过程或函数中。

Case 语句的 Else 子句只用于默认情况或错误检测。

3．While 语句

建议不要使用 Exit 过程来退出 While 循环。如果需要的话，应当使用循环条件退出循环。

所有对 While 循环进行初始化的代码应当位于 While 入口前，并且不要被无关的语句隔开。

4．For 语句

如果循环次数是确定的，应当用 For 语句代替 While 语句。

5．Repeat 语句

Repeat 语句类似于 While 循环，并且循环同样的规则。

6．With 语句

使用 With 语句应该小心，要避免过度使用 With 语句，尤其是在 With 语句中使用多个对象或记录。如下所示：

```
with Record1，Record2 do
```

这个情况很容易迷惑编程人员，并且导致检测 bug 困难。

4.2.8　结构化异常处理

异常处理主要用于纠正错误和包含资源。这意味着，凡是分配资源的地方，都必须使用 Try...Finally 来保护资源得到释放。不过，如果是在单元的 Initialization / Finalization 部分或者对象的构造/析构中来分配/释放资源则例外。

1．Try…Finally 的用法

可能的情况下，每个资源分配应当与 Try…Finally 结构匹配。例如，下面的代码可能导致错误，如下所示：

```
SomeClass1 := TSomeClass.Create;
SomeClass2 := TSomeClass.Create;
Try
        DoSomething…
Finally
        SomeClass1.Free;
        SomeClass2.Free;
End;
```

比较安全的资源的分配途径应当如下所示：

```
SomeClass1 := TSomeClass.Create;
Try
SomeClass2 := TSomeClass.Create;
Try
                DoSomething…
Finally
                SomeClass2.Free;
        End;
Finally
        SomeClass1.Free;
End;
```

2．Try…Except 的用法

如果用户希望在发生异常时执行一些任务，可以使用 Try…Except 方法。通常，没有必要为了简单地显示一个错误信息而使用 Try…Except，因为 Application 对象能够自动根

据上下文做到这一点。如果要在 Except 子句中激活默认的异常处理，可以再次触发异常。

3．Try…Except...Else 的用法

通常不鼓励使用带 Else 子句的异常处理语句，因为它会阻塞所有的异常，包括一些设想不到的异常。

4.2.9 类

1．命名和格式

类的名称应当表达出类的用途。类名前要加字母“T”，表示它是一个类型。例如：

```
type
  TCustomer = class(TObject);
```

类的实例名称与类名相同，只不过没有前缀“T”，例如：

```
var
  Custom: TCustomer;
```

2．字段

字段的命名遵循与变量相同的规则，只不过要加前缀“F”，表示这是字段。

类的字段必须都是私有的，如果要从类的作用域外访问字段，可以借助于类的特性来实现。

3．方法

方法是属于某个类的函数或过程，遵循与过程和函数相同的命名规则。

当用户不希望一个方法被派生类重载的时候，应当使用静态方法。

4．虚拟方法和动态方法

如果用户希望一个方法能被派生类重载，应当使用虚拟方法；如果类的方法要被多个派生类直接或者间接地使用，则应当使用动态方法。例如，某一个类含有一个被频繁重载的方法，并且有很多的派生类，则应当将该方法定义为动态的，这样可以减少内存的开销。

5．抽象方法

如果一个类创建实例，则不要使用抽象方法。抽象方法只能在那些从不创建实例的基类中使用。

6．特性访问方法

所有特性访问方法应当定义在类的私有和保护部分。

特性访问方法遵循与过程和函数相同的规则。用于读的方法应当加 Get 前缀，用于写的方法应当加 Set 前缀，并且有一个名为 Value 的参数，其类型与特性的类型相同。

7．特性

特性作为私有字段的访问器，遵循与字段相同的命名规则，只是没有“F”前缀。

特性名应为名词，而不是动词。特性是数据，而方法是动作。数组特性名应当为复数形式，而一般的特性应当为单数形式。

4.3 文　件

4.3.1 项目文件

项目文件的名称应当具有描述意义，例如系统信息程序的名称为 SysInfo.dpr，也可以是缩写的形式，例如，The Delphi Developer's Guide Bug Manager 的项目名称为 DDGBugs.dpr。

4.3.2 Form 文件

Form 文件的名称应当表达出 Form 的用途，并且具有 Frm 的后缀，或者 fm 的前缀，例如，About 的文件名为 AboutFrm 或者 fm_About，主 Form 的文件名为 MainFrm 或者 fm_Main。

4.3.3 数据模块文件

数据模块文件的名称应当表达出数据模块的作用，并且具有 DM 的后缀，例如，Customers 数据模块的文件名为 CustomersDM。

远程数据模块文件的名称与普通的数据模块的不同在于后者是 RDM。

4.3.4 单元文件

1．单元名

单元的名称应当具有描述性。例如，应用程序的主 Form 的单元为 MainFrm.pas。

2．Uses 子句

在 Interface 部分的 Uses 子句应当只包含该部分需要的单元，不要包含由 Delphi 自动添加的单元名。

Implementation 部分的 Uses 子句应当只包含该部分需要的单元，不要有多余的单元。

3．Interface 部分

Interface 部分应当只包含需要被外部单元访问的类型、变量、过程和函数的声明。而且，这些声明应当在 Implementation 部分之前。

4．Implementation 部分

Implementation 部分包括本单元私有的类型、变量、过程和函数的声明。

5．Initialization 部分

不要在单元的 Initialization 部分放置花费很多的代码。否则，将导致应用程序启动非常慢。

6．Finalizatioin 部分

在 Finalization 部分中的代码要确保释放所有在 Initialization 部分中分配的资源。

7．组件单元

组件单元应放在单独的路径中，以区别于定义组件的单元。它们一般不和项目放置在同一路径中，单元文件名应表达出单元内容。

4.4 Form 和数据模块

4.4.1 Form

1．Form 类型的命名标准

Form **类型的名称应当表达出** Form **的用途，并且要加“** T **”为前缀，后跟描述性的名称，最后是** Form**。例如，**About **的** Form **类型名称为：**

```
TAboutForm = class(TForm)
```

主 Form **的类型名称为：**

```
TMainForm = class(TForm)
```

客户登陆 Form **的类型名称为：**

```
TCustomerEntryForm = class(TForm)
```

2．Form 实例的命名标准

Form **实例的名称与相应的类型名称相同，只是没有前缀“** T **”。例如，前面提到的** Form **类型与** Form **实例的名称分别为：**

```
TAboutForm、AboutForm
TMainForm、MainForm
TCustomerEntryForm、CustomerEntryForm
```

3．自动创建的 Form

除非特别的原因，只有主 Form **才自动生成。其他所有** Form **必须从** Project Options **对话框的自动生成列表中去掉。**

4．模式 Form **实例化函数**

所有 Form **单元都应当含有实例化函数，用于创建、设置、显示模式和释放** Form**。这个函数将返回由** Form **返回的模式结果。传递这个函数的参数遵循“参数传递”的规则。之所以要这样封装，是为了便于代码的重用和维护。**

Form **的变量应当从单元中移走，再在实例化函数中作为局部变量定义。注意，这要求该** Form **从** Project Options **对话框中的自动生成列表中删除。**

4.4.2　数据模块

1．数据模块的命名标准

数据模块类型名称应当表达出它的用途，并且加前缀“T**”，后跟描述性名称，最后是** Module**，例如，**Customer **数据模块的类型名为：**

```
TCustomerDataModule = class(TDataModule)
```

2．数据模块实例的命名标准

数据模块实例的名称应当与相应的类型名称相同，但没有前缀“T**”。**

4.5　包

4.5.1　运行期包与设计期包

运行期包中应当只包含所需要的单元。那些特性编辑器和组件编辑器的单元应当放在设计期包中。注册单元也应当放在设计期包中。

4.5.2　文件命名标准

包的命名遵循下列模式：

- iiilibvv.pkg　**设计期包**
- iiistdvv.pkg　**运行期包**

其中，iii **代表一个** 3 **个字符的前缀，用于标识公司、个人或其他需要标识的事情。**vv **代表包的版本号。名称中** lib **和** std **分别表示这是设计期包还是运行期包。**

4.6　组　　件

4.6.1　自定义组件

1．组件类型的命名标准

组件的命名和类的命名类似，只不过它有 3 **个小写字符的前缀。这些前缀用以标识公司、个人或者其他实体。例如，一个时钟单元可以这样声明：**

```
TddgClock = class(TComponent)
```

2．组件单元

组件单元只能含有一个主要组件，这是指出现在组件选项板上的组件。其他辅助性的组件或者对象也可以包含在同一个单元中。

3．注册单元

组件的注册过程应当从组件单元中移到一个单独的单元中，这个注册单元用于注册所有组件、特性编辑器、组件编辑器和向导等。

组件注册应当在设计期包中进行。因此，注册组件应当包含在设计期包而不是运行期包中。

4.6.2 组件实例的命名规则

组件的名称应当具有描述性。Delphi 没有为组件指定默认的名称，组件应当有一个小写的前缀以表明其类型，这是为了便于在 Object Inspector 和 Code Explorer 中查找特定的组件。

4.6.3 组件的前缀

下面是部分常用的 Delphi 标准组件的前缀。

1．Standard 选项卡

该选项卡的组件及其前缀见表 4.1。

表 4.1 Standard 选项卡的组件

组件	前缀
Tframe	frm
TMainMenu	mm
TPopupMenu	pm
TMainMenuItem	mmi
TPopupMenuItem	pmi
TLabel	lbl
TEdit	edt
TMemo	mem
TButton	btn
TCheckBox	cb
TRadioButton	rb
TListBox	lb
TComboBox	cb
TScrollBox	scb
TGroupBox	gb
TRadioGroup	rg
TPanel	pnl
TActionList	al

2．Additional 选项卡

该选项卡的组件及其前缀见表 4.2。

表 4.2　Additional 选项卡的组件

组件	前缀
TBitBtn	bbtn
TSpeedButton	sb
TMaskEdit	me
TDrawGrid	dg
TStringGrid	sg
TImage	img
TScrollBox	sbx
TBevel	bvl
TSplitter	spl
TShape	shp
TApplicationEvents	ae
TValueListEditor	vle
TLabeledEdit	le
TColorBox	clb
TActionManager	am

3．Win32 选项卡

该选项卡的组件及其前缀见表 4.3。

表 4.3　Win32 选项卡的组件

组件	前缀
TTabControl	tbc
TPageControl	pgc
TImageList	il
TRichEdit	re
TProgressBar	prb
TTrackBar	tbr
TUpDown	ud
THotKey	hk
TAnimate	ani
TDateTimePicker	dtp
TTreeView	tv
TListView	lv
THaderControl	hdr
TStatusBar	sb
TToolBar	tlb

4．System 选项卡

该选项卡的组件及其前缀见表 4.4。

表 4.4 System 选项卡的组件

组件	前缀
TTimer	tm
TPaintBox	pb
TMediaPlayer	mp
TOleContainer	olec
TDDEClientConv	ddcc
TDDEClientItem	ddci
TDDEServerConv	ddsc
TDDEServerItem	ddsi

5．Internet 选项卡

该选项卡的组件及其前缀见表 4.5。

表 4.5 Internet 选项卡的组件

组件	前缀
TClientSocket	csk
TServerSocket	ssk
TWebDispatcher	wbd
TPageProducer	pp
TQueryTableProducer	tp
TDataSetTableproducer	dstp
TNMDayTime	nmdt
TNMEcho	nec
TNMFinger	nf
TNMFtp	nftp
TNMHttp	nhttp
TNMMsg	nMsg
TNMMSGServ	nmsg
TNMNNTP	nntp
TNMPop3	npop
TNMUUProcessor	nuup
TNMSMTP	smtp
TNMStrm	nst
TNMTime	ntm
TNMStrmServ	nsts
TNMUdp	nudp
TPowerSock	psk
TNMGeneralServer	ngs
THtml	html
TNMUrl	url

（续表）

组件	前缀
TSimpleMail	sml

6．Data Access 选项卡

该选项卡的组件及其前缀见表 4.6。

表 4.6　Data Access 选项卡的组件

组件	前缀
TDataSource	ds
TTable	tbl
TQuery	qry
TStoredProce	sp
TDataBase	db
TSession	ssn
TBatchMove	bm
TUpdateSQL	usql

7．Data Controls 选项卡

该选项卡的组件及其前缀见表 4.7。

表 4.7　Data Controls 选项卡的组件

组件	前缀
TDBGrid	dbg
TDBNavigaotr	dbn
TDBText	dbt
TDBEdit	dbe
TDBMemo	dbm
TDBImage	dbi
TDBListBox	dblb
TDBComboBox	dbcb
TDBCheckBox	dbch
TDBRadioGroup	dbrg
TDBLookupListBox	dbll
TDBLookupComboBox	dblc
TDBRichEdit	dbre
TDBCtrlGrid	dbcg
TDBChart	dbch

8. QReport 选项卡

该选项卡的组件及其前缀见表 4.8。

表 4.8 QReport 选项卡的组件

组件	前缀
TQuickReport	qr
TQRSubDetail	qrsd
TQRBand	qrb
TQRChildBand	qrcb
TQRGroup	rqg
TQRLable	qrl
TQRText	qrt
TQRExpr	qre
TQRSysData	qrs
TQRMemo	qrm
TQRRichText	qrrt
TQRDBRichText	qrdr
TQRShape	qrsh
TQRImage	qri
TQRDBMImage	qrdi
TQRCompositeReport	qrcr
TQRPreview	qrp
TQRChart	qrch

9．Dialogs 选项卡

对话框组件实际上是以组件形式封住起来的 Form，因此它遵循 Form 的命名规则，其类型已经由组件的名称定义了。实例的名称和类型的名称相同，但没有前缀“T”。

10．Win31 选项卡

该选项卡的组件及其前缀见表 4.9。

表 4.9 Win31 选项卡的组件

组件	前缀
TDBLookupList	dbll
TDBLookupCombo	dblc
TTabSet	ts
TOutline	ol
TTabbedNoteBook	tnb
TNoteBook	nb
THeader	hdr
TFileListBox	flb
TDirctoryListBox	dlb
TDriveComboBox	dcb
TFilterComboBox	fcb

11．Samples 选项卡

该选项卡的组件及其前缀见表 4.10。

表 4.10　Samples 选项卡的组件

组件	前缀
TGauge	gg
TColorGrid	cg
TSpinButton	spb
TSpinEdit	spe
TDirectoryOutline	dol
TCalendar	cal
TIBEventAlerter	ibea

12．ActiveX 选项卡

该选项卡的组件及其前缀见表 4.11。

表 4.11　ActiveX 选项卡的组件

组件	前缀
TChartFX	cfx
TVSSpell	vsp
TFlBook	flb
TVTChart	vtc
TGraph	grp

13．Midas 选项卡

该选项卡的组件及其前缀见表 4.12。

表 4.12　Midas 选项卡的组件

组件	前缀
Tprovider	prv
TclientDataSet	cds
TQueryClientDataSet	qcds
TDCOMConnection	dcom
TOleEnterpriseConnection	olee
TSocketConnection	sck
TRemoteServer	rms
TMidasConnection	mid

4.7　本 章 小 结

本章讨论了 Delphi 开发的编码标准，学习并努力遵循这些标准将使用户的开发工作更加标准，更加容易被其他人所理解，也使用户更加容易理解别人标准化的代码。这些标准中的很多东西并不需要刻意去记，只要在日常的开发工作中尽量使用并遵循它们，就会很容易记住它们。

第 5 章　使用 Delphi 6.0 的组件

Delphi 6.0 中包含的组件瀚如烟海，本书不可能对它们逐一介绍。这一章将介绍其中若干类比较重要、比较常用的组件，如果想了解更多的关于 Delphi 组件的信息，可以参考 Delphi 的在线帮助。

Delphi 6.0 根据应用程序运行的平台不同对组件选项板中的组件做了一些不同的处理，例如，如果开发一套运行于 Linux 平台下的应用程序，会发现 Additional 组件组中的组件少了很多，而 Win32 和 System 组件组都没有出现，取而代之的是一套 Common Controls 组件组。这些不同往往是由于组件本身使用了 Windows 内部提供的组件的缘故，因为应用程序不在 Windows 系统中运行，自然就不能够使用 Windows 提供的各种组件或者函数等，但是不必担心，Borland 在这方面做了大量的工作，移植了大量的 Windows 下的组件和函数到 Delphi 6.0 中，也就是说，脱离了 Windows 环境，仍然可以继续使用大量类似 Windows 组件的东西，甚至使用方法完全一样，而实际上，这些组件或者函数其实是 Borland 公司重新设计和开发的。

5.1　Standard 组件组

Standard 组件组中提供了很多开发界面应用程序所必需的组件，其中包括 TMainMenu（主菜单）、TPopMenu（右键菜单）、TLabel（标签）、TEdit（编辑框）、TButton（按钮）、TCheckBox（多选框）、TRadioBox（单选框）、TListBox（列表框）、TComboBox（下拉框）、TGroupBox（分组框）、TPanel（面板）以及 TActionList（动作列表）。除非用户开发的应用程序不需要任何显示界面或者没有任何界面操作，例如控制台程序，除此以外，只要有界面和操作，就不可避免地要使用上面列举的组件中的多种组件。很多第三方组件在这些组件的基础上进行了继承，做出了很漂亮的效果，能够使用户的应用程序界面看起来更加美观，不过除非用户非得用它提供的某个特别的功能，笔者还是建议用户尽量使用这些最基本的 Standard 组件，因为它们在系统资源保护和消耗方面做得非常好。基于 Windows 和 Linux 平台的应用程序开发中的 Standard 组件组基本相同，该组组件如图 5.1 所示。

图 5.1　Standard 组件组

5.1.1　TFrame

TFrame 是在 Delphi 5 和 Delphi 6.0 中新加入的组件。TFrame 的一个功能可以进行功能封装，就像一个组件一样，它能够把一组功能包装在一起，实现这组功能的复用，甚至可以把一个应用程序封住进一个 TFrame，然后在其他的地方引用它；TFrame 的另一个功能是，它提供了一种拆分和组合应用程序的途径，可以把一个比较大，而且有较独立模块的应用程序拆分成多个 Frame，然后分发给不同的人，每个人以 TFrame 的形式进行独立开发，最后可以把所有这些 Frame 组合到一起，作为一个整体的应用程序。

首先，要创建一个 Frame，需要执行 File | New | Other 命令，或者直接单击 New 按钮来打开 New Items 对话框，然后在 New 选项卡上选择 Frame 进行创建，如图 5.2 所示。

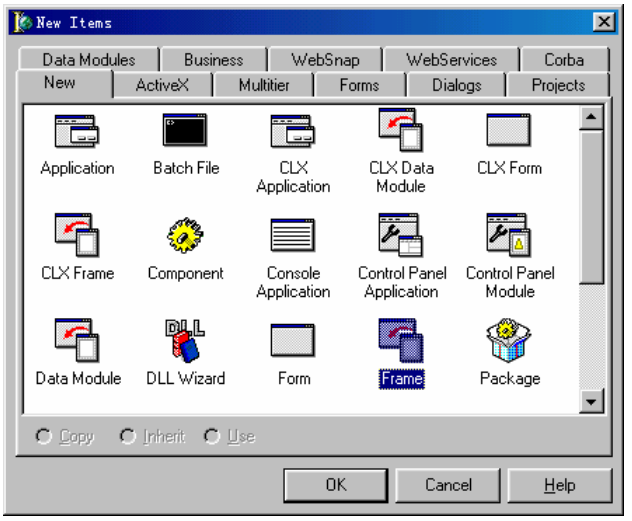

图 5.2　创建一个 Frame

这时，可以看到一个和普通 TForm 很相似的 TFrame，然后就可以在它上面进行模块的开发工作，例如放置一个 TButton，在它的单击事件中加入一个简单的显示信息框的动作。下图显示了这个 Frame 的界面，它命名为 MyFrame，并且为了显示明显，笔者将它的界面颜色设置为了红色，如图 5.3 所示。

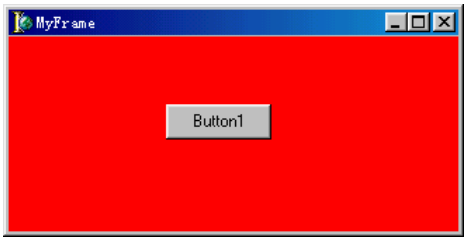

图 5.3　MyFrame 的显示界面

下面是 MyFrame 的代码实现，可以看到 MyFrame 声明为 Tframe，如下所示。

```
type
    TMyFrame = class(TFrame)
        Button1: TButton;
        procedure Button1Click(Sender: TObject);
    private
        { Private declarations }
    public
        { Public declarations }
    end;

implementation

{5R *.dfm}

procedure TMyFrame.Button1Click(Sender: TObject);
begin
    ShowMessage('I am from MyFrame');
end;
```

这样，一个简单的 Frame **就已经完成了。用户可以在任何一个普通的应用程序中引用它，不过必须要保证** MyFrame **单元已经加在了当前项目中，这样在向一个** Form **上放置一个** TFrame **组件时，当前项目才能够找到** MyFrame**，如图** 5.4 **所示。**

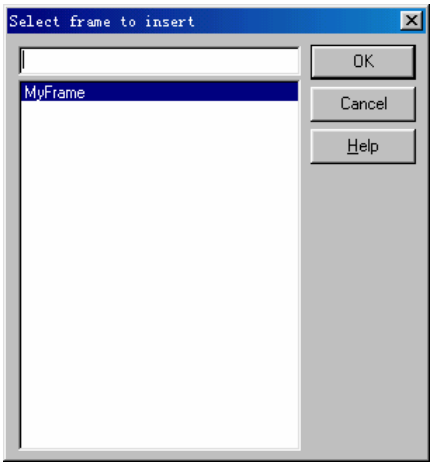

图 5.4　**选择一个** Frame **插入到项目中**

选择 MyFrame**，然后单击** OK **按钮以后，可以看到** MyFrame **已经嵌进了当前的** Form **中了，如图** 5.5 **所示。现在，用户完全可以把** MyFrame **当做一个独立的组件来使用，例如可以从外部访问它内部的各种子组件及其属性，下面的代码将改变** MyFrame **中** Button1 **的** Caption **属性，改变以后的显示如图** 5.6 **所示。**

```
type
```

```
    TForm1 = class(TForm)
        MyFrame1: TMyFrame;
        btnChangeButtonCaption: TButton;
        procedure btnChangeButtonCaptionClick(Sender: TObject);
    private
        { Private declarations }
    public
        { Public declarations }
    end;

var
    Form1: TForm1;

implementation

{5R *.dfm}

procedure TForm1.btnChangeButtonCaptionClick(Sender: TObject);
begin
    MyFrame1.Button1.Caption := 'Changed';
end;
```

图 5.5　Form 中插入 MyFrame

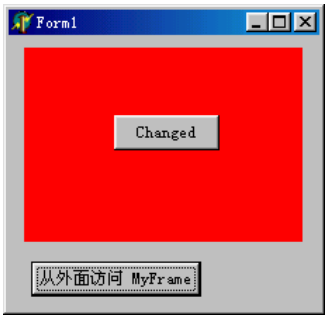

图 5.6　改变 MyFrame1 中的按钮

读者已经看到了，上面的示例中这样声明了 MyFrame1: MyFrame，**也就是说，**MyFrame **已经是作为一个整体来使用了。**

上面只是一个比较简单的 TFrame **的示例，它旨在介绍** TFrame **能够实现什么样的功能，以及如何使用它。它更加强大的功能还需要用户在开发过程中慢慢体验和积累。**

5.1.2　TMainMenu 和 TPopMenu 菜单

大多数应用程序都具有菜单。如果用户想开发一个高质量的软件，那么必须掌握菜单组件的使用，本节将介绍 TMainMenu **主菜单与** TPopMenu **右键弹出菜单。尽管主菜单与弹**

出菜单有许多差异，但它们与其它标准 Windows 控件仍然有许多共同之处。本节主要介绍这两类菜单的一些共同点。

在一个 Form 上放置一个 TMainMenu 或者 TPopMenu 组件，然后双击这个组件，就可以进入菜单编辑器中，这里可以对菜单中各项命令进行设置，如图 5.7 所示。

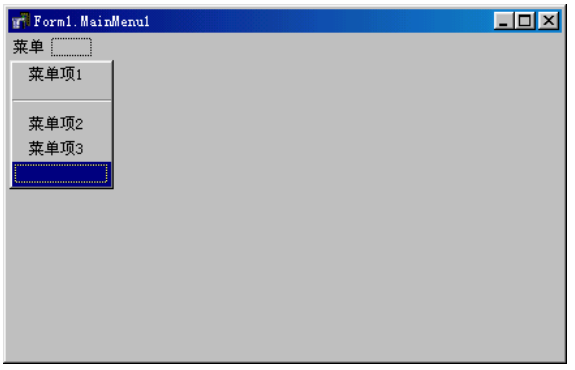

图 5.7 菜单编辑器

可以在对象监视器中设置命令的 Caption 属性，它将显示在菜单上，就像“文件”、“编辑”等一样，一般情况下需要把命令的缺省名改为一个更有意义的名字，也可以改变命令的其它属性。

1．菜单分隔符

大多数 Windows 菜单都用若干条下陷的水平直线把命令分为多组，称这些直线为 break（菜单分隔符）。Delphi 的分隔符有好几种风格。可以在用户自己的菜单中使用菜单分隔符，方法是创建一个的命令，设置它的 Caption 属性为短横线“-”。当程序运行时这个命令的标题就显示为一条水平直线。

类似的，可以在对象监视器中设置命令的 Break 属性把命令分成好几列。Break 属性有 3 个选项：mbNone、mbBreak 和 mbBarBreak。mbNone 是默认选项，它表示不创建菜单分隔，mbBreak 选项将把菜单分隔成多列，但列与列之间没有垂直分隔棒。mbBarBreak 选项与 mbBreak 选项类似，只是在列与列之间显示了一条垂直分隔棒。如果设置了一个命令的 Break 属性，该命令将会另起一列。下图为设置成 mbBarBreak 的情况，其中，设置了命令（即菜单项）1，2，3 的 Break 属性为 mbBarBreak，而命令 4，5，6 的 Break 属性为 mbNone，如图 5.8 所示。

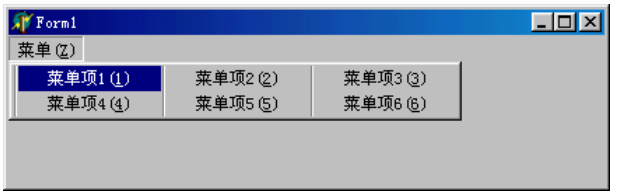

图 5.8 将 Break 属性设置成 mbBrBreak

2．级联子菜单

Cascading Sub-Menu（级联子菜单）是 Windows 菜单的一个普遍特征。Delphi 同样可以创建级联子菜单。如果用户想在主菜单或弹出菜单中创建级联子菜单，那么需要这样做：首先右击需要创建级联子菜单的命令，然后从弹出的快捷菜单中选择 Create Submenu 命令之后，会看到新的插入点移到了命令的右边。运行的时候，Delphi 会在具有级联子菜单的命令上放置一个三角形起到指示作用。如图 5.9 所示。

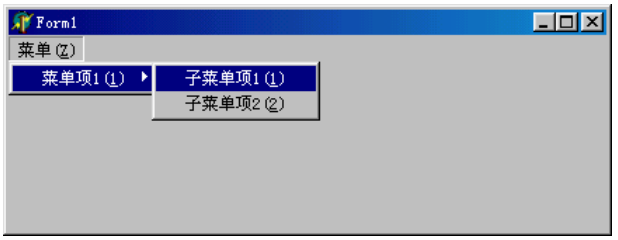

图 5.9　级联子菜单

3．复选命令

Delphi 的菜单命令有一个 Checked 属性，当命令的 Checked 属性设为 True 时，这个命令的旁边会出现一个复选标记。如果 Checked 属性的值为 False，那么命令旁边不出现复选标记，如图 5.10 所示。

图 5.10　命令的 Checked 属性

命令（即菜单项）2 和 3 前面都标记了复选标记（见图 5.10），但是有时候几个命令是互斥的，也就是说，任何时候都只允许其中的一个命令被选中，这时候，需要对命令的 RadioItem 和 GroupIndex 这两个属性进行设置。为实现菜单的这种行为，把这些互斥命令的 RadioItem 属性设为 True，而 GroupIndex 属性都设为一个相同的值，以此将这几个互斥命令归为一组，然后把其中一个命令的 Checked 属性设为 True，执行菜单时，选中的命令左边将出现一个圆点而不是复选标记。因为这个组中只允许一个命令被选中，所以当另一个命令的 Checked 属性切换为 True 时，圆点就出现在这个命令上，而原来被选中的那个命令及其他命令的 Checked 属性都将设为 False。

4．命令的快捷键

许多应用程序允许用户使用键盘选择菜单命令，实际上，用键盘选择命令是非常方便的，例如，许多 Windows 应用程序允许使用 Ctrl+S 键保存活动文件，称这些组合键为 shortcut key（快捷键）。可以在对象监视器中设置命令的快捷键，也可以在运行时用 Delphi 提供的函数设置命令的 ShortCut 属性提供快捷键方式。

Windows 还提供了另一种使用快捷键的方法。大多数 Windows 命令都有一个带下划线的字母，用户同时按下 Alt 键与这个字母键可以选择相应的命令。建立这种快捷键的方法是：设置命令的 caption 属性时，在某个字母前面加上&。例如一个命令的标题为“ Edit ”，使用 Alt+E 键就可以选择该命令。建立这个快捷键的方法是把命令的 caption 属性设置为&Edit。

5．弹出菜单

上面介绍的菜单设置方法对于弹出菜单 TPopMenu 都是适用的。但是要注意，放置在窗体上的弹出菜单不会自动关联一个特定的组件。在 Delphi 中，所有的可视组件都具有 PopupMenu 属性，设置这个属性可以把该组件与一个弹出菜单关联。这样，当用户右击这个组件时就会显示这个弹出菜单。

5.1.3 TLabel、TEdit 和 TlabeledEdit 组件

TLabel 组件用来在窗体上显示一行文字，用户是不能直接在 Lable 上修改这些文字的，它只用作显示。TEdit 组件也可以在窗体上显示一行文字，但它与 TLable 不同之处在于，TEdit 组件可以允许用户在它上面进行编辑修改。

图 5.11 显示了这两种组件在窗体上的样子。

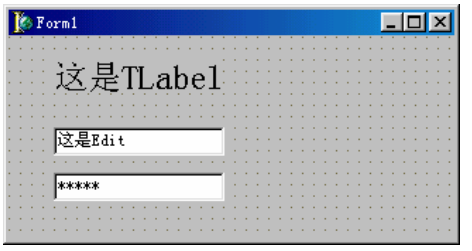

图 5.11 TLabel 和 TEdit 组件

其中，第一个 TEdit 没有做任何设置，第二个 TEdit 的 PasswordChar 属性设置为“ * ”，它的作用是将 Edit 中所有的可见字符用“ * ”字符来替代，也就是通常用来输入密码的形式，它并不代表 Edit 中实际的文字变成了“ * ”，而只是表面上被它替代了。用户同样可以将它设置为其他的字符，例如“ # ”等。

在实际应用中，TLabel 和 TEdit 常常要一起使用，而 Delphi 6.0 提供了一个新的组件 TLabeledEdit，它实际上就是组合了 TLabel 和 TEdit 两个组件在一起，该组件包含在 Additional 组件组里。

5.1.4 TCheckBox 和 TradioButton 组件

TCheckBox 和 TradioButton 组件的区别在于 CheckBox 可以多选，而 RadioButton 只能单选，如图 5.12 所示。

放置在 Form 上的所有的 RadioButton 都将是互斥的（见图 5.12），即它们当中只能有一个被选中。但是实际情况中，有时候会有几组 RadioButton 分别互斥，但组和组之间不互斥，这时候，可以运用 TGroupBox 组件或者 TPanel 组件，放置在不同的 TGroupBox 或者 TPanel 上的 RadioButton 之间是相互独立的，如图 5.13 所示。

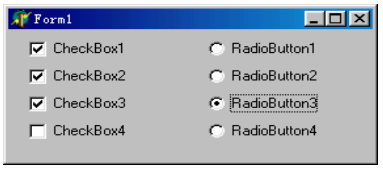

图 5.12　TCheckBox 和 TRadioBox 的区别

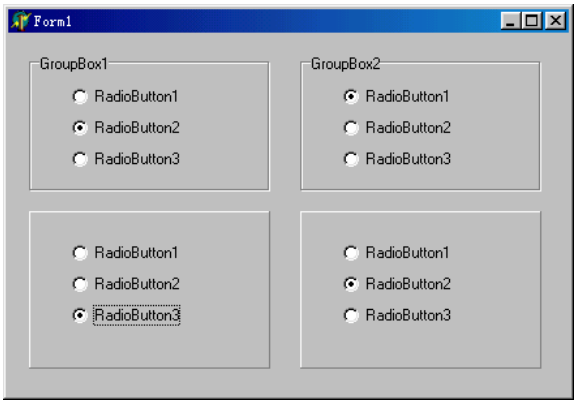

图 5.13　多组 RadioButton

可以看到，GroupBox1 和 GroupBox2 以及另外两个 Panel 上的 RadioButton 组之间并没有关联，它们可以是内部互斥的。GroupBox 和 Panel 在这里实际上起到了分组的作用。

5.1.5　TListBox 和 TcomboBox 组件

习惯上称 TListBox 为列表框、TComboBox 为下拉列表框。图 5.14 显示了它们的样子。

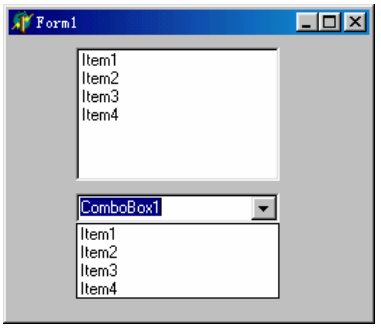

图 5.14　TListBox 和 TComboBox

TListBox 将所有的可选项都列在一个列表框中，而 TComboBox 则放在一个下拉列表框中。这两个组件都有一个 Items: TStrings 属性，其中定义了所有的可选项，如图 5.15 所示：

TComboBox 组件有一个 Style 属性，它定义了下拉框的特性，其中包括以下选项：

- csDropDown：下拉列表框可以进行编辑，并且下拉项的高度全部相同。

- csDropDownList：下拉列表框不能进行编辑，只能从下拉选项中选取，下拉项的高度全部相同。

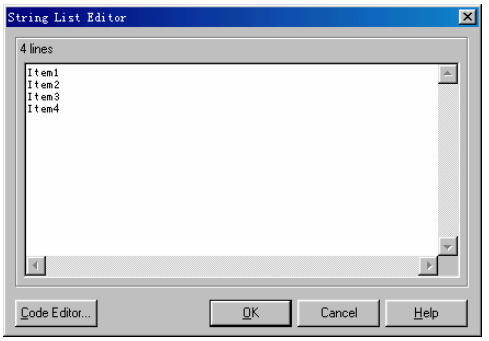

图 5.15 设置 TListBox 和 TComboBox 的 Items 属性

- csOwnerDrawFixed：下拉列表框不能进行编辑，只能从下拉选项中选取，但是下拉项的高度可以由 ItemHeight 属性来指定。
- csOwnerDrawVariable：下拉列表框不能进行编辑，只能从下拉选项中选取，其中每个下拉项的高度可以不同。

在实际应用中，TComboBox 比 TListBox 应用的要广泛一些，因为它比 TListBox 占用的地方小很多。

5.1.6 Tpanel 组件

TPanel 是一个比较简单的组件，但是它在 Delphi 应用程序中却起着非常重要的作用，尤其是在界面设计方面，它主要起到了分组的作用。在一个 Panel 上相对设计好了的组件可以随 Panel 一起移动，而它们的相对位置不会变。结合 Additional 组件组中的 TSplitter 组件，它们组成了大部分应用程序常用的界面，如图 5.16 所示。

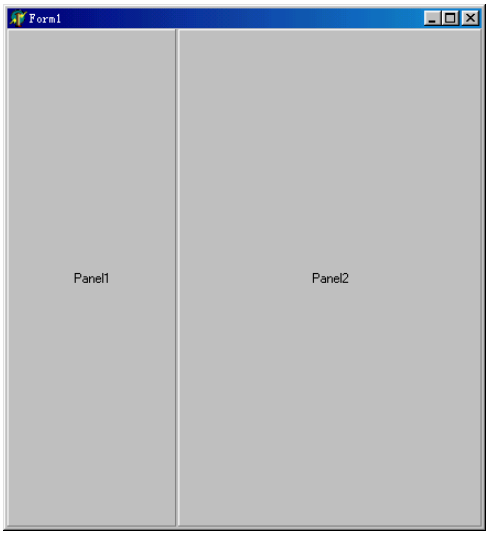

图 5.16 Panel 与 Splitter 结合使用

需要设置左边的 Panel1 的 Align 特性为 alLeft，而右边的 Panel2 的 Aligan 特性为 alClient，它的作用是 Panel1 以左靠齐，而 Panel2 填充空白的区域，中间的 Splitter 的作用是控制左右两个 Panel 的大小，就像 Windows 的资源浏览器的效果一样。

5.1.7　TactionList 组件

TActionList 在 Delphi 中是一个功能非常强大的组件，它的作用在于可以集中的创建和管理应用程序中会用到的事件、方法等，即 Action。而且，更重要的是，Delphi 中的大多数控件，例如 TMenuItem、TButton 等，都可以直接通过它们的 Action 特性关联在某一个 Action 上，不仅仅事件本身关联起来了，而且 Action 的各项设置，例如 Caption、Hint、ImageIndex 等都能够直接从 Action 上读取。而且，一个 Action 可以关联在多个控件上，例如一个菜单上和一个按钮上都可以关联一个 Action，当程序需要设置该事件的可操作性时，用户只需要对 Action 进行设置，而所有关联在该 Action 上的控件都会同步相应。例如，要把一个事件关掉，在程序中不允许启动该事件，那么只需要将 Action 的 Enabled 属性设置为 False 即可，关联该 Action 的按钮和菜单都将自动置为不可操作的状态。

另外，在程序别的地方同样可以把 Action 当做一个函数、方法来调用，只要执行 Action 的 Execute 方法即可。图 5.17 所示为 TActionList 中的 Action 列表。

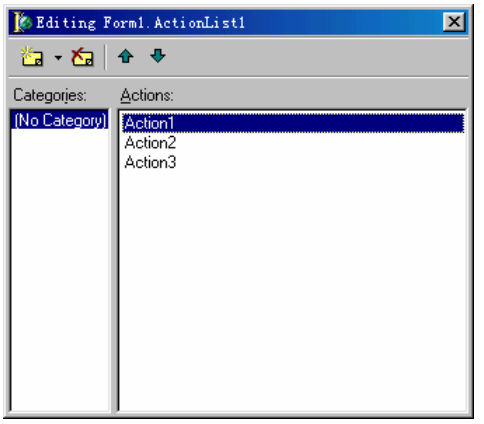

图 5.17　TactionList 中的 Action 列表

图中的 ActionList1 中包含有 Action1、Action2、Action3 3 个 Action，通过对象监视器，可以设置这 3 个 Action 的 Caption、Hint、Checked、ImageIndex、ShortCut 等特性，它们都可以传递给关联 Action 的控件，例如按钮等。

使用 TActionList 可以使用户的应用程序更加清晰，更加有条理，更加便于维护。

5.2　Additional 组件组

Additional 组件组中包含了 TBitButton、TSpeedButton、TMaskEdit、TStringGrid、TDrawGrid、TImage、TShape、TBevel、TScrollBox、TCheckListBox、TSplitter、TStaticText、TControlBar、TApplicationEvent、TValueListEditor、TLabeledEdit、TColorBox、TChart、

TActionManager、TActionMainMenuBar、TActionToolBar、TCustomizeDlg 等，下图中第一行为 Windows 应用程序开发中的 Additional 组件组，第二行为 Linux 应用程序开发的 Additional 组件组，如图 5.18 所示。

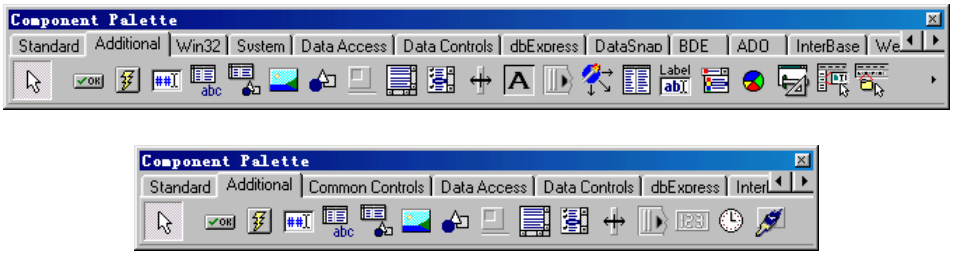

图 5.18 Additional 组件组

5.2.1 TmaskEdit 组件

TmaskEdit 组件实际上只是一个附加了 Mask 功能的 TEdit，它能够按照特定的格式输入编辑 TEdit 内容，Mask 格式可以从系统预置的模板中选择，如图 5.19 所示。可以双击 TMaskEdit 的 EditMask 属性来打开这个对话框。

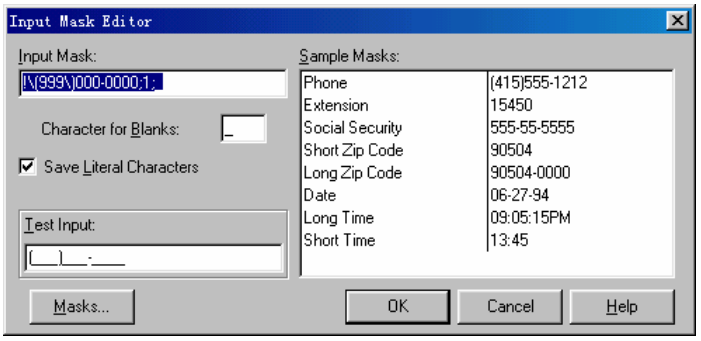

图 5.19 Mask 编辑器

图 5.20 是选择 Phone 掩模以后，MaskEdit 在运行时的显示。

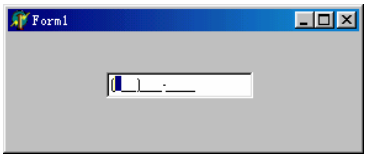

图 5.20 运行时的 MaskEdit

可以看到，Edit 中固定显示了几个特殊字符，它们是 Phone 掩模中的固定格式，在编辑它的时候，这些固定字符是不能被修改和删除的，只能在要求填写的字符位置上编辑。

5.2.2 Timage 组件

TImage 组件用来显示一幅图像，它支持 jpg、jpeg、bmp、ico、emf 和 wmf 等格式的

图像文件，双击 Picture属性，可以看到一个选择图像文件的对话框，如图 5.21 所示。这里可以 Load（选择）一个图像文件，或者 Save（保存）当前图像到一个文件。

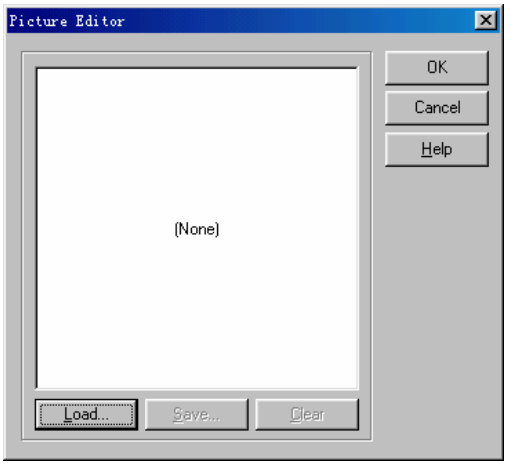

图 5.21　图像编辑器

在程序运行期间，如果需要动态地 Load 一个图像文件或者清空 TImage 图像，TImage 提供了一些函数方法来完成这些功能。其中，Load 一个图像文件，如下所示：

```
Image1.Picture.LoadFromFile('c:\picfile.bmp');
```

TImage 能够根据要读取的文件的扩展名（bmp、jpg、ico 等），生成一个不同的 TGraphic 类来显示图像，如果 TImage 无法识别扩展名，那么该方法将触发一个异常。

除了从一个文件中读取图像以外，TImage 还提供了一个 LoadFromStream 方法用来从数据流中读取图像。

TImage 在一般的应用程序中有很广泛的应用，最常用的是应用程序的启动界面。为了在启动界面中按照一个图像的大小尺寸来显示它，需要设置 TImage 的 AutoSize 属性为 True，它会使 TImage 的大小尺寸完全和当前图像的大小一样。

同样，设置启动 Form 的 AutoSize 属性为 True，它的作用是将 Form 的大小设置为和 TImage 的大小完全一样。最后设置 Form 的 BorderStyle 为 bsNone，这样，将不会看到 Form 的边缘，而显示在屏幕上的只有图像本身，这就是一个由 TImage 组成的启动界面。

5.2.3　Tsplitter 组件

TSplitter 组件的作用是产生一个可变大小的区域的界线，该界线可以自由调整两个区域的大小，就像在 Windows 的资源管理器中看到的一样，左边的目录树和右边的文件列表可以相互调整大小。

在 Delphi 中，TSplitter 一般和 TPanel 一起布置窗体界面，这在 TPanel 一节已经介绍过了，图中两个 TPanel 中间的“缝”就是一个 Tsplitter（见图 5.16）。

5.2.4　TapplicationEvents 组件

TApplicationEvents 组件的作用是为程序员提供一个处理当前应用程序 TApplication 的

事件的途径，它能够截获当前 Application 的所有事件，例如 OnActivate、OnDeactivate、OnException 和 OnIdle 等，只要将一个 TApplicationEvents 组件放到了一个 Form 上，那么 Application 将会把它所有的事件都传递给这个组件，也就是说，TApplicationEvents 组件得到的事件和 Application 的事件是完全一样的。

这样，如果想处理当前 Application 的某些事件，只需要简单地处理 TApplicationEvents 提供的相应的事件就可以了。例如，要处理 Application 在空闲时的事件，当然可以按照传统的方式来处理，即处理 Application 的 OnIdle 事件，但是使用 TApplicationEvents 组件，只需要简单地处理 TApplicationEvents 组件的 OnIdle 事件就可以了。在 TApplicationEvents 组件的对象监视器 Events 选项卡上，可以看到它所能够处理的这些 Application 事件，如图 5.22 所示。

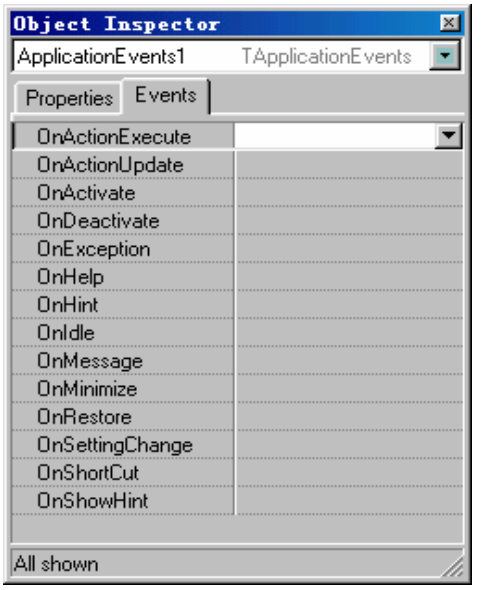

图 5.22 TApplicationEvents 组件的事件

5.2.5 TActionManager 组件

Delphi 6.0 提供了一组全新的 Action 组件，它们包括了 4 个组件：TActionManager、TActionMainMenuBar、TActionToolBar 和 TCustomizeDlg。与 TActionList 类似，TActionManager 能够让用户创建一系列的 Action 事件，但除此以外，它借助于另外两个组件 TActionMainMenuBar 和 TActionToolBar，提供了创建窗体主菜单和工具栏的功能，而且能够直接将创建的 Action 拖放到主菜单或者工具栏上。

图 5.23 是 TActionManager 编辑器中用来管理 Action 的界面，可以通过双击这个组件来打开这个编辑器；图 5.24 为编辑器中管理工具栏的界面，单击 New 按钮，系统将在默认位置自动创建一个 TActionToolBar，并且该 TActionToolBar 的 ActionManager 特性自动设置为了当前的 TActionManager。

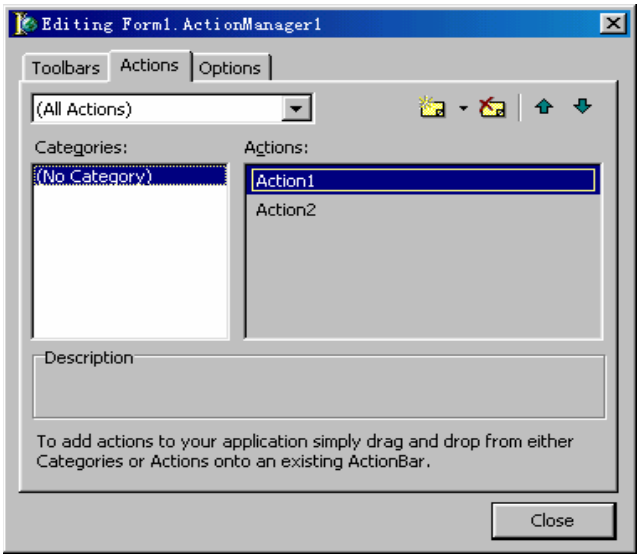

图 5.23　TActionManager 编辑器的 Action 选项卡

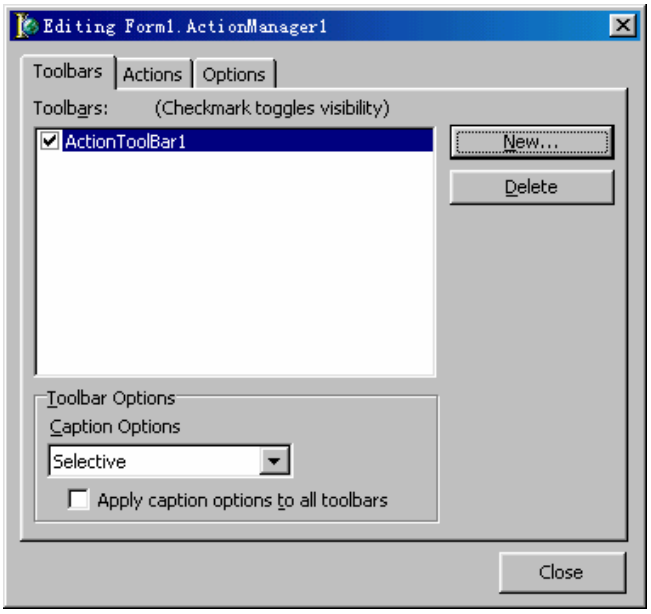

图 5.24　TActionManager 编辑器的 Toolbars 选项卡

下图显示了如何将一个 Action 从 TActionManager 编辑器中拖放到一个 TActionToolBar 和 TactionMainMenuBar 中，如图 5.25 所示。

读者已经看到了，TActionMainMenuBar 相当于 Standard 组件组中的 TMainMenu 组件，它创建了当前窗体的主菜单，并且能够接受来自 TActionManager 组件中的 Action 停泊在上面，而 TActionToolBar 就是能够停泊 Action 的工具栏。

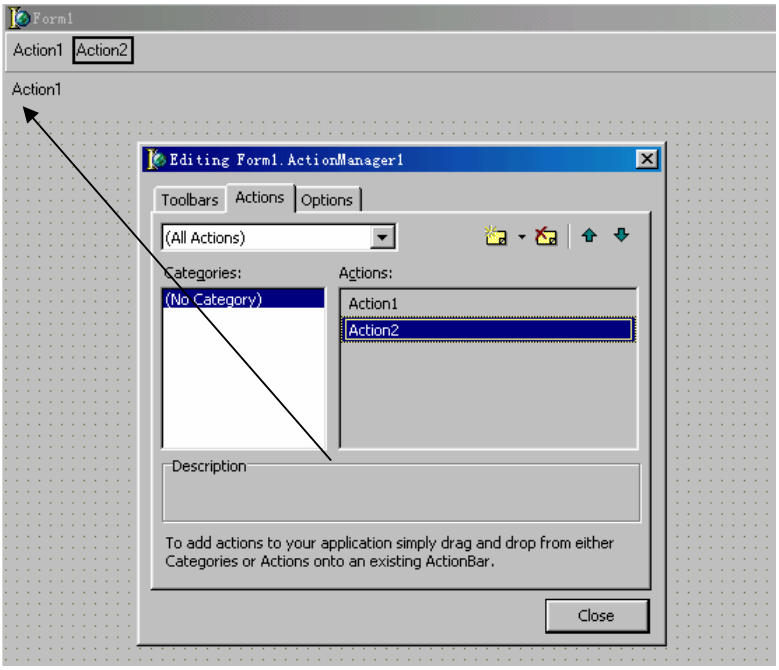

图 5.25 拖放 Action 到 TActionMainMenuBar

5.3 Win32 组件组

Win32 组件组提供了在 Win32 下编程应用非常广泛的一组组件，其中包括 TPageControl、TImageList 等，而在 Linux 应用程序开发中，相应的这些组件大部分列在了 Common Controls 组件组中。图 5.26 中，分别显示了 Windows 和 Linux 应用程序开发中这些组件在组件选项板中的样子。

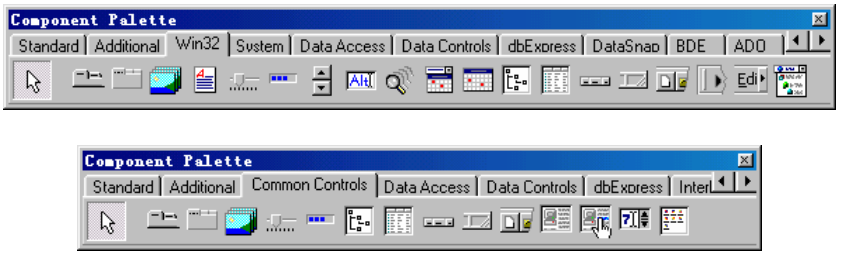

图 5.26 Win32 和 Common Controls 组件组

5.3.1 TpageControl 组件

在应用中，经常会出现信息过多以致窗体无法容纳的情况。在 Delphi 6.0 中，可以使用 PageControl 组件来解决这一问题。PageControl 能够将信息重叠显示在窗体的一个区域中，通过 TabSheet 选项卡来切换当前的选项卡。

首先，放置一个 PageControl 组件到一个窗体 Form 上，这时候可以对该 PageControl 进行一些属性设置，例如 Align、Style 等。右击该组件，单击弹出菜单中的 New Page 命令，系统将在当前 PageControl 上创建一个新的选项卡 TabSheet1，可以设置该 TabSheet 的显示标题 Caption 和名称 Name。然后，在当前的 TabSheet1 选项卡上，放置一系列的需要用到的控件或者别的东西。例如，现在在第一个 TabSheet1 选项卡上放置一个 GroupBox，在其中放入几个 RadioButton，如图 5.27 所示。

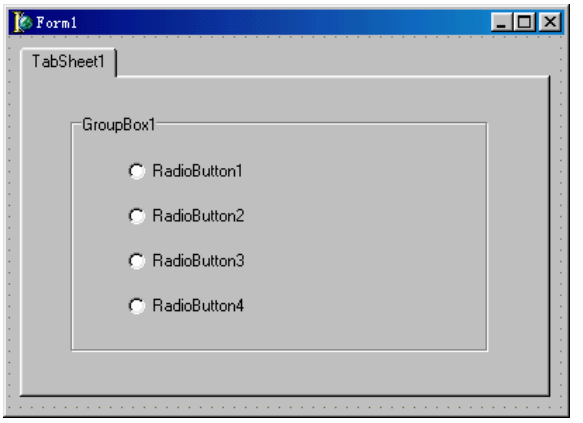

图 5.27　PageControl 中的 TabSheet1 选项卡

然后，仍然按照上面的方法，通过右键菜单中的 New Page 命令来加入第二个选项卡 TabSheet2，在这一选项卡上加入其他的一些控件，如图 5.28 所示。

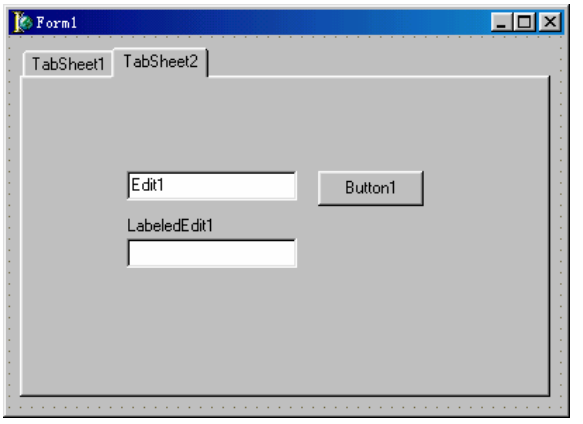

图 5.28　PageControl 的 TabSheet2 选项卡

在 PageControl 的右键菜单中，除了上面已经用到的 New Page 命令以外，还会看到另外 3 个命令，它们分别是 Next Page、Previous Page 和 Delete Page，它们提供了查看前一选项卡、后一选项卡和删除当前选项卡的功能。

下面简单介绍几个 TPageControl 组件中的特性。

1．ActivePage 和 ActivePageIndex

ActivePage 特性表示 PageControl 当前显示的 TabSheet 选项卡，可以在对象监视器中静态地设置该特性，也可以在运行期间通过代码来实现选项卡的切换，例如：

```
PageControl1.ActivePage := TabSheet2;
```

或者可以通过 ActivePageIndex 特性来定位，如下所示。但该方法适用于不知道选项卡的名称的情况：

```
PageControl1.ActivePageIndex := 1;
```

2．MultiLine

MultiLine 特性用来设置当 PageControl 中的 TabSheet 选项卡很多的时候，所有这些选项卡应该如何显示。如果该特性设置为 False（默认情况），那么所有的 TabSheet 选项卡将显示在一行上，这时候可以看到在 PageControl 的右上角出现了两个小按钮，用来向左向右移动 TabSheet，如下图中上面的 PageControl 的显示。如果 MultiLine 特性为 True，那么所有的 TabSheet 将自动多行显示在 PageControl 中，如下图中下面的 PageControl 的显示，这里 9 个 TabSheet 排列在两行上，而且每个 TabSheet 按钮的宽度都会自动调整，保证两行的宽度相同，如图 5.29 所示。

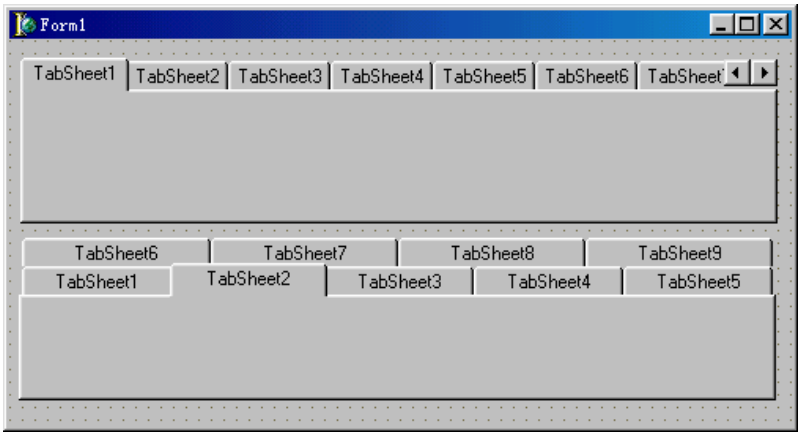

图 5.29　MultiLine 特性设置

3．Style

Style 特性用来设置 PageControl 组件的显示风格，它包括 tsTabs、tsButtons、tsFlatButtons 3 种风格。图 5.30 显示了这 3 种风格的样子。

4．TabPosition

TabPosition 特性有 4 个值可以选择：tpTop、tpBottom、tpLeft、tpRight，它们分别用来设置 PageControl 的选项卡按钮显示在上边、下边、左边和右边。可以根据窗体的实际情况来设置按钮的位置。

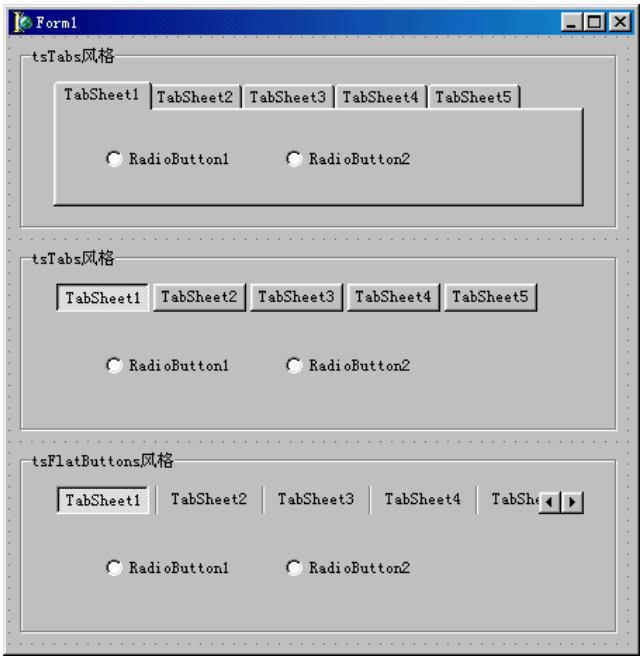

图 5.30　PageControl 的 Style 特性设置

5.3.2　TimageList 组件

TimageList 组件在 Delphi 应用程序中的应用非常广泛，几乎每一种控件都提供了对 ImageList 的支持，也就是说，都可以显示图标。TImageList 提供了一种集中管理图标的方法，控件可以通过 ImageList 和 ImageList 中图标的 Index 来指定一个特定的图标，而且，不同的控件可以共享一个图标，这大大避免了为每一个控件都指定图标文件所带来的麻烦。

图 5.31 为双击 ImageList 打开的管理图标的对话框。

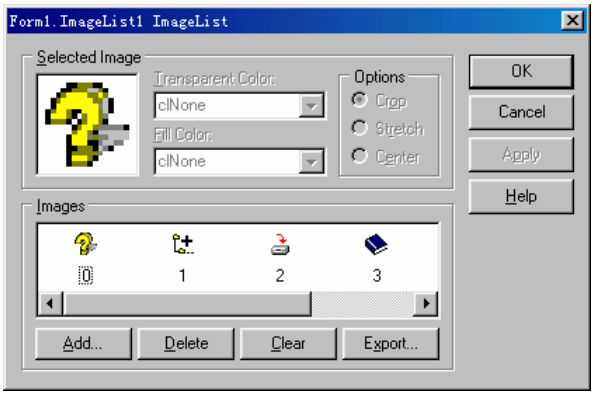

图 5.31　ImageList 图标管理器

在这里，可以添加一个图标到 ImageList 中，或者删除一个图标、清空全部图标，或者还可以将 ImageList 中的一个图标导出为一个图标文件。

下面以一个 PageControl 为例来说明如何使用 ImageList。

首先，指定 PageControl 的 ImageList 特性为当前窗体上的一个 ImageList 组件，然后选定这个 PageControl 中的一个 TabSheet，这时可以在对象监视器中看到这个 TabSheet 有一个 ImageIndex 特性，打开这个属性下拉列表框，它将显示出 PageControl 所关联的 ImageList 中的所有图标，如图 5.32 所示。在这里，可以很清楚地指定一个图标，ImageIndex 特性将是所选定图标的在 ImageList 中的索引 Index 值。

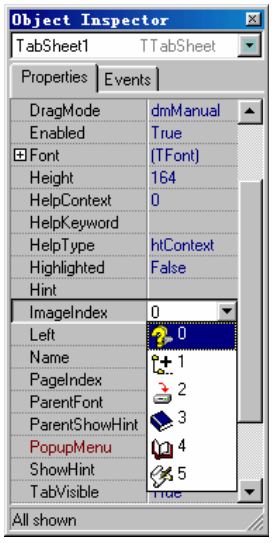

图 5.32 ImageIndex 下拉列表框

很多控件都是这样使用 ImageList 的。这里，需要特别地讲到 TActionList 组件对 ImageList 的使用。TActionList 组件同样可以指定一个 ImageList 组件，对于 TActionList 中的每一个 Action，同样可以通过一个 ImageIndex 来指定 ImageList 中的一个图标。所有的这些设置完成以后，当将 Action 关联到某一个控件上去的时候，如果该控件支持 ImageList，那么会看到它的 ImageIndex 已经从 Action 的 ImageIndex 关联下来了。无论该 Action 应用到什么地方，只要能够显示图标，那么指定好的图标都将自动引用在 Action 中的设置图标。

图 5.33 为一个 PageControl 关联了 ImageIndex 后的显示。

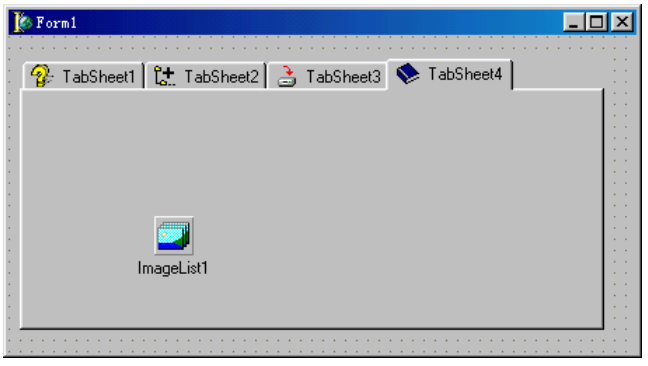

图 5.33 PageControl 关联 ImageList

5.3.3　TstatusBar 组件

大部分 Windows 应用程序中，主界面最下面总会有一个状态栏，它用来显示系统当前的各种状态，一般是用文字来显示，Win32 组件组中的 TStatusBar 组件就是 Delphi 为实现此功能而提供的一个组件。当然，TStatusBar 组件在显示形式方面有比较大的局限性，它仅仅支持显示文字，或者 Owner Draw（自己绘画），而常常会用到的进度条功能则没有提供出来，如果通过自己绘画的方法来实现则相当的麻烦，将会浪费不必要的开发时间。其实，完全可以到相关的网站上查找相应的组件，例如笔者知道一个 TDfsStatusBar 组件，它是在 TStatusBar 的基础上继承下来的一个组件，除了 TStatusBar 本身所带有的功能以外，增加了很多很有用的功能，例如进度条显示、小键盘数字键状态、插入/改写状态等，使用户在使用时能够非常方便地应用各种状态。

当放置一个 TStatusBar 组件到窗体上时，它将默认地靠在窗体的最下边，如图 5.34 所示。

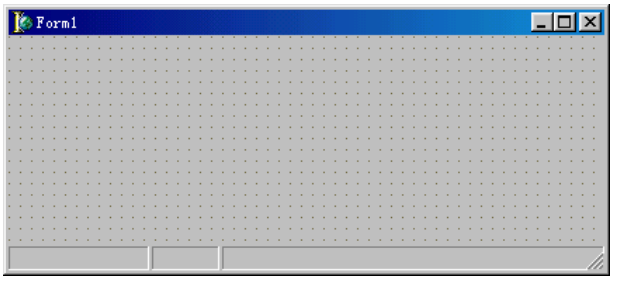

图 5.34　TStatusBar 组件

双击 TStatusBar 组件打开状态栏 TStatusPanel 编辑器，如图 5.35 所示。其中可以对每一个 TStatusPanel 进行设置，例如显示文字 Text、宽度 Width 等。

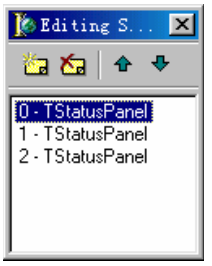

图 5.35　TStatusPanel 编辑器

5.3.4　TMonthCalendar 和 TdateTimePicker 组件

TMonthCalendar 和 TDateTimePicker 这两个组件都可以用来表示日前或者时间，其中前者按照月份来表示日期，后者可以通过下拉列表框来指定月份日期，就像 TMonthCalendar 一样，另外它还能用来指定一天中的时间。图 5.36 分别显示了这两个组件的样子，以及 TDateTimePicker 的两种显示类型（日期和时间），它是通过 TDateTimePicker 的 Kind 属性来区分的。

图 5.36 时间和日期组件

现在看到 TMonthCalendar 组件显示了当前月份的日历，其实，如果将这个组件的显示区域足够扩大，它将会同时显示出多个月份的日历，例如图 5.37 同时显示了六个月份的日历。

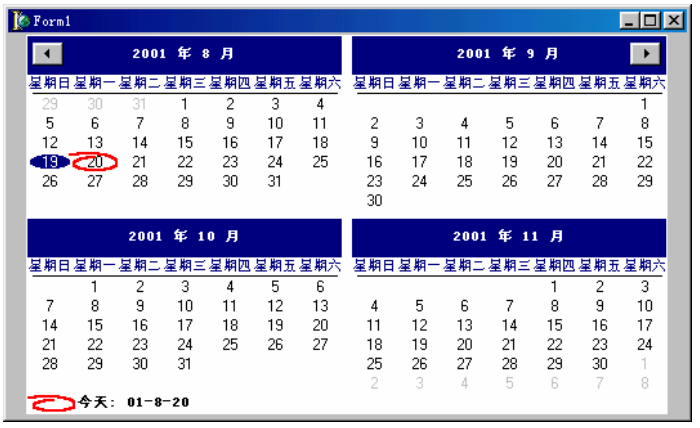

图 5.37 同时显示多个月份的 TMonthCalendar 组件

当应用程序中需要确定一个比较详细的时间，例如包括日期和时间，那么就需要同时使用 TDateTimePicker 的两种显示类型，例如上图中（见图 5.36），一个 TDateTimePicker 用来显示日期 2001/8/20，另一个用来显示时间 0:13:04，当然还可以调整它们两个来指定任何一个时间。

在程序中，这两个组件都提供了一个 Date 方法来取到组件指定的日期，另外 TdateTimePicker 组件还提供了 DateTime 和 Time 两个方法来取组件指定的日期和时间或者仅仅时间，它们分别是 TDate、TDateTime 和 TTime 类型。

5.3.5 TComboBoxEx 组件

TComboBoxEx 组件在 TComboBox 组件的基础上增加了很多特性，它使这一应用非常广泛的组件更加美观和好用。

首先，它增加了 ImageList 特性，它表示 TComboBoxEx 中的每一条 Item 都可以关联相应的图标。与 TComboBox 不同的是，它需要通过 ItemsEx 特性来编辑下拉列表框中的各个 Item，下图是这个 Item 编辑器的样子以及其中每条的属性，对其中的每一个 Item，都可

以在对象监视器中设定它的图标 Index 和 Caption，如图 5.38 所示。

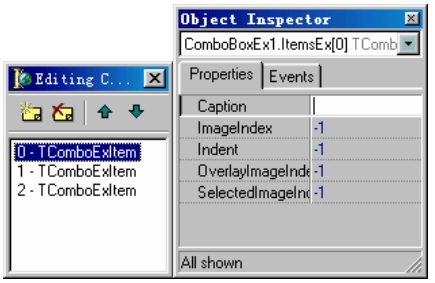

图 5.38　TComboBoxEx 中的 Item

另外，对下拉列表框中的每一条 Item，都可以设置它的显示高度，以及是否区分大小写等。

5.4　Dialogs 组件组

Dialogs 组件组中提供了 Windows 系统中的各种标准信息对话框，例如打开文件、保存文件、打开图片、选择颜色、打印机预览等。它们在 Windows 应用程序开发中非常重要，尤其是当用户必须要与文件或者输入输出设备打交道的时候。图 5.39 是这个组件组的样子。

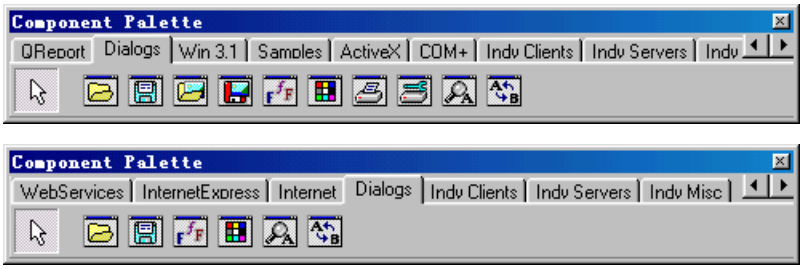

图 5.39　Dialogs 组件组

大部分 Dialog 组件都具有下面几个属性：

- Title

Title 是显示在对话框上的标题，例如图 5.40 中的“选择一个文件”。

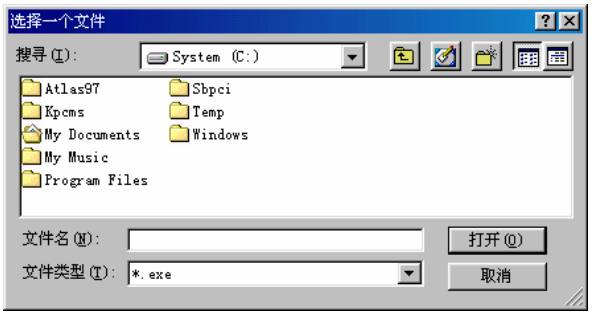

图 5.40　一个 Dialog 对话框

注意：在任何 dialog 组件上右键单击鼠标，可以在弹出的快捷菜单中看到第一个命令为 Test Dialog，运行这个命令，系统将演示当前这个 Dialog 组件在执行 Execute 方法后的运行界面。通过此命令可以快速地检测组件的属性设置是否正确，而不必运行整个程序来检测。

- InitialDir

InitialDir **属性用来指定 Dialog 对话框在打开的时候初始的默认路径，如上图中的 InitialDir 就是“ C:\ ”（见图 5.40），当然可以在打开以后重新更改这个路径。**

- Filter

Filter **为 Dialog 组件的文件过滤器，它将使对话框中只显示特定类型的文件。上图中 Filter 设置为“ *.exe|*.exe ”（见图 5.40），当然不必自己来编写这个属性，而可以通过一个设置对话框来完成这个工作，如图 5.41 所示。**

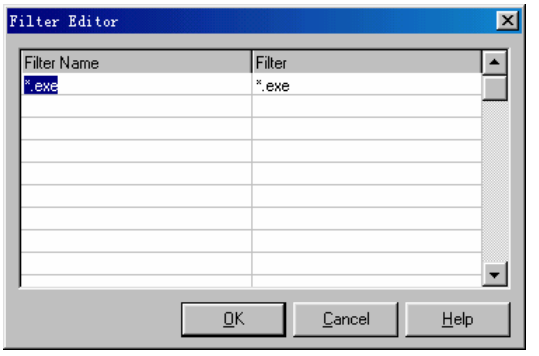

图 5.41 Filter 设置对话框

- FileName

FileName **属性是 Dialog 组件在运行之后返回的文件名，其中包括了文件的绝对路径以及文件名称。**

- DefaultExt

DefaultExt **属性用于指定 Dialog 组件的默认扩展名，例如当在 SaveDialog 对话框中输入 MyFile 时，如果该组件的 DefaultExt 为 exe，那么文件将按照“ MyFile.Exe ”的形式进行存储，而如果 DefaultExt 为空，那么文件将存储为 MyFile。**

Dialogs **组件组中的其他组件，例如 TFontDialog、TColorDialog、TPrintDialog、TFindDialog 以及 TReplaceDialog 等，分别用于打开标准的字体对话框、颜色对话框、打印机对话框、查找对话框以及替代对话框等。用户可能经常在 Windows 的其他地方看到过这些对话框，这并不奇怪，因为它们实际上都是 Windows 系统的标准组件，任何地方都可以直接调用它们。**

TOpenPictureDialog **和 TSavePictureDialog 两个组件在标准的 TOpenDialog 和 Tsave-Dialog 基础上附加了预览图片的功能。**

5.5　本 章 小 结

本章中详细地介绍了几组在开发工作中常常会用到的组件，读者现在已经能够了解这些组件具备什么功能、能够完成什么工作，或者有什么样的界面显示，在此基础上，随着对这些组件的实际使用，读者会学到更多更深入的知识。

Delphi 本身自带了非常丰富的组件库，但更重要的是，无穷无尽的第三方组件是 Delphi 最具魅力的地方。要编写一个应用程序，用户往往根本不需要做或者很少做底层的东西，用户需要做的只是找到适当的组件，然后将它们规则地摆放在用户自己的应用程序中就可以了。如果遇到比较特殊的情况，也可能需要在现有组件的基础上进行功能或者界面扩展、进行组件的再加工，也就是继承，有关组件继承的知识，可以参考“ VCL 组件基础 ”和“ 编写自己的组件 ” 两章中的相关内容。

第 6 章　VCL 组件基础

自从 Broland 在 Trubo Pascal for Windows 中第一次引入了 Object Windows Library 即 OWL，传统的 Windows 编程得到了大大地简化。例如，用户再也不必写一个很长的 case 语句来捕获每一个消息，因为 OWL 已经代劳了。

从 Delphi 1.0 开始引入的 Visual Component Library 即 VCL，是 OWL 的延续。原则上 VCL 和 OWL 都是基于对象模型的，但实现的途径有所不同。

VCL 是专为 Delphi 的可视化开发环境设计的。用户不必写代码来建立窗口或对话框，VCL 中定义了许多 Component（组件），可以使用户可视化地设计应用程序。

在软件工程中，创建一个精巧实用的应用框架至今仍是最困难的任务之一。VCL 是迄今为止做得最好的，事实上，VCL 已经影响了其它语言中的基于组件的标准框架的建立。

对 VCL 的了解程度取决于如何使用它们。首先，必须认识到有两种 Delphi 的开发者：应用程序开发者和 VCL 组件编写者。应用程序开发者的任务是通过可视化开发环境来编写程序，他们通过 VCL 来建立 UI（用户界面）或者数据库的连接。而组件编写者的任务是创建新的组件来扩展已有的 VCL。

不管是开发应用程序还是编写组件，掌握 VCL 都是非常有必要的。应用程序开发者应该知道一个组件有哪些特性、方法和事件，最好还应该知道 VCL 组件的继承关系。组件编写者则需要更加深入地了解 VCL，诸如 VCL 如何处理消息、内部的通知、组件的拥有关系和父子关系、特性编辑器等。

本章将详细介绍 VCL 的各个方面，包括组件的继承关系、每一层的作用，以及组件的一些共有的特性、方法和事件等。

6.1　VCL 应用框架

问世以来，VCL 的基本结构变化很小。但是，随着 Delphi 的改进，VCL 也同样得到了改进，添加了一些新功能，但并没有改变它的基础模块。在介绍 VCL 的基础结构之前，本节先对面向对象的应用框架与基于组件的应用框架之间的区别做一个简单的探讨。

如果用户曾经作过一些 Windows 的开发工作，可能使用过某些应用框架。从头开始编写 Windows 程序是一件极其耗费时间而且不实际的作法。

面向对象库与基于组件库的区别在于对事物的看法不同。组件模型基于面向对象的规则，但组件并不会用任何新的语言特征来增加语言的面向对象的能力，相反，组件结构会在系统的对象上添加一些标准接口。

下面举一个组件的实例。如果要购买一个立体声音响组件，顾客不用关心组件的制造商，因为所有的立体声音响组件的制造都遵循一定的标准规范，它们都能相互兼容。这样，

顾客就可以随意挑选喜欢的组件，将这些组件及其功能都加入到自己的应用程序中。由于组件能自动知道如何与其它组件相互沟通，因此，不必担心它们之间会不兼容。计算机硬件也是由各个不同厂商的构件组成的，但是，只要这些构件正确遵守规定的总线标准，它们就能共同运转。

这也正是 VCL 赋予 Delphi 的优势。VCL 充当的角色就像是标准总线，所有的组件都能插进去，因为所有的组件归根到底都是由 TComponent 继承而来的，所以，它们之间必然存在一些所有组件都能支持的功能，例如将组件添加入组件列表中、在窗体和容器中组件之间相互交流的标准方式、设置组件的标准方式（通过对象监视器或属性编辑器或组件编辑器）以及保存组件状态（例如创建持久对象）的标准方式。

基于组件的结构是对面向对象结构的进一步改良。对结构化及模块化编程来说，面向对象的结构是一场革命，而基于组件的结构是建立在面向对象的语言基础之上的，它仅仅是一种改良活动，而不是去代替它。

6.2　组 件 简 介

组件是可视化编程的基础，应用程序的用户界面和其他功能就是由它构成的。对于应用程序开发者来说，可以从组件选项板上选择一些组件放到窗体上，然后通过修改组件的特性、建立事件句柄来满足应用程序的要求。从组件编写者的角度看，组件就是 Object Pascal 语言中的对象。其中，有些对象可以操纵 Windows，有些对象只是组件内部使用的。

不同的组件的复杂程度相差很多。有些组件非常简单，有些组件封装了精心设计的任务，组件所能做的事情几乎没有什么限制。有的组件就很简单，像 TLabel，有的组件就很复杂，它可以具有一个电子表格的所有功能。

Delphi 组件有 3 个非常好的优势：

- Delphi 组件是在 Delphi 环境中开发出来的本地组件。这意味着可以在标准的 Delphi 程序内部编写和调试。
- Delphi 组件是完全面向对象的，可以通过创建派生对象的方法很容易地改变和提供现有组件的功能。
- Delphi 组件小、快而且轻，它们可以直接连接到执行程序中去。

与 ActiveX 控件相比，学习和编写自己的 Delphi 组件要比理解和编写 ActiveX 控件容易的多，而且 ActiveX 控件显得很笨重、缓慢，更重要的是，ActiveX 控件不能直接连接到程序中去而必须单独发布。

当然，这并不是说 ActiveX 控件技术不重要，只不过 Delphi 组件创建起来相对容易一些，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 组件，包括从串行通信到数据库连接、到多媒体等方面。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。

掌握 VCL 的关键是知道组件的种类。应当掌握组件的继承关系以及每一层的作用。下面几节就详细介绍这些内容。

6.3 组件的种类

Delphi 有 4 种组件：标准控件、自定义控件、图像控件和非可视组件。

注意：Component（组件）和 Control（控件）的概念经常被互换使用，尽管它们并不完全是一回事。控件是指可视的用户接口组件。在 Delphi 中，控件一定属于组件，因为它们是从 TComponent 继承下来的。组件是一种对象，它能够出现在组件选项板中，能够被放在 From 设计器上。组件是 TComponent 类型，不一定是控件，也不一定是可视的用户接口组件。

- **标准组件**

Delphi 提供了一些标准的 Windows 组件，例如 TRichEdit、TTrackBar 和 TListView 等。这些组件可以在组件选项板的 Win32 选项卡上找到。这些组件实际上是把 Windows 公共控件加上了一层 Object Pascal 外套。如果用户可以打开 VCL 的源代码，就会看到 Borland 是怎样在 ComCtrls.pas 文件里给这些组件加外套的。

- **自定义组件**

自定义组件是一个通用的概念，是指那些不是由 Delphi 自带提供的组件。换句话说，这些组件是用户自己或者其他人编写的。

- **图像组件**

图像组件是可视的，但没有输入焦点。这些组件可以在屏幕少年宫显示一些东西，但不会像标准组件和自定义组件那样消耗很多的 Windows 资源。这是因为，图像组件不需要用到窗口句柄。典型的图像组件有 TLable、TShape 等，它们不能作为容器。

- **非可视组件**

非可视组件不具有可视化的特征。这样的组件往往封装了某些功能，但在运行期不直接表现出来。典型的非可视组件有 TOpenDialog、TTable、TTimer 等。

注意：要真正掌握 VCL，尤其是要编写组件的时候，最后要有 VCL 的源代码。看看 Borland 是如编写组件的，这是学会编写组件的最佳途径。

Handle（句柄）：在 Win32 环境中，句柄是一个 32 位的数字，用于标识某个对象实例。这里所说的对象是指 Win32 对象，例如内核对象、用户对象和 GDI 对象等，而非 Delphi 对象。内核对象是指事件、文件映射、进程等。用户对象是指编辑框、列表框、按钮等。GDI 对象是指位图、画刷、字体等。在 Win32 环境中，每一个窗口的句柄是惟一的。许多 Windows API 需要传递一个窗口句柄作为参数。Delphi 封装了 Windows API 并且管理窗口句柄。只有 TWinControl 和 TCustomControl 及其派生类才有句柄特性。

6.4　组件的结构

正如前面提到的那样，组件就是 Object Pascal 的类。所有的组件都具有某种结构。下面就讨论 Delphi 的组件是怎样构成的。

注意：组件和类的概念是有区别的。组件是一种能够在 Delphi 环境中使用的类，而一般的类只是 Object Pascal 的一种数据类型。

6.4.1　组件的特性

特性是访问组件内部字段的接口。通过特性，组件的使用者就可以读或者写组件内部的字段，一般情况下，用户不能直接访问组件内部的字段，因为它们往往声明为私有的。

特性是字段的访问器。有的是直接访问的，有的要经过内部的方法来访问。例如下面这个类的声明：

```
TCustomEdit = class(TWinControl)
private
    FmaxLength: Integer;
protected
    Procedure SetMaxlength(Value: Integer);
        . . .
published
    Property MaxLength: Integer read FMaxLength write SetMaxLength default 0;
        . . .
end;
```

MaxLength 特性就是 FMaxLength 字段的访问器。特性定义包括了特性名称、特性类型、读和写声明以及一个可选的预定义值。读声明指明了读存储字段的方法。可以看出，MaxLength 特性是直接读 FMaxLength 字段，但写 FMaxLength 字段需要借助于一个叫 SetMaxLength 的方法。对特性 MaxLength，一个访问方法 SetMaxLength 通常把这个值赋给存储字段 FMaxLength。当然，读 FMaxLength 字段也可以借助于一个方法，例如：

```
Property MaxLength: Integer read GetMaxLength write SetMaxLength default 0;
```

这里的 GetMaxLength 就是用来读 FMaxLength 字段的方法，它是这样声明的：

```
function GetMaxLength: integer;
```

使用特性来访问一个组件内部数据的方法的好处是，即使以后更改了特性的访问方法，也不会影响组件使用者对此特性的访问；另外一个好处是组件的使用者可以在设计期修改特性的值，只要特性是在类的 published 部分声明的，它就会出现在 Object Inspector 中。

关于特性访问方法，将在第 7 章中详细介绍。

凡是与 Object Pascal 数据类型有关的规则也适用于特性。特性的数据类型决定了它们在 Object Inspector 中被编辑的方式。表 6.1 中列出了特性的各个种类。

表 6.1 特性的种类

种类	Object Inspector 的处理方式
简单	包括数字、字符和字符串等。用户可以直接输入特性的值
枚举	可以从下拉列表中选择，即从一组值中选择一个值作为特性的值
集合	在 Object Inspector 中可以展开集合类型的特性,集合中的每个元素就好像 Boolean 类型的特性，设为 True 表示集合中包含此元素。可选择多个值
对象	要编辑的对象类型的特性需要打开一个特定的编辑器。如果对象本身的特性是公开的，用户也可以在 Object Inspector 中展开分别设置每个子特性
数组	数组特性必须有专门的特性编辑器。Object Inspector 不能编辑这种类型的特性

6.4.2 组件的事件

事件表示某个动作的发生，例如单击一个按钮、按下键盘的一个键等。事件也可以看作特殊的特性。组件的使用者可以指定事件发生时要执行的代码。

对事件的处理可以在设计期进行，也可以在运行期进行。在设计期中，以 TEdit 组件为例，用户可以在 Object Inspector 中发现它有 OnChange()、OnClick()和 OnDBlClick()等事件。对于组件编写者来说，事件实际上是指向方法的指针。当组件的使用者把代码指派给一个事件时，就称为建立事件的句柄。例如，当在 Object Inspector 中双击一个事件的时候，Delphi 会自动生成一个方法的框架，让用户加入代码，如下所示：

```
TForm1 = class(TForm)
        Button1: TButton;
        procedure Button1Click(Sender: TObject);
private
        { Private declarations }
public
        { Public declarations }
end;
    …
procedure TForm1.Button1Click(Sender: TObject);
begin
        //事件详细的代码
end;
```

在运行期，当用户通过编程把一个事件句柄指派给一个事件的时候，可以清楚地看到，事件实际上就是方法的指针。例如要处理 TButton 的 OnClick()事件，首先要声明和定义一个方法，这个方法应该在拥有 TButton 组件的 Form 中声明中：

```
TForm1 = class(TForm)
        Button1: TButton;
```

```
        ...
Private
        procedure MyOnClickEvent(Sender: TObject);    //声明一个方法
      …
procedure TForm1.MyOnClickEvent(Sender: TObject);
begin
        //方法的详细代码
end;
```

上面的代码中，MyOnclickEvent()就是一个事件的句柄，它用来处理 Button1.OnClick()事件，只要将 MyOnClickEvent 赋给 Button1.OnClick()事件就可以了。

可以根据不同的情况指派不同的事件句柄给同一个事件。另外，也可以通过把事件设置为 nil 来禁止事件句柄。

在设计期建立事件的句柄和在运行期指派事件句柄在本质上是一致的。但是需要注意，并不是任意的方法都可以赋给事件，由于事件实际上是方法的指针，所以事件句柄也必须符合一定的要求，传递的参数个数、类型和顺序都要和事件本身的个数、类型和顺序完全一致才可以。

前面说过，事件也是特性。就像一般的特性一样，事件也可以声明为一个组件的私有数据。例如，下面的代码中声明了一个 TMouseEvent 类型的事件：

```
TControl = class(TComponent)
Private
        FOnMouseDown: TMouseEvent;
Protected
        Property OnMouseDown: TMouseEvent read FOnMouseDown write FOnMouseDown;
      …
end;
```

6.4.3　组件的拥有关系

一个组件可以用其他组件。一个组件的拥有者是由 Owner 特性指定的。一般情况下，Form 就是 Form 上的组件的拥有者。当用户把一个组件加到 Form 上，Form 就自动成为这个组件的拥有者。如果是在运行期动态地创建一个组件的实例，必须在调用这个组件的 Create()事件时指定这个组件的拥有者。下面的代码在调用 TButton.Create()时传递的参数是 self，表示这个组件的拥有者是 From：

```
MyButton := TButton.Create(self);
```

当 Form 的实例被释放的时候，TButton 的实例也会被释放，这是由 VCL 内部处理的。Form 会遍历它所拥有的所有组件，然后逐个释放。当然，创建一个组件的实例也可以不指定任何拥有者，只要传递 nil 给组件的 Create()事件即可，不过，如果没有指定拥有者，用户就必须负责最后释放组件实例。

一般情况下，除非实在无法作到这一点，应该指定组件的拥有者。

另一个有关所有者的特性是 Components 特性。Components 特性是一个数组，它包含的是一个组件所拥有的所有其他组件。例如，一个 Form 上的所有组件就包含在 Components[indx]中。

6.4.4 组件的父子关系

不要把拥有关系和父子关系混淆起来，组件可以成为其他组件的父。只有从 TWinControl 继承下来的窗口组件才可以成为其他组件的父。作为父的组件必须负责调用子组件的方法来画出子组件自己。一个组件的父是通过 Parent 特性指定的。

一个组件的父组件不一定非得是一个组件的拥有者，一个组件的拥有者和父完全可以不同，这是允许的。

6.5 组件的继承关系

VCL 库具有一个可扩展的组件层次结构，它们扮演着某种幕后角色。VCL 库的另外一个重要特征是，它具有定义良好的封装函数层，允许编程人员直接选择所需函数的子组件，而不必走许多冤枉路。然而，用户必须熟悉一些重要的类。

下图显示了 VCL 的继承关系，如图 6.1 所示。

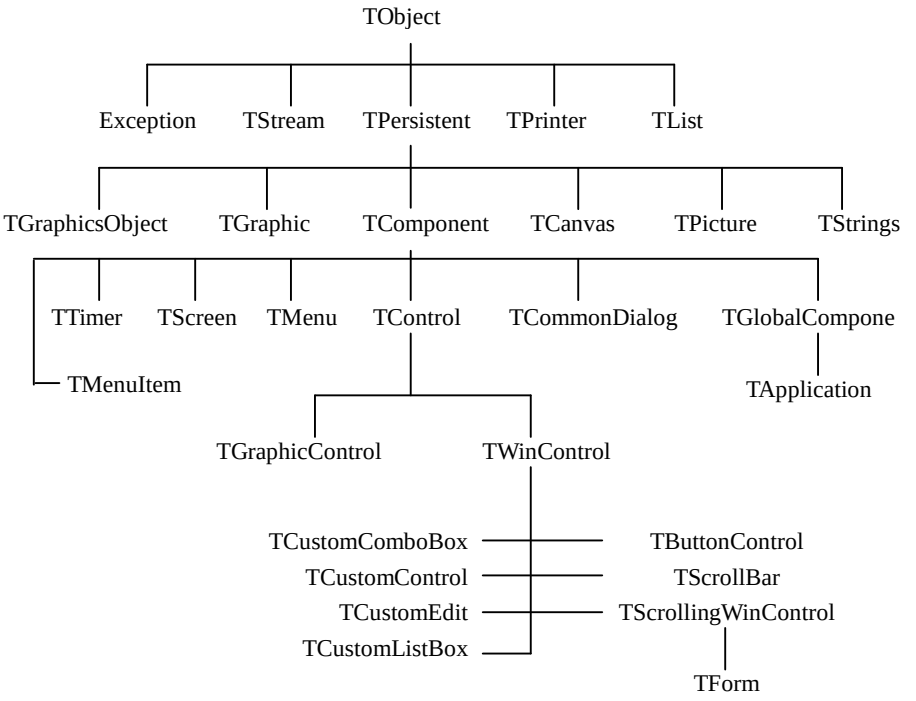

图 6.1 VCL 的继承关系

Delphi 中所有类的基本行为都继承自 TObject 类，这个类封装了 VCL 分层结构中所有类共同的特征和行为。但是作为一个组件编写者，一般不需要直接从 TObject 继承组件下

来，VCL 提供了一些 TObject 的派生类让用户继承，除非想创建一个非组件的类，才考虑从 TObject 继承。

TObject 的 Create()和 Destroy()的作用是创建和删除对象实例。TObject 的 Create()将返回对新创建的对象实例的引用。

TObject 的方法大部分是 VCL 内部使用的，其中一些方法可以获取一个对象的类、类名和祖先类等信息。

> 注意：建议用 TObject.Free()而不要用 TObject.Destory()方法来删除一个对象实例，因为 Free()方法会检查要删除的对象实例是否为 nil，这样会避免对一个不存在的对象进行操作。

6.5.1　TPersistent 类

VCL 层次结构中 TObject 下面第一层类中有一个 TPersistent 类。TPersistent 为所有子类增加了两个非常重要的方法：SaveToStream 和 LoadFromStream，这两个方法使得凡是从 TPersistent 继承下来的对象都具有进行流操作的能力。

对象能一直使用 SaveToStream 和 LoadFromStream 这两个方法。例如，窗体设计器需要创建一个 DFM 文件时，它会在自己拥有的组件数组中循环，并向窗体上所有的组件调用 SaveToStream 方法，而每个组件都知道如何将修改过的属性输出到数据流中（这种情况下是一个二进制文件）。反过来，当窗体设计器需要从 DFM 文件中装入组件属性时，它会在组件数组中循环，并向窗体上所有的组件调用 LoadFromStream 方法。这样，所有来源于 TPersistent 的类都能保存自己的状态信息，并能在需要的时候恢复。

TPersistent 没有增加新的特性和事件，但定义了一些新的方法，见表 6.2。

表 6.2　TPersistent 的方法

方法	用途
Assign()	该方法用于将另一个组件的数据赋给自己。
AssignTo()	TPersistent 的派生类必须实现这个方法。把自己的数据赋给另一个组件。
DefineProperties	指定怎样存储特性的值。该方法用于是组件能存储诸如二进制数据等非简单数据类型的数据。

下面向读者介绍有关流的特性。

组件的特征之一就是能够进行流操作。流是一种存储组件及其有关特性值信息的方式。Delphi 提供了强大的流操作能力。事实上，由 Delphi 建立的 dfm 文件只不过是一种资源文件，它包括 Form 组件的流信息。

6.5.2　TComponent 组件

TComponent 是直接从 TPersistent 继承下来的，是 Delphi 中所有组件的祖先，它是 VCL 层次结构中一个非常重要的部分。TComponent 为所有的组件封装了以下行为：

- 安装到组件面板的能力

- 在设计时能在窗体上拖拽的能力
- 通过对象检视器和组件编辑器设置属性的能力

从 TComponent 检测下来的对象即组件的特征是，它们的特性可以在设计期通过 Object Inspector 来操纵，可以拥有其他组件。

注意，所有的组件都必须能实现上述行为。将它们封装入 TComponent 后，就能通过修改 TComponent 并重新编译起源于它的派生类来改变所有组件的行为。这也是面向对象的程序设计方法的优点之一。

TComponent 充当了一个所有组件都能插入的总线角色。TComponent 里有一些用来支配组件在设计时的行为的方法。例如 Name 属性。每一个 TComponent 派生的组件都有一个 Name 属性。不需要可视界面的组件（例如对话框组件或计时器组件）也都是由 TComponent 派生而来的，因此它们也可以在设计期被操纵。而可视控件则出现在继承关系结构的再下一层。

TComponent 定义了一些有趣的特性和方法，见表 6.3。

表 6.3 TComponent 的特性

特性	用 途
Owner	组件的拥有者
ComponentCount	所拥有的组件个数
ComponentIndex	组件在它的拥有者组件列表中的序号。以 0 开始
Components	这是一个数组，包含组件所拥有的组件
ComponentState	组件的当前状态
ComponentStyle	组件的风格
Name	组件的名称
Tag	一个整数，本身没有明确的用途，组件编写者不能使用它，但应用程序开发者可以利用它来区分不同组件
DesignInfo	Form 设计期内部使用

TComponent 定义了若干的方法，可以用来拥有其他组件或被 Form 设计期操纵。

TComponent 的 Create()方法用于创建组件的实例，并指定组件的拥有者。与 TObject.Create()方法不同的是，TComponent.Create()方法是虚拟的，TComponent 的派生类必须重载这个方法。尽管用户可以在派生类中另外声明一个 Constructors（构造），但是要使 VCL 能够在设计期和运行期创建基于流的组件的实例，只能使用 TComponent.Create()方法。

TComponent 的 Destroy()方法用于把组件及其所拥有的组件设为一种状态，以表明组件实例正在被删除。TComponent.DestroyComponents()方法用于删除每个拥有的组件。一般情况下，用户不用直接调用 Create()和 Destroy()这两种方法。

TComponent.FindComponent()方法用于在所拥有的组件中查找一个组件；TComponent. GetParentComponet()方法用于返回父的实例，如果组件没有父，这个方法将返回 nil；TComponent. HasParent()方法返回一个 Boolean 类型的值，表明组件是否有父。

TComponent.InsertComponent 方法的作用是拥有一个组件，而 TcomponentRemove-Component 方法的作用是取消拥有关系。一般情况下不要调用这两个方法，因为拥有关系是由 Create 方法和 Destroy 方法自动确立或取消的。

6.5.3　TControl 类

从 TControl 继承下来的组件称为控件，它们是可视的。控件可以显示自己、可以设置它们在屏幕的位置和尺寸、可以设置它们的客户区。与 TComponent 定义所有组件的行为一样，TControl 定义了所有可视组件的行为。它包括绘图例程，标准 Windows 事件及组件包装等。通过 Visible、Enabled、Color 等特性可设置控件的外观，通过 Font 特性可以设置字体。它的文本则是通过 TControl 的特性 Text 和 Caption 设定的。

TControl 引入了一些设置鼠标的事件，例如 OnClick、OnDblClick、OnMouseDown、OnMouseMove 和 OnMouseUp 等。此外，还引入了进行拖放的事件，例如 OnDragOver、OnDragDrop 等。

TControl 本身是一个虚拟类，决不能直接创建它的实例。

从 TControl 继承下来的组件可以有父，但作为父的控件必须是从 TWinControl 继承下来的（父控件必须是窗口控件）。TControl 引入 Parent 特性。

TControl 下面又可以分为两种类：TGraphicControl 和 TWinControl，大部分控件是从这两个类继承下来的。

6.5.4　TGraphicControl 类

TGraphicControls 组件必须由自己完成绘图功能，而且不可能获得控制焦点。例如 Timage、TLabel 和 TBevel 等组件。注意，这些组件都具有全部的绘图例程（Paint，Repaint，Invalidate 等）。但是 Delphi 并不会从 Windows 中为它们分配窗口句柄，因为它们永远不会要求获得控制焦点。在选择使用 TBevel 与 TPanel 中的哪一个组件时，如果不需要使用 Panel 的 ContainShip 属性，建议使用 Bevel 组件，因为该组件使用较少的 Windows 资源。

因为从 TGraphicControl 继承下来的控件没有窗口句柄，因此它们不能成为其他控件的父。当需要在 Form 上显示一些东西，但又不需要用户和它交互，这种情况下应当使用从 TGraphicControl 继承下来的控件。这种控件的好处是不需要窗口句柄，因而可以节省系统资源。同时这些控件画出自己的速度也较快。从 TGraphicControl 继承下来的控件可以响应鼠标事件，实际上，是它们的父控件把鼠标消息传递给它们的。

为了在屏幕上画出自己，TGraphicControl 提供了一个 TCanvas 类型的特性叫 Canvas，同时，还提供了一个虚拟方法 Paint()，TGraphicControl 的派生类必须重载它。

6.5.5　TWinControl 类

TWinControl 除了能够接受控制焦点外，其它与 TGraphicControls 相同。这意味可应用的标准事件将会更多，同时 Windows 必须为它们分配一个窗口句柄。这些组件包括那些 Windows 自动绘制的控件（包括 TEdit、TListBox、TCombobox、TPageControl 等）以及那些必须由 Delphi 绘制的自定义控件（包括 TDBNavigator、TMediaPlay 和 TGauge 等）。然

而，用户根本不用关心这些组件使自己生效或响应事件的任何实现细节，因为 Delphi 已经将这些行为为用户完全封装好了。

标准的窗口控件 TForm 就是从 TWinControl 继承下来的，这类控件构成了 Windows 应用程序的用户界面，例如编辑框、列表框、组合框、按钮等。用户根本不需要使用 Windows API 函数，只要通过这些控件的特性、方法和事件就可以操纵程序的用户界面。

从 TWinControl 继承下来的控件具有 3 个特征：有窗口句柄、可以接受输入焦点、可以成为其他控件的父。

应用程序开发者一般使用的是 TWinControl 的派生类，而组件编写者则必须掌握 TWinControl 的一个叫 TCustomControl 的派生类。

通过 TWinControl 的特性，用户可以切换输入焦点或者改变控件的外观。

- TWinControl.Brush 特性代表控件的刷子，它可以设置控件内部怎样填充。
- TWinControl.Controls 特性是一个数组，包含了控件的所有子控件。
- TWinControl.ControlCount 特性是控件的子控件的个数。
- TWinControl.Ctl3D 特性用于设置控件是否要具有三维外观。
- TWinControl.Handle 特性是控件的窗口句柄。调用某些 Win32 API 时需要传递窗口句柄。
- TWinControl.Showing 特性表明控件是否是可见的。
- TWinControl.TabStop 特性是一个 Boolean 类型的值，表明是否可以通过 TAB 键切换输入焦点。
- TWinControl. TabOrder 特性是一个序号，当用户按 Tab 键时将依次切换输入焦点。

通过 TWinControl 的方法，用户可以创建窗口、使一个控件具有焦点、指派事件恶化定位控件。TWinControl 的方法很多，与窗口创建、重建和删除有关的方法一般只有组件编写者才会用到，它们包括 Createparams()、CreateWnd()、CreateWindowHandle()、DestroyWnd()、DestroyWindowHandle()和 RecreateWnd()。

与窗口焦点、定位和对齐有关的方法有 CanFocus()、Focused()、AlignControls()、EnableAlign()、DisabledAlign()和 ReAlign()。

在 TWinControl 中，用于处理键盘的事件有 OnKeyDown()、OnKeyPress()和 OnKeyUp()。用于处理焦点转换的事件有 OnEnter()和 OnExit()。

6.5.6 TCustomControl 类

读者可能注意到了，TWinControl 的有些派生类的名称是以 TCustom 开头的，例如 TCustomComboBox、TCustomControl、TCustomEdit 和 TCustomListBox 等。除了具有 TWinControl 派生类的特性以外，这些以 TCustom 开头的派生类主要是作为自定义控件的基类而存在的。组件编写者经常要用到这些类。

6.5.7 VCL 助手类

VCL 库中包括大量的助手类。虽然它们并不是 VCL 层次结构中的必要部分，但是有些 VCL 类会用它们来辅助一些操作。当然这些助手也很有用处，使用它可以大大减少为管

理这些类的行为所必须输入的代码量。

这一节将简要介绍这些类和它们的一些用法。这里，本节仅仅是让读者知道有这些类存在而且可以使用它们。

1．TStrings

TStrings 是一个抽象类，用于操纵一个字符串列表。TStrings 本身并不直接管理字符串的内存，而是由拥有 TStrings 对象实例的组件管理。TStrings 只是提供了一些特性和方法来访问和操纵这些字符串，而不需要借助于 Win32 API。

TListBox.Items、TMemo.Lines 和 TComboBox.Items 等特性的类都是 TStrings。

2．TStringList

TStringList 类，顾名思义是一个字符串的容器类。读者可能已经注意到了，很多标准组件都使用了一些 TStringList 来容纳字符串列表（例如 TListBoxed、TTabControl 等）。如果愿意，可以自己创建这样的列表。如果对使用动态数组的语言很熟悉，那么就会发现 TStringList 是个很不错的组件，因为它们在内部是作为一个列表来实现的，而且能够随意增加而不用用户去编写任何管理列表的代码。

下面这个示例显示如何创建一个 TStringList 类并加入列表内容：

```
procedure Tform1.Button1Click(Sender: TObject);
var
        List: TStringLIst;
begin
        List := TStringList.Create;
        with List do begin
                Add('Neal');
                Add('Ed');
                Add('Terry');
                Add('Jennifer');
                Add('Casey');
        end;
        ListBox1.Items := List;
end;
```

上例中，创建了一个 TStringList，然后向里面加了一些名字，最后将它赋给 ListBox 的 Items 特性，这样就能让 ListBox 显示这些名字。TStringLists 不仅是字符串的容器，而且能容纳除字符串外的能创建词典风格数据结构的对象引用。如果用户不用 Add()而用 AddObject()方法，能够将字符串及任何对象（任何 TObject 的子类）加入到列表中并且能通过索引或字符串值检索它们。这种方法常用于为那些既要显示字符串又要显示位图的 ListBox。

3．TList

TList 在内部实现上与 TStringList 相似，但是它能容纳任何类型的对象，包括混合的对象类型或类指针（该对象并不需要最终继承于 TObject，这不同于 TStringList）。下面的

程序片段显示了使用 TList 来容纳一些窗体控件的方法：

```
procedure Tform1.Button1Click(Sender: TObject);
var
    List: TList;
    i: integer;
begin
    List := TList.create;
    with List do begin
        Add(Button1)；
            Add(Label1);
            Add(Edit1);
    end;

    for I := 0 to List.count - 1 do
        ShowMessage('Component Name is' + TControl(List.Items[I]).Name);
end;
```

4．TCanvas

TCanvas 封装了 Windows 的设备环境（Device Context），它能处理所有窗体、可视容器（例如 TPanel）以及 TPrinter 对象的绘制。使用画布对象，用户就不再需要担心画笔、颜色刷、调色板的分配，因为所有的分配和再分配任务都已经处理好了。TCanvas 包括图形基础例程，用来在包含一个画布的任何控件上绘制线条、形状、多边形、字体等。

关于 TCanvas 类的使用方法，在“图像编程”章中有更详细地讲解。

5．TPrinter

TPrinter 对象封装了 Windows 打印机的所有细节。可以使用 Printers 特性得到已安装及可以使用的打印机的列表。该打印机对象使用了一个 TCanvas，因此，任何能够绘制在窗体上的东西也能被打印出来。打印图像时，首先调用 BeginDoc 方法，然后在后面加上用户希望打印的画布图形（可以使用 TextOut 方法输出文字），接着调用 EndDoc()方法向打印机发送任务。想要了解更多的 TPrinter 对象的使用信息，可以参考 TPrinter 的在线帮助。

6．TINIFile

Windows3.1 和它的应用程序将配置信息储存在一种特殊格式的文本文件，即 INI 文件中。这种 INI 文件格式在 Windows95 中依然很常用，很多 Delphi 的配置文件（例如 DSK 桌面设置文件）就是这种格式。由于这种格式曾经很流行而且现在仍然很流行，因此 VCL 提供了一个类，使读写这些文件便得更加简单。当用户要创建一个 TINIFiles 对象时，需要将 INI 文件的路径和文件名称作为参数传递给构造函数 Create()。如果该文件不存在，将自动创建该文件。然后就可以用诸如 ReadString()、ReadInteger()或 ReadBool()等函数任意读取文件中的值。另外，如果想读 INI 文件的完整的一节，可以使用 ReadSection()方法。同样，可以使用 WriteBool()、WriteInteger()或 WriteString()等方法向文件中写入值。

下面是一个示例，第一个函数用来从一个 INI 文件中读取配置信息，第二个函数向文件中写入信息。

```
procedure ReadIt;
var
    IniFile: TiniFile;
begin
    IniFile := TiniFile.Create(ChangeFileExt(Application.ExeName, '.INI'));
    with IniFile do begin
        Top := ReadInteger('Form', 'Top', 100);
        Left := ReadInteger('Form', 'Left', 100);
        if ReadBool('Form', 'InitMax', false) then begin
          windowState := wsMaximized;
        end else WindowState := wsNormal;
    end;
    IniFile.free;
end;

procedure WriteIt;
var
    IniFile: TIniFile;
begin
    IniFile := TIniFile.Create(ChangeFileExt(Application.ExeName, '.INI'));
    with IniFile do begin
        WriteInteger('Form', 'Top', Top);
        WriteInteger('Form', 'Left', Left);
        WriteBool('Form', 'InitMax', WindowState = wsMaximized);
    end;
    IniFile.free;
end;
```

在每一个 Read **例程中都有** 3 **个参数，第一个参数指向** INI **文件中的节** Section**，第二个参数指向希望读出的值，第三个参数是一个缺省值，当** INI **文件中节或值不存在时，该参数将作为返回值。同样，**Write **例程在节或值不存在时也会创建节或值。第一次运行上述代码后，生成的** INI **文件如下所示：**

```
[Form]
Top=100
Left=100
InitMax=0
```

7．TRegistry

Windows95 **和** WindowsNT **都使用注册表来管理配置信息。注册表是一个由操作系统管理的二进制数据库。由于注册表是分层次的，具有很好的健壮性，而且不会像** INI **文件那样大小受限制，因此** Windows95 **及大多数** 32 **位应用程序都是将配置信息保存在注册表中而不是** INI **文件中。**VCL **提供了一个** TRegistry **类来操纵注册表。**TRegistry **对象提供了**

包括打开、关闭、保存、移动、复制和删除键值的方法。操纵注册表比使用 INI 文件困难得多，而且危险得多，如果不慎破坏了注册表，可能会使 Windows 无法正常工作，所以一定要十分小心。具体 TRegistry 的使用方法，可以参考 VCL 帮助中 TRegistry 主题下的内容。

8．TRegINIFile

如果习惯于使用 INI 文件，而又希望将配置信息移到注册表中，用户可以使用 TRegINIFile 类。设计 TRegINIFile 的目的就是使得对注册表的读取方式同读取 INIFile 的方式类似。所有 TINIFile 中有的方法（读和写），TRegINIFile 都提供有。当构造了一个 TRegINIFile 对象时，传递的参数（INIFile 对象的名字）在注册表中将会成为 User 键下的一个键值，而且所有的节和值将以上述位置为根，依次向下扩展。事实上，该对象能极大地简化注册表的接口，因此，即使在不需要使用当前存在的代码时，用户也会愿意使用它而不是使用 TRegistry 类。

6.6 运行期类型信息

在 Object Pascal 一章中，本书曾经提到过运行期类型信息 RTTI（Runtime Type Information），本章将更加深入地讨论它，这里读者将学习到如何获得一个对象的类型信息，就好像 Delphi IDE 获取这些信息一样。

VCL 中的 TObject 类提供了运行期类型信息。它使 Delphi 环境通过查询运行时的对象类找出这些类的重要信息，例如类名、是否采用了某种方法等。RTTI 对环境来说很重要，因为它能使编译器直到运行时才去判定对象的实际类型。这能使运行时组件和对象的类型设定更加灵活。

RTTI 至少在两个地方有用。第一个地方是 Delphi 的 IDE，通过 RTTI，IDE 就能知道一个对象和组件的任何事情；第二个地方是程序员编写的代码，例如 Is 和 As 运算符。

下面通过 Is 运算符来说明 RTTI 的作用。

假设要使一个 Form 上的所有 TEdit 组件变成只读的，只需要 Form 上的所有组件、并且用 Is 运算符逐个判断是不是 TEdit 组件、然后把 ReadOnly 特性设置为 True 就可以了。如下所示：

```
for I := 0 to ComponentCount – 1 do
    if Components[I] is TEdit then
        TEdit(Component[I]).ReadOnly := True;
```

as 运算符的典型用法是在一个事件句柄中把 Sender 参数强制转换为某种类型。假设已经知道所有的组件都是从一个共同的祖先类继承下来的，要访问其中一个组件的某个特性，应该用 as 运算符把 Sender 参数强制转换为某种类型，然后再访问这个特性。

上面的方法隐含使用了 RTTI 信息，下面来讨论直接使用 RTTI 的问题。

假设一个 Form 上既有数据感知组件又有非数据感知组件，而用户现在要针对 Form 上的数据感知组件进行操作。当然，仍然可以遍历 Form 上的所有组件，然后判断每一个组件的类型。不过，必须要与每一种数据感知组件的类型进行比较，这是非常麻烦的。

要测试一个组件是不是数据感知的，最方便的方法就是检查它是不是具有 DataSource 特性，因为只有数据感知组件才有 DataSource 特性，这时候，就可以直接用到 RTTI。该功能的具体实现将在后面的 6.6.2 一节中讲解。

6.6.1　TypInfo.pas 单元

任何一个对象都有它自己的类型信息，Delphi IDE 和应用程序都可以查询这些类型信息。在 TObject 类的定义中，有一个 ClassInfo()方法，如下所示：

```
class function ClassInfo: Pointer;
```

它的返回值为一个指针，指向了 RTTI 信息，也就是 TTypeInfo 记录。TypInfo.pas 单元中定义了 RTTI 信息的结构，如下所示：

```
PPTypeInfo = ^PTypeInfo;
PTypeInfo = ^TTypeInfo;
TTypeInfo = record
        Kind: TTypeKind;
        Name: ShortString;
      {TypeData: TTypeData}
end;
```

其中，Kind 域是一个枚举，它可以是下面的值：

```
TTypeKind = (tkUnknown, tkInteger, tkChar, tkEnumeration, tkFloat,
        tkString, tkSet, tkClass, tkMethod, tkWChar, tkLString, tkWString,
        tkVariant, tkArray, tkRecord, tkInterface, tkInt64, tkDynArray);
```

在 TTypeInfo 的声明中，最后一项是用括号括起来的 TypeData 域，它就是实际的类型信息，具体内容取决于 Kind 域的值，也就是说，Kind 域决定了 TypeData 域到底包含有哪些信息。下面是 TTypeData 类型在 TypInfo.pas 单元中的声明：

```
PTypeData = ^TTypeData;
TTypeData = packed record
    case TTypeKind of
      tkUnknown, tkLString, tkWString, tkVariant: ();
      tkInteger, tkChar, tkEnumeration, tkSet, tkWChar: (
        OrdType: TOrdType;
        case TTypeKind of
          tkInteger, tkChar, tkEnumeration, tkWChar: (
            MinValue: Longint;
            MaxValue: Longint;
            case TTypeKind of
              tkInteger, tkChar, tkWChar: ();
              tkEnumeration: (
                BaseType: PPTypeInfo;
```

```
                    NameList: ShortStringBase;
          EnumUnitName: ShortStringBase));
              tkSet: (
                  CompType: PPTypeInfo));
          tkFloat: (
              FloatType: TFloatType);
          tkString: (
              MaxLength: Byte);
          tkClass: (
              ClassType: TClass;
              ParentInfo: PPTypeInfo;
              PropCount: SmallInt;
              UnitName: ShortStringBase;
            {PropData: TPropData});
          tkMethod: (
              MethodKind: TMethodKind;
              ParamCount: Byte;
              ParamList: array[0..1023] of Char
            {ParamList: array[1..ParamCount] of
                record
                    Flags: TParamFlags;
                    ParamName: ShortString;
                    TypeName: ShortString;
                end;
              ResultType: ShortString});
          tkInterface: (
              IntfParent : PPTypeInfo; { ancestor }
              IntfFlags : TIntfFlagsBase;
              Guid : TGUID;
              IntfUnit : ShortStringBase;
            {PropData: TPropData});
          tkInt64: (
              MinInt64Value, MaxInt64Value: Int64);
          tkDynArray: (
              elSize: Longint;
              elType: PPTypeInfo;           // nil if type does not require cleanup
              varType: Integer;             // Ole Automation varType equivalent
              elType2: PPTypeInfo;          // independent of cleanup
        DynUnitName: ShortStringBase);
    end;
```

正如读者所看到的，TTypeData **是一个非常复杂的记录，而且是一个可变的记录，为了很好地理解** RTTI **信息，需要熟悉可变记录和指针。**

6.6.2　获取 RTTI

下面介绍一个示例，来演示怎样获取 RTTI 信息，这里将会看到如何获取一个对象的基本信息、祖先类、特性信息以及特性类型等，如下所示：

```
unit Un_Main;

interface

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

type
    Tfm_Main = class(TForm)
        lbClasses: TListBox;
        Label1: TLabel;
        btnGetClasses: TButton;
        lbClassInfo: TListBox;
        lbProperty: TListBox;
        Label2: TLabel;
        Label3: TLabel;
        procedure btnGetClassesClick(Sender: TObject);
        procedure lbClassesClick(Sender: TObject);
    private
        { Private declarations }
    public
        { Public declarations }
        function CreateClass(const AClassName: string): TObject;
        procedure GetClassInfo(AClass: TObject; AStrings: TStrings);
        procedure GetClassAncestor(AClass: TObject; AStrings: TStrings);
        procedure GetClassProperty(AClass: TObject; AStrings: TStrings);
    end;

var
    fm_Main: Tfm_Main;

implementation

{6R *.dfm}

procedure Tfm_Main.btnGetClassesClick(Sender: TObject);
begin
    with lbClasses.Items do begin
        Clear;
        Add('TApplication');
        Add('TForm');
        Add('TButton');
        Add('TComponent');
        Add('TGraphicControl');
```

```
            Add('TListBox');
            Add('TComboBox');
            Add('TLabel');
            Add('TActionList');
        end;

            RegisterClasses([TApplication, TForm, TButton, TComponent,
  TGraphicControl, TListBox, TComboBox, TLabel, TActionList]);
end;

function Tfm_Main.CreateClass(const AClassName: string): TObject;
var
        C: TFormClass;
        Obj: TObject;
begin
        C := TFormClass(FindClass(AClassName));
        Obj := C.Create(nil);
        Result := Obj;
end;

procedure Tfm_Main.GetClassAncestor(AClass: TObject; AStrings: TStrings);
var
        AC: TClass;
begin
        AC := AClass.ClassParent;
        AStrings.Add('Class Ancestry');
        while AC <> nil do begin
            AStrings.Add(Format('     %s', [AC.ClassName]));
            AC := AC.ClassParent;
        end;
end;

procedure Tfm_Main.GetClassInfo(AClass: TObject; AStrings: TStrings);
var
        TI: PTypeInfo;
        TD: PTypeData;
        EnumName: string;
begin
        TI := AClass.ClassInfo;
        TD := GetTypeData(TI);
        with AStrings do begin
            Add(Format('Class Name:            %s', [TI.Name]));
            EnumName := GetEnumName(TypeInfo(TTypeKind), Integer(TI.Kind));
            Add(Format('Kind:                  %s', [EnumName]));
            Add(Format('Size:                  %d', [AClass.InstanceSize]));
            Add(Format('Defined in:            %s.pas', [TD.UnitName]));
            Add(Format('Num Properties:     %d', [TD.PropCount]));
        end;
end;
```

```
procedure Tfm_Main.GetClassProperty(AClass: TObject; AStrings: TStrings);
var
        PropList: PPropList;
        TI: PTypeInfo;
        TD: PTypeData;
        i: integer;
        NumProps: integer;
begin
        TI := AClass.ClassInfo;
        TD := GetTypeData(TI);

        if TD.PropCount <> 0 then begin
                GetMem(PropList, SizeOf(PPropInfo) * TD.PropCount);
                try
                   GetPropInfos(AClass.ClassInfo, PropList);
                   for i := 0 to TD.PropCount - 1 do
                      if not (PropList[i]^.PropType^.Kind = tkMethod) then
                         AStrings.Add(Format('%s: %s', [PropList[i]^.Name,
PropList[i]^.PropType^.Name]));
                   NumProps := GetPropList(AClass.ClassInfo, [tkMethod], PropList);
                   if NumProps <> 0 then begin
                      AStrings.Add('');
                      AStrings.Add('---------- Events -------');
                      AStrings.Add('');
                   end;
                   for i := 0 to NumProps - 1 do
                      AStrings.Add(Format('%s: %s', [PropList[i]^.Name,
 PropList[i]^.PropType^.Name]));
                finally
                   FreeMem(PropList, SizeOf(PPropInfo) * TD.PropCount);
                end;
        end;
end;

procedure Tfm_Main.lbClassesClick(Sender: TObject);
var
        Cls: TObject;
        ClassName: string;
begin
        lbClassInfo.Items.Clear;
        lbProperty.Items.Clear;

        try
                ClassName := lbClasses.Items[lbClasses.ItemIndex];
                Cls := CreateClass(ClassName);
                try
                   GetClassInfo(Cls, lbClassInfo.Items);
                   GetClassAncestor(Cls, lbClassInfo.Items);
                   GetClassProperty(Cls, lbProperty.Items);
                finally
```

```
                Cls.Free;
            end;
        except on E: EClassNotFound do
                ShowMessage('Class not Found');
        end;
end;

end.
```

程序运行后的显示界面如图 6.2 **所示。**

图 6.2 RTTI **信息**

首先单击“得到类型”按钮，在类型列表中将显示出几个类，这些是在程序中自己加进去的，也可以增加别的类。在将类加入到列表中的同时，程序调用了 RegisterClasses()**函数来注册了这些类。选择一个类以后，在基本类型信息列表中将显示出当前选定的类型的基本信息，包括长度、祖先类层次等，例如上图中（见图** 6.2**），**TActionList **类的继承关系为：**TObject->TPersistent->TComponent->TCustomActionList**。在对象特性及其类型列表中显示了类型包含的特性、特性类型以及事件等信息。**

CreateClass()**函数用来基于一个给定的类名创建一个对象实例，但在此之前必须保证要创建的类已经注册过了。**

为了获得对象的 RTTI **信息，程序调用了** GetClassInfo()**函数，该函数把** TObject.ClassInfo()**函数的返回值传递给** GetTypeData()**函数。**GetTypeData()**函数是在** TypInfo.pas **单元中声明的，它的作用是返回** TTypeData **的结构指针。**

GetClassAncestor()函数用来获得一个对象的祖先类。它实际上调用了 TObject 的 ClassParent()函数，该函数返回的是一个 TClass 类型的值，即当前对象的祖先类。GetClassAncestor()函数将逐级追溯到最顶级的祖先，然后把所有祖先的类名加到一个列表中。

上面的示例中，相对较复杂的是获得对象的特性信息。这里需要用到一个 PPropList 类型的变量 PropList，PPropList 类型在 TypInfo.pas 单元中这样声明：

```
PPropList = ^TPropList;
TPropList = array[0..16379] of PPropInfo;
```

TPropInfo 就是特性的 RTTI，其中 PPropInfo 声明为：

```
PPropInfo = ^TPropInfo;
TPropInfo = packed record
PropType: PPTypeInfo;
    GetProc: Pointer;
    SetProc: Pointer;
    StoredProc: Pointer;
    Index: Integer;
    Default: Longint;
    NameIndex: SmallInt;
    Name: ShortString;
end;
```

GetClassProperty()方法调用 GetPropInfos()函数来填充 TPropList 数组，然后遍历这个数组，取出了每一个特性的名称和类型。

6.6.3　检查特性

要判断一个组件是否是数据感知组件，可以检查它是否具有 DataSource 特性，也就是说可以检查一个对象是否具有某个特性。下面给出一个函数，它的作用是判断一个组件是否是数据感知的。

```
function IsDataAware(AComponent: TComponent): Boolean;
var
    PropInfo: PPropInfo;
begin
    PropInfo := GetPropInfo(AComponent.ClassInfo, 'DataSource');
    Result := PropInfo <> nil;

//确保特性名称为 DataSource 的特性是否属于 TDataSource 类型
    if Result then
        if not ((PropInfo^.PropType^.Kind = tkClass)
and (GetTypeData(PropInfo^.PropType^).ClassType.InheritsFrom
(TDataSource))) then
```

```
            Result := False;
end;
```

将上面的函数稍作修改就可以用来判断任何组件是否具有某一个特性，如下所示：

```
function CheckProperty(AComponent: TComponent;
APropertyName: string): Boolean;
var
    PropInfo: PPropInfo;
begin
    PropInfo := GetPropInfo(AComponent.ClassInfo, APropertyName);
    Result := PropInfo <> nil;
end;
```

不过因为只有公开的特性才有 RTTI 信息，这个函数只适用于检查公开的特性，即由 Published 部分声明的特性。

6.7 本 章 小 结

本章详细讨论了 VCL 的应用结构、VCL 组件的特性、事件等，以及组件之间的拥有关系和父子关系。

在了解了 VCL 的继承关系图以后，本章就几个非常重要的类（TPersistent、TComponent、TControl、TGraphicControl、TWinControl 和 TcustomControl），以及 VCL 的助手类（TStrings、TStringList、TCanvas 等）进行了介绍和讲解。

在本章的最后，深入详细地讲解了运行期类型信息 RTTI。通过一个实例，读者在这里能够学到 RTTI 在 Delphi 中起到了什么重要作用，以及如何获得 RTTI、如何通过 RTTI 检查一个对象是否具有某个特性等。

第 7 章　编写自己的组件

在 Delphi 中可以轻松地编写自定义的组件，而其他编程环境则很难做到这一点。这是由于 Delphi 能够把自定义组件应用到程序中，使用户能够完全控制应用程序的用户界面。使用自己编写的控件能够使应用程序满足自己的要求。

在这一章里，读者将学习到设计组件的基本概念，以及如何把组件集成到 Delphi 环境中去。另外，本章还会介绍一些提示和技巧。

即使只是一个应用程序开发者，不是组件编写者，也会从本章中学到许多有用的东西。尤其是当现有的组件总是不能完全满足特殊的需要的时候，在应用程序中使用自定义的组件往往比较有意思。同时，组件应当设计得比较巧妙、通用，以便让其他的应用程序也可以使用它。

7.1　组件设计基础

下面介绍编写组件的基本技巧，并通过实例演示如何使用这些技巧来设计组件。

7.1.1　编写组件的时机

并不是任何时候都需要编写自己的组件，只有在下列情况下才应该考虑编写自定义的组件：

- 希望设计一个能够被其他应用程序共享的组件。
- 希望把某部分用户界面变成逻辑上独立的对象。
- 现有的 Delphi 组件、ActiveX 控件以及所能找到的第三方组件不能满足某种特殊的需要。
- 想设计一个组件作为商品出售。
- 只是想更深入地了解 Delphi、VCL 和 Win32 API。

学习创建自定义组件的最好途径就是打开 VCL 的源代码。对于组件编写者来说，VCL 的源代码是无价之宝。编写自定义的组件看起来有些困难，其实不然，难易取决于用户自己。

7.1.2　编写组件的步骤

假如已经知道问题的所在，并且准备创建一个新的组件，接下来还应考虑关于创建一个组件的几件事情：

（1）首先，要确信确实需要创建一个独特的新组件。

（2）然后，坐下来好好规划这个组件的工作方式。

（3）先做好准备工作，不要一下子就实际创建组件。先要搞清楚这个组件到底要做什么。

（4）把组件从逻辑上分成几个部分。这不仅是为了把创建组件的工作模块化、简单化，同时也是为了使代码干净、组织良好。设计组件时，一定要考虑到可能会有其他程序员要基于您创建的组件派生一个新的组件。

（5）创建了一个组件后，要在一个程序中测试它，不要一下子就加到组件选项板上。

（6）最后，把组件（包括图标）加到组件选项板上。至此，新创建的组件就可以用了。

创建一个组件分为以下 6 个步骤：

（1）确定一个祖先类。
（2）创建一个组件单元。
（3）在新的组件中加入特性、方法和事件。
（4）测试这个组件。
（5）在 Delphi 环境中注册这个组件。
（6）为这个组件提供一个帮助文件。

本章将详细讨论前面的 5 个步骤，第 6 个步骤超出了本章的范围，但这并不表示这一步骤不如前 5 步重要。另外，用户应当采用适当的第三方工具来帮助制作帮助文件。

7.1.3 确定组件的祖先类

在第 6 章中，本书讨论了 VCL 的继承关系以及不同层次类的不同用途，其中介绍了 4 种基本类型的组件：标准控件、自定义组件、图像控件和非可视组件。例如，如果需要扩展一个现有的 Win32 控件，如 TMemo，应当创建一个标准控件；如果要创建一个全新的组件，应当创建一个自定义组件；如果一个组件能够在设计期用 Object Inspector 来编辑但在运行期不需要显示，应当创建一个非可视组件。表 7.1 中列出了 VCL 中的各级公共祖先类。

表 7.1 VCL 中的公共祖先类

VCL 类	描述
Tobject	直接从 TObject 继承下来的类不是组件，不过，有的对象不需要在设计期使用，例如 TIniFile，这时候就要直接继承于 TObject
Tcomponent	是非可视组件的起点，它提供了流的能力
TGraphicControl	用这个祖先类来创建一个不需要窗口句柄、但需要在屏幕上显示的组件
TWinControl	凡是需要窗口句柄的组件，应当用这个类作为祖先类
TCustomControl	这个类是从 TWinControl 继承下来的，它具有 Canvas 特性和 Paint 方法，使用户能够控制组件的外观
TCustomClassName	VCL 中有些类的特性并不是公开的，它们主要用来作为组件的祖先类。用户可以以此为祖先类创建出多个组件，每个组件只公开自己所需要的特性

（续表）

VCL 类	描述
TComponentName	一个已有的组件，例如 TEdit、TPanel 或者 TScrollBox 等。从已有的组件派生一个新的组件，可以节省很多时间，大部分自定义的组件都是这样的

应当非常深刻地理解这些不同的祖先类的作用，大部分情况下，用户会感到选择一个已有的组件作为新组件的起点是最合适的。惟一需要考虑的是选择哪个已有的组件作为起点，这需要根据实际情况而定，因此，需要把 VCL 中的所有组件都弄懂。

确定了祖先类以后，就可以开始创建一个新的组件了。7.1.4 小节将以一个示例来说明创建一个组件的各个步骤。

7.1.4　创建组件单元

这个示例中的组件名称为 TMyComponent，它从 TCustomControl 继承下来，也就是说，这个组件既有窗口句柄，又有画出自己的能力。TCustomControl 的特性、方法和事件也被这个组件所继承。

创建一个组件单元最方便的方法就是使用 Component Expert（组件专家），可以从 Delphi 中运行 Component | New Component 命令来启动它，如图 7.1 所示。

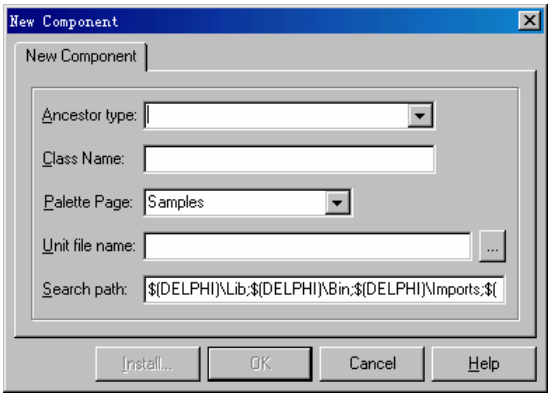

图 7.1　组件专家

在组件专家中，可以选择一个 Ancestor type（祖先类名），指定组件的 Class Name（类名），指定把这个新的组件放到组件选项板上的哪一 Palette Page（选项卡）上，以及指定组件的 Unit file name（单元名称）等。设置好以上的选项以后，单击 OK 按钮，Delphi 将自动创建这个组件的单元，其中声明了组件的类，并包含一个注册过程。下面是这个单元的代码：

```
unit MyComponent;
interface

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

```
type
    TMyComponent = class(TCustomControl)
    private
        { Private declarations }
    protected
        { Protected declarations }
    public
        { Public declarations }
    published
        { Published declarations }
    end;

procedure Register;

implementation

procedure Register;
begin
    RegisterComponents('Samples', [TMyComponent]);
end;

end.
```

到此为止，已经创建了一个新的组件，当然，这个组件现在还是空的，它什么事情也没有做，在后面的章节中，将逐步加入一些特性、方法和事件。

7.1.5 加入新的特性

在第 6 **章中，本书介绍了特性的用法，这里将介绍如何在组件中加入各种类型的特性，包括简单类型、枚举类型、集合类型、对象和数组类型等。在** Object Inspector **中不同类型的特性的编辑方式是不同的。**

1．加入简单类型的特性

简单类型的特性是指数字、字符串和字符。用户可以在 Object Inspector **中直接修改它们的值，而不需要特殊的访问方法。这里，向** TMyComponent **组件中加入** 3 **个简单类型的特性：整型、字符串和字符，如下所示。**

```
    TMyComponent = class(TCustomControl)
    private
        { Private declarations }
        FIntegerProp: Integer;
        FStringProp: String;
        FCharProp: Char;
    protected
```

```
        { Protected declarations }
public
        { Public declarations }
published
        { Published declarations }
        property IntegerProp: Integer read FIntegerProp write FIntegerProp;
        Property StringProp: String read FStringProp write FStringProp;
        property CharProp: Char read FCharProp write FCharProp;
end;
```

在第 6 章中已经讨论过了声明特性的语法，因此，读者对此应当比较熟悉了，如果仍然有不清楚的问题的话，建议返回上一章，将相应的内容再看一遍。

组件的内部数据是在 Private 部分声明的，而特性是在 Published 部分声明的，这部分中的特性在用户把组件安装到 Delphi 的环境中以后，可以用 Object Inspector 直接编辑它们。图 7.2 中显示了 TMyComponent 组件中上面 3 个简单类型的特性。

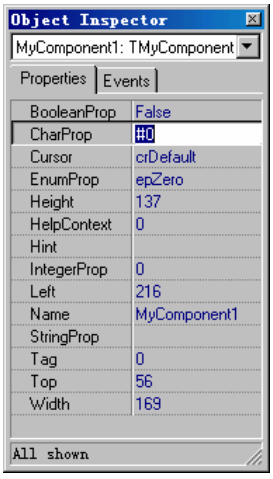

图 7.2　简单类型特性

注意：编写组件的时候，命名组件内部的数据字段最好用“F”开头，而组件和类最好用 T 开头，这样，程序代码就比较容易阅读。

2．加入枚举类型的特性

枚举型或者布尔型的特性在 Object Inspector 中可以通过反复双击相应的栏，直到需要的值出现为止，也可以从一个下拉列表框中选择一个值。大部分可视的组件都有这个特性。要在组件中加入枚举类型的特性，首先要声明一个枚举类型：

```
TEnumProp = (epZero, epOne, epTwo, epThree);
```

然后还需要声明一些内部的字段来代表每一个枚举值。下面列出了 TMyComponent 组件中的两个枚举型特性：

```
type
    TEnumProp = (epZero, epOne, epTwo, epThree);
    TMyComponent = class(TCustomControl)
    private
        { Private declarations }
        …
        FEnumProp: TEnumProp;
        FBooleanProp: Boolean;
    protected
        { Protected declarations }
    public
        { Public declarations }
    published
        { Published declarations }
        …
        property EnumProp: TEnumProp read FEnumProp write FEnumProp;
        property BooleanProp: Boolean read FBooleanProp write FBooleanProp;
    end;
```

将该组件安装到 Delphi 组件选项板中以后，枚举型的特性将出现在 Object Inpector 中，如图 7.3 所示。

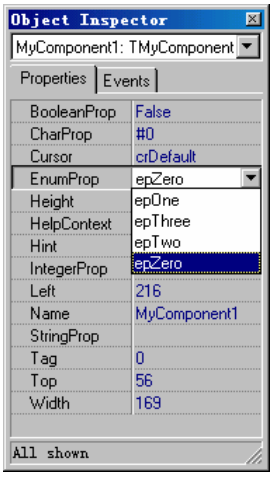

图 7.3　枚举类型特性

3．加入集合类型的特性

要在 Object Inspector 中编辑集合型特性，可以把这个特性展开，这样，集合的每个元素就好像 Boolean 类型的特性一样。典型的集合型的特性是 Options。

要加入一个集合型的特性，首先要声明一个集合类型，如下所示：

```
TSetPropItems = (PoZero, poOne, poTwo, poThree, poFour);
```

```
TSetPropOption = set of TSetPropItems;
```

上面，首先声明了一个枚举类型 TSetPropItems**，列出了集合中可能的元素。现在把这个** TSetPropOption **加到组件** TMyComponent **中，如下所示：**

```
type
    TSetPropOption = set of TSetPropItems;
    TMyComponent = class(TCustomControl)
    private
        { Private declarations }
        …
        FOptions: TSetPropOption;
    protected
    public
        { Public declarations }
    published
        { Published declarations }
        …
        property Options: TSetPropOption read FOptions write FOptions;
    end;
```

图 7.4 **显示了在** Object Inspector **中这个特性** Options **展开的样子。**

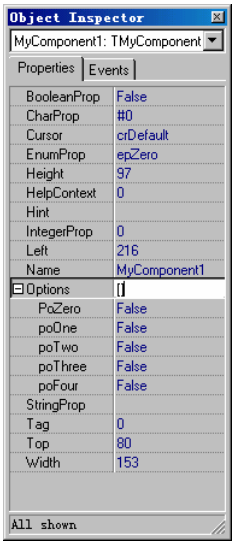

图 7.4　**集合类型的特性**

4．加入对象类型的特性

特性可以是对象甚至是另一个组件。例如，TShape **组件的** Brush **特性和** Pen **特性就分别是** TBrush **对象和** TPen **对象。典型的对象类型特性还有** Font **字体特性。如果一个特性是对象，它就可以在** Option Inspector **中被展开，这样就可以进一步编辑对象本身的特性；如**

果特性是对象类型，该对象必须是从 persistent 或其派生类继承下来的，只有对象的公开特性才能够被流操作并显示在 Object Inspector 中。

下面的示例将向 TMyComponent 中加入一个自定义的对象型特性 TMyObject。首先必须先声明该对象，如下所示：

```
TMyObject = class(TPersistent)
    private
      FProp1: Integer;
      FProp2: string;
    public
      procedure Assign(Source: TPersistent);
    published
      property Prop1: Integer read FProp1 write FProp1;
      property Prop2: string read FProp1 write FProp2;
end;
```

这个对象本身还包含两个特性：Prop1 和 Prop2，它们都是简单类型的特性。

现在，可以向 TMyComponent 中加入这个 TMyObject 对象特性了。不过，由于这个特性是一个对象，因此，必须创建这个对象的实例，否则，当用户把 TMyComponent 组件放到 Form 上，就无法编辑这个特性了。要创建这个对象的实例很简单，只有重载 TMyComponent 组件的 Create()方法，下面是具体实现的代码：

```
type
    …
    TMyObject = class(TPersistent)
        private
          FProp1: Integer;
          FProp2: string;
        public
          procedure Assign(Source: TPersistent);
        published
          property Prop1: Integer read FProp1 write FProp1;
          property Prop2: string read FProp1 write FProp2;
    end;

    TMyComponent = class(TCustomControl)
    private
        { Private declarations }
        …
        FMyObject: TMyObject;
        procedure SetMyObject(Value: TMyObject);
        procedure TMyObject(const Value: TMyObject);
    protected
```

```
        { Protected declarations }
    public
        { Public declarations }
        constructor Create(AOwner: TComponent); override;
        destructor Destroy; override;
    published
        { Published declarations }
        …
        property MyObject: TMyObject read FMyObject write SetMyObject;
    end;

procedure Register;

implementation

procedure Register;
begin
    RegisterComponents('Samples', [TMyComponent]);
end;

{ TMyComponent }

constructor TMyComponent.Create(AOwner: TComponent);
begin
    inherited;
    FMyObject := TMyObject.Create;
end;
destructor TMyComponent.Destroy;
begin
    FMyObject.Free;
    inherited;
end;

procedure TMyComponent.SetMyObject(Value: TMyObject);
begin
    if Assigned(Value) then
        FMyObject.Assign(Value);
end;

procedure TMyComponent.TMyObject(const Value: TMyObject);
begin
    FMyObject := Value;
end;
```

```
procedure TMyObject.Assign(Source: TPersistent);
begin
    if Source is TMyObject then
    begin
        FProp1 := TMyObject(Source).Prop1;
        Fprop2 := TMyObject(Source).Prop2;
        inherited Assign(Source);
    end;

end.
```

这里重载了 Create()和 Destroy()方法，这是为了在释放 TMyComponent 组件时保证要释放 TMyObject 的实例。另外，这里用 SetMyObject()方法来设置 SomeObject 特性的值，这就是所谓的写方法，相应的，读访问方法也叫做读方法。

在第 2 章“Object Pascal 语言”中，本书讨论过对象实例实际上是一个指针，当把一个对象赋给一个变量的时候，只是使指针指向另外一个对象而已，而原来的对象仍然还在内存中。因此，在设计组件的时候，应该避免用户直接访问内部特性，而要通过特性的写方法来设置特性。上例中 MyObject 的写方法就是 TMyComponent.SetMyObject(Value: TMyObject)，该方法调用了 TMyObject.Assign()方法，并且传递了新的对象参数给它。

TMyObject.Assign()方法首先检查用户是否传递了一个合法的 TMyObject 对象实例。然后逐个复制参数对象的特性。

在一个特性的写方法中千万不要直接对特性赋值，这将导致无限循环。例如：

```
property SomeProp: integer read FSomeProp write SetSomeProp;
    …
procedure SetSomeProp(Value: integer);
begin
    SomeProp := Value;    //这将导致无限循环
end;
```

这是由于写 SomeProp 特性会调用 SetSomeProp()方法，而 SetSomeProp()方法又去写 SomeProp 特性，这就导致了无限循环。

5．加入数组类型的特性

如果特性本身包含若干歌相同类型的项，这些项都有相关的序号，就像数组一样，典型的数组型特性是 TDBGrid.Columns。编辑这种特性需要专门的编辑器，下面将讨论如何向一个组件中加入一个数组类型的特性。

数组类型的特性和其他类型特性的不同在于：

- 数组型特性要带下标。下标必须是简单类型，例如整型、字符串，但不能是记录或类。
- 数组型特性必须带读写访问方法，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。

这里用一个新的示例 TPlanets 来介绍数组型特性。这个组件有两个特性：PlanetName 和 PlanetPos。PlanetName 特性就是一个数组型的特性，它的下标是一个整数，代表各个行星的名称；PlanetPos 特性也是一个数组型的特性，它的下标是一个字符串，代表行星的名称，PlanetPos 特性就是行星的位置。下面是 TPlanets 组件的代码：

```
unit planet;

interface

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

type
    TPlanet = class(TComponent)
    private
        { Private declarations }
        function GetPlanetName(const AIndex: integer): String;
        function GetPlanetPos(const APlanetName: string): integer;
    protected
        { Protected declarations }
    public
        { Public declarations }
    published
        { Published declarations }
        property PlanetName[const AIndex: integer]: string read GetPlanetName; default;
        property PlanPos[const APlanetName: string]: Integer read GetPlanetPos;
    end;

procedure Register;

implementation

const
    PlanetNames: array [1..9] of String[7] = ('Mercury', 'Venus', 'Earth',
        'Mars', 'Jupiter', 'Saturn', 'Uranus', 'Neptune', 'Pluto');

procedure Register;
begin
    RegisterComponents('Samples', [TPlanet]);
end;
```

```
{ TPlanet }

function TPlanet.GetPlanetName(const AIndex: integer): String;
begin
        result := PlanetNames[AIndex];
end;

function TPlanet.GetPlanetPos(const APlanetName: string): integer;
var
        i: integer;
begin
        i := 0;
        repeat
                inc(i);
        until CompareStr(UpperCase(APlanetName), UpperCase(PlanetNames[i])) = 0;
        Result := i;
end;
end.
```

上面的代码演示了两种数组型的特性：一种是以整数为下标，另一种是以字符串为下标。

注意：这些特性的值是由读方法返回的，而不是由内部字段返回的。

6．默认值

要使一个特性有默认值，可以在组件的 Create()**方法中对特性赋值，假如在** Create()**方法中有如下的代码：**

```
FIntegerProp := 100;
```

当这个组件被创建以后，FIntegerProp **特性的值就默认为“** 100 **”了。**

下面来解释指示字 Default **和** NoDefault**。可以发现，有些特性使用** Default **指示字来声明，例如：**

```
property Tag: Integer read FTag write FTag default 100;
```

不要误以为 Tag **特性的默认值是“** 100 **”，这个值只是影响当保存包含当前组件的窗体时，是否要保存这个特性的值。如果** Tag **特性的值不是“** 100 **”，就把这个特性的值保存到** DFM **文件中，否则，就不保存这个值。建议尽量使用** Default **指示字来声明特性，因为这能够加快打开窗体的速度。**

注意：Default **指示字并没有设置特性的默认值。要设置默认值，必须在组件的** Create() **中进行。**

不管特性的值是多少，NoDefault **指示字的作用时保存窗体时都将保存特性的值。其实一般情况下不需要使用** NoDefault **指示字，因为不使用这个指示字和使用它的效果是一样的。**

7.1.6　向组件中加入事件

事件是一种特殊的特性，当某个动作发生时，就会执行预先指定的代码。这一节将更详细地讨论事件，包括事件是如何产生的，以及怎样在自定义的组件中加入自定义的事件。

事件有可能产生于操作系统、用户的操作甚至程序代码本身。Delphi 的事件机制就是把事件与特定的代码相联系，当事件发生的时候，相应的代码就被调用。处理事件的代码实际上就是一个方法，称为 Event Handle（事件句柄）。

例如，当用户单击鼠标的时候，一个 WM_MOUSEDOWN 消息就被发送给 Win32 系统，Win32 系统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。这个控件首先将会检查相应的事件特性是否指向了一段代码，即事件句柄。如果是的话，就执行这个事件句柄。

事件特性实际上是一个方法指针。作为一个组件编写者，必须自己声明事件特性以及事件的调度方法。而组件的使用者则只需要建立事件句柄。

在声明一个事件特性之前，需要确定是否真的需要一个特殊的事件。很多情况下，自定义的组件往往是从一个已有的组件继承下来的，也就是说，自定义的组件已经继承了一些事件，除非已有的事件不能满足要求，才需要定义自己的事件。所以，最好先熟悉一下 VCL 中已有的事件。

下面将介绍怎样在自定义组件中加入自定义的事件。

假设一个组件 TMyTimer，它每一秒触发一次事件，传递了一个参数，该参数表示该组件启动后的秒数。这个组件的功能并没有太大的意义，这里只是为了说明如何加入自定义的事件。

首先，需要定义一个事件句柄，如下所示。

```
TTimeEvent = procedure(Sender: TObject; Times: Integer) of Object;
```

可以看出，TTimeEvent 是一个过程，传递了两个参数：Sender 和 Times，后面加上 of Object，表示这个过程是一个方法。这样，调用这个过程时会隐含传递一个 Self 参数，这个参数代表这个方法实际所在的对象，如下所示：

```
unit MyTimer;

interface

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

type
    TTimeEvent = procedure(Times: Integer) of Object;

    TMyTimer = class(TComponent)
    private
        { Private declarations }
```

```
        FTimer: TTimer;
        FOnTimer: TTimeEvent;
        FTimes: Integer;

        procedure FOnTime(Sender: Tobject);
    protected
        { Protected declarations }
    public
        { Public declarations }
        constructor Create(AOwner: TComponent); override;
        destructor Destroy; override;

        procedure Start;
    published
        { Published declarations }
        property OnSecond: TTimeEvent read FOnTimer write FOnTimer;
    end;

procedure Register;

implementation

procedure Register;
begin
    RegisterComponents('Other', [TMyTimer]);
end;

{ TMyTimer }

constructor TMyTimer.Create(AOwner: TComponent);
//用于创建 TTimer 的实例，然后建立处理它的 OnTimer 事件的句柄。
begin
    inherited;
    if not (csDesigning in ComponentState) then begin    // 避免在设计期对组件进行操作
        FTimer := TTimer.Create(nil);
        FTimer.Interval := 1000;
        FTimer.OnTimer := FOnTime;
    end;
end;

destructor TMyTimer.Destroy;
begin
    FTimer.free;
```

```
        inherited;
end;

procedure TMyTimer.FOnTime(Sender: Tobject);
begin
        inc(FTimes);
        if Assigned(OnSecond) then //检查组件使用者是否已经建立了 OnSecond 的事件句柄
OnSecond(FTimes);
end;

procedure TMyTimer.Start;
begin
        FTimes := 0;
        FTimer.Enabled := True;
end;

end.
```

当建立一个自定义的事件时，一定要考虑事件句柄要有哪些参数，也就是说，要提供哪些信息给组件使用者。例如，OnKeyDown()事件的句柄就传递了 Sender: TObject 和 Var Key: Char 两个参数，这两个参数不但使用户可以知道是哪个对象触发了这个事件，而且还知道用户按下的是什么键。如果深入到 VCL 中的话，就会发现，这个事件实际上是由 WM_CHAR 消息触发的。Delphi 会把必要的信息以参数的形式传递给组件的使用者。

注意：KeyDown 方法中的参数 Key 是用 Var 声明的，那么为什么不直接用一个函数来返回这个参数值呢？这是因为，如果把事件句柄设计为函数将引发不确定性。例如，当把一个函数类型的方法指针赋值给一个事件特性时，被赋值的是函数指针还是函数的返回值呢？这就是不确定性。

7.1.7　加入自定义的方法

把方法加到组件中与把方法加到一般的对象中没有什么区别。不过，当设计一个组件的方法时需要考虑下面的这些规则。

1．不要相互依赖

要创建一个自定义的组件，一个关键的问题是要使最终用户使用起来很简单。因此，应当避免程序方法之间的相互依赖。例如，不能强迫用户非要调用某个方法，也不能使一个方法的调用与另一个方法的调用顺序有关；另外，一个方法调用后不能使组件的其他方法和事件无法使用；最后，应当给方法取一个有意义的名称，使用户从名称上就能了解到这个方法大致是作什么用的。

2．方法的可见性

必须确定方法是在 private、public **还是在** protected **部分声明。不但要考虑到使用这个组件的最终用户，还有考虑到那些以此组件作为祖先类派生出其他组件的用户。表** 7.2 **中列出了** Private、Protected、Public **和** Published **部分的区别。**

表 7.2　Private、Protected、Public **和** Published **的区别**

关键字	描　述
Private	这部分的成员对派生类来说是不能访问的
Protected	这部分的成员对派生类是可访问的，但对组件的使用者来说是不可访问的
Public	要使其他对象能够访问组件的成员，并且只需要在运行期而不是在设计期访问它们，这些成员就只有在这个部分中声明
Published	这部分中声明的特性可以出现在 Object Inspector 中

7.1.8　构造和析构

当用户创建一个新组件的时候，可以重载它的祖先类的构造，从而定义自己的构造函数。这里，需要用到 override **关键字。如下所示：**

```
TSomeComponent = class(TComponent)
Private

Protected

Public
        constructor Create(AOwner: TComponent); override;
        destructor Destroy; override;
published

end;
```

注意：TComponent **的** Create **方法本身是虚拟的，它的构造是不能重载的。**

关键字 override **从语法上讲可以省略，但是这样有可能引来麻烦。这是因为，当使用这个组件时，无法通过类来引用这个构造。**

在重载祖先类的构造时，务必要调用祖先类的构造，否则将引发无法预料的异常错误。例如：

```
constructor TSomeComponent.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);
  …
end;
```

构造是当组件被创建时调用的，包括用户在设计期把组件放在 Form 上。在组件正在被创建的时候，应当避免对组件进行任何操作，否则可能会引发 Delphi 环境的不稳定。要防止在设计期对组件的操作，可以通过检查组件的 ComponentState 特性来判断组件当前的状态。表 7.3 中列出了组件的各种状态。

表 7.3　组件的状态

状态标志	状态
CsAncestor	组件是以祖先的形式引入的
CsDesigning	处于设计状态，即组件在 Form 上，被 Form 设计器操纵
CsDestroying	组件将要被删除
CsFixups	组件被连接到一个还没有打开的 Form 上
CsLoading	正在从文件中调入内存
CsReading	正在从一个流中读特性值
CsUpdating	组件正在更新
CsWriting	正在把特性写到一个流中

最经常用到的是 CsDesigning，它表示组件正处于设计状态。在前面的示例中有以下的一段代码，如下所示：

```
  inherited;
  if not (csDesigning in ComponentState) then begin
// 避免在设计期对组件进行操作
      FTimer := TTimer.Create(nil);
      FTimer.Interval := 1000;
      FTimer.OnTimer := FOnTime;
  end;
```

其中就利用了组件的设计状态来避免在组件创建时进行内部组件 FTimer 的创建。

注意：重载构造时，通常是首先调用 inherited Create()方法。而重载析构时，通常是最后调用 inherited Destroy。这确保了在用户修改它之前类已经被创建了，在所有操作结束之后再释放类。

7.1.9　注册组件

所谓注册组件，就是让 Delphi 知道把哪个组件放到组件选项板上。如果是用组件专家创建的组件，就不需要自己去注册组件，因为 Delphi 已经自动生成了有关的代码。如果组件是手工创建的，就需要把 Register()过程加到组件的单元中。用户所要做的就是把 Register()过程加到组件的单元的 Interface 部分。

在 Register()过程中，每注册一个组件，就要调用一遍 RegisterComponent()过程，这个过程包含两个参数：第一个参数用于指定要把组件加到组件选项板上的哪一页上，第二个参数用于指定组件的类型，如下所示：

```
…
```

```
procedure Register;

implementation

procedure Register;
begin
    RegisterComponents('Other', [TMyTimer, TOtherComponent]);
end;
```

这个示例注册了两个组件：TMyTimer **和** TOtherComponent**，并把它们放到组件选项板上的** Other **页上。**

7.1.10 组件图标

要提供给自定义组件一个图标，可以使用 Delphi **附带的** Image Editor **或其他位图工具来创建一个** 24*24 **的位图，并把它保存到一个** DCR **文件中。**DCR **和** RES **文件一样，也是资源文件。**

创建了一个位图之后，还要给这个位图命名。位图的名称必须与组件的类名相同，但要全部改为大写。DCR **文件的名称必须与组件的单元名称相同。而且，位图文件必须与组件的单元文件放在同一个命令中。**Delphi **在编译这个单元时，会自动把位图资源加到组件库中。**

7.2 定 制 组 件

现在读者已经了解了创建一个组件的所有步骤，并且已经可以从头开始创建一个自己的组件了。实际情况中，往往需要在一个现存组件的基础上进行一些特别的设置来产生符合自己要求的组件。例如，在 TEdit、TPanel、TLabel **的基础上定制自己的组件，对这些组件所作的改变包括：改变它们的颜色以及字体等。这一节的目的是为了介绍如何创建一组定制组件，并将它们放在组件选项板上作为特殊的效果来使用，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。有了** 7.1 **节的基础，本节中给出的示例显得非常简单。**

您依然可以从组件专家 Component Expert **开始创建一个新的组件，它的祖先为** TEdit**，名称为** TMyEdit**。这里，以下的示例重载了组件的构造函数，里面对** TMyEdit **的颜色和字体进行了特殊的设置。**

```
unit MyEdit;

interface

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

type
```

```
    TMyEdit = class(TEdit)
    private
        { Private declarations }
    protected
        { Protected declarations }
    public
        { Public declarations }
        constructor Create(AOwner: TComponent); override;
    published
        { Published declarations }
    end;

procedure Register;

implementation

procedure Register;
begin
    RegisterComponents('Samples', [TMyEdit]);
end;

{ TMyEdit }

constructor TMyEdit.Create(AOwner: TComponent);
begin
    inherited;
    Color := clBlue;
    Font.Color := clYellow;
    Font.Name := '宋体';
    Font.Size := 12;
    Font.Style := [fsBold];
end;

end.
```

安装这个组件以后，就可以在组件选项板的 Sample **选项卡上看到一个新的组件** TMyEdit**，默认情况下，它的图标将是和** TEdit **相同的图标。当把它放到一个窗体上的时候，** TMyEdit **的外观、颜色和字体与普通的** TEdit **不同，但用户依然可以对它们进行设置。**

7.3　复 合 组 件

像创建定制组件一样，可以将几个通常是组合在一起的组件来生成一个单一的组件。例如，在一个 Panel **上放置一个或者多个按钮、或者复选框，然后将它们合并成一个组件。**

下面是一个复合组件的实例，它将两个 RadioButton 合并到了一个 Panel 上，如图 7.5 所示。

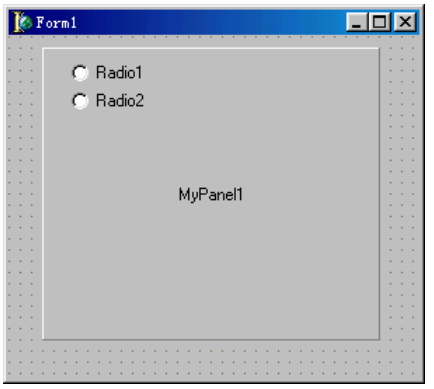

图 7.5　TMyPanel 组件

TMyPanel 组件是从 TPanel 继承下来的，TRaidoButton 组件的实现是在对 Create 构造函数的重载中完成的。组件代码如下所示，代码之后将对实现的关键部分进行解释。

```
unit MyPanel;

interface

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

type
    TMyPanel = class(TPanel)
    private
        { Private declarations }
        FRadio1: TRadioButton;
        FRadio2: TRadioButton;
        function GetRadio1Caption: string;
        function GetRadio2Caption: string;
        procedure SetRadio1Caption(const Value: string);
        procedure SetRadio2Caption(const Value: string);
    protected
        { Protected declarations }
        procedure WmSize(var msg: TMessage); message wm_size;
    public
        { Public declarations }
        constructor Create(AOwner: TComponent); override;
```

```
        destructor Destroy; override;
        property radio1: TRadioButton read FRadio1;
        property radio2: TRadioButton read FRadio2;
    published
        { Published declarations }
        property Radio1Caption: string read GetRadio1Caption
Write SetRadio1Caption;
        property Radio2Caption: string read GetRadio2Caption
Write SetRadio2Caption;
    end;

procedure Register;

implementation

procedure Register;
begin
    RegisterComponents('Samples', [TMyPanel]);
end;

{ TMyPanel }

constructor TMyPanel.Create(AOwner: TComponent);
begin
    inherited;
    Width := 175;
    Height := 60;

    FRadio1 := TRadioButton.Create(self);
    FRadio1.parent := self;
    FRadio1.Caption := 'Radio1';
    FRadio1.Left := 20;
    FRadio1.Top := 10;
    FRadio1.Show;

    FRadio2 := TRadioButton.Create(self);
    FRadio2.parent := self;
    FRadio2.Caption := 'Radio2';
    FRadio2.Left := 20;
    FRadio2.Top := 30;
    FRadio2.Show;
end;
```

```
destructor TMyPanel.Destroy;
begin
    FRadio1.Free;
    FRadio2.Free;
    inherited;
end;

function TMyPanel.GetRadio1Caption: string;
begin
    Result := FRadio1.Caption;
end;

function TMyPanel.GetRadio2Caption: string;
begin
    Result := FRadio2.Caption;
end;

procedure TMyPanel.SetRadio1Caption(const Value: string);
begin
    Radio1.Caption := Value;
end;

procedure TMyPanel.SetRadio2Caption(const Value: string);
begin
    Radio2.Caption := Value;
end;

procedure TMyPanel.WmSize(var msg: TMessage);
begin
    inherited;
    FRadio1.Width := Width - (FRadio1.Left + 10);
    FRadio2.Width := Width - (FRadio2.Left + 10);
end;

end.
```

在这里，RadioButton **实际上是作为** private **数据来声明的，通过** Radio1 **和** Radio2 **属性来与之关联，这就是** Delphi **表现出来的一种称为聚合的东西。聚合是一种多重继承的替代方法，它不仅继承了** TPanel **的所有特性，而且也继承了** TRadioButton **的特性。例如用户在这里可以设置** Panel **的边框以及边框外观等。从所有的意图和目的来说，聚合与多重继承都是同一种东西，只是聚合更加清楚、麻烦更少。**

修改 RadioButton **的特性并不需要编写与** TRadioButton **特性相连的程序，可以通过** TMyPanel.Radio1 **和** TMyPanel.Radio2 **特性来对** RadioButton **进行各种允许的设置，所以不**

需要增加 write 方法。例如要对 Radio1 的 Caption 和 Checked 特性进行设置：

```
MyPanel.Radio1.Caption := 'hello';
MyPanel.Radio1.Checked := True;
```

TMyPanel 的构造函数中除继承于 TPanel 的构造以外，分别又对 Panel 的宽度和高度进行了设置，然后又创建和设置了 FRadio1 和 FRadio2 属性。会发现代码把 Self 作为所有者来传递、并且将父类设置为 MyPanel 自己。

组件中处理了一个 WMSize 消息程序，这是因为这两个 RadioButton 都有标题。该标题伸出来覆盖了组件的底部，造成一个很麻烦的问题。通过处理 WM_SIZE 消息，就可以解决这个问题。

组件的 published 部分中加入了两个组件属性：Radio1Caption 和 Radio2Caption。

```
    published
        { Published declarations }
        property Radio1Caption: string read GetRadio1Caption
Write SetRadio1Caption;
        property Radio2Caption: string read GetRadio2Caption
Write SetRadio2Caption;
```

这两个 published 属性现在就会出现在 Object Inspector 中。Delphi 向组件中增加了额外的运行类型信息，使得 IDE 能够在涉及的时候读到该信息，并且将这些属性的信息显示给用户。

当然，用户甚至可以用相同的方法给 Radio1 和 Radio2 赋予 published 的特性。但是，当第一次这么做的时候，它们并没有 Property Editor 可用，因为 Delphi 没有 TRadioButton 内嵌的 Property Editor。为了创建自己的 Property Editor，可以参考与 Delphi 一起出售的 DSGNINTE.PAS 单元。

7.4 组 件 包

在 Delphi1 和 Delphi2 中，所有在 Component Palette 中显示的组件都放在一个称为 COMBLIB.DCL 或 CMPLIB32.DCL 的库里面。到 Delphi3 的时候，这些库已经被放弃了，取而代之的是一种称为“包”的更灵活但是也更复杂的系统。

包是一个包含一个或者更多组件、对象或者功能的特殊种类的 DLL，它能被应用程序和 IDE 使用。使用包，可以把应用程序的一部分放到一个单独的模块中，以便被其他应用程序共享。包类似于动态链接库，但用法不一样，包主要用于收集组件，就像动态链接库一样，包也是在运行期动态地链接到应用程序中的，而不是在编译器链接的。由于部分代码放在 BPL(包文件)中，因此，EXE 或者 DLL 文件可以很小。

与 DLL 不同的是，包是 Delphi VCL 专用的，用其他语言编写的应用程序无法使用 Delphi 的包（C++Builder 除外）。

7.4.1 使用包的好处

使用包的最主要的原因是可以缩减应用程序和 DLL 文件的长度。Delphi 把 VCL 分成了若干个包。事实上，当编译应用程序的时候，就已经使用了其中的许多包。

许多应用程序都是通过 Internet 发布的，例如升级程序、演示程序等。通过把应用程序拆分成多个包，用户可以只更新他们所需要的部分，只需要下载程序的片断而不是整个程序。

使用包的另外一个重要原因是分发第三方组件。如果用户是一个组件销售商，必须知道怎样创建包。组件、特性编辑器、向导、专家等一般都是放在包中分发的。

7.4.2 包的类型

包有两种类型：运行时包和设计时包。运行时包用来为运行的应用程序提供一定的功能。它允许多个应用程序访问同一段代码，减少获取执行结果所需的代码量。另外，它还能缩短编译时间、节省硬盘空间。

运行时包是可以选择的，但设计时包却无法选择。如果创建了自己的自定义组件，希望将它们加到 IDE 中，必须将它们作为设计时包安装。设计时包还能为自定义组件创建特殊的属性编辑器。创建新包时，必须在 Package Options（包选项）对话框中选择包的类型：运行时包或设计时包或者两种都包括。

- 运行期包：运行期包包含了应用程序执行时所需要的代码。如果一个应用程序使用了运行期包，这个包必须随应用程序一起分发，否则，应用程序将无法运行。
- 设计期包：设计期包中包含有组件、特性编辑器、专家等，这些都是 Delphi 的 IDE 需要用到的。这种类型的包只能用于 Delphi 环境，并且不需要随应用程序一起分发。
- 运行期和设计期包：这种包即是运行期包，又是设计期包。这种包中往往不包含设计期专用的东西，例如特性编辑器、专家等。使用这种包可以简化应用程序的开发和分发。不过，如果这种包中含有特性编辑器、专家等，则应用程序就会很大。
- 既不是运行期包也不是设计期包：这种包往往并不是由应用程序或者设计期环境直接使用的，而是由其他包使用的。

7.4.3 包文件

创建包时，将生成一个 DPK 文件。编译器用它生成一个扩展名为 bpl（Borland Package Library）的可执行文件以及一个扩展名为 dcp 的二进制映象文件。包的源文件中并没有类型、数据、子程序或函数的声明，而是由包名、该包所需的其他包名及单元文件组成。

表 7.4 中列出了与包有关的文件及其扩展名。

表 7.4 包文件

扩展名	文件类型	描述
Dpk	包的源文件	这个文件是包编辑器创建的，可以看作是包的项目文件

（续表）

扩展名	文件类型	描述
dcp	编辑过的包文件	这个文件中包含包的符合信息和 IDE 需要的头信息
dcu	编译过的单元	对应于包中的每一个单元
bpl	包的代码	这就是我们所说的包，相当于 DLL。如果是运行期包，必须把它与应用程序一起发布。如果这是一个设计期包，必须把它分发给需要用它编程的程序员，同时还要分发每一个 DCP 文件或者单元文件

7.4.4　安装包

要把包安装到 IDE 中，这非常简单。首先，要把包文件放到一个合适的位置。表 7.5 中列出了各种类型的包应当放置的位置。

表 7.5　包文件的位置

包文件	位置
设计期包（*.bpl）	\Windows\System\或者\WinNT\System32\
设计期包（*.bpl）	由于包有可能是从不同的地方得到的，因此最好在\Delphi\BIN 下建立一个目录来放置设计期包文件
包符号文件（*.dcp）	这些文件应当与包文件放在同一个目录下
编译后的单元（*.dcu）	分发设计期包的时候应当同时分发这些编译过的单元。建议把它们放置在\Delphi\Lib 下

要安装包，需要使用 Component | Install Packages 命令打开 Project Options 对话框中的 Packages 选项卡，如图 7.6 所示。

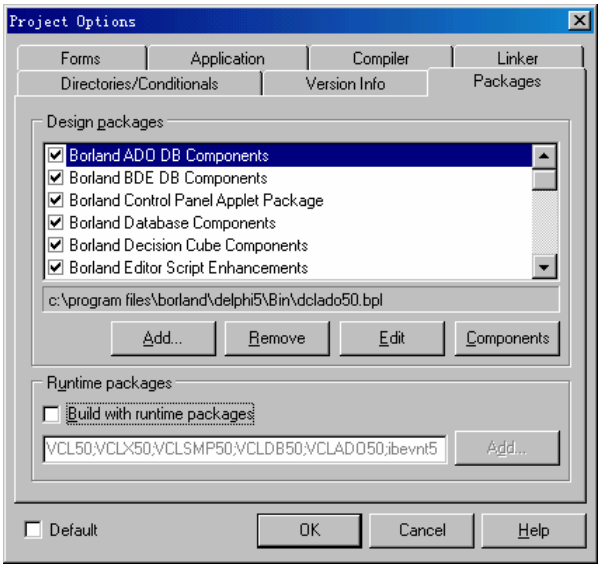

图 7.6　安装包

单击 Add 按钮，选择一个要安装的 BPL 文件。单击 OK 按钮，所选的包就安装到 IDE 中了，如果这个包中含有组件，这时候，就可以看到组件选项板上出现新的组件。

只要单击 Build with runtime packages 复选框（见图 7.6）以后，当 Delphi 编译应用程序时，运行期包中的代码就不会静态链接到 EXE 或者 DLL 中，这将使应用程序变得非常小，但应用程序的运行必须需要很多包。

7.4.5 设计包

在创建包之前，需要先确定几件事情。首先，要确定包的类型，其次，要确定怎样命名新创建的包以及把包放在什么地方。包最终分发的目标位置很可能不同于这个包被创建的初始位置。最后，要确定包中需要包含哪些单元，是否要引用其他的包。

Delphi 中提供了包编辑器来创建新的包。执行 File | New 命令，然后双击 New Items 对话框中的 Packages 图标。包编辑器由两部分组成：Contains 和 Requires，如图 7.7 所示。

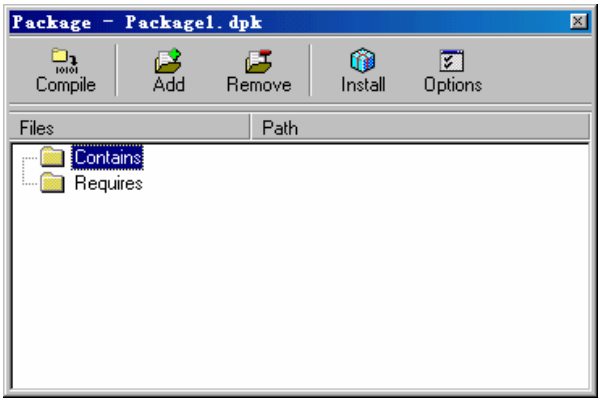

图 7.7 包编辑器（Package Editor）

包的源文件中的 Contained 子句列出了该包所包括的单元文件。在描述包中包括哪些单元时必须遵守一定的规则。所有通过 Contained 子句或用单元文件中 Use 子句包括进包中的所有单元文件都是包的一部分。

在这里，要注意以下几条规则：

- 只能包含单元，不能包含另一个包，并且包含的单元中也不能引用另一个包。
- 出现在 Contains 中的单元不能再出现在 Requires 中。
- 同一个应用程序使用的两个包不能同时包含同一个单元。

在 Requires 部分中，可以指定当前这个包还需要引用的其他包。它和 Delphi 单元中的 Uses 子句相似。任何包括在列表中的包都将与使用当前包的应用相连接。如果某个包中任何单元文件引用了其他包的单元文件，那么这些包也应包括在 Requires 子句中。

默认情况下，每个包中都需要 VCL 包，因为这个包中含有标准的 VCL 组件。典型的用法是：把所有自定义的组件放到一个运行期包中，然后再创建一个设计期包，在 Requires 子句中列出这个运行期包。注意：

- 避免循环引用。一个包不能引用自己。
- 引用链不能闭合。例如，包 A 引用了包 B，包 B 引用了包 C，那么包 C 就不能引用包 A。

上图所示的对话框用来管理用户所拥有的包（见图 7.7），这里，可以从包中添加或移掉单元文件、构建包、设置包的相关选项及安装包。保存包时 Delphi 会要求为.DPK 文件（包源文件）提供一个名称。

包的类型与实际情况有关，下面介绍 3 种可能的情况。

1．为组件创建设计期和运行期包

如果是一个组件编写者并且遇到下面的情况之一，就可能需要创建一个设计期和运行期包。

- 想使应用程序能够把组件编译并链接到应用程序种，或者把组件随应用程序一起分发。
- 不想使包中的特性编辑器、专家等内容编译到应用程序中。

2．为组件创建设计期包

当用户想分发不想在运行期包中分发的组件时就要考虑这种方案。此时，设计期包可以含有 IDE 专用的特性编辑器、组件编辑器、专家、注册单元等内容。

3．没有组件的包

有时候只是想扩展 IDE 的功能，例如安装一个新的专家。在这种情况下，可以在一个注册单元中注册用户的专家。要分发这样的包很简单，只有分发*.bpl 文件就可以了。

如果把应用程序分割成逻辑上的几个部分，每个部分就可以单独分发。下面的情况就应当分割应用程序：

- 为了便于维护。
- 用户可以只购买他们需要的功能。以后，可以增加功能，可以下载需要的包、而不是整个应用程序。
- 升级应用程序的时候，可以只下载一部分、而不是整个应用程序全面升级。

7.4.6　维护包的版本

包的版本文件比较难于理解。可以把包的版本设想为单元的版本。也就是说，某个版本的包只能在同一个版本的 Delphi 环境中使用。这样，用 Delphi 6.0 中编写的包在其他的 Delphi 5.0 或者 Delphi 4.0 环境中就无法使用。读者也许注意到了，VCL 包的版本是由代码基数决定的，用 Delphi 6.0 编写的包的代码基数就是 6.0，相应的 VCL 包就是 VCL6.0。建议用户采用这样的约定来命名自定义的包。

7.5　本 章 小 结

知道组件是如何工作的，这对于 Delphi 编程来说非常必要。这一章中详细讨论了如何

编写一个自定义的组件，如何向组件中加入简单类型、枚举类型、集合类型和对象类型的特性，以及如何加入自定义的事件和方法。本章中特别演示了几个示例，来介绍如何创建一个改变父类默认设置的组件，以及创建包括多重继承或者多集合组件的复合组件。

在包的基础上，文章接着讨论了包。包括包文件的各种类型以及安装包的方法。

不可否认，Delphi 中的基本组件开发是非常简单的，然而，它们需要一点关于 VCL 如何结合的背景知识的理解。即使是对 Windows API 基本知识的少许了解在这里都能够派上用场，特别是碰到像 WM_SIZE 这样的消息的时候。

事实上，组件开发有时候需要一点小技巧，对 VCL 和 Windows API 的知识了解得越多，组件开发工作就会做得越好。

第8章 异 常 处 理

错误处理是软件开发中必然会遇到的情况。为保证程序能在各种环境下运行，程序员必须对那些不可避免出现的错误提供处理方法。

本章将介绍如何向程序中增加错误处理的功能，它是通过一种叫做 Exception（异常）的机制来实现的。

在很大程度上，Delphi 和 VCL 使用户在编写程序时几乎可以完全忽略错误检查的问题。之所以能做到这一点，是因为大多数类和独立的例程中都内建了异常，无论哪里发生了错误都能自动触发异常。不过，专业的程序员还是希望能够超过这种安全程度，给代码添加额外的错误检查措施，或者是改变 Delphi 通过的默认的错误处理程序。而且，程序可能会需要触发它自己的错误，所以还需要添加一些异常类。

处理异常并不困难，而且，对于异常，需要有一个正确的认识，那就是对于任何程序，异常都是报告错误的正确模型。

8.1 异 常 理 论

8.1.1 错误处理方法

在 Delphi 以前的 Pascal 版本中，跟踪处理运行时的错误是非常麻烦的。对于一些操作系统的错误，程序员必须求助于编译器开关和状态变量来检测。换句话说，程序员需要告诉编译器关闭哪些代码段的出错报告，然后再检查一些全局状态变量的值来确定什么地方出了错误。

实现这一功能的代码极其烦琐，而且可读性差，难以维护。并且，过多的出错检测代码放置在代码的开头，有时比真正完成用户任务的代码还要多，以至于很难确定该代码实际要实现的功能。而下面将介绍的 Object Pascal 中的结构化异常处理将能从根本上解决上述问题。

结构化异常处理是 Object Pascal（面向对象的 Pascal 语言）的一个新的特性。它主要包括两个方面的内容：首先，它能确保在程序运行时即使发生了严重错误，仍能正确恢复已经分配或改变了的资源；其次，它提供了一种良好的结构化方式来处理包括最严重的运行错误。

利用异常，可以指定代码中的特定区域来处理错误，特别是，可以通过这种方法来保护代码的整个部分，如果其中发生了错误，那么就可以在别的区域中来处理这些问题，而不会影响到整个程序的运行。异常能够让用户的程序变得更健壮、更具可读性。

Object Pascal 中的异常语法包括资源保护和异常处理两种，分别针对于 Try…Finally 和 Try…Except 两个语法块，下面就分别介绍它们的语法和用途。

8.1.2 Try…Finally 块

Try…Finally 块的语法如下所示：

```
try
    <statement>;
    <statement>;
    . . .
finally
    <statement>;
    <statement>;
    . . .
end;
```

它以保留字 Try 开头，后面跟一行或多行代码。注意，这里在 Try 后面没有用到 Begin，这是 Object Pascal 中的一个以命令块而非 Begin 开头的语法结构。Try…Finally 块以 Finally 保留字结尾，Finally 后也可跟一行或多行代码。整个 Try…Finally 块以 End 结束。同样，这是一个以 End 结束却没有相对应 Begin 的 Object Pascal 结构，与之相似的有 Case 语句等。

Try…Finally 块能保证 Finally 部分的代码在任何情况下都将被执行。也就是说，当一个 Try…Finally 块执行时，无任是否发生错误或出现了不正常的情况，Finally 部分的代码都将执行。所以，如果没有出现任何异常，Finally 部分的代码的执行情况就如同没有 Try…Finally 代码块一样。但是，当 Try 部分的代码运行时发生异常，这种结构的作用将会显示出来。在这种情况下，Finally 部分的代码都将执行，也就是说，即使 Try 部分的代码发生了致命的错误，Finally 部分的代码仍会执行。

在这种功能的基础上，Try…Finally 块的作用主要体现在两个方面，第一是资源保护，第二是封闭操作符。

程序中很多情况下都存在一些保证要执行的特定代码块，例如应用程序动态创建一些东西以后，需要释放这些对象所占用的内存。然而，如果异常在分配内存和释放内存之间被引发，那么释放内存的代码有可能永远都不会执行。为了保证不会发生这种情况，就要用到 Try…Finally 块，也就是说，它为程序提供了资源保护。

首先介绍一些有关 Windows 资源的背景知识。在 Windows 应用中，所有资源的分配和使用都是由 Windows 来管理的。当应用程序对某种资源（例如内存、图形句柄等）有更多的需求时，它会向 Windows 提出请求，并由 Windows 分配。这些资源会在应用结束时被释放掉。

如果分配资源的 Windows 应用程序没有在适当的时侯把资源释放掉（称为资源的“泄漏”），那么该资源将一直被占用直到整个应用程序结束。例如，如果某个程序向 Windows 申请了一个图形句柄（以便在屏幕上绘图）但却没有将其释放，那么可用的图形句柄将会减少。

Try…Finally 块能为 Delphi 应用程序中的资源保护提供几乎完美的解决方案。事实上，在 Delphi 应用中有一个基本的要求：谁分配了资源，谁就有责任释放它。这同样适用于调用对象的构造函数时的资源分配。如果用户显式地调用了构造函数，那么同样有责任调用

析构函数来释放它；相反，如果构造函数是 Delphi 应用程序自己调用的，那么用户也就没有必要自己调用析构函数来释放它，因为 Delphi 应用程序会自动在适当的时候做这个工作。

举个简单的资源保护的示例。希望用户在单击一个按钮的时候显示出一个 Form，如果让工程自动创建该窗体将会浪费资源，因为该 Form 并不经常用到。所以在需要显示该 Form 时为该窗体分配资源（调用它的构造函数），比在结束时调用它的析构函数将会好得多。创建、显示和释放窗体的代码如下所示：

```
procedure ShowForm;
begin
  Form1 := TForm1.create(Application);
  Form1.ShowModel;
  Form1.Free;
end;
```

如上面的代码所示，任何时候调用了构造函数，就有责任调用析构函数。但是，如果当显示窗体时产生了一个运行错误，析构函数将不会被调用，这时，所有分配给窗体的资源（包括内存和图形资源）将会被“泄漏”。解决这个问题的方法是使用资源保护的异常处理。以下是对上面代码的改进：

```
procedure ShowForm;
begin
    with TForm1.create(Application) do try
        ShowModel;
    finally
        Free;   //无论窗体在显示时出现什么错误，这行程序最终总会执行。
    end;
end;
```

修改后的代码将更为安全和健壮。现在，无论在激活窗体时发生什么情况，Delphi 都将保证在适当的时候调用窗体的析构函数。

Try…Finally 块的用途并不仅仅限于上面提到的资源保护，当用户改变了一种资源并希望在代码模块结束时恢复它的初始状态时，Try…Finally 块将能提供极大的帮助。例如，如果在一个过程中为读取信息而打开了一个文本文件，并希望无论程序中间发生什么事情，在离开过程的时候都能保证正确地关闭该文件，这时，就可以使用 Try…Finally 块，如下所示：

```
procedure ReadFile;
begin
    Assign(TextFile，TextFileName);
    reset(TextFile);
    try
        readIn(TextFile，buf);
        ...
```

```
        finally
            CloseFile(TextFile);
end;
```

Try…Finally 块常常被称为封闭操作符，因为通过使用 Try…Finally 块，可以保证所有过程和函数（以及其他任何逻辑程序块）的用于清扫或者缮后的代码，都会被执行以便将某些东西恢复到它们原来的状态。

总之，当分配了资源或改变了应用程序并希望能恢复到初始状态时，都可以使用 Try…Finally 块。

8.1.3 Try…Except 块

异常处理的另一个方面与多数开发人员对异常处理的想法一样，就是用来处理运行时出现的错误。Try...Except 块和 Try...Finally 块的语法一样，如下所示：

```
try
    <statement>;
    <statement>;
    . . .
except
    <statement>;
    <statement>;
    . . .
end;
```

像 Try...Finally 块一样，Try...Except 块由 End 结尾而没有相应的 Begin。从功能上说，如果 Try 部分的代码没有产生任何错误（即为产生运行错误），Except 部分的代码块将被忽略。在了解错误出现后如何处理之前，先介绍一些基础的 Delphi 中异常类的背景知识，以及更加详细的异常类的信息。

在 Delphi 中，就像 Delphi 中其它的类一样，异常是一种类。异常类的命名习惯是在类名前加一个“E”而不是像其它的类一样在前面加一个“T”，除此以外，异常类和其它类没有什么区别。

在 Delphi 中，如果发生了一个运行错误，Delphi 的运行环境将会产生一个和运行异常相匹配的异常类。当一个运行错误发生时，Delphi 会触发一个异常。代码执行流程将会跳出发生错误的地方并立即转向 Try...Except 的 Except 部分。Try...Except 块下的每一个入口都会被检测以确定异常的类型。匹配了异常类型后，程序的执行将转向到该异常类型下。一旦错误处理代码被执行，由 Delphi 产生的异常对象将自动被解析，同时程序将转到紧接着整个 Try…Except 块后的第一条语句执行。匹配了异常的类型后，该异常将被处理，正确的代码将被执行。

另外，因为 exception 类是所有其它异常类的基类，能够替代所有其它异常类，所以，仅用这个基类就可以捕捉到各种异常类。

下面的代码介绍了如何配置一个 Try…Except 模块，这个模块提供了一个处理错误的地方，如下所示：

```
procedure TForm1.Button1Click(Sender: TObject);
var
    i, j, k: Integer;
begin
    j := 0;
    try
        K := i div j;
        ShowMessage(IntToStr(K));
    except on EDivByZero do
        MessageDlg('Error', mtError, [mbOK], 0);
    end;
end;
```

这段代码的运行结果是出现一个 Error **的对话框，而不是显示** K **的值，因为，当执行时，系统将发生一个被零除的错误。在错误发生时，将执行在** Except **后面的代码，这里只是显示了一个** Error **的对话框，完全可以向程序的使用者显示一些更具体的提示，例如“除数不能为零”等。**

异常的发生与代码中使用的一系列函数调用的深度是没有关系的。例如，如果不是在最高级上引发一个错误，而是调用了另外一个函数，这个函数可能是调用了更深的一个函数，这个模块中的任何一个例程引发了一个错误，代码都会自动跳转到 Except **后面来执行，如下所示：**

```
procedure TForm1.Button1Click(Sender: TObject);
var
    i, j, k: Integer;
begin
    j := 0;
    try
        K := i div j;
        SubFunction1;
        SubFunction2;
        SubFunction3;
        . . .
        ShowMessage(IntToStr(K));
    except on EDivByZero do
        MessageDlg('Error', mtError, [mbOK], 0);
    end;
end;
```

上面的代码中，SubFunction1、SubFunction2、SubFunction3 **函数都不会执行到，因为程序在走到这些函数之前已经跳转到** Except **后面了。假如程序依次执行了这** 3 **个函数，如果它们当中任何一个函数引发了被零除的错误，程序都将跳转到** Except **后面；如果** SubFunction1

中调用了另外一个例程，而这个例程又调用了别的例程，这个例程引发了被零除的错误，那么代码会立即跳转到 except 后面。

在 Delphi 中，如果发生了异常，而用户又没有明确地处理它，那么就会有一个默认的异常处理程序来处理异常。而且，大多数异常都不是明确处理的。

8.1.4 混合使用资源保护和异常处理

8.1.2 和 8.1.3 小节分别单独介绍了 Try…Finally 资源保护和 Try…Except 异常处理。实际上，也可以将 Try...Finally 和 Try...Except 块组合起来使用，而且在实际应用中有很多组合使用的时候。两种类型的异常块都将像其它 Object Pascal 块一样工作，但是，不可以只使用一个共同的 Try 开始两个块，因为每个块都需要寻找属于自己的 Try，必须各自设置自己的 Try 头。下面就是一个资源保护块和异常处理块混合使用的示例：

```
function GetAverageFromFile(FileName: string): Double;
var
    inputFile: Text;
    buf: string;
    sum， numItems: integer;
begin
    sum := 0;
    numItems := 0;
    assignFile(inputFile， FileName);
    reset(inputFile);
    try
        while (not eof(inputFile)) do
        begin
          readIn(inputFile， buf);
          sum := StrToInt(buf);
          inc(numItems);
        end;

        // 包含在资源保护块中的一个异常处理块
        try
          result := sum / numItems;
        except
          on EdivByZero do
          begin
            result := 0.0
              raise;
          end; { EdivByZero }
        end; { except }
    finally
        closeFile(inputFile);
```

```
        end; {finally}
end;
```

在这个示例中，如果 numItems **为零，函数的返回值为** 0.0**，但是异常还将传播给调用者以指明发生了错误。一旦异常发生，**Delphi **会在将异常传播给调用过程之前执行** Finally **块中的代码。**Finally **块中的代码总是会在退出块之前执行。上面这个示例的代码可以确保在任何情况下，文件都将正确地关闭。**

下面再来看一个示例：

```
procedure TForm1.Button1Click(Sender: TObject);
var
        i, j, k: integer;
        S: string;
begin
        try
           S := '';

                //   开始 try. . .finally 块
                j := 0;
                try
                   k := i div j;
                   Caption := IntToStr(k);
                finally
                   S := 'Finally Called';
                end;
                S := 'After Finally';
                // 结束 try. . .finally 块

        except on EDivByZero do
                   MessageDlg('Div by Zero Error occurred.', mtError, [mbOK], 0);
        end;
        Caption := S;
end;
```

在上面这个示例中，Try…Except **块中嵌套了一个** Try…Finally **块，实际上** Delphi **程序中所有的** Try…Finally **块总是嵌套在** Try…Except **块中的，因为事实上，即使用户没有明显地调用这个块，每一个** Delphi **程序都是在** Try…Except **块中的。**

为了比较清楚地描述这种嵌套，可以把 Try…Finally **块单独写成一个函数，然后在** Try…Except **块中调用这个函数，例如上面的示例可以改写成这样：**

```
var
        S: string;
    . . .
procedure TForm1.Button1Click(Sender: TObject);
```

```
var
        i, j, k: integer;
        S: string;
begin
        try
                S := '';
                FinallySection;
        except on EDivByZero do
                  MessageDlg('Div by Zero Error occurred.', mtError, [mbOK], 0);
        end;
        Caption := S;
end;
procedure FinallySection;
begin
        j := 0;
        try
                k := i div j;
                Caption := IntToStr(k);
        finally
                S := 'Finally Called';
        end;
        S := 'After Finally';
end;
```

这段程序的运行结果是，首先显示了 Try…Except **块的异常信息** Div by Zero Error occurred.**，然后会看到窗体的** Caption **变为了** Finally Called**，它表示最终“** S **”变量的赋值是在** Finally **块中进行的，也就是说，** Try…Finally **保证了代码“** S := 'Finally Called' **”一定能够在最后执行到。**

8.1.5 异常处理的必要性

一个程序不可能没有任何异常，即使是最简单的程序，也有可能因为外部或者系统的原因而引起异常，如果在程序中不对异常进行必要的处理，很有可能导致非常严重的后果，例如在没有任何提示的情况下关闭程序，从而导致数据丢失等问题。

在 Delphi **中解决这个问题有一个很方便的方法：用一个** Try…Except **块封装整个程序，这样，无论在程序的什么地方、什么时候出现异常，它都会被截获到异常处理模块中，从而避免不可预测的错误；相反，如果没有将整个程序封装在一个** Try…Except **块中，那么异常就会制造一个雷区，它可能会出其不意地关闭程序。因此，如果将整个程序封装在一个良好的异常处理模块中，那么就只需要很少的努力，就可以很安全地完成工作。**

Delphi **的最大的好处之一就是所有** VCL **程序自动存在于一个构造良好的** Try…Except **块中，而且，整个** VCL **和大多数支持它的例程也很好地利用了异常。**

8.2　异 常 类

Delphi 中附带了大量的异常类系列，用于处理大范围的错误条件。用户也可以很容易地创建自己的异常类，来处理程序中易于受错误影响的关键事件。下面是 Delphi 的异常基类，它是在 SysUtils.pas 中声明的，如下所示：

```
Exception = class(TObject)
private
        FMessage: string;
        FHelpContext: Integer;
public
        constructor Create(const Msg: string);
        constructor CreateFmt(const Msg: string; const Args: array of const);
        constructor CreateRes(Ident: Integer); overload;
        constructor CreateRes(ResStringRec: PResStringRec); overload;
        constructor CreateResFmt(Ident: Integer; const Args: array of const);
      overload;
        constructor CreateResFmt(ResStringRec: PResStringRec; const Args:
array of const); overload;
        constructor CreateHelp(const Msg: string; AHelpContext: Integer);
        constructor CreateFmtHelp(const Msg: string; const Args: array of const;
                AHelpContext: Integer);
        constructor CreateResHelp(Ident: Integer; AHelpContext: Integer);
overload;
        constructor CreateResHelp(ResStringRec: PResStringRec; AHelpContext:
Integer); overload;
        constructor CreateResFmtHelp(ResStringRec: PResStringRec; const Args:
array of const;    AHelpContext: Integer); overload;
        constructor CreateResFmtHelp(Ident: Integer; const Args: array of const;
                AHelpContext: Integer); overload;
        property HelpContext: Integer read FHelpContext write FHelpContext;
        property Message: string read FMessage write FMessage;
end;
```

读者可能发现了，所有异常都有一个可以显示给用户的消息 Message，在后面的示例中将看到 Message 中的内容。

在 Exception 的基础上派生了很多异常类，包含有被零除、文件 I/O 错误、无效类型处理和多种其他的异常类，如下所示：

```
EAbort = class(Exception);


EHeapException = class(Exception)
```

```
private
        AllowFree: Boolean;
public
        procedure FreeInstance; override;
end;

EOutOfMemory = class(EHeapException);

EInOutError = class(Exception)
public
        ErrorCode: Integer;
end;

EExternal = class(Exception)
public
        ExceptionRecord: PExceptionRecord;
end;

EExternalException = class(EExternal);
EIntError = class(EExternal);
EDivByZero = class(EIntError);
ERangeError = class(EIntError);
EIntOverflow = class(EIntError);
EMathError = class(EExternal);
EInvalidOp = class(EMathError);
EZeroDivide = class(EMathError);
EOverflow = class(EMathError);
EUnderflow = class(EMathError);
EInvalidPointer = class(EHeapException);
EInvalidCast = class(Exception);
EConvertError = class(Exception);
EAccessViolation = class(EExternal);
EPrivilege = class(EExternal);
EStackOverflow = class(EExternal);
EControlC = class(EExternal);
EVariantError = class(Exception);
EPropReadOnly = class(Exception);
EPropWriteOnly = class(Exception);
EAssertionFailed = class(Exception);
EAbstractError = class(Exception);
EIntfCastError = class(Exception);
EInvalidContainer = class(Exception);
EInvalidInsert = class(Exception);
```

```
EPackageError = class(Exception);

EWin32Error = class(Exception)
public
    ErrorCode: DWORD;
end;
ESafecallException = class(Exception);
```

8.3　异常的实例

8.3.1　一个异常的实例

下面将以一个实例来说明异常基本语法的使用，当用户在 Delphi 中运行这个实例的时候，遇到异常，该异常便把 Delphi 编译器跳出运行状态，并带到发生异常的代码处，用户只要继续运行代码，就会看到程序中的异常报告。建议不要在 Delphi 中运行该程序，而应在编译完成后直接运行编译产生的可执行文件。

下图是该实例的程序主界面，如图 8.1 所示。

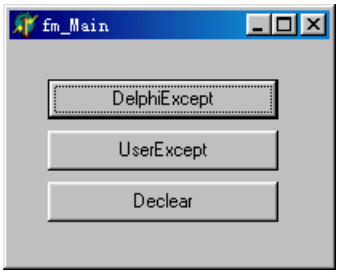

图 8.1　演示异常的程序主界面

可以看到，主窗体上有 3 个按钮，分别包含了 Delphi 的内置例程处理异常、用户自定义异常模块处理异常，以及异常信息显示等功能。代码如下所示：

```
unit Main_Except;

interface

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

type
    Tfm_Main = class(TForm)
        btnDelphiExcept: TButton;
        btnUserExcept: TButton;
```

```
        btnDeclear: TButton;
        procedure btnDelphiExceptClick(Sender: TObject);
        procedure btnUserExceptClick(Sender: TObject);
        procedure btnDeclearClick(Sender: TObject);
    private
        { Private declarations }
    public
        { Public declarations }
    end;

var
    fm_Main: Tfm_Main;

implementation

{8R *.DFM}

procedure Tfm_Main.btnDelphiExceptClick(Sender: TObject);
var
    i, j, k: Integer;
begin
    j := 0;
    K := i div j;
    ShowMessage(IntToStr(K));
    ShowMessage('Program did not get here. . .');
end;

procedure Tfm_Main.btnUserExceptClick(Sender: TObject);
var
    i, j, k: Integer;
begin
    j := 0;
    try
        K := i div j;
        ShowMessage(IntToStr(K));
    except on EDivByZero do
        MessageDlg('User got the Exception!', mtError, [mbOK], 0);
    end;
end;

procedure Tfm_Main.btnDeclearClick(Sender: TObject);
var
    i: integer;
begin
    try
```

```
            i := StrToInt('abc');
        except on E:EConvertError do
            MessageDlg('Error occurred at: ' + Self.Classname + #13#13 +
              'Type of Error: ' + E.ClassName + #13#13 +
              'Message: ' + E.Message, mtError, [mbOK], 0);
        end;
    end;

    end.
```

第一个按钮演示了 Delphi **内置的异常处理，下图为运行结果，可以会发现，**Delphi **内置的异常出现以后，该函数过程就已经中止，而没有继续运行下面的** ShowMessage **代码，如图** 8.2 **所示。**Delphi **中的大多数异常都是这样处理的，如果发生了一个错误，**VCL **就会自动捕获它，并向用户显示一个错误信息，用户不必做任何事情。**

图 8.2　Delphi **内置异常处理**

第二个按钮演示了用户自定义了一段异常模块用来处理系统发生的被零除的异常，运行结果如图 8.3 **所示：**

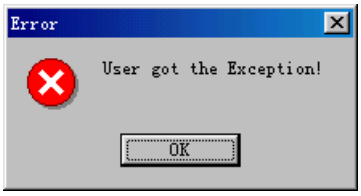

图 8.3　**自定义异常处理模块**

第三个按钮的事件中，StrToInt('abc')**代码将引发一个** EConvertError **异常，因为系统无法把字符串** abc **转换为整型值。异常发生后，程序将按照用户的要求显示异常信息，包括异常类型以及异常消息等，如图** 8.4 **所示。**

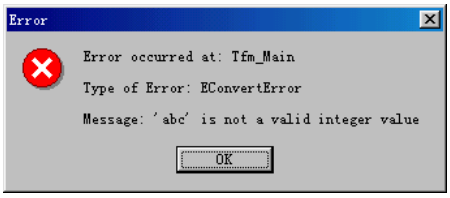

图 8.4　**异常信息**

在这里，Delphi 使用了一个标识符“E”映射到了一个已被引发的异常对象实例上。它不是一个变量声明，因为没有分配任何新的存储器给它。Delphi 只是提供了一个访问异常对象实例的方便的方法，以便用户可以引用这个对象实例所携带的附加信息，例如错误消息 E.Message 等。

注意：上例中，“E”是随便取的名字，可以用任何其他有效的变量名来代替。采用这种方式处理异常，可以通过“E”这个异常变量来访问异常对象的所有属性和方法。另外还需注意，异常对象是在处理时而不是在创建时命名的。这样做是为了避免异常对象名的范围问题。如果它在产生时命名，经过传递后，最后处理时可能会出现名字越界的情况。

8.3.2 找到异常的地址

用户可能希望显示异常引发的实际位置，下面的代码演示了如何返回这个地址：

```
procedure GetExpetionAddr;
var
    I, j, k: integer;
begin
    j := 0;
    try
        k := I div j;
    except on E:EDivByZero do
        MessageDlg(E.Message + #13#13 + 'Address:' + Format('%p', [ExceptAddr]),
mtError, [mbOK], 0);
end;
```

这段代码中使用了 VCL ExceptAddr 函数，该函数返回了异常发生的地址。其中使用 Format 函数来执行一个常规的字符串操作，它返回了一个地址的字符串型信息。

8.4 引 发 异 常

除了程序中 Delphi 自动引发的异常以外，实际情况中还有可能需要主动地引发一些异常，进而使程序进入异常处理模块中去执行。

8.4.1 引发 VCL 异常类

可以引发的最简单的异常是 VCL Exception 类。例如下面的代码：

```
Function StrToInt(const S: string): Integer;
const
    S = 'Could not convert string to integer, it's: %d';
var
    E: Integer;
```

```
begin
    Val(S, Result, E);
    if E <> 0 then
        raise EConvertError.Create(Format(S, E));
end;
```

触发异常的关键是 Object Pascal 中规定的一个异常保留字 raise。raise 有两个不同的用途。首先，raise 可以用来再次触发已经被处理的异常，其次，raise 还可以用来人为触发异常，模拟错误发生的条件。

上面的代码中，raise 语句自动创建了 Exception 类的一个示例，并用一个简单的字符串调用它的构造函数。用户可以根据自己的需要显示异常的信息。

又如下面的示例：

```
procedure ThrowVCLException;
begin
    try
        raise Exception.Create('VCL Class');
    except on E: Exception do
        ShowMessage(E.ClassName + ' ' + E.Message);
    end;
end;
```

在 8.2 小节中，列举了一个很长的程序清单，它包含了大量的 VCL 异常类，如下所示：

```
EIntError = class(Exception);
EDivByZero = class(EIntError);
ERangeError = class(EIntError);
EIntOverflow = class(EIntError);
```

在所有这些类中，它们都是从一个叫做 Exception 的类或者它的子类中派生下来的，这种多态性为程序员提供了很大的好处。所有的 VCL 异常都是从 Exception 类派生出来的，所以，利用多态性的规则，可以将它们传递给任何带有 Exception 类型参数的 except 块，而不必区分到底是哪种具体的异常类型。更重要的是，当系统收到用户传递的异常类型时，它能够正确地报告出 Exception 的具体类型，例如，本节第一个示例中，虽然类的类型声明为了 Exception 类型，依然能够正确的报告异常的名字是 EConvertError。换句话说就是，EConvertError 类型的异常可以通过 Exception 类型的变量进行传递，而且系统仍然能够正确识别。

8.4.2　创建和引发异常

通过前面章节的介绍，可以知道：在 Delphi 应用的出错处理中，异常是首选方法。但是，到目前为止，本章只是介绍了一些固定的异常类。和普通的类一样，用户也可以通过

继承固定的异常类来创建定制的异常类。Delphi 内置的很多异常类对用户会非常有用，但是很多时候，还是需要创建自己的异常类。

在 Delphi 中，没有什么东西可以迫使用户在特定的情况下使用一个特定的异常类。即使程序中没有出现任何问题，用户仍然可以引发一个特定的异常。

创建一个自定义异常只需让它成为异常类的一个子类即可，如下所示：

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

这样就创建了一个和固定的异常类一样的异常，它是 Exception 类型的派生类，这里的一行代码是该类的完全的声明和实现。如果用户不想加入自己的属性，那么上面的代码就已经足够了，但是这样做实际上不过是给异常类换了一个名字，不过，即使是这样，一个自定义的类也比仅仅依赖异常对象要好得多。因为当触发该异常时，可以通过名字来捕获它，而如果通过异常类型，那么，将会捕获到所有的异常，其中包括许多用户并不关心的异常。

当然，Delphi 不会自动触发自定义异常，这个任务必须由程序员来完成。触发异常的语法和上面触发固定异常类的语法相似，如下所示：

```
raise EMyException.create ('My Error Information');
```

同样，参数是异常传播给应用对象时将显示的消息。和固定异常一样，用户不用调用它的析构方法，一旦它被处理，Delphi 会自动析构它。

使用自定义异常的好处可能现在没有立即体现出来，但是，它们确实在事件驱动的 Windows 环境中起着非常重要的作用。

读者可以看到创建自己的异常并引发它们是非常容易的，而且，可以为任何目的创建异常类。无论什么时候，只有感觉到需要描述一种新的错误，都可以通过在前面的代码中所描述的简单方法来引发一个异常。

如果需要，可以创建一个比较复杂的异常类型。例如，Delphi 中的 EInOutError 异常在对象声明中加入了 ErrorCode 字段，如下所示：

```
  EInOutError = class(Exception)
  public
    ErrorCode: Integer;
  end;
```

这样，在 except 块中就可以引用这个错误代码了，如下所示：

```
try
  . . .
except on E: EInOutError do
    Code := E.ErrorCode;
end;
```

在引发异常的时候明确表示出不同类型的异常并不是非常必要的，但在 Try…Except 块中的 Except 部分是必要的。

引发异常最方便的方法是简单地输入下面的代码，如下所示：

```
raise Exception.Create('Error occurred');
```

这种方法在很多情况下效果都很好，然而，创建自己的异常类是最好的，这样就可以创建一个 Try…Except 块，专门处理特殊的异常类型。例如，如果希望程序忽略掉所有的 EMyException 异常，可以这样处理，如下所示：

```
try
  . . .
except on EMyException do;
end;
```

这段代码的作用是在程序遇到 EMyException 异常的时候，自动绕过它，不作任何处理，就好像这个异常从来没有发生过一样。

注意：用上面的方法来忽略掉一些异常并不是一个很好的避免异常出现的方法，因为大部分 Delphi 内置的异常如果不特意地处理，都会产生一些不好的后果，有些可能严重到导致系统崩溃。

8.4.3　再次引发异常

请看下面的代码：

```
procedure Tfm_Main.btnDeclearClick(Sender: TObject);
var
        i: integer;
begin
        try
                i := StrToInt('abc');
        except on E:Exception do begin
                MessageDlg('Error occurred.', mtError, [mbOK], 0);
                Raise;    //再次引发异常，显示详细错误信息
                end;
        end;
end;
```

在这个示例中，程序导致了一个类型转换错误，并通过异常处理向用户显示了一个“Error occurred.”的信息框，在这之后，程序使用了 raise 关键字，它在这里的作用是向用户进一步显示了标准的 EConvertError 的错误信息。这就是再次引发异常。

再次引发异常的好处是：执行一些定制好的处理后，再让 Delphi 自动确切地通知用户出错之处。在大多数的情况下，再次引发异常是一个很好的主意，因为不能保证对错误的处理是完全的。事实上，最好是让系统自动处理异常，用户在任何地方都不加干涉。如果

必须要编写 Try…Except 块来逐步进行该过程，那么最好考虑再引发异常，以便其他的例程知道它。

在这里，笔者要进一步强调一点：处理 VCL 引发异常的最好的方法就是不要特意处理它，让 VCL 自己来处理 VCL 异常。禁止 VCL 异常，或者处理它们然后不再进行再次引发异常，都不是很好的作法，除非用户可以确信忽略该异常是正确的。

8.5 高级异常处理技术

到现在为止，读者已经基本了解了 Delphi 的异常理论以及异常的处理方式，下面将在此基础上，进行更深入的异常处理技术的讨论。

8.5.1 事件驱动环境下的异常

在探讨高级异常处理技术前，首先必须了解异常在 Windows 事件驱动环境下的工作方式。同其他 Object Pascal 类一样，异常具有模块化语法结构。然而，在事件驱动环境中，理解模块化语法作用于事件的方式有时比较困难。

每一个应用程序总是尽量达到一种“休整状态”，处于该状态时，应用程序不执行任何用户代码，并时刻等待事件（来自用户或系统）的发生。事件发生后，Delphi 将执行所有与该事件相关的代码并最终返回到休整状态。假设有这样一个过程：应用程序开始处于休整状态，然后用户改变了窗体上 DBEdit 字段，接着单击 Post 按钮。下图显示了上述事件发生的过程，如图 8.5 所示。

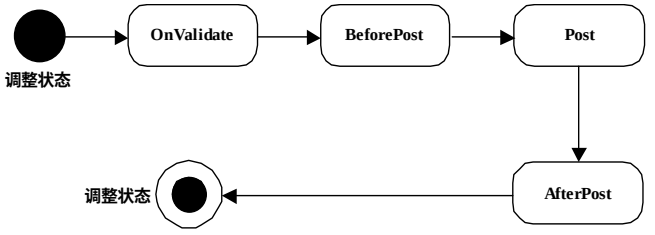

图 8.5 对应 Post 命令的事件触发的通常顺序

用户执行改变操作时，将触发 OnValidate 事件，接着触发 BeforePost、Post 事件，最后触发 AfterPost 事件，一旦这一过程结束，应用程序将返回休整状态。

使用异常可以让用户在上述事件链的任何一点生成一个异常，破坏该事件链，影响事件的发生。例如，如果 Validation 校验失败，则产生一个异常，图 8.6 显示了异常影响后的事件触发顺序图。

从图中您可以看到，异常使本应触发的事件（BeforePost、Post 和 AfterPost）不再执行，而使应用程序直接返回到休整状态。由此可见，异常能干扰事件执行的正常顺序。

无论是 Delphi 自带的还是用户自定义的异常都可以破坏事件发生的正常顺序，并可以将程序停留在某一预定位置，该位置由异常来管理。不过，程序员可以采取措施向用户发送一定的消息或执行其他的操作。

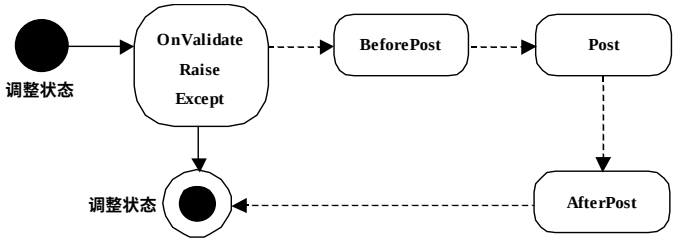

图 8.6　OnValidate 产生一个异常后的事件触发顺序

8.5.2　哑异常

Delphi 的哑异常是异常中一个非常特殊的异常类。哑异常的工作方式和传递方式与常规异常是一样的，但它并不提供一个包括可见结果信息的窗体或对话框。

EAbort 异常类是 Delphi 中所有哑异常的基类。如果希望创建一个自己的哑异常，那么必须将它作为 EAbort 的子类。哑异常除了不向终端用户显示结果外，工作方式与其他常规异常一样，也就是说，它能够破坏未执行完毕的事件链，但却是静悄悄地完成。

可以采用和生成其他异常一样的方式来生成哑异常，如下所示：

```
raise EAbort.create('异常信息，但是它不会显示出来');
```

虽然哑异常不提供对话框向用户显示信息，但用户仍然需要向构造函数中提供消息。哑异常处理方式与其他异常一样，惟一区别在于它们向用户提供的显示方式。

还有一个产生哑异常的更简便的方法，Delphi 内置的 Abort 方法能够为用户产生一个 EAbort 异常，因此，可以不用显式地调用 EAbort 异常，而直接调用 Abort 方法就可以了，如下所示：

```
if . . . then Abort;
```

8.5.3　应用对象的错误处理

前面已经提到，应用对象可以处理所有传递过来的未处理异常。对于这些异常，可以通过两个方法和一个事件：HandleException()、ShowException()和 OnException()来控制应用对象处理它们的方式。

1．HandleException()方法

HandleException 方法是应用程序处理那些传递到最外层异常时调用的指令。它通过调用 ShowException 方法，将上述异常传递给相应的异常对象。

2．ShowException()方法

ShowException 方法的功能是将异常信息显示在屏幕上的对话框中。HandleException 能自动调用该方法，但如果用户希望对某种异常规定缺省的处理方式，也可以自己手工调用它。

3．OnException()事件

OnException 事件允许用户自定义 Delphi 对最外层异常的处理方式。由于无法在对象监视器中访问应用对象（它所有的属性、方法和事件都只能在执行期可用），所以，必须书写代码指定事件的任务。于是，可以创建一个事件处理程序并接受以下两个参数：类型为 TObject 的 Sender 参数和异常对象 E 参数。下面的代码显示了事件处理程序进程和调用事件的方法：

```
procedure CustomExceptionHandler(    Sender: TObject;
E: Exception);
begin
        if E is EMyException then
                ShowMessage('Error information')
        else
                application.ShowException(E);
end;

procedure FormCreate(Sender: TObject);
begin
        application.OnException := CustomExceptionHandler;
end;
```

注意，在处理客户异常时，首先检查该异常是否是希望执行客户操作时需要的异常，如果不是，则通过调用 ShowException 将该异常传递给缺省的异常处理模块。这种作法很重要，它能保证用户正确地用自定义代码或缺省方法（例如 ShowException）来处理异常。

8.6 处理数据库异常

Delphi 中处理数据库错误同处理其他错误有些不同。除使用前面已经介绍过的所有属性外，还必须在很多方面作额外的处理。

8.6.1 EDatebaseError 和 EDBEngineError 异常

EDatabaseError 是 Delphi 中最高级别的数据库异常，所有其他的数据库异常都起源于 EDataseError。该异常会在组件发现数据库错误时产生。下面是一个打开表时处理 EDatabaseError 的示例。例中代码将反复执行 try 中的指令，直到成功地连接上表或用户在对话框中单击了 Cancel 按钮放弃开表操作为止。

```
//尝试打开数据表 tblConnTest
```

```
repeat //直至成功或用户选择 Cancel 按钮
    try
        tblConnTest.Active := True; //打开
        Break; //如果没有错误，离开循环
    except
        on EDatabaseError do
          //询问用户是否重试
          if MessageDlg('无法打开数据表。'，mtError，
            [mbOK，mbCancel]，0) <> mrOK then
            raise; // 如果不继续， 再次触发异常来放弃操作
    end; //except
until False;
```

不过，EDatabaseError 异常只提供了数据库错误的一些通用信息，并不会详细说明发生的数据库错误的类型。由于 Delphi 是通过 BDE 访问数据的，因此，大多数数据库错误将提示为 DBEngine Errors。

异常 EDBEngineError 直接由 EDatabaseError 继承而来，但它还添加了一些有用的特性，其中用途最大的特性是 Errors 数组和 ErrorCount 特性。

在某些错误条件下，BDE 可能会发送不止一个错误。例如，如果用户执行某个操作导致了 3 个不同的 BDE 错误，那么 BDE 将把所有的错误都发送回调用的应用线程，而 BDE 返回的多个错误将会放置在 Error 数组和 ErrorCount 特性共同作用的错误列表中。

8.6.2　OnPostError()、OnEditError()和 OnDeleteError()事件

OnPostError()、OnEditError()和 OnDeleteError()事件是为数据集对象（即 TTable、TQuery 和 TStoredProc）定义的。它们具有同样的参数，如下所示：

```
procedure TForm1.Table1PostError(DataSet: TDataSet; E: EDatabaseError;
    var Action: TDataAction);
```

参数 Action 规定了各种错误条件下的处理方式，可以有 3 个取值：daFail、daAbort 和 daRetry。缺省值 daFail 将生成通用的 EDatabaseError 对象，daAbort 值将产生一个哑异常而不是 EDatabaseError 异常，来消除数据库异常，daRetry 值的使用必须小心，它将重新执行产生异常的操作，重新产生同样的异常。如果用户不在指令中采取一些操作加以限制，这可能会导致无休止的循环。终止循环必须将 Action 值改为 daFail 或 daAbort。

8.6.3　错误常量

BDE 返回的错误都有自己的错误代码。Borland 公司已经将这些数值命名在一个常量目录清单里，这个清单定义于 BDE 单元文件（VCL 单元文件之一）里。在 VCL 源代码中，找不到 BDE 单元文件，只能看到 DCU 文件，但在 Delphi 6.0 下的 DOC 目录，有一个 BDE.INT 文件，它是 BDE 单元文件的接口部分，该部分中定义了所有 BDE 能返回的错误代码。这些代码用十六进制表示，并进行了分类，所有的错误代码值都由一个错误类型和错误代码组成。以下是错误分类目录列表的一部分内容：

```
ERRBASE_NONE = 0; {No error}
ERRBASE_SYSTEM = 82100; {System relate (Fatal Error)}
ERRBASE_NOTFOUND = 82200; {Object of interest Not Found }
ERRBASE_DATACORRUPT = 82300; {Physical Data Corruption}
```

定义好基础分类错误代码后，接着根据上述分类定义错误代码偏移值：

```
ERRCODE_SYSFILEOPEN = 1; {Cannot open a system file}
ERRCODE_SYSFILEIO = 2; {I/O error on a system file}
ERRCODE_SYSCORRUPT = 3; {Data structure corruption}
ERRCODE_NOCONFIGFILE = 4; {Cannot find config file}
```

将基础代码值和某个错误偏移值相加即为实际的错误代码值。例如，以下是系统错误代码列表的一部分内容：

```
DBIERR_SYSFILEOPEN = {ERRBASE_SYSTEM + ERRCODE_SYSFILEOPEN }
DBIERR_SYSFILEIO = {ERRBASE_SYSTEM + ERRCODE_SYSFILEIO}
DBIERR_SYSCORRUPT = {ERRBASE_SYSTEM + ERRCODE_SYSCORRUPT}
DBIERR_NOCONFIGFILE = {ERRBASE_SYSTEM + ERRCODE_NOCONFIGFILE}
```

这些 DBIERR_**常量是** BDE **返回的实际数值。但上述这种方法比较繁琐，而且，**BDE **返回的是十进制数，上面的数却都是十六进制数。**Delphi **还提供了一个更简单的方法，即通过对象浏览器获取实际的错误值。**

使用对象浏览器的 UNIT **接口，查看** constants **窗口，可以得到** EDBEngineError **异常实际返回值，如图** 8.7 **所示。**

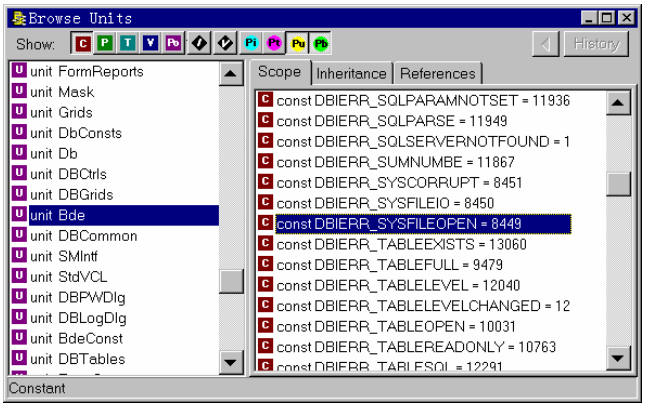

图 8.7　**显示** BDE **错误常量及其数值的对象浏览器**

通过错误常量，可以知道发生错误的具体类型。这一点很重要，因为用户绝不希望去处理不正确的数据库引擎错误。例如，考虑以下代码：

```
try
    tblConnect.active := true;
```

```
except
  on E: EDBEngineError do
        MessageDlg('不能找到数据表', mtError, [mbOk], 0);
end;
```

如果指定目录上的表实际上并不存在，或因某种原因毁坏了，那么如何处理呢？上述代码假设了某种特定类型的异常，但这种假设可能并不准确。如果使用 BDE 返回的实际的错误代码，就能确切知道发生了哪种类型的错误，如下所示：

```
try
      tblConnect.active := true;
except on E: EDBEngineError do
            case E.errors[0].errorCode of
              DBIERR_OSENOPATH: MessageDlg('错误的路径', mtError, [mbOK], 0);
              DBIERR_OSENOENT: MessageDlg('文件没有找到', mtError, [mbOK], 0);
            else
              raise;
            end;
end;
```

代码中对错误类型的检查依据于 BDE 返回的错误代码，如果返回的错误并不在设定的错误陷阱中，则重新生成一个异常，这一步非常重要。因为，出现 BDE 错误时，错误类型可能有很多种，因此，将 BDE 返回错误码假定为用户所期望的值的作法并不可靠，考虑不在设定范围以内的错误异常是很有必要的。

8.6.4　自定义数据库服务器异常

有些服务器平台（例如 InterBase）允许用户创建自定义异常并将它作为数据库的一部分。这和在 Delphi 中创建自定义异常类很相似。

InterBase **中，创建异常的语法如下所示：**

```
create exception EXCEPTION_NAME "Error message returned";
```

然后使用触发器或存储过程产生异常，如下所示：

```
exception EXCEPTION_NAME;
```

InterBase **平台上的异常具有 Delphi 中异常相同的功能，即破坏即将发生的事件链。例如，如果在 BeforePost 触发器中产生一个异常，它将阻止 Post 指令的执行。**

自定义服务器异常以 EDatabaseErrors 的形式传回 Delphi，由于其 Message 属性值与服务器异常部分定义的错误消息一样，因此，在应用程序中，可以通过为 EDatabaseError 设置陷阱、检查错误消息来判定实际发生的异常。

下面是实际应用中的一个实例。企业制定了一条商业规则，规定不能更改类型为 COPPER 的零件。于是创建了一个名为 CANNOT_CHANGE_PART_PRICE 的 InterBase 异常，声明如下所示：

```
CREATE TRIGGER UPDATE_CHECK_PART_PRICE "Cannot change the
price of parts of this type";
```

接着在 Parts **零件表中声明一个更新前触发器，如下所示：**

```
CREATE TRIGGER UPDATE_CHECK_PARTTYPE for PARTS
        BEFORE UPDATE POSITION 0
        as
begin
        if (NEW.RETALL_PRICE <> OLD.RETALL_PRICE and
                NEW.TYPE = 'COPPER') then
                EXCEPTION CANNOT_CHANGE_PART_PRICE;
end;
```

触发器首先对新旧零售价格进行比较，检查是否发生了变化，如果用户试图修改改变 COPPER **类型的零件，**InterBase **将产生异常。**

实例中的异常实际上是在 Parts **零件表的** OnPostError **事件中处理的，如下所示：**

```
procedure TForm1.tblPartsPostError(DataSet: TDataSet;
        E: EDatabaseError; var Action: TDataAction);
begin
        if (pos('Cannot change the price of parts of this type',
                E.Message) <> 0) then
        begin
                Application.MessageBox('Cannot change the Retail Price
                  for COPPER parts', 'Update Error', MB_ICONERROR + MB_OK);
                  Action := daAbort;
        end;
end;
```

代码中，检查的消息来自 EDatabaseError，Delphi **将这些自定义异常错误提示为** "General SQL Errors"，**并返回异常在数据库的序号以及异常定义的消息。**

8.7 本 章 小 结

本章向读者介绍了异常。异常是所有优秀程序中的极其重要的部分。由于 Delphi **自身的优势，以及它为处理发生在程序中的错误而声明的大量的异常，无须用户的干涉，许多错误就可以被处理得很好。而且，一个好的建议就是不要干涉异常，让系统自己处理它们。**

在了解了异常基本原理的基础上，本章 8.4.2 **小节接着讲述了如何创建自己的自定义异常的方法和调用过程。**

异常是 Delphi **句法中最重要的部分之一，应该花些时间来学习它们是如何工作的。**

本章花了较大的篇幅讲解了高级的异常处理技术细节，包括事件驱动环境下的异常处理，以及数据库异常处理。在数据库应用程序中，出现各种异常的情况是很多的，所以，建议读者一定要对数据库异常有很好的了解。

第 9 章　动态链接库

本章将讨论 Win32 Dynamic Link Library（动态链接库），也称为 DLL。DLL 是编写任何 Windows 应用程序的关键组成部分。这里将介绍怎样创建和使用 DLL，读者将学到怎样调入 DLL 并链接 DLL 输出的例程，以及如何使用回调函数、怎样在不同的进程之间共享 DLL 的数据。

9.1　DLL 简介

动态链接库是一个程序模块，它包含代码、数据或者资源，可以被其他应用程序共享。DLL 的优势之一是，应用程序能够在运行期动态地调入代码，而不是在编译期静态地链接代码。这样，多个应用程序可以同时共享同一个 DLL 的代码。事实上，像 Kernel32.dll、User32.dll 和 GDI32.dll 就是 Win32 系统所共享的动态链接库。Kernel32.dll 负责管理内存、进程和线程，User32.dll 包含的例程用于实现用户界面、创建窗口和处理 Win32 消息，GDI32.dll 负责处理图形。读者可能还听说过其他 DLL，例如 AdvAPI32.dll 和 ComDlg32.dll，前者负责对象的安全性和注册，后者用于提供公共对话框。

使用动态链接库的另一个优势是有利于程序的模块化。当修改应用程序的时候，就只需要修改其中的一个模块，而不是整个应用程序。Windows 本身就是一个高度模块化的实例。当安装一个新的设备的时候，只需要安装一个设备驱动程序，它本身就是一种 DLL，实现了设备和 Windows 之间的相互通信。

从文件的角度看，一个动态链接库与一个 EXE 文件（可执行文件）非常类似。一个主要的区别在于，尽管 DLL 中包含了可执行的代码，但 DLL 不能单独执行。动态链接库文件的扩展名一般是 dll，也可能是 drv（设备驱动程序）、sys（系统文件）和 fon（字体文件）。

DLL 与其他应用程序通过 Dynamic Linking（动态链接技术）来共享代码，这将在后面进行介绍。一般情况下，如果一个应用程序使用了动态链接库，Win32 系统会保证内存中只有该动态库的一份复制品，这是通过内存映射文件实现的。DLL 首先被调入 Win32 系统的全局堆，然后映射到调用这个 DLL 的进程的地址空间。在 Win32 系统中，每个进程拥有自己的 32 位线形地址空间。如果一个 DLL 被多个进程调用，每个进程都会收到该 DLL 的一份映像。因此，与 16 位 Windows 不同的是，在 Win32 中，DLL 可以看作是每一个进程自己的代码。不过，这并不意味着物理内存中会分配 DLL 的多个实例。

下面是一些有关 DLL 的术语：

- 应用程序：一个可执行的 Windows 程序，扩展名为 exe。
- 可执行文件：一个包含可执行代码的文件，包括*.dll 和*.exe 文件。

- 实例：是指可执行文件在内存中的出现。每个实例可以由一个实例句柄来引用，而实例句柄是由 Win32 系统维护管理的。如果一个应用程序运行了两次，就会有该应用程序的两份实例句柄。当一个 DLL 被调入的时候，就会生成 DLL 的一份实例以及相应的实例句柄。注意，这里所说的实例，和类的实例不是同一个概念。
- 模块：在 Win32 系统中，模块和实例基本上是同义的，但在 16 位的 Windows 系统中，系统会自动建立和维护一个模块数据库，每个模块都有一个模块句柄。在 Win32 中，应用程序的每个实例都有自己的地址空间。因此，模块没有必要有自己单独的标识符。
- 任务：Windows 是一个多任务的环境，它必须能够为每一个在它下面运行的实例分配系统资源和时间。Windows 会建立一个任务数据库，其中包括每一个任务的实例句柄和其他必要的信息。

9.2 静态链接和动态链接

所谓静态链接，是指在编辑期把要调用的函数或过程链接到可执行文件中，成为可执行文件的一部分。换句话说，函数和过程的代码就在程序的.exe 文件中。

当程序调入内存中的时候，函数和过程也被调入内存中，它们在内存中的位置是与应用程序的位置相应的。当程序中需要调用函数或者过程的时候，执行流程就会跳转到函数或者过程所在的位置，然后再返回到调用的位置。至于这些函数或者过程的相对位置，在编译期就确定了。

在静态链接中，假设有两个应用程序，它们都要调用一个动态库中的一个函数，当这两个应用程序同时运行的时候，这个函数就会在内存中存在两个实例，如果还有第 3 个应用程序也调用这个函数，那么内存中就有这个函数的 3 个实例。

而动态链接就可以解决这个问题。此时，当一个应用程序把这个函数调入内存中后，如果还有其他的应用程序也需要调用这个函数，Win32 就会把 DLL 映射到每个进程的地址空间，而物理内存中只有这个 DLL 的一个实例。

如果采用动态链接的方式，被调用的函数是在运行期才链接到可执行文件中的。要引入一个外部的函数，必须在应用程序中或者在应用程序的引用单元中声明这个函数：

```
function DllFunction(Para: Integer): Boolean; external 'MyLib.dll';
```

在这里，您不需要也没有必要定义 DllFunction 这个函数，关键字 external 将向程序解释函数 DllFunction 是在一个外部的动态库 MyLib.dll 中。当这个应用程序运行时，Win32 系统会自动把动态链接库 MyLib.dll 调入内存，并且链接其中的 DllFunction 函数。

9.3 使用 DLL 的必要性

使用 DLL 有若干个理由，其中有的理由在前面已经提到了。使用 DLL 能够共享代码或资源数据、隐藏实现的细节或低级系统例程、设计自定义控件。

9.3.1　共享代码、资源和数据

使用 DLL 的主要目的是为了共享代码。DLL 的代码可以被所有的 Windows 应用程序共享。

另外，通过 DLL 可以共享资源，例如位图、字体或图标等。这些资源本来可以放在资源文件中，并直接链接到应用程序中。但是如果把资源放在 DLL 中，许多应用程序就可以使用 DLL 中的资源，但内存中不会重复分配这些资源。

在 16 位 Windows 中，DLL 有自己的数据段，因此，所有调用同一个 DLL 的应用程序可以访问同样的全局变量和静态变量。而 Win32 系统就不同，由于 DLL 被映射到每一个进程的地址空间，DLL 的所有数据属于它映射的进程。由于每个线程的执行是相互独立的，当线程访问某个全局变量时要特别小心，避免引起冲突。

提示：尽管 DLL 的数据不能被几个进程共享，但它可以被一个进程内的所有线程共享。

这并不意味着没有办法使几个进程共享同一个 DLL 的数据。一个办法是使用内存映射文件在内存中创建一个共享的区域。凡是调用 DLL 的应用程序都可以读这个共享区域的数据。

9.3.2　隐藏实现的细节

某些情况下，用户可能不希望别人知道例程的实现细节，DLL 就能够达到这个目的。DLL 中的例程可以被应用程序调用访问，但应用程序并不知道这些例程是如何实现的。从这个角度看，DLL 和 Delphi 的已编译单元（即*.dcu 文件）很相似，*.dcu 文件也可以隐藏代码细节，但是，它只适用于 Delphi 程序，而且往往还限于同一个版本。而 DLL 是语言无关的，用户可以创建一个 DLL，被 VC、VB 或任何支持动态链接库的语言调用。

9.3.3　自定义控件

自定义控件通常放在 DLL 中，这些控件不同于 Delphi 的自定义组件，自定义控件是在 Windows 下注册的，可以用在任何 Windows 开发环境。之所以要把自定义控件放在 DLL 中，是因为即使有多个应用程序使用这些自定义的控件，内存中也只有一份控件的实例。

提示：现在，Microsoft 已经用 OLE 和 ActiveX 控件的方式替代了把自定义控件放在 DLL 中的方式。

9.4　创建和使用 DLL

下面介绍创建一个 DLL 的全部步骤，读者还将学到怎样把 Delphi 的 Form 加到 DLL 中去。

9.4.1　创建 DLL

首先通过 File | New 命令打开 New Items 对话框，然后双击 DLL Wizard 图标，系统新

开始了一个 DLL 程序，如图 9.1 所示。

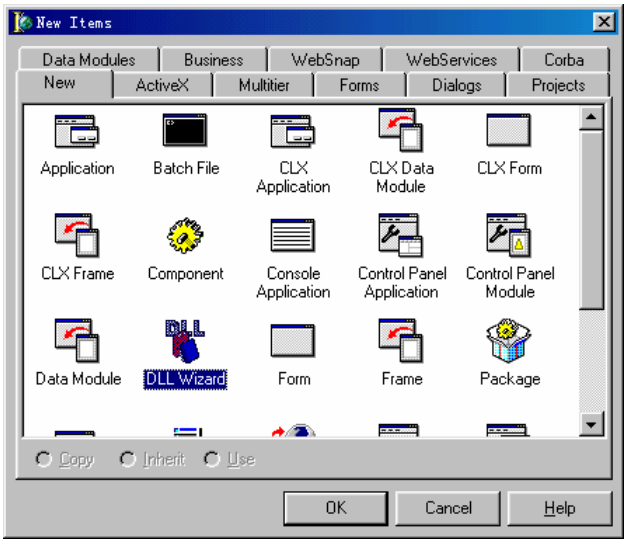

图 9.1 创建 DLL

下面是一个简单的 DLL 程序，其中有一个函数 GetMax()，它的功能是输出两个输入参数中的最大值。

```
library ExampleDLL;

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

uses
    SysUtils,
    Classes;

{$R *.RES}

function GetMax(Para1: Integer; Para2: Integer): Integer;
begin
    if Para1 > Para2 then Result := Para1
    else Result := Para2;
```

```
end;

exports
        GetMax;
end.
```

其中，exports **子句用于引出** DLL **中要被其他应用程序调用的函数或者过程。**

9.4.2　定义接口单元

为了便于其他应用程序调用 DLL **中的输出函数，可以定义一个接口单元。只要把该接口单元加到其他应用程序中的** Uses **子句中，应用程序就可以方便地调用** DLL **中的函数了。在接口单元中还可以定义一些** DLL **和调用** DLL **的应用程序都能使用的公共结构。下面就是一个非常简单的接口单元，其中只定义了输出函数** GetMax**。**

```
unit DLLInterface;

interface
function GetMax(Para1: Integer; Para2: Integer): Integer; stdcall;

implementation

function GetMax: Integer; external 'ExampleDLL.dll' Name 'GetMax';

end.
```

有两种引入 DLL **中例程的方法。第一种就是上面示例中的方法：**

```
function GetMax: Integer; external 'ExampleDLL.dll' Name 'GetMax';
```

这种方法按名称引入例程。另一种方法是按序号引入，如下所示：

```
function GetMax: Integer; external 'ExampleDLL.dll' Index 1;
```

按序号引入例程能够减少 DLL **的调入时间，因为不需要在** DLL **的名称表中查找例程的名称，不过，在** Win32 **中这并不是最好的方法，最好的方法还是按照名称引入例程，当** DLL **有什么修改之后，应用程序不必担心例程的序号发生变化。**

在发布上面的这个 DLL **的时候，必须要同时发布** ExampleDLL.dll **和** DLLInterface.pas **这两个文件。这样，如果程序员需要使用其他语言，例如** C++**，他就可以把这个** PAS **文件转为其他语言，从而使这个** DLL **能够在其他的开发环境中使用。**

9.5　在动态库中显示 Form

这一节将讨论怎样显示 DLL **中的** Form**。把** Delphi **的** Form **加到** DLL **中后，这个** Form **可以被任何** Windows **应用程序或者其他开发环境，例如使用** C++**、**VB **等。**

9.5.1 显示模式 Form

要把一个 Form 编译进 DLL，这个 Form 必须自己创建和释放。下面是一个包含模式 Form 的 DLL 例程：

```
library ExampleDLL;

uses
    SysUtils,
    Classes,
    Windows,
    Forms,
    Un_Main in 'Un_Main.pas' {fm_Main};

{$R *.RES}

function ShowMainForm(AHandle: THandle): Boolean; stdcall;
begin
    Result := True;
    Application.Handle := AHandle;
    with Tfm_Main.Create(nil) do try
        ShowModal;
    finally
        Free;
    end;
end;

exports
    ShowMainForm;
end.
```

这个 DLL 中的主 Form 被包含进了一个输出函数 ShowMainForm()中，该函数包含一个 THandle 参数，该参数用于将包含模式 Form 的 DLL 的 Handle 与主调程序的句柄合并到一起。在第 3 章中介绍 Delphi 项目的时候，曾经提到过应用程序的一个全局变量 Application 和它的一个特性 Handle，DLL 也有 Application 变量和 Handle 特性，而且这个 Application 对象与调用这个 DLL 的应用程序的 Application 对象是不同的。要使 DLL 中的 Form 作为一个模式 Form 打开，必须把调用这个 DLL 的应用程序的 Application.Handle 句柄赋给 DLL 的 Application.Handle 特性。这样，DLL 中的 Form 将会与应用程序合并到一起。而如果不做这一步操作的话，会看到调用 DLL 的应用程序和 DLL 中的模式 Form 将分开地显示在 Windows 中，而且在把 Form 最小化的时候，将会出现不可预测的结果。

因为 DLL 中的 Form 都是用户自己创建的，所以 DLL 本身不会在程序结束的时候去释放它们，用户必须自己在适当的时候释放创建的 Form 实例。上面的 DLL 中，使用了 Try…Finally 结构来保证在 Form 关闭的时候释放它。

下面的程序将调用该 DLL 中的 ShowMainForm()函数：

```
function ShowMainForm(AHandle: THandle): Boolean; stdcall; external 'ExampleDLL.dll' Name 'ShowMainForm';
. . .
procedure TForm1.Button1Click(Sender: TObject);
begin
    ShowMainForm(Application.Handle);
end;
```

注意：如果 DLL 中的例程需要传递字符串或者动态数组作为参数或者返回值，则 DLL 必须要通过 Uses 子句来引用 ShareMem 单元，而且，这个单元必须是在 Uses 子句中的第一个单元。ShareMem **单元实际上是内存管理器 Borlndmm.dll 的接口单元，**Borlndmm.dll **必须与 DLL 一起分发。如果不想使用 Borlndmm.dll，那应当把字符串改用 PChar 或者 ShortString 来代替。**

9.5.2　显示无模式 Form

9.5.1 **小节介绍了怎样打开 DLL 中的模式 Form，现在来介绍如何打开无模式 Form。要打开 DLL 中的无模式 Form，DLL 必须提供两个输出例程，一个用于创建和显示 Form，另外一个用于释放 Form。仍然以** 9.5.1 **小节中的程序为例来说明，如下所示：**

```
function ShowMainForm(AHandle: THandle): Longint; stdcall;
begin
    Application.Handle := AHandle;
    fm_Main := Tfm_Main.Create(nil);
    Result := Longint(fm_Main);
    fm_Main.Show;
end;

function CloseMainForm(AFormIns: Longint): Boolean; stdcall;
begin
    if AFormIns > 0 then
        Tfm_Main(AFormIns).Free;
end;

exports
    ShowMainForm, CloseMainForm;
end.
```

这个 DLL 中包含了两个例程，ShowMainForm()**和 CloseMainForm()例程。前者类似与** 9.5.1 **小节中介绍的** ShowMainForm()**函数，它的作用是把应用程序的句柄赋值给 DLL 中的** Application.Handle **特性，然后创建并显示** Form **实例，这里是调用** Show()**方法而不是**

ShowModal()方法。另外，该函数将返回一个 Longint 值即 Form 的实例，因为应用程序需要知道这个 Form 的实例是否存在，以保证在释放 Form 时不会对一个空的 Form 实例进行操作。

CloseMainForm()函数首先检查 Form 的实例释放存在，如果存在的话，将调用 Free()函数来释放它。

相应的调用该 DLL 的程序如下所示：

```
var
    Form1: TForm1;
    FormIns: Longint;

implementation

function ShowMainForm(AHandle: THandle): Longint; stdcall; external 'ExampleDLL.dll' Name 'ShowMainForm';
function CloseMainForm(AFormIns: Longint): Boolean; stdcall; external 'ExampleDLL.dll' Name 'CloseMainForm';

{$R *.DFM}

procedure TForm1.Button1Click(Sender: TObject);
begin
    FormIns := ShowMainForm(Application.Handle);
end;

procedure TForm1.Button2Click(Sender: TObject);
begin
    CloseMainForm(FormIns);
end;

end.
```

注意：只有调用 DLL 的应用程序才能关闭 Form，而 DLL 本身不能擅自关闭 Form，否则，调用 CloseMainForm()函数将失败。

9.6 DLL 的入口和出口函数

如果 DLL 需要在调入时进行初始化，或者在释放时进行一些清理工作，那么可以给 DLL 加上入口和出口函数。

典型的初始化操作是注册窗口类和初始化全局变量，或者初始化入口和出口函数。初始化的工作是在 DllEntryPoint 过程中进行的，而 DllEntryPoint 过程可以在 DLL 的项目文

件的 begin…end 之间调用，它需要传递一个 DWord 类型的参数。

全局变量 DLLProc 是一个过程的指针，通过这个变量可以建立 DLL 的入口和出口函数。这个变量默认为 nil，除非建立了入口和出口函数。入口和出口函数可以对应表 9.1 中的事件。

表 9.1　DLL 的入口和出口事件

事件	用途
DLL_PROCESS_ATTACH	一个进程已经把 DLL 映射到它的地址空间，此时可以对 DLL 进行初始化
DLL_PROCESS_DETACH	DLL 正在从进程的地址空间分离出来，这可能是因为进程本身退出或者调用了 FreeLibrary()方法，此时应当释放先前分配的资源
DLL_THREAD_ATTACH	进程创建了一个新的线程，此时，系统就会调用 DLL 的入口函数
DLL_THREAD_DETACH	一个线程正在退出，此时，系统将调用 DLL 的出口函数

下面的代码将演示如何通过 DLLProc 这个全局变量来建立 DLL 的入口和出口函数。在这个示例中，首先对变量 DLLProc 进行赋值。然后在 DLLEntryPoint()过程中，先检查 dwReason 参数的值，来判断当前是什么事件。这里对每个事件的处理比较简单，只是演示一下事件如何触发。当 DLL 被调入和删除时，分别显示一个信息框，当线程被创建和退出时，分别发出“嘀”的一声，代码如下所示：

```
library DllEntry;

uses
    SysUtils,
    Windows,
    Dialogs,
    Classes;

{$R *.RES}

procedure DLLEntryPoint(dwReason: DWORD);
begin
    case dwReason of
        DLL_PROCESS_ATTACH: ShowMessage('Attaching to process');
        DLL_PROCESS_DETACH: ShowMessage('Detaching from process');
        DLL_THREAD_ATTACH: MessageBeep(0);
        DLL_THREAD_DETACH: MessageBeep(0);
    end;
end;

begin
    DllProc := @DLLEntryPoint;    //把一个过程的指针赋给 DLLProc 变量
    DLLEntryPoint(DLL_PROCESS_ATTACH);
```

```
end.
```

下面的示例将说明怎样使用上面的这个 DLL：

```
unit Un_Client;

interface

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

type
    TMyThread = class(TThread)
    end;

    TForm1 = class(TForm)
        btnLoadDLL: TButton;
        btnFreeDLL: TButton;
        btnCreateThread: TButton;
        btnFreeThread: TButton;
        procedure btnLoadDLLClick(Sender: TObject);
        procedure btnFreeDLLClick(Sender: TObject);
        procedure btnCreateThreadClick(Sender: TObject);
        procedure btnFreeThreadClick(Sender: TObject);
    private
        { Private declarations }
    public
        { Public declarations }
    end;

var
    Form1: TForm1;
    DllHandle: THandle;
    MyThread: TMyThread;

implementation

{$R *.DFM}

procedure TForm1.btnLoadDLLClick(Sender: TObject);
begin
    if DllHandle = 0 then
        DLLHandle := LoadLibrary('DllEntry.dll');    //调入 DLL
end;

procedure TForm1.btnFreeDLLClick(Sender: TObject);
```

```
begin
    if DllHandle <> 0 then begin
        FreeLibrary(DllHandle);    //释放 DLL
        DllHandle := 0;
    end;
end;

procedure TForm1.btnCreateThreadClick(Sender: TObject);
begin
    MyThread := TMyThread.Create(False);    //创建一个线程的实例
end;

procedure TForm1.btnFreeThreadClick(Sender: TObject);
begin
    DllHandle := 0;
    MyThread := nil;      //删除一个线程的实例
end;

end.
```

在这个示例中包含 4 个按钮。其中，btnLoadDLL 按钮用来调入动态链接库 DLLEntry.dll，btnFreeDLL 按钮用来释放动态库，btnCreateThread 按钮用来创建一个线程的实例，btnFreeThread 用来删除一个线程的实例。

btnLoadDLL 按钮调用了 LoadLibrary()函数来调入动态库 DllEntry.dll，并把它映射到进程的地址空间，此时，将会执行 DLL 的初始化代码。对于一个进程来说，初始化的代码只会执行一次，如果另一个进程也调用了这个 DLL，初始化代码又会执行一次。

btnFreeDLL 按钮调用了 FreeLibrary()函数，用来从内存中卸载动态链接库，此时将调用 DLLProc 变量所指的 DLLEntryProc()过程，传递的参数是 DLL_PROCESS_DETACH。

btnCreateThread 按钮用于创建一个线程的实例，此时将调用 DLLEntryProc()过程，传递的参数是 DLL_THREAD_ATTACH。btnFreeThread 按钮调用 DLLEntryProc()，传递的参数是 DLL_THREAD_DETACH。

上面示例中的入口和出口函数只是简单地打开一个信息框或者使系统发出声音，但用户可以根据需要对线程或者进程进行任何初始化或者清理的操作。

9.7　本 章 小 结

DLL 是 Windows 应用程序实现代码重用的重要手段，本章讨论了如何创建 DLL、如何使用 DLL 以及如何调用 DLL 等方面的内容。现在，读者应当能够创建动态链接库，并且在应用程序中调用动态链接库了。

第 10 章　数据库开发

与其他开发工具相比，Delphi 6.0 在数据库开发方面具有无与伦比的优势，这很大程度上归功于 VCL 提供的功能强大的数据库组件，它们将大部分数据库的底层设置封装了起来，使开发者只要进行简单地设置数据源就可以方便快速地开发出数据库应用程序。它们包含在几个组件组中：DataAccess、DataControls、DBExpress、DataSnap、BDE、ADO 以及 InterBase 等。

本章将向读者介绍数据集以及操纵数据集的基础知识和技术，例如 BDE、ODBC、ADO 等，然后介绍如何运用数据库表和查询。最后，读者将通过本章的学习了解到作为一个专业的 Delphi 数据库开发人员所需了解的重点。

10.1　配置 ODBC 数据源

ODBC（Open Database Connectivity）是由 Microsoft 公司开发的一套独立于数据库的驱动支持。在 Windows 平台上，Microsoft 公司已经预装了 ODBC 对很多类型数据库的驱动，例如 Access、FoxPro、Excel File、Paradox、DBase 等。用户可以进入系统“控制面板”，然后双击“ODBC 数据源(32 位)”图标，将出现一个“ODBC 数据源管理器”对话框，如图 10.1 所示。

图 10.1　“ODBC 数据源管理器”对话框

下面以创建一个 Access 数据源为例来说明如何配置 ODBC。

在 ODBC 数据源管理器对话框中单击“添加”按钮，将打开“创建新数据源”对话框，如图 10.2 所示。从 Windows 默认提供的驱动程序列表中选择 Microsoft Access Driver（*.mdb），并单击“完成”按钮，这时将出现如图 10.3 所示的对话框，在这里对新加的数据源进行命名和描述，以及制定创建或连接 Access 数据库的位置。示例中，命名新的数据源为 MyAccess，描述为 Test for ODBC。

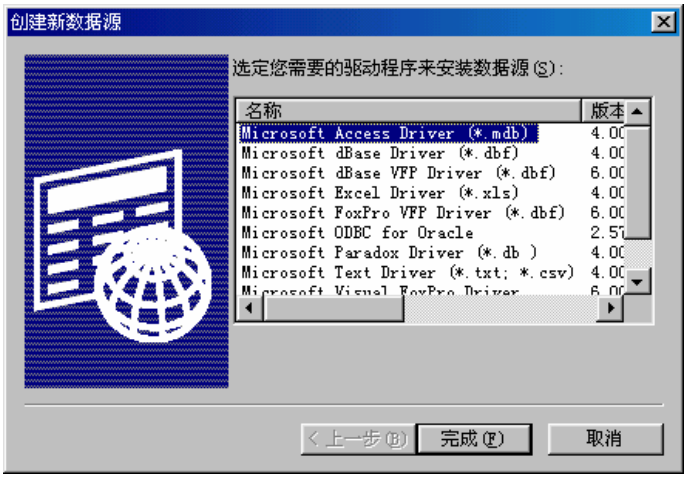

图 10.2　“创建新数据源”对话框

图 10.3　配置 Access 数据源

如果已经有一个 Access 数据库存在，单击“选取”按钮，然后制定已有的数据库文件所在的位置，确认之后，在 ODBC 中的新数据源 MyAccess 将连向用户所指定的 Access 文件；如果想创建一个新的 Access 数据库，只要单击“创建”按钮，然后指定一个 Access 文件名，另外还需要选择一种排序语言，这里选择“汉语拼音”。确认之后，系统提示用户数据库创建成功，这时候，一个新的 Access 数据库就已经创建完成了，在本示例中，用户会在 C 盘根目录下找到一个新创建的 MyAccess.mdb 文件，如图 10.4 所示。

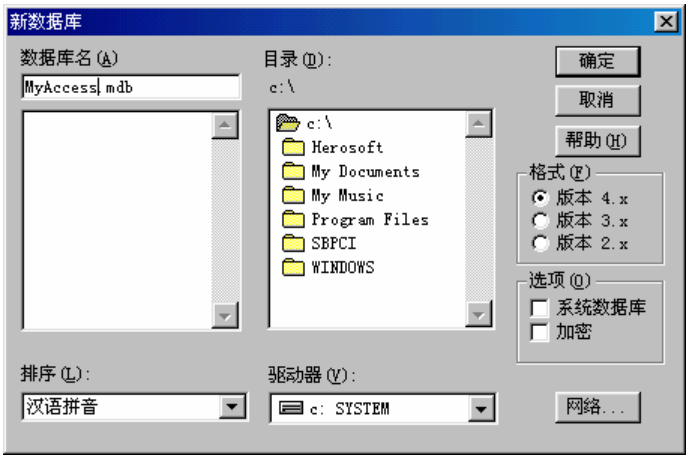

图 10.4 创建一个新的数据库

这里，笔者想再更深入地讨论 ODBC 是如何对数据源进行配置的。其实很简单，首先，驱动数据库的关键是数据库驱动程序，而 ODBC 对数据源的配置只是使用这些驱动程序而已，例如 Access 等数据库。无论是 Windows98 还是 Windows NT（2000），在系统中都已经安装注册了一个 odbcjt32.dll 文件，它可以驱动很多数据库类型，例如 Access、Paradox、Excel 文件、Dbase 等，但是其他一些专业性更强的数据库诸如 SQL Server、Sybase 等则需要专门配置的驱动程序。这时候 ODBC 只是起到一个中间接口的作用，它的奥妙就在系统的注册表中。运行 regedit 命令打开系统的注册表，展开 HKEY_CURRENT_USER 根键，在 Software\ODBC\ODBC.INI 下，就列出了所有已经配置好了的 ODBC 数据源，每个数据源有一个主键与之相对应，主键中的数据就是数据源的配置细节，例如：驱动程序位置（Driver）以及 DriverID 等。另外，在 ODBC.INI 下还有一个主键 ODBC Data Sources，在这里罗列了所有的数据源名称，以及所对应的数据驱动名称。对照 ODBC 数据源管理器中“用户 DSN”选项卡，就会发现其中的规律了。因此，在 Delphi 中，完全可以通过写注册表的方法来配置已知的数据源，下面的示例就是配置一个名为 MyAccess 的 Access 数据源：

```
procedure TFrom1.InitialODBC;
var
    SystemPath: PChar;
Begin
    GetMem(SystemPath, 255);
    GetSystemDirectory(SystemPath, 255);
    with TRegistry.Create do try
        RootKey := HKEY_CURRENT_USER;
        if OpenKey('\Software\ODBC\ODBC.INI\ODBC Data Sources', True) then if not
  ValueExists('MyAccess') then
          WriteString('MyAccess','Microsoft Access Driver(*.mdb)');
          if OpenKey('\Software\ODBC\ODBC.INI\MyAccess', True) then begin
        if not ValueExists('Driver') then WriteString('Driver', SystemPath + '\odbcjt32.dll');
        if not ValueExists('UID') then WriteString('UID', '');
```

```
                    if not ValueExists('DriverID')
then WriteInteger('DriverID', 25);
                    if not ValueExists('SafeTransactions')
then WriteInteger('SafeTransactions', 0);
                end;
        finally
                Free;
        end;
end;
```

或者通过 ODBC **数据源管理器、或者通过注册表配置完** ODBC **数据源以后，在** Delphi **中如何使用这个数据源呢？事实上，**Delphi **提供了很多方法让用户通过** ODBC **来使用数据源。最常用的一种方法是使用** BDE**，当然使用** BDE **可以不需要** ODBC **的配置而直接连接** Access **数据库，但借助于** ODBC **的配置无疑是比较便捷的方法。**

10.2　Borland **数据库引擎**

Delphi **是通过** BDE（Borland **数据库引擎）来完成对数据库的访问的。无论对于存储格式诸如** DB **和** DBF **文件的本地数据，还是更为复杂的客户机/服务器体系的异地数据，**BDE **都能提供性能优异的访问服务。**

几年前，Borland **公司发现他们的软件在处理数据库访问时存在一个问题。虽然他们开发出的一些产品都能实现访问数据库的功能，但是，每一种产品在链接、使用数据的方法上都各有不同。**Borland **公司认为如果能够将这些方法统一起来，将会使产品更具有竞争力。于是，他们决定推出一个新的软件，将所有的数据库功能都集成到一个引擎中，这就是后来的** BDE**。**

下面就来看看如何配置 BDE**，并通过它来访问数据库。**

10.2.1　BDE **管理器**

打开“控制面板”，双击 BDE　Adminstrator **图标进入** BDE **管理员系统，如图** 10.5 **所示；也可以通过** Delphi **中的** Database | Explore **命令进入** BDE **的数据浏览系统，如图** 10.6 **所示。在** BDE **管理系统中可以配置新的数据源驱动，在** BDE **数据浏览系统中可以浏览数据源中的数据。**

打开 BDE　SQL　Explorer **以后，会发现一个** MyAccess **别名已经存在于** Databases **选项卡中了，它是** BDE **根据** ODBC **的配置自动创建的，它所连的数据源就是刚刚创建的** MyAccess **数据源。当然也可以自己动手创建和它相同的数据库别名，步骤如下：执行** Object | New **命令，然后在弹出的数据库驱动对话框中选择** Mircosoft Access Driver (*.mdb)**，如图** 10.7 **所示。**

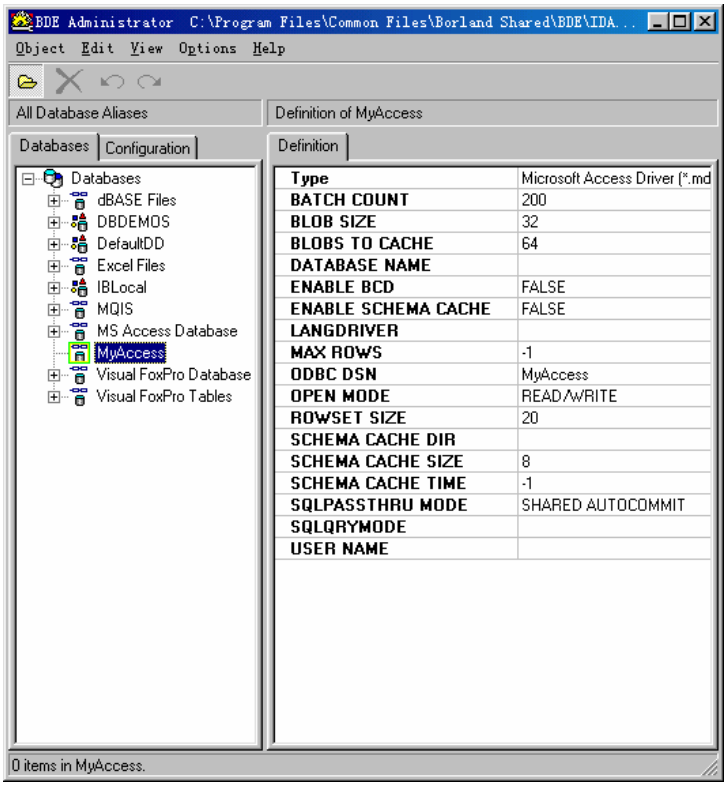

图 10.5　BDE **管理员系统**

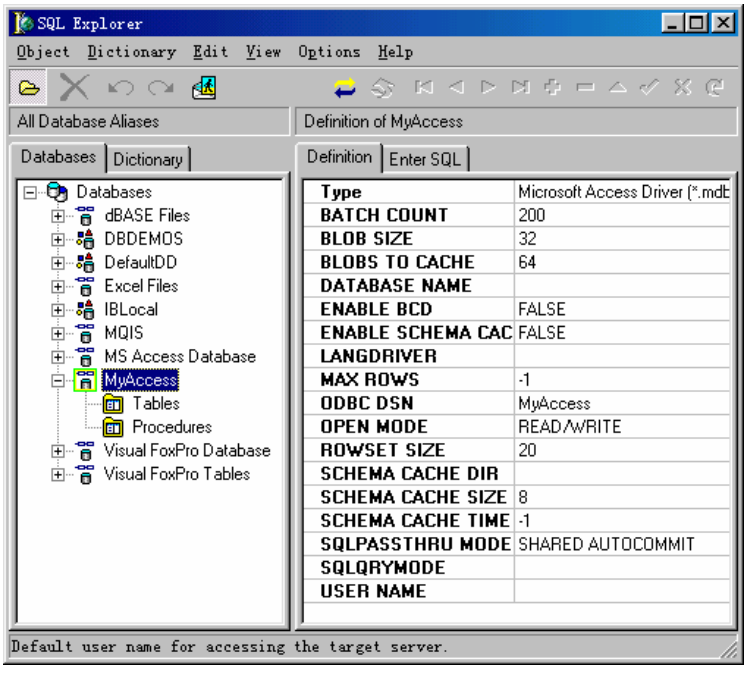

图 10.6　BDE **的数据浏览系统**

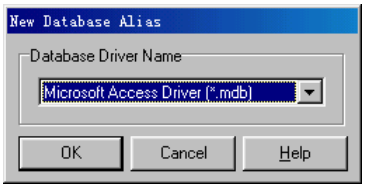

图 10.7　数据库驱动对话框

单击 OK 按钮后，系统将创建一个名为 ODBC1 的数据库别名，将 ODBC1 改名为 MyAccess1，然后执行 Object|Apply 命令保存该别名。保存以后，在右面的 ODBC DSN 中通过下拉列表框可以列出当前 ODBC 中所有的 Access 数据库的数据源名称。选择 MyAccess 后，再次保存别名，这时候，用户所创建的连接 MyAccess 数据源的别名 MyAccess1 就已经完成了，它与 BDE 自动创建的 MyAccess 别名完全相同。

注意：只有在 MyAccess1 保存（Apply）之后，ODBC DSN 中才会有数据源名，否则，它总是为空。

除此以外，BDE 还提供了一种方法让用户创建自己的数据库驱动，就像系统提供的 Microsoft Access Driver(*.mdb)一样。步骤如下：从控制面板中双击 BDE Adminstrator，在左边的面板中翻到 Configuration 选项卡，可以看到一个树状的视图，展开 Drivers 节点并选中其中的 ODBC，执行 Object|New 命令，这将打开一个 New ODBC Driver 对话框，如图 10.8 所示。Driver Name 文本框中的名称就是新的数据库驱动的名称，可以任意指定，这里把它命名为 MyAccessDriver，ODBC Driver Name 设置为 Microsoft Access Driver(*.mdb)，这时候，在最下面的列表框中列出了所有以 Access Driver 为驱动的数据源，这里选择 MyAccess，它就是在上一步创建的。单击 OK 按钮，回到 BDE 配置程序的主窗口。将左边的面板翻到 Databases 选项卡，这时候，BDE 已经自动创建了一个数据源别名 MyAccess2，它的数据库驱动程序就是刚刚创建的 MyAccessDriver。当然，同样可以自己动手在 MyAccessDriver 的基础上创建一个新的数据源别名，具体方法在前面已经描述过了。

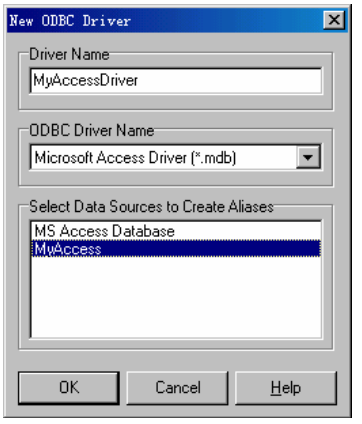

图 10.8　新建 ODBC 驱动

通过 BDE 创建好数据源以后，现在要做的工作是编辑它，向数据库中加入数据记录。对应这种 Access 数据库，可以直接通过 Office Access 97（2000）打开数据库，然后进行编辑，可以创建新的数据表、视图、向表中添加记录等。

Delphi 提供了一套桌面数据库工具 Database Desktop 来创建 Access 数据库表。这个工具可以在 Borland Delphi 的菜单中找到。它可以直接根据 BDE 中的数据库别名打开数据库进行操作。

设置完数据源以后，在 Delphi 中，放置一个 TTable 到窗体上，从 Database 属性下拉列表框中就可以选择到上面创建的 MyAccess 数据源，然后设置 TTable 的 TableName 属性，它是数据库中的数据表。最后，通过一个 TDataSource 组件将这个 TTable 连接到一个 DBGrid 上，设置 TTable 的 Active 属性为 True 以后，就可以在 DBGrid 中浏览查看数据表的数据了。

10.2.2 Databases 选项卡

在 BDE 管理员系统中，Database 选项卡列出了所有可使用的数据库别名，在图中（见图 10.5），它们呈分层树状显示，类似 Windows 资源管理器。查看数据库别名，只需单击 Databases 左边的“＋”号，别名显示后，可以通过选择某一别名，在 BDE 管理器的右侧面板方便地查看和修改它的定义和配置。

上面不止一次提到了数据库别名，数据库别名赋予强大的处理数据库的能力。在商业应用中，随着商业机构不断配备新的文件服务器，将各种数据库操作转移到特定的数据库服务器上，数据库的重新定位变得极其频繁。一些开发环境要求在每次改变数据库位置时都必须重新编译应用程序，或者在代码中集成一个附加的中间层。而 BDE 可以通过使用别名自动地赋予这样的能力。在探讨 BDE 别名的工作机制时，有必要对数据库和数据库表的本质稍加研究。

Wayne Radcliff 在兼容 PC 时代的早期设计出了 dBASE 文件结构，并用扩展名 DBF（Database File）来描述它。现在看来，这个名称不是很合适，因为现在的数据库概念并不等同于 DBF 文件。

每个 DBF 文件都包括一张表，它是数据的有序集合。表的列称作字段，行称作记录。在某些方面，数据库这一术语常被描述为多个相互关联的表的集合。除了最简单的应用，数据库通常是由多个相互关联并包含多条记录和字段的表组成。数据库与表的区别请务必理解。

数据库是相关表的集合，那么这种集合又是如何实现的呢？像 Oracle、Interbase 这样的数据库服务器都拥有一种由多表构成的数据库数据结构类型。数据库开发人员常常需要用到它们。但是，脱离了服务器环境，连接多表的机制通常并不存在。这时，用户可能会想到创建自己的数据库格式来管理所有的表格。但是，如果这样的话，将无法兼容旧版本的数据库程序。这些由其他开发人员编写的旧版本程序在提取或更新用户的数据库时会带来很多麻烦，因为，它们并不是为用户的文件格式设计的。

BDE 对这类问题提供了一个简洁明了的解决方案。将一组相关表放进一个 DOS 目录里，就可以用 BDE 将这些表连接起来。BDE 通过给某一特定目录下的一组文件赋别名，或数据库名创建了一个逻辑表集合。BDE 将这样的集合称为虚拟数据库。虚拟数据库并不具有一个完整数据库服务器的高级特征，例如参照完整性保存，但它确实能够让用户使用数据库名作为指针来定位数据位置。

BDE 别名的目标是创建一个指针的指针。任何 Delphi 数据库访问事务都将引用那些以表为引用的数据库别名。即使当数据从一个地方移到另一个地方时，也可以通过修改别名，迅速找到用户的数据。

在创建新数据库表或访问旧表时，都应该用别名而不是路径名来引用它。例如，在 Database Desktop 的 Open Table 对话框中，既可以选别名，也可以选驱动器名，如图 10.9 所示。这样，当用户准备打开一张数据库表时，可以选驱动器名或者只使用别名强行转到特定的目录。

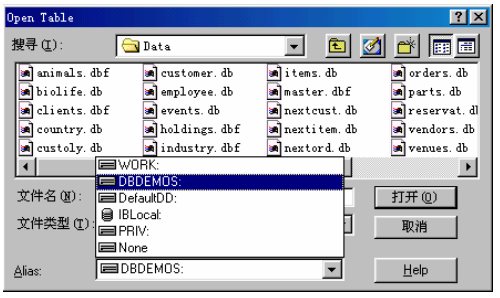

图 10.9　Database Desktop 的 Open Table 对话框

当在下拉列表框中改变别名时，可以看到目录也随着改变。在 Delphi 需要引用表的任何地方，都可以定义别名以取代目录名。

10.2.3　Configuration 选项卡

BDE 管理器上的 Configuration 选项卡可以显示和配置 BDE 用来管理表格的驱动。这些驱动可分为两类：Native 驱动和 ODBC 驱动。选择驱动后，其设置将显示在右边面板的 Definition 选项卡中，如图 10.10 所示。

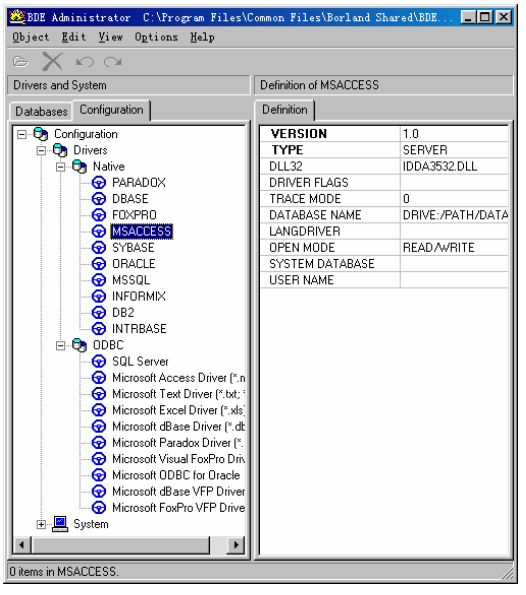

图 10.10　Configuration 选项卡

10.2.4 分发 BDE

Delphi 在编制程序时，会生成一个单机运行的可执行文件。由于 BDE 的特殊性，要求它独立地工作于其他软件包之外。BDE 的核心由一整套 DLL 组成，必须将它们分发给任一需要访问 BDE 函数的程序。

Borland 公司在 Delphi 6.0 Professional 版和 Client/Server 版中提供了一个 InstallShield Express Custom Edition for Delphi 工具，这使得 BDE 与应用程序的分发极其简单。在使用 InstallShield Express Custom Edition for Delphi 来安装应用程序时，用户可以很容易地将 BDE 放入安装程序包中。如果对 InstallShield Express Custom Edition for Delphi 不是很熟悉的话，建议找相关的书籍或者资料了解一下。

10.3 数据库应用程序体系结构

上面了解了 ODBC 和 BDE 数据驱动的知识，下面将从开发 Delphi 数据库应用程序的角度进一步介绍开发数据库应用程序所必须要了解的体系结构。

10.3.1 数据集

简单的说，数据集就是是数据行和列的集合。数据集里的每一列具有相同的数据类型，称之为字段；每一行由所有的列的类型的数据组成，称为一条记录。VCL 把数据集封装在一个叫做 TDataSet 的抽象组件中，TDataSet 引入了很多用于操作和浏览数据集的特性和方法。

为了保证术语统一、明确、清晰，这里先介绍说明一些基本概念，以及常用的数据库术语：

- 数据集：是一组离散的数据记录的集合，每个记录由多个字段构成，每个字段可以是各种数据类型（整型、字符、字符串、图像等）的数据。数据集在 Delphi 中由 VCL 中的抽象类 TDataSet 来表达。
- 表：是一种特殊的数据集类型。一个表可以是一个单独的文件，例如 Paradox，FoxPro 等数据库，另外有些数据库，例如 Access、Interbase 等，一个数据库是一个文件，其中可以包含有众多的数据库表。VCL 中通过 TTable 封装了表。
- 查询：与表相似，也是一种特殊的数据集，它可以看作是执行特定命令后产生的数据库表，这些命令通常对一个和一组数据库表进行操作。VCL 用 TQuery 封装处理查询。
- 索引：定义了数据库表排序的规则。通常基于特定字段建立索引就是指用该字段的值作为排序的依据。

10.3.2 BDE 数据访问组件

Delphi 6.0 的组件选项板中的 Data Access、Data Control 和 BDE 组件组包含了用来访问和管理 BDE 数据集的 VCL 组件。

下图为其中的 BDE 组件组，这里有 3 种组件用来表达数据集：TTable、TQuery 和 TStoredProc，如图 10.11 所示。这些组件都是从 TDBDataset 继承下来的，而 TDBDataSet 是从 TBDEDataSet 继承下来的，TBDEDataSet 是从 TDataSet 继承下来的。正如前面所说，TDataSet 是一个封装了数据集管理、导航和操作的抽象类，TBDEDataSet 则是一个表达特定的 BDE 数据集的抽象组件。

图 10.11　组件选项板上的 BDE 组件组

1．TTable

正如名字所暗示的，TTable 是一个表达数据库表中的数据和结构的组件。它有两个比较重要的特性为：DatabaseName 和 TableName。其中 DatabaseName 特性用来指定 TTable 链接的数据源，可选的数据源可以是在 BDE 管理器中设置的数据库别名，也可以是基于文件目录存储的数据库所在的路径，例如 Paradox、Foxbase 等。TableName 特性用来选择 TTable 连接的是数据库中的哪个表，在下拉列表框中会列出当前数据库中所有的数据表，只要从中选择一个即可。图 10.12 中显示出了两个特性下拉列表框的样子。

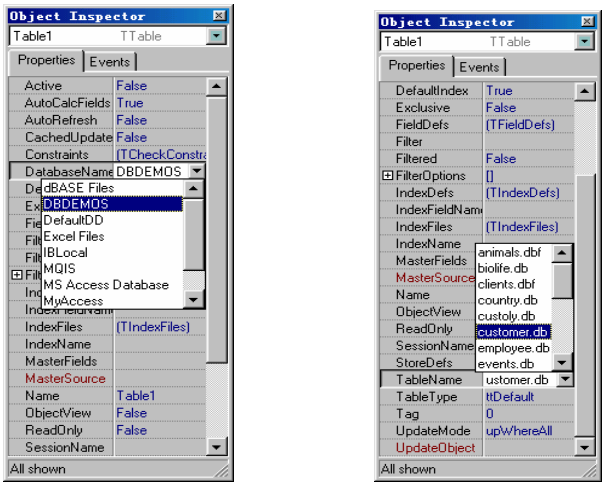

图 10.12　TTable 的 Database Name 和 Table Name 特性

2．TQuery

TQuery 是一个用来表达 SQL 查询操作的组件，它可以有返回的数据集，也可以没有。与 TTable 一样，TQuery 也有 DatabaseName 特性，同样是用来指定数据库。但 TQuery 没有 TableName 特性，也就是说，TQuery 并不局限在某个数据表上，而是在整个数据库中进行查询或者操作。TQuery 有个 SQL 特性，它是 TStrings 类型，单击 SQL 特性框边上的小按钮，可以打开一个文本编辑框，在这里可以输入所要操作的 SQL 语句，如图 10.13 所示。

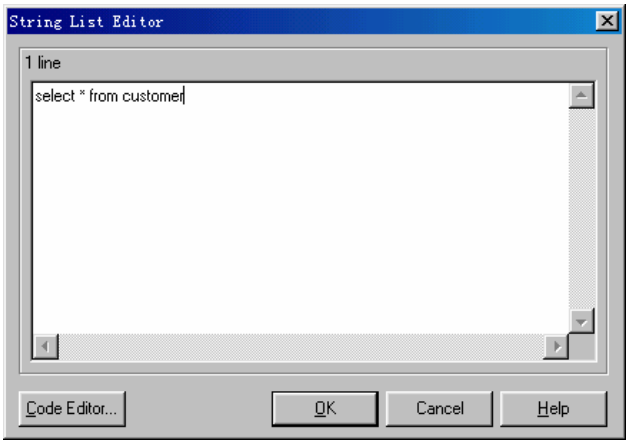

图 10.13 TQuery 的 SQL 特性编辑框

在文本编辑框的左下角有一个 Code Editor 按钮，它的作用是将当前编辑框中的文字转移到 Delphi 的代码编辑器中进行编辑。它适用于所编辑的文本非常长的情况，因为在上面的编辑框中可能显示不下。

这个编辑框在 Delphi 的很多地方都会遇到，它一般用来编辑 TStrings 类型特性中的文本，例如 TListBox 的 Items 特性。

TQuery 组件中有一个 Params 特性，它用来维护 TQuery 中可能用到的参数列表。在很多动态查询中，每次查询的条件都不同，那么 TQuery 的 SQL 语句中可以引用参数列表中的参数，只要在查询之前，通过将相应的参数设置为新的值，就可以动态地获得不同的查询结果。例如下面的 SQL 语句，它通过“:”号引用了参数列表中的一个参数 Param1，它的类型为 String 字符串，如下所示。

```
//设置参数 Param1 的值
Query1.ParamByName('Param1').AsString := '10000';
Query1.Close;
//通过参数来查询数据集
Query1.SQL.Text := 'select * from Customer where CustNo = :Param1';
Query1.Open;
```

TQuery 的 ParamCheck 是一个布尔类型的特性，它用来控制 TQuery 的参数在操作过程中是否起作用，如果将它设置为 False，那么 SQL 语句中通过“:”号引用的单词将不会被认为是参数，而会当做普通的字符串来看待。这种情况适用于 SQL 语句中包含文件路径的示例，用户当然不希望 TQuery 把 SQL 语句中的“C:\File”分解为参数“\File”，所以就必须将 ParamCheck 特性设置为 False。

TQuery 的另外一个重要的特性是 RequestLive，该特性的作用是表明返回的数据集是否是可修改的，如果希望返回的数据集可以修改，只要将 RequestLive 设为 True 即可。

提示：当需要对数据库表进行大数据量的操作的时候，例如删除一批数据、插入一批数据或者需要统一地编辑一批数据，使用 TQuery 提供的数据集操作方法，要

比利用 TTable 的 Insert、Edit 或 Delete 方法快速、方便、稳定得多。只不过，在 TQuery 对数据库表进行了改动之后，需要刷新数据库表才能使更改生效。

3．TStoredProc

TStoredProc 与 TTable 比较相似，TTable 用于连接数据库中的数据表，而 TStoredProc 则用于连接数据库中的存储过程。一个设计比较完善的数据库中应该包含一些具有一定功能的存储过程，它们一定程度上分担了应用程序操作数据库的负担，例如大型数据库中一般没有定义自动增长的字段类型，那么完全可以在数据库内部创建一个生成某个字段当前惟一值的存储过程，它的功能就是模拟一个自动增长字段，而不用在程序中自己来写这部分代码。

TStoredProc 通过 DatabaseName 特性关联到数据库上后，StoredProcName 特性可以从数据库中选择一个存储过程，执行存储过程很简单，如下所示：

```
StoredProc1.ExecProc;
```

4．TDataSource

图 10.14 为 Data Access 组件组，本节只介绍其中的 TDataSource 数据源组件。

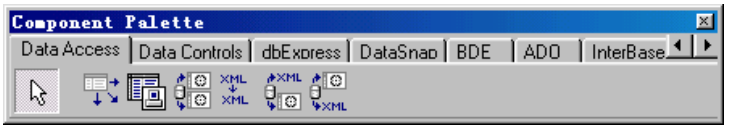

图 10.14　Data Access 组件组

TDataSource 的功能是作为一个中间起关联作用的中介。通过它，各种各样的 DataSet（数据集）中的数据可以输出到各种各样的数据感知组件中去。TDataSource 的特性相对少很多，最主要的一个特性是 DataSet，它用来指定 TDataSource 连接的数据集，可以是 TTable、TQuery 或者 TstoredProc 组件。

TDataSource 的 State 特性表示当前连接的数据集的状态，它的全部取值在表 10.1 中列出。这个特性的值能够表明数据集是处于插入、编辑还是正常浏览状态。

表 10.1　State 特性的取值

取值	含义
dsBrowse	数据集处于浏览状态
dsCalcFields	OnCalcFields 事件已发生，正在进行记录的计算
DsEdit	数据集处于编辑状态，表示 Edit 方法已经调用，但数据还没有保存
dsInactive	数据集已经关闭
dsInsert	数据集处于插入状态，表示 Insert 方法已经调用，但数据还没保存
dsSetKey	数据集处于设置键值状态，SetKey 已经调用，但 GotoKey 还没有调用
dsUpdateNew	数据集正在向对应的数据库表提交一个更新操作
dsUpdateOld	数据集正在撤消一个延迟的更新操作，并把数据恢复为原来的取值
dsFilter	数据集正在处理一个记录过滤器、查找字段或其他需用的过滤器的操作

5．数据感知组件

为了能使用户查看和操纵数据表中的数据，还必须加上 TData-Aware（数据感知）组件。下图显示了 Data Controls 组件组，它其中包含了大量的数据感知组件，例如最常用到的 TDBGrid 等，如图 10.15 所示。

图 10.15　Data Controls 组件组

TDBGrid 组件可以显示一个表的所有行和列，它是通过 DataSource 特性来关联数据表的。对每一列，即数据表中的字段，TDBGrid 可以直接显示字段名称，同时也可以创建对应于字段的 TColumn，并且设置显示标题和标题的颜色、字体和宽度等。如果不自己创建 TColumn，那么 TDBGrid 将自动显示全部字段，每一列都按照字段的名称和默认长度显示。如图 10.16 所示，它显示了这两种不同的情况，图 10.17 是对应于第二种显示的 TColumn 设置，可以通过双击 TDBGrid 组件来打开这个列表。

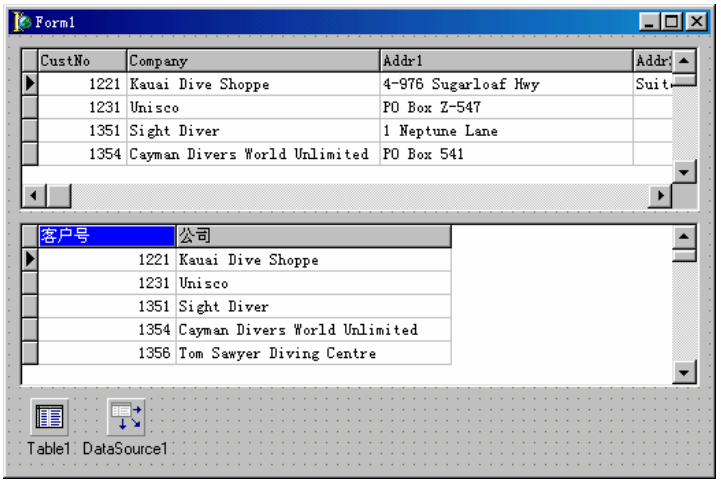

图 10.16　TDBGrid 的两种显示方式

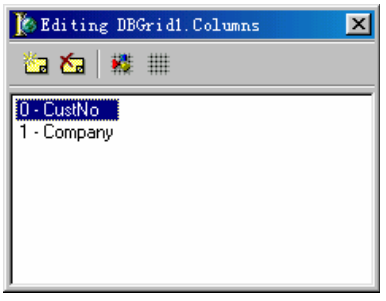

图 10.17　设置 TDBGrid 的 Column

TDBNavigator 组件用来操作数据表中的记录，它包括：向前、向后移动记录，将当前记录置到第一条和最后一条，删除当前记录，插入一条记录，编辑当前记录，提交修改结果，取消修改以及刷新记录等操作。

用户对 TDBNavigator 所需做的工作只是设置它的 DataSource 特性，而该组件能够根据数据表的当前状态自动地设置显示状态，例如，如果数据表的当前记录在第一条，那么向前移动记录和到第一条记录位置这两种操作自动失效，而如果数据表没有进入编辑状态，那么提交修改和取消修改两种操作也会失效，如图 10.18 所示就是这种状态。

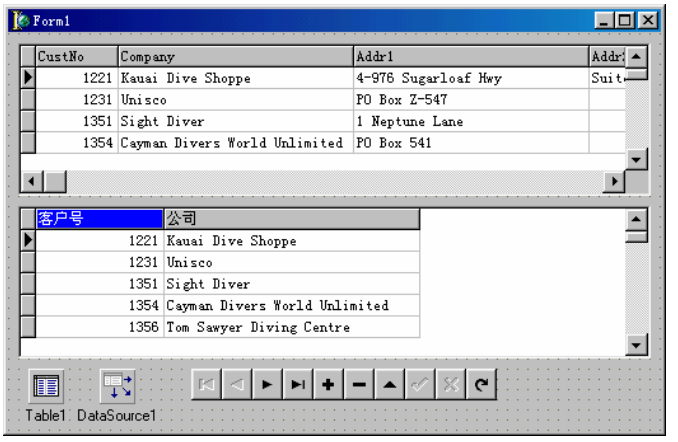

图 10.18　TDBNavigator 状态

TDBLabel、TDBEdit、TDBMemo 以及 TDBImage 等数据感知组件和普通的 TLable、TEdit、TMemo 以及 TImage 很类似，仅仅只是多了 DataSource 和 DataField 两个特性，它们分别用来对应数据表和数据表中的字段。

到现在为止，已经介绍了关于 BDE 的数据访问组件：数据表、数据源以及数据感知组件等，那么 10.3.3 小节将通过理论介绍基于这些组件基础上的数据库的连接。

10.3.3　数据库的连接

Delphi 系统中用来连接数据库的组件有 5 类：TSession（会话）、TDatabase（数据库）、TDataSet（数据集）、TDataSource（数据源）和 TData-Aware（数据感知）组件。这些组件的相互关系如图 10.19 所示。

TDataSet（数据集）对象，是 Delphi 数据库互连的基础，它为程序开发人员提供了一个反映数据库内部各实体的简单的表现形式；TDataSource 组件是联系数据集对象和数据感知组件等的桥梁；数据感知组件为用户提供输入和查看数据的界面。

在组件和数据库表之间还存在别的层可能有点让人糊涂，但实际上，这种复杂的设计避免了不少麻烦。通过改变 TDataSource 组件中 DataSet 属性，可以很容易地改变所有与 TdataSource 组件相连的数据感知组件。一旦改变了 DataSet 属性，所有的组件将会显示来源于新的 TTable、TQuery 或 TStored Proc 组件的数据。

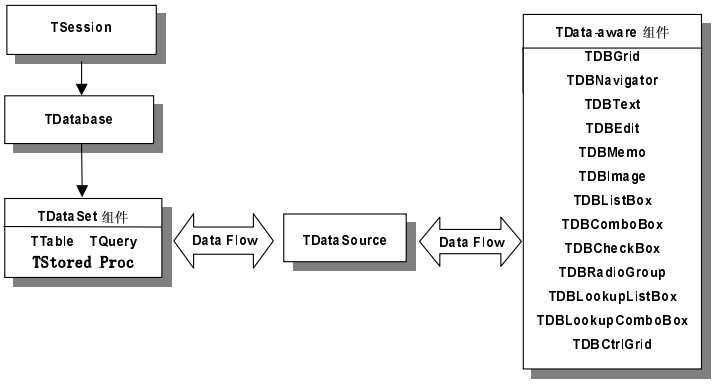

图 10.19　Tsession、TDatabase、TDataset、TDataSource **和** TData-aware **组件之间的关系**

10.4　数据库应用程序实例

了解了数据库应用程序的体系结构之后，现在到一个实例里面来学习创建一个数据库应用程序的方法，以及学习如何设置计算字段、查找字段、过滤器和主从表等数据库知识。

10.4.1　打开和关闭数据集

在对一个数据集进行操作之前，首先要调用 Open()**方法打开该数据集，同样，也可以把数据集的** Active **特性设为** True**，形式如下所示：**

```
TTable1.Open;
TTable1.Active := True;
```

使用后一种方式的开销稍微小一点，这是因为 Open()**方法最终也是把** Active **特性设为** True **来完成的，不过差别非常小，以至于可以不予考虑。**

数据集打开以后，就可以自由地操作浏览数据了。当完成了对数据集的使用后，一定要调用 Close()**方法来关闭它，如下所示：**

```
TTable1.Close;
```

同样有另外一种方法，如下所示：

```
TTable1.Active := False;
```

10.4.2　浏览数据集

TDataSet **提供了一些简单的方法来浏览数据集中的记录。其中，**First()**方法、**Last()**方法分别用于把当前的记录定位到数据集的第一个记录和最后一个记录。**Next()**方法和** Prior()**方法分别用于把当前记录向后和向前移动一条记录。另外，**MoveBy()**方法能够向前或向后跳过一定数量的记录。**

数据集中的两个特性 BOF **和** EOF **都是布尔类型，分别表示当前记录释放数据集的第**

一条或最后一条记录。例如，如果需要遍历数据集中的每一条记录，最容易的方法就是使用一个 while 循环从起点不断向后移动记录，直到 EOF 为真，如下所示：

```
begin
TTable1.First;
While not TTable1.EOF do begin
{Do something here}
TTable1.Next;
End;
end;
```

另外一个很有用的特性是数据集的书签 BookMark，它用来保存数据集中的记录位置，以便使数据集在移动以后可以重新回到那个位置。在 Delphi 中用 TBookMarkStr 来表达书签，TTable 的 BookMark 特性就是这种类型。当读取 BookMark 特性时就取到一个书签，而设置 BookMark 特性则可以跳到该书签所指示的记录位置，如下所示：

```
var BM: TBookMarkStr;
begin
BM := Table1.Bookmark;    //记录当前书签位置
Table1.BookMark := BM;    //回到 BM 书签记录的位置
End;
```

TBookMarkStr 是 AnsiString 类型分配给它的内存时自动管理的，因此不需要手动地释放它们。还有一种设置书签的方法，就是利用 BookMark、GetBookMark、GotoBookMark 和 FreeBookMark，不过，建议尽量少用这些方法，在稳定性和方便性上，它们远不如 TBookMarkStr。

10.4.3　一个实例

现在用一个实例来具体地说明数据集的各种特性和方法的使用。

图 10.20 为这个实例的主窗体。

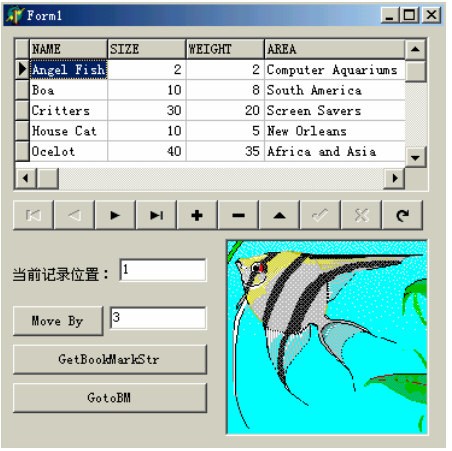

图 10.20　数据集应用程序实例

程序中的 DBGrid 显示了 Table 中的数据集，下面是具体实现数据显示的步骤：

（1）第一步，Table1 的 DatabaseName 属性设为一个已有的数据库别名或目录。实例中通过 DatabaseName 的下拉列表框选中了 DBDEMOS 这个别名。

（2）第二步，Table1 的 TableName 属性通过下拉列表框选中一个数据表，实例中选择了 animals.dbf 表。

（3）第三步，把一个 TDataSource 组件放在窗体上，设置 DataSet 特性为 Table1。

（4）第四步，在窗体上放置一个 TDBNavigator 组件，设置它的 DataSource 属性为 DataSource1，这个组件可以用来移动操作数据集中的数据记录。

（5）第五步，把 DBGrid1 的 DataSource 特性设置为 DataSource1，通过 DataSource1 建立了 DBGrid1 和 Table1 之间的联系。

（6）第六步，把 Table1 的 Active 属性设为 True，这样就打开了数据库表，同时在 DBGrid1 中就会看到 Table1 中的数据了。

技巧：对于 DataSet 或 DataSource 这样的特性，通过双击 Object Inspector 中相应的特性栏可以很便捷地设置它的值，而不必通过下拉框来选择。

由这个实例可以看出，Delphi 中的数据库类能够完成大量的数据操作，这些操作只需要很少的代码。

下面是这个实例的源代码：

```
unit Unit1;

interface

uses
    Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
    ExtCtrls, DBCtrls, Grids, DBGrids, Db, DBTables, StdCtrls;
type
    TForm1 = class(TForm)
        Table1: TTable;
        DataSource1: TDataSource;
        DBGrid1: TDBGrid;
        DBImage1: TDBImage;
        DBNavigator1: TDBNavigator;
        btMoveBy: TButton;
        Edit1: TEdit;
        btGetBM: TButton;
        btGotoBM: TButton;
        Edit2: TEdit;
        Label1: TLabel;
        procedure btMoveByClick(Sender: TObject);
        procedure btGetBMClick(Sender: TObject);
```

```
        procedure btGotoBMClick(Sender: TObject);
        procedure Table1AfterScroll(DataSet: TDataSet);

    private
{ Private declarations }

    public
{ Public declarations }

    end;

var
    Form1: TForm1;
    BM: TBookMarkStr;

implementation
{10R *.DFM}

procedure TForm1.btMoveByClick(Sender: TObject);
begin
    Table1.MoveBy(StrToInt(edit1.Text));
end;

procedure TForm1.btGetBMClick(Sender: TObject);
begin
    BM := Table1.Bookmark;
end;

procedure TForm1.btGotoBMClick(Sender: TObject);
begin
    Table1.Bookmark := BM;
end;

procedure TForm1.Table1AfterScroll(DataSet: TDataSet);
begin
    Edit2.Text := IntToStr(Table1.RecNo);
end;

end.
```

10.4.4 对数据集的操作

通过 TField 及其派生对象，可以访问任何数据集的字段，可以通过数据集的字段对象取得当前字段的值、设置字段的值，还可以改变字段的显示顺序或创建计算字段和查找字段来修改数据集。

访问数据集某一字段的值非常容易。可以通过 FieldbyName 的方法来获得，例如通过 Table1.FieldByName('fieldname').AsString 取得一个字符串类型的字段值，也可以通过 AsInteger 得到整型值的字段值。另外还有一种更简单的方法访问字段：通过一对中括号引用字段，例如访问 Table1 的 FieldName 字段的值，就是 Table1[FieldName]。实际上 TDataSet 类提供了一个默认的数组特性叫做 FieldValues[]，该数值按字段顺序返回了所有的字段值，但在代码中不需要显式地写出 FieldValues[]。后一种方法甚至不用作类型声明，函数自身能够判断和转换。

对上面的第二种方法进行扩展，可以把多个字段同时读入一个数组中去，因为多个字段的数据类型可能不同，所以必须将该数组定义为 Variant 类型，它能够接纳任何类型的数据。如下所示：

```
var MyFields: Variant;
MyFields := VarArrayCreate([0,2], varVariant);
MyFields := Table1['FieldName1; FieldName2'];
```

对于一个给定的字段对象，可以用 TField 的有关特性来取得字段的值或给字段赋值。它们是 AsBoolean（布尔型）、AsFloat（双精度浮点数）、AsInteger（整型）、AsString（字符串）、AsDateTime（日期时间类型）、Value（Variant 类型）。

编辑当前记录中一个或多个字段可以分为如下几个操作步骤：

（1）调用数据集的 Edit()方法，使数据集进入编辑状态。

（2）给字段赋值。

（3）调用 Post()方法来保存数据集。如果不调用该方法，而直接移动记录，数据集将自动调用 Post()方法。

下面是典型的编辑数据集代码：

```
Table1.Edit;
Table1.FieldByName('fieldName').asinteger := newValue;
Table1.Post;
```

在一个数据集中插入和追加记录的操作和上面的步骤类似，只需将调用 Edit()方法改为调用 Insert()方法和 Append()方法即可。

如果在数据保存之前放弃对数据集所作的修改，可以调用 Cancel 方法。如下所示：

```
Table1.Edit;
Table1[Fieldname] := newValue;
Table1.Candel;
```

这段程序中，对字段 FieldName 所作的修改因为调用 Cancel 方法而被取消。

10.4.5　TField 类型

TField 本身是个抽象类，无法创建它的一个实例，它是那些与数据库中字段类型相对应的字段类的祖先。字段类型既可以在 Delphi 运行时由 Delphi 自动创建，也可以由用户自己在程序设计时定义，而且可以通过对象监视器来管理它们的属性。例如，可以改变它们的显示宽度、对齐方式（左、右、中对齐），以及是否可见等属性。

10.4.6　字段编辑器

通过使用 Fields Editor(字段编辑器)，在使用数据集中的字段时就有了强大的控制功能和灵活性。在字段编辑器中，可以查看一个特定的数据集所包含的字段。要打开字段编辑器，可以双击一个数据集，或者右击数据集，在弹出的菜单中单击 Fields Editor 命令，如图 10.21 所示。

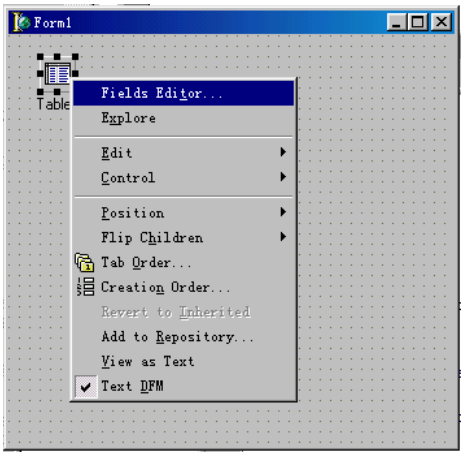

图 10.21　运行字段编辑器

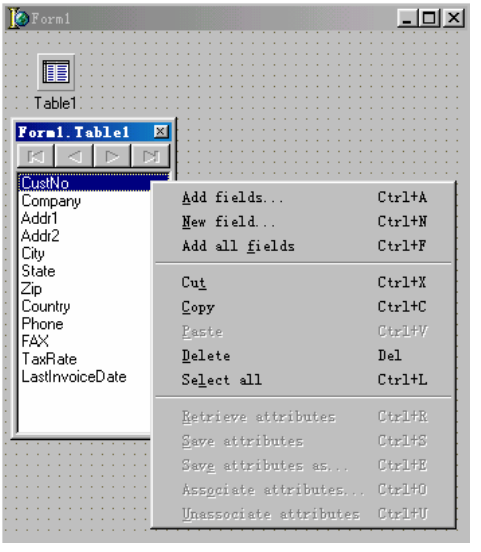

图 10.22　字段编辑器

在字段编辑器中，可以选中一个要操作的字段，进行该字段的各项特性设置。还可以增加新的计算字段和查找字段。图 10.22 为字段编辑器和它的右键菜单。图中，单击 Add fields 命令将弹出一个对话框，其中列出了除了在字段编辑器中显示的字段以外的所有字段，可以选择一个或者多个字段加入到字段编辑器中，也可以通过 Add all fields 命令添加全部字段。菜单中的 New field 命令用来创建新的字段，例如计算字段或者查询字段等，如图 10.22 所示。

Delphi 会为字段编辑器中的每一个字段建立一个字段对象，这些字段对象都是从 TField 继承下来的。例如图中的字段（见图 10.22），Delphi 在窗体单元中自动产生如下的声明，如下所示：

```
Table1CustNo: TFloatField;
Table1Company: TStringField;
Table1Addr1: TStringField;
Table1Addr2: TStringField;
Table1City: TStringField;
Table1State: TStringField;
Table1Zip: TStringField;
Table1Country: TStringField;
Table1Phone: TStringField;
Table1FAX: TStringField;
Table1TaxRate: TFloatField;
Table1LastInvoiceDate: TDateTimeField;
```

可以会发现，字段对象的名称是由表名和字段名两部分组成的，而字段对象的类型为 TField 的派生类，例如 TStringField 和 TFloatField，分别表示字符串字段和浮点数字段，Delphi 会根据实际字段的类型自动产生相应的字段类型。对于每个字段类型来说，都有一个或者多个 TField 派生对象与之对应，字段类型同时又对应于一个 Object Pascal 数据类型。例如：TStringField 对应字符串类型（String），TIntegerField 对应整数类型（Integer），TDateField、TTimeField 和 TDataTimeField 都对应于日期类型（TDataTime），还有一些字段类型没有相对应的 Object Pascal 类型，例如 TBlobField。

在字段编辑器中选中一个字段后，在 Object Inspector 中就可以访问到相应的 TField 派生类对象的特性和事件。

决定数据的输入和显示格式是创建数据访问窗体工作中最常见的一项任务，下面讨论几个常常用来设置显示格式的属性。

1．Currency 属性

Currency 属性只会出现在数字型的字段上，它是一个布尔类型的属性，有 True 和 False 两种状态。如果把某一个字段的 Currency 属性设置为 True，那么该字段将按照当前系统的货币设置显示。如果当前为中文系统，那么在数字前将会显示一个“￥”符号，而英文系统中将会显示一个“$”符号。并且，如果字段的实际值有多于 2 位的小数位，那么将自动

缩至 2 个小数位，按照正确的货币格式显示。

2．EditMask 属性

我们可以使用 EditMask 属性来修改字符串数据输入时的显示方式。该方法可以在字段中嵌入特殊字符并使它成为字段的一部分。例如，EditMask 中定义为文字符号的破折号和分号能作为表中字段的一部分显示出来。

为字符串字段创建 EditMask，首先在字段属性编辑器中选中该字段，然后在对象监视器中双击 EditMask 属性，系统将弹出一个 Mask 编辑器对话框，如图 10.23 所示。

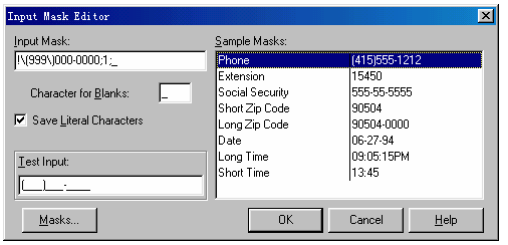

图 10.23　Mask 编辑器

Input Mask（输入掩模）编辑器的右边提供了一些系统中预先定义好了的样本掩模。选定就能使用。下端的 Test Input 文本框用来检查掩模的工作方式。定义好掩模后，可以在框中输入测试字符串，看一看掩模是否正确格式化了文本。Masks 按钮允许载入其他国家格式的掩模文件。这些文件的扩展名为 dem，存储格式为文本文件。例如，US.dem 文件包括了美国数据格式的掩模。可以用写字板等文本编辑器工具来打开它们查看，如下所示：

```
Phone |4155551212|!\(999\)000-0000;1;_
Extension |15450|!99999;1;_
Social Security |555555555|000\-00\-0000;1;_
Short Zip Code | 90504|00000;1;
Date | 062794|!99/99/00;1;
Long Time |090515PM|!90:00:00>LL;l;_
Short Time | 1345 | !90;00;1;_
```

如果不使用样本掩模，还可以输入掩模字符来定义用户自己的掩模，这里建议参考 EditMask 的在线帮助。例如，大于号“ > ”能将输入字符强制转换为大写，小于号则转换为大写，大写的“ L ”只能用做字母的掩模符号，而“ A ”既可用于字母也能用于数字。“ 9 ”只能用于数字。因此，如果要建一个掩模，强制用户输入 3 个大写字母，后面跟 4 个数字，则该掩模为：“ >LLL-9999 ”。然后就可以输入值“ NCC-1701 ”，而不是“ NCC-1701D ”。Delphi 6.0 提供了足够多的掩模字符供用户输入使用，甚至可以按照上面的示例使用文本编辑器建立自己的 DEM 掩模文件。

将电话号码字段的掩模定义为国际格式：99-999-000-000。这将强制用户输入后 7 位数字，而前 5 位数字的输入并不是必须的。建好掩模后，单击 Click 按钮返回到对象检视器中。

Delphi 的 TField 对象有一个重要特征：计算字段。应用程序可以像对表中的其他字段一样对计算字段操作，但表并不会保存该字段的值。也就是说，计算字段并不是真正存在于数据表中的字段，实际上，正如其名所示，它是应用程序基于表中的数值计算出的字段。

10.4.7 计算字段和查找字段

使用计算字段可以让用户在看到表中数据的同时，方便地得到由多个字段共同作用的值，而不必在表中占用空间、增加字段。

在字段编辑器中可以向数据集中加入一个计算字段。以数据库 DBDEMOS 中的表 orders.db 为例，操作如下：

首先，在字段编辑器中加入两个字段：ItemsTotal 和 TaxRate，当然也可以加入其他更多的字段。然后，在字段编辑器的右键菜单中执行 New field 命令，这时会出现一个 New Field 对话框，如图 10.24 所示。只需要在 Field Properties 中的 Name 文本框中写入新的计算字段的名称，这里是 TaxTotal，在 Type 下拉列表框中选择字段类型，这里选定的是 Currency 钱币类型。最后在 Field type 选项组中单击 Calculated 单选按钮，表示该字段为计算字段。确定以后，新的字段 TaxTotal 就加到了字段编辑器中。在这个示例中，计算字段 TaxTotal 的值表示税的总量，它的计算公式为 ItemsTotal 和 TaxRate 的乘积，即总钱数与税率的乘积。

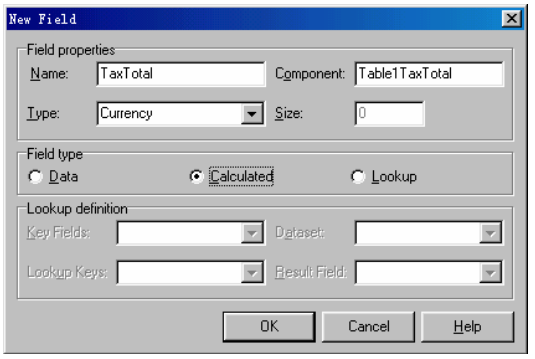

图 10.24 添加计算字段

创建好新字段后，仅仅告诉 Delphi 该字段为计算字段是不够的，还必须告诉它如何计算该字段、如何给该字段赋值，这里需要用到一个 OnCalcFields 事件。所有的 TDataSet 类都有一个 OnCalcFields 事件。该事件在应用程序需要得到计算字段的值时将被触发。例如上面的示例中，需要这样设置该事件，如下所示：

```
procedure TForm1.Table1CalcFields(DataSet: TDataSet);
begin
Table1['TaxTotal']:= Table1['ItemsTotal']* Table1['TaxRate'];
end;
```

下图为实现的结果界面，可以看到，创建的计算字段 TaxTotal 的值为前两个字段 ItemsTotal 和 TaxRate 的乘积，如图 10.25 所示。

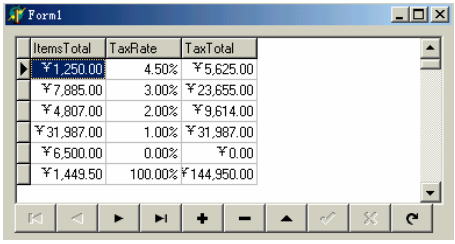

图 10.25　计算字段 TaxTotal 的值

如果对关系数据库中主从表理解得很好的话，那么对于查找字段的概念就很容易理解了。查找字段实际上是由一个字段作关联，从另外一个数据集中获取其他字段的数据，例如，当前数据集中只有一些人的 ID（标识号）和其他业务数据，而另外一个表中则有相应标识号的人的详细信息，例如名字、家庭住址等。这时候，希望将人的标识号和名字同时列在一个数据集中，查找字段在这里就起了关联数据的作用，即通过标识号获取名字信息。仍旧以上面的示例来说明查找字段的用法。

与计算字段不同，Field type 中要单击 Lookup，表示字段类型为查找字段。Key Fields 表示主表中的主字段，即客户号 CustNo，Dataset 为要关联的从表 Table2，Lookup Keys 为查找依据，表示 Table2 中要和主表 Table1 的主字段关联的字段，Result Field 表示要返回显示的字段。

在这里，需要再加入一个数据集 Table2，链接数据库 DBDEMOS 中的 customer.db 数据表，将其中的字段 CustNo 和 Contact 加入到字段编辑器中。在第一个数据集 Table1 中，通过字段编辑器的右键菜单中的 New Field 命令加入一个新的字段，如图 10.26 所示。图 10.27 为加入查询字段后的显示。

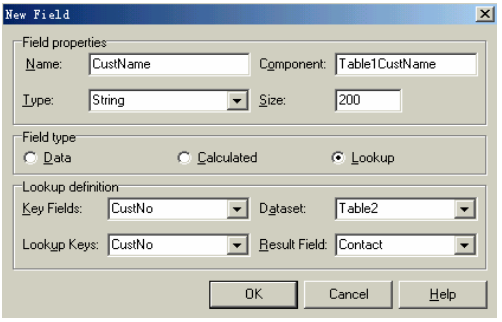

图 10.26　添加查找字段

图 10.27　查找字段 CustName

字段对象提供了一种带有强制性的有效性验证事件，它能保证字段中的数据符合预先定义的格式或值域等条件，例如保证输入的价格在某个取值范围内。有效性验证事件在 OnValidate 事件中进行处理，用户可以根据自己的需要对验证代码进行修改，例如检查字段之间的关系等。

10.4.8 过滤器

通过 Filter（过滤器），只要使用简单的 Object Pascal 代码就能完成一些简单的数据集过虑和搜索。使用过滤器最大的好处是不需要索引，当然如果有索引到话，查找的速度会更快一些。

过滤器的一个最普遍的作用是只显示数据集中的一些特定记录。下面介绍操作步骤：

（1）把数据集的 Filter 特性设为 True。

（2）建立一个处理数据集的 OnFilterRecord 事件句柄。在这个句柄中，根据一个和多个字段的值来决定是否要包含某个记录。

除了第（2）步的处理方法以外，也可以简单地设置数据集的 Filter 特性，它是一个字符串类型，例如：Table1.Filter ＝ ‘fieldname > 3’，表示 Table1 中只包含字段 fieldname 的值大于“3”的记录。

除此以外，TDataSet 还提供了一组方法用来筛选符合特定条件的记录，它们是：FindFirst、FindNext、FindPrior 和 FindLast。这些方法可以在处理 OnFilterRecord 事件的句柄中调用，根据这里给出的过滤条件，它们可以分别找到第一个、下一个、前一个和最后一个匹配的记录。这些方法都没有输入参数，其返回值则是一个表示是否找到匹配记录的布尔值。

使用过滤器不仅可以定义一个数据集的子集，而且也可以根据一个或多个字段的取值查找数据集中的记录。为此，TDataSet 提供了一个 Locate 方法()。由于 Locate()方法使用过滤器来进行查找，所有它与数据集中的任何索引都无关。Locate()方法的声明如下所示：

```
function TBDEDataSet.Locate(const KeyFields: string;
const KeyValues: Variant; Options: TLocateOptions): Boolean;
```

第一个参数 KeyFields 用于指定根据哪个字段进行查找，它可以是一个字段，也可以是一组字段。第二个参数 KeyValues 用于给出这些字段的值。最后一个参数 Options 用于指定查找的方式，它是一个 TLocateOptions 类型，该类型声明如下所示：

```
TLocateOption = (loCaseInsensitive, loPartialKey);
TLocateOptions = set of TLocateOption;
```

其中 loCaseInsensitive 表示区分大小写，loPartialKey 表示支持部分匹配。

该方法具体用法可以在 Delphi 提供的帮助中找到。

提示：要搜索记录，建议尽量使用 Locate()方法，因为它能够以最快的速度找到特定的记录，而且，它可以自动切换不同的索引。

10.4.9　主从表

在编写数据库应用程序时，经常会遇到这样的情况，即用到的数据分散在多个彼此相关的表中，这些表之间通过某一个或多个字段关联在一起。典型的示例是 Customers（客户表）和 Orders（订单表）。其中，每一个客户在客户表中对应一条记录，而一个客户在订单表中可能对应多条订单记录，即每个客户有多条订单，是一种一对多的关系。在这种情况下，称这种关联关系为主从表，这里 Customers 为主表，Orders 为从表，它们之间通过 CustID（客户号）关联在一起。

Delphi 可以很方便地建立数据表之间的主从关系，甚至在设计期通过 Object Inspector 简单地设置几个特性就可以建立起这种关系，不需要编写任何代码。起关联作用的特性就是 TTable 的 MasterSource 和 MasterFields 特性。下面就以客户和订单系统为例来说明主从表的实现。

首先，在窗体上放置两套 TTable、TDataSource 和 TDBGrid，它们分别是 Table1 和 Table2、DataSource1 和 DataSource2、DBGrid1 和 DBGrid2。第一套数据集连 DBDEMOS 库中的 Customer.db 表，第二套数据集连 Orders.db 表，它们分别显示在 DBGrid1 和 DBGrid2 中。具体的设置方法在本章前面的内容中已经介绍过了，这里不再赘述。设置完成以后，窗体显示如图 10.28 所示。

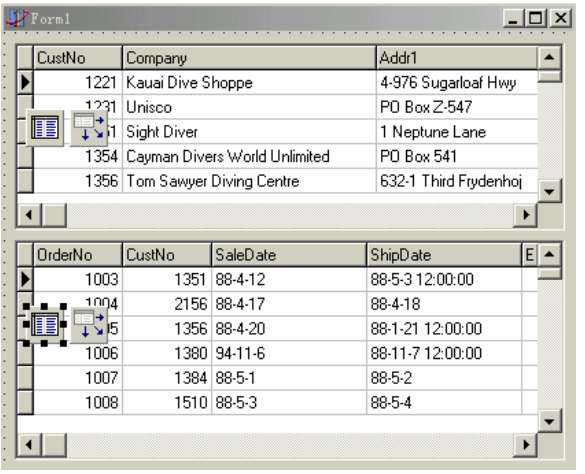

图 10.28　主从表实例

现在，窗体上放置的两个数据库表之间没有任何关联关系，接下来要做的就是建立它们之间的关联。首先，将 Table2 的 MasterSource 特性设为 DataSource1，表示 Table2 的主表数据源为 DataSource1；然后，双击 Table2 的 MasterFields 特性框对它进行编辑，这时将会出现一个 Field Link Designer 对话框，如图 10.29 所示。

在这个对话框中，需要指定通过哪些字段来把两个数据库表连接起来。本例中是通过两个表中的 CustNo 字段连接起来的，当然主表和从表的关联字段的名称不一定必须相同。在 Available Indexes 下拉列表框中选择 CustNo 字段。一旦选择了 CustNo 索引，就可以从 Detail Fields 和 Master Fields 列表框中分别选中 CustNo 字段，然后单击中间的 Add 按钮，

会发现 Joined Fields 文本框中出现了“CustNo->CustNo”，这就表示了两个数据表之间的连接。最后，单击 OK 按钮完成主从表关联字段的设定。

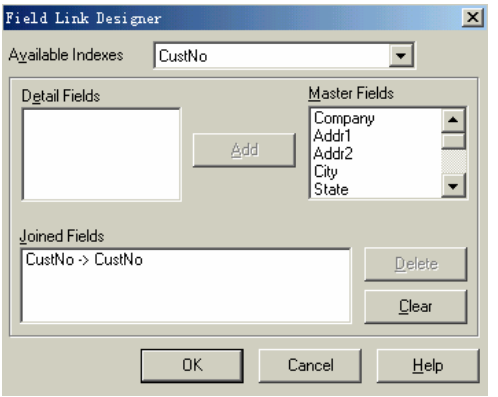

图 10.29 设置关联字段

运行这个实例，会看到当对 Table1 进行浏览的时候，Table2 中所显示的数据都是 CustNo 字段的取值和 Table1 中 CustNo 字段取值相同的记录，即 Table2 中显示了 Table1 中的客户所有的订单记录。

10.5 数 据 模 块

一个比较大的数据库应用程序中用到的各种数据集可能非常多，在这种情况下，就有必要使用数据模块。数据模块可以用来集中维护程序中用到的所有的数据库规则以及相互之间的关系。一个数据模块可以被多个项目、开发组甚至整个企业共享。

数据模块由 TDataModule 创建，可以把 TDataModule 看作一个不可见的 Form，在它上面可以放置整个项目所需要的所有的数据访问组件。执行 File|New 命令，然后从 Data Module 选项卡中选择一个 Data Module 就可以，在这一选项卡中，会看到一个 Customer Data 对象，可以直接创建它，然后将它上面的全部数据组件删除后加进自己需要的组件就可以了。

要在一个项目的多个窗体和单元之间共享数据，比较简单的办法就是把数据访问组件放在数据模块上，然后在其他的单元中直接引用它就可以了。在数据模块中，可以对多个 TTable、TQuery 和 TStoredProc 等组件进行适当地设置安排，包括定义它们之间的关联关系、字段显示格式等。当完成了这些设置工作之后，为了避免重复劳动，可以把创建的数据模块加入到对象库中以备今后使用。如果在一个团队环境下工作，也可以将数据模块共享给开发组内所有的开发者使用。

10.6 SQL 语 句

10.6.1 SQL 语句语法

SQL 是一组符合工业标准的数据库操作命令集，它可以在很多的开发环境中使用，例

如 Delphi、C＋＋Builder 等。

标准的 SQL 提供了功能强大的操作数据库的方法，例如查询、插入、编辑、删除等，以及大量的非常实用的函数，例如，求和、求最大最小值等。掌握这些方法和函数对数据库编程非常有用和方便。这里，仅介绍基本的 SQL 语句语法和使用，因为详细全面地介绍 SQL 需要一本书的篇幅。

1．查询

假设数据库表 table，其中有 fieldname1 和 fieldname2 两个字段，其中 fieldname1 为整型，fieldname2 为字符串类型。

查询语法如下所示：

```
select 字段列表 from table group by(字段列表) where 限制条件 order by (字段列表)
```

其中，字段列表中可以是一个或者多个字段的罗列，字段之间以“，”间隔（例如：fieldname1，fieldname2），如果需要将数据表中的全部字段选定，可以用字符“＊”代替；

group by(字段列表)子句表示按照括号内所列的字段进行分组显示，例如 group by(fieldname1)表示按字段 fieldname1 进行分组，具有相同的 fieldname1 字段取值的记录将只返回一条，代表一组。

Order by 子句表示查询的返回数据集将按指定的字段进行排序。

限制条件部分为对返回数据集的要求，有如下几种格式：

- 对某一个字段或多个字段取值的限制：fieldname1 = 1 and fieldname2 = ‘abc’。
- 对字段取值范围的限制：fieldname1 in (1,2,3)，表示要求 fieldname1 的取值在 1，2，3 的范围内；fieldname1 not in (1,2,3)，表示 fieldname1 的取值不在 1，2，3 的范围内。
- 嵌套条件：fieldname1 in (select fieldname3 from table2)，表示要求 fieldname1 的取值必须在从 table2 中选择的 fieldname3 的全部取值之内。当然括号内的子查询可以是更复杂的查询，但它必须要返回一个与 fieldname1 相符的字段。

2．插入

插入语法如下所示：

```
insert into table (fieldname1, fieldname2) values(10, ‘delphi’)
```

或者：

```
insert into table (fieldname1, fieldname2) select (fieldname3, fieldname4) from table1
```

第一种方法是直接向数据集中插入一条记录，第二种方法向数据集中插入一批由子查询返回的数据子集。

3．编辑

编辑语法如下所示：

```
update table set fieldname1 = 2, fieldname2 = ‘abc’ where 限制条件
```

限制条件子句的用法和查询语句中的限制条件相同。

Set 子句是对字段进行的实质操作，即将符合限制条件的数据集中的 fieldname1 字段全部设为 2，fieldname2 字段设为“ abc ”。该子句支持在原字段基础上进行编辑，例如：set fieldname1 = fieldname1 + 10，表示将字段 fieldname1 的值在原来的基础上加“ 10 ”。

4．删除

删除语法如下所示：

```
delete from table where 限制条件
```

5．函数

Max(fieldname1)：该函数返回字段 fieldname1 所有值中的最大值。相应的 Min(fieldname1)返回最小值，如下所示：

```
select Max(fieldname1) as maxvalue, Min(fieldname1) as minvalue from table;
```

该 SQL 返回的数据集只有一条记录，即 table 中字段 fieldname1 的最大值和最小值。As 的作用是将最大最小值分别附给一个新的字段名，以便程序中引用它们。这样，就可以用字段 maxvalue 来引用 fieldname1 的最大值，用 minvalue 来引用 fieldname1 的最小值。

Sum(fieldname1)：该函数返回字段 fieldname1 的所有值的总和。

10.6.2 动态 SQL

动态 SQL 是指在运行过程中根据不同的情况来修改 SQL 语句。通过 TQuery 的 SQL 编辑器可以输入静态的 SQL 语句，除非在运行期对 SQL 进行修改，否则，这个 SQL 语句将不会改变。例如：Select * from Customer where country = ‘China’，就是静态的。

动态 SQL 可以通过 SQL Params（参数）来实现。上面的示例这样修改就变成动态的了，如下所示：

```
Select * from Customer where country = :iCountry;
```

在这个语句中，没有把要查找的值固定，而是用一个参数来代替，这个参数叫 iCountry，前面紧跟一个冒号，用来标识参数。

对于参数化查询来说，可以有多种方式来提供参数值，一种是通过 TQuery.Params 特性的特性编辑器，另一种是在运行期提供参数，还可以通过 TDataSource 组件从其他数据集中得到参数值。下面就简要说明几种参数提供方式：

1．通过 Params 特性编辑器提供参数值

在 TQuery 的 SQL 特性中设置好 SQL 之后，打开 TQuery.Params 的特性编辑器，在 Parameter Name 列表框中列出了该查询用到的参数。对于每一个参数，必须通过 Data Type 下拉列表框选择它的数据类型，在 Value 文本框中指定参数的初始值，当然参数也可以没有初始值。调用 TQuery.Open 方法时，TQuery 将根据设置好的参数值自动返回一个数据集。

2．通过 Params 提供参数

TQuery 中有一个 Params 对象，它是一个数组，每个 Params 对象代表 SQL 语句中用到的一个参数，参数在数组中的位置就是它在 SQL 中的出现位置，因此可以通过索引号到 Params 数组中取得指定的参数，索引号从“0”开始，即第一个用到的参数的索引为“0”。例如下面的 SQL 语句：

```
Insert into Country(Name, Capital, Population) Values(
:NAME, :CAPITAL; POPULATION)
```

在 Params 特性中，生成了 3 个参数，它们分别是：

- Params[0]为:NAME;
- Params[1]为:CAPITAL;
- Params[2]为:POPULATION;

这样，就可以通过 Params 数组来对参数进行设置了，如下所示：

```
with TQuery1 do begin
Params[0].AsString := ‘China’;
Params[1].AsString := ‘Beijing’;
Params[2].AsInteger := 120,000,000;
end;
```

3．调用 ParamByName()方法提供参数

除了用 Params 特性以外，还可以用 ParamByName()方法来提供参数。使用 ParamByName()的好处是可以直接按照参数名称来对参数赋值，而不必考虑参数的顺序。这种方法增加了代码的可读性，使程序一目了然，但是一定程度上也降低了代码的效率，因为 Delphi 必须对参数名进行分析。仍然以上面的示例来说明 ParamByName()的用法，代码如下所示：

```
With TQuery1 do begin
ParamByName(‘NAME’).AsString := ‘China’;
ParamByName(‘CAPITAL’).AsString := ‘Beijing’;
ParamByName(‘POPULATION’).AsInteger := 120,000,000;
End;
```

读者也许注意到了，ParamByName 的使用和数据集的 FieldByName 的使用方法很相似。

4．从另外一个数据集中获取参数

查询的参数也可以来自另外一个数据集，例如 TQuery、TTable 等。首先通过 TDataSource 在两个数据集之间建立主从表的关系。TQuery 的 DataSource 特性应设置成连接另外一个数据集的 TDataSource。这样，Delphi 将在 DataSource 特性所指定的数据集中查找与 SQL 语句中的参数名相匹配的字段，如果找到，就绑定该字段和参数。如下所示：

```
Query1.SQL.Text := ‘Select * from orders where CustNo = :CustNo’;
```

如果 Query1 的 DataSource 特性为连接 Table1 的 DataSource1，这时候，Delphi 将在 Table1 中查找一个叫 CustNo 的字段，并绑定该字段和 SQL 语句中的:CustNo 参数，自动取该字段的值赋给参数。

当一个查询操作返回结果集之后，可以使用 TQuery 组件来访问结果集中的数据。这里的 TQuery 完全可以看作一个 TTable，可以用 FieldByName、FieldValues['fieldname']或 TQuery['fieldname']等方法来获取其中的数据。

注意：FieldValues()方法的返回类型为 Variant，它可以自动转换成相应的数据类型。但是，如果一个字段的值为空，当通过 FieldValues()方法获取这个字段的值时，会触发一个 EVariantError 异常。这时候，要特别小心地处理这种情况。

FieldByName()方法如果是针对整型值，如果字段的值为空，该函数会自动将空转换为“0”，但是在 SQL 数据库中，空是合法的，即空和“0”不是一回事。

10.7 连接数据库

ADO 作为一种连接数据库的技术，它的使用必须结合 OLE DB、ODBC 或者 MS Jet 才能起作用。Delphi 提供了一组 ADO 组件来使用 ADO 技术，这组组件如图 10.30 所示。

图 10.30 ADO 组件组

从 ADO 组件组中选择一个 ADOConnection 组件放到一个窗体上，然后打开 ConnectionString 特性编辑器，或者双击该组件，将出现一个 ConnectionString 对话框，如图 10.31 所示，Source of Connection 选项组默认选择的是 Use Connection String 来关联数据源。Connection String 特性是一组用分号隔开的信息，它们包括 OLE DB 提供者、数据源、数据库位置、用户名和密码等。当 ADO 通过 Connction String 的信息连接到指定的数据库上之后，ADO 会根据实际的连接情况更改 Connction String 的设置，增加一些连接信息。

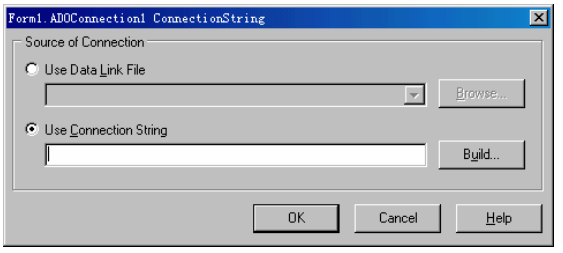

图 10.31 ADOConnection 的 ConnectionString 编辑器

单击 Build 按钮，将进入数据连接属性编辑器，如果 ADOConnection 是第一次进行 ConnectionString 的设置，将显示 Provider（提供者）选项卡，如图 10.32 所示；如果

ConnectionString 中已经包含了关于 OLE DB Provider 的设置，那么这时候将直接进入编辑器的第二个选项卡——Connection（连接），这一选项卡的内容根据 OLE DB 提供者的不同而有所不同。

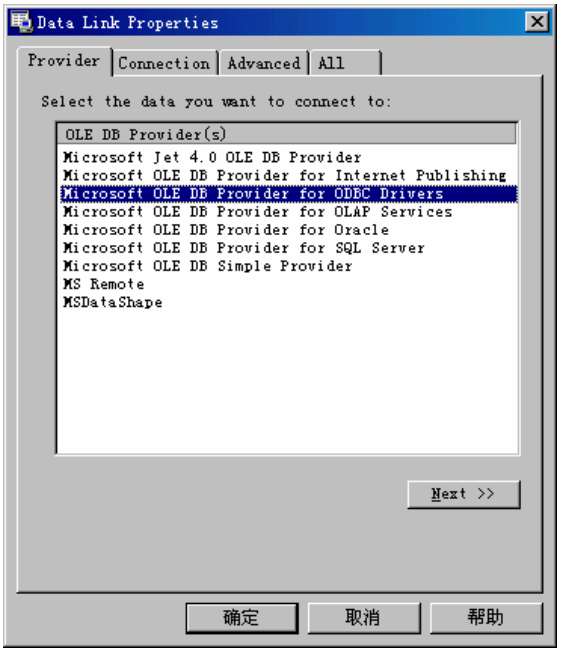

图 10.32　Provider 选项卡

先来看 Provider 选项卡（见图 10.32），编辑器自动列出了系统中已经安装的所有 OLE DB Provider，这里主要对比讨论其中的两种：Microsoft Jet 4.0 和 ODBC Drivers。

其中后者——Microsoft OLE DB Provider for ODBC Drivers 是 ADO 和 ODBC 的结合，它底层实际上使用了 ODBC 中的各种数据源或者数据驱动，本质上与 BDE 和 ODBC 的结合使用是相同的，但是 ADO 比 BDE 在存取数据的速度上要快很多。它的 Connection 选项卡如图 10.33 所示，通过 Use data source name 下拉列表框列出了系统 ODBC 中所有创建的用户数据源，其中有的数据源已经关联到特定的数据库上了，但有的只是一个数据驱动而已。对于前者，这里只要选定数据源实际上就已经选定了数据库，单击 Test Connection（测试连接）按钮，就会出现一个“测试连接成功”的对话框，它表示已经连接到指定的数据库上了。对于后者，当选定数据源以后，还必须通过下拉列表框 Enter the initial catalog to use 来选择一个目录或者文件，来确定具体的数据库的位置。如果数据库的访问限定了特定用户，可以在登录信息中设定用户名和密码。对于已经关联到特定数据库上的数据源，仍然可以通过数据库目录来设定新的数据库。

图 10.34 是 Provider 为 Microsoft Jet 4.0 OLE DB Provider 时，Connection 选项卡的内容。它提供了另外一种连接 Access 数据库的方法。只要指定一个 Access 数据库的路径，然后根据情况设定访问用户名和密码后，就可以连接到数据库上。Microsoft Jet 与 ODBC 相比，Microsoft Jet 在访问和存取数据的速度方面要快一些。

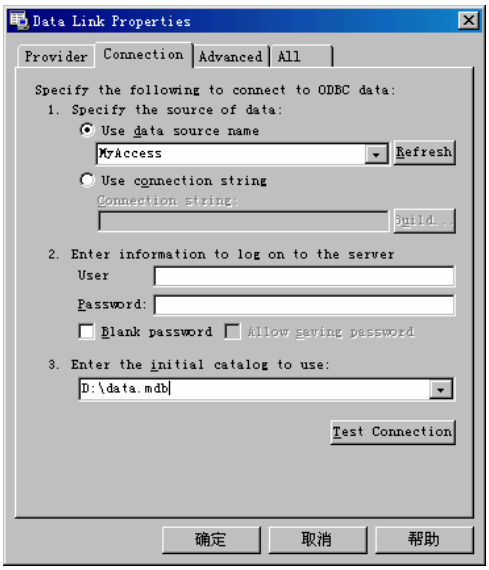

图 10.33　Connection **选项卡**

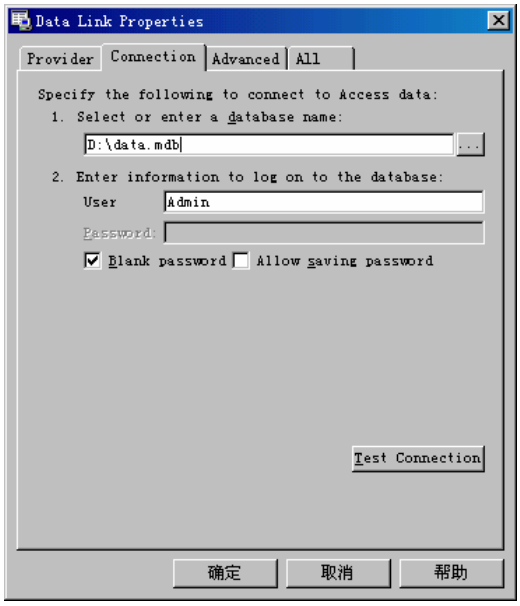

图 10.34　MS Jet 4.0 **连接数据库**

这时候，单击“ 确定 ”按钮后，将结束 ConnectionString **的设置，会发现** ConnectionString **特性的值已经增加了很多信息。**

使用 ADOConnection **组件在运行期如何动态的设置数据库呢？其实，只要根据在设计期设置数据库后** ConnectionString **的格式，在程序中把其中** Data Source **部分相应地替换成要连接的数据库路径就可以了，而根本不需要进行其他的特性设置，因为** ConnectionString **中包括了所有的连接设置。**

TADOTable 组件相当于 DataAccess 组件组中的 TTable，它具有一个 Connection 特性，用来指定已有的 ADOConnection 组件，通过这个特征来确定数据库。同时 TADOTable 还有一个 ConnectionString 特性，同 TADOConnection 的 ConnectionString 特性一样，ADOTable 可以通过它来指定数据源和数据库。TableName 特性用来指定数据库中的数据表。

TADOQuery 组件与 TQuery 组件一样用来执行 SQL 语句，实现对数据库的操作。

技巧：在 TADOQuery 的 SQL 语句中，通过一个数据库的路径就可以直接访问和存取指定的任意一个数据库，如下所示：

```
Select * from C:\AccessFile.mdb.TableName;
```

这个语句将从位于 C:\AccessFile.mdb 的数据库中取出表 TableName 的所有记录。但是如果在路径名中存在“%”、“*”或空格等特殊字符，SQL 语句在运行时将认为有语法错误，如下所示：

```
Select * from C:\My Documents\AccessFile.mdb.TableName;
```

执行该 SQL 时会出现语法错误的提示。解决这个问题的办法是将路径名用一对中括号括起来，这样修改上面的 SQL 语句，如下所示：

```
Select * from [C:\My Documents\AccessFile.mdb].TableName;
```

这时，将成功地取出位于 C:\My Documents\AccessFile.mdb 的数据库中的 TableName 表的全部记录。

注意：应用上面的技巧的时候，和 TQuery 组件一样，为避免 SQL 语句中的冒号使 TADOQuery 将 C 盘符后面的路径名误认为是参数，必须将 TADOQuery 组件的 ParamCheck 特性设为 False，这样 TADOQuery 将不会按冒号来检查参数。

10.8　dbExpress

dbExpress 是 Delphi 6.0 提供的一套全新的数据存取技术，它目前能够对 DB2、Interbase、MySQL 和 Oracle 等数据库进行数据存取，而且数据存取的速度完全可以与传统的 BDE 相比，甚至能够超过 BDE 的速度。图 10.35 是 Delphi 6.0 中的 dbExpress 组件组。

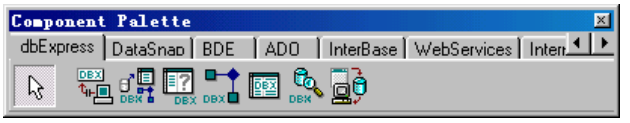

图 10.35　dbExpress 组件组

与普通的数据库组件一样，dbExpress 在应用方面同样非常简单，它包含 TSQLConnection、TSQLDataSet、TSQLQuery、TSQLStoredProc 以及 TSQLTable 等组件，它很类似于 ADO 组中的一套数据库组件。其中，TSQLConnection 用来连接一个数据库，TSQLTable 用于关联数据库中的数据表，TSQLQuery 用于在数据库中进行 SQL 查询或者数据操作，TSQLStoredProc 用于关联数据库中的存储结构。

就像 TADOConnection 一样，TSQLConnection 通过设置 ConnectionName 特性指定数

据驱动来存取一个数据库，可以在对象监视器中进行静态设置，也可以在程序中动态地设置 ConnectionName，下图中分别显示了这个特性的设置方法，如图 10.36 所示。

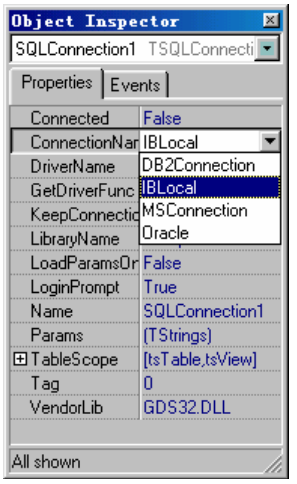

图 10.36　TSQLConnection 的 ConnectionName 连接

图中显示了系统默认的 4 种 ConnectionName（见图 10.36），它们分别对应于 DB2、Interbase、MySQL 和 Oracle 4 种数据库。ConnectionName 一旦确定后，DriverName 会自动设置成相应的驱动名称。

用户也可以编辑自己的 ConnectionName。具体方法是双击 TSQLConnection 组件，将弹出一个对话框，如图 10.37 所示。

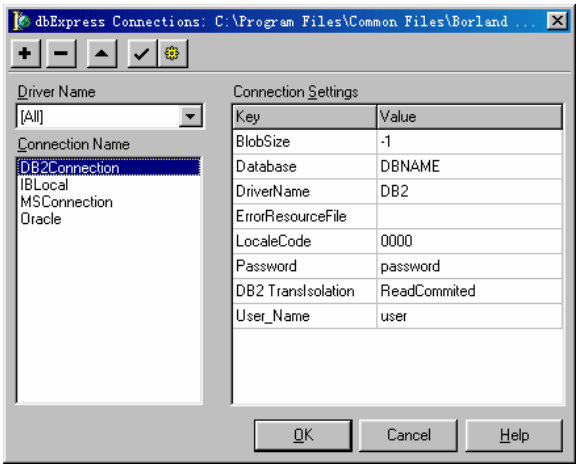

图 10.37　编辑 dbExpress Connection

图中左面显示了一个 Connection Name 列表（见图 10.37），其中每一个 ConnectionName 都关联了一个 Driver Name，即数据驱动名称。在右面显示了对应于左面的 Connection Name 的设置，因为每种数据库的各种设置项都不完全相同，它实际上是由数据驱动类型，即数据库类型决定的。

选定了 ConnectionName，即确定了数据库类型以后，通过 TSQLConnection 的 Params 特性可以进行数据库的各种具体设置，也就是图中右面的各项设置项（见图 10.37）。以 Interbase 数据库为例来介绍 Params 特性的设置方法，图 10.38 为 Params 特性的设置界面。

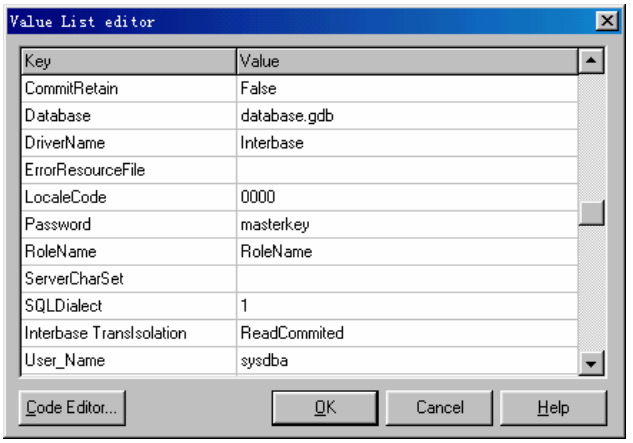

图 10.38　Interbase 数据库的 Params 设置

图中（见图 10.38），系统已经默认设置好了很多项，例如，DriverName 为 Interbase，用户名和密码都是系统默认的 sysdba 和 masterkey，这是 Interbase 数据库默认的管理员名和密码，该用户具有超级用户的一切权限。这时，用户必须自己来设置数据库文件路径，即 Database 项，它其实就是有关 Interbase 数据库文件（即*.gdb 文件）的路径位置。

读者可能已经发现了，Params 特性其实是一个 TStrings 类型的对象，查看它的 String 属性，就是以回车（换行）分隔开的一系列设置项，例如上面的 Params 的 String 如下所示：

```
BlobSize=-1
CommitRetain=False
Database=database.gdb
DriverName=Interbase
ErrorResourceFile=
LocaleCode=0000
Password=masterkey
RoleName=RoleName
ServerCharSet=
SQLDialect=1
Interbase TransIsolation=ReadCommited
User_Name=sysdba
WaitOnLocks=
```

了解了这一点以后，就知道该如何在程序运行期间对 Params 进行动态地设置了。当然，如果用户做的应用程序专业性比较强的话，甚至可以模仿 Params 编辑器做一个 Connection Setting 对话框让用户来自己配置数据库，这里其实就是一个 TValueListEditor 组件而已。

像 TADOTable 设置 Connection 属性来关联一个 TADOConnection 一样，类似的，TSQLTable 同样需要通过 SQLConnection 属性来指定一个 TSQLConnection 组件，然后设置 TableName 来指定一个数据库中的表。

相应的，TSQLQuery 和 TSQLStoredProc 分别类似于 ADO 中的 TADOQuery 和 TADOStoredProc 组件，具体设置方法几乎完全相同，读者可以参考前面章节中的相关内容。

虽然 dbExpress 在表面上的配置和设置方法都没有什么很特别的地方，但是实际上，dbExpress 在内部的数据驱动方面却比 BDE 有很大的改进，这些改进导致了 dbExpress 强大快速的数据存取速度和性能。

10.9 本 章 小 结

本章阐述了通过 ODBC 创建数据源的各种方法和步骤，以及使用 BDE 管理器来创建数据源的方法。

本章详细地介绍了数据库编程中最基本的知识，其中包括 TDataSet 的组件细节，以及由它派生出来的 TTable、TQuery 等组件。在此基础上，进一步介绍了怎样操作 TTable 对象、怎样管理字段并介绍 TQuery 中简单的 SQL 语句语法。

通过本章的学习，读者已经具备了开发简单的传统数据库应用程序所需的基础知识和技巧，它是以后更加深入地学习 Client/Server 或其他技术性更高的数据库知识的一个坚实基础。

最后介绍了 Delphi 6.0 全新的 dbExpress 数据存取技术和相应的 dbExpress 组件组。

第 11 章　COM 基 础

支持 COM、ActiveX 和 OLE（对象链接与嵌入）Delphi 的重要功能。这一章将完整地介绍这些技术，并帮助用户把这些技术应用到程序中。传统的 OLE 技术能够把一个应用程序链接或嵌入到另一个应用程序中，例如把一个 Excel 电子表格嵌入到 Word 文档中去。不过，本章要讨论的 COM 技术远没有这么简单。

本章将首先介绍有关 COM、OLE 和 ActiveX 基本的背景知识，然后简单地介绍分布式 COM（DCOM）技术，更详细的 DCOM 技术将在第 12 章专门介绍，除此以外，还会介绍 Automation（自动化操作）的实现。本章的最后将介绍一个典型 OLE 的 VCL 组件——TOLEContainer，它是一个 ActiveX 容器。在这里并不打算讲述关于 COM、OLE 和 ActiveX 的所有知识，因为这些知识完全可以写一本书来讲解，所以只将重点放在 COM、OLE 和 ActiveX 的主要特点上，尤其是这些技术在 Delphi 中的应用。

学完本章后，读者应该可以理解关于 COM 的 Delphi 实现是如何构成的。即使读者对 COM 技术不是特别感兴趣，也应该非常了解接口以及在程序中如何使用接口。

11.1　COM 基 础

在进入 COM 技术主题之前，先要介绍有关这些技术的一些基本概念。其中包括 COM、OLE 和 ActiveX 等基本概念。

11.1.1　COM（组件对象模型）

COM（Component Object Model，组件对象模型）是 OLE 和 ActiveX 技术的基石。它定义了一组 API 和一个二进制标准，使来自不同语言、不同平台的彼此独立的对象之间相互通讯。COM 类似于 VCL 对象，不同之处是 COM 对象只能有特性和方法，不能有字段。

COM 既可以在 EXE（可执行文件）中实现，也可以在 DLL（动态链接库）中实现，这种实现对于 COM 对象的用户来说是透明的，因为 COM 提供了一种 Marshaling（聚合）的服务。COM 的这种聚合机制能够实现跨进程边界甚至跨机器边界的函数调用，它能够在 16 位的应用程序中访问 32 位的对象，或者在一台机器上访问另外一台机器上的对象，这种跨机器的通讯称为 DCOM。

从历史上看，Windows 操作系统经历了从 16 位到 32 位的变迁，在 16 位平台即 Windows 3.x 上，虽然系统已经按照组件模块的结构建立起来，但模块与模块之间大多并没有采用 COM 接口，它仅仅提供了对 OLE 的支持，因此这种组件模型的优势并没有充分发挥出来。在 32 位 Windows 版本中，不管是 Windows95/98 还是 NT，很多系统部件都是以 COM 的形式实现的，除了考虑与以前版本的 SDK 兼容之外，一些新增的组件均提供了 COM 接口，这样做的好处是，不仅使各种开发语言可直接调用系统提供的功能，

而且也有利于在特殊情况下对组件的单独升级，而这种部分升级对于 MS-DOS 系统和 16 位 windows 系统来说是很困难的事情。

在 Windows 操作系统平台上，有一些用 COM 形式提供的组件模型极大地丰富了系统的功能，而且也使 Windows 功能扩展得更加灵活，如下所示：

- DirectX（多媒体软件包）。它以 COM 接口的形式为 Windows 平台提供了强大的多媒体功能，现广泛用于游戏娱乐软件以及其他多媒体软件的开发中。
- RDO（Remote Data Object，远程数据对象）和 DAO（Data Access Object，数据访问对象）数据库访问对象库。它以 COM 自动化对象的形式为数据库应用提供了便捷的操作方法。而数据访问一致接口——OLE DB/ADO（ActiveX Data Object，活动数据对象）更淋漓尽致地发挥了 COM 接口的作用。
- Internet Client SDK。它提供了一组 COM 库，为应用系统增加 Internet 特性提供了底层透明的一致操作。
- 其他还有一些组件例如 MAPI（Message API，消息应用编程接口），ADSI（Active Directory Service Interface，活动目录服务接口）等，它们都提供了一致、高效的服务。从整个 Windows 操作系统看，COM 成了系统的基本软件模型，它带来的是灵活性和高效率，以及应用开发的一致性。

在所有的编程工作中，很少有东西能像 COM 一样庞大，COM 可能是编程世界中曾经采用的最大的编程成就。在 Delphi 中可以很容易地使用 COM 技术，实际上，Delphi 从本质上重新编写了 COM。

那么 COM 技术到底能为用户做什么呢？COM 赋予了程序员完成以下 4 种事情的能力：

- 编写可以供多种编程语言使用的代码。
- 创建 ActiveX 控件。
- 通过 OLE 自动化来控制其他程序。
- 与其他设备上的对象或者程序会话。

当把这些特性结合在一起的时候，就可以找到解决许多问题的方案。这种方案很强大，甚至 Microsoft 以后可能会以 COM 作为 Windows 的基本接口，也就是说，Windows 将是一个 COM 接口的集合。即使是在现在，Windows 中的许多服务都是以 COM 为基础的。例如，资源管理器主要是通过 COM 进行访问的，可以使用 Shell API 的 COM 接口集合对系统界面的某些特性进行编程设置；Word、Excel 可以通过 COM 实现完全自动化。这么说也许不是很好理解，下面举几个示例来说明 COM。

读者也许已经知道，在一个 Word 文档中可以嵌入一个 Excel 表格、画笔图片等。如果还不知道，现在就打开一个 Word 文档，然后执行“插入”|“对象”命令，选择插入画笔图片，确认后，一个画笔图片连同画笔程序环境一起出现在 Word 里；简单地在图片上画几条线之后，使焦点离开画笔图片，这时候，就会看到这副图片已经嵌在了用户的文档里面了。这就是 COM 实现的结果。同样，如果用户正在编写一个 Delphi 程序，其中需要用到

强大的字处理或者电子表格功能，达到这一目的的一种方法就是利用 COM 使 Word 或者 Excel 嵌入进应用程序中来。

在理论上，没有任何理由可是使程序员不用一系列的 COM 服务器来创建他们的应用程序。这些子程序可以根据需要插接到一个主程序中，就像堆积木一样，每一个木块就是一个 COM 服务器。

11.1.2　COM 的问题和未来

COM 并不是完美的，指出它的一些缺陷是值得的。COM 最大的问题就是它是以 Windows 为中心的。Microsoft 的主要目标也许就是在 Windows 上开发 COM，并将它的用户锁定在 Windows 操作系统上。

COM 的另一个问题是一个插入型或者分布式结构可能有潜伏的错误。这些错误经常是由于一个程序或者一个融合系统的不同部分之间缺乏真正的兼容性而产生的。

如果将一个有错误或者错误地使用了 OLE 规范的程序和 Word 融合到一起，那么 Word 和 Windows 都将崩溃。崩溃可能是由于粗心地编程或者 OLE 规范本身的错误导致的。

而且，目前的软件供应商明显地缺乏合作的精神，而这正是解决类似这样的问题所需要的。例如，今天的软件供应商一般只在程序中加入只有他们才知道的客户接口或者他们没有正确地提供资料的接口。

11.1.3　COM、OLE 和 ActiveX 的异同

许多初学者问的最多的问题也许是：“COM、OLE 和 ActiveX 到底有什么区别呢？”，这个问题很难说清楚，因为 Microsoft 本身就没有把这些事情说清楚。前面说过，COM 是一组 API 和二进制标准，它也是其他相关技术的基石。在 1995 年以前，OLE 仅仅是指与对象链接和嵌入有关的技术，例如容器、服务器、就地编辑、拖放和菜单合并。1996 年，Microsoft 提出了 ActiveX 的概念。ActiveX 又成为了一个新的神秘术语来描述建立在 COM 基础上的非 OLE 技术。ActiveX 技术包括自动化控件、文档、容器、脚本以及一些 Internet 技术。

ActiveX 技术最伟大的地方就是使应用程序能够轻松地操纵许多类型的数据。例如，一个应用程序可以包含现成的具有文字处理等功能的 ActiveX 控件，而不需要把文字处理、电子表格、图象处理的功能都加到应用程序中。

ActiveX 完全符合 Delphi 的传统，即最大程度地实现代码重用。不需要专门写代码来操纵某种类型的数据，只要使用相应的 OLE 服务器程序就可以了。

11.1.4　COM 技术中的术语

COM 技术带来了许多新的技术，因此，在进一步讨论 ActiveX 和 OLE 之前，先来介绍一些术语。

尽管 COM 对象的实例通常像一个普通对象那样被引用，但通常所说的 COMduix 的类型是指 CoClass（组件类）。因此，要创建一个 COM 对象的实例，必须知道要创建的 COM 对象的 CLSID。

称在应用程序之间共享的一大块数据为一个 OLE 对象，称能够包含 OLE 对象的应用

程序为 OLE 容器，反过来，称能够让它的数据包含到另一个应用程序中的程序为 OLE 服务器。

称一个包含一个和几个 OLE 对象的文档为复合文档。一个文档可以包含 OLE 对象，也可以包含一个应用程序，这种应用程序称为 ActiveX 文档。

正如它们的名字所暗示的那样，OLE 对象可以链接或嵌入到一个复合文档中，链接的对象仍然保存在一个磁盘文件中。通过对象链接，多个容器甚至服务器程序可以链接同一个 OLE 对象。当一个应用程序修改了链接的 OLE 对象，所作的修改将被反应到所有包含这个链接对象的应用程序中。嵌入的对象保存在 OLE 容器中。只有容器程序才可以编辑 OLE 对象。嵌入的好处是其他程序无法访问数据，但也增加了容器程序管理数据的负担。

关于 ActiveX 的一个重要内容是 Automation。Automation 允许一个应用程序操纵另一个应用程序的对象或库。前者叫做 Automation 控制器，后者叫做 Automation 服务器。

11.1.5 COM 的线程模式

每个 COM 对象都是在一个特定的线程模式下运作的。线程模式决定了一个对象在多线程环境下被操纵的方式。当要注册一个 COM 服务器时，应当为服务器所包含的每个 COM 对象指定一种线程模式。如果用 Delphi 编写 COM 对象，Automation、ActiveX 控件和 COM 对象向导会提示用户指定一种线程模式。COM 线程模式可以设为以下几种：

- Single　整个 COM 服务器工作在单线程下。
- Apartment　也称为 MTA（单线程单元）。每个 COM 对象在一个单独的线程中执行，同一类型的 COM 对象的多个实例也运行在各自单独的线程中。因此，凡是需要在对象实例之间共享的数据必须用线程同步对象来保护。
- Free　也称多线程单元。一个 OLE 对象的多个实例可以同时运行。这意味着 COM 对象必须保护自己的实例数据，以避免多个线程冲突。
- Both　同时支持 Apartment 和 Free 两种线程模式。

任何一种线程模式，并不能保证 COM 对象在那种线程模式下一定是安全的，必须自己编写必要的代码来保证 COM 服务器在指定的线程模式下正常工作。这通常需要借助于线程同步对象来保护 COM 对象中的全局变量或实例数据。

11.2 接　口

经过上面的介绍，现在读者已经对 COM 有所了解了，现在开始研究它在操作系统和 Delphi 代码中是如何实现的。接口是这个问题的核心，它将极大地扩展程序的功能，当完全理解接口的时候，就可以利用 COM 和 Delphi 来执行强大的功能。

11.2.1 接口简介

COM 定义了一个对象的函数怎样在内存中布置的标准。这些函数被安排在 vTable 虚方法表中，类似于 Delphi 的 VMT（虚方法表）。编程语言对每个 vTable 的描述就是所谓的接口。Delphi 程序员可以把接口仅仅看成一个用于管理对象的真正的抽象公共方法，例

如把一个标准的 VCL 对象除去 protected 和 private 两个部分，然后把剩下的方法声明为一个虚拟抽象对象，这个对象就非常类似于接口。

接口可以看作是一种特别的类，它所包含的一组函数和过程可以用来操纵一个对象。例如，一个表达位图的 COM 对象可能支持两个接口：一个接口包含了使位图能够输出到屏幕或打印机的方法，另一个接口用于位图的文件管理。

和普通的 VCL 对象的公共接口不同，COM 接口没有实现部分，它的方法是真正抽象的。一个接口只需为访问一个对象而定义一组公共方法，但对那个对象的实际实现没有任何说明，在这个意义上，接口非常类似一个标准 VCL 对象的真正抽象的公共接口。

一个 COM 接口实际上分为两个部分：第一个部分是接口定义部分，这部分按一定顺序声明了一些方法，这部分是 COM 对象与这个对象的用户共享的；第二部分是接口实现的部分，实际上就是具体实现接口定义部分所声明的方法。接口定义就好像 COM 对象与它的客户之间的约定，它可以保证客户按照事先约定的名称和顺序来调用方法。

Delphi 中 Interface 关键字使用户可以轻松地声明一个 COM 接口。声明一个接口非常类似于声明一个类，但有几个地方不同：接口只有特性和方法，没有数据。由于接口不包含数据，特性只能通过方法被访问。最重要的是，接口并不实现自己，因为接口只是一个约定。

就像 Object Pascal 所有的类都是从 TObject 继承下来的一样，所有的 COM 接口都是从 IUnknown 接口继承下来的。IUnknown 是一个基类，任何没有实现 IUnknown 的接口都不是一个真正的 COM 接口，因为 Delphi 中的所有接口的实现必须支持 IUnknown，所以 Delphi 接口总是支持 IUnknown。

在 System 单元中，IUnknown 是这样声明的，如下所示：

```
type
    IUnknown = interface
        ['{00000000-0000-0000-C000-000000000046}']
        function QueryInterface(const IID: TGUID; out Obj): Integer; stdcall;
        function _AddRef: Integer; stdcall;
        function _Release: Integer; stdcall;
    end;
```

关于 IUnknown，在 11.2.5 一节中将更加详细地讨论。

除了 Interface 关键字之外，接口与类的另一个区别在于：接口必须有全局惟一标识符 GUID。GUID 是一个 128 位的整数，它用来标识一个接口、coClass（组件类）或者其他实体。由于它的长度达到了 128 位，因此可以完全保证每个 GUID 是惟一的。

可以通过 CoCreateGUID()的 API 函数产生 GUID，API 函数能够综合考虑当前的时间、日期、CPU 时钟序列、网卡编号等因素来生成 GUID。如果用户的机器中安装了网卡，因为每块网卡的编号都是惟一的，所生成的 GUID 肯定是惟一的；如果没有安装网卡，函数就会考虑其他硬件信息。产生这个数字的算法是由 Open Software Foundation 创建的。

在调用 CoCreateGUID()函数之前，必须首先通过调用 CoInitialize()来初始化 COM，然后才能得到 GUID 的一个实例。

由于没有任何一种编程语言能够表达 128 位的数据，所以，Object Pascal 用一个叫作 TGUID 的记录来表达。在 System 单元中，TGUID 是这样声明的，如下所示：

```
PGUID = ^TGUID;
TGUID = packed record
        D1: LongWord;
        D2: Word;
        D3: Word;
        D4: array[0..7] of Byte;
end;
```

除了上面记录的表示形式以外，Object Pascal 还允许把这个记录用字符串的形式来表示，如下所示：

['{15AF0BC1-81FD-11D5-BF59-FD54F6B9DBC9}']。

在 COM 中，任何一个接口或者类都有一个 GUID，所以，即使两个接口需要相同的名称，但它们的 GUID 也肯定是不相同的。当 GUID 表示一个接口的时候，它也称为 IID，即 Interfact ID（接口标识符），当 GUID 表示一个类的时候，也称为 CLSID，即类标识符。

提示：在 Delphi 中，快捷键 Ctrl + Shift + G 键可以自动产生一个 GUID。

11.2.2 声明接口类型

可以使用关键字 interface 来声明一个新的接口类型，如下所示：

```
IMyObject = interface(IUnknown)
        Procedure MyProcedure;
end;
```

这个简单的接口具有一个 MyProcedure 的方法。实际上，完全可以省略 IUnknown 关键字，可以这样声明，如下所示：

```
IMyObject = interface
        Procedure MyProcedure;
end;
```

上面两个接口的声明是完全一样的，就像下面的两个类的声明是完全一样的，如下所示：

```
TMyObject = class
end;

TMyObject = class(TObject)
end;
```

这两个声明完全一样，因为全部的 Delphi 类在默认情况下都是从一个名为 TObject 的类中继承下来的，所以，即使没有声明 TMyObject 从什么类中继承，它仍然是从 TObject 继承而来。这种情况和 IUnknown 完全一样。

接口看起来很像类，但却不是类，例如接口中没有数据，没有字段，也没有实例数据。接口不能从类中继承，而必须从零开始或者从另一个接口中继承。这样的声明是完全非法的，如下所示：

```
IMyObject = Interface(IUnknown)
        FData: Integer; //这是非法的。
        Procedure MyProcedure;
End;
```

这个声明不正确，是因为不能在一个接口内部声明任何数据，这违反了规则。

另外，接口的全部成员默认为公共的，实际上，不能在接口上加入任何范围语句。这样的声明是非法的，如下所示：

```
IMyObject = Interface(IUnknown)
        Private //  这个范围限定是非法的。
Procedure Myprocedure;
End;
```

这个声明不正确，是因为不能在一个接口声明中使用范围限定命令，例如 private、public、protected 和 published 等。所有方法默认声明为公共的。

11.2.3　实现接口

一定要记住，永远不能通过接口自己实现一个接口，相反，而是应使它成为一个类的一部分。使用接口是为一个对象创建规范的方法之一，而不是声明一个对象的方法。下列部分将不会被编译，如下所示：

```
interface
type
        IMyobject = interface(IUnknown)
                Procedure Myprocedure;
        End;

Implementation

Procedure IMyobject.Myprocedure;
Begin
        ShowMessage('Hello');
End;

End.
```

这里的问题是企图声明一个 IMyObject.Myprocedure 方法的实现，不能直接实现任何接口方法，相反的，应利用类来实现接口。

下面的一个示例介绍了如何利用类来实现接口。它包括两个单元，第一个单元 Un_MyInterface 中声明了一个新的接口，第二个单元 Un_Main 中引用了第一个单元，并实现了 3 种调用接口或者对象的方法。

1. 单元一

声明一个接口并实现其中简单的组件，如下所示：

```
unit Un_MyInterface;

interface

type
    IMyInterface = interface
        function GetName: string;
    end;

    TMyClass = class(TInterfacedObject, IMyInterface)
        function GetName: string;
    end;

implementation

function TMyClass.GetName: string;
begin
    Result := '这是我创建的接口';
end;

end.
```

2. 单元二

使用 3 种方法来调用上面的接口中的方法，如下所示：

```
unit Un_Main;

interface

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

type
    TForm1 = class(TForm)
```

```
        btnUseObject: TButton;
        btnUseInterface1: TButton;
        btnUseInterface2: TButton;
        procedure btnUseObjectClick(Sender: TObject);
        procedure btnUseInterface1Click(Sender: TObject);
        procedure btnUseInterface2Click(Sender: TObject);
    private
        { Private declarations }
    public
        { Public declarations }
    end;

var
    Form1: TForm1;

implementation

uses Un_MyInterface;

{11R *.DFM}
```

//直接调用对象的方法

```
procedure TForm1.btnUseObjectClick(Sender: TObject);
var
    MyClass: TMyClass;
begin
    MyClass := TMyClass.Create;
    ShowMessage(MyClass.GetName);
    MyClass.Free;
end;
```

//实现一个接口，并调用其中的方法

```
procedure TForm1.btnUseInterface1Click(Sender: TObject);
var
    MyInterface: IMyInterface;
    MyClass: TMyClass;
begin
    MyClass := TMyClass.Create;
    MyInterface := MyClass;
    ShowMessage(MyInterface.GetName);
end;

procedure TForm1.btnUseInterface2Click(Sender: TObject);
```

```
var
    MyInterface: IMyInterface;
begin
    MyInterface := TMyClass.Create;
    ShowMessage(MyInterface.GetName);
end;

end.
```

程序的主界面如图 11.1 **所示。**

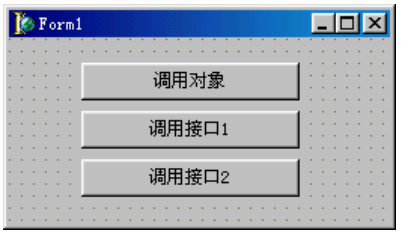

图 11.1　**接口调用演示程序**

在这个程序中，接口的声明如下所示：

```
IMyInterface = interface
    function GetName: string;
end;

TMyClass = class(TInterfacedObject, IMyInterface)
    function GetName: string;
end;
```

这个简单的接口声明了一个 GetName()**的简单方法，该方法在** TMyClass **中的实现如下所示：**

```
function TMyClass.GetName: string;
begin
    Result := 'You got my Interface';
end;
```

可以看到，这个方法的实现没有特殊的地方，它非常简单。但是，是什么将 TMyClass.GetName()**方法连接到** IMyInterface.GetName()**方法上的呢？**TMyClass **和** IMyInterface **这两个语法实体之间是什么关系呢？为了理解这个问题，需要很好地分析** TMyClass **的声明，如下所示：**

```
TMyClass = class(TInterfacedObject, IMyInterface)
```

这行代码说明 TMyClass 从另外一个称为 TInterfacedObject 的类继承下来的，并且它实现了一个 IMyInterface 的接口。

所以，TMyClass 是从一个标准的 TInterfacedObject 类和一个 IMyInterface 的接口继承而来。它声明了一个方法，这个方法实现了在 IMyInterface 接口中声明的 GetName()方法。将 TMyClass.GetName()和 IMyInterface.GetName()连接起来的方法很简单，因为这两种方法具有相同的名字。

对于 TInterfacedObject 这个 Delphi 类，它是一个比较特殊的类，它实现了某些工作，使从它继承来的类可能自动和容易地实现一个接口。特别是，TInterfacedObject 实现了 IUnknown 的方法，正如 TMyClass 实现了 IMyInterface 的惟一方法一样。

> 注意：TMyClass 并不支持真正的多重继承，虽然它看起来好像多重基础性。因为当一个类从至少两个类继承而来时，它才具有真正的多重继承性。但这里 TMyClass 是从一个类（TInterfacedObject）和一个接口（IMyInterface）继承而来的，而接口和类是完全不同的，接口不能实现任何方法。

上面示例的第二单元 Un_Main 调用了该接口的方法，其中包括 3 个按钮，分别提供了 3 种不同调用接口的方法。第一个 UseObject 的方法除了创建 TMyClass 的一个实例和调用它的方法以外，什么也不做。

第一种调用方法如下所示：

```
procedure TForm1.btnUseObjectClick(Sender: TObject);
var
    MyClass: TMyClass;
begin
    MyClass := TMyClass.Create;
    ShowMessage(MyClass.GetName);
    MyClass.Free;
end;
```

这个示例是一个简单的代码，在 Delphi 中实现非常容易而且很容易理解，在这里写出它只是为了和后面的方法进行对比。

第二种调用方法是从 TMyClass 中获得接口然后调用 IMyInterface.GetName()方法，如下所示：

```
procedure TForm1.btnUseInterface1Click(Sender: TObject);
var
    MyInterface: IMyInterface;
    MyClass: TMyClass;
begin
    MyClass := TMyClass.Create;
    MyInterface := MyClass;
    ShowMessage(MyInterface.GetName);
end;
```

在这里，首先创建了一个 TMyClass 的 MyClass 实例，然后从这个实例中得到接口，如下所示：

```
MyInterface := MyClass;
```

这行代码非常重要，它允许用户通过一个简单的赋值语句创建一个到接口的引用。

在得到接口的一个实例以后，就可以直接调用该接口实例的方法了。这实际上和调用 TMyClass.GetName()是一样的。也就是说上面两种调用方法其实完成了相同的事情，惟一的区别在于其中一个利用接口，而另一个不是；或者说，一个是普通的 Delphi 对象，而另一个是合法的 COM 接口。

第三种方法是从 TMyClass 中获得接口，然后调用 IMyInterface.GetName()方法的另外一种方法：

```
procedure TForm1.btnUseInterface2Click(Sender: TObject);
var
    MyInterface: IMyInterface;
begin
    MyInterface := TMyClass.Create;
    ShowMessage(MyInterface.GetName);
end;
```

接口不需要手动地释放，因为 Delphi 会自动地完成这一操作。如果需要手动地释放一个接口，不能调用 Free()方法或者 Release()方法，这些方法的使用都会产生非法访问，正确的方法应该是将接口设置为 nil，如下所示：

```
MyInterface := nil;
```

这个方法会自动调用 Release 这个 COM 方法，也会调用 TMyClass 的析构函数。必须清楚地记住这一点，它非常重要。

注意：可以通过将一个接口设置为 nil 的方法来释放接口，但是不能将一个类设置为 nil 来释放类。这是因为一个接口仅仅是一个抽象、一个方法。接口不能离开实现它的对象而独立存在。上面示例中能够创建的惟一的东西是 TMyClass，IMyInterface 仅仅是 TMyClass 和 TInterfacedObject 的功能的一些子集的抽象。

11.2.4 使用接口的原因

此时，读者已经了解了足够的关于接口的知识。下面进一步学习为什么要使用接口。

使用接口的原因大概有下面几个。第一个也是最明显的原因是：它们在 Object Pascal 内部为 COM 对象提供了实现的方法，第二个原因是：它们为创建标准或者规范提供了一个简单的方法。如果定义了一个接口，那么用户本人或者别人就可以根据一个预先定义的容易理解的标准来实现它。比较相似的是 TStrings 和 TStream 类在 VCL 中扮演了类似的角色，它们是一种抽象的类，这些类主要用于定义行为，而其他一些更加具体完整的类，例如 TFileStream、TStringList，则是行为的执行者。

除此以外，接口还提供了定义一组面向对象类的行为的很好的方法。这种类型的典型的示例是 ActiveX 类和 Windows Shell Extensions。

实际上，与传统的 OOP 提供的技术相比，接口为定义一组对象提供了更好的方法，随着不断地使用这个技术，用户会更多地看到它的价值。

不可否认，接口具有某些方面明显的限制，但是在它们自己的基础结构上找到接口崩溃的情况是非常困难的；另一方面，标准的 OOP 看起来对每个问题都有答案，但是没有好的方法以限制 OOP，有时候 OOP 的复杂程度会呈几何增长。

编程的最终目的总是为了解决某个实际问题，而因为许多现实世界问题所固有的复杂性，编程其实是很难的。最好的方法是不要企图用复杂而难以管理的工具解决那些复杂的问题。最好的工具是最简单的工具，它提供了优良的可变性和性能。

通常接口创建的代码比标准的 OOP 要快，深而复杂的分层结构降低了 OOP 程序的速度。在本质上，接口并不鼓励创建深的分层结构，而鼓励相对简单的、具有良好性能的程序。

11.2.5　接口的维护和更新

任何一个应用程序或者组件，都不可避免的在维护过程中进行版本更新，但是对于接口，作为一个规则，决不能试图更新或者改善它，在创建它并将它公布给公众以后，它就是最终的方案，而且永远不能再修改。

如果确实需要更新接口，那么通常应该创建接口的一个新版本，并用一个新的名字来公布它，以区别与旧的的版本。

例如，笔者已经公布了一个接口 IMyInterface，它的声明如下所示：

```
IMyInterface = interface(IUnknown)
        procedure GetName;
end;
```

一段时间之后，这个接口需要更新为下面的声明，如下所示：

```
IMyInterface 1= interface(IUnknown)
        procedure GetName;
        procedure GetID;
end;
```

这个新接口除了支持一个称为 GetID 的新函数之外，与原来的 IMyInterface 接口是完全一样的。

如果一个开发者希望访问 IMyInterface1 中的新特性，它便可以利用这个新接口，但是与此同时，对象必须仍然支持 IMyInterface 接口，以便不会破坏原来的代码，也就是说，必须考虑创建一个新的对象服务器，它支持 IMyInterface 和 IMyInterface1 两个接口。

理论上，在这种系统后面找到一个漏洞是很容易的，但是事实上，它在实践中工作得非常好。例如上面的两个接口，它们提供了 GetName()方法的同样实现，而只有新的或者更新的方法才不得不需要重新实现，以支持新接口。所以，系统并不会非常浪费，而且，这里说明的技术只是一个标准或者容易理解的版本控制方法。标准 OOP 没有为版本类确定协议。

11.2.6 理解 IUnknown

实际上，IUnknown **是** COM **用来在它的对象上进行计数的一种机制。一个** COM **对象可以有很多个客户，因此，不能因为一个客户释放了这个对象就简单地破坏这个对象，相反，必须只在整个对象破坏的时候释放它。为了更加准确地理解这个过程是如何进行的，本节准备了下面这个示例。在这个示例中包括两个单元，其中** UserUnknown **单元实现了** IUnknown **接口的简单** COM **对象，如下所示：**

```
unit UserUnknown;

interface

uses
    windows, Dialogs;

type
    TNotifyEvent = procedure(Sender: TObject) of object;

    TUserUnknown = class(TObject, IUnknown)
    protected
        FRefCount: Integer;
        function QueryInterface(const IID: TGUID; out obj): HResult; stdcall;
        function _AddRef: Integer; stdcall;
        function _Release: Integer; stdcall;
    public
        property RefCount: Integer read FRefCount;
    end;

    IName = interface(IUnknown)
        ['{F17EF920-82CB-11D5-BF59-E3228A875871}']
        function GetName: string;
    end;

    INumber = interface(IUnknown)
        ['{F17EF921-82CB-11D5-BF59-E3228A875871}']
        function GetNumber: integer;
    end;

    TMyClass = class(TUserUnknown, IName, INumber)
    private
        FOnDestroy: TNotifyEvent;
    public
        constructor Create;
```

```
        destructor Destroy; override;
        function GetName: string;
        function GetNumber: Integer;
        property OnDestroy: TNotifyEvent read FOnDestroy write FOnDestroy;
    end;

implementation

{ TUserUnknown }

function TUserUnknown._AddRef: Integer;
begin
    Inc(FRefCount);
    Result := FRefCount;
end;

function TUserUnknown._Release: Integer;
begin
    Dec(FRefCount);
    Result := FRefCount;
    if Result = 0 then
        Destroy;
end;

function TUserUnknown.QueryInterface(const IID: TGUID; out obj): HResult;
const
    E_NOINTERFACE = 1180004002;
begin
    if GetInterface(IID, obj) then
        Result := 0
    else
        Result := E_NOINTERFACE;
end;

{ TMyClass }

constructor TMyClass.Create;
begin
    inherited Create;
    MessageBeep(0);
end;

destructor TMyClass.Destroy;
```

```
begin
    if Assigned(FOnDestroy) then
        FOnDestroy(self);
    inherited Destroy;
end;

function TMyClass.GetName: string;
begin
    Result := 'IName';
end;

function TMyClass.GetNumber: Integer;
begin
    Result := 42;
end;

end.
```

UserUnknown COM **对象的客户程序主单元如下所示：**

```
unit un_Main;

interface

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

type
    Tfm_Main = class(TForm)
        gbName: TGroupBox;
        gbNumber: TGroupBox;
        btnNameCreate: TButton;
        btnNameRelease: TButton;
        btnNumberRelease: TButton;
        btnNumberCreate: TButton;
        gbReCount: TGroupBox;
        leNameCount: TLabeledEdit;
        leNumberCount: TLabeledEdit;
        btnCreateMyClass: TButton;
        btnUpdate: TButton;
        procedure btnCreateMyClassClick(Sender: TObject);
        procedure btnUpdateClick(Sender: TObject);
        procedure btnNameCreateClick(Sender: TObject);
```

```
        procedure btnNameReleaseClick(Sender: TObject);
        procedure btnNumberCreateClick(Sender: TObject);
        procedure btnNumberReleaseClick(Sender: TObject);
    private
        { Private declarations }
        FMyClass: TMyClass;
        FName: IName;
        FNumber: INumber;
        procedure EnableButtons(Value: Boolean);
        procedure MyClassDestroyed(Sender: TObject);
    public
        { Public declarations }
    end;

var
    fm_Main: Tfm_Main;

implementation

{11R *.dfm}
procedure Tfm_Main.EnableButtons(Value: Boolean);
begin
    btnNameCreate.Enabled := Value;
    btnNumberCreate.Enabled := Value;
    btnNameRelease.Enabled := Value;
    btnNumberRelease.Enabled := Value;
end;

procedure Tfm_Main.btnCreateMyClassClick(Sender: TObject);
begin
    btnCreateMyClass.Enabled := False;
    FMyClass := TMyClass.Create;
    btnUpdateClick(nil);
    EnableButtons(True);
    FMyClass.OnDestroy := MyClassDestroyed;
end;

procedure Tfm_Main.MyClassDestroyed(Sender: TObject);
begin
    EnableButtons(False);
    btnCreateMyClass.Enabled := False;
end;
```

```
procedure Tfm_Main.btnUpdateClick(Sender: TObject);
begin
    if FName <> nil then begin
        leNameCount.Text := IntToStr(FName._AddRef - 1);
        FName._Release;
    end else
        leNameCount.Text := '0';

    if FNumber <> nil then begin
        leNumberCount.Text := IntToStr(FNumber._AddRef - 1);
        FNumber._Release;
    end else
        leNumberCount.Text := '0';
end;

procedure Tfm_Main.btnNameCreateClick(Sender: TObject);
begin
    if FMyClass <>  nil then
        FName := FMyClass;
    btnUpdateClick(nil);
end;

procedure Tfm_Main.btnNameReleaseClick(Sender: TObject);
begin
    FName := nil;
    btnUpdate.Enabled := True;
end;

procedure Tfm_Main.btnNumberCreateClick(Sender: TObject);
begin
    if FMyClass <>  nil then
        FNumber := FMyClass;
    btnUpdateClick(nil);
end;

procedure Tfm_Main.btnNumberReleaseClick(Sender: TObject);
begin
    FNumber := nil;
    btnUpdateClick(nil);
end;

end.
```

程序的主界面如图 11.2 所示。

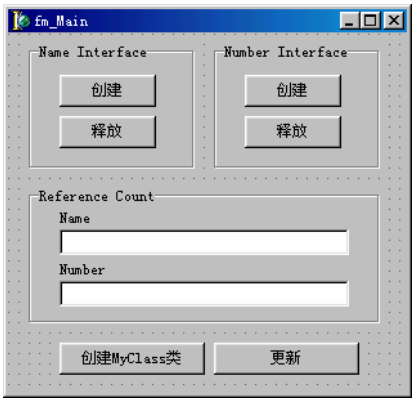

图 11.2　UserUnknown COM 对象客户程序

这个示例允许创建和破坏两个接口：IName 和 INumber。它告诉用户如何自己实现 IUnknown，但是需要强调的是，笔者并不建议用户在自己的程序中使用它，只需要使用 TInterfacedObject 或者它的某个子类就行了。

下面在上面示例的基础上来进一步讨论 IUnknown 是如何工作的。

下面的类声明取代了通常分配给 TInterfacedObject 的角色，如下所示：

```
TUserUnknown = class(TObject, IUnknown)
protected
        FRefCount: Integer;
        function QueryInterface(const IID: TGUID; out obj): HResult; stdcall;
        function _AddRef: Integer; stdcall;
        function _Release: Integer; stdcall;
public
        property RefCount: Integer read FRefCount;
end;
```

这个对象看起来非常类似于 TInterfacedObject，但声明并不完全一样。

_AddRef 方法是这样声明的，如下所示：

```
function TUserUnknown._AddRef: Integer;
begin
    Inc(FRefCount);
    Result := FRefCount;
end;
```

可以看到，这个方法只增加了对象上的引用计数，除此以外什么也没有做，每次调用这个类的某个继承类所支持的接口的一个实例时，Delphi 将自动调用_AddRef。换句话说，并不是接口的客户程序调用了_AddRef，而是 TUserUnknown 的继承类调用了它。

_Release 方法的实现如下所示：

```
function TUserUnknown._Release: Integer;
begin
    Dec(FRefCount);
    Result := FRefCount;
    if Result = 0 then
        Destroy;
end;
```

上面的这个方法是在将接口赋值为 nil **的时候调用的。例如运行** MyInterface := nil **的时候，就将调用** MyInterface **的**_Release()**方法。一般不需要亲自调用**_Release()**方法，**Delphi **会自己来完成它。**

TUserUnknown **实现的最后一个** IUnknown **方法是** QueryInterface**，如下所示：**

```
function TUserUnknown.QueryInterface(const IID: TGUID; out obj): HResult;
const
    E_NOINTERFACE = 1180004002;
begin
    if GetInterface(IID, obj) then
        Result := 0
    else
        Result := E_NOINTERFACE;
end;
```

其中调用了一个 GetInterface()**的** TObject **的方法，它执行了** QueryInterface()**方法中所有必要的繁琐的工作。**GetInterface()**方法返回一个小型的** VMT**，这个** VMT **中只包含用户需要的接口的方法。**

现在读者已经基本了解了如何实现 IUnknown**，那么这时候就比较容易理解在实现接口的类中所发生的事情，如下所示：**

```
IName = interface(IUnknown)
    ['{F17EF920-82CB-11D5-BF59-E3228A875871}']
    function GetName: string;
end;

INumber = interface(IUnknown)
    ['{F17EF921-82CB-11D5-BF59-E3228A875871}']
    function GetNumber: integer;
end;

TMyClass = class(TUserUnknown, IName, INumber)
private
    FOnDestroy: TNotifyEvent;
public
```

```
    constructor Create;
    destructor Destroy; override;
    function GetName: string;
    function GetNumber: Integer;
    property OnDestroy: TNotifyEvent read FOnDestroy write FOnDestroy;
end;
```

可以看到，TMyClass 继承了 TUserUnknown 的功能，因此它知道如何实现 IUnknown。实际上，TMyClass 实现了 3 个不同的接口：IUnknown、IName 和 INumber。

Delphi 允许按照用户的愿望在一个类中实现多个接口。上面的示例中，TMyClass 通过继承来自 TUserUnknown 的实现方法隐式地实现了 IUnknown 接口，同时显式地实现了 IName 和 INumber 两个接口。

最好不要在两个接口中同时声明相同名称的方法，这将会增加程序的复杂性。

11.2.7　IDispatch、双重接口和 DispInterface

IDispatch 是一个在 OLE 自动化中具有举足轻重的重要接口。这个接口最初是为 VB 用户而设立的，但 Object Pascal 和 C++程序员也可以访问它。IDispatch 对于 VB 程序是非常有用的，因为它允许程序员可以不使用对象而访问 COM 接口。但是，VB 并不是面向对象的编程语言，因此它并不具备 COM 的所有优点，不过它可以使用 Dispinterface。IDispatch 是一个将 VB 带到 COM 中的更有帮助的类，许多其他工具，例如 Word 和 Excel，也可以通过 IDispatch 来调用 COM 接口。

> 注意：IDispatch 比其他接口要慢。但穿越程序和网络界限去访问 COM 接口比访问程序或者 DLL 要更慢一些，以至于使 IDispatch 本身的开销显得不是很重要了。

在以下两种情况中，IDispatch 是十分有用的：

- 要找到一种 Quick-and-Dirty（迅速但不利索）的方法来得到一个自动化对象。
- 确保所创建的对象可被第三方开发者使用。也就是说，希望自己的代码可以从其他不支持真正对象的语言环境（例如 VB）里也可以调用。

Delphi 使用了一种叫做“双重接口”的技术，它可以使一个单独的接口支持 IDispatch 和用户自定义的接口。因此，所创建的对象可以通过 IDispatch 从 VB 或者 Word 来访问，或者可以直接从 Object Pascal 或 C++访问相同的对象。

另外一种技术称为 Dispinterface，也是在 Delphi 上自动获得的。DispInterface 介于 IDispatch 接口与真正的 Object Pascal 对象之间。IDispatch 允许查询一个对象并要求一系列可以用来访问对象的函数以及参数的数字标记。用于查询对象的方法称为 IDispatch.GetIDsOfNames。

在知道哪些标记可以使用时，就可以将它们送至对象，来访问对象的数据和方法。用于访问对象的方法调用称为 IDispatch.Invoke。

有关 IDispatch 的问题是，为访问一个方法必须对对象作两个调用：第一个调用得到要使用的标记，第二个调用用来实际访问方法。

11.2.8 HResult 类型

HResult 在 ActiveX、OLE 和 COM API 中经常用作返类型。在 System 单元中 HResult 被声明为 LongWord 类型。在 Windows 单元中可以找到 HResult 的可能值。值为 S_OK 或 NOERROR(0)表示调用成功。如果 HResult 值的高位被设置，表示调用失败。Windows 单元中的 Succeeded()和 Failed()函数需要传递 HResult 类型的参数，并且返回布尔值来表示成功还是失败。

当然，如果对每个函数都检查返回值是非常单调乏味的。同样，这种方法也破坏了 Delphi 的异常处理机制。正是这个原因，ComObj 单元中另外定义了一个叫 OleCheck()的过程，能够把 HResult 错误转换为异常，调用的语法如下所示：

```
OleCheck(MyFunction: HResult);
```

注意：如果读者是一个有经验的 COM 开发者，可能注意到_AddRef()和_Release()方法都有一个下划线，这与其他 COM 编程语言甚至与 Microsoft 的 COM 文档都不同，这是因为 Object Pascal 是“ IUnknown 感知 ”的，所以没有必要直接调用这两个方法，加下划线是为了提醒注意。

11.2.9 虚方法表

本章已经对 COM 和接口讨论了很多，但一直没有提到 VTable。VTable 在接口技术的实现中起着重要的作用，扮演着关键的角色。

VTable 是当用户用虚方法声明一个对象时所得到的东西。就是说，如果要声明一个对象，把它的所有或者一部分方法声明为虚或者虚抽象，那么其实已经创建了一个 VTable。

在前面的讨论中，读者已经知道了声明一个接口和声明一系列公共虚抽象的方法的对象很相似。实际上，接口非常类似 Object Pascal 中的虚抽象类。

可以用一个虚抽象类在 Object Pascal 中定义一个 COM 对象，编译器会产生直接访问这个类的方法的代码，就好像它是一个标准 Object Pascal 类一样，也可以在 Object Pascal 或者其他编程语言中直接调用这个 COM 对象的方法。

11.3 COM 对象和类工厂

COM 对象可以支持一个或几个接口，并且具有生存期管理功能。此外，COM 对象可以通过一种成为类工厂的特殊对象来创建。每个 COM 对象都有一个类工厂，类工厂负责创建 COM 对象的实例。类工厂是一种特殊的 COM 对象，它的特点是支持 IClassFactory 接口。在 ActiveX 单元里，IClassFactory 是这样定义的，如下所示：

```
Type
    IClassFactory ＝ interface（IUnknown）
        [‘{0000001-0000-0000-C000-000000000046’}]
        function CreateInstance(const unkOUter: IUnknown; const iid: TIID;out obj): HResult; stdcall;
        function LockServer(fLock: Bool): HResult; stdcall;
```

```
end;
```

CreateInstance 方法的作用是创建类工厂所关联的 COM 对象的实例。如果要创建的 COM 对象是聚合的一部分，这个方法的 unkOuter 参数需要引用一个接口。iid 参数用于指向一个接口的 iid。相应的，obj 参数返回这个接口的指针。

LockServer( )方法的作用是把 COM 服务器锁定在内存中，即使已经没有客户引用这个服务器。如果 fLock 参数为 True，表示把锁定计数加“1”，如果为 False，表示把锁定计数减“1”。当锁定计数减到“0”并且确实没有客户在引用 COM 服务器时，这个 COM 服务器将卸载。

11.3.1　TComObject 和 TComObjectFactory

Delphi 用 TComObject 来封装 COM 对象、用 TComObjectFactory 来封装类工厂。TComObject 包含了对 IUnknown 接口的支持。同样，TComObjectFactory 支持 IClassFactory 接口，并且能够创建 TComObject 对象的实例。

声明和实现一个 COM 对象就像声明和实现一个普通 VCL 对象一样。要使 COM 对象与一个类工厂对象关联，需要在调用 TComObjectFactory 的 Create( )时传递一个参数。第一个参数用于指定一个 TComServer 对象，一般情况下，这个参数设为一个全局变量 ComServer，它在 ComServ 单元中声明。第二个参数用于指定一个要关联的 TComObject 类。第三个参数用于指定 TComObject 类的类标识符 CLSID。第四和第五个参数用于指定在系统注册表中注册的类名和描述。

至于 COM 服务器是如何被激活的，这取决于 COM 服务器是 in-process（进程内）的还是 out-process（进程外）的。

11.3.2　进程内 COM 服务器

in-process（进程内 COM 服务器）的形式是 DLL，它可以输出 COM 对象供它的宿主程序使用。这种类型的服务器之所以被称为 in-process，是因为 DLL 与调用它的应用程序在同一个进程内。一个进程内 COM 服务器必须输出以下 4 个例程，如下所示：

```
DllGetClassObject；
DllCanUnloadNow；
DllRegisterServer；
DllUnregisterServer；
```

它们在 ComServ 单元中已经实现了，所以，只要保证把这几个例程加到 COM 服务器单元的 Exports 子句中就可以了。

要创建进程内 COM 服务器的实例，可以调用 CreateComObject()函数。这个函数在 ComObj 单元中是这样声明和实现的，如下所示：

```
function CreateComObject(const ClassID: TGUID): IUnknown;
begin
OleCheck(CoCreateInstance(ClassID, nil, CLSCTX_INPROC_SERVER or
        CLSCTX_LOCAL_SERVER, IUnknown, Result));
```

```
end;
```

ClassID 用于指定一个 COM 对象的标识符 CLSID，这个函数返回所创建的 COM 对象的接口。如果对象不能被创建，将触发一个异常。

如果要创建一个 COM 对象的多个实例，CreateComObject 就显得比较低效了，因为它在创建了 COM 对象的实例后就释放了 IClassFactory 接口的指针。应当调用 CoGetClassObject 后再调用 IClassFactory.CreateInstance()函数来创建一个 COM 对象的多个实例。

11.3.3 进程外 COM 服务器

out-process（进程外 COM 服务器）的形式是 EXE，它们可以输出 COM 对象供其他应用程序使用。所谓 out-process 就是 COM 服务器与客户程序不在同一个线程内。

EXE 形式的 COM 服务器是通过调用 Application.Initialize()函数注册的。应用程序的项目文件的第一行会调用这个函数，给应用程序传递命令行参数/regserver，Application.Initialize()将在注册表中注册 COM 类，然后应用程序将终止退出。参数/unregserver，将取消 COM 类在注册表中的注册。如果没有任何参数，应用程序注册 COM 类后继续执行程序。

从表明上看，和进程内 COM 服务器的实例一样，要创建进程外 COM 服务器的实例，也要调用 ComObj 单元中的 CreateComObject()函数。不过，它们背后的操作却不是一样的。

11.4 DCOM（分布式 COM）

DCOM（分布式 COM）是从 Windows NT 4 开始引入的。DCOM 适用于访问其他机器和网络上的 COM 对象。除了能够访问远程对象外，DCOM 还能指定哪个客户有权限创建服务器的实例、能够进行什么操作。Windows NT 4 和 Windows 98 以及 2000 都支持 DCOM，但 Windows 95 需要另外安装一个软件才能支持 DCOM。

创建远程对象需要调用 CreateRemoteComObject()函数，这个函数在 ComObj 单元中是这样声明和实现的：

```
function CreateRemoteComObject(const MachineName: WideString;
        const ClassID: TGUID): IUnknown;
const
LocalFlags = CLSCTX_LOCAL_SERVER or CLSCTX_REMOTE_SERVER or
CLSCTX_INPROC_SERVER;
        RemoteFlags = CLSCTX_REMOTE_SERVER;
var
        MQI: TMultiQI;
        ServerInfo: TCoServerInfo;
        IID_IUnknown: TGuid;
        Flags, Size: DWORD;
        LocalMachine: array [0..MAX_COMPUTERNAME_LENGTH] of char;
```

```
begin
    if 11CoCreateInstanceEx = nil then
        raise Exception.CreateRes(11SDCOMNotInstalled);
    FillChar(ServerInfo, sizeof(ServerInfo), 0);
    ServerInfo.pwszName := PWideChar(MachineName);
    IID_IUnknown := IUnknown;
    MQI.IID := 11IID_IUnknown;
    MQI.itf := nil;
    MQI.hr := 0;
    { If a MachineName is specified check to see if it the local machine.
If it isn't, do not allow LocalServers to be used. }
    if Length(MachineName) > 0 then
    begin
        Size := Sizeof(LocalMachine);    // Win95 is hypersensitive to size
        if GetComputerName(LocalMachine, Size) and
          (AnsiCompareText(LocalMachine, MachineName) = 0) then
          Flags := LocalFlags else
          Flags := RemoteFlags;
    end else
        Flags := LocalFlags;
    OleCheck(CoCreateInstanceEx(ClassID, nil, Flags, 11ServerInfo, 1, 11MQI));
    OleCheck(MQI.HR);
    Result := MQI.itf;
end;
```

其中，MachineName **参数用于指定** COM **类所在机器的名称，**ClassID **参数用于指定** COM **类的** CLSID**，这个函数返回** COM **对象的** IUnknown **接口指针。**

11.5　COM Automation

Automation **是应用程序或动态链接库将可编程对象提供给其他应用程序使用的一种手段。提供者称为** Automation **服务器，操纵** Automation **服务器的应用程序称为** Automation **控制器。**Automation **控制器采用类似于宏命令的方式对** Automation **服务器进行操作。**

Automation **的最大优势是它的语言无关性。一个** Automation **服务器可以操纵用任何一种语言编写的** Automation **服务器。下面就介绍在** Delphi **中如何创建** Automation **服务器和控制器。**

DDE **是与** Automation **相似的一项技术，但是笔者建议不要使用** DDE**，而最好使用** COM**。**DDE **是自动化最初失败的尝试，它最初的价值体现在进程间通讯。而且，**DDE **过于复杂而且充满了** Microsoft **从未合理解决的错误，它是一项再也不可能有任何发展的技术。但** COM **与之不同，**COM **是一项正在不断向前发展的技术。**

下面将看到如何创建一个简单的自动化服务器和控制端（客户程序）。

11.5.1 创建 Automation 服务器

在 Delphi 中，无论要创建 out-process 还是 in-process 服务器都是很方便的。这正是 Delphi 无与伦比的优势所在。创建一个 Automation 服务器大致分为以下几个步骤：

（1）创建一个应用程序或者动态链接库。如果是 out-process 服务器，就创建一个应用程序；如果是 in-process 服务器，就创建一个动态链接库。除此以外，out-process 和 in-process 后面的步骤都是一样的。

（2）创建一个 Automation 对象并加到应用程序或动态链接库中，Delphi 的 Automation Object Expert 将帮助用户完成这一步骤。

（3）通过 type library（类型库）编辑器在 Automation 对象中加入特性和方法，这些特性和方法将让 Automation 控制器知道。

（4）实现 Automation 对象中的方法。

下面是如何实现以上每一步的方法：

这里将创建一个简单的 out-process Automation 服务器。首先，创建一个新的项目，然后把一个 TEdit 组件放到窗体上，如图 11.3 所示，将项目保存为 ComSrv.dpr。

图 11.3 Srv.dpr 的主窗体

现在，单击 File|New 命令打开 New Items 对话框，切换到 ActiveX 选项卡，双击 Automation Object 图标，如图 11.4 所示。此时将打开 Automation 对象向导，如图 11.5 所示。

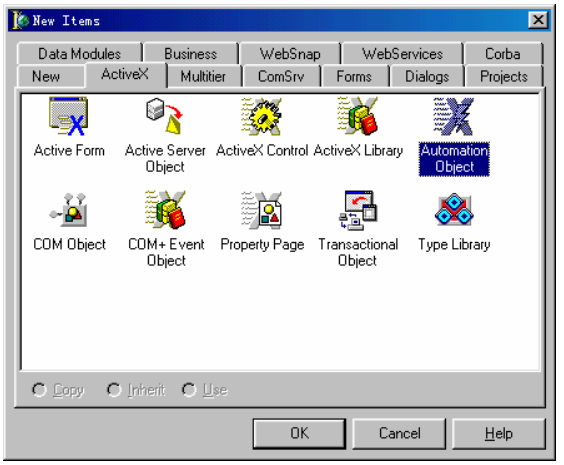

图 11.4 加入 Automation 对象

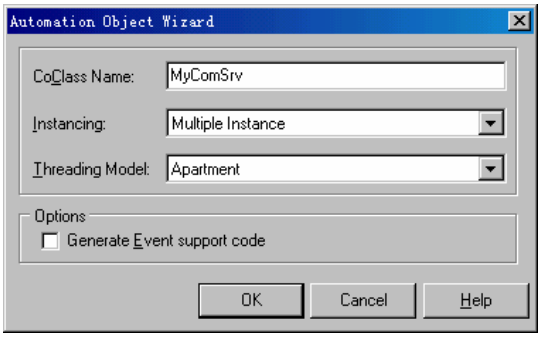

图 11.5　Automation 对象向导

在 CoClass Name 文本框内，输入这个 Automation 对象的 COM 类名，这里叫作 MyComSrv，向导会自动在类别前加大写的“T”，在相应的接口名称前加大写的“I”。Instancing 下拉列表框内可以选择以下 3 个值：

- Internal OLE　对象是应用程序内部使用的，不需要在注册表中注册。外部的进程不能访问这个内部的 Automation 对象。
- Single Instance　每个服务器的实例只能输出同一个 OLE 对象的一个实例。如果控制器程序请求 OLE 对象的另一个实例，Windows 就会启动服务器的另一个实例。
- Multiple Instance　每个服务器的实例可以创建和输出同一个 OLE 对象的多个实例。In-process Automation 服务器通常要设为这种模式。

从向导退出后，Delphi 会创建一个新的类型库（如果原来没有的话）并加入一个接口。新创建的 Automation 对象叫 TMyComSrv，它是从 TAutoObject 继承下来的。TAutoObject 是 Automation 对象的祖先类。当通过类型库编辑器在接口中加入一个方法，Delphi 将在 Automation 对象的实现单元中生成这个方法的框架，程序清单如下所示：

```
unit Unit2;

interface

uses
    ComObj, ActiveX, ComSrv_TLB, StdVcl;

type
    TMyComSrv = class(TAutoObject, IMyComSrv)
    protected
        { Protected declarations }
    end;

implementation

uses ComServ;
```

```
initialization
    TAutoObjectFactory.Create(ComServer, TMyComSrv, Class_MyComSrv,
        ciMultiInstance, tmApartment);
end.
```

创建了一个新的 Automation 对象后，下一步就是用 Type Library Editor（类型库编辑器）在它的主接口中加入一些特性或方法。对于 TMyComSrv 来说，在 IMyComSrv 中加入一个方法 MyShow，此方法传递一个整型值的参数“I”，保存项目或者刷新数据之后，Delphi 在 Automation 的实现单元中将自动加入该函数的框架，用户所要做的只是填充该函数的内容。

```
procedure TMyComSrv.MySend(i: Integer);
begin
  form1.edit1.Text := IntToStr(i);
end;
```

这里 MyShow 函数的功能仅仅是将参数“I”显示在 form1 上的 Edit1 中，所用到的数据类型转换函数 IntToStr 在 SysUtils 单元中声明，而 form1 在 unit1 中定义，所以，必须在 uses 子句中加入 SysUtils 和 unit1 两个单元。

注意：在类型库中加入特性和方法时，特性和方法的参数或返回值的类型一定要是与 Automation 相容的类型。

这时候，会发现，Automation 的实现单元 unit2 里引用了一个叫 ComSrv_TLB 的单元。这个单元是 Delphi 自动生成的 Object Pascal 外套，当然没有必要自己修改里面的代码，但是能够理解它对于理解 Automation 的实现是非常有帮助的。

在这里，首先声明了类型库的 GUID，在注册这个类型库的时候必须要有一个 GUID 来惟一标识它。接着又声明了一个 IMyComSrv 的接口，其中声明了若干个特性和方法。其中包括刚才的 MyShow 函数。

在这里要解释一下 safecall 的调用方式，正如在 IMyComSrv 接口中看到的那样，safecall 是类型库编辑器的默认调用约定方式。它不仅仅是一种调用约定方式，而且还暗示了两件事情。假设有这样一个方法：

```
function MyCall(S: Integer): Integer; safecall;
```

这个方法经过编译后，实际上是这样，如下所示：

```
function MyCall(S:Integer;out RetVal:Integer):HResult;safecall;
```

它的优势就是能够捕捉异常。当这个方法中触发的异常未被处理的时候，它就可以转换为 HResult 值返回给调用者。

将上面的项目编译成功之后，这个应用程序至少要运行一遍才能注册其中的 Automation 对象。如果希望将这个 Automation 安装到 VCL 组件面板上去，那么执行 Project|Import Type Library 命令，这时将出现导出类型库的界面，如图 11.6 所示，单击 Add

按钮后，找到刚才编译的 ComSrv.exe 或者 ComSrv.tlb 文件，选定后单击 Install 命令，确认后，该 Automation 服务器就安装到了 ActiveX(默认)组件组中去了，以后如果想在其他程序中调用该服务器，只要从 VCL 的 ActiveX 组件组中选择 MyComSrv 放到程序中就可以了。本章将在 11.5.2 小节中介绍如何操纵这个 Automation 服务器。

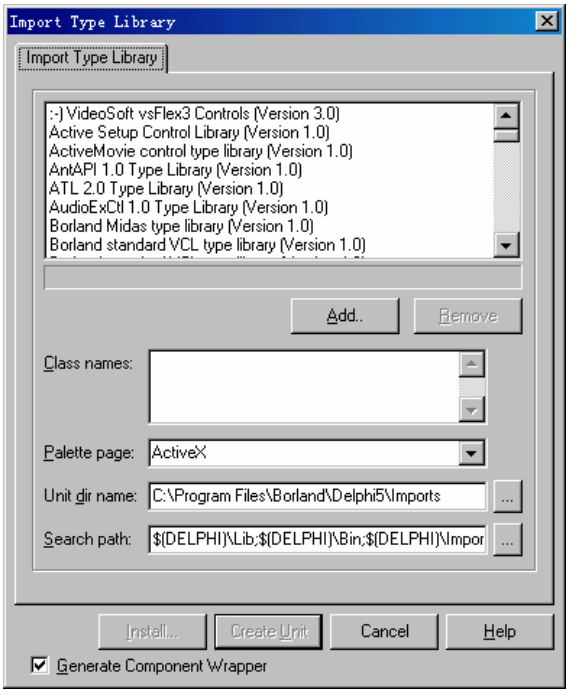

图 11.6　导入 Automation 服务器

Out-process Automation 服务器以一个可执行文件为起点，而一个 in-process 服务器则是以一个动态库为起点。可以通过 File|New 命令，然后从 New Items 中切换至 ActiveX 选项卡，选择创建一个 ActiveX Library 来得到 in-process 服务器的动态库起点。对于 in-process 服务器，它的注册方式和 out-process 服务器不同，必须调用 DllRegisterServer()函数来注册。在 Delphi 的 IDE 中，只要调用 Run|Register ActiveX Server 命令就可以注册一个 in-process 服务器，也可以利用 Windows 中的 RegSvr32.exe 程序来注册它。

11.5.2　创建 Automation 控制器

要操纵 Automation 服务器是非常容易和灵活的。共有 3 种方式来调用服务器，它们分别是 Interface（接口）、Dispatch Interface（调度接口）和 Variants。下面就介绍一个 Automation 控制器程序（即 MyControl），它分别演示这 3 种调用方式来操纵前面创建过的 MyComSrv 服务器。

MyControl 程序的主界面如图 11.7 所示，其中 3 个按钮(Interface、Dispatch Interface 和 Variants)分别代表 3 种调用方式。

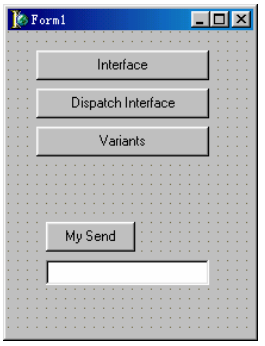

图 11.7 控制器主界面

下面是这个控制器程序的创建部分的源代码：

```
procedure TForm1.btInterfaceClick(Sender: TObject);
var
    MyComSrv: TMyComSrv;
begin
    MyComSrv := TMyComSrv.Create(nil);
    MyComSrv.Connect;
end;

procedure TForm1.btDispIntfClick(Sender: TObject);
var
    MyComSrv: IMyComSrvDisp;
begin
    MyComSrv := CreateComObject(Class_MyComSrv) as IMyComSrvDisp;
end;
procedure TForm1.btVariClick(Sender: TObject);
var
    MyComSrv: OleVariant;
begin
    MyComSrv := CreateOleObject('ComSrv.MyComSrv');
end;
```

如果用户已经把 MyComSrv 服务器装到了 VCL 中，那么就可以非常方便地将该组件拖放到窗体上。在窗体创建时，程序将按 Interface 方式自动创建该服务器，用户只要调用该服务器的 Connect 函数就可以了。

要断开 Automation 服务器，对于接口和调度接口来说，只要将它们的变量设为 nil 就可以了，对于 Variants 来说，只要把它的变量设为 unassigned 就可以了。当然，当 MyControl 程序关闭的时候，服务器的实例也会自动释放。

提示：用接口来控制 Automation 服务器比用调度接口和 Variants 来控制 Automation 服务器要好一些，建议尽可能使用接口方式。相对而言，Variants 的性

能最差，因为当 Variants 调用 Automation 服务器的方法的时候，实际上是先调用 GetIDsOfNames()方法再把方法名称转换成标识符，然后再调用 Invoke()方法来执行方法。

下面是操纵 Automation 服务器的方法。假设用户已经在窗体上放置了一个 MyComSrv:TMyComSrv 类型的服务器组件，这时候，通过按钮 MySend 触发以下的程序，如下所示：

```
procedure TForm1.btMySendClick(Sender: TObject);
begin
    MyComSrv.MySend(StrToInt(edit1.Text));
end;
```

这时候，MySend 函数将控制器上 Edit1 中的整型值传递给了 MyComSrv 服务器，服务器收到函数触发后，将执行内部的 MySend 函数，实际操作是把参数显示在了服务器窗体的 Edit1 中。程序界面如图 11.8 所示。

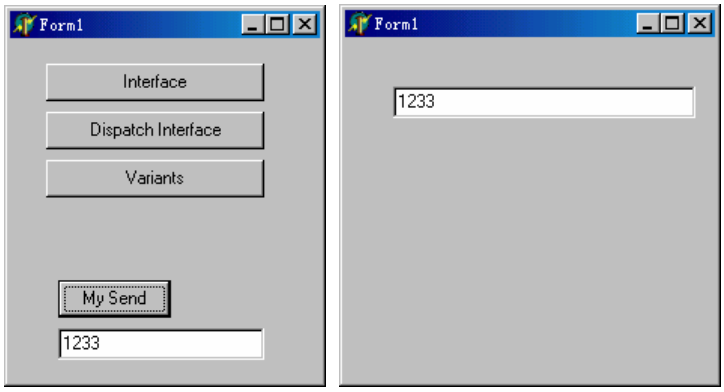

图 11.8　控制器和服务器

图 11.8 中左侧为控制器 MyControl，右侧为 Automation 服务器 MyComSrv，这时候，会发现，左侧中的参数“1233”通过 My Send 按钮传递给了右侧的服务器。

11.6　TOleContainer

现在读者已经具备了一些 ActiveX 和 OLE 的知识，下面来讨论 TOleContainer 组件。它位于 OleCntrs 单元中，是一个容器，可以容纳 OLE 文档和 ActiveX 文档。

11.6.1　一个简单的程序示例

首先创建一个新的项目，然后从组件选项板上的 System 选项卡选择 TOleContainer 放到 Form 上，双击这个组件，或者右击它，在弹出的快捷菜单中执行 Insert Object 命令，Delphi 将打开“插入对象”对话框，如图 11.9 所示。

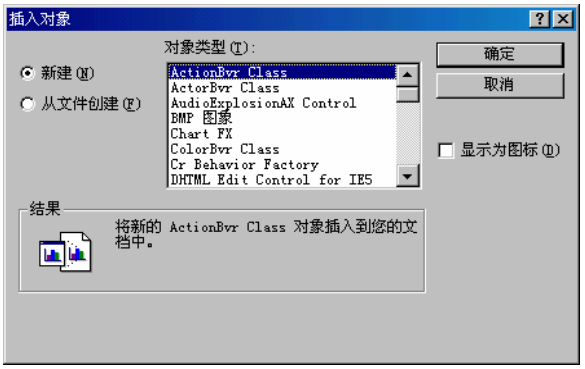

图 11.9 “插入对象”对话框

“插入对象”对话框里缺省地列出了所有在 Windows 中注册的 OLE 服务器程序。要嵌入一个新的 OLE 对象，只要从对象类型列表中选择一个 OLE 服务器程序就可以了。单击“确定”按钮，所选的 OLE 服务器将运行并创建一个新的 OLE 对象插入到 TOleContainer 中，要嵌入一个已有的 OLE 文件，可以在“插入对象”对话框中选择“从文件创建”单选按钮，这时将弹出一个对话框，如图 11.10 所示，然后选定一个已经存在的 OLE 文件，确定后，TOleContain 将创建一个同选定文件完全相同的新的 OLE 对象。如果要链接而不是嵌入一个 OLE 对象，只要在图中（见图 11.10）单击“链接”复选框就可以了，多个应用程序可以链接和编辑同一个文件。

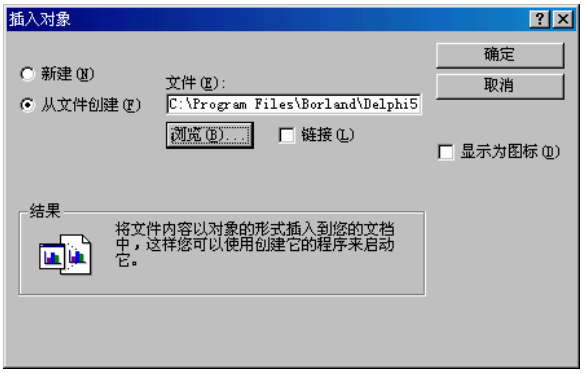

图 11.10 嵌入一个文件

上面所介绍的操作都是在设计期进行的，如果要在运行期进行同样的动作，例如打开“插入对象”对话框，嵌入或链接 OLE 服务器等操作，可以利用 TOleContainer 提供的一系列的方法和函数，如下所示：

```
function InsertObjectDialog:Boolean;
```

以下函数用来打开 Insert Object 对话框：

```
Procedure CreateObjectFromFile(const FileName: string; Iconic: Boolean);
Procedure CreateLinkToFile(const FileName: string; Iconic: Boolean);
```

以下两个方法分别实现了嵌入和链接文件的动作。

11.6.2　OLE 对象的操作方法

OLE 对象可以进行流操作，也就是说可以进行文件操作。TOleContainer 已经提供了 4 个方法：SaveToStream()、LoadFromStream()、SaveToFile()和 LoadFromFile()，这 4 个方法提供了从文件读写 OLE 对象的渠道。

Windows 的剪贴板同样可以用来传输 OLE 对象。TOleContainer 将繁琐复杂的操作细节打包处理之后，取而代之的是几个简单的函数，从而大大简化了开发操作。

例如，要把一个 OLE 对象从 TOleContainer 组件复制到剪贴板中，只要简单地调用 Copy 方法就可以了。假如剪贴板里已经有了 OLE 对象，现在需要把它从剪贴板中读到一个 TOleContainer 组件中，在此之前，应当检查 CanPaste 特性的值，看看剪贴板中的数据是否适合于 OLE 对象来粘贴。然后可以调用 PasteSpecialDialog( )方法来打开“选择性粘贴”对话框，如图 11.11 所示，该对话框显示当前的 OleContainer 中的 OLE 对象为一 Word 文档对象。

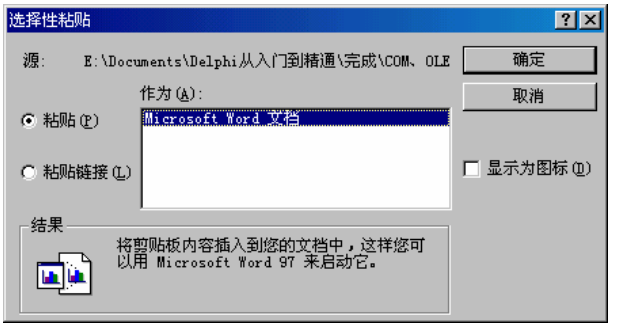

图 11.11　“选择性粘贴”对话框

11.7　本 章 小 结

这一章综合介绍了 COM、OLE 和 ActiveX。通过本章的学习，读者可以基本了解 COM 技术的奥秘，为基于 COM 的应用程序开发打下扎实的基础。更重要的是，这里为读者提供了一些第一手的资料和信息，让读者知道怎样在 Delphi 中应用 ActiveX 技术。

除了应当深入了解 COM 和 Automation 外，还应当熟悉 VCL 中的 TOleContainer 组件，它是 COM 技术很直观的体现。

可能在没有完全理解本章的所有内容的情况下，读者已经可以熟练地使用 ActiveX 甚至 DCOM 了，但是，理解这些知识将加深对所有 Delphi 中与 COM 相关的技术的理解，进而更加充分地利用它们。具有良好扎实的基础以后，才有可能创建出真正可靠健壮的程序。

第 12 章　DCOM

前面“COM 基础”一章中曾经简单地介绍过 DCOM，即分布式 COM。DCOM 适用于访问其他机器或者网络上的 COM 对象，除了能够访问远程对象以外，DCOM 还能够指定哪个客户有权限创建服务器的实例、能够进行什么操作等。

本章中将介绍一些关于 DCOM 技术以及该技术的使用等方面的知识，例如创建简单的 DCOM 程序、基于 DCOM 的分布式数据库应用程序，以及在多个客户间共享数据集等。

12.1　COM 和分布式体系结构

首先，来看看 DCOM 在实际中是如何发挥作用的。

12.1.1　DCOM 简介

当 COM 从一个机器上转移到网络上的时候，COM 的强大的生命力就显现出来了。DCOM 就是这一领域中的最稳定和最成熟的工具之一。

DCOM 技术支持应用程序在网络上相互通信。具体来说，它运行共享驻留在两个独立的机器上的对象，这意味着可以在一个应用程序中或者 DLL 中创建对象，并从驻留在另外一台机器上的应用程序中调用这个对象的共享方法和数据。这种操作并不会在客户机上消耗资源，而是加载于服务器机器的地址空间里。对于 DCOM 客户程序来说，只要有统一的接口调用方式，它不必知道 COM 的存在形式，客户程序甚至不知道 COM 对象的位置，可以在同一台计算机上，也可以在地球的另一面。

如果比较熟悉 COM，那么读者会发现学习 DCOM 其实非常容易，DCOM 遵循与 COM 相同的规则。

由于 DCOM 本身就是 COM 的一种存在形式，具有许多共同点，但同时也存在着一些区别，如下所示：

- COM 有两种存在形式，动态链接库和可执行程序。但 DCOM 必须是可执行程序，因为 DCOM 不可能在客户程序的内存空间运行，所以不能是动态链接库。
- COM（动态链接库形式）可以不用 RPC 通迅，而 DCOM 必须使用 RPC 远程调用。
- COM（动态链接库形式）与客户共同存在于同一内存空间，调用速度快，DCOM 的速度只有 COM 的万分之一。
- COM（动态链接库形式）的安全性不高，客户程序可能造成服务 COM 发生错误，DCOM 安全性高，原因也是 COM 与客户程序共用内存空间造成的。
- COM 程序配置简单，DCOM 配置较复杂。毕竟 DCOM 牵涉到网络和安全性。

12.1.2 DCOM 的系统设置

DCOM 最难的部分就是如何正确设置系统。在已经设置好系统以后，其他的事情就相对容易很多了。

可以通过以下 3 种方式来设置 DCOM：

- DCOM 服务器和客户端程序可以在两台 NT 机器上正常运行。
- 如果客户端使用 Windows 98 系统，只要服务器是在 NT 机器上，DCOM 就可以正常运行。
- 如果使用 Windows 98 作为服务器，会有一些问题，尤其是在安全性方面。

就 NT 来说，如果使用的是 NT 4，那么至少需要装载 Service Pack3。DCOM 内建于 Windows 98 和 Windows NT 中，但并没有装载到 Windows 95 中，所以如果在 Windows 95 上使用 DCOM，需要到 Microsoft 的网站上下载 DCOM95 组件。

Windows 98 的共享设置应为用户级访问，而不是默认的共享级访问，这一设置可以通过控制面板上的网络程序来进行。

用户级共享要求的 NT Domain Server 或者一些其他访问列表可以从相关的网站上获得。当用户访问服务器时，管理员赋予每个用户一定的权限，具体来说，用户可以有权运行一个或者多个 DCOM 程序。如果赋予用户在服务器上运行所有的 DCOM 程序的权力，那么实际上就允许了用户在服务器上做任何事情。

为了帮助配置服务器机器 DCOM 的安全级别，用户可以使用 DComCnfg.exe 应用程序，它是 Windows 98 和 NT 附带的。

12.2 DCOM 服务器和客户程序

在“COM 基础”一章中，读者学习了 IDispatch 接口的基础。IDispatch 是 COM 对象，它使 OLE 在许多过程中可以自动进行，因为 Delphi 支持双重接口，所以使用 IDispatch 是构造分布式程序的较好的方式。本节将介绍如何实现既适用于两个应用程序之间，又适用于运行、驻留在独立的机器上的应用程序之间的 OLE 自动化。

读者会发现在每个简单的示例里面所展示的客户/服务器程序中的服务器实际上都是一个标准的自动化服务器。这里笔者将详细地描述它的创建过程，并特别强调它的方法声明为 safecall 类型，在 Delphi 中的 DCOM 应用程序里，safecall 具有举足轻重的地位。

12.2.1 创建 DCOM 服务器

创建一个 DCOM 服务器的开始过程和创建一个简单的自动化服务器是一样的。Delphi 自动化服务器与 DCOM 服务器没有什么区别，同样是通过 File | New | Automation Object 命令来创建，如图 12.1 所示。

在 Automation Object 的设置对话框中，将服务器命名为 EasyDCOM，其他设置项保持默认，然后单击 OK 按钮确认。建议创建之后马上保存服务器，将主窗体保存为 Main-Server.pas、将项目保存为 EasyDCOMServer.dpr。另外，建议将 Unit2 保存为 MainIMPL.pas。

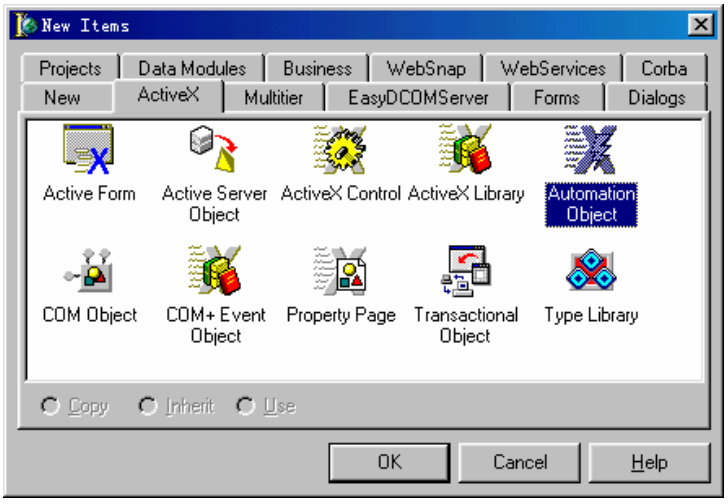

图 12.1 创建一个 Automation Object

如果这时候 Delphi 中没有出现 Type Library Editor 对话框，可以通过单击 View | Type Library 命令打开它。使用类型库编辑器创建两个方法，一个是 GetName()，另外一个是 Square()。其中，GetName()返回 WideString 类型，Square()包含两个整型值作为参数，第一个参数叫做 InX，第二个叫做 OutX 的 out 参数。

在 IEasyDCOM 下加入两个 Method，分别如图 12.2 所示命名。对于 GetName 方法，将右边部分翻到 Parameters 选项卡。在参数列表中加入一个 Value 参数，类型为 BSTR*，为指针类型；Modifier 值设为[out，retval]。注意，如果参数为 out 类型，参数类型必须为指针。对于 Square 方法，参数设置如图 12.3 所示。

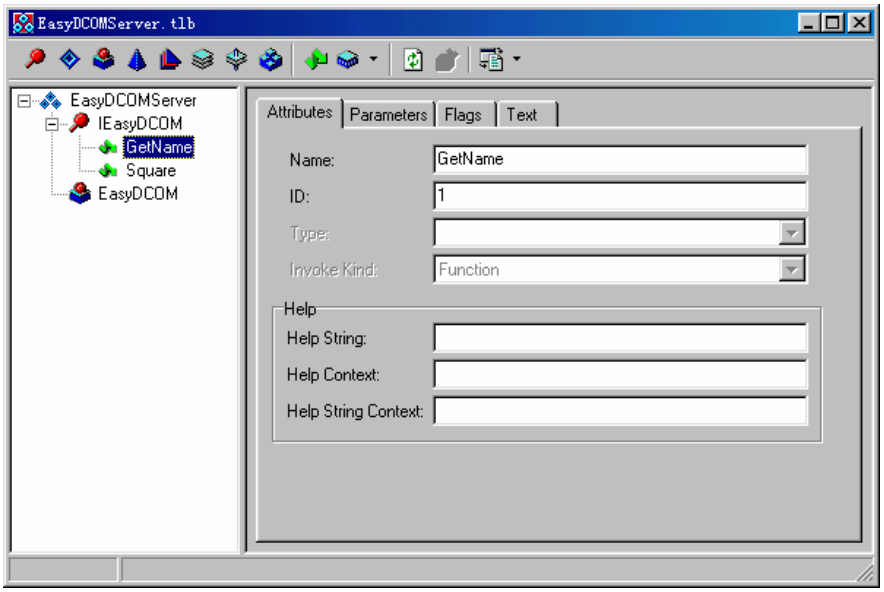

图 12.2 使用类型库编辑器创建方法

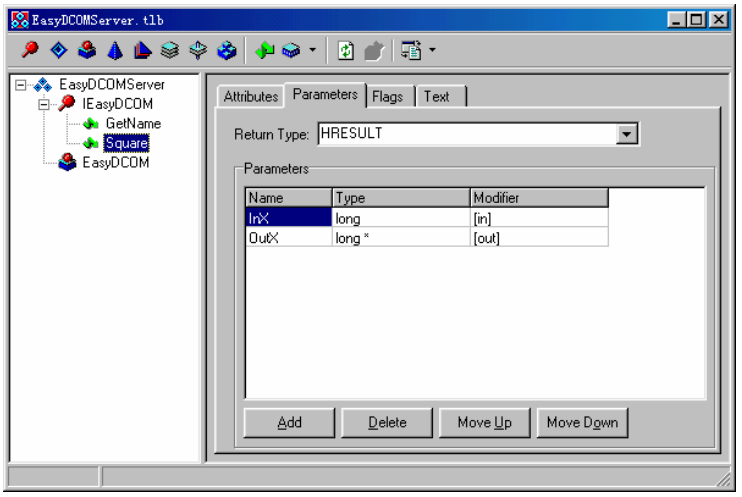

图 12.3　Square 方法的参数设置

TEasyDCOMServer **的完整的源文件中包含了** TEasyDCOMServer **的实现文件和类型库文件，如下所示：**

```
unit MainIMPL;

interface

uses
    ComObj， ActiveX， EasyDCOMServer_TLB， StdVcl;

type
    TEasyDCOM = class(TAutoObject， IEasyDCOM)
    protected
        function GetName: WideString; safecall;
        procedure Square(InX: Integer; out OutX: Integer); safecall;
        { Protected declarations }
    end;

implementation

uses ComServ;

{ TEasyDCOM }

function TEasyDCOM.GetName: WideString;
begin
    Result := 'TEasyDCOM';
end;
```

```
procedure TEasyDCOM.Square(InX: Integer; out OutX: Integer);
begin
    OutX := InX * InX;
end;

initialization
    TAutoObjectFactory.Create(ComServer， TEasyDCOM， Class_EasyDCOM，
        ciMultiInstance， tmApartment);
end.
```

下面是类型库 EasyDCOMServer_TLB **代码：**

```
unit EasyDCOMServer_TLB;

{12TYPEDADDRESS OFF} // Unit must be compiled without type-checked pointers.
interface

uses Windows， ActiveX， Classes， Graphics， OleServer， OleCtrls， StdVCL;

const
    // TypeLibrary Major and minor versions
    EasyDCOMServerMajorVersion = 1;
    EasyDCOMServerMinorVersion = 0;
    LIBID_EasyDCOMServer: TGUID = '{1742AF80-68A0-11D5-BF57-D692FD02CF71}';

    IID_IEasyDCOM: TGUID = '{1742AF81-68A0-11D5-BF57-D692FD02CF71}';
    CLASS_EasyDCOM: TGUID = '{1742AF83-68A0-11D5-BF57-D692FD02CF71}';
type

    IEasyDCOM = interface;
    IEasyDCOMDisp = dispinterface;

    EasyDCOM = IEasyDCOM;

    IEasyDCOM = interface(IDispatch)
        ['{1742AF81-68A0-11D5-BF57-D692FD02CF71}']
        function    GetName: WideString; safecall;
        procedure Square(InX: Integer; out OutX: Integer); safecall;
    end;

    IEasyDCOMDisp = dispinterface
        ['{1742AF81-68A0-11D5-BF57-D692FD02CF71}']
        function    GetName: WideString; dispid 1;
```

```
        procedure Square(InX: Integer; out OutX: Integer); dispid 2;
    end;

    CoEasyDCOM = class
        class function Create: IEasyDCOM;
        class function CreateRemote(const MachineName: string): IEasyDCOM;
    end;

{12IFDEF LIVE_SERVER_AT_DESIGN_TIME}
    TEasyDCOMProperties= class;
{12ENDIF}
    TEasyDCOM = class(TOleServer)
    private
        FIntf:          IEasyDCOM;
{12IFDEF LIVE_SERVER_AT_DESIGN_TIME}
        FProps:          TEasyDCOMProperties;
        function        GetServerProperties: TEasyDCOMProperties;
{12ENDIF}
        function        GetDefaultInterface: IEasyDCOM;
    protected
        procedure InitServerData; override;
    public
        constructor Create(AOwner: TComponent); override;
        destructor  Destroy; override;
        procedure Connect; override;
        procedure ConnectTo(svrIntf: IEasyDCOM);
        procedure Disconnect; override;
        function  GetName: WideString;
        procedure Square(InX: Integer; out OutX: Integer);
        property  DefaultInterface: IEasyDCOM read GetDefaultInterface;
    published
{12IFDEF LIVE_SERVER_AT_DESIGN_TIME}
        property Server: TEasyDCOMProperties read GetServerProperties;
{12ENDIF}
    end;

{12IFDEF LIVE_SERVER_AT_DESIGN_TIME}
TEasyDCOMProperties = class(TPersistent)
    private
        FServer:    TEasyDCOM;
        function    GetDefaultInterface: IEasyDCOM;
        constructor Create(AServer: TEasyDCOM);
    protected
```

```
        public
            property DefaultInterface: IEasyDCOM read GetDefaultInterface;
        published
        end;
{12ENDIF}

procedure Register;

implementation

uses ComObj;

class function CoEasyDCOM.Create: IEasyDCOM;
begin
        Result := CreateComObject(CLASS_EasyDCOM) as IEasyDCOM;
end;

class function CoEasyDCOM.CreateRemote(const MachineName: string): IEasyDCOM;
begin
        Result := CreateRemoteComObject(MachineName，  CLASS_EasyDCOM) as IEasyDCOM;
end;

procedure TEasyDCOM.InitServerData;
const
        CServerData: TServerData = (
            ClassID:     '{1742AF83-68A0-11D5-BF57-D692FD02CF71}';
            IntfIID:     '{1742AF81-68A0-11D5-BF57-D692FD02CF71}';
            EventIID:    '';
            LicenseKey: nil;
            Version: 500);
begin
        ServerData := @CServerData;
end;
procedure TEasyDCOM.Connect;
var
        punk: IUnknown;
begin
        if FIntf = nil then
        begin
            punk := GetServer;
            Fintf:= punk as IEasyDCOM;
        end;
end;
```

```
procedure TEasyDCOM.ConnectTo(svrIntf: IEasyDCOM);
begin
        Disconnect;
        FIntf := svrIntf;
end;
procedure TEasyDCOM.DisConnect;
begin
        if Fintf <> nil then
        begin
                FIntf := nil;
        end;
end;

function TEasyDCOM.GetDefaultInterface: IEasyDCOM;
begin
        if FIntf = nil then
                Connect;
        Assert(FIntf <> nil，'DefaultInterface is NULL. Component is not connected
       to Server. You must call "Connect" or "ConnectTo" before this
       operation');
        Result := FIntf;
end;

constructor TEasyDCOM.Create(AOwner: TComponent);
begin
        inherited Create(AOwner);
{12IFDEF LIVE_SERVER_AT_DESIGN_TIME}
        FProps := TEasyDCOMProperties.Create(Self);
{12ENDIF}
end;

destructor TEasyDCOM.Destroy;
begin
{12IFDEF LIVE_SERVER_AT_DESIGN_TIME}
        FProps.Free;
{12ENDIF}
        inherited Destroy;
end;

{12IFDEF LIVE_SERVER_AT_DESIGN_TIME}
function TEasyDCOM.GetServerProperties: TEasyDCOMProperties;
begin
```

```
        Result := FProps;
end;
{12ENDIF}

function    TEasyDCOM.GetName: WideString;
begin
        Result := DefaultInterface.GetName;
end;

procedure TEasyDCOM.Square(InX: Integer; out OutX: Integer);
begin
        DefaultInterface.Square(InX， OutX);
end;

{12IFDEF LIVE_SERVER_AT_DESIGN_TIME}
constructor TEasyDCOMProperties.Create(AServer: TEasyDCOM);
begin
        inherited Create;
        FServer := AServer;
end;

function TEasyDCOMProperties.GetDefaultInterface: IEasyDCOM;
begin
        Result := FServer.DefaultInterface;
end;
{12ENDIF}

procedure Register;
begin
        RegisterComponents('Servers'，[TEasyDCOM]);
end;

end.
```

12.2.2 理解 Safecall

单击 Type Library Editor **中最上部的工具栏中最右边的按钮，查看类型库的** IDL **文件——**MainDCOMServer.idl**。下面是这个** IDL **文件中的两个例程的** IDL**：**

```
interface IEasyDCOM: IDispatch
{
        [id(0x00000001)]
        HRESULT _stdcall GetName([out， retval] BSTR * value );
        [id(0x00000002)]
```

```
        HRESULT _stdcall Square([in] long InX， [out] long * OutX );
};
```

在 Type Library Editor **中单击刷新按钮，或者保存全部项目以后，在** MainIMPL.pas **文件中将出现下面的** TEasyDCOM **类的声明，如下所示：**

```
TEasyDCOM = class(TAutoObject， IEasyDCOM)
protected
        function GetName: WideString; safecall;
        procedure Square(InX: Integer， out OutX); safecall;
        { Protected declarations }
end;
```

其中包含了在 Type Libray Editor **中创建的两个方法的声明。可以看到，它们都声明** safecall**，这意味着它们都会自动包装在一个** Try…Except **程序块中，而且每一个都暗中返回一个** HResult **值。如下所示：**

```
function TEasyDCOM.GetName: WideString;
begin
        Result := 'TEasyDCOM';
end;
```

这段程序将返回字符串 TEasyDCOM**。假如** GetName **方法没有声明** safecall**，程序代码如下所示：**

```
function TEasyDCOM.GetName(out Name: WideString): HResult;
begin
        try
                Result := S_OK;
                Name := 'TEasyDCOM';
        except
        Result := E_UNEXPECTED;
end;
```

这段代码将 GetName **方法的整个功能都包装在一个** Try…Except **程序块中，如果出现错误的话，异常将会被截获，并且作为** HResult **值返回。这个方法是必要的，因为在一个远程服务器上出现异常是非常严重的事情。假如，某用户正在北京使用一个** DCOM **应用程序的客户程序，而该** DCOM **应用程序的服务器是在上海运行，如果在客户端的一个操作导致了服务器端的一个异常，这时候，北京和上海的应用程序部分同时出现了一个异常的对话框，那么为了让程序继续运行，这个用户不得不从北京飞到上海去，在服务器端的异常对话框上单击** OK **按钮，然后再飞回北京，继续运行程序。可以想象这将是非常糟糕的事情。理想情况是，**Delphi **自动将方法包装在一个** Try…Except **程序块中，并抑制异常情况，**Delphi **随后会自动返回给客户端一个异常情况，客户端处理完该异常以后，程序就可以继续运行，而服务器端也没有因为这个异常而停止。**

12.2.3 创建 DCOM 客户程序

这一节将创建一个名称为 TEasyDCOMClient 的客户程序，用来调用上面创建的 TEasyDCOMServer 服务器中的 GetName()和 Square()方法。这个客户程序很类似于标准的 Delphi 自动化程序，但它是用于远程而不是本地检索对象。

这个程序并不是非常复杂，它的运行界面如图 12.4 所示。

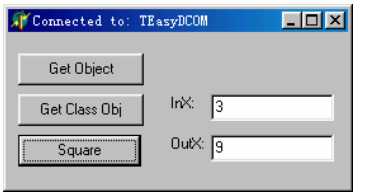

图 12.4 TEasyDCOM Client 界面

程序中使用了 3 种办法来启动远程 DCOM 服务器。一种是 VCL 远程访问对象，即图中的按钮 Get Object；一种是使用 Windows API 函数远程访问对象，按钮 Get Class Obj 就是使用这种方法；第三种是按钮 Square 调用远程服务器上的 Square 方法。

客户程序使用了来自服务器端的 TEasyDCOMServer_TLB 文件，需要把这个文件放置在本地 Delphi 能够找到的地方，并写进客户程序的 uses 部分。TEsayDCOMClient 的源程序如下所示：

```
unit Client_Main;

interface

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

type
    Tfm_Main = class(TForm)
        btnGetObject: TButton;
        btnGetClassObj: TButton;
        Edit1: TEdit;
        Edit2: TEdit;
        btnSquare: TButton;
        Label1: TLabel;
        Label2: TLabel;
        procedure btnGetObjectClick(Sender: TObject);
        procedure btnGetClassObjClick(Sender: TObject);
        procedure btnSquareClick(Sender: TObject);
    private
        { Private declarations }
    public
        { Public declarations }
    end;
```

```
var
    fm_Main: Tfm_Main;
    FEasyDCOM: IEasyDCOM;

implementation

uses
    ActiveX， ComObj;

{12R *.DFM}

// VCL 远程访问对象方法
procedure Tfm_Main.btnGetObjectClick(Sender: TObject);
var
    S: string;
    ServerName: string;
begin
    if (InputQuery('Enter Server Name'， 'Server Name'， ServerName)) then begin
        FEasyDCOM := CoEasyDCOM.CreateRemote(WideString(ServerName));
        S := FEasyDCOM.GetName;
        Caption := 'Connected to: ' + S;
    end;
end;

// Win32 API 函数远程访问对象方法
procedure Tfm_Main.btnGetClassObjClick(Sender: TObject);
var
    hr: HResult;
    ClassFactory: IClassFactory;
    CoServerInfo: TCoServerInfo;
    ServerName: string;
    AServerName: WideString;
begin
    ServerName := '';
    if (InputQuery('Enter Server Name'， 'Server Name'， ServerName)) then begin
        ClassFactory := nil;
        FillChar(CoServerInfo， sizeof(CoServerInfo)， 0);
        AServerName := ServerName;
        CoServerInfo.pwszName := pWideChar(AServerName);

        hr := CoGetClassObject(CLASS_EasyDCOM， CLSCTX_REMOTE_SERVER，
          @CoServerInfo， IClassFactory， ClassFactory);
        OleCheck(hr);
```

```
            if ClassFactory = nil then
               ShowMessage('no ClassFactory');

            hr := ClassFactory.CreateInstance(nil， IID_IEasyDCOM， FEasyDCOM);
            if Succeeded(hr) then begin
               ClassFactory := nil;
               Caption := 'Connected to: ' + FEasyDCOM.GetName;
            end else
               ShowMessage('No Object');
      end;
end;

// 调用远程 DCOM 服务器上的 Square()方法
procedure Tfm_Main.btnSquareClick(Sender: TObject);
var
      value， Num: Integer;
begin
      try
            Num := StrToInt(Edit1.Text);
            if FEasyDCOM <> nil then begin
               FEasyDCOM.Square(Num， Value);
               Edit2.Text := IntToStr(Value);
            end;
      except
            on E: Exception do
               ShowMessage(E.Message);
      end;
end;

end.
```

这个示例中没有给出本地调用服务器的方法，其实如果想在本地创建一个 COM 对象的实例，只需要从 TEasyDCOMServer_TLB 文件中调用预先声明的 CoEsayDCOM 的构造函数即可，如下所示：

```
procedure GetLocalObject;
var
      S: WideString;
begin
      FEasyDCOM := CoEasyDCOM.Create();
      S := FEasyDCOM.GetName;
      Caption := 'Connected to: ' + s;
end;
```

可以把这个函数加进上面的源程序中，进行 DCOM 的本地调用。

程序运行以后，单击 Get Object 按钮，然后在显示出来的对话框中输入 Server Name，即远程的计算机名，确定以后，服务器程序将被启动，会看到标题栏上显示出：“ Connected to：TEasyDCOM ”的信息，表示客户端已经连接到服务器上了。这时候，在 InX 文本框中输入一个整型值，例如“ 3 ”；单击 Square 按钮后，在 OutX 文本框中将显示“ 9 ”，即调用 Square()函数成功。

12.2.4 深入 DCOM

仍以 12.2.3 中的示例来深入到 DCOM 内部，看看 DCOM 内部是如何实现远程调用的。第一种调用方法使用 VCL 技术用来创建远程对象，如下所示：

```
procedure Tfm_Main.btnGetObjectClick(Sender: TObject);
var
    S: string;
    ServerName: string;
begin
    if (InputQuery('Enter Server Name', 'Server Name', ServerName)) then begin
        FEasyDCOM := CoEasyDCOM.CreateRemote(WideString(ServerName));
        S := FEasyDCOM.GetName;
        Caption := 'Connected to: ' + S;
    end;
end;
```

这里调用了来自 TEasyDCOMServer_TLB 文件以前建立的代理对象的一个类方法。这些类使用起来非常容易，但应该继续进行到最后一步来看看它们是如何工作的。CreateRemote()函数初始化了一个称为 FEasyDCOM 的 fm_Client 字段，它属于 TEasyDCOM 类型。下面看一看 CreateRemote()函数是如何工作的。

当调用 CreateRemote()函数时，系统将映射到 ComObj.pas 文件中的 CreateRemote-ComObject()函数，如下所示：

```
function CreateRemoteComObject(const MachineName: WideString;
    const ClassID: TGUID): IUnknown;
const
    LocalFlags = CLSCTX_LOCAL_SERVER or CLSCTX_REMOTE_SERVER or
CLSCTX_INPROC_SERVER;
    RemoteFlags = CLSCTX_REMOTE_SERVER;
var
    MQI: TMultiQI;
    ServerInfo: TCoServerInfo;
    IID_IUnknown: TGuid;
    Flags, Size: DWORD;
    LocalMachine: array [0..MAX_COMPUTERNAME_LENGTH] of char;
begin
```

```
        if @CoCreateInstanceEx = nil then
            raise Exception.CreateRes(@SDCOMNotInstalled);
        FillChar(ServerInfo， sizeof(ServerInfo), 0);
        ServerInfo.pwszName := PWideChar(MachineName);
        IID_IUnknown := IUnknown;
        MQI.IID := @IID_IUnknown;
        MQI.itf := nil;
        MQI.hr := 0;
        { If a MachineName is specified check to see if it the local machine.
            If it isn't， do not allow LocalServers to be used. }
        if Length(MachineName) > 0 then
        begin
            Size := Sizeof(LocalMachine);    // Win95 is hypersensitive to size
            if GetComputerName(LocalMachine, Size) and
                (AnsiCompareText(LocalMachine, MachineName) = 0) then
              Flags := LocalFlags else
              Flags := RemoteFlags;
        end else
            Flags := LocalFlags;
        OleCheck(CoCreateInstanceEx(ClassID, nil, Flags, @ServerInfo, 1, @MQI));
        OleCheck(MQI.HR);
        Result := MQI.itf;
end;
```

这个函数是通过一个 Windows API **函数** CoCreateInstanceEx()**来创建远程对象的，这个函数的定义如下所示：**

```
        CoCreateInstanceEx: TCoCreateInstanceExProc = nil;
        TCoCreateInstanceExProc = function (
const clsid: TCLSID;
            unkOuter: IUnknown;
dwClsCtx: Longint;
ServerInfo: PCoServerInfo;
            dwCount: Longint;
rgmqResults: PMultiQIArray): HResult stdcall;
```

这个函数包含以下 5 **个参数：**

- CLSID**：这是创建的** CoClass **的** LCSID**；**
- UnkOuter**：总是为** nil**，只能在聚合时使用。**
- DwClsCtx**：这是一个常数，它指定了要访问的服务器的种类。**
- PCoServerInfo**：参数定义如下所示：**

```
  PCoServerInfo = ^TCoServerInfo;
```

```
_COSERVERINFO = record
        dwReserved1: Longint;
        pwszName: LPWSTR;
        pAuthInfo: Pointer;
        dwReserved2: Longint;
end;
```

这个记录的第一个字段是一个版本的检查字段，包含 TCoServerInfo 记录的大小；第二个字段包含一个 Unicode 字符串，它具有服务器名称或者嵌于其中的 IP 地址。pAuthInfo 字段用于安全性，这里不会涉及它。

- 最后一个参数：包含了一个 MULTI_QI 结构的数组，如下所示。

```
PMultiQI = ^TMultiQI;
tagMULTI_QI = record
        IID: PIID;
        Itf: IUnknown;
        hr: HRESULT;
end;
TMultiQI = tagMULTI_QI;
MULTI_QI = TMultiQI;
```

这个参数提供了一个位置用来声明想要检索接口的 GUID 和声明当检索时包含接口的指针。如果想要在一个对 CoCreateInstanceEx()函数的调用中检索多个接口，只需要扫描一个填充了多个 MULTI_QI 结构的数组，这是非常有效的，因为在客户程序与服务器之间的每次传输都消耗时间，因此检索多个接口将在多层应用程序中进行明显的优化。

假使不理解 CoCreateInstanceEx()函数，也可以创建一个远程服务器，但如果弄懂了它，那么一切会简单得多。

12.3　本 章 小 结

这一章主要介绍了如何使用 Delphi 来建立具有 DCOM 优点的应用程序。读者已经看到了 Delphi、DCOM 和 OLE 自动化的结合可提供一个简单的方法来允许一个应用程序控制或者使用驻留于其他机器的另一个应用程序。

DCOM 使用起来很容易，在多个机器之间分布一个特定应用程序的工作也是非常容易的。

第 13 章　分布式编程

今天，随着计算机的软硬件、网络技术的发展，企业的需求也在不断地增长与变化。多层架构的分布式应用正在作为一种解决方案逐渐流行起来。MIDAS 正是作为一种构造这种多层架构分布式应用系统而被推出的快速开发工具。

实际上，MIDAS 只是在 Delphi 6.0 之前版本中应用的一个名词，而在 Delphi 6.0 中，取而代之的是 DataSnap 组件组。DataSnap 不但强化了 MIDAS 原有的功能，而且还加入了许多新的组件，例如 TConnectionBroker、SharedConnection 和 LocalConnection 等，这些组件能够帮助程序员开发出功能更加强大的 Delphi 应用程序。本章将详细介绍这一组组件。另外，在功能上，DataSnap 改善了 MIDAS 的执行效率，使开发出的应用程序执行起来更加快速；在 XML 方面，DataSnap 提供了以前 MIDAS 所没有的功能，就是直接以 XML 的形式表示 MIDAS 的 Data Packet。

虽然 DataSnap 比 MIDAS 功能更加强大，但是它的基本技术仍然是建立在多层计算的基础上的，因此，在本章中，仍然以 MIDAS 技术为主体来讨论分布式计算。这里，读者将学习到多层数据库的计算，它使应用程序划分成几个部分，使繁烦琐复杂的数据库工具脱离客户机，为客户提供尽可能简单的安装配置。它同时也允许集中业务规划与处理，在网络间分配负载。

Delphi 一直鼓励开发者把数据访问和数据操纵分开，以实现 3 层，甚至更多层的逻辑结构。本章讨论的主题是基于现实世界的开发工作，帮助用户成功地设计和开发多层分布式应用程序。

13.1　MIDAS 多层应用

13.1.1　MIDAS 的概念

MIDAS 是 Multi-tier Distributed Application Services Suite（多层分布式应用服务包）的缩写，这也正是 MIDAS 的本质含义，有的人故意把它理解为 Multi－tier Made Easy ，因为它真正简化了多层的开发工作。

MIDAS 是由 Borland 公司开发的在 Windows 平台之上利用 DCOM、TCP/IP、OLE Enterprise 或 CORBA 等技术构造分布式应用程序的一个工具，或者说是一组工具。其实它是被 Borland 的 Delphi、C++Builder、J++Builder（MIDASII）所支持的一组高级组件，这些组件的服务和核心技术都使用 MIDAS。MIDAS 是专门为提高和更容易构造多层、瘦客户的分布式数据库应用系统而设计的。利用它，使用极少的代码，就能更快速地开发出一套具有良好的可扩展性的应用程序。

MIDAS 支持基于数据库商业逻辑的数据库约束条件的自动传播，这样在客户端就减少了操作错误的可能，并且增强了网络效率。基于数据库服务器的数据存取不用在客户端安装 BDE，从而减少了客户应用程序的维护费用。另外，MIDAS 提供的“公文包模式”的工作方式，使得数据库的流动应用更加完美。

通过 Delphi 所提供的强大的 MIDAS 数据感应组件，可以在不做任何请求的情况下使服务器和客户端保持良好的数据一致性。

MIDAS 典型的 3 层结构包括：

- 某台机器上的一个数据库服务器
- 在另外一台机器上的应用程序服务器
- 在第三台机器上的瘦客户

数据库服务器可以是诸如 SQL Server、Oracle、Sybase 或 Interbase 服务器之类的工具。应用程序服务器和瘦客户将在 Delphi 中建立。应用程序服务器包含商业规则和用来操作数据的工具。客户程序只需要将数据送至用户，以供查看和编辑。

13.1.2　MIDAS 的核心技术

下面的文字将比较深入地介绍 MIDAS 技术的核心。如果读者并不是非常熟悉 MIDAS 技术在 Delphi 中的应用，那么完全可以跳过这一节，进入后面关于 DataSnap 组件的介绍和 MIDAS 应用程序开发的章节。从那些示例里，可以比较快地熟悉 DataSnap 组件的使用，进而了解 MIDAS 技术的应用。之后，读者就可以回到本节中，阅读下面的文字。这也正是 Delphi 的一个优越之处，即用户可以在不是很了解技术细节的情况下，很快地开发出高效率的应用程序来。

使用 MIDAS 可以建立“瘦客户端+应用程序服务器（支持商业规则，自动约束条件更新）+数据库服务器系统”的多层分布应用程序，这些应用系统，可以拥有很多优良的特性，如下所示：

- 跨平台，跨产品的分布式应用程序(通过支持 CORBA 标准的 VisiBroker)
- 为高效可用的服务器提供失败恢复(fail-over)的安全措施
- 公文包模式(非连接)的数据操作
- 智能缓冲更新
- 负载平衡
- 分布式数据库和事务处理
- 跨平台的瘦客户端应用程序或 Applets
- 自动的数据库约束条件的传播
- 高速数据库连接
- 降低网络流量
- 高级冲突的解决方案
- 数据库连接 Pooling 或资源管理

提示：VisiBroker 是 Borland 公司用来开发分布式应用程序的 ORB 技术。VisiBro-

ker 实现了由 OMG 所发展的 CORBA2.0 和 IIOP 标准，并且可以在分布式对象运算环境中与其他 CORBA 兼容的 ORB 一起运作。ORB 把客户端应用程序和其所要使用的对象连在一起，ORB 会处理找到对象内容，传送返回结果。

上面所介绍的这些特性其实都是 MIDAS 通过它提供的 3 个 Broker（代理）作为核心技术来实现的，它们是：

- Remote DataBroker
- Business ObjectBroker
- ConstraintBroker

图 13.1 是这 3 个代理在 MIDAS 体系结构中的分布。

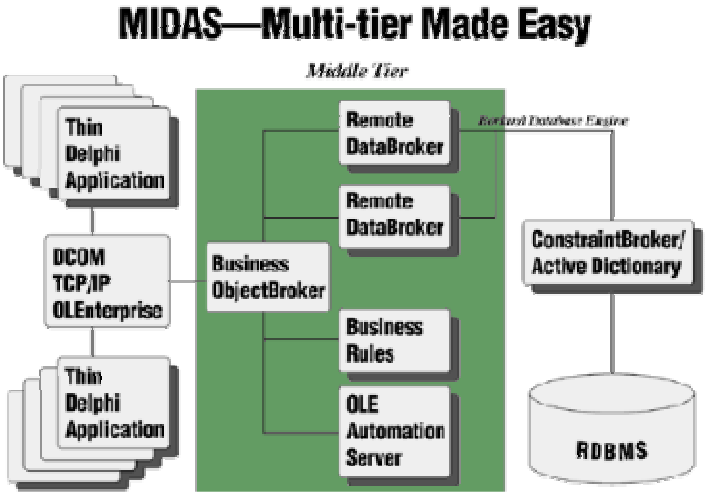

图 13.1 MIDAS 核心技术体系结构

1．Remote DataBroker

Remote DataBroker 为 MIDAS 的开发者提供了创建分布式数据库应用程序的能力，这些应用程序一般具有以下特点：

- 应用程序以多层建造：客户层，中间层和数据库层
- 中间层可能包括多个物理机器和多个不同应用程序服务器
- 瘦客户应用程序

其中，瘦客户应用程序有以下两层意思：

- 表示客户端机器上技术量很小。在 Remote DataBroker 的情况下，客户端几乎不需要手动安装或配置，因为 Delphi 的 Web 分发能力能够使客户端自动进行配置。
- 表示客户端的应用程序没有商业规则嵌入。简单的说就是，对数据而言客户端仅仅是一个图形用户界面，仅有的客户端代码是用来表现效果的。

利用 MIDAS，开发者能建立支持这些意义上的瘦客户应用程序，所有实际操作以及操

作规则都被编码到中间层应用程序中去了。瘦客户应用程序有很多优点：客户端仅需很少的安装、配置和维护，这对广大的普通用户来说具有很大的意义。需要安装、配置和维护的部分仅仅是数据库客户端连接软件。Remote DataBroker 技术为客户端减少了这一步而将这一负担留给了应用程序服务器。

2．Remote DataBroker

Remote DataBroker 技术可被分割成 3 个明显的部分：客户端、数据传输协议和服务端。Remote Data Broker 结构的精髓是让每一个客户端不再需要 BDE，取而代之的是一个中央化的 BDE，以集中管理的方式降低每一个客户在 BDE 上所需的开销和复杂度。

它通常的基本形式为：

- 客户端对服务端提出（DCOM，Socket，OLEnterprise，CORBA）请求。
- 服务端通过收集数据和将数据打包到 Data Packet 处理请求。Data Packet 是从服务端传输数据到客户端的协议。
- 一旦 Remote Broker 收集到数据就将其返回给客户端。
- 当客户端收到 Data packet，通过 ClientDataset 它就变成可用的了。

客户端的 Remote Broker 由一个 TClientDataset 组件和一个 dbClient.dll 动态库组成。中间层应用服务器提供了连接到各种不同数据库的能力，例如 Oracle、Interbase、Microsoft SQL Server、IBM DB2、Sybase 、Informix 等，以及基于文件的数据库，例如 Patadox、Microsoft Access、Borland DBase 和 Foxpro 以及任何 ODBC 数据源。

3．Business ObjectBroker

BusinessObjectBroker 的目的是给一些关键性的商业应用程序提供一个快速且可信赖的使用环境。为了满足这种高层次的要求，BusinessObjectBroker 会自动地将应用程序作适当的划分，并复制重要的业务规则到每一个区间，以达到速度的要求。

多层环境的优点之一就是多个服务器分布处理的能力。利用标准的 DCOM，当客户端试图连接到应用服务器时，它必须知道服务器的名字，这就产生了两个问题：

- 当一个机器运行不正确时会发生什么？
- 因为别的用户的请求，此机器过载怎么办？

MIDAS 技术已经解决了此问题。解决问题的办法就是 Business Object Broker 和 OLEnterprise。

Business ObjectBroker 和 OLEnterprise 能对任何 OLE 自动服务器（COM、DCOM 等）或 DCE/RPC 服务器提供负载平衡和失败恢复的功能。

提供此技术的主要组件是 TSimpleObjectBroker。TSimpleObjectBroker 提供了一个应用服务器的全局注册。客户应用程序在这里能够找到网络上注册的可以连接的应用程序服务器。TSimpleObjectBroker 将返回给提出请求的客户一个可用的应用程序服务器。

Business ObjectBroker 提供给所有自动化服务开发者动态负载平衡和失败恢复的功能。这些特性确保持续高的性能和高的可用性。其结果是终端用户应用程序有了更快和高的运行效率。

4．ConstrainBroker

ConstrainBroker 所扮演的角色是保证所有客户数据的一致性及数据的完整性。

好的数据库的设计原则之一是数据完整性规则应该集中在数据库上。通过在数据库集中设置有效性规则，确保无论客户更新了数据库的什么，数据永远是正确有效的。但是当这样确保数据库的完整性时，如果没有提供彻底的解决方案，例如一个客户应用程序对数据库进行了无效的数据更新，那么将会发生以下的情况，如下所示：

（1）客户端将数据存储到数据库服务器。
（2）数据库服务器开始一个事务处理。
（3）然后他试图进行更新。
（4）更新因为无效的数据而失败。
（5）事务处理回滚。
（6）数据库服务器给客户端应用程序发回一个错误信息。
（7）要求终端用户修改数据。
（8）整个处理重新开始。

这样就使得网络变忙并且客户端用户不得不等待响应，从而降低了用户的工作效率。通常用以降低这些障碍的技术是开发者将数据的有效性验证加到客户应用程序中去。这样做只是利用网络发送那些很可能是正确的数据，此外客户端用户将会对无效的数据有一个更快的反映以提高其工作效率，但这样同样有缺点：

- 开发者在实现这些规则时必须付双倍的代价。
- 大批数据库作更改时，所有的应用程序包括在客户端的这些规则必须更新，重新分发、重新安装和重新配置。

一个理想的解决方案就是 ConstrainBroker 技术，这项技术允许开发者在集中的地方去维护数据的完整性规则，从而减少网络的流量。

ConstrainBroker 为开发者提供了一个简单自动地分发数据完整性规则的方法，动态传播规则的特性使得开发者能够很容易地维护大量的应用程序。例如，当数据库的一个规则改变时，开发者可简单地更新数据字典。数据字典被更新后，Remote Broker 进行简单的功能调用来进行内部更新。当客户对数据提出请求时，新规则在不需要重新编译、分发、安装、配置的情况下便被复制到了客户端。

对于 MIDAS 的核心技术体系结构，可以做如下的总结：

- Business ObjectBroker 在没有任何附加代码的环境下，提供负载平衡和失败恢复的功能。
- 为把数据分布到客户端而且保持很好的可用性，Remote DataBroker 使建立中间层业务对象变得很容易。
- ConstraintBroker 提供简单的机制来增强在客户端的基础规则，以减少网络交通问题，它通过集中客户应用端数据一体化规则而减少了维修的费用。

13.1.3　简单理解 MIDAS

从一个稍微不同的角度来看，MIDAS 是一项将数据集从服务器上的 TTable 或 TQuery 对象转移到客户机上的 TClientDataSet 对象中的技术。在客户端，TClientDataSet 看起来、操作起来都和 TTable 或 TQuery 一样，但它并不需要 BDE 或其他数据驱动的支持，这样就大大地简化了客户端的配置，这也就是瘦客户。

MIDAS 允许在客户应用程序里使用所有的标准 Delphi 组件，包括数据库工具，但是客户机没有必要包含 Borland Database Engine、ODBC 或任何客户数据库。在网络上的某些地方，需要 BDE 或一个相似的引擎存在，但在客户机上并不需要。简而言之，现在需要的只是一套服务器数据库工具，在客户端需要一个很小的文件 dbclient.dll，但比 BDE 所需的或其他数据库软件所需的文件要少得多，而且小得多。

MIDAS 技术包含以下两个层次：

- 在 Component Palette 中的组件，内建于 VCL 中。
- 在 Internet 上传送数据的协议。它们可以是 DCOM、OLEnterprise、CORBA 或者 TCP/IP 中的一种。

表 13.1 是客户端与服务器连接时所用的 4 种组件对应的通讯协议。

表 13.2 是 3 种远程数据模块各自所需的通讯协议。

表 13.1　Connection 组件的通讯协议

Connection 组件	协议
TDCOMConnection	DCOM
TSocketConnection	Windows Sockets(TCP/IP)
TOLEnterpriseConnection	OLEnterprise(RPCs)
TCORBAConnection	CORBA(IIOP)

表 13.2　远程数据模块的通讯协议

远程数据模块	连接客户端的协议
TRemoteDataModle	DCOM、Sockets、OLEnterprise
TMTSDataModle	通过 DCOM、Sockets、OLEnterprise 使用 MTS
TCORBADataModule	CORBA

13.1.4　MTS、COM/DCOM、CORBA、OLEnterprise

1．MTS

MTS 是由 Microsoft 提供的一个中间件，它运行并管理业务层组件，负责最底层的工作，例如接收、连接、安全、同步、线程调度、内容和队列等。MTS 是利用 DCOM 在网络上进行通讯的，同时并为 DCOM 提供事务处理的功能。

就安全性方面，MTS 为应用程序提供了 Role-base Security（以角色为基础的安全性），它决定了是否能存取远端数据模块的界面，MTS 数据模块实现了一个 IsCallerInRole 方法，

它让用户检查当前连接客户的角色，并根据该角色有条件地允许某些功能和禁止某些功能。

MTS 提供了数据库的 Pooling（处理聚集），MTS 数据模块自动集中数据库连接，使得当一客户结束数据库连接时，另一客户端可以再次利用此连接，这样就减少了网络的流量，因为中间层不需要在登录前退出远端数据库服务器。

2．COM/DCOM

有关 COM 和 DCOM 的知识已经在第 11 章“COM 基础”和第 12 章“DCOM”两章中进行了详细地讲解，所以这里只简单地介绍一下。

COM 是由 Microsoft 设计的客户/服务器对象的模式。它能使软件组件与应用程序之间进行通讯，这个技术也称为 ActiveX，这一般认为是通过 OLE 和 OCX 实现的。它可以使开发人员利用其中的通讯机制组装不同开发商提供的构件，核心是一组应用程序调用接口。

常常有人弄不太清楚 ActiveX、COM、DCOM 之间的关系，其实 ActiveX 是 MicrosoftOLE 技术的扩展，它已经成为了一个行业标准，是部件化程序设计的技术基础。而 COM 是 ActiveX 之间交互数据的接口。DCOM 是 ActiveX 在网络计算机之间交互数据的接口。使用 DCOM，可实现部件和部件之间、应用程序和部件之间在网络位置上的透明，即应用程序无需知道每个部件在网络中的位置，就能调用它们。

3．CORBA

CORBA(Common Object Request Broker Architecture，通用对象请求代理结构)是由 OMG 提出的开发分布式应用程序的另一个标准。CORBA 提供了一个以面向对象方式来编写分布式应用程序的方式。

4．OLEnterprise

OLEnterprise 是 DCOM 的替代物，如果使用 OLEnterprise，那么系统上就没有必要有 DCOM。在 DCOM 出现之前，OLEnterprise 确实是一个非常有价值的工具，而且，它的确有几条远胜于 DCOM 的优点：

- 使用 OLEnterprise 可以连接两台 Windows 95 机器，即使在 NT 服务器不可用的时候也是可以的。而没有 NT 服务器的连接在 DCOM 里是不可能的，事实上，在有服务器时，两个 Windows 95 机器的连接在 DCOM 里也是极其困难的。
- OLEnterprise 有一个 Object Broker（对象代理），它可以在多个机器之间分配连接负载，特别是，每次一个新用户进入一个数据集的时候，它可以随机路由到一个可用的服务器上去，因此可以在多个服务器之间分配负载。
- OLEnterprise 具有错误处理的功能，只需要编写几句简单的代码就可以应用这个功能。

上面简单地讨论了 MTS、COM/DCOM、CORBA 和 OLEnterprise，那么 MIDAS 与它们之间有些什么关系呢？首先，可以说 TCP/IP、COM/DCOM、CORBA 和 OLEnterprise 是 MIDAS 得以实现的技术基础，MIDAS 提供了一种机制，可以使用它们的通讯功能，然后为它们加上事务处理的功能。或者说是 MIDAS 为开发人员提供了一种更加简化了的方法来利用 TCP/IP、DCOM、CORBA、OLEnterprise 开发分布式应用程序。

因为 MTS 是基于 DCOM 来开发分布式应用程序的，并提供了相应事务处理能力，所以 MIDAS 也可以利用 MTS 来开发分布式应用程序。MIDAS 和 MTS 是完全互补的技术，MTS 充当事务服务器管理对象的事务，而 MTS 开发人员则利用 MIDAS 得到瘦客户端的公文包模型，自动加上约束条件及进行商业规则的更新、数据的更新等。

13.1.5　MIDAS 的应用和未来

因为 Delphi、C++Builder 等开发环境提供了直接开发 COM/DCOM，CORBA 构件的能力，所以使用 MIDAS 来开发基于 COM/DCOM，CORBA 的多层分布式应用非常简单。利用 MIDAS 建立多层分布式应用程序，主要是创建客户端和服务器端。关于 MIDAS 客户程序和服务器端的创建步骤和细节，将在后面的 13.2.5 一节中详细介绍。

MIDAS 应该说是 Borland 公司推出的一项在多层分布式服务中重要的战略性产品，随着该项技术的日趋完善，和 MIDASII、MIDASClientforJava 的推出，MIDAS 同时支持 CORBA 及 COM/MTS，使 MIDAS 可以轻易地集成跨平台分布式对象的各项资源，让企业信息资源在不同平台架构中顺利集成应用。

总之，MIDAS 的优势可以总结为以下几条：

- MIDAS 是一个跨平台的中间件产品，允许开发人员使用单一界面和技术来存取各种分布式对象。
- MIDAS 简化了各种分散式对象的开发工作，并且能够大大简化发布式对象异质数据库的工作。
- MIDAS 在未来将会不断强化它的效率和功能。
- MIDAS 为多层分布结构的应用开发提供了强大的功能，这使得开发者再也无需为越来越庞大的数据及应用发愁了。

13.2　分布式应用程序基础

13.2.1　DataSnap 组件组

在本章的开头篇中，讲到 Delphi 6.0 提供了一组崭新的组件 DataSnap，它们提供了比 MIDAS 更加强大的多层应用的功能。下面就一一介绍这些组件。

图 13.2 中显示了 DataSnap 组件组中的组件组成。

图 13.2　DataSnap 组件组

DataSnap 组件组如下所示：

- TDCOMConnection：在多层分布式数据库应用程序中，TDCOMConnection 是连接数据服务器和用户应用层的主要中间层组件。它可以通过 ComputerName 特性来指

定远程服务器所在的计算机，通过 ServerName 特性关联计算机上的所有服务器，其中包含了服务器所提供的 TDataSetProvider（数据提供者）。客户应用程序使用 IAppServer 接口来通过 TDCOMConnection 与数据服务器进行通讯。

- TSocketConnection：与 TDCOMConnection 不同的是，TSocketConnection 是通过 TCP/IP 协议与服务器进行通讯的。它在多层分布式数据库应用程序中的作用和 TDCOMConnection 是相同的，都是作为一个中间层，连接客户端和数据服务器。
- TSampleObjectBroker：前面关于“MIDAS 内核技术”的讨论中已经涉及到了该组件。它提供了一个应用服务器的全局注册，其中维护了一组可用的数据服务器列表，当客户应用程序请求一个服务器时，TSampleObjectBroker 将随机地从服务器列表中选取一个可用服务器，提供给客户应用程序。该组件使得客户应用程序可以在运行过程中动态地获取服务器，甚至当当前正在使用的服务器被关闭以后，可以继续获得另外一个服务器。
- TWebConnection：该组件为客户应用程序提供了一种通过 HTTP 协议来连接数据服务器的途径。
- TConnectionBroker：该组件为各种中间层组件和客户端的连接提供了一种连接代理。当程序中有一个以上的服务器可用的时候，可能需要在这些服务器之间进行切换。例如，如果客户端应用程序中的 TDataSetProvider 有不止一个的服务器 Connection 可选，那么利用 TConnectionBroker 作为上面 TDataSetProvider 和 Connection 服务器之间的连接代理，TDataSetProvider 可以直接通过 ConnectionBroker 特性连接到一个 TConnectionBroker 上，而 TConnectionBroker 再通过 Connection 特性连接到一个 Connection 服务器。这样，当需要切换 Connection 服务器的时候，只需要重新设置 TConnectionBroker 的 Connection 特性，使之指向一个新的服务器即可，而不需要对应用程序中所有的 TDataSetProvider 进行 Connection 特性进行设置。图 13.3 显示出了 TConnectionBroker 在程序中的作用。

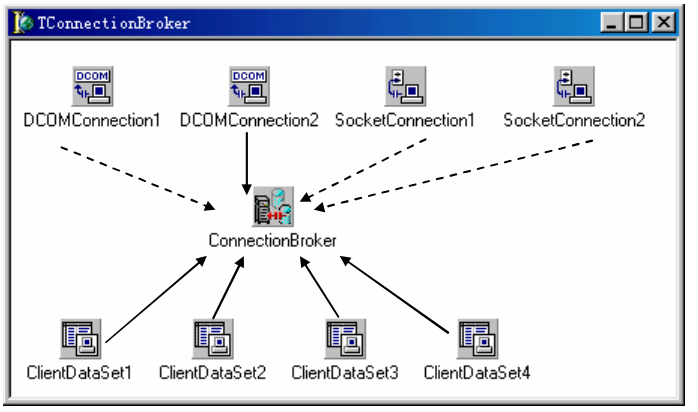

图 13.3 TConnectionBroker 所处的位置

- TSharedConnection：该组件不是直接连接到一个远程数据服务器上，而是连接到通过 ParentConnection 特性设定的数据服务器上的一个子数据模板上。

- TLocalConnection：该组件用来连接同处在一台计算机上的客户程序和数据服务器。
- TCorbaConnection：用来连接 CORBA 客户应用程序和数据服务器。

除了上面介绍的 DataSnap 组件以外，这里笔者还要特意介绍 TDataSetProvider 和 TClientDataSet 组件。TDataSetProvider 包含在 Data Access 组件组中，它的作用是在数据库服务器中，将一个 Table 或者 Query（本地表或者查询）通过服务器提供到外界，从远程计算机的客户应用程序中，通过 TClientDataSet 来连接到每一个 TDataSetProvider 上。在图中（见图 13.3），读者已经看到了 TClientDataSet 的使用，它的作用很明显，就是多层应用程序的客户端去连接服务器上的每一个数据提供者的对象。通过 TClientDataSet 组件，客户端就可以像访问本地数据库一样来访问远程数据服务器。

上面这几类组件，在后面的实际示例中都会应用到。通过那些示例，读者就会更好地理解这些组件的作用和使用方法。

13.2.2　建立 3 层 MIDAS 结构

传统的 Client/Server 体系结构已经不再满足现代应用程序的需要了，随着用户数量的增加，要把数兆的应用程序和支持文件数据发布给每一个用户的计算机上，而且还有配置和维护这些数据，已经变得不可能了。Web 的出现使很多人可以访问用户的应用程序，这要求应用程序要小而且尽量不需要配置。这也是为什么 Delphi 要设计分布式应用程序开发工具的原因。

Delphi 为实现分布式数据集提供了 4 种主要工具，下面分为两种情况来介绍：

1．在服务器端

Remote data module（远程数据模块）类似于标准数据模块，不同之处在于，它不是把数据传送到当前应用程序，而是传送到网络上的某一个位置。具体来说，它将一个简单的数据模块转换成一个 COM 对象，允许一个远程服务器通过 DCOM 来访问数据模块。

TDataSetProvider 组件驻留在远程数据模块上，就如同 TTable 对象可以驻留在一个标准数据模块上一样。不同之处在于，TDataSetProvider 在网络上传播一个表。提供者对象也可作为属性包括在 TTable 和 TQuery 对象中，以便网络上的其他程序可以通过 DCOM 从 TTable 和 TQuery 访问数据。远程数据模块的工作是让客户程序访问专用服务器上的提供者。客户程序首先连接至远程数据模块，然后查询远程数据模块以寻找服务器上的可获取的提供者列表。

2．在客户端，使用两个组件来访问服务器所提供的数据

TRemoteServer 组件给客户机提供连接至服务器的能力。更具体地说，它连接到 COM 接口，这一 COM 接口为远程数据模块所支持。尽管它名称中有 Server，但 TRemoteServer 是在客户端存在，而不是在服务器上的。TRemoteServer 知道如何在可获得的服务器中浏览注册表的组件。当服务器找到后，TRemoteServer 将连接到服务器上。

TClientDataSet 组件是挂接到 TRemoteServer 上，并且附着于服务器上的特定提供者。它们在连接一个远程数据集时，给予客户应用程序数据源以插入的能力。

下图描述了一个远程数据集应用程序的体系结构。其中，TDCOMConnection 代表一组连接远程数据模块的组件，它也可以是 TCorbaConnection、TSocketConnection 和 TOLEEnterpriseConnection 中的任一种，不过它们的设置不尽相同。

如果多个客户同时连接一个应用程序服务器，那么每个客户都将有一份远程数据模块的实例。更准确地说，每一个连接应用程序的 TDCOMConnection（或其他远程数据模块）组件的实例将有一份远程数据模块的实例被创建。

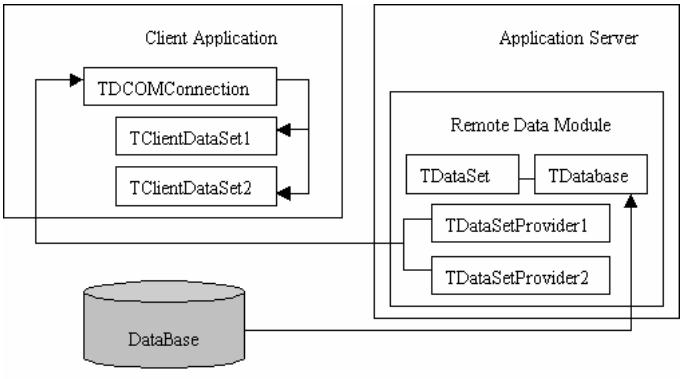

图 13.4 一个分布式应用程序的体系结构

13.2.3 创建 MIDAS 服务器

在这一节里，笔者将要描述 Delphi5 中创建一个 MIDAS 服务器的步骤：

（1）执行 File | New Application 命令，启动一个新的应用程序。

（2）执行 File | New | Multitier | Remote Data Module 命令，创建一个远程数据模块。设置 CoClass Name 为该服务器的名字，其他设置保持默认值。

（3）在远程数据模块上放置一个或多个的 TTable 或者 TQuery 组件，和相应个数的 TDataSetProvider 组件，如图 13.5 所示。

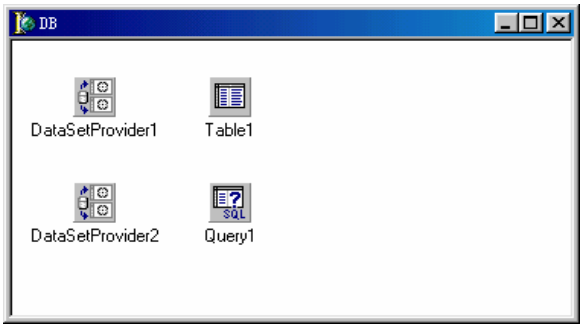

图 13.5 Remote Data Module 远程数据模块

（4）TDataSetProvider 的 DataSet 属性设置成相应的 TTable 或 TQuery 对象。

（5）保存项目，并运行服务器以便系统注册它。可以通过查找注册表来确认是否已经注册成功。

这样一个 MIDAS 数据服务器中将包含以下的代码：

1．远程数据模块

```
unit DBModule;

interface

uses
    Windows， Messages， SysUtils， Classes， ComServ， ComObj， VCLCom， DataBkr，
    DBClient， MidasServer_TLB， StdVcl， DBTables， Provider， Db;
type
    TDB = class(TRemoteDataModule， IDB)
        Table1: TTable;
        Query1: TQuery;
        DataSetProvider1: TDataSetProvider;
        DataSetProvider2: TDataSetProvider;
    private
        { Private declarations }
    protected
        class procedure UpdateRegistry(Register: Boolean; const ClassID，
ProgID: string); override;
    public
        { Public declarations }
    end;

implementation

{13R *.DFM}

class procedure TDB.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， TDB，
            Class_DB， ciMultiInstance， tmApartment);
end.
```

2．MidasServer 的库文件

这是服务器的主要实现部分，在这里可以看到服务器的实现细节。

```
unit MidasServer_TLB;
{13TYPEDADDRESS OFF}
// Unit must be compiled without type-checked pointers.
interface

uses Windows， ActiveX， Classes， Graphics， OleServer， OleCtrls， StdVCL，
        MIDAS;

const
        // TypeLibrary Major and minor versions
        MidasServerMajorVersion = 1;
        MidasServerMinorVersion = 0;

        LIBID_MidasServer: TGUID = '{250793EE-64A6-11D5-BF57-A1F39F25E171}';

        IID_IDB: TGUID = '{250793EF-64A6-11D5-BF57-A1F39F25E171}';
        CLASS_DB: TGUID = '{250793F1-64A6-11D5-BF57-A1F39F25E171}';
type
        IDB = interface;
        IDBDisp = dispinterface;
        DB = IDB;

        IDB = interface(IAppServer)
            ['{250793EF-64A6-11D5-BF57-A1F39F25E171}']
        end;

        IDBDisp = dispinterface
            ['{250793EF-64A6-11D5-BF57-A1F39F25E171}']
            function    AS_ApplyUpdates(const ProviderName: WideString;
Delta: OleVariant; MaxErrors: Integer; out ErrorCount: Integer;
var OwnerData: OleVariant): OleVariant; dispid 20000000;
```

```
            function    AS_GetRecords(const ProviderName: WideString; Count: Integer;
out RecsOut: Integer; Options: Integer;
const CommandText: WideString; var Params: OleVariant;
var OwnerData: OleVariant): OleVariant; dispid 20000001;
            function    AS_DataRequest(const ProviderName: WideString;
Data: OleVariant): OleVariant; dispid 20000002;
            function    AS_GetProviderNames: OleVariant; dispid 20000003;
            function    AS_GetParams(const ProviderName: WideString;
var OwnerData: OleVariant): OleVariant; dispid 20000004;
            function    AS_RowRequest(const ProviderName: WideString; Row: OleVariant;
RequestType: Integer; var OwnerData: OleVariant): OleVariant;
dispid 20000005;
            procedure AS_Execute(const ProviderName: WideString;
const CommandText: WideString; var Params: OleVariant;
var OwnerData: OleVariant); dispid 20000006;
        end;

        CoDB = class
            class function Create: IDB;
            class function CreateRemote(const MachineName: string): IDB;
        end;

{13IFDEF LIVE_SERVER_AT_DESIGN_TIME}
        TDBProperties= class;
{13ENDIF}
        TDB = class(TOleServer)
        private
            FIntf:          IDB;
{13IFDEF LIVE_SERVER_AT_DESIGN_TIME}
            FProps:          TDBProperties;
            function        GetServerProperties: TDBProperties;
{13ENDIF}
            function        GetDefaultInterface: IDB;
        protected
            procedure InitServerData; override;
        public
            constructor Create(AOwner: TComponent); override;
            destructor    Destroy; override;
            procedure Connect; override;
            procedure ConnectTo(svrIntf: IDB);
            procedure Disconnect; override;
            function    AS_ApplyUpdates(const ProviderName: WideString;
Delta: OleVariant; MaxErrors: Integer; out ErrorCount: Integer;
```

```
var OwnerData: OleVariant): OleVariant;
        function    AS_GetRecords(const ProviderName: WideString; Count: Integer;
out RecsOut: Integer; Options: Integer;
const CommandText: WideString; var Params: OleVariant;
var OwnerData: OleVariant): OleVariant;
        function    AS_DataRequest(const ProviderName: WideString;
Data: OleVariant): OleVariant;
        function    AS_GetProviderNames: OleVariant;
        function    AS_GetParams(const ProviderName: WideString;
var OwnerData: OleVariant): OleVariant;
        function    AS_RowRequest(const ProviderName: WideString; Row: OleVariant;
RequestType: Integer; var OwnerData: OleVariant): OleVariant;
        procedure AS_Execute(const ProviderName: WideString;
const CommandText: WideString; var Params: OleVariant;
var OwnerData: OleVariant);
        property    DefaultInterface: IDB read GetDefaultInterface;
    published
{13IFDEF LIVE_SERVER_AT_DESIGN_TIME}
        property Server: TDBProperties read GetServerProperties;
{13ENDIF}
    end;

{13IFDEF LIVE_SERVER_AT_DESIGN_TIME}

TDBProperties = class(TPersistent)
    private
        FServer:      TDB;
        function      GetDefaultInterface: IDB;
        constructor Create(AServer: TDB);
    protected
    public
        property DefaultInterface: IDB read GetDefaultInterface;
    published
    end;
{13ENDIF}

procedure Register;

implementation

uses ComObj;
class function CoDB.Create: IDB;
begin
```

```
        Result := CreateComObject(CLASS_DB) as IDB;
end;

class function CoDB.CreateRemote(const MachineName: string): IDB;
begin
        Result := CreateRemoteComObject(MachineName，  CLASS_DB) as IDB;
end;

procedure TDB.InitServerData;
const
        CServerData: TServerData = (
                ClassID:     '{250793F1-64A6-11D5-BF57-A1F39F25E171}';
                IntfIID:     '{250793EF-64A6-11D5-BF57-A1F39F25E171}';
                EventIID:    '';
                LicenseKey: nil;
                Version: 500);
begin
        ServerData := @CServerData;
end;
procedure TDB.Connect;
var
        punk: IUnknown;
begin
        if FIntf = nil then
        begin
                punk := GetServer;
                Fintf:= punk as IDB;
        end;
end;

procedure TDB.ConnectTo(svrIntf: IDB);
begin
        Disconnect;
        FIntf := svrIntf;
end;

procedure TDB.DisConnect;
begin
        if Fintf <> nil then
        begin
                FIntf := nil;
        end;
end;
```

```
function TDB.GetDefaultInterface: IDB;
begin
        if FIntf = nil then
                Connect;
        Assert(FIntf <> nil， 'DefaultInterface is NULL. Component is not connected
to Server. You must call "Connect" or "ConnectTo" before this
operation');
        Result := FIntf;
end;

constructor TDB.Create(AOwner: TComponent);
begin
        inherited Create(AOwner);
{13IFDEF LIVE_SERVER_AT_DESIGN_TIME}
        FProps := TDBProperties.Create(Self);
{13ENDIF}
end;

destructor TDB.Destroy;
begin
{13IFDEF LIVE_SERVER_AT_DESIGN_TIME}
   FProps.Free;
{13ENDIF}
        inherited Destroy;
end;

{13IFDEF LIVE_SERVER_AT_DESIGN_TIME}
function TDB.GetServerProperties: TDBProperties;
begin
        Result := FProps;
end;
{13ENDIF}
function    TDB.AS_ApplyUpdates(const ProviderName: WideString;
Delta: OleVariant; MaxErrors: Integer; out ErrorCount: Integer;
var OwnerData: OleVariant): OleVariant;
begin
        Result := DefaultInterface.AS_ApplyUpdates(
ProviderName， Delta， MaxErrors， ErrorCount， OwnerData);
end;

function    TDB.AS_GetRecords(const ProviderName: WideString; Count: Integer;
```

```
    out RecsOut: Integer; Options: Integer; const CommandText: WideString; var Params: OleVariant; var
OwnerData: OleVariant): OleVariant;
    begin
        Result := DefaultInterface.AS_GetRecords(
    ProviderName，Count，RecsOut，Options，CommandText，
    Params， OwnerData);
    end;

    function  TDB.AS_DataRequest(const ProviderName: WideString;
    Data: OleVariant): OleVariant;
    begin
        Result := DefaultInterface.AS_DataRequest(ProviderName， Data);
    end;

    function  TDB.AS_GetProviderNames: OleVariant;
    begin
        Result := DefaultInterface.AS_GetProviderNames;
    end;

    function  TDB.AS_GetParams(const ProviderName: WideString;
    var OwnerData: OleVariant): OleVariant;
    begin
        Result := DefaultInterface.AS_GetParams(ProviderName， OwnerData);
    end;

    function  TDB.AS_RowRequest(const ProviderName: WideString;
    Row: OleVariant; RequestType: Integer;
    var OwnerData: OleVariant): OleVariant;
    begin
        Result := DefaultInterface.AS_RowRequest(ProviderName， Row，
    RequestType， OwnerData);
    end;

    procedure TDB.AS_Execute(const ProviderName: WideString;
    const CommandText: WideString; var Params: OleVariant;
    var OwnerData: OleVariant);
    begin
        DefaultInterface.AS_Execute(ProviderName， CommandText，
    Params， OwnerData);
    end;

    {13IFDEF LIVE_SERVER_AT_DESIGN_TIME}
    constructor TDBProperties.Create(AServer: TDB);
```

```
begin
    inherited Create;
    FServer := AServer;
end;

function TDBProperties.GetDefaultInterface: IDB;
begin
    Result := FServer.DefaultInterface;
end;

{13ENDIF}

procedure Register;
begin
    RegisterComponents('Servers',[TDB]);
end;

end.
```

13.2.4 理解服务器

实际上，建立一个 MIDAS **服务器，用户只需要知道一点细节，并不需要详细的知识。以下代码为实现单元中的主要类：**

```
TServerName = class (TRemoteDataModule， IserverName)
    myTable: TTable;
    myProvider: TDataSetProvider;
protected
    function Get_myProvider:Iprovider; safecall;
end;
```

这个类型的类声明可提供一个 COM **对象的实现，这个** COM **接口称为** IServerName**。读者会注意到** IserverName **是双重接口的** COM **对象，这意味着它支持** IDispatch **和** IServerName**。其他应用程序可通过** OLE **自动化或从远程机器上通过分布式** COM **来访问。**ServerName **有一个附带的类型库，在独立的单元里声明。**

即便没有任何方法，也可以从另一个应用程序或另一台机器上来访问 IServerName **对象。在这种情况下，只需要启动服务器。**

为了可以使用服务器，必须有一种机制来允许客户得到放置在远程数据模块中的TDataSetProvider **对象，具体来说，就是需要被** TDatasetProvider **组件支持的** TDatasetProvider **接口。**

13.2.5 创建和理解 MIDAS 客户程序

创建可与 MIDASServer **会话的** MIDAS **客户程序并不很困难。在这个过程中，建议在同一台机器上运行客户程序和服务器程序，这样对于程序的调试非常方便，因为往往需要**

同时调试服务器和客户端。调试结束后，然后再在网络上分发应用程序。

以下是创建客户程序的步骤：

（1）创建一个新的应用程序并保存，将主窗体命名为 MainClient.pas，保存程序部分为 MidasServer.pas。

（2）从组件选项板的 DataSnap 选项卡上选择一个 TDComConnection 组件放在主窗体上，并设置其 ComputerName 属性为服务器端的机器名；然后，展开 ServerName 属性下拉列表框，将看到所选择的服务器机器上所有的注册 MidasServer，其中包括前面创建过的 MidasServer.DB，选择它，如图 13.6 所示。

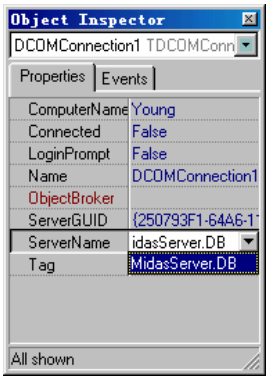

图 13.6　TDCOMConnection 的 ServerName

（3）从 DataAccess 选项卡选择并放置一个 TClientDataSet 组件，设置其 RemoteServer 属性为上一步中放置的 DComConnection1 组件，此时打开 TClientDataSet 的 ProviderName 属性下拉列表框，将看到所有 MidasServer.DB 中所提供出来的数据集，选择其中一个，如图 13.7 所示。

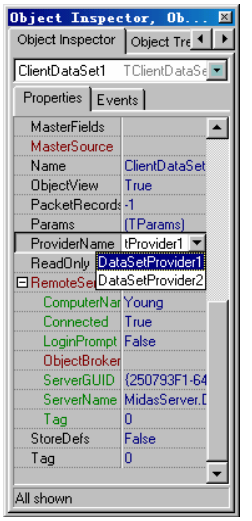

图 13.7　TClientDataSet 的 ProviderName

（4）最后，就可以放置一个 TDataSource 组件和一个 TDBGrid 组件，将它们进行连接之后就可以查看服务器端的数据了，如图 13.8 所示。

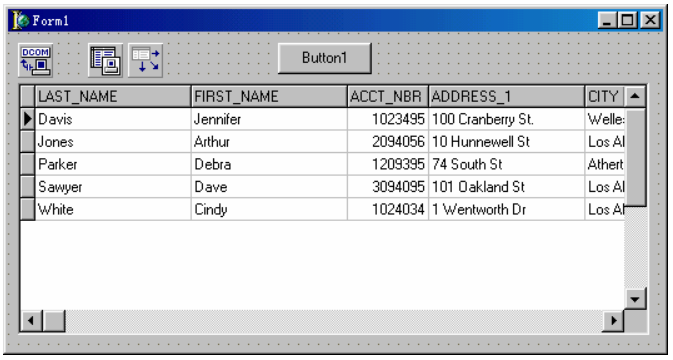

图 13.8 客户端显示 MIDAS 服务器中的数据集

TClientDataSet 组件支持强大的 SQL 功能，可以通过它的 CommandText 特性将合法的 SQL 语句传递给服务器端，服务器将执行 SQL 语句进行相应的操作或者查询返回特定的数据集。

对于 TDCOMConnection 组件，当 Connected 特性置为 True 的时候，它将在 VCL 中调用 CoCreateInstance 函数。不难发现，它使用了与 COM、DCOM 和 OLE 自动化相同的技术，在 COM 和 DCOM 的相应章节中已经讨论过这个技术了。

当使用其他技术（例如 OLEnterprise、TCP/IP 或者 CORBA）时，MIDAS 同样可以工作。之所以存在这样的灵活性，是因为 MIDAS 技术的核心是在 OleVariant（或一个 CORBA）中包装了一个数据集，用于网络间通信的传输并不十分重要。MIDAS 技术的基础其实是 COM 程序模型，当 MIDAS 使用 TCP/IP 时，TCP/IP 看起来和感觉起来就像 COM 一样。

13.2.6 远程访问服务器

必须使服务器同时在客户机和服务器上注册。如果注册失败，客户程序仍可查找并启动服务器，但如果服务器的类型库并没有在客户机上注册，COM 将不能够配置数据进出。只需要将服务器程序在机器上运行一次就可以注册服务器。

假如已经正确地设置了每一项，那么连接客户程序与服务器所要做的只是设置 TDCOMConnection 组件的 ComputerName 特性。例如下面的示例：

```
procedure TForm1.Button1Click(Sender: TObject);
var
    S: string;
begin
    S := '';
    if InputQuery('Enter the Machine Name', 'Machine Name', S) then begin
        DCOMConnection1.ComputerName := S;
        DComConnection1.Open;
        ClientDataSet1.Open;
```

```
        end;
end;
```

如果在网络上没有可用的 NT 域服务器，那么就不应该使用 DCOM，而应使用 TCP/IP。即便是 NT 域服务器，Socket（套接字）连接也会起作用，而且它通常比 DCOM 连接建立起来要容易得多。但是，在 Socket 连接中，安全性相对要差一些。

建立 Socket 为基础的 MIDAS 程序，需要在服务器上运行 ScktSrvr.exe 程序，否则服务器将不会工作。

将一个 DCOM 基础上的 MIDAS 程序转换为 Socket 程序非常简单，只需要将 TDCOMConnection 组件替换为 TSocketConnection 组件，它的 ServerName 特性的设置和 TDCOMConnection 组件一样。服务器 IP 地址可以通过 Address 特性设定。

13.3　建立一对多应用程序

下面将介绍一个使用 MIDAS 建立的主从表程序，主要目的是为了使读者清楚一点，在 MIDAS 应用程序中可以做和在一个标准的 Delphi 数据库应用程序中相同的事情。

13.3.1　创建步骤

首先创建一个一对多服务器，具体步骤在本章前面的小节中已经描述过，如果现在还不是很清楚的话，建议读者回头重新看一遍相关的文字。

服务器命名为 CustOrderRemoteData，在远程数据模块上放置两个 TTable 对象，通过 DBDEMOS 别名分别连接一个 Customer 表、一个 Order 表。TTable 命名为 tbCustomer 和 tbOrder。

需要注意，在这里的一对多关系并不是在服务器上体现的，而是在客户端上体现的，所以，不要在服务器上创建一对多关系，而应该在客户程序上创建。在本章后面“服务器端和客户端逻辑”小节里所提及的示例中，情况正好与此相反，笔者将说明这两种方法的相对优点。

如果愿意的话，可以在应用程序的主窗体上添加一个画面或一些文字，例如：“MIDAS 服务器”。这时，可以保存工作，并运行应用程序来注册它。

运行服务器后，在 Windows 注册表里将新增加一个条目。可以运行 Windows 程序 RegEdit.exe，打开注册表，展开 HKEY_CLASSES_ROOT 并搜索包含服务器的可执行文件名，例如，在这个示例里，要搜索的键为：CustOrderServer.CustOrdersRemoteData，它也叫做服务器的 Prog ID。这个主键的前半节来自服务器的可执行文件名，后半节来自用户所给出的远程数据模块的名称。这一个名称也同样是默认时导出的 COM 接口名。

现在来创建一个客户程序，来连接上面创建的服务器。

虽然在前面 13.2.5 一节中已经介绍了创建客户程序的步骤，但为了更加具体地说明，这里将详细写出这个示例中的各个步骤：

（1）从 DataSnap 选项卡上选择一个 TDCOMConnection 组件放置在主窗体上，设置 ServerName 特性为 MidasServer.CustomerOrderRemoteData，选定以后，将 TDCOMConnection

的 Connected 特性设为 True，这时候服务器程序将自动启动，如果没有启动的话，那么确认一下已经在当前机器上运行了服务器程序以便注册它。

（2）从 Data Access 选项卡选择两个 TClientDataSet 组件，分别命名为 cdsCustomer 和 cdsOrder，设置它们的 RemoteServer 特性为刚刚连接的 TDCOMConnection 组件，分别设置 ProviderName 特性为 ProviderCustomer 和 ProviderOrder，这是服务器上的数据提供者。

（3）从 Data Controls 选项卡上选择两个 TDBGrid 放置在主窗体上，然后分别挂接到上面的两个 TClientDataSet 上，这里的具体操作和 TTable、TQuery 完全一样。也就是说，通过 TDataSource 关联它们。

（4）设置两个 TClientDataSet 之间的一对多关系。将 cdsOrder 组件的 MasterSource 设置为连接 cdsCustomer 的 TDataSource 组件。单击 cdsOrder 的 MasterFields 特性，在两个表的 CustNo 字段之间建立关联。如图 13.9 所示。

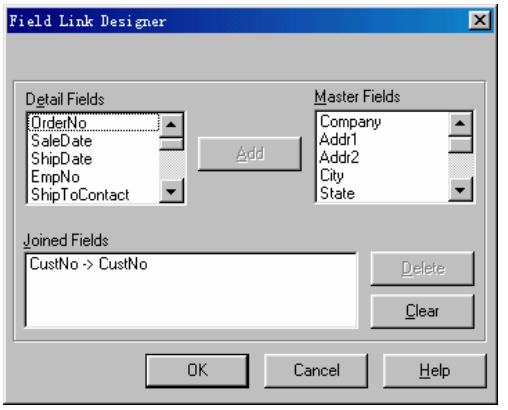

图 13.9 设置 MasterFields 特性建立一对多关联

这个示例最吸引人的地方或许是它使用 TClientDataSet 组件的 MasterSource 和 MasterField 特性来建立一对多关系的能力。其关键之处在于 Delphi 使用户可以加强规则并在分布式程序中使用可视的组件，正如在本地数据库程序中一样。这种能力和灵活性也是 MIDAS 技术的优点。

13.3.2 刷新和更新数据

这一节中，将讨论在客户程序上编辑数据以后如何更新和刷新服务器数据，Delphi 已经尽可能地使整个过程完整地传递，但是仍需要学习几条简单的规则。

客户端向服务器提交数据更改，应该调用 TClientDataSet 的 ApplyUpdates 方法，如下所示：

```
cdsCustomer.ApplyUpdates(-1);
```

这个方法中包含了一个 MaxErrors 参数，传递“-1”表示在错误发生的时候马上停止更新数据进程。这种情况下，服务器上的数据将不会发生任何变化。客户端会向用户提示错误信息。如果将 MaxErrors 参数设置为一个正数，那么更新过程将一直进行下去，直到达到数值指定的错误数目，如果发生这种情况，那么无论是服务器的数据还是改变日志都

不会发生任何改变。如果错误数目小于 MaxErrors，那么将更新所有成功改变的记录，并从改变日志中删除这些记录，ApplyUpdates 函数返回它所遇到的错误数目。

为了弄清楚这里发生的事情，需要明白 Delphi 缓存了对数据集所做的所有改变。换句话说，它保存了原始数据及更新数据。当调用 ApplyUpdates 时，可以报告错误，读者将有机会来恢复原始记录或者试图完成整个改变。TClientDataSet 还提供了一个 Undo 函数，如下所示：

```
function UndoLastChange(FollowChange: Boolean): Boolean;
```

这个函数将取消 TClientDataSet 最后的数据操作，包括插入、删除和编辑。参数 FollowChange 如果为 True，那么取消操作以后，数据集当前记录将定位到相应的恢复记录上；如果为 False，那么当前记录不会变化。

如果存在其他的用户也在操作服务器上的数据集，或许希望及时地看到最新的数据情况。这时候，可以调用方法 Refresh，该方法将重新从服务器上读取最新的数据，包括其他用户通过 ApplyUpdates 提交的操作，如下所示：

```
if cdsCustomer.ApplyUpdates(-1) = 0 then
    cdsCustomer.Refresh;
```

13.3.3　公文包模式

前面提到过 MIDAS 的“公文包模式”，它提供了一种数据移动操作的途径，也就是使数据脱离开原有的数据库，还仍然能够正常地进行浏览和编辑。

TClientDataSet 的公文包模型允许加载或者保存客户数据集内容到磁盘上。它依赖于两个方法：LoadFromFile 和 SaveToFile，如下所示：

```
cdsCustomer.SaveToFile('C:\Customer.cds');
cdsCustomer.LoadFromFile('C:\Customer.cds');
```

公文包模式对于笔记本计算机用户显得尤为重要，因为从服务器上断开连接以后，仍然可以通过简单地调用 Load 方法来访问数据。例如，如果正在使用笔记本计算机，可以连接服务器一次并将检索到的数据集保存到磁盘上，然后关闭机器并带回家，在路上或者回到家中，可以再次打开数据集浏览或者编辑它。当重新连接到服务器上的时候，可以通过调用 ApplyUpdates 来更新这些文件。

在一些应用程序中，如果想要保存主从表，应该在分离的文件中保存和读取它们。

13.3.4　PacketRecords

TClientDataSet 重要的 PacketRecords 特性是需要花费一些时间分析的，下面将从几个不同的角度来讨论它，其中将特别对使用公文包模型时它的重要性进行说明。

简单地说，PacketRecords 特性表示了当 ClientDataSet 从服务器上的 Provider 中取数据的时候，每次提取多少条数据记录。如果设置 PacketRecords 为“−1”，表示一次全部提取全部数据；如果设置 PacketRecords 为“0”，表示获得元数据。

为了使公文包模型正常工作，有时需要确保服务器文件不以一对多的关系安排，并确保设置在两个 ClientDataSet 上的 PacketRecords 属性为“-1”。设置 PacketRecords 特性为“0”来获得元数据，设置为“-1”来获得所有数据，设置为某个整数“n”可获得所需的 n 个记录。

如果已得到一个数据集的数据，设置它的 PacketRecords 特性为“-1”或某个正整数来检索数据，而不要设置为“0”；但是如果还没有得到元数据，那么设置 PacketRecords 为“-1”或者某个正整数将会检索元数据和记录。

注意：使用分布式数据集时，有时候用户并不想一次看到 10000 或者 20000 条记录，应该查找方法用查询来过滤数据或者设置 PacketRecords 为一个比较小的数目“n”，以得到一个合理的用户浏览记录数。Delphi 会自动保存状态，因此，随后的要求会得到随后的 n 个记录，而不是检索到和第一次一样的 n 个记录。

如果要使用公文包模型，那么应得到整个数据集，这意味着应设置 PacketRecords 为“-1”；如果只是想要建立一对多的关系，而且并不对公文包模型感兴趣，那么可以在从表上设置 PacketRecords 为“0”，而主表上设置为“-1”，这样将从主表上检索全部数据，从表上检索元数据。最后在查看一个特定的记录时，Delphi 会调用 TclientDataSet.AppendData 方法来得到想要的数据。

在使用公文包模型时，通常设置 PacketRecords 特性为“-1”，并且会完成主从表关系，但整个主从表记录将一次从服务器上获得，这对于很大的数据集来说显然是不实际的。因为这个问题非常重要，所以这里介绍一个示例来说明这一问题。下面的代码代表了一种“不敏感”的情况，这里首先检索元数据，然后检索记录。之所以不敏感是因为在首次访问数据集时，就可以自动检索元数据，如下所示：

```
with ClientDataSet1 do begin
        close;
        PacketRecords := 0;
        Open;
        PacketRecords := -1;
        GetNextPacket;
end;
```

如果想再完善一点，可以参考以下代码：

```
var
        RecsOut: Integer;
        V: OleVariant;
begin
        ClientDataSet1.Close;
        V := ClientDataSet1.Provider.GetMetaData;
        ClientDataSet1.AppendData(V， False);
        V := ClientDataSet1.Provider.GetRecords(-1， RecsOut);
        ClientDataSet1.AppendData(V， True);
```

```
end;
```

在这个示例中，首先关闭了数据集，然后使用 GetMetaData 函数在一个 OleVariant 中检索元数据，这时候，可以直接将这一函数的结果送至 AppendData，它会将所检索到的记录附加到任何当前的数据集中。

AppendData 函数包括两个参数：第一个是从服务器上检索到的数据，第二个表示当把检索到的数据加到数据集中的时候是否触发数据集的 EOF，即表示已经到数据集的尾部。

在这一部分中，笔者没有深入讨论 constraint（约束），正如用户可以自动为应用程序下载元数据，也可以下载约束。为达到这一目的，需要设置 IProvider 接口的 Constrains 字段为 True，这样，在服务器上所创建的约束将会自动建立在客户机上了。

13.4　错 误 处 理

在处理远程数据集时，不可避免地会出现错误，例如，如果两个用户同时访问一个表，他们可能想要更改同一条记录，在这种情况下，首先执行更新的人会成功地更改记录，而第二个人将会得到一个错误记录。

可以通过处理 TClientDataSet 组件的 OnReconcileErrorEvent 事件来及时地响应错误。处理 Application Server 返回的错误的技巧是使用一个存储于 Delphi Object Repository 的窗体。可以通过 File | New | Other 命令打开 New Items 对话框，再转到 Dialogs 选项卡，然后选择 ReConcile Error Dialog 窗体，如图 13.10 所示，将该窗体保存到当前工程并从自动创建窗体列表中删除它。

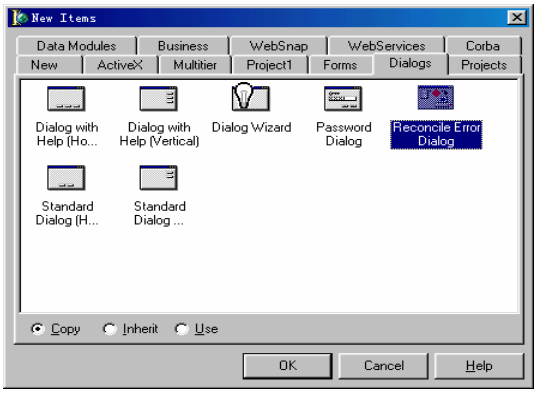

图 13.10　选择 Reconcile Error Dialog 窗体

将 Reconcile Error Dialog 窗体添加到相应窗体的 Uses 中，就可以通过下面的事件来响应一个 OnReconcileError 事件，代码如下所示：

```
procedure TForm1.ClientDataSet1ReconcileError(
        DataSet: TCustomClientDataSet; E: EReconcileError;
        UpdateKind: TUpdateKind; var Action: TReconcileAction);
begin
```

```
        Action := HandleReconcileError(DataSet， UpdateKind， E);
    end;
```

这时，将会启动在 Object Repository 中找到的对话框，并允许用户在其中处理任何错误，如图 13.11 所示。

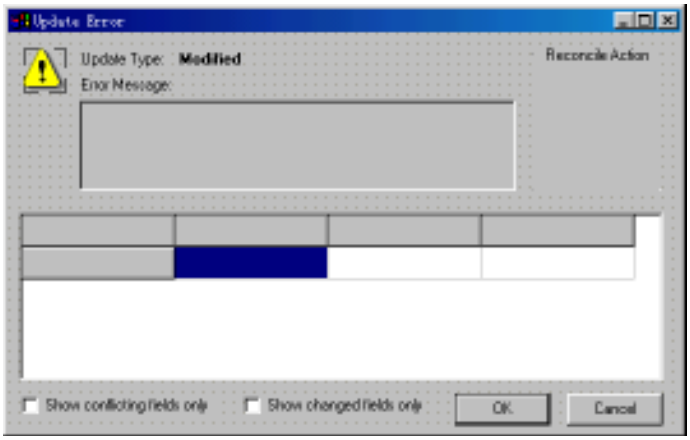

图 13.11 Update Error 对话框

对话框中心的网格说明了错误发生的字段名称。其中将包含客户应用程序想要在记录中插入的值，以及其他用户冲突的值，其他用户是在之前已经成功更新插入的用户。另外还包含有该记录在更新之前的值。在 Reconcile Action 列表框中，可以选择 Skip、Cancel、Correct、Refresh 或者 Merge 等操作。

完全可以通过编写自己的代码来进行所有的这些更新以及访问所有选项。但是，如果完全使用这一个对话框或者使用它作为自己代码的基础，会使工作变得容易很多。

13.5 服务器端和客户端逻辑

用户可以决定有多少逻辑放在应用程序的中间层上，即服务器上，例如，可以决定是否在服务器上建立表和表之间的主从关系。如果建立了主从关系，当从客户机上查询服务器上的数据集时，对于从表，将只能得到服务器上当前可见的关系，也就是说，主从逻辑关系在服务器端就已经确定了，实际上这一结果也许正是用户所希望的，特别是在从表相当大的情况下。

但是，如果主表和从表都非常小，那么用户也许宁愿从两个表上访问到所有的数据。下面将介绍一个示例，它在服务器端上创建一对多的主从逻辑，但是一次传送了整个数据集。而且，这个程序允许使用嵌套的数据集，这里的从表嵌套在主表的一个字段中。

服务器端可以直接利用前面创建的 Customer-Order 的程序，需要在两个表之间建立主从关系，这里 Customer 表为主表，Order 为从表，它们通过 CustNo 字段关联在一起。

其实，在这个示例中，服务器端可以只设置一个 TDataSetProvider，连接 Customer 表，因为 Order 表已经嵌进了 Customer 表中的一个字段。

在客户端，设置 TDCOMConnection 和一个 TClientDataSet，具体设置方法和前面的 MIDAS 客户端程序设置方法相同，TClientDataSet 连接服务器端的 Customer 表，然后连接到一个 TDBGrid 对象上。设置 TclientDataSet 的 Active 特性为 True 后，运行程序，可以看到，在 DBGrid 表格的最右面新加了一个字段 tbOrder，数据标识为 DATASET。单击这个字段，会看到一个省略号按钮；单击该按钮，将会出现一个新的表格，其中显示了当前 Customer 表记录所关联的 Order 从表中的数据，如图 13.12 所示。

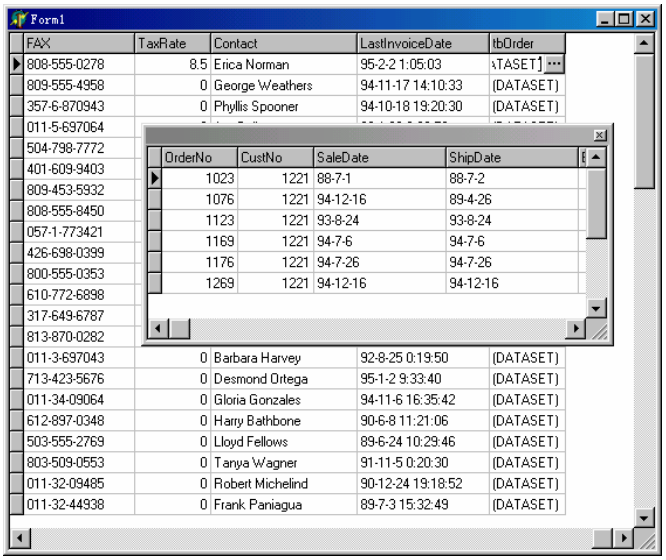

图 13.12　服务器端的主从表逻辑在客户端的显示

已经看到了，不需要在客户端做任何特殊的工作来实现上面的这一切，用户所需要做的只是创建一个标准的 MIDAS 客户端程序，就可以拥有一个所有逻辑都齐备的嵌套数据集。

上面的这个示例也包括了在服务器上调用定制方法的逻辑，如果已经读过了前面讨论的 COM 以及 DCOM 的章节，那么应当感觉到这种调用方法是可行的。毕竟，TRemoteDataModule 对象只是支持标准 COM 接口的标准的 COM 实现，因此，对此接口添加自己的定制方法是可行的。

首先，在 Type Library Editor 中，通过 New Method 按钮加入一个 GetName 方法，在 Parameters 选项卡上指定这一方法的返回值类型为 WideString，即 BSTR，填充该方法的代码如下所示：

```
function TCustomerOrderRemoteData.GetName: WideString;
begin
  Result := 'Nested Data';
end;
```

到此为止，服务器上要做的工作就已经完成了。如果不是很清楚前面提到的 COM 以及类型库 Type Library 的某些方面，建议查看 COM 和 DCOM 的相关章节。

在客户端，Delphi 已经使许多事情变得十分容易了，特别是，TDCOMConnection 的 AppServer 字段是一个包含有服务器接口的 IDispatch 包装的实例，可以这样调用它，如下所示：

```
ShowMessage(DCOMConnection.AppServer.GetName);
```

AppServer 是一个包含了 IDispatch 实例的 Variant，可以用它来从服务器上访问相应的接口，调用接口中的方法。

13.6 本 章 小 结

本章介绍了 Borland 的多层应用开发技术，详细讨论了 MIDAS 的核心技术及其体系结构。

在理解了 MIDAS 的应用结构的基础上，本章详细地创建了 3 层 MIDAS 应用结构，并给出了服务器端和客户端应用程序的示例，以便读者深入理解它们。同时示例中包括了 DataSnap 组组件和 MIDAS 客户端组件的使用。

第 14 章　创建 ActiveX 控件

对于许多开发者来说，能够轻松地创建 ActiveX 控件是 Delphi 最重要的功能之一。ActiveX 是一种与语言无关的编写标准，它适用于许多不同的环境，包括 Delphi、C++Builder、VB 以及 Internet Explorer 等。ActiveX 可以是一个简单的文本框，用来显示简单的文字信息，也可以是一个复杂的 Word 字处理工具或者是一个电子表格设计器。Delphi 使 ActiveX 控件非常容易创建和使用，用户完全可以在现有的 VCL 控件或者 Form 的基础上将它们转换为 ActiveX 控件。

本章将不会把 ActiveX 控件的方方面面都讲到，因为它同 COM 一样也需要一本书来讲解，本章的重点将是在前面讲解 COM、ActiveX 和 VCL 组件的基础上，介绍如何在 Delphi 中创建 ActiveX 控件。

14.1　创建 ActiveX 控件的原因

作为一个 Delphi 开发者，也许会想，既然已经有了 VCL 组件和 Form，为什么还要创建 ActiveX 控件呢？最重要也是最主要的一个原因是，创建 ActiveX 控件不仅仅可以在 Delphi 和 C++Builder 环境中使用，而且可以在其他的 Win32 的开发环境中使用，这将大大地扩大用户所开发的控件的适用潜力。Delphi 所提供的由现有的 VCL 控件转换为 ActiveX 控件的功能，可以节省巨大的付出。

14.2　创建 ActiveX 控件

在 Delphi 中，只需要简单的一步向导就可以创建一个 ActiveX 控件。但是，要创建一个专业的 ActiveX 控件，这个向导是远远不够的。

可以通过 File | New 命令来打开 New Items 对话框，其中的 ActiveX 选项卡如图 14.1 所示，它列出了 Delphi 中的 ActiveX 功能。

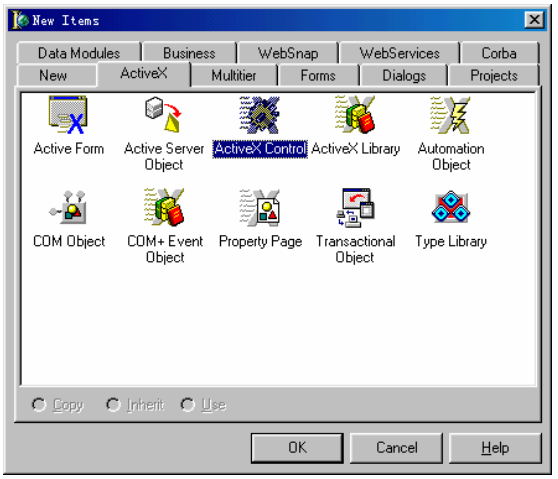

图 14.1 New Items 对话框中的 ActiveX 选项卡

14.2.1 ActiveX 控件向导

在 New Items 对话框中的 ActiveX 选项卡上，双击 ActiveX Control 图标将打开 ActiveX 控件向导，如图 14.2 所示。

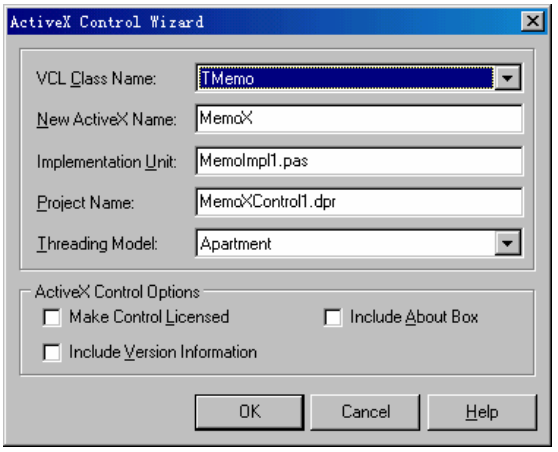

图 14.2 ActiveX 控件向导

这个向导能够把一个 VCL 控件转换为一个 ActiveX 控件，另外，向导能够让用户指定 ActiveX 控件的类名、实现单元的名称、ActiveX 控件所在项目的名称等。

在这个向导中，如果其中的下拉 VCL Class Name 列表框，就会发现，并不是所有的 VCL 组件都列出来了，而是只有其中的一部分。只有满足下面 3 个条件的 VCL 控件才会列在 VCLClass Name 下拉列表框里，如下所示：

- VCL 控件必须已经安装在了已经选项板上，即组件必须位于已安装的设计期包中。
- VCL 控件必须继承于 TWinControl，非窗口的控件不能转换为 ActiveX 控件。

- VCL 控件没有调用过 RegisterNonActiveX()方法。

许多标准的 VCL 控件无法转换为 ActiveX 控件，其中有些是因为没有转换的意义，有些是因为转换起来非常困难。例如，TDBGrid 控件就没有必要转换，因为它还需要有另外一个 VCL 组件 TDataSource 与它一起工作。TTreeView 组件则很难转换，因为它很难表达节点的概念。

ActiveX 向导中有 3 个复选框，它们允许用户设置一些选项。

- Include Design-Time License：如果选中这个复选框，向导将生成一个许可文件，用来防止未经授权的人在开发环境中使用 ActiveX 控件，应当生成一个 LIC 文件（许可文件），随同 OCX 文件一起发布给合法用户。
- Include Version Information：如果选中这个复选框，OCX 中将包含版本信息，版本信息可以在 Project Options 对话框中的 VersionInfo 选项卡上进行设置。
- Include About Box：如果选中这个复选框，ActiveX 控件将有一个 About 对话框，在开发环境中，右击 ActiveX 控件，从弹出的快捷菜单中单击 About 命令可以打开它。

设置完基本信息以后，单击 OK 按钮，ActiveX 控件向导就会将所选的 VCL 控件转换为 ActiveX 控件。看到的最终结果就是一个 ActiveX 库，其中包括一个 ActiveX 控件。向导把一个 VCL 组件转换为一个 ActiveX 控件，实际上经过了以下几个步骤：

（1）首先判断所选的 VCL 控件在哪个单元中，然后把这个单元传给编译器，让编译器为 VCL 控件中的特性、方法和事件生成特殊的符号信息。

（2）创建一个类型库，其中包含一个接口、一个调度接口和一个组件类。

（3）向导遍历 VCL 控件的所有符号信息，把其中的特性和方法加到接口中，把事件加到调度接口中。

（4）类型库编辑器把类型库中的信息翻译成 Object Pascal 代码。

（5）最后，向导生成一个 ActiveX 控件的实现单元，实现其中的接口。

注意：并不是 VCL 控件中的所有特性、方法和事件都会加到类型库中，只有当那些特性、方法的参数和返回值、事件的类型与 Automation 兼容时，它们才会被加到类型库中。与 Automation 兼容的数据类型有：Byte、SmallInt、Integer、Single、Double、Currency、TDateTime、WideString、WordBool、PSafeArray、TDecimal、OleVariant、IUnknown 和 IDispatch。另外，如果参数的类型是 TStrings、TPicture 和 TFont，也是允许加到类型库中的。

14.2.2　ActiveX 控件示例

为了说明上面介绍的内容，下面列出了一个由 VCL TPanel 控件转换来的 ActiveX 控件 TPanelX 的程序清单，其中详细地描述了类型库中如何定义接口、调度接口以及组件类等信息如下所示。图 14.3 是这个 ActiveX 控件的类型库。

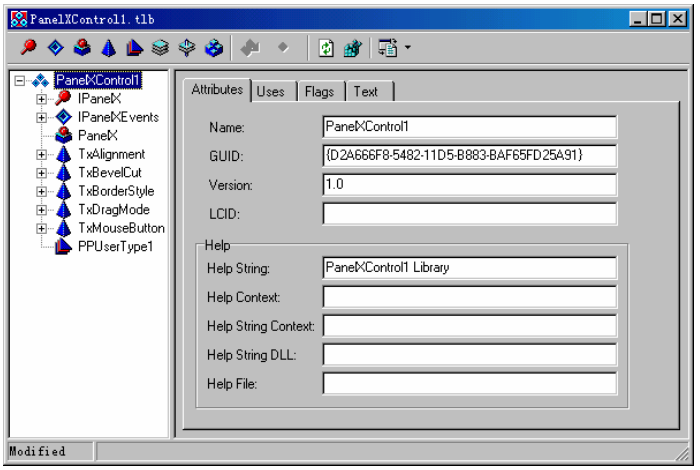

图 14.3　ActiveX **控件的类型库**

```
unit PanelXControl1_TLB;

// ************************************************************************ //
// WARNING
// -------
// The types declared in this file were generated from data read from a
// Type Library. If this type library is explicitly or indirectly (via
// another type library referring to this type library) re-imported, or the
// 'Refresh' command of the Type Library Editor activated while editing the
// Type Library, the contents of this file will be regenerated and all
// manual modifications will be lost.
// ************************************************************************ //

// PASTLWTR : 14Revision:      1.88    14
// File generated on 01-5-29 22:39:44 from Type Library described below.

// ************************************************************************ //
// NOTE:
// Items guarded by 14IFDEF_LIVE_SERVER_AT_DESIGN_TIME are used by properties
// which return objects that may need to be explicitly created via a function
// call prior to any access via the property. These items have been disabled
// in order to prevent accidental use from within the object inspector. You
// may enable them by defining LIVE_SERVER_AT_DESIGN_TIME or by selectively
// removing them from the 14IFDEF blocks. However, such items must still be
// programmatically created via a method of the appropriate CoClass before
// they can be used.
// ************************************************************************ //
// Type Lib: C:\Program Files\Borland\Delphi5\Projects\PanelXControl1.tlb (1)
```

```
// IID\LCID: {D2A666F8-5482-11D5-B883-BAF65FD25A91}\0
// Helpfile:
// DepndLst:
//     (1) v2.0 stdole, (C:\WINDOWS\SYSTEM\STDOLE2.TLB)
//     (2) v4.0 StdVCL, (C:\WINDOWS\SYSTEM\STDVCL40.DLL)
// ************************************************************ //
{14TYPEDADDRESS OFF} // Unit must be compiled without type-checked pointers.
interface

uses Windows, ActiveX, Classes, Graphics, OleServer, OleCtrls, StdVCL;

// ************************************************************//
// GUIDS declared in the TypeLibrary. Following prefixes are used:
//     Type Libraries         : LIBID_xxxx
//     CoClasses                : CLASS_xxxx
//     DISPInterfaces         : DIID_xxxx
//     Non-DISP interfaces: IID_xxxx
// ************************************************************//
const
        // TypeLibrary Major and minor versions
        PanelXControl1MajorVersion = 1;
        PanelXControl1MinorVersion = 0;

        LIBID_PanelXControl1: TGUID = '{D2A666F8-5482-11D5-B883-BAF65FD25A91}';

        IID_IPanelX: TGUID = '{D2A666F9-5482-11D5-B883-BAF65FD25A91}';
        DIID_IPanelXEvents: TGUID = '{D2A666FB-5482-11D5-B883-BAF65FD25A91}';
        CLASS_PanelX: TGUID = '{D2A666FD-5482-11D5-B883-BAF65FD25A91}';

// ************************************************************//
// Declaration of Enumerations defined in Type Library
// ************************************************************//
// Constants for enum TxAlignment
type
        TxAlignment = TOleEnum;
const
        taLeftJustify = 1400000000;
        taRightJustify = 1400000001;
        taCenter = 1400000002;

// Constants for enum TxBevelCut
type
        TxBevelCut = TOleEnum;
```

```
const
    bvNone = 1400000000;
    bvLowered = 1400000001;
    bvRaised = 1400000002;
    bvSpace = 1400000003;

// Constants for enum TxBorderStyle
type
    TxBorderStyle = TOleEnum;
const
    bsNone = 1400000000;
    bsSingle = 1400000001;

// Constants for enum TxDragMode
type
    TxDragMode = TOleEnum;
const
    dmManual = 1400000000;
    dmAutomatic = 1400000001;
// Constants for enum TxMouseButton
type
    TxMouseButton = TOleEnum;
const
    mbLeft = 1400000000;
    mbRight = 1400000001;
    mbMiddle = 1400000002;

type

// *********************************************************//
// Forward declaration of types defined in TypeLibrary
// *********************************************************//
    IPanelX = interface;
    IPanelXDisp = dispinterface;
    IPanelXEvents = dispinterface;

// *********************************************************//
// Declaration of CoClasses defined in Type Library
// (NOTE: Here we map each CoClass to its Default Interface)
// *********************************************************//
    PanelX = IPanelX;

// *********************************************************//
```

```
// Declaration of structures, unions and aliases.
// ************************************************************//
    PPUserType1 = ^IFontDisp; {*}

// ************************************************************//
// Interface: IPanelX
// Flags:     (4416) Dual OleAutomation Dispatchable
// GUID:      {D2A666F9-5482-11D5-B883-BAF65FD25A91}
// ************************************************************//
    IPanelX = interface(IDispatch)
        ['{D2A666F9-5482-11D5-B883-BAF65FD25A91}']
        function    Get_Alignment: TxAlignment; safecall;
        procedure Set_Alignment(Value: TxAlignment); safecall;
        function    Get_AutoSize: WordBool; safecall;
        procedure Set_AutoSize(Value: WordBool); safecall;
        function    Get_BevelInner: TxBevelCut; safecall;
        procedure Set_BevelInner(Value: TxBevelCut); safecall;
        function    Get_BevelOuter: TxBevelCut; safecall;
        procedure Set_BevelOuter(Value: TxBevelCut); safecall;
        function    Get_BorderStyle: TxBorderStyle; safecall;
        procedure Set_BorderStyle(Value: TxBorderStyle); safecall;
        function    Get_Caption: WideString; safecall;
        procedure Set_Caption(const Value: WideString); safecall;
        function    Get_Color: OLE_COLOR; safecall;
        procedure Set_Color(Value: OLE_COLOR); safecall;
        function    Get_Ctl3D: WordBool; safecall;
        procedure Set_Ctl3D(Value: WordBool); safecall;
        function    Get_UseDockManager: WordBool; safecall;
        procedure Set_UseDockManager(Value: WordBool); safecall;
        function    Get_DockSite: WordBool; safecall;
        procedure Set_DockSite(Value: WordBool); safecall;
        function    Get_DragCursor: Smallint; safecall;
        procedure Set_DragCursor(Value: Smallint); safecall;
        function    Get_DragMode: TxDragMode; safecall;
        procedure Set_DragMode(Value: TxDragMode); safecall;
        function    Get_Enabled: WordBool; safecall;
        procedure Set_Enabled(Value: WordBool); safecall;
        function    Get_FullRepaint: WordBool; safecall;
        procedure Set_FullRepaint(Value: WordBool); safecall;
        function    Get_Font: IFontDisp; safecall;
        procedure _Set_Font(const Value: IFontDisp); safecall;
        procedure Set_Font(var Value: IFontDisp); safecall;
        function    Get_Locked: WordBool; safecall;
```

```
        procedure Set_Locked(Value: WordBool); safecall;
        function  Get_ParentColor: WordBool; safecall;
        procedure Set_ParentColor(Value: WordBool); safecall;
        function  Get_ParentCtl3D: WordBool; safecall;
        procedure Set_ParentCtl3D(Value: WordBool); safecall;
        function  Get_Visible: WordBool; safecall;
        procedure Set_Visible(Value: WordBool); safecall;
        function  Get_DoubleBuffered: WordBool; safecall;
        procedure Set_DoubleBuffered(Value: WordBool); safecall;
        function  Get_VisibleDockClientCount: Integer; safecall;
        function  Get_Cursor: Smallint; safecall;
        procedure Set_Cursor(Value: Smallint); safecall;
        property Alignment: TxAlignment read Get_Alignment write Set_Alignment;
        property AutoSize: WordBool read Get_AutoSize write Set_AutoSize;
        property BevelInner: TxBevelCut read Get_BevelInner write Set_BevelInner;
        property BevelOuter: TxBevelCut read Get_BevelOuter write Set_BevelOuter;
        property BorderStyle: TxBorderStyle read Get_BorderStyle
      write Set_BorderStyle;
        property Caption: WideString read Get_Caption write Set_Caption;
        property Color: OLE_COLOR read Get_Color write Set_Color;
        property Ctl3D: WordBool read Get_Ctl3D write Set_Ctl3D;
        property UseDockManager: WordBool read Get_UseDockManager
      write Set_UseDockManager;
        property DockSite: WordBool read Get_DockSite write Set_DockSite;
        property DragCursor: Smallint read Get_DragCursor write Set_DragCursor;
        property DragMode: TxDragMode read Get_DragMode write Set_DragMode;
        property Enabled: WordBool read Get_Enabled write Set_Enabled;
        property FullRepaint: WordBool read Get_FullRepaint
      write Set_FullRepaint;
        property Font: IFontDisp read Get_Font write _Set_Font;
        property Locked: WordBool read Get_Locked write Set_Locked;
        property ParentColor: WordBool read Get_ParentColor
      write Set_ParentColor;
        property ParentCtl3D: WordBool read Get_ParentCtl3D
      write Set_ParentCtl3D;
        property Visible: WordBool read Get_Visible write Set_Visible;
        property DoubleBuffered: WordBool read Get_DoubleBuffered
      write Set_DoubleBuffered;
        property VisibleDockClientCount: Integer
  read Get_VisibleDockClientCount;
        property Cursor: Smallint read Get_Cursor write Set_Cursor;
    end;
```

```
// ******************************************************//
// DispIntf:   IPanelXDisp
// Flags:       (4416) Dual OleAutomation Dispatchable
// GUID:          {D2A666F9-5482-11D5-B883-BAF65FD25A91}
// ******************************************************//
      IPanelXDisp = dispinterface
            ['{D2A666F9-5482-11D5-B883-BAF65FD25A91}']
            property Alignment: TxAlignment dispid 1;
            property AutoSize: WordBool dispid 2;
            property BevelInner: TxBevelCut dispid 3;
            property BevelOuter: TxBevelCut dispid 4;
            property BorderStyle: TxBorderStyle dispid 5;
            property Caption: WideString dispid -518;
            property Color: OLE_COLOR dispid -501;
            property Ctl3D: WordBool dispid 6;
            property UseDockManager: WordBool dispid 7;
            property DockSite: WordBool dispid 8;
            property DragCursor: Smallint dispid 9;
            property DragMode: TxDragMode dispid 10;
            property Enabled: WordBool dispid -514;
            property FullRepaint: WordBool dispid 11;
            property Font: IFontDisp dispid -512;
            property Locked: WordBool dispid 12;
            property ParentColor: WordBool dispid 13;
            property ParentCtl3D: WordBool dispid 14;
            property Visible: WordBool dispid 15;
            property DoubleBuffered: WordBool dispid 16;
            property VisibleDockClientCount: Integer readonly dispid 17;
            property Cursor: Smallint dispid 26;
      end;

// ******************************************************//
// DispIntf:   IPanelXEvents
// Flags:       (0)
// GUID:          {D2A666FB-5482-11D5-B883-BAF65FD25A91}
// ******************************************************//
      IPanelXEvents = dispinterface
            ['{D2A666FB-5482-11D5-B883-BAF65FD25A91}']
            procedure OnCanResize(var NewWidth: Integer; var NewHeight: Integer; var Resize:
   WordBool); dispid 1;
            procedure OnClick; dispid 2;
            procedure OnConstrainedResize(var MinWidth: Integer;
        var MinHeight: Integer; var MaxWidth: Integer;
```

```
        var MaxHeight: Integer); dispid 3;
            procedure OnDblClick; dispid 7;
            procedure OnResize; dispid 16;
        end;

// ******************************************************************//
// OLE Control Proxy class declaration
// Control Name         : TPanelX
// Help String          : PanelX Control
// Default Interface: IPanelX
// Def. Intf. DISP? : No
// Event     Interface: IPanelXEvents
// TypeFlags            : (34) CanCreate Control
// ******************************************************************//
    TPanelXOnCanResize = procedure(Sender: TObject; var NewWidth: Integer;
var NewHeight: Integer; var Resize: WordBool) of object;
    TPanelXOnConstrainedResize = procedure(Sender: TObject; var MinWidth:
Integer; var MinHeight: Integer; var MaxWidth: Integer;
var MaxHeight: Integer) of object;

    TPanelX = class(TOleControl)
    private
        FOnCanResize: TPanelXOnCanResize;
        FOnClick: TNotifyEvent;
        FOnConstrainedResize: TPanelXOnConstrainedResize;
        FOnDblClick: TNotifyEvent;
        FOnResize: TNotifyEvent;
        FIntf: IPanelX;
        function    GetControlInterface: IPanelX;
    protected
        procedure CreateControl;
        procedure InitControlData; override;
    public
        property    ControlInterface: IPanelX read GetControlInterface;
        property    DefaultInterface: IPanelX read GetControlInterface;
        property DoubleBuffered: WordBool index 16 read GetWordBoolProp
      write SetWordBoolProp;
        property VisibleDockClientCount: Integer index 17 read GetIntegerProp;
  published
        property Alignment: TOleEnum index 1 read GetTOleEnumProp
      write SetTOleEnumProp stored False;
        property AutoSize: WordBool index 2 read GetWordBoolProp
      write SetWordBoolProp stored False;
```

```
        property BevelInner: TOleEnum index 3 read GetTOleEnumProp
write SetTOleEnumProp stored False;
        property BevelOuter: TOleEnum index 4 read GetTOleEnumProp
write SetTOleEnumProp stored False;
        property BorderStyle: TOleEnum index 5 read GetTOleEnumProp
write SetTOleEnumProp stored False;
        property Caption: WideString index -518 read GetWideStringProp
write SetWideStringProp stored False;
        property Color: TColor index -501 read GetTColorProp
write SetTColorProp stored False;
        property Ctl3D: WordBool index 6 read GetWordBoolProp
write SetWordBoolProp stored False;
        property UseDockManager: WordBool index 7 read GetWordBoolProp
write SetWordBoolProp stored False;
        property DockSite: WordBool index 8 read GetWordBoolProp
write SetWordBoolProp stored False;
        property DragCursor: Smallint index 9 read GetSmallintProp
write SetSmallintProp stored False;
        property DragMode: TOleEnum index 10 read GetTOleEnumProp
write SetTOleEnumProp stored False;
        property Enabled: WordBool index -514 read GetWordBoolProp
write SetWordBoolProp stored False;
        property FullRepaint: WordBool index 11 read GetWordBoolProp
write SetWordBoolProp stored False;
        property Font: TFont index -512 read GetTFontProp
write SetTFontProp stored False;
        property Locked: WordBool index 12 read GetWordBoolProp
write SetWordBoolProp stored False;
        property ParentColor: WordBool index 13 read GetWordBoolProp
write SetWordBoolProp stored False;
        property ParentCtl3D: WordBool index 14 read GetWordBoolProp
write SetWordBoolProp stored False;
        property Visible: WordBool index 15 read GetWordBoolProp
write SetWordBoolProp stored False;
        property Cursor: Smallint index 26 read GetSmallintProp
write SetSmallintProp stored False;
        property OnCanResize: TPanelXOnCanResize read FOnCanResize
write FOnCanResize;
        property OnClick: TNotifyEvent read FOnClick write FOnClick;
        property OnConstrainedResize: TPanelXOnConstrainedResize
read FOnConstrainedResize write FOnConstrainedResize;
        property OnDblClick: TNotifyEvent read FOnDblClick write FOnDblClick;
        property OnResize: TNotifyEvent read FOnResize write FOnResize;
```

```
    end;

procedure Register;

implementation

uses ComObj;

procedure TPanelX.InitControlData;
const
    CEventDispIDs: array [0..4] of DWORD = (
        1400000001, 1400000002, 1400000003, 1400000007, 1400000010);
    CTFontIDs: array [0..0] of DWORD = (
        14FFFFFE00);
    CControlData: TControlData2 = (
        ClassID: '{D2A666FD-5482-11D5-B883-BAF65FD25A91}';
        EventIID: '{D2A666FB-5482-11D5-B883-BAF65FD25A91}';
        EventCount: 5;
        EventDispIDs: @CEventDispIDs;
        LicenseKey: nil (*HR:1480040154*);
        Flags: 140000001D;
        Version: 401;
        FontCount: 1;
        FontIDs: @CTFontIDs);
begin
    ControlData := @CControlData;
    TControlData2(CControlData).FirstEventOfs :=
Cardinal(@@FOnCanResize) - Cardinal(Self);
end;

procedure TPanelX.CreateControl;

    procedure DoCreate;
    begin
        FIntf := IUnknown(OleObject) as IPanelX;
    end;
begin
    if FIntf = nil then DoCreate;
end;

function TPanelX.GetControlInterface: IPanelX;
begin
    CreateControl;
```

```
        Result := FIntf;
end;

procedure Register;
begin
        RegisterComponents('ActiveX',[TPanelX]);
end;

end.
```

从上面的代码清单中，会发现除了类型库的信息以外，还有一个 TPanelX 类，它是从 TOleControl 继承下来的。这样，就可以把这个 ActiveX 控件加到组件选项板上了。具体操作有很多种，可以先保存这个 ActiveX 控件，然后通过 Project | Import Type Library 命令打开导入类型库的对话框，通过 Add 按钮找到保存过的.tlb 文件就可以将该 ActiveX 控件加到组件选项板上了。

有时候，过多的代码让人望而生畏，不过，如果仔细看的话，这里面并没有多么高深的知识。这时候，用户已经拥有了一个功能齐全的 ActiveX 控件，它包括了接口、类型库和事件。

上面的代码中包含了一些函数，它们把 IStrings 和 IFont 转换为 Delphi 中的 TStrings 和 TFont 类，这些函数为 Object Pascal 的类和 Automation 兼容的调度接口之间搭建了一个桥梁。表 14.1 中列出了一些 VCL 类及其对应的 Automation 接口。

表 14.1　VCL 类及其对应的 Automation 接口

VCL 类	Automation 接口
Tfont	Ifont
Tpicture	Ipicture
Tstrings	Istrings

14.2.3　ActiveX 框架

Delphi 的 ActiveX 框架（Delphi ActiveX FrameWork、DAX）位于 AxCtrls 单元中。一个 ActiveX 控件可以被描述为一个 Automation 对象，因为 ActiveX 控件也必须实现 IDispatch 接口。因此，DAX 类似于 Automation 对象的框架。TActiveXControl 就是从 TAutoObject 继承下来的。DAX 是一个双向的框架，借此 ActiveX 控件与 VCL 控件彼此间可以进行通讯。

就像所有的 COM 对象一样，ActiveX 控件也必须由类工厂来创建。DAX 中的 TActiveXControlFactory 就是 TActiveXControl 对象的类工厂，而类工厂本身的实例就是在控件的实现单元的 Initialization 部分创建的。类工厂是这样创建的，如下所示：

```
constructor TActiveXControlFactory.Create(ComServer: TComServerObject;
        ActiveXControlClass: TActiveXControlClass;
        WinControlClass: TWinControlClass; const ClassID: TGUID;
```

```
        ToolboxBitmapID: Integer; const LicStr: string; MiscStatus: Integer;
        ThreadingModel: TThreadingModel);
begin
        FWinControlClass := WinControlClass;
        inherited Create(ComServer, ActiveXControlClass,
ClassID, ciMultiInstance, ThreadingModel);
        FMiscStatus := MiscStatus or
                OLEMISC_RECOMPOSEONRESIZE or
                OLEMISC_CANTLINKINSIDE or
                OLEMISC_INSIDEOUT or
                OLEMISC_ACTIVATEWHENVISIBLE or
                OLEMISC_SETCLIENTSITEFIRST;
        FToolboxBitmapID := ToolboxBitmapID;
        FVerbs := TStringList.Create;
        AddVerb(OLEIVERB_PRIMARY, SPropertiesVerb);
        LicString := LicStr;
        SupportsLicensing := LicStr <> '';
        FLicFileStrings := TStringList.Create;
end;
```

其中：

- ComServer **参数用于指定** TComServer **的实例，它通常设为全局变量** ComServer。
- ActiveXControlClass **参数用于指定** ActiveX **控件的类，它一定是从** TActiveXControl **继承下来的。**
- WinControlClass **参数用于指定** ActiveX **控件的类，它一定是从** TWinControl **类继承下来的。**
- ClassID **参数用于指定** ActiveX **控件的组件类的类标识符，即** CLSID。
- ToolboxBitmapID **参数用于指定一个位图的资源标识符，这个位图用于作为** ActiveX **控件被安装到组件选项板上后的图标。**
- LicStr **参数用于指定** ActiveX **控件的许可字符串，如果这个字符串为空，表示不许可。**
- MiscStatus **参数用于指定** ActiveX **控件的** OLEMISC_XXX **状态标志。这些标志决定了一个** ActiveX **控件怎样在窗口上画出、是否能够包含其他控件。**
- ThreadingModel **参数用于指定** ActiveX **控件的线程模式。这个参数只是设置线程模式，并不会保证** ActiveX **控件在该线程模式下一定是安全的。**

有些 VCL **控件需要处理消息才能正常工作，相应的，在** ActiveX **控件中，**DAX **可以创建一个反射窗口来接受消息，然后把它们反射给** VCL **控件。凡是需要反射窗口的标准** VCL **控件的** ControlStyle **特性都包含了** csReflector **标志。对于自定义的** TWinControl **控件来说，如果需要的话，可以把这个标志加到它的** ControlStyle **特性中。**

VCL **控件可以让用户知道一个控件现在是在设计期还是在运行期，只要检查它的**

ComponentState 特性，看它是否包含 csDesigning 标志即可。而对于 ActiveX 控件来说，就比较麻烦了。

首先，用户要获取一个指向容器的 IAmbientDispatch 接口的指针，然后访问这个接口的 UserMode 特性。例如下面的代码：

```
Function IsControlRunning(Control: IUnknown): Boolean;
var
        OleObj: IOleObject;
        Site: IOleClientSite;
begin
        Result := True;
        if (Control.QueryInterface(IOleObject, OleObj) = S_OK) and
                (OleObj.GetClientSite(Site) = S_OK) and (Site <> nil) then
            Result := (Site as IAmbientDispatch).UserMode;
        //  先获取控件的 IOleObject 接口，再由此获得容器的 IOleClientSite 接口，进而再获取//
IAmbientDispatch 接口
end;
```

14.3　ActiveX 控件在 Web 上的应用

ActiveForm 的真正用途是在 WWW 上作为载体来传递较小的应用程序。ActiveX 控件也能增强 Web 网页的美观和有效性。不过，要充分发挥 ActiveX 控件在 Web 上的作用，除了 ActiveX 控件以外，还需要知道如何与 Web 浏览器进行交互。

14.3.1　与 Web 浏览器交互

ActiveX 控件可以在 Web 浏览器的环境中运行，那么如果 ActiveX 控件能够调用 Web 浏览器的函数和接口，这将是非常有意义的一件事情。Web 浏览器的函数和接口大部分是在 UrlMon 单元中定义的，其中比较简单的函数例如 HLinkXXX()，它用于使 Web 浏览器跳转到一个超级链接所指定的位置。例如，HLinkGoForward()和 HLinkGoBack()函数分别可以使浏览器在历史记录中向前或向后跳转。HLinkNavigateString()函数能够使浏览器跳转到一个特定的 URL。

14.3.2　Web 分发

Delphi 的 IDE 可以很方便地在 Web 上分发 ActiveX 控件，其中包含 ActiveX Form。可以使用 Project | Web Deployment Options 命令来设置有关的选项，如图 14.4 所示。

- 在 Project 选项卡上，Target Dir 文本框用于指定 ActiveX 控件要分发的目标路径。
- Target URL 文本框用于输入 Target Dir 文本框中所指定的路径的 URL。合法的 URL 必须以 http://、file://、ftp://等为前缀。URL 中不能包含文件名。
- HTML Dir 文本框用于指定一个路径，Delphi 将生成一个测试用的 HTML 文档放在制定的路径中。HTML Dir 文本框指定的路径通常与 Target Dir 文本框指定的路

径相同。

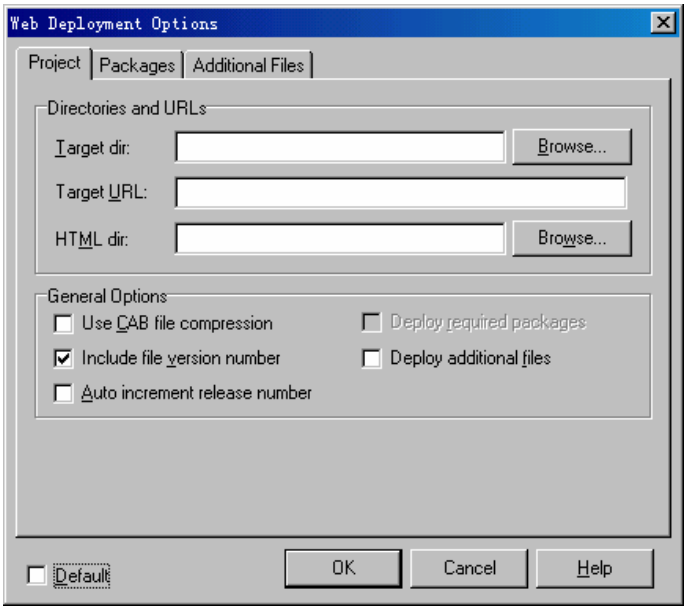

图 14.4 Web Deployment Options 对话框

除此以外，还可以设置下列选项：

- Use CAB File Compression：如果选中这个复选框，将把 OCX 文件以 Microsoft Cabine（CAB）格式压缩。在低带宽的网络上最好选中这个选项。
- Include File Version Number：生成的 HTML 文件或者 INF 文件中将包含版本号，这样，用户就可以避免下载他已经有的版本。
- Auto Increment Release Number：版本信息中的发布号将自动增大。
- Code Sign Project：如果用户有数字签名的授权证书，可以让 IDE 自动给 ActiveX 控件加上数字签名。
- Deploy Required Packages：如果 ActiveX 控件使用了包，发布 ActiveX 控件时将自动包括所需要的包。
- Deploy Additional Files：如果还需要其他文件一起发布，就选中这个复选框。

Packages 选项卡和 Additional Files 选项卡如图 14.5 和 14.6 所示。

以上两个选项卡的主要区别在于，Packages 选项卡已经列出了 ActiveX 项目中所使用的包，而 Additional Files 选项卡需要用户自己指定文件。

如果 Project 选项卡上选中了 Use CAB file compression 复选框，Packages 选项卡和 Additional Files 选项卡上的 CAB Options 选项组就可以让用户选择是把文件与 OCX 一起打包还是单独打包。单独打包的好处是，用户不必每次都把文件全部下载。此外，还可以设置下面的选项：

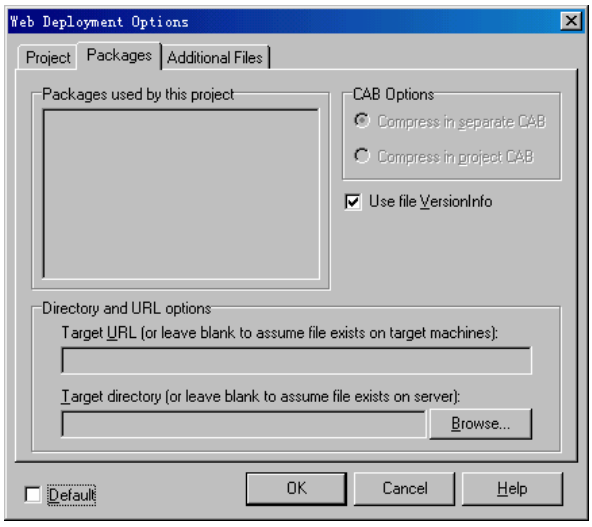

图 14.5　Packages 选项卡

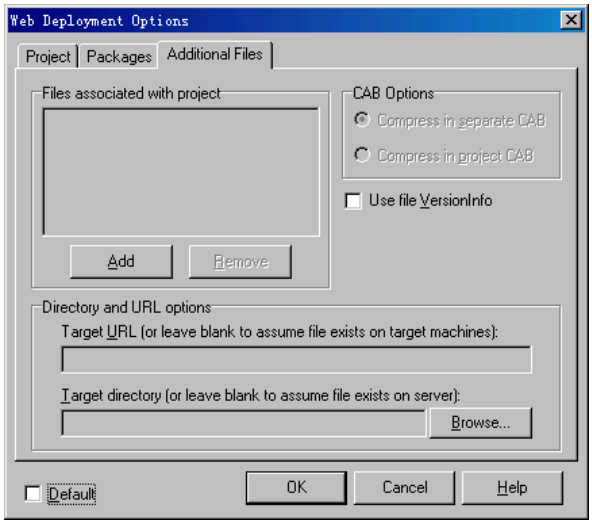

图 14.6　Additional Files 选项卡

- 如果单击 Use File VersionInfo 复选框，Delphi 将检查这些文件是否包含版本信息。如果有的话，就把它们的版本信息加到 INF 文件中。
- Target URL 文本框一般就是 Project 选项卡的 Target URL 文本框输入的内容，即文件可以被下载的 URL。如果可以保证用户已经安装了这个文件，可以空着这个文本框。
- Target Directory 文本框用于指定文件将复制到服务器的哪个路径中。如果这个文本框空着，这个文件将不被复制到服务器上。

14.4 本 章 小 结

本章介绍了如何在 Delphi **中创建和使用** ActiveX **控件。在这里揭示了** ActiveX **控件向导的奥妙，以帮助用户在** DAX **框架内工作。**

第 15 章　文 件 处 理

程序中经常会碰到处理文件、目录、驱动器的事情，本章中将讲述如何处理各种不同类型的文件，例如文本文件、有类型和无类型的文件。本章的内容涵盖了如何使用 TFileStream 封装文件的输入输出，以及如何利用 32 位 Windows 的最佳特征——内存映射文件，并利用它在文件中进行文本查询。本章还提供了一些非常有用的例程，介绍了关于选择可用驱动器、在树型目录中查询文件以及获取文件版本信息的方法。通过本章的学习，读者将对 Delphi 的文件、目录和驱动器处理功能得到充分的认识。

15.1　文件的输入/输出

需要处理的文件一般有 3 种：文本文件、有类型文件和二进制文件。下面几节将讲述这些文件的输入和输出处理。

文本文件顾名思义，就是包含了可被任意文本编辑器读取的 ASCII 文本。有类型文件是包含了程序员所定义的数据类型的文件。二进制文件则包含了其他所有的类型，它是对包含任意格式或者无格式数据的文件的通称。

15.1.1　文本文件

这一节中将介绍利用内建于 Obejct Pascal Run Time Library（运行时库）中的过程和函数来处理文本文件的方法。在对文本文件进行任何操作之前，先要打开这个文本文件。首先，必须声明一个变量，如下所示：

```
Var
    MyTextFile: TextFile;
```

使用这个变量来引用一个文本文件。

要打开一个文件，需要函数 AssignFile()，它的作用是将文件变量与一个文件联系在一起，如下所示：

```
AssignFile(MyTextFile, 'c:\MyTextFile.txt');
```

现在就可以打开这个文件了。有 3 种方法可以来打开文件：

- ReWrite()：创建并打开一个文件。如果该文件已经存在，它将被重新覆盖。
- ReSet()：以只读方式打开一个文件。
- Append()：向已存在的文件追加文本。

要关闭一个开启的文件，可以调用 CloseFile()过程。

下面的示例以只读方式打开一个文件：

```
procedure TForm1.btnReadOnlyClick(Sender: TObject);
var
    MyTextFile: TextFile;
begin
    AssignFile(MyTextFile, 'c:\MyTextFile.txt');
    Reset(MyTextFile);
    try
        ...
    finally
        CloseFile(MyTextFile);
    end;
end;
```

下面的示例创建了一个新的文件：

```
procedure TForm1.btnCreateNewFileClick(Sender: TObject);
var
    MyTextFile: TextFile;
begin
    AssignFile(MyTextFile, 'c:\MyTextFile.txt');
    Rewrite(MyTextFile);
    try
        ...
    finally
        CloseFile(MyTextFile);
    end;
end;
```

下面的示例向一个已存在的文件中追加新文本：

```
procedure TForm1.btnAppendTextClick(Sender: TObject);
var
    MyTextFile: TextFile;
begin
    AssignFile(MyTextFile, 'c:\MyTextFile.txt');
    Append(MyTextFile);
    try
        ...
    finally
        CloseFile(MyTextFile);
    end;
end;
```

下面的示例演示了如何使用 Rewrite()函数创建并添加 5 行文本到一个文件中：

```
procedure TForm1.btnRewriteClick(Sender: TObject);
var
    MyTextFile: TextFile;
    S: string;
    i: integer;
begin
    AssignFile(MyTextFile, 'C:\MyTextFile.txt');
    Rewrite(MyTextFile);
    try
        for i := 1 to 5 do begin
          S := 'This is the line #';
          Writeln(MyTextFile, S, i);
        end;
    finally
        CloseFile(MyTextFile);
    end;
end;
```

该程序运行结束后，文件中将包含图 15.1 所示的文本。

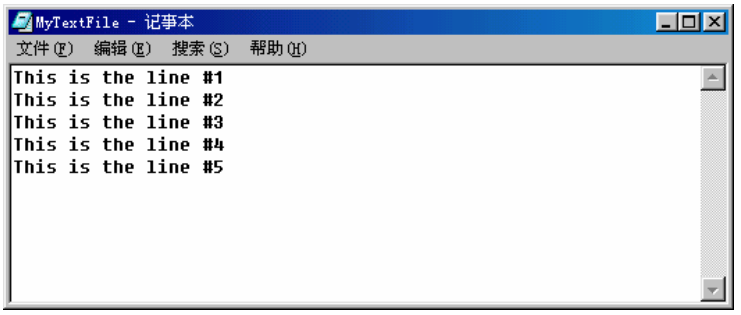

图 15.1　添加文本后的文件

后面的代码将向上面创建的文件中添加更多的文本行，如下所示：

```
procedure TForm1.btnAddMoreClick(Sender: TObject);
var
    MyTextFile: TextFile;
    S: string;
    i: integer;
begin
    AssignFile(MyTextFile, 'C:\MyTextFile.txt');
    Append(MyTextFile);
    try
        for i := 6 to 10 do begin
```

```
            S := 'This is the line #';
            Writeln(MyTextFile, S, i);
        end;
    finally
        CloseFile(MyTextFile);
    end;
end;
```

上面的两个程序段都可以向文件中写入字符串和整数。实际上，Object Pascal **对于所有的数据类型都是可以这样做的。下面的代码将读取文件中的文本：**

```
procedure TForm1.btnReadTextClick(Sender: TObject);
var
    MyTextFile: TextFile;
    S: string[15];
    i, j: integer;
begin
    AssignFile(MyTextFile, 'C:\MyTextFile.txt');
    Reset(MyTextFile);
    try
        while not Eof(MyTextFile) do begin
            Readln(MyTextFile, S, j);
            Memo1.Lines.Add(S + IntToStr(j));
        end;
    finally
        CloseFile(MyTextFile);
    end;
end;
```

上面的代码中，变量“S**”声明为一个数组** String[15]**，这样可避免将文件中的所有行都读入变量“**S**”中；否则，把一个值读入整数变量“**j**”时将会出错。同时，这里用到了** Eof()**函数，它的作用是检查文件的指针是否处在文件中的末尾，如果是，则停止循环，因为此时已无文本可读。**

15.1.2 处理有类型文件

可以把 Object Pascal **的数据结构保存到磁盘文件中，也可以将数据从文件中直接读入数据结构中，就好像这些数据是表中的记录一样。保存有** Pascal **数据结构的文件称为基于记录的文件。关于这类文件的使用，可以参考下面声明的记录，如下所示：**

```
TPersonRec = packed record
    FirstName: String[20];
    LastName: String[20];
    MI: String[1];
    Birthday: TDateTime;
```

```
    Age: Integer;
end;
```

注意：包含有 AnsiString、变量、类实例、接口或者动态数组的记录不能写入文件中。

现在要在一个文件中保存一个或者多个这样的记录，可以通过声明一个记录来实现，如下所示：

```
DataFile: File of TPersonRec;
```

要读取单独一条 TPersonRec 类型的记录，可以参考下面的代码：

```
procedure TForm1.Button1Click(Sender: TObject);
var
    PersonRec: TPersonRec;
    DataFile: file of TPersonRec;
begin
    AssignFile(DataFile, 'c:\PersonFile.Dat');
    Reset(DataFile);
    try
        if not Eof(DataFile) then
          Read(DataFile, PersonRec);
    finally
        CloseFile(DataFile);
  end;
end;
```

下面的代码演示了如何向一个文件中添加一条记录：

```
procedure TForm1.Button2Click(Sender: TObject);
var
    PersonRec: TPersonRec;
    DataFile: file of TPersonRec;
begin
    AssignFile(DataFile, 'c:\PersonFile.Dat');
    Reset(DataFile);
    Seek(DataFile, FileSize(DataFile));
    try
        if not Eof(DataFile) then
          Write(DataFile, PersonRec);
    finally
        CloseFile(DataFile);
    end;
end;
```

在写记录之前调用了 Seek()过程，使文件的当前位置处于末尾。

为了演示有类型文件的用法，下面创建一个简单的应用程序，这个程序以 Object Pascal 格式保存了一些个人的信息，允许用户浏览、添加和编辑其中的记录。

这里，首先要声明一个 TFileStream 的派生类，它的作用是操纵文件的输入和输出。TFileStream 是一种可用来存储非对象内容的流。记录本身没有提供将自身保存到磁盘或内存的方法，所以这里就利用了 TFileStream 类的存储功能来保存记录。

程序中声明了一个 TPersonRec 的记录和一个 TFileStream 的派生类 TRecordStream，用来处理文件的输入和输出，如下所示：

```
unit Un_PersonRec;

interface

uses Classes, Dialogs, sysutils;

type
    TPersonRec = packed record
        FirstName: string[20];
        LastName: string[20];
        MI: string[1];
        Birthday: TDateTime;
        Age: Integer;
    end;

    TRecordStream = class(TFileStream)
    private
        function GetRecsCount: LongInt;
        function GetCurRec: LongInt;
        procedure SetCurrec(RecNo: LongInt);
    protected
        function GetRecSize: Longint; virtual;
    public
        function SeekRec(RecNo: Longint; Origin: Word): Longint;
        function WriteRec(const Rec): Longint;
        function AppendRec(const Rec): Longint;
        function ReadRec(var Rec): Longint;
        procedure First;
        procedure Last;
        procedure NextRec;
        procedure previousRec;

        property RecsCount: Longint read GetRecsCount;
        property CurRec: Longint read GetCurRec write SetCurRec;
```

```
    end;

implementation

{ TRecordStream }

function TRecordStream.GetCurRec: LongInt;
begin
    Result := (Position div GetRecSize) + 1;
end;

function TRecordStream.GetRecsCount: LongInt;
begin
    Result := Size div GetRecSize;
end;

function TRecordStream.GetRecSize: Longint;
begin
    // 流中的记录数目
    Result := Size div GetRecSize;
end;

function TRecordStream.AppendRec(const Rec): Longint;
begin
    // 将记录写入流中
    Seek(0, 2);
    Result := Write(Rec, GetRecSize);
end;

procedure TRecordStream.First;
begin
    Seek(0, 0);
end;

procedure TRecordStream.Last;
begin
    Seek(0, 2);
    Seek(-GetRecSize, 1);
end;

procedure TRecordStream.NextRec;
begin
    if ((Position + GetRecSize) div GetRecSize) = GetRecsCount then
```

```
            raise Exception.Create('超出文件界限')
        else
            Seek(GetRecSize, 1);
end;

procedure TRecordStream.previousRec;
begin
        if (Position - GetRecSize >= 0) then
            Seek(-GetRecSize, 1)
        else
            raise Exception.Create('超出文件界限');
end;

function TRecordStream.ReadRec(var Rec): Longint;
begin
        Result := Read(Rec, GetRecSize);
        Seek(-GetRecSize, 1);
end;

function TRecordStream.SeekRec(RecNo: Integer; Origin: Word): Longint;
begin
        Result := Seek(RecNo * GetRecSize, Origin);
      end;

procedure TRecordStream.SetCurrec(RecNo: Integer);
begin
        if RecNo > 0 then
            Position := (RecNo - 1) * GetRecSize
        else
            raise Exception.Create('文件超出界限');
end;

function TRecordStream.WriteRec(const Rec): Longint;
begin
        Result := Write(REc, GetRecSize);
end;

end.
```

上面的程序中，首先声明了一个 TPersonRec **记录，然后声明了一个** TRecordStream **类型。**TRecordStream **类是从** TFileStream **继承下来的，用来为** TPersonRec **进行文件的输入和输出。**TRecordStream **有两个特性，其中** RecsCount **特性用来表示流中记录的数量，**CurRec **特性表示当前流中正在查看的记录。**

GetRecsCount()方法是 RecsCount 特性的访问方法，用于得到流中记录的数量。这个方法实际上是把从 TStream.Size 特性获取的流的字节数除以 TPersonRec 记录的字节数。

注意：TPersonRec 类型必须声明为 Packed Record，这是因为，为了便于访问，记录和数组这样的结构化类型，内存中都是按照字或者双字排列的，这意味着，将消耗比实际需要更多的磁盘空间，只有 Packed 以后，就可以保证数据是紧凑存储的。否则，将导致 GetRecsCount 方法得到不正确的结果。

GetCurRec()方法用于判断当前的记录。这是通过 TStream.Position 特性除以 TPersonRec 记录的字节数来获得的。SetCurRec()方法可将文件指针置于 RecNo 特性所指的位置。

SeekRec()方法用于将文件指针置于 RecNo 和 Origin 参数所指的位置，这个方法可以将文件指针前后移动。它实际上是调用了 TStream 的 Seek()方法。

WriteRec()方法永远在文件的当前位置写入 TPersonRec 的内容，而当前位置已有的记录则将被覆盖。AppendRec()用于在文件的末尾添加新的记录。

ReadRec()方法用于从流中将数据读入 TPersonRec 记录，然后再调用 Seek()方法将文件指针置于该记录的起始位置。

First()和 Last()方法分别用于将文件指针置于文件的开始和末尾处。

下面是一个应用程序的示例，它使用了上面的 TRecordStream 类型，实现了浏览文件中的记录的功能，如下所示：

```
unit Un_Main;

interface

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

type
    Tfm_Main = class(TForm)
        leFirstName: TLabeledEdit;
        leLastName: TLabeledEdit;
        leMI: TLabeledEdit;
        leAge: TLabeledEdit;
        lbBirthday: TLabel;
        dtBirthday: TDateTimePicker;
        btnFirst: TButton;
        btnLast: TButton;
        btnPrev: TButton;
        btnNext: TButton;
        btnAppend: TButton;
        btnUpdate: TButton;
```

```
            procedure FormCreate(Sender: TObject);
            procedure FormDestroy(Sender: TObject);
            procedure FormShow(Sender: TObject);
            procedure btnFirstClick(Sender: TObject);
            procedure btnLastClick(Sender: TObject);
            procedure btnPrevClick(Sender: TObject);
            procedure btnNextClick(Sender: TObject);
            procedure btnAppendClick(Sender: TObject);

        private
            { Private declarations }
        public
            { Public declarations }
            RecordStream: TRecordStream;
            PersonRec: TPersonRec;
            procedure ShowCurrentRecord;
        end;

var
  fm_Main: Tfm_Main;

implementation

{15R *.dfm}

procedure Tfm_Main.FormCreate(Sender: TObject);
begin
    if FileExists('c:\persons.dat') then
        RecordStream := TRecordStream.Create('c:\persons.dat', fmOpenReadWrite)
    else
        RecordStream := TRecordStream.Create('c:\persons.dat', fmCreate);
end;

procedure Tfm_Main.FormDestroy(Sender: TObject);
begin
    RecordStream.Free;
end;

procedure Tfm_Main.FormShow(Sender: TObject);
begin
    if RecordStream.RecsCount <> 0 then
        ShowCurrentRecord;
end;
```

```
procedure Tfm_Main.ShowCurrentRecord;
begin
        RecordStream.ReadRec(PersonRec);
        with PersonRec do begin
                leFirstName.Text := FirstName;
                leLastName.Text := LastName;
                leMI.Text := MI;
                leAge.Text := IntToStr(Age);
                dtBirthday.Date := Birthday;
        end;
end;

procedure Tfm_Main.btnFirstClick(Sender: TObject);
begin
        if RecordStream.RecsCount <> 0 then begin
                RecordStream.First;
                ShowCurrentRecord;
        end;
end;

procedure Tfm_Main.btnLastClick(Sender: TObject);
begin
        if RecordStream.RecsCount <> 0 then begin
                RecordStream.Last;
                ShowCurrentRecord;
        end;
end;

procedure Tfm_Main.btnPrevClick(Sender: TObject);
begin
        if RecordStream.RecsCount <> 0 then begin
                RecordStream.PreviousRec;
                ShowCurrentRecord;
        end;
end;

procedure Tfm_Main.btnNextClick(Sender: TObject);
begin
        if RecordStream.RecsCount <> 0 then begin
                RecordStream.NextRec;
                ShowCurrentRecord;
        end;
```

```
end;

procedure Tfm_Main.btnAppendClick(Sender: TObject);
begin
    with PersonRec do begin
        FirstName := leFirstName.Text;
        LastName := leLastName.Text;
        MI := leMI.Text;
        Birthday := dtBirthday.Date;
        Age := StrToInt(leAge.Text);
    end;
    RecordStream.AppendRec(PersonRec);
    ShowCurrentRecord;
end;

end.
```

图 15.2 **显示了上面的应用程序的主窗体。**

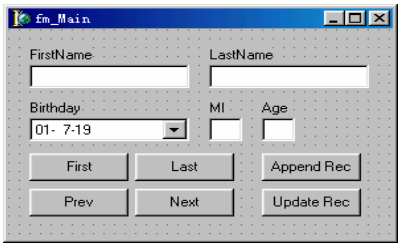

图 15.2 **主窗体**

程序的主单元中包含了一个 TPersonRec **类型和一个** TRecordStream **类。**TPersonRec **中保存着当前记录中的内容。**

ShowCurrentRecord()**通过调用** TRecordStream.ReadRec()**方法将当前记录从流中取出。它首先读出了整个记录，然后将指针置于记录的开头。**

这个程序演示了如何利用文件的输入输出功能，实现对有类型文件的简单数据库操作；同时，也演示了在文件中如何利用 TFileStream **对象来控制记录的输入和输出。**

15.1.3 处理无类型文件

本章已经介绍了如何对文本文件和有类型文件进行操作。文本文件用于存储 ASCII **字符序列，有类型文件用于存储符合** pascal **记录格式的数据，因此，这两种类型的文件所存储内容的字节数都可以通过代码来获得。**

与上面的文件不同，有些文件不遵从任何格式，例如 RTF **文件。尽管** RTF **文件也包含文本，但同时其中也包含了文本的属性。因此，不是任何一种文本编辑器都能够浏览** RTF **文件中的内容，而必须使用可以识别** RTF **文件格式的编辑器。下面就介绍如何操作无类型的文件。**

无类型文件可以这样声明：

```
var
UnTypedFile: File;
```

这里声明了一个含有数据块序列的文件，每个数据块含有 128 **字节的数据。**

要从无类型的文件中读取数据，可以调用 BlockRead()**方法，写数据可以调用** BlockWrite()**方法，这两个过程是这样定义的，如下所示：**

```
procedure BlockRead(var F: File; var Buf; Count: Integer
[; var AmtTransferred: Integer]);
procedure BlockWrite(var f: File; var Buf; Count: Integer
[; var AmtTransferred: Integer]);
```

这里至少要传递 3 **个参数：第一个参数是无类型文件的变量，第二个参数是缓冲区变量，用于存储读出或者写入的数据，第三个参数用于指定需要从文件中读出记录的数量。**Result **是可选的参数，它用于返回实际读取的记录数。如果** Result **参数和** Count **参数不一致，可能是因为磁盘空间不足了。**

下面的示例演示了如何从一个文件中读取出一条 128 **字节的记录，然后将读出的数据写到一个** Buffer **缓冲区中：**

```
procedure TForm1.btnReadClick(Sender: TObject);
var
        UnTypedFile: file;
        Buffer: array[0..128] of byte;
        NumRecsRead: Integer;
begin
        AssignFile(UnTypedFile, 'c:\myfile.dat');
        Reset(UnTypedFile);
        try
                BlockRead(UnTypedFile, buffer, 1, NumRecsRead);
        finally
                CloseFile(UnTypedFile);
        end;
end;
```

下面的示例演示如何向无类型文件中写入 128 **字节的数据：**

```
procedure TForm1.btnWriteClick(Sender: TObject);
var
        UnTypedFile: file;
        Buffer: array[0..128] of byte;
        NumRecsWriten: Integer;
begin
        AssignFile(UnTypedFile, 'c:\myfile.dat');
```

```
        //如果文件不存在，则创建新文件，否则，以可读写方式访问该文件
        if FileExists('c:\myfile.dat') then
            Reset(UnTypedFile)
        else
            Rewrite(UnTypedFile);

        try
            //将文件指针置于文件末尾
            Seek(UntypedFile, FileSize(UnTypedFile));
            FillChar(Buffer, SizeOf(Buffer), 'D');
            BlockWrite(UnTypedFile, buffer, 1, NumRecsWriten);
        finally
            CloseFile(UnTypedFile);
        end;
end;
```

数据块默认大小为 128 字节，这样，当从文件中读取数据的时候就会出现一个问题，即文件的大小必须是 128 的倍数，否则，读到文件末尾时就可能会超出文件以外。解决这个问题的方法是调用 Reset()方法来指定一个字节为单位，这样，任意大小的数据块都会是一个字节的倍数。

结合上面的两个示例，就可以实现复制文件的功能。

15.2 TTextRec 和 TFileRec 结构

大多数文件管理的例程都是来自操作系统的函数或者中断（加上了 Object Pascal 外套），例如，Reset()方法就是 Win32 Kernel32 中的 CreateFileA()函数的外套。将 Win32 函数加上 Object Pascal 外套，使开发者不必了解这些函数的具体细节。不过，这样一来，开发者也就很难了解文件的很多细节，例如文件句柄等，因为这些细节在 Object Pascal 中都是隐含的。

当使用非原生的 Object Pascal 函数时，需要传递一个文件句柄。例如，对于 LZCopy() 来说，可以分别把文本文件和二进制文件变量转换为 TTextRec 和 TFileRec。这两个记录类型包含了文件句柄以及有关文件的其他细节。除了文件句柄外，程序员很少需要访问其他的数据域。

获得一个文件句柄的代码如下所示：

```
TFileRec(MyFileVar).Handle;
```

TFileRec **记录是这样声明的：**

```
  TFileRec = packed record (* must match the size the compiler generates:
 332 bytes *)
        Handle: Integer;
```

```
        Mode: Word;
        Flags: Word;
        case Byte of
          0: (RecSize: Cardinal);     //   files of record
          1: (BufSize: Cardinal;      //   text files
              BufPos: Cardinal;
              BufEnd: Cardinal;
              BufPtr: PChar;
              OpenFunc: Pointer;
              InOutFunc: Pointer;
              FlushFunc: Pointer;
              CloseFunc: Pointer;
              UserData: array[1..32] of Byte;
              Name: array[0..259] of Char; );
  end;
```

TTextRec **记录是这样声明的：**

```
  PTextBuf = ^TTextBuf;
  TTextBuf = array[0..127] of Char;
  TTextRec = packed record (* must match the size the compiler generates:
460 bytes *)
        Handle: Integer;            (* must overlay with TFileRec *)
        Mode: Word;
        Flags: Word;
        BufSize: Cardinal;
        BufPos: Cardinal;
        BufEnd: Cardinal;
        BufPtr: PChar;
        OpenFunc: Pointer;
        InOutFunc: Pointer;
        FlushFunc: Pointer;
        CloseFunc: Pointer;
        UserData: array[1..32] of Byte;
        Name: array[0..259] of Char;
        Buffer: TTextBuf;
  end;
```

15.3　驱动器和目录

应用程序中往往需要访问系统中的驱动器和其中的目录。下面就介绍关于驱动器和目录的内容。

15.3.1 获得驱动器列表

为了获得可用的驱动器列表，可以调用 Win32 API 函数 GetDriveType()，这个函数需要传递一个 PChar 类型的参数 lpRootPathName，它指示驱动器的盘符号。函数返回一个整数值来表明驱动器的类型，返回值中可能有表 15.1 中的某些值。

表 15.1 GetDriveType()的返回值

返回值	含义
0	不可确定的驱动器
1	驱动器不存在
DRIVE_REMOVABLE	可移动驱动器
DRIVE_FIXED	不可移动驱动器
DRIVE_REMOTE	远程驱动器
DRIVE_CDROM	CDROM 光驱
DRIVE_RAMDISK	RAM 磁盘

下面的代码演示了 GetDriveType()函数的用法。运行结果如图 15.3 所示。

```
procedure Tfm_Main.btnGetDriveTypeClick(Sender: TObject);
var
    i: Integer;
    C: string;
    DriveType: Integer;
    DriveString: string;
begin
    for i := 65 to 90 do begin //控制磁盘符从 A~Z 循环
        C := chr(i)+ ':\';
        DriveType := GetDriveType(PChar(c));
        case DriveType of
          0: DriveString := C + '不确定驱动器';
          1: DriveString := C + '驱动器不存在';
          DRIVE_REMOVABLE: DriveString := C + '可移动驱动器';
          DRIVE_FIXED: DriveString := C + '不可移动驱动器';
          DRIVE_REMOTE: DriveString := C + '远程驱动器';
          DRIVE_CDROM: DriveString := C + 'CDROM 光驱';
          DRIVE_RAMDISK: DriveString := C + 'RAM 磁盘';
        end;
        if not (DriveType = 0) or (DriveType = 1) then
          lbDrives.Items.addObject(DriveString, Pointer(i));
    end;
end;
```

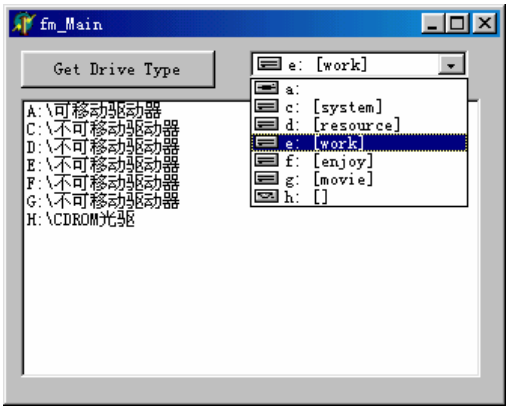

图 15.3　GetDriveType 函数运行结果

上面的示例中，以 26 个字母为顺序进行循环，依次判断它们是不是合法的驱动器，如果是，GetDriveType()函数返回驱动器的类型，并依次显示在了一个 TListBox 中。在上面的图中（见图 15.3）放置了一个 Delphi 组件 TDriveComboBox，其中列出了当前系统中所有存在的驱动器。

15.3.2　获得驱动器信息

获取了可用的驱动器以及类型以后，就可以进一步获得它们的某些信息，包括：

- **每一簇的扇区数**
- **每一扇区的字节数**
- **可用簇数**
- **簇的总数**
- **可用磁盘字节数**
- **磁盘大小**

调用 Win32 API 函数 GetDiskFreeSpace()可以得到上述前 4 项信息，最后面项信息可以由函数 GetDiskFreeSpace()取到的信息计算出来。

在上面的示例中，加入另外一个例程，它演示了函数 GetDiskFreeSpace()函数的用法，这段程序是在包含驱动器列表的 TListBox 的 OnClick 句柄中实现的，如下所示：

```
procedure Tfm_Main.lbDrivesClick(Sender: TObject);
var
    RootPath: string; //驱动器的根目录
    SectorsPerCluster: DWord; //每簇的扇区数
    BytesPerSector: DWord; //每扇区的字节数据
    FreeClusters: DWord; //可用簇数
    TotalClusters: DWord; //总簇数
    DriveByte: Byte; //驱动器字节数
    FreeSpace: Int64; //驱动器上的可用空间
    TotalSpace: Int64; //驱动器容量
```

```
        DriveNum: Integer; //驱动器序号
    begin
        with lbDrives do begin
            // 驱动器的序号就是盘符的 ASCII 码减去 64
            DriveByte := Integer(Items.Objects[ItemIndex]) - 64;
            RootPath := chr(Integer(Items.Objects[ItemIndex])) + ':\';

            if GetDiskFreeSpace(PChar(RootPath), SectorsPerCluster,
              BytesPerSector, FreeClusters, TotalClusters) then begin
              //调用成功，驱动器可确定
              leSectorsPerCluster.Text := Format('%.0n', [SectorsPerCluster * 1.0]);
              leBytesPerSector.Text := Format('%.0n', [BytesPerSector * 1.0]);
              leFreeCluster.Text := Format('%.0n', [FreeClusters * 1.0]);
              leTotalCluster.Text := Format('%.0n', [TotalClusters * 1.0]);

              FreeSpace := DiskFree(DriveByte);
              TotalSpace := DiskSize(DriveByte);
              leFreeSpace.Text := Format('%.0n', [FreeSpace * 1.0]);
              leTotalSpace.Text := Format('%.0n', [TotalSpace * 1.0]);
            end else begin   // 不能确定驱动器的信息
              leSectorsPerCluster.Text := '';
              leBytesPerSector.Text := '';
              leFreeCluster.Text := '';
              leTotalCluster.Text := '';
              leFreeSpace.Text := '';
              leTotalSpace.Text := '';
            end;
        end;
    end;
```

程序运行以后的界面如图 15.4 所示。

在这个示例中，用户只需要用鼠标选中驱动器列表中的一个驱动器，程序将会产生当前驱动器的盘符，将该盘符传递给 GetDiskFreeSpace()函数。如果该函数成功地取得了驱动器的信息，则在 Form 元件的下部将更新显示出这些信息；如果没有执行成功，则清空这些显示信息。

注意：这里没有调用 GetDiskFreeSpace()函数来获得驱动器的大小和可用空间，而是调用了在 SysUtils.pas 单元中声明的 DiskFree 和 DiskSize 方法，这是因为，GetDiskFreeSapce()函数在 Win95 下是有缺陷的，它不能报告超过 2G 大小的驱动器空间，而 DiskFree()方法和 DiskSize()方法则是经过改进的。

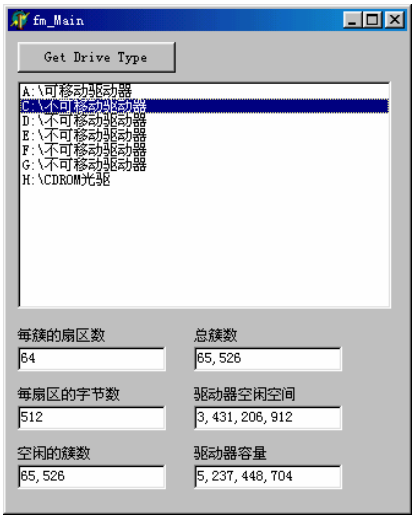

图 15.4　显示选定驱动器的信息

15.3.3　获取 Windows 目录

为了获得 Windows 目录，可以调用 Win32 API 函数 GetWindowsDirectory()，这个函数是这样声明的：

```
function GetWindowsDirectory(lpBuffer: PChar; uSize: UINT): UINT; stdcall;
```

其中，第一个参数是一个以 null 结束的字符串，用于返回 Windows 目录；第二个参数用于指定缓冲区的大小。下面的代码演示了这个函数的用法：

```
procedure TForm1.btnGetWindowsDirClick(Sender: TObject);
var
    WDir: String;
begin
    SetLength(WDir, 144);
    if GetWindowsDirectory(PChar(WDir), 144) <> 0 then begin
        SetLength(WDir, StrLen(PChar(WDir)));
        ShowMessage(WDir);
    end else
RaiseLastWin32Error;
end;
```

注意：上面的代码中，在调用 GetWindowsDirectory()函数后有一次 SetLength 函数调用。这是因为，对于 String 长字符串，Delphi 无法确定该字符串是否已经修改了，这时，必须调用 StrLen()来获得该字符串的长度，然后通过 SetLength()函数来调整字符串的长度。

提示：GetWindowsDirectory()**函数如果调用成功，函数将返回一个整数，表示目录路径的长度；否则，它将返回“**0**”，表示有错误发生，然后通过调用** RaiseLastWin32Error()**函数可以查明出错原因。**

15.3.4 获取系统目录

调用 Win32 API **函数** GetSystemDirctory()**可以获取系统目录的位置。**GetSystemDirectory()**函数与** GetWindowsDirectory()**函数的不同之处是，它返回的是系统目录的完整路径。**

下面的代码演示了该函数的使用：

```
procedure TForm1.btnGetSystemDirClick(Sender: TObject);
var
    SDir: String;
begin
    SetLength(SDir, 144);
    if GetSystemDirectory(PChar(SDir), 144) <> 0 then begin
        SetLength(SDir, StrLen(PChar(SDir)));
        ShowMessage(SDir);
    end else
RaiseLastWin32Error;
end;
```

这个函数的返回值与 GetWindowsDirectory()**的返回值含义相同。**

15.3.5 获取当前目录

编写应用程序的时候经常会需要知道当前的目录位置，即应用程序执行的位置，为此，可以调用 Win23 API **函数** GetCurrentDirectory()**。该函数与上面两个函数的调用方法完全相同，只是参数顺序相反，如下所示：**

```
procedure TForm1.btnGetCurrentDirClick(Sender: TObject);
var
    CDir: String;
begin
    SetLength(CDir, 144);
    if GetCurrentDirectory(144, PChar(CDir)) <> 0 then begin
        SetLength(CDir, StrLen(PChar(CDir)));
        ShowMessage(CDir);
    end else    RaiseLastWin32Error;
end;
```

除了调用 GetCurrentDirectory()**函数来获得当前目录位置以外，**Delphi **还提供了一些别的函数，例如** CurDir()**、**GetCurrentDir()**函数等。另外，**Delphi **可以从文件中提取目录信息，这时，用户只需要提供出当前应用程序的可执行文件，即** Application.ExeName **即可。如何提取目录信息根据所需信息的不同而不同。这些基本的函数包括：**ExtractFileDir**、**

ExtractFileDrive、ExtractFileExt、ExtractFileName、ExtractFilePath。**下面以一个示例来说明它们的返回结果。假设一个可执行文件的路径是** C:\Borland\Delphi6\Project1.exe，**表** 15.2 **列出了上述函数的返回结果。**

表 15.2　Delphi **提供的文件目录信息函数**

函数	返回结果
ExtractFileDir()	C:\Borland\Delphi6
ExtractFileDrive()	C:
ExtractFileExt()	.exe
ExtractFileName()	Project1.exe
ExtractFilePath	C:\Borland\Delphi6\

除了以上几节中所提到的几个函数以外，相关操作的函数还有很多，例如：获得系统临时目录的 GetTempPath()**函数、获取文件时间的** GetFileTime()**函数、获取系统信息的** GetSystemInfo()**、获得系统版本信息的** GetVersionEx()**函数等。读者可以参考** Win32 Help **的相关内容，这里就不再介绍。**

15.3.6　从目录中查找文件

有时，需要在一个目录中查找某种格式的文件，下面的代码演示了如何通过递归调用来实现对一个目录和其中子目录的搜索，如下所示：

```
unit Un_Main;

interface

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

type
    Tfm_Main = class(TForm)
        DriveComboBox1: TDriveComboBox;
        lDir: TDirectoryListBox;
        leFile: TLabeledEdit;
        lbFiles: TListBox;
        btnFind: TButton;
        procedure btnFindClick(Sender: TObject);
    private
        { Private declarations }
    public
        { Public declarations }
        procedure FindFiles(APath: string; AFile: string);
```

```
        end;

var
        fm_Main: Tfm_Main;
        TargetFile: string;

implementation

{15R *.dfm}

procedure Tfm_Main.btnFindClick(Sender: TObject);
begin
        Screen.Cursor := crHourGlass;
        try
                lbFiles.Clear;
                TargetFile := leFile.Text;
                FindFiles(lDir.Directory, TargetFile);
        finally
                Screen.Cursor := crDefault;
        end;
end;

//查找文件的主函数
procedure Tfm_Main.FindFiles(APath, AFile: string);
var
        FindResult: integer;
        FSearchRec, DSearchRec: TSearchRec;

        function IsDirNotation(ADirName: string): Boolean;
        begin
                Result := ((ADirName = '.') or (ADirName = '..'));
        end;
begin
        if APath[Length(APath)] <> '\' then
                APath := APath + '\';

        //在根目录中查找指定文件
        FindResult := FindFirst(APath + AFile, faAnyFile + faHidden +
                faSysFile + faReadOnly, FSearchRec);
        try
                while FindResult = 0 do begin
                  lbFiles.Items.Add(APath + FSearchRec.Name);
                  FindResult := FindNext(FSearchRec); //  查找下一个指定文件
```

```
            end;

            //进入当前目录的子目录继续查找
            FindResult := FindFirst(APath + '*.*', faDirectory, DSearchRec);
            while FindResult = 0 do begin
              if ((DSearchRec.Attr and faDirectory) = faDirectory) and
                 not IsDirNotation(DSearchRec.Name) then
                 //递归调用 FindFiles 函数
                 FindFiles(APath + DSearchRec.Name, TargetFile);
              FindResult := FindNext(DSearchRec);
            end;
        finally
            FindClose(FSearchRec);
        end;
end;

end.
```

图 15.5 是文件的搜索结果。

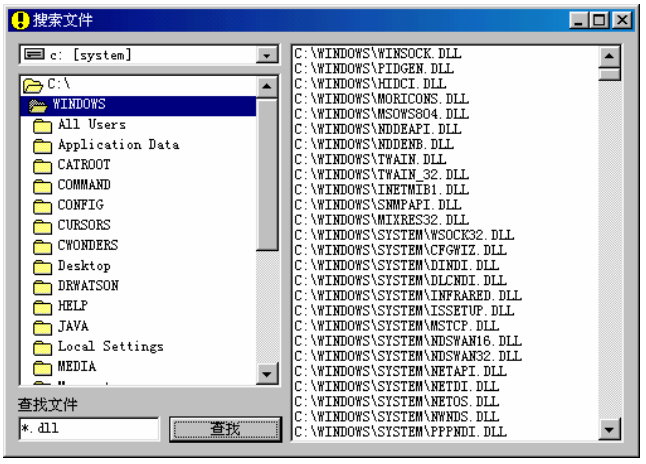

图 15.5　文件搜索结果

上面的示例中，有一个主函数 FindFiles()，其中包含两个 While…Do 循环。第一个循环用于在由 APath 参数指定的目录中查找 AFile 文件，然后将找到的文件及其目录加到列表框中；第二个循环用于在指定目录 APath 的子目录中进行查找，然后，递归调用 FindFiles 函数，其中第一个参数为 APath 的子目录。

上面用到了两个函数，分别是 FindFirst()和 FindNext()函数，还有一个结构 TSearchRec，下面就逐一介绍它们。

TSeachRec **结构实际上是 FindFirst()和 FindNext()函数的返回类型，注意不是返回值，这两个函数的返回值是一个整型值，这个值如果为“ 0 ”，表示已经搜索结束。**TSearchRec **结构是这样声明的：**

```
  TSearchRec = record
              Time: Integer;
              Size: Integer;
              Attr: Integer;
              Name: TFileName;
              ExcludeAttr: Integer;
{15IFDEF MSWINDOWS}   //在 Windows 系统中
              FindHandle: THandle    platform;
              FindData: TWin32FindData    platform;
{15ENDIF}
{15IFDEF LINUX} //在 Linux 系统中
              Mode: mode_t    platform;
              FindHandle: Pointer    platform;
              PathOnly: String    platform;
              Pattern: String    platform;
{15ENDIF}
  end;
```

可以看到，TSearchRec **类型的结构在** Windows **系统和** Linux **系统中的定义是不同的。**

其中，Time **域包含了创建和修改文件的时间，**Size **域是文件的大小，**Name **域是文件的名称，**Attr **域是文件的属性。文件属性包括表** 15.3 **中所列的值：**

表 15.3　**文件属性**

属性	值	描述
FaReadOnly	1501	只读文件
FaHidden	1502	隐藏文件
faSysFile	1504	系统文件
faVolumeID	1508	卷标
faDirectory	1510	目录
faArchive	1520	档案文件
faAnyFile	153F	任意文件

FindHandle **和** ExcludeAttr **域是在** FindFirst()**函数内部使用的。**

15.3.7　复制和删除目录

在 Win32 **之前，要复制和删除目录，需要对目录树进行分析，然后调用** FindFirst()**和** FindNext()**函数。现在，可以调用** ShFileOperation()**函数，使用它非常方便。在使用这个函数之前，需要了解一个结构** TSHFileOpStruct。TSHFileOpStmet **是** ShFileOperation()**函数运行的惟一参数，其中包含了函数功能的设置，如下所示：**

```
  TSHFileOpStruct = TSHFileOpStructA;
     SHFILEOPSTRUCTA = _SHFILEOPSTRUCTA;
```

```
_SHFILEOPSTRUCTA = packed record
    Wnd: HWND;
    wFunc: UINT;
    pFrom: PAnsiChar;
    pTo: PAnsiChar;
    fFlags: FILEOP_FLAGS;
    fAnyOperationsAborted: BOOL;
    hNameMappings: Pointer;
    lpszProgressTitle: PAnsiChar; { only used if FOF_SIMPLEPROGRESS }
end;
```

其中：

- Wnd **表示函数运行时要显示的对话框。它的句柄应附属于谁，一般设置为当前应用程序的句柄** Application.Handle **或者** Handle**。**
- wFunc **表示要运行的操作，其中有表** 15.4 **中所列的可能值。**

表 15.4　wFunc **域的可能值**

wFunc **值**	**操作**
FO_COPY	**拷贝文件或者目录**
FO_DELETE	**删除文件或者目录**
FO_MOVE	**移动文件或者目录**
FO_RENAME	**文件或者目录重命名**

- pFrom **和** pTo **分别表示源文件和目标文件的指针。**
- fFlag **是一个包含下面标记中的一个或者多个的组合，**FOF_ALLOWUNDO、FOF_CONFIRMMOUSE、FOF_FILESONLY、FOF_MULTIDESTFILES、FOF_NO-CONFIRMATION、FOF_NOCONFIRMMKDIR、FOF_RENAMEONCOLLISION、FOF_SILENT、FOF_SIMPLEPROGRESS、FOF_WANTMAPPINGHANDLE**，每个标记代表了不同的设置，具体细节可以参考** Win32 Help **中相关的帮助文件。**

下面的代码演示了 ShFileOperation()**函数的用法。注意，要使用该函数，必须在** Uses **单元中包含** ShellAPI **单元：**

```
procedure TForm1.CopyDir(AHandle: THandle; AFromDir, AToDir: String);
var
    ShFileOpStruct: TShFileOpStruct;
begin
    with ShFileOpStruct do begin
        Wnd := AHandle;
        wFunc := FO_COPY; //执行复制功能
        pFrom := PChar(AFromDir);
        pTo := PChar(AToDir);
```

```
            fFlags := FOF_NOCONFIRMATION or FOF_RENAMEONCOLLISION;
            fAnyOperationsAborted := False;
            hNameMappings := nil;
            hpszProgressTitle := nil;
        end;
        ShFileOperation(SHFileOpStruct);
end;
```

下面的代码调用了 ShFileOperation()**函数把一个目录移到回收站：**

```
procedure TForm1.DelToRecycle(AHandle: THandle; AFileName: String);
var
        ShFileOpStruct: TShFileOpStruct;
begin
        with ShFileOpStruct do begin
            Wnd := AHandle;
            wFunc := FO_DELETE;    //执行删除功能
            pFrom := PChar(AFileName);
            fFlags := FOF_ALLOWUNDO;    //允许取消操作
        end;
        ShFileOperation(SHFileOpStruct);
end;
```

15.4 内存映射文件

Win32 **环境下一个最方便的功能要算是它提供了一种访问文件的能力，让人感到访问磁盘文件就像访问内存中的文件一样。这种能力就是由内存映射文件提供的。**

使用内存映射文件后，可以避免进行所有的文件输入和输出操作，而只是保留虚拟的址空间，然后将磁盘文件提交给这段内存空间。这样，通过一个指向该区域的指针就可以访问文件的内容。

15.4.1 内存映射文件的作用

映射文件有 3 **个典型的用途：**

- Win32 **通过内存映射文件调入和执行** EXE **文件和** DLL **文件。这样，节省了页交换文件的空间，从而减少了文件调入的时间。**
- **通过指向内存区域的指针，可以访问内存映射文件中的数据。这样，不仅简化了对文件的访问，而且不再需要编写各种文件缓冲方案。**
- **通过映射文件，运行在同一台机器上的多个过程可以共享数据。**

第二个用途对程序员的开发工作很有帮助，本章将着重讨论。

15.4.2　创建内存映射文件

创建一个内存映射文件，实际上就是把文件和进程虚拟空间关联在一起。要查看或者编辑文件的内容，必须获得文件映射对象的视图。这样，通过指针访问文件内容的时候，就好像访问内存区域一样了。

当用户向文件视图中写入数据的时候，系统负责处理数据的缓存、缓冲和调入以及内存的分配和释放。文件的输入和输出全部由系统复杂处理，这就是映射文件的好处。前面讨论的文件的输入和输出技术大大简化了文件的操作，而且速度也有所提高。下面将介绍创建或者打开一个内存映射文件的步骤：

（1）创建或者打开文件。

创建或者打开一个文件的第一步工作是获取映射文件的句柄。在获得了合法的文件句柄以后，才可以创建文件映射对象。用户可以调用 FileCreate()或者 FileOpen()函数来获得文件句柄，它们都是在 SysUtils.Pas 单元中声明的，如下所示：

```
function FileCreate(const FileName: string): Integer;
function FileOpen(const FileName: string; Mode: LongWord): Integer;
```

FileCreate()函数用于创建一个新的文件，文件名由 FileName 给定。如果函数调用成功，则返回一个合法的文件句柄；否则，返回 INVALID_HANDLE_VALUE 常量。

FileOpen()函数能以某种模式打开一个已有的文件。如果函数调用成功，则返回一个合法的文件句柄，否则，返回 INVALID_HANDLE_VALUE 常量。其中第二个参数用于指定文件打开模式，表 15.5 中列出了 Mode 的可能值。

表 15.5　文件打开模式

访问模式	含义
fmOpenRead	只读
fmOpenWrite	只写
fmOpenReadWrite	只读写

如果 Mode 参数设置为“0”，则表示既不能读也不能写，这种模式适用于只需要获得文件的属性的情况；如果要在不同的应用程序之间共享数据，可以使用上表中列出的模式（见表 15.5）和表 15.6 中的文件共享模式进行“或”运算。

表 15.6　文件共享模式

共享模式	含义
fmShareCompat	与 DOS 1.x 和 2.x 文件控制块兼容的文件共享机制
fmShareExclusive	禁止共享
fmShareDenyWrite	不允许其他人使用以 fmOpenWrite 模式打开的文件
fmShareDenyRead	不允许其他人使用以 fmOpenRead 模式打开的文件
fmShareDenyNone	允许以任何模式打开的文件

（2）创建文件映射对象。

要创建一个文件映射对象，可以使用 CreateFileMapping()函数。该函数适用于创建有名称的和未命名的文件映射对象。该函数声明如下所示：

```
function CreateFileMapping(
hFile: THandle;
lpFileMappingAttributes: PSecurityAttributes;
flProtect, dwMaximumSizeHigh, dwMaximumSizeLow: DWORD;
lpName: PChar): THandle; stdcall;
```

传递给函数的参数用于给出了要创建的文件映射对象所需要的信息。

第一个参数 hFile 为一个合法的文件句柄，可以是前面 FileOpen()或者 FileCreate()函数的返回值，也可以通过系统页交换文件来创建文件映射对象；第二个参数 lpFileMappingAttributes 是一个 PSecurityAttributes 类型的指针，指向文件映射对象的安全属性结构，这个参数通常设为“0”；第三个参数 flProtect 用于指定文件视图的保护类型，这个值必须与前面创建或打开文件以获得文件句柄时用到的参数保持一致。表 15.7 中列出了 flProtect 参数可能的值。

表 15.7 保护属性

保护属性	含义
Page_ReadOnly	文件只读。文件必须用 FileCreate()创建，或者用 FileOpen()以 fmOpenRead 模式打开
Page_ReadWrite	文件可读写。文件必须以 fmOpenReadWrite 模式打开
Page_WriteCopy	文件可读写。但进行写操作时，会复制一个修改过的页面，这样，多个进程之间共享的映射文件就不会双倍消耗系统内存和其他资源。文件必须以 fmOpenWrite 或者 fmOpenReadWrite 模式打开

当获得了合法的文件映射对象以后，就可以将文件的数据映射到进程的地址空间中去了。

（3）将文件的视图映射到进程地址空间。

MapViewOfFile()函数可以将文件的视图映射到进程地址空间。这个函数是这样声明的，如下所示：

```
function MapViewOfFile(
        hFileMappingObject: THandle;
        dwDesiredAccess: DWORD;
        dwFileOffsetHigh,
        dwFileOffsetLow,
        dwNumberOfBytesToMap: DWORD): Pointer;
```

hFileMappingObject 参数就是通过 CreateFileMapping()或者 OpenFileMapping()函数返回的文件映射对象的句柄。

dwDesiredAccess 参数用于指定访问文件数据的模式，表 15.8 中列出了所有的文件视图访问模式。

表 15.8　文件视图访问模式

dwDesiredAccess 值	含义
File_Map_Write	可读写。必须用 Page_Read_Write 属性调用 CreateFileMapping()函数
File_Map_Read	只读。必须用 Page_Read_Write 或 Page_Read 属性调用 CreateFileMapping()函数
File_Map_All_Access	与 File_Map_Write 相同
File_Map_Copy	允许 Copy-on-Write 模式，即在对文件写入的同时进行内容复制。必须用 Page_Read_Only、Page_Read_Write 或者 Page_Write_Copy 属性调用 CreateFileMapping()函数

dwFileOffsetHigh 参数用于指定文件映射起始位置的偏移量的高 32 位。DwFileOffsetLow 参数用于指定文件映射起始位置的偏移量的低 32 位。dwNumberOfBytesToMap 参数用于指定需要映射的字节数，0 表示文件的全部。

MapViewOfFile()函数返回文件视图的起始地址。如果函数调用失败，则返回 nil，此时，必须通过调用 GetLastError()函数来确定出错的原因。

（4）取消文件视图的映射。

UnmapViewOfFile()函数可以取消文件视图与进程地址空间的映射关系。该函数的声明如下所示：

```
function UnmapViewOfFile(lpBaseAddress: Pointer): BOOL; stdcall;
```

这个函数惟一的参数 lpBaseAddress 必须设置为映射区域的起始地址，即 MapViewOfFile()函数的返回值。

在结束对文件的处理以后，必须调用 UnmapViewOfFile()函数来释放文件的映射区域所占用的内存。

（5）关闭文件映射和文件内核对象。

调用 FileOpen()和 CreateFileMapping()函数将打开内核对象，因此，程序必须负责关闭这些内核对象，为此，可以调用 CloseHandle()函数，这个函数是这样定义的，如下所示：

```
function CloseHandle(hObject: THandle): BOOL; stdcall;
```

如果函数调用成功，则返回 True，否则函数返回 False。

了解了上面创建或者打开映射文件的操作步骤以后，下面来看一个比较简单的示例。这个示例中依次应用了上述的基本步骤，它的功能是将一个文本文件中的字符按照要求全部转换为大写或者小写，程序运行界面如图 15.6 所示。

图 15.6　内存映射文件程序主界面

```
unit Un_Main;

interface

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

const
    FileName = 'test.txt';

type
    Tfm_Main = class(TForm)
        btnUpperCase: TButton;
        btnLowerCase: TButton;
        btnOpenFile: TButton;
        reContents: TRichEdit;
        procedure btnOpenFileClick(Sender: TObject);
        procedure btnUpperCaseClick(Sender: TObject);
        procedure btnLowerCaseClick(Sender: TObject);
    private
        { Private declarations }
    public
        { Public declarations }
        UpperCase: Boolean;
        procedure ChangeAllCase;
    end;

var
    fm_Main: Tfm_Main;
```

```
implementation

{15R *.dfm}

procedure Tfm_Main.btnOpenFileClick(Sender: TObject);
begin
    reContents.Lines.LoadFromFile(FileName);
end;

procedure Tfm_Main.btnUpperCaseClick(Sender: TObject);
begin
    UpperCase := True;
    ChangeAllCase;
end;

procedure Tfm_Main.btnLowerCaseClick(Sender: TObject);
begin
    UpperCase := False;
    ChangeAllCase;
end;

procedure Tfm_Main.ChangeAllCase;
var
    FileHandle: THandle; //文件句柄
    MapHandle: THandle;    //文件映射大小的句柄
    FileSize: Integer;
    FData: PByte; //数据映射指针
    PData: PChar; //标识文件数据的指针
begin
    // 确保文件存在，并打开获得一个文件句柄
    if not FileExists(FileName) then
        raise Exception.Create('文件不存在')
    else
        FileHandle := FileOpen(FileName, fmOpenReadWrite);

    if FileHandle = INVALID_HANDLE_VALUE then
        raise Exception.Create('文件打开失败');

    // 创建文件映射对象
    try
        FileSize := GetFileSize(FileHandle, nil);
        MapHandle := CreateFileMapping(FileHandle, nil, Page_ReadWrite, 0, FileSize, nil);
        if MapHandle = 0 then
```

```
            raise Exception.Create('文件映射失败');
        finally
            CloseHandle(FileHandle);
        end;

        //映射文件视图到进程地址空间，返回文件数据指针
        try
            FData := MapViewOfFile(MapHandle, FILE_MAP_ALL_ACCESS, 0, 0, FileSize);
            if FData = nil then
              raise Exception.Create('映射文件视图失败');
        finally
            CloseHandle(MapHandle);
        end;

        try
            PData := PChar(FData);

            // 在文件的结尾加一个结束字符
            inc(PData, FileSize);
            PData^ := #0;

            //字符大小写转换
            if UpperCase then
              StrUpper(PChar(FData))
            else
              StrLower(PChar(FData));
        finally
            UnmapViewOfFile(FData);
        end;

        reContents.Lines.Clear;
        reContents.Lines.LoadFromFile(FileName);
    end;

    end.
```

程序的运行效果是，单击“打开文件”按钮将在 RichEdit **文本框中显示出当前目录中** test.txt **文件中的内容，然后单击“全部大写”按钮将会看到所有字符全部变为大写，单击“全部小写”按钮后所有字符将以小写显示。**

从上面的程序清单中可以看到，程序的第一步是获取一个文件句柄，这是通过调用 FileOpen()**来完成的。文件访问模式设置成了** fmOpenWrite**，以便对文件进行读写。**

第二步是通过调用 CreateFileMapping()**函数来获得文件映射对象。如果调用失败，则将触发一个异常；否则，进入下一步，将文件映射对象映射为一个视图。**

注意：在文件映射视图数据的结尾，程序将最后一个字符改写为了“＃0”结束符，这是因为，程序是把文件作为了以 Null 结束的字符串来处理的，所以，必须确保字符串中存在一个 Null 字符，用它来表示字符串的结束。

现在就可以来改写字符的大小写了，实现这个功能的是两个函数：StrUpper()和 StrLower()函数。在执行完该方法以后，如果使用文本编辑器来查看文件，就会发现文件中的字符都已转变成了指定的大写或小写。

最后，调用 UnmapViewOfFile()函数来取消文视图的映射。

用户也许已经注意到了，程序中即使是文件被映射为一个视图的时候，也需要释放在这之前的文件句柄和文件映射对象句柄。这是因为，当调用 MapViewOfFile()函数的时候，系统中保留了文件句柄和文件映射对象句柄的引用计数。因此，用户最好调用 CloseHandle()函数来关闭对象，以减少系统资源的浪费。

15.5　本 章 小 结

本章主要讲解了获取文件、目录和驱动器信息的编程技术，掌握这方面的技术对于开发各种应用程序都是非常重要的。因为无论是通用还是专业应用程序，都不可能离开文件或者目录操作。

本章还介绍了如何利用 TFileStream 的派生类来操纵文件的输入和输出，以及利用 Win32 API 函数来实现文件和目录的拷贝、删除等操作。最后，介绍了 Win32 内存映射文件的用法。

第16章 图像编程

计算机图像编程非常有意思，这也许是编程中最令人愉快的一种，程序员常常会从千变万化的图像中得到愉悦，随着对图像编程理解的深入，会越来越感到图像设计的乐趣，并会对计算机图像的美妙之处赞叹不已。

本章将详细介绍与图像相关的知识，例如 TCanvas 类、TImage 类、以及字体等。Win32 使用 GDI（图像设备接口）在计算机屏幕上画图，传统的 Windows 编程则是直接使用 GDI 函数和工具。在 Delphi 中，TCanvas 封装并简化了这些工具和技术的使用。本章将介绍如何利用 TCanvas 来实现一些有用的图像功能。用户也可以使用 Delphi 和 Win32 GDI 来创建高级的图像程序。

16.1 GDI 和 TCanvas 类

Windows 使用一个叫做“GDI（图形设备接口）”的系统来提供在屏幕上绘图的技术，Delphi VCL 中提供了一个 Tcanvas 类，它封装了 GDI 并使之非常容易地被程序员使用。Delphi 中，很多对象都有 TCanvas 特性，例如 TForm、TGraphic 等。Canvas 是一块画布，为 Form 等对象提供了一个绘制图像的平面。TCanvas 中用于绘画的工具有画笔 TPen、刷子 TBrush 和字体 TFont 等。

16.1.1 理解 GDI

GDI 是 Windows 标准图形编程的关键。就像要在一张纸上绘画、涂抹、着色一样，计算机图形编程也是使用一些绘画工具在一个界面上进行的，这个界面就是窗口本身，而在这个窗口上绘画所需的任何一件工具都必须通过一个 DC（Device Context，设备环境）来获得，窗口的界面则需要通过 GDI 进入。

GDI 函数组包含了在一个窗口界面上绘制图形所有必需的功能，这些功能使用户能够在屏幕上绘制出文本、各种形状以及位图等，它们还能够控制字型、颜色、线条粗细、阴影、比例、方向等各种相关的因素。

在绘制之前，必须准备做下面几件事情。首先，必须得到这个窗口的设备环境，然后在窗口界面上绘画，最后要释放设备环境。释放设备环境是非常重要的，因为如果不释放设备环境，程序会在系统中留下一个资源漏洞，严重的话会导致蓝屏的出现，即系统崩溃。

下面是一个简单的示例，它的功能是利用 GDI 将在屏幕的（10，10）坐标点上的像素颜色变为红色：

```
procedure TForm1.Button1Click(Sender: TObject);
var
```

```
        FDC: HDC;
begin
        // 得到设备环境 DC
        FDC := GetDC(Handle);
        setPixel(FDC, 10, 10, clRed);
        // 释放设备环境
        ReleaseDC(Handle, FDC);
end;
```

在这个示例中，第一步调用了函数 GetDC，它返回了一个设备环境，函数需要一个句柄参数，用来指定要在上面绘画的窗口句柄。得到设备环境以后就可以在上面进行各种绘画操作了。最后一步是释放设备环境。

还有另一种得到设备环境的方法。VCL 提供的 TCanvas 对象包装了 GDI，并且处理了所有的设备环境和幕后资源管理问题，同时，利用 TCanvas 的 Handle 特性可以得到任何设备环境。实际上，这个特性正是该对象画布的句柄。例如，上面的示例也可以这样实现：

```
Canvas.Pixels[10,10] := clRed;
```

可以看到，上面的语句提供了一种在屏幕上绘画的很便捷的方法，甚至没有必要担心设备环境的问题，它将上面示例中的 3 行代码合并成了一行。

16.1.2　画笔 TPen

用户可以使用画笔在画布上画线，也可以通过 Canvas.Pen 特性来访问画笔。通过修改 Tpen 的 Color、Width、Style 和 Mode 特性就可以控制如何画线。

1．Color 特性

Color 特性用于指定画笔的颜色。Delphi 提供了预先定义好的颜色常量，例如 clRed 表示正红色、clBlue 表示正蓝色；另外还提供了表示 Win32 系统屏幕元素颜色的常量，例如 clActiveCaption 和 clHighlightText，它们对应于 Win32 活动的标题和突出显示的文本。

下面的代码将画笔的颜色设为蓝色：

```
Canvas.Pen.Color := clBlue;
```

Win32 用长整型来表示颜色，其中，最低的 3 个字节分别代表红、绿、蓝的强度。这 3 个值组合起来就构成了合法的 Win32 颜色。Delphi 提供了 RGB(R，G，B)函数来供用户组合颜色，其中的 3 个参数分别表示红、绿、蓝 3 种颜色的强度，函数的返回值为一个长整型值，作为 Win32 颜色。每种颜色强度有 255 个值，RGB()函数可以返回大概 1600 万种颜色。RGB(255,255,255)返回白色，RGB(0,0,0)返回黑色，RGB(255,0,0)返回红色，RGB(0,255,0)返回绿色，RGB(0,0,255)返回蓝色。

函数 ColorToRGB()可以把 Win32 系统的颜色转换为 RGB 颜色。

2．Style 特性

通过 Style 特性，画笔就可以以不同的风格画线。表 16.1 中列出了 Pen.Style 特性可能的值。

表 16.1 画笔的 Style 特性值

特性值	绘制
psClear	不可视线
psDash	虚线
psDashDot	点划线
psDashDotDot	双点划线
psDot	点线
psInsideFrame	内框线
psSolid	实线

用户可以通过下面的代码来设置画笔的样式：

```
Canvas.Pen.Style := psDashDot;
```

3．Width 特性

Pen.Width 特性用于指定画笔的笔宽。它以象素为单位，此特性值越大，画笔画出的线就会越粗。

注意：点划线只适用于宽度为“1”的画笔，如果把宽度设置为“2”，画出的将是实线。

4．Mode 特性

有 3 个因素决定了 Win32 如何在画布上画出相似或者线：画笔的颜色、画布的颜色和 Win32 对两个颜色值进行的逐位操作，这个操作称为 ROP（光栅操作），即两种颜色的叠加效果。Pen.Mode 特性就是用于指定画布的 ROP 模式。表 16.2 中列出了 Win32 预定义的 16 种模式。

表 16.2 Pen.Color(S)在画布颜色(D)上的画笔模式

模式	像素颜色	布尔运算
pmBlack	总是黑色	0
pmWhite	总是白色	1
pmNOP	不变	D
pmNOT	反转屏幕上的颜色	not D
pmCopy	以 Color 特性设定的颜色代替	S
pmNotCopy	反转画笔的颜色	not S
pmMergePenNot	画笔的颜色和屏幕颜色反转后的颜色进行“或”操作	S or not D
pmMaskPenNot	画笔的颜色和屏幕的颜色反转后的颜色进行“与”操作	S and not D
pmMergeNotPen	屏幕颜色和画笔颜色反转后的颜色进行或操作	not S or D
pmMaskNotPen	屏幕颜色和画笔颜色反转后的颜色进行与操作	not S and D
pmMerge	画笔颜色和屏幕颜色进行或操作	S or D

（续表）

模式	像素颜色	布尔运算
pmNotMerge	反转 pmMerge	not (S or D)
pmMask	画笔颜色和屏幕颜色进行与操作	S and D
pmNotMask	反转 pmMask	not (S and D)
pmXor	画笔颜色和屏幕颜色进行非或操作	S XOR D
pmNotXor	反转 pmXor	not (S XOR D)

假如将 Pen.Mode 设为 pmCopy，画笔的颜色将由它的 Color 特性决定。如果要在一个白色的屏幕上画两条黑线，当两条线相交的时候，屏幕上的交点应该为白色，而不是黑色，这时候，就应该把 Pen.Color 设为 clBlack，而把 Pen.Mode 设为 pmNot，这就能使画笔将画笔和画布的颜色合并后的颜色取反后再进行绘制。

16.1.3　TCanvas.Pixels 特性

TCanvas.Pixels 特性是一个二维数组，它的每个元素代表 Form 表面或者客户区的一个像素的 TColor 值。Form 最左上角的像素为：Canvas.Pixels[0,0]。

一般情况下很少需要访问 Form 的单个像素，因此通常不需要用到 Pixels 特性。而且访问 Pixels 特性速度很慢，如果使用 GetPixel()和 SetPixel()来访问像素，效率非常低，因为这两个函数都依赖于 24 位的 RGB 值，如果不是 24 位 RGB 设备，当把 RGB 转换为设备像素格式时会出现颜色失真。为了以较快的速度操纵大量像素，可以使用 TBitmap.Scanline 数组。

16.1.4　TCanvas.Brush 特性

画布的刷子 TBrush 用于填充画布上的区域和图形。与 TPen 不同的是，画笔用来在画布上画线，而刷子则用不同的颜色、样式和形状来填充画布的区域。

Canvas 的 TBrush 对象有 3 个重要的特性：Color、Style、Bitmap 和一个方法 Assign()，这 3 个特性决定了刷子在画布上如何绘制。其中，Color 用于指定一个刷子的颜色，Style 用于指定刷子的形状，而 Bitmap 特性用于指定一个位图来充当刷子的背景。

Style 特性有 8 个可能的值：bsSolid、bsClear、bsHorizontal、bsVertical、bsFDiagonal、bsBDiagonal、bsCross 和 bsDiagCross。图 16.1 表示出了以上 8 种 Style 值所代表的图形。

位图是图形图像的一种二进制表示，它具有一个.bmp 的扩展名，可以通过下面的代码来为一个刷子设置位图：

```
Canvas.Brush.Bitmap.LoadFromFile('c:\MyBmp.bmp');
```

当刷子使用位图完毕以后，必须将 TBrush.Bitmap 特性设为 nil，因为刷子并不拥有该位图。

Windows 规定，作为刷子图案的位图的大小只能是 8×8，并且必须是与设备无关的位图。Windows 95 会拒绝接受比这个尺寸大的位图，Windows NT 可以接受较大的位图，但只能用右上角的 8×8 部分。

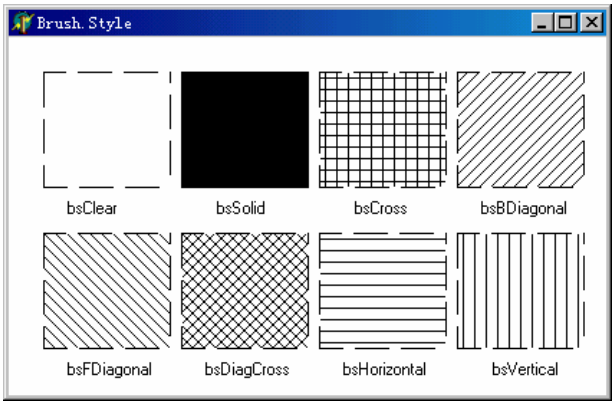

图 16.1 Brush.Style

提示：默认情况下，刷子的颜色为 clWhite，**样式为** bsSolid，**没有位图。**

最后介绍一下 TBrush.Assign()**方法，这个相当方便的方法允许将一个画刷的内容复制到另一个** TBrush **实例上去。可以创建几个定制的画刷，通过在它们之间的转换同时来使用它们。下面的示例将当前的画刷复制到** MyBrush **变量中：**

```
MyBrush := TBrush.Create;
MyBrush.Assign(Image1.Canvas.Brush);
```

这样，MyBrush **就具有了当前画刷的所有特性。**

16.1.5 Font 字体

Canvas.Font **特性使用户可以用** Win32 **字体画出文字，然后可以通过修改字体的颜色、名称、大小、高度、样式等来设置文字的外观。**Font.Color **特性可以是任何** Delphi **预定义的颜色。**

Font.Name **特性用于指定** Windows **字体名。例如，下面的代码把字体设置为** New Times Roman**：**

```
Canvas.Font.Name := 'New Times Roman';
```

Font.Size **以磅为单位指定字体的大小。**Font.Style **是由一种或者多种样式组成的集合，这些样式包括：**fsBold(**黑体**)、fsItalic(**斜体**)、fsUnderLine(**下划线**)、**fsStrickOut(**删除线**)。用户可以使用下面的代码来组合以上的样式：**

```
Canvas.Font.Style := [fsBold, fsItalic];
```

Delphi **提供了一个组件** TFontDialog **对话框来选择当前系统中所安装的所有的字体。**

注意：当要复制 TBitmap、TBrush、TPen **等对象时，应当使用** Assign()**方法。如果直接通过赋值语句来操作，例如** Brush1 := Brush2，**看起来合法，但它直接进行指针复制，将使两个实例都指向了同一个刷子对象，这将导致堆不足。而通过**

Assign 方法，即 Brush1.Assign(Brush2)，就可以保证释放前面已经分配给 Brush1 的资源。

但当要复制 TFont 对象时，比较特殊，用户完全可以这样来复制，如下所示：

```
Form1.Font := Form2.Font;
```

这是合法的，因为 Font 特性的写方法内部会调用 Assign()来进行复制。但要注意，只有在复制 TFont 对象时才可以这么做。一般情况下，最好还是用 Assign()方法比较安全。

16.1.6　TCanvas.CopyMode 特性

TCanvas.CopyMode 特性决定了画布如何从另一个画布中复制图像给自己。它的所有可能的值包括：cmDstInvert、cmMergeCopy、cmMergePaint、cmNotSrcCopy、cmNotSrcErase、cmPatCopy、cmPatInvert、cmPatPaint、cmSrcAnd、cmSrcCopy、cmSrcErase、cmSrcInvert、cmSrcPaint 和 cmWhiteness 等，它们所显示的效果将在本节后面的程序中演示。如果 CopyMode 设置为 cmSrcCopy，那么源图将完全复制到目标图上；如果设置为 cmSrcInvert，源图将和目标图的像素进行按位 XOR 运算，图 16.2 和图 16.3 分别是以上两种设置的显示效果。

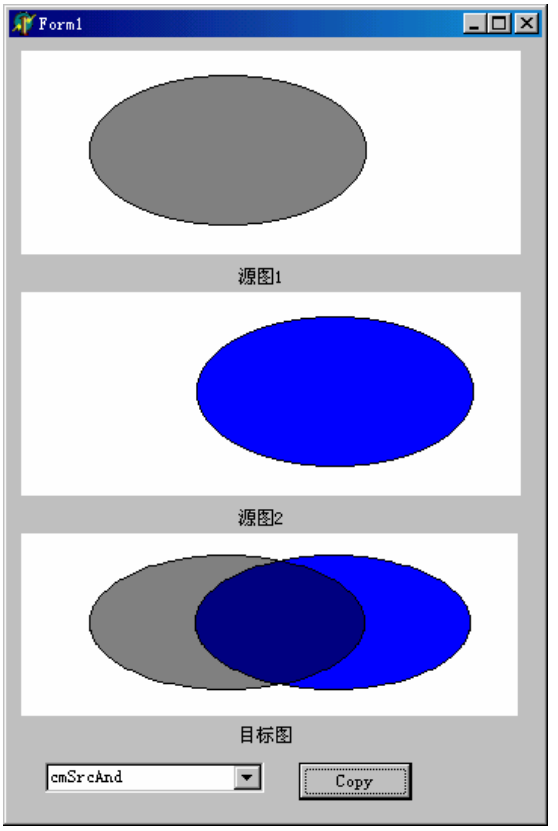

图 16.2　cmSrcAnd 模式

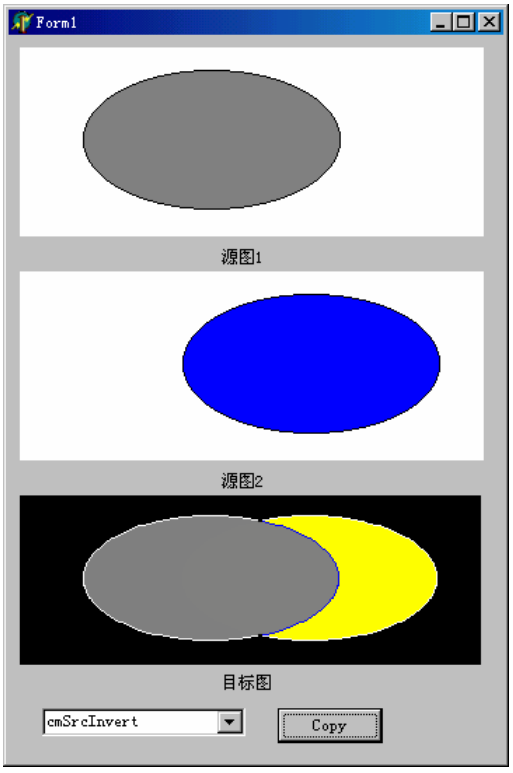

图 16.3　cmSrcInvert **模式**

下面的代码清单演示了所有各种 CopyMode **值的复制模式。程序中使用了** 3 **个** Image**，其中** Image1 **和** Image2 **分别是要复制到一起去的两幅图像，**Image3 **是复制到一起后的显示效果图，如下所示：**

```
unit Unit1;

interface

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

type
    TForm1 = class(TForm)
        Image1: TImage;
        Image2: TImage;
        Image3: TImage;
        cbCopyMode: TComboBox;
        btnCopy: TButton;
        Label1: TLabel;
```

```
        Label2: TLabel;
        Label3: TLabel;
        procedure btnCopyClick(Sender: TObject);
    private
        { Private declarations }
    public
        { Public declarations }
    end;

var
    Form1: TForm1;
    cm: TCopyMode;

implementation
{16R *.DFM}

procedure TForm1.btnCopyClick(Sender: TObject);
var
    Image1Rect, Image2Rect, Image3Rect: TRect;
begin
//首先将 Image1 复制到 Image3 上
    Image1Rect.TopLeft := Point(0, 0);
    Image1Rect.BottomRight := Point(Image1.Width, Image1.Height);

    Image2Rect.TopLeft := Point(0, 0);
    Image2Rect.BottomRight := Point(Image2.Width, Image2.Height);

    Image3Rect.TopLeft := Point(0, 0);
    Image3Rect.BottomRight := Point(Image3.Width, Image3.Height);

    Image3.Canvas.CopyMode := cmSrcCopy;
    Image3.Canvas.CopyRect(Image3Rect, Image1.Canvas, Image1Rect);

//然后将 Image2 也复制到 Image3 上
    case cbCopyMode.ItemIndex of
        0: cm := cmBlackness;
        1: cm := cmDstInvert;
        2: cm := cmMergeCopy;
        3: cm := cmMergePaint;
        4: cm := cmNotSrcCopy;
        5: cm := cmNotSrcErase;
        6: cm := cmPatCopy;
        7: cm := cmPatInvert;
```

```
            8: cm := cmPatPaint;
            9: cm := cmSrcAnd;
            10: cm := cmSrcCopy;
            11: cm := cmSrcErase;
            12: cm := cmSrcInvert;
            13: cm := cmSrcPaint;
            14: cm := cmWhiteness;
        else
            cm := cmSrcCopy;
        end;

        Image3.Canvas.CopyMode := cm;
        Image3.Canvas.CopyRect(Image3Rect, Image2.Canvas, Image2Rect);
end;

end.
```

16.2 TCanvas 的方法

TCanvas 对象封装了许多 GDI 函数，通过 TCanvas 的方法，可以画线、画图形、显示文字、把一个图像的某个区域复制到另一个图像上，甚至在画布上放大或者填充区域。

16.2.1 TCanvas 画线

TCanvas.MoveTo(A,B)方法用于在画布上改变 Canvas.Pen 画笔的位置到坐标（A,B）点，TCanvas.LineTo(X,Y)方法用于在画布上画出一条从（A,B）点到（X,Y）点的直线。下面的代码演示了从 Form 客户区的左上角到右下角的直线，如下所示：

```
Canvas.MoveTo(0,0);
Canvas.LineTo(ClientWidth, ClientHeight);
```

注意：在有的版本的 Windows 系统下，MoveTo(0,0)将画笔移到窗口的右上角，例如中东版的 Windows，这与地区文化有关，因为有的地区习惯于从右至左写字。

16.2.2 TCanvas 画几何形状

TCanvas 提供了多种方法用来在画布上画几何图形，包括：Arc、Chord、Ellipse、Pie、Polygon、Polyline、Rectangle 和 RevendRect 等，这些方法分别能够画出弧线、弦、椭圆、饼图、矩形等图形。例如下面的代码在 Form 的客户区上画一个椭圆，它调用了 Ellipse() 方法：

```
Canvas.Ellipse(0, 0, ClientWidth, ClientHeight);
```

可以通过设定 Canvas.Brush.Style 特性来指定刷子形状填充椭圆的内部，如下所示：

```
Canvas.Brush.Style := bsCross;
Canvas.Ellipse(0, 0, ClientWidth, ClientHeight);
```

另外也可以用 TCauvas.Brush.Bitmap 特性指定一个位图，用它来填充画布的一个区域或者几何图形。

一些 TCanvas 的方法需要传递一些参数来描述要画的图形，例如，Polyline()方法需要传递 TPoint 类型的数组，用于指定要连成线的点的坐标。

16.2.3　TCanvas 输出文字

TCanvas 封装了 Win32 GDI 中用于输出文字的例程，下面就介绍如何使用这些例程以及那些没有被 TCanvas 封装起来的 Win32 GDI 函数。

TCanvas 中有一个 TextOut()函数用来在 Form 的客户区上输出文字，另外还有 TextWidth()和 TextHeight()两个函数来读取文字的宽度和高度（以像素为单位）。

TextRect()函数也可以用来在 Form 上输出文字，但只限于由 TRect 结构指定的矩形中。超出 TRect 边界之外的文字将被裁剪掉。

下面是一个文字输出的程序实例：

```
unit Unit1;

interface

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

type
    TfmMain = class(TForm)
        btnTextRect: TButton;
        btnTextOut: TButton;
        gbDrawText: TGroupBox;
        btnCenter: TButton;
        btnLeft: TButton;
        btnRight: TButton;
        procedure btnTextRectClick(Sender: TObject);
        procedure FormCreate(Sender: TObject);
        procedure btnTextOutClick(Sender: TObject);
        procedure btnCenterClick(Sender: TObject);
        procedure btnLeftClick(Sender: TObject);
        procedure btnRightClick(Sender: TObject);
    private
        { Private declarations }
    public
        { Public declarations }
```

```
            procedure ClearCanvas;
        end;

var
        fmMain: TfmMain;
        S: string;
        R: TRect;

implementation

{16R *.DFM}


procedure TfmMain.FormCreate(Sender: TObject);
begin
        S := 'Example Text';
end;

//清除画布
procedure TfmMain.ClearCanvas;
begin
        with Canvas do begin
                Brush.Style := bsSolid;
                Brush.Color := clWhite;
                FillRect(ClipRect);
        end;
end;

//使用 TextRect 方法来输出文字，如图 16.4
procedure TfmMain.btnTextRectClick(Sender: TObject);
var
        TW, TH: Integer;
begin
        ClearCanvas;
        Canvas.Font.Size := 20;
        TW := Canvas.TextWidth(S);
        TH := Canvas.TextHeight(S);

        R := Rect(1, TH div 2, TW + 1, TH + TH div 2);
        Canvas.Rectangle(R.Left - 1, R.Top - 1, R.Right + 1, R.Bottom + 1);
        Canvas.TextRect(R, 0, 0, S);
end;
```

```
//使用 TextOut 方法来输出文字，并显示文字高度和宽度，如图 16.5
procedure TfmMain.btnTextOutClick(Sender: TObject);
begin
    ClearCanvas;
    with Canvas do begin
        Font.Size := 18;
        TextOut(10, 10, S);
        TextOut(50, 50, 'TextWidth = ' + IntToStr(TextWidth(S)));
        TextOut(100, 100, 'TextHeight = ' + IntToStr(TextHeight(S)));
    end;
end;

//DrawText 函数输出文字，文字居中显示
procedure TfmMain.btnCenterClick(Sender: TObject);
begin
    ClearCanvas;
    Canvas.Font.Size := 10;
    R := Rect(10, 10, 80, 100);
    Canvas.Rectangle(R.Left - 2, R.Top - 2, R.Right + 2, R.Bottom + 2);
    DrawText(Canvas.Handle, PChar(S), -1, R, DT_WORDBREAK or DT_CENTER);
end;

//DrawText 函数输出文字，文字靠左显示，如图 16.6
procedure TfmMain.btnLeftClick(Sender: TObject);
begin
    ClearCanvas;
    Canvas.Font.Size := 10;
    R := Rect(10, 10, 80, 100);
    Canvas.Rectangle(R.Left - 2, R.Top - 2, R.Right + 2, R.Bottom + 2);
    DrawText(Canvas.Handle, PChar(S), -1, R, DT_WORDBREAK or DT_LEFT);
end;

//DrawText 函数输出文字，文字靠右显示，如图 16.7
procedure TfmMain.btnRightClick(Sender: TObject);
begin
    ClearCanvas;
    Canvas.Font.Size := 10;
    R := Rect(10, 10, 80, 100);
    Canvas.Rectangle(R.Left - 2, R.Top - 2, R.Right + 2, R.Bottom + 2);
    DrawText(Canvas.Handle, PChar(S), -1, R, DT_WORDBREAK or DT_RIGHT);
end;

end.
```

首先，这个程序中调用了 ClearCanvas()方法来清除画布上的内容。

btnTextRect 按钮演示了如何使用 TCanvas 的 TextRect()方法来输出文字。它首先判断文字的高度和宽度，并以字体大小的一半高度来显示文字，如图 16.4 所示。

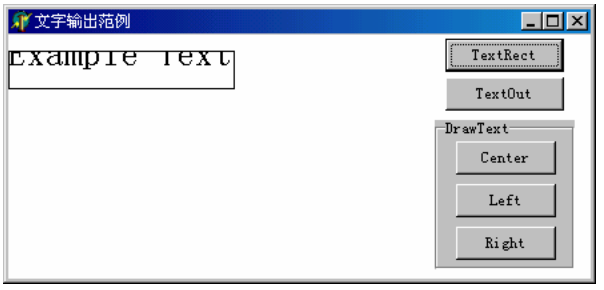

图 16.4 TextRect 方法输出文字

btnTextOut 按钮演示了如何使用 TCanvas 的 TextWidth()和 TextHeight()的方法来判断字符串的大小，如图 16.5 所示。

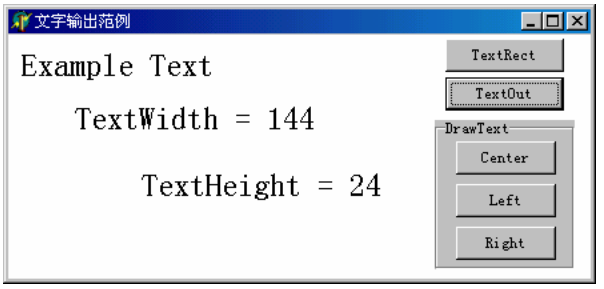

图 16.5 TextOut 方法输出文字

在这个程序中，分别使用了 3 个类似的按钮来演示 Win32 GDI 函数 DrawText()的使用方法。DrawText()函数方法没有封装进 TCanvas 中，但用户仍然可以通过 Win32 GDI 直接调用它。

3 个按钮的区别在于 DrawText()函数传递的参数有所不同，它们分别使文字在一个 Rect 中居中、靠左和靠右显示。图 16.6 和图 16.7 分别是靠左和靠右的显示情况。

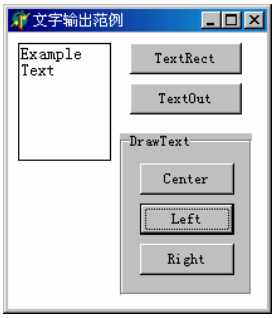

图 16.6 DrawText 函数输出文字（靠左）

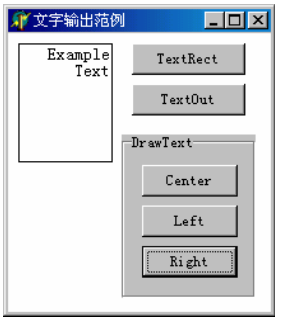

图 16.7 DrawText 函数输出文字（靠右）

查看一下 DrawText()函数的源代码，会发现，它需要传递一个设备描述表，或者叫作 DC 的参数，也就是示例中的 Canvas.Handle 参数，也就是说，设备描述表可以通过画布的 Handle 特性得到。16.2.4 小节将介绍关于设备描述表的一些内容。

16.2.4　定制图形

结合前面介绍的 TBrush 画刷的知识，读者已经了解了如何在画布上画出线条，但是，似乎用户能够操作的惟一一个属性就是 Width，由它来控制线条的粗细，仅此而已。很多时候用户可能需要画出一条定制的线条，例如一条看上去好像用书法笔画出的线条。事实上，用户可以画出任何一种想要的线条，当然仅仅知道了几个 TCanvas 特性的使用方法是远远不够的，还需要一些技巧，例如上面提到的书法笔效果。其实不只是需要画一条线，而是需要画一系列的形状，将这些形状衔接起来，就会形成各种效果，有些效果用户可能根本想象不到，这也就是图像编程的美妙之处。

画出一条像是用书法笔画出来的线条实际上并不困难，可以使用 TCanvas 的 Polygon 方法，画出一系列的平行四边形。图 16.8 所示就是这种方法产生的效果。

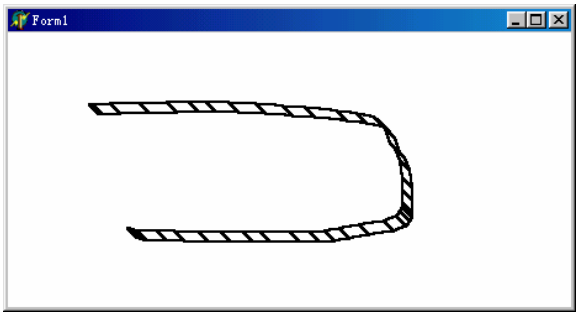

图 16.8　一系列相互衔接的多边形形成的效果

当把画笔和画刷设置为一样的颜色的时候，就会看到了定制的书法线条，如图 16.9 所示。

图 16.9　定制书法线条

下面是上面这个示例的详细的程序清单：

```
unit Un_Main;
```

```
interface

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

type
    TForm1 = class(TForm)
        Image1: TImage;
        procedure Image1MouseDown(Sender: TObject; Button: TMouseButton;
          Shift: TShiftState; X, Y: Integer);
        procedure Image1MouseMove(Sender: TObject; Shift: TShiftState; X,
          Y: Integer);
        procedure Image1MouseUp(Sender: TObject; Button: TMouseButton;
          Shift: TShiftState; X, Y: Integer);
        procedure FormCreate(Sender: TObject);
    private
        { Private declarations }
    public
        { Public declarations }
    end;

var
    Form1: TForm1;
    MouseIsDown: Boolean;
    StartPos, NextPos: TPoint;

implementation

{16R *.dfm}

procedure TForm1.Image1MouseDown(Sender: TObject; Button: TMouseButton;
    Shift: TShiftState; X, Y: Integer);
begin
    MouseIsDown := True;
    StartPos := Point(X, Y);
    NextPos := Point(X, Y);
end;

procedure TForm1.Image1MouseMove(Sender: TObject; Shift: TShiftState; X,
    Y: Integer);
var
```

```
        Points: array[1..4] of TPoint;
begin
        if MouseIsDown then begin
                NextPos := Point(X, Y);
                Points[1] := StartPos;
                Points[2] := NextPos;
                Points[3] := Point(NextPos.X + 8, NextPos.Y + 8);
                Points[4] := Point(StartPos.X + 8, StartPos.Y + 8);
                Image1.Canvas.Polygon(Points);
                StartPos := Point(X, Y);
        end;
end;

procedure TForm1.Image1MouseUp(Sender: TObject; Button: TMouseButton;
        Shift: TShiftState; X, Y: Integer);
begin
        MouseIsDown := False;
end;

procedure TForm1.FormCreate(Sender: TObject);
begin
        Image1.Canvas.Brush.Color := clBlack;
        Image1.Canvas.Pen.Color := clBlack;
end;

end.
```

程序在 FormCreate()**事件中，将画笔和画刷设置为相同的颜色。在** MouseDown()**事件中，设置变量** MouseIsDown **为** True**，表示当前鼠标为按下状态，并记录了开始绘制处的点。**

MouseMove()**事件是真正画的动作，这里实际的动作是画一系列的多边形，并且将上一个多边形的终点作为下一个多边形的起点，这样来保证多边形之间的相互衔接。**

用户可以发挥自己的想象力，利用各种不同的形状、形状之间的组合、画笔画刷的颜色以及宽度等因素，画出想要的任何样式的线条，甚至可以利用位图来填充这些形状。

16.2.5　设备描述表

设备描述表是由 Win32 **通过的句柄，用于标识应用程序通过设备驱动程序与诸如显示器、打印机、绘图仪等设备的连接。在传统的** Windows **编程中，当需要在** Windows **窗口的表面绘图时，程序员需要请求设备描述表。使用完后，必须把设备描述返回给** Windows**。**Delphi TCanvas **中封装了设备描述，从而简化了设备描述的管理。事实上，**TCanvas **能够缓存设备描述表，以备以后使用，减少向** Win32 **申请的次数，提高程序的运行速度。**

16.2.3 **小节的示例中使用了带格式输出文字的** GDI **例程** DrawText ()**。**DrawText()**需要传递以下** 5 **个参数，见表** 16.3**。**

表 16.3 DrawText()函数中的参数

参数	说明
DC	设备描述表
Str	指向要输出的文字。如果 Count 参数为–1，它必须以 NULL 结束
Count	Str 的字节数。如果为–1，则 Str 为 NULL 结束的字符串
Rect	指向 TRect 结构的指针。在这个矩形中，文本被格式化
Format	以位的形式指定 Str 字符的格式化标志

在 16.2.3 小节的这个示例中，首先用 Rect () 函数对 TRect 结构进行初始化。然后使用这个结构在用 DrawText()函数输出的文字周围画一个矩形。对于不同的方法来说，传递给 DrawText()函数的格式化标志是不同的。dt_WordBreak 和 dt_Center 标志用于使文本在指定的矩形中居中，dt_WordBreak 和 dt_Right 标志用于使文本在指定的矩形中右对齐，而 dt_WordBreak 与 dt_Left 标志则用于使文本左对齐。dt_WordBreak 能够根据矩形中的文本调整矩形的高度和宽度。

16.3 坐标系统和映射模式

大多数 GDI 函数需要指定坐标集，以此来确定绘图的位置。这些坐标基于一个度量单元，例如像素。通常，GDI 以垂直方向和水平方向上的轴来定位，也就是说，通过增减 X、Y 坐标值来移动绘图位置。Win32 依赖于两个因素来执行绘图函数：坐标系统和映射模式。

Win32 坐标系统与任何其他坐标系统没有什么区别。指定了一个点的 X、Y 坐标后，Win32 就在绘图表面定位这个点。Win32 使用 3 种坐标系统，分别是设备、逻辑和屏幕坐标系。Windows 95 不支持通用坐标系（例如位图旋转、裁剪、扭曲等）。

16.3.1 设备坐标系

设备坐标系，正如其名称所揭示的那样，是指 Win32 运行所在的设备。它以像素为单位进行度量，定位方向是水平轴从左到右、垂直轴从上到下增加。例如，如果在 640×480 的显示器上运行 Windows，设备左上角坐标为（0，0），而右下角坐标则为（639，479）。

16.3.2 逻辑坐标系

在 Win32 中，有设备描述表（或称 DC）的区域通常使用逻辑坐标系。例如，屏幕、Form、Form 的客户区。设备坐标系与逻辑坐标系的区别稍后解释。

16.3.3 屏幕坐标系

屏幕坐标系是指显示设备，它是以像素为度量单位的坐标系。在 640×480 的显示器上，Screen.Width 和 Screen.Hight 分别为“640”和“480”像素。要获取屏幕的设备描述表，可以使用 Win32API 中的 GetDC()函数。相应地，用户必须调用 RaleaseDc()函数来释放所获取的设备描述表。下面的代码演示了这一点：

```
var
```

```
        ScreenDC: HDC;
begin
        Screen DC := GetDC(0);
        try
                . . .
        finally
                releaseDC(0,ScreenDC);
        end;
end;
```

16.3.4　Form 坐标系

Form 坐标系也叫窗口坐标系，指整个 From 或窗口，包括标题栏和边框。Delphi 的 TFrom 没有直接提供访问绘图区域 DC 的特性，但用户可以通过使用 Win32 API 中的 GetWindowsDC()函数来获得 DC。例如：

```
MyDC := GetWindowsDC(form1.Handle);
```

这个函数返回给定窗口的设备描述表。

注意：用户可以通过 TCanvas 对象来操纵由 GetDC()和 GetWindowsDC()所获得的设备描述表，进而通过设备描述表来调用 TCanvas 的方法。用户只要创建一个 TCanvas 实例，然后将 GetDC()或 GetWindowsDC()的结果赋给 TCanvas.Handle 特性即可。这个方法可行，因为 TCanvas 拥有这个句柄，当画布对象释放时将释放该 DC。下面的代码说明了这个技术：

```
var
  c:TCanvas;
begin
  c := TCanvas.Create;
  try
        c.Handle:=GetDC(0);
        c.TextOut(10,10,‘Hello World’);
        finally
          c.Free;
        end;
end;
```

Form 的客户区坐标是指 From 的客户区域，其 DC 为 Form 画布的 Handle 特性，其尺寸可以由 Canvas.Client 和 Canvas.ClientHeight 获得。

16.3.5　坐标映射

为什么在调用 GDI 例程时不用设备坐标系代替逻辑坐标系呢？代码如下所示：

```
Form1.Canvas.TextOut(0, 0, ‘UpperLaftCorner of Form’);
```

上面的代码把字符串显示在 Form 的左上角。逻辑坐标系的（0，0）位置对应于 Form 设备描述表的（0，0）位置。不过，Form 的左上角（0，0）不同于设备坐标系的（0，0）。如果 Form 是在屏幕的左上角出现的，Form 的（0，0）对应于设备上完全不同的位置。

提示：通过 Win32 API 函数 ClientToScreen()或 ScreenToClient()，可以把一个基于逻辑坐标系的坐标转换为基于设备坐标系的坐标。当然，也可以用 TControl 的方法。注意，这只对屏幕的设备描述表和可视的控件有效。对于打印机或图元文件的设备描述表来说，由于不是以屏幕为参照，要把逻辑像素转换为设备像素，可以调用 Win32 的 LPtoDP()函数。反之，可以调用 DPtoLP()函数。

在调用 Canvas.TextOut()方法的背后，Win32 实际上使用的是设备坐标系。为了让 Win32 实现这一点，必须把逻辑坐标系映射成设备坐标系。通过 DC 的映射模式可以实现这一点。

使用逻辑坐标系的另一个原因是，用户可以不像在调用 GDI 例程时一样是使用像素，而是使用英寸或毫米。正如用户将看到的那样，Win32 让用户通过改变映射模式来改变度量单位。

映射模式有两个属性：一是 Win32 把逻辑单位转换为设备单位的转换器，二是 DC 的 X、Y 轴方向。

注意：在 Delphi 中，与 DC 有关的绘画例程、映射模式、方向等看起来好像不大重要，因为用户是使用 TCanvas 对象来绘画。记住，当 Win32 GDI 例程与对应的 Canvas 例程比较时，就可以明显看出 TCanvas 实际上是 DC 的外套。例如：

Canvas 例程　　Canvas.Rectangle(0,0,50,50);

GDI 例程　　Rectangle(ADC,0,0,50,50);

当使用 GDI 例程时，要传递 DC 给例程，而 Canvas 例程则使用封装的 DC。

Win32 给 DC 或 TCanvas.Handle 定义了映射模式。实际上，Win32 定义了 8 种可用的映射模式。这些影射模式及其属性列在表 16.4 中。

表 16.4　Win32 映射模式

映射模式	逻辑单位长度	方向（X，Y）
MM_ANISOTROPIC	arbitrary(x<>y)or(x=y)	definable/definable
MM_HIENGLISH	0.001inch	Right/Up
MM_HIMETRIC	0.01 mm	Right/Up
MM_ISOTROPIC	arbitrary(x=y)	definable/definable
MM_LOENGLISH	0.01 inch	Right/Up
MM_LOMETRIC	0.1 mm	Right/Up
MM_TEXT	1 pixel	Right/Down
MM_TWIPS	1/1440 inch	Right/Up

Win32 定义了一些函数，让用户改变或者获取给定 DC 的映射模式。下面列出了这些函数：

- SetMapMode()　　为给定设备描述表设置映射模式
- GetMapMode()　　获取给定设备描述表的影射模式
- SetWindowOrgEx()　　设置给定 DC 的窗口原点（0，0）
- SetWindowExtEx()　　设置给定 DC 的窗口范围
- SetViewPortExtEx()　　设置给定 DC 的视区范围

这些函数提到窗口或视区的概念。窗口或视区的作用是，Win32 GDI 通过它们可以把逻辑单位转换为设备单位。涉及窗口的函数指逻辑坐标系统，而涉及视区的函数则指设备坐标系统。除 MM_ANISOTROPIC 和 MM_ISOTROPIC 这两种映射模式以外，不必考虑这些区别。事实上，Win32 默认使用 MM_TEXT 映射模式。

注意：MM_TEXT 是默认的映射模式，逻辑坐标与设备坐标映射为 1:1。因此，除非改变了映射模式，在所有 DC 上都可使用设备坐标。有些 API 函数比较特殊。例如，字体高度总是以设备像素为单位，而不是以逻辑像素为单位。

16.3.6　设置映射模式

用户将注意到每个映射模式使用不同的逻辑单位。有些情况下，由于某种原因，使用不同的映射模式可能更方便。例如，用户可能想显示一条 2 英寸宽的线，而不管输出设备的分辨率。这时，最好使用 MM_LOENGLISH 映射模式。

假设要在 From 上面放一个 1 英寸的矩形，首先要把 Form1.Canvas.Handle 的映射模式设为 MM_HIENGLISH 或 MM_LOENGLISH，如下所示：

```
SetMapMode（Canvas.Handle, MM_LOENGLISH）;
```

然后，用适当的度量单位画一个 1 英寸的矩形。由于 MM_LOENGLISH 使用 1/100 英寸，所以用户只可以传递值“ 100 ”，如下所示：

```
Canvas.Rectangle(0,0,100,100);
```

由于 MM_TEXT 以像素作为其度量单位，所以，可以使用 Win32 API 函数 GetDeviceCaps()来获得把像素转换为英寸或毫米的消息，这样就可以执行用户想要的计算。映射模式是一个让 Win32 为用户工作的方式。然而要注意，用户永远也得不到屏幕显示的精确度量。这是因为，Windows 无法知道屏幕的显示尺寸，它只能估算。同样，在比较矮胖的显示器上，Windows 通过扩大显示比例来提高文字的可读性。例如，屏幕上为 10 磅的字与纸上 12~14 磅的字的高度是一样的。

16.3.7　设置窗口/视区范围

SetWindowExtEx()和 SetViewPortExtEx()函数可以定义 Win32 如何把逻辑单位转换为设备单位。这些函数只在窗口映射模式设为 MM_ANISOTROPIC 或 MM_ISOTROPIC 时才有效，否则无效。因此，下面的代码意味着一个逻辑单位对应着两个设备单位（像素）：

```
SetWindowExtEx(Canvas.Handle, 1, 1, nil);
SetVieportExtEx(Canvas.Handle, 2, 2, nil);
```

同样，下面的代码意味着 5 个逻辑单位需要 10 个设备单位，如下所示：

```
SetWindowExtEx(Canvas.Handle, 5, 5, nil);
SetViewportExtEx(Canvas.Handle, 10, 10, nil);
```

注意，这与前面的示例完全一样。两者的逻辑与设备单位比都为 1:2。下面的示例可以用来改变 Form 的单位，如下所示：

```
SetWindowExtEx(Canvas.Handle, 500, 500, nil);
SetWindowExtEx(Canvas.Handle, ClientWidth, ClientHeight, nil);
```

这样，不管 Form 的尺寸多大，上面的代码都能让用户工作在客户区的宽度和高度为 500×500 单位（不是像素）的 Form 上。

SetWindowOrgEx()和 SetViewportOrgEx()函数可重定位默认的原点（0，0）。此位置在 MM_TEXT 映射模式下位于 Form 客户区的左上角。通常情况下，建立坐标系只需要修改视区的原点即可。例如，下面的代码用于建立如图 16.10 所示的四象限坐标系统。

```
SetViewportOrgEx(Canvas, ClientWidth div 2, ClientHeight div 2, nil);
```

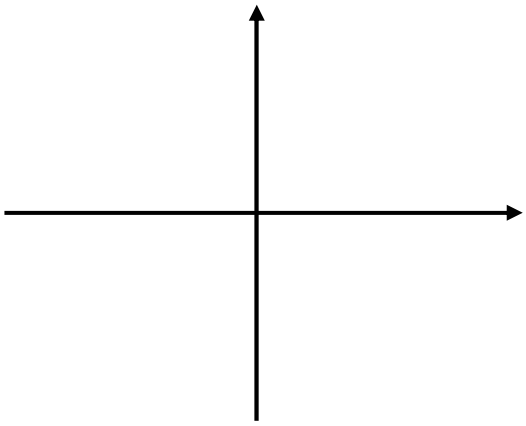

图 16.10 一个四象限坐标系统

注意：传递给 SetWindowOrgEx()、SetWindowExtEx()和 SetViewportExtEx()函数的最后一个参数是 nil。SerWindowOrgEx()和 SetViewportOrgEx()函数需要传递一个 TPoint 结构，用来保存上次设置的原点，以便必要时恢复 DC 的原点。另外，SetWindowExtEx()和 SetViewportExtEx()函数需要传递一个 TSize 结构，用于保存 DC 的原先范围。

16.4 高 级 字 体

尽管 VCL 可以方便地操纵字体，但它没有提供 Win32 API 中的字体润色功能。这一节将介绍 Win32 字体的基础以及如何操纵它们。

16.4.1　Win32 字体类型

在 Win32 中，有两种基本类型的字体：GDI 字体和设备字体。GDI 字体保存在字体资源文件中，扩展名是 fon（对光栅和矢量字体而言）或者 tot 和 ttf（对 TrueType 字体而言）。设备字体是指诸如打印机等设备而言。与 GDI 字体不同的是，当 Win32 用设备字体打印文本时，只需要把 ASCII 字符传给设备，由设备负责以某种字体打印。Win32 能够把字体转换为位图或者用 GDI 函数来画字体。与 GDI 字体一样，用位图或 GDI 函数来输出字体是比较费时的。

尽管设备字体更快，但它们只使用于某种设备，而且支持的字体有限。

16.4.2　基本字体元素

在使用 Win32 的不同字体之前，应了解与字体有关的术语和元素，它们是字体的字样、家族和尺寸。

可以把字体看作是代表字符的图片。字符有两个特征：字样和尺寸。

在 Win32 中，字样是指字体的样式和大小。在 Win32 的帮助文件中，详细定义了字样的概念以及它与字体之间的关系：“字样是共享设计特征的字符的集合”。例如，Courier 为一普通字样。所谓字样，就是具有相同字样和大小的字符的集合。

Win32 把不同字样分为 5 个家族：Decorative、Modern、Roman、Script 和 Swiss。区分这些家族的字体特征是字体的衬线和笔划宽度。

衬线是指字体主要笔划的开始或结尾处的小细线。笔划是组成字体的主要线。图 16.11 演示了这两个特征。表 16.5 中列出了字体家族中的一些典型字体。

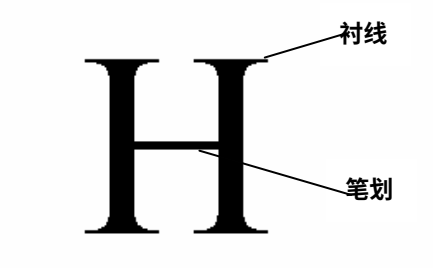

图 16.11　衬线和笔划

表 16.5　字体家族和典型字体

字体家族	典型字体
Decorative	新式字体：老式英语
Modern	有固定笔划，可能有也可能没有衬线的字体：Pice、Elite、CourierNew
Roman	有不同笔划和衬线的字体：Times New Roman、New Centary Schoolbook
Script	看起来像手写的字体：Scipt、Cursive
Swiss	有不同笔划，但没有衬线的字体：Arial、Helvetica

字体的大小以 1/72 英寸的磅来表示。字体高度由上升部分和下降部分组成。上升和下降部分分别由 tmAscent 和 tmDescent 值表示，如图 16.12 所示，图中还显示了测量字符的其他几个值。

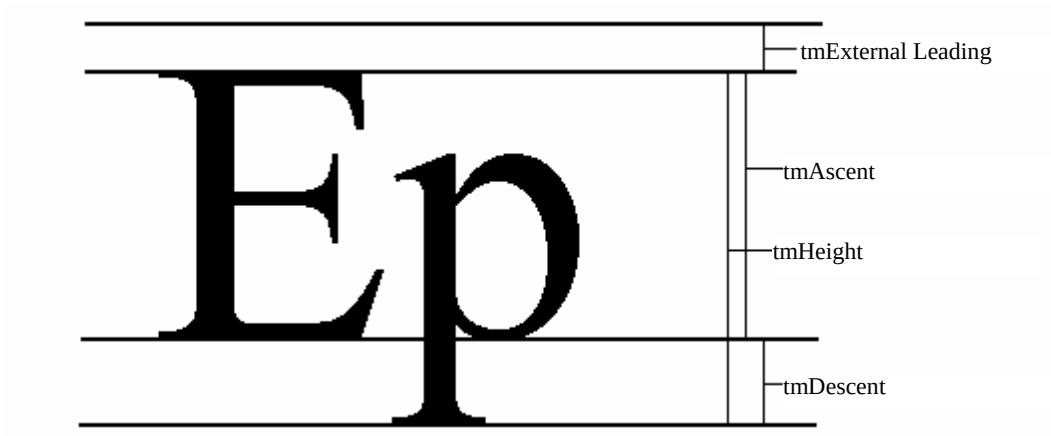

图 16.12 字符度量值

字符位于所谓的 cell（字符单元）中，字符周围的区域由空白组成。要注意的是，字符的大小可能包括字符图形（字符的可见部分）和字符单元。其他情况则可能指其中之一，见表 16.6。

表 16.6 不同字符大小的含义

大小	含义
tmExternalLeading	文本线之间的空格
tmInternalLeading	字符图形与字体单元之间的距离
tmAscent	从基准线到字符单元顶部的距离
tmDescent	从基准线到字符单元底部的距离
tmPorntSize	字符高度减去 tmlnternalLeading
tmHeight	Ascent、Descent 与 Internal Leading 的高度
tmBaseLine	字符所在的基准线

16.4.3 GDI 字体分类

GDI 字体主要有 3 种，分别是光栅字体、矢量字体和 TrueType 字体。前两种字体存在于 Win32 以前的版本中，而最后一种字体则是在 Windows 3.1 中引入的。

1．光栅字体

光栅字体是一种位图，它的分辨率、长度比和字体大小是确定的。由于这些字体以固定的尺寸提供，Win32 可以按需要的大小来合成新的字体，但只能由大字体合成小字体，反过来则不行，因为 Win32 是通过复制原始字体位图的行和列来合成字体的。合成光栅字体很方便，缺点是，当把它放大后有点不光滑。图 16.13 显示的是 Win32 系统光栅字体。

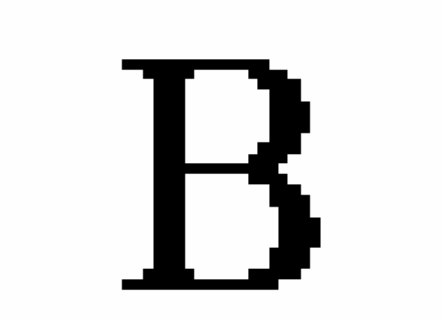

图 16.13　光栅字体

2．矢量字体

矢量字体是 Win32 通过 GDI 函数生成的一系列直线来创建的。这些字体比光栅字体有更好的延伸性，但它们的显示密度较低，有的人希望如此，而另一些人可能不希望如此。而且，矢量字体的操作也比光栅字体慢。

3．TrueType 字体

TrueType 字体可能是 3 种字体中最好的。它的优势是，TrueType 字体可以按任意大小虚拟地显示任何类型的字体，看上去较平滑。Win32 实际上是用一组点的集合来显示 TrueType 字体，并隐藏了轮廓。图 16.14 显示了一种 TrueType 字体。

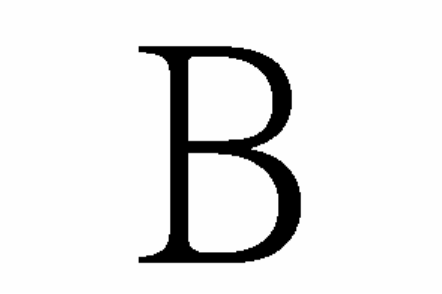

图 16.14　True Type 字体

16.4.4　显示不同字体

至此，本章已介绍了 Windows 的字体技术。如果有兴趣想进一步了解字体的话，读者可以参阅 Win32 在线帮助中的 Fonts Overview，那里会提供大量有关字体的信息。

16.5　本 章 小 结

本章介绍了很多关于 Win32 图形设备接口 GDI 的信息，讨论了 Delphi 的 TCanvas 类的特性和方法，还讨论了映射模式以及 Win32 坐标系等概念。最后，还讨论了如何创建字体和显示字体的信息。掌握了 GDI 编程技术以后，用户就可以做很多有趣的事情了。

第 17 章　多媒体编程

Delphi 强大的封装功能，使多媒体编程变得非常容易，例如 TMediaPlayer 组件，这么一个小巧的组件中，封装进了 Windows MCI（媒体控制接口）的大量函数。

本章将帮助读者学习编写一个简单但功能强大的媒体播放器，还要编写一个功能齐全的 CD 播放器。除此以外，读者还将学习到 TMediaPlayer 的用法，当然，在学习这些东西以前，读者必须确保自己的计算机上安装了多媒体设备，例如声卡和 CDROM 等。

17.1　一个简单的媒体播放器

下面就开始创建一个简单的媒体播放器，通过它读者将了解到如何在一个 Form 上放置一个 TMediaPlayer 组件来迅速建立起一个媒体播放器。程序的主 Form 如图 17.1 所示。

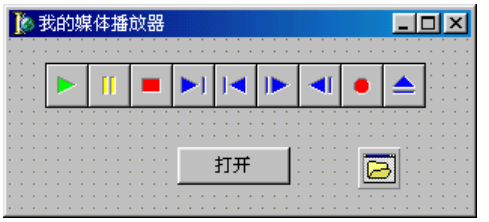

图 17.1　媒体播放器主界面

TMediaPlayer 支持所有的多媒体设备类型，但现在只想播放 WAV、AVI、MIDI 和 MP3 文件，所以，应当将 TOpenDialog 的 Filter 特性设置为：WAV; AVI; MIDI; MP3 | *.wav; *.avi; *.mid; *.mp3。播放的程序代码如下所示：

```
procedure Tfm_Main.btnOpenClick(Sender: TObject);
begin
    if OpenDialog.Execute then begin
        MediaPlayer.FileName := OpenDialog.FileName;
        MediaPlayer.Open;
    end;
end;
```

至此，一个可以播放多种媒体类型文件的播放器就做好了，本章下面的章节将针对 WAV 以及 AVI 等作更加详细地介绍。

17.2　播放 WAV 文件

WAV 文件是 Windows 中的标准声音格式，WAV 文件是二进制格式的，类似于数字声音。WAV 文件的最大好处是它符合工业标准，无论在什么地方都可以正常使用，但缺点是 WAV 文件体积太大，占用了大量的磁盘空间。

TMediaPlayer 已经可以很容易地把声音集成到应用程序中去，正如上面读者所看到的，只要简单地设置一个文件名，打开就可以播放了。

提示：如果只是需要播放 WAV 文件，用户根本没有必要使用 TMediaPlayer 组件，实际上，只需要调用 mmSystem 单元中的 PlaySound()函数就可以了，PlaySound()函数可以播放文件、内存或者应用程序连接的资源文件中的 WAV 声音。函数需要提供 3 个参数，分别是 pszSound、hMod 和 fdwSound，具体内容可以参考 Win32 Help 的帮助。

17.3　播放 AVI 文件

AVI（Audio-Video Interleave）是一种同时交换音频和视频信息的常用格式。用户可以利用前面编写的媒体播放器程序来播放一个 AVI 文件。这时候，AVI 文件将在自己的窗口中播放，用户可以通过修改 MediaPlayer 的 Display 和 DisplayRec 特性来设置图像显示的位置和大小。下面的代码是一个简单的播放 AVI 文件的示例：

```
unit un_Main;

interface

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

type
    Tfm_Main = class(TForm)
        MediaPlayer: TMediaPlayer;
        btnOpen: TButton;
        OpenDialog: TOpenDialog;
        plView: TPanel;
        procedure btnOpenClick(Sender: TObject);
        procedure MediaPlayerNotify(Sender: TObject);
    private
        { Private declarations }
    public
```

```
        { Public declarations }
    end;

var
    fm_Main: Tfm_Main;

implementation

{17R *.dfm}

procedure Tfm_Main.btnOpenClick(Sender: TObject);
begin
    if OpenDialog.Execute then begin
        MediaPlayer.FileName := OpenDialog.FileName;
        MediaPlayer.Open;
        MediaPlayer.DisplayRect := Rect(0, 0, plView.Width, plView.Height);
        MediaPlayer.Notify := True;
    end;
end;

procedure Tfm_Main.MediaPlayerNotify(Sender: TObject);
begin
    ShowMessage('Media control method executed!');
end;

end.
```

TMediaPlayer 组件的 Display 特性用来指定一个窗口，使 AVI 文件在这个指定的窗口中播放，这个窗口可以是 Form 上的任意控件（注意不是所有组件）。通过 Object Inspector 可以在 Display 特性的下拉框中找到当前 Form 上的任意控件。在上面的示例中，设置 Display 特性为一个 Panel，AVI 文件将在这个 Panel 上播放，但是 AVI 文件并没有占据 Panel 的全部空间，而是以默认的尺寸播放。

接下来，通过设置 DisplayRec 特性就可以设置文件播放的尺寸了。DisplayRec 特性是 TRect 类型，用户可以通过设置它的左、上、右、下 4 个位置参数来定义一个尺寸。例如，上面的示例中，要使 AVI 文件占据 Panel 的整个客户区，就应把 DisplayRec 特性设置为 plView 的尺寸，如下所示：

```
Rect(0, 0, plView.Width, plView.Height)
```

注意：只有在调用了 TMediaPlayer 的 Open 方法后才可以设置 Display 特性。

图 17.2 是上面的示例的显示窗口。

图 17.2　在 Panel 上播放 AVI

下面来进一步了解 TMediaPlayer 的两个事件：OnPostClick 和 OnNotify。

OnPostClick 事件非常类似于 OnClick 事件，不同的是，OnClick 事件是在用户单击某个组件时发生的，而 OnPostClick 事件只是在单击引起的动作执行后才发生的。例如，如果单击 TMediaPlayer 的 Play 按钮，将触发 OnClick 事件，但只有在媒体播放完后才会发生 OnPostClick 事件。

在 TMediaPlayer 的 Notify 特性设置为 True 的情况下，当 TMediaPlayer 的一个媒体控制方法（例如 Back、Close、Eject、Next 等）执行完毕时，就会触发 OnNotify 事件。上面的示例中，在 OnNotify 的事件句柄中触发了一个对话框，可以很清楚地了解到什么时候触发了 OnNotify 事件。

17.4　设 备 支 持

TMediaPlayer 支持大量被 MCI 支持的媒体设备，这些媒体设备可以通过 DeviceType 特性来确定。表 17.1 列出了 DeviceType 特性可能的值。

表 17.1　DeviceType 特性

设备特性	描述
dtAutoSelect	TMediaPlayer 根据要播放的文件名自动选择适当的设备类型
dtAVIVideo	AVI 文件。这些文件的扩展名为 AVI，含有音频和视频
dtCDAudio	从 CDROM 驱动器中播放的音频 CD
dtDAT	与计算机相连的 DAT（数字录音机）

（续表）

设备特性	描述
dtDigitalVideo	数字视频设备，例如数字相机
dtMMMovie	多媒体电影格式
dtOther	未确定的多媒体格式
dtOverlay	视频特技设备
dtScanner	与计算机相连的扫描仪
dtSequencer	能播放 MIDI 文件的设备
dtVCR	与计算机相连的录像机
dtVideodisc	与计算机相连的视盘播放机
dtWAVAudio	WAV 视频文件

尽管 TMediaPlayer 支持众多的媒体格式，但这里只讨论 WAV、AVI 和 CD 音频等格式，因为它们在 Windows 下是最常用的。

注意：TMediaPlayer 是从 TWinControl 继承下来的，这意味着，通过 Delphi 提供的向导，可以方便地把它封装成 ActiveX 控制。这样做的好处是可以把媒体播放器嵌入到 Web 网页中，使网页支持自定义的媒体格式。另外，使用 JavaScrip 和 VBScript，可以为在 Internet/Intranet 上运行 Web 浏览器的人编写一个 CD 播放器。

17.5 音频 CD 播放器

为了让读者更好地掌握 TMediaPlayer 组件，下面就来创建一个功能齐全的音频 CD 播放器，图 17.3 是这个程序的主界面。

图 17.3 CD 播放器主界面

17.5.1 编写 CD 播放器程序

首先，处理 Form 的 OnCreate 事件，以便打开并初始化 CD 播放器。接着，将 TMediaPlayer

的 DeviceType 特性设置为 dtAudioCD；然后调用 TMediaPlayer 的 Open()方法，对设备进行初始化，检查系统是否能够播放音频 CD。如果 Open()调用失败，则会触发 EMCIDeviceError 异常，此时将终止应用程序的运行，如下所示。

```
mpCDPlayer: TMediaPlayer;
procedure Tfm_Main.FormCreate(Sender: TObject);
begin
    try
        mpCDPlayer.Open;
    except on EMCIDeviceError do begin
        MessageDlg('初始化失败，程序将终止！', mtError, [mbOK], 0);
        Application.Terminate;
      end;
    end;
end;
```

一个 CD 播放器必须具备必要的 CD 信息显示，例如曲目数、曲目时间、当前曲目、播放进度等。17.5.2 小节将介绍如何获得这些信息。

17.5.2　获取 CD 播放信息

首先介绍一些 Windows API 提供的一些时间转换例程，它们用于获取表 17.2 中所列的 Packed 信息。这些函数都在 MMSystem.dll 中定义的，因此必须在 Uses 子句中包含 MMSystem 单元。

表 17.2　多媒体时间格式转换函数

函数	格式	返回值
mci_HMS_Hour()	tfHMS	Hours
mci_HMS_Minute()	tfHMS	Minutes
mci_HMS_Second()	tfHMS	Seconds
mci_MSF_Frame()	tfMSF	Frames
mci_MSF_Minute()	tfMSF	Minutes
mci_MSF_Second()	tfMSF	Seconds
mci_TMSF_Frame()	tfTMSF	Frames
mci_TMSF_Minute()	tfTMSF	Minutes
mci_TMSF_Second()	tfTMSF	Seconds
mci_TMSF_Track()	tfTMSF	Tracks

下面的一段函数 GetCDTotals()用于获得当前 CD 的时间长度和音轨数，这个方法中用到了 TMediaPlayer.TimeFormat 特性和时间转换例程，如下所示：

```
procedure Tfm_Main.GetCDTotals(Sender: TObject);
var
```

```
    TimeValue: longint;
    TotalLengthM: Byte;
    TotalLengthS: Byte;
    TotalTracks: integer;
begin
    mpCDPlayer.TimeFormat := tfTMSF; //设置时间格式
    TimeValue := mpCDPlayer.Length; //得到 CD 长度

    TotalLengthM := mci_msf_Minute(TimeValue); // 获得时间的分钟长度
    TotalLengthS := mci_msf_Second(TimeValue); // 获得时间的分钟长度
    TotalTracks := mci_Tmsf_Track(mpCDPlayer.Tracks); //获得音轨数目

    leTotalTrack.Text := IntToStr(TotalTracks);
    leCDTime.Text := IntToStr(TotalLengthM) + ':' + IntToStr(TotalLengths);
    pbTotalProgress.Max := (TotalLengthM * 60) + TotalLengthS;
end;
```

下面的函数 ShowCDCurrentTime()**用于获得** CD **的当前播放时间，这里也使用了时间转换例程，如下所示：**

```
procedure Tfm_Main.ShowCDCurrentTime;
var
    m: Byte;
    S: Byte;
begin
    mpCDPlayer.TimeFormat := tfMSF;
    //得到曲目的当前时间点
    //分钟
    m := mci_MSF_Minute(mpCDPlayer.Position);

    //秒
    s := mci_MSF_Second(mpCDPlayer.Position);

    leCDCurrentTime.Text := IntToStr(m) + ':' + IntToStr(s);
    pbCurrentProgress.Position := (60 * m) + s; //指示 CD 进度
end;
```

下面的函数 ShowTrackTime()**用于获得当前音轨的序号以及总时间长度，并刷新界面信息显示，这里也使用了时间转换例程，如下所示：**

```
procedure Tfm_Main.ShowTrackTime;
var
    Min, Sec: Byte;
    Len: Longint;
```

```
    CurrentTrackNo: Integer;
begin
    //得到当前曲目序号
    CurrentTrackNo := mci_Tmsf_Track(mpCDPlayer.Position);
    leCurrentTrackNo.Text := IntToStr(CurrentTrackNo);

    //得到当前曲目的时间长度
    Len := mpCDPlayer.TrackLength[mci_TMSF_track(mpCDPlayer.Position)];
    Min := mci_msf_Minute(Len);
    Sec := mci_msf_Second(Len);

    //显示当前曲目的总时间长度
    leCurrentTotalTime.Text := IntToStr(Min) + ':' + IntToStr(Sec);
    pbCurrentProgress.Max := (60 * Min) + Sec;
end;
```

下面的函数 ShowCurrentTime()**用于获得当前播放的音轨的时间，这里也用到了由** MMSystem **单元中提供的时间转换例程，如下所示：**

```
procedure Tfm_Main.ShowCurrentTime;
var
    m: Byte;
    S: Byte;
begin
    mpCDPlayer.TimeFormat := tfTMSF; //设置时间格式
    //得到曲目的当前时间点

    //分钟
    m := mci_TMSF_Minute(mpCDPlayer.Position);
    //秒
    s := mci_TMSF_Second(mpCDPlayer.Position);

    leCurrentTime.Text := IntToStr(m) + ':' + IntToStr(s);
    pbCurrentProgress.Position := (60 * m) + s; //指示当前曲目进度
end;
```

另外，为了实时地刷新当前 CD **播放的进度和状态，程序还需要一个计时器** Timer**，它每隔一秒刷新界面显示进度，如下所示：**

```
procedure Tfm_Main.TimerTimer(Sender: TObject);
begin
    ShowCurrentTime; //  刷新当前曲目的进度
    ShowCDCurrentTime;   //刷新 CD 的整体进度
end;
```

17.6 本 章 小 结

本章主要讨论了 Delphi 6.0 中的 TMediaPlayer 组件，介绍了该组件的使用方法和一些基本多媒体编程知识，从几个示例中，读者可是看到 TMediaPlayer 组件功能非常强大，但它使用起来却非常简单。

另外，本章还介绍了 WAV 音频、AVE 音频/视频和 CD 音频的常见格式。

附录 1　Delphi 函数方法参考手册

名称	类型	说明
Abort	函数	引起放弃的意外处理
Abs	函数	绝对值函数
AddExitProc	函数	将一过程添加到运行时库的结束过程表中
Addr	函数	返回指定对象的地址
AdjustLineBreaks	函数	将给定字符串的行分隔符调整为 CR/LF 序列
Align	属性	使控件位于窗口某部分
Alignment	属性	控件标签的文字位置
AllocMem	函数	在堆栈上分配给定大小的块
AllowGrayed	属性	允许一个灰度选择
AnsiCompareStr	函数	比较字符串（区分大小写）
AnsiCompareText	函数	比较字符串（不区分大小写）
AnsiLowerCase	函数	将字符转换为小写
AnsiUpperCase	函数	将字符转换为大写
Append	函数	以附加的方式打开已有的文件
ArcTan	函数	余切函数
AssignFile	函数	给文件变量赋一外部文件名
Assigned	函数	测试函数或过程变量是否为空
AutoSize	属性	自动控制标签的大小
BackgroundColor	属性	背景色
BeginThread	函数	以适当的方式建立用于内存管理的线程
BevelInner	属性	控件方框的内框方式
BevelOuter	属性	控件方框的外框方式
BevelWidth	属性	控件方框的外框宽度
BlockRead	函数	读一个或多个记录到变量中
BlockWrite	函数	从变量中写一个或多个记录
BorderStyle	属性	边界类型
BorderWidth	属性	边界宽度
Break	命令	终止 For、While、Repeat 循环语句
Brush	属性	画刷
Caption	属性	标签文字的内容
ChangeFileExt	函数	改变文件的后缀
ChDir	函数	改变当前目录
Checked	属性	确定复选框选中状态
Chr	函数	返回指定序数的字符
CloseFile	命令	关闭打开的文件
Color	属性	标签的颜色

(续表)

名称	类型	说明
Columns	属性	显示的列数
CompareStr	函数	比较字符串（区分大小写）
Concat	函数	合并字符串
Continue	命令	继续 For、While、Repeat 的下一个循环
Copy	函数	返回一字符串的子串
Cos	函数	余弦函数
Ctl3D	属性	是否具有 3D 效果
Cursor	属性	鼠标指针移入后的形状
Date	函数	返回当前的日期
DateTimeToFileDate	函数	将 Delphi 的日期格式转换为 DOS 的日期格式
DateTimeToStr	函数	将日期时间格式转换为字符串
DateToStr	函数	将日期格式转换为字符串
DayOfWeek	函数	返回星期的数值
Dec	函数	递减变量值
DecodeDate	函数	将日期格式分解为年月日
DecodeTime	函数	将时间格式分解为时、分、秒、毫秒
Delete	函数	从字符串中删除子串
DeleteFile	命令	删除文件
DiskFree	函数	返回剩余磁盘空间的大小
DiskSize	函数	返回指定磁盘的容量
Dispose	函数	释放动态变量所占的空间
DisposeStr	函数	释放字符串在堆栈中的内存空间
DitherBackground	属性	使背景色的色彩加重或减轻
DragCursor	属性	当鼠标按下时光标的形状
DragMode	属性	拖动的作用方式
DropDownCount	属性	容许的显示数据项的数目
EditMask	属性	编辑模式
Enabled	属性	是否使标签呈现打开状态
EncodeDate	函数	将年、月、日合成为日期格式
EncodeTime	函数	将时、分、秒、毫秒合成为时间格式
EndMargin	属性	末尾边缘
Eof	函数	对有类型或无类型文件测试是否到文件尾
Eoln	函数	返回文本文件的行结束状态
Erase	命令	删除外部文件
ExceptAddr	函数	返回引起当前意外的地址
Exclude	函数	从集合中删除一些元素
ExceptObject	函数	返回当前意外的索引
Exit	命令	立即从当前的语句块中退出
Exp	函数	指数函数
ExpandFileName	函数	返回包含绝对路径的字符串

（续表）

名称	类型	说明
ExtendedSelect	属性	是否允许存在选择模式。属性为 True 时，MultiSelect 才有意义
ExtractFileDir	函数	返回驱动器和路径
ExtractFileExt	函数	返回文件的后缀
ExtractFileName	函数	返回文件名
ExtractFilePath	函数	返回指定文件的路径
FileAge	函数	返回文件已存在的时间
FileClose	命令	关闭指定的文件
FileCreate	命令	用指定的文件名建立新文件
FileDateToDateTime	函数	将 DOS 的日期格式转换为 Delphi 的日期格式
FileExists	函数	检查文件是否存在
FileGatAttr	函数	返回文件的属性
FileGetDate	函数	返回文件的 DOS 日期时间标记
FileOpen	命令	用指定的存取模式打开指定的文件
FilePos	函数	返回文件的当前指针位置
FileRead	命令	从指定的文件读取
FileSearch	命令	在目录中搜索指定的文件
FileSeek	函数	改变文件的指针
FileSetAttr	函数	设置文件属性
FileSetDate	函数	设置文件的 DOS 日期时间标记
FileSize	函数	返回当前文件的大小
FileWrite	函数	对指定的文件做写操作
FillChar	函数	用指定的值填充连续字节的数
FindClose	命令	终止 FindFirst/FindNext 序列
FindFirst	命令	对指定的文件名及属性搜索目录
FindNext	命令	返回与文件名及属性匹配的下一入口
FloatToDecimal	函数	将浮点数转换为十进制数
FloatToStr	函数	将浮点数转换为字符串
FloatToText	函数	将给定的浮点数转换为文本
FloatToTextFmt	函数	将给定的浮点数转换为文本格式
Flush	函数	将缓冲区的内容刷新到输出的文本文件中
FmtLoadStr	函数	从程序的资源字符串表中装载字符串
FmtStr	函数	格式化一系列的参数，其结果以参数 Result 返回
Font	属性	设置字体
Format	函数	格式化一系列的参数并返回 Pascal 字符串
FormatBuf	函数	格式化一系列的参数
FormatDateTime	函数	用指定的格式来格式化日期和时间
FormatFloat	函数	指定浮点数格式
Frac	函数	返回参数的小数部分
FreeMem	函数	按给定大小释放动态变量所占的空间
GetDir	函数	返回指定驱动器的当前目录

（续表）

名称	类型	说明
GetHeapStatus	函数	返回内存管理器的当前状态
GetMem	函数	建立一指定大小的动态变量，并将指针指向该处
GetMemoryManager	函数	返回内存管理器的入口点
Glyph	函数	按钮上的图象
Halt	函数	停止程序的执行并返回到操作系统
Hi	函数	返回参数的高地址位
High	函数	返回参数的上限值
Hint	属性	提示信息
Int	函数	返回参数的整数部分
Include	函数	添加元素到集合中
Insert	函数	在字符串中插入子串
IntToHex	函数	将整型数转换为十六进制数
IntToStr	函数	将整型数转换为字符串
IOResult	函数	返回最新的 I/O 操作完成状态
IsValidIdent	属性	测试字符串是否为有效的标识符
Items	属性	默认显示的节点
Kind	属性	摆放样式
LargeChange	属性	最大改变值
Layout	属性	图象布局
Length	函数	返回字符串的动态长度
Lines	属性	缺省显示内容
Ln	函数	自然对数函数
Lo	函数	返回参数的低地址位
LoadStr	函数	从应用程序的可执行文件中装载字符资源
LowerCase	函数	将给定的字符串变为小写
Low	函数	返回参数的下限值
Max	属性	最大值
MaxLength	属性	最大长度
Min	属性	最小值
MkDir	命令	建立一子目录
Move	函数	从源到目标复制字节
MultiSelect	属性	允许同时选择几个数据项
Name	属性	控件的名字
New	函数	建立新的动态变量并设置一指针变量指向他
NewStr	函数	在堆栈上分配新的字符串
Now	函数	返回当前的日期和时间
Odd	函数	测试参数是否为奇数
OnActivate	事件	焦点移到窗体上时触发

（续表）

名称	类型	说明
OnClick	事件	单击窗体空白区域触发
OnDblClick	事件	双击窗体空白区域触发
OnCloseQuery	事件	使用者试图关闭窗体触发
OnClose	事件	窗体关闭后才触发
OnCreate	事件	窗体第一次创建时触发
OnDeactivate	事件	用户切换到另一应用程序触发
OnDragDrop	事件	鼠标拖放操作结束时触发
OnDragOver	事件	有其他控件从他上面移过触发
OnMouseDown	事件	按下鼠标键时触发
OnMouseUp	事件	释放鼠标键时触发
OnMouseMove	事件	移动鼠标时触发
OnHide	事件	隐藏窗体时触发
OnKeyDown	事件	按下键盘某键时触发
OnKeyPress	事件	按下键盘上的单个字符键时触发
OnKeyUp	事件	释放键盘上的某键时触发
OnPaint	事件	窗体上有新部分暴露出来触发
OnResize	事件	重新调整窗体大小触发
OnShow	事件	在窗体实际显示之前瞬间触发
Ord	函数	返回序数类的序数
OutlineStyle	属性	类型
ParamCount	函数	返回在命令行上传递给程序的参数数量
ParamStr	函数	返回指定的命令行参数
Pen	属性	画刷设置
Pi	函数	返回圆周率 Pi
Picture	属性	显示图象
PictureClosed	属性	设置 Closed 位图
PictureLeaf	属性	设置 Leaf 位图
PictureMinus	属性	设置 Minus 位图
PictureOpen	属性	设置 Open 位图
PicturePlus	属性	设置 Plus 位图
Pos	函数	在字符串中搜索子串
Pred	函数	返回先前的参数
Random	函数	返回一随机函数
Randomize	函数	用一随机数初始化内置的随机数生成器
Read	函数	对有格式的文件，读一文件组件到变量中；对文本文件，读一个或多个值到一个或多个变量中
Readln	函数	执行 Read 过程，然后跳到文件下一行
ReadOnly	属性	只读属性
ReAllocMem	函数	分配一动态变量
Rename	函数	重命名外部文件
RenameFile	函数	对文件重命名

（续表）

名称	类型	说明
Reset	函数	打开已有的文件
Rewrite	函数	建立并打开一新的文件
RmDir	函数	删除空的子目录
Round	函数	将实数值舍入为整型值
RunError	函数	停止程序的执行
ScrollBars	属性	滚动条状态
Seek	函数	将文件的当前指针移动到指定的组件上
SeekEof	函数	返回文件的文件结束状态
SeekEoln	函数	返回文件的行结束状态
SelectedColor	属性	选中颜色
SetMemoryManager	函数	设置内存管理器的入口点
SetTextBuf	函数	给文本文件指定 I/O 缓冲区
Shape	属性	显示的形状
ShowException	函数	显示意外消息与地址
Sin	函数	正弦函数
SizeOf	函数	返回参数所占的字节数
SmallChange	属性	最小改变值
Sorted	属性	是否允许排序
Sqr	函数	平方函数
Sqrt	函数	平方根函数
StartMargin	属性	开始边缘
State	属性	控件当前状态
Str	函数	将数值转换为字符串
StrAlloc	函数	给以 NULL 结束的字符串分配最大长度为−1 的缓冲区
StrBufSize	函数	返回存储在由 StrAlloc 分配的字符缓冲区的最大字符数
StrCat	函数	将一字符串附加到另一字符串尾并返回合并的字符串
StrComp	函数	比较两个字符串
StrCopy	函数	将一个字符串复制到另一个字符串中
StrDispose	函数	释放堆栈上的字符串
StrECopy	函数	将一字符串复制到另一个字符串并返回结果字符串尾部的指针
StrEnd	函数	返回指向字符串尾部的指针
Stretch	属性	自动适应控件的大小
StrFmt	函数	格式化一系列的参数
StrIComp	函数	比较两个字符串（不区分大小写）
StringToWideChar	函数	将 ANSI 字符串转换为 UNICODE 字符串
StrLCat	函数	将一字符串中的字符附加到另一字符串尾并返回合并的字符串
StrLComp	函数	以最大长度比较两个字符串
StrLCopy	函数	将一个字符串中的字符复制到另一个字符串中
StrLen	函数	返回字符串中的字符数
StrLFmt	函数	格式化一系列的参数，其结果中包含有指向目标缓冲区的指针
StrLIComp	函数	以最大长度比较两个字符串（不区分大小写）

（续表）

名称	类型	说明
StrLower	函数	将字符串中的字符转换为小写
StrMove	函数	将一个字符串中的字符复制到另一个字符串中
StrNew	函数	在堆栈上分配一个字符串
StrPas	函数	将以 NULL 结束的字符串转换为 Pascal 类的字符串
StrPCopy	函数	将 Pascal 类的字符串复制为以 NULL 结束的字符串
StrPLCopy	函数	从 Pascal 类的最大长度字符串复制为以 NULL 结束的字符串
StrPos	函数	返回一个字符串在另一个字符串中首次出现指针
StrRScan	函数	返回字符串中最后出现字符的指针
StrScan	函数	返回字符串中出现首字符的指针
StrToDate	函数	将字符串转换为日期格式
StrToDateTime	函数	将字符串转换为日期/时间格式
StrToFloat	函数	将给定的字符串转换为浮点数
StrToInt	函数	将字符串转换为整型
StrToIntDef	函数	将字符串转换为整型或默认值
StrToTime	函数	将字符串转换为时间格式
StrUpper	函数	将字符串中的字符转换为大写
Style	属性	类型选择
Suce	函数	返回后继的参数
Swap	函数	交换参数的高低地址位
Tabs	属性	标记每一项的内容
TabIndex	属性	标记索引
Text	属性	显示的文本
TextToFloat	函数	将字符串（以 NULL 结束的格式）转换为浮点数
Time	函数	返回当前的时间
TimeToStr	函数	将时间格式转换为字符串
Trim	函数	从给定的字符串中删除前导和尾部的空格及控制字符
TrimLeft	函数	从给定的字符串中删除首部的空格及控制字符
TrimRight	函数	从给定的字符串中删除尾部的空格及控制字符
Trunc	函数	将实型值截取为整型值
Truncate	函数	截去当前文件位置后的内容
UnSelectedColor	属性	未选中颜色
UpCase	函数	将字符转换为大写
UpperCase	函数	将给定的字符串变为大写
Val	函数	将字符串转换为整型值
VarArrayCreate	函数	以给定的界限和维数建立变体数组
VarArrayDimCount	函数	返回给定变体的维数
VarArrayHighBound	函数	返回给定变体数组维数的上界
VarArrayLock	函数	锁定给定的变体数组
VarArrayLowBound	函数	返回给定变体数组维数的下界
VarArrayOf	函数	返回指定变体的数组元素
VarArrayRedim	函数	通过改变上限来调整变体的大小

（续表）

名称	类型	说明
VarArrayUnlock	函数	解锁指定的变体数组
VarAsType	函数	将变体转换为指定的类型
VarCase	函数	将变体转换为指定的类型并保存他
VarClear	函数	清除指定的变体
VarCopy	函数	将指定的变体复制为指定的变体
VarFormDateTime	函数	返回包含日期时间的变体
VarIsArray	函数	测试变体是否为数组
VarIsEmpty	函数	测试变体是否为 UNASSIGNED
VarIsNull	函数	测试变体是否为 NULL
VarToDateTime	函数	将给定的变体转换为日期时间
VarType	函数	将变体转换为指定的类型并保存他
Visible	属性	控件的可见性
WantReturns	属性	为 True 时，按回车键产生一个回车符； 为 False 时，按 Ctrl+Enter 键才产生回车符
Write	命令	对有格式的文件，写一变量到文件组件中；对文本文件，写一个或多个值到文件中
Writeln	命令	执行 Write 过程，然后输出一行结束标志
WideCharLenToString	函数	将 ANSI 字符串转换为 UNICODE 字符串
WideCharLenToStrWar	函数	将 UNICODE 字符串转换为 ANSI 字符串变量
WideCharToString	函数	将 UNICODE 字符串转换为 ANSI 字符串
WideCharToStrVar	函数	将 UNICODE 字符串转换为 ANSI 字符串变量

附录 2　Win32 API 函数库

第一部分　文件相关的 API 函数

Api 函数名	函数说明	适用范围		
		W16	W98	WNT
mmioWrite	写文件	否	是	是
WriteFile	写文件	否	是	是
ExtractAssociatedIcon	从文件或相关 EXE 中获取图标句柄	否	是	是
ExtractIcon	从可执行文件中返回图标句柄	否	是	是
LZRead	从压缩文件中读入数据	是	是	是
GetPrivateProfileString	从私有文件中获取字符串	是	是	是
GetPrivateProfileInt	从私有文件中获取整数	是	是	是
UnlockFile	开锁文件	否	是	是
UnlockFileEx	开锁文件	否	是	是
LZOpenFile	打开文件	是	是	是
mmioOpen	打开多媒体文件	否	是	是
SetFileApisToOEM	设置文件 API 为 OEM 字符集	否	是	是
SetFileSecurity	设置文件或目录安全属性	否	是	是
FindFirstChangeNotification	设置文件或目录修改等待	否	是	是
SetFileTime	设置文件的 64 位时间	否	是	是
mmioSetInfo	设置文件信息	否	是	是
SetTextColor	设置文件前颜色	是	是	是
SetFilePointer	设置文件指针位置	否	是	是
SetFileAttributes	设置文件属性	否	是	是
SetFileApisToOEM	设置文件 API 为 OEM 字符集	否	是	是
SetFileSecurity	设置文件或目录安全属性	否	是	是
FindFirstChangeNotification	设置文件或目录修改等待	否	是	是
SetFileTime	设置文件的 64 位时间	否	是	是
mmioSetInfo	设置文件信息	否	是	是
SetTextColor	设置文件前颜色	是	是	是

（续表）

Api 函数名	函数说明	适用范围		
		W16	W98	WNT
SetFilePointer	设置文件指针位置	否	是	是
SetFileAttributes	设置文件属性	否	是	是
DeleteFile	删除文件	否	是	是
mmioSeek	改变当前文件位置	否	是	是
MoveFile	更名文件	否	是	是
MoveFileEx	更名文件	否	是	是
GetFileTime	返回文件 64 位时间	否	是	是
GetFileTitle	返回文件名	否	是	是
GetVolumeInformation	返回文件系统信息	否	是	是
GetFileVersionInfo	返回文件的版本信息	否	是	是
GetFullPathName	返回文件的路径名	否	是	是
GetFileInformationByHandle	返回文件信息	否	是	是
GetFileType	返回文件类型	否	是	是
GetFileAttributes	返回文件属性	否	是	是
GetShortPathName	返回文件短路径	否	是	是
mmioRead	读入文件	否	是	是
ReadFile	读文件	否	是	是
WriteFileEx	写文件	否	是	是

第二部分　打印相关的 API 函数

Api 函数名	函数说明	适用范围		
		W16	W98	WNT
AddPrinterConnection	为当前用户建立与打印机的联系	否	是	是
StartPagePrinter	开始打印机	否	是	是
StartDoc	开始打印作业	是	是	是
StartDocPrinter	开始打印作业	否	是	是
AddPrintProvidor	加入一个打印机支持器	否	是	是
AddForm	加入一个打印机窗体	否	是	是
AddPort	加入一个打印机端口	否	是	是
AddMonitor	加入一个打印机管理器	否	是	是
ShellExecute	打开或打印指定文件	否	是	是

（续表）

Api 函数名	函数说明	适用范围		
		W16	W98	WNT
ClosePrinter	关闭打开的打印机	否	是	是
WritePrinter	向打印机输出数据	否	是	是
AddPrinter	在打印机服务器上建立一个打印机	否	是	是
SetAbortProc	设备打印作业的放弃函数	否	是	是
SetPrinter	设置打印机信息	否	是	是
SetPrinterData	设置打印机配置	否	是	是
SetJob	设置打印作业信息	否	是	是
ResetPrinter	设置打印数据类型和设备模式值	否	是	是
DeletePrinterConnection	删除与打印机的连接	否	是	是
DeletePrintProcessor	删除打印机处理器	否	是	是
DeletePrinterDriver	删除打印机驱动程序	否	是	是
DeletePrinter	删除打印机服务器上的打印机	否	是	是
DeleteMonitor	删除打印机监视器	否	是	是
DeletePrintProvidor	删除打印机提供者	否	是	是
DeleteForm	删除打印机窗体层差	否	是	是
AbortPrinter	删除打印机缓冲文件	否	是	是
DeletePort	删除打印机端口	否	是	是
AddJob	启动一个打印作业	否	是	是
AdvancedDocumentProperties	进行打印机高级设置	否	是	是
PrintDlg	建立打印文本对话框	否	是	是
EnumPrintProcessors	枚举已安装的打印机处理器	否	是	是
EnumPrinterDrivers	枚举已安装的打印机驱动程序	否	是	是
EnumPorts	枚举可用打印机端口	否	是	是
EnumPrintProcessorDatatypes	枚举打印机所支持的数据类型	否	是	是
EnumForms	枚举所支持的打印机窗体	否	是	是
AbortDoc	终止一项打印作业	是	是	是
PrinterProperties	修改打印机属性	否	是	是
AddPrintProcessor	将打印处理器复制到打印机服务器中	否	是	是
AddPrinterDriver	将打印机驱动程序复制到打印机服务器中	否	是	是
PrinterMessageBox	显示打印作业出错信息	否	是	是
ConnectToPrinterDlg	显示浏览对话并连接网络打印机	否	是	是

（续表）

Api 函数名	函数说明	适用范围		
		W16	W98	WNT
EndPagePrinter	结束打印页	否	是	是
EndDoc	结束打印作业	是	是	是
EndDocPrinter	结束打印作业	否	是	是
StartPage	准备打印机接收数据	是	是	是
WaitForPrinterChange	监测打印机或打印机服务器变化	否	是	是
GetPrintProcessorDirectory	获取打印机驱动处理器路径	否	是	是
GetPrinterDriver	获取打印机驱动程序信息	否	是	是
GetPrinterDriverDirectory	获取打印机驱动程序路径	否	是	是
GetPrinter	获取打印机信息	否	是	是
GetPrinterData	获取打印机配置信息	否	是	是
GetForm	获取打印机窗口信息	否	是	是
EnumJobs	获取打印作业信息	否	是	是
GetJob	获取打印作业信息	否	是	是
OpenPrinter	获取指定打印机的句柄	否	是	是
ReadPrinter	读打印机数据	否	是	是
DocumentProperties	配置打印机设置	否	是	是
ConfigurePort	配置打印机端口	否	是	是

第三部分　其它 API 函数

Api 函数名	函数说明	适用范围		
		W16	W98	WNT
DdeImpersonateClient	DDE 服务器模拟客户机	否	是	是
timeKillEvent	中止计时器事件	否	是	是
TerminateProcess	中止进程	否	是	是
KillTimer	中止定时器	是	是	是
TerminateThread	中止线索	否	是	是
waveOutBreakLoop	中断声音输出循环	否	是	是
DdeKeepStringHandle	为字符串句柄增加可用记录	否	是	是
AllocConsole	为当前进程建立控制台	否	是	是
CreateHalftonePalette	为设备描述表建立中间色调调色板	否	是	是
CreateCaret	为系统脱字号建立新的形状	是	是	是

（续表）

Api 函数名	函数说明	适用范围		
		W16	W98	WNT
GetConsoleCP	为控制台输入获取代码页	否	是	是
GetConsoleOutputCP	为控制台输出获取代码页	否	是	是
PolyTextOut	书写字符串	否	是	是
UnpackDDElParam	从 DDE 消息 IPARAM 中获取数据	否	是	是
CreateDIBitmap	从 DIB spec 中建立位图句柄	否	是	是
CreateDIBPatternBrush	从 DIB 中建立图案刷子	是	是	是
DeleteService	从 SC MANAGER 数据库中删除服务	否	是	是
GetProfileSection	从 WIN.INI 中返回关键字和值	否	是	是
GetProfileString	从 WIN.INI 中获取字符串	是	是	是
GetProfileInt	从 WIN.INI 中获取整数	是	是	是
DeleteAce	从已存在的 ACL 中删除 ACE	否	是	是
DeleteObject	从内存删除一个对象	是	是	是
DialogBoxIndirectParam	从内存模块中建立对话框	否	是	是
CreateDialogIndirectParam	从内存模块中建立非模态对话框	否	是	是
ExtCreateRegion	从区域数据中建立一个区域	否	是	是
FindAtom	从本地原子表中返回字符串原子	是	是	是
DlgDirSelectComboBoxEx	从目录列表框中返回用户选择	是	是	是
DlgDirSelectEx	从目录列表框中返回用户选择	是	是	是
RegUnLoadKey	从记录中卸载关键字	否	是	是
GetLocaleInfo	从记录中获取本机信息	否	是	是
GlobalFindAtom	从全局原子表中返回字符串原子	是	是	是
LineTo	从当前位置画一条线	是	是	是
CreatePatternBrush	从位图中建立图案刷子	是	是	是
CreateDIBPatternBrushPt	从位图中建立逻辑刷子	否	是	是
GlobalDeleteAtom	从系统原子表中删除原子	是	是	是
GetWindowLong	从附加窗口内存中返回长型数值	是	是	是
GetWindowWord	从附加窗口内存中返回字值	是	是	是
GetMessage	从线索消息队列中返回一条消息	是	是	是
UnhookWindowsHookEx	从钩子链中删除函数	是	是	是
ChangeClipboardChain	从剪贴板查看窗口上删除一个窗口	是	是	是
ExcludeUpdateRgn	从剪裁区中排斥更新区域	是	是	是

（续表）

Api 函数名	函数说明	适用范围		
		W16	W98	WNT
HeapAlloc	从堆中分配内存	否	是	是
HeapReAlloc	从堆中重分配内存	否	是	是
LocalAlloc	从堆分配内存	是	是	是
RegDeleteValue	从登录关键字中删除一个值	否	是	是
GetProp	从窗口属性表列中返回数据句柄	是	是	是
ClearCommError	允许出错后进行通讯	否	是	是
Escape	允许访问设备	是	是	是
ExtEscape	允许访问私有设备	否	是	是
AllocateLocallyUniqueId	分配 LUID	否	是	是
CreatePrivateObjectSecurity	分配并初始化保护 SD	否	是	是
AllocateAndInitializeSid	分配和初始化 SID	否	是	是
TlsAlloc	分配线索本地存储索引	否	是	是
DisconnectNamedPipe	切断命名管道的服务器终端	否	是	是
DdeClientTransaction	开始 DDE 数据事务	否	是	是
midiInStart	开始 MIDI 输入设备	否	是	是
BeginPath	开始一个路径等级	是	是	是
WNetConnectionDialog	开始网络连接对话框	否	是	是
StartService	开始运行服务	否	是	是
ResumeThread	开始暂停的线索	否	是	是
LocalUnlock	开锁本地内存块	是	是	是
GlobalUnlock	开锁全局内存块	是	是	是
VirtualUnlock	开锁虚拟页	否	是	是
UnlockServiceDatabase	开锁数据库	否	是	是
CompareFileTime	比较两个 64 位文件时间	否	是	是
DdeCmpStringHandles	比较两个 DDE 字符串句柄	否	是	是
lstrcmp	比较两个字符串	是	是	是
lstrcmpi	比较两个字符串	是	是	是
CompareString	比较两个局部指定字符串	否	是	是
EqualRgn	比较两区域是否相等	是	是	是
ScrollDC	水平或垂直移动矩形	是	是	是
IntersectRect	计算两矩形交叠处	是	是	是

（续表）

Api 函数名	函数说明	适用范围		
		W16	W98	WNT
AdjustWindowRect	计算所需窗口矩形的大小	是	是	是
AdjustWindowRectEx	计算所需窗口矩形的大小	是	是	是
LineDDA	计算线中的连续点	否	是	是
RegFlushKey	写入关键字	否	是	是
waveOutWrite	写入声音输出设备	否	是	是
ReportEvent	写入事件记录项目	否	是	是
WriteConsole	写控制台屏幕缓冲区	否	是	是
WriteTapemark	写磁带标记	否	是	是
EnterCriticalSection	加入临界部分	否	是	是
InsertMenu	加入新菜单	是	是	是
EscapeCommFunction	发送扩展 COMM 函数	是	是	是
midiOutLongMsg	发送系统专用 MIDI 消息	否	是	是
Shell_NotifyIcon	发送修改任务栏图标	否	是	是
TranslateMDISysAccel	处理 MDI 键盘加速器	是	是	是
TranslateAccelerator	处理加速关键字	是	是	是
joySetCapture	对指定窗口捕获操纵杆消息	否	是	是
OpenClipboard	打开 CLIPBOARD	是	是	是
midiInOpen	打开 MIDI 设备	否	是	是
midiStreamOpen	打开 MIDI 流	否	是	是
midiOutOpen	打开 MIDI 输出设备	否	是	是
OpenDriver	打开可安装驱动程序	是	是	是
RegOpenKey	打开关键字	否	是	是
RegOpenKeyEx	打开关键字	否	是	是
OpenProcessToken	打开过程令牌对象	否	是	是
waveInOpen	打开声音输入设备	否	是	是
waveOutOpen	打开声音输出设备	否	是	是
OpenEvent	打开事件对象	否	是	是
OpenEventLog	打开事件登记句柄	否	是	是
OpenMutex	打开命名 MUTEX 对象	否	是	是
OpenFileMapping	打开命名文件的映像对象	否	是	是
OpenSemaphore	打开命名信号量对象	否	是	是

（续表）

Api 函数名	函数说明	适用范围		
		W16	W98	WNT
OpenBackupEventLog	打开备份事件句柄	否	是	是
OpenService	打开服务	否	是	是
OpenThreadToken	打开线索令牌对象	否	是	是
mixerOpen	打开混合设备	否	是	是
CreateBitmapIndirect	用 BITMAP 结构建立位图	是	是	是
DdeInitialize	用 DDEML 登记应用程序	否	是	是
CreateFontIndirect	用 LOGFONT 结构建立字体	是	是	是
CreatePenIndirect	用 LOGPEN 结构建立画笔	是	是	是
CreateRectRgnIndirect	用 RECT 结构建立一个区域	是	是	是
EnumMetaFile	用 WINDOWS 图元文件返回 GDI 调用	否	是	是
mciGetDeviceIDFromElementID	用元素引用返回设备 ID	否	是	是
CheckDlgButton	用对话按钮修改复选标记	是	是	是
ExtFloodFill	用当前刷子填充区域	是	是	是
FloodFill	用当前刷子填充区域	是	是	是
PaintRgn	用设备描述表中的刷子填充区域	是	是	是
FillRgn	用刷子填充区域	是	是	是
CheckRadioButton	用单选按钮放置一个复选标记	是	是	是
FrameRect	用指定刷子画一个窗口边框	否	是	是
FillRect	用指定刷子填充矩形区域	是	是	是
CreateCursor	用指定的尺寸建立一个光标	是	是	是
OffsetRgn	用指定偏移量移动区域	是	是	是
CreateBrushIndirect	用指定属性建立一个刷子	是	是	是
CreateSolidBrush	用指定颜色建立实心刷子	是	是	是
FlushViewOfFile	用映像视图填充文件	否	是	是
OffsetRect	用偏移量移动矩形	是	是	是
DrawFocusRect	用焦点风格画矩形	是	是	是
CreateDirectoryEx	用模板属性建立一个目录	否	是	是
EnumEnhMetaFile	用增强图元文件返回 GDI 调用	否	是	是
SetWinMetaFileBits	由图元文件数据建立增强图元文件	否	是	是
mouse_event	记录鼠标事件	否	是	是
SwapMouseButton	交换鼠标按钮功能	是	是	是

（续表）

Api 函数名	函数说明	适用范围		
		W16	W98	WNT
Beep	产生一个声调	否	是	是
MessageBeep	产生声音	是	是	是
PrivilegedServiceAuditAlarm	产生特权系统服务声音警报	否	是	是
BackupWrite	产生磁带备份写	否	是	是
BackupSeek	产生磁带备份查寻	否	是	是
BackupRead	产生磁带备份读	否	是	是
PlgBlt	传输像素	否	是	是
FindCloseChangeNotification	光闭文件或目录修改等待	否	是	是
CloseDriver	光闭可装入的多媒体驱动程序	是	是	是
DeregisterEventSource	光闭事件句柄	否	是	是
CloseEventLog	光闭事件记录句柄	否	是	是
CloseFigure	光闭路径中的一个数	否	是	是
CloseEnhMetaFile	光闭增强型图元文件 DC	否	是	是
StrokeAndFillPath	关闭、填充路径	否	是	是
midiInClose	关闭 MIDI 输入设备	否	是	是
midiOutClose	关闭 MIDI 输出设备	否	是	是
mmioClose	关闭 MM 文件	否	是	是
CloseServiceHandle	关闭 Service control manager 对象	否	是	否
ExitWindows	关闭 WINDOWS	是	是	是
ExitWindowsEx	关闭 WINDOWS	否	是	是
CloseMetaFile	关闭 WINDOWS 图元文件 DC	是	是	是
LZClose	关闭文件	是	是	是
midiStreamClose	关闭打开的 MIDI 流	否	是	是
CloseHandle	关闭打开的对象句柄	否	是	是
waveInClose	关闭声音输入设备	否	是	是
waveOutClose	关闭声音输出设备	否	是	是
InitiateSystemShutdown	关闭系统	否	是	是
FindClose	关闭查找文件描述表	否	是	是
CloseClipboard	关闭剪贴板	是	是	是
mixerClose	关闭混合设备	否	是	是
RegCloseKey	关闭登录关键字	否	是	是

（续表）

Api 函数名	函数说明	适用范围		
		W16	W98	WNT
UnionRect	关联两个矩形	是	是	是
WNetOpenEnum	列出网络资源	否	是	是
EnumResourceLanguages	列出语言资源	否	是	是
EnumResourceNames	列出资源名称	否	是	是
EnumResourceTypes	列出资源类型	否	是	是
RegEnumKey	列举指定关键字的子关键字	否	是	是
RegEnumKeyEx	列举指定关键字的子关键字	否	是	是
RegEnumValue	列举指定关键字的值	否	是	是
BeginDeferWindowPos	创建一个窗口位置结构	否	是	是
HeapCompact	压缩内存堆	否	是	是
BuildCommDCB	向 DCB 中传送设备定义字符串	是	是	是
DdeAddData	向 DDE 数据对象中加入数据	否	是	是
mmioSendMessage	向 I/O 过程发送消息	否	是	是
mciSendString	向 MCI 设备发出一条命令字符串	否	是	是
mciSendCommand	向 MCI 设备发出一条命令消息	否	是	是
midiInAddBuffer	向 MIDI 设备发送输入缓冲	否	是	是
midiInMessage	向 MIDI 设备驱动程序发送消息	否	是	是
midiOutMessage	向 MIDI 设备驱动程序发送消息	否	是	是
midiStreamOut	向 MIDI 流发送数据	否	是	是
midiOutShortMsg	向 MIDI 输出设备发送短消息	否	是	是
SendDriverMessage	向可安装驱动程序发送消息	是	是	是
SendDlgItemMessage	向对话框控件发送消息	是	是	是
waveInMessage	向声音输入设备发送信息	否	是	是
waveInAddBuffer	向声音输入设备发送缓冲区	否	是	是
waveOutMessage	向声音输出设备发送消息	否	是	是
WritePrivateProfileString	向私有 INI 文件输出字符串	是	是	是
WritePrivateProfileSection	向私有 INI 文件输出数据	否	是	是
ControlService	向服务器发送控件	否	是	是
PostThreadMessage	向线索发出消息	否	是	是
OutputDebugString	向调试发送字符串	是	是	是
FatalExit	向调试者返回控件	是	是	是

（续表）

Api 函数名	函数说明	适用范围		
		W16	W98	WNT
WriteConsoleOutputCharacter	向控制台写入字符串	否	是	是
WriteConsoleOutputAttribute	向控制台写入属性字符串	否	是	是
GenerateConsoleCtrlEvent	向控制台进程组发送信号	否	是	是
mixerMessage	向混合设备发送消息	否	是	是
DispatchMessage	向窗口发送消息	是	是	是
SendMessage	向窗口进程发送消息	是	是	是
SendMessageCallback	向窗口进程发送消息	否	是	是
SendMessageTimeout	向窗口进程发送消息	否	是	是
SendNotifyMessage	向窗口进程发送消息	否	是	是
auxOutMessage	向输出设备发送消息	否	是	是
GdiComment	向增强型图元文件中加入注释	否	是	是
SetClipboardViewer	在 CLIPBOARD 浏览器链中加入窗口	是	是	是
FrameRgn	在区域四周画出边框	是	是	是
CreateRemoteThread	在另一进程中建立线索	否	是	是
BeginUpdateResource	在可执行文件中开始资源文件更新	否	是	是
EndUpdateResource	在可执行文件中结束资源更新	否	是	是
DdeQueryNextServer	在对话表列中获得下一个句柄	否	是	是
SetWindowText	在目录标题或控制窗口中设置窗口文本	是	是	是
TransmitCommChar	在传输队列中加入字符	是	是	是
BitBlt	在设备描述表间复制位图	是	是	是
InvalidateRect	在更新区域中加入一个矩形	是	是	是
InvalidateRgn	在更新区域加入一个区域	是	是	是
GlobalAddAtom	在系统原子表中加入字符串	是	是	是
ReadProcessMemory	在进程中读内存	否	是	是
DrawIcon	在指定设备描述表中画一个图标	是	是	是
GrayString	在指定位置画灰色文本	否	是	是
WriteProcessMemory	在指定进程中写内存	否	是	是
EndPaint	在指定窗口中标记图画结尾标志	是	是	是
DrawText	在矩形中画出已格式化文本	是	是	是
ExtTextOut	在矩形区域中输出一个字符串	是	是	是
GlobalAlloc	在堆中分配内存	是	是	是

（续表）

Api 函数名	函数说明	适用范围		
		W16	W98	WNT
AppendMenu	在菜单中加入新的命令	是	是	是
FindResource	在模块中寻找资源	是	是	是
FindResourceEx	在模块中寻找资源	否	是	是
CallNamedPipe	多管道操作	否	是	是
VerInstallFile	安装文件	否	是	是
SetTimer	安装系统定时器	否	是	是
SetWindowsHook	安装钩子过程	否	是	是
SetWindowsHookEx	安装钩子过程	否	是	是
ReadFileEx	异步读文件	否	是	是
mciExecute	执行 MCI 设备命令	否	是	是
SHFileOperation	执行系统文件对象的操作	否	是	是
ExpandEnvironmentStrings	扩充环境变量字符串	否	是	是
InterlockedExchange	自动交换 32 位数值	否	是	是
StrokePath	行使路径	否	是	是
SetAclInformation	设备 ACL 信息	否	是	是
EnableWindow	设备窗口使能状态	是	是	是
SetSecurityDescriptorDacl	设置 DACL 信息	否	是	是
SetDIBitsToDevice	设置 DIB 位到设备	是	是	是
midiOutSetVolume	设置 MIDI 输出设备卷	否	是	是
SetSecurityDescriptorSacl	设置 SACL 信息	否	是	是
SetSecurityDescriptorGroup	设置 SD 主组信息	否	是	是
SetSecurityDescriptorOwner	设置 SD 所有者	否	是	是
TlsSetValue	设置 TLS 值	否	是	是
SetKernelObjectSecurity	设置内核对象安全属性	否	是	是
SetTextAlign	设置文本对齐标志	是	是	是
SetEndOfFile	设置文本尾指针	否	是	是
timeSetEvent	设置计时回调事件	否	是	是
timeBeginPeriod	设置计时器分辨率	否	是	是
SetWorldTransform	设置世界传送	否	是	是
SetTokenInformation	设置令牌信息	否	是	是
SetErrorMode	设置出错模式	是	是	是

（续表）

Api 函数名	函数说明	适用范围		
		W16	W98	WNT
SetHandleCount	设置可用文件句柄	是	是	是
SetDlgItemText	设置对话框标题或项目	是	是	是
SetLocalTime	设置本地时间	否	是	是
SHAppBarMessage	设置任务栏消息	否	是	是
SetPriorityClass	设置优先级类	否	是	是
waveOutSetPlaybackRate	设置回放率	否	是	是
SetPolyFillMode	设置多边形填充	是	是	是
SetMapperFlags	设置字体映像标志	是	是	是
SetTextCharacterExtra	设置字符间隔	是	是	是
SetUserObjectSecurity	设置安全描述值	否	是	是
PulseEvent	设置并复位事件	否	是	是
SetComputerName	设置当前计算机名称	否	是	是
SetBrushOrgEx	设置当前刷子的起点	否	是	是
SetROP2	设置当前绘图模式	是	是	是
SetBkColor	设置当前背景色	是	是	是
WidenPath	设置当前路径	否	是	是
SetLastError	设置扩展出错代码	否	是	是
SetLastErrorEx	设置扩展出错代码及类型	否	是	是
mciSetYieldProc	设置过程地址	否	是	是
SetDIBits	设置位图位	是	是	是
SetStretchBltMode	设置位图拉伸模式	是	是	是
SetBitmapBits	设置位图的值	是	是	是
SetBitmapDimensionEx	设置位图的宽和高	是	是	是
SetTimeZoneInformation	设置时区	否	是	是
SetSystemTime	设置系统时间和日期	否	是	是
SetSystemPaletteUse	设置系统调色板状态色	否	是	是
SetSysColors	设置系统颜色	是	是	是
SetProcessShutdownParameters	设置进程关闭参数	否	是	是
SetClassLong	设置附加类内存长数值	是	是	是
SetClassWord	设置附加类内存字数值	是	是	是
SetWindowWord	设置附加窗口内存字值	是	是	是

（续表）

Api 函数名	函数说明	适用范围		
		W16	W98	WNT
auxSetVolume	设置附属设备卷	否	是	是
SetEvent	设置事件对象	否	是	是
SetVolumeLabel	设置卷标	否	是	是
SetGraphicsMode	设置图形模式	否	是	是
SetICMMode	设置图形颜色匹配	否	是	是
waveOutSetPitch	设置波形输出强度	否	是	是
SetEnvironmentVariable	设置环境变量	否	是	是
SetArcDirection	设置画弧方向	否	是	是
SetThreadLocale	设置线索本地信息	否	是	是
SetViewportExtEx	设置视口宽度	是	是	是
SetViewportOrgEx	设置视口起点	是	是	是
SetMailslotInfo	设置信箱读时间	否	是	是
SetConsoleTextAttribute	设置屏幕文本属性	否	是	是
SetThreadContext	设置指定线索描述表	否	是	是
SetMapMode	设置映像模式	是	是	是
SetStdHandle	设置标准设备句柄	否	是	是
SetRect	设置矩形大小	是	是	是
SetBkMode	设置背景模式	是	是	是
SetDebugErrorLevel	设置调试事件出错等级	否	是	是
SetCommState	设置通讯设备状态	是	是	是
SetCommMask	设置通讯事件屏蔽	否	是	是
SetupComm	设置通讯参数	否	是	是
SetCommTimeouts	设置通讯读写时间范围	否	是	是
SetClipboardData	设置剪贴板中数据	是	是	是
SetPaletteEntries	设置彩色调色板和标志	是	是	是
SetConsoleCursorInfo	设置控制台光标大小	否	是	是
SetConsoleCursorPosition	设置控制台光标位置	否	是	是
SetConsoleCtrlHandler	设置控制台进程的单个句柄	否	是	是
SetConsoleWindowInfo	设置控制台窗口大小	否	是	是
SetConsoleTitle	设置控制台窗口标题字符串	否	是	是
SetConsoleCP	设置控制台输入代码页	否	是	是

（续表）

Api 函数名	函数说明	适用范围		
		W16	W98	WNT
SetConsoleMode	设置控制台输入输出模式	否	是	是
SetConsoleOutputCP	设置控制台输出代码页	否	是	是
SetMiterLimit	设置斜面接合长度	否	是	是
SetCaretBlinkTime	设置脱字号闪烁时间	是	是	是
SetCaretPos	设置脱字号位置	是	是	是
RegSetKeySecurity	设置登录关键字的安全属性	否	是	是
SetWindowPos	设置窗口大小、位置、顺序	是	是	是
SetWindowLong	设置窗口附加内存长型数值	是	是	是
SetWindowPlacement	设置窗口显示状态及最小/最大位置	是	是	是
SetWindowExtEx	设置窗口宽度	是	是	是
SetMenu	设置窗口菜单	是	是	是
SetWindowOrgEx	设置窗口源起点	是	是	是
SetForm	设置窗体信息	否	是	是
SetUnhandledExceptionFilter	设置筛选器异常函数	否	是	是
GdiSetBatchLimit	设置缓冲 GDI 函数数量	否	是	是
SetDeviceGammaRamp	设置辉等级	否	是	是
SetPixel	设置像素颜色	是	是	是
SetPixelV	设置像素颜色	否	是	是
SetScrollPos	设置滚动条位置	是	是	是
SetScrollRange	设置滚动条最大和最小位置	是	是	是
waveOutSetVolume	设置输出音量	否	是	是
SetKeyboardState	设置键盘状态表	是	是	是
SetFocus	设置键盘焦点	是	是	是
SetDoubleClickTime	设置鼠标双击时间	是	是	是
SetCursorPos	设置鼠标指针位置	是	是	是
SetCapture	设置鼠标捕获	是	是	是
SetTapeParameters	设置磁带机和介质信息	否	是	是
SetTapePosition	设置磁带位置	否	是	是
SetNamedPipeHandleState	设置管道读取/阻塞模式，控制局部缓存	否	是	是
SetICMProfile	设置颜色外观	否	是	是
SetColorSpace	设置颜色空间	否	是	是

（续表）

Api 函数名	函数说明	适用范围		
		W16	W98	WNT
joySetThreshold	设置操纵杆运动临界值	否	是	是
DdeAccessData	访问 DDE 数据对象	否	是	是
ObjectOpenAuditAlarm	访问对象时产生审查/警报	否	是	是
GetSystemMenu	访问系统菜单	是	是	是
SubtractRect	两个矩形相减	否	是	是
InitializeSid	初始化 SID	否	是	是
InitAtomTable	初始化本地原子杂凑表	是	是	是
LZInit	初始化压缩数据结构	是	是	是
InitializeSecurityDescriptor	初始化安全描述	否	是	是
InitializeCriticalSection	初始化临界段对象	否	是	是
DeleteAtom	删除一个原子	是	是	是
ValidateRgn	删除区域	是	是	是
ObjectCloseAuditAlarm	删除对象时产生审查/警报	否	是	是
RemoveDirectory	删除目录	否	是	是
RegDeleteKey	删除关键字	否	是	是
RemoveFontResource	删除字体资源	是	是	是
DeleteDC	删除设备描述表	是	是	是
DeleteCriticalSection	删除临界部分	否	是	是
DeleteColorSpace	删除指定色彩空间	否	是	是
ValidateRect	删除矩形	是	是	是
DestroyPrivateObjectSecurity	删除被保护的服务器对象的 SD	否	是	是
DeleteMenu	删除菜单	是	是	是
RemoveMenu	删除菜单和弹出式菜单	是	是	是
RemoveProp	删除属性列入口	是	是	是
UnregisterClass	删除窗口类	是	是	是
UnhookWindowsHook	删除筛选函数	否	是	是
UnloadKeyboardLayout	删除键盘布置	否	是	是
EraseTape	删除磁带的指定段	否	是	是
BroadcastSystemMessage	广播系统消息	否	是	是
CascadeWindows	层叠窗口	否	是	是
ChangeMenu	改变菜单	否	是	是

（续表）

Api 函数名	函数说明	适用范围		
		W16	W98	WNT
CheckMenuRadioItem	选中可选菜单	否	是	是
ChoosePixelFormat	选择像素格式	否	是	否
CloseDesktop	关闭桌面	否	是	是
CloseWindowStation	关闭窗口	否	是	是
CommConfigDialog	公用配置窗口	否	是	是
CopyImage	复制图片	否	是	是
CreateDesktop	创建桌面	否	是	是
CreateIoCompletionPort	创建 IO 端口	否	是	是
DescribePixelFormat	描述像素格式	否	是	是
DisableThreadLibraryCalls	关闭线程库调用	否	是	是
DoEnvironmentSubst	设置环境子集	否	是	是
DragDetect	拖拉监视	否	是	是
DragObject	拖拉对象	否	是	是
DrawAnimatedRects	绘制动画区域	否	是	是
DrawCaption	绘制标题	否	是	是
DrawEdge	绘制边框	否	是	是
DrawFrameControl	绘制框架控制	否	是	是
DrawIconEx	绘制图表	否	是	是
DrawState	绘制状态	否	是	是
DrawTextEx	绘制文字	否	是	是
DuplicateIcon	复制图表	否	是	是
EnumCalendarInfo	枚举日历信息	否	是	是
EnumDesktopWindows	枚举桌面窗口	否	是	是
EnumDesktops	枚举桌面	否	是	是
EnumPrinterPropertySheets	枚举打印机属性表	否	是	是
EnumPrinters	枚举打印机	否	是	是
EnumWindowStations	枚举窗口位置	否	是	是
ExtractIconEx	提取图表	否	是	是
FindClosePrinterChangeNotification	查找关闭打印机状态变化的通知	否	是	是
FindEnvironmentString	查找环境字符串	否	是	是

（续表）

Api 函数名	函数说明	适用范围		
		W16	W98	WNT
FindFirstPrinterChangeNotification	查找第一个打印机状态变化通知	否	是	是
FindNextPrinterChangeNotification	查找下一个打印机状态变化通知	否	是	是
FindWindowEx	查找窗口	否	是	是
FixBrushOrgEx	维护画刷组织	否	是	是
FreeEnvironmentStrings	释放环境字符串	否	是	是
FreeLibraryAndExitThread	释放库并退出线程	否	是	是
FreeResource	释放资源	否	是	是
GetBrushOrgEx	得到画刷组织	否	是	是
GetCommConfig	得到公用配置	否	是	是
GetCompressedFileSize	得到压缩文件长度	否	是	是
GetCurrencyFormat	得到当前格式	否	是	是
GetDIBColorTable	得到 DIB 颜色表	否	是	是
GetDefaultCommConfig	得到缺省公用配置	否	是	是
GetHandleInformation	得到句柄信息	否	是	是
GetKeyboardLayout	得到键盘布局	否	是	是
GetKeyboardLayoutList	得到键盘布局列表	否	是	是
GetMenuContextHelpId	得到菜单上下文帮助	否	是	是
GetMenuDefaultItem	得到菜单缺省命令项	否	是	是
GetMenuItemInfo	得到菜单命令信息	否	是	是
GetMenuItemRect	得到菜单项区域	否	是	是
GetNumberFormat	得到数字格式	否	是	是
GetPixelFormat	得到像素格式	否	是	是
GetProcessHeaps	得到进程堆	否	是	是
GetProcessWorkingSetSize	得到进程工作集尺寸大小	否	是	是
GetQueuedCompletionStatus	得到序列中的完成状态	否	是	是
GetScrollInfo	得到滚动条信息	否	是	是
GetStringTypeEx	得到字符串类型	否	是	是
GetSysColorBrush	得到系统颜色画刷	否	是	是
GetSystemTimeAdjustment	得到系统时间调整	否	是	是
GetTextCharset	得到文本字符集	否	是	是
GetUserObjectInformation	得到用户对象信息	否	是	是

（续表）

Api 函数名	函数说明	适用范围		
		W16	W98	WNT
GetWindowContextHelpId	得到窗口上下文帮助	否	是	是
GlobalCompact	全局压缩	否	是	是
InsertMenuItem	插入菜单命令	否	是	是
IsTextUnicode	文字 Unicode 格式	否	是	是
LoadCursorFromFile	装载光标	否	是	是
LoadImage	装载图像	否	是	是
LookupAccountSid	把 SID 转为帐户名	否	是	是
CharToOem	把字符串转换成 OEM 字符	否	是	是
CharToOemBuff	把字符串转换成 OEM 字符	否	是	是
LookupAccountName	把帐户名转为 SID	否	是	是
SetParent	改变父窗口	是	是	是
ModifyWorldTransform	改变世界变换式	否	是	是
SetCurrentDirectory	改变当前目录	否	是	是
SetConsoleScreenBufferSize	改变屏幕缓冲区大小	否	是	是
SetConsoleActiveScreenBuffer	改变显示屏幕缓冲区	否	是	是
InflateRect	改变矩形大小	是	是	是
ExcludeClipRect	改变剪裁区	是	是	是
ModifyMenu	改变菜单	是	是	是
MoveWindow	改变窗口位置及大小	是	是	是
SetCursor	改变鼠标指针	是	是	是
RegReplaceKey	更改关键字	否	是	是
ResetDC	更新设备描述表	否	是	是
SetServiceStatus	更新服务状态	否	是	是
RedrawWindow	更新客户窗口	是	是	是
UpdateResource	更新资源	否	是	是
DeferWindowPos	更新窗口位置结构	是	是	是
EndDeferWindowPos	更新窗口的位置及大小	是	是	是
UpdateWindow	更新窗口的客户区域	是	是	是
UpdateColors	更新颜色	是	是	是
PlayMetaFileRecord	运行 WINDOWS 图元文件记录	是	是	是

（续表）

Api 函数名	函数说明	适用范围		
		W16	W98	WNT
PlayMetaFile	运行 WINDOWS 图元文件到 DC	是	是	是
Netbios	运行指定 NCB	否	是	是
WinExec	运行程序	是	是	是
PlayEnhMetaFile	运行增强图元文件	否	是	是
PlayEnhMetaFileRecord	运行增强图元文件记录	否	是	是
EnumClipboardFormats	返回 CLIPBOARD 格式	是	是	是
GetCommState	返回 COMM 设备控制块	否	是	是
GetCommProperties	返回 COMM 设备属性	否	是	是
GetCommTimeouts	返回 COMM 设备超时特性值	否	是	是
GetCommMask	返回 COMM 事件屏蔽	否	是	是
GetSidIdentifierAuthority	返回 ID 字段地址	否	是	是
mciGetErrorString	返回 MCI 出错代码的文本描述	否	是	是
midiInGetErrorText	返回 MIDI 出错代码的文本描述	否	是	是
midiInGetID	返回 MIDI 设备句柄 ID	否	是	是
midiInGetNumDevs	返回 MIDI 设备数量	否	是	是
midiOutGetErrorText	返回 MIDI 输出出错文本	否	是	是
midiOutGetID	返回 MIDI 输出设备 ID	否	是	是
midiOutGetVolume	返回 MIDI 输出设备卷	否	是	是
midiOutGetDevCaps	返回 MIDI 输出设备性能	否	是	是
mmioGetInfo	返回 MM 文件信息	否	是	是
mmsystemGetVersion	返回 MM 软件版本	否	是	是
GetSecurityDescriptorGroup	返回 SD 个人组信息	否	是	是
GetSecurityDescriptorLength	返回 SD 长度	否	是	是
GetSecurityDescriptorSacl	返回 SD 系统 ACL	否	是	是
GetSecurityDescriptorOwner	返回 SD 所有人	否	是	是
GetSecurityDescriptorDacl	返回 SD 离散 ACL	否	是	是
GetLengthSid	返回 SID 长度	否	是	是
TlsGetValue	返回 TLS 值	否	是	是
GetOutlineTextMetrics	返回 TRUETYPE 字体的公制类型	是	是	是
GetSystemDirectory	返回 WINDOWS 系统目录路径	是	是	是
GetTickCount	返回 WINDOWS 运行时间	是	是	是

（续表）

Api 函数名	函数说明	适用范围		
		W16	W98	WNT
timeGetSystemTime	返回 WINDOWS 运行时间	否	是	是
timeGetTime	返回 WINDOWS 运行时间	否	是	是
GetVersion	返回 WINDOWS 和操作系统版本	是	是	是
GetWinMetaFileBits	返回 WINDOWS 格式的图元文件内容	否	是	是
FindWindow	返回一个与类和窗口名相关的窗口句柄	是	是	是
GetDlgCtrlID	返回儿子窗口的 ID 值	否	是	是
GetTextExtentExPoint	返回子字符串长度数组	否	是	是
GetKernelObjectSecurity	返回内核对象 SD	否	是	是
GetRgnBox	返回区域边框矩形	是	是	是
GetRegionData	返回区域数据	否	是	是
GetTextAlign	返回文本对齐标志	是	是	是
GetTextCharacterExtra	返回文本字符间隔	是	是	是
GetParent	返回父口句柄	是	是	是
timeGetDevCaps	返回计时器性能	否	是	是
QueryPerformanceFrequency	返回计数频率	否	是	是
GetKeyNameText	返回代表键盘名字的字符串	是	是	是
CommDlgExtendedError	返回出错数据	否	是	是
WindowFromPoint	返回包含点的窗口	是	是	是
LookupPrivilegeDisplayName	返回可见特权名	否	是	是
GetFileVersionInfoSize	返回可用版本信息大小	否	是	是
FindExecutable	返回可执行文件名及句柄	否	是	是
LookupPrivilegeName	返回可编程特权名	否	是	是
GetDlgItemText	返回对话框控件文本	否	是	是
GetDlgItem	返回对话框控件句柄	否	是	是
GetDialogBaseUnits	返回对话基础单元	否	是	是
GetObject	返回对象信息	是	是	是
GetObjectType	返回对象类型	否	是	是
GetOpenClipboardWindow	返回打开 CLIPBOARD 的窗口句柄	是	是	是
LocalSize	返回本地内存块大小	是	是	是
LocalFlags	返回本地内存块信息	是	是	是
GetLocalTime	返回本地时间和日期	否	是	是

（续表）

Api 函数名	函数说明	适用范围		
		W16	W98	WNT
LocalHandle	返回本地指针句柄	是	是	是
GetAtomName	返回本地原子字符串	是	是	是
GetUserName	返回用户名	否	是	是
DdeGetLastError	返回由 DDEML 函数设置的出错代码	否	是	是
GetConsoleCursorInfo	返回光标大小	否	是	是
GlobalSize	返回全局内存块大小	是	是	是
GlobalFlags	返回全局内存块信息	是	是	是
GlobalGetAtomName	返回全局原子字符串	是	是	是
DdeQueryConvInfo	返回关于 DDE 对话的信息	否	是	是
RegQueryValueEx	返回关键字的类型和值	否	是	是
mciGetCreatorTask	返回创建任务	否	是	是
GetLogicalDriveStrings	返回合法驱动器字符串	否	是	是
GetFontData	返回字体数据	是	是	是
GetTextExtentPoint	返回字符串大小	是	是	是
GetTextExtentPoint32	返回字符串大小	否	是	是
lstrlen	返回字符串中字符数	是	是	是
GetCharABCWidths	返回字符宽度	否	是	是
GetCharABCWidthsFloat	返回字符宽度	否	是	是
GetCharWidth	返回字符宽度	否	是	是
GetCharWidth32	返回字符宽度	否	是	是
GetAspectRatioFilterEx	返回当前 ASPECT-RATIO 过滤器	是	是	是
GetMiterLimit	返回当前 MITER-JOIN 长度	否	是	是
GetTextColor	返回当前文本色彩	是	是	是
GetCurrentObject	返回当前对象	否	是	是
GetCurrentDirectory	返回当前目录	否	是	是
GetCursor	返回当前光标的句柄	否	是	是
GetCursorPos	返回当前光标的位置	否	是	是
GetPolyFillMode	返回当前多边形填充模式	是	是	是
GetKerningPairs	返回当前字体内核对	是	是	是
GetTextCharsetInfo	返回当前字体设置信息	否	是	是
GetTextMetrics	返回当前字体的公制	是	是	是

（续表）

Api 函数名	函数说明	适用范围		
		W16	W98	WNT
GetTextFace	返回当前字体的字样	是	是	是
GetTimeZoneInformation	返回当前时区信息	否	是	是
GetCurrentProcessId	返回当前进程 ID 值	否	是	是
GetCurrentProcess	返回当前进程的句柄	否	是	是
GetMetaRgn	返回当前图元区域	否	是	是
GetCurrentThreadId	返回当前线索 ID 值	否	是	是
GetCurrentThread	返回当前线索的句柄	否	是	是
GetWorldTransform	返回当前变换式	否	是	是
GetBoundsRect	返回当前相邻矩形	否	是	是
GetROP2	返回当前绘图模式	是	是	是
GetBkColor	返回当前背景色	否	是	是
GetClipRgn	返回当前剪辑域	否	是	是
GetCaretPos	返回当前脱字号位置	否	是	是
GetFocus	返回当前焦点窗口句柄	是	是	是
GetTapePosition	返回当前磁带位置	否	是	是
DragQueryFile	返回托动的文件名	否	是	是
GetLastError	返回扩充出错代码	否	是	是
GetVersionEx	返回扩展操作系统版本信息	否	是	是
GetSidSubAuthorityCount	返回次字段地址	否	是	是
GetSidSubAuthority	返回次规范数组地址	否	是	是
WNetGetLastError	返回网络函数最近错误	否	是	是
mciGetDeviceID	返回设备名对应的 ID	否	是	是
GetICMProfile	返回设备场景的色彩映像	否	是	是
DeviceCapabilities	返回设备驱动程序的功能	否	是	是
GetDeviceCaps	返回设备性能	否	是	是
GetDCOrgEx	返回设备描述表的转换起点	否	是	是
WindowFromDC	返回设备描述标窗口	否	是	是
GetAclInformation	返回访问控件表信息	否	是	是
GetExitCodeProcess	返回过程中断代码	否	是	是
GetEnvironmentVariable	返回过程环境变量	否	是	是
GetPriorityClass	返回过程的优先级类	否	是	是

（续表）

Api 函数名	函数说明	适用范围		
		W16	W98	WNT
GetProcessShutdownParameters	返回过程停止参数	否	是	是
GetProcessWindowStation	返回过程窗口站句柄	否	是	是
GetStretchBltMode	返回位图拉伸模式	是	是	是
GetBitmapDimensionEx	返回位图宽和高	否	是	是
waveInGetNumDevs	返回声音输入设备数量	否	是	是
LoadLibraryEx	返回库模块句柄	否	是	是
GetPrivateProfileSection	返回私有键和值	否	是	是
GetRasterizerCaps	返回系统 TRUETYPE 状态	是	是	是
GetSystemMetrics	返回系统公制	是	是	是
GetSystemTime	返回系统时间和日期	否	是	是
GetSystemInfo	返回系统信息	否	是	是
GetSystemPaletteEntries	返回系统调色板入口	是	是	是
auxGetNumDevs	返回附属设备数量	否	是	是
GetDriverModuleHandle	返回驱动程序模块实例句柄	否	是	是
GetProcAddress	返回函数地址	否	是	是
GetNamedPipeHandleState	返回命名管道句柄消息	否	是	是
GetNamedPipeInfo	返回命名管道句柄消息	否	是	是
LookupIconIdFromDirectory	返回图标或光标 ID 坐标	否	是	是
GetIconInfo	返回图标或光标信息	否	是	是
GetArcDirection	返回弧和矩形的绘画方向	否	是	是
GetSidLengthRequired	返回所需 SID 长度	否	是	是
GetUserObjectSecurity	返回服务器对象 SD 信息	否	是	是
GetDiskFreeSpace	返回空闲磁盘空间	否	是	是
GetExitCodeThread	返回线索中断代码	否	是	是
GetThreadTimes	返回线索计时信息	否	是	是
GetThreadDesktop	返回线索桌面句柄	否	是	是
GetThreadSelectorEntry	返回线索描述入口	否	是	是
EnumThreadWindows	返回线索窗口	否	是	是
GetViewportOrgEx	返回视口源	是	是	是
GetViewportExtEx	返回视窗宽度	是	是	是
GetClipCursor	返回限制光标的矩形的坐标	否	是	是

（续表）

Api 函数名	函数说明	适用范围		
		W16	W98	WNT
GetTempPath	返回临时文件路径	否	是	是
GetPrivateObjectSecurity	返回保护服务器对象 SD	否	是	是
GetMailslotInfo	返回信箱信息	否	是	是
GetNextDlgTabItem	返回前或后一个 WS_TABSTOP 控件	是	是	是
GetNextDlgGroupItem	返回前或后一组控件的句柄	是	是	是
GetNextWindow	返回前或后一窗口管理器窗口	是	是	是
GetForegroundWindow	返回前景窗口句柄	否	是	是
GetConsoleScreenBufferInfo	返回屏幕缓冲区信息	否	是	是
GetAce	返回指向 ACL 中的 ACE 的指针	否	是	是
GetCommandLine	返回指向命令行的指针	否	是	是
GetEnvironmentStrings	返回指向环境块的指针	否	是	是
FindFirstFreeAce	返回指向第一个空闲 ACL 字节的指针	否	是	是
GetFileSize	返回指定文件大小	否	是	是
GetTokenInformation	返回指定令牌信息	否	是	是
RegQueryValue	返回指定关键字名	否	是	是
GetDriveType	返回指定驱动器类型	否	是	是
GetThreadPriority	返回指定线索优先级	否	是	是
GetThreadContext	返回指定线索描述表	否	是	是
EnumFontFamilies	返回指定家族的字体	否	是	是
EnumFontFamiliesEx	返回指定家族的字体	否	是	是
GetMenuState	返回指定菜单的菜单标记	是	是	是
GetWindow	返回指定窗口句柄	是	是	是
GetTopWindow	返回指定窗口的顶端儿子句柄	是	是	是
GetMenu	返回指定窗口的菜单句柄	是	是	是
GetPixel	返回指定像素的 RGB 值	是	是	是
GetModuleHandle	返回指定模块的句柄	是	是	是
GetModuleFileName	返回指定模块的路径	是	是	是
GetMapMode	返回映像模式	是	是	是
GetSysColor	返回显示元素色彩	是	是	是
GetDCEx	返回显示描述表的句柄	否	是	是
GetStdHandle	返回标准 I/O 句柄	否	是	是

（续表）

Api 函数名	函数说明	适用范围		
		W16	W98	WNT
GetActiveWindow	返回活动窗口句柄	是	是	是
GetKeyboardLayoutName	返回活动键盘版面名	否	是	是
GetBkMode	返回背景模式	否	是	是
GetDesktopWindow	返回桌面窗口句柄	否	是	是
LookupPrivilegeValue	返回特权名 LUID	否	是	是
GetMenuCheckMarkDimensions	返回缺省复选标记位图尺寸	是	是	是
GetPaletteEntries	返回调色板入口范围	是	是	是
GetCommModemStatus	返回调制解调器控制登录值	否	是	是
SizeofResource	返回资源大小	是	是	是
LoadResource	返回资源句柄	是	是	是
LockResource	返回资源地址	是	是	是
GetCharWidthFloat	返回部分字符宽度	否	是	是
GetClipboardOwner	返回剪贴板所有者窗口句柄	否	是	是
CountClipboardFormats	返回剪贴板格式的数量	是	是	是
GetClipboardData	返回剪贴板数据的句柄	否	是	是
HeapSize	返回堆对象的大小	否	是	是
GetStockObject	返回常用画笔、刷子或字段的句柄	是	是	是
GetSubMenu	返回弹出式菜单句柄	是	是	是
GetNumberOfConsoleInputEvents	返回控制台队列事件数	否	是	是
GetConsoleTitle	返回控制台窗口标题	否	是	是
GetConsoleMode	返回控制台输入输出模式	否	是	是
mixerGetLineControls	返回混合器线控件	否	是	是
GetPriorityClipboardFormat	返回第一个 CLIPBOARD 格式	是	是	是
GetClipboardViewer	返回第一个剪贴板浏览窗口句柄	否	是	是
GetCaretBlinkTime	返回脱字号闪烁时间	否	是	是
GetMenuItemID	返回菜单命令标识符	是	是	是
GetMenuItemCount	返回菜单命令数目	是	是	是
VirtualQueryEx	返回虚拟保护	否	是	是
VirtualQuery	返回虚拟信息	否	是	是
GetKeyState	返回虚拟键的状态	是	是	是
GetKeyboardState	返回虚拟键盘状态	是	是	是

（续表）

Api 函数名	函数说明	适用范围		
		W16	W98	WNT
GetCurrentPositionEx	返回逻辑单元位置	否	是	是
GetOverlappedResult	返回最后重叠结果	否	是	是
GetMessagePos	返回最后消息的光标位置	是	是	是
GetMessageTime	返回最后消息的时间位置	是	是	是
GetOldestEventLogRecord	返回最早的记录数	否	是	是
GetNearestPaletteIndex	返回最近的匹配色彩	是	是	是
GetNearestColor	返回最近的可用色彩	是	是	是
GetScrollPos	返回滑块位置	是	是	是
GetScrollRange	返回滑块运动范围	是	是	是
RegisterEventSource	返回登记的事件记录句柄	否	是	是
RegQueryInfoKey	返回登录字信息	否	是	是
GetClipboardFormatName	返回登录的剪贴板格式名	否	是	是
GetMessageExtraInfo	返回硬件消息的信息	是	是	是
GetWindowsDirectory	返回窗口 WINDOWS 目录	是	是	是
GetWindowTextLength	返回窗口工具栏文本长度	是	是	是
GetLargestConsoleWindowSize	返回窗口尺寸的最大可能性	否	是	是
GetWindowDC	返回窗口设备厂描述表	是	是	是
GetWindowRect	返回窗口坐标	是	是	是
GetUpdateRect	返回窗口更新区域大小	是	是	是
GetUpdateRgn	返回窗口更新区域大小	是	是	是
GetWindowThreadProcessId	返回窗口线索及过程 ID	否	是	是
GetClientRect	返回窗口客户区坐标	否	是	是
GetWindowPlacement	返回窗口显示状态及最小/最大位置	是	是	是
GetClassWord	返回窗口类内存字	否	是	是
GetClassName	返回窗口类名称	否	是	是
GetClassInfo	返回窗口类信息	否	是	是
GetClassLong	返回窗口类数据	否	是	是
GetWindowExtEx	返回窗口宽度	是	是	是
EnumProps	返回窗口属性表列	否	是	是
EnumPropsEx	返回窗口属性表列	否	是	是
GetDC	返回窗口描述表句柄	否	是	是

（续表）

Api 函数名	函数说明	适用范围		
		W16	W98	WNT
GetWindowOrgEx	返回窗口源	是	是	是
GdiGetBatchLimit	返回缓冲 GDI 函数数量	否	是	是
CreateFileMapping	返回新文件映像对象一个句柄	否	是	是
CreateEvent	返回新事件对象一个句柄	否	是	是
GetPath	返回路径中的所有直线和曲线	否	是	是
GetKeyboardType	返回键盘信息	是	是	是
GetDoubleClickTime	返回鼠标双击时间	否	是	是
DragQueryPoint	返回鼠标位置	否	是	是
GetNumberOfConsoleMouseButtons	返回鼠标按钮数	否	是	是
GetTapeStatus	返回磁带机状态	否	是	是
GetTapeParameters	返回磁带驱动器或介质信息	否	是	是
GetEnhMetaFileHeader	返回增强图元文件头	否	是	是
GetEnhMetaFileDescription	返回增强图元文件的标题及建立者	否	是	是
GetEnhMetaFilePaletteEntries	返回增强图元文件调色板入口	否	是	是
joyGetPos	返回操纵杆位置及按钮活动	否	是	是
joyGetPosEx	返回操纵杆位置扩展信息	否	是	是
joyGetThreshold	返回操纵杆运动临界值	否	是	是
joyGetNumDevs	返回操纵杆数目	否	是	是
ObjectPrivilegeAuditAlarm	进行特权操作时产生审查/警报	否	是	是
midiConnect	连接 MIDI 设备	否	是	是
RegConnectRegistry	连接远程登录	否	是	是
OpenSCManager	连接服务控件管理器	否	是	是
StartServiceCtrlDispatcher	连接线索	否	是	是
DebugActiveProcess	连接调试进程	否	是	是
IsValidSid	使 SID 有效	否	是	是
IsValidSecurityDescriptor	使安全描述有效	否	是	是
GdiFlush	使当前 GDI 闪烁	否	是	是
IsValidAcl	使访问控件表有效	否	是	是
EnableMenuItem	激活菜单命令	是	是	是
EnableScrollBar	激活滚动条	否	是	是

（续表）

Api 函数名	函数说明	适用范围		
		W16	W98	WNT
AdjustTokenGroups	调整令牌中的群	否	是	是
AdjustTokenPrivileges	调整令牌权限	否	是	是
DdeEnableCallback	激活 DDE 对话	否	是	是
ContinueDebugEvent	使调试线索继续	否	是	是
FlashWindow	使窗口闪烁一次	是	是	是
GetExpandedName	取压缩文件的原文件名	是	是	是
GetSecurityDescriptorControl	取回 SD 校正及控制信息	否	是	是
GetComputerName	取回当前计算机名	否	是	是
GetGlyphOutline	取回轮廓数据	是	是	是
GetGraphicsMode	取回指定 DC 的图形模式	否	是	是
DestroyCursor	取消 CREATECURSOR 建立的光标	是	是	是
DdeDisconnectList	取消 DDE 对话表列	否	是	是
DeleteMetaFile	取消 WINDOWS 图元文件句柄	是	是	是
DestroyAcceleratorTable	取消加速表	否	是	是
DestroyIcon	取消由 CREATEICON 建立的图标	是	是	是
DdeAbandonTransaction	取消异步交互	否	是	是
DestroyCaret	取消当前脱字号	是	是	是
CancelDC	取消指定 DC 上的任何悬而未决的操作	否	是	是
HeapDestroy	取消堆	否	是	是
DestroyMenu	取消菜单并释放内存	是	是	是
DestroyWindow	取消窗口	是	是	是
DeleteEnhMetaFile	取消增强图元文件句柄	否	是	是
GetClipBox	取剪裁区域框	否	是	是
DefineDosDevice	定义、重定义或删除 DOS 的设备名	否	是	是
DefDriverProc	定义缺省消息处理器	是	是	是
RegisterWindowMessage	定义新的窗口消息	是	是	是
DrawEscape	实现非 GDI 可画图设备方法	否	是	是
OpenFile	建立、打开或删除文件	是	是	是
CreateFile	建立、打开或截断文件	否	是	是
BuildCommDCBAndTimeouts	建立 COMMDCB 并设置超时值	否	是	是
DdeCreateStringHandle	建立 DDE 字符串句柄	否	是	是

（续表）

Api 函数名	函数说明	适用范围		
		W16	W98	WNT
DdeCreateDataHandle	建立 DDE 数据句柄	否	是	是
CreateMailslot	建立 Mailsolt	否	是	是
mmioCreateChunk	建立 RIFF 文件块	否	是	是
CreateMetaFile	建立 WINDOWS 图元 DC	是	是	是
GetMetaFile	建立 WINDOWS 图元文件	是	是	是
SetMetaFileBitsEx	建立 WINDOWS 图元文件	否	是	是
ChooseFont	建立一个字体选择对话框	否	是	是
ChooseColor	建立一个色彩选择对话框	否	是	是
CreateDirectory	建立一个新目录	否	是	是
CreateCompatibleBitmap	建立与 DC 相兼容的位图	是	是	是
DdeConnect	建立与服务器的对话	否	是	是
CreateCompatibleDC	建立与指定 DC 相兼容的 DC	是	是	是
PathToRegion	建立区域	否	是	是
ReplaceText	建立文本对话框	否	是	是
GetOpenFileName	建立文件名对话框	否	是	是
GetSaveFileName	建立文件名保存对话框	否	是	是
CreateAcceleratorTable	建立加速表	否	是	是
CreateDiscardableBitmap	建立可放弃位图	是	是	是
CreatePolyPolygonRgn	建立由多边形组成的区域	是	是	是
RegCreateKey	建立关键字	否	是	是
RegCreateKeyEx	建立关键字	否	是	是
DdeConnectList	建立多个 DDE 对话	否	是	是
CreatePolygonRgn	建立多边形区域	是	是	是
PageSetupDlg	建立并显示选项卡的设置对话框	否	是	是
RaiseException	建立异常条件	否	是	是
CreateIcon	建立有指定属性的图标	是	是	是
MakeSelfRelativeSD	建立自相关 SD	否	是	是
CreateDC	建立设备描述表	是	是	是
PatBlt	建立位图图案	是	是	是
CreateNamedPipe	建立命名管道实例	否	是	是
CreateIconFromResource	建立图标或光标	否	是	是

（续表）

Api 函数名	函数说明	适用范围		
		W16	W98	WNT
CreateIconIndirect	建立图标或光标	否	是	是
CreateService	建立服务对象	否	是	是
SetRectEmpty	建立空的矩形	是	是	是
CreateDialogParam	建立非模态对话框	否	是	是
GetTempFileName	建立临时文件名	是	是	是
CreateIC	建立信息上下文	是	是	是
CreateScalableFontResource	建立带字体信息的资源文件	是	是	是
CreateProcessAsUser	建立指定用户的新进程	否	是	是
FindText	建立查找文本对话框	否	是	是
MakeAbsoluteSD	建立独立 SD	否	是	是
CreateBitmap	建立独立于设备的内存位图	是	是	是
CreateRectRgn	建立矩形区域	是	是	是
CreatePipe	建立匿名管道	否	是	是
CreateRoundRectRgn	建立圆角矩形	是	是	是
MessageBox	建立消息框窗体	是	是	是
MessageBoxEx	建立消息框窗体	否	是	是
IntersectClipRect	建立剪辑区域	是	是	是
HeapCreate	建立堆	否	是	是
CreatePopupMenu	建立弹出式菜单	是	是	是
DebugBreak	建立断点	是	是	是
CreateMenu	建立菜单	是	是	是
CreateFont	建立逻辑字体	是	是	是
CreatePalette	建立逻辑色彩调色板	是	是	是
CreateColorSpace	建立逻辑色影区域	否	是	是
CreatePen	建立逻辑画笔	是	是	是
ExtCreatePen	建立逻辑画笔	否	是	是
CreateEllipticRgn	建立椭圆区域	是	是	是
CreateEllipticRgnIndirect	建立椭圆区域	是	是	是
CreateWindowEx	建立窗口	是	是	是
CreateMDIWindow	建立新的 MDI 窗口	否	是	是
InitializeAcl	建立新的访问控件表	否	是	是

（续表）

Api 函数名	函数说明	适用范围		
		W16	W98	WNT
CreateProcess	建立新的进程和线索对象	否	是	是
CreateThread	建立新的线索	否	是	是
CreateTapePartition	建立新的磁带分区	否	是	是
GetEnhMetaFile	建立增强图元文件	否	是	是
SetEnhMetaFileBits	建立增强图元文件	否	是	是
CreateEnhMetaFile	建立增强型图元文件 DC	否	是	是
CreateHatchBrush	建立影线刷子	是	是	是
mixerSetControlDetails	放置混合器控件	否	是	是
EnumServicesStatus	枚举 SC 管理器数据库中的服务	否	是	是
EnumSystemCodePages	枚举已安装的可用系统代码页	否	是	是
EnumSystemLocales	枚举已安装的系统局部	否	是	是
EnumICMProfiles	枚举可用的颜色分布	否	是	是
EnumMonitors	枚举可用监视器	否	是	是
EnumTimeFormats	枚举本地指定时间格式	否	是	是
EnumObjects	枚举设备描述表中的画笔和刷子	否	是	是
EnumDateFormats	枚举局部指定数据格式	否	是	是
EnumDependentServices	枚举依赖于设备的服务	否	是	是
EnumFonts	枚举指定设备上的字体	否	是	是
PolyBezier	画 BEZIER 曲线	否	是	是
PolyBezierTo	画 BEZIER 曲线	否	是	是
Rectangle	画一个矩形	是	是	是
Chord	画一条弦	是	是	是
PolylineTo	画一条或多条直线	否	是	是
PolyPolygon	画一系列多边形	是	是	是
PolyDraw	画一系列直线及 BEZIER 曲线	否	是	是
Polygon	画多边形	是	是	是
Arc	画弧	是	是	是
Polyline	画线段	是	是	是
PolyPolyline	画相连线段	否	是	是
RoundRect	画圆角矩形	是	是	是
Ellipse	画椭圆	是	是	是

（续表）

Api 函数名	函数说明	适用范围		
		W16	W98	WNT
ArcTo	画椭圆弧	否	是	是
mmioAdvance	直接 I/O 缓冲	否	是	是
DeviceIoControl	直接调用驱动程序	否	是	是
WriteConsoleOutput	直接控制屏幕缓冲区	否	是	是
DdeDisconnect	终止 DDE 对话	否	是	是
AbortPath	终止或取消 DC 中的一切路径	否	是	是
CharPrev	转到字符串的上一个字符	否	是	是
CharNext	转到字符串的下一个字符	否	是	是
OemToChar	转换 OEM 字符串	否	是	是
OemToCharBuff	转换 OEM 字符串	否	是	是
MaskBlt	转换位图	否	是	是
SystemTimeToFileTime	转换系统时间为 64 位时间	否	是	是
TranslateCharsetInfo	转换结构为给定字符串	否	是	是
ToUnicode	转换虚拟关键字代码为 ANSI 字符	否	是	是
ToAscii	转换虚拟关键字代码为 WINDOWS 字符	是	是	是
MapVirtualKey	转换虚拟关键字代码或扫描代码	是	是	是
TranslateMessage	转换虚拟关键字消息	是	是	是
SaveDC	保存设备描述表	是	是	是
LeaveCriticalSection	保留前一个输入临界段	否	是	是
VirtualAlloc	保留虚拟页	否	是	是
SetPrivateObjectSecurity	修改 SD	否	是	是
SetRectRgn	修改区域为矩形	是	是	是
LocalReAlloc	修改本地内存大小及属性	是	是	是
GlobalReAlloc	修改全局内存块大小/属性	是	是	是
SetServiceObjectSecurity	修改服务对象安全描述符	否	是	是
ChangeServiceConfig	修改服务参数	否	是	是
HiliteMenuItem	修改顶级菜单命令高亮度	是	是	是
CheckMenuItem	修改菜单命令的复选标记属性	是	是	是
VirtualProtect	修改虚拟保护	否	是	是
VirtualProtectEx	修改虚拟保护	否	是	是
ResizePalette	修改逻辑调色板大小	是	是	是

（续表）

Api 函数名	函数说明	适用范围		
		W16	W98	WNT
FreeLibrary	卸载库模块	是	是	是
FoldString	变换字符串	否	是	是
ReplyMessage	响应通过 SENDMESSAGE 发送的消息	是	是	是
ResetEvent	复位事件对象	否	是	是
UnrealizeObject	复位逻辑调色板	是	是	是
CopyMetaFile	复制 Windows 图元文件	是	是	是
CopyAcceleratorTable	复制一个加速表	否	是	是
CopyCursor	复制一个光标	是	是	是
CopyFile	复制文件	否	是	是
LZCopy	复制文件或压缩	是	是	是
DuplicateHandle	复制对象句柄	否	是	是
DuplicateToken	复制访问令牌	否	是	是
StretchBlt	复制位图	是	是	是
CopyIcon	复制图标	是	是	是
CopyRect	复制矩形大小	是	是	是
CopyEnhMetaFile	复制增强型图元文件	否	是	是
FileTimeToSystemTime	将 64 位时间转换为系统时间	否	是	是
AddAccessAllowedAce	将 ACCESS_ALLOWED_ACE 加入 ACL	否	是	是
AddAccessDeniedAce	将 ACCESS_DENIED_ACE 加入 ACL	否	是	是
AddAce	将 ACE 加入一个已存在的 ACL	否	是	是
StretchDIBits	将 DIB 从源矩形移至目的矩形	是	是	是
GetDIBits	将 DIB 位复制到缓冲区中	否	是	是
DosDateTimeToFileTime	将 MS-DOS 日期时间转换为 64 位格式	否	是	是
CopySid	将 SID 复制到缓冲区中	否	是	是
AddAuditAccessAce	将 SYSTEM_AUDIT_ACE 加入 ACL	否	是	是
FileTimeToLocalFileTime	将 UTC 文件时间转换成本地文件时间	否	是	是
GetMetaFileBitsEx	将 WINDOWS 图元文件复制入缓冲区	否	是	是
AddAtom	将一个字符串加入本地原子表	是	是	是
ZeroMemory	将一块内存置零	否	否	否

（续表）

Api 函数名	函数说明	适用范围		
		W16	W98	WNT
AddFontResource	将一种字体加入字体表	是	是	是
EnumChildWindows	将子窗口句柄传给回调函数	否	是	是
RegSetValue	将文本字符串与指定关键字关联	否	是	是
RegSetValueEx	将文本字符串与指定关键字关联	否	是	是
FileTimeToDosDateTime	将文件时间转换为 MS-DOS 日期	否	是	是
CreateMutex	将句柄返回给 MUTEX 对象	否	是	是
OpenProcess	将句柄返回给过程对象	否	是	是
CreateSemaphore	将句柄返回给新的信号量	否	是	是
CreateConsoleScreenBuffer	将句柄返回给新的屏幕缓冲区	否	是	是
GetDlgItemInt	将对话框文本转换为整数	否	是	是
MapDialogRect	将对话框映像至像素	是	是	是
LocalFileTimeToFileTime	将本地文件时间转移为 UTC 文件时间	否	是	是
DdeSetUserHandle	将用户定义句柄与事务建立关联	否	是	是
ClipCursor	将光标限制在矩形内	是	是	是
GlobalHandle	将全局指针转换为句柄	是	是	是
MultiByteToWideChar	将多媒体字符串映像为通配字符串	否	是	是
FillConsoleOutputCharacter	将字符写入屏幕缓冲区	否	是	是
WriteProfileString	将字符串写入 WIN.INI	是	是	是
DdeQueryString	将字符串句柄文本复制到缓冲区	否	是	是
CharLowerBuff	将字符串变为小写	否	是	是
CharUpperBuff	将字符串变为小写	否	是	是
mmioStringToFOURCC	将字符串变为四个字符代码	否	是	是
CharUpper	将字符或字符串变为大写	否	是	是
CharLower	将字符或字符串变为小写	否	是	是
VkKeyScan	将字符转换为虚拟关键字代码	是	是	是
DPtoLP	将设备坐标转换位逻辑坐标	是	是	是
CombineRgn	将两个区域合成一个区域	是	是	是
CombineTransform	将两个变换式结合在一起	否	是	是
MulDiv	将两数相乘并并除以结果	是	是	是
GetBitmapBits	将位图复制到缓冲区	否	是	是
BackupEventLog	将事件记录保存至后备文件	否	是	是

（续表）

Api 函数名	函数说明	适用范围		
		W16	W98	WNT
FlattenPath	将弧变为线	否	是	是
AttachThreadInput	将线索彼此相连	否	是	是
MapViewOfFile	将视图映像入地址空间	否	是	是
MapViewOfFileEx	将视图映像入地址空间	否	是	是
SetMenuItemBitmaps	将复选标记位图于菜单项关联	是	是	是
ClientToScreen	将客户点转换成屏幕坐标	是	是	是
ScreenToClient	将屏幕点转换为客户坐标	是	是	是
WriteProfileSection	将段写入 WIN.INI	否	是	是
MapWindowPoints	将点转换到另一坐标系统	是	是	是
VerLanguageName	将语言 ID 转换为文本描述	否	是	是
PostMessage	将消息加入线索消息队列	是	是	是
CallMsgFilter	将消息传给消息过滤过程	是	是	是
CallWindowProc	将消息传答窗口函数	否	是	是
ConvertDefaultLocale	将缺省局部转换为实际的局部值	否	是	是
WideCharToMultiByte	将通配符映像为多字节	否	是	是
SetMetaRgn	将剪裁区选作图元区域	否	是	是
GetMenuString	将菜单字符串复制入缓冲区	是	是	是
LPtoDP	将逻辑指针变为设备指针	是	是	是
RealizePalette	将逻辑调色板映像为系统调色板	是	是	是
FillConsoleOutputAttribute	将属性写入屏幕缓冲区	否	是	是
RegSaveKey	将登录保存了的树存入文件	否	是	是
GetWindowText	将窗口工具栏文本复制到缓冲区	是	是	是
SetForegroundWindow	将窗口置于前台	否	是	是
DdeGetData	将数据从 DDE 数据对象中复制到缓冲区	否	是	是
PackDDElParam	将数据打包装入 DDE 消息 IPARAM	否	是	是
GetEnhMetaFileBits	将增强图元文件值复制到缓冲区	否	是	是
SetDlgItemInt	将整数转换为对话框文本字符串	是	是	是
ClearCommBreak	恢复字符传输	是	是	是
RestoreDC	恢复设备描述表	是	是	是
BringWindowToTop	恢复重叠窗	是	是	是
SetCommBreak	挂起字符传送	是	是	是

（续表）

Api 函数名	函数说明	适用范围		
		W16	W98	WNT
SuspendThread	挂起线索	否	是	是
GetLogicalDrives	指定合法驱动器	否	是	是
DdeSetQualityOfService	指明服务的 DDE 质量	否	是	是
AngleArc	按指定角度画弧	否	是	是
MapGenericMask	映像对专用/标准的一般性访问	否	是	是
LCMapString	映像字符串	否	是	是
LoadLibrary	映像模块到进行地址空间	是	是	是
ShowCursor	显示光标	是	是	是
ShowOwnedPopups	显示弹出式窗口	是	是	是
TrackPopupMenu	显示弹出式窗口	是	是	是
ShowCaret	显示脱字号	是	是	是
RegNotifyChangeKeyValue	显示登记关键字的变化	否	是	是
ShowWindow	显示窗口	是	是	是
ShowScrollBar	显示滚动条	是	是	是
SystemParametersInfo	查寻系统参数信息	是	是	是
FindNextFile	查找下一个匹配文件	否	是	是
SearchPath	查找文件	否	是	是
FindFirstFile	查找第一个匹配文件	否	是	是
PtVisible	查询点是否在剪辑区域内	是	是	是
RectInRegion	查询矩形是否有重叠区域	是	是	是
RectVisible	查询矩形是否有剪辑区中	是	是	是
AnyPopup	标识弹出式窗口是否存在	是	是	是
midiInStop	结束 MIDI 输入	否	是	是
midiInReset	结束 MIDI 输入和标准输入缓冲区	否	是	是
midiOutReset	结束 MIDI 输出和标记缓冲区	否	是	是
FatalAppExit	结束一个应用程序	否	是	是
EndPage	结束一页	是	是	是
WNetCloseEnum	结束网络资源列表	否	是	是
EndPath	结束路径	否	是	是

（续表）

Api 函数名	函数说明	适用范围		
		W16	W98	WNT
Pie	绘制饼状锲形图	是	是	是
ExitProcess	退出当前进程	否	是	是
ExitThread	退出当前线索	否	是	是
SelectObject	选定对象	是	是	是
SelectClipPath	选定当前路径为剪辑区域	否	是	是
SelectPalette	选定调色板	是	是	是
SelectClipRgn	选定剪辑区域	是	是	是
ExtSelectClipRgn	选择一块区域作为剪裁区	否	是	是
waveOutRestart	重开始声音回放	否	是	是
ReuseDDElParam	重利用 DDE 消息 IPARAM	否	是	是
midiStreamRestart	重启动指定 MIDI 流	否	是	是
mmioRename	重命名多媒体文件名	否	是	是
WNetAddConnection	重定向本地设备网络资源	是	是	是
WNetAddConnection2	重定向本地设备网络资源	否	是	是
LZSeek	重定位文件中的指针	是	是	是
DdeReconnect	重建 DDE 对话	否	是	是
DrawMenuBar	重显示菜单栏	是	是	是
midiInPrepareHeader	准备 MIDI 输入缓冲区	否	是	是
midiOutPrepareHeader	准备 MIDI 输出数据块	否	是	是
BeginPaint	准备一个画图窗	是	是	是
PrepareTape	准备磁带设备	否	是	是
HeapValidate	校验指定堆结构	否	是	是
wsprintf	格式化字符串	是	否	否
FormatMessage	格式化消息字符串	否	是	是
midiOutUnprepareHeader	消除 MIDI 输出头	否	是	是
ClearEventLog	消除事件记录	否	是	是
PurgeComm	消除通讯队列	否	是	是
WaitCommEvent	监视屏蔽的事件	否	是	是
WaitForSingleObject	监测一个对象	否	是	是
WaitForSingleObjectEx	监测一个对象或 I/O 结束	否	是	是

（续表）

Api 函数名	函数说明	适用范围		
		W16	W98	WNT
WaitForMultipleObjects	监测多个对象	否	是	是
WaitForMultipleObjectsEx	监测多个对象或 I/O 结束	否	是	是
WaitForInputIdle	监测进程空闲	否	是	是
WaitNamedPipe	监测命名管道	否	是	是
WNetEnumResource	继续列表网络资源	否	是	是
DefMDIChildProc	缺省 MDI 子窗口消息进程	是	是	是
DefFrameProc	缺省 MDI 框架窗口消息进程	是	是	是
GetStringTypeA	获取 ANSI 字符串类型	否	是	是
GetACP	获取 ANSI 系统代码页	否	是	是
QueryDosDevice	获取 DOS 设备名信息	否	是	是
midiStreamPosition	获取 MIDI 流当前位置	否	是	是
midiStreamProperty	获取 MIDI 流属性	否	是	是
midiOutGetNumDevs	获取 MIDI 输出设备数量	否	是	是
GetKBCodePage	获取 OEM 系统代码页	是	是	是
GetOEMCP	获取 OEM 系统代码页	否	是	是
GetStringTypeW	获取 UNICODE 字符串类型	否	是	是
SHGetFileInfo	获取工作台外壳文件夹界面	否	是	是
GetFileSecurity	获取文件或目录安全信息	否	是	是
QueryPerformanceCounter	获取计数值	否	是	是
GetCPInfo	获取代码页信息	否	是	是
GetDateFormat	获取本地日期格式	否	是	是
GetTimeFormat	获取本地指定的时间字符串	否	是	是
GetUserDefaultLCID	获取用户缺省本地 ID	否	是	是
GetUserDefaultLangID	获取用户缺省语言 ID	否	是	是
GetProcessAffinityMask	获取任务可用的处理表列	否	是	是
GetFontLanguageInfo	获取字体显示描述表的信息	否	是	是
GetCharacterPlacement	获取字符串属性数据	否	是	是
WNetGetUser	获取当前网络用户名	否	是	是
GetStartupInfo	获取当前过程的启始信息	否	是	是
auxGetVolume	获取当前卷设置	否	是	是
mixerGetNumDevs	获取当前混合设备数量	否	是	是

（续表）

Api 函数名	函数说明	适用范围		
		W16	W98	WNT
WNetGetConnection	获取网络资源名	是	是	是
GetColorAdjustment	获取设备描述表调整值	否	是	是
GetProcessTimes	获取过程计时器	否	是	是
mciGetYieldProc	获取过程地址	否	是	是
waveOutGetErrorText	获取声音出错文本	否	是	是
waveInGetErrorText	获取声音出错的信息文本	否	是	是
waveOutGetPosition	获取声音回放位置	否	是	是
waveOutGetPlaybackRate	获取声音回放率	否	是	是
waveInGetPosition	获取声音设备输入位置	否	是	是
waveOutGetVolume	获取声音音量	否	是	是
waveInGetID	获取声音输入设备 ID	否	是	是
waveInGetDevCaps	获取声音输入设备性能	否	是	是
waveOutGetID	获取声音输出设备 ID	否	是	是
waveOutGetDevCaps	获取声音输出设备性能	否	是	是
waveOutGetNumDevs	获取声音输出设备数量	否	是	是
waveOutGetPitch	获取声音输出的强度	否	是	是
GetSystemPowerStatus	获取系统 AC 或 DC 电源状态	否	是	是
GetSystemDefaultLCID	获取系统缺省本地 ID	否	是	是
GetSystemDefaultLangID	获取系统缺省语言 ID	否	是	是
auxGetDevCaps	获取附属设备容量	否	是	是
GetNumberOfEventLogRecords	获取事件记录中的记录数	否	是	是
QueryServiceObjectSecurity	获取服务对象安全描述	否	是	是
QueryServiceStatus	获取服务状态	否	是	是
GetServiceDisplayName	获取服务显示名称	否	是	是
QueryServiceConfig	获取服务配置参数	否	是	是
GetServiceKeyName	获取服务登录关键名称	否	是	是
QueryServiceLockStatus	获取服务数据库锁定状态	否	是	是
GetThreadLocale	获取线索本地信息	否	是	是
GetColorSpace	获取指定色彩空间	否	是	是
mixerGetControlDetails	获取指定混合器控件	否	是	是
GetDeviceGammaRamp	获取显示器的辉度系数	否	是	是

（续表）

Api 函数名	函数说明	适用范围		
		W16	W98	WNT
ImpersonateSelf	获取调用过程的模拟令牌	否	是	是
GetProcessHeap	获取调用过程堆句柄	否	是	是
VerQueryValue	获取资源项目	否	是	是
mixerGetID	获取混合器 ID 值	否	是	是
mixerGetDevCaps	获取混合器性能	否	是	是
mixerGetLineInfo	获取混合器信息	否	是	是
LoadMenuIndirect	获取菜单模板句柄	是	是	是
GetLogColorSpace	获取逻辑色彩空间的信息	否	是	是
RegGetKeySecurity	获取登录关键安全属性	否	是	是
DrvGetModuleHandle	获得可安装驱动程序的实例句柄	否	是	是
FindNextChangeNotification	请求对下一个文件或目录变化的通知	否	是	是
TransactNamedPipe	读写命名管道	否	是	是
ReadEventLog	读事件记录	否	是	是
ReadConsoleOutputCharacter	读屏幕缓冲区字符串	否	是	是
ReadConsoleOutput	读屏幕缓冲区数据	否	是	是
ReadConsoleOutputAttribute	读控制台属性字符串	否	是	是
ReadConsole	读控制台输入数据	否	是	是
CallNextHookEx	调中链中的下一个挂钩过程	是	是	是
DefWindowProc	调用缺省窗口过程	是	是	是
ScheduleJob	调度作业	否	是	是
SetTextJustification	调整文本输出	是	是	是
ScaleViewportExtEx	调整视口大小	是	是	是
ScaleWindowExtEx	调整窗口大小	是	是	是
SetColorAdjustment	调整颜色	否	是	是
PostQuitMessage	通知 WINDOWS 线索将中断	是	是	是
NotifyBootConfigStatus	通知响应引导配置	否	是	是
waveOutPrepareHeader	预备声音回放数据块	否	是	是
waveInPrepareHeader	预备声音输入缓冲区	否	是	是
PeekNamedPipe	预显示管道队列数据	否	是	是
midiOutCachePatches	预装入 MIDI 修正码	否	是	是
midiOutCacheDrumPatches	预装入 MIDI 碰撞修正码	否	是	是

（续表）

Api 函数名	函数说明	适用范围		
		W16	W98	WNT
UnmapViewOfFile	停止文件查看映像	否	是	是
waveOutReset	停止声音回放	否	是	是
waveInStop	停止声音输入	否	是	是
waveInReset	停止声音输入设备工作	否	是	是
waveInStart	停止声音输入设备工作	否	是	是
AbortSystemShutdowna	停止系统工作	否	是	是
RevertToSelf	停止模拟	否	是	是
midiStreamStop	停止播放 MIDI 流	否	是	是
InterlockedDecrement	减少 LONG	否	是	是
mmioDescend	减少 RIFF 块	否	是	是
ArrangeIconicWindows	排列最小化的子窗口	是	是	是
ColorMatchToTarget	控件预览设备描述表	否	是	是
mmioSetBuffer	控制 I/O 缓冲	否	是	是
SetBoundsRect	控制相邻矩形重叠	是	是	是
midiDisconnect	断开 MIDI 设备	否	是	是
WNetDisconnectDialog	断开网络对话框	否	是	是
WNetCancelConnection	断开网络连接	否	是	是
WNetCancelConnection2	断开网络连接	是	是	是
GlobalMemoryStatus	检查内存状态	否	是	是
AreAnyAccessesGranted	检查任何要求的访问	否	是	是
CheckColorsInGamut	检查设备调色板中是否有色彩	否	是	是
EqualSid	检查两 SLD 安全 ID 是否相等	否	是	是
EqualPrefixSid	检查两个 SLD 前缀是否相等	否	是	是
AreAllAccessesGranted	检查所有要求的访问	否	是	是
PeekMessage	检查消息队列	是	是	是
PrivilegeCheck	检查特权安全描述表	否	是	是
AccessCheckAndAuditAlarm	检验访问，产生声音或警报	否	是	是
AccessCheck	检验客户访问权限	否	是	是
EmptyClipboard	清空剪贴板并释放数据句柄	是	是	是
FlushFileBuffers	清除文件缓冲区	否	是	是
timeEndPeriod	清除计时器分辨率	否	是	是

（续表）

Api 函数名	函数说明	适用范围		
		W16	W98	WNT
midiInUnprepareHeader	清除预备头	否	是	是
waveOutUnprepareHeader	清除预备声音数据块	否	是	是
waveInUnprepareHeader	清除预备的声音文件头	否	是	是
FlushConsoleInputBuffer	清除控制台输入缓冲区	否	是	是
MoveToEx	移动当前位置	是	是	是
OffsetViewportOrgEx	移动视口区域	是	是	是
OffsetClipRgn	移动剪辑区	是	是	是
OffsetWindowOrgEx	移动窗口区域	是	是	是
ScrollWindow	移动窗口客户区	是	是	是
ScrollWindowEx	移动窗口客户区	是	是	是
keybd_event	综合击键事件	否	是	是
EndDialog	隐藏对话框	是	是	是
HideCaret	隐藏脱字号	是	是	是
DdePostAdvise	提示服务器向客户机发送建议数据	否	是	是
DefDlgProc	提供缺省窗口消息进程	是	是	是
midiStreamPause	暂停 MIDI 流	否	是	是
waveOutPause	暂停声音回放	否	是	是
WaitMessage	暂停应用程序运行并产生控件	是	是	是
SetSystemPowerState	暂停系统工作	否	是	是
Sleep	暂停线索	否	是	是
SleepEx	暂停线索直到 I/O 结束	否	是	是
AnimatePalette	替换逻辑调色板中的项目	是	是	是
CloseWindow	最小化窗口	是	是	是
RegLoadKey	登记关键字和子关键字	否	是	是
DdeNameService	登记取消服务器名称	否	是	是
RegisterServiceCtrlHandler	登记服务控制请求句柄	否	是	是
RegisterHotKey	登记热键	否	是	是
DragAcceptFiles	登记窗口是否接受托动文件的内容	否	是	是
RegisterClass	登记窗口类	否	是	是
RegisterClipboardFormat	登记新的 CLIPBOARD 格式	是	是	是

（续表）

Api 函数名	函数说明	适用范围		
		W16	W98	WNT
LogonUser	登录用户	否	是	是
RegRestoreKey	登录保存树	否	是	是
midiInGetDevCaps	确定 MIDI 设备性能	否	是	是
GetBinaryType	确定二进制可执行文件类型	否	是	是
VerFindFile	确定文件安装路径	否	是	是
IsValidCodePage	确定代码页是否有效	否	是	是
IsBadWritePtr	确定写指针的合法性	是	是	是
ChildWindowFromPoint	确定包含有点的窗口	是	是	是
IsMenu	确定句柄是否是菜单	是	是	是
IsValidLocale	确定本地代码是否有效	否	是	是
IsBadStringPtr	确定字符串指针的合法性	是	是	是
IsCharUpper	确定字符串是否是大写	是	是	是
IsCharLower	确定字符串是否是小写	是	是	是
IsCharAlpha	确定字符串是否是字母	是	是	是
IsCharAlphaNumeric	确定字符串是否是数字	是	是	是
IsDBCSLeadByte	确定字符是否 DBCS 引导字节	是	是	是
EqualRect	确定两矩形是否相等待	是	是	是
IsBadHugeWritePtr	确定进程是否有写动作	否	是	是
IsBadReadPtr	确定进程是否有读动作	是	是	是
GetSystemPaletteUse	确定使用整个系统调色板	是	是	是
GetTabbedTextExtent	确定制表串大小	是	是	是
IsDlgButtonChecked	确定按钮控件状态	是	是	是
PtInRect	确定点是否在矩形内	是	是	是
IsRectEmpty	确定矩形是否为空	是	是	是
IsClipboardFormatAvailable	确定格式是否可用	是	是	是
GetQueueStatus	确定消息队列内容	是	是	是
IsDialogMessage	确定消息是否用于对话框	是	是	是
IsBadCodePtr	确定读指针的合法性	否	是	是
IsBadHugeReadPtr	确定读指针的合法性	否	是	是
GetLastActivePopup	确定最近的活动弹出式窗口	是	是	是
IsWindow	确定窗口句柄是否有效	是	是	是

（续表）

Api 函数名	函数说明	适用范围		
		W16	W98	WNT
IsWindowVisible	确定窗口是否可见	是	是	是
InSendMessage	确定窗口是否在进行 SENDMESSAGE	是	是	是
IsWindowUnicode	确定窗口是否使用 UNICODE	否	是	是
IsChild	确定窗口是否是儿子窗	是	是	是
IsIconic	确定窗口是否被最小画	是	是	是
IsWindowEnabled	确定窗口是否接收用户输入	是	是	是
IsZoomed	确定窗口是否最大化	是	是	是
GetAsyncKeyState	确定键的状态	是	是	是
GetInputState	确定鼠标、键盘、定时器状态	是	是	是
joyGetDevCaps	确定操纵杆是否可用	否	是	是
MsgWaitForMultipleObjects	等待多个对象句柄	否	是	是
ConnectNamedPipe	等待要连接的客户机	否	是	是
UnhandledExceptionFilter	筛选异常条件	否	是	是
mmioInstallIOProcA	装入及删除自定义 I/O 过程	否	是	是
LoadAccelerators	装入加速表	是	是	是
LoadCursor	装入光标资源	是	是	是
LoadString	装入字符串资源	是	是	是
LoadModule	装入并运行程序	是	是	是
LoadBitmap	装入位图资源	是	是	是
LoadIcon	装入图标资源	是	是	是
LoadMenu	装入菜单资源	是	是	是
mmioAscend	超出 RIFF 块	否	是	是
DdeFreeStringHandle	释放 DDE 字符串句柄	否	是	是
FreeDDElParam	释放 DDE 消息 IPARAM	否	是	是
DdeFreeDataHandle	释放 DDE 数据对象	否	是	是
DdeUnaccessData	释放 DDE 数据对象	否	是	是
FreeSid	释放 SID	否	是	是
HeapFree	释放从堆中分配的内存	否	是	是
ReleaseMutex	释放公制对象	否	是	是
DragFinish	释放分配用于托动文件的内容	否	是	是
SHFreeNameMappings	释放文件名映像对象	否	是	是

（续表）

Api 函数名	函数说明	适用范围		
		W16	W98	WNT
LocalFree	释放本地内存信息	是	是	是
GlobalFree	释放全局内存块	是	是	是
ReleaseDC	释放设备描述表	是	是	是
DdeUninitialize	释放应用程序的 DDEML 资源	否	是	是
TlsFree	释放线索本地存储索引	否	是	是
ReleaseSemaphore	释放信号量对象	否	是	是
UnregisterHotKey	释放热键	否	是	是
HeapUnlock	释放堆锁	否	是	是
FreeConsole	释放控制台	否	是	是
VirtualFree	释放虚拟页	否	是	是
ReleaseCapture	释放鼠标捕获	是	是	是
joyReleaseCapture	释放操纵杆捕获	否	是	是
LockFile	锁定一个字节范围	否	是	是
LockFileEx	锁定一个字节范围	否	是	是
GlobalLock	锁定内存对象并返回一个指针	是	是	是
LocalLock	锁定本地内存对象并返回指针	是	是	是
LockServiceDatabase	锁定指定 SC 管理器数据库	否	是	是
HeapLock	锁定堆	否	是	是
VirtualLock	锁定虚拟页	否	是	是
DlgDirList	填充目录列表框	是	是	是
DlgDirListComboBox	填充目录列表框	是	是	是
FillPath	填充当前路径	否	是	是
FlushInstructionCache	填满指令缓冲区	否	是	是
ScrollConsoleScreenBuffer	滚动屏幕缓冲区中的数据	否	是	是
LockWindowUpdate	禁止或使能在窗口中的绘画	是	是	是
TabbedTextOut	输出字符串	是	是	是
TextOut	输出字符串	是	是	是
LoadKeyboardLayout	键盘布置装入内存	否	是	是
ImpersonateDdeClientWindow	模拟 DDE 客户窗口	否	是	是
ImpersonateLoggedOnUser	模拟指定用户	否	是	是
ImpersonateNamedPipeClient	管道服务器模拟客户	否	是	是

（续表）

Api 函数名	函数说明	适用范围		
		W16	W98	WNT
InterlockedIncrement	增加 LONG	否	是	是
SetProp	增加或修改属性列项	是	是	是
PlaySound	播放声音文件	否	是	是
sndPlaySound	播放声音文件	否	是	是
ActivateKeyboardLayout	激活一个新的键盘设备	否	是	是
SetActiveWindow	激活顶级窗口	是	是	是
OpenIcon	激活最小化窗口	是	是	是
InvertRgn	翻转区域颜色	是	是	是
InvertRect	翻转矩形区域	是	是	是

附录 3　Delphi 网络资源

- http://www.borland.com

Borland 公司的官方网站，包含大量信息和资源。

- http://vcl.vclxx.com

32bit 深度历险站是中文 Delphi 的第一大站，包含大量的 Delphi 组件，内容非常丰富，而且更新速度很快，基本上每周更新 1~2 次。组件分类很细，查找起来很方便。该站点有多个镜像。

- www.gexpert.org

著名的 Delphi 专家网站，提供了目前应用很广泛的 GExpert Delphi 专家，在开发过程中提供了非常好的帮助。

- http://sunsite.icm.edu.pl/delphi/

Delphi Super Page，位于波兰华沙大学内，其镜像站遍布于全世界各地，总数达到了几十个。该站组件、资料收集数量庞大，分类完整清晰，查找十分方便，最重要的一点是由于该站的规模和名气都很大，许多组件、资料的作者都乐于把他们的作品首先在这个站上发表，所以该站的更新速度很快。

- http://www.torry.ru/

Torry's Delphi Page，这是个和 Delphi Super Page 齐名的 Delphi 大站，内容十分丰富，分类很多，查找很方便。

- http://develepor.href.com/

非常出色的技术站点，可以查找 Borland、Microsoft 新闻组和 Internet 开发新闻组的所有资料，是查找技术资料的首选站点。

- http://www.expert-exchange.com/

Expert-Exchange 是一个十分有特色的技术论坛站点，采用积分制，提出问题必须给分，回答了问题可以得分。其中有非常多的技术专家，只要用户提出问题，就能够得到答案。

- http://www.gislab.ecnu.edu.cn/delphibbs/

Delphi 大富翁站，这是国内仿 Delphi Expert-Exchange 的站点，站长是孙以义博士，比起 Expert-Exchange，有自己的中国特色，例如 E-mail 通知、分组讨论专题等。

- http://202.96.154.12:88/forum/borland/main.html

中国大陆程序员论坛，是中国人气最旺的 Inprise **技术论坛，有很多有用信息，开发** Delphi **程序不可不去**!

- http://www.drbob42.com/

Drbob's Clinic，**国外著名的** Delphi **个人站点，以高质量技术文件闻名，讨论的都是高级问题，近来专注于网上程序的开发。**

- http://www.undu.com/

Undu，**著名的** Delphi **电子杂志《**UNDU**》，非常丰富的技术资料，备有** HLP、HTML **两种格式的文件可供选择。**

- http://www.delphideli.com/

The Delphi Deli，**非常实用的英文** Delphi **站，收罗十分丰富。**

- http://www.user.xpoint.at/r.fellner/delphi2.htm

Delphi Box，**典型的技术派主页，整洁干净，内容极其丰富，尤其是网络资源收罗极多，有上千个网络连接。**

- http://www.delphipages.com/

Delphi Pages，Delphi **资源大站，包括各种控件和文档。**

- http://www.efg2.com/lab/

efg's Computer Lab，**包含大量** Delphi **示例，算法和网络资源等。**

- http://www.delphi32.com/

Delphi32.com，Delphi **技术论坛和资料下载。**

- http://www.chih.com/

钱达智的 Delphi **学习笔记** (Big5)，**钱达智在台湾** Delphi **界有很高的知名度，他的问答集是学习** Delphi **的不可多得的宝贵资料。**

- http://www.ncc.com.tw/delphi/chinese.htm

Delphi **星际总部（**Big5**），台湾又一个中文** Delphi **大站，它的最大特点是组件收藏数量非常庞大，而且该站还配置了一个搜索引擎。**

- http://dbc.copystar.com.tw/

Delphi & BCB Chat Mail List（Big5），**著名的** Delphi & BCB Chat Mail List **的管理站。站长已经将邮件组的信件全部整理成** HTM **格式，可以下载浏览。**

- http://www.pobox.com/~bstowers/delphi/

Brad Stowers，著名的 Delphi 组件作者，这里有许多高质量带源代码的组件。

- http://members.aol.com/charliecal/index.html

Charlie Calvert，著名技术作者，著有《Delphi Unleashed》、《C++Builder Unleashed》、《Teach Yourself Windows 95 Programming in 21 days》等。

- http://super.sonic.net/ann/

Ann Lynnworth，著名 Delphi/Internet 开发者，Href 创始人。

- http://www.geocities.com/SiliconValley/lakes/1636

Alex Fedorov，著名俄国技术作家和 Delphi 开发者。

- http://www.execpc.com/~dmiser/

Dan Miser，著名 MIDAS 和 DCOM 技术专家（well-known guru in MIDAS and DCOM）

- http://www.teleport.com/~greglief/

Greg Lief，《Delphi Object Lessons》技术刊物主编，著有《Delphi Database Development》。

- http://www.marcocantu.com/

Marco Cantu，著名 Delphi 作家，著有《Mastering Delphi 系列》and 《Delphi Developer's Handbook》。

- http://ourworld.compuserve.com/homepages/mikes/

Mike Scott Team Borland 的主要成员。

- http://www.apci.net/~nhodges/home.htm

Nick Hodges Team Borland 的主要成员。

- http://www.geocities.com/SiliconValley/Peaks/8307/

Sergey Kolchin，《Lotus Notes Delphi classes》的作者。

- http://www.geocities.com/SiliconValley/Way/7092/

Sergey Kucherov，ABAP/4 技术专家，也是《Lotus Notes Delphi classes》的作者。

- http://www.castle.net/~bly/Programming/index.html

Binh Ly，著名 COM 专家，著有《Implementing COM Component Callbacks in Delphi》and 《Understanding COM Threading using Delphi》。

- http://delphi21.163.net/

Delphi 每日更新，包含很多 Delphi 源代码控件。每天更新!

- http://personal.gz168.net/jianl/

Delphi WWW 报告各家 Delphi 站点的更新情况。工作很辛苦。但对 Delphi 网上用户来说太方便了!

- http://www.nease.net/~wanghs/

王寒松 Delphi，对 Linux 更感兴趣的 Delphi 站点。

- http://www.nbip.net/michaeljia

C++ Builder & Delphi 之家,有许多有用的 DELPHI 资料及程序分析,控件设计课程等。

- http://www.963.net/~yinchh/

DELPHI 圣殿，相当出色的站点，资料非常丰富，可惜久未更新，不过还是值得一去。

- http://www.nease.net/~bozhi/

深圳同丰实业有限公司在网易上的个人主页，有不少 DELPHI 的编程心得发表。软件下载里的好东西也不少。

- http://www.zhanglong.com/softwork/

四眼工作室，整理 Delphi 的 Tips。内容很不错。

- http://202.120.100.49/tqz/

Delphi Forever，Delphi 的忠实拥护者，控件库有数百个 Delphi 控件。

- http://www.nease.net/~gui/

学程序来旋风，有 Delphi 的一些心得、控件，主持一份网上杂志《联合网路期刊》的编程版。

- http://www.netease.com/~seawave/

SeaWave 个人主页，有编程交流、新手指南、网站品评、精品软件等栏目，主要研究 DirectX。

- http://www.chinacars.com/delphi/

先锋软件工作室，最大的特点是开发了很多游戏软件，例如 24 点、飞行棋、俄罗斯方块、四连贯、国际象棋等。

- http://www.netease.com/~huangkai/

Delphi 天空，提供编程资料 Tips、控件库等。

- http://www.netease.com/~lws/

DELPHI/PASCAL PAGE，站长一直用 PASCAL 编程，因此有许多独特的东西：最佳 32bit 保护模式编译器、DOS 实模式下访问 4G 内存、XMODE 图形编程技术、VESA 标准、Windows 95 标准图形函数 GDI 等。

- http://www.netease.com/~zjfeng/

科技书库，仓库管理员邹建峰在这里专门提供电脑类书籍和资料以供下载，尤其是 Delphi 和 VB 的资料。

- http://www.turbopower.com/

Turbopower，国外著名 Delphi 控件开发商，主要产品有 ASYC PRO(通讯极品)、Orpheus (增强型控件集)、FlashFiler (取代 BDE 的数据库控件)、SysTools and Memory Sleuth。

- http://www.woll2woll.com/

Woll2Woll，国外著名 Delphi 控件开发商，主要产品有 Inforpower。

- http://home.sprynet.com/sprynet/rrm/index.html

QDB，Robert Marsh S.J.，数据库控件 QDB 和 Snoop 正式站点。

- http://www.raize.com/

Raize，著名控件集合 Raize Component，还出品 CodeSite（强大的 Delphi 调试工具）。

- http://www.lmdtools.com/

LmdTools，著名控件包 LMD Tools 厂商。

- http://www.ahm.co.za/

AHM Tools，著名控件包 AHM Tools 厂商。

- http://www.dreamcompany.com/

Dream Compnay，著名 Delphi 控件厂商。出品 Dream 系列控件，包括：Dream Tree、Dream Designer 等。

- http://rx.demo.ru/

Rxlib，最有名的免费控件包 RxLib 的官方站点。

- http://skylinetools.com/

ImageLib，著名图像控件 ImageLib。

- http://www.elevatesoft.com/

DBISAM DBISAM Database Systems for Delphi，Elevate Software

- http://www.luxent.com/default.htm

Apollo Apollo，Luxent Software

- http://www.oopsoft.com/

SQL Express SQL Express，OOPSoft Inc.

- http://www.delphi-jedi.org/

Project JEDI，the Delphi API Library JEDI 是一个非盈利志愿组织。主要工作是将各种新 Windows API 和 SDK 从 C/C++程序翻译成 Pascal/Delphi 接口程序。

以下是有关 Delphi 的新闻组

borland.public.delphi.activex.controls.using
borland.public.delphi.activex.controls.writing
borland.public.delphi.database.desktop
borland.public.delphi.database.multi-tier
borland.public.delphi.database.sqlservers
borland.public.delphi.graphics
borland.public.delphi.ide
borland.public.delphi.internet
borland.public.delphi.jobs
borland.public.delphi.non-technical
borland.public.delphi.objectpascal
borland.public.delphi.oleautomation
borland.public.delphi.opentoolsapi
borland.public.delphi.reporting-charting
borland.public.delphi.thirdparty-tools
borland.public.delphi.vcl.components.using
borland.public.delphi.vcl.components.writing
borland.public.delphi.winapi
alt.comp.lang.borland-delphi
comp.lang.pascal.delphi.components.misc
comp.lang.pascal.delphi.components.usage
comp.lang.pascal.delphi.components.writing
comp.lang.pascal.delphi.databases
comp.lang.pascal.delphi.misc