

组件编写者指南

翻译：王朋武

E-mail: darlingpeng@sina.com

25/4, 2010



Borland®
Delphi™ 7
for Windows®

Borland Software Corporation
100 Enterprise Way, Scotts Valley, CA 95066-3249
www.borland.com

组件编写者指南

参考位于 Delphi 7 产品根目录下的配置 (DEPLOY) 文档中完整的文件列表, 依照与 Delphi 7 的许可声明和授权保证, 你可以进行分发。

Borland 软件公司可以拥有含盖此文档主题的专利权和/或待授权专利应用软件。请参阅产品 CD 或可应用专利列表的‘关于’对话框。本文档内容不授权你对于这些专利的任何许可。

版权©1983-2002 属于 Borland 软件公司, 版权所有。所有 Borland 品牌和产品名称都是 Borland 软件公司在美国或其它国家的商标或注册商标。所有的其它商标都为它们各自拥有者所有。

在美国印刷

D7-CWG-0802

关于《组件编写者指南》翻译的说明

1. 译文中**粉红色**的部分，是译者认为原文有错误的地方，我已改过，只是用粉红色作了标记；
2. 对于有关“参看”的各章节的页码，我也没有建立页码的关联，因为麻烦，所以只是用原书中的页码值，同时用**蓝色**加以标记，可以直接删除；
3. 由于原书只分章，没有分节。阅读起来不便。所以，我加了章节的编号，这样更清楚。
4. 书中的“**第 V 部分的其余章节**”没有真正的对应部分，所以照译出来，只是在后面进行了说明（译者说明：**红色部分有问题，因为本书没有 Part V**）。
5. 原书中的编码注释主要使用的是{}符号，我都更换成了//符号，因为排版过程中有时出现超过一行的情况，看起来不整齐。对于长注释部分，还是用{}注释符号。
6. 源程序代码中的所有局部变量 I 改为 i，因为这是几乎所有编程者的惯例；
7. 最后的索引，没有建立与页的关联，只是翻译成了中文。

目录

第1章 组件创建概述.....	13
1.1 类库.....	13
1.2 组件和类.....	14
1.3 创建组件.....	14
1.3.1 修改已有的控件.....	14
1.3.2 创建窗口控件.....	15
1.3.3 创建图形控件.....	15
1.3.4 子类化 Windows 控件.....	15
1.3.5 创建非可视组件.....	16
1.4 组件的构成.....	16
1.4.1 去除依赖.....	16
1.4.2 设定属性、方法、事件.....	17
1.4.2.1 属性.....	17
1.4.2.2 方法.....	17
1.4.2.3 事件.....	17
1.4.3 封装图形.....	18
1.4.4 注册组件.....	18
1.5 创建新组件.....	18
1.5.1 用组件向导创建组件.....	19
1.5.2 手工创建组件.....	21
1.5.2.1 创建单元文件.....	21
1.5.2.2 派生组件.....	22
1.5.2.3 注册组件.....	22
1.5.3 为组件创建位图.....	23
1.6 在组件面板上安装组件.....	24
1.6.1 使源文件可用.....	24
1.7 测试未安装的组件.....	24
1.8 测试已安装的组件.....	26
第2章 为组件编写者的面向对象编程.....	26
2.1 定义新类.....	27
2.1.1 派生新类.....	27
2.1.1.1 改变类的缺省值以避免重复.....	27
2.1.1.2 给类添加新能力.....	28
2.1.2 声明新的组件类.....	28
2.2 祖先、子孙和类层次关系.....	28
2.3 控制访问.....	29
2.3.1 隐藏实现细节.....	29
2.3.2 定义组件编写者的接口.....	30
2.3.3 定义运行时接口.....	30
2.3.4 定义设计时接口.....	31
2.4 分派方法.....	31
2.4.1 静态方法.....	32

2.4.1.1 静态方法的一个示例.....	32
2.4.2 虚拟方法.....	32
2.4.2.1 重载方法.....	33
2.4.3 动态方法.....	33
2.5 抽象类成员.....	33
2.6 类和指针.....	34
第3章 创建属性.....	34
3.1 为什么创建属性?	34
3.2 属性的类型.....	35
3.3 发布继承的属性.....	35
3.4 定义属性.....	36
3.4.1 属性声明.....	36
3.4.2 内部数据保存.....	36
3.4.3 直接访问.....	37
3.4.4 存取方法.....	37
3.4.4.1 读的方法.....	38
3.4.4.2 写的方法.....	38
3.4.5 属性的缺省值.....	39
3.4.4.1 无缺省值的指定.....	39
3.5 创建数组属性.....	40
3.6 为子组件创建属性.....	40
3.7 为接口创建属性.....	42
3.8 存储与装载属性.....	42
3.8.1 使用存储与装载机制.....	43
3.8.2 指定缺省值.....	43
3.8.3 决定保存的内容.....	44
3.8.4 装载后的初始化.....	44
3.8.5 存储与装载未发布的属性.....	45
3.8.5.1 创建方法来存储与装载属性值.....	45
3.8.5.2 重载 DefineProperties 方法.....	46
第4章 创建事件.....	46
4.1 什么是事件?.....	47
4.1.1 事件是方法指针.....	47
4.1.2 事件是属性.....	48
4.1.3 事件类型是方法指针类型.....	48
4.1.3.1 事件处理程序类型是过程.....	48
4.1.4 事件处事程序是可选的.....	49
4.2 实现标准事件.....	49
4.2.1 识别标准事件.....	49
4.2.1.1 为所有控件的标准事件.....	49
4.2.1.2 为标准控件的标准事件.....	50
4.2.2 使事件可见.....	50
4.2.3 改变标准事件的处理.....	50
4.3 定义自己的事件.....	51

4.3.1 触发事件.....	51
4.3.1.1 二种事件.....	51
4.3.2 定义处理程序类型.....	52
4.3.2.1 简单通告.....	52
4.3.2.2 事件特定的处理程序.....	52
4.3.2.3 从处理程序返回信息.....	52
4.3.3 声明事件.....	52
4.3.3.1 事件名称以 On 开头.....	53
4.3.4 调用事件.....	53
4.3.4.1 空的处理程序必须有效.....	53
4.3.4.2 用户可以重载缺省处理.....	53
第 5 章 创建方法.....	54
5.1 消除依赖.....	54
5.2 命名方法.....	54
5.3 保护方法.....	55
5.3.1 应公开的方法.....	55
5.3.2 应受保护的方法.....	55
5.3.3 抽象方法.....	55
5.4 使方法为虚拟的.....	56
5.5 声明方法.....	56
第 6 章 在组件中使用图形.....	57
6.1 图形概述.....	57
6.2 使用画布.....	58
6.3 用图片工作.....	59
6.3.1 使用图片、图形或画布.....	59
6.3.2 装载与保存图形.....	59
6.3.3 处理调色板.....	60
6.3.3.1 为控件指定调色板.....	60
6.3.3.2 响应调色板的改变.....	60
6.4 离屏位图.....	60
6.4.1 创建和管理离屏位图.....	61
6.4.2 复制位图图象.....	61
6.5 响应改变.....	62
第 7 章 处理消息和系统通告.....	63
7.1 理解消息处理系统.....	63
7.1.1 Windows 消息中是什么?	63
7.1.2 分派消息.....	64
7.1.2.1 跟踪消息流程.....	64
7.2 改变消息处理.....	64
7.2.1 重载处理程序方法.....	65
7.2.2 使用消息参数.....	65
7.2.3 捕获消息.....	65
7.3 创建新的消息处理程序.....	66
7.3.1 定义自己的消息.....	66

7.3.1.1 声明消息标识符.....	66
7.3.1.2 声明消息记录类型.....	67
7.3.2 声明新的消息处理方法.....	67
7.3.3 发送消息.....	68
7.3.3.1 给窗体中的所有控件广播消息.....	68
7.3.3.2 直接调用控件的消息处理程序.....	69
7.3.3.3 使用 Windows 消息队列发送消息.....	69
7.3.3.4 发送不立即执行的消息.....	69
7.4 使用 CLX 响应系统通告.....	70
7.4.1 响应信号.....	70
7.4.1.1 分派定制的信号处理程序.....	71
7.4.2 响应系统事件.....	71
7.4.2.1 常用事件.....	72
7.4.2.2 重载 EventFilter 方法.....	73
7.4.2.3 产生 Qt 事件.....	74
第 8 章 制作设计时可用的组件.....	74
8.1 注册组件.....	75
8.1.1 声明 Register 过程.....	75
8.1.2 编写 Register 过程.....	75
8.1.2.1 指定组件.....	76
8.1.2.2 指定组件面板页.....	76
8.1.2.3 使用 RegisterComponents 函数.....	76
8.2 为组件提供帮助.....	77
8.2.1 创建帮助文件.....	77
8.2.1.1 创建条目.....	77
8.2.1.2 使组件的帮助上下文关联.....	78
8.2.2 添加组件的帮助文件.....	78
8.3 添加属性编辑器.....	79
8.3.1 派生属性编辑器类.....	79
8.3.2 作为文本编辑属性.....	80
8.3.2.1 显示属性值.....	80
8.3.2.2 设置属性值.....	80
8.3.3 作为整体编辑属性.....	81
8.3.4 指定编辑器特性.....	82
8.3.5 注册属性编辑器.....	83
8.4 属性分类.....	84
8.4.1 一次注册一个属性.....	84
8.4.2 一次注册多个属性.....	85
8.4.3 指定属性分类.....	85
8.4.4 使用 IsPropertyInCategory 函数.....	86
8.5 添加组件编辑器.....	86
8.5.1 给上下文菜单添加项.....	87
8.5.1.1 指定菜单项.....	87
8.5.1.2 实现命令.....	87

8.5.2 改变双击行为.....	88
8.5.3 添加剪贴板格式.....	89
8.5.4 注册组件编辑器.....	89
8.6 把组件编译到包中.....	89
第 9 章 修改已有的组件.....	90
9.1 创建并注册组件.....	90
9.2 修改组件类.....	91
9.2.1 重载构造函数.....	91
9.2.2 给属性指定新的缺省值.....	92
第 10 章 创建图形控件.....	92
10.1 创建并注册组件.....	92
10.2 发布继承的属性.....	93
10.3 添加图形能力.....	93
10.3.1 确定画什么.....	94
10.3.1.1 声明属性类型.....	94
10.3.1.2 声明属性.....	94
10.3.1.3 编写实现方法.....	95
10.3.2 重载构造函数和析构函数.....	95
10.3.2.1 改变缺省的属性值.....	95
10.3.3 发布画笔和画刷.....	96
10.3.3.1 声明类域.....	96
10.3.3.2 声明访问属性.....	96
10.3.3.3 初始化自有类.....	97
10.3.3.4 设定自有类的属性.....	98
10.3.4 绘制组件图象.....	99
10.3.5 改进形状绘制.....	100
第 11 章 定制网格.....	101
11.1 创建并注册组件.....	101
11.2 发布继承的属性.....	102
11.3 改变初始值.....	103
11.4 重定网格单元的大小.....	103
11.5 填充网格单元.....	104
11.5.1 跟踪日期.....	105
11.5.1.1 保存内部日期.....	105
11.5.1.2 访问年、月、日.....	106
11.5.1.3 产生日数字.....	107
11.5.1.4 选择当前日.....	109
11.6 操纵年、月.....	110
11.7 操纵日.....	110
11.7.1 移动选择.....	111
11.7.2 提供 OnChange 事件.....	111
11.7.3 排除空白单元.....	112
第 12 章 制作数据感知控件.....	112
12.1 创建数据浏览控件.....	113

12.1.1 创建并注册组件.....	113
12.1.2 使控件只读.....	114
12.1.2.1 添加 ReadOnly 属性.....	114
12.1.2.2 允许需要的更新.....	115
12.1.3 添加数据链接.....	116
12.1.3.1 声明类域.....	116
12.1.3.2 声明访问属性.....	116
12.1.3.3 声明访问属性的示例.....	117
12.1.3.4 初始化数据链接.....	118
12.1.4 响应数据改变.....	118
12.2 创建数据编辑控件.....	119
12.2.1 改变 FReadOnly 的缺省值.....	120
12.2.2 处理鼠标按下和键盘按下消息.....	120
12.2.2.1 响应鼠标按下消息.....	120
12.2.2.2 响应键盘按下消息.....	121
12.2.3 更新域数据链接类.....	122
12.2.4 修改 Change 方法.....	122
12.2.5 更新数据集.....	123
第 13 章 使对话框成为组件.....	124
13.1 定义组件接口.....	125
13.2 创建并注册组件.....	125
13.3 创建组件接口.....	126
13.3.1 包括窗体单元.....	126
13.3.2 添加接口属性.....	126
13.3.3 添加 Execute 方法.....	127
13.4 测试组件.....	128
第 14 章 扩充 IDE.....	129
14.1 Tools API 概述.....	129
14.2 编写向导类.....	130
14.2.1 实现向导接口.....	131
14.2.2 安装向导包.....	131
14.3 获取 Tools API 服务.....	132
14.3.1 使用固有的 IDE 对象.....	132
14.3.1.1 使用 INTAServices 接口.....	133
14.3.1.2 给图象列表添加图象.....	133
14.3.1.3 给动作列表添加动作.....	133
14.3.1.4 删除工具条按钮.....	134
14.3.2 调试向导.....	135
14.3.3 接口的版本号.....	136
14.4 使用文件和编辑器工作.....	136
14.4.1 使用模块接口.....	136
14.4.2 使用编辑器接口.....	137
14.5 创建窗体和工程.....	138
14.5.1 创建模块.....	138

14.6 给向导通报 IDE 事件	141
索引	146

表索引

- 1.1 组件创建开始点
- 2.1 对象内部可见性的级别
- 3.1 对象查看器中属性如何显示
- 6.1 画布能力总结
- 6.2 图象复制方法
- 7.1 为响应系统通告的 `protected TWidgetControl` 方法
- 7.2 为响应来自控件的事件的 `protected TWidgetControl` 方法
- 8.1 预定义的属性编辑器类型
- 8.2 读写属性值的方法
- 8.3 属性编辑器特性标志
- 8.4 属性分类
- 14.1 四种向导
- 14.2 Tools API 服务接口
- 14.3 Notifier 接口

图索引

- 1.1 可视组件库类的层次
- 1.2 组件向导
- 7.1 信号路由
- 7.2 系统事件路由

第 1 章 组件创建概述

本章提供对 Delphi 应用程序的组件设计与组件编写过程的概述。在这里，假设读者熟悉 Delphi 及其标准组件。

- 类库
- 组件和类
- 创建组件
- 组件的构成
- 创建新组件
- 测试未安装的组件
- 测试已安装的组件
- 在组件面板上安装组件

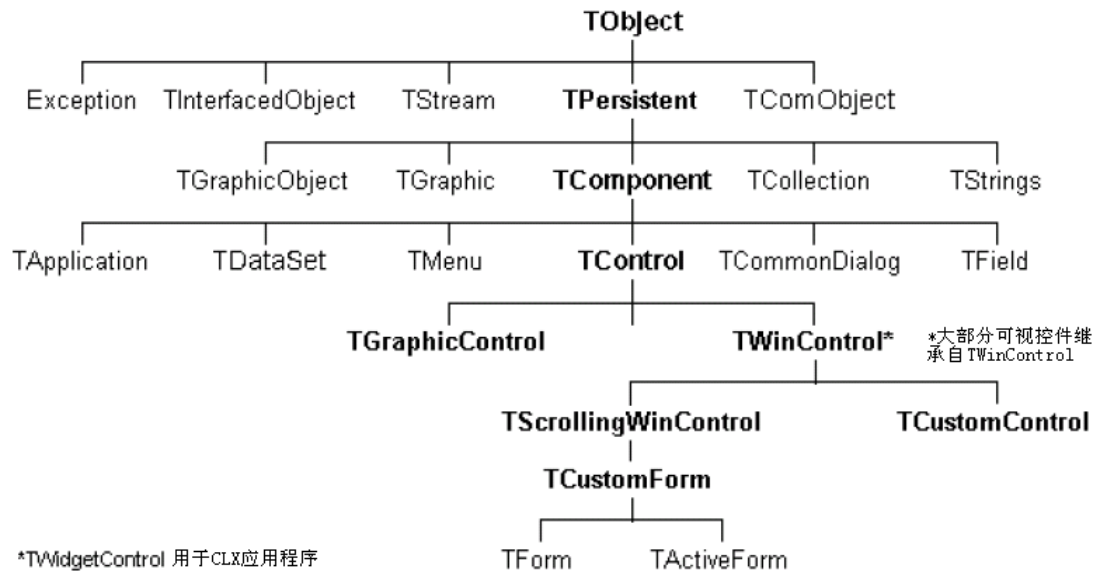
有关安装新组件的知识，参见《开发者指南》中 16-10 页的“安装组件包”。

1.1 类库

Delphi 的组件驻留在包括可视组件库（VCL）和跨平台组件库（CLX）的组件库中。图 1.1 显示了构成 VCL 层次结构的被选择类的关系。CLX 的层次关系与 VCL 的相类似，但是 Windows 控件被称为部件（所以，例如 TWinControl 被称为 TWidgetControl），并且还存在着其它方面的差异。对于类之间的类层次和继承关系的更详细的讨论，参见第 2 章“为组件编写者的面向对象编程”。就相互之间层次关系如何不同的概述，参见《开发者指南》的 15-7 页的“WinCLX 与 VisualCLX”，并为有关组件的更详细的内容，参阅 CLX 在线参考。

TComponent 类是组件库中每一个类的共同祖先。TComponent 提供组件要在 IDE 工作所必需的最少的属性和事件。这个库的各个不同的分支提供其它的、更专门的能力。

图 1.1 可视组件库类层次关系



当创建组件时，通过从层次关系中已存在的一个类类型派生出新的类，添加到组件库。

1.2 组件和类

因为组件是类，组件编写者在与应用程序开发人员不同的级别上用对象进行工作。创建新组件需要派生新的类。

简单地说，在创建组件和于应用程序中使用组件之间存在二点主要的不同。当创建组件时：

- 访问应用程序编程人员无权访问的类的部分
- 给组件添加新的部分（比如，属性）

由于这些不同，所以就需要知道更多的约定，并考虑应用程序开发人员会如何使用你编写的组件。

1.3 创建组件

组件可以是在设计时你想操作的几乎任何程序元素。创建组件意味着从已存在的一个组件派生新的类。可以用以下几种途径派生新组件：

- 修改已有的控件
- 创建窗口控件
- 创建图形控件
- 子类化 Windows 控件
- 创建非可视组件

表 1.1 总结了不同的组件和类，你可以把它们用作创建每一种组件的起点。

表 1.1 组件创建的起点

做什么	以这个类型开始
修改已有的组件	任何已有的组件，比如 TButton、TListBox，或抽象组件类型，如 TCustomListBox
创建窗口控件（或 CLX 应用程序中基于部件的控件）	TWinControl（在 CLX 应用程序中的 TWidgetControl）
创建图形控件	TGraphicControl
子类化控件	任何 Windows 控件（VCL 应用程序）或基于部件的控件（CLX 应用程序）
创建非可视组件	TComponent

也可以派生不是组件并且不能在窗体上被操作的类，如 TRegIniFile 和 TFont。

1.3.1 修改已有的控件

创建组件最简单的方法是定制一个已有的组件。可以从组件库提供的任何组件中派生新的组件。

一些控件，比如，列表框和网格，在基本主题上有几个变化。在这些情况下，组件库包括一个抽象类（在其名字中含有字“custom”，比如 TCustomGrid），由此抽象类派生定制的版本。

例如，你可能想创建一个特殊的列表框，它没有标准 `TListBox` 类的有些属性。由于不能删除（隐藏）从祖先类继承的属性，所以，就需要从层次关系中 `TListBox` 上面的一些类中派生组件，而不是强迫你从抽象类 `TWinControl`（或 CLX 应用程序中的 `TWidgetControl`）开始并重写全部的列表框的函数。组件库提供 `TCustomListBox`，它实现列表框的属性，但是并不发布全部属性。当从 `TCustomListBox` 这样的抽象类派生组件时，只发布你想使其在你的组件中可用的属性，而置其余的属性为受保护的（`protected`）。

第 3 章“创建属性”解释如何发布继承的属性。

第 9 章“修改已有组件”和第 11 章“定制网格”给出修改已有控件的示例。

1.3.2 创建窗口控件

组件库中的窗口控件是在运行时显示的对象，而且用户可与其交互。每一个窗口控件都有一个窗口句柄，通过其 `Handle` 属性进行访问，它让操作系统识别控件并在此控件上操作。如果使用 VCL 控件，句柄允许控件接收输入焦点，并可被传递到 Windows API 函数。CLX 控件是基于部件的控件，每一个基于部件的控件都有一个句柄，通过其 `Handle` 属性被访问，它标识底层的部件。

所有窗口控件都来自于 `TWinControl` 类（在 CLX 中是 `TWidgetControl`）。这些控件包括了大部分的标准控件，比如，按钮、列表框和编辑框。由于能够直接从 `TWinControl`（在 CLX 的 `TWidgetControl`）派生原始控件（original control）（即不与已有的任何控件关联的控件），Delphi 为此提供 `TCustomControl` 组件。`TCustomControl` 是一个特殊的窗口控件，它使绘制复杂的可视图象变得容易。

第 11 章“定制网格”提供了创建窗口控件的一个示例。

1.3.3 创建图形控件

如果控件不需要接收输入焦点，那么就可以使它成为图形控件。图形控件与窗口控件类似，但是没有窗口句柄，因此就较少地消耗资源。象 `TLabel` 这样的组件绝不会接收输入焦点，所以它们是图形控件。尽管这些控件不能接收输入焦点，但是可以把它们设计为对鼠标消息有反应。

可以经由 `TGraphicControl` 组件来创建定制控件。`TGraphicControl` 是从 `TControl` 派生的抽象类，虽然可以直接从 `TControl` 派生控件，但是最好还是从 `TGraphicControl` 开始，它提供了在 Windows 上不停地描绘的画布，处理 `WM_PAINT` 消息。所有需要做的只是重载 `Paint` 方法。

第 10 章“创建图形控件”提供了创建图形控件的一个示例。

1.3.4 子类化 Windows 控件

在传统的 Windows 编程过程中，可以通过定义新的‘窗口类’并向 Windows 注册这个类来创建定制的控件。窗口类（它与面向对象编程中的对象或类相似）包含被相同类型控件的实例所共享的信息。可以在已有类的基础上建立新的窗口类，此被称为子类化。然后把你的控件放在动态链接库（DLL）中，这非常象标准的 Windows 控件，并给它提供了接口。

可以创建围绕在任何已有窗口类周围的组件“包装器（wrapper）”。所以，如果已经有

了想用在 Delphi 应用程序中的定制控件的库,就可以创建象你自己的控件一样运行的 Delphi 组件,并且象对待任何其它组件一样从这些组件派生新的控件。

有关用于子类化 Windows 控件的技术的例子,参见提供标准 Windows 控件的 StdCtrls 单元中的组件,比如 TEdit。对于 CLX 应用程序,参见 QStdCtrls。

1.3.5 创建非可视组件

非可视组件被用作象数据库 (TDataSet 或 TSQLConnection)、系统时钟 (TTimer) 这样的元素的接口,并被用作对话框 (TCommonDialog (VCL 应用程序) 或 TDialog (CLX 应用程序) 和其子孙) 的占位符。你编写的大部分组件很可能都是可视控件。非可视组件可以从所有组件的抽象基类 TComponent 直接派生。

1.4 组件的构成

要使你的组件成为 Delphi 环境下的可靠部分,需要遵循设计时的一些约定。本节讨论下列主题:

- 去除依赖
- 设定属性、方法、事件
- 封装图形
- 注册组件

1.4.1 去除依赖

使组件可用的一个品质是对其代码能够做的事情不做任何限制。按其性质,组件以多样的组合、顺序和上下文被组合进应用程序。你应该设计组件使其可在任何环境下无条件地工作。

去除依赖的一个例子是 TWinControl 的 Handle 属性。如果你以前编写过 Windows 应用程序,就应该知道,使程序运行最困难并易于犯错的一个方面是:确信不要试图访问窗口控件,除非通过调用 CreateWindow API 函数已经创建了这个窗口控件。Delphi 减轻了用户在这方面的负担,它确保在需要时总是有一个有效的窗口句柄是可用的。通过使用属性来提供窗口句柄,这样,控件可以检查窗口是否已经被创建。如果句柄无效,那么控件就创建一个窗口,并返回句柄。因此,无论任何时候,当应用程序的代码访问窗口的 Handle 属性时,都会保证有一个有效的句柄。

通过去除象创建窗口这样的后台任务,Delphi 组件容许开发人员集中注意力在他们确实想做的事情上。在传递窗口句柄到 API 函数以前,不需要检查这个句柄是否存在,也不需要创建窗口。应用程序开发人员可以假定需要的准备已经就绪,而不必经常性地检查可能会发生问题的事情。

尽管创建没有依赖的组件可以从容进行,但是一般还是要花费很多时间。它不仅免除了应用程序开发人员做重复性和繁重的工作,而且减少了文档和技术支持的负担。

1.4.2 设定属性、方法、事件

除了在窗体设计器中被操作的可见图象以外，组件的最显而易见的特性是其属性、事件和方法。它们每一个在本书中都有一章进行专门论述，但是下面的讨论，解释其使用的动机。

1.4.2.1 属性

属性给应用程序开发人员设置和读出变量值的错觉，在这个过程中，容许组件编写者隐藏底层的数据结构，或者实现访问数据时的特殊处理。

使用属性有以下几个优点：

- 属性在设计时可用。应用程序开发人员可以不必编写代码而设置或改变属性的初值
- 当应用程序开发人员设置属性时，可检查其值或格式。在设计时验证输入，防止错误
- 在需要时组件可以构造合适的值。程序员最普遍的错误类型或许是在变量还没有初始化时就引用这个变量。借助属性提供数据，可以保证值总是在需要时可用。
- 属性容许把数据隐藏在简单、一致的接口中。你可以改变数据在属性中被构造的方式，但不让这个改变对于应用程序开发人员可见。

第 3 章“创建属性”解释了如何给组件添加属性。

1.4.2.2 方法

类方法是对类而不是对类的特定实例操作的过程和函数。例如，每一个组件的构造函数方法（Create）是一个类方法。组件方法是对组件实例本身操作的过程和函数。应用程序开发人员使用方法来把组件应用于执行特定的动作或返回不被任何属性所包含的值。

因为方法需要代码的执行，所以方法只可在运行时被调用。方法是有用的，有以下几方面的原因：

- 方法封装与数据驻留在同一对象中的组件的功能
- 方法可以隐藏复杂的过程于简单、一致的接口中。应用程序开发人员可以调用组件的 `AlignControls` 方法，而不用了解 `AlignControls` 方法如何工作，或者它与另一个组件中的 `AlignControls` 方法有什么不同。
- 方法允许用单独一个调用来更新几个属性

第 5 章“创建方法”解释如何给组件添加方法。

1.4.2.3 事件

事件是特殊属性，它调用代码来响应运行时的输入或者其它活动。事件为应用程序开发人员提供给特定的在运行时发生的事情（比如，鼠标操作和键盘击键动作）附加特殊代码块的途径。当事件发生时，执行的代码被称为事件处理程序。

事件允许应用程序开发人员指定对不同类型输入的响应，而不用定义新组件。

第 4 章“创建事件”解释了如何实现标准事件和如何定义新事件。

1.4.3 封装图形

Delphi 通过封装各种图形工具到画布而简化了 Windows 图形。画布代表窗口或者控件的绘图表面，并包含其它类，比如画笔、画刷和字体。画布就象 Windows 设备上下文，但是它为你处理了所有的细节。

如何你以前编写过 Windows 图形应用程序，那么就已经熟悉了 Windows 图形设备接口（GDI）的需求。例如，GDI 限制可用的设备上下文数量，并需要你在销毁它们以前恢复图形对象到其初始状态。

使用 Delphi，就不必担心这些事情。要在窗体或者其它组件上绘制，就访问组件的 Canvas 属性。如果想定制画笔或者画刷，就设置其颜色或者样式。当使用完成以后，Delphi 释放这些资源。如果应用程序频繁地使用相同类型的资源，则 Delphi 缓存资源，以避免重建这些资源。

你还拥有对 Windows GDI 的所有访问权限，但是，如果使用内置于 Delphi 组件中的画布，你会经常发现，你的代码会更简单，运行速度会更快。

CLX 图形封装工作会有所不同，这时画布是绘图工具（painter）。要在窗体或者其它组件上绘制，就访问组件的 Canvas 属性。Canvas 是属性，同时也是被称为 TCanvas 的对象。TCanvas 是围绕 Qt 绘图工具的包装器，它可以借助 Handle 属性访问。你可以使用这个句柄来访问低级的 Qt 图形库函数。

如果想定制画笔或者画刷，就设置其颜色或样式。当使用完时，Delphi 或者 Kylix 会释放这些资源。CLX 应用程序也缓存这些资源。

你可以从这些组件继承来使用内置于 CLX 组件的画布。图形图象如何在组件中工作，依赖于组件继承自的对象的画布。图形的特征在第 6 章“在组件中使用图形”中有详细描述。

1.4.4 注册组件

组件在可以安装于 IDE 以前，必须进行注册。注册告诉 Delphi 在组件面板的哪里放置这个组件。你也可以定制 Delphi 保存组件在窗体文件中的途径。有关注册组件的信息，参见第 8 章“制作在设计时可用的组件”。

1.5 创建新组件

可用二种途径创建组件：

- 用组件向导创建组件
- 手工创建组件

可以使用这二个方法中的任一个来创建准备安装于组件面板上的最简单功能的组件。在安装以后，可以在设计时和运行时把新组件添加到窗体上，并进行测试。然后，可以给组件添加更多的特征，更新组件面板，并继续测试。

无论何时创建新组件，有几个必须执行的基本步骤，如下所示。这个文档中的其它示例假定，你已经知道如何执行它们。

1. 为新组件创建单元。
2. 从已有组件类型中派生组件。

3. 添加属性、方法和事件。
4. 向 IDE 注册组件。
5. 为组件创建位图。
6. 创建包（一个特殊的动态链接库），从而可以在 IDE 中安装组件。
7. 为组件与其属性、方法和事件创建帮助文件。

注释 创建帮助文件用于指导组件用户如何使用这个组件，此是可选的。

当你完成时，完整的组件包括下列文件：

- 包文件（.BPL）或者包集文件（.DPC）
- 编译过的包文件（.DCP）
- 编译过的单元文件（.DCU）
- 组件面板位图文件（.DCR）
- 帮助文件（.HLP）

你也可以创建一个位图来代表你的新组件。参见 1-13 页的“为组件创建位图”。

在第 V 部分的其余章节中，解释构建组件的所有方面，并提供几个完整的编写不同类型组件的示例。（译者说明：红色部分有问题，因为本书没有 Part V）

1.5.1 用组件向导创建组件

组件向导简化创建组件的入门台阶。当你使用组件向导时，需要指定：

- 从其派生组件的类
- 新组件的类名称
- 你想显示组件的组件面板页
- 在其中创建组件的单元的名称
- 查找单元文件的搜索路径
- 你想存放组件的包名称

组件向导执行与你手工创建组件时相同的任务：

1. 创建单元
2. 派生组件
3. 注册组件

组件向导不能给已有单元添加组件。必须手工给已有单元添加组件。

1. 要启动组件向导，选择这二个方法中的一个：
 - 选择“组件 | 新组件”菜单项
 - 选择“文件 | 新的 | 其它”菜单项，并双击组件
2. 填充组件向导中的域：
 - 在‘祖先类型’域中，指定你派生新组件的类。

注释 在下拉列表中，很多组件用不同的单元名被列了二次，一个用于 VCL 应用程序，另一个用于 CLX 应用程序。CLX 特定的单元以 Q 开头（比如，QGraphics 取代 Graphics）。要确保从正确的组件继承而来。

- 在‘类名’域中，指定新组件类的名称
- 在‘组件面板页’域中，指定你想让新组件被安装到其中的组件面板的页
- 在‘单元文件名’域中，指定你想让组件类被声明在其中的单元的名称。如果单元不在搜索路径上，则编辑‘搜索路径’域中的搜索路径为需要的路径。

图 1.2 组件向导

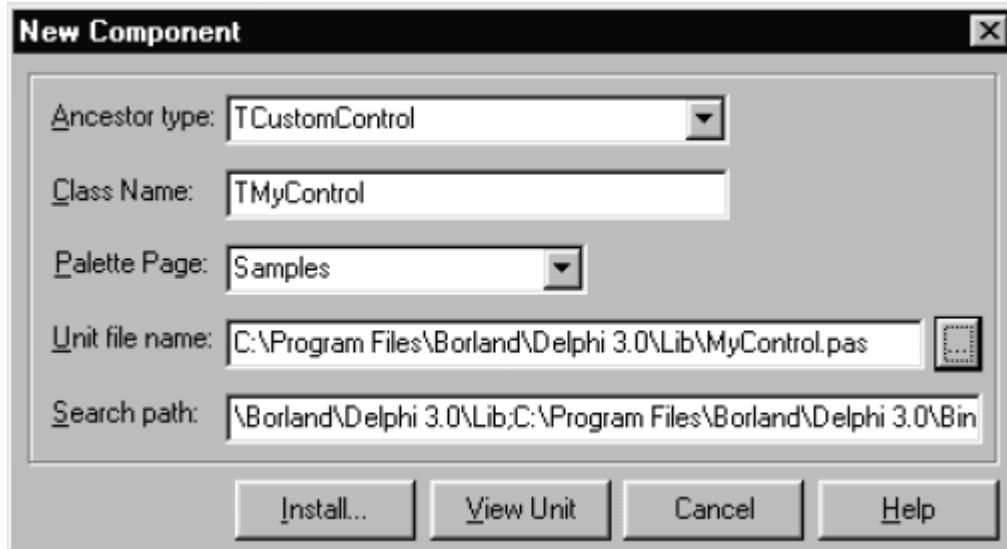


图 1- 1 组件向导

3. 填充了组件向导中的域以后，也可以：
 - 点击‘安装’。要把组件放在新包或者已存在的包中，点击‘组件 | 安装’，并用显示的对话框来指定包。参见 [1-16 页](#)的“测试未安装组件”。
4. 点击‘确认’。IDE 创建新单元。

警告 如果从名字以“custom”开头的类中派生了组件（比如，TCustomControl），就不要在窗体中试图放置此新组件，除非你已经重载了原始组件中的任何抽象方法。

Delphi 不能创建有抽象的属性或者方法的类的实例对象。

要检查单元的源代码，点击‘查看单元’（如果组件向导已经关闭，选择‘文件 | 打开’，在代码编辑器中打开单元文件）。Delphi 创建包含类声明和 Register 过程的新单元，并添加包括所有标准 Delphi 单元的 uses 语句。

单元看起来如下：

```
unit MyControl;
interface
uses
    Windows, Messages, SysUtils, Types, Classes, Controls;
Type
    TmyControl = class(TCustomControl)
    private
        { Private declarations }
    protected
        { Protected declarations }
    public
        { Public declarations }
    published
        { Published declarations }
    end;
```

```

procedure Register;
implementation
procedure Register;
begin
    RegisterComponents('Samples', [TmyControl]); // In CLX, use a different page
    than 'Samples'
end;
end

```

注释 在 CLX 应用程序中使用不同的单元，它们被以 Q 开头的有相同名称的单元取代，比如，QControls 取代 Controls。如果来自 QControls 单元中的 TCustomControl，则唯一的不同是 uses 语句，它们看起来如下：

```

uses
    SysUtils, Types, Classes, QGraphics, QControls, QForms, QDialogs,
    QStdCtrls;

```

1.5.2 手工创建组件

创建新组件最容易的方法是使用组件向导，但是，你也可以手工执行相同的步骤。

要手工创建组件，按下列步骤：

1. 创建单元文件
2. 派生组件
3. 注册组件

1.5.2.1 创建单元文件

单元是 Delphi 代码被分别编译的模块。Delphi 使用单元有几个目的。每一个窗体有其自己的单元，大部分组件（或相关单元的组）也有它们自己的单元。

当创建组件时，可以为此组件创建一个新单元，或者给已有的单元添加新组件。

要为组件创建新单元：

1. 选择下面任一菜单项：
 - 文件 | 新的 | 单元
 - 文件 | 新的 | 其它，显示出‘新项目’对话框，选择‘单元’，并选择‘确认’
 IDE 创建新单元文件，并在代码编辑器中打开它。
2. 用有意义的名字保存文件。
3. 派生组件类

要打开已有单元：

1. 选择‘文件 | 打开’，并选择你要给其添加组件的源代码单元。

注释 当给已有单元添加组件时，确保单元只包含组件代码。例如，给一包含窗体的单元添加组件代码，会在组件面板中引起错误。

2. 派生组件类

1.5.2.2 派生组件

每一个组件都是一个类，它派生自 `TComponent`，或者派生自其更复杂的子孙（比如，`TControl` 或 `TGraphicControl`），或者派生自己有组件类的类。1-3 页的“创建组件”描述了派生不同类型组件的类。

有关派生类的内容，在 2-2 页的“定义新类”中有更详尽的解释。

要派生组件，给将包含组件的单元中的 `interface` 部分添加对象类型声明。

一个简单的组件类是直接从 `TComponent` 继承来的非可视组件。

要创建简单组件类，给组件单元的 `interface` 部分添加下面的类声明：

```
type
    TNewComponent = class(TComponent)
    end;
```

到目前为止，新组件还没有做什么不同于 `TComponent` 的任何事情。你已经创建了一个框架，在这个框架下来构建新组件。

有关派生类的内容，在 2-2 页的“定义新类”中有更详尽的解释。

1.5.2.3 注册组件

注册是一个简单的过程，它告诉 IDE 哪一个组件要添加到其组件库中，并应该出现在组件面板的哪一页上。对于注册过程更详细的讨论，参见第 8 章“制作在设计时可用的组件”。

要注册组件：

1. 给组件单元的 `interface` 部分添加名为 `Register` 的过程。`Register` 没有参数，所以声明很简单：

```
procedure Register;
```

如果你正在给已经包含了组件的单元中添加组件，那么就应该已经有一个声明过的 `Register` 过程，所以不需要改变声明。

注释 尽管 Delphi 是大小写不敏感的语言，但是 `Register` 过程是大小写敏感的，并且必须以大写字母 `R` 开头。

2. 在单元的 `implementation` 部分编写 `Register` 过程，对每一个你想注册的组件，调用 `RegisterComponents`。`RegisterComponents` 是一个过程，它有二个参数：组件面板页的名称、一套组件类型。如果你正在给已存在的注册添加组件，可以在已有语句的设置中添加新的组件，或者添加调用 `RegisterComponents` 的新语句。

要注册名为 `TMyControl` 的组件，并且把它放在组件面板的‘Samples’页上，你应该给包含 `TMyControl` 声明的单元中添加下列 `Register` 过程：

```
procedure Register;
begin
    RegisterComponents('Sample', [TNewControl]);
end;
```

这个 `Register` 过程，把 `TMyControl` 放在组件面板的 `Samples` 页上。

一旦注册了组件，就可以把它编译进一个包（参见第 8 章“制作在设计时可见的组件”）中，并把它安装于组件面板上。

1.5.3 为组件创建位图

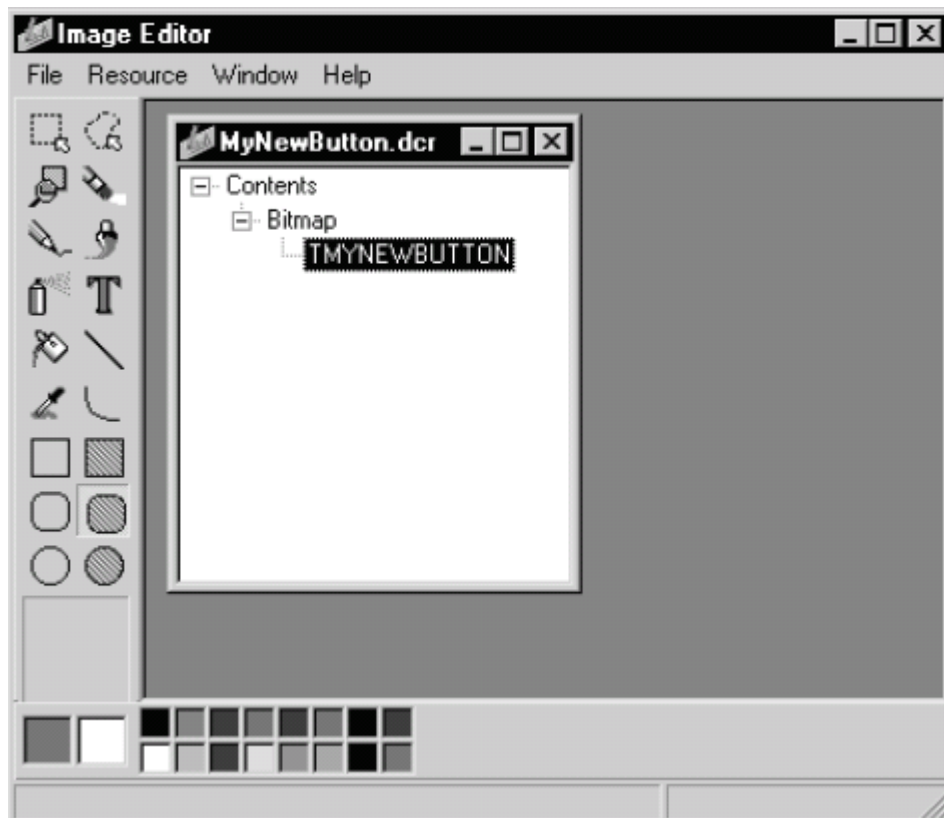
每一个组件都需要有在组件面板上代表它自己的位图。如果不指定自己的位图，IDE 会使用缺省的位图。因为面板的位图只有在设计时才需要，所以就不用把它们编译进组件的编译单元中，而是在与单元同名的 Windows 资源文件中提供它们，但是这个文件用 .dcr（动态组件资源）扩展名。可以使用图像编辑器创建资源文件。

当创建新的组件时，可以为定制的组件定义自己的位图。

要创建新位图：

1. 选择‘工具 | 图像编辑器’
2. 在图像编辑器对话框中，选择‘文件 | 新的 | 组件资源文件 (.dcr)’
3. 在 untitled1.dcr 对话框中，用鼠标右键点击‘内容’，选择‘新的 | 位图’
4. 在‘位图属性’对话框中，改变宽度和高度值都为 24 像素，确定‘VGA (16 色)’核选框被选。点击‘确认’。
5. Bitmap 和 Bitmap1 显示在‘Content’的下面，选择 Bitmap1，用鼠标右键点击，并选择‘更名’。给位图起一个与新组件类名相同的名字，包括前缀 T，全部都使用大写字母。例如，如果新类名打算取 TMyNewButton 的名字，那么就把此位图命名为 TMYNEWBUTTON。

注释 名字必须全部用大写字母，不管在‘新组件’对话框中你是如何命名类名的。



6. 用鼠标双点 TMYNEWBUTTON，显示出一个带有空白位图的对话框。
7. 使用‘位图编辑器’底部的调色板设计你的图标。
8. 选择‘文件 | 另存为’，并给资源文件 (.dcr 或 .res) 起一个与你想在其中声明组件类的单元的基本名字相同的基本名字。例如，命名资源文件为 MyNewButton.dcr。
9. 选择‘组件 | 新组件’。使用 [1-9 页](#)的“组件向导”，按使用说明创建新组件。确保

组件的源文件 `MyNewButton.pas` 与 `MyNewButton.dcr` 在同一目录下。

对名字为 `TMyNewButton` 的类来说，组件向导命名组件源文件或者单元为 `MyNewButton.pas`，放在 `LIB` 目录下缺省的位置。用鼠标点击‘浏览’按钮，在新的位置查找产生的组件单元。

注释 如果你为位图正使用 `.res` 文件而不是 `.dcr` 文件，那么给组件源添加一个引用，绑定这个源。例如，如果给 `.res` 文件起名字为 `MyNewButton.res`，在保证 `.pas` 和 `.res` 在相同的目录情况下，在 `type` 部分下面给 `MyNewButton.pas` 添加下列内容：

```
{*R *.res}
```

10. 选择‘组件 | 安装组件’，把组件安装进新包或者已有的包中。点击‘确认’。

新包已被构建，并且已被安装。代表新组件的位图显示在你于组件向导中指定的组件面板页上。

1.6 在组件面板上安装组件

要把组件安装进包中，并把它安装在组件面板上：

1. 选择‘组件 | 安装组件’。
‘安装组件’对话框出现。
2. 选择适当的页，安装新组件到已有包或新包中
3. 输入包含新组件的 `.pas` 文件的名字，或者选择‘浏览’查找单元
4. 如果新组件的 `.pas` 文件没有在显示的缺省的位置中，调整搜索路径
5. 输入要安装组件在其中的包的名称，或者选择‘浏览’查找包
6. 如果组件已被安装到新包中，如果愿意，可输入有意义的包说明
7. 选择‘确认’，关闭‘安装组件’对话框，这样就会编译 / 重建包，并会把组件安装到组件面板上。

注释 新安装的组件最初显示在由组件编写者指定的组件面板页上。在面板上安装组件以后，可以用‘组件 | 配置面板’对话框来移动组件到不同的页上。

对于需要分发组件到用户的组件编写者，要把组件安装到‘组件面板’中，参见 [1-16 页](#)“使源文件可用”。

1.6.1 使源文件可用

组件编写者应使被组件使用的所有源文件放置在相同的目录下。这些文件包括源代码文件（`.pas`）和其它工程文件（`.dfm/.xfm`，`.res/.rc` 和 `.dcr`）。

添加组件的过程导致大量文件的创建，这些文件被自动存放在由 IDE 环境选项指定的目录下（使用菜单命令‘工具 | 环境选项’，导航到‘库’标签页上）。`.lib` 文件被存放在 DCP 输出目录下。如果添加组件会导致创建一个新包（与安装组件到已有包相反），则 `.bpl` 文件被存放在 BPL 输出目录下。

1.7 测试未安装的组件

你可以在组件被安装到组件面板上以前测试其运行时行为。这尤其对于调试新创建的组

件有用，但是相同的技巧适用于任何组件，不论它是否在组件面板上。有关测试已经安装组件的知识，参见 1-18 页的“测试已安装组件”。

当从面板上选择组件并把它放在一个窗体上时，通过模拟由 Delphi 执行的动作来测试未安装的组件。

要测试未安装组件：

1. 给窗体单元的 `uses` 语句添加此组件单元的名字
2. 给窗体添加一个对象域，代表这个组件

这是你添加组件的方法与 Delphi 添加组件的方法之间的最主要的一个区别。你给窗体类型声明的底部的 `public` 部分添加这个对象域，而 Delphi 把它添加到它所管理的类型声明部分的上面。

绝不要给窗体类型声明的 Delphi 管理部分添加域。类型声明的这个部分的项相应于保存在窗体文件中的项。添加窗体中不存在的组件的名字可能会致命窗体文件无效。

3. 给窗体的 `OnCreate` 事件附加处理程序。
4. 在窗体的 `OnCreate` 处理程序中构造组件

当调用组件的构造函数时，必须传递一个指定组件拥有者的参数(当时间到时，此组件负责销毁它自己)。几乎总是传递 `Self` 作为拥有者。在方法中，`Self` 是对包含这个方法的对象的一个引用。这时，在窗体的 `OnCreate` 处理程序中，`Self` 代表这个窗体。

5. 指派 `Parent` 属性

设置 `Parent` 属性总是在构造控件以后最先做的一件事。`Parent` 是可见地包含此控件的一个组件，通常它是显示此控件的窗体，但是也可能是组框 (`group box`) 或者面板(`panel`)。通常，设置 `Parent` 为 `Self`，即这个窗体。总是在设置控件的其它属性以前就设置 `Parent`。

警告 如果你的组件不是控件 (即，`TControl` 不是其祖先之一)，跳过这一步。如果偶然设置了窗体的 `Parent` 属性 (而不是组件的 `Parent` 属性) 为 `Self`，可能会引起操作系统错误。

6. 设置组件的其它希望的属性

假设你想测试 `MyControl` 单元中的 `TMyControl` 类型的新组件，创建一个工程，然后按步骤完成窗体单元如下：

```
unit Unit1;
interface
uses
  SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls,
  Forms, Dialogs, MyControl;    // 1. Add NewTest to uses clause
Type
  TForm1 = class(TForm)
    procedure FormCreate(Sender: TObject); // 3. Attach a handler to OnCreate
  private
    { Private declarations }
  public
    { Public declarations }
    MyControl1: TMyControl1; // 2. Add an object field
end;
```

```

var
    Form1: TForm1;
Implementation
{$R *.DFM}

procedure TForm1.FormCreate(Sender: TObject);
begin
    MyControl1 := TMyControl.Create(Self); // 4. Construct the component
    MyControl1.Parent := Self; // 5. Set Parent property if component is a control
    MyControl1.Left := 12; // 6. Set other properties...
    ... // ... continue as need
end;
end.

```

1.8 测试已安装的组件

在把组件安装到组件面板上以后，你可以测试组件的设计时行为。这尤其对于调试新创建的组件有用，但是相同的技巧适用于任何组件，不论它是否已在组件面板上。有关测试尚未安装组件的内容，参见 [1-16 页](#) 的“测试未安装组件”。

在安装以后测试你的组件，容许你调试这些组件，它们在被拖放到窗体上时只产生设计时异常。

使用 IDE 的第二个运行实例，测试已安装的组件：

1. 选择‘工程 | 选项’，并在‘目录 / 条件’页上，给组件的源文件设置‘调试源路径’。
2. 然后选择‘工具 | 调试器选项’。在‘语言异常’页，使能你想跟踪的异常。
3. 打开组件源文件，设置断点。
4. 选择‘运行 | 参数’，并设置‘主机应用程序’域为 Delphi 可执行文件的名字和位置。
5. 在‘运行参数’对话框中，点击‘装载’按钮，启动 Delphi 的第二个实例。
6. 然后把要测试的这个组件放到窗体上，在源代码的断点处将中断。

第 2 章 为组件编写者的面向对象编程

如果你使用 Delphi 编写过应用程序，就会知道，类包含数据和代码，而且可以在设计时和运行时操作类。在这个意义上说，你已经成为了一个组件用户。

当创建新组件时，你用应用程序开发人员从不需要的方式处理类，也可以尝试对使用它的开发人员隐藏组件的内部工作方式。通过对组件选择合适的祖先，并设计只展现开发人员需要的属性和方法的接口，同时遵循本章的其它指导原则，就可以创建通用的、可重用的组件。

在开始创建组件以前，应该熟悉这些主题，它们与面向对象编程（OOP）有关：

- 定义新类

- 祖先、子孙和类层次关系
- 控制访问
- 分派方法
- 抽象类成员
- 类和指针

2.1 定义新类

组件编写者和应用程序开发人员的区别是组件编写者创建新类，而应用程序开发人员操作类的实例。

类本质上就是类型。作为一个程序员，总是用类型和实例工作，即使你不使用这个术语。例如，创建某类型的变量，就象 `Integer`。类通常比简单的数据类型更复杂，但是它们的工作方式是一样的：通过给相同类型的实例分派不同的值，可以执行不同的任务。

例如，创建一个窗体，它包含二个按钮：一个有标签‘确认’，另一个有标签‘取消’，这是很平常的事。每一个都是 `TButton` 类的实例，但是，通过指派不同的值给它们的 `Caption` 属性，并分派不同的处理程序给它们的 `OnClick` 事件，就会使这二个实例产生不同的行为。

2.1.1 派生新类

派生新类有二个理由：

- 改变类的缺省值，避免重复
- 给类添加新能力

在这二种情况的任一情况下，目的都是创建可重用的对象。如果你用重用的思想设计组件，就可以节省以后的工作量。给类可用的缺省值，但是允许它们被定制。

2.1.1.1 改变类的缺省值以避免重复

大部分程序员都设法避免重复。因此，如果你发现自己在反复地重写相同的代码行，就把这些代码放在子程序或函数内，或者构建一个可以在很多程序中使用的子程序库。对组件也做相同的推理。如果你发现自己正改变相同的属性或者进行相同的方法调用，就可以创建一个新组件，由它缺省地做这些事情。

例如，假设每次创建一个应用程序，都会添加一个对话框来执行特定的操作，尽管每次重建这个对话框并不困难，但也是不必要的。可以一次性设计这个对话框，设定其属性，并安装一个关联它的包装器组件在组件面板上。通过使这个对话框成为可重用的组件，就不仅消除了重复性的作业，而且促进了标准化，并减少了每次重建对话框时出现错误的可能性。

第 9 章“修改已有组件”给出一个改变组件缺省属性的例子。

注释 如果只想修改已有组件中已发布的属性，或者为组件或组件组保存特定的事件处理程序，通过创建‘组件模板’，或许能够更容易地完成这个任务。

2.1.1.2 给类添加新能力

创建新组件普遍的理由是添加在已有组件中原来没有的能力。当这样做时，就从已有组件或者抽象基类中派生新组件，比如 TComponent 或 TControl。

从包含与你想要的特征子集最接近的类中派生新组件。可以给一个类添加能力，但不能从其中取消某功能。所以，如果在已有的组件类中包含你不想要的属性，就应该从这个组件的祖先中派生。

例如，如果想给列表框添加特征，就应该从 TListBox 派生组件。但是，如果想添加新特征而排除标准列表框中的一些功能，就需要从 TCustomListBox（即 TListBox 的祖先）派生组件。然后就可以只重建（或使可见）你想要的列表框能力，并添加新的特征。

第 11 章“定制网格”给出一个定制抽象组件类的例子。

2.1.2 声明新的组件类

除标准组件以外，Delphi 还提供了很多抽象类，它们被设计为用作派生新组件的基类。在 1-3 页的表 1.1 展示了这些类，当你创建自己的组件时，可从这些类开始。

要声明新的组件类，给组件的单元文件添加类的声明。

下面是一个简单的图形组件的声明：

```
type
    TSampleShape = class(TGraphicControl)
    end;
```

已完成的组件声明通常在 end 前面包括属性、事件和方法声明。但是，象上面这样的声明也是有效的，并提供了添加组件特征的开始点。

2.2 祖先、子孙和类层次关系

应用程序开发人员都会认同，每一个控件都有在窗体上确定其位置的名为 Top 和 Left 的属性，对它们来说，也许不会在乎是否所有控件都是从一个共同的祖先 TControl 继承了这些属性。但是，当你创建组件时，就必须知道要从哪个类派生，才会继承合适的属性。同时，必须知道你的控件所继承的全部事情，这样，你就可以不用重新创建它们而是利用所继承的特征。

派生组件的类被称为它的‘直接祖先’。每一个组件都从其直接祖先继承，并从其直接祖先的直接祖先等继承。被组件所继承的所有这些类都被称为它的祖先。这个组件是其祖先的子孙。

归纳一下，在应用程序中的所有这些祖先—子孙关系组成了类的层次关系。在层次中的每一代都比其祖先包括更多的特征。类从其祖先继承了全部特征，然后添加新的属性和方法，或者重新定义已存在的属性和方法。

如果没有指定一个直接祖先，Delphi 就从缺省的祖先 TObject 派生组件。TObject 是对象层次关系中所有类的终极祖先。

选择从哪一个对象派生的一般规则是简单的：选择尽可能多地包含了你想要在新对象中包含的对象，但是，它不能包含你不想在新对象中包含的任何内容。你可以总是给你的对象

添加东西，但是不能从中取消东西。

2.3 控制访问

对属性、方法和域的访问控制有 5 个级别，也称为‘可见性’。可见性确定哪些代码可访问类的哪些部分。通过指定可见性，就可以给你的组件定义‘接口’。

表 2.1 显示了可见性的级别，从最严格限制到最可访问：

表 2.1 对象内部的可视性级别

可见性	意义	目的
private	只有在类被定义的单元中的代码可访问	隐藏实现细节
protected	在类和其子孙被定义的单元中的代码可访问	定义组件编写者的接口
public	所有代码可访问	定义运行时接口
automated	所有代码可访问。自动化类型信息被产生	只有 OLE 自动化
published	所有代码可访问，并在对象查看器中可用。被保存在窗体文件中	定义设计时接口

如果只想让成员在定义它的类内部可用，就声明成员为 `private`。如果想让成员只在类和其子孙内部可用，声明成员为 `protected`。记住，虽然成员可在其单元文件内部的任何地方可用，但是它也只是在这个文件中的任何地方。因此，如果你在同一单元中定义了二个类，那么这二个类就可相互访问对方的 `private` 方法。并且，如果你在一个不同的单元中从其祖先派生了一个类，那么在新单元中的所有类将能够访问这个祖先的 `protected` 方法。

2.3.1 隐藏实现细节

作为 `private` 的类的声明部分，使这部分对于类单元文件以外的代码不可见。在包含声明的单元内部，代码就好像是 `public` 一样，可以访问这个 `private` 部分。

下面的例子说明了如何把一个域声明为 `private`，使它对应用程序开发人员隐藏起来。源代码清单显示了二个 VCL 窗体单元，每一个窗体都有一个其 `OnCreate` 事件的处理程序，它用来给 `private` 域赋值。编译器只允许对窗体中被声明的域进行赋值。

```
unit HideInfo;
interface
uses SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls, Forms, Dialogs;
type
  TSecretForm = class(TForm) // declare new form
  procedure FormCreate(Sender: TObject);
  private // declare private part
    FSecretCde: Integer; // declare a private field
  end;
var
  SecretForm: TSecretForm;
implementation
{$R *.dfm}
procedure TSecretForm.FormCreate(Sender: TObject);
```

```

begin
    TSecretCode := 42;      // this compiles correctly
end;
end.      // end of unit

unit TestHide;      // this is the main form file
interface
uses SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls, Forms, Dialogs,
    HideInfo;      // use the unit with TSecretForm
type
    TTestForm = class(TForm)
        Procedure FormCreate(Sender: TObject);
    end;
var
    TestForm: TTestForm;
implementation
procedure TTestForm.FormCreate(Sender: TObject);
begin
    SecretForm.FSecretCode := 13;    // compiler stops with "Field identifier excepted"
end;
end.      // end of unit

```

尽管使用 HideInfo 单元的程序可以使用 TSecretForm 类型的类，但是，它不能访问在任何这些类中的 FSecretCode 域。

注释 一些单元的名称和位置与 CLX 应用程序中的有所不同。例如，Controls 单元是 CLX 应用程序中的 QControls。

2.3.2 定义组件编写者的接口

作为类的 protected 声明部分，使这部分只对类本身和其子孙（还有共享其单元文件的其它类）可见。

可以使用 protected 声明来给这个类定义一个‘组件编写者的接口’。应用程序单元没有访问 protected 部分的权限，但是，派生的类有这个权限。这就是说，组件编写者可以改变类的工作方式，而不用使细节对应用程序开发人员可见。

注释 一个普遍的错误是试图从事件处理程序来访问 protected 方法。事件处理程序是窗体的典型的方法，而不是接收事件的组件。结果，它们不能访问组件的 protected 方法（除非组件被声明在与窗体相同的单元中）。

2.3.3 定义运行时接口

总体来说，作为类的 public 声明部分，使这部分对于有权访问此类的任何代码可见。

public 部分在运行时对于全体代码都是可用的，所以，类的 public 部分定义它的‘运行时接口’。运行时接口对于设计时无意义或者不适合的项很有用，就象依赖于运行时输入的属性或者只读属性。打算供应用程序开发人员调用的方法也必须是 public。

这里有一个示例，它展示了作为组件的运行时接口被声明的二个只读属性：

```
type
  TSampleComponent = class(TComponent)
  private
    FTempCelsius: Integer; // implementation details are private
    function GetTempFahrenheit: Integer;
  public
    property TempCelsius: Integer; read FTempCelsius;    // properties are public
    property TempFahrenheit: Integer read GetTempFahrenheit;
  end;
  ...
function TSampleComponent.GetTempFahrenheit: Integer;
begin
  Result := FTempCelsius * 9 div 5 + 32;
end;
```

2.3.4 定义设计时接口

作为类的 `published` 声明部分，使这个部分是 `public`，同时也产生运行时类型信息。在其它事情中，运行时类型信息允许对象查看器访问属性和事件。

因为它们在对象查看器中显示，所以类的 `published` 部分定义这个类的‘设计时接口’。设计时接口应该包括应用程序开发人员可能想在设计时定制的类的任何方面，但是，必须排除依赖于有关运行时环境的特定信息的任何属性。

由于应用程序开发人员不能直接给只读属性赋值，所以它们不能是设计时接口的部分。因此，只读属性应该是 `public`，而不是 `published`。

这里是一个被称为 `Temperature` 的 `published` 属性的例子。因为它是 `published`，所以在设计时的对象查看器中显示。

```
type
  TSampleComponent = class(TComponent)
  private
    FTemperature: Integer; // implementation details are private
  published
    property Temperature: Integer read FTemperature write FTemperature; // writable !
  end;
```

2.4 分派方法

分派指的是当程序遇到方法调用时由程序确定应该在哪儿调用方法的途径。调用方法的代码看起来象其它任何的过程或者函数调用。但是，类有分派方法的不同途径。

有三种类型的方法分派，它们是：

- 静态的 (Static)
- 虚拟的 (Virtual)

- 动态的 (Dynamic)

2.4.1 静态方法

所有方法都是静态方法，除非在声明它们时指定为其它类型的方法。静态方法就象规范的过程或者函数一样工作。编译器确定方法的精确地址，并在编译时链接方法。

静态方法的主要优点是分派它们的速度很快。因为编译器能够确定方法的精确地址，所以就会直接链接这些方法。与此相对，虚拟的和动态的方法使用间接的手段在运行时查找这些方法的地址，这会花费更长的时间。

当由子孙类继承时，静态方法不会改变。如果你声明了一个包含静态方法的类，然后从它派生新类，那么派生的类就在相同的地址严格地共享同一方法。这意味着你不能重载静态方法。不论什么类被调用进来，静态方法总是精确地做相同的事情。如果你用与祖先类中的静态方法相同的名字在派生类中声明了一个方法，那么新的方法就在派生类中完全取代继承的方法。

2.4.1.1 静态方法的一个示例

在下列代码中，第一个组件声明了二个静态方法。第二个组件用相同的名字声明了二个静态方法，它们取代从第一个组件中继承的方法。

```
type
  TFirstComponent = class(TComponent)
    procedure Move;
    procedure Flash;
  end;

  TSecondComponent = class(TFirstComponent)
    procedure Move; // different from the inherited method, despite same declaration
    function Flash(HowOften: Integer): Integer; // this is also different
  end;
```

2.4.2 虚拟方法

虚拟方法使用比静态方法更复杂、更灵活的分派机制。虚拟方法可以在子孙类中被重定义，但还是在祖先类中被调用。虚拟方法的地址不是在编译时确定，而是由在其中定义方法的对象在运行时查找地址。

要使方法成为虚拟的，在方法声明的后面添加 **virtual** 指示。**virtual** 指标在对象的‘虚拟方法表’或 **VMT**（它保存有对象类型中的所有虚拟方法的地址）中创建一个条目。

当你从已有类中派生新类时，新类得到它自己的 **VMT**，它包含祖先的 **VMT** 加上在新类中声明的任何其它虚拟方法的所有条目。

2.4.2.1 重载方法

重载方法意味着扩展或者改进这个方法，而不是代替这个方法。子孙类可以重载它继承的任何虚拟方法。

要在子孙类中重载方法，在方法声明的末尾添加 `override` 指标。

如果发生以下情况，重载方法会引起编译错误：

- 方法在祖先类中不存在
- 相同名字的祖先类的方法是静态的
- 声明不一样（变量参数的数量和类型不同）

下面的代码显示了二个简单组件的声明。第一个声明了三个方法，每一个有不同类型的分派。第二个是从第一个派生的，它代替其静态方法，并重载其虚拟方法。

type

```
TFirstComponent = class(TCustomControl)
    procedure Move;           // static method
    procedure Flash; virtual; // virtual method
    procedure Beep; dynamic;  // dynamic virtual method
end;

TSecondComponent = class(TFirstComponent)
    procedure Move;           // declare new method
    procedure Flash; override; // override inherited method
    procedure Beep; override; // override inherited method
end;
```

2.4.3 动态方法

动态方法是分派机制稍微有些不同的虚拟方法。因为动态方法在对象的虚拟方法表中没有条目，所以它们可以减少对象消耗的内存数量。但是，分派动态方法比分派规范的虚拟方法稍有些慢。如果一个方法被频繁调用，或者其运行时间很关键，那么你或许就应该把它声明为虚拟的而不是动态的。

对象必须保存其动态方法的地址。动态方法没有虚拟方法表中的接收条目，而是被分别列出。动态方法列表只包含由特定类引入或者重载的方法的条目（相反，虚拟方法表包含继承的和引入的对象的全部虚拟方法）。通过遍历继承树，向后查找每一个祖先的动态方法列表，继承的动态方法被分派。

要使方法成为动态的，在方法声明后面添加 `dynamic` 指标。

2.5 抽象类成员

当方法作为 `abstract` 在祖先类中被声明时，必须在可能使用这个新组件于应用程序中以前，在任何的子孙组件中把它展示出来（借助重声明和实现）。Delphi 不能创建包含抽象成员的类的实例。有关展示类的继承部分的更详细的信息，参见第 3 章“创建属性”和第 5

章“创建方法”。

2.6 类和指针

每一个类（从而，每一个组件）确实是一个指针。编译器自动地为你去除类指针的引用，所以，大部分时间你不需要考虑这个问题。当作为参数传递类的时候，作为指针的类的状态就变得重要了。通常，应该用值而不是用引用传递类。理由是类已经是指针，它本来就是引用。借助引用传递类相当于给引用传递引用。

第 3 章 创建属性

属性是组件中最明显的部分。应用程序开发人员可以在设计时看到并操作它们，同时，组件在窗体设计器中一有反应，立即得到反馈。设计良好的属性会使你的组件容易被其他人使用，并且容易维护。

为了充分地使用组件中的属性，应该理解以下事项：

- 为什么创建属性
- 属性的类型
- 发布继承的属性
- 定义属性
- 创建数组属性
- 保存与装载属性

3.1 为什么创建属性？

从程序开发人员的立场来说，属性就象变量。就好象是域一样，开发人员可以设置或者读取属性的值。（变量可以做而属性不能做的唯一一件事是以 var 参数传递属性）

属性比简单的域提供更多的功能，因为：

- 应用程序开发人员可以在设计时设置属性。不象方法只能在运行时可用，在运行应用程序以前，属性让开发人员定制组件。属性可以显示在对象查看器中，它简化了程序员的工作。对象查看器提供需要的值，而不是处理几个参数来构造对象。对象查看器在属性被赋值以后也立即验证其值的有效性。
- 属性可以隐藏实现细节。例如，以加密方式于内部保存的数据可以作为属性值以非加密的方式显示出来。尽管这个值只是简单的数字，但是组件可以在数据库中查找这个值，或者执行复杂的计算来求出。属性让你在表面上通过简单的赋值来达到复杂的效果。看起来象给一个域赋值的事可以是调用一个实现精细处理过程的方法。
- 属性可以是虚拟的。因此，对应用程序开发人员来说，看起来象一个简单属性的事，或许在不同的组件中会有不同的实现。

一个简单的例子是所有控件的 Top 属性。给 Top 赋新值，不只是改变了已保存的值，它还要重新配置和重绘控件。设置属性的效果不需要局限地单个组件中，比如，设置一个加速按钮的 Down 属性为 True，就自动设置其群组中的所有其它加速钮的 Down 属性为 False。

3.2 属性的类型

属性可以是任何类型。不同的类型在对象查看器中有不同的显示，对象查看器验证属性在设计时的赋值。

表 3.1 在对象查看器中属性如何显示

属性类型	对象查看器的处理
Simple	数字、字符和串属性显示为数字、字符和串。应用程序开发人员可以直接编辑这些属性的值
Enumerated	枚举类型(包括布尔值)的属性显示为可编辑串。开发人员也可以通过双击输入值框来循环访问可能的值，并有一个下拉列表显示其所有可能的取值
Set	集合类型的属性显示为集合。通过在属性上双击，开发人员可以展开这个集合，并作为布尔值处理其每一个元素（如在集合中包含则为 True）
Object	是它们自己的类的属性经常有其自己的属性编辑器，它们在组件的注册过程中被指定。如果被属性拥有的类有其自己的 <code>published</code> 属性，则对象查看器让开发人员展开列表(通过双击)来包括这些属性并分别编辑它们。对象属性必须由 <code>TPersistent</code> 继承而来
Interface	只要值是由组件(<code>TComponent</code> 的子孙)实现的接口，则是接口的属性就可显示在对象查看器中。接口属性经常有其自己的属性编辑器
Array	数组属性必须有其自己的属性编辑器。对象查看器没有给其提供内置的编辑支持。当注册你的组件时，可以为其指定一个属性编辑器

3.3 发布继承的属性

所有组件都从其祖先类继承属性。当从已有组件派生新组件时，新组件继承其直接祖先的全部属性。如果从一个抽象类派生，很多继承的属性要么是 `protected`，要么是 `public`，但不是 `published`。

要使 `protected` 或 `public` 属性于设计时在对象查看器中可用，必须重新声明这些属性为 `published`。重新声明意味着为继承的属性在子孙类的声明中再添加一个声明。

如果从 `TWinControl` 派生组件，比如，它继承了 `protected DockSite` 属性。通过在新组件中重声明 `DockSite`，就可以改变其保护级别为 `public` 或 `published`。

下面的代码显示了 `DockSite` 作为 `published` 的重声明，使其在设计时可用。

```
type
    TSampleComponent = class(TWinControl)
    published
        property DockSite;
    end;
```

当重声明属性时，只要指定属性名即可，而不用指定在“定义属性”中描述的类型和其它信息。你也可声明其新的缺省值，并指定是否保存属性。

重声明可以使属性有较少的限制，而不是限制更多。这样，可使 `protected` 的属性变为 `public`，但是，不能通过重声明为 `protected` 而隐藏 `public` 属性。

3.4 定义属性

本节说明如何声明新属性，并解释在标准组件中遵循的一些约定。主题包含：

- 属性声明
- 内部数据保存
- 直接访问
- 访问方法
- 属性的缺省值

3.4.1 属性声明

属性在其组件类的声明中被声明。要声明属性，需指定三项：

- 属性的名称
- 属性的类型
- 读写属性值的方法。如果没有声明写方法，则属性是只读的。

被声明在组件类声明的 `published` 部分的属性于设计时在对象查看器中是可编辑的。`published` 属性的值与组件一起被保存在窗体文件中。被声明在 `public` 部分的属性在运行时是可用的，并可在程序代码中被读写。

这里是被称为 `Count` 的属性的典型声明：

```
type
  TYourComponent = class(TComponent)
  private
    FCount: Integer;    // used for internal storage
    procedure SetCount(Value: Integer);    // write method
  public
    property Count: Integer read FCount write SetCount;
  end;
```

3.4.2 内部数据保存

关于如何为属性保存数据没有任何限制。但是，通常情况下，Delphi 组件遵循下列约定：

- 属性数据被保存在类域中
- 用于保存属性数据的域是 `private`，并且应该只从组件本身的内部访问。派生的组件应该使用继承的属性，它不需要对属性的内部数据保存进行直接访问
- 域的标识符由 `F` 后跟属性名组成。比如，被定义在 `TControl` 中的 `Width` 属性的原始数据被保存在 `FWidth` 域中

构成这些约定的原则是：只有属性的实现方法应该访问属性背后的数据。如果方法或者另外的属性需要改变这个数据，那么就应该通过属性而不是通过直接访问保存的数据来达到。这确保继承属性的实现不用通过使派生的组件无效而可以改变。

3.4.3 直接访问

使属性数据可用的最简单的途径就是直接访问。即，属性声明的‘读’和‘写’部分指定直接给属性赋值，或者直接读取内部保存域，而不用调用存取方法。当你想使属性在对象查看器中可用，但是在改变其值时不会触发即时处理的时候，直接访问是有用的。

通常，属性声明的‘读’部分使用直接访问，而‘写’部分使用存取方法。这就允许当属性值改变时组件的状态被更新。

下面的组件类型声明说明了一个属性，它对读、写部分使用直接存取。

```
type
  TSampleComponent = class(TComponent)
  private
    FMyProperty: Boolean; // declare field to hold property value
  published
    // make property available at design time
    property MyProperty: Boolean read FMyProperty write FMyProperty;
  end;
```

3.4.4 存取方法

可指定存取方法代替属性声明的读、写部分的域。存取方法应该是 `protected`，并通常被声明为 `virtual`。这允许子孙组件重载属性的实现。

要避免使存取方法为 `public`。保持存取方法为 `protected`，确保应用程序开发人员不会因调用其中一个存取方法而不注意地修改了属性。

这里有一个类，它使用 `index` 指示声明了三个属性，它们允许所有三个属性有相同的读、写存取方法：

```
type
  TSampleCalendar = class(TCustomGrid)
  public
    property Day: Integer index 3 read GetDateElement write SetDateElement;
    property Month: Integer index 2 read GetDateElement write SetDateElement;
    property Year: Integer index 1 read GetDateElement write SetDateElement;
  private
    function GetDateElement(Index: Integer): Integer; // note the index parameter
    procedure SetDateElement(Index: Integer; Value: Integer);
  ...
```

由于日期的每一个元素（年、月、日）都是整数，并且因为设定每一个元素都需要编码日期，所以，对所有这三个属性通过共享其读、写方法来避免代码重复。你只需要一个方法来读日期元素，而另一个写日期元素。

这里是获取日期元素的读方法：

```
function TSampleCalendar.GetDateElement(Index: Integer): Integer;
var
  AYear, AMonth, ADay: Word;
begin
  DecodeDate(FDate, AYear, AMonth, ADay); // break encoded date into elements
```

```

        case Index of
            1: Result := AYear;
            2: Result := AMonth;
            3: Result := ADay;
            else Result := -1;
        end;
    end;
end;
这个设定合适的日期元素的写方法:
procedure TSampleCalendar.SetDateElement(Index: Integer; Value: Integer);
var
    AYear, AMonth, ADay: Word;
begin
    if Value > 0 then           // all elements must be positive
    begin
        DecodeDate(FDate, AYear, AMonth, ADay); // get current date elements
        case Index of          // set new element depending on Index
            1: AYear := Value;
            2: AMonth := Value;
            3: ADay := Value;
            else Exit;
        end;
        FDate := EncodeDate(AYear, AMonth, ADay); // encode the modified date
        Refresh;           // update the visible calendar
    end;
end;
end;

```

3.4.4.1 读的方法

属性的读方法是一个函数，它没有参数(除下面解释的以外)，并且返回与属性同类型的值。按约定，函数名称以 **Get** 开头，后跟属性名。比如，属性 **Count** 的读方法是 **GetCount**。读方法按需要操作内部保存的数据，以合适的类型产生属性值。

无参数规则的唯一例外是对数组属性和使用 **index** 限定符的属性的情况（参见 3-8 页的“创建数组属性”），它们都使用 **index** 值作为参数。(使用 **index** 限定符来创建单个的读方法，它被几个属性所共享。有关 **index** 限定符的更多的内容，参见《Delphi 语言指南》)

如果没有声明读的方法，则为只写属性。只写属性很少被使用。

3.4.4.2 写的方法

属性的写方法是一个过程，它采用与属性同类型的单个参数(除下面解释的以外)。参数可以由引用或由值传递，并且可以有您选择的任何名称。按约定，写方法的名称以 **Set** 开头，后跟属性名。比如，**Count** 属性的写方法是 **SetCount**。用参数传递的值成为属性的新值。写方法必须执行需要的任何的处理，把合适的数据放在属性的内部存储中。

单一参数规则的唯一例外是对数组属性和使用 **index** 限定符的属性的情况，它们都传递

index 值作为其第 2 个参数。(使用 index 限定符创建单个的写方法, 它由几个属性所共享。有关 index 限定符的更多信息, 参见《Delphi 语言指南》)

如果没有声明写方法, 则为只读属性。

写方法通常在改变属性以前先测试新值是否与当前值不同。例如, 下面是为整数属性 Count 的简单的写方法, 它保存当前值在 FCount 域中:

```
procedure TMyComponent.SetCount(Value: Integer);
begin
    if Value <> FCount then
    begin
        FCount := Value;
        update;
    end;
end;
```

3.4.5 属性的缺省值

当声明属性时, 可以为其指定缺省值。Delphi 使用缺省值来决定是否保存属性在窗体文件中。如果没有给属性指定缺省值, Delphi 总是保存这个属性。

要为属性指定缺省值, 添加 default 指示在属性声明(或重声明)的后面, 并且后跟缺省值。比如,

```
property Cool Boolean read GetCool write SetCool default True;
```

注释 声明缺省值并不是把属性的值设为这个值。组件的构造函数方法应该在适当的时候初始化属性值。但是, 由于对象总是初始化其域为 0, 严格来说, 不需要构造函数来设定整数属性为 0, 串属性为 null 或者布尔属性为 False。

3.4.4.1 无缺省值的指定

当重声明属性时, 可以指定此属性没有缺省值, 即使被继承的属性指定了缺省值。

要指出属性没有缺省值, 在属性声明的末尾添加 nodefault 指示。比如,

```
property FavoriteFlavor: string nodefault;
```

当第一次声明属性时, 不必要包含 nodefault。声明缺省值的缺失意味着没有缺省值。

这里是一个组件的声明, 它包含被称为 IsTrue 的单个的布尔属性, 有缺省值 True。在声明(在单元的 implementation 部分)的下面是初始化属性的构造函数。

```
type
    TSampleComponent = class(TComponent)
    private
        FIsTrue: Boolean;
    public
        constructor Create(AOwner: TComponent); override;
    published
        property IsTrue: Boolean read FIsTrue write FIsTrue default True;
    end;
...

```

```

constructor TSampleComponent.Create(AOwner: TComponent);
begin
    inherited Create(AOwner); // call the inherited constructor
    FIsTrue := True; // set the default value
end;

```

3.5 创建数组属性

有些属性会给自己添加索引，就象数组一样。比如，TMemo 的 Lines 属性就是可索引的串列表，它构成 memo 的文本。你可以把它看作串数组进行处理。Lines 提供对较大的数据集（memo 文本）中的特定元素（串）的自然访问。

数组属性的声明就象其它属性的声明一样，但以下情况例外：

- 声明包含一个或多个有指定类型的索引。这些索引可以是任何类型的索引。
- 属性声明的读、写部分，如果要指定，必须是方法，不能是域。

对于数组属性的读、写方法，相应于索引项，采用另外的参数。这些参数必须与声明中指定的索引有相同的顺序和类型。

在数组属性和数组间有少许重要的区别。不象数组中的索引，数组属性的索引不必一定是整数类型，比如，你可以按串对属性建立索引。另外，你可以只引用数组属性的单个元素，而不是属性的全部元素。

```

type
    TDemoComponent = class(TComponent)
    private
        function GetNumberName(Index: Integer): string;
    public
        property NumberName[Index: Integer]: string read GetNumberName;
    end;
...
function TDemoComponent.GetNumberName(Index: Integer): string;
begin
    Result := 'Unknown';
    case Index of
        -MaxInt..-1: Result := 'Negative';
        0:           Result := 'Zero';
        1..100:      Result := 'Small';
        101..MaxInt: Result := 'Large';
    end;
end;
end;

```

3.6 为子组件创建属性

缺省地，当属性的值是另外一个组件时，通过添加另外那个组件的实例给窗体或者数据

模块，然后指派那个组件作为属性的值，给此属性赋值。但是，你的组件也可以创建实现属性值的对象自己的实例。这样一个专门的组件称为子组件。

子组件可以是任何永久对象(TPersistent 的任何子孙)。不象个别的组件偶而作为属性值被分配，子组件的 **published** 属性与创建他们的组件被一起保存。但是，为了让这个起作用，以下的条件必须满足：

- 子组件的 **Owner** 必须是创建它并使用它作为 **published** 属性的值的组件。对于是 TComponent 子孙的子组件，可以通过设置此子组件的 **Owner** 属性来完成。对于其它子组件，必须重载永久对象的 **GetOwner** 方法，从而返回创建的组件。
- 如果子组件是 TComponent 的子孙，必须指出它是通过调用 **SetSubComponent** 方法实现的子组件。典型地，这个调用是通过创建子组件时的 **Owner**，或是通过子组件的构造函数进行的。

典型地，其值是子组件的属性是只读的。如果允许其值是子组件的属性被改变，那么，当另外一个组件作为属性值被分配时，此属性的设定函数必须释放此子组件。另外，当属性被设为 **nil** 时，此组件经常重新实例化它的子组件。另一方面，一旦这个属性被变为另一个组件，那么这个子组件在设计时决不可再被恢复。下面的例子为值是 TTimer 的属性说明了这样一个属性设定函数：

```
procedure TDemoComponent.SetTimerProp(Value: TTimer);
begin
    if Value <> FTimer then
    begin
        if Value <> nil then
        begin
            if Assigned(FTimer) and (FTimer.Owner = Self) then
                FTimer.Free;
            FTimer := Value;
            FTimer.FreeNotification(self);
        end
        else // nil value
        begin
            if Assigned(FTimer) and (FTimer.Owner <> Self) then
            begin
                FTimer := TTimer.Create(self);
                FTimer.Name := 'Timer'; // optional bit, but makes result much nicer
                FTimer.SetSubComponent(True);
                FTimer.FreeNotification(self);
            end;
        end;
    end;
end;
```

注意，上面的属性设定函数调用了组件的 **FreeNotification** 方法，而这个组件作为属性值被设置。这个调用保证是属性值的组件如果被销毁会发送通告。它通过调用 **Notification** 方法来发送这个通告。重载 **Notification** 方法，处理这个调用。如下所示：

```
procedure TDemoComponent.Notification(AComponent: TComponent; Operation: TOperation);
begin
```

```

    inherited Notification(AComponent, Operation);
    if (Operation = opRemove) and (AComponent = FTimer) then
        FTimer := nil;
end;

```

3.7 为接口创建属性

可以把接口用作已发布属性的值，几乎就象你可以使用对象一样。但是，组件从接口的实现接收通告的机制不同。在前面的主题中，属性设定函数调用被作为属性值分派的组件的 `FreeNotification` 方法。这允许当作为属性值的组件被释放时组件更新自身。但是，当属性值是接口时，就不能对实现接口的组件进行访问。结果，不能调用它的 `FreeNotification` 方法。

要处理这个情况，可以调用组件的 `ReferenceInterface` 方法：

```

    procedure TDemoComponent.SetMyIntfProp(const Value: IMyInterface);
    begin
        ReferenceInterface(FIntfField, opRemove);
        FIntfField := Value;
        ReferenceInterface(FIntfField, opInsert);
    end;

```

用指定的接口调用 `ReferenceInterface` 完成的工作与调用另一组件的 `FreeNotification` 方法一样。因此，从属性设定函数中调用 `ReferenceInterface` 以后，可以重载 `Notification` 方法，处理从接口的实现来的通告：

```

    procedure TDemoComponent.Notification(AComponent: TComponent; Operation: TOperation);
    begin
        inherited Notification(AComponent, Operation);
        if (Assigned(MyIntfProp)) and (AComponent.IsImplementorOf(MyIntfProp)) then
            MyIntfProp := nil;
    end;

```

注意，`Notification` 代码给 `MyIntfProp` 属性赋值 `nil`，不给 `private` 域(`TIntfField`)赋 `nil` 值。这确保 `Notification` 调用属性设定函数，当属性值在前面已被设定时，这个设定函数调用 `ReferenceInterface` 来删除已经建立的通告请求。所有对接口属性的赋值必须通过属性设定函数进行。

3.8 存储与装载属性

Delphi 在窗体文件 (在 VCL 应用程序中为 `.dfm`，在 CLX 应用程序中为 `.xfm`) 中保存窗体及其组件。窗体文件保存窗体及其组件的属性。当 Delphi 开发人员给其窗体添加你编写的组件时，你的组件必须具有在保存时把自己的属性写入窗体文件的能力。类似地，当组件被装载进 Delphi 或者作为应用程序的一部分被执行时，组件必须从窗体文件中恢复自己。

大部分时间，使你的组件与窗体一起工作，你不需要做任何事情，因为保存 (对组件的) 描述和从描述中装载的能力是组件所继承行为的一部分。但是，有时你可能想改变组件保存自己的方法，或者想改变在装载时初始化的方法，因此，应该了解底层的机制。

下面是你需要了解的属性存储方面的事情：

- 使用存储与装载机制
- 指定缺省值
- 决定保存什么
- 装载后的初始化
- 存储与装载未发布的属性

3.8.1 使用存储与装载机制

窗体的描述由这个窗体的一系列属性连同窗体上每一个组件的类似描述组成。每一个组件，包括窗体本身，都负责存储与装载自己的描述。

缺省地，当保存自己时，组件按其声明的顺序写入与其缺省值不同的所有已发布属性的值。当装载自己时，组件首先构造自身，设定所有属性为其缺省值，然后读取保存的、非缺省的属性值。

这个缺省机制满足大部分组件的需求，并且不需要组件编写者做任何工作。但是，有几个方法可以用来定制存储与装载过程，以满足对特殊组件的需要。

3.8.2 指定缺省值

Delphi 组件只保存不同于其缺省值的属性的值。如果你不指定其它的，Delphi 就假定属性没有缺省值，这就意味着这个组件永远保存属性，不论其值如何。

要为属性指定缺省值，在属性的声明末尾加 **default** 指示，并给出新的缺省值。

你也可以在重声明属性时指定缺省值。事实上，重声明属性的一个原因就是为分派不同的缺省值。

注释 指定属性的缺省值并不是在对象一旦被创建时就自动给属性分配那个值，你必须确保由组件的构造函数分配需要的值。不是由构造函数设定其值的属性假定它具有 0 值，换句话说，无论什么值的属性这时都假定其内存值被设定为 0。即数字值缺省为 0，布尔值缺省为 **False**，指针缺省为 **nil**，等等。如果有任何疑问，那么在构造函数方法中赋值。

下面的代码显示了一个组件声明，它为 **Align** 属性指定缺省值，并显示了用于设置其缺省值的这个组件构造函数的实现。这里，新组件是特殊情况下的标准面板组件，它将被用于窗口中状态条，所以，它缺省的对齐基准（**alignment**）应该是其拥有者的底部。

```
type
  TStatusBar = class(TPanel)
  public
    constructor Create(AOwner: TComponent); override; // override to set new default
  published
    property Align default alBottom; // redeclare with new default value
  end;
...
constructor TStatusBar.Create(AOwner: TComponent);
begin
  inherited Create(AOwner); // perform inherited initialization
```

```

Align := alBottom;           // assign new default value for Align
end;

```

3.8.3 决定保存的内容

你可以控制 Delphi 是否保存你的组件的每一个属性。缺省地，在类声明的 **published** 部分的所有属性都被保存。你可以选择完全不保存某一给定的属性，或者，你可以分派一个函数，由它来动态确定是否保存属性。

要控制 Delphi 是否保存属性，给属性的声明中添加 **stored** 指示，后跟 **True**、**False** 或者布尔函数的名称。

下面的代码展示了一个组件，它声明了三个新的属性。一个总是被保存，一个从不被保存，第三个是否被保存由布尔函数的值决定。

```

type
  TSampleComponent = class(TComponent)
  protected
    function StoreIt: Boolean;
  public
    ...
  published
    property Important: Integer stored True;    // always stored
    property Unimportant: Integer stored False; // never stored
    property Sometimes: Integer stored StoreIt; // storage depends on function value
  end;

```

3.8.4 装载后的初始化

组件从保存的描述中读取其所有的属性值以后，调用虚拟函数 **Loaded**，它执行任何需要的初始化工作。对 **Loaded** 的调用是在窗体和其控件被显示出来以前发生，所以，不必担心初始化会引起屏幕的闪烁。

要在组件装载其属性值以后初始化组件，重载 **Loaded** 方法。

注释 在任何的 **Loaded** 方法中，第一项要做的事情是调用继承的 **Loaded** 方法。这确保在初始化你自己的组件以前，任何继承的属性都已被正确初始化。

下面的代码来自 **TDatabase** 组件。在装载以后，数据库试图重新建立在其被保存时已打开的任何连接，并且指定如何处理连接过程中发生的任何异常。

```

procedure TDatabase.Loaded;
begin
  inherited Loaded;    // call the inherited method first
  try
    if FStreamedConnected then Open    // reestablish connections
    else CheckSessionName(False);
  except
    if csDesigning in ComponentState then    // at design time ...
      Application.HandleException(Self)    // let Delphi handle the exception

```

```

        else raise;
    end;
end;
// otherwise, reraise

```

3.8.5 存储与装载未发布的属性

缺省地，只有发布的属性随组件一起被装载和保存。但是，可以装载与保存未发布的（unpublished）属性。这允许你具有不在对象查看器中显示的永久属性，也允许组件存储和装载因属性值太复杂而 Delphi 不知道如何读写的属性值。比如，TStrings 对象不能依赖 Delphi 的自动行为来保存与装载它表示的串，并必须使用下列机制。

你可以通过添加代码，告诉 Delphi 如何装载与保存你的属性值来保存未发布的属性。为编写你自己的代码来装载与保存属性，使用以下步骤：

1. 创建方法来存储与装载属性值
2. 重载 DefineProperties 方法，把这些方法传递给文件读写对象（Filer）。

3.8.5.1 创建方法来存储与装载属性值

为存储与装载未发布的属性，首先必须创建一个方法来保存你的属性值，并且创建另一个方法来装载你的属性值。你有二个选择：

- 创建 TWriterProc 类型的方法来存储你的属性值，创建 TReaderProc 类型的方法来装载你的属性值。此方法让你利用 Delphi 内置的保存与装载简单类型的能力。如果你的属性值不是 Delphi 知道如何存储与装载的类型，就使用这个方法。
- 创建 TStreamProc 类型的二个方法，一个用于存储你的属性值，另一个用于装载你的属性值。TStreamProc 采用流式对象(stream)作为变量，你可使用流式对象的方法来读写你的属性值。

例如，考虑在运行时被创建的组件的属性。Delphi 知道如何写这个值，但是并不会自动地这样做，因为这个组件不是在窗体设计器中被创建的。由于流系统（streaming system）可能已经装载与保存了组件，所以你可以采用第一个方法。下面的方法装载与保存被动态创建的组件，它是名为 MyCompProperty 的属性的值：

```

procedure TSampleComponent.LoadCompProperty(Reader: TReader);
begin
    if Reader.ReadBoolean then
        MyCompProperty := Reader.ReadComponent(nil);
end;

procedure TSampleComponent.StoreCompProperty(Writer: TWriter);
begin
    Writer.WriteBoolean(MyCompProperty <> nil);
    if MyCompProperty <> nil then
        Writer.WriteComponent(MyCompProperty);
end;

```

3.8.5.2 重载 DefineProperties 方法

一旦你已经创建了方法来存储与装载你的属性值，就可以重载组件的 DefineProperties 方法。当 Delphi 装载和保存这个组件时，它就调用此方法。在 DefineProperties 方法中，你必须调用当前文件读写对象的 DefineProperty 或者 DefineBinaryProperty 方法，给它传递用于装载或保存你的属性值的方法。如何你的装载与存储方法是 TWriterProc 类型和 TReaderProc 类型，那么就调用文件读写对象的 DefineProperty 方法。如果你创建了 TStreamProc 类型的方法，就调用 DefineBinaryProperty。

不论你使用哪一个方法定义属性，你都要给它传递一个装载与保存你的属性值的方法，和一个指示此属性值是否需要被写的布尔值。如果此值可以被继承或者有缺省值，你都不必写它了。

例如，给出的是 TReaderProc 类型的 LoadCompProperty 方法和 TWriterProc 类型的 StoreCompProperty 方法，那么你将重载 DefineProperties 方法如下：

```
procedure TSampleComponent.DefineProperties(Filer: TFiler);
function DoWrite: Boolean;
begin
    if Filer.Ancestor <> nil then // check Ancestor for an inherited value
    begin
        if TSampleComponent(Filer.Ancestor).MyCompProperty = nil then
            Result := MyCompProperty <> nil
        else if MyCompProperty = nil or
            TSampleComponent(Filer.Ancestor).MyCompProperty.Name <>
                MyCompProperty.Name then
            Result := True
        else Result := False;
    end
    else // no inherited value -- check for default (nil) value
        Result := MyCompProperty <> nil;
    end;
begin
    inherited; // allow base classes to define properties
    Filer.DefineProperty('MyCompProperty', LoadCompProperty, StoreCompProperty,
        DoWrite);
end;
```

第 4 章 创建事件

事件就是在系统中所发生事情 (如用户动作、焦点改变)与响应这些事情的一段代码间的链接。响应代码是事件处理程序，并且几乎总是由应用程序开发人员编写的。事件让应用程序开发人员不必改变类本身而定制组件的行为。这就是委托 (delegation)。

为最普通的用户动作(如鼠标动作)的事件都已被内建在全部的标准组件中，但是，你也可以定义新的事件。要在组件中创建事件，需要了解下列事项：

- 什么是事件?
- 实现标准事件
- 定义自己的事件

事件被实现为属性，所以在你希望创建事件或者改变组件的事件以前，应该熟悉第 3 章“创建属性”的内容。

4.1 什么是事件?

事件是把所发生事情链接到一些代码的机制，更确切地说，事件就是方法指针，它指向特定类实例中的方法。

从应用程序开发人员的观点来说，事件只是关联到系统所发生事情的一个名称，比如 `OnClick`，对此事情可附上特定的代码。例如，名为 `Button1` 的按钮有一个 `OnClick` 方法。缺省地，当你给 `OnClick` 事件赋值时，窗体设计器就在包含此按钮的窗体中产生一个名为 `Button1Click` 的事件处理程序，并把它指派给 `OnClick`。当在此按钮上发生鼠标点击事件时，此按钮就调用分派给 `OnClick` 的方法，这时就是 `Button1Click`。

要写事件，需要理解下列事项：



- 事件是方法指针
- 事件是属性
- 事件类型是方法指针类型
- 事件处理程序类型是过程
- 事件处理程序是可选的

4.1.1 事件是方法指针

Delphi 使用方法指针来实现事件。方法指针是一个特殊的指针类型，它指向特定类实例中的特定方法。作为组件编写人员，你可以把方法指针当作占位符，当你的代码侦测到事件发生时，就可以调用由用户为此事件指定的方法(如果有的话)。

方法指针就象任何其它的过程类型一样地工作，但是维护一个指向类实例的隐藏的指针。当应用程序开发人员给组件的事件分派处理程序时，不只是分派给有特定名称的方法，而且分派给特定类实例的方法。此实例通常是但不必是包含这个组件的窗体。

比如，所有控件都继承了一个用于处理点击事件的名为 `Click` 的动态方法：

```
procedure Click; dynamic;
```

`Click` 的实现调用用户的点击事件处理程序（如果存在的话）。如果用户已给控件的 `OnClick` 事件分派了处理程序，那么，点击控件就会导致此方法被调用。如果没有分派处理程序，那么什么也不会发生。

4.1.2 事件是属性

组件使用属性来实现它们的事件。不象大部分其它属性一样，事件不使用方法来实现其读写部分，而是事件属性使用与属性相同类型的私有类域。

按约定，域的名字是属性的名字前加字母 F。例如，OnClick 方法的指针被保存在 TNotifyEvent 类型的 FOnClick 域中，并且 OnClick 事件属性的声明看起来象这样：

```
type
    TControl = class(TComponent)
    private
        FOnClick: TNotifyEvent;           // declare a field to hold the method pointer
    ...
    protected
        property OnClick: TNotifyEvent read FOnClick write FOnClick;
    end;
```

要学习 TNotifyEvent 和其它事件类型，参见下一节的“事件类型是方法指针类型”。

就象对任何其它属性一样，你可以在运行时设置或者改变事件的值。但是，使事件成为属性的主要优点是组件用户可以在设计时通过使用对象查看器来给事件分派处理程序。

4.1.3 事件类型是方法指针类型

由于事件是指向事件处理程序的指针，所以事件属性的类型必须是方法指针类型。同样，被用作事件处理程序的任何代码必须是被正确输入的类的方法。

所有事件处理程序方法都是过程。为了与给定类型的事件兼容，事件处理程序方法必须以相同的方式，按相同的顺序，传递相同数量与类型的参数。

Delphi 为其所有标准事件定义了方法类型。当你创建自己的事件时，如果已存在类型合适的话，可以使用已存在的类型，或者定义你自己的类型。

4.1.3.1 事件处理程序类型是过程

尽管编译器允许你声明是函数的方法指针类型，但是，你绝不可以也这样处理事件。因为空的函数返回未定义的结果，是函数的空的事件处理程序并不总是有效的，因此，你的所有事件和其相应的事件处理程序都应该是过程。

虽然事件处理程序不能是函数，但是仍然可以通过使用 var 参数从应用程序开发人员的代码中获取信息。当这样做的时候，在调用处理程序以前，要确保给参数分派了有效值，因此，就不需要用户代码来改变此值。

给事件处理程序传递 var 参数的一个例子是 TKeyPressEvent 类型的 OnKeyPress 事件。TKeyPressEvent 定义了二个参数，一个指示哪一个对象产生了这个事件，另一个指示哪个键被按下：

```
type
    TKeyPressEvent = procedure(Sender: TObject; var Key: Char) of object;
```

通常，Key 参数包含用户按下的字符。但是，在某特定情况下，组件的用户或许想改变这个字符。一个例子或许是在编辑器中使所有字符变成大写。这时，用户为键击可以定义如

下的处理程序：

```
procedure TForm1.Edit1KeyPressed(Sender: TObject; var Key: Char);
begin
    Key := UpCase(Key);
end;
```

你也可以使用 `var` 参数来让用户重载缺省的处理。

4.1.4 事件处理程序是可选的

当创建事件时，要记住，使用你组件的开发人员或许没有给它们附加处理程序。这就是说，你的组件不应该因为没有给某一特定事件附加处理程序就简单地失败或者产生错误。（有关调用处理程序和处理没有附加处理程序的事件的机制在 [4-9 页](#) 的“调用事件”有解释）

事件几乎经常性地 GUI 应用程序中发生。只是鼠标指针移过某可视组件就会发送大量鼠标移动消息，这时这个组件把此消息转换为 `OnMouseMove` 事件。大多数情况下，开发人员并不想处理鼠标移动事件，这不应该引起错误。因此，你创建的组件应该不需要有对这些事件的处理程序。

此外，应用程序开发人员可以在事件处理程序中写入他们想写的任何代码。类库中的组件使事件以减少事件处理程序产生错误的机会这样一种方式被编写。显然，你不能避免应用程序代码中的逻辑错误，但是，你可以保证在调用事件以前数据结构被初始化，这样，应用程序开发人员就不会试图访问无效的数据。

4.2 实现标准事件

与组件库一起提供的控件为普遍发生的事情继承事件，这被称为标准事件。尽管所有这些事件被内建在控件中，但它们经常是 `protected`，这意味着是开发人员不能给它们附加处理程序。当你创建控件时，可以选择使事件对于使用你控件的用户可见。

当把标准事件结合进你的控件中时，有三个事项需要考虑：

- 识别标准事件
- 使事件可见
- 改变标准事件的处理

4.2.1 识别标准事件

有二类标准事件：为所有控件定义的标准事件、只为标准窗口控件定义的标准事件。

4.2.1.1 为所有控件的标准事件

最基本的事件被定义在类 `TControl` 中。不论是窗口控件、图形控件，还是定制控件，所有的控件都继承这些事件。以下事件在所有的控件中都可用：

<code>OnClick</code>	<code>OnDragDrop</code>	<code>OnEndDrag</code>	<code>OnMouseMove</code>
<code>OnDblClick</code>	<code>OnDragOver</code>	<code>OnMouseDown</code>	<code>OnMouseUp</code>

标准事件有被声明在 TControl 中的 protected virtual 方法，它们有与事件名称关联的名称。例如，OnClick 事件调用名为 Click 的方法，OnEndDrag 事件调用名为 DoEndDrag 的方法。

4.2.1.2 为标准控件的标准事件

除对所有控件公有的事件以外，标准窗口控件(在 VCL 应用程序中是从 TWinControl 继承的控件，在 CLX 应用程序中是从 TWidgetControl 继承的控件)有下列事件：

OnEnter	OnKeyDown	OnKeyPress
OnKeyUp	OnExit	

就象在 TControl 中的标准事件一样，窗口控件事件有相应的方法。以上列出的这些标准的键盘事件响应所有普通的键击。

注释 但是，为响应特殊键击（如 Alt 键），你必须响应来自 Windows 的 WM_GETDLGCODE 或者 CM_WANTSPECIALKEYS 消息。对有关编写消息处理程序的内容，参见第 7 章“处理消息和系统通告”。

4.2.2 使事件可见

在 TControl 和 TWinControl（在 CLX 应用程序中的 TWidgetControl）中的标准事件的声明是 protected，响应这些事件的方法也一样是 protected。如果你正从这些抽象类中的一个类继承，并想让它们的事件在运行时或者在设计时可用，那么就需要重声明这些事件为 public 或 published。

重声明属性而不指定其实现，则保持与其原实现方法相同的名称，但是改变了其保护级别。因此，你可以取一个在 TControl 中定义的但不可见的事件，通过声明它为 public 或 published 而使其展示出来。

例如，要创建一个在设计时展示 OnClick 事件的组件，应给组件的类声明添加下列代码：

```
type
    TMyControl = class(TCustomControl)
    ...
    published
        property OnClick;
    end;
```

4.2.3 改变标准事件的处理

如果你想改变你的组件响应某种事件的途径，你可能想尝试写一些代码，并把它分派给此事件。作为一个应用程序开发人员，此确实是你应该做的，但是，当创建组件时，你必须保持对于使用此组件的开发人员来说这个事件是可用的。

这就是 protected 实现方法关联每一个标准事件的原因。通过重载实现方法，你可以修改内部事件的处理，并通过调用继承的方法，你可以继续标准的处理，包括为应用程序开发人员的代码的事件（including the event for the application developer's code）。

调用方法的顺序是非常重要的。通常，首先调用继承的方法，此允许应用程序开发人员

的事件处理程序在你的定制以前执行（并有时，阻止定制的执行）。但是，有些时候，你或许还想在调用继承的方法以前就执行你的代码。比如，如果继承的代码有些依赖于组件的状态，而你的代码改变此状态，那么你就应该进行这个改变，并且允许用户的代码响应这个改变。

假如你正在编写一个组件，并想更改它响应鼠标点击的方法。作为应用程序开发人员，你本应该给 `OnClick` 事件分派处理程序，然而你并没有这样做，而是重载了 `protected` 方法 `Click`：

```
procedure Click override      // forward declaration
...
procedure TMyControl.click;
begin
    inherited Click;          // perform standard handling, including calling handler
    ...                       // you customizations go here
end;
```

4.3 定义自己的事件

定义全新的事件相对来说不经常发生，但是，还是有这样一些时候，当一个组件引入完全不同于任何其它组件的行为时，就需要为其定义一个事件。

定义事件时需要考虑的问题：

- 触发事件
- 定义处理程序类型
- 声明事件
- 调用事件

4.3.1 触发事件

你需要知道什么触发了事件。对于一些事件，答案是明显的。例如，当用户按压鼠标左键时发生鼠标按下事件，并且 Windows 给应用程序发送 `WM_LBUTTONDOWN` 消息。一旦收到此消息，组件就调用它的 `MouseDown` 方法，这个方法依次调用用户已附加给 `OnMouseDown` 事件的任何代码。

但是，有一些事件不是很明显地能联系到外部发生的特定事情上。比如，滚动条有 `OnChange` 事件，此事件由几种事项触发，包括键盘键击、鼠标点击、以及在其它控件中发生的变化。当定义你的事件时，你必须保证所有发生的有关事项调用正确的事件。

注释 对 CLX 应用程序，参见 [7-10 页](#) 的“使用 CLX 响应系统通告”。

4.3.1.1 二种事件

有二种你需要提供事件的事项：用户交互、状态改变。用户交互事件几乎总是由来自 Windows 的消息触发，这个消息指示用户做了某事，而你的组件可能需要对其作出响应。状态改变事件也可能涉及来自 Windows 的消息（比如，输入焦点改变和使能），但是，它们也

可能由于属性的改变或者其它代码而发生。

你对于自己定义事件的触发有完全的控制权。要仔细地定义事件，这样开发人员才能够理解并使用你定义的事件。

4.3.2 定义处理程序类型

一旦决定了事件在什么时候发生，那么就必须定义你想怎样处理此事件，即意味着确定事件处理程序的类型。大多数时候，你自己为事件定义的处理程序要么是简单的通告，要么是事件特定类型。也有可能从处理程序回馈信息。

4.3.2.1 简单通告

通告事件是这样一种事件，它只告诉你特定事件发生了，并没有关于此事件何时发生或者在哪里发生的特定信息。通告使用 `TNotifyEvent` 类型，它只带有一个参数，即事件的发送者。通告的处理程序关于事件全部‘知道’的只是它是何种类型，事件发生在哪个组件上。例如，点击事件就是通告。当你为点击事件编写处理程序时，你全部知道的就是点击发生了，哪个组件被点击了。

通告是一个单向过程，没有提供回调或者防止对通告作进一步处理的机制。

4.3.2.2 事件特定的处理程序

有时，仅知道哪个事件发生了以及在哪个组件中发生事件还是不够的。比如，事件是键盘按压事件，很可能处理程序想知道用户按下了哪个键。这时，你就需要处理程序类型，它包含有提供附加信息的参数。

如果你创建事件来响应消息，那么传递给事件处理程序的参数就可能直接来自这个消息参数。

4.3.2.3 从处理程序返回信息

由于所有事件处理程序都是过程，所以，从处理程序传递回信息的唯一途径只有通过 `var` 参数。你的组件可以使用这样的信息来决定在用户处理程序运行以后如何处理或者是否处理事件。

比如，所有的键盘事件（`OnKeyDown`、`OnKeyUp`、`OnKeyPress`）用名为 `Key` 的参数以引用方式传递按下的键值。事件处理程序可以改变 `Key`，这样应用程序就看到一个涉及此事件的不同的键。例如，强制键入的字符变为大写。

4.3.3 声明事件

一旦确定了事件处理程序的类型，就要准备为事件声明方法指针和属性。确保给事件起一个有意义的、描述性的名字，这样，用户就可推定事件是干什么的。要设法与其它组件中的类似属性取一致的名字。

4.3.3.1 事件名称以 On 开头

Delphi 中大部分事件的名称以“On”开头。这只是一个约定，编译器并不强制这样。对象查看器通过查看属性的类型决定属性是一个事件：假设所有的方法指针属性为事件，并且在‘事件’页中出现。

开发人员希望在以字母顺序排列的具有以“On”开头的名字的列表中查找事件。

注释 这个规则的主要例外是，发生在一些事情以前和以后的很多事件以 Before 和 After 开头。

4.3.4 调用事件

应该集中对事件的调用权限，即，在组件中创建调用应用程序中事件处理程序的虚拟方法(如果分派的话)，并且提供任何缺省的处理。

把所有的事件调用放在一起，保证当某人从你的组件继承新组件的时候，可以通过重载单个方法来定制事件的处理，而不用经由代码搜索以查找调用事件的地方。

当调用事件时，还有二个其它方面的考虑：

- 空的处理程序必须有效
- 用户可以重载缺省处理

4.3.4.1 空的处理程序必须有效

你决不应该造成空的处理程序引起错误这样一种情形，也不应该使组件的基本功能依赖于来自应用程序中事件处理程序的特定响应。

空的处理程序应该产生与没有处理程序完全一样的结果。因此，调用应用程序事件处理程序的代码应该看起来象这样：

```
if Assigned(OnClick) then OnClick(Self);  
...           // perform default handling
```

你绝不应该做象下面这样的事情：

```
if Assigned(OnClick) then OnClick(Self)  
else ...           // perform default handling
```

4.3.4.2 用户可以重载缺省处理

对有些种类的事件，开发人员可能想取代缺省的处理，甚至想禁止所有的响应。为能做到这样，你应该给处理程序传递一个引用参数，并在处理程序返回时检查某特定值。

这个和‘空的处理程序应该产生与没有处理程序完全一样的结果’的规则保持一致。因为空的处理程序不会改变由引用传递的参数值，所以，缺省的处理过程总是在调用空的处理程序以后发生。

例如，当处理键盘按压事件时，应用程序开发人员可通过设置 var 参数 Key 为 null 值(#0)来禁止组件击键的缺省处理。对此观点的支持逻辑如下：

```
if Assigned(OnKeyPress) then OnKeyPress(Self, Key);  
if Key <> #0 then ...           // perform default handling
```

由于处理 Windows 消息，实际的代码与此会有一点不同，但是，逻辑是一样的。缺省地，组件调用用户分派的任何处理程序，然后执行其标准处理。如果用户处理程序设定 Key 为 null 字符，那么组件就跳过其缺省处理。

第 5 章 创建方法

组件方法是内置于类结构中的过程和函数。尽管对组件的方法能做什么没有做根本性的限制，但是，Delphi 确实使用了一些你应该遵守的标准。这些原则包括：

- 消除依赖
- 命名方法
- 保护方法
- 使方法为虚拟的
- 声明方法

通常，组件不应该包括很多方法，而且应该使应用程序需要调用的方法的数量最少。可能倾向于以方法实现的特征，最好经常被封装为属性。属性提供适合 Delphi 并且在设计时可用的接口。

5.1 消除依赖

一直以来，在编写组件时，都要求影响开发人员的前提条件减至最小。在最大可能的范围内，开发人员应该能够在任何时候做他们想对组件做的任何事情。然而，也会存在某些情形，虽然你不可能满足它，但是应该尽量向你的目标靠拢。

这个列表给出要避免的依赖的种类型的观点：

- 要使用组件，用户必须调用的方法
- 必须按一定顺序执行的方法
- 使组件处于某状态或模式的方法，在某些状态或模式下，某些事件或者方法可能会无效

处理这些情况的最好的途径是保证提供避开这些依赖的方法。比如，如果调用一个方法会使你的组件处于调用另一个方法时可能会无效的状态，那么编写第二个方法，这样，当组件处在一个不合适状态时，如果应用程序调用它，那么这个方法就在执行其主要代码以前纠正此状态。至少，当用户调用无效的方法时，你应该引出一个异常。

换句话说，如果你造成一种情形，在这里你的代码部分相互依赖，那么你应该负责确保以不正确的方式使用代码不致于引起问题。例如，如果用户没有适应你的依赖，那么引起系统错误的告警信息会是一个更好的选择。

5.2 命名方法

Delphi 对于如何命名方法或者方法的参数并没有任何限制，但是，对开发人员有使创建方法更容易的少量约定。记住，组件结构的性质决定了有很多不同的用户会使用你的组件。

如果你习惯于编写只有你自己或者小范围的程序员使用的代码，你可以不必认真考虑命

名的事。让方法的名字意思明确是一个不错的想法，因为不熟悉你的代码（甚至不熟悉编写代码）的用户可能必须使用你的代码。

以下是使方法名容易明白的一些建议：

- 起描述性的名称。使用有意义的动词

使用象 `PasteFromClipboard` 的名称比简单的 `Paste` 或 `PFC` 更有信息量

- 函数名称应该反映返回值的性质

尽管作为程序员把返回某物横坐标位置的函数命名为 `X` 是很显而易见的，但是象 `GetHorizontalPosition` 这样的名称更易被普遍理解。

最后一项考虑，确定此方法确实需要作为一个方法被定义。一个好的原则是方法名称中有动词。如果发现你创建的很多方法名称中没有动词，就考虑此方法是否应该是属性。

5.3 保护方法

类的所有部分（包括域、方法、属性）都有一个保护级别或者‘可见性’，如在 [2-4 页](#) “控件访问”中的说明。为方法选择合适的可见性是简单的。

在组件中编写的大部分方法是 `public` 或 `protected`。很少需要编写 `private` 方法，除非它确实是这种组件专有的，甚至派生的组件都不应访问它。

5.3.1 应公开的方法

应用程序开发人员需要调用的任何方法都必须声明为 `public`。记住，大部分方法调用发生在事件处理程序中，所以，方法应该避免绑定系统资源，或者使操作系统处于不能响应用户的状态。

注释 构造函数和析构函数应该总是 `public`。

5.3.2 应受保护的方法

对于组件的任何实现方法都应该为 `protected`，所以，应用程序在错误的时间就不能调用它们。如果你有应用程序代码不应该调用而在派生的类中被调用的方法，那么就把它声明为 `protected`。

比如，假定你有一个方法，它依赖于预先为其建立的某些数据，如果你让此方法为 `public`，那么在构造此数据以前，应用程序就有调用它的机会。另外，使此方法为 `protected`，就保证了应用程序不能直接调用它。然后，你可以创建其它的 `public` 方法，保证在调用 `protected` 方法以前先构造此数据。

属性实现的方法应声明为 `virtual protected` 方法。方法被如此声明，是让应用程序开发人员重载属性实现，或者增加其功能，或者完全取代它。这样的属性完全是多态的。保持访问方法为 `protected`，确保开发人员不会意外地调用方法，或者不注意地修改了属性。

5.3.3 抽象方法

有时，方法在 Delphi 组件中被声明为 `abstract`。在组件库中，抽象方法通常在名称以

‘custom’开头的类中出现，比如 TCustomGrid。这样的类本身就是抽象类，从这个意义上说，它们被打算只是用于派生子孙类。

虽然你可以创建包含抽象成员的类的实例对象，但是并不建议这样做。调用抽象成员会导致 EAbstractError 异常。

abstract 指示被用于说明应该被展示出来并且在子孙组件中被定义的类的部分。它迫使组件编写者在类的实例创建以前重声明子孙类中的抽象成员。

5.4 使方法为虚拟的

当你想让能执行不同代码的不同类型响应相同的方法调用时，你就使方法为 virtual。

如果创建希望由应用程序开发人员直接使用的组件，你或许可以使所有的方法为非虚拟的。另外，如果你创建从它派生其它组件的抽象组件，那么考虑使添加的方法为 virtual。这样，派生的组件可以重载继承的 virtual 方法。

5.5 声明方法

在组件中声明方法与声明任何类方法一样。

要在组件中声明新方法，做下列事情：

- 给组件的对象类型声明添加声明
- 在组件单元的 implementation 部分实现方法

下面的代码展示一个组件，它定义了二个新方法，一个为 protected static 方法，另一个是 public virtual 方法：

```
type
    TSampleComponent = class(TControl)
    protected
        procedure MakeBigger;      // declare protected static method
    public
        function CalculateArea: Integer; virtual;    // declare public virtual method
    end;
...
implementation
...
procedure TSampleComponent.MakeBigger;    // implement first method
begin
    Height := Height + 5;
    Width := Width + 5;
end;

function TSampleComponent.CalculateArea: Integer; // implement second method
begin
    Result := Width * Height;
```


end;

第 6 章 在组件中使用图形

Windows 为绘制设备独立的图形提供了强大的图形设备接口(GDI)。但是，GDI 对程序员提出了额外的要求，如管理图形资源。Delphi 承担了所有繁重的 GDI 工作，容许你集中精力在更有价值的工作上，而不必搜寻丢失的句柄或者未释放的资源。

对于 Windows API 的任何部分，可在你的 Delphi 应用程序中直接调用 GDI 函数。但是，你可能会发现，使用 Delphi 对图形函数的封装会更快、更容易。

注释 GDI 函数是 Windows 专有的，并且不能应用于 CLX 应用程序。但是，CLX 组件使用 Qt 库。

本节的主题包括：

- 图形概述
- 使用画布
- 用图片工作
- 离屏位图
- 响应改变

6.1 图形概述

Windows 在几个级别上封装 GDI（在 CLX 应用程序中为 Qt）。作为一个组件编写者，最重要的是组件在屏幕上显示其图象的方法。当直接调用 GDI 函数时，需要有一个指向设备上下文的句柄，在此句柄中，选择了各种绘图工具，如画笔、画刷、字体。绘制了图形图象以后，你必须恢复设备上下文到其使用以前的初始状态。

Delphi 并不强制你详细地处理图形，而是提供了简单然而完备的接口：组件的画布（Canvas）属性。画布确保它具有一个有效的设备上下文，并在不被使用时释放此上下文。同样地，画布有其自己的代表当前画笔、画刷、字体的属性。

画布为你管理所有这些资源，所以，你自己不必关心如何创建、选择并释放象画笔句柄这类事情。你只要告诉画布应该使用什么种类的画笔，然后就什么也不用管了。

让 Delphi 管理图形资源的一个好处是它能为以后使用而缓冲资源，此会加快重复性的操作。比如，你有一个程序，它重复地创建、使用、释放某种画笔工具，那么你就需要每次重复这些步骤。因为 Delphi 缓存了图形资源，就有你要使用的工具还在缓存中这样的好机会，这样，Delphi 就不用重新创建工具，而是使用已存在的工具。

使用此机制的一个例子是一个应用程序有几十个打开了的窗体，有几百个控件。每一个这些控件可能有一个或多个 TFont 属性。尽管这可能导致出现几百或者几千个 TFont 对象实例，但是大部分应用程序最终只使用二、三个字体句柄，此归功于字体缓存。

这里是 Delphi 的图形代码可以是多么简单的二个示例。第一个使用标准的 GDI 函数，在窗口上绘制有色蓝轮廓的黄色底椭圆，此方法你可以使用其它的开发工具。第二个在用 Delphi 写的一个应用程序中使用画布绘制同样的椭圆。

```
procedure TMyWindow.Paint(PaintDC: HDC; var PaintInfo: TPaintStruct);  
var
```

```

    PenHandle, OldPenHandle: HPEN;
    BrushHandle, OldBrushHandle: HBRUSH;
begin
    PenHandle := CreatePen(PS_SOLID, 1, RGB(0,0,255)); // create blue pen
    OldPenHandle := SelectObject(PaintDC, PenHandle); // tell DC to use blue pen
    BrushHandle := CreateSolidBrush(RGB(255,255,0)); // create a yellow brush
    OldBrushHandle := SelectObject(PaintDC, BrushHandle); // tell DC to use yellow brush
    Ellipse(HDC, 10, 10, 50, 50); // draw the ellipse
    SelectObject(OldBrushHandle); // restore original brush
    DeleteObject(BrushHandle); // delete yellow brush
    SelectObject(OldPenHandle); // restore original pen
    DeleteObject(PenHandle); // destroy blue pen
end;

procedure TForm1.FormPaint(Sender: TObject);
begin
    with Canvas do
    begin
        Pen.Color := clBlue; // make the pen blue
        Brush.Color := clYellow; // make the brush yellow
        Ellipse(10,10,50,50); // draw the ellipse
    end;
end;

```

6.2 使用画布

画布类在几个级别上封装了图形控件，包括绘制单线条、形状、文本的高级函数；操纵画布绘图能力的中级属性；以及在组件库中提供对于 Windows GDI 的低级访问。

表 6.1 总结了画布的能力。

6.1 画布能力总结

级别	操作	工具
高	绘制线条和形状	如 MoveTo、LineTo、Rectangle、Ellipse 方法
	显示和测量文本	TextOut、TextHeight、TextWidth、TextRect 方法
	填充区域	FillRect、FloodFill 方法
中	定制文本和图形	Pen、Brush、Font 属性
	操纵像素	Pixels 属性
	复制与合并图像	Draw、StretchDraw、BrushCopy、CopyRect 方法；CopyMode 属性
低	调用 Windows GDI 函数	Handle 属性

对于有关画布类和其方法与属性的更详细的内容，参见在线帮助。

6.3 用图片工作

在 Delphi 中你做的大部分图形工作限于直接在组件和窗体的画布上绘图。Delphi 也提供处理独立的图形图象，如位图、元文件、图标，包括调色板的自动管理。

在 Delphi 中用图形工作有三个重要方面：

- 使用图片、图形、画布
- 装载与保存图形
- 处理调色板

6.3.1 使用图片、图形或画布

在 Delphi 中处理图形有三种类：

- **canvas**（画布）代表在窗体、图形控件、打印机、或位图上的位图绘制表面。**canvas** 总是其它项中的一个属性，绝不会是一个独立的类。
- **graphic**（图形）代表通常在文件或资源中发现的那种图形图象，比如位图、图标、元文件。Delphi 定义了类 **TBitmap**、**TIcon**、**TMetafile**（只用于 VCL），它们都是通用类 **TGraphic** 的子孙类。你也可以定义自己的图形类。通过为所有的图形定义最小的标准接口，**TGraphic** 为应用程序容易地使用不同种类的图形而提供了一种简单的机制。
- **picture**（图片）是图形的容器，意思就是它可以包含任何图形类。即，**TPicture** 类型的一个项可包含一个位图、图标、元文件或者一个用户定义的图形类型，并且，应用程序借助此图形类可以用相同的方式访问它们。比如，**image** 控件有一个 **TPicture** 类型的 **Picture** 属性，能控制很多种图形的图象显示。

记住，**picture** 类总有一个 **graphic**，**graphic** 可能有一个 **canvas**（唯一有 **canvas** 的标准图形是 **TBitmap**）。通常，当处理图片时，你只用通过 **TPicture** 展现的 **graphic** 类的部分工作。如果你需要访问图形类本身的细节，可参看图片的 **Graphic** 属性。

6.3.2 装载与保存图形

在 Delphi 中所有图片和图形可从文件中装载其图象并再保存回去（或者保存到不同的文件中），你可以在任何时候装载或者保存图片的图象。

注释 你也可以从 Qt MIME 源装载图象，并保存它们到 Qt MIME 源中，或者，如果创建跨平台组件的话，同样操作可施于流对象。

要从文件装载图象到图片，就调用图片的 **LoadFromFile** 方法。要从图片保存图象到文件，调用图片的 **SaveToFile** 方法。

LoadFromFile 和 **SaveToFile** 方法，每一个都把文件名作为其唯一参数。**LoadFromFile** 利用文件的扩展名确定要创建和装载的图形对象的类型。可保存任何类型文件的 **SaveToFile** 也适用于保存图形对象类型的文件。

例如，装载一个位图到 **image** 控件的图片中，传递位图文件的文件名给此图片的 **LoadFromFile** 方法：

```
procedure TForm1.LoadBitmapClick(Sender: TObject);
begin
```

```
Image1.Picture.LoadFromFile('RANDOM.BMP');  
end;
```

图片把.bmp 认为是位图文件的标准扩展名，所以它作为 TBitmap 创建其图片，然后调用此图形的 LoadFormFile 方法。因为图形是位图，所以它从文件中作为位图装载图象。

6.3.3 处理调色板

对 VCL 和 CLX 组件，当在基于调色板的设备（典型的是 256 色视频模式）上运行时，Delphi 控件自动支持调色板的实现。即，如果你有一个有调色板的控件，你可使用从 TControl 继承的二个方法来控制如何使 Windows 适应此调色板。

调色板对控件的支持有二个方面：

- 为控件指定调色板
- 响应调色板的改变

大部分控件不需要调色板，但是包含‘富色彩’图形图象的控件（如 image 控件）可能需要与 Windows 和屏幕设备驱动程序交互来保证控件恰当的外观表示。Windows 把这个过程称为‘实现’调色板。

实现调色板是这样一个过程，它保证最上面的窗口使用其全部调色板，底部的窗口使用尽量多的调色板，然后映射任何其它色彩到最接近真实的颜色。当窗口一个在一个前面相互移动时，Windows 连续实现调色板。

注释 Delphi 本身对于创建和维护调色板没有提供特别的支持，除在位图中以外。但是，如果你有一个调色板句柄，那么 Delphi 控件可为你管理它。

6.3.3.1 为控件指定调色板

要给控件指定调色板，重载此控件的 GetPalette 方法以返回此调色板的句柄。

为给控件指定调色板，你的应用程序应做下列事情：

- 告诉应用程序，你的控件的调色板需要被实现
- 指定用于实现的调色板

6.3.3.2 响应调色板的改变

如果你的 VCL 和 CLX 控件通过重载 GetPalette 指定了一个调色板，Delphi 会自动地关照对来自 Windows 的调色板消息的响应。处理调色板消息的方法是 PaletteChanged。

PaletteChanged 的主要任务是确定是在前台还是在后台实现控件的调色板。Windows 通过使最上面的窗口有前台调色板，其它窗口使用后台调色板来进行调色板的实现。Delphi 更进一步，因为它在窗口中按 Tab 顺序也实现控件的调色板。只有一个时候，你或许才需要重载这个缺省的特性，即，如果你想让一个控件有前台调色板，但它不是在第一 Tab 顺序。

6.4 离屏位图

当绘制复杂的图形图象时，在图形编程中一个通用的技巧是创建离屏位图，在此位图上

绘制图象，然后从此位图把完整的图象复制到最后的目标屏幕上。使用离屏图象减少因直接在屏幕上重复绘制所引起的闪烁。

在 Delphi 中的位图类，它代表资源和文件中位图图象，也可以作为离屏图象工作。

用离屏位图工作主要有二个方面：

- 创建和管理离屏位图
- 复制位图图象

6.4.1 创建和管理离屏位图

当创建复杂的图形图象时，应该避免直接在屏幕上显示的画布上绘制它们。不在窗体或者控件的画布上绘制，取而代之的是，可构建一个位图对象，在其画布上绘制，然后复制其完整的图象到在屏的画布上。

离屏位图被最常使用于图形控件的 Paint 方法中。就象使用任何临时对象一样，位图都应该使用 try...finally 块被保护起来。

```
type
    TFancyControl = class(TGraphicControl)
    protected
        procedure Paint; override;    // override the Paint method
    end;

procedure TFancyControl.Paint
var
    Bitmap: TBitmap;                // temporary variable for the off-screen bitmap
Begin
    Bitmap := TBitmap.Create;        // construct the bitmap object
    try
        { draw on the bitmap }
        { copy the result into the control's canvas }
    finally
        Bitmap.Free;                // destroy the bitmap object
    end;
end;
```

6.4.2 复制位图图象

Delphi 提供了四种不同的途径来从一个画布到另一个画布复制图像。根据你想产生的效果的不同，可调用不同的方法：

表 6.2 总结了在画布对象上的图象复制方法。

表 6.2 图象复制方法

要产生的效果	调用方法
复制完整图形	Draw
复制并改变图形大小	StretchDraw
复制画布的一部分	CopyRect

用光栅操作复制位图	BrushCopy (VCL)
重复复制图形使其平铺满一个区域	TiledDraw (CLX)

6.5 响应改变

所有的图形对象，包括画布和画布拥有的对象（画笔、画刷、字体），都有内置于其中的事件来响应在对象中的改变。通过使用这些事件，可以使你的组件（或使用这些组件的应用程序）重绘其图象来响应改变。

如果你的组件作为设计时接口的一部分被发布，那么响应图形对象中的改变尤其重要。保证组件在设计时的外观显示与在对象查看器中的属性集匹配的唯一途径，就是响应在对象中的改变。

要响应在图形对象中的改变，给类的 **OnChange** 事件指派方法。

形状（**shape**）组件发布用于绘制其形状的代表其画笔和画刷的属性。此组件的构造函数给每一个画笔或画刷的 **OnChange** 事件指派一个方法，从而引起此组件在画笔或画刷改变时刷新其图象：

```

type
  TShape = class(TGraphicControl)
  public
    procedure StyleChanged(Sender: TObject);
  end;
...
implementation
...
constructor TShape.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);           // always call the inherited constructor
  Width := 65;
  Height := 65;
  FPen := TPen.Create;                 // construct the pen
  FPen.OnChange := StyleChanged;       // assign method to OnChange event
  FBrush := TBrush.Create;             // construct the brush
  FBrush.OnChange := StyleChanged;     // assign method to OnChange event
end;

procedure TShape.StyleChanged(Sender: TObject);
begin
  Invalidate();                       // erase and repaint the component
end;
```

第 7 章 处理消息和系统通告

组件经常需要响应来自于底层操作系统的通告。操作系统通知应用程序发生的事情，比如用户使用鼠标和键盘做了什么。有些控件也产生通告，比如，象用户在列表框中选择列表项这样的由用户动作所产生的结果。组件库已经处理了通用通告的大部分，但是，可能也需要编写你自己的处理这些通告的代码。

对 VCL 应用程序，通告以消息的形式送达。这些消息可以来自于任何源，包括 Windows、VCL 组件、你自己定义的组件。用消息工作有三个方面：

- 理解消息处理系统
- 改变消息处理
- 创建新的消息处理程序

对 CLX 应用程序，通告以信号和系统事件的形式送达，它们取代 Windows 消息。有关在 CLX 中如何用系统通告工作的详细内容，参见 7-10 页的“使用 CLX 响应系统通告”。

7.1 理解消息处理系统

所有 VCL 类都有一个处理消息的内建机制，被称为消息处理方法，或者消息处理程序。消息处理程序的基本思想是，类接收某种消息，并分发消息，然后根据接收到的消息调用一套方法中的一个方法。如果对于某特定的消息不存在指定的方法，则使用缺省的处理程序。

下图显示消息分派系统：



可视组件库（VCL）定义了一个消息分派系统，它把指向特定类的全部 Windows 消息（包括用户定义的消息）翻译为方法调用。你从不需要改变此消息分派机制。所有你需要做的只是创建消息处理方法。对此主题的更多的内容，参见 7-7 页的“声明新的消息处理方法”。

7.1.1 Windows 消息中是什么？

Windows 消息是一个数据记录，它包含几个域。其中最重要的是一个用于识别消息的整数大小的值。Windows 定义了很多消息，并在 Messages 单元中为所有消息声明了标识符。消息中其它的有用信息是二个参数域和一个结果域。

一个参数含有 16 位，另一个参数含有 32 位。你经常会看到把这二个值当作 wParam 和 lParam 的 Windows 代码，它们分别代表字参数和长整型参数。经常，每个参数包含多个信息，你看到对名称的引用，比如 lParamHi，它代表在长整型参数中的高位字。

刚开始，Windows 程序员不得不记住或者在 Windows API 中查找每一个参数所包含的信息。现在，微软给每一个参数命名。这个所谓的消息分离方法（message cracking）使得理解伴随每一个消息的信息是什么变得更简单了。例如，对 WM_KEYDOWN 消息的参数，现在被命名为 nVirtKey 和 lKeyData，它比 wParam 和 lParam 给出更多的细节信息。

为每一种类型的消息，Delphi 都定义了一个记录类型，它为每一个参数都提供一个助记名。例如，鼠标消息在长整型参数中传递鼠标事件的 x 和 y 坐标，并且一个放在高位字中，

另一个在低位字中。使用鼠标消息结构，你不必担心哪个字是哪个，因为你用名字 XPos 和 YPos 取代了 lParamLo 和 lParamHi 来代表参数。

7.1.2 分派消息

当应用程序创建窗口时，它向 Windows 核心注册一个窗口过程。此窗口过程是为窗口处理消息的子程序。传统的窗口过程包含一个巨大的 case 语句，它有这个窗口必须处理的每一个消息的入口。记住，在这个意义上的‘窗口’意味着屏幕上的几乎任何东西：每个窗口、每个控件，等等。每一次创建一种新类型的窗口，你都必须创建一个完整的窗口过程。

VCL 用几个途径简化了消息分派：

- 每个组件都继承一个完整的消息分派系统
- 分派系统有缺省的处理。你只为需要特别响应的消息定义处理程序
- 你可以修改消息处理的一小部分，对大部分处理，依赖继承的方法

这个消息分派系统的最大好处是你可以在任何时候向任何组件安全地发送任何消息。如果组件没有对消息定义的处理程序，那么就使用缺省的处理，通常忽略此消息。

7.1.2.1 跟踪消息流程

VCL 在应用程序中为每一种组件注册一个被称为 MainWndProc 的方法作为其窗口过程。MainWndProc 包含一个异常处理块，从 Windows 传递消息结构给虚拟方法 WndProc，并且通过调用应用程序类的 HandleException 处理任何异常。

MainWndProc 是一个非虚拟方法，它不包含对于任何特定消息的特殊处理。在 WndProc 中发生定制，因为每一个组件类型都可重载这个方法，以满足其特殊的需要。

WndProc 方法检查影响其处理的任何特殊条件，这样它们可‘捕捉’到不想要的消息。比如，在被拖拽的过程中，组件忽略键盘事件，所以，TWinControl 的 WndProc 方法只有在组件不被拖拽时才传递键盘事件。最终，WndProc 调用从 TObject 继承的非虚拟方法 Dispatch，它决定调用哪个方法来处理此消息。

Dispatch 使用消息结构的 Msg 域来决定如何分派特定的消息。如果组件为特定的消息定义了处理程序，Dispatch 就调用这个方法。如果组件没有为此消息定义处理程序，Dispatch 就调用 DefaultHandler。

7.2 改变消息处理

在改变你的组件的消息处理以前，确定你确实想做什么。VCL 翻译大部分 Windows 消息为组件编写者和组件用户都能处理的事件。相对于改变消息处理的行为，你或许更应该改变事件处理的行为。

要改变 VCL 组件中的消息处理，重载消息处理方法。你也可以通过捕获消息来阻止组件处理在某特定情况下的消息。

7.2.1 重载处理程序方法

要改变组件处理特定消息的途径，重载对此消息的消息处理方法。如果组件还没有处理这个特定的消息，你需要声明一个新的消息处理方法。

要重载消息处理方法，用与被重载的方法相同的消息索引在你的组件中声明一个新方法。不使用 `override` 指示，你必须使用 `message` 指示和一个匹配的消息索引。

注意，方法的名称和单个 `var` 参数的类型不必匹配被重载的方法。只有消息索引是重要的。不过，为清楚起见，在它们处理消息的后面，最好还是遵循消息处理方法的命名约定。

例如，要重载组件对 `WM_PAINT` 消息的处理，重新声明 `WMPaint` 方法：

```
type
  TMyComponent = class(...)
  ...
  procedure WMPaint(var Message: TWMPaint); message WM_PAINT;
end;
```

7.2.2 使用消息参数

一旦在消息处理方法里面，你的组件就有权访问消息结构的所有参数。因为传递给消息处理程序的参数是 `var` 参数，所以，如果需要的话，处理程序可以改变参数的值。唯一频繁改变的参数是消息的 `Result` 域：由发送消息的 `SendMessage` 调用返回的值。

由于消息处理方法中的 `Message` 参数的类型随被处理的消息而改变，所以，你应该参考有关 Windows 消息的文档，查阅各参数的名称和含义。如果因为某些原因，你需要查阅有旧式名称的消息参数（`Wparam`、`LParam`，等），你可转换 `Message` 为一般类型 `TMessage`，它们使用那些旧的参数名称。

7.2.3 捕获消息

在某些情况下，你可能想让你的组件忽略消息。即，你想阻止组件给其处理程序分派消息。要捕获消息，重载虚拟方法 `WndProc`。

对 VCL 组件来说，`WndProc` 方法在把消息传递给 `Dispatch` 方法以前隐藏消息，它依次决定哪一个方法着手处理此消息。通过重载 `WndProc`，你的组件在分派消息以前取得滤除消息的机会。对从 `TWinControl` 派生的控件，一个重载其 `WndProc` 的过程样子如下：

```
procedure TMyControl.WndProc(var Message: TMessage);
begin
  {tests to determine whether to continue processing}
  inherited WndProc(Message);
end;
```

`TControl` 组件定义了全部的鼠标消息，这样当用户拖放控件时就过滤消息。重载 `WndProc` 以二个方式对此提供帮助：

- 可过滤大范围的消息，从而不必为每一个消息指定处理程序
- 可根除消息分派，所以处理程序从不会被调用

例如，这里是 `TControl` 的 `WndProc` 方法部分：

```

procedure TControl.WndProc(var Message: TMessage);
begin
    ...
    if (Message.Msg >= WM_MOUSEFIRST) and (Message.Msg <= WM_MOUSELAST) then
        if Dragging then           // handle dragging specially
            DrawMouseMsg(TWMMouse(Message))
        else
            ...                     // handle other normally
        end;
    ...                             // otherwise process normally
end;

```

7.3 创建新的消息处理程序

由于 VCL 为大部分普通消息提供了处理程序，所以当定义自己的消息时，就很可能需要创建新的消息处理程序。用用户定义的消息工作有三个方面：

- 定义自己的消息
- 声明新的消息处理程序方法
- 发送消息

7.3.1 定义自己的消息

很多标准组件为内部使用定义了消息。定义消息最普通的理由是广播的信息没有包括在标准消息和因状态改变而产生的通告中。你可以在 VCL 中定义自己的消息。

定义消息分二个步骤，这些步骤是：

1. 声明消息标识符
2. 声明消息记录类型

7.3.1.1 声明消息标识符

消息标识符是一个整数大小的常数，Windows 保留 1024 以下的值为其自己的消息使用，所以当你声明自己的消息时，应该取大于 1024 的值。

常数 WM_APP 代表用户定义消息的起始值。当定义消息标识符时，应该基于 WM_APP。

大家知道，一些标准的 Windows 控件使用用户定义范围的消息，这些控件包括列表框、组合框、编辑框和命令按钮。如果从这些控件中的某组件派生一个组件，并想为其定义一个新消息，那么一定要检查 Messages 单元，看一下 Windows 已经为此控件定义了哪些消息。

下列代码显示了二个用户定义的消息：

```

const
    MY_MYFIRSTMESSAGE = WM_APP + 400;
    MY_MYSECONDMESSAGE = WM_APP + 401;

```

7.3.1.2 声明消息记录类型

如果你想给你的消息参数起一个有用的名字，就需要为此消息声明一个消息记录类型。消息记录是要传递给消息处理方法的参数的类型。如果你不想使用消息的参数，或者你想使用老式的参数符号 (wParam、lParam 等)，就可以使用缺省的消息记录 TMessage。

要声明消息记录类型，遵循下列约定：

1. 在消息后面命名记录类型，加前缀 T
2. 调用 TMsgParam 类型的记录 Msg 中的第一个域
3. 定义下面的二个字节以符合 Word 参数，后面二个字节为 unused。或者定义下面的四个字节以符合 Longint 参数
4. 添加最后一个域 Result，为 Longint 类型

例如，这里是对所有鼠标消息的消息记录 TWMMouse，它使用一个变体记录对相同的参数定义二组名字。

```
type
    TWMMouse = record
        Msg: TMsgParam;           // first is the message ID
        Keys: Word;               // this is the wParam
        case Integer of           // two ways to look at the lParam
            0: (
                XPos: Integer;     // either as x- and y-coordinates...
                YPos: Integer;)
            1: (
                Pos: TPoint;       // ... or as a single point
                Result: Longint;)  // and finally, the result field
        end;
```

7.3.2 声明新的消息处理方法

有二类情况需要声明新的消息处理方法：

1. 你的组件需要处理还没有被标准组件处理过的 Windows 消息
2. 你已为自己的组件定义了自己的消息

要声明消息处理方法，做下列事情：

1. 在组件类声明的 protected 部分声明方法
2. 使方法为过程
3. 在它处理消息的后面命名方法，但是不能带下划线。
4. 传递消息记录类型的单个的 var 参数 Message
5. 在消息方法的实现里，为组件的任何特殊处理写代码
6. 调用继承的消息处理程序

例如，下面是用户定义消息 CM_CHANGEColor 的消息处理程序的声明：

```
const
    CM_CHANGEColor = WM_APP + 400;
type
    TMyComponent = class(TControl)
```

```

...
protected
    procedure CMChangeColor(var Message: TMessage); message CM_CHANGECOLOR;
end;

procedure TMyComponent.CMChangeColor(var Message: TMessage);
begin
    Color := Message.LParam;
    inherited;
end;

```

7.3.3 发送消息

典型地，应用程序发送消息用于发送状态改变的通告，或者用于广播信息。你的组件可以给窗体里的所有控件广播消息，给特定控件（或应用程序本身）发送消息，甚至给你的组件本身发送消息。

发送 Windows 消息有几个不同的途径，你所采用的方法依赖于你发送消息的原因。以下的主题描述了发送 Windows 消息的不同的途径：

7.3.3.1 给窗体中的所有控件广播消息

当你的组件要改变影响窗体或者其它容器里的所有控件的全局设定时，你或许想向这些控件发送消息，所以，它们会适当地更新自身。并不是每一个控件都需要响应这个通告，但是，通过广播这个消息，你可以通知那些知道如何响应的所有控件，也允许其它控件忽略此消息。

要广播消息给另一个控件中的其它所有控件，使用 **Broadcast** 方法。在你广播消息以前，用要传送的信息填写消息记录。（对有关消息记录的信息，参见 [7-6 页上的“声明消息记录类型”](#)）

```

var
    Msg: TMessage;
begin
    Msg.Msg := MY_MYCUSTOMMESSAGE;
    Msg.WParam := 0;
    Msg.LParam := Longint(Self);
    Msg.Result := 0;

```

然后，传送这个消息记录到你通知的所有控件的父对象，这可以是应用程序中的任何控件。例如，可以是你正在编写的控件的父对象：

```
Parent.Broadcast(Msg);
```

可以是包含你的控件的窗体：

```
GetParentForm(self).Broadcast(Msg);
```

可以是活动的窗体：

```
Screen.ActiveForm.Broadcast(Msg);
```

甚至可以是你的应用程序中的所有窗体：

```
for i := 0 to Screen.FormCount - 1 do  
    Screen.Forms[i].Broadcast(Msg);
```

7.3.3.2 直接调用控件的消息处理程序

有时，只有一个控件需要响应你的消息。如果你知道应该接收你消息的这个控件，那么，发送消息的最简单和最直接的途径就是调用此控件的 **Perform** 方法。

为什么要调用控件的 **Perform** 方法，主要有二个原因：

- 你想触发与控件产生给标准的 **Windows**（或其它）消息相同的响应。例如，当网格控件接收到一个键击消息时，它就创建一个内嵌编辑控件，然后给此编辑控件转送键击消息。
- 你可能知道你想通知的控件，但是不知道它是什么类型的控件。因为不知道目标控件的类型，所以就不可能知道它的任何特定的方法，但是所有控件有消息处理的能力，所以你总可以发送消息。如果控件有你要发送消息的消息处理程序，那么它会适当地响应此消息。否则，它将忽略你发送的这个消息，并返回 0。

要调用 **Perform** 方法，不需要创建消息记录。你只需要传递消息标识符和参数 **WParam**、**LParam**，**Perform** 返回消息结果。

7.3.3.3 使用 **Windows** 消息队列发送消息

在多线程应用程序中，你不能只是调用 **Perform** 方法，因为目标控件是在与正运行线程不同的另一个线程里。但是，通过使用 **Windows** 消息队列，你可以安全地与其它线程通信。消息处理总是发生在主 **VCL** 线程中，但是，你可以使用 **Windows** 消息队列，从应用程序的任何线程中发送消息。对 **SendMessage** 的调用是同步的，即，直到目标控件已经处理了消息，甚至控件是在另一个线程中，**SendMessage** 才会返回。

使用 **Windows** API 调用 **SendMessage**，借助 **Windows** 消息队列给控件发送消息。除了你必须通过传递其窗口句柄来指定目标控件以外，**SendMessage** 采用与 **Perform** 方法相同的参数。这样，不要写下列代码：

```
MsgResult := TargetControl.Perform(MY_MYMESSAGE, 0, 0);
```

而应该这样写：

```
MsgResult := SendMessage(TargetControl.Handle, MYMESSAGE, 0, 0);
```

对于 **SendMessage** 函数的更多的信息，参见微软 **MSDN** 文档。有关编写可同时运行的多线程程序的更多的内容，参见《开发者指南》中 13-11 页的“定位线程”。

7.3.3.4 发送不立即执行的消息

在有些时候，你可能想发送消息，但是并不知道它立即执行后对于消息的目标是否安全。例如，如果发送消息的代码是从目标控件上的事件处理程序中被调用的，那么，在此控件执行你的消息以前，你可能想确定事件处理程序已经运行完成了。只要你不需要知道消息的结果，你就可以处理这种情况。

使用 **Windows** API 调用 **PostMessage** 给控件发送消息，但是允许此控件等待，直到在处理你的消息以前它已经完成了任何其它的消息。**PostMessage** 采用与 **SendMessage** 完全相同

的参数。

有关 `PostMessage` 函数的更多的信息，参见微软 MSDN 文档。

7.4 使用 CLX 响应系统通告

当使用 Windows 时，操作系统使用 Windows 消息直接向你的应用程序和其包含的控件发送通告。但是，这个方法不适合 CLX 应用程序，因为 CLX 是一个跨平台的库，而 Windows 消息在 Linux 上不能使用。代之，CLX 采用平台中立的方法来响应系统通告。

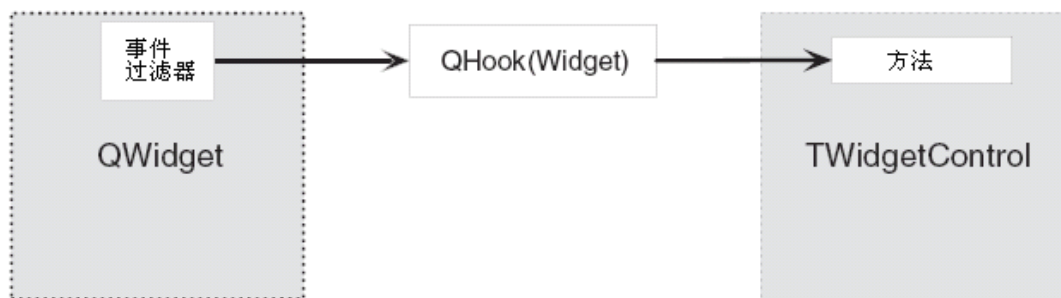
对 CLX 来说，与 Windows 消息相类似的是来自底层部件层的信号的信号系统。然而，在 VCL 中，Windows 消息可以来自操作系统，也可以来自被 VCL 包裹的固有 Windows 控件（native Windows controls），CLX 使用的部件层在这两个之间产生不同。如果通告来自于部件，就被称为信号。如果通告来自于操作系统，则被称为系统事件。部件层作为事件类型的信号沟通系统事件和你的 CLX 组件。

7.4.1 响应信号

底层的部件层发出各种信号，每一个信号代表一个不同类型的通告。这些信号包括系统事件（事件信号）以及各部件产生的特定的通告。例如，所有部件在释放时都产生被销毁信号，`trackbar` 部件产生 `valueChanged` 信号，`header` 控件产生 `sectionClicked` 信号，等等。

每一个 CLX 组件通过为信号分派方法作为其处理程序来响应从底层部件来的信号。这样做是通过使用一个特殊的钩子对象，它与底层的部件关联。钩子对象是一个轻量级对象，它确实只是方法指针的一个集合，每一个方法指针对应一个特定的信号。当 CLX 组件的方法已经作为特定信号的处理程序被分派给钩子对象时，那么每次部件一产生特定的信号，有关此 CLX 组件的方法就获得调用。具体描述见如图 7.1:

Figure 7.1 信号路由



注释 每一个钩子对象的方法被声明在 Qt 单元中。这些方法被修正为全局子程序，而这些子程序有反映其所属于的钩子对象的名称。例如，与应用程序部件（`QApplication`）关联的有关钩子对象的所有方法以 `'QApplication_hook'` 开头。这个修正是需要的，所以，Delphi CLX 对象可以访问 C++ 钩子对象的方法。

7.4.1.1 分派定制的信号处理程序

很多 CLX 控件已经给来自底层部件的信号分派了方法用于处理此信号。典型地，这些方法是私有的、而不是虚拟的。这样，如果你想编写自己的方法来响应信号，那么就必须指派自己的方法给与你的部件关联的勾子对象。要这样做，重载 HookEvents 方法。

注释 如果你想响应的信号是系统事件通告，就不必使用 HookEvents 方法的重载。有关如何响应系统事件的更详细的知识，参见后面的“响应系统事件”。

在对 HookEvents 方法的重载中，声明 TMethod 类型的一个变量。然后，为你想作为信号处理程序把其指派给勾子对象的每一个方法，做以下事情：

1. 初始化 TMethod 类型的变量，为信号提供一个方法处理程序。
2. 把此变量指派给勾子对象。你可以通过使用你的组件从 THandleComponent 或者 TWidgetControl 继承的 Hooks 属性来访问此勾子对象。

在你的重载中，总是调用继承的 HookEvents 方法，这样，基类分派的信号处理程序也被勾住。

下列代码是 TTrackBar 的 HookEvents 方法。它描述了如何重载 HookEvents 方法来添加定制的信号处理程序。

```
procedure TTrackBar.HookEvents;
var
    Method: TMethod;
begin
    // initialize method to represent a handler for the QSlider valueChanged signal
    // valueChangedHook is a method of TTrackBar that responds to the signal
    QSlider_valueChanged_Event(Method) := ValueChangedHook;
    // Assign method to the hook object.note that you can cast Hooks to the
    // type of hook object associated with the underlying widget
    QSlider_hook_hook_valueChanged(QSlider_hookH(Hooks), Method);
    // repeat the process for the sliderMoved event
    QSlider_sliderMoved_Event(Method) := ValueChangedHook;
    QSlider_hook_hook_valueChanged(QSlider_hookH(Hooks), Method);
    // call the inherited method so that inherited signal handlers are hooked up
    inherited HookEvents;
end;
```

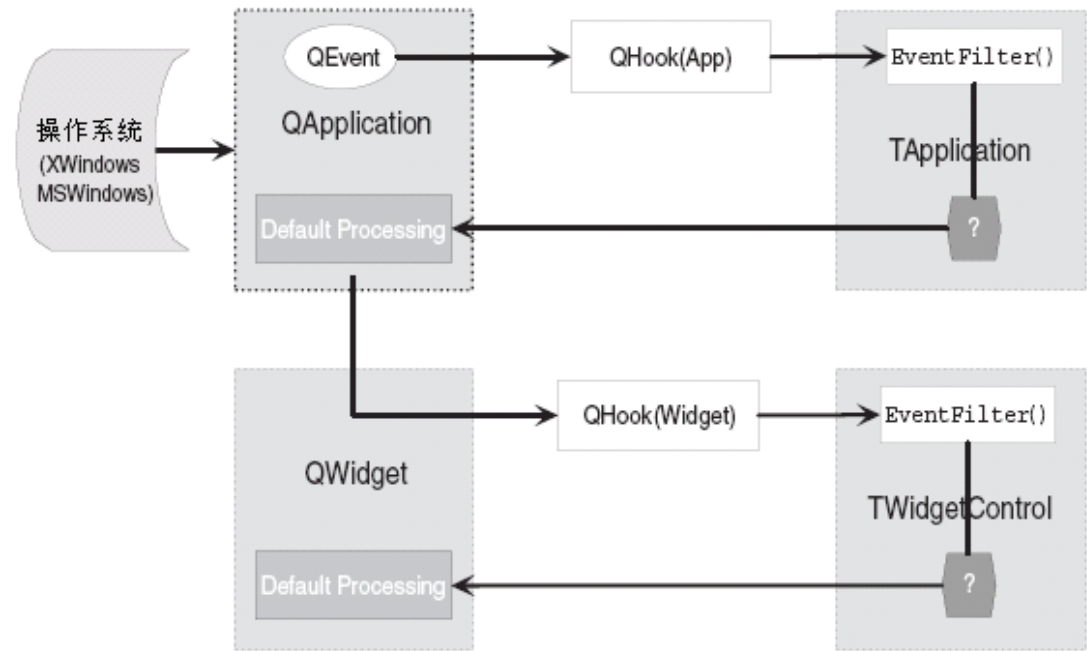
7.4.2 响应系统事件

当部件层从操作系统接收到事件通告时，它就产生一个特定事件对象（QEvent 或者一个其它子孙）代表此事件。这个事件对象包括有关发生事件的只读信息，事件对象的类型指示所发生事件的类型。

部件层通过使用一个特殊信号的类型事件来给你的 CLX 组件通知系统事件。它把 QEvent 对象传递给事件的信息处理程序。因为它最先到达应用程序对象，所以事件信号的处理过程比处理其它信号有一点复杂。这就是说，应用程序有二个机会来响应系统事件：一个在应用程序级 (TApplication)，另一个在各组件级 (你的 TWidgetControl 或 THandleComponent 子孙)。所有这些类 (TApplication、TWidgetControl、THandleComponent)

已经从部件层为事件信号分派了信号处理程序。即，所有系统事件都被自动引导到 EventFilter 方法，它与 VCL 控件上的 WndProc 方法有类似的作用。系统事件的处理过程解释如图 7.2:

Figure 7.2 系统事件路由



EventFilter 处理大部分被普遍使用的系统通告，把它们转换为由你的组件的基类引入的事件。这样，比如，TWidgetControl 的 EventFilter 方法通过产生 OnMouseDown、OnMouseMove、OnMouseUp 事件来响应鼠标事件(QMouseEvent)，通过产生 OnKeyDown、OnKeyPress、OnKeyString、OnKeyUp 事件来响应键盘事件(QKeyEvent)，等。

7.4.2.1 常用事件

TWidgetControl 的 EventFilter 方法通过访问由 TControl 或 TWidgetControl 引入的 protected 方法处理很多的公共系统通告。大部分这些方法是 virtual 或 dynamic，所以，当你编写自己的组件并实现对系统事件的你自己的响应时，你就可以重载它们。当重载这些方法时，你不必担心用事件对象或（大多数情况下）在底层部件层的任何其它对象工作的问题。

当你想让你的 CLX 组件响应系统通告时，首先检查是否有已经响应了这个通告的 protected 方法是一个不错的想法。你可以检查 TControl 或 TWidgetControl（以及你的组件派生自的任何其它基类）的文档来看是否有响应你感兴趣事件的 protected 方法。表 7.1 列出了很多被最普遍使用的 protected 方法，它们来自你可以使用的 TControl 和 TWidgetControl。

表 7.1 响应系统通告的 TWidgetControl 的 protected 方法

方法	说明
BeginAutoDrag	如果控件的 DragMode 属性有 dmAutomatic 值，当用户点击鼠标左键时被调用
Click	当用户在控件上释放鼠标键时被调用
DblClick	当用户在控件上双击鼠标时被调用
DoMouseWheel	当用户滚动鼠标轮时被调用

DragOver	当用户在控件上拖动鼠标光标时被调用
KeyDown	当控件有输入焦点并用户按下键盘键时被调用
KeyPress	如果 KeyDown 没有处理键击，则在 KeyDown 之后被调用
KeyString	当系统使用多字节字符系统并在用户用键盘输入字符时被调用
KeyUp	控件具有焦点并当用户释放键盘键时被调用
MouseDown	当用户在控件上点击鼠标键时被调用
MouseMove	当用户在控件上移动鼠标光标时被调用
MouseUp	当用户在控件上释放鼠标键时被调用
PaintRequest	当系统需要重绘控件时被调用
WidgetDestroyed	当在控件底层的部件销毁时被调用

在重载中，调用继承的方法，所以任何缺省的处理仍然继续进行，即，响应信号。

注释 除响应系统事件的方法以外，控件包括大量相似的方法，这些方法由 TControl 或 TWidgetControl 发起，给控件通知各种事件。尽管这些方法不响应系统事件，但是它们执行与很多被发送到 VCL 控件的 Windows 消息相同的任务。表 7.2 列出其中一些方法。

表 7.2 来自于控件的响应事件的 TWidgetControl 的 protected 方法

方法	说明
BoundsChanged	当控件改变大小时被调用
ColorChanged	当控件色彩变化时被调用
CursorChanged	当光标改变形状时被调用。当处在窗口部件上时鼠标光标显示此形状
EnabledChanged	当应用程序改变窗口或控件的使能状态时被调用
FontChanged	当字体资源的集合改变时被调用
PaletteChanged	当部件的调色板改变时被调用
ShowHintChanged	当有关控件的帮助提示条显示或隐藏时被调用
StyleChanged	当窗口或控件的 GUI 样式改变时被调用
TabStopChanged	当窗体上的 tab 顺序改变时被调用
TextChanged	当控件的文本改变时被调用
VisibleChanged	当控件被隐藏或显示时被调用

7.4.2.2 重载 EventFilter 方法

如果你想响应事件通告，但是对你可以重载的此事件没有任何的 protected 方法，那么，你可以重载 EventFilter 方法本身。在你的重载中，要检查 EventFilter 方法中 Event 参数的类型，并且当它代表你想响应的通告的类型时，执行你自己的特殊处理。你可通过使你的 EventFilter 方法返回 True 值来阻止对此事件通告作进一步的处理。

注释 关于不同类型的 QEvent 对象的详细内容，参见 TrollTech 中的 Qt 文档。

下面的代码是有关 TCustomControl 的 EventFilter 方法，它说明了当重载 EventFilter 时如何从 QEvent 对象获得事件类型。注意，尽管这里没有显示，但是一旦你确定了事件类型，你就可以把 QEvent 对象转换为合适的、特殊的 QEvent 子孙（如 QMouseEvent）。

```
function TCustomControl.EventFilter(Sender: QObjectH; Event: QEventH): Boolean;
begin
    Result := inherited EventFilter(Sender, Event);
    case QEvent_type(Event) of
```

```

        QEventType_Resize,
        QEventType_FocusIn,
        QEventType_FocusOut:
            UpdateMask;
    end;
end;

```

7.4.2.3 产生 Qt 事件

与 VCL 控件可以定义并发送定制的 Windows 消息的方法类似，你可以使你的 CLX 控件定义并产生 Qt 系统事件。第一步要为事件定义一个唯一的 ID（与定义定制的 Windows 消息时你必须定义消息 ID 的方法类似）。

```

const
    MyEvent_ID = Integer(QCLXEventType_ClxUser) + 50;

```

在你想产生事件的代码中，用新的事件 ID，使用 QCustomEvent_create 函数（被声明在 Qt 单元中）产生一个事件对象。可选的第二个参数让你用数据值补充事件对象，此数据值是指向你想与此事件关联的信息的一个指针。

```

var
    MyEvent: QCustomEventH;
begin
    MyEvent := QCustomEvent_create(MyEvent_ID, self);

```

一旦创建了这个事件对象，就可以通过调用 QApplication_postEvent 方法来发布它。

```

    QApplication_postEvent(Application.Handle, MyEvent);

```

对于响应此通告的任何组件，只需要重载其 EventFilter 方法，检查 MyEvent_ID 事件类型。EventFilter 方法通过调用声明在 Qt 单元中的 QCustomEvent_data 方法可找回你提供给构造函数的数据。

第 8 章 制作设计时可用的组件

本章描述使创建的组件在 IDE 中可用的步骤。制作在设计时可用的组件需要几个步骤：

- 注册组件
- 给组件提供帮助
- 添加属性编辑器
- 添加组件编辑器
- 把组件编译进包

并不是所有这些步骤都将应用于每一个组件。例如，如果你没有定义任何新的属性或者事件，你就不需要为其提供帮助。总是需要的唯一步骤是注册和编译。

一旦你的组件已被注册并编译进包中，它们就可以被分发给其他开发人员使用，并被安装在 IDE 里。有关在 IDE 中安装包的内容，参见《开发者指南》中的第 16-10 页“安装组件包”。

8.1 注册组件

注册在编译单元的基础上进行工作，所以，如果你在单独一个编译单元中创建了几个组件，你就可以一次性完成几个组件的全部注册。

要注册组件，给单元添加 **Register** 过程。在 **Register** 过程中注册组件，并决定把它们安装在组件面板的哪个地方。

注释 如果你在 IDE 中通过选择‘组件 | 新组件’来创建你的组件，注册组件所需要的代码就会被自动地添加进去。

手动注册组件的步骤是：

- 声明 **Register** 过程
- 编写 **Register** 过程

8.1.1 声明 **Register** 过程

注册涉及到在单元中编写一个过程，此过程必须有 **Register** 名称。**Register** 过程必须出现在此单元的 **interface** 部分，其名称与在 Delphi 其它部分不同，它是大小写敏感的。

注释 尽管 Delphi 是一个大小写不敏感的语言，但是 **Register** 过程是大小写敏感的，并且必须以大写字母 **R** 开头。

下列代码显示了创建和注册新组件的一个简单单元的框架：

```
unit MyBtns;
interface
type
    ...                // declare your component types here
    procedure Register; // this must appear in the interface section
implementation
    ...                // component implementation goes here
    Procedure Register;
    begin
        ...            // register the components
    end;
end.
```

在 **Register** 过程内部，为你想安装到组件面板里的每一个组件调用 **RegisterComponents**。如果单元包含几个组件，你可以在一个步骤中完成对它们的全部注册。

8.1.2 编写 **Register** 过程

在包含组件的单元中的 **Register** 过程内部，你必须注册你想添加到组件面板上的每一个组件。如果此单元包含几个组件，你可以同时注册它们。

要注册组件，为你想添加组件到组件面板中的每一页调用一次 **RegisterComponents** 过程。**RegisterComponents** 涉及到三个重要事情：

1. 指定组件
2. 指定组件面板页

3. 使用 RegisterComponents 函数

8.1.2.1 指定组件

在 Register 过程中，用开放数组传递组件名称，在调用 RegisterComponents 内部你可以构造这个数组。

```
RegisterComponents('Miscellaneous', [TMyComponent]);
```

你也可一次在同一页上注册几个组件，或者把组件注册在不同的页上，如下列代码所示：

```
procedure Register;
```

```
begin
```

```
    RegisterComponents('Miscellaneous', [TFirst, TSecond]); // two on this page ...
```

```
    RegisterComponents('Assorted', [TThird]);                // ... one on another ...
```

```
    RegisterComponents(LoadStr(srStandard), [TFourth]);      // ... and one on the Standard
```

```
page
```

```
end;
```

8.1.2.2 指定组件面板页

组件面板页的名称是一个字符串，如果你给出的面板页的名称还不存在，Delphi 就用此名称创建一个新页。Delphi 把标准页的名称保存在 string-list 资源中，所以产品的国际版可以用使用者计算机本机语言（native language）命名此页。如果你想把组件安装到一个标准页上，应该通过调用 LoadStr 函数并给其传递代表此页串资源的常数获得此页的名称，如 srSystem 代表 System 页。

8.1.2.3 使用 RegisterComponents 函数

在 Register 过程中，调用 RegisterComponents 来注册用类数组表示的组件。RegisterComponents 是一个函数，它有二个参数：组件面板页的名称、组件类的数组。

设置 Page 参数为要显示组件的组件面板页的名称。如果命名的页已存在，则组件被添加到此页上。如果命名的页不存在，Delphi 用此名称创建一个新的面板页。

在定义定制组件的一个单元中，从 Register 过程的实现里调用 RegisterComponents。然后，定义组件的单元必须被编译进一个包中，并且此包必须在定制组件被添加到组件面板以前被安装。

```
procedure Register;
```

```
begin
```

```
    RegisterComponents('System', [TSystem1, TSystem2]); //add to system page
```

```
    RegisterComponents('MyCustomPage', [TCustom1, TCustom2]); //new page
```

```
end;
```

8.2 为组件提供帮助

当你在窗体上选择一个标准组件，或者在对象查看器中选择一个属性或事件时，你可以按 F1 键来得到有关此项的帮助。如果你创建了格式合理的帮助文件，就可以给开发人员为你的组件也提供相同类型的文档。

你可以提供一个小的帮助文件来描述你的组件，你的帮助文件变成了 Delphi 用户的全部帮助系统的一部分。

有关如何为组件的使用写作帮助文件，参见 8-4 页的“创建帮助文件”。

8.2.1 创建帮助文件

你可以使用任何你想使用的工具来为 Windows 帮助文件创建源文件(.rtf 格式)。Delphi 包含 Microsoft Help Workshop，它编译帮助文件，并提供一个在线帮助作者指南。你可以在 Help Workshop 的在线指南中找到有关创建帮助文件的完整信息。

为组件编写帮助文件，包括下列步骤：

- 创建条目
- 使组件的帮助上下文关联
- 添加组件帮助文件

8.2.1.1 创建条目

要使组件的帮助与库中其它组件的帮助无缝地结合在一起，遵循以下约定：

1. 每一个组件都应该有一个帮助主题

组件主题应该显示组件被声明在哪一个单元中，并且简要描述此组件。组件主题应该链接到在对象层次关系中描述组件位置的次级窗口，并且列举出其全部属性、事件和方法。应用程序开发人员通过在窗体上选择组件并按 F1 键可访问这个主题。举一个组件主题的例子，在窗体上放任一个组件，按 F1 键。

组件主题必须有一个#脚注，它有此主题独特的唯一值。借助帮助系统，#脚注唯一地识别每一个主题。

组件主题应该有一个 K 脚注，可用于在包括组件类的名称的帮助系统索引中用关键字搜索。例如，对 TMemo 组件的关键字脚注是"TMemo"。

组件主题也应该有一个\$脚注，它提供主题的标题，这个标题显示在‘主题查找’对话框、‘书签’对话框、‘历史’窗口中。

2. 每一个组件都应该包括下面的次级导航主题

- 层次主题，它具有到组件层次中的每一个组件祖先的链接
- 在组件中可用的所有属性的列表，它具有到描述这些属性的条目的链接
- 在组件中可用的所有事件的列表，它具有到描述这些事件的条目的链接
- 在组件中可用的方法的列表，它具有到描述这些方法的条目的链接

在 Delphi 帮助系统中，到对象类、属性、方法或者事件的链接，可使用 Alink 制作。当链接到对象类时，Alink 使用此对象的类的名称，接着是下划线和字符串"object"。比如，到 TCustomPanel 对象的链接，使用下面的代码：

```
!AL(TCustomPanel_object, 1)
```

当链接到属性、方法或者事件时，在此属性、方法或者事件的名称前面加上实现它们的对象的名称和下划线。比如，到由 TControl 实现的 Text 属性的链接，使用下面的代码：

```
!AL(TControl_Text, 1)
```

要看一个次级导航主题的示例，则显示出任一组件的帮助，点击标记的层次、属性、方法或者事件的链接即可。

3. 在组件中被声明的每一个属性、方法和事件都应该有一个主题

属性、事件或者方法的主题应该显示此属性、事件或者方法的声明，并描述其用法。应用程序开发人员查看这些主题，可通过在对象查看器中加亮此项并按 F1 键查看，或者在代码编辑器中把光标置于此项的名称上并按 F1 来查看。看一个属性主题的示例，在对象查看器上选择任一项并按 F1 即可。

属性、事件和方法的主题应该包括一个 K 脚注，它列出属性、方法或者事件的名称和其与组件名结合的名称。这样，TControl 的 Text 属性具有下列 K 脚注：

```
Text,TControl;TControl,Text;Text,
```

属性、方法、事件的主题也应该包括一个 \$ 脚注，它指示此主题的标题，如 TControl.Text。

所有这些主题都应该有一个主题 ID，它对于此主题是唯一的，并作为 # 主题被输入。

8.1.2.2 使组件的帮助上下文关联

每一个组件、属性、方法和事件的主题都必须有一个 A 脚注。A 脚注被用于在用户选择了组件并按下 F1 键时或者事件在对象查看器中被选择并按下 F1 键时显示主题。A 脚注必须遵循一定的命名约定。

如果帮助的主题是为组件的，则 A 脚注由两个被分号隔开的条目组成，使用如下句法：

```
ComponentClass_Object;ComponentClass
```

其中，ComponentClass 是组件类的名称。

如果帮助的主题是为属性或事件的，A 脚注由三个由分号隔开的条目组成，使用如下句法：

```
ComponentClass_Element;Element_Type;Element
```

其中，ComponentClass 是组件类的名称，Element 是属性、方法或者事件的名称，Type 是 Property、Method 或 Event。

例如，对 TMyGrid 组件的 BackgroundColor 属性，A 脚注如下：

```
TMyGrid_BackgroundColor;BackgroundColor_Property;BackgroundColor
```

8.2.2 添加组件的帮助文件

要把你的帮助文件添加到 Delphi 中，使用位于 bin 目录下的 OpenHelp 程序（程序名为 oh.exe），或者使用 IDE 中的‘帮助 | 定制’菜单访问。

你会在 OpenHelp.hlp 文件中发现有关使用 OpenHelp 的信息，其中包括在帮助系统中添加帮助文件的内容。

8.3 添加属性编辑器

对象查看器为所有类型的属性提供缺省的编辑处理。但是，你可以通过编写并注册属性编辑器为特定的属性提供一个代用的编辑器。你可以注册只应用于你所编写组件的属性的属性编辑器，但是，你也可以创建应用于某特定类型的所有属性的属性编辑器。

从最简单的层面来说，属性编辑器可以用任何一种或者二种方式工作：显示并允许用户作为文本串编辑当前值；显示允许进行其它一些类型的编辑操作的一个对话框。根据被编辑属性的不同，你可能会发现，提供任一种方式、或者二种方式都提供，这是很有用的。

编写属性编辑器需要五个步骤：

1. 派生属性编辑器类
2. 作为文本编辑属性
3. 作为整体编辑属性
4. 指定编辑器的特性
5. 注册属性编辑器

8.3.1 派生属性编辑器类

二个组件库都定义了几种属性编辑器，它们全部都来自 `TPropertyEditor`。当你创建属性编辑器时，你的属性编辑器类可以直接来自 `TPropertyEditor`，或者间接来自在表 8.1 中描述的一个属性编辑器类。在 `DesignEditors` 单元中的类可被用于 `VCL` 和 `CLX` 应用程序，但是，其中有一些属性编辑器类提供专门的对话框，并因此专用于 `VCL` 或 `CLX`。这些可分别在 `VCLEditors` 和 `CLXEditors` 单元中发现。

注释 属性编辑器来自 `TBasePropertyEditor` 并且支持 `IProperty` 接口是完全必要的。但是，`TPropertyEditor` 提供 `IProperty` 接口的缺省实现。

表 8.1 的列表并不完整。`VCLEditors` 和 `CLXEditors` 单元也定义了一些非常专门化的用于独特属性（如组件名称）的属性编辑器。表中列出的属性编辑器对用户定义的属性是最有用的。

表 8.1 预定义的属性编辑器的类型

类型	被编辑的属性
<code>TOrdinalProperty</code>	继承自 <code>TordinalProperty</code> 的所有顺序属性编辑器（用于整数、字符、枚举属性）
<code>TIntegerProperty</code>	所有整数类型，包括预定义的和用户定义的子界类型
<code>TCharProperty</code>	字符类型和字符的子界类型，如 'A'..'Z'
<code>TEnumProperty</code>	任何枚举类型
<code>TFloatProperty</code>	所有浮点数
<code>TStringProperty</code>	字符串
<code>TSetElementProperty</code>	以布尔值显示的集合中的单个元素
<code>TSetProperty</code>	所有集合。集合不可以直接编辑，但可以扩展为一列集合元素属性
<code>TClassProperty</code>	类。显示类的名称，并允许类属性的扩展
<code>TMethodProperty</code>	方法指针，非常特殊的事件
<code>TComponentProperty</code>	在同一窗体中的组件。用户不能编辑组件的属性，但是可以指向一个兼容类型的特定组件

TColorProperty	组件颜色。如果可能，则显示颜色常数，否则显示 16 进制值。下拉列表中包含颜色常数。双击可打开颜色选择对话框
TFontNameProperty	字体名。下拉列表显示当前已安装的所有字体
TFontProperty	字体。允许单个字体属性的扩展，并允许访问字体对话框

下面的示例显示被命名为 TMyPropertyEditor 的简单属性编辑器的声明：

```

type
  TFloatProperty = class(TPropertyEditor)
  public
    function AllEqual: Boolean; override;
    function GetValue: string; override;
    procedure SetValue(const Value: string); override;
  end;

```

8.3.2 作为文本编辑属性

所有属性都需要为了在对象查看器中的显示提供其值的字符串表示。大部分属性也允许用户为属性输入一个新值。TPropertyEditor 和其子孙提供了可重载的虚拟方法来在文本表示和其实际值间实现转换。

你重载的方法被称为 GetValue 和 SetValue。你的属性编辑器为了赋值和读取 TPropertyEditor 中不同种类的值也继承方法，见表 8.2 中所示：

表 8.2 读、写属性值的方法

属性类型	Get 方法	Set 方法
浮点数	GetFloatPoint	SetFloatPoint
方法指针(事件)	GetMethodValue	SetMethodValue
顺序类型	GetOrdValue	SetOrdValue
字符串	GetStrValue	SetStrValue

当重载 GetValue 方法时，调用其中一个 Get 方法。当重载 SetValue 时，调用其中一个 Set 方法。

8.3.2.1 显示属性值

属性编辑器的 GetValue 方法返回代表属性当前值的串。对象查看器在属性的数值栏中使用这个串。GetValue 缺省返回"unknown"。

要提供属性的串表示，重载属性编辑器的 GetValue 方法。

如果属性不是串值，GetValue 必须把此值转换为串表示。

8.3.2.2 设置属性值

属性编辑器的 SetValue 方法采用用户在对象查看器中输入的串，把它转换为合适的类型，并设定属性的值。如果这个串没有表示为属性合适的值，SetValue 就应该抛出一个异常，

并放弃这个不合适的值。

要把串值读进属性，重载属性编辑器的 `SetValue` 方法。

`SetValue` 应该在调用其中一个 `Set` 方法以前转换串，并验证其值。

下面的例子是 `GetValue` 和 `SetValue` 方法，它们用于 `Integer` 类型的 `TIntegerProperty`。`Integer` 是顺序类型，因此 `GetValue` 调用 `GetOrdValue`，并转换其结果为串。`SetValue` 转换串为 `Integer`，并执行一些范围检查，调用 `SetOrdValue`。

```
function TIntegerProperty.GetValue: string
begin
    with GetTypeData(GetProperty)^ do
        if OrdType = otULong then           // unsigned
            Result := IntToStr(Cardinal(GetOrdValue))
        else
            Result := IntToStr(GetOrdValue);
    end;

procedure TIntegerProperty.SetValue(const Value: string);
    procedure Error(const Args: array of const);
    begin
        raise EPropertyError.CreateResFmt(@SOutOfRange, Args);
    end;
var
    L: Int64;
begin
    L := StrToInt64(Value);
    with GetTypeData(GetProperty)^ do
        if OrdType = otULong then
            begin // unsigned compare and reporting needed
                if (L < Cardinal(MinValue)) or (L > Cardinal(MaxValue)) then
                    // bump up to Int64 to get past the %d in the format string
                    Error([Int64(Cardinal(MinValue)), Int64(Cardinal(MaxValue))]);
            end
            else if (L < MinValue) or (L > MaxValue) then
                Error([MinValue, MaxValue]);
        SetOrdValue(L);
    end;
```

上面这个特殊例子的细节没有原理重要：`GetValue` 把值转换为串；`SetValue` 在调用一个“`Set`”方法以前，转换串并验证其值。

8.3.3 作为整体编辑属性

你可以随你的意愿提供一个对话框，在此对话框中用户能够直观地编辑属性。属性编辑器最普遍的应用是用于属性本身类的属性。例如 `Font` 属性，用户可打开一个字体对话框来一次性选择字体的所有属性。

要提供一个对所有属性进行编辑的编辑器对话框，重载此属性编辑器类的 `Edit` 方法。

Edit 方法使用与编写 GetValue 和 SetValue 方法相同的 Get 和 Set 方法。事实上，一个 Edit 方法调用一个 Get 方法和一个 Set 方法。因为编辑器是类型特定的，所以通常不需要把属性值转换为串。编辑器一般处理“取回”的值。

当用户点击属性后的“...”按钮，或者双击输入值框时，对象查看器调用属性编辑器的 Edit 方法。

在你的 Edit 方法实现中，遵循下面这些步骤：

1. 构造你正用于属性的编辑器
2. 使用 Get 方法读取当前值，并把此值赋给属性
3. 当用户选择新值时，使用 Set 方法把新值赋给属性
4. 销毁编辑器

在大部分组件中都会有的 Color 属性使用标准的 Windows 颜色对话框作为属性编辑器。下面是来自 TColorProperty 的 Edit 方法，它调用这个对话框并使用其结果：

```
procedure TColorProperty.Edit;
var
  ColorDialog: TColorDialog;
begin
  ColorDialog := TColorDialog.Create(Application); // construct the editor
  try
    ColorDialog.Color := GetOrdValue;           // use the existing value
    if ColorDialog.Execute then                  // if the user OKs the dialog ...
      SetOrdValue(ColorDialog.Color);           // ... use the result to set value
  finally
    ColorDialog.Free;                           // destroy the editor
  end;
end;
```

8.3.4 指定编辑器特性

属性编辑器必须提供对象查看器可以使用的信息来确定用什么工具来显示内容。例如，对象查看器需要知道属性是否有子属性，或者是否可显示一系列可能的值。

要指定编辑器的特性，重载此属性编辑器的 GetAttributes 方法。

GetAttributes 是一个方法，它返回一组类型为 TPropertyAttributes 的值，它可能包含下列任一个值或者所有的值：

表 8.3 属性编辑器的特性标志

标志	相关方法	可能包含的意义
paValueList	GetValues	编辑器可以给出一列枚举值
paSubProperties	GetProperties	属性有可以显示的子属性
paDialog	Edit	编辑器可以显示一个用于编辑全部属性的对话框
paMultiSelect	N/A	当用户选择多个组件时，此属性应该显示
paAutoUpdate	SetValue	在每一个改变后更新组件，而不是等待值的确认
paSortList	N/A	对象查看器应该对值列表进行排序

paReadOnly	N/A	用户不能修改属性值
paRevertable	N/A	使能对象查看器上的 Revert to Inherited 菜单项。此菜单项告诉属性编辑器放弃当前属性值，并返回一些以前建立的缺省值或标准值
paFullWidthName	N/A	值不需要显示。对象查看器使用整个宽度显示属性名
paVolatileSubProperties	GetProperties	无论何时属性值一改变，对象查看器就重新取其全部子属性的值
paReference	GetComponentValue	值是对其它东西的引用。当与 paSubProperties 一起使用时，引用的对象应作为此属性的子属性被显示

Color 属性比大部分属性都容易被改变，它们允许用户在对象查看器中使用几种方式选择：直接输入、从列表中选择、定制编辑器。因此，TColorProperty 的 GetAttributes 方法在其返回值中包含几个特性：

```
function TColorProperty.GetAttributes: TPropertyAttributes;
begin
    Result := [paMultiSelect, paDialog, paValueList, paRevertable];
end;
```

8.3.5 注册属性编辑器

一旦创建了属性编辑器，就需要向 Delphi 注册。注册属性编辑器把属性的类型与特定的属性编辑器关联。你可以注册属性编辑器关联某给定类型的全部属性，或者只是关联组件的一个特定类型的特定的属性。

要注册属性编辑器，调用 RegisterPropertyEditor 过程。

RegisterPropertyEditor 采用 4 个参数：

- 指向要编辑属性类型的类型信息指针
这总是调用内置的 TypeInfo 函数，如 TypeInfo(TMyComponent)
- 应用编辑器的组件的类型。如果此参数为 nil，则此编辑器应该用于给定类型的所有属性
- 属性名。只有前面的参数指定某特定类型的组件，此参数才有意义。这时，你可以在应用此编辑器的这个组件类型中指定某特定属性的名称。
- 用于编辑指定属性的属性编辑器的类型

下面是此过程的一段摘录，它在组件面板上为标准组件注册编辑器：

```
procedure Register;
begin
    RegisterPropertyEditor(TypeInfo(TComponent), nil, '', IComponentProperty);
    RegisterPropertyEditor(TypeInfo(TComponentName), TComponent, 'Name',
                           TComponentNameProperty);
    RegisterPropertyEditor(TypeInfo(TMenuItem), TMenuItem, '', TMenuItemProperty);
end;
```

在这个过程中的三个语句包含了 RegisterPropertyEditor 的不同的用法：

- 第一句最典型，它为 TComponent 类型（或没有注册其自己编辑器的 TComponent 的子孙）的所有属性注册 TComponentProperty 属性编辑器。一般，当注册属性编辑器时，你只是为某一特定类型创建编辑器，但是你想把它用于此类型的所有属性，所以第 2、3 个参数分别为 nil 和空串。
- 第二句是最特别的注册，它为组件的特定类型的特定属性注册编辑器。这时，此编辑器用于所有组件的 TComponentName 类型的 Name 属性。
- 第三句比第一句特别，但不象第二句受限制。它为 TMenu 类型的组件中的 TMenuItem 类型的所有属性注册编辑器

8.4 属性分类

在 IDE 中，对象查看器让你选择性地隐藏和显示基于属性分类的属性。新定制组件的属性通过按分类注册属性可适合此方案。在做这个的同时，通过调用 RegisterPropertyInCategory 或者 RegisterPropertiesInCategory 注册组件。使用 RegisterPropertyInCategory 来注册一个属性。使用 RegisterPropertiesInCategory 在一个函数调用中注册多个属性。这些函数都定义在 DesignIntf 单元中。

注意，当进行注册时，注册属性或者注册定制组件的所有属性并不是强制性的。不显式地与一个分类关联的任何属性都被包括在 TMiscellaneousCategory 分类中。这样的属性在对象查看器中基于缺省的分类被隐藏或者显示。

除了这两个注册属性的函数外，还有一个 IsPropertyInCategory 函数。这个函数在创建本地化应用程序时很有用。在创建本地化程序过程中，你必须确定属性是否被注册在一个给定的属性分类中。

8.4.1 一次注册一个属性

一次注册一个属性，并使用 RegisterPropertyInCategory 函数把它与一个属性分类关联。RegisterPropertyInCategory 有四个不同的重载函数，每一个提供一套在关联属性分类的定制组件中识别属性的不同标准。

第一个函数让你通过属性名识别属性。下面一行代码注册一个涉及组件的可视性显示的属性，识别此属性通过其名称 “AutoSize”。

```
RegisterPropertyInCategory('Visual', 'AutoSize');
```

第二个函数与第一个很相似，不同的是它限制分类为只出现在某给定类型组件上的给定名称的属性。下面的例子注册一个定制类 TMyButton 的组件的名称为 “HelpContext” 的属性（属于 “Help and Hints” 分类）。

```
RegisterPropertyInCategory('Help and Hints', TMyButton, 'HelpContext');
```

第三个函数使用其类型而不是名称识别属性。下面的例子注册一个基于 Integer 类型的属性。

```
RegisterPropertyInCategory('Visual', TypeInfo(Integer));
```

第四个函数同时使用属性的类型和名称识别属性。下面的举例注册一个基于 TBitmap 类型和 “Pattern” 名称的属性。

```
RegisterPropertyInCategory('Visual', TypeInfo(TBitmap), 'Pattern');
```

参见“指定属性分类”节，有一列可用的属性分类和对其用法的简单描述。

8.4.2 一次注册多个属性

一次注册多个属性，并使用 `RegisterPropertiesInCategory` 函数把它们与属性分类关联。`RegisterPropertiesInCategory` 有三个重载函数，每一个提供一套不同的在关联属性分类的定制组件中识别属性的标准。

第一个函数让你基于属性名称或者类型识别属性。列表作为一组常数数组被传递。在下面的例子中，名为“Text”或者属于 `TEdit` 类型的任何属性被注册在‘Localizable’分类中。

```
RegisterPropertiesInCategory('Localizable', ['Text', TEdit]);
```

第二个函数让你限制注册的属性为只属于特定组件的属性。要注册的属性列表只包括名称，不包括类型。例如，下面的代码把很多属性注册到所有组件的‘Help and Hints’分类中：

```
RegisterPropertiesInCategory('Help and Hints', TComponent, ['HelpContext', 'Hint', 'ParentShowHint', 'ShowHint']);
```

第三个函数让你限制注册的属性为有特定类型的属性。与第二个函数一样，要注册的属性列表只可以包含名称。

```
RegisterPropertiesInCategory('Localizable', TypeInfo(String), ['Text', 'Caption']);
```

参见“指定属性分类”节，有一列可用属性分类，并有其用法的简短描述。

8.4.3 指定属性分类

当在某一分类中注册属性时，你可以使用任何你想使用的串作为此分类的名称。如果你使用一个以前没有用过的串，对象查看器就用此名称产生一个新属性分类的类。但是，你也可以注册属性到某内置分类中。内置属性分类描述见表 8.4：

表 8.4 属性分类

分类	目的
Action	涉及运行时动作的属性。TEdit 的 Enabled 和 Hint 属性在此分类中
Database	涉及数据库操作的属性。TQuery 的 DatabaseName 和 SQL 属性在此分类
Drag 、 Drop 和 Docking	涉及拖放和停靠操作的属性。TImage 的 DragCursor 和 DragKind 属性在此分类中
Help 和 Hints	涉及使用在线帮助或提示的属性。TMemo 的 HelpContext 和 Hint 属性在此分类中
Layout	涉及设计时控件的可视显示的属性，TDBEdit 的 Top 和 Left 属性在此分类中
Legacy	涉及过时操作（obsolete operations)的属性。TComboBox 的 Ctl3D 和 ParentCtl3D 属性在此分类中
Linkage	涉及从一个组件到另一个组件关联或者链接的属性。TDataSource 的 DataSet 属性在此分类中
Locale	涉及国际语言本地化的属性。TMainMenu 的 BiDiMode 和 ParentBiDiMode 属性在此分类中
Localizable	应用程序本地化版本可能需要修改的属性。就象确定控件的大小和位置的属性一样，很多串属性(如 Caption)在此分类中。
Visual	涉及运行时控件的可视显示的属性。TScrollBar 的 Align 和 Visible 属性在此分类中

Input	涉及数据输入(不需要涉及数据库操作)的属性。TEdit 的 Enabled 和 ReadOnly 属性在此分类中
Miscellaneous	不适合某一分类或者不需要被分类(以及不被显式地注册到某一指定分类的属性)的属性。TSpeedButton 的 AllowAllUp 和 Name 属性在此分类中

8.4.4 使用 IsPropertyInCategory 函数

应用程序可以查询现有的被注册的属性，确定给定的属性是否已被注册于某特定分类。这在象执行本地化操作以前由本地化程序检查属性分类这样的情况下可能特别有用。IsPropertyInCategory 函数的两个重载函数都是可用的，它们在确定属性是否属于某一分类时采用不同的标准。

第一个函数基于比较所拥有组件的类的类型和属性的名称组合的标准。在下面的命令行中，IsPropertyInCategory 要返回 True，属性必须属于 TCustomEdit 的子孙，具有“Text”名称，并属于‘Localizable’属性分类。

```
IsItThere := IsPropertyInCategory('Localizable', TCustomEdit, 'Text');
```

第二个函数基于比较所拥有组件的类的名称和属性的名称组合的标准。在下面的命令行中，IsPropertyInCategory 要返回 True，属性必须为 TCustomEdit 的子孙，具有“Text”名称，并属于‘Localizable’属性分类。

```
IsItThere := IsPropertyInCategory('Localizable', 'TCustomEdit', 'Text');
```

8.5 添加组件编辑器

组件编辑器确定当组件在设计器中被双击时会发生什么，并且当鼠标右键点击组件时给显示的上下文菜单添加命令。组件编辑器也可以用定制格式复制你的组件到 Windows 剪贴板。

如果你没有给你的组件一个组件编辑器，Delphi 会使用缺省的组件编辑器。缺省的组件编辑器是由类 TDefaultEditor 实现的。TDefaultEditor 不给组件的上下文菜单添加任何新的项。当组件被双击时，TDefaultEditor 查询组件的属性，并产生（或导航到）它发现的第一个事件处理程序。

要给上下文菜单添加项，当组件被双击时改变其特性，或者添加新的剪贴板格式，从 TComponentEditor 导出新的类，并用你的组件注册其用法。在你的重载方法中，可以使用 TComponentEditor 的 Component 属性来访问正被编辑的组件。

添加定制组件编辑器，需要下面几步：

- 给上下文菜单添加项
- 改变双击行为
- 添加剪贴板格式
- 注册组件编辑器

8.5.1 给上下文菜单添加项

当用户用鼠标右键点击组件时，组件编辑器的 `GetVerbCount` 和 `GetVerb` 方法被调用来构建上下文菜单。你可以重载这些方法，给上下文菜单添加命令（动词）。

给上下文菜单添加项需要的步骤：

- 指定菜单项
- 实现命令

8.5.1.1 指定菜单项

重载 `GetVerbCount` 方法，返回你正向上下文菜单添加命令的数目。重载 `GetVerb` 方法，返回给每一个这些命令添加的串。当重载 `GetVerb` 方法时，添加一个记号 `&` 给串，以引起在上下文菜单中 `&` 后面的字符显示下划线，并作为选择此菜单项的快捷键。如果某命令会引出一个对话框，则确保在命令后面添加一个省略号（`...`）。`GetVerb` 有一个单独的参数，用于指示命令的索引。

下面的代码重载 `GetVerbCount` 和 `GetVerb` 方法，给上下文菜单添加两个命令：

```
function TMyEditor.GetVerbCount: Integer;
begin
    Result := 2;
end;

function TMyEditor.GetVerb(Index: Integer): string;
begin
    case Index of
        0: Result := '&DoThis ...';
        1: Result := 'Do&That';
    end;
end;
```

注释 你的 `GetVerb` 方法为由 `GetVerbCount` 指示的每一个可能存在的索引返回一个值。

8.5.1.2 实现命令

当由 `GetVerb` 提供的命令在设计器中被选择时，`ExecuteVerb` 方法被调用。对于你在 `GetVerb` 方法中提供的每一个命令，在 `ExecuteVerb` 方法中实现一个动作。你可以使用编辑器的 `Component` 属性来访问正被编辑的组件。

例如，下面的 `ExecuteVerb` 方法为前面示例中的 `GetVerb` 方法实现命令。

```
procedure TMyEditor.ExecuteVerb(Index: Integer);
var
    MySpecialDialog: TMyDialog;
begin
    case Index of
        0: begin
```

```

        MyDialog := TMySpecialDialog.Create(Application); // instantiate the editor
        if MySpecialDialog.Execute then // if the user OKs the dialog ...
            // ...use the value
            MyComponent.FThisProperty := MySpecialDialog.ReturnValue;
        MySpecialDialog.Free;
    end;
1:   That;
end;
end;
end;

```

8.5.2 改变双击行为

当组件被双击时，组件编辑器的 `Edit` 方法被调用。缺省地，`Edit` 方法执行被添加到上下文菜单中的第一个命令。因此，在前面的示例中，双击组件就执行 `DoThis` 命令。

虽然执行第一个命令通常是一个不错的想法，但是你或许想改变这个缺省的行为。例如，在以下情况下，可以提供一个备选的行为：

- 没有给上下文菜单添加任何命令
- 当组件被双击时，想显示组合了几个命令的对话框

当组件被双击时，重载 `Edit` 方法来指定一个新的行为。例如，当用户双击组件时，下面的 `Edit` 方法会引出一个字体对话框：

```

procedure TMyEditor.Edit;
var
    FontDialog: TFontDialog;
begin
    FontDialog := TFontDialog.Create(Application);
    try
        if FontDialog.Execute then
            MyComponent.FFont.Assign(FontDialog.Font);
    finally
        FontDialog.Free;
    end;
end;
end;

```

注释 如果你想在组件上双击时显示事件处理程序的代码编辑器，那么就使用 `TDefaultEditor` 作为你的组件编辑器的基类，而不是使用 `TComponentEditor`。然后，不重载 `Edit` 方法，而是重载 `protected` 的 `TDefaultEditor.EditProperty` 方法。`EditProperty` 扫描组件的所有事件处理程序，并唤起所发现的第一个事件处理程序。当然你可以改变这个来关注某特定的事件。例如：

```

procedure TMyEditor.EditProperty(PropertyEditor: TPropertyEditor; Continue,
    FreeEditor: Boolean);
begin
    if (PropertyEditor.ClassName = 'TMethodProperty') and
        (PropertyEditor.GetName = 'OnSpecialEvent') then
        // DefaultEditor.EditProperty(PropertyEditor, Continue, FreeEditor);
end;
end;

```


8.5.3 添加剪贴板格式

缺省地，在 IDE 中，如果一个组件在被选中期间，当用户选择‘复制’时，组件就以 Delphi 的内部格式被复制，然后它就可以用另一种格式或者数据模块被粘贴。你的组件可以通过重载 Copy 方法复制其它的格式到剪贴板。

例如，下面的 Copy 方法允许 TImage 组件复制其图片到剪贴板，这个图片被 Delphi 的 IDE 忽略，但是可被粘贴到其它的应用程序中。

```
procedure TMyComponent.Copy;
var
    MyFormat: Word;
    AData, APalette: THandle;
begin
    TImage(Component).Picture.Bitmap.SaveToClipboardFormat(MyFormat, AData, APalette);
    Clipboard.SetAsHandle(MyFormat, AData);
end;
```

8.5.4 注册组件编辑器

一旦组件编辑器被定义，它就可以被注册来用特定的组件类工作。当这个被注册的组件编辑器在窗体设计器中被选择时，它就可以为此类的每一个组件创建组件编辑器。

要在组件编辑器和组件类之间建立关联，就调用 RegisterComponentEditor。RegisterComponentEditor 采用使用此编辑器的组件类的名称和你定义的组件编辑器类的名称作为参数。例如，下面的语句注册一个名为 TMyEditor 的组件编辑器类，适用于 TMyComponent 类型的所有组件。

```
RegisterComponentEditor(TMyComponent, TMyEditor);
```

在注册你的组件的 Register 过程中调用 RegisterComponentEditor。例如，如果新组件 TMyComponent 和它的组件编辑器 TMyEditor 在相同的单元中被实现，下面的代码就注册这个组件和与这个组件编辑器的关联。

```
procedure Register;
begin
    RegisterComponents('Miscellaneous', [TMyComponent]);
    RegisterComponentEditor(classes[0], TMyEditor);
end;
```

8.6 把组件编译到包中

一旦组件被注册，你就必须在它们被安装到 IDE 以前把它们编译为包。一个包可能包含一个或多个组件和定制的属性编辑器。有关包的更多的信息，参见《开发者指南》第 16 章“用包和组件工作”。

要创造和编译包，参见《开发者指南》16-11 页的“创建并编辑包”。把你定制组件的源代码单元放在包的‘内容’列表中。如果你的组件依赖其它包，就包括这些包在‘需求’列

表中。

要在 IDE 中安装你的组件，参见《开发者指南》16-11 页的“安装组件包”。

第 9 章 修改已有的组件

创建组件最简单的方法是从几乎可做你想做的任何事的组件中派生组件，然后进行你需要的任何改变。后面给出一个简单的示例，它修改标准的 memo 组件来创建一个新的 memo 组件，此组件缺省地不自动回绕字符。

Memo 组件的 WordWrap 属性的值初始被设为 True。如果你频繁地使用不回绕的 memo 组件，那么你就可以创建一个新的 memo 组件，它在缺省情况下不回绕字符。

注释 要为已存在的组件修改已发布的属性，或者保存特定的事件处理程序，经常是简单地使用一个组件模板，而不是创建一个新类。

修改已有组件，只用两个步骤：

- 创建并注册组件
- 修改组件类

9.1 创建并注册组件

创建每一个组件都用相同的方法：创建单元、派生组件类、注册组件、在组件面板上安装组件。这个过程在 1-8 页的“创建新组件”中有大概描述。

注释 一些单元的名称和存放在 CLX 应用程序中会有所不同。例如，Controls 单元在 CLX 应用程序中是 QControls。

对于这个例子，遵循创建组件的一般过程，但有以下特点：

- 把组件的单元称为 Memos
- 从 TMemo 派生新的组件类型 TWrapMemo
- 在组件面板的 Samples 页上注册 TWrapMemo
- 结果单元应看起来象下面这样：

```
unit Memos;
interface
uses
    SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls,
    Forms, StdCtrls;
type
    TWrapMemo = class(TMemo)
    end;
procedure Register;
implementation
procedure Register;
begin
    RegisterComponents('Samples', [TWrapMemo]);
end;
```

end.

如果你要立即编译并安装新组件，它的行为就与其祖先 TMemo 极其相似。在下一节，会对组件做一个简单的改变。

9.2 修改组件类

一旦已经创建了一个新的组件类，就可以用几乎任何方式修改它。这时，只是修改 memo 组件中一个属性的初始值，这涉及到对此组件类的两个小的改变：

- 重载构造函数
- 给属性指定新的缺省值

构造函数实际设置属性的值，缺省值告诉 Delphi 要把什么值保存在窗体文件（在 VCL 应用程序中是 .dfm，在 CLX 应用程序中是 .xfm）中。Delphi 只是保存不同于缺省值的值，所以执行这两个步骤非常重要。

9.2.1 重载构造函数

当在设计时组件被放在窗体上，或者应用程序在运行时构造了一个组件的时候，组件的构造函数设置其属性的值。当组件从窗体文件中被装载时，应用程序设置在设计时被改变的属性。

注释 当你重载构造函数时，新的构造函数在做任何其它事情以前，必须先调用继承的构造函数。有关更多的信息，参见 2-8 页的“重载方法”。

对这个例子，你的新组件需要重载继承自 TMemo 的构造函数来设置 WordWrap 属性为 False。要实现这个任务，给前面声明添加 constructor 和 override，然后在这个单元的 implementation 部分编写新的构造函数：

```
type
  TWrapMemo = class(TMemo)
  public
    // constructors are always public
    constructor Create(AOwner: TComponent); override; // this syntax is always the same
  end;
...
constructor TWrapMemo.Create(AOwner: TComponent); // this goes after implementation
begin
  inherited Create(AOwner); // ALWAYS do this first !
  WordWrap := False; // set the new desired value
end;
```

现在，你可以把这个新组件安装到组件面板上，并可以把它添加到窗体中。注意，WordWrap 属性现在被初始化为 False。

如果你改变属性的一个初始值，也应该把此值作为缺省值分派。如果你没有使用由构造函数设置的值与指定的缺省值相等，Delphi 就不能保存与恢复这个缺省值。

9.2.2 给属性指定新的缺省值

当 Delphi 在窗体文件中保存窗体的描述时，它只保存与属性的缺省值不同的属性值。只保存这些不同的值，则保持窗体文件尺寸小并在被装载时快速。如果你创建一个属性或者改变其缺省值，那么更新声明以包含新的缺省值是一个不错的主意。窗体文件、装载、缺省值在第 8 章“制作在设计时可用的组件”有详尽的解释。

要改变属性的缺省值，重新声明此属性名称，后跟 `default` 指示和新的缺省值。不需要重声明全部属性，只是重声明其名称和缺省值即可。

对于字回绕 `memo` 组件，在对象声明的 `published` 部分重声明 `WordWrap` 属性，并给出缺省值 `False`：

```
type
    TWrapMemo = class(TMemo)
    ...
    published
        property WordWrap default False;
    end;
```

指定缺省属性值并不影响组件的工作方式，你仍然必须在组件的构造函数中初始化此值。重声明缺省值，确保 Delphi 知道什么时候把 `WordWrap` 写到窗体文件中。

第 10 章 创建图形控件

图形控件是一种简单的组件。由于纯图形控件绝不会获得焦点，所以它就没有或者不需要自己的窗口句柄。用户仍然可以用鼠标操作这个控件，但是没有任何键盘接口。

本章介绍的图形控件是 `TShape`，即在组件面板的‘Additional’页上的形状组件。尽管被创建的这个组件与标准形状组件一样，但是你需要以稍微不同的方式调用，以避免标识符重复。本章调用其形状组件 `TSampleShape`，并演示涉及到创建此形状组件的所有步骤：

- 创建并注册组件
- 发布继承的属性
- 添加图形能力

10.1 创建并注册组件

创建每一个组件都是用相同的方法：创建单元、派生组件类、注册组件、编译组件、在组件面板上安装组件。这个过程在 1-8 页的“创建新组件”中有大概的论述。

对这个例子，按一般程序创建组件，但有下面这些特点：

1. 把此组件的单元称为 `Shapes`
 2. 从 `TGraphicControl` 派生新的组件类型 `TSampleShape`
 3. 在组件面板的‘Samples’页（或 CLX 应用程序的其它页）上注册 `TSampleShape`
- 结果单元看起来应该是这样的：

```
unit Shapes;
```

```

interface
uses SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls, Forms;
type
    TSampleShape = class(TGraphicControl)
    end;
procedure Register;
implementation
procedure Register;
begin
    RegisterComponent('Samples', [TSampleShape]);
end;
end.

```

注释 一些单元的名称和位置在 CLX 应用程序中会有所不同。例如，Controls 单元在 CLX 应用程序中是 QControls。

10.2 发布继承的属性

一旦派生了一个组件类型，就可以决定把在祖先类的 `protected` 部分声明的哪些属性和事件你想在新组件中展现出来。TGraphicControl 已经发布了使组件功能表现得象控件一样的所有的属性，这样，所有你需要发布的是响应鼠标事件和处理拖放的能力。

发布继承的属性和事件在 [3-3 页](#)的“发布继承的属性”和 [4-6 页](#)的“使事件可见”中有解释。这两个过程都在类声明的 `published` 部分只重声明属性名。

对于形状控件的例子，你可以发布 3 个鼠标事件，3 个拖放事件，2 个拖放属性：

```

type
    TSampleShape = class(TGraphicControl)
    published
        property DragCursor;           // drag-and-drop properties
        property DragMode;
        property OnDragDrop;           // drag-and-drop events
        property OnDragOver;
        property OnEndDrag;
        property OnMouseDown;         // mouse events
        property OnMouseMove;
        property OnMouseUp;
    end;

```

例子中的形状控件现在使得用户可以使用鼠标和拖放进行交互。

10.3 添加图形能力

一旦声明了图形组件，并发布了你想使其可见的任何继承的属性，就可以添加区分组件的图形能力。当创建图形控件时，有两个任务要执行：

1. 确定画什么

2. 绘制组件图象

另外，对于这个形状控件示例，给它添加一些能使应用程序开发人员在设计时定制形状外观的属性。

10.3.1 确定画什么

图形控件能改变其外观以反映动态条件，包括用户输入。一个看起来总是一样的图形控件或许根本就不是一个组件。如果你想要一个静态图象，可以引入图象而不是使用一个控件。

一般，图形控件的外观依赖于其一些属性的组合。比如，`gauge` 控件有决定其形状和方向以及是否既用数字又用图形显示其进度的属性。同样，图形控件有决定应绘制哪种形状的属性。

要给控件一个决定其绘制形状的属性，就添加 `Shape` 属性。这需要：

1. 声明属性类型
2. 声明属性
3. 编写实现方法

创建属性在第 3 章“创建属性”有更详细的讲解。

10.3.1.1 声明属性类型

当你声明用户定义类型的属性时，必须在包括此属性的类前面首先声明其类型，对属性来说，最普通的用户定义类型是枚举。

对这个形状控件，需要一个枚举类型，它对此控件能绘制的每一种形状都有一个元素对应。

在形状控件类的声明的上面添加下列类型定义：

```
type
    TSampleShapeType = {sstRectangle, sstSquare, sstRoundRect, sstRoundSquare,
                        sstEllipse, sstCircle};
    TSampleShape = class(TGraphicControl)           // this is already there
```

现在，你可以在类中使用这个类型来声明一个新的属性。

10.3.1.2 声明属性

当声明属性时，通常需要声明一个私有域来为此属性保存数据，然后为读写此属性值指定方法。经常，你不需要使用方法来读此值，而可以只指向被保存的数据。

对于此形状控件，你声明一个保存当前形状的域，然后声明一个属性，用此属性读取域，并通过方法调用给其写值。

给 `TSampleShape` 添加下列声明：

```
type
    TSampleShape = class(TGraphicControl)
private
    FShape: TSampleShapeType;           // field to hold property value
    procedure SetShape(Value: TSampleShapeType);
```

```

published
    property Shape: TSampleShapeType read FShape write SetShape;
end;

```

现在，剩下的事情就是添加 SetShape 的实现。

10.3.1.3 编写实现方法

当属性定义的读、写部分是使用方法而不是直接访问保存的属性数据时，就需要实现这个方法。

把 SetShape 方法的实现加到单元的 implementation 部分：

```

procedure TSampleShape.SetShape(Value: TSampleShapeType);
begin
    if FShape <> Value then           // ignore if this isn't a change
    begin
        FShape := Value;             // store the new value
        Invalidate;                  // force a repaint with the new shape
    end;
end;

```

10.3.2 重载构造函数和析构造函数

要为组件改变缺省的属性值并初始化拥有的类，必须重载继承的构造函数和析构造函数。在这两种情况下，记住，在新的构造函数和析构造函数中，永远都要调用继承的方法。

10.3.2.1 改变缺省的属性值

图形控件的缺省尺寸相当小，所以可以在构造函数中改变其宽度和高度。改变缺省属性值在第 9 章“修改已有的组件”中有详细的讲解。

在这个例子中，形状控件设置其大小为每边都是 65 像素的正方形。

给组件类的声明中添加被重载的构造函数：

```

type
    TSampleShape = class(TGraphicControl)
    public
        // constructors are always public
        constructor Create(AOwner: TComponent); override; // remember override directive
    end;

1. 用其新的缺省值重声明 Height 和 Width 属性：
    type
        TSampleShape = class(TGraphicControl)
        ...
        published
            property Height default 65;
            property Width default 65;

```

```

        end;
2. 在单元的 implementation 部分编写新的构造函数:
    constructor TSampleShape.Create(AOwner: TComponent);
    begin
        inherited Create(AOwner); // always call the inherited constructor
        Width := 65;
        Height := 65;
    end;

```

10.3.3 发布画笔和画刷

缺省地，画布有一个细的、黑色的画笔和一个实心的、白色的画刷。要让开发人员改变画笔和画刷，则必须为它们提供类，用于在设计时的操作，然后在绘图过程中复制此类到画布上。象辅助的画笔和画刷这样的类被称为自有类（owned classes），因为组件拥有它们，并为创建和销毁其本身而负责。

管理自有类需要：

1. 声明类域
2. 声明访问属性
3. 初始化自有类
4. 设置自有类的属性

10.3.3.1 声明类域

组件拥有的每一个类都必须有一个在此组件中为其声明的类域。这个类域确保组件总是有一个指向自己所拥有对象的指针，这样它就可以在销毁自身以前销毁这个类。一般，组件在其构造函数中初始化其所拥有的对象，并在析构函数中销毁这些对象。

拥有对象的域几乎总是被声明为 **private**。如果应用程序（或其它组件）需要有权访问拥有的对象，就必须为此目的声明 **published** 或 **public** 属性。

为画笔和画刷给形状控件添加域：

```

type
    TSampleShape = class(TGraphicControl)
    private
        // fields are nearly always private
        FPen: TPen;      // a field for the pen object
        FBrush: TBrush;  // a field for the brush object
        ...
    end;

```

10.3.3.2 声明访问属性

你可以通过声明这种类型对象的属性来提供对组件所拥有对象的访问。这给开发人员一个途径来在设计时或者运行时访问这些对象。通常，属性的读部分只是引用类域，而写部分是调用一个方法，它能使此组件在拥有的对象中对于改变作出反应。

对这个形状控件添加提供访问画笔和画刷域的属性，也声明方法来对画笔和画刷的改变作出反应。

```
type
    TSampleShape = class(TGraphicControl)
    ...
    private           // these methods should be private
        procedure SetBrush(Value: TBrush);
        procedure SetPen(Value: TPen);
    published        // make these available at design time
        property Brush: TBrush read FBrush write SetBrush;
        property Pen: TPen read FPen write SetPen;
    end;
```

然后，在单元的 implementation 部分编写 SetBrush 和 SetPen 函数：

```
procedure TSampleShape.SetBrush(Value: TBrush);
begin
    FBrush.Assign(Value);      // replace existing brush with parameter
end;

procedure TSampleShape.SetPen(Value: TPen);
begin
    FPen.Assign(Value);        // replace existing pen with parameter
end;
```

直接给 FBrush 赋值 Value 的内容：

```
FBrush := Value;
```

这将重写内部指针 FBrush，造成内存泄露，并产生大量的权属问题。

10.3.3.3 初始化自有类

如果给你的组件添加类，组件的构造函数必须初始化这些类，这样用户就可以在运行时与其对象交互。同样地，组件的析构函数也必须在销毁组件本身以前销毁拥有的对象。

因为你已经给形状控件添加了画笔和画刷，所以，需要在形状控件的构造函数中初始化它们，并在控件的析构函数中销毁它们：

1. 在形状控件的构造函数中构造画笔和画刷：

```
constructor TSampleShape.Create(AOwner: TComponent);
begin
    inherited Create(AOwner);  // always call the inherited constructor
    Width := 65;
    Height := 65;
    FPen := TPen.Create;        // construct the pen
    FBrush := TBrush.Create;    // construct the brush
end;
```

2. 给组件类的声明中添加重载的析构函数：

```
type
    TSampleShape = class(TGraphicControl)
```

```

public                                // destructors are always public
    constructor Create(AOwner: TComponent); override;
    destructor Destroy; override; // remember override directive
end;
3. 在单元的 implementation 部分编写新的析构函数:
destructor TSampleShape.Destroy;
begin
    FPen.Free;                        // destroy the pen object
    FBrush.Free;                      // destroy the brush object
    inherited Destroy;                // always call the inherited destructor, too
end;

```

10.3.3.4 设定自有类的属性

作为处理画笔和画刷类的最后一步, 需要确保画笔和画刷的改变会引起形状控件重绘自身。画笔和画刷两个类都有 **OnChange** 事件, 因此, 可以在形状控件中创建一个方法, 并把这两个 **OnChange** 事件指向它。

添加下列方法给形状控件, 更新组件的构造函数, 在新方法中设置画笔和画刷事件:

```

type
    TSampleShape = class(TGraphicControl)
    published
        procedure StyleChanged(Sender: TObject);
    end;
...
implementation
...
constructor TSampleShape.Create(AOwner: TComponent);
begin
    inherited Create(AOwner);          // always call the inherited constructor
    Width := 65;
    Height := 65;
    FPen := TPen.Create;               // construct the pen
    FPen.OnChange := StyleChanged;     // assign method to OnChange event
    FBrush := TBrush.Create;           // construct the brush
    FBrush.OnChange := StyleChanged;   // assign method to OnChange event
end;

procedure TSampleShape.StyleChanged(Sender: TObject);
begin
    Invalidate;                       // erase and repaint the component
end;

```

对于这些变化, 组件进行重绘来反映画笔或者画刷的改变。

10.3.4 绘制组件图象

图形控件的基本要素是在屏幕上绘制其图象的方法。抽象类型 `TGraphicControl` 定义了 `Paint` 方法，重载此方法以在你的控件中绘制你想要的图象。

这个形状控件的 `Paint` 方法需要做以下几件事情：

1. 使用由用户选择的画笔和画刷
2. 使用选择的形状
3. 调整坐标，这样正方形和圆使用相同的宽和高

重载 `Paint` 方法需要两步：

1. 把 `Paint` 添加到组件的声明中
2. 在单元的 `implementation` 部分编写 `Paint` 方法

对示例形状控件来说，在类的声明中添加下列声明：

```
type
    TSampleShape = class(TGraphicControl)
    ...
protected
    procedure Paint; override;
    ...
end;
```

然后，在单元的 `implementation` 部分编写这个方法：

```
procedure TSampleShape.Paint;
begin
    with Canvas do
    begin
        Pen := FPen;                // copy the component's pen
        Brush := FBrush;            // copy the component's brush
        case FShape of
            sstRectangle, sstSquare:
                Rectangle(0, 0, Width, Height);    // draw rectangles and squares
            sstRoundRect, sstRoundSquare:
                // draw rounded shapes
                RoundRect(0, 0, Width, Height, Width div 4, Height div 4);
            sstCircle, sstEllipse:
                Ellipse(0, 0, Width, Height);      // draw round shapes
        end;
    end;
end;
```

每当控件需要更新其图象时，`Paint` 就被调用。当控件第一次出现或者在控件前面的窗口离开时控件被绘制。另外，你可以通过调用 `Invalidate` 来强制重绘控件，就象 `StyleChanged` 方法进行的那样。

10.3.5 改进形状绘制

标准的形状控件还做了一件你的简单的形状控件还没有做的事情：它处理了正方形、圆形以及矩形和椭圆形。要做这些，你需要编写找到图象的短边和中心的代码。

下面是改进了的 Paint 方法，它为正方形和椭圆而做了调整：

```
procedure TSampleShape.Paint;
var
  X, Y, W, H, S: Integer;
begin
  with Canvas do
    begin
      Pen := FPen;           // copy the component's pen
      Brush := FBrush;       // copy the component's brush
      W := Width;            // use the component width
      H := Height;           // use the component's height
      if W < H then S := W else S := H; // save smallest for circles/squares
      case FShape of         // adjust height,width and position
        sstRectangle, sstRoundRect, sstEllipse:
          begin
            X := 0; // origin is top-left for these shapes
            Y := 0;
          end;
        sstSquare, sstRoundSquare, sstCircle:
          begin
            X := (W - S) div 2; // center these horizontally...
            Y := (H - S) div 2; // ...and vertically
            W := S;             // use shortest dimension for width...
            H := S;             // ...and for height
          end;
      end;
      case FShape of
        sstRectangle, sstSquare:
          Rectangle(X, Y, X+W, Y+H); // draw rectangles and squares
        sstRoundRect, sstRoundSquare:
          RoundRect(X, Y, X+W, Y+H, S div 4, S div 4); // draw rounded shapes
        sstCircle, sstEllipse:
          Ellipse(X, Y, X+W, Y+H); // draw round shapes
      end;
    end;
  end;
```

第 11 章 定制网格

组件库提供了可用作定制组件的基础的抽象组件。这其中最重要的是网格和列表框。在本章，你将看到如何从基础网格组件 TCustomGrid 来创建一个小的单月日历。

创建这个日历涉及到这些任务：

- 创建并注册组件
- 发布继承的属性
- 改变初始值
- 重定单元格大小
- 填充单元格
- 操纵年和月
- 操纵日

在 VCL 应用程序中，结果组件与组件面板上 Samples 页中的 TCalendar 组件相似。在 CLX 应用程序中，把此组件保存到不同的页里或者创建一个新的面板页。参见 8-3 页的“指定组件面板页”或者在线帮助中的“组件面板、添加页”。

11.1 创建并注册组件

创建每一个组件都使用相同的方法：创建单元、派生组件类、注册组件、编译组件、把组件安装在组件面板上。这个过程在 1-8 页的“创建新组件”有大概的描述。

对于这个示例，按照创建组件的一般过程，但还有以下特点：

1. 把组件的单元称为 CalSamp
2. 从 TCustomGrid 派生新的组件类型 TSampleCalendar
3. 在组件面板的 Samples 页（或 CLX 应用程序的其它页）上注册 TSampleCalendar。

在 VCL 应用程序中，从 TCustomGrid 继承的结果单元应看起来如下：

```
unit CalSamp;
interface
uses
    Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs, Grids;
type
    TSampleCalendar = class(TCustomGrid)
    end;
procedure Register;
implementation
procedure Register;
begin
    RegisterComponents('Samples', [TSampleCalendar]);
end;
end.
```

注释 如果从 TCustomGrid 的 CLX 版本中继承，只有 uses 因显示 CLX 单元而不同。同样，一些单元的名称和位置在 CLX 应用程序中也有所不同。例如，Controls 单元

在 CLX 应用程序中是 QControls。

如果你立即安装这个日历组件，就会发现它显示在 **Samples** 页上。唯一有用的属性是最基本的控件属性。下一步是使一些更专门的属性对这个日历的用户有用。

注释 虽然你可以安装这个只是被编译了的示例日历组件，但是还不能试图把它放在窗体上。TCustomGrid 组件有一个抽象方法 DrawCell，它必须在实例对象能被创建以前被重新声明。重载 DrawCell 方法在 [11-6 页](#) 的“填充单元格”有说明。

11.2 发布继承的属性

抽象网格组件 TCustomGrid 提供了相当多的 `protected` 属性，你可以选择你想要哪些属性，并使它们对日历控件的用户可用。

要使继承的 `protected` 属性对组件的用户可用，在组件声明的 `published` 部分重新声明这些属性。

对这个日历控件，发布下面这些属性和事件：

```
type
    TSampleCalendar = class(TCustomGrid)
    published
        property Align;           { publish properties }
        property BorderStyle;
        property Color;
        property Font;
        property GridLineWidth;
        property ParentColor;
        property ParentFont;
        property OnClick;         { publish events }
        property OnDblClick;
        property OnDragDrop;
        property OnDragOver;
        property OnEndDrag;
        property OnKeyDown;
        property OnKeyPress;
        property OnKeyUp;
    end;
```

有许多其它属性你也可以发布，但是它们不适用于日历，象 `Options` 属性，它可使用户选择绘制哪些网格线。

如果你安装修改过的日历组件到组件面板上，并在应用程序中使用它们，你会发现，还有更多的属性和事件在这个日历中也可用，它们都有强大的功能。现在，你可以开始给自己的设计添加新的能力了。

11.3 改变初始值

日历本质上就是一个有固定数量的行和列的网格，尽管并不是所有的行总是含有日期。由于这个原因，没有发布网格属性 `ColCount` 和 `RowCount`，因为日历用户想显示除了每周七天外的其它事情是极其不可能的。但是，你还必须设定这些属性的初始值，所以周总是被设置为 7。

要改变组件属性的初始值，重载构造函数以设定希望的值。构造函数必须是 `virtual`。

记住，你需要给组件的对象声明的 `public` 部分添加构造函数，然后在组件单元的 `implementation` 部分编写新的构造函数。在新构造函数中的第一句应该总是调用继承的构造函数，然后给 `uses` 语句添加 `StdCtrls` 单元。

```
type
    TSampleCalendar = class(TCustomGrid)
    public
        constructor Create(AOwner: TComponent); override;
    ...
    end;
...
constructor TSampleCalendar.Create(AOwner: TComponent);
begin
    inherited Create(AOwner);           // call inherited component
    ColCount := 7;                       // always seven days/week
    RowCount := 7;                       // always six weeks plus the headings
    FixedCols := 0;                      // no row labels
    FixRows := 1;                        // one row for day names
    ScrollBars := ssNone;                // no need to scroll
    // disable range selection
    Options := Options - [goRangeSelect] + [goDrawFocusSelected];
end;
```

现在，日历有 7 行、7 列，顶行是固定的，或不滚动的。

11.4 重定网格单元的大小

注释 当用户或者应用程序改变窗口或控件的大小时，Windows 给受影响的窗口或控件发送 `WM_SIZE` 消息，这样，它就可调整任何需要的设定值，以适应随后用新尺寸重绘的新图形。你的 VCL 组件可以通过改变网格单元的大小来响应此消息，从而它们都全部安装在控件边界里。要响应 `WM_SIZE` 消息，给此组件添加消息处理的方法。

创建消息处理方法在 7—6 页的“创建新的消息处理程序”中有详尽描述。

既然如此，日历控件需要响应 `WM_SIZE`，所以给控件添加 `protected` 方法 `WMSize`，作为 `WM_SIZE` 消息的索引，然后编写这个方法，计算合适的网格单元大小，使得所有的网格单元在新尺寸下都可见：

```

type
  TSampleCalendar = class(TCustomGrid)
  protected
    procedure WMSize(var Message: TWMSize); message WM_SIZE;
    ...
  end;
...
procedure TSampleCalendar.WMSize(var Message: TWMSize);
var
  GridLines: Integer;           // temporary local variable
begin
  GridLines := 6 * GridLineWidth; // calculate combines size of all lines
  DefaultColWidth := (Message.Width - GridLines) div 7; // set new default cell width
  DefaultRowHeight := (Message.Height - GridLines) div 7; // and cell height
end;

```

现在，当日历尺寸被改变时，它就以能安装到控件里的最大尺寸显示所有的网格单元。

注释 在 CLX 应用程序中，窗口或者控件的尺寸改变是通过调用 `protected` 的 `BoundsChanged` 方法自动得到通知的。你的 CLX 组件可以通过改变网格单元的大小响应这个通告，从而这些网格单元都能安装在控件的边界里。

这时，日历控件需要重载 `BoundsChanged`，这样它会计算出合适的网格单元大小，使所有的网格单元在新尺寸下都可见。

```

type
  TSampleCalendar = class(TCustomGrid)
  protected
    procedure BoundsChanged; override;
    ...
  end;
...
procedure TSampleCalendar.BoundsChanged;
var
  GridLines: Integer;           // temporary local variable
begin
  GridLines := 6 * GridLineWidth; // calculate combines size of all lines
  DefaultColWidth := (Width - GridLines) div 7; // set new default cell width
  DefaultRowHeight := (Height - GridLines) div 7; // and cell height
  inherited;                     // now call the inherited method
end;

```

11.5 填充网格单元

网格控件一个单元一个单元地填充其内容。对于这个日历来说，就是对于每个单元进行计算，判断哪个日期（如果有的话）应归入哪个单元。网格单元的缺省绘制发生在 `virtual` 方法 `DrawCell` 中。

要填充网格单元的内容，重载 `DrawCell` 方法。

填充操作最容易的部分是在固定行的标题单元。运行时库中包含一个有日名称缩写的数组，所有，对此日历的每一列使用其合适的日名：

```
type
    TSampleCalendar = class(TCustomGrid)
    protected
        procedure DrawCell(ACol, ARow: Longint; ARect: TRect;
                           AState: TGridDrawState); override;
    end;
...
procedure TSampleCalendar.DrawCell(ACol, ARow: Longint; ARect: TRect;
                                   AState: TGridDrawState);
begin
    if ARow = 0 then
        // use RTL strings
        Canvas.TextOut(ARect.Left, ARect.Top, ShortDayNames[ACol + 1]);
    end;
```

11.5.1 跟踪日期

为了使此日历控件有用，用户和应用程序必须有一个设定年、月、日的机制。Delphi 在 `TDateTime` 类型的变量中保存日期和时间，`TDateTime` 是日期和时间经过编码的数字表示，它对于编程操作有用，但不便于人的使用。

因此，你可以用编码格式保存日期值，提供运行时对此值的访问，但也可以提供年、月、日属性，使日历组件的用户可以在设计时设置。

跟踪日历中的日期包含以下步骤：

- 保存内部日期
- 访问年、月、日
- 产生日数字
- 选择当前日

11.5.1.1 保存内部日期

要保存日历的日期，需要一个保存日期的私有域和一个提供对此日期访问的运行时属性。

给日历添加内部日期需要三个步骤：

1. 声明一个私有域用来保存日期：

```
type
    TSampleCalendar = class(TCustomGrid)
    private
        FDate: TDateTime;
```

2. 在构造函数中初始化此日期域：

```

constructor TSampleCalendar.Create(AOwner: TComponent);
begin
    inherited Create(AOwner); // this is already here
    ...                       // other initializations here
    FDate := Date;           // get current date from RTL
end;

```

3. 声明一个运行时属性，使有权访问编码日期。

需要一个设定日期的方法，因为设定日期需要更新控件在屏幕上的图象：

```

type
    TSampleCalendar = class(TCustomGrid)
    private
        procedure SetCalendarDate(Value: TDateTime);
    public
        property CalendarDate: TDateTime read FDate write SetCalendarDate;
    ...
    procedure TSampleCalendar.SetCalendarDate(Value: TDateTime);
    begin
        FDate := Value;           // set new date value
        Refresh;                 // update the onscreen image
    end;

```

11.5.1.2 访问年、月、日

经过编码的数字日期很适合应用程序，但是人们更喜欢使用年、月、日。你可以通过创建属性提供对备选的方法来访问那些被保存并编码的日期的各元素。

因为日期的每一个元素(年、月、日)都是整数，并由于在设定每一个元素时都需要编码日期，所以你可以通过对所有这三个属性共享其实现方法来避免每次的重复编码。这就是说，你可以写两个方法，一个用于读元素，另一个用于写元素，并使用这些方法来获取和设置所有这三个属性。

要提供在设计时对年、月、日的访问，做如下操作：

1. 声明这三个属性，给每一个分配一个唯一的索引（index）数

```

type
    TSampleCalendar = class(TCustomGrid)
    public
        property Day: Integer index 3 read GetDateElement write SetDateElement;
        property Month: Integer index 2 read GetDateElement write SetDateElement;
        property Year: Integer index 1 read GetDateElement write SetDateElement;
    ...

```

1. 声明并编写其实现方法，为每一个索引值设定不同的元素：

```

type
    TSampleCalendar = class(TCustomGrid)
    private
        function GetDateElement(Index: Integer): Integer; // note the Index parameter
        procedure SetDateElement(Index: Integer; Value: Integer);

```

```

...
function TSampleCalendar.GetDateElement(Index: Integer): Integer;
var
    AYear, AMonth, ADay: Word;
begin
    DecodeDate(FDate, AYear, AMonth, ADay);    // break encoded date into elements
    case Index of
        1: Result := AYear;
        2: Result := AMonth;
        3: Result := ADay;
        else Result := -1;
    end;
end;

procedure TSampleCalendar.SetDateElement(Index: Integer; Value: Integer);
var
    AYear, AMonth, ADay: Word;
begin
    if Value > 0 then        // all elements must be positive
    begin
        DecodeDate(FDate, AYear, AMonth, ADay);    // get current date elements
        case Index of
            1: AYear := Value;
            2: AMonth := Value;
            3: ADay := Value;
            else Exit;
        end;
        FDate := EncodeDate(AYear, AMonth, ADay);    // encode the modified date
        Refresh;        // update the visible calendar
    end;
end;

```

现在，可以在设计时使用对象查看器，或者在运行时使用代码，设定日历的年、月、日。当然，还没有添加代码来把日期绘制到网格单元，但是现在已经有了需要的数据。

11.5.1.3 产生日数字

把日数字放到日历中涉及到几个方面的考虑。日的数量根据它所在月的不同而不一样，并且还要考虑所给定的年是否闰年的问题。另外，因年、月的不同，月以周的不同的星期值开始。使用 `IsLeapYear` 函数来确定某年是否为闰年，使用 `SysUtils` 单元中的 `MonthDays` 数组来获取某月的日的数量。

一旦有了关于闰年、每月天数这些信息，就可以计算各日期属于哪个网格单元。这个计算是基于此月此周是从哪日开始的。

因为你会需要欲填充的每一个网格单元的月偏离值，所以最好的方法是：一旦当你改变了年或月时，就计算这个值，然后在每次需要时再查阅它。你可以保存此值在一个类域中，

然后在每次日期改变时再更新此域值。

要在合适的网格单元中填充日，需要做如下操作：

1. 给对象添加月偏离域和更新此域值的方法：

type

```
TSampleCalendar = class(TCustomGrid)
private
    FMonthOffset: Integer;           // storage for the offset
...
protected
    procedure UpdateCalendar; virtual; // property for offset access
end;
...
procedure TSampleCalendar.UpdateCalendar;
var
    AYear, AMonth, ADay: Word;
    FirstDate: TDateTime;           // date of the first day of the month
begin
    if FDate <> 0 then               // only calculate offset if date is valid
    begin
        DecodeDate(FDate, AYear, AMonth, ADay); // get elements of date
        FirstDate := EncodeDate(AYear, AMonth, 1); // date of the first
        FMonthOffset := 2 - DayOfWeek(FirstDate); // generate the offset into the grid
    end;
    Refresh;                         // always repaint the control
end;
```

2. 给构造函数和每当日期改变时都调用新的更新方法的 SetCalendarDate 和 SetDateElement 方法添加语句：

```
constructor TSampleCalendar.Create(AOwner: TComponent);
begin
    inherited Create(AOwner);        // this is already here
    ...                             // other initializations here
    UpdateCalendar;                  // set proper offset
end;

procedure TSampleCalendar.SetCalendarDate(Value: TDateTime);
begin
    FDate := Value;                 // this is already here
    UpdateCalendar;                  // this previously called Refresh
end;

procedure TSampleCalendar.SetDateElement(Index: Integer; Value: Integer);
begin
    ...
    FDate := EncodeDate(AYear, AMonth, ADay); // encode the modified date
```

```

        UpdateCalendar;           // this previously called Refresh
    end;
end;
3. 给日历添加一个在被传递网格单元的行、列坐标时返回此单元的日值的方法：
function TSampleCalendar.DayNum(ACol, ARow: Integer): Integer;
begin
    Result := FMonthOffset + ACol + (ARow - 1) * 7;    // calculate day for this cell
    if (Result < 1) or (Result > MonthDays[IsLeapYear(Year), Month]) then
        Result := -1;                                // return -1 if invalid
    end;
    记住，给组件的类型声明中添加 DayNum 的声明。
4. 既然可以计算日期所在单元的位置，就可以更新 DrawCell 来填充这些日期：
procedure TSampleCalendar.DrawCell(ACol, ARow: Longint; ARect: TRect;
                                    AState: TGridDrawState);
var
    TheText: string;
    TempDay: Integer;
begin
    if ARow = 0 then                                // if this is the header row ...
        TheText := ShortDayNames[ACol + 1] // just use the day name
    else begin
        TheText := "";                               // blank cell is the default
        TempDay := DayNum(ACol, ARow); // get number for this cell
        if TempDay <> -1 then
            TheText := IntToStr(TempDay); // use the number if valid
        end;
        with ARect, Canvas do
            TextRect(ARect, Left + (Right - Left - TextWidth(TheText)) div 2,
                    Top + (Bottom - Top - TextHeight(TheText)) div 2, TheText);
        end;
    end;

```

现在，如果你重新安装此日历组件，并在一个窗体置上放一个此组件，你就会看到，当前月的合适的日期信息。

11.5.1.4 选择当前日

既然你已经在日历的单元中有了日数字，那么移动选择焦点到包含当前日的单元中就有意义了。缺省地，选择从左上角单元开始，所以，当初始构造日历和改变日期时，你就需要设置行、列这二个属性。

要把选择放在当前日，改变 UpdateCalendar 方法，使在调用 Refresh 以前先设定行、列。

```

procedure TSampleCalendar.UpdateCalendar;
begin
    if FDate <> 0 then
        begin
            ...                // existing statements to set FMonthOffset

```

```

        Row := (ADay - FMonthOffset) div 7 + 1;
        Col := (ADay - FMonthOffset) mod 7;
    end;
    Refresh;           // this is already here
end;

```

注意，现在你正在重用以前通过解码日期设定的 `ADay` 变量。

11.6 操纵年、月

为操作组件，属性是有用的，尤其是在设计时。但是，有时有非常普通或者自然的操作类型，经常涉及到多个属性，这样就有必要提供方法来处理它们。象这样自然操作的一个例子是日历的“下一月”特性。处理月的循环和年的增加是简单，但是它非常便于使用此组件的开发人员。

把普通操作封装到方法的唯一缺点是方法只能在运行时可用。然而，这样的操作一般只有在重复执行时才令人讨厌，而且在设计时是很少用到的。

对这个日历，添加下面四个方法，用于上、下月和前、后年的处理，每一个方法以稍许不同的方式使用 `IncMonth` 函数，通过年、月值的增量来加、减 `CalendarDate`。

```

procedure TSampleCalendar.NextMonth;
begin
    CalendarDate := IncMonth(CalendarDate, 1);
end;
procedure TSampleCalendar.PrevMonth;
begin
    CalendarDate := IncMonth(CalendarDate, -1);
end;
procedure TSampleCalendar.NextYear;
begin
    CalendarDate := IncMonth(CalendarDate, 12);
end;
procedure TSampleCalendar.PrevYear;
begin
    CalendarDate := IncMonth(CalendarDate, -12);
end;

```

要确保给类的声明中添加新方法的声明。

现在，当你创建使用此日历组件的应用程序时，就可以很容易地实现通过年、月浏览日期。

11.7 操纵日

在某一给定月中，有二个明显的方法来在日间导航。第一个是使用键盘的方向键，另一个是响应对鼠标的点击。标准的网格组件处理这两个方法就象键盘和鼠标都被点击一样。即，

键盘方向键的运动被处理成就象在相邻单元上用鼠标点击一样。

操纵日的过程，由下列步骤组成：

- 移动选择
- 提供 OnChange 事件
- 排除空白单元

11.7.1 移动选择

网格继承的特性处理移动选择，以响应键盘方向键或者鼠标的点击，但是，如果你想改变选择的日，需要修改其缺省特性。

要处理在日历内部的移动，重载网格的 Click 方法。

当重载象 Click 这样的与用户交互结合在一起的方法时，几乎总是包括对一个继承方法的调用，以便不丢失标准的特性。

下面是为日历网格重载的 Click 方法。要确保给 TSampleCalendar 添加 Click 的声明，并在后面包括 override 指示。

```
procedure TSampleCalendar.Click;
var
    TempDay: Integer;
begin
    inherited Click;           // remember to call the inherited method
    TempDay := DayNum(Col, Row); // get the day number for the clicked cell
    if TempDay <> -1 then Day := TempDay; // change day if valid
end;
```

11.7.2 提供 OnChange 事件

既然日历用户可以在日历内改变日期，就有必要允许应用程序响应这些变化。

给 TSampleCalendar 添加 OnChange 事件。

1. 声明事件、保存事件的域，和调用事件的动态方法：

```
type
    TSampleCalendar = class(TCustomGrid)
    private
        FOnChange: TNotifyEvent;
    protected
        procedure Change; dynamic;
    ...
    published
        property OnChange: TNotifyEvent read FOnChange write FOnChange;
    ...
```

2. 编写 Change 方法：

```
procedure TSampleCalendar.Change;
begin
    if Assigned(FOnChange) then FOnChange(Self);
```

```

        end;
3. 在 SetCalendarDate 和 SetDateElement 方法的最后添加调用 Change 的语句:
    procedure TSampleCalendar.SetCalendarDate(Value: TDateTime);
    begin
        FDate := Value;
        UpdateCalendar;
        Change;           // this is the only new statement
    end;

    procedure TSampleCalendar.SetDateElement(Index: Integer; Value: Integer);
    begin
        ...               // many statements setting element values
        FDate := EncodeDate(AYear, AMonth, ADay);
        UpdateCalendar;
        Change;           // this is new
    end;
end;

```

使用此日历组件的应用程序现在可以通过给 OnChange 事件附加处理程序来响应组件中日期的变化。

11.7.3 排除空白单元

象编写的这个日历，用户可以选择空白单元，但是，其日期不变。那么，有必要阻止选择空白单元。

要控制给定的单元是否可选，重载网格的 SelectCell 方法。

SelectCell 是一个函数，它以行、列作为参数，并返回指定单元是否可选的布尔值。

你可以重载 SelectCell，如果某单元不包含有效日期，则返回 False 值：

```

function TSampleCalendar.SelectCell(ACol, ARow: Longint): Boolean;
begin
    if DayNum(ACol, ARow) = -1 then Result := False    // -1 indicates invalid date
    else Result := inherited SelectCell(ACol, ARow); // otherwise, use inherited value
end;

```

现在，如果用户用鼠标点击空白单元，或者试图用键盘方向键移动到空白单元，此日历维持当前选择的单元不变。

第 12 章 制作数据感知控件

当用数据库连接进行工作时，有数据感知的控件经常是很方便的。即，应用程序可以在控件与数据库某部分间建立链接。Delphi 包括数据感知的标签、编辑框、列表框、组合框、查找控件和网格。你也可以制作自己的数据感知控件。有关使用数据感知控件的更详细的信息，参见《开发者指南》的第 19 章“使用数据控件”。

有几个数据感知的级别。最简单的是只读数据感知，或者数据浏览，即反映数据库当前

状态的能力。更复杂的是可编辑的数据感知，或者数据编辑，在此控件中，用户通过操作此控件，可以编辑数据库中的值。也要注意，关联数据库的级别可以改变，即从只链接单一字段的最简单的级别到象多记录控制这样的更复杂的级别的改变。

本章首先说明最简单的情况，制作一个链接到数据集中单一字段的只读控件。所用的特殊控件是在第 11 章“定制网格”中创建的 `TSampleCalendar` 日历。你也可以使用在组件面板中 `Samples` 页上的标准日历控件 `TCalendar`（只用在 VCL 中）。

然后，在本章继续解释如何使新的数据浏览控件成为数据编辑控件。

12.1 创建数据浏览控件

创建一个数据感知的日历控件，不论它是只读控件，还是用户可改变数据集中底层数据的控件，都涉及到下列步骤：

- 创建并注册组件
- 添加数据链接
- 响应数据的改变

12.1.1 创建并注册组件

创建每一个组件都是用相同的方法：创建单元、派生组件类、注册组件、编译组件、把组件安装到组件面板上。这个过程在 [1-4 页](#) 的“创建新组件”中有大概的描述。

对这个例子，按照创建组件的一般过程，但有下列特点：

- 给组件的单元起名字 `DBCAl`
- 从组件 `TSampleCalendar` 派生新的组件类 `TDBCcalendar`。第 11 章“定制网格”演示了如何创建 `TSampleCalendar` 组件。
- 在组件面板的 `Samples` 页（或 CLX 应用程序的其它页）上注册 `TDBCcalendar`。

在 VCL 应用程序中，从 `TCustomGrid` 继承的结果单元应看起来象这样：

```
unit CalSamp;
interface
uses
    Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs, Grids;
type
    TSampleCalendar = class(TCustomGrid)
    end;
procedure Register;
implementation
procedure Register;
begin
    RegisterComponents('Samples', [TSampleCalendar]);
end;
end.
```

注释 如果从 `TCustomGrid` 的 CLX 版本中继承，只有 `uses` 语句因显示 CLX 单元而不同。如果现在安装此日历组件，你会发现它显示在 `Samples` 页上。唯一可用的属性是最基本

的控件属性。下一步是使一些更特殊的属性对此日历的用户可用。

注释 虽然您可以安装只是编译过的这个示例日历组件，但是还不能试图把它放到窗体上。TCustomGrid 组件有一个抽象方法 DrawCell，它必须在实例对象被创建以前被重新声明。重载 DrawCell 方法在 11-6 页的“填充网格单元”中有描述。

注释 一些单元的名称和位置在 CLX 应用程序中有所不同。例如，在 CLX 应用程序中，Controls 是 QControls，并没有 Windows 或 Messages 单元。

最终的单元应看起来象这样：

```
unit DBCal;
interface
uses SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls,
    Forms, Grids, Calendar;
type
    TDBCcalendar = class(TSampleCalendar)
    end;
procedure Register;
implementation
procedure Register;
begin
    RegisterComponents('Samples', [TDBCcalendar]);
end;
end.
```

现在，你可以继续使这个新日历成为数据浏览器。

12.1.2 使控件只读

因为这个数据日历涉及的数据是只读的，有必要使控件本身只读，所以，用户不会在控件内部产生改变，并且希望它们在数据库中有所反映。

使此日历只读涉及到：

- 添加 ReadOnly 属性
- 允许需要的更新

注释 既然是从 Delphi 的 Samples 页而不是 TSampleCalendar 的 TCalendar 组件开始，它已经有了 ReadOnly 属性，所以你可以忽略这些步骤。

12.1.2.1 添加 ReadOnly 属性

借助添加 ReadOnly 属性，你将提供一个在设计时使控件只读的途径。当此属性被设置为 True 时，你可以使控件中的所有单元都不能被选择。

1. 添加属性声明和一个私有域来保存此值

```
type
    TDBCcalendar = class(TSampleCalendar)
    private
        FReadOnly: Boolean;           // field for internal storage
    public
```

```

        constructor Create(AOwner: TComponent); override; // must override to set default
published
        property ReadOnly: Boolean read FReadOnly write FReadOnly default True;
end;
...
constructor TDBCalendar.Create(AOwner: TComponent);
begin
    inherited Create(AOwner);          // always call the inherited constructor
    FReadOnly := True;                 // set the default value
end;

```

2. 重载 SelectCell 方法，如果控件是只读的，就不允许选择。SelectCell 的用法在 11-14 页的“排除空白单元”中有解释。

```

function TDBCalendar.SelectCell(ACol, ARow: Longint): Boolean;
begin
    if FReadOnly then Result := False    // cannot select if read only
    else Result := inherited SelectCell(ACol, ARow); // otherwise, use inherited method
end;

```

记住，给 TDBCalendar 的类型声明中添加 SelectCell 声明，并在最后加 override 指示。

如果现在把这个日历加到一个窗体上，你会发现这个组件忽略鼠标点击和键盘键击。当你改变日期时，它也不能更新选择的位置。

12.1.2.2 允许需要的更新

只读日历为了各种改变使用 SelectCell 方法，包括设置行、列属性。每次日期改变时，使用 UpdateCalendar 方法设置行、列，但是，由于 SelectCell 不允许改变，所以选择就保持在原位，甚至日期发生了改变选择也不变化。

要绕过这个对于改变的完全禁止，可以给此日历添加一个内部的布尔标记，并在此标记被设置为 True 时容许改变：

```

type
    TDBCalendar = class(TSampleCalendar)
    private
        FUpdating: Boolean;          // private flag for internal use
    protected
        function SelectCell(ACol, ARow: Longint): Boolean; override;
    public
        procedure UpdateCalendar; override; // remember the override directive
    end;
...
function TDBCalendar.SelectCell(ACol, ARow: Longint): Boolean;
begin
    if (not FUpdating) and FReadOnly then
        Result := False    // allow select if updating
    else
        Result := inherited SelectCell(ACol, ARow);    // otherwise, use inherited

```

```

        method
    end;

    procedure TDBCalenadr.UpdateCalendar;
    begin
        FUpdating := True;           // set flag to allow updates
        try
            inherited UpdateCalendar; // update as usual
        finally
            FUpdating := False;       // always clear the flag
        end;
    end;
end;

```

此日历依然不允许用户改变，但是，通过改变日期属性，现在可以正确地反映在日期中的变化。既然已有了一个真正的只读日历控件，就准备给其添加数据浏览的能力。

12.1.3 添加数据链接

在控件和数据库间的连接是通过一个被称为‘数据链接’的类进行处理的。连接控件与数据库的单一字段的数据链接类是 `TFieldDataLink`。也有对于整个表的数据链接。

数据感知控件拥有其数据链接类。即，这个控件有构建和销毁其数据链接的责任。有关自有类的管理的详细内容，参见第 10 章“创建图形控件”。

建立数据链接为自有类，需要下面 3 个步骤：

1. 声明类域
2. 声明访问属性
3. 初始化数据链接

12.1.3.1 声明类域

就象 10-6 页“声明类域”中说明的那样，组件需要为其每一个自有类分配一个域。这样，这个日历为其数据链接就需要一个 `TFieldDataLink` 类型的域。

在日历中为数据链接声明一个域：

```

type
    TDBCalenadr = class(TSampleCalendar)
    private
        FDataLink: TFieldDataLink;
    ...
    end;

```

在你可以编译应用程序以前，需要给单元的 `uses` 语句中添加 `DB` 和 `DBCtrls`。

12.1.3.2 声明访问属性

每一个数据感知控件都有一个 `DataSource` 属性，它指定应用程序中的哪一个数据源类

给此控件提供数据。另外，访问单一字段的控件需要 **DataField** 属性来指定数据源中的域。

不象在第 10 章“创建图形控件”的例子中的自有类的访问属性，这些访问属性不提供对自有类本身的访问权限，但是有对自有类中相应属性的访问权限。这就是说，你会创建使这个控件和其数据链接共享相同的数据源和域的属性。

声明 **DataSource** 和 **DataField** 属性和其实现方法，然后，给数据链接类的相应属性编写所谓“贯通法（pass-through method）”的方法。

12.1.3.3 声明访问属性的示例

声明 **DataSource** 和 **DataField** 属性和其实现方法，然后，给数据链接类的相应属性编写所谓“贯通法”的方法：

```
type
  TDBCalendar = class(TSampleCalendar)
  private
    // implementation methods are private
    ...
    function GetDataField: string; // returns the name of the data field
    function GetDataSource: TDataSource; // returns reference to the data source
    procedure SetDataField(const Value: string); // assigns name of data field
    procedure SetDataSource(Value: TDataSource); // assigns new data source
  published
    // make properties available at design time
    property DataField: string read GetDataField write SetDataField;
    property DataSource: TDataSource read GetDataSource write SetDataSource;
  end;
...
function TDBCalendar.GetDataField: string;
begin
  Result := FDataLink.FieldName;
end;

function TDBCalendar.GetDataSource: TDataSource;
begin
  Result := FDataLink.DataSource;
end;

procedure TDBCalendar.SetDataField(const Value: string);
begin
  FDataLink.FieldName := Value;
end;

procedure TDBCalendar.SetDataSource(Value: TDataSource);
begin
  FDataLink.DataSource := Value;
end;
```

既然已经在日历和其数据链接间建立了链接，还有一个更重要的步骤：当日历控件被构

造时，必须构建数据链接类，并在销毁此日历以前销毁这个数据链接。

12.1.3.4 初始化数据链接

数据感知控件需要有在它存在期间的任何时间访问其数据链接的权限，所以，必须作为自己构造函数的一部分构造此数据链接对象，并在其本身被销毁以前销毁此数据链接对象。

分别重载日历的 Create 和 Destroy 方法，构造和销毁数据链接对象：

```
type
    TDBCcalendar = class(TSampleCalendar)
    public
        // constructors and destructors are always public
        constructor Create(AOwner: TComponent); override;
        destructor Destroy; override;
        ...
    end;
...
constructor TDBCcalendar.Create(AOwner: TComponent);
begin
    inherited Create(AOwner); // always call the inherited constructor first
    FDataLink := TFieldDataLink.Create; // construct the datalink object
    FDataLink.Control := self; // let the datalink know about the calendar
    FReadOnly := True; // this is already here
end;

destructor TDBCcalendar.Destroy;
begin
    FDataLink.Free; // always destroy owned objects first
    inherited Destroy; // ... then call inherited destructor
end;
```

现在，你有了一个完整的数据链接，但是，还没有告诉这个控件它应该从这个被链接的域读取什么数据。下一节将解释如何做这个事情。

12.1.4 响应数据改变

一旦控件有了数据链接和属性来指定数据源和数据域，就需要响应域中数据的改变，这些改变或者是因为移动到不同的记录，或者是由于域内容的改变。

所有数据链接类都有名为 OnDataChange 的事件，当数据源指示数据的改变时，数据链接对象就调用附加到其 OnDataChange 事件上的任何事件处理程序。

要更新控件响应数据改变，则给数据链接的 OnDataChange 事件附加处理程序。

在这种情况下，给日历添加一个方法，然后把它指派为此数据链接的 OnDataChange 的处理程序。

声明和实现 DataChange 方法，然后在构造函数中把它分派给此数据链接的 OnDataChange 事件。在析构函数中，在销毁此对象以前，分离 OnDataChange 处理程序。

type

```

TDBCalendar = class(TSampleCalendar)
private      // this is an internal detail, so make it private
    procedure DataChange(Sender: TObject);    // must have proper parameters for event
end;

...
constructor TDBCalendar.Create(AOwner: TComponent);
begin
    inherited Create(AOwner);    // always call the inherited constructor first
    FReadOnly := True;          // this is always here
    FDataLink := TFieldDataLink.Create;    // construct the datalink object
    FDataLink.OnDataChange = DataChange;    // attach handler to event
end;

destructor TDBCalendar.Destroy;
begin
    FDataLink.OnDataChange := nil;    // detach handler before destroying object
    FDataLink.Free;                  // always destroy owned objects first
    inherited Destroy;    // ...then call inherited destructor
end;

procedure TDBCalendar.DataChange(Sender: TObject);
begin
    if FDataLink.Field = nil then    // if there is no field assigned...
        CalendarDate := 0    // ...set to invalid date
    else
        CalendarDate := FDataLink.Field.AsDateTime;    // otherwise, set calendar to the date
end;

```

现在，你就有了数据浏览控件。

12.2 创建数据编辑控件

当创建数据编辑控件时，就象为数据浏览控件做的那样，创建并注册此组件，添加数据链接。你也用类似的方法响应底层域中数据的改变，但是，还必须再处理几个事项。

例如，你可能想让你的控件既能响应键盘事件，也能响应鼠标事件。当用户改变控件的内容时，你必须响应，当用户退出控件时，你想让控件中的改变能在数据集中有所反映。

这里描述的数据编辑控件与本章第一部分描述的日历控件是一个控件，只是控件被修改了，这样，它既能编辑，也能查看其链接域中的数据。

修改这个已有的控件，使其成为一个数据编辑控件，包括：

- 改变 FReadOnly 的缺省值
- 处理鼠标按下和键盘按下消息
- 更新域数据链接类
- 修改 Change 方法
- 更新数据集

12.2.1 改变 FReadOnly 的缺省值

因为这是一个数据编辑控件，ReadOnly 属性应该缺省地被设置为 False。要使 ReadOnly 属性为 False，在构造函数中改变 FReadOnly 的值：

```
constructor TDBCcalendar.Create(AOwner: TComponent);
begin
    ...
    FReadOnly := False;    // set the default value
    ...
end;
```

12.2.2 处理鼠标按下和键盘按下消息

当控件的用户开始与控件交互时，控件就从 Windows 收到鼠标按下消息（WM_LBUTTONDOWN、WM_MBUTTONDOWN、WM_RBUTTONDOWN）或者键盘按下消息（WM_LKEYDOWN）。要使控件响应这些消息，就必须编写响应这些消息的处理程序。

- 响应鼠标按下消息
- 响应键盘按下消息

注释 如果编写 CLX 应用程序，通告是以系统事件的形式来自操作系统。有关编写响应系统和部件事件的组件方面的内容，参见 [7-10 页](#) 的“使用 CLX 响应系统通告”。

12.2.2.1 响应鼠标按下消息

MouseDown 方法是为控件的 OnMouseDown 事件的 protected 方法。控件本身调用 MouseDown，响应 Windows 的鼠标按下消息。当你重载继承的 MouseDown 方法时，可以包括一些代码。这些代码提供除调用 OnMouseDown 事件外的其它的响应。

要重载 MouseDown，给 TDBCcalendar 类添加 MouseDown 方法：

```
type
    TDBCcalendar = class(TSampleCalendar)
    ...
    protected
        procedure MouseDown(Button: TButton; Shift: TShiftState; X, Y: Integer); override;
    ...
    end;

procedure TDBCcalendar.MouseDown(Button: TButton; Shift: TShiftState; X, Y: Integer);
var
    MyMouseDown: TMouseEvent;
begin
    if not ReadOnly and FDataLink.Edit then
        inherited MouseDown(Button, Shift, X, Y)
```



```

else
begin
    MyMouseDown := OnMouseDown;
    if Assigned(MyMouseDown) then
        MyMouseDown(Self, Button, Shift, X, Y);
end;
end;

```

当 MouseDown 响应鼠标按下消息时，只有在控件的 ReadOnly 属性为 False，并且数据链接对象处于编辑模式，即此域可被编辑时，继承的 MouseDown 方法才被调用。如果这个不能编辑，程序员放在 OnMouseDown 事件处理程序中的代码（如果存在的话）就被执行。

12.2.2.2 响应键盘按下消息

KeyDown 方法是为控件的 OnKeyDown 事件的 protected 方法。控件本身调用 KeyDown 来响应 Windows 的键盘键按下消息。当重载继承的 KeyDown 方法时，你可以包括一些代码，这些代码提供除调用 OnKeyDown 事件以外的其它的响应。

要重载 KeyDown，按下列步骤：

1. 给 TDBCcalendar 类添加 KeyDown 方法：

```

type
    TDBCcalendar = class(TSampleCalendar);
    ...
protected
    procedure KeyDown(var Key: Word; Shift: TShiftState); override;
    ...
end;

```

2. 实现 KeyDown 方法：

```

procedure TDBCcalendar.KeyDown(var Key: Word; Shift: TShiftState);
var
    MyKeyDown: TKeyEvent;
begin
    if not ReadOnly and (Key in [VK_UP, VK_DOWN, VK_LEFT, VK_RIGHT,
        VK_END, VK_HOME, VK_PRIOR, VK_NEXT])
        and FDataLink.Edit then
        inherited KeyDown(Key, Shift)
    else
    begin
        MyKeyDown := OnKeyDown;
        if Assigned(MyKeyDown) then
            MyKeyDown(Self, Key, Shift);
    end;
end;

```

当 KeyDown 响应鼠标按下消息时，只有在控件的 ReadOnly 属性为 False，并且按下的键是光标控制键之一，同时数据链接对象处于编辑模式，即域可被编辑时，继承的 KeyDown

方法才被调用。如果域不能被编辑，或是其它一些键被按下，则程序员放在 OnKeyDown 事件处理程序中的代码（如果存在的话）就被执行。

12.2.3 更新域数据链接类

有两种类型的数据改变：

- 必须被反映在数据感知控件中的域值的改变
- 必须被反映在域值中的数据感知控件的改变

TDBCalendar 组件已经有了 DataChange 方法，它通过给 CalendarDate 属性赋值来处理数据集中字段值的改变。DataChange 方法是为 OnDataChange 事件的处理程序，所以，日历组件可以处理第一种类型的数据改变。

同样地，域数据链接类也有 OnUpdateData 事件，它是在控件的用户修改数据感知控件的内容时发生的。日历控件有 UpdateData 方法，它成为 OnUpdateData 事件的处理程序。UpdateData 给域数据链接分配数据感知控件中变化的值。

1. 要在域值中反映日历中值的改变，给日历组件的 private 区添加 UpdateData 方法：

```
type
    TDBCalendar = class(TSampleCalendar);
private
    procedure UpdateData(Sender: TObject);
    ...
end;
```

2. 实现 UpdateData 方法：

```
procedure TDBCalendar.UpdateData(Sender: TObject);
begin
    FDataLink.Field.AsDateTime := CalendarDate; //set field link to calendar date
end;
```

3. 在 TDBCalendar 的构造函数中，给 OnUpdateData 事件分派 UpdateData 方法：

```
constructor TDBCalendar.Create(AOwner: TComponent);
begin
    inherited Create(AOwner);
    FReadOnly := True;
    FDataLink := TFieldDataLink.Create;
    FDataLink.OnDataChange := DataChange;
    FDataLink.OnUpdateData := UpdateData;
end;
```

12.2.4 修改 Change 方法

TDBCalendar 的 Change 方法只有在设置新的日期值时才被调用。Change 调用 OnChange 事件处理程序（如果存在的话）。组件用户可以在 OnChange 事件处理程序中编写代码来响应日期的改变。

当日历的日期改变时，应通知底层的数据集改变已经发生了。你可以通过重载 Change 方法并增加多行代码的方式来达到此目的。这些步骤如下：

1. 给 TDBCcalendar 组件添加新的 Change 方法:

```
type
    TDBCcalendar = class(TSampleCalendar);
private
    procedure Change; override;
    ...
end;
```

2. 编写 Change 方法, 在其中调用 Modified 方法, 它通知数据集数据已经改变, 然后, 调用继承的 Change 方法:

```
procedure TDBCcalendar.Change;
begin
    FDataLink.Modified;           // call the modified method
    inherited Change;              // call the inherited Change method
end;
```

12.2.5 更新数据集

迄今为止, 数据感知控件内的变化已经改变了域数据链接类中的值。创建数据编辑控件的最后一步是用新值更新数据集。这应该在正改变数据感知控件中值的用户通过在控件外点击鼠标或者按下 Tab 键离开控件以后发生。这个过程在 VCL 和 CLX 应用程序间有不同的操作。

注释 VCL 应用程序为有关控件的操作定义消息控件 ID, 比如, 当用户离开控件时, 给此控件发送 CM_EXIT 消息。你可以编写响应此消息的消息处理程序, 这样, 当用户离开控件时, 为 CM_EXIT 编写的消息处理程序, 即 CMExit 方法, 通过用域数据链接类中改变后的值更新数据集的记录来作出响应。关于消息处理程序的更多信息, 参见第 7 章“处理消息和系统通告”。

要在消息处理程序内更新数据集, 按以下步骤:

1. 给 TDBCcalendar 组件添加消息处理程序:

```
type
    TDBCcalendar = class(TSampleCalendar);
private
    procedure CMExit(var Message: TWMNoParams); message CM_EXIT;
    ...
end;
```

2. 实现 CMExit 方法, 看起来象这样:

```
procedure TDBCcalendar.CMExit(var Message: TWMNoParams);
begin
    try
        FDataLink.UpdateRecord; // tell data link to update database
    except
        on Exception do SetFocus; // if it failed, don't let focus leave
    end;
    inherited;
end;
```

注释 在 CLX 应用程序中，TWidgetControl 有一个 protected DoExit 方法，它在输入焦点从控件移走时被调用。这个方法为 OnExit 事件调用事件处理程序。你可以在产生 OnExit 事件处理程序以前重载这个方法来更新数据集中的记录。

当用户离开控件时，要更新数据集，执行下列步骤：

1. 给 TDBCalendar 组件添加对 DoExit 方法的重载

```
type
    TDBCalendar = class(TSampleCalendar);
private
    procedure DoExit; override;
    ...
end;
```

2. 实现 DoExit 方法，它看起来如下：

```
procedure TDBCalendar.CMExit(var Message: TWMNoParams);
begin
    try
        FDataLink.UpdateRecord;           // tell data link to updfate database
    except
        on Exception do SetFocus;         // if it failed, don't let focus leave
    end;
    inherited;
end;
```

第 13 章 使对话框成为组件

你会发现，使被频繁使用的对话框成为添加到组件面板上的组件是很方便的。你的对话框组件会象代表标准普通对话框的组件一样工作。目的是创建一个简单的组件，用户可把它添加到工程中，并在设计时设置属性。

使对话框成为组件需要这些步骤：

1. 定义组件接口
2. 创建并注册组件
3. 创建组件接口
4. 测试组件

与对话框结合的 Delphi 的“包装器 (wrapper)”组件在运行时创建并运行对话框，传递用户指定的数据。因此，这个对话框组件可以重用，并且可定制。

在本章，你将看到，如何创建围绕 Delphi 对象库中提供的一般 About Box 窗体的包装器组件。

注释 把 ABOUT.PAS 和 ABOUT.DFM 文件复制到你的工作目录。

为设计一个将被包装进组件的对话框没有很多特别的考虑，几乎任何窗体在此情况下都可当作对话框进行操作。

13.1 定义组件接口

在可以为对话框创建组件以前，需要决定你想让开发人员如何使用它。在对话框和使用此对话框的应用程序间创建一个接口。

例如，考虑为普通对话框组件的属性。它们能使开发人员设置对话框的初始状态，比如标题和初始的控件设定，然后，在对话框关闭以后读回任何需要的信息。没有与对话框中的各控件有直接的交互，只是与包装器组件中的属性有交互。

因此，这个接口必须包含足够的信息，这样，对话框窗体可以按开发人员指定的方式显示，并返回任何应用程序需要的信息。你可以把包装器组件中的属性看成瞬时对话框的持久稳定数据。

就 About box 来说，不需要返回任何信息，所以，只有包装器的属性必须包含需要合理地显示 About box 的信息。因为在 About box 中有应用程序或许会影响的 4 个独立的域，所以你要为它们提供 4 个串类型的属性。

13.2 创建并注册组件

每一个组件的创建都以相同的方式开始：创建单元、派生组件类、注册组件、编译组件、安装组件于组件面板上。这个过程在 [1-8 页](#) 的“创建新组件”中有大概的描述。

对于这个例子，遵循创建组件的一般程序，但有下列特点：

- 给组件的单元起名字为 AboutDlg
- 从 TComponent 派生新的组件类型 TAboutDlg
- 在组件面板的 Samples 页上注册 TAboutDlg

结果单元将看起来如下：

```
unit AboutDlg;
interface
uses
    SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls, Forms;
type
    TAboutBoxDlg = class(TComponent)
    end;
procedure Register;
implementation
procedure Register;
begin
    RegisterComponents('Samples', [TAboutBoxDlg]);
end;
end.
```

注释 一些单元的名字和位置在 CLX 应用程序有所不同。比如，Controls 单元在 CLX 应用程序中是 QControls。

现在，新组件只有 TComponent 中内置的功能，它是最简单的非可视组件。在下一节，将创建组件与对话框间的接口。

13.3 创建组件接口

这些是创建组件接口的步骤：

1. 包括窗体单元
2. 添加接口属性
3. 添加 Execute 方法

13.3.1 包括窗体单元

要为包装器组件初始化并显示被包裹的对话框，必须给包装器组件单元的 `uses` 语句添加此窗体的单元。

添加 `About` 到 `AboutDlg` 单元的 `uses` 语句后。

现在，`uses` 语句现在看起来如下：

```
uses
    Windows, SysUtils, Messages, WinTypes, WinProcs, Messages, Classes, Graphics,
    Controls, Forms, About;
```

窗体单元总是声明窗体类的实例，对 `About box` 来说，窗体类是 `TAboutBox`，`About` 单元包括下列内容：

```
var
    AboutBox: TAboutBox;
```

因此，通过给 `uses` 语句添加 `About`，使 `AboutBox` 可用于包装器组件。

13.3.2 添加接口属性

在继续进行以前，确定包装器需要的属性，使开发人员在其应用程序中能够把此对话框用作组件。然后，可以给组件的类声明添加这些属性的声明。

包装器组件中的属性比编写规则的组件所创建的属性稍有些简单。记住，这时你只是创建包装器可以来回传给对话框的一些永久数据，你能使开发人员在设计时设置数据，所以，在运行时包裹器能把它们传递到对话框。

声明接口属性需要给组件类声明中添加二个声明：

- 一个 `private` 类域，它是包装器用于保存属性值的变量
- 发布的属性声明本身，它指定属性名，并告诉它哪一个域用于保存

这种接口属性不需要访问方法。它们使用对其保存数据的直接访问。按约定，保存属性值的类域有与此属性相同的名称，但是在前面有字母 `F`。域和属性必须是同一类型。

例如，要声明整型接口属性 `Year`，做如下声明：

```
type
    TMyWrapper = class(TComponent)
    private
        FYear: Integer;           // field to hold the Year-property data
    published
        property Year: Integer read FYear write FYear; // property matched with storage
    end;
```

对于这个 About box，需要 4 个串类型的属性，分别表示：产品名称、版本信息、版权信息、任何的说明。

```
type
    TAboutBoxDlg = class(TComponent)
    private
        FProductName, FVersion, FCopyright, FComments: string; // declare fields
    published
        property ProductName: string read FProductName write FProductName;
        property Version: string read FVersion write FVersion;
        property Copyright: string read FCopyright write FCopyright;
        property Comments: string read FComments write FComments;
    end;
```

当把这个组件安装到组件面板板中，并把它放在一个窗体上时，你就可以设置其属性，并且其属性值将自动与窗体保存在一起。当运行被包裹的对话框时，包装器可以使用这些值。

13.3.3 添加 Execute 方法

组件接口的最后部分是打开对话框并在它关闭时返回结果的途径。就象普通对话框组件一样，使用布尔函数 Execute，它在用户点击‘确认’时返回 True，在用户撤消对话框时返回 False。

Execute 方法的声明看起来总是如下的样子：

```
type
    TMyWrapper = class(TComponent)
    public
        function Execute: Boolean;
    end;
```

对 Execute 的最小实现需要构造对话框窗体，把它显示为模态对话框，根据 ShowModal 的返回值，返回 True 或 False。

这里是 TMyDialogBox 类型的对话框窗体的最简单 Execute 方法：

```
function TMyWrapper.Execute: Boolean;
begin
    DialogBox := TMyDialogBox.Create(Application); // construct the form
    try
        Result := (DialogBox.ShowModal = IDOK); // execute; set result based on how closed
    finally
        DialogBox.Free; // dispose of the form
    end;
end;
```

注意 try..finally 块的使用，确保应用程序释放对话框对象，甚至在异常发生时。一般，每当用这种方法构建对象时，都应该使用 try..finally 块来保护代码块，并确保应用程序释放它所分配的任何资源。

实际上，在 try..finally 块中会有更多的代码，尤其在调用 ShowModal 以前，包装器将设置一些对话框的属性，它们都基于包装器组件的接口属性。在 ShowModal 返回以后，包装器可能会设置基于对话框运行结果的一些接口属性。

对于 About box，需要使用包装器组件的 4 个接口属性来设置 About box 窗体中标签的内容。因为 About box 不给应用程序返回任何信息，所以在调用 ShowModal 以后不需要做任何事情。编写 About box 包装器的 Execute 方法，如下。

在 TAboutDlg 类的 public 部分，为 Execute 方法添加声明：

```
type
    TAboutDlg = class(TComponent)
public
    function Execute: Boolean;
end;

function TAboutBoxDlg.Execute: Boolean;
begin
    AboutBox := TAboutBox.Create(Application);           //construct About box
    try
        if ProductName = '' then           // if product name's left blank...
            ProductName := Application.Title; // ...use application title instead
        AboutBox.ProductName.Caption := ProductName;    // copy product name
        AboutBox.Version.Caption := Version;           // copy version info
        AboutBox.Copyright.Caption := Copyright;       // copy copyright info
        AboutBox.Comments.Caption := Comments;         // copy comments
        AboutBox.Caption := 'About ' + ProductName; // set About-box caption
        with AboutBox do begin
            ProgramIcon.Picture.Graphic := Application.Icon; // copy icon
            Result := (ShowModal = IDOK);           // execute and set result
        end;
    finally
        AboutBox.Free;           // dispose of About box
    end;
end;
```

13.4 测试组件

一旦你安装了对话框组件，把它放在窗体上，并运行，就可以象使用任何普通对话框一样使用它们。测试 About box 的一个快速方法是给窗体添加命令钮，当用户点击此按钮时运行这个对话框。

例如，如果你创建了一个 About 对话框，把其做成组件，并把它添加到组件面板上，就可用下面的步骤进行测试：

1. 创建新工程
2. 把 About box 组件放在主窗体中
3. 在窗体上放一个命令按钮
4. 鼠标双击命令按钮，创建一个空的点击事件处理程序
5. 在点击事件处理程序中，输入下行代码：
AboutBoxDlg1.Execute;

6. 运行应用程序

当主窗体显示时，点击命令按钮。About box 显示出来，带有缺省的工程图标和名称 Project1。选择‘确认’按钮，关闭对话框。

你可以通过设置此 About box 组件的各种属性并再次运行应用程序来对此组件做进一步的测试。

第 14 章 扩充 IDE

你可以通过使用 Open Tools API（经常缩写为 Tools API），用你自己的菜单项、工具条按钮、动态窗体创建向导等等来扩充和定制 IDE。Tools API 是一套超过 100 个的接口，它们与 IDE 交互并控制 IDE，包括主菜单、工具条、主动作列表、图象列表、源代码编辑器的内部缓存、键盘宏和绑定、窗体和在窗体编辑器中的组件、调试器和被调试进程、代码完成、消息浏览和行事列表（To-Do list）。

使用 Tools API，简单地就是编写实现特定接口的类，访问由其它接口提供的服务。你的 Tools API 代码必须在设计时被编译，并作为设计时包或以 DLL 的形式被装载进 IDE 中。这样，编写一个 Tools API 扩展有点象编写一个属性或者组件编辑器。在着手本章的学习以前，确信你熟悉使用包（《开发者指南》的第 16 章“使用包和组件”）和注册组件（第 8 章“制作在设计时可用的组件”）的基础。

本章包括下列主题：

- Tools API 概述
- 编写向导类
- 获取 Tools API 服务
- 用文件和编辑器
- 创建窗体和工程
- 给向导通报 IDE 事件

14.1 Tools API 概述

所有的 Tools API 声明都放在一个单元 ToolsAPI 中。要使用 Tools API，特别需要使用 designide 包，也就是说，作为设计时的包，或者作为使用运行时包的 DLL，你必须构建自己的 Tools API 插件（add-in）。有关包和库发行的有关信息，参见 14-4 页的“安装向导包”。

为编写 Tools API 扩展的主要接口是 IOTAWizard，所以，大部分 IDE 附加件被称为向导。C++ Builder 和 Delphi 向导，大部分都是可互操作的。你可以在 Delphi 中编写并编译一个向导，然后在 C++ Builder 中使用它。反之亦然。互操作性对于同一版本的（C++ Builder 和 Delphi）效果最好，但是也可以编写向导在二个产品的更高版本中使用。要使用 Tools API，就要编写向导类，它实现定义在 ToolsAPI 单元中的一个或者多个接口。向导利用了 Tools API 提供的服务。每一个服务都是提供了一套相关函数的一个接口。接口的实现被隐藏于 IDE 内部。Tools API 只发布接口，你可以使用这些接口来编写向导，而不用关心接口的实现。各种服务提供对源代码编辑器、窗体设计器、调试器等的访问。14-5 页的“获取 Tools API 服务”节会深入地讨论这个主题。

服务和其它接口归入两个基本分类中，你可以借助用于类型名的前缀把它们分辨开来。

- NTA (Native tools API) 授权对实际的 IDE 对象的直接访问，比如，IDE 的 TMainMenu 对象。当使用这些接口时，向导必须使用 Borland 包，这也意味着向导被绑定到特定版本的 IDE 上。向导可以驻留在设计时包中，或者驻留在使用运行时包的 DLL 中。
- OTA (open tools API) 不需要包，只通过接口访问 IDE。理论上，倘若你也能用 Delphi 调用约定和 Delphi 类型（比如 AnsiString）工作，就能用支持 COM 风格接口的任何语言编写向导。OTA 接口不授权对 IDE 的完全访问，但是，Tools API 几乎所有的功能性都通过 OTA 接口而可用。如果向导只使用 OTA 接口，就有可能编写不依赖于 IDE 特定版本的 DLL。

Tools API 有二种接口：你（即程序员）必须实现的接口和 IDE 实现的接口。大部分接口是属于后一种：接口定义 IDE 的能力，但是隐藏了实现细节。你必须实现的接口的种类分为三类：向导、通告者（notifiers）、创建者（creators）。

- 如在本节前面提过的，向导类实现 IOTAWizard 接口和有可能派生的接口。
- 通告者是 Tools API 中的另一种接口。当某些有趣的事发生时，IDE 使用通告者来回调你的向导。你编写实现通告者接口的类，用 Tools API 注册通告者，并当用户打开文件、编辑源代码、修改窗体、开始调试任务等的时候，IDE 回调你的通告者对象。通告者在 14-15 页的“给向导通知 IDE 事件”中有论述。
- 创建者是你必须实现的另一种接口。Tools API 使用创建者来创建新单元、工程、或其它文件、或者打开已存在的文件。14-12 页的“创建窗体和工程”一节会更深入地讨论这个主题。

其它重要的接口是模块（modules）和编辑器（editors）。模块接口代表打开的单元，它含有一个或多个文件。编辑器接口代表打开的文件。不同种类的编辑器接口给你访问 IDE 的不同部分的权限：源代码编辑器用于源文件，窗体设计器用于窗体文件，工程资源用于资源文件。14-10 页的“用文件和编辑器工作”一节含盖更深入的对这些主题的讨论。

以下各节带你领略编写向导的步骤。就每一接口的完整细节，参见在线帮助文件。

14.2 编写向导类

有四种向导，根据向导类所实现接口的不同而不同。表 14.1 描述这四种向导。

表 14.1 四种向导

接口	说明
IOTAFormWizard	典型地创建一个新单元、窗体或其它文件
IOTAMenuWizard	自动地给帮助添加菜单
ITOAProjectWizard	典型地创建新应用程序或其它工程
ITOAWizard	不能归入其它类的混合向导

这四种向导只在用户如何调用向导方面有所不同。

- 菜单向导被添加到 IDE 的帮助菜单上。当用户选取菜单项时，IDE 就调用这个向导的 Execute 函数。简单的向导提供更多的灵活性，所以，菜单向导典型地被只用于原型和调试。
- 窗体和工程向导被称为仓库向导，是由于它们都驻留在对象仓库中。用户从‘新项目’对话框中调用这些向导。用户也可以在对象仓库中（借助选择‘工具 | 仓库’

菜单项)看到这些向导。用户可为窗体向导检查‘新窗体’核选框,当用户选择‘文件|新窗体’菜单项时,它告诉 IDE 去调用窗体向导。用户也可以检查‘主窗体’核选框,这告诉 IDE,对新的应用程序,把这个窗体向导当作缺省窗体使用。用户为工程向导可以检查‘新工程’核选框,当用户选择‘文件|新应用程序’时,IDE 调用选择的工程向导。

- 第四种向导是不能归入其它分类的情形。简单的向导不会自动地或独立地做任何事情,所以,你必须定义这个向导如何被调用。

Tools API 不给向导强加任何限制,比如,需要工程向导创建工程。你可以非常容易地编写工程向导来创建窗体,编写窗体向导来创建工程(如果这事是你确实想做的的话)。

14.2.1 实现向导接口

每一个向导类最少必须实现 IOTAWizard,它也需要实现它的祖先: IOTANotifier 和 IInterface。窗体和工程向导必须实现它们全部的祖先接口,即 ITOARepositoryWizard、IOTAWizard、ITOANotifier 和 IInterface。

IInterface 的实现必须遵循 Delphi 接口的一般规则,这些规则与 COM 接口的规则一样。就是: QueryInterface 完成类型转换, _AddRef 和 _Release 管理引用计数。你可能想使用一个通用基类来简化编写向导和通告者类。为了这个目的, ToolsAPI 单元定义类 TNotifierObject,它用空的方法体(empty method bodies)实现 IOTANotifier 接口。

尽管向导从 IOTANotifier 继承,并因此必须实现其全部函数,但是 IDE 通常不使用这些函数,所以你的实现可以是空的(就象它们在 TNotifierObject 中一样)。因此,当你编写自己的向导类时,只需要声明和实现这些由向导接口引入的接口方法,接受 IOTANotifier 的 TNotifierObject 实现。

14.2.2 安装向导包

就象对待任何其它的设计时包一样,向导包必须有一个 Register 函数(有关 Register 函数的更详细的说明,参见第 8 章“制作在设计时可用的组件”)。在 Register 函数中,通过调用 RegisterPackageWizard,并把向导对象作为其唯一的变量传递给它,可以注册任何数量的向导,如下所示:

```
procedure Register;  
begin  
    RegisterPackageWizard(MyWizard.Create);  
    RegisterPackageWizard(MyOtherWizard.Create);  
end;
```

在同一包中,你也可以注册属性编辑器、组件,等。

记住,设计时包是 Delphi 主应用程序的部分,这意味着任何窗体名称都必须在整个应用程序和所有其它的设计时包中是唯一的。这是使用包的主要优点:你从不知道任何其它人可能给其窗体的命名。

在开发过程中,安装向导包的方法与任何其它设计时包一样:点击包管理器中的‘安装’按钮。IDE 将编译和链接包,并试图装载它。IDE 显示一个对话框,它告诉你是否成功地装载了这个包。

14.3 获取 Tools API 服务

要做任何有用的事，向导需要有权使用 IDE：它的编辑器、窗口、菜单，等。这就是服务接口的任务。Tools API 包括很多服务，比如执行文件动作的动作服务、访问源代码编辑器的编辑器服务、访问调试器的调试器服务，等。表 14.2 总结了所有的服务接口。

表 14.2 Tools API 服务接口

接口	说明
INTAServices	提供对固有的 IDE 对象的访问：主菜单、动作列表、图象列表、工具条
IOTAActionServices	执行基本的文件动作：打开、关闭、保存、重新装载文件
IOTACodeCompletionServices	提供对代码完成的使用，容许向导安装定制代码完成管理器
IOTADebuggerServices	提供对调试器的使用
IOTAEditorServices	提供对源编辑器及其内部缓存的访问
IOTAKeyBindingServices	容许向导注册定制的绑定
IOTAKeyboardServices	提供对键盘宏和绑定的访问
IOTAKeyboardDiagnostics	开关对键盘键击的调试
IOTAMessageServices	提供对消息视图的访问
IOTAModuleServices	提供对打开文件的访问
IOTAPackageServices	查询所有安装包及其组件的名称
IOTAServices	杂项服务
IOTAToDoService	提供对行事列表的访问，容许向导安装定制的行事列表管理器
IOTAToolsFilter	注册工具过滤器通告者
IOTAWizardServices	注册和卸载注册向导

要使用服务接口，使用定义在 SysUtils 单元中的全局函数 Supports，转换 BorlandIDEServices 变量为希望的服务。例如：

```
procedure set_keystroke_debugging(debugging: Boolean);
var
    diag: IOTAKeyboardDiagnostics;
begin
    if Supports(BorlandIDEServices, IOTAKeyboardDiagnostics, diag) then
        diag.KeyTracing := debugging;
    end;
```

如果向导需要经常使用特定的服务，可以保存一个指向此服务的指针，作为你的向导类的数据成员。

14.3.1 使用固有的 IDE 对象

向导有对主菜单、工具条、动作列表和 IDE 的图象列表的完全访问的权限（注意，IDE 的很多上下文菜单是不能通过 Tools API 访问的）。本节给出一个简单的示例，说明向导可以如何使用这些固有的 IDE 对象来与 IDE 交互。

14.3.1.1 使用 INTAServices 接口

用固有的 IDE 对象工作的出发点是 INTAServices 接口。使用这个接口来给图象列表添加图象，给动作列表添加动作，给主菜单添加菜单项，给工具条添加按钮。可以把这个动作绑定到菜单项和工具按钮上。当向导被销毁时，就必须清除它创建的对象，但是，不必删除添加给图象列表的图象。删除图象将弄乱在使用这个向导以后所有添加图象的索引。

向导使用来自 IDE 的实际的 TMainMenu、TActionList、TImageList 和 TToolBar 对象，所以，可以使用与任何其它应用程序一样的方法来编写代码。这也意味着你有损毁 IDE 或者取消重要特征的许多机会，比如删除‘文件’菜单。

14.3.1.2 给图象列表添加图象

假设你想添加菜单项来调用向导，也想让用户能够添加调用此向导的工具条按钮，那么第一步先给 IDE 的图象列表添加图象，然后图象的索引就可被用于动作，它们依次被菜单项和工具条按钮使用。使用‘图象编辑器’，创建包含 16×16 位图资源的资源文件。给向导的构造函数添加下列代码：

```
constructor MyWizard.Create;
var
  Services: INTAServices;
  Bmp: TBitmap;
  ImageIndex: Integer;
begin
  inherited;
  Supports(BorlandIDEServices, INTAServices, Services);
  // add an image to the image list
  Bmp := TBitmap.Create;
  Bmp.LoadFromResourceName(HInstance, 'Bitmap1');
  ImageIndex := Services.AddMasked(Bmp, Bmp.TransparentColor,
                                   'Tempest Software.into wizard image');
  Bmp.Free;
end;
```

确保由在资源文件中指定的名称或者 ID 来装载资源。必须选择一个颜色，它将作为图象的背景色。如果不想要背景色，就选择一个在位图中不存在的颜色。

14.3.1.3 给动作列表添加动作

图象索引被常用于创建动作，如下所示。向导使用 OnExecute 和 OnUpdate 事件。一个共同的方案是对向导使用 OnUpdate 事件来使能或者取消这个动作。要确保 OnUpdate 事件快速返回，否则，用户会注意到在装载向导以后 IDE 变得行动迟缓。动作的 OnExecute 事件与向导的 Execute 方法类似。如果你正使用菜单项来调用窗体或工程向导，你可能甚至想让 OnExecute 直接调用 Execute。

```
NewAction := TAction.Create(nil);
```

```

NewAction.ActionList := Services.ActionList;
NewAction.Caption := GetMenuText();
NewAction.Hint := 'Display a silly dialog box';
NewAction.ImageIndex := ImageIndex;
NewAction.OnUpdate := action_update;
NewAction.OnExecute := action_execute;

```

菜单项给新创建的动作设定其‘动作’属性。创建菜单项时棘手的部分是要知道把它插在哪里。下面的示例查找‘查看’菜单，并把新菜单项插入到‘查看’菜单的第一项。（一般，按绝对位置不是一个好的想法：你绝不会知道什么时候有哪一个菜单项会插入菜单。Delphi 以后的版本也可能重新排序菜单。一个更好的方法是用指定名称在菜单中查找菜单项。为清楚起见，在下面显示简化了的方法）

```

for i := 0 to Services.MainMenu.Items.Count - 1 do
begin
    with Services.MainMenu.Items[i] do
    begin
        if CompareText(Name, 'ViewMenu') = 0 then
        begin
           NewItem := TMenuItem.Create(nil);
           NewItem.Action := NewAction;
            Insert(0, NewItem);
        end;
    end;
end;

```

由于给 IDE 的动作列表中添加了动作，当定制工具条时，用户就可以看到动作。用户可以选择动作，并作为按钮把它添加到任何工具条中。这在向导被卸载时会引起问题：所有的工具按钮最后都带有悬空的指针，它们指向不存在的动作和不存在的 OnClick 事件处理程序。要禁止访问冲突，向导必须找到全部关联动作的工具按钮，并删除这些按钮。

14.3.1.4 删除工具条按钮

没有从工具条删除按钮的简便函数，你必须发送 CM_CONTROLCHANGE 消息，其中，第一个参数是要改变的控件；第二个参数如果是 0，从工具条删除按钮，如果是非 0 值，则添加按钮到工具条。删除了工具条上的按钮以后，由析构函数删除动作和菜单项。在删除这些项的同时，会自动从 IDE 的动作列表（ActionList）和主菜单（MainMenu）中把它们删除掉。

```

procedure remove_action(Action: TAction; ToolBar: TToolBar);
var
    i: Integer;
    Btn: TToolButton;
begin
    for i := ToolBar.ButtonCount - 1 downto 0 do
    begin
        Btn := ToolBar.Buttons[i];
        if Btn.Action = Action then

```

```

        begin
            // Remove 'Btn' from 'ToolBar'
            ToolBar.Perform(CM_CONTROLCHANGE, WPARAM(Btn), 0);
            Btn.Free;
        end;
    end;
end;

destructor MyWizard.Destroy;
var
    Services: INTAServices;
    Btn: TToolButton;
begin
    Supports(BorlandIDEServices, INTAServices, Services);
    // check all the toolbars, and remove any buttons that use this action
    remove_action(NewAction, Services.ToolBar[sCustomToolBar]);
    remove_action(NewAction, Services.ToolBar[sDesktopToolBar]);
    remove_action(NewAction, Services.ToolBar[sStandardToolBar]);
    remove_action(NewAction, Services.ToolBar[sDebugToolBar]);
    remove_action(NewAction, Services.ToolBar[sViewToolBar]);
    remove_action(NewAction, Services.ToolBar[sInternetToolBar]);

    NewItem.Free;
    NewAction.Free;
end;

```

正如你会从这个简单的例子中看到的，你在向导如何与 IDE 交互方面有很多灵活性。但是，这个灵活性伴随着责任，它很容易造成悬空指针或者其它的访问冲突。下一节介绍如何帮助诊断这些问题的一些小技巧。

14.3.2 调试向导

当编写使用 native Tools API 的向导时，你可能会编写出引起 IDE 崩溃的代码。也有可能你编写的向导可以安装但是不能按你所想进行动作。用设计时代码工作的一个挑战是调试，但是它是一个容易解决的问题。因为向导被安装于 Delphi 本身，你只不过需要从‘运行 | 参数...’菜单项给 Delphi 执行程序（delphi32.exe）设置包的主应用程序。

当你想（或需要）调试包的时候，不用安装它，而是从菜单条中选择‘运行 | 运行’，这就启动了一个新的 Delphi 实例。在这个新的实例中，通过从菜单条中选择‘组件 | 安装包...’，安装已经编译过的包。返回到 Delphi 最初的实例，现在你会看到**蓝点指示**（telltale blue dots），它们告诉你可以向导的源代码中的哪里设置断点。（如果没有看到，仔细检查你的编译选项，确定你能够调试；确保你已装载了正确的包；仔细检查进程模块，更加确信你已经装载了你想装载的.bpl 文件）

你不能用这个方法调试进入 VCL、CLX 或 RTL 代码，但是你有全面调试向导本身的能力，它足以告诉你什么正在出错。

14.3.3 接口的版本号

如果你仔细地留意一些接口的声明，比如 `IOTAMessageServices`，你会看到它们从有相似名称的其它接口继承，如 `IOTAMessageServices50`，它继承自 `IOTAMessageServices40`。这个版本号的使用，帮助你使你的代码不受 Delphi 版本间变化的影响。

Tools API 遵循 COM 的基本原理，即接口和其 GUID 从不改变。如果一个新版本给接口添加特征，Tools API 就声明一个继承自旧接口的新接口。原 GUID 保持不变，它连接到旧的、没有变化的接口。新接口获得一个全新的 GUID。使用旧 GUID 的旧向导继续工作。

Tools API 也改变接口名称，以设法保护源代码的兼容性。要明白此怎样工作，在 Tools API 中区分二种不同的接口是很重要的：Borland 实现的接口和用户实现的接口。如果 IDE 实现接口，名称就与接口的最新版本放在一起。新功能不影响已有代码。老的接口有旧版本号跟随。

但是，对用户实现的接口，在基接口中的新成员函数需要用你的代码实现新函数。因此，名称趋于用旧接口名，新接口在名称的尾部附加版本号。

例如，考虑消息服务，Delphi 6 引入了一个新特征：消息组。因此，基本消息服务接口需要新的成员函数，这些函数被声明在新的接口类中，它保留名称 `IOTAMessageServices`。旧的消息服务接口被重命名为 `IOTAMessageServices50`（对版本 5）。因为成员函数相同，所以旧 `IOTAMessageServices` 的 GUID 与新 `IOTAMessageServices50` 的 GUID 相同。

作为用户实现接口的一个示例，考虑 `IOTAIDENotifier`。Delphi 5 添加了新的重载函数：`AfterCompile` 和 `BeforeCompile`。使用了 `IOTAIDENotifier` 的原有代码不需要改变，但是需要新功能的新代码必须被修改来重载从 `IOTAIDENotifier50` 继承的新函数。版本 6 没有添加更多的函数，所以要用的当前版本是 `IOTAIDENotifier50`。

经验法则是：当编写新代码时，使用最底层的（most-derived）类。如果只是在新版本下编译原有向导时，就不用管源代码。

14.4 使用文件和编辑器工作

在进一步学习以前，需要理解 Tools API 如何用文件工作。主要接口是 `IOTAModule`。一个模块代表一组逻辑关联的打开文件。例如，一个单独的模块代表一个单独的单元。每个模块有一个或多个编辑器，其中，一个编辑器代表一个文件，比如单元的源代码文件（.pas）或窗体文件（.dfm 或 .xfm）。编辑器接口影响 IDE 编辑器的内部状态，所以，向导可以知道用户看到的修改过的代码和窗体，甚至在用户没有保存任何改变时也会知道。

14.4.1 使用模块接口

要获得模块接口，从模块服务（`IOTAModuleServices`）开始。你可以在模块服务中查询所有打开的模块，从文件名或窗体名称中查找，或者打开文件来获得其模块接口。

不同种类的文件（如工程、资源、类型库）有不同种类的模块。把模块接口转变成特定种类的模块接口，了解此模块是否是这种类型。例如，获取当前工程组接口的一个方法如下：

```
// Return the current project group, or nil if there is no project group
function CurrentProjectGroup: IOTAProjectGroup;
```



```

var
    i: Integer;
    Svc: IOTAModuleServices;
    Module: IOTAModule;
begin
    Supports(BorlandIDEServices, IOTAModuleServices, Svc);
    for i := 0 to Svc.ModuleCount - 1 do
        begin
            Module := Svc.Modules[i];
            if Supports(Module, IOTAProjectGroup, Result) then
                Exit;
        end;
        Result := nil;
    end;
end;

```

14.4.2 使用编辑器接口

每一个模块都至少有一个编辑器接口。有些模块还有几个编辑器，比如，源文件(.pas)和窗体描述文件(.dfm)。所有编辑器都实现 `IOTAEditor` 接口。把编辑器转换为某特定类型的接口，了解它是哪种编辑器，比如，要为单元获得窗体编辑器接口，可以做如下事情：

```

// Return the form editor for a module, or nil if the unit has no form
function GetFormEditor(Module: IOTAModule): IOTAFormEditor;
var
    i: Integer;
    Editor: IOTAEditor;
begin
    for i := 0 to Module.ModuleFileCount - 1 do
        begin
            Editor := Module.ModuleFileEditors[i];
            if Supports(Editor, IOTAFormEditor, Result) then
                Exit;
        end;
        Result := nil;
    end;
end;

```

编辑器接口给你访问编辑器内部状态的权限。你可以检查用户正编辑的源代码或组件，改变源代码、组件或属性，改变源和窗体编辑器的选择，并执行最终用户可执行的几乎任何编辑器动作。

由于使用窗体编辑器接口，向导可以访问窗体上的所有组件。每一个组件（包括根窗体或者数据模块）各自有一个关联的 `IOTAComponent` 接口。向导可以检查或改变大部分组件的属性。如果你需要完全控制组件，可以把 `IOTAComponent` 接口转变为 `INTAComponent`。固有的组件接口（native component interface）能使你的向导直接访问 `TComponent` 指针。如果你需要读取或修改类类型的属性，比如 `TFont`，它只能通过 `NTA` 样式的接口读取或修改，这是很重要的。

14.5 创建窗体和工程

Delphi 带有大量已安装的窗体和工程向导，并且你也可以编写自己的窗体和工程向导。‘对象仓库’让你创建可用于工程的静态模板，但是向导提供更多的功能，因为它是动态的。向导可以给用户提示，并根据用户的响应创建不同种类的文件。本节描述如何编写窗体或工程向导。

14.5.1 创建模块

窗体或工程向导典型地创建一个或多个新文件，但是代替实际文件最好还是创建没有名称、没有保存的模块。当用户保存它们时，IDE 提示用户输入文件名。向导使用创建者对象来创建这样的模块。

创建者类实现创建者接口，它从 `IOTACreator` 继承。向导给模块服务的 `CreateModule` 方法传递一个创建对象，IDE 为它需要创建模块的这些参数回调创建者对象。

例如，创建新窗体的窗体向导典型地实现 `GetExisting` 来返回 `false`，并实现 `GetUnnamed` 来返回 `true`。这创建一个没有名字模块（所以，在此文件被保存以前，用户必须给出一个名称），并不会被已有的文件支持（所以，用户必须保存这个文件，甚至在用户没有做任何改变的情况下）。创建者的其它方法告诉 IDE，正在创建什么类型的文件（例如，工程、单元、窗体），提供文件的内容，或返回窗体名称、祖先名称和其它重要的信息。另外的回调让向导给新创建的工程添加模块，或者给新创建的窗体添加组件。

要创建在窗体或工程向导中经常需要的新文件，通常需要提供新文件的内容。为了这样做，编写一个实现 `IOTAFile` 接口的新类。如果向导能设法应付缺省的文件内容，就可以从返回 `IOTAFile` 的任何函数中返回 `nil`。

例如，假设你的组织有一个必须出现在每个源文件顶端的标准说明块。你可以用在对象仓库中的静态模板这样做，但是，手工更新说明块来反映作者或创建日期。代之，当文件被创建时，你可以使用创建者来动态地填充说明块的内容。

第一步是编写创建新单元和窗体的向导。大部分创建者的函数返回 0、空串或者其它缺省值，它告诉 Tools API 为创建新单元或窗体来使用它的缺省特征。重载 `GetCreatorType`，通知 Tools API 要创建哪种模块：单元或窗体。要创建单元，返回 `sUnit`。要创建窗体，返回 `sForm`。要简化代码，使用一个类，它给构造函数提供一个创建者类型变量。

创建者类型保存在数据成员中，这样 `GetCreatorType` 可以返回它的值。实现 `NewImplSource` 和 `NewIntfSource`，返回希望的文件内容。

```
TCreateoe = class(TInterfaceObject, IOTAModuleCreator)
```

```
public
```

```
    constructor Create(const CreatorType: string);
```

```
    // IOTAModuleCreator
```

```
    function GetAncestorName: string;
```

```
    function GetImplFileName: string;
```

```
    function GetIntfFileName: string;
```

```
    function GetFormName: string;
```

```
    function GetMainForm: Boolean;
```

```
    function GetShowForm: Boolean;
```

```

function GetShowSource: Boolean;
function NewFormFile(const FormIdent, AncestorIdent: string): IOTAFile;
function NewImplSource(const ModuleIdent, FormIdent, AncestorIdent: string): IOTAFile;
function NewIntfSource(const ModuleIdent, FormIdent, AncestorIdent: string): IOTAFile;
procedure FormCreated(const FormEditor: IOTAFormEditor);

```

```

// IOTACreator
function GetCreatorType: string;
function GetExisting: Boolean;
function GetFileSystem: string;
function GetOwner: IOTAModule;
function GetUnnamed: Boolean;

```

private

```

    FCreateoeType: string;

```

end;

TCreator 的大部分成员返回 0、nil 或空串。布尔方法都返回 true，除 GetExisting 之外，它返回 false。最有趣的方法是 GetOwner，它返回指向当前工程模块的指针，或者在没有工程时返回 nil。没有简单的方法来发现当前工程或当前工程组，代之，GetOwner 必须在所有打开的模块上迭代。如果工程组被发现，它必须是唯一打开的工程组，所以，GetOwner 返回其当前工程。否则，此函数返回它发现的第一个工程模块，或者在没有打开的工程时返回 nil。

```

function TCreator.GetOwner: IOTAModule;
var
    i: Integer;
    Svc: IOTAModuleServices;
    Module: IOTAModule;
    Project: IOTAProject;
    Group: IOTAProjectGroup;
begin
    // Return the current project
    Supports(BorlandIDEServices, IOTAModuleServices, Svc);
    Result := nil;
    for i := 0 to Svc.ModuleCount - 1 do
        begin
            Module := Svc.Modules[i];
            if Supports(Module, IOTAProject, Project) then
                begin
                    // Remember the first project module
                    if Result = nil then
                        Result := Project
                end
            else if Supports(Module, IOTAProjectGroup, Group) then
                begin

```

```

        // Found the project group, so return its active project
        Result := Group.ActiveProject;
        Exit;
    end;
end;
end;
end;

```

创建者从 `NewFormSource` 返回 `nil`，产生一个缺省的窗体文件。有趣的方法是 `NewImplSource` 和 `NewIntfSource`，它们创建返回文件内容的 `IOTAFile` 实例。

`TFile` 类实现 `IOTAFile` 接口。返回 -1 表示的文件寿命（意思是此文件不存在），并返回用串表示的文件内容。要保持 `TFile` 类简洁，创建者产生这个串，`TFile` 类简单地继续传递它。

```

TFile = class(TInterfacedObject, IOTAFile)
public
    constructor Create(const Source: string);
    function GetSource: string;
    function GetAge: TDateTime;
private
    FSource: string;
end;

constructor TFile.Create(const Source: string);
begin
    FSource := Source;
end;

function TFile.GetSource: string;
begin
    Result := FSource;
end;

function TFile.GetAge: TDateTime;
begin
    Result := TDateTime(-1);
end;

```

你可以把文件内容以文本方式保存在资源中，使其容易修改，但是为了简化，这个示例把源代码硬编码在向导中。下面的例子产生源代码，假定有一个窗体。你可以很容易地添加普通单元的较简易的代码。测试 `FormIdent`，如果它是空的，就创建一个普通单元，否则创建一个窗体单元。代码的基本框架与 IDE 的缺省的框架一样（当然，在顶部添加了说明），但是，你可以按你希望的任何方式修改它。

```

function TCreator.NewImplSource(
    const ModuleIdent, FormIdent, AncestorIdent: string): IOTAFile;
var
    FormSource: string;
begin

```

```

FormSource :=
'{'-----'+ #13#10 +
'%s - description'+ #13#10 +
'Copyright © %y Your company, Inc.'+ #13#10 +
'Created on %d'+ #13#10 +
'By %u'+ #13#10 +
'-----}' + #13#10 +
#13#10;
return TFile.Create(Format(FormSource, ModuleIdent, FormIdent,
                          AncestorIdent));
end;

```

最后一步是创建两个窗体向导：一个使用 `sUnit` 作为创建者类型，另一个使用 `sForm`。作为对用户额外的好处，你可以使用 `INTAServices` 来给‘文件|新的’菜单添加菜单项来调用每一个向导。菜单项的 `OnClick` 事件处理程序可以调用向导的 `Execute` 函数。

根据 IDE 上发生的其它事情，有些向导需要使能或抑制菜单项。例如，如果 IDE 中没有文件被打开，则把工程登记进源代码控制系统的向导应抑制其‘Check in’菜单项。可以通过使用通告者给向导添加这个能力，具体见下一节的主题。

14.6 给向导通报 IDE 事件

编写行为良好向导的重要方面是使向导响应 IDE 事件。尤其，跟踪模块接口的任何向导必须知道什么时候用户关闭模块，所以，向导可以释放此接口。要做到这样，向导就需要通告者，此意味着你必须编写一个通告者类。

所有通告者类实现一个或多个通告者接口。通告者接口定义回调方法。向导用 `Tools API` 注册通告者对象，并当某些重要的事情发生时，IDE 回调这个通告者。

每一个通告者接口都继承自 `IOTANotifier`，尽管并不是其所有的方法都用于某个特殊的通告者。表 14.3 列出了全部通告者接口，并对每一个给出了简洁的说明。

表 14.3 通告者接口

接口	说明
<code>IOTANotifier</code>	所有通告者的抽象基类
<code>IOTABreakpointNotifier</code>	触发或改变调试器中的断点
<code>IOTADebuggerNotifier</code>	在调试器中运行程序，或添加、删除断点
<code>IOTAEditLineNotifier</code>	跟踪源代码编辑器中行的移动
<code>IOTAEditorNotifier</code>	修改或保存源文件，或转换编辑器中的文件
<code>IOTAFormNotifier</code>	保存窗体，或者修改窗体或窗体（或数据模块）上的任何组件
<code>IOTAIDENotifier</code>	装载工程，安装包和其它全局 IDE 事件
<code>IOTAMessageNotifier</code>	添加或删除消息示图中的标签（消息组）
<code>IOTAModuleNotifier</code>	改变、保存或重命名模块
<code>IOTAProcessModNotifier</code>	在调试器中装载进程模块
<code>IOTAProcessNotifier</code>	创建或销毁在调试器的线程和进程
<code>IOTAThreadNotifier</code>	改变调试器中线程的状态
<code>IOTAToolsFilterNotifier</code>	调用工具过滤器

要了解如何使用通告者，考虑以前的例子。使用模块创建者，这个例子创建了向导，它给每一个源文件都添加了说明。说明包括单元的初始名称，但是，用户几乎总是用不同的名字保存文件。这样的话，如果向导更新了说明以匹配文件真实的名字，则它对用户就是一个尊重。

要这样做，就需要一个模块通告者。向导保存 `CreateModule` 返回的模块接口，并使用这个接口来注册一个模块通告者。当用户修改文件或保存文件时，模块通告者接收通告，但是这些事件对这个向导并不重要，所以，`AfterSave` 和相关的函数全部都有空的函数体(empty body)。重要的函数是 `ModuleRenamed`，当用户用一个新名字保存文件时，IDE 调用它。对模块通告者类的声明如下所示：

```
TModuleIdentifier = class(TNotifierObject, IOTAModuleNotifier)
public
    constructor Create(const Module: IOTAModule);
    destructor Destroy; override;
    function CheckOverwrite: Boolean;
    procedure ModuleRenamed(const NewName: string);
    procedure Destroyed;
private
    FModule: IOTAModule;
    FName: string;
    FIndex: Integer;
end;
```

编写通告者的一个途径是在其构造函数中使它自动注册本身。析构函数卸载对通告者的注册。在模块通告者的情况下，当用户关闭文件时，IDE 调用 `Destroyed` 方法。这时，通告者必须卸载对它本身的注册，并释放到模块接口的引用。IDE 释放它对通告者的引用，它减少其引用数到 0 并释放此对象。因此，需要防御性地编写析构函数：通告者可能已经被卸载了。

```
constructor TModuleNotifier.Create(const Module: IOTAModule);
begin
    FIndex := -1;
    FModule := Module;
    // Register this notifier
    FIndex := Module.AddNotifier(self);
    // Remember the module's old name
    FName := ChangeFileExt(ExtractFileName(Module.FileName), "");
end;

destructor TModuleNotifier.Destroy;
begin
    // Unregister the notifier if that hasn't happened already
    if FIndex >= 0 then
        FModule.RemoveNotifier(FIndex);
end;
```

```

procedure TModuleNotifier.Destroyed;
begin
    // The module interface is being destroyed, so clean up the notifier
    if FIndex >= 0 then
        begin
            // Unregister the notifier
            FModule.RemoveNotifier(FIndex);
            FIndex := -1;
        end;
        FModule := nil;
    end;
end;

```

当用户更改文件名时，IDE 回调通告者的 `ModuleRenamed` 函数。这个函数采用新名字作为参数，向导用它来更新文件中的说明。要编辑源缓存，向导使用编辑位置接口。向导找到正确的位置，仔细检查它找到位置处的正确的文本，并用新名字替换这个文本。

```

procedure TModuleNotifier.ModuleRenamed(const NewName: string);
var
    ModuleName: string;
    i: Integer;
    Editor: IOTAEditor;
    Buffer: IOTAEditBuffer;
    Pos: IOTAEditPosition;
    Check: string;
begin
    // Get the module name from the new file name
    ModuleName := ChangeFileExt(ExtractFileName(NewName, ""));
    for i := 0 to FModule.GetModuleFileCount - 1 do
        begin
            //Update every source editor buffer
            Editor := FModule.GetModuleFileEditor(i);
            if Supports(Editor, IOTAEditBuffer, Buffer) then
                begin
                    Pos := Buffer.GetEditPosition;
                    {The module name is on line 2 of the comment.
                     Skip leading white space and copy the old module name,
                     to double check we have the right spot. }
                    Pos := Move(2, 1);
                    Pos.MoveCursor(mmSkipWhite or mmSkipRight);
                    Check := Pos.RipText(", rfIncludeNumericChars or rfIncludeAlphaChars");
                    if Check = FName then
                        begin
                            Pos.Delete(Length(Check));    // Delete the old name
                            Pos.InsertText(ModuleName);    // Insert the new name
                            FName := ModuleName;           // Remember the new name
                        end;
                end;
            end;
        end;
    end;
end;

```

```

        end;
    end;
end;

```

如果用户在模块名上面插入另外的说明会发生什么呢？这时，需要使用一个编辑行通告者来追踪模块名所在的行号。要这样做，使用 `IOTAEditLineNotifier` 和 `IOTAEditLineTracker` 接口，在线帮助中对它们有说明。

当编写通告者时，需要小心。必须确保没有通告者在其向导销毁后还存在。例如，如果用户准备使用向导创建一个新单元，然后卸载这个向导，将仍然还有一个通告者附在这个单元上。其结果将是不可预料的，但是极有可能，IDE 会蹦垮。因此，向导需要跟踪其所有通告者，并必须在向导被销毁以前卸载对每一个通告者的注册。另外，如果用户首先关闭了文件，则模块通告者会收到 `Destroyed` 通告，此意味着通告者必须卸载自身，并释放对此模块的所有引用。通告者也必须从向导的主通告者列表（`master notifier list`）中删除它自己。

下面是向导的 `Execute` 函数的最终版本。它创建新的模块，使用模块的接口，并创建一个模块通告者，然后保存模块通告者在接口列表（`TInterfaceList`）中。

```

procedure DocWizard.Execute;
var
    Svc: IOTAModuleServices;
    Module: IOTAModule;
    Notifier: IOTAModuleNotifier;
begin
    // Return the current project
    Supports(BorlandIDEServices, IOTAModuleServices, Svc);
    Module := Svc.CreateModule(TCreator.Create(creator_type));
    Notifier := TModuleNotifier.Create(Module);
    list.Add(Notifier);
end;

```

向导的析构函数在接口列表上迭代，并卸载列表中的每一个通告者。简单地让接口列表释放它保存的接口还不够，因为 IDE 也保存有相同的接口。为了最终释放通告者对象，你必须告诉 IDE 释放通告者接口。这时，析构函数哄骗通告者，让它认为它们的模块已经被销毁了。在更复杂的情况下，你可能会发现，最好还是为通告者类编写一个单独的 `Unregister` 函数。

```

destructor DocWizard.Destroy; override;
var
    Notifier: IOTAModuleNotifier;
    i: Integer;
begin
    // RUnregister all the notifier in the list
    for i := list.Count - 1 downto 0 do
        begin
            Supports(list.Items[i], IOTANotifier, Notifier);
            { Pretend the associated object has been destroyed.
              That convinces the notifier to clean itself up. }
            Notifier.Destroyed;
            list.Delete(i);
        end;
    end;
end;

```



```
end;  
list.Free;  
FItem.Free;  
end;
```

向导的其余部分操纵注册向导、安装菜单项等的一般细节。

索引

A

About box

- adding properties

- executing

About unit

AboutDlg unit

abstract classes

ancestor classes

applications

- graphical

- realizing palettes

arrays

assignment statements

attributes, property editors

B

BeginAutoDrag method

bitmaps

- adding to components

- drawing surfaces

- graphical controls vs.

- loading

- offscreen

Boolean values

BorlandIDEServices variable

BoundsChanged method

Broadcast method

Brush property

BrushCopy method

brushes

- changing

C

caching resources

calendars

- adding dates

- defining properties and events

- making read-only

- moving through

- resizing

- selecting current day

Canvas property

canvases

‘关于’对话框

添加属性

运行

‘关于’单元

AboutDlg 单元

抽象类

祖先类

应用程序

图形的

实现调色板

数组

赋值语句

特性、属性编辑器

BeginAutoDrag 方法

位图

给组件添加

绘制表面

图形控件与...

装载

离屏

布尔值

BorlandIDEServices 变量

BoundsChanged 方法

Broadcast 方法

Brush 属性

BrushCopy 方法

画刷

改变

缓存资源

日历

添加日期

定义属性和事件

使只读

移过

重新改变尺寸

选择当前日

Canvas 属性

画布

default drawing tools	缺省绘图工具
palettes	调色板
Change method	Change 方法
characters	字符
circles, drawing	圆, 绘制
class fields	类域
declaring	声明
naming	命名
class pointers	类指针
classes	类
abstract	抽象的
accessing	访问
ancestor	祖先
creating	创建
default	缺省的
defining	定义
static methods and	静态方法和...
virtual methods and	虚拟方法和...
deriving	分派
descendant	子孙
hierarchy	层次关系
inheritance	继承
instantiating	实例化
passing as parameters	作为参数传递
properties as	作为...属性
property editors as	作为...属性编辑器
public part	公开部分
published part	发布部分
click events	点击事件
Click method	Click 方法
overriding	重载
Clipboard formats, adding	剪贴板格式, 添加
CLX	跨平台组件库 (在 Linux 操作系统下使用)
signals	信号
system notifications	系统通告
CM_EXIT message	CM_EXIT 消息
CMEExit method	CMEExit 方法
code	代码
Code editor, displaying	代码编辑器、显示
ColorChanged method	ColorChanged 方法
common dialog boxes	普通对话框
creating	创建
executing	运行
compile-time errors, override directive and	编译时错误, 重载指示和...

component editors	组件编辑器
default	缺省的
registering	注册
component interfaces	组件接口
creating	创建
properties, declaring	属性, 声明
component libraries, adding components	组件库, 添加组件
Component palette	组件面板
adding components	添加组件
adding custom bitmaps	添加定制位图
moving components	移动组件
component templates	组件模板
Component wizard	组件向导
component wrappers	组件包装器
initializing	初始化
components	组件
abstract	抽象的
adding to Component Palette	给组件面板添加
adding to existing unit	给已有单元添加
adding to units	给单元添加
bitmaps	位图
changing	改变
context menus	上下文菜单
creating	创建
customizing	定制
data-aware	数据感知
data-browsing	数据浏览
data-editing	数据编辑
dependencies	依赖
derived classes	派生类
double-click	双击
initializing	初始化
installing	安装
interfaces	接口
design-time	设计时
runtime	运行时
moving	移动
nonvisual	非可视的
online help	在线帮助
packages	包
palette bitmaps	组件面板位图
registering	注册
registration	注册
resources, freeing	资源, 释放

responding to events	响应事件
testing	测试
constructors	构造函数
overriding	重载
owned objects and	拥有的对象和...
Contains list (packages)	Contains list (包)
context menus, adding items	上下文菜单, 添加菜单项
controls	控件
changing	改变
custom	定制
data-browsing	数据浏览
data-editing	数据编辑
graphical	图形的
creating	创建
drawing	绘制
events	事件
palettes and	调色板和...
receiving focus	接受输入焦点
repainting	重绘
resizing	重定尺寸
shape	形状 (控件)
windowed	窗口 (控件)
CopyMode property	CopyMode 属性
CopyRect method	CopyRect 方法
creators	创建者 (接口)
CursorChanged method	CursorChanged 方法
custom controls	定制控件
libraries	库
customizing components	定制的组件
D	
data links	数据链接
initializing	初始化
data, accessing	数据, 访问
data-aware controls	数据感知控件
creating	创建
data-browsing	数据浏览
data-editing	数据编辑
destroying	销毁
responding to changes	响应改变
databases	数据库
access properties	访问属性
DataChange method	DataChange 方法
DataField property	DataField 属性
DataSource property, data-aware controls	DataSource 属性, 数据感知控件

Day property	Day 属性
DbClick method	DbClick 方法
.dcr files	.dcr 文件
debugging wizards	调试向导
declarations	声明
classes	类
public	公开的
published	发布的
event handlers	事件处理程序
message handlers	消息处理程序
methods	方法
dynamic	动态的
public	公开的
static	静态的
virtual	虚拟的
new component types	新的组件类型
properties	属性
stored	stored 指示
user-defined types	用户定义类型
default	缺省的
ancestor class	祖先类
directive	指示
handlers	处事程序
events	事件
message	消息
overriding	重载
property values	属性值
changing	改变
specifying	指定
reserved word	保留字
DefaultHandler method	DefaultHandler 方法
delegation	委托
deploying IDE extensions	进行 IDE 扩展
dereferencing object pointers	废弃的对象指针
deriving classes	派生类
descendant classes	子孙类
redefining methods	重定义方法
design-time interfaces	设计时接口
destructors	析构函数
owned objects and	拥有的对象和...
device contexts	设计上下文
device-independent graphics	设备独立图形
.dfm files	.dfm 文件
dialog boxes	对话框

creating	创建
property editors as	作为...属性编辑器
setting initial state	设定初始状态
Windows common	Windows 公共的
creating	创建
executing	运行
directives	指示
default	缺省的
dynamic	动态的
override	重载
protected	受保护的
public	公开的
published	发布的
stored	stored 指示
virtual	虚拟的
Dispatch method	Dispatch 方法
DoExit method	DoExit 方法
DoMouseWheel method	DoMouseWheel 方法
double-clicks	鼠标双击
components	组件
responding to	响应...
drag-and-drop events	拖放事件
DragOver method	DragOver 方法
Draw method	Draw 方法
drawing tools	绘图工具
changing	改变
dynamic directives	dynamic 指示
dynamic methods	动态方法
E	
Edit method	Edit 方法
editors, Tools API	编辑器, Tools API
Ellipse method	Ellipse 方法
ellipses, drawing	椭圆, 绘制
EnabledChanged method	EnabledChanged 方法
enumerated types	枚举类型
event handlers	事件处理程序
declarations	声明
default, overriding	缺省的, 重载
displaying the Code editor	显示代码编辑器
empty	空的
methods	方法
parameters	参数
notification events	通告事件
passing parameters by reference	用引用传递参数

pointers	指针
types	类型
EventFilter method, system events	EventFilter 方法，系统事件
events	事件
accessing	访问
defining new	定义新的
graphical controls	图形控件
implementing	实现
inherited	继承的
message handling	消息处理
naming	命名
providing help	提供帮助
responding to	响应...
retrieving	找回
standard	标准的
exceptions	异常
Execute method dialogs	Execute 方法对话框
F	
field datalink class	域的数据链接类
fields	域
databases	数据库
message records	消息记录
files, graphics	文件、图形
FillRect method	FillRect 方法
finally reserved word	finally 保留字
flags	标记
FloodFill method	FloodFill 方法
focus	焦点
Font property	Font 属性
FontChanged method	FontChanged 方法
form wizards	窗体向导
forms, as components	窗体，作为组件
FReadOnly	FReadOnly
freeing resources	释放资源
functions	函数
events and	事件和
graphics	图形
naming	命名
reading properties	读取属性
Windows API	Windows 应用程序编辑接口
G	
GDI applications	GDI 应用程序
geometric shapes, drawing	几何形状，绘制
GetAttributes method	GetAttributes 方法

GetFloatValue method	GetFloatValue 方法
GetMethodValue method	GetMethodValue 方法
GetOrdValue method	GetOrdValue 方法
GetPalette method	GetPalette 方法
GetProperties method	GetProperties 方法
GetStrValue method	GetStrValue 方法
GetValue method	GetValue 方法
Graphic property	Graphic 属性
graphical controls	图形控件
bitmaps vs.	位图与...
creating	创建
drawing	绘图
events	事件
saving system resources	保存系统资源
graphics	图形
complex	复杂的
containers	容器
drawing tools	绘图工具
changing	改变
functions, calling	函数, 调用
loading	装载
methods	方法
copying images	复制图象
palettes	调色板
redrawing images	重绘图象
resizing	重定尺寸
saving	保存
standalone	独立的
storing	保存
graphics methods, palettes	图形方法, 调色板
grids	网格
H	
Handle property	Handle 属性
HandleException method	HandleException 方法
Help	帮助
Help systems	帮助系统
files	文件
keywords	关键字
hierarchy (classes)	层次关系 (类)
HookEvents method	HookEvents 方法
I	
icons	图标
adding to components	给组件添加
IDE	集成开发环境

adding actions	添加动作
adding images	添加图象
customizing	定制
deleting buttons	删除按钮
identifiers	标识符
class fields	类域
events	事件
message-record types	消息记录类型
methods	方法
property settings	属性设定
Image Editor, using	图象编辑器, 使用
images	图象
drawing	绘图
redrawing	重绘
reducing flicker	减少屏幕闪烁
index reserved word	index 保留字
indexes	索引
inherited	继承的
events	事件
methods	方法
properties	属性
publishing	发布
inheriting from classes	从类继承...
input focus	输入焦点
Install Components dialog box	安装组件对话框
instances	实例
INTAComponent	INTAComponent
INTAServices	INTAServices
interfaces	接口
design-time	设计时
nonvisual program elements	非可视程序元素
properties	属性
properties, declaring	属性, 声明
runtime	运行时
Tools API	Tools API
version numbers	版本号
Invalidate method	Invalidate 方法
IOTAActionServices	IOTAActionServices 接口
IOTABreakpointNotifier	IOTABreakpointNotifier 接口
IOTACodeCompletionServices	IOTACodeCompletionServices 接口
IOTAComponent	IOTAComponent 接口
IOTACreator	IOTACreator 接口
IOTADebuggerNotifier	IOTADebuggerNotifier 接口
IOTADebuggerServices	IOTADebuggerServices 接口

IOTAEditLineNotifier	IOTAEditLineNotifier 接口
IOTAEditor	IOTAEditor 接口
IOTAEditorNotifier	IOTAEditorNotifier 接口
IOTAEditorServices	IOTAEditorServices 接口
IOTAFile	IOTAFile 接口
IOTAFormNotifier	IOTAFormNotifier 接口
IOTAFormWizard	IOTAFormWizard 接口
IOTAIDENotifier	IOTAIDENotifier 接口
IOTAKeyBindingServices	IOTAKeyBindingServices 接口
IOTAKeyboardDiagnostics	IOTAKeyboardDiagnostics 接口
IOTAKeyboardServices	IOTAKeyboardServices 接口
IOTAMenuWizard	IOTAMenuWizard 接口
IOTAMessageNotifier	IOTAMessageNotifier 接口
IOTAMessageServices	IOTAMessageServices 接口
IOTAModule	IOTAModule 接口
IOTAModuleNotifier	IOTAModuleNotifier 接口
IOTAModuleServices	IOTAModuleServices 接口
IOTANotifier	IOTANotifier 接口
IOTAPackageServices	IOTAPackageServices 接口
IOTAProcessModNotifier	IOTAProcessModNotifier 接口
IOTAProcessNotifier	IOTAProcessNotifier 接口
IOTAProjectWizard	IOTAProjectWizard 接口
IOTAServices	IOTAServices 接口
IOTAThreadNotifier	IOTAThreadNotifier 接口
IATAToDoServices	IATAToDoServices 接口
IOTAToolsFilter	IOTAToolsFilter 接口
IOTAToolsFilterNotifier	IOTAToolsFilterNotifier 接口
IOTAWizard	IOTAWizard 接口
IOTAWizardServices	IOTAWizardServices 接口
K	
K footnotes (Help systems)	K 脚注（帮助系统）
keyboard events	键盘事件
key-down messages	键盘按下事件
KeyDown method	KeyDown 方法
KeyPress method	KeyPress 方法
KeyString method	KeyString 方法
KeyUp method	KeyUp 方法
keywords	关键字
protected	受保护的
L	
labels	标签
leap years	闰年
libraries, custom controls	库，定制控件
Lines property	Lines 属性

LineTo method	LineTo 方法
Linux system notifications	Linux 系统通告
list boxes	列表框
Loaded method	Loaded 方法
LoadFromFile method, graphics	LoadFromFile 方法，图形
LParam parameter	LParam 参数
lParam parameter	lParam 参数
M	
MainWndProc method	MainWndProc 方法
memo controls	memo 控件（多行编辑控件）
memory management, dynamic vs. virtual	内存管理，动态的与虚拟的
methods	方法
menu wizards	菜单向导
message handlers	消息处理程序
creating	创建
declarations	声明
default	缺省的
methods, redefining	方法，重定义
overriding	重载
message handling	消息处理
messages	消息
cracking	分离
defined	定义的
handlers declarations	处理程序声明
identifiers	标识符
key	键
Linux See system notifications	Linux See 系统通告
mouse	鼠标
mouse- and key-down	鼠标按下和键盘按下
multithreaded applications	多线程应用程序
record types, declaring	记录类型，声明
records	记录
sending	发送
trapping	捕捉
user-defined	用户定义的
Windows	Windows 操作系统
metafiles	元文件
method pointers	方法指针
methods	方法
calling	调用
declaring	声明
dynamic	动态的
public	公开的
static	静态的

virtual	虚拟的
dispatching	分派
drawing	绘制
event handlers	事件处理程序
overriding	重载
graphics	图形
palettes	调色板
inherited	继承的
initialization	初始化
message-handling	消息处理
naming	命名
overriding	重载
properties and	属性和...
protected	受保护的
public	公开的
redefining	重定义
virtual	虚拟的
Modified method	Modified 方法
modules	模块
Tools API	Tools API
Month property	Month 属性
months, returning current	月，返回当前的
mouse events	鼠标事件
mouse messages	鼠标消息
mouse-down messages	鼠标按下消息
MouseDown method	MouseDown 方法
MouseMove method	MouseMove 方法
MouseUp method	MouseUp 方法
MoveTo method	MoveTo 方法
Msg parameter	Msg 参数
multi-threaded applications, sending	多线程应用程序，发送
messages	消息
MyEvent_ID type	MyEvent_ID 类型
N	
naming conventions	命名约定
events	事件
fields	域
message-record types	消息记录类型
methods	方法
properties	属性
native tools API	Native Tools API
New command	‘新的’命名
New Unit command	‘新单元’命令
nonvisual components	非可视组件

notification events	通告事件
notifiers	通告者接口
Tools API	Tools API
writing	编写
numbers	数字
property values	属性值
O	
Object Inspector	对象查看器
editing array properties	编辑数组属性
help with	帮助...
Object Repository wizards	对象仓库向导
object-oriented programming	面向对象编程
declarations	声明
classes	类
methods	方法
objects	对象
instantiating	实例化
owned	拥有的、自有的
initializing	初始化
temporary	临时的
offscreen bitmaps	离屏位图
OnChange event	OnChange 事件
OnClick event	OnClick 事件
OnCreate event	OnCreate 事件
OnDataChange event	OnDataChange 事件
OnDbClick event	OnDbClick 事件
OnDragDrop event	OnDragDrop 事件
OnDragOver event	OnDragOver 事件
OnEndDrag event	OnEndDrag 事件
OnEnter event	OnEnter 事件
OnExit event	OnExit 事件
OnKeyDown event	OnKeyDown 事件
OnKeyPress event	OnKeyPress 事件
OnKeyString event	OnKeyString 事件
OnKeyUp event	OnKeyUp 事件
online help	在线帮助
OnMouseDown event	OnMouseDown 事件
OnMouseMove event	OnMouseMove 事件
OnMouseUp event	OnMouseUp 事件
Open Tools API See Tools API	Open Tools API See Tools API
optimizing system resources	优化系统资源
override directive	override 指示
overriding methods	重载方法
owned objects	拥有的对象

initializing	初始化
Owner property	Owner 属性
P	
packages	包
components	组件
Contains list	Contains 链表
Requires list	Requires 链表
Paint method	Paint 方法
PaintRequest method	PaintRequest 方法
palette bitmap files	组件面板位图文件
PaletteChanged method	PaletteChanged 方法
palettes	组件面板
default behavior	缺省行为
specifying	指定
parameters	参数
classes as	作为...类
event handlers	事件处理程序
messages	消息
property settings	属性设置
array properties	数组属性
Parent property	Parent 属性
Pen property	Pen 属性
pens	画笔
changing	改变
Perform method	Perform 方法
picture objects	图片对象
pictures	图片
Pixel property	Pixel 属性
pointers	指针
classes	类
default property values	缺省属性值
method	方法
PostMessage method	PostMessage 方法
preexisting controls	预先存在的控件
private properties	私有属性
procedures	过程
naming	命名
property settings	属性设定
project wizards	工程向导
properties	属性
accessing	访问
array	数组
as classes	作为类
changing	改变

common dialog boxes	普通对话框
declaring	声明
stored	stored 指示
user-defined types	用户定义类型
default values	缺省值
redefining	重定义
editing as text	作为文本进行编辑
events and	事件和...
inherited	继承的
interfaces	接口
internal data storage	内部数据保存
loading	装载
nodefault	nodefault 指示
overview	概述
providing help	提供帮助
published	发布的
read and write	读、写
reading values	读取值
read-only	只读
redeclaring	重声明
specifying values	指定值
storing	保存
storing and loading unpublished	保存与装载未发布的
subcomponents	子组件
types	类型
updating	更新
viewing	查看
wrapper components	包装器组件
write-only	只写
writing values	写入值
property editors	属性编辑器
as derived classes	作为派生类
attributes	特性
dialog boxes as	作为...对话框
registering	注册
property settings	属性设置
reading	读取
writing	写入
protected	受保护的
directive	指示
events	事件
keyword	关键字
public	公开的
directive	指示

keyword	关键字
part of classes	类的部分
properties	属性
published	发布的
directive	指示
keyword	关键字
part of classes	类的部分
properties	属性
example	示例
Q	
QApplication_postEvent method	QApplication_postEvent 方法
QCustomEvent_create function	QCustomEvent_create 函数
QEvent	QEvent
QKeyEvent	QKeyEvent
QMouseEvent	QMouseEvent
Qt events messages	Qt 事件消息
R	
raster operations	光栅操作
read method	读的方法
read reserved word	read 保留字
reading property settings	读取属性设定
read-only properties	只读属性
ReadOnly property	ReadOnly 属性
realizing palettes	实现调色板
Rectangle method	Rectangle 方法
rectangles, drawing	长方形, 绘制
redefining methods	重定义方法
redrawing images	重绘图象
Register procedure	Register 过程
RegisterComponents procedure	RegisterComponents 过程
registering	注册
component editors	组件编辑器
components	组件
property editors	属性编辑器
RegisterPropertyEditor procedure	RegisterPropertyEditor 过程
repainting controls	重绘控件
Requires list (packages)	Requires 链表 (包)
.res files	.res 文件
resizing controls	重定控件尺寸
graphics	图形
resources	资源
caching	缓冲
freeing	释放
system, optimizing	系统, 优化

Result parameter
RTTI
runtime interfaces
runtime type information
S
SaveToFile method, graphics
search lists (Help systems)
SelectCell method
Self parameter
sending messages
SendMessage method
services, Tools API
set types
SetFloatValue method
SetMethodValue method
SetOrdValue method
sets
SetStrValue method
SetValue method
ShowHintChanged method
signals, responding to (CLX)
simple types
squares, drawing
standard events
 customizing
static methods
stored directive
StretchDraw method
strings
 returning
StyleChanged method
subclassing Windows controls
subcomponent s properties
system events, customizing
system notifications
system resources, conserving

T

TabStopChanged method
TApplication system events
TBitmap
TCalendar
TCharProperty type
TClassProperty type
TColorProperty type

Result 参数
运行时类型识别
运行时接口
运行时类型信息
SaveToFile 方法，图形
搜索列表（帮助系统）
SelectCell 方法
Self 参数
发送消息
SendMessage 方法
服务，Tools API
设定类型
SetFloatValue 方法
SetMethodValue 方法
SetOrdValue 方法
设定
SetStrValue 方法
SetValue 方法
ShowHintChanged 方法
信号，响应...（CLX）
简单类型
正方形，绘制
标准事件
定制
静态方法
stored 指示
StretchDraw 方法
串
返回
StyleChanged 方法
子类化 Windows 控件
子组件属性
系统事件，定制
系统通告
系统资源，保存

TabStopChanged 方法
TApplication 系统事件
TBitmap
TCalendar 类
TCharProperty 类型
TClassProperty 类型
TColorProperty 类型

TComponent	TComponent
TComponentProperty type	TComponentProperty 类型
TControl	TControl
TCustomControl	TCustomControl
TCustomGrid	TCustomGrid
TCustomListBox	TCustomListBox
TDateTime type	TDateTime 类型
TDefaultEditor	TDefaultEditor
temporary objects	临时对象
TEnumProperty type	TEnumProperty 类型
Testing	测试
components	组件
values	值
TextChanged method	TextChanged 方法
TextHeight method	TextHeight 方法
TextOut method	TextOut 方法
TextRect method	TextRect 方法
TextWidth method	TextWidth 方法
TFieldDataLink	TFieldDataLink
TFloatProperty type	TFloatProperty 类型
TFontNameProperty type	TFontNameProperty 类型
TFontProperty type	TFontProperty 类型
TGraphic	TGraphic
TGraphicControl	TGraphicControl
THandleComponent	THandleComponent
TIcon	TIcon
TiledDraw method	TiledDraw 方法
TIntegerProperty type	TIntegerProperty 类型
TKeyPressEvent type	TKeyPressEvent 类型
TLabel	TLabel
TListBox	TListBox
TMessage	TMessage
TMetafile	TMetafile
TMethod type	TMethod 类型
TMethodProperty type	TMethodProperty 类型
TNotifyEvent	TNotifyEvent
TObject	TObject
Tools API	Tools API
creating files	创建文件
creators	创建者接口
debugging	调试
editors	编辑器
modules	模块
notifiers	通告者接口

services	服务
wizards	向导
ToolsAPI unit	ToolsAPI 单元
TOrdinalProperty type	TOrdinalProperty 类型
TPicture type	TPicture 类型
TPropertyAttributes	TPropertyAttributes
TPropertyEditor class	TPropertyEditor 类
transfer records	传送记录
try reserved word	try 保留字
TSetElementProperty type	TSetElementProperty 类型
TSetProperty type	TSetProperty 类型
TStringProperty type	TStringProperty 类型
TWinControl	TWinControl
type declarations, properties	类型声明, 属性
types	类型
message-record	消息记录
properties	属性
user-defined	用户定义的
U	
units, adding components	单元, 添加组件
UpdateCalendar method	UpdateCalendar 方法
user-defined messages	用户定义消息
user-defined types	用户定义类型
V	
values	值
Boolean	布尔
default property	缺省属性
redefining	重定义
testing	测试
var reserved word event handlers	var 保留字 事件处理程序
VCL	可视组件库
virtual	虚拟的
directive	指示
method tables	方法表
methods	方法
properties as	作为...属性
property editors	属性编辑器
VisibleChanged method	VisibleChanged 方法
W	
WidgetDestroyed property	WidgetDestroyed 属性
window	窗口
class	类
controls	控件
handles	处理程序

message handling	消息处理
procedures	过程
Windows	Windows 操作系统
API functions	API 函数
common dialog boxes	普通对话框
creating	创建
executing	运行
controls, subclassing	控件, 子类化
device contexts	设备上下文
events	事件
messages	消息
messaging	消息处理
wizards	向导
Component	组件
creating	创建
debugging	调试
installing	安装
responding to IDE events	响应 IDE 事件
Tools API	Tools API
Types	类型
WM_APP constant	WM_APP 常数
WM_KEYDOWN message	WM_KEYDOWN 消息
WM_LBUTTONDOWN message	WM_LBUTTONDOWN 消息
WM_MBUTTONDOWN message	WM_MBUTTONDOWN 消息
WM_PAINT message	WM_PAINT 消息
WM_RBUTTONDOWN message	WM_RBUTTONDOWN 消息
WM_SIZE message	WM_SIZE 消息
WndProc method	WndProc 方法
WordWrap property	WordWrap 属性
WParam parameter	WParam 参数
wParam parameter	wParam 参数
wrappers	包装器
initializing	初始化
See also component wrappers	也参见组件包装器
write method	写的方法
write reserved word	write 保留字
write-only properties	只写属性
Y	
Year property	Year 属性