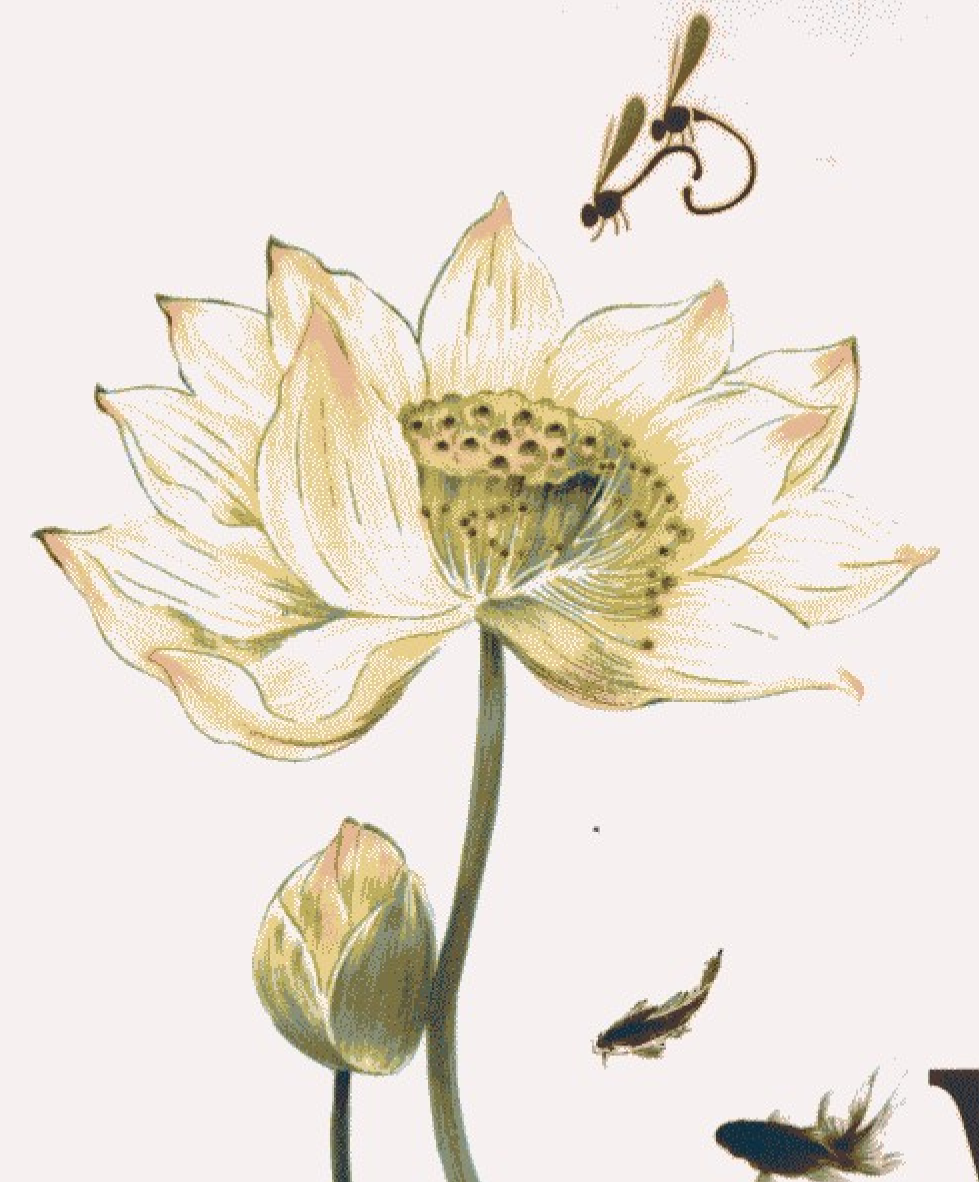
全面系统，深入实用
通俗易懂，实例丰富



WPF专业编程指南

李应保 著

博文视点 · IT出版旗舰品牌

技术凝聚实力 · 专业创新出版

WPF专业编程指南

本书特色：

本书介绍WPF编程的各种基本概念，重在实用，尽量避免枯燥的叙述，代之以大量的图表和程序实例；在采用UML语言形象地展现WPF中各种对象间的相互关系时，尽量把WPF作为一个整体呈现在读者的面前，从而避免了孤立地理解WPF中的某个类或单一技术。软件开发工程师在初次接触一门技术的时候，往往从某个概念及相关技术开始，但要组织一个实用的项目，并综合使用相关技术还有相当的距离。为此，本书专门写了第18章，在最新开源代码的基础上，改写了一个实用的工程项目，为读者打开了综合运用WPF技术的思路。

◇ 学习WPF可以获取新的工作机遇

◇ 应用WPF可以降低企业的开发成本

◇阅读本书是通向WPF专业编程的捷径

光盘： 光盘中含有本书所有实例的完整源代码。

上架建议：程序设计>.NET技术

责任编辑：高洪霞
责任美编：侯士卿

本书贴有激光防伪标志，凡没有防伪标志者，属盗版图书。

定价：68.00元（含光盘1张）

WPF
专业编程指南

李应保　著

電子工業出版社
Publishing House of Electronics Industry
北京•BEIJING

内 容 简 介

《WPF 专业编程指南》是一本 WPF 编程的专业参考书，全书通过大量的实例深入阐述了 WPF 中的传递事件、传递命令、相关属性、附加属性、逻辑树和视觉树等基本概念；介绍了各种画笔、画刷的使用方法；深入讨论了 WPF 中的各种控件以及这些控件在窗口或页面上的排版，并进而阐述了控件的风格和模板及数据绑定等相关技术。

本书对 WPF 中的图形系统及图形和排版的变换原理也进行了深入的探讨，并在此基础上讨论了 WPF 中的动画技术。多媒体不是 WPF 专有的技术，但本书介绍了在 WPF 中使用多媒体的实用技术。用户控件和自定义控件是 WPF 中比较深入的内容，本书最后两章对这一课题进行了深层次的研究，通过对 Ribbon 控件的开发，不仅可以了解开发用户控件和自定义控件的方法，而且可以体会 WPF 项目的组织及多种 WPF 技术细节的综合运用。

本书可供.NET 桌面及互联网应用程序的开发人员、项目管理人员或准备进入这一领域的相关工程技术人员，以及大专院校相关专业的师生参考学习。

未经许可，不得以任何方式复制或抄袭本书之部分或全部内容。
版权所有，侵权必究。

图书在版编目（CIP）数据

WPF 专业编程指南 / 李应保著.—北京：电子工业出版社，2010.1
ISBN 978-7-121-10011-6

I. W… II.李… III.窗口软件，Windows Vista—用户界面—程序设计 IV.TP316.7

中国版本图书馆 CIP 数据核字（2009）第 220308 号

策划编辑：袁金敏
责任编辑：高洪霞
特约编辑：顾慧芳
印　　刷：北京天宇星印刷厂
装　　订：涿州市桃园装订有限公司
出版发行：电子工业出版社
　　　　　北京市海淀区万寿路 173 信箱　　邮编 100036
开　　本：860×1092　1/16　　印张：34　　字数：828 千字
印　　次：2010 年 1 月第 1 次印刷
印　　数：4000 册　　定价：68.00 元（含光盘 1 张）

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

服务热线：（010）88258888。

前 言

2008年在世界经济历史上是一个不平凡的一年，在这一年中，美国的五大投资银行有两家破产。金融危机席卷全球，美国的失业率在过去的一年从4%飙升到了10%，有些城市的失业率甚至高达20%；加拿大的就业数字也是惨不忍睹，多伦多的失业率剧升到了12%（2009年8月数据）。在这样暗淡的经济背景下，有关WPF的工作却在快速增长，过去两年，和WPF相关的工作职位在北美和欧洲每半年翻一番，WPF初级职位的薪水在5万美元到7万美元之间，高级职位在10万美元以上。一般来说，新的开发平台在中国的应用比欧美要晚2~3年，但随着互联网的普及和软件开发外包到中国，这一迟后时间正在缩短。可以预见，在未来几年内，和WPF相关的工作职位也会在中国快速增长；所以，学习WPF编程技术正当其时，拥有WPF技术必将使你在职场上更加挥洒自如，也就是说，**学习WPF就是获取新的工作机遇。**

WPF是基于.NET的新一代界面开发平台，它实现了桌面应用程序和互联网应用程序的统一编程，实现了程序员长期梦想的数据驱动用户界面，融合了动画、多媒体的功能，跨越了图形和控件、控件和排版等技术上的界限，在很短的时间内实现并超越了Flash和PDF的相关功能。近年来微软在软件开发上的投资额已和中等国家的GDP相当，其中.NET是微软最大的投资项目。在.NET 2.0之后，微软停止了WinForm的开发，而WPF实际上是微软在今后一段时间内唯一要不断投入的用户界面开发平台，这一平台在Vista、Windows 7 和Window Server上得到了广泛的支持。建立在WPF上的应用程序将会自动随着.NET的不断开发而自动拓展新的功能：把WPF和WCF（Windows Communication Foundation）技术相结合，可以很容易地实现面向服务的软件架构（SOA，Service Oriented Architecture）；WPF对图形流的支持，使得开发GIS应用程序更加方便；基于SilverLight和Ajax技术是互联网开发的新热点。所以，企业把应用程序的界面建立在WPF之上，不仅可以极大地缩短开发周期，而且可以把同一技术用在不同的项目上，从而极大地降低开发成本。比如过去开发桌面应用程序和互联网应用程序一直是两个不同的开发团队，而使用WPF，我们只需要一个开发团队。在过去的20年内，微软一直是用户界面开发的领跑者；若企业把应用程序建立在WPF之上，就不会担心落伍。所以，**应用WPF就是降低企业的开发成本**。

本书特点

本书深入浅出地介绍了WPF中的各种新概念，使用了大量图表和实例力图以整体的形式把WPF展现在读者的面前。笔者推崇Scott Meyers的写作风格（Scott Meyers的C++系列丛书在软件界有广泛的影响——笔者注），即以散文的笔调描述技术细节，以避免枯燥的叙述；希望读者在阅读本书时有一种像朋友在一起聚会的感觉，边喝啤酒边聊天，在聚会结束时，您会发现自己已经掌握了WPF技术。因此，**阅读本书是通向WPF专业编程的捷径。**

由于WPF是在.NET 3.0 之后引入的，故读者在使用WPF之前应该已经熟悉 .NET的编程环境、C#语言等基本知识。本书使用简单的UML（Unified Modeling Language）描述WPF类和类间的关系，若您具有UML的基本知识，对阅读本书会有帮助。书中的例子在Visual Studio 2008 和.NET FrameWork

3.5上调试过，有时笔者也使用了微软的Expression Blend 2.0调试，但后者不是必需的。

本书光盘使用说明

本书的配书光盘含有约100MB的源程序，所用的语言为C#和XAML。所有的例程在Visual Studio 2008 和.NET Framework 3.5上调试通过，笔者在创建某些例程时，使用过Microsoft Blend 2.0。Microsoft Blend工具在创建WPF界面时非常有用，但对于运行本书的例程不是必需的。本光盘的内容是对本书的补充，因书中着重介绍WPF编程模型和基本概念，光盘中则含有完整的源代码。

1．光盘中的目录使用Yingbao.Chapterxx格式，如第1章的例程在Yingbao.Chapter1的目录下，第2章的例程在Yingbao.Chapter2的目录下等。全书共18章，整个光盘含有18个目录。

2．在每个目录下有一个相应的Visual Studio解决方案文件，其文件名采用Yingbao.Chapterxx.sln格式，如第1章Visual Studio解决方案文件名为Yingbao.Chapter1.sln，第2章Visual Studio解决方案文件名为Yingbao.Chapter2.sln等。该文件中含有一个或多个项目，如Yingbao.Chapter13解决方案中含有六个项目。运行某个项目，您需要在Visual Studio中的Solution explore窗口下单击鼠标右键，在弹出菜单中选择“Set as StartUp Project”条目，然后，您只要按下“F5”功能键，或在Visual Studio 的“Debug”菜单下选择“Start Debug”即可。

3．例程中命名空间的名字采用Yingbao.Chapterxx的格式，如第1章所有例程中的类都在Yingbao.Chapter1命名空间中，第2章的所有例程中的类都在Yingbao.Chapter2命名空间中。虽然对于本书的例程来说使用不同的命名空间不是必需的，但对于组织大型项目，使用命名空间是良好的习惯。

4．例程中的类名、属性名、域名、方法名采用通用电气（GE）编程规范，读者也可用其他的公司或自己公司的编程规范。

感谢

写作是一个费时费力的工作，笔者在写作本书的时候，得到了家人的支持和理解。电子工业出版社的杨福平副总编和袁金敏编辑对本书的出版做了大量的工作，笔者在此表示衷心感谢。

联系方式

最后，若您有什么建议和意见或者发现书中的错误，请和笔者联系：Yingbao.Li@gmail.com。

李应保

2009年9月8日于加拿大

目　录

第一篇　WPF编程基础

第 1 章　WPF 应用程序 ······ 2

1.1　WPF 应用程序的创建 ······ 2

1.2　Dos 窗口 ······ 3

1.3　WPF 应用程序的启动和终止 ······ 4

1.4　输入参数 ······ 5

1.5　在 Xaml 中创建 Application ······ 7

1.6　窗口大小 ······ 10

1.7　互联网应用程序 ······ 10

1.8　应用程序的异常处理 ······ 11

1.9　应用程序中的资源 ······ 12

1.10　应用程序的发布 ······ 13

1.11　WPF 开发环境 ······ 14

1.12　本章小结 ······ 15

第 2 章　XAML 语言 ······ 16

2.1　XAML 是一种界面描述语言 ······ 16

2.2　XAML 的根元素 ······ 17

2.3　XAML 命名空间（NameSpace） ······ 17

2.4　XAML 和代码分离技术（code behind） ······ 18

2.5　子元素 ······ 19

2.6　相关属性（Dependency Property） ······ 20

2.7　附加属性（Attached Property） ······ 21

2.8　XAML 标记扩展 ······ 21

2.8.1　静态资源扩展（StaticResourceExtension） ······ 22

2.8.2　动态资源扩展（DynamicResourceExtension） ······ 23

2.8.3　数据绑定扩展（Binding） ······ 24

2.8.4　相对数据源扩展（RelativeSource） ······ 24

2.8.5　模板绑定（TemplateBinding） ······ 25

2.8.6　x:Type 扩展 ······ 26

2.8.7　x:Static 扩展 ······ 26

2.8.8　x:null 扩展 ······ 26

2.8.9　x:Array 扩展 ······ 26

2.9　本章小结 ······ 27

第 3 章 WPF 排版 28
3.1 排版基础 28
3.2 堆积面板（StackPanel） 29
3.3 WrapPanel 34
3.4 停靠面板（DockPanel） 35
3.5 表格式面板（Grid） 39
3.5.1 设定 UI 元素在 Grid 中的位置 40
3.5.2 设定 Grid 行或列的尺寸 40
3.5.3 元素横跨多个行列时的设定 41
3.5.4 在 Grid 中保持多行或多列大小的一致性 44
3.6 UniformGrid 46
3.7 画布面板（Canvas） 47
3.8 本章小结 48
第 4 章 WPF 中的属性系统 49
4.1 CLR 属性 49
4.2 相关属性的概念 50
4.2.1 相关属性的传递 50
4.2.2 WPF 对相关属性的支持 51
4.3 自定义相关属性 52
4.4 附加属性 58
4.5 本章小结 67
第 5 章 画笔和画刷 68
5.1 WPF 中的颜色 68
5.2 画刷 75
5.2.1 实心画刷（SolidColorBrush） 76
5.2.2 梯度画刷（GradientBrush） 77
5.2.3 线性梯度画刷（LinearGradientBrush） 77
5.2.4 圆形梯度画刷（RadialGradientBrush） 80
5.2.5 自制画刷（DrawingBrush） 81
5.2.6 粘贴模式（TileMode） 82
5.2.7 伸展方式（Stretch） 83
5.2.8 图像画刷（ImageBrush） 83
5.2.9 控件画刷（VisualBrush） 85
5.3 画笔 88
5.4 本章小结 95

第二篇　WPF专业程序员必备

第 6 章　WPF 控制 ······ 98

6.1　WPF 控件概述 ······ 98

6.2　内容控件（Content Control） ······ 100

6.2.1　框架控件（Frame） ······ 100

6.2.2　WPF 按钮（Button） ······ 101

6.2.3　拨动按钮（ToggleButton） ······ 104

6.2.4　CheckBox 控件 ······ 104

6.2.5　RadioButton 控件 ······ 104

6.2.6　重复按钮（RepeatButton） ······ 105

6.2.7　带有标题栏的内容控件（HeaderedContentControl） ······ 106

6.2.8　分组框（GroupBox） ······ 107

6.2.9　伸展控件（Expander） ······ 109

6.2.10　标签控件（Label） ······ 110

6.2.11　为按钮设置热键 ······ 111

6.2.12　ToolTip ······ 113

6.2.13　ScrollViewer ······ 115

6.3　条目控件（Items Controls） ······ 116

6.3.1　菜单（Menu） ······ 117

6.3.2　工具条（ToolBar） ······ 123

6.3.3　Selector ······ 126

6.3.4　组合框（ComboBox） ······ 126

6.3.5　TabControl ······ 129

6.3.6　列表框（ListBox） ······ 132

6.3.7　ListView ······ 135

6.3.8　状态条（StatusBar） ······ 138

6.3.9　树形控件 TreeView 和 TreeViewItem ······ 140

6.4　文本控件（Text Controls） ······ 143

6.4.1　口令输入框（PasswordBox） ······ 143

6.4.2　文字输入框（TextBox） ······ 144

6.4.3　RichTextBox ······ 145

6.5　范围控件（Range Controls） ······ 146

6.5.1　滚动条（ScrollBar） ······ 146

6.5.2　滑动条（Slider） ······ 147

6.5.3　进展条（ProgressBar） ······ 152

6.6　本章小结 ······ 152

第 7 章　传递事件和传递命令系统 ······ 153

7.1　WPF 中的元素树 ······ 153

7.2 传递事件（Routed Event）……165
7.2.1 RoutedEventArgs……166
7.2.2 终止事件传播……166
7.2.3 处理传递事件……167
7.2.4 附加传递事件（Attached Routed Event ）……168
7.3 考察传递事件……168
7.3.1 键盘事件的产生和传递……174
7.4 自定义传递事件……174
7.5 管理键盘和鼠标输入事件……182
7.5.1 键盘输入……182
7.5.2 鼠标输入……182
7.6 传递命令……184
7.6.1 ICommand 接口……186
7.6.2 ICommandSource 接口……186
7.6.3 CommandTarget……186
7.6.4 命令绑定（CommandBinding）……186
7.6.5 传递命令（Routed Command）……187
7.6.6 WPF 命令仓库（Command Repository）……187
7.7 本章小结……190
第 8 章 资源……191
8.1 资源定义及其类型……191
8.2 统一资源标识（Unified Resource Identifier）……192
8.3 .NET 开发平台对资源国际化的支持……196
8.3.1 WinForm 下的资源管理……197
8.3.2 用 XAML 创建本地资源……200
8.4 WPF 元素中定义的资源……202
8.4.1 静态资源（StaticResource）……203
8.4.2 资源的作用范围……204
8.4.3 静态扩展标识（Static markup extension）……205
8.4.4 动态资源扩展标识（DynamicResource Markup Extension）……208
8.5 本章小结……210
第 9 章 风格……211
9.1 Style 类……211
9.2 Setters……211
9.3 TargetType……215
9.4 BasedOn……218
9.5 触发器（Triggers）……220
9.5.1 使用单一条件的触发器……221

9.5.2 使用多个条件的触发器 ······ 222
9.5.3 使用数据触发器（DataTrigger） ······ 223
9.6 风格中的资源 ······ 225
9.7 IsSealed ······ 227
9.8 把风格定格定义在单独的文件中 ······ 227
9.9 在 FrameworkContentElement 中使用风格 ······ 228
9.10 再谈 Setter 属性 ······ 229
9.11 本章小结 ······ 230
第 10 章 模板 ······ 231
10.1 模板概述 ······ 231
10.2 控件模板 ······ 232
10.2.1 在控件中使用模板 ······ 232
10.2.2 在资源中使用模板 ······ 234
10.2.3 在控件模板中使用 TargetType ······ 235
10.2.4 在模板中显示控件的内容 ······ 236
10.2.5 在模板中使用 ContentPresenter ······ 237
10.2.6 模板中元素名 Name 属性 ······ 239
10.2.7 在模板中绑定控件的其他属性 ······ 239
10.2.8 使用模板显示电力系统的断路器和刀闸开关 ······ 240
10.2.9 在风格中使用模板 ······ 242
10.2.10 获取 WPF 控件的模板 ······ 243
10.3 数据模板（DataTemplate） ······ 244
10.3.1 我们所面临的问题 ······ 244
10.3.2 定义数据模板 ······ 247
10.3.3 在资源中使用数据模板 ······ 248
10.3.4 数据模板触发器 ······ 249
10.3.5 根据数据属性选择相应的模板 ······ 250
10.3.6 在数据模板中使用类型转换技术 ······ 253
10.4 ItemsPanelTemplate ······ 258
10.5 层次结构数据模板（HierarchicalDataTemplate） ······ 259
10.6 本章小结 ······ 262
第 11 章 数据绑定（Data Binding） ······ 263
11.1 数据绑定概述 ······ 263
11.2 最简单的数据绑定——从界面元素到界面元素 ······ 264
11.2.1 一对一数据绑定 ······ 264
11.2.2 在 C#中，实现数据绑定 ······ 265
11.2.3 对不是 FrameworkElement 和 FrameworkContentElement 元素实现数据绑定 ······ 266
11.3 使用不同的绑定模式 ······ 266

11.4 动态绑定 ············ 267
11.5 最简单的数据绑定——从.NET 对象到界面元素 ············ 268
11.6 DataContext ············ 271
11.7 控制绑定时刻 ············ 272
11.8 开发自己的 IValueConverter ············ 273
11.9 在数据绑定中加入校验 ············ 275
11.9.1 开发业务规则类 ············ 276
11.9.2 在绑定中添加任意多个业务规则 ············ 279
11.9.3 在控件上显示校验信息 ············ 279
11.9.4 触发错误处理事件 ············ 280
11.9.5 清除控件上的错误信息 ············ 282
11.10 对集合对象的绑定 ············ 283
11.10.1 使用 DisplayMemberPath 属性 ············ 286
11.10.2 显示当前条目 ············ 286
11.10.3 遍历集合中的记录 ············ 288
11.10.4 增加或删除记录 ············ 290
11.10.5 对集合对象分组 ············ 293
11.10.6 对集合对象排序 ············ 294
11.10.7 对集合对象过滤 ············ 295
11.11 数据源 ············ 296
11.11.1 XML 数据源 ············ 296
11.11.2 .NET 对象数据源 ············ 301
11.12 层次结构数据的绑定 ············ 303
11.13 本章小结 ············ 303
第 12 章 窗口对话框和打印 ············ 304
12.1 窗口（Window） ············ 304
12.1.1 窗口的状态变化和事件 ············ 304
12.1.2 确定视窗的位置 ············ 309
12.1.3 确定视窗的大小 ············ 310
12.1.4 视窗状态属性（WindowState） ············ 310
12.1.5 视窗大小模式（ResizeMode） ············ 310
12.1.6 视窗风格（WindowStyle） ············ 311
12.2 网页（Page） ············ 311
12.2.1 创建网页 ············ 312
12.2.2 KeepAlive 属性 ············ 312
12.2.3 NavigationService 属性 ············ 312
12.2.4 ShowsNavigationUI 属性 ············ 313
12.3 浏览窗口（NavigationWindow） ············ 313

12.3.1 使用统一风格 ……314
12.3.2 设置 NavigationWindow 的标题 ……314
12.3.3 浏览网页 ……315
12.3.4 使用 HyperLink 类 ……315
12.3.5 使用 NavigationService 转到不同的网页 ……318
12.3.6 使用浏览日志转换到不同的网页 ……319
12.3.7 浏览窗口的浏览事件 ……319
12.4 对话框（DialogBox） ……320
12.4.1 消息框（MessageBox） ……320
12.4.2 通用对话框 ……320
12.4.3 自定义对话框 ……322
12.5 打印输出 ……323
12.5.1 XPS 文档简介 ……323
12.5.2 创建 XPS 文档 ……324
12.5.3 显示 XPS 文档 ……328
12.5.4 打印 ……333
12.6 本章小结 ……333

第三篇　图形和动画

第 13 章　二维图形 ……336
13.1 WPF 图形系统概述 ……336
13.1.1 统一编程模型 ……336
13.1.2 坐标系统 ……338
13.1.3 Shape 和 Geometry ……338
13.2 Shape 及其派生类 ……339
13.2.1 直线（Line） ……340
13.2.2 矩形（Rectangle） ……340
13.2.3 椭圆（Ellipse） ……341
13.2.4 折线（Polyline） ……341
13.2.5 多边形（Polygon） ……342
13.2.6 填充规则（FillRule） ……342
13.2.7 路径（Path） ……343
13.3 Geometry 及其派生类 ……343
13.3.1 直线（LineGeometry） ……344
13.3.2 矩形（RectangleGeometry） ……344
13.3.3 椭圆（EllipseGeometry） ……344
13.3.4 几何图形组（GeometryGroup） ……345

13.3.5 合并图形（CombinedGeometry）……346
13.3.6 几何路径（PathGeometry）……348
13.3.7 分段路径（PathSegment）……350
13.3.8 弧线（ArcSegment）……350
13.3.9 直线段（LineSegment）……352
13.3.10 折线段（PolyLineSegment）……353
13.3.11 柏之线（BezierSegment）……353
13.3.12 多段柏之线（PolyBezierSegment）……354
13.3.13 二次柏之线（QuadraticBezierSegment）……354
13.3.14 多段二次柏之线（PolyQuadraticBezierSegment）……355
13.3.15 迷你绘图语言……356
13.3.16 流几何图形（StreamGeometry）……360
13.4 绘制（Drawing）……361
13.4.1 使用 DrawingImage 显示几何图形……362
13.4.2 使用 DrawingVisual 来显示几何绘制……363
13.4.3 创建 DrawingVisual 宿主……363
13.4.4 绘制几何图形……364
13.4.5 把 DrawingVisual 对象加到 FrameworkElement 中的视觉树和逻辑树中……364
13.4.6 选择视觉元素（Visual Hit Testing）……366
13.4.7 简单选择判断……366
13.4.8 多个视觉元素的选择判断……367
13.4.9 视觉元素重叠时的选择判断……367
13.5 本章小结……368
第 14 章 图形转换……369
14.1 图形转换概述……369
14.2 项目管理器……370
14.3 旋转转换（RotateTransform）……376
14.4 位移转换（TranslateTransform）……378
14.5 缩放转换（ScaleTransform）……380
14.6 扭曲转换（SkewTransform）……382
14.7 组合转换（TransformGroup）……384
14.8 矩阵转换（MatrixTransform）……385
14.8.1 矢量操作……385
14.8.2 H 坐标系……386
14.8.3 位移转换矩阵……387
14.8.4 旋转转换矩阵……388
14.8.5 缩放转换矩阵……388
14.8.6 扭曲转换矩阵……389

14.8.7 矩阵操作 ······ 389
14.9 本章小结 ······ 394
第 15 章 动画 ······ 395
15.1 WPF 中的动画 ······ 395
15.2 动画类继承树 ······ 396
15.3 一个简单的动画 ······ 397
15.4 控制动画 ······ 398
15.4.1 动画所用的时间（duration） ······ 399
15.4.2 设定动画开始时间 BeginTime ······ 399
15.4.3 设定自动返回（AutoReverse） ······ 399
15.4.4 设定动画速度（SpeedRatio） ······ 400
15.4.5 加快和减慢动画（AccelarationRatio 和 DecelarationRatio） ······ 400
15.4.6 设定动画的重复特性（RepeatBehavior） ······ 402
15.4.7 设定动画的终止状态（FillBehavior） ······ 402
15.4.8 设定相关属性的动画范围（From 和 To） ······ 402
15.4.9 设定相关属性的动画范围（By） ······ 403
15.4.10 设定 IsAdditive 和 IsCumulative 属性 ······ 403
15.4.11 WPF 动画的时间片类 ······ 403
15.5 故事板（Storyboard） ······ 404
15.5.1 使用故事板的一般格式 ······ 404
15.5.2 设定 Target 和 TargetName ······ 406
15.5.3 操作 Storyboard ······ 406
15.6 KeyFrame ······ 408
15.6.1 线性 KeyFrame ······ 409
15.6.2 非线性 KeyFrame（Spline KeyFrame） ······ 412
15.6.3 离散 KeyFrame（Discrete KeyFrame） ······ 414
15.7 本章小结 ······ 416

第四篇 开发WPF产品

第 16 章 多媒体技术及其应用 ······ 418
16.1 播放.wav 声音格式的 SoudPlayer 和 SoundPlayerAction ······ 418
16.1.1 装载.wav 文件 ······ 418
16.1.2 播放.wav 文件 ······ 419
16.1.3 停止播放 ······ 419
16.1.4 在 XAML 中使用 SoundPlayerAction ······ 419
16.2 播放多种格式的声音和图像 ······ 420
16.2.1 播放模式 ······ 421

16.2.2 使用 MediaPlayer 实例 ······ 422
16.2.3 使用 MediaElement 和 MediaTimeline 实例 ······ 426
16.3 语音合成和语音识别 ······ 430
16.3.1 尝试 Windows Vista 的语音功能 ······ 431
16.3.2 使你的程序发音 ······ 432
16.3.3 PromptBuilder 和 SSML ······ 433
16.3.4 语音识别中的语法 ······ 434
16.4 本章小结 ······ 436
第 17 章 定制控件和排版 ······ 437
17.1 用户控件和自定义控件 ······ 437
17.2 创建用户控件（User Control） ······ 439
17.2.1 设计用户控件 UI ······ 439
17.2.2 开发支持用户控件 UI 的逻辑 ······ 442
17.3 创建自定义控件（Custom Control） ······ 446
17.4 创建自定义排版（Custom Panel） ······ 459
17.4.1 照片浏览器 ······ 461
17.5 本章小结 ······ 471
第 18 章 综合应用 ······ 472
18.1 Ribbon 界面概览 ······ 472
18.2 项目的组织 ······ 473
18.3 管理 Generic.XAML 文件 ······ 475
18.4 开发自定义控件 ······ 476
18.4.1 自定义控件间的关系 ······ 476
18.4.2 Ribbon 按钮 ······ 477
18.4.3 Ribbon 分组（Group） ······ 483
18.4.4 RibbonTabItem ······ 492
18.4.5 RibbonApplicationMenuItem ······ 493
18.4.6 RibbonApplicationMenu ······ 495
18.4.7 RibbonQAToolBar ······ 496
18.4.8 RibbonBar ······ 497
18.4.9 RibbonWindow ······ 498
18.4.10 支持不同皮肤 ······ 514
18.5 使用 Ribbon 自定义控件实例 ······ 516
18.6 本章小结 ······ 525
参考文献 ······ 526

第一篇　WPF编程基础

第 1 章　WPF 应用程序

第 2 章　XAML 语言

第 3 章　WPF 排版

第 4 章　WPF 中的属性系统

第 5 章　画笔和画刷

第1章 WPF应用程序

本章讨论WPF应用程序的创建，运行及退出时的技术细节。涉及创建WPF应用程序的三个重要的类：Application、Window和Page、应用程序的异常处理、发布等话题。

1.1 WPF应用程序的创建

在Windows操作系统中，所有的应用程序都在自己独立的进程中运行，WPF也不例外。每个进程都有自己独立的内存地址，进程间的数据是隔离的，进程之间不会相互影响。当应用程序创建进程时，同时创建一个或多个线程。线程在自己的进程空间中运行，在底层，WPF仍然使用Windows的消息驱动机制来实现事务处理。

创建WPF应用程序从创建Application类开始。WPF的主要功能是人机交互，我们可以把当代人机界面（UI）程序归为两大类：一类是桌面（desktop）应用程序；一类是互联网（Web）应用程序。WPF首次实现了对这两类应用程序的统一编程。

让我们来创建一个简单的WPF程序：

```
using System;
using System.Windows;
namespace Yingbao.Chapter1
{
     public class HelloWPF
    {
        [STAThread]
        public static void Main()
        {
            Window win = new Window();
            win.Title = "WPF application";
            win.Content = "Hello WPF!";
            win.Show();
            Application app = new Application();
            app.Run();
        }
    }
}
```

首先，用Yingbao.Chapter1声明命名空间，并将在本书的所有例程中使用这一惯例。第1章的例程在Yingbao.Chapter1的命名空间中，第2章的例程将在Yingbao. Chapter2的命名空间中等。

然后，引入了.NET中的两个命名空间: System和System.Windows。 System命名空间提供.NET的基本服务，System.Windows中含有WPF的基本服务。Microsoft把WPF所用的类，都放在System.Windows的命名空间或其下面的命名空间中，如System.Windows.Controls、System.Windows.Input，等等。在Visual Studio的项目下，引入相应的Assembly, 这样就可以使用这些命名空间中的类了。

Main函数是WPF程序的入口点，它必须是静态（static）的。所有WPF程序，都必须在单一线程公寓(STA)模型中运行。STA来源于COM，若对此概念不熟悉，可以参考COM的相关著作或文章。在

Main函数前加上STAThread属性，就是满足WPF的这一要求，即UI不能在多线程的环境中运行。

在这个程序里，创建了一个Window类实例； Window类是管理WPF桌面应用程序的窗口类。其Title属性就是窗口标题，而Content属性是窗口中所要显示的内容。和.NET 1.0中的Form显示窗口一样，需要调用Show()方法，最后创建了Application类实例，并调用其Run() 方法。Run() 方法创建了和Win32一样的消息分配机制，它会接收操作系统发给应用程序的消息，并对相关消息进行响应。这个程序的运行结果如图1-1所示。

图1-1 Hello WPF!

你也可以不用调用Window类中的Show方法，而使用更为简洁的形式：

app.Run(win);

在这种情况下，Run方法将会自动调用win.Show()。

1.2 Dos窗口

当用Visual Studio创建WPF应用程序的时候，可以设置所创建的应用程序为带有Dos窗口的应用程序，还是桌面应用程序，方法如下：

在Visual Studio中，选择所要设定的项目，单击鼠标右键，选择“Property”，Visual Studio显示图1-2所示的窗口。在output type下拉控件里，可以选择Console Application、Windows Application和Class Library。

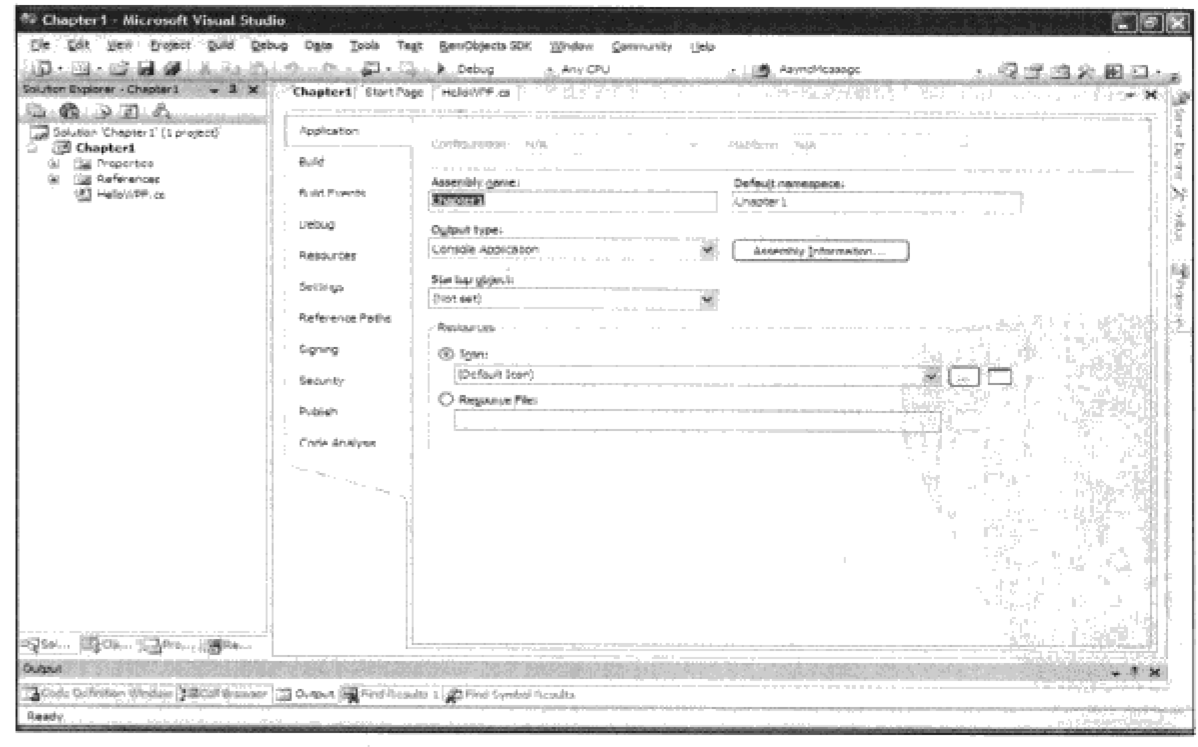


图1-2 在Visual Studio中设置应用程序项目的类型

有意思的是，当你选择Console Application时，WPF并不阻止你创建窗口，而是在创建桌面窗口的时候同时创建Dos命令行窗口，这一点和Java相似。

笔者觉得使用Dos命令行窗口来调试程序非常方便，比如在上面的这段程序里加入Console.WriteLine("…. "); 则这些信息会在Dos命令行窗口中显示。当调试好WPF程序之后，再把Output Type改为Windows Application。若项目是创建一个类库，就要把Output Type设为Class Library。

在一个进程中，只能创建一个Application实例，但是可以创建多个Window实例，每个Window实例就是一个可以显示的窗口。

1.3 WPF应用程序的启动和终止

表1-1列出了和应用程序生存期相关的事件和方法，通常在相关事件发生时，WPF首先调用Application类中的相应方法，然后再产生相关事件。所以，这些方法常常有相应的事件相对应。例如OnActivated方法和Activated事件、OnDeactivated方法和DeActivated事件，等等。

要考察WPF应用程序的启动和终止过程，有两种方法：其一是从Application类中派生出自己的类，并使用虚函数覆盖技术来观察Application类在运行时调用虚函数的过程；其二是处理Application类中的相关事件。

表1-1 Application类中与应用程序生存期相关的事件

方法名	事件	功能描述
OnActivated	Activated	在应用程序获得输入焦点时调用
OnDeactivated	DeActivated	在应用程序失去输入焦点前调用
OnSessionEnding	SessionEnding	在用户退出登录，或系统关机前调用
OnExit	Exit	在应用程序退出前调用
Shutdown		使应用程序退出
OnStartup	Startup	当调用Run方法时，系统调用此方法

让我们来看看第一种方法：

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Windows;
namespace Yingbao.Chapter1
{
   public class HelloWPF
   {
        [STAThread]
        public static void Main()
        {
           Window win = new Window();
           win.Title = "WPF application";
           win.Content = "Hello WPF!";
           Application app = new MainApp();
           app.Run(win);
        }
   }
   public class MainApp:Application
   {
      public MainApp()
         : base()
      {
```

```
        }
        protected override void OnActivated(EventArgs e)
        {
            Console.WriteLine("OnActivated");
            base.OnActivated(e);
        }
        protected override void OnExit(ExitEventArgs e)
        {
            Console.WriteLine("OnExit");
            base.OnExit(e);
        }
        protected override void OnDeactivated(EventArgs e)
        {
            Console.WriteLine("OnDeactivated");
            base.OnDeactivated(e);
        }
        protected override void OnStartup(StartupEventArgs e)
        {
            Console.WriteLine("OnStartup");
            base.OnStartup(e);
        }
        protected override void
        OnSessionEnding(SessionEndingCancelEventArgs e)
        {
            Console.WriteLine("OnSessionEnding");
            base.OnSessionEnding(e);
        }
    }
}
```

这段程序的运行结果如图1-3所示。

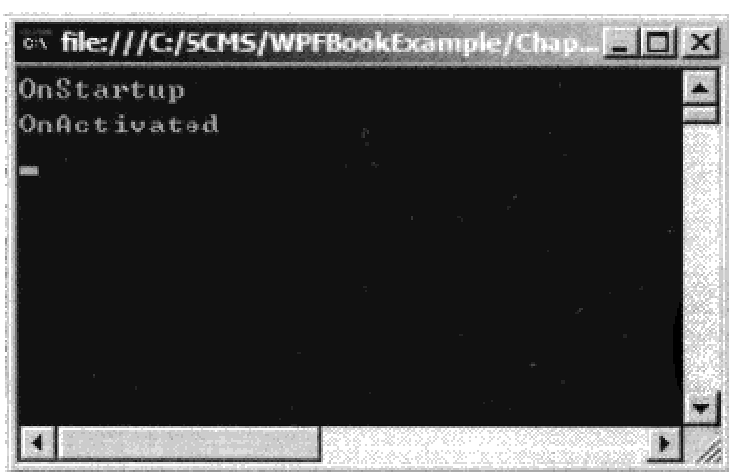


图1-3 考察Application中的事件

1.4 输入参数

在WinForm应用程序中，应用程序的入口函数是一个Main方法，我们可以向该方法传递一个命令行参数。WPF也可以用类似的方法，而且更加灵活。

方法1：在Application类中定义带参数的Main方法，如下面的DesktopApp：

```
namespace Yingbao.Chapter1.WpfAppWithInputParam
{
    public class DesktopApp: Application
    {
       [STAThread]
       public static void Main(string[] command )
       {
        //...加入你的程序
       }
    }
}
```

方法2：调用GetCommandLineArgs，如下面的Main中所用的方法：

```
namespace Yingbao.Chapter1.WpfAppWithInputParam
{
    public class DesktopApp: Application
    {
       [STAThread]
       public static void Main( )
       {
          string[] commandline = Environment.GetCommandLineArgs();
          foreach (string cmd in commandline)
          {
             Console.Write(cmd);
          }
       }
    }
}
```

方法3：移植OnStartUp方法

方法1和方法2都需要处理入口函数Main。我们在Visual Studio中创建WPF应用程序，通常使用项目模板，该项目模板创建的Application类由两部分组成，一部分是XAML，一部分是后台C#。WPF在编译时，会自动产生一个Main函数，即在这种情况下，我们不能定义自己的Main方法。这时候若要处理命令行参数，则要移植OnStartUp方法：

```
namespace Yingbao.Chapter1.WPFStartUp
{
    public partial class App : Application
    {
       protected override void OnStartup(StartupEventArgs e)
       {
          base.OnStartup(e);
          string[] commands = e.Args;
          foreach (string command in commands)
          {
             Console.Write(command);
          }
          //你自己的逻辑
       }
    }
}
```

1.5 在Xaml中创建Application

上面的例子是使用C#代码来创建Application的类。实际上在WPF应用程序中更常用的是使用Xaml来创建Application类实例。上面提到在Visual Studio里创建WPF应用程序通常使用项目模板，现在来考察使用Visual Studio项目模板创建WPF应用程序的过程。在File菜单中选择“New Project”，Visual Studio会显示图1-4所示的会话框，你可以选择WPF Application(桌面应用程序）或WPF Browser Application（Silverlight 应用程序）。

当选择WPF Application项目模板时，Visual Studio自动创建4个文件：App.xaml、App.xaml.cs、Window1.xaml和Window1.xaml.cs。这4个文件移植了两个类，App.xaml和App.xaml.cs创建的是App类，它从Application类中派生出来。Window1.xaml和Window1.xaml.cs创建的是Window1类，它从类Window中派生出来。让我们来看一下App.xaml：

```
<Application x:Class=" Yingbao.Chapter1.WPFStartUp.App"
   xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
   xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
   StartupUri="Window1.xaml">
   <Application.Resources>
   </Application.Resources>
</Application>
```

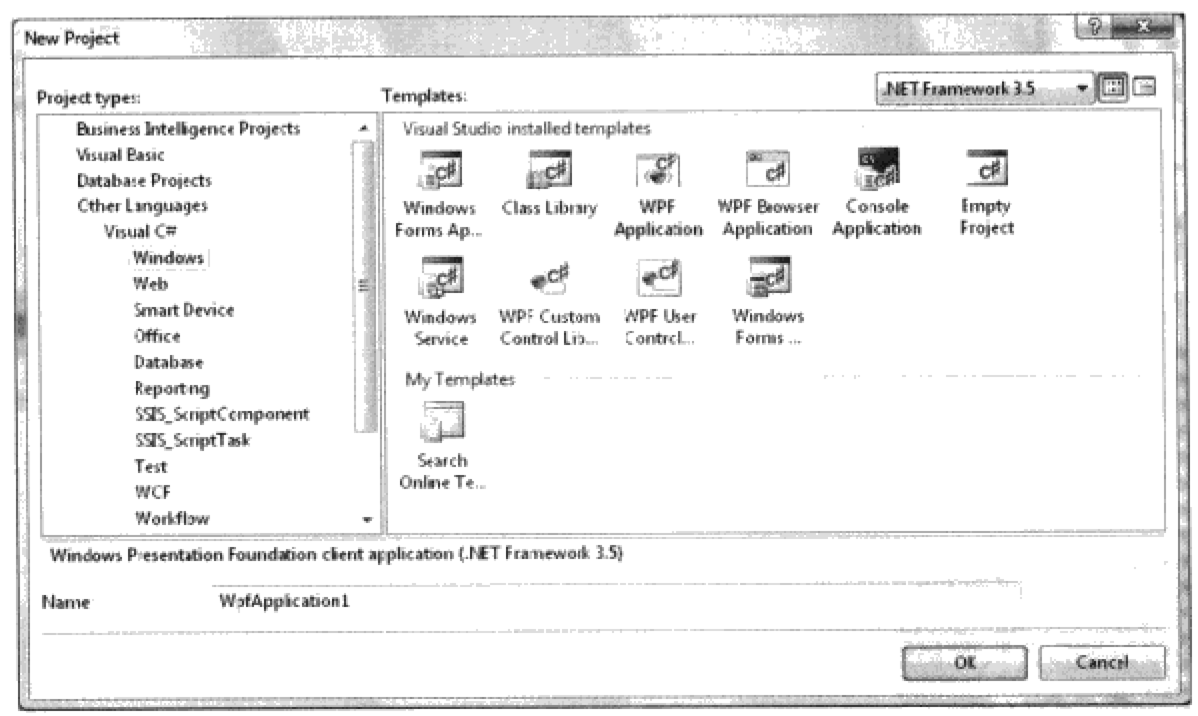


图1-4 Visual Studio中的项目模板

x:Class属性把xaml所对应的后台C#类联系起来，在这里是Yingbao.Chapter1.WPFStartUp.App，其中App是类名，Yingbao.Chapter1.WPFStartup是命名空间。App.xaml在编译后产生一个部分类，它位于所创建的项目目录下的obj\Debug子目录中，文件名为App.g.cs。这个文件中的内容如下：

```
public partial class App : System.Windows.Application {

       [System.Diagnostics.DebuggerNonUserCodeAttribute()]
```

```
        public void InitializeComponent() {

           #line 4 "..\..\App.xaml"
           this.StartupUri = new System.Uri("Window1.xaml",
            System.UriKind.Relative);

           #line default
           #line hidden
        }

        [System.STAThreadAttribute()]
        [System.Diagnostics.DebuggerNonUserCodeAttribute()]
        public static void Main() {
           Yingbao.Chapter1.WPFStartUp.App app = new
            Yingbao.Chapter1.WPFStartUp.App();
           app.InitializeComponent();
           app.Run();
        }
    }
```

它在App类中加入了两个方法，InitializeComponent和Main。Main方法和我们在前面直接创建HelloWPF类时所移植的Main方法类似，由于一个应用程序只能有一个入口点，所以在App.xaml.cs中不能再移植Main方法。在InitializeComponent()方法中，设置了StartupUri属性：

```
this.StartupUri = new System.Uri("Window1.xaml",
          System.UriKind.Relative);
```

它是对App.xaml中的StartupUri="Window1.xaml"翻译。

我们可以在App.cs中移植应用程序的逻辑， 下面是App.cs中的App部分类：

```
namespace Yingbao.Chapter1.WPFStartUp
{
   public partial class App : Application
   {
   }
}
```

Visual Studio为我们产生一个框架。

与App类似，Visual Studio产生的Window1.xaml，也是一个部分类，Window1的另一部分类在C#中。

```
<Window x:Class="Yingbao.Chapter1.WPFStartUp.Window1"
   xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
   xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
   Title="Window1" Height="300" Width="300">
   <Grid>

   </Grid>
</Window>
```

其中的x:Class属性也是把XAML和后台的C#类联系起来。

```
namespace Yingbao.Chapter1.WPFStartUp
{
  public partial class Window1 : Window
   {
      public Window1()
      {
         InitializeComponent();
      }
   }
}
```

它只产生一个构造函数Window1，其中调用了InitializeComponent方法，InitializeComponent在哪里呢？在xaml编译后产生的AppWin.g.cs文件中：

```
namespace Yingbao.Chapter1.WPFStartUp {
    public partial class AppWin : System.Windows.Window,
       System.Windows.Markup.IComponentConnector {

        private bool _contentLoaded;
       [System.Diagnostics.DebuggerNonUserCodeAttribute()]
        public void InitializeComponent() {
           if (_contentLoaded) {
              return;

           _contentLoaded = true;
           System.Uri resourceLocater = new
              System.Uri("/WPFStartUp;component/window1.xaml",
              System.UriKind.Relative);

           #line 1 "..\..\Window1.xaml"
           System.Windows.Application.LoadComponent(this,
                 resourceLocater);

           #line default
           #line hidden
        }

        [System.Diagnostics.DebuggerNonUserCodeAttribute()]

[System.ComponentModel.EditorBrowsableAttribute(System.ComponentModel.Ed
itorBrowsableState.Never)]

[System.Diagnostics.CodeAnalysis.SuppressMessageAttribute("Microsoft.Des
ign", "CA1033:InterfaceMethodsShouldBeCallableByChildTypes")]
        void System.Windows.Markup.IComponentConnector.Connect(int
connectionId, object target) {
           this._contentLoaded = true;
        }
    }
}
```

本书以下的章节都是以Visual StudioWPF项目模板所产生的代码为基础，小结一下：

- xaml中的x:Class属性把xaml和后台C#类联系起来。
- Visual Studio利用.NET 2.0引入的部分类（partial class）技术，把一个类分为两部分表述：一个是xaml文件，一个是C#文件。编译后，xaml文件会产生一个后缀为.g.cs的C#文件，是xaml所产生的部分类。
- Application中的StartupUri设置了应用程序启动时的主窗口。

1.6 窗口大小

Window类中的Width和Height属性确定窗口的大小，可以在xaml或C#中设置该属性：

```
<Window x:Class="Yingbao.Chapter1.WPFStartUp.Window1"
  ...
Height="300" Width="300"> <!-- ( XAML ) -->
```

或：

```
public partial class Window1 : Window
    {
        public AppWin()
        {
            InitializeComponent();
                 this.Height = 300;
            this.Width = 300;
        }
    }                                                    // (C#)
```

在上面这段程序中，我们把窗口的高度和宽度都设为300。你也许会问，300这个数字代表多大？首先，300并不是屏幕上的300个点(pixel)， 否则Width和Height属性就不会是浮点数。WPF用的单位是1/96英寸。这个单位和显示器上实际的图素点没有关系，因此上面的300×300(3.125英寸×3.125英寸)窗口在1024×768和1920×1080分辨率的显示器上显示的结果是一样的。

1.7 互联网应用程序

在WPF中创建互联网应用程序和创建桌面程序是一样的。在桌面应用程序里，XAML的顶层元素为Window，在互联网应用程序里，XAML的顶层元素为Page。例如下面的网页：

```
<Page
   xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
   xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
   Title="Page1">
   <Grid>
      <Label Height="28" Margin="80,108,100,0" Name="label1"
          VerticalAlignment="Top">Hello WPF!</Label>
    </Grid>
</Page>
```

该网页在IE 7 中的显示结果如图1-5所示。

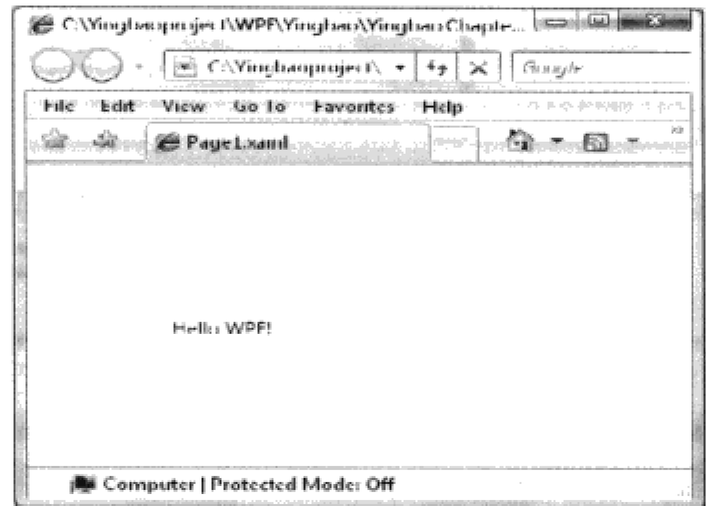


图1-5 IE 7中显示的WPF网页

1.8 应用程序的异常处理

现代软件工程有一个基本的出错处理原则：就近处理程序异常。比如说，你有如下Class A和Class B两段程序：

```
public Class A
{
 //...
       protected  float Divide( float x, float y )
     {
       float z = x/y;
       return z;
     }
}

public Class B: A
{
   //...

   public void Calc()
   {
        float[]  x={100.2, 22.5,.....};
        float[]  y =( 500.0, 2.5,....};
        for( int i = 0; i < x.Length; i++ )
        {
               z= divide( y[i],x[i]);
               //...
        }
   }
}
```

这两段程序在正常的情况下，没什么问题，但如果Class A中的y为0，那么z=x/y就会出错，一般情况下，我们需要把Class A中的方法Divide改为：

```
protected float Divide(float x, float y)
       {
          float z = 0;
          try
          {
              z = y / x;
          }
```

```
            catch
            {
                //处理异常
            }
            return z;
        }
```

即我们需要在最可能出现异常的地方加上try{} catch{}，从而对程序异常加以处理。 这一原则说起来简单，做起来难，一旦有一个地方没有对异常进行适当的处理，可能会导致整个软件，甚至整个操作系统的崩溃。WPF的Application类中有一个事件：DispatcherUnhandledException，这个事件在应用程序未对其中的异常加以处理的情况下发生。我们可以对该事件进行处理，从而为应用程序把好最后的大门：

```
protected override void OnStartup(StartupEventArgs e)
        {
            ...
            this.DispatcherUnhandledException += new
            System.Windows.Threading.
             DispatcherUnhandledExceptionEventHandler(
             App_DispatcherUnhandledException);
        }
        void App_DispatcherUnhandledException(object sender,
                System.Windows.Threading.
                DispatcherUnhandledExceptionEventArgs e)
        {
        //对程序异常进行处理
         }
```

1.9 应用程序中的资源

WPF应用程序，除了WinForm中的资源文件外，还有其自身所特有的资源，如模板、风格等都可以放到资源中，本书的第8章将对资源进行详细的讨论。FrameWorkElement中有一个相关属性：Resources, 可以把资源放在其中。由于FrameworkElement是WPF中非常重要的基类，所有从FrameworkElement中派生出来的类都继承了Resources属性，其中可以根据需要加入相应的资源。

例如，当使用Visual Studio模板创建程序时，其中含有Application. Resources标记，在这个标记内可以放入各种WPF资源：

```
<Application x:Class=" Yingbao.Chapter1.WPFStartUp.App"
   xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
   xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
   StartupUri="AppWin.xaml">
   <Application.Resources>
   </Application.Resources>
</Application>
```

同样也可以在Windows.Resources中放入窗口级的资源：

```
   <Window.Resources>
    /Window.Resources>
```

Application.Resources中的资源在整个应用程序中都是可见的，Window.Resource中的资源仅在该窗口内可见。一个Application中可以含有多个窗口或网页，一个窗口或网页中又可含有多个控件，区分不同层次的资源是很重要的。

1.10 应用程序的发布

当应用程序开发到一定的程度，我们希望能够在一个相对独立于开发的环境下测试。这里相对独立于开发的环境可以是同一台机器上的不同子目录，也可以是不同的机器。过去我们一般把相关的软件手动复制到目的地，现在Visual Studio为WPF软件提供了简单的发布程序的方法。在Visual Studio的solution explore窗口，选择要发布的项目，单击鼠标右键，选择“Publish…”菜单项，Visual Studio就会启动发布向导，该向导由三个对话框组成，每个对话框上都有 “Previous” 和“Next”按键，用户可以方便地在这些对话框间切换。第一个对话框如图1-6所示：

在这个对话框里，你要说明的是所要发布的路径。这里的路径可以是本机硬盘上的某个节点，也可以是局域网上的某个节点，或FTP服务器，甚至还可以是网站。第二个对话框如图1-7所示。

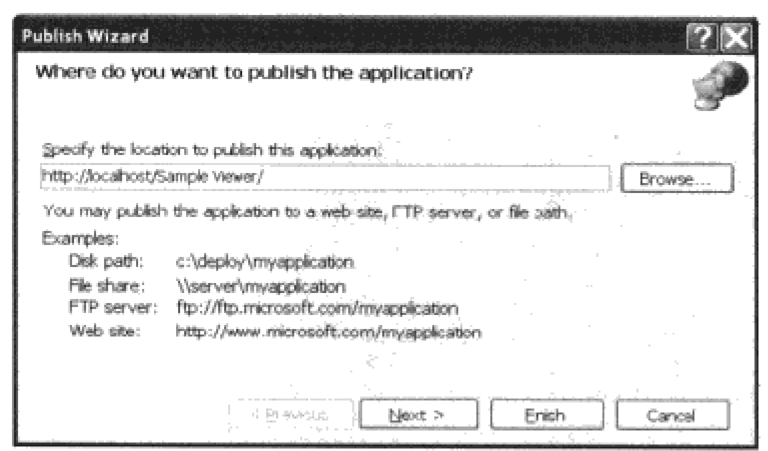


图1-6 发布向导（1）

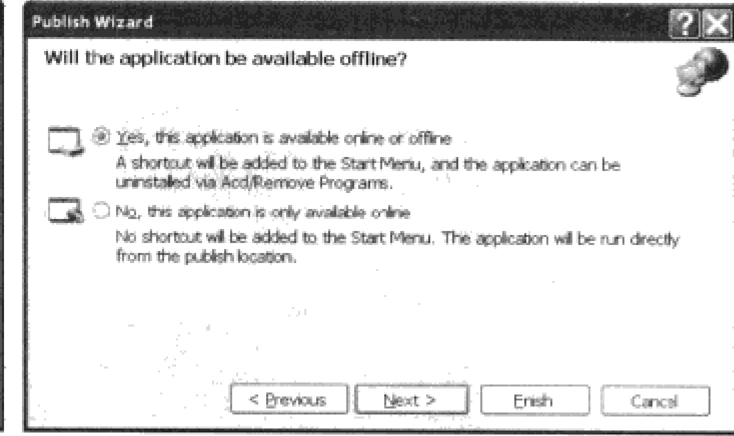


图1-7 发布向导（2）

这个对话框让你说明是否要在操作系统的开始菜单上为WPF应用程序创建一个入口。第三个对话框如图1-8所示。

第三个对话框显示小结信息，这个时候单击“Finish”按键，那么你的WPF应用程序就自动发布到目的地。

在目标目录下，WPF生成了一个“publish.htm”，如图1-9所示。

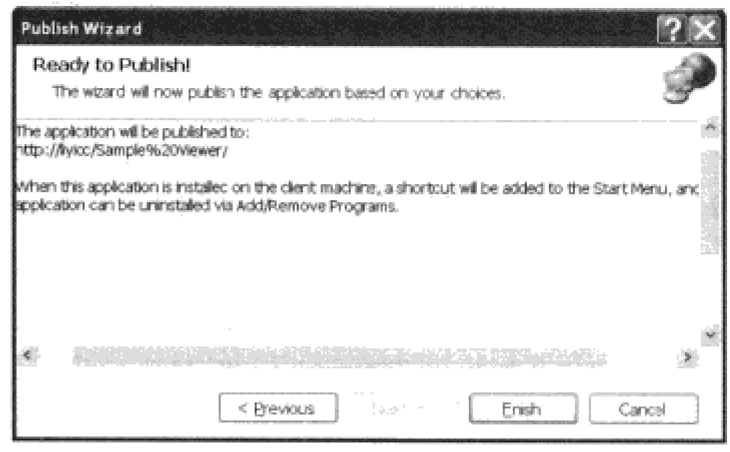


图1-8 发布向导（3）

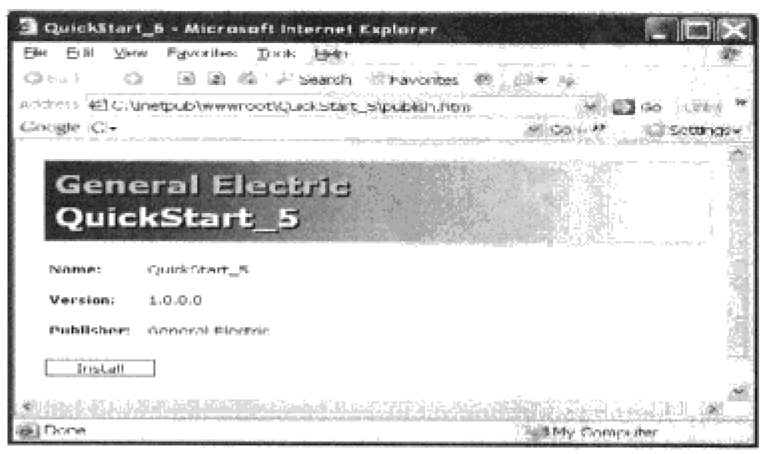


图1-9 发布向导（4）

1.11 WPF 开发环境

从理论上说开发WPF只要使用.NET framework 3.5 及其以上的版本即可，但实际上，需要使用Visual Studio 2005 或Visual Studio 2008以提高你的工作效率。如果有Microsoft Expression Blend，就可以使用Expression Blend 提供的工具产生XAML，从而获得事半功倍的效果。

本书的例子均在Visual Studio上运行，并且开发WPF应用程序，Visual Studio是必须的。

Express blend

Express Blend的主界面如图1-10所示，上面是菜单，左边是作图工具，右边是Project, Property和资源。Blend对项目的组织和Visual Studio一样，可在Blend中打开Visual Studio所产生的项目，也可以在Visual Studio里打开Blend所产生的项目。

Blend的下半部为交互部分，可以对各个控件加入事件处理、触发器（Trigger)和动画，使用Blend设计动画是十分方便的。

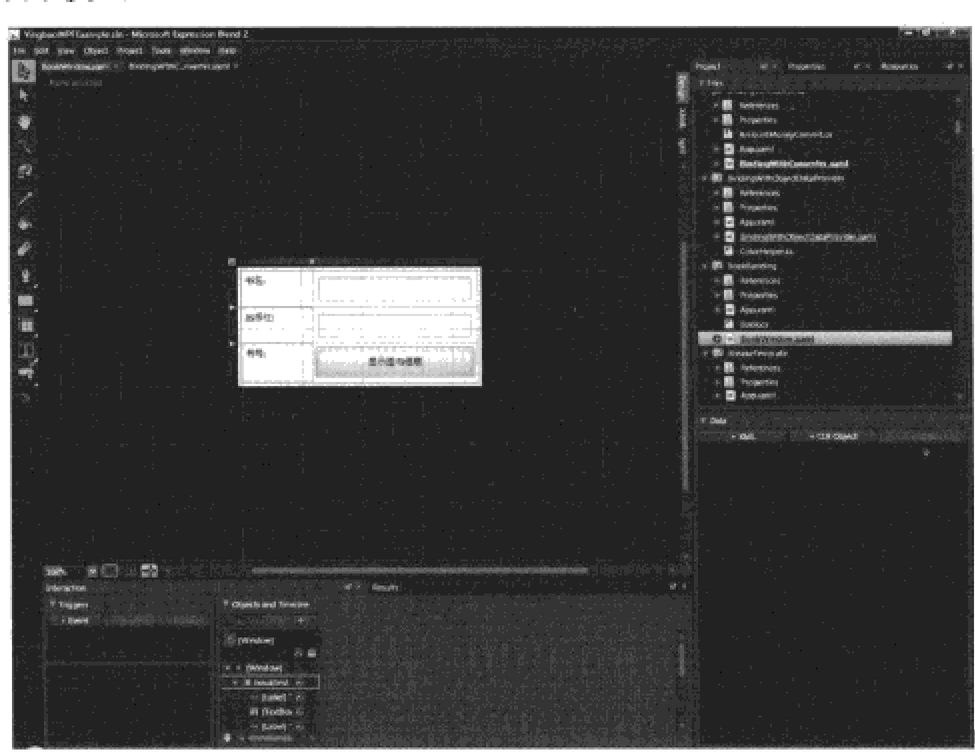


图1-10 Express Blend主界面

使用Blend的另外一个好处是，可以很方便地操作界面元素的属性。 在设计图上选中对象，然后在右边的面板上选择“Property”，就可以很方便地设置所选择的界面元素的属性，并同时看到其在UI上的效果。在Blend上产生直线和梯度画刷也非常方便。

有两种方法可以观察Blend所产生的XAML，其一是在画布上选择“XAML”Tab，这时画布上整个显示XAML；其二是选择“Split”，这时同时显示XAML和所设计的画面，如图1-11所示。

Blend在设计风格和模板时也非常有用，总之，如果你要开发大型WPF应用程序，或者有专门的界面设计人员，Blend会提高工作效率。笔者在设计本书的例子时，有时用到了Blend，但为了不增加读者的负担，并不要求读者安装Blend（blend软件并不便宜）。

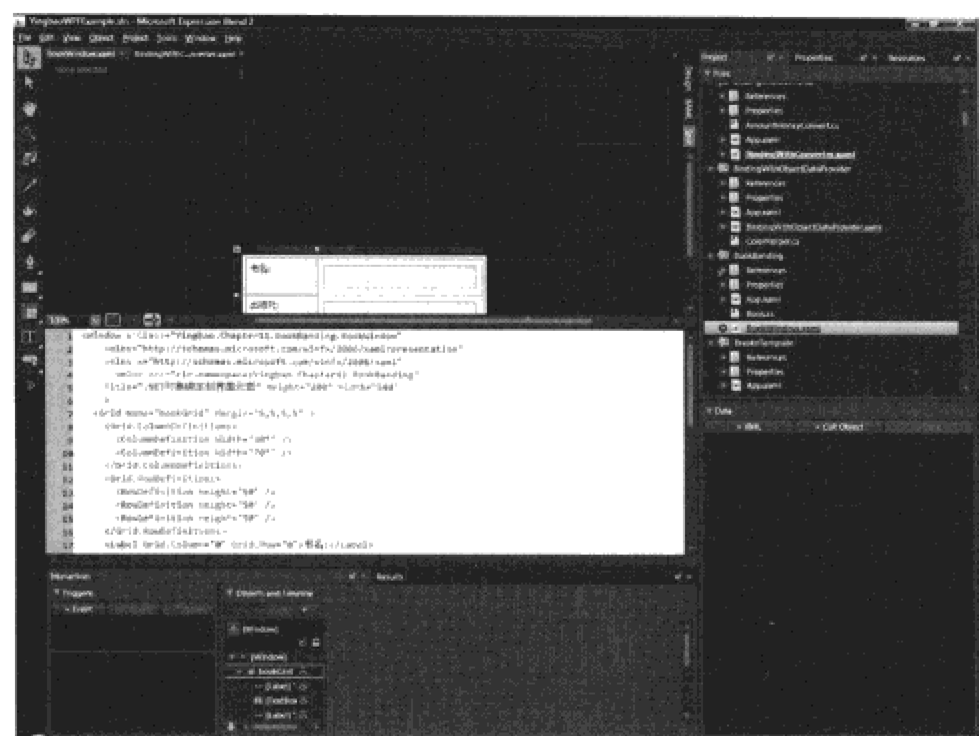


图1-11 Express Blend产生XAML

1.12 本章小结

本章介绍了WPF程序的创建及其组成部分，初步接触了XAML、事件处理资源及发布应用程序。需要指出的是，这里的发布应用程序并不能代替用户的最终安装程序；当然对于程序员来说，提供了一种简单的测试环境。WPF的典型开发环境是使用Expression Blend和Visual Studio，但学习本书，只要在Visual Studio 2005 安装.NET 3.0或3.5，或使用Visual Studio 2008即可。

第2章 XAML语言

.NET 技术平台3.0引入了一种新的语言——XAML，这是一种基于XML的标记语言。我们知道，互联网中广泛应用的HTML或XHTML语言就是标记语言，成千上万的网页都是用这种语言写的。.NET有一个设计目标，就是统一桌面应用程序和互联网应用程序的编程。ASP.NET使用后台编码技术达到了在.NET平台上的统一编程，但就人机界面而言，真正实现桌面应用程序和互联网界面的统一编程是从.NET 3.0开始的，而其核心就是应用XAML语言。微软在创建WPF及XAML时，虽然在业界有打击竞争对手的作用，如基于WPF的Silverlight技术和XPS规范针对的是Flash和PDF。但实际上，实现统一桌面和互联网界面编程，可以极大地缩短开发应用程序的开发周期，从而减少投资。

本章将概要地介绍XAML语言，讨论XAML语言的主要特征，以及XAML的各个细节。

2.1 XAML是一种界面描述语言

在大型软件工程中，通常涉及两类不同性质的工作。一类是用户界面设计人员，他们关心的是软件和用户之间的交互；另一类是软件开发人员，他们关心的是软件功能的实现。在互联网中，用户界面设计人员使用HTML及其工具来设计界面，开发人员使用Java，C#，VB或其他语言来实现其中的逻辑，HTML网页可以用到最终的产品中。

在桌面应用程序中，过去我们一直没有分开这两种不同性质的工作。用户界面设计人员通常和开发人员使用不同的工具，当界面设计人员设计好用户界面时，他们的工作并没有用到最终的产品中，而只是用来展现某种概念或工作流程。

XAML实现了互联网应用程序和桌面应用程序的统一，界面设计人员可以使用XAML或基于XAML的工具（如微软的Blend）来设计桌面或互联网应用程序的界面。程序开发人员则可以在此基础上使用C#或VB.NET来开发相应的功能，这样，界面设计人员的工作便自然过渡到最终产品中。

在XAML中，用户界面用XML的元素或属性来表示。WPF引擎把XAML描述的UI元素解释为相应的.NET对象，从而在桌面程序或Silverlight网页上创建相应的控件。

WPF编程模型实际上是XAML标记语言和逻辑编程语言（C#或VB.NET）的混合。XAML的节点，属性以及相互关系用来描述界面元素及其相互关系。例如，若要创建一个按钮，则可以用下面的XAML：

```
<Button Name="btn1" Background="Pink" BorderBrush="Black"
BorderThickness="1" Click="OnClick1" > 按钮 </Button>
```

XAML中的元素名为CLR中的类名，如上例中的Button，它实际上是WPF中的Button类。XAML的属性是相应类中的相关属性，如上例中的Name、Background、BorderBrush等实际上是Button类中相应的相关属性（有关相关属性的概念，将在第4章详细介绍）。在这句XAML中，我们还放置了事件处理程序，Click="OnClick1"，即XAML支持声明事件处理程序，具体逻辑在C#或VB.NET的OnClick1方法中。

XAML中的元素名和CLR中的类名是一一对应的，其中的属性也是CLR类中的属性。

2.2 XAML的根元素

我们知道XML文件总是从一个单一元素开始的，在这个单一元素的里面可以放置任意个子元素。子元素中又可以包含其他子元素，如此下去，整个XML文件就像一棵倒挂的树。开始的这个单一元素就叫做根元素，XAML也遵循XML这一规范。

通常XAML的根元素有两个：一个是Window，说明这是一个桌面应用程序；另一个是Page，主要用在互联网应用程序中，有时候桌面应用程序也用到Page。

```
<Window x:Class="Yingbao.Chapter2.Window1"
   xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
   xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
   Title="送元二使安西" Height="300" Width="300" >
<StackPanel  Orientation ="Vertical"  >
  </StackPanel.LayoutTransform>
 </StackPanel>
</Window>
```

上面这段XAML为桌面应用程序，其根元素为Window。

```
<Page x:Class="Yingbao.Chapter2.MyFrirstWebPage"
xmlns=http://schemas.microsoft.com/winfx/2006/xaml/presentation
xmlns:x=http://schemas.microsoft.com/winfx/2006/xaml Title="网页"
Height="80" Width="300">
<StackPanel Orientation ="Horizontal " >
  </StackPanel>
</Page>
```

上面这段为XMAL网页，它的根元素为Page。

由此可见，XAML网页和桌面程序的区别只是XAML根元素的不同（Silverlight 在使用CLR对象时目前还有些限制，但从大的方面来说两者的编程模型是一样的）。

XAML编译器并不限制根元素的类型，在一般WPF应用程序中，常见的根元素除了上面的Window和Page之外，还有FlowDocument，Application和Grid等。

2.3 XAML命名空间（NameSpace）

长期以来，各大公司或个人开发了大量面向对象的软件。在C#或C++里，为了区分模块间的同名类，我们引入命名空间。本书采用的命名空间规则是笔者本人的名字加上章节名。比如第3章中的例子，都放在Yingbao.Chapter3这个命名空间中；而第4章中的例子，都放在Yingbao.Chapter4这个命名空间中；这样，即使有两个在第3章和第4章同名的类，比如TreeView。那么可以用Yingbao.Chapter3.TreeView和Yingbao.Chapter4.TreeView来区分用的究竟是哪个类。

同样的道理，在XAML中也使用命名空间这个概念。 XML的命名空间，则用xmlns来表示，例如前面的例子：

```
<Window x:Class="Yingbao.Chapter2.Window1"
   xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
   xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"    Title="
   送元二使安西" Height="300" Width="300" >
```

在这个例子中，使用了两个命名空间，一个是http://schemas.microsoft.com/winfx/2006/xaml/presentation；另一个是http://schemas. microsoft.com/winfx/2006/xaml。注意，XAML中的命名空间和.NET的命名空间密切相关，但XAML的命名空间和.NET的命名空间之间并不是一一对应的，而是一对多的关系，即一个XAML命名空间对应多个.NET的命名空间。这样做的好处是，不必在XAML中书写过多的命名空间。一般来说在XAML里使用这两个命名空间就包括了WPF中所有的命名空间。

这种让一个XAML命名空间对应多个.NET命名空间的做法不仅微软可以用，任何软件开发人员也都可以使用。比如我可以把Yingbao.Chapter2和Yingbao.Chapter3两个命名空间合并为一个XAML命名空间，方法是在项目的AssemblyInfo.cs文件中使用XmlnsDefinition属性：

```
[assembly: XmlnsDefinition("http://Yingbao.Com/WPFExample",
           "Yingbao.Chapter3")]
[assembly: XmlnsDefinition("http://Yingbao.Com/WPFExample",
           "Yingbao.Chapter4")]
```

这里我把Yingbao.Chapter3和Yingbao.Chapter4两个.NET命名空间合并到http://Yingbao.Com/WPFExample这一个命名空间中。注意：这里的http://Yingbao.Com并不是真的有一个这样的网址。

WPF使用这一技术，把所有WPF类命名空间映射到一个XAML命名空间：

```
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
```

注意xmlns后面没有冒号，这表示WPF在XAML中默认命名空间。另一个WPF中常用的命名空间是：xmlns:x=http://schemas.microsoft.com/ winfx/2006/xaml，它包含了XAML的某些特定功能，比如控制XAML编译器等。

如果要在XAML中使用非WPF命名空间中的类，那么你就要在XAML中引入相应的命名空间，例如，我们要在Window中使用ADO.NET中的类，我们可以直接在XMAL中引入System.Data命名空间：

```
<Window x:Class="Yingbao.Chapter2.Window1"
   xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
   xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
   xmlns:ado="clr-namespace:System.Data;assembly= System.Data,
   Version=2.0.0.0,Culture=Meutral,PublicKeyToken=b77a5c561934e089"
   Title="送元二使安西" Height="300" Width="300" >
```

可以使用同样的方法在XAML中引用笔者自己开发的类，例如在本书第11章中的例子：

```
<Window x:Class="Yingbao.Chapter11.BookBanding.BookWindow"
    xmlns=http://schemas.microsoft.com/winfx/2006/xaml/presentation
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:src="clr-namespace:Yingbao.Chapter11.BookBanding"
    Title=".NET对象绑定到界面元素" Height="200" Width="340" >
```

这里xmlns:src就是在XAML中引入Yingbao.Chapter11.BookBanding命名空间。

2.4 XAML和代码分离技术（code behind）

和ASP.NET一样，XAML也使用代码分离技术。WPF应用程序一般由两大部分组成，一部分用XAML描述UI元素在界面上的位置，大小等；另一部分用来处理程序的逻辑、对传递事件的反应，等等。

让我们来看一看前面的例子：

```
<Window
   xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
   xmlns:x=" http://schemas.microsoft.com/winfx/2006/xaml"
   x:Class="Yingbao.Chapter2.Window1"
   Title="送元二使安西" Height="300" Width="300" >
```

XAML在编译这个XAML时，在Yingbao.Chapter2命名空间中创建一个Window1的类。这是由XAML中的x:Classs说明的，前缀x是XAML的命名空间 （xmlns: x= "http://schemas.microsoft.com/winfx/2006/xaml "）。属性x:Class确定要产生的类名，如这里的Window1。其次，XAML在产生类Window1时，把元素名Window作为Window1的基类。若我们在Visual Studio里创建XAML，Visual Studio会自动产生下面的部分类代码：

```
namespace Yingbao.Chapter2
{
     public partial class Window1 : System.Windows.Window
    {
        public Window1 ()
        {
            InitializeComponent();
        }
    }
}
```

显然，Window1从Window类中派生出来。

在这个例子中，若我们省去x:Class属性，那么根元素就是Window类，而不是Window1类。代码分离技术提倡我们使用派生类。

2.5 子元素

在XAML中，除了根元素之外的所有元素都是子元素。根元素只有一个，而子元素理论上可以有无限多个。子元素又可以包含一个或多个子元素，某个元素可以含有子元素的多少由WPF中具体类决定。排版类元素可以包含多个子元素，而内容控件只能含有一个子元素。让我们来看一个例子：

```
<Window x:Class="Yingbao.Chapter2.DockPanelProperties"
   xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="停靠面板属性" Height="500" Width="600"  >
  <DockPanel Background="White">
    <TextBlock FontSize="16" FontWeight="Bold" DockPanel.Dock="Top"
               Margin="20,0,0,10">停靠面板属性</TextBlock>
    <TextBlock DockPanel.Dock="Top" Margin="0,0,0,10"
      TextWrapping="Wrap">
      选择下面的控件及其停靠属性，观察控件在停靠控制面板中的位置
    </TextBlock>
  </DockPanel>
  </Window>
```

在这个例子中，Window是XAML的根元素，其中含有DockPanel这个子元素。由于Window是一

个内容控件，它只能含有一个子元素。DockPanel是一个排版控件，它可以含有任意多个子控件，这里，我在DockPanel中放入了两个TextBlock。

如果我们用C#来写这些元素之间的关系，上面的程序可以改写为：

```
namespace Yingbao.Chapter2
{

    public partial class DockPanelProperties :
              System.Windows.Window
    {
        public DockPanelProperties()
        {
            InitializeComponent();
            DockPanel dp = new DockPanel();
            dp.Background = "White";
            this.Content = dp;
            TextBlock tb = new TextBlock();
                                  // 设置tb属性
                                  dp.Children.Add(tb);
            TextBlock tb = new TextBlock(); //下一个TextBlock
                                  // 设置tb属性
                                  dp.Children.Add(tb);
        }
    }
}
```

从C#程序，我们可以清楚地看出Window、DockPanel和TextBlock之间的关系。有关Window、DockPanel和TextBlock，本书后面章节将会详细介绍，在这里，只要了解XAML中子元素之间的关系就可以了。

2.6 相关属性（Dependency Property）

XAML中元素的属性大部分为相关属性，有关相关属性的概念，将在第4章进行详细介绍。这里，先讲述XAML中相关属性的表示方法。一种表示方法是：

```
  <DockPanel Background="White">
  ...
</ DockPanel>
```

这里的Background就是DockPanel中的相关属性，也可以用第二种表示方法：

```
  <DockPanel>
     <DockPanel.Background>White </ DockPanel.Background >
  ...
</ DockPanel>
```

笔者比较喜欢第一种，因为它比较简洁。但有时必须要用第二种。比如，若要用梯度画刷来绘制DockPanel的背景：

```
<DockPanel>
   <DockPanel.Background>
```

```
        <LinearGradientBrush StartPoint ="0,0" EndPoint ="0,1">
          <GradientStop Color="Red" Offset="0.0" />
          <GradientStop Color="Orange"  Offset="0.17" />
          <GradientStop Color="Yellow " Offset="0.33" />
          <GradientStop Color="Green" Offset="0.5" />
          <GradientStop Color="Blue" Offset="0.67" />
          <GradientStop Color="Indigo" Offset="0.84" />
          <GradientStop Color="Violet" Offset="1" />
        </LinearGradientBrush>
      </DockPanel.Background>
</DockPanel>
```

这时，必须使用第二种表示方法。有关梯度画刷，将在第5章详细讨论。

2.7 附加属性（Attached Property）

WPF属性系统引入了附加属性的概念，有关附加属性将在第4章详细讨论，这里介绍附加属性在XAML中的表示。并列举一个使用网格（Grid）排版的例子：

```
<Grid Name="MyGrid" Background="Wheat"
 ShowGridLines="False">
      <Grid.ColumnDefinitions>
      <ColumnDefinition/>
      <ColumnDefinition/>
      <ColumnDefinition/>
    </Grid.ColumnDefinitions>
    <Grid.RowDefinitions>
      <RowDefinition/>
      <RowDefinition/>
      <RowDefinition/>
    </Grid.RowDefinitions>
 <TextBox  Name ="InputText"  Grid.Column ="0" Grid.Row ="0"
    Grid.ColumnSpan ="9" FontSize ="12" FontWeight =
         "DemiBold" Margin ="5,2,10,3"/>
 <Button Name="B7" Click="DigitBtn_Click"  Grid.Column="1"
          Grid.Row="2" Margin ="2">7</Button>
 <Button Name="B8" Click="DigitBtn_Click"  Grid.Column="1"
         Grid.Row="2" Margin ="2">8</Button>
 <Button Name="B9" Click="DigitBtn_Click"  Grid.Column="1"
        Grid.Row="2" Margin ="2">9</Button>
</Grid>
```

Grid中的Column和Row都是附加属性。Grid中的控件在说明其位置时，直接设置该附加属性。写法如上面的XAML，TextBox和Button并不含有Grid.Row或Grid.Column属性，却可以很方便地应用Grid中的附加属性。

2.8 XAML标记扩展

有时候我们要在XAML里引用静态或动态对象实例，或在XAML中创建带有参数的类。这时，我们需要用到XAML扩展。XAML扩展常用来设定属性值。

标记扩展本身是一系列类，其基类为MarkupExtension，这是一个抽象类。从这个类中派生出十

二个类，即：ResourseKey、TypeExtension、StaticExten-sion、BindingBase、RelativeSourse、ColorConvertedBitmapExtension、Dyna-micResourceExtension、ArrayExtension、NullExtension、StaticResource-Extension、Template-BindingExtension和ThemeDictionaryExtension。其中ResourceKey、BindingBase等类又派生出其他的一些类。这些标记扩张可分为两大类：

- WPF标记扩展。这类扩展包括：StaticResource、DynamicResource、Binding、RelativeSource和TemplateBinding。
- XAML本身定义的标记扩展。这类扩展包括：x:Type、x:Static、x:null和x:Array。

在语法上，XAML使用大括号{}来表示扩展。例如，下面这句XAML：

```
<TextBlock Text="{Binding Source={StaticResource myDataSource}, Path
          =PersonName}"/>
```

这里有两处使用了XAML扩展，一个是Binding，另一个是StaticResource，这种用法又称为嵌套扩展，TextBlock元素的Text属性的值为{}中的结果。当XAML编译器看到大括号{}时，把大括号中的内容解释为XAML标记扩展。

必要时，你也可以使用自己的扩展， 其方法是从MarkupExtension中派生出你自己的标记扩展类，并覆盖基类中的ProvideValue方法。

2.8.1 静态资源扩展（StaticResourceExtension）

前面已经用了StaticResource标记扩展，这个扩展用来获取静态属性值，这是XAML编译器在编译时完成的。为了让读者感受一下这种用法，可以看一个完整的例子：

```
<Window x:Class="Yingbao.Chapter2.SimpleBindingEx.AppWin"
   xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:sys="clr-namespace:System;assembly=mscorlib"
    Title="Static Markup Extension" Height="100" Width="200">
    <Window.Resources>
        <sys:String x:Key="personName">李白</sys:String>
    </Window.Resources>
    <Grid>
        <TextBlock Width="200" Height="50" Text="{Binding
             Source={StaticResource personName}}"/>
    </Grid>
</Window>
```

C#代码是Visual Studio产生的：

```
namespace Yingbao.Chapter2.SimpleBindingEx
{
    public partial class AppWin : Window
    {
        public AppWin()
        {
            InitializeComponent();
        }
    }
}
```

笔者在窗口的资源部分定义了一个简单的字符串，注意字符串位于.NET平台的System命名空间中；因此，在window标记中加了xmlns:sys。这段XAML的运行结果如图2-1所示。

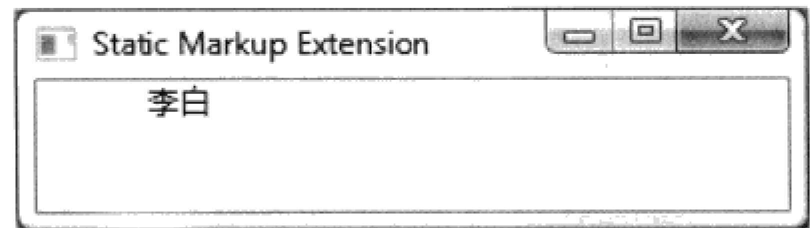


图2-1 使用静态资源标记扩展

2.8.2 动态资源扩展（DynamicResourceExtension）

动态资源扩展和静态资源扩展相对，其区别是获取资源的值一个是在编译时完成的（静态），另一个是在运行时完成的。当使用动态资源扩展时，若该资源属性在运行时发生了变化，那么其获取的值也会发生相应的变化。让我们来看一个例子：

```
<Window x:Class="Yingbao.Chapter2.DynamicResourceEx.AppWin"
   xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
   xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
   Title="动态资源扩展" Height="100" Width="300">
   <StackPanel>
        <TextBlock Name="exText" Background="
          {DynamicResource {x:Static
             SystemColors.ActiveCaptionBrushKey}}" Height="30"
             FontSize="24">清泉石上流
         </TextBlock>
    </StackPanel>
</Window>
```

在这个例子中，笔者利用动态资源扩展把TextBlock的背景色设置为具有输入焦点的视窗标题的背景色。上述XAML的运行结果如图2-2所示。

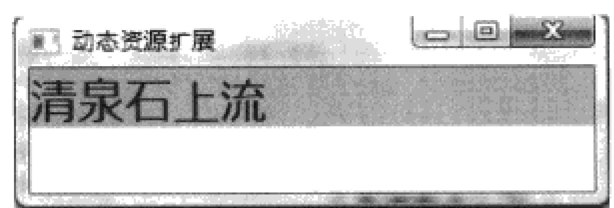


图2-2 使用动态资源扩展

接下来，笔者在运行这段程序的同时，来改变Vista的“Appearance Settings”。方法是在“Control Panel”中，选择“Appearance and Personalization / Custom Color/Open classic appearance properties for more color options”，设置Vista窗口属性的方法如图2-3所示。

把Color Scheme设置为“High Contrast #2”时的结果如图2-4所示。

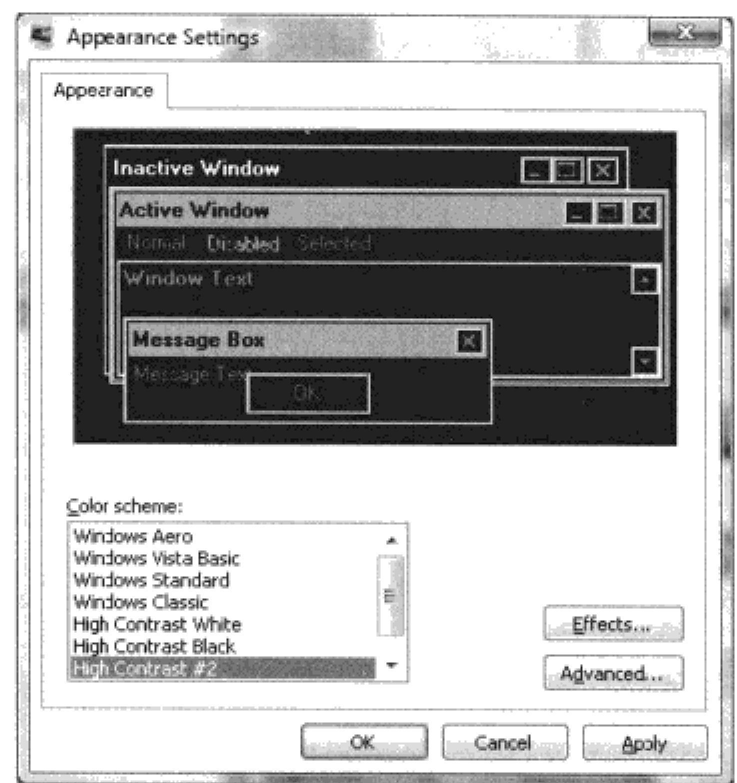


图2-3 设置Vista窗口属性

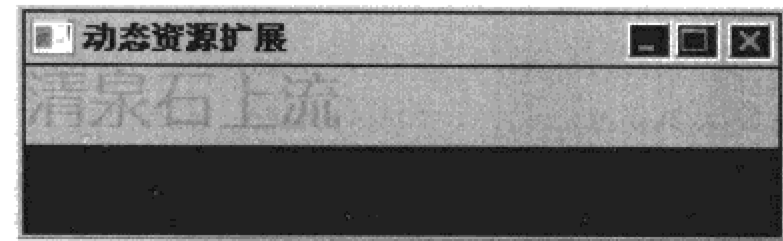


图2-4 TextBlock背景色被动态置换

2.8.3 数据绑定扩展（Binding）

Binding扩展专门在XAML中用作数据绑定，将在第11章中详细讨论WPF中的数据绑定。现在来看一个简单绑定的XAML语法：

```
<Window x:Class="Yingbao.Chapter2.SimpleUIBanding.AppWin"
   xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="绑定" Height="373" Width="337">
    <StackPanel>
      <ScrollBar Name="scroll"  Orientation ="Horizontal " Margin
        ="24" Maximum ="100" LargeChange ="10" SmallChange ="1"/>
      <Label Height ="30">OneWay:</Label>
      <TextBox Name="scrollValue1" Height ="20" Width ="200"
          HorizontalAlignment="Center"  Text ="{Binding
            ElementName=scroll, Path=Value, Mode=OneWay}"/>
```

使用Binding扩展，设定了Mode、Path，等等属性。

2.8.4 相对数据源扩展（RelativeSource）

在数据绑定时，有时候需要把目标对象绑定到和目标对象自身有相关关系的对象上，这时就需要用到相对数据源扩展。例如：

```
<Window x:Class="Yingbao.Chapter2.RelativeEx.AppWin"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="相对绑定" Height="100" Width="300">
    <StackPanel Orientation="Horizontal"
                HorizontalAlignment="Center">
      <TextBlock FontSize="20" Text="{Binding
```

```
            RelativeSource={RelativeSource self}, Path=FontSize}"/>
        <TextBlock Margin="10,1,1,5"  FontSize="20" Text="{Binding
            RelativeSource={RelativeSource AncestorType={x:Type
            StackPanel}}, Path=Orientation}"/>
    </StackPanel>
</Window>
```

在这个例子中，笔者使用了两个相对数据源扩展，第一个TextBlock的Text绑定到自身的字体大小上；第二个TextBlock的Text则绑定到其父节点StackPanel的Orientation属性上。这段XAML的运行结果如图2-5所示。

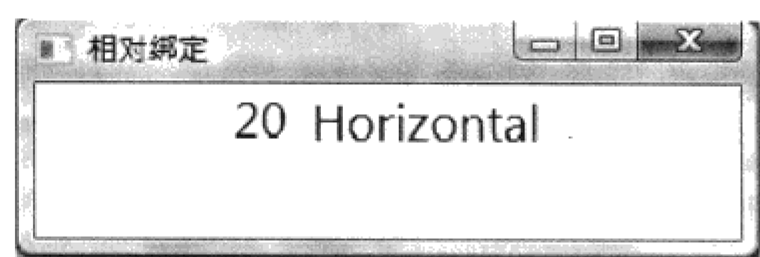


图2-5 使用相对数据源

关于数据绑定，本书将在第11章详细讨论。

2.8.5 模板绑定（TemplateBinding）

WPF提供强大的模板功能，本书第10章，将详细介绍WPF模板。模板绑定扩展是用来把原对象中的属性和模板对象中的属性联系起来。请看下面的例子：

```
<Window x:Class="Yingbao.Chapter2.TemplateBindingEx.AppWin"
   xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="模板绑定扩展" Height="134" Width="226">
    <Window.Resources>
        <ControlTemplate TargetType="{x:Type Button}"
                x:Key="buttonTemp">
            <Border BorderThickness="3" Background
                   ="{TemplateBinding Foreground}">
                <TextBlock Name="txtblk" FontStyle ="Italic"
                    Text="{TemplateBinding Content}"
                    Foreground="{TemplateBinding Background}"/>
            </Border>
        </ControlTemplate>
    </Window.Resources>
    <StackPanel Orientation="Vertical" Height="362" Width="394">
        <Button Height="50"  Foreground="Red" FontSize="24"
           Template="{StaticResource buttonTemp}">按钮1</Button>
        <Button Height="50"  Foreground="Yellow" FontSize="24"
           Template="{StaticResource buttonTemp}">按钮2</Button>
    </StackPanel>
</Window>
```

在ControlTemplate中，笔者使用TemplateBinding扩展把TextBlock的前景色设置为控件的背景色。

2.8.6 x:Type扩展

x:Type扩展在XAML中取对象的类型， 相当于C#中的typeof操作， 这种操作发生在编译时刻，如前面的例子中的：

```
<ControlTemplate TargetType="{x:Type Button}"
                 x:Key="buttonTemp">
```

它等价于下面的C#语句：

```
ControlTemplate tmplt = new ControlTemplate();
tmplt.TargetType = typeof(Button);
.....
```

注意：类型名字和命名空间有关。

2.8.7 x:Static扩展

静态扩展用来把某个对象中的属性或域的值赋给目标对象的相关属性。这个扩展总是而且只有一个参数，这个参数就是源对象的属性。例如：

```
<TextBlock Name="exText" Background="
          {DynamicResource {x:Static
             SystemColors.ActiveCaptionBrushKey}}" Height="30"
             FontSize="24">清泉石上流
        </TextBlock>
```

这是前面见过的例子。{ x:Static SystemColors.ActiveCaptionBrushKey} 静态扩展取的ActiveCaptionBrush值。

2.8.8 x:null扩展

x:null扩展是一种最简单的扩展，其作用就是把目标属性设置为null。例如：

```
<TextBlock Name="exText" Background="{x:null} />
```

这句XAML把TextBlock的Background属性设置为null。如果用C#来重写这句，那就是下面这段程序：

```
TextBlock tb= new TextBlock();
tb.Name = "exText";
tb.Background = null;
```

在WPF中，把相关属性设置为null会打断相关属性的继承链，关于这一点，将在本书第4章详细讨论。

2.8.9 x:Array扩展

Array扩展就是在XAML里创建一个数组。使用Array扩展创建数组很容易，但在词法上和其他XAML扩展不同，它不需要使用大括号"{}"，原因在于Array里面含有多个元素。一个整数数组的例子如下：

```
<Window x:Class="Yingbao.Chapter2.ArraryEx.AppWin"
```

```
   xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
   xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
   xmlns:sys="clr-namespace:System;assembly=mscorlib"
   Title="在XAML中使用数组" Height="151" Width="244">
    <Window.Resources>
        <x:ArrayExtension Type="{x:Type sys:Int32}"
            x:Key="numArray">
            <sys:Int32>10</sys:Int32>
            <sys:Int32>20</sys:Int32>
            <sys:Int32>30</sys:Int32>
            <sys:Int32>40</sys:Int32>
        </x:ArrayExtension>
    </Window.Resources>
    <Grid>
        <ListBox ItemsSource="{StaticResource numArray}">
            <ListBox.ItemTemplate>
                <DataTemplate>
                    <TextBlock Text="{Binding}" Width="100"
                    Height="30" Margin="4"/>
                </DataTemplate>
            </ListBox.ItemTemplate>
        </ListBox>
    </Grid>
</Window>
```

在这段XAML中，笔者用ListBox列出了数组numArray，其运行结果如图2-6所示。

图2-6 在XAML中创建数组

2.9 本章小结

本章概要介绍了XAML的语法及其简单的应用，XAML语言是开发WPF应用程序常用的语言。编写XAML可以用简单的NotePad程序，就像使用HTML一样。XAML背后有强大的.NET开发平台支持，从而实现互联网和桌面应用程序的统一编程。对于界面设计人员来说，可以使用某些工具，如Blend来产生XAML程序。在后面的章节中，将详细讨论WPF所引入的新概念。

第3章 WPF排版

WPF吸取了HTML、Office及Java等多种软件的排版经验，开发了专门管理版面的类，可以很方便地使用这些类来布置用户界面上的元素。这种做法和Win32或 WinForm中的界面设计有根本的区别，写网页的程序员可能觉得这是非常自然的事，但对于桌面开发人员来说，则需要转换思路。本章将讨论与排版相关的WPF类：StackPanel、WrapPanel、DockPanel、Grid、UniformGrid、Canvas等。

3.1 排版基础

WPF使用控制面板来进行排版，控制面板实际上是一种可以放入WPF界面元素的容器。当用户把界面元素放入控制面板后，WPF会自动把这些界面元素放在它认为合适的地方。WPF开发人员需要根据自己对用户界面的要求来选择合适的控制面板，因而，理解这些控制面版并正确地使用它们的功能，就成为开发者的首要任务。此外，WPF控制面板本身也可以放到控制面板中，因此可以灵活地根据需要来对控制面板进行组合，从而可以从这些基本的控制面板出发，来获得多种不同的版面设计。

WPF中的基本控制面板类如图3-1所示，这些类都是从Panel类中派生出来的，Panel本身是UIElement。控制面板的概念来自于Java和HTML，过去，在Windows操作系统中开发桌面应用程序，无论是使用Delphi、Visual Basic，还是C++，在视窗中放置控件都使用绝对坐标，.NET中的WinForm用的也是绝对坐标。WPF排版仍然支持绝对坐标这种机制，但同时引入了新的版面布置类。图3-1中的Canvas就是使用绝对坐标的控制面板，使用Canvas，便可以确切地制定控件在界面上的x、y坐标。

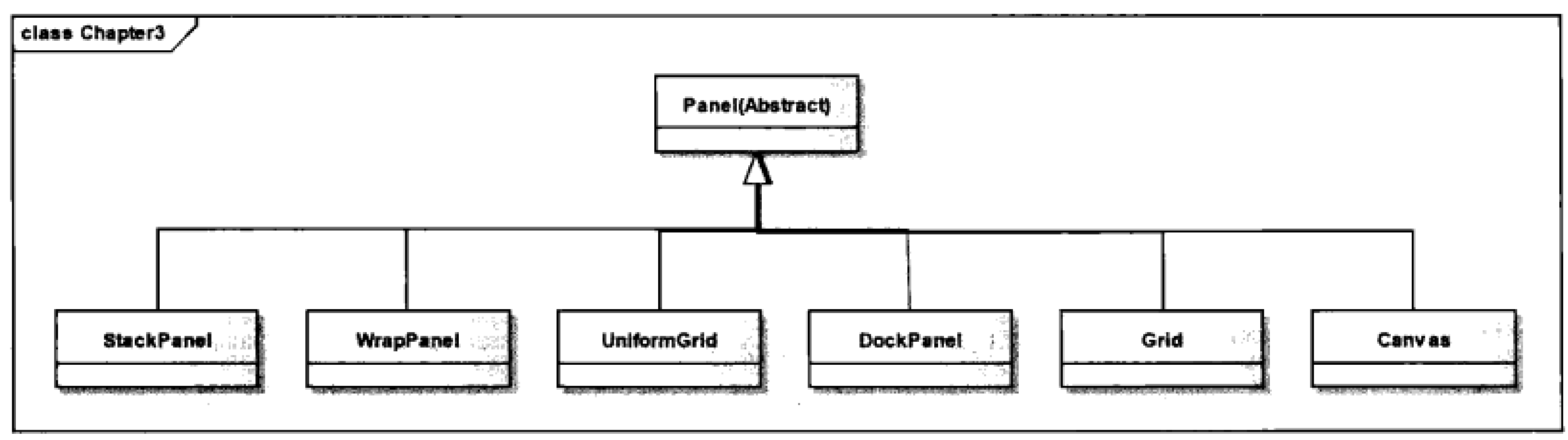


图3-1 WPF中的控制面板类

使用绝对坐标在某些情况下会有问题。比如会话框，一般在其资源文件中定义其中控件的位置。当软件需要支持多种语言的时候，是不能简单地在软件中替换字符串就能切换到另一种文字的，往往需要对每个UI元素的位置进行调整。结果就需要不同的资源文件来支持不同的语言。WPF中的StackPanel、DockPanel、WrapPanel及Grid则支持另外一种排版机制，使用这些排版类，不需要设置控件在视窗上的绝对位置，只需要说明其相对位置，WPF的控制面板类负责最终确定这些控件的位置。

控制面板中的Children属性是一个UIElement的集合，即所有从UIElement中派生出来的UI元素都可以加入到控制面板中，也就是说，控制面板中既可以放入各种图形，又可以放入像Button、

TextBlock这样的常规控件。由于Panel本身是从UIElement中派生出来的，所以Panel可以含有Panel。

在本章及以后的章节中将接触到各种控制面板，首先来讨论最简单的控制面板——StackPanel。

3.2 堆积面板（StackPanel）

StackPanel是最简单的一种控制面板，它把其中的UI元素按横向或纵向堆积排列。现列举一个例子，在这个例子中，笔者用的是王维的七绝《送元二使安西》。首先，在XAML中使用StackPanel：

```
<Window x:Class="Yingbao.Chapter3.Window1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="送元二使安西" Height="300" Width="300">

  <StackPanel  Orientation ="Vertical"  >
    <TextBlock   FontSize="16" Foreground="Brown">渭城朝雨浥轻尘,</TextBlock >
    <TextBlock   FontSize="16" Foreground="Brown">客舍青青柳色新.</TextBlock >
    <TextBlock   FontSize="16" Foreground="Brown">劝君更尽一杯酒,</TextBlock >
    <TextBlock   FontSize="16" Foreground="Brown">西出阳关无故人.</TextBlock >
    <Button Background ="Coral" Click ="OnVertical">竖排</Button>
  </StackPanel>
</Window>
```

在上面的这段程序里，笔者在StackPanel里加了四个TextBlock控件，其内容为王维的四句诗，最后加了一个Button控件，上面这段XAML的运行结果如图3-2所示。

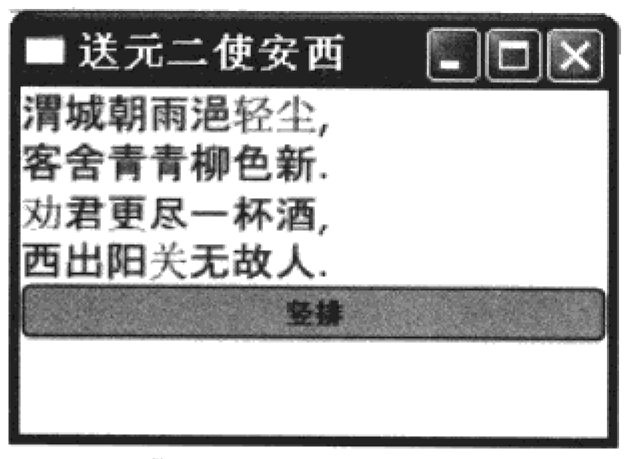


图3-2 StackPanel示例

注意：在这段程序中，并没有指定TextBlock或Button的绝对位置坐标，而是在StackPanel中把Orientation属性设为纵向（Vertical），StackPanel就会自动地把其中的UI元素按照先后次序堆积起来，先进入StackPanel的元素放在最上面，后进入StackPanel的元素放在最下面。

Orientation的属性有两个值：Vertical和Horizontal。当笔者把Orientation属性设为Horizontal时，StackPanel会自动地把其中的元素从左到右一次排列。当Orientation设为 Vertical时，StackPanel中的元素占满StackPanel的水平方向空间，即其中元素的宽度Width是相同的。当Orientation设为Horizontal时，StackPanel中的元素则占满垂直方向空间，其中元素的高度Height都相同。

也可以用C#来写这一段程序，StackPanel和其中元素间的父子关系（这里的父子关系指的是一种包容关系，而不是继承关系）更加清晰：

```
namespace Yingbao.Chapter3
{
    using System;
    using System.Windows;
    using System.Windows.Controls;
    using System.Windows.Input;
    using System.Windows.Media;

   public class StackLayoutCShap:Window
    {
        public StackLayoutCShap()
        {
            this.Title = "送元二使安西";
            StackPanel sp = new StackPanel();
            this.Content = sp;
            sp.Children.Add(CreateTextBlock("渭城朝雨浥轻尘"));
            sp.Children.Add(CreateTextBlock("客舍青青柳色新"));
            sp.Children.Add(CreateTextBlock("劝君更尽一杯酒"));
            sp.Children.Add(CreateTextBlock("西出阳关无故人"));
            Button btn = new Button();
            btn.Height = 40;
            btn.Content = "竖排";
          btn.Background = new SolidColorBrush(Color.FromRgb(200,100,100));
            btn.Click += new RoutedEventHandler(OnVertical);
            sp.Children.Add(btn);
        }
        private TextBlock CreateTextBlock(string text )
        {
            TextBlock tb = new TextBlock();
            tb.FontSize = 16;
            tb.Foreground = new SolidColorBrush(Color.FromRgb(200, 100,
                100));
            tb.Text = text;
            return tb;
        }

        private void OnVertical(object sender, RoutedEventArgs rea)
        {
        }
    }
```

StackLayoutCShap类是从WPF的Window类中派生出来的，在其构造函数中，笔者首先创建StackPanel实例，然后把Window类的Content属性设为该StackPanel实例。在CreateTextBlock方法里，笔者创建了TextBlock实例，并设置字体及颜色。用CreateText Block方法创建TextBlock实例，该实例被加入到StackPanel的Children集合中。之后，创建一个Button实例并加入到StackPanel的Children集合中。

使用XAML和C#所达到的效果是一样的，比较C#程序和XAML程序，可以发现XAML更为简洁，在大多数多情况下都是如此；所以，在下面的章节中将主要用XAML来创建WPF界面。

有时候需要把视窗的大小根据其中的内容来自动调整，方法是设置视窗的SizeToContent属性．SizeToContent是枚举类型，表3-1列出了其可取的值及其意义。

表3-1 SizeToContent值及其意义

SizeToContent值	描 述
Manual	视窗的大小不随其中内容的变化而变化，而由其他的属性如Width、Height等来确定
Width	视窗的宽度由WPF自动根据其中的内容来调整
Height	视窗的高度由WPF自动根据其中的内容来调整
WidthAndHeight	视窗的宽度和高度都由WPF自动根据其中的内容来调整

若你对StackPanel和窗口间或UI元素与UI元素间的距离不满意，可以调整StackPanel或界面元素的Margin属性。

UI元素Margin属性的意义如图3-3所示，Margin的类型为Thickness，可以对其左、右、上、下分别设置。WPF内部对Margin的赋值操作进行了重载，若只指定一个数值，WPF会自动把left、right、top、bottom都相应地设为该值，如：

Margin=5 （XAML)

或：

Margin = new Thickness(5); （C#)

这时UI元素的四周Margin都被设为5个单位。当然也可以分别指定Margin的上下左右的值,如：

Margin="3,0,0,0" (XAML)

或：

Margin.Left=10.1；（C#）

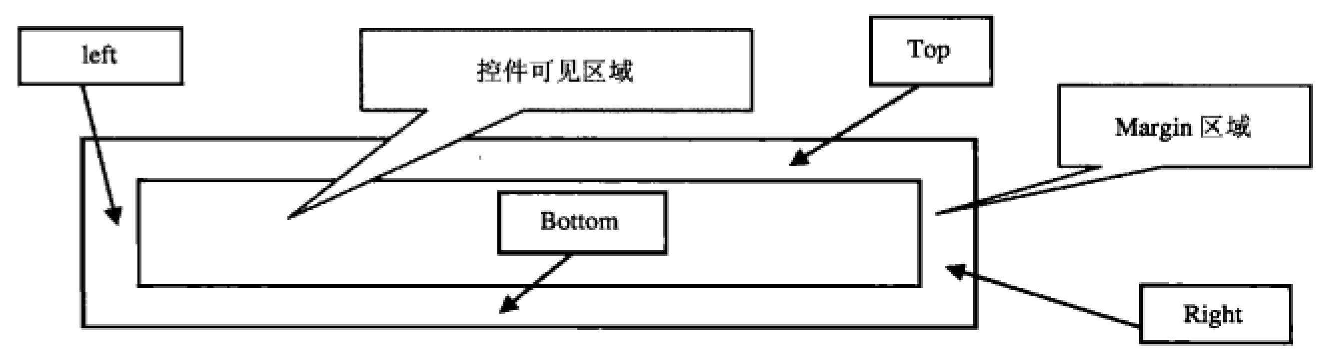


图3-3 WPF中UI元素的Margin属性

UI元素的另外一个属性是Padding。Padding和Margin一样也是Thickness类型。两者的区别是Margin定义UI元素边界外面的矩形区域，而Padding定义的是UI元素边界内部的区域。由于Padding和Margin都是Thickness类型，所以两者在XAML和C#中使用的语法一样。

上面谈到当视窗里的内容比视窗小时，可以用设置视窗的SizeToContent的属性让视窗根据其中的内容进行自动调整。但更常遇到的是另外一种情况：视窗里的内容比计算机可用屏幕的尺寸还要大，这时候就要在视窗内加滚动条。WPF提供ScrollBar和ScrollViewer来实现屏幕滚动，笔者认为ScrollViewer比ScrollBar在滚动视窗中的内容时用起来方便。在视窗中加入ScrollViewer对象的例子如下：

```
<Window x:Class="Yingbao.Chapter3.StackPanelWithScrollViewer"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
```

```
    Title="Yingbao.Chapter3" Height="200" Width="300">
  <ScrollViewer>
    <StackPanel Name ="addButtonSp" Margin ="5">
      <Button FontSize ="20" Foreground ="Blue"
          Click="OnButtonClick">在窗口里加入按钮  </Button>
    </StackPanel>
  </ScrollViewer>
  </Window>
```

在上面的这段XAML程序里，笔者在Window对象中加入了ScrollViewer,，然后在ScrollViewer里加入StackPanel，运行这段程序，可以看到视窗里只显示一按钮，而且视窗的滚动条是灰色的（Disabled），如图3-4所示。

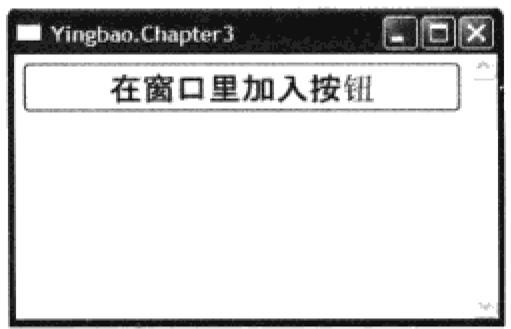


图3-4 ScrollViewer示例（1）

现在，在后台C#中，加入按钮Click事件的响应逻辑：

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Data;
using System.Windows.Documents;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Media.Imaging;
using System.Windows.Shapes;
namespace Yingbao.Chapter3
{
    public partial class StackPanelWithScrollViewer : System.Windows.Window
    {

        public StackPanelWithScrollViewer()
        {
            InitializeComponent();
        }

        private void OnButtonClick(object sender, RoutedEventArgs rea)
        {
            Button btn = new Button ();
            btn.Background = Brushes.LightBlue ;
            btn.Foreground = Brushes.Red;
            btn.Content = "新加入的按钮";
```

```
                addButtonSp.Children.Add(btn);
            }
        }
}
```

OnButtonClick方法是对按钮事件的响应，当用户单击“在窗口里加入按钮”的按钮时，此方法被调用。在这个方法里，笔者创建了一个新的名为“新加入的按钮”的按钮，并把它加入到StackPanel中。运行该程序，并单击名为“在窗口里加入按钮”的按钮，可以看到每单击一次鼠标，视窗里就有一个“新加入的按钮”的按钮出现。当视窗不能显示全部新加入的按钮的时候，视窗中自动出现滚动条，如图3-5所示。

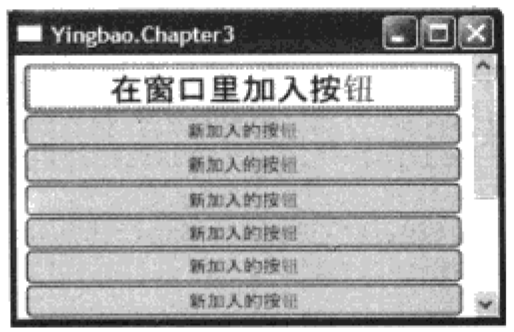


图3-5 ScrollViewer示例（2）

ScrollViewer有两个控制滚动条显示方式的相关属性：HorizontalScrollBarVisibility和VerticalScrollBarVisibility。VerticalScrollBarVisibility用来控制竖直滚动条，而HorizontalScrollBarVisibility则用来控制水平滚动条。这两个属性为ScrollBarVisibility枚举类型，其可能取的值如表3-2所示。HorizontalScrollBarVisibility的默认值是Disbaled，而VerticalScrollBarVisibility的默认值是“auto”。

ScrollViewer还提供了8个用于控制每次滚动范围的方法：LineUp、LineDown、LineLeft、LineRight、PageUp、PageDown、PageLeft和PageRight这些方法可以在程序中模拟人工操作滚动条。

表3-2 ScrollBarVisibility值及其意义

ScrollBarVisibility值	描 述
Disabled	这时滚动条不会出现
Auto	若视窗的内容比视窗的尺寸大，则自动显示相应的滚动条
Hidden	这时滚动条不会出现
Visible	总是显示滚动条

当视窗的大小，滚动条的位置或视点发生改变时，ScrollViewer会产生ScrollChanged事件。

前面提到StackPanel中UI元素排列的顺序是：若Orientation属性设为Vertical，默认的顺序是自上向下排列；若Orientation属性设为Horizontal，默认的顺序是自左向右排列。可以用StackPanel的FlowDirection属性来控制UI元素在StackPanel中出现的顺序：若Orientation属性为Horizontal，把FlowDirection的属性设为RightToLeft，则后加入的UI元素位于先加入的UI元素的左面，这和中国繁体书的排版一样。

StackPanel里还有一个属性：LayoutTransform。这个属性可以把StackPanel内的UI元素一起放大、缩小或旋转，本书的第14章将系统地介绍这些转换功能。

3.3 WrapPanel

WrapPanel是和StackPanel最相近的一个控制面板，StackPanel把其中的UI元素按行或列排列，而WrapPanel则可根据其中UI元素的尺寸和其自身可能的大小自动地把其中的UI元素排列到下一行或下一列。Windows操作系统中的文件管理器就是这样的例子，当选择图标显示模式时，文件管理器将根据视窗的宽度和文件图标的大小，自动布置文件在其中的位置。

在下面的例子中，笔者在WrapPanel中放入了七个矩形图形，每个图形的宽为70、高为40。为了更好地展示WrapPanel面板的功能，设置了Window的MinHeight和MaxWidth两个属性，这两个属性表示视窗所能改变的最小高度和宽度。

```
<Window x:Class="Yingbao.Chapter3.WrapPanelLayout"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="WrapPanel 的例子" Height="270" Width="212" MinHeight="250"
    MaxWidth="555" >
    <WrapPanel x:Name="ButtonPanel" Background="Gainsboro">
    <Rectangle Width="70" Height ="40">
      <Rectangle.Fill>
        <SolidColorBrush Color="Red" />
      </Rectangle.Fill>
    </Rectangle>
    <Rectangle Width="70" Height ="40">
      <Rectangle.Fill>
        <SolidColorBrush Color="Aqua" />
      </Rectangle.Fill>
    </Rectangle>
    <Rectangle Width="70" Height ="40">
      <Rectangle.Fill>
        <SolidColorBrush Color="Blue" />
      </Rectangle.Fill>
    </Rectangle>
    <Rectangle Width="70" Height ="40">
      <Rectangle.Fill>
        <SolidColorBrush Color="LightGreen" />
      </Rectangle.Fill>
    </Rectangle>
    <Rectangle Width="70" Height ="40">
      <Rectangle.Fill>
        <SolidColorBrush Color="Yellow" />
      </Rectangle.Fill>
    </Rectangle>
    <Rectangle Width="70" Height ="40">
      <Rectangle.Fill>
        <SolidColorBrush Color="Navy" />
      </Rectangle.Fill>
    </Rectangle>
    <Rectangle Width="70" Height ="40">
      <Rectangle.Fill>
        <SolidColorBrush Color="Gold" />
      </Rectangle.Fill>
```

```
    </Rectangle>
  </WrapPanel >
</Window>
```

上面这小段XAML程序的运行结果如图3-6所示。可用鼠标调整视窗的大小，当视窗的宽度小于其中矩形宽度的总和时，排在右面的矩形会自动排列到下一行；当视窗的宽度大于其中矩形宽度的总和时，排在下一行最左边的矩形会自动进入上一行。当视窗的宽度只能放置1个矩形时，WrapPanel的效果和StackPanel的效果一样，如图3-7所示。

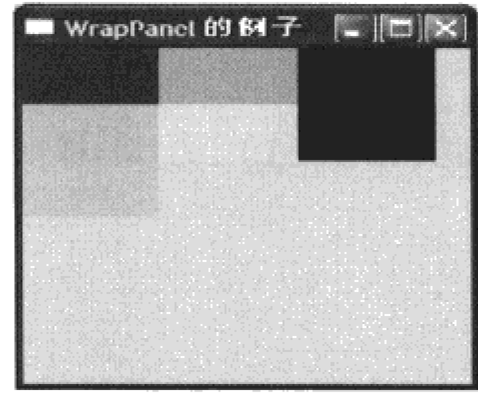


图3-6 WrapPanel示例（1）

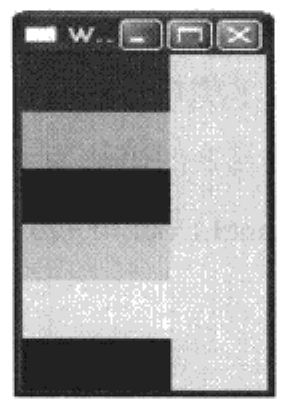


图3-7 WrapPanel示例（2）

WrapPanel中有三个属性ItemWidth、ItemHeight和Orientation，这三个属性来安排其中UI元素的位置：

- ItemWidth：定义所有子元素的宽度。每个子元素在其中显示的宽度由子元素自己的Width及HorizontalAlignment属性确定，若子元素的宽度大于ItemWidth，WrapPanel就会自动剪掉子元素超过ItemWidth的部分。ItemWidth的默认值为NaN，在这种情况下，WrapPanel使用其中最大子元素的宽度来作为列的宽度。
- ItemHeight：定义所有子元素的高度。每个子元素在其中显示的高度由子元素自己的Height及VerticalAlignment确定，若子元素的宽度大于ItemHeight，WrapPanel就会自动剪掉子元素超过ItemHeight的部分。ItemHeight的默认值为NaN，在这种情况下，WrapPanel使用其中最大子元素的高度来作为行的高度。
- Orientation：和StackPanel相同，唯一的区别是在WrapPanel中，Orientation 的默认值为Horizontal（水平放置）。

3.4 停靠面板（DockPanel）

停靠（Dock）这个概念并不是新的东西，在传统的视窗应用程序中，通常在视窗的顶端有菜单，最下面有状态行，有时还有工具条，工具条一般放在菜单的下面等。控制这些UI元素在视窗中的位置就是停靠，只不过在WPF之前，需要自己做大量工作，或使用第三方提供的工具。

WPF的DockPanel定义了一个Dock附加属性（Attached Property见第4章），其类型为Dock，是枚举类型，可取Left、Right、Top和Bottom四个值。注意Dock并没有一个Fill或Center的值，当LastChildFill属性设为True时，DockPanel用最后一个加入的UI元素填充所有剩下的地方。

在C#里，可以用两种方法来设定UI元素在DockPanel中的停靠位置，一种是用UI元素中的

SetValue()方法，比如设定控件myControl的停靠位置：

myControl.SetValue (DockPanel.DockProperty, Dock.Top)

SetValue方法是在DependencyObject类中定义的，DependencyObject是WPF中的一个重要的基类，很多对象都是从DependencyObject中派生出来。这里的语法有点不一样，SetValue设置的属性Dock并不在myControl中，而是在DockPanel中，这就是附加属性的特点（参见第4章）。

另外一种方法是在DockPanel里设置：

DockPanel.SetDock(myControl, Dock.Top);

这种方法看起来似乎更自然些。

在XAML中设置UI元素的语法为（以TextBlock为例，其他UI元素都一样）：

<TextBlock DockPanel.Dock=Dock.Top>

...

</TextBlock>

笔者设计的一个观察DockPanel停靠属性位置的程序如下，第一部分为XAML：

```
<Window x:Class="Yingbao.Chapter3.DockPanelProperties"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="停靠面板属性" Height="500" Width="600" >
  <DockPanel Background="White">
    <TextBlock FontSize="16" FontWeight="Bold" DockPanel.Dock="Top"
         Margin="20,0,0,10">停靠面板属性</TextBlock>
    <TextBlock DockPanel.Dock="Top" Margin="0,0,0,10" TextWrapping="Wrap">
      选择下面的UI元素及其停靠属性，观察UI元素在停靠控制面板中的位置
    </TextBlock>
    <StackPanel Orientation="Horizontal" DockPanel.Dock="Top"
         Margin="0,0,0,10">
      <TextBlock Margin ="0,0,0,5">停靠</TextBlock>
      <ComboBox  Name="dockCombo" Width ="60" >
        <ComboBoxItem  Name="dockTop">顶部</ComboBoxItem>
        <ComboBoxItem  Name="dockbottom">底部</ComboBoxItem>
        <ComboBoxItem  Name="dockLeft">左边</ComboBoxItem>
        <ComboBoxItem  Name="dockRight">右边</ComboBoxItem>
      </ComboBox>
      <TextBlock Margin ="0,0,0,5">UI元素</TextBlock>
      <ComboBox  Name="controlCombo" Width ="60"  SelectionChanged
         ="OnSelectControl">
        <ComboBoxItem  Name="button1">按钮</ComboBoxItem>
        <ComboBoxItem  Name="blockText1">字符框</ComboBoxItem>
        <ComboBoxItem  Name="circle">园</ComboBoxItem>
      </ComboBox>
      <TextBlock Margin ="5,0,0,5">LastChileFill</TextBlock>
      <ComboBox Text="Is not open" Width ="60"  SelectionChanged
         ="OnSelectLastChildFill">
```

```
        <ComboBoxItem  Name="LastChildTrue">True</ComboBoxItem>
        <ComboBoxItem  Name="LastChildFalse">False</ComboBoxItem>
      </ComboBox>
    </StackPanel>
    <Border Background="LightGoldenRodYellow" BorderBrush="Black"
         BorderThickness="1">
      <DockPanel Name="myDP">
        <Button Name ="btn" MinHeight ="200" MinWidth ="200"  Background
         ="GreenYellow" FontSize ="20">按钮</Button>
        <TextBlock Name ="tb" MinHeight ="200"  MinWidth ="200" Background
         ="Cyan" FontSize ="20">字符框</TextBlock>
        <Ellipse Name="ellps" MinWidth="200" MinHeight="200" Stroke="Black"
         Fill="LightSkyBlue" />
      </DockPanel>
    </Border>
  </DockPanel>
</Window>
```

第二部分为C#:

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Data;
using System.Windows.Documents;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Media.Imaging;
using System.Windows.Shapes;
namespace Yingbao.Chapter3
{
   public partial class DockPanelProperties : System.Windows.Window
    {
        public DockPanelProperties()
        {
            InitializeComponent();
        }

        private void OnSelectControl(object sender,
                SelectionChangedEventArgs args)
        {
             ComboBoxItem lbi = ((sender as ComboBox).SelectedItem  as
                  ComboBoxItem);
             if (lbi == this.button1)
             {
                 this.btn.SetValue(DockPanel.DockProperty,
                      GetCurrentDock());
             }
             if (lbi == this.blockText1)
             {
                 this.tb.SetValue(DockPanel.DockProperty, GetCurrentDock());
```

```
            }
            if (lbi == this.circle)
            {
                this.circle.SetValue(DockPanel.DockProperty,
                        GetCurrentDock());
            }
        }

        private void OnSelectLastChildFill(object sender,
                SelectionChangedEventArgs args)
        {
            ComboBoxItem lbi = ((sender as ComboBox).SelectedItem as
                                ComboBoxItem);
            if (lbi.Name  =="LastChildTrue")
            {
                myDP.LastChildFill = true;
            }
            else if (lbi.Name == "LastChildFalse")
            {
                myDP.LastChildFill = false;
            }
        }

        Dock GetCurrentDock()
        {
                Dock dockValue=Dock.Top;
                if (dockCombo.SelectedItem == this.dockTop )
                {
                    dockValue = Dock.Top;
                }
                else if (dockCombo.SelectedItem == this.dockbottom )
                {
                    dockValue = Dock.Bottom;
                }
                else if (dockCombo.SelectedItem == this.dockLeft )
                {
                    dockValue = Dock.Left ;
                }
                else if (dockCombo.SelectedItem == this.dockRight )
                {
                    dockValue = Dock.Right;
                }
                return dockValue;
        }
    }
}
```

在这个例子中，笔者使用XAML来设计用户界面，而用C#来处理UI元素的事件响应，这是WPF程序的通常做法。该视窗分为上下两个部分，上面部分的UI元素位于StackPanel中，提供对停靠属性、UI元素和LastChildFill的操作；下面的部分，在BorderUI元素内，有一个DockPanel(myDP)，其中含有按钮、字符框和园三个UI元素。上下两部分UI元素都是DockPanel的子元素。

笔者用组合框（dockCombo）来选择停靠属性，该组合框中有4个选项：顶部、底部、左边和右边；在选择了停靠属性后，你可以选择该停靠属性所要施加的对象,即用controlCombo组合框。

在C#的OnSelectControl方法中，对controlCombo组合框的事件响应代码时要对所选择的元素设置停靠属性。方法GetCurrentDock读出在组合框（dockCombo）中所选定的停靠属性。

OnSelectLastChildFill方法根据用户的选择，把myDP的LastChildFill设为true或false。可以看到当LastChildFill设为true时，myDP内的元素被三个元素填满；当LastChildFill设为false时，myDP中还剩下一块地方。

若把DockPanel中的元素的Dock都设为Dock.Top，则可以看到，元素在DockPanel中的排列呈从上到下的方式，这时的DockPanel和StackPanel的排版一样。所以StackPanel实际上可看作是DockPanel的特例。

DockPanel也可以设置Margin、LayoutTransform、HorizontalAlignment、Vertical-Alignment等属性，可以用这些属性来调整其中元素的排版效果。

上面这段程序的运行结果如图3-8所示。

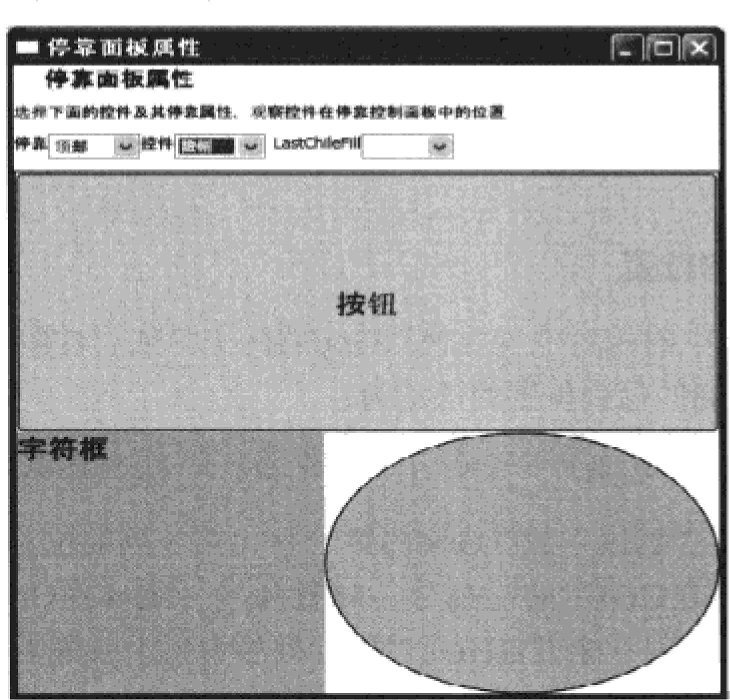


图3-8 DockPanel示例

3.5 表格式面板（Grid）

Grid控制面板是WPF中最常用的控制面板，当用Visual Studio或Express Blend创建WPF程序时，Visual Studio或Express Blend会自动在XAML中添加<Grid>标记。Grid排版和HTML表格排版是一样的，尽管现在有无数的网站，提供的内容丰富多彩，但网页版面设计都是基于表格排版的。

WPF中还有一个和Grid的类似的类——Table。Table类常用于文档，而Grid则多用于用户界面。

在XAML中定义Grid的语法为：

```
<Grid>
  <Grid.RowDefinitions>
    <RowDefinition/>
```

```
    <RowDefinition/>
    <RowDefinition/>
    </Grid.RowDefinitions>
  <Grid.ColumnDefinitions >
    <ColumnDefinition />
    <ColumnDefinition />
  </Grid.ColumnDefinitions>
</Grid>
```

上面的这段XAML定义了3行2列的Grid 。在C#里创建Grid的语法和创建其他对象一样：

```
Grid myGrid = new Grid();
for (int i = 0; i < 3; i++)
{
     RowDefinition rowdef = new RowDefinition();
     myGrid.RowDefinitions.Add(rowdef);
}
for (int i = 0; i < 2; i++)
{
     ColumnDefinition coldef = new ColumnDefinition();
     myGrid.ColumnDefinitions.Add(coldef);
 }
```

Grid类中含有行容器RowDefinitions和列容器ColumnDefinitions。 行和列分别用RowDefinition和ColumnDefinition类来表达。

3.5.1 设定UI元素在Grid中的位置

设定Grid中UI元素在Grid中的相对位置要使用Grid的两个附加属性Row和Column,若Grid里面含有Label，则使用XAML设置Label的行列位置的语法为：

```
<Label Grid.Row="0" Grid.Column= "0" FontSize ="20">测试</Label>
```

上面的XAML句子把标签“测试”放在Grid的第一行、第一列的位置。注意：Grid中的行列号为从0开始的整数，设定UI元素在Grid中的行列号，该行列号一定要在Grid所定义的范围内；换句话说，若你定义了3行3列的Grid，则不能设定UI元素的行列号为大于或等于3。

3.5.2 设定Grid行或列的尺寸

和普通UI元素可以设定Height和Width属性不同，RowDefinition类和ColumnDefinition类中相应的属性的类型不是Double，而是System.Window.GridLength。Grid调整Grid行列尺寸的方法有如下三种：

- 绝对尺寸　把Grid行列大小尺寸设为一个数值，这时Grid的行列尺寸不会随着其中UI元素的大小进行自动调整。
- 自动尺寸　把Grid的Height和Width设为Auto。这时WPF会根据Grid中的UI元素自动调整其行列的高度或宽度。其原则为：Grid的行高度由该行中元素的最大高度决定，Grid的列宽度由该列中元素的最大宽度决定。使用这种方法可以保证Grid中的UI元素不会只显示一部分。
- 按比例分割行列尺寸　把有限的平面大小按照一定的比例划分行的高度或列的宽度，其比例的

数值可以是浮点数。如第一列的宽度设为“*”、第二列的宽度设为“1.2*”、第三列的宽度设为“2.5*”，等等。

在C#中设定行列尺寸的语法：

```
RowDefinition rowdef = new RowDefinition();
rowdef.Height = new GridLength(100, GridUnitType.Pixel);   (绝对尺寸)
rowdef.Height = GridLength.Auto; (自动尺寸)
rowdef.Height = new GridLength (2,GridUnitType.Star);(按比例尺寸)
```

WPF默认设置行的高度和列的宽度为1个*，即每行或每列的大小一样。

在XAML中设定行列尺寸的语法：

```
<RowDefinition Height="100"/>                 (绝对尺寸)
<RowDefinition Height ="Auto">                (自动尺寸)
<RowDefinition Height = "2*">                 (按比例尺寸)
```

在设定Grid行列的尺寸时，还有一对重要参数：最小/最大宽度（MinWidth和MaxWidth）；最大/最小高度（MinHeight和MaxHeight）；当用户设定某个单元的MinWidth和MaxWidth值后，Grid允许该单元宽度的变化范围为MinWidth和MaxWidth之间。若用户的设定值大于MaxWidth, Grid用MaxWidth作为该单元的宽度；若用户的设定值小于MinWidth, Grid用MinWidth作为该单元的宽度。对于高度来说，上述规则也适用。

当Grid行列大小设为按比例排版时，WPF划分区域的方法为：

- 若行的高度都为*，就按Grid中的行数来计算出每行的高度。如整个Grid区域的高度是100，你要显示10行，那么每行的高度就是10（100/10）。
- 若Grid中有两行，第一行的高度设为*，第二行的高度设为1.5*，而整个Grid区域的高度仍为100，那么第一行的高度就是100/2.5= 40；第二行的高度就是100/2.5 * 1.5 = 60。

上述计算每行高度的规则也适用于计算每列的宽度，实际上在使用按比例分割行列尺寸时，行的高度或列的宽度的绝对数值并不重要，重要的是其相对比例。

3.5.3 元素横跨多个行列时的设定

有时候需要设置某个UI元素横跨多行或多列，Grid支持这种操作。用C#设置跨行和跨列的例子如下：

```
Rectange rect = new Rectangle();
rect.Width = 70;
rect.Height = 75;
Grid.SetRowSpan(rect,2);                    (横跨两行)
Grid.SetColumnSpan(rect,3);                 (横跨三列)
```

用XAML设置跨行和跨列的例子如下：

```
<Rectangle Width="70" Height="75"Grid.Row="0"Grid.Column="0"
Grid.RowSpan="2" Grid.ColumnSpan="3"/>
```

ColumnsSpan和RowSpan也是附加属性。

下面是一个用Grid排版的例子，这是一个计算器程序的用户界面。在XAML中，笔者定义了一个七行九列的Grid，把Grid的背景色设为蛋黄色，把ShowGridLines属性设为False，即不显示表格线。然后，又定义一个文字输入框，该文字输入框位于Grid的第一行，第一列，横跨9列。

```
<Window x:Class="Yingbao.Chapter3.GridCalculator"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Grid计算器" Height="300" Width="300">
  <DockPanel  Name="MyPanel">
    <Grid Name="MyGrid" Background="Wheat"  ShowGridLines="False">
      <Grid.ColumnDefinitions>
        <ColumnDefinition/>
        <ColumnDefinition/>
        <ColumnDefinition/>
        <ColumnDefinition/>
        <ColumnDefinition/>
        <ColumnDefinition/>
        <ColumnDefinition/>
        <ColumnDefinition/>
        <ColumnDefinition/>
      </Grid.ColumnDefinitions>
      <Grid.RowDefinitions>
        <RowDefinition/>
        <RowDefinition/>
        <RowDefinition/>
        <RowDefinition/>
        <RowDefinition/>
        <RowDefinition/>
      </Grid.RowDefinitions>
      <TextBox  Name ="InputText"  Grid.Column ="0" Grid.Row ="0"
        Grid.ColumnSpan ="9" FontSize ="12" FontWeight ="DemiBold" Margin
         ="5,2,10,3"/>
      <Button Name="B7" Click="DigitBtn_Click"  Grid.Column="4"
        Grid.Row="2" Margin ="2">7</Button>
      <Button Name="B8" Click="DigitBtn_Click"  Grid.Column="5"
        Grid.Row="2" Margin ="2">8</Button>
      <Button Name="B9" Click="DigitBtn_Click"  Grid.Column="6"
        Grid.Row="2" Margin ="2">9</Button>
      <Button Name="B4" Click="DigitBtn_Click"  Grid.Column="4"
        Grid.Row="3"  Margin ="2">4</Button>
      <Button Name="B5" Click="DigitBtn_Click"  Grid.Column="5"
        Grid.Row="3" Margin ="2">5</Button>
      <Button Name="B6" Click="DigitBtn_Click"  Grid.Column="6"
        Grid.Row="3" Margin ="2">6</Button>
      <Button Name="B1" Click="DigitBtn_Click"  Grid.Column="4"
        Grid.Row="4" Margin ="2">1</Button>
      <Button Name="B2" Click="DigitBtn_Click"  Grid.Column="5"
        Grid.Row="4" Margin ="2">2</Button>
      <Button Name="B3" Click="DigitBtn_Click"  Grid.Column="6"
        Grid.Row="4" Margin ="2">3</Button>
      <Button Name="B0" Click="DigitBtn_Click"  Grid.Column="4"
```

```
        Grid.Row="5" Margin ="2">0</Button>
      <Button Name="BPeriod" Click="DigitBtn_Click"  Grid.Column="5"
        Grid.Row="5" Margin ="2">.</Button>
      <Button Name="BPM"     Click="OperBtn_Click" Background="Darkgray"
        Grid.Column="6" Grid.Row="5" Margin ="2" >+/-</Button>
      <Button Name="BDevide"    Click="OperBtn_Click" Background="Darkgray"
        Grid.Column="7" Grid.Row="2"  Margin ="2">/</Button>
      <Button Name="BMultiply"  Click="OperBtn_Click" Background="Darkgray"
        Grid.Column="7" Grid.Row="3" Margin ="2">*</Button>
      <Button Name="BMinus"  Click="OperBtn_Click" Background="Darkgray"
        Grid.Column="7" Grid.Row="4" Margin ="2">-</Button>
      <Button Name="BPlus"      Click="OperBtn_Click" Background="Darkgray"
        Grid.Column="7" Grid.Row="5" Margin ="2">+</Button>
      <Button Name="BSqrt" Click="OperBtn_Click" Background="Darkgray"
        Grid.Column="8" Grid.Row="2" Margin ="2" ToolTip="Usage: 'A
        Sqrt'" >Sqrt</Button>
      <Button Name="BPercent" Click="OperBtn_Click" Background="Darkgray"
        Grid.Column="8" Grid.Row="3"  Margin ="2" ToolTip="Usage: 'A % B
        ='" >%</Button>
      <Button Name="BOneOver" Click="OperBtn_Click" Background="Darkgray"
        Grid.Column="8" Grid.Row="4"  Margin ="2" ToolTip="Usage: 'A
        1/X'">1/X</Button>
      <Button Name="BEqual"   Click="OperBtn_Click" Background="Darkgray"
        Grid.Column="8" Grid.Row="5" Margin ="2">=</Button>
      <Button Name="BC"  Click="OperBtn_Click" Background="Darkgray"
        Grid.Column="8" Grid.Row="1" Grid.ColumnSpan="1" ToolTip="Clear
        All">C</Button>
      <Button Name="BCE" Click="OperBtn_Click" Background="Darkgray"
        Grid.Column="7" Grid.Row="1" Grid.ColumnSpan="1"  ToolTip="Clear
        Current Entry">CE</Button>
      <Button Name="BMemClear" Click="OperBtn_Click" Background="Darkgray"
          Grid.Column="3" Grid.Row="2"  ToolTip="Clear Memory">MC</Button>
      <Button Name="BMemRecall" Click="OperBtn_Click" Background="Darkgray"
          Grid.Column="3" Grid.Row="3"  ToolTip="Recall Memory">MR</Button>
      <Button Name="BMemSave" Click="OperBtn_Click" Background="Darkgray"
          Grid.Column="3" Grid.Row="4" ToolTip="Store in
          Memory">MS</Button>
      <Button Name="BMemPlus" Click="OperBtn_Click" Background="Darkgray"
          Grid.Column="3" Grid.Row="5"  ToolTip="Add To Memory">M+</Button>
      <TextBlock  Name="BMemBox" Grid.Column="3" Grid.Row="1"
          Margin="10,17,10,17" Grid.ColumnSpan="2"></TextBlock>
    </Grid>
  </DockPanel>
</Window>
```

在这段XAML中，笔者并没有设定Grid中的行的宽度或列的高度，在这种情况下，WPF自动把行的高度或列的宽度设为比例方式且每行的高度或每列的宽度一样；在Grid里面定义了数字键0-9和计算器常用的操作键，其中还定义了操作的事件处理函数接口。后台C#程序如下：

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Windows;
```

```
using System.Windows.Controls;
using System.Windows.Data;
using System.Windows.Documents;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Media.Imaging;
using System.Windows.Shapes;

namespace Yingbao.Chapter3
{
    public partial class GridCalculator : System.Windows.Window
    {
        public GridCalculator()
        {
            InitializeComponent();
        }

        private void DigitBtn_Click(object sender, RoutedEventArgs e)
        {

        }

        private void OperBtn_Click(object sender, RoutedEventArgs e)
        {

        }
       }
}
```

笔者在这里略去了处理按钮即TextBox的细节，所以DigitalBtn_Click和OperBtn_Click两个方法为空。上面这段程序的运行结果如图3-9所示。

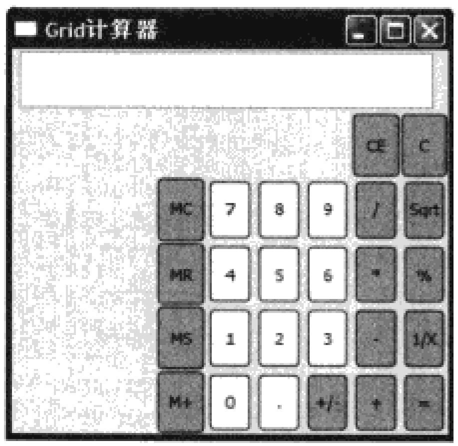


图3-9 使用Grid排版的计算器

3.5.4 在Grid中保持多行或多列大小的一致性

有时候希望Grid中的一些行或列无论在什么情况下都保持相同的高度或宽度，使用RowDefinition和ColumnDefinition中的SharedSizedGroup 可以达到这一目的。

下面的XAML创建了一个1行4列的Grid，每个Grid单元含有一个矩形，想保持第1列和第3列，第2列和第4列的宽度相同；为了演示效果，笔者在第2列和第3列间增加了GridSplitter：

```
<Window x:Class="Yingbao.Chapter3.GridSharedGroupSize"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Grid宽度" Height="50" Width="300" >
    <Grid>
      <Grid.ColumnDefinitions >
        <ColumnDefinition Width ="*"/>
        <ColumnDefinition Width ="2*"/>
        <ColumnDefinition Width ="*"/>
        <ColumnDefinition Width ="2*"/>
      </Grid.ColumnDefinitions>
      <Grid.RowDefinitions >
        <RowDefinition Height ="auto"/>
      </Grid.RowDefinitions>
      <Rectangle Grid.Column ="0" Grid.Row ="0" Fill ="Cyan"  MinWidth
         ="20" Height ="15"/>
      <Rectangle Grid.Column ="1" Grid.Row ="0" Fill ="DeepPink"  MinWidth
         ="20" Height ="15"/>
      <GridSplitter Grid.Column ="1" Width ="5"/>
      <Rectangle Grid.Column ="2" Grid.Row ="0" Fill ="Yellow" MinWidth
         ="20" Height ="15"/>
      <Rectangle Grid.Column ="3" Grid.Row ="0" Fill ="Green" MinWidth
         ="20" Height ="15"/>
    </Grid>
</Window>
```

这段程序的运行结果如图3-10所示。

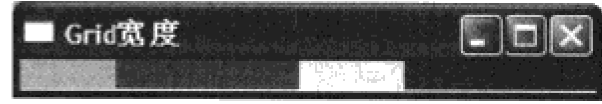


图3-10 初始宽度，第1列和第3列；第2列和第4列的宽度相同

可以用鼠标左右移动第1列和第3列间的GridSplitter，得到如图3-11所示的结果。这时第1列和第3列；第2列和第4列的宽度不再相同。

图3-11 调整中间GridSplitter的位置

要在调整GridSplitter的位置后，Grid列间的宽度仍然保持相同的关系，就要用到SharedSizeGroup属性，并把上面的列定义改为：

```
<Grid  Grid.IsSharedSizeScope="True">
      <Grid.ColumnDefinitions >
        <ColumnDefinition Width="20"   SharedSizeGroup ="mgroup1"/>
        <ColumnDefinition  Width="40"  SharedSizeGroup ="mgroup2"/>
        <ColumnDefinition Width="20" SharedSizeGroup ="mgroup1"/>
        <ColumnDefinition Width="40" SharedSizeGroup ="mgroup2"/>
</Grid.ColumnDefinitions>
...
</Grid>
```

设定SharedSizeGroup后的结果如图3-12所示。由图可见，第1列和第3列，第2列和第4列的宽度保

持相同的比例。

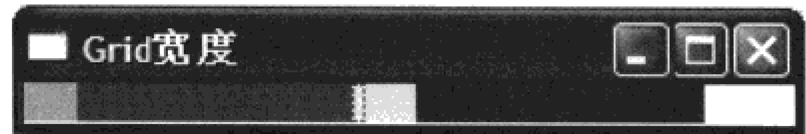


图3-12 设定SharedSizeGroup后，位于同一组中的列的宽度保持相同的比例

注意：首先在Grid层面设置了一个IsSharedSizeScope属性为True,表明在Grid中，会有行或列使用SharedSizeGroup属性。在ColumnDefinition中，笔者把第1列和第3列的SharedSizeGroup设为“mgroup1”，把第2列和第4列的SharedSize-Group设为“mgroup2”。SharedSizedGroup可以设为任何字符串，具有相同Shared-SizedGrou字符串的行或列，其高度或宽度相同。字符串大小写敏感，如mgroup1和mGroup1是不同的字符串。

3.6 UniformGrid

有时候觉得使用Grid过于烦琐的话，可以设置行列数，行列的宽度或高度等属性，实际上就是需要一种大小相等在平面上均匀排列的表格。UniformGrid支持这种表格的排版类，它是一般Grid的一个特例。这时，不必要定义行列的集合；不必设定每行的列数或行数，UniformGrid总是把行数和列数设为相等；每个单元只含有一个子元素，所以也不必用附加属性来说明哪个元素位于哪个单元。请看下面的例子：

```
<Window x:Class="Yingbao.Chapter3.UsingUniformGrid"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Yingbao.Chapter3" Height="300" Width="300">
  <UniformGrid>
    <TextBlock Background ="Yellow " Margin ="5">1</TextBlock>
    <TextBlock Background ="Yellow " Margin ="5">2</TextBlock>
    <TextBlock Background ="Yellow " Margin ="5">3</TextBlock>
    <TextBlock Background ="Yellow " Margin ="5">4</TextBlock>
    <TextBlock Background ="Yellow " Margin ="5">5</TextBlock>
    <TextBlock Background ="Yellow " Margin ="5">6</TextBlock>
    <TextBlock Background ="Yellow " Margin ="5">7</TextBlock>
    <TextBlock Background ="Yellow " Margin ="5">8</TextBlock>
    <TextBlock Background ="Yellow " Margin ="5">9</TextBlock>
  </UniformGrid >
</Window>
```

由于笔者在其中加入了9个TextBlock, UniformGrid自动把它分成3行3列，若再加入一个元素，情况会怎么样呢？UnformGrid会在每行里添加一列，然后把元素从左到右，从上到下排列，直到4行4列为止，其元素的排列如图3-13所示。

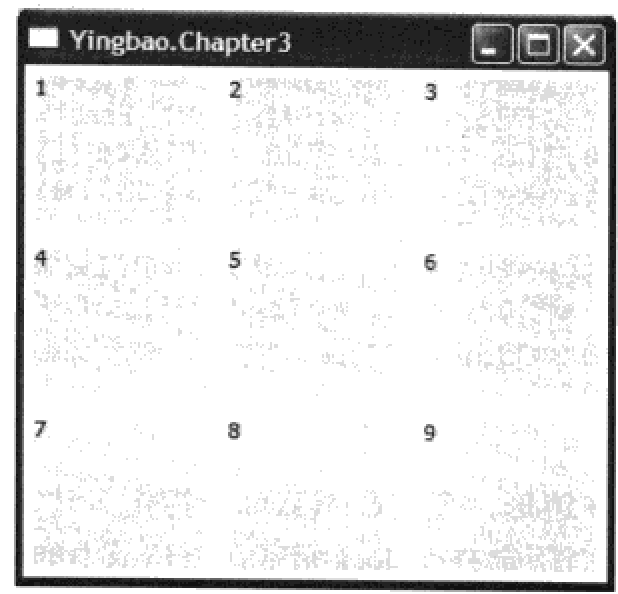


图3-13 UniformGrid中元素排列

3.7 画布面板（Canvas）

过去视窗界面排版都是用绝对坐标，比如，用户设计的对话框，每个控件在对话框中的位置，都是用x，y值确定的；又比如用户要在窗口内显示文字或图形，也要指出所要显示图素的确切位置。WPF也支持这种精确定位的排版，Canvas面板就是为此设计的。

Canvas中的坐标值是一种与设备无关的单位值，其坐标原点位于Canvas的左上角。X坐标轴从原点指向屏幕的右边，Y坐标轴从原点指向屏幕的下方。UI元素并没有指定其位置的坐标属性，要把某个元素放在Canvas上的某个位置，需要使用Canvas的Left和Top附加属性。

XAML：

```
<TextBlock Canvas.Left= "15" Canvas.Top= "15">字符框</TextBlock>
```

C#：

```
TextBlock tb = new TextBlock();
Tb.Text= "字符框";
Canvas.SetLeft( tb, 15);
Canvas.SetTop( tb, 15);
```

上面的代码把TextBlock元素放置在Canvas左上角x=15,y=15的位置。

在WPF中,Canvas排版常用在对图形元素的版面布置上。虽然可以设置图形元素在Canvas上的左上角或右下角的位置，但若设置了图形元素的左上角的位置，同时又设置了图形元素右下角的位置，WPF就会自动忽略所设的右下角的位置坐标。

现在列举一个使用Canvas排版的例子：

```
<Window x:Class="Yingbao.Chapter3.PolygonCanvas"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Yingbao.Chapter3" Height="300" Width="300">
    <Canvas>
      <Polygon Points="100,0 75,75 100,100 125,75"
```

```
        Stroke="Black" StrokeThickness="2" Fill="Yellow"/>
        <Polygon Points="100,100 125,125 100,200 75,125"
             Stroke="Yellow" StrokeThickness="2" Fill="Black"/>
        <Polygon Points="100,100 125,75 200,100 125,125"
             Stroke="Red" StrokeThickness="2" Fill="Blue"/>
        <Polygon Points="100,100 75,125 0,100 75,75"
             Stroke="Blue" StrokeThickness="2" Fill="Red"/>
        <TextBlock FontSize ="20" FontStyle="Italic" Foreground ="Red"
           Canvas.Left ="130" Canvas.Top ="10"> Canvas中的图形</TextBlock>
    </Canvas>
</Window>
```

这段XAML程序创建了4个多边形，使用Canvas可以指定多边形中每个点的位置；在TextBlock中，笔者设置了左上角坐标，从而确定TextBlock元素在Canvas中的确切位置。这段程序的运行结果如图3-14所示，如果要在计算机上绘图，使用Canvas是最好的选择。

图3-14 在Canvas中确定图形元素的位置

3.8 本章小结

本章介绍了WPF中的排版系统，重点在StackPanel、WrapPanel、DockPanel、Grid、UniformGrid等元素的使用方法。WPF允许这些面板类可以互相包含，甚至在UI元素(如Button)中也可以包含面板元素作为其自身的子元素,这些排版元素组合起来使得WPF很容易支持各种版面设计。还有与排版相关的一些UI类，如LayoutTransform，该类支持FrameWorkElement在版面上的变换，将在第14章一并介绍。还有一些处在控件和版面设计之间的类，如TabControl；使用TabControl的例子如IE 7.0。这个类可用作版面设计，但在属性上应把它归于UI控件元素，将在第6章加以讨论。

第4章 WPF中的属性系统

WPF中包含一个属性子系统，其中有两个重要的属性，即相关属性（Dependency Property）和附加属性（Attached Property）。WPF中的很多特性，如校验、默认值、数据绑定、风格和动画等都是通过这个属性系统来实现的。WPF开发属性系统的目的之一是支持用XAML语言来描述人际界面，显然在XAML中，使用属性更为直观；目的之二是节省内存开销；目的之三是支持一种新的属性继承机制，从而进一步节省内存开销。

本章介绍属性系统中的一些新概念及其使用方法。

4.1 CLR属性

.NET开发平台从1.0开始就支持属性的概念，相对于WPF的相关属性，我们把 .NET中的属性称为CLR属性，CLR属性主要实现了面向对象的封装：

```
Class A {
    private int  x;
    public int X
        {
                get{ return x;}
                set{ x= value;}
        }
}
```

在类A之外，其私有域x不可见，要获取或改变x的值，必须要通过成员大X来实现：

```
A  a = new A();
a.X= 10;
int x = a.X;
```

这样做的好处是，可以在改变域x之前在set中放入一些校验代码，从而避免错误地改变x的值。比如，我们可以对x的值加上在0和100之间的约束：

```
public int X
{
    get{return x;}
    set{
      if( value>0 && value<100)
      {
           x= value;
      }
    else{
           //产生错误
      }
}
```

还可以实现只读功能（把CLR属性X的set去掉，从而无法从类A的外面改变x的值）或只写（即把CLR属性X的get去掉）。

我们可以利用CLR属性和.NET中的委托（delegate）、事项（event）来实现当属性发生改变时通知相关对象的功能；还可以把CLR属性绑定到某个控件上，从而自动实现属性的值和某个元素的特性自动关联（数据绑定）。

4.2 相关属性的概念

WPF引入了相关属性的概念。为了支持XAML，WPF中每个UI元素都含有大量的属性。如我们前面用的按钮（Button）控件，竟然含有117个属性之多，而且有的属性本身就是一个类，其中又含有大量的属性。这些属性都可以在XAML中直接设定，而不需要任何过程代码；但如果没有相关属性系统，这种简单设定是不可能完成相应的功能的。

对于开发WPF控件的用户来说，深入理解相关属性尤其重要。对于一般使用WPF的用户来说，了解WPF的相关属性及其工作原理，可以更好地使用WPF中诸如风格、动画等功能，因为这些功能和相关属性密切相关。

现在让我们先来看一个使用相关属性的例子：

```
<Window x:Class="Yingbao.Chapter4.DependentProperty1"
   xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
   xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
 Title="WPF中的相关属性(1)" Height="150" Width="300" FontSize ="12" >
  <StackPanel >
   <Label Content ="使用窗口类中定义的字体" />
   <Label Content ="在WPF中, Microsoft引入了相关属性的概念。"/>
   <Label Content ="使用自己的字体" FontSize ="15" />
   <Button>
        <TextBlock >使用窗口类中定义的字体</TextBlock>
   </Button>
  </StackPanel>
</Window>
```

在这个例子中，在Window类中定义了大小为12的字体。视窗下的StackPanel中有三个标签（Label）控件和一个按钮（Button)控件，Button控件中包含了另外一个控件TextBlock，上述XAML的运行结果如图4-1所示。

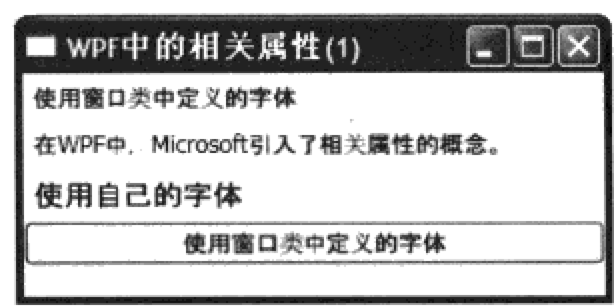


图4-1 WPF中字体大小相关属性

4.2.1 相关属性的传递

由图4-1可见，在没有设置字体大小的两个标签控件中，使用的是Window类中定义的字体，而在设置字体大小的标签控件中，显示的文字是用自己定义的字体。这就是WPF中相关属性的继承规

则：相关属性会沿着视觉树，从包容器类向其中的子元数传递。这种传递可以隔代进行，如上例中按钮的TextBlock类，它通过按钮Button类来继承Window类中定义的FontSize属性。虽然按钮中含有FontSize相关属性，但对于相关属性的传递来说，这不是必须的。如上例中的StackPanel并不含有FontSize属性，但字体大小还是通过StackPanel传递给了其中的子元素。上例中可能的视觉树传递情况如图4-2所示（有关视觉树，我将在第7章中和传递事件一起讨论）：

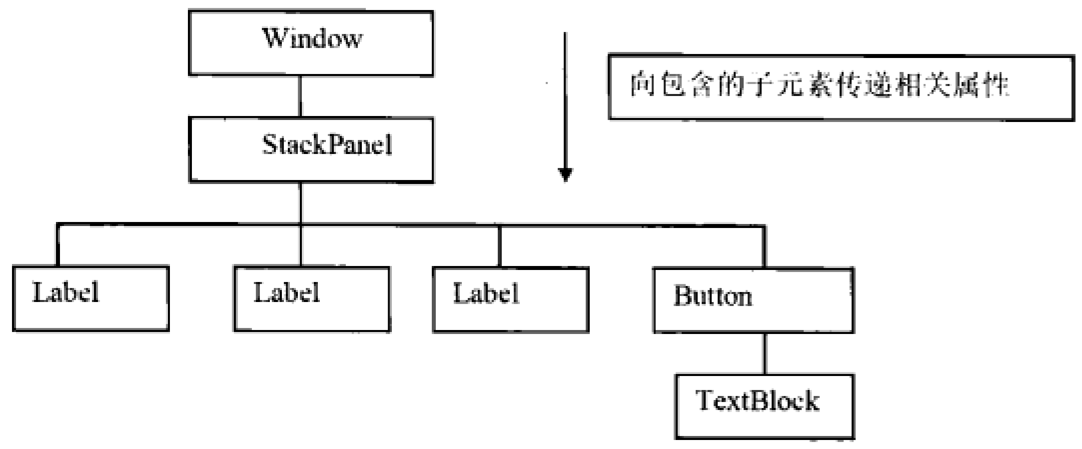


图4-2 相关属性沿着视觉树传递

需要注意的是，这里使用的是相关属性的传递一词，在WPF的很多文档中用的都是继承（Inheritance），笔者认为“传递”比继承更为确切。继承一词容易和面向对象编程（OOP）中的继承概念相混淆，面向对象编程中的继承是派生类继承基类(有时也叫父类)的属性和方法，而相关属性的传递，则是沿着WPF中的视觉树进行的。一旦相关属性在传递链上被覆盖，则后面视觉树中的元素就不会传递原来相关属性的值，而会传递新的相关属性的值。

4.2.2 WPF对相关属性的支持

WPF对相关属性的支持是WPF最初设计的目标之一，这一点我们可以从如图4-3所示的WPF中的类树中看出来：

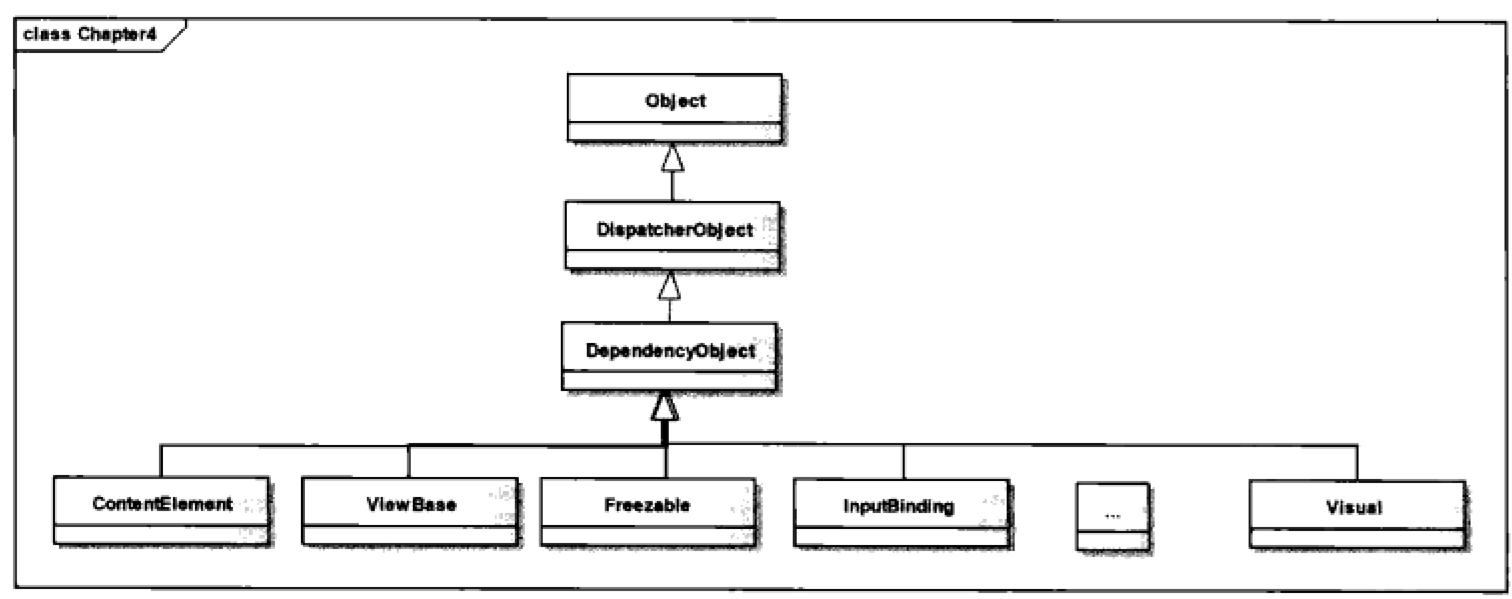


图4-3 WPF中的类继承树

从图4-3可见，DependencyObject非常靠近继承树的根部，该类提供对相关属性的支持，WPF中大量的类在其继承树上都有DependencyObject，DependencyObject是构建WPF相关属性系统的最重要

的类。表4-1列出了DependencyObject中与相关属性有关的操作。

表4-1 DependencyObject中与相关属性有关的操作

方法名	描 述
ClearValue	该方法清除相关属性的值（相关属性将回到默认值）
CoerceValue	设定某个相关属性的强制值，通过调用CoerceValueCallBack函数来实现
GetValue	该方法返回某个相关属性的值
SetValue	该方法设定某个相关属性的值
OnPropertyChanged	当有相关属性的值发生改变时，该方法被调用
ReadLocalValue	该方法返回某个相关属性的本地值

这些方法在使用相关属性时非常重要，在后面会用到。

4.3 自定义相关属性

要深入理解相关属性，最好的办法是自己定义一个相关属性，并且使用自己定义的相关属性。在举例说明自定义相关属性之前，先来考察一下WPF中的FrameworkPropertyMetadata类。这个类在WPF属性系统中建立相关属性和元数据间的联系，它是PropertyMetadata的派生类。FrameworkPropertyMetadata中包括如表4-2所示的一些属性：

表4-2 FrameworkPropertyMetadata中的属性

属性名	描 述
Affect Arrange	表示该相关属性对排版会有影响
Affect Measure	表示该相关属性会对元素的大小有影响
AffectParentArrange	表示该相关属性会对包含该UI元素的元素的排版有影响
Affect Render	表示对显示该UI元素有影响
BindsTwoWayByDefault	表示在默认的情况下为双向绑定（在第11章将说明什么是双向绑定）
DefaultUpdateSourceTrigger	设置默认时的UpdateSourceTrigger值
Inherit	表示该相关属性是否可以向子元素传递
IsDataBindingAllowed	表示该相关属性是否可以用来数据绑定

自定义相关属性的步骤：

- 声明相关属性变量（总是public static的）；
- 使用FrameworkPropertyMetadata在WPF相关属性系统中注册。注册时用DependencyProperty.Register方法注册读写相关属性或用DependencyProperty.RegisterReadOnly方法注册只读相关属性；
- 使用DependencyObject类中GetValue, SetValue方法来读写相关属性的值。

这就是在类中说明一个相关属性所要遵循的基本步骤。

下面来看一个自定义相关属性的实例，并考察自定义相关属性的传递。笔者希望用相关属性来实现在窗口中显示时间，首先声明一个窗口类：TimerWindow。

```
using System;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Input;
using System.Windows.Media;
```

```
using System.Windows.Threading;
using System.Threading;

namespace Yingbao.Chapter4
{
    public partial class TimerWindow : Window
    {
       private delegate void SetTimeDelegate(DateTime time);

        public static  DependencyProperty TimeProperty;
        private Timer myTimer;

        public DateTime Time
        {
            set {
                SetValue(TimeProperty, value);
                }
            get {
                return (DateTime)GetValue(TimeProperty);
            }
        }

        static TimerWindow()
        {
            FrameworkPropertyMetadata metadata = new
                  FrameworkPropertyMetadata();
            metadata.Inherits = true;
            metadata.DefaultValue = DateTime.Now;
            metadata.AffectsMeasure = true;
            metadata.PropertyChangedCallback +=
                 OnTimerPropertyChanged;
            TimeProperty =
                  DependencyProperty.Register("Timer",
                  typeof(DateTime),
                  typeof(TimerWindow),
                  metadata,
                  ValidateTimeValue);
        }

        public TimerWindow()
            : base()
        {
            InitializeComponent();

            myTimer =
                    new Timer(new TimerCallback (RefreshTime), new
                    AutoResetEvent(false ),0, 1000);
        }

        static bool ValidateTimeValue(object obj)
        {
            DateTime dt = (DateTime)obj;
            if (dt.Year > 1990 && dt.Year < 2200)
```

```
            {
                return true;
            }
            return false;
        }

        static void OnTimerPropertyChanged(DependencyObject obj,
                    DependencyPropertyChangedEventArgs args)
        {
        }

        private void RefreshTime(Object stateInfo)
        {
            SetTimeDelegate d = new
                                SetTimeDelegate( SetTimerProperty);
            this.Dispatcher.BeginInvoke(DispatcherPriority.Send, d,
                        DateTime.Now);
            }

        private void SetTimerProperty(DateTime dt)
        {
            this.Time = dt;
        }
    }
}
```

在TimeWindow类中首先声明一个相关属性变量TimeProperty，其类型为DependencyProperty；如前所述，这个变量必须是public static的；然后声明一个Time的CLR属性，其中使用DependencyObject的Set和Get方法来设定或获取TimeProperty的值。在TimeWindow中，定义了一个静态构造函数：

```
static TimerWindow()
 {
  FrameworkPropertyMetadata metadata = new FrameworkPropertyMetadata();
  metadata.Inherits = true;
  metadata.DefaultValue = DateTime.Now;
  metadata.AffectsMeasure = true;
  metadata.PropertyChangedCallback += OnTimerPropertyChanged;
  //register property
  TimeProperty = DependencyProperty.Register("Timer",
                  typeof(DateTime),
                  typeof(TimerWindow),
                  metadata,
                  ValidateTimeValue);
 }
```

在这个函数中实现TimeProperty相关属性在WPF属性系统中注册，要注册一个属性，需要说明一个类型为FrameworkPropertyMetadata 的元数据与之相关。这里把Inherits设为true，表示TimePerty可以向TimerWindow中的子元素传递。然后把元数据的初始值设为DateTime.Now，TimeProperty的类型为DateTime。元数据的AffectdMeasure设为true，表示TimeProperty相关属性的改变会影响UI元素的尺寸。OnTimePropertyChanged方法为当属性值改变时的事件处理程序，在这里笔者主要是为了演示在

元数据中接入用户处理程序的技术，在OnTimePropertyChanged 方法中并没有做什么工作：

```
static void OnTimerPropertyChanged(DependencyObject obj,
                                   DependencyPropertyChangedEventArgs args)
      {

      }
```

这个方法是静态的，这是因为笔者在注册TimeProperty时还没有创建TimerWindow类的实例！

Register函数的最后一个参数为ValidateTimeValue, 这个参数为WPF属性系统要用的回调方法。在改变相关属性之前，WPF属性系统需要询问用户软件，所要设定的属性值是否有效？用户软件可以在该回调函数中放入相关校验逻辑，如笔者在这里假定时间在1990年到2200年之间是有效的（返回true），在此时间范围之外为无效（返回false）：

```
static bool ValidateTimeValue(object obj)
 {
      DateTime dt = (DateTime)obj;
      if (dt.Year > 1990 && dt.Year < 2200)
      {
         return true;
      }
      return false;
 }
```

为了让TimeProperty这个相关属性随着时间实时地变化，笔者在TimerWindow中使用了另外一个线程来不断自动刷新TimeProperty的值：

```
myTimer =  new Timer(new TimerCallback (RefreshTime), new
                     AutoResetEvent(false ),0, 1000);
```

这个计时器的作用是每秒钟调用RefreshTime方法。在RefreshTime方法中，更新TimeProperty属性值。由于RefreshTime和WPF的UI并不在一个线程内，故需要使用Window类的继承树中的DispatcherObject对象中的Dispatcher.BeginInvoke把此任务委托给SetTimerProperty函数来完成（有关在.NET 用户界面中使用多线程的问题，笔者曾有专文加以讨论）。

```
private void RefreshTime(Object stateInfo)
{
  SetTimeDelegate d = new SetTimeDelegate(SetTimerProperty);
  this.Dispatcher.BeginInvoke(DispatcherPriority.Send, d,DateTime.Now);
}
```

读者需要明白的是：通过这一委托过程，SetTimerProperty方法和WPF的UI处在同一个线程之内。

```
private void SetTimerProperty(DateTime dt)
{
    this.Time = dt;
}
```

现在我想在另一个类中从TimeWindow中传递TimeProperty相关属性，换句话说，是希望

TimeWindow能向其中的子类传递TimeProperty相关属性的值。为此，笔者写了另一个类CustomeTextBlock类：

```
using System;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Input;
using System.Windows.Media;
using System.Threading;

namespace Yingbao.Chapter4
{
    public class CustomeTextBlock: TextBlock
    {
        static CustomeTextBlock()
        {
           FrameworkPropertyMetadata metaData = new
                FrameworkPropertyMetadata ();
           metaData.Inherits = true;

           TimeProperty = TimerWindow.TimeProperty.AddOwner(
                          typeof(CustomeTextBlock));
           TimeProperty.OverrideMetadata(
                         typeof( CustomeTextBlock ),metaData );
          }

        public CustomeTextBlock()
            : base()
        {

        }

        #region Inherited Dependency Property  declaration

        public static  DependencyProperty TimeProperty;

        public DateTime Timer
        {
            get {
                 return (DateTime)GetValue(TimeProperty);
             }
            set {
                 SetValue(TimeProperty, value);
            }
        }
        #endregion
    }
}
```

CustomTextBlock是从TextBlock类中派生出来的。在这个类中，声明了Time-Property为相关属性类型，并且声明了一个CLR属性Timer，其类型DateTime，这些都和TimerWindow类中的用法完全一样，所不同的是在CustomeTextBlock静态构造函数中，笔者把TimerWindow类的TimeProperty相关属

性加了一个拥有者：

```
TimeProperty = TimerWindow.TimeProperty.AddOwner(
                        typeof(CustomeTextBlock));
```

然后，对这个相关属性的元数据进行了修改：

```
TimeProperty.OverrideMetadata(typeof( CustomeTextBlock ),metaData );
```

这个静态构造函数如下：

```
static CustomeTextBlock()
{
  FrameworkPropertyMetadata metaData = new
                        FrameworkPropertyMetadata ();
  metaData.Inherits = true;
  TimeProperty = TimerWindow.TimeProperty.AddOwner(
                        typeof(CustomeTextBlock));
  TimeProperty.OverrideMetadata(
                        typeof( CustomeTextBlock ),metaData );
}
```

有了上面的工作，就可以用XAML来布置窗口的版面了。这段程序如下：

```
<Window x:Class="Yingbao.Chapter4.TimerWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="clr-namespace:Yingbao.Chapter4"
    Title="自定义相关属性" Height="200" Width="350" >
  <StackPanel >
    <local:CustomeTextBlock  Text="{Binding RelativeSource={x:Static
       RelativeSource.Self},Path=Timer}"   FontSize="32"
       Background="Wheat" />
    <local:CustomeTextBlock   Text="{Binding RelativeSource={x:Static
       RelativeSource.Self},Path=Timer}" Timer="2008/06/30"
       FontSize="32" Background="Wheat" />
  </StackPanel>
</Window>
```

首先在Window标记下，用x:Class把XAML中的Windows和TimerWindow联系起来，为了使用CustumeTextBlock，需要引入Yingbao.Chapter4命名空间：

```
xmlns:local="clr-namespace:Yingbao.Chapter4"
```

在CustomeTextBlock中，笔者使用了数据绑定技术。如果读者对这些语法不了解，可参考本书第11章，在那里将详细讨论各种绑定技术。上述自定义相关属性的示例如图4-4所示。

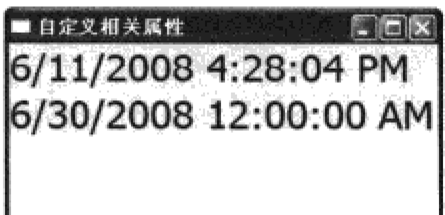


图4-4 自定义相关属性示例

第一个CustomeTextBlock中，Text属性绑定到CustomeTextBlock的TimeProperty相关属性上，由于该相关属性的值是从TimerWindow中的TimeProperty相关属性的值传递过来的，所以CustimeTextBlock中的TimeProperty会随着TimerWindow中的TimeProperty的值的变化而变化。由于笔者在TimerWindow类中用了一个独立的线程来每秒钟更新TimeProperty的值，所以，窗口中第一个TextBlock中显示的值随着时间的变化而变化。

第二个CustomeTextBlock中，由于笔者设定了TimeProperty的值，从而切断了相关属性的传递链，可以看到，其显示的时间停留在2008年的6月30号不变，而这正是在XAML中设定的日期：

```
Timer="2008/06/30"
```

由于没有设置时分秒，故WPF默认为“12：00：00 AM”。

上面的例子中，讨论了自定义相关属性技术，同时讨论了如何为相关属性提供默认值，对相关属性进行校验，以及相关属性发生改变时通知相关对象的技术，这一切都是通过FrameworkPropertyMetadata类来实现的。WPF对动画、风格、模板，以及排版的支持都是经由相关属性来实现的。

需要指出的是，并不是所有的相关属性的值都能够向其中的子元素传递的，能否传递相关属性的值实际上是由注册相关属性时所使用的FrameworkPropertyMetadata中的Inherits标志确定的。如在上面的例子中，若把TimerWindow或CustomTextBlock类中注册相关属性TimeProperty时用的元数据Inherits标志设为false，则结果视窗中所显示的时间就不会随时变化了。

4.4 附加属性

附加属性（Attached Property）其实是相关属性的另外一种形式。所有从DependencyObject类中派生出来的类，都可以使用附加属性。

为什么要引入附加属性呢？可以看这样一个例子：若有一些图形，比如说矩形，要在视窗上显示出来。若使用Canvas排版，那么要告诉Canvas该矩形的左上角坐标。一种实现方式是定义一个基类，其中含有x，y坐标，然后把具体的图形类从该基类中派生出来（笔者过去在用C++开发电力系统软件时就用过这种解决方案），这样所有的图形在Canvas上的位置就确定了。然而，这种解决方案，在WPF中会行不通：WPF支持多种排版。比如说，同样的图形元素，现在要改在Grid上显示了，这时，就不能使用左上角x，y坐标，而要指出图形在Grid中的行列号。又如，同样的图形要在DockPanel中显示时，所要给出的居然是Top、Left、Right或Bottom这样的相对位置。显然，由于WPF排版的多样性，过去所用的解决问题的方法已经不够用了。

WPF引入了附加属性来解决这个问题：图形元素，比如矩形，就不需要预先说明它们在窗口内的位置，当要在某个特定的排版环境中显示时，可以把排版类的某些相关属性引入进来，从而确定自己在窗口中的位置。

在Grid中显示矩形的例子如下：

```
<Window x:Class="Yingbao.Chapter4.AttachedPropertyExample"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Yingbao.Chapter4" Height="300" Width="300">
```

```
    <Grid>
      <Grid.RowDefinitions>
        <RowDefinition  Height ="*"/>
        <RowDefinition  Height ="2*"/>
      </Grid.RowDefinitions>
      <Grid.ColumnDefinitions>
        <ColumnDefinition Width ="*"/>
        <ColumnDefinition Width ="2*"/>
      </Grid.ColumnDefinitions>
      <Rectangle Stroke ="Aqua" Fill ="Brown " Grid.Column ="0"
           Grid.Row ="0"/>
      <Rectangle Stroke ="Blue" Fill ="Yellow " Grid.Column ="1"
           Grid.Row ="0"/>
      <Rectangle Stroke ="Aqua" Fill ="Brown " Grid.Column ="1"
           Grid.Row ="1"/>
      <Rectangle Stroke ="Blue" Fill ="Yellow " Grid.Column ="0"
           Grid.Row ="1"/>
    </Grid>
</Window>
```

本例定义了一个2行2列的Grid，用4个矩形填充这4个网格；每个矩形在Grid中的位置，是由Grid.Column和Grid.Row确定的。显然，这里的Grid.Column和Grid.Row不是矩形的属性，而是Grid的属性。为什么可以把Grid的属性用到矩形Rectangle中来呢？难道WPF有什么魔法不成？在Grid中显示的矩形如图4-5所示。

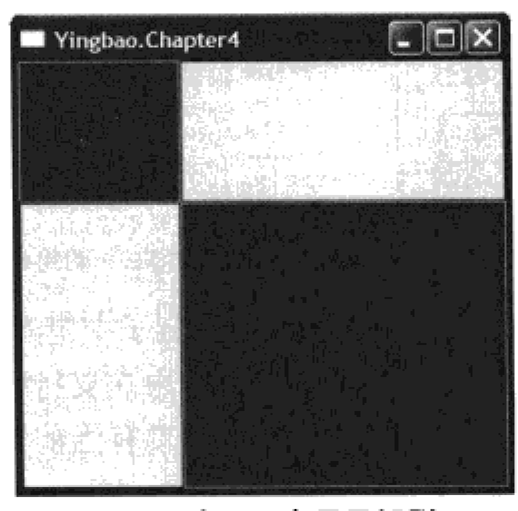


图4-5 在Grid中显示矩形

问题的关键在于Rectangle类具有DependencyObject作为它的基类，图4-6是矩形Rectangle类在WPF中的完整类继承树，由图4-6可见，DependencyObject是其中的一个基类，换句话说，Rectangle是DependencyObject（这是UML的通常表述）。

当我们在XAML中使用下面的语句时：

```
<Rectangle Stroke ="Aqua" Fill ="Brown " Grid.Column ="0" Grid.Row
             ="0"/>
```

设置Grid.Column="0"，实际上调用的是DependencyObject中的(Grid.Column, "0")方法。类似地，要读取附加属性的值，可以调用DependencyObject中的GetValue方法。笔者认为DependencyObject中应该有一个维护相关属性和值之间的哈希表。由此可见，要把相关属性引入到某

个类中，这个类必须是DependencyObject，这一点非常重要。

其次，如要使相关属性能被引入到DependencyObject类中，则一定要在注册该相关属性时。注册相关属性为附加属性的方法是调用DependencyProperty类中的RegisterAttached方法。

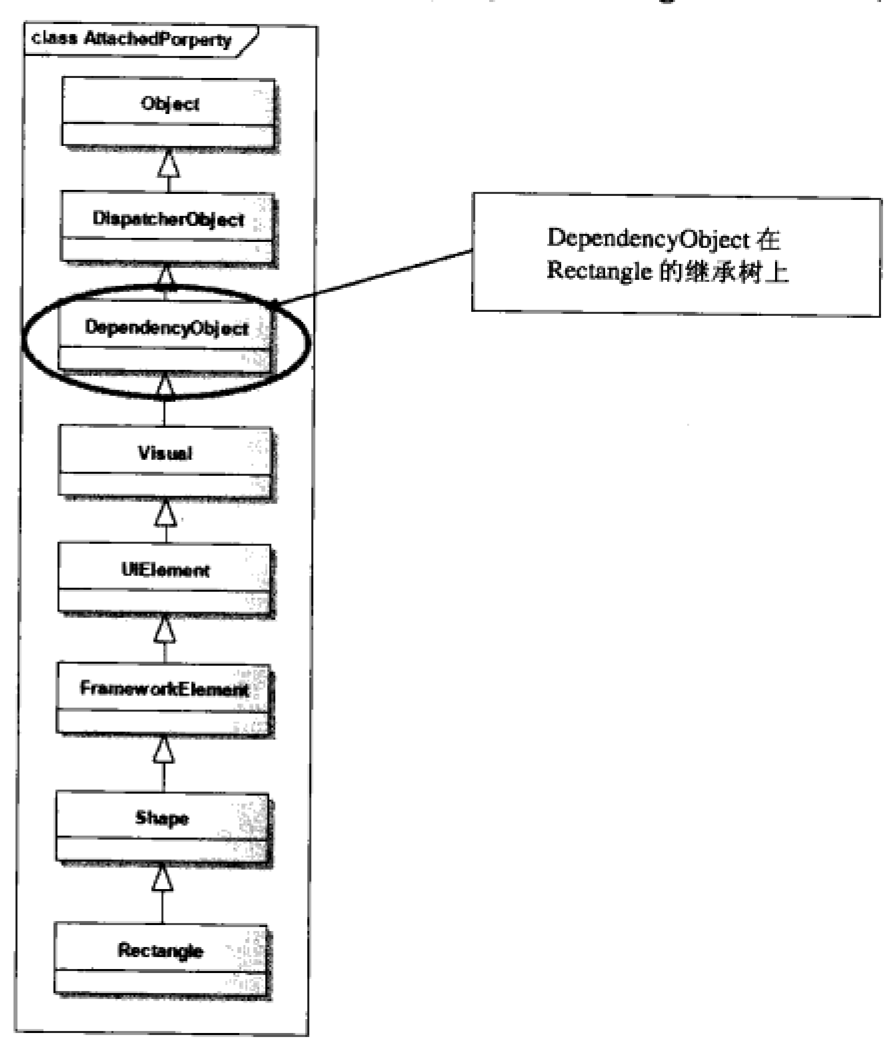


图4-6 WPF中矩形（Rectangle）的继承树

下面来看一个使用附加属性的实际的例子，在这个例子中笔者使用附加属性来实现编辑软件中常用的“重做/取消”功能。WPF的Text Box及相关元素都内在支持“重做/取消”和“拷贝/复制”这样一些常用的编辑功能，但WPF的“重做/取消”功能只对具有当前输入焦点的TextBox有效。

笔者在这里要做的是：“取消和重做”可以对多个TextBox同时进行。

为此，要把用户的操作用堆栈管理起来，当用户输入的时候，把用户的操作放入堆栈，当用户要取消一个输入的时候可把该输入放到另外一堆栈；在用户要进行重做时，只要把用户的输入从另外一个堆栈中取出来就可以了。由于堆栈的先入先出功能，可以很方便地管理用户“最近”的 操作。

首先我们定义一个UndoOperation操作类：

```
public class UndoOperation
  {
      public TextBoxBase Sender;
      public UndoAction Action;
      public UndoOperation(TextBoxBase sender, UndoAction action)
      {
          this.Sender = sender;
```

```
            this.Action = action;
        }
    }
```

这个类中有两个域，一个是Sender表示当前操作哪一个UI元素，其类型为TextBoxBase；另一个是UndoAction。WPF中TextBoxBase是TextBox及RichTextBox两个类的基类（见第6章）。WPF中定义了一个UndoAction枚举类型，可取Create、Clear、none、Undo，Redo、Merge这样一些值。

UndoService是主要完成“取消/重做”操作的类，其中含有两个管理“取消/重做”操作的堆栈：

```
Stack<UndoOperation> undoStack = new Stack<UndoOperation>();
Stack<UndoOperation> redoStack = new Stack<UndoOperation>();
```

如前所述，附加属性是相关属性的一种。和相关属性一样，需要定义一个public static类型为DependencyProperty的变量：

```
public static readonly DependencyProperty SharedUndoRedoScopeProperty =
            DependencyProperty.RegisterAttached("SharedUndoRedoScope",
            typeof(UndoService),
            typeof(UndoService),
            new FrameworkPropertyMetadata(null,
            FrameworkPropertyMetadataOptions.Inherits, new
            PropertyChangedCallback(OnUseGlobalUndoRedoScopeChanged)));
```

与相关属性不同，附加属性需要使用相关属性的RegisterAttached方法注册。在注册的时候同样需要使用FrameworkPropertyMetadata类，来说明附加属性的一些特性。这里笔者说明SharedUndoRedoScopeProperty是可以传递的（设定Inherits标志），并且设置了该属性值发生改变时要调用的处理函数。

定义附加属性时，还要求定义两个读取附加属性的方法，其格式是固定的：

```
public static void SetSharedUndoRedoScope(DependencyObject depObj, bool
        isSet)
   {
     // never place logic in here, because these methods are not called
when things are done in XAML
     depObj.SetValue(SharedUndoRedoScopeProperty, isSet);
   }

public static UndoService GetSharedUndoRedoScope(DependencyObject
                              depObj)
   {
      // never place logic in here, because these methods are not
         called when things are done in XAML
     return depObj.GetValue(SharedUndoRedoScopeProperty) as UndoService;
   }
```

注意：这两个方法都有一个Dependency类型的参数，由此可知，附加属性只能附加到DependencyObject上，而且附加属性的值存储在该DependencyObject实例内！理解这一点，是理解附加属性的关键，这就是附加的由来：

```
depObj.SetValue(SharedUndoRedoScopeProperty, isSet);
```

在SharedUndoRedoScopeProperty属性值发生变化时，WPF属性系统会调用在注册附加属性时设置的回调函数：

```
OnUseGlobalUndoRedoScopeChanged
```

在这个回调函数中，笔者设置或去除了相应的事项处理函数。WPF中的传递事件是一种新的事件类型，将在第7章深入讨论，这里的重点是附加属性。完整的UndoService类源程序如下：

```
using System;
using System.Collections.Generic;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Controls.Primitives;
using System.Windows.Documents;
using System.Windows.Input;

namespace Yingbao.Chapter4
{
    public class UndoService
    {
        Stack<UndoOperation> undoStack = new Stack<UndoOperation>();
        Stack<UndoOperation> redoStack = new Stack<UndoOperation>();

        #region Attached Properties
        public static readonly DependencyProperty
              SharedUndoRedoScopeProperty =
                 DependencyProperty.RegisterAttached(
                "SharedUndoRedoScope", typeof(UndoService),

              typeof(UndoService),
              new FrameworkPropertyMetadata(null,
               FrameworkPropertyMetadataOptions.Inherits, new
               PropertyChangedCallback(
               OnUseGlobalUndoRedoScopeChanged)));

        public static void SetSharedUndoRedoScope(DependencyObject
                       depObj, bool isSet)
        {

            depObj.SetValue(SharedUndoRedoScopeProperty, isSet);
        }

        public static UndoService GetSharedUndoRedoScope(
                     DependencyObject depObj)
        {
           return depObj.GetValue(SharedUndoRedoScopeProperty) as
                 UndoService;
        }

        private static void OnUseGlobalUndoRedoScopeChanged(
                  DependencyObject depObj,
                  DependencyPropertyChangedEventArgs args)
        {
```

```
    if (depObj is TextBoxBase)
    {
        if (args.OldValue != null)
        {
            RemoveEventHandlers(depObj as TextBoxBase,
               args.OldValue as UndoService);
        }
        if (args.NewValue != null)
        {
            AttachEventHandlers(depObj as TextBoxBase,
              args.NewValue as UndoService);
        }
    }
}

private static void AttachEventHandlers(TextBoxBase textBox,
         UndoService service)
{
    if (textBox != null && service != null)
    {
        textBox.AddHandler(CommandManager.PreviewExecutedEvent,
                new ExecutedRoutedEventHandler(
                  service.ExecutedHandler), true);
         textBox.TextChanged += new TextChangedEventHandler(
                  service.TextChangedHandler);
    }
}

private static void RemoveEventHandlers(TextBoxBase textBox,
          UndoService service)
{
    if (textBox != null && service != null)
    {

      textBox.RemoveHandler(CommandManager.PreviewExecutedEvent,
             new  ExecutedRoutedEventHandler(
             service.ExecutedHandler));
      textBox.TextChanged -= new
            TextChangedEventHandler(service.TextChangedHandler);
    }
}

#endregion

void TextChangedHandler(object sender, TextChangedEventArgs e)
{
    this.AddUndoableAction(sender as TextBoxBase, e.UndoAction);
}

private void ExecutedHandler(object sender,
              ExecutedRoutedEventArgs e)
{
    if (e.Command == ApplicationCommands.Undo)
```

```
    {
        e.Handled = true;
        Undo();
    }

    if (e.Command == ApplicationCommands.Redo)
    {
        e.Handled = true;
        Redo();
    }
}

private void AddUndoableAction(TextBoxBase sender,
              UndoAction action)
{
    if (action == UndoAction.Undo)
    {
        redoStack.Push(new UndoOperation(sender, action));
    }
    else
    {
        if (undoStack.Count > 0)
        {
            UndoOperation op = undoStack.Peek();
            if ((op.Sender == sender) && (action ==
                    UndoAction.Merge))
            {
                // no-op
            }
            else
            {
                PushUndoOperation(sender, action);
            }
        }
        else
        {
            PushUndoOperation(sender, action);
        }
    }
}

private void PushUndoOperation(TextBoxBase sender,
            UndoAction action)
{
    undoStack.Push(new UndoOperation(sender, action));
    System.Diagnostics.Debug.WriteLine("PUSHED");
}

public void Undo()
{
    if (undoStack.Count > 0)
    {
        UndoOperation op = undoStack.Pop();
```

```
                op.Sender.Undo();
                op.Sender.Focus();
            }
        }

        public void Redo()
        {
            if (redoStack.Count > 0)
            {
                UndoOperation op = redoStack.Pop();
                op.Sender.Redo();
                op.Sender.Focus();
            }
        }
    }

    public class UndoOperation
    {
        public TextBoxBase Sender;
        public UndoAction Action;

        public UndoOperation(TextBoxBase sender, UndoAction action)
        {
            this.Sender = sender;
            this.Action = action;
        }
    }
}
```

有了这个UndoService类，就可以对在窗口中的多个TextBox元素进行统一管理了。下面是使用UndoService的XAML的一个示例：

```
<Window x:Class="Yingbao.Chapter4.AttachedPropertyWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="UndoRedo" Height="600" Width="400"
    xmlns:undo="clr-namespace:Yingbao.Chapter4">
  <StackPanel>
    <Border BorderBrush="Orange" BorderThickness="3" Margin="5"
            CornerRadius="2">
      <Grid Background="#feca00" >
        <undo:UndoService.SharedUndoRedoScope>
          <undo:UndoService />
        </undo:UndoService.SharedUndoRedoScope>
        <Grid.ColumnDefinitions>
          <ColumnDefinition Width="75" />
          <ColumnDefinition Width="*" />
        </Grid.ColumnDefinitions>
        <Grid.RowDefinitions>
          <RowDefinition Height="25" />
          <RowDefinition Height="25" />
          <RowDefinition Height="25" />
          <RowDefinition Height="25" />
```

```
      <RowDefinition Height="*" />
    </Grid.RowDefinitions>
    <TextBlock Text="设定重做范围" FontSize="22" Foreground="#58290a"
           Grid.ColumnSpan="2" />
    <TextBlock Grid.Row="1" Grid.Column="0">姓名:</TextBlock>
    <TextBox Grid.Row="1" Grid.Column="1" Margin="1" />
    <TextBlock Grid.Row="2" Grid.Column="0">毕业院校:</TextBlock>
    <TextBox Grid.Row="2" Grid.Column="1" Margin="1" />
    <TextBlock Grid.Row="3" Grid.Column="0">工作简历:</TextBlock>
    <RichTextBox Grid.Column="0" Grid.Row="4" Grid.ColumnSpan="2"
            Margin="1" Height="70" />
  </Grid>
</Border>
<Border BorderBrush="#58290a" BorderThickness="3" Margin="5"
            CornerRadius="2">
  <Border.Resources>
   <!-- undo managers can also be created and used as resources -->
    <undo:UndoService x:Key="undoService" />
  </Border.Resources>
  <Grid undo:UndoService.SharedUndoRedoScope="{StaticResource
        undoService}" Background="LightGray " >
    <Grid.ColumnDefinitions>
      <ColumnDefinition Width="75" />
      <ColumnDefinition Width="*" />
    </Grid.ColumnDefinitions>
    <Grid.RowDefinitions>
      <RowDefinition Height="25" />
      <RowDefinition Height="25" />
      <RowDefinition Height="25" />
      <RowDefinition Height="25" />
      <RowDefinition Height="*" />
    </Grid.RowDefinitions>
    <TextBlock Text="设定重做范围" FontSize="22" Foreground="#58290a"
           Grid.ColumnSpan="2" />
    <TextBlock Grid.Row="1" Grid.Column="0">姓名:</TextBlock>
    <TextBox Grid.Row="1" Grid.Column="1" Margin="1" />
    <TextBlock Grid.Row="2" Grid.Column="0">毕业院校:</TextBlock>
    <TextBox Grid.Row="2" Grid.Column="1" Margin="1" />
    <TextBlock Grid.Row="3" Grid.Column="0">工作简历:</TextBlock>
    <RichTextBox Grid.Column="0" Grid.Row="4" Grid.ColumnSpan="2"
        Margin="1" Height="70" />
  </Grid>
</Border>
<Border BorderBrush="DarkGreen" BorderThickness="3" Margin="5"
        CornerRadius="2">
  <Grid Background="LightGreen" >
    <Grid.ColumnDefinitions>
      <ColumnDefinition Width="75" />
      <ColumnDefinition Width="*" />
    </Grid.ColumnDefinitions>
    <Grid.RowDefinitions>
      <RowDefinition Height="25" />
```

```
            <RowDefinition Height="25" />
            <RowDefinition Height="25" />
            <RowDefinition Height="25" />
            <RowDefinition Height="*" />
          </Grid.RowDefinitions>
          <TextBlock Text="未设定重做范围" FontSize="22"
              Foreground="DarkGreen"  Grid.ColumnSpan="2" />
          <TextBlock Grid.Row="1" Grid.Column="0">姓名:</TextBlock>
          <TextBox Grid.Row="1" Grid.Column="1" Margin="1" />
          <TextBlock Grid.Row="2" Grid.Column="0">毕业院校:</TextBlock>
          <TextBox Grid.Row="2" Grid.Column="1" Margin="1" />
          <TextBlock Grid.Row="3" Grid.Column="0">工作简历:</TextBlock>
          <RichTextBox Grid.Column="0" Grid.Row="4" Grid.ColumnSpan="2"
              Margin="1"  Height="70" />
        </Grid>
      </Border>
    </StackPanel>
</Window>
```

图4-7是这段程序的运行结果，这个程序可以作为人事部门管理软件的基本界面。需要指出的是：要对“取消/重做”功能进行测试，要用到标准的快捷键。“取消”的快捷键是“Ctrl+Z”；“重做”的快捷键是“Ctrl+Y”。这些快捷键在WPF中是内嵌的，不需要我们做什么工作。在上面的窗口中，前两个输入框都使用了SharedUndoRedoScope属性，而最后一个输入框没有使用SharedUndoRedoScope属性。在前两个输入框的姓名、毕业院校、工作简历中输入信息，然后用快捷键“Ctrl+Z”和“Ctrl+Y”进行“取消/重做”操作，可以看到，这种操作对这三个TextBox都有效。而在最后一组中，这种操作只对具有当前输入焦点的TextBox有效。

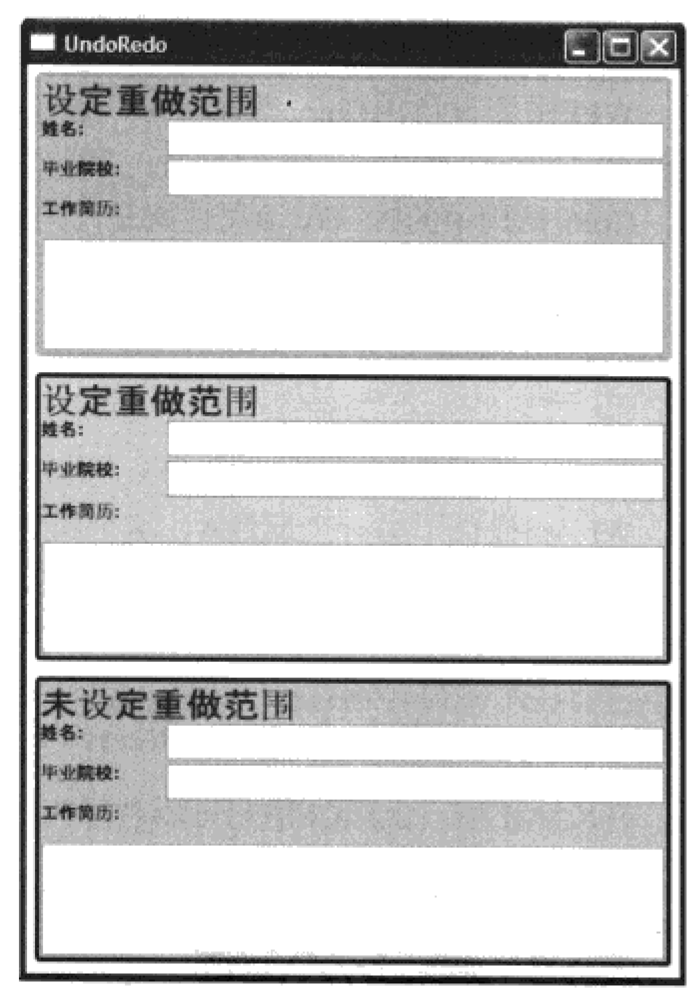


图4-7 使用附加属性实现的“取消/重做”功能

4.5 本章小结

本章介绍了WPF中两个重要的概念——相关属性和附加属性，附加属性是通过相关属性来实现的，也可以说附加属性是相关属性的一种特殊的形式。WPF的属性系统是XAML得以实现的基础，很多功能都建立在相关属性之上。

第5章 画笔和画刷

假如你在海边度假，海风习习，阳光和煦。孩子们在海滩上奔跑着，在海水里嬉闹着。极目远处，一艘艘游艇在海天相接处，仿佛静止在目光的尽头；一只只海鸥，不经意地在你的视野里飞翔。这时你突然有一种作画的冲动，想要把这祥和的大自然描绘下来。你不必是专业画家，画好画坏对你来说都不重要，但你需要画笔、画刷、画板和颜料，你需要用它们来宣泄自己的情感。在计算机里写图形界面程序也一样，你也需要画笔、画刷、画板和颜料。本书第3章讨论了画布面板类（Canvas），本章将讨论WPF作图的基本工具：画笔和画刷，颜料就是色彩，WPF 提供的色彩可丰富了！

5.1 WPF中的颜色

WPF中的颜色用Color结构表示，这个Color结构位于System.Windows.Media命名空间中。我们知道颜色是用红（Red），绿（Green）和蓝（Blue）三种颜色调配出来的，WPF也支持这种颜色调配方法。Color结构中的R、G、B属性就是用来说明你要使用的颜色的，比如，你要用红色：

```
Color lColor =  new Color();
lColor.R= 255;
lColor.G= 0;
lColor.B= 0;
```

R、G、B属性的范围从0到255，这和常规Windows操作系统中编程一样。除了R、G、B属性之外，WPF还提供了另一个属性A。这个属性表示颜色的透明度（或叫不透明度），其值也在0和255之间。若A取为0，则表示颜色透明，若A取为255，则表示颜色不透明。不过，还可以使用带有参数的构造函数来创建颜色：

```
Color lColor =  Color.FromRgb(r,g,b);
Color lColor =  Color.FromArgb(r,g,b);
```

在XAML里，上述C#语句可以表示为：

```
<Color B ="0" G ="0" R ="255"/>
<Color B ="0" G ="0" R ="255" A ="100" />
```

实际上，你可能很少需要在XAML中创建颜色对象，更可能是创建画笔或画刷。由于ARGB的组合值可以表示4228250625（255*255*255*255）种颜色，最好的办法是使用程序来实际观察这些组合的效果。为此，笔者写了下面的测试程序：

```
<Window x:Class="Yingbao.Chapter5.TestColorProperty"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="TestColorOpacity" Height="300" Width="600">
<Window.Resources>
 <Color B ="100" G ="100" R ="100" A ="100" x:Key="myColor" />
</Window.Resources>
<Grid>
```

```
<Grid.RowDefinitions>
      <RowDefinition Height ="auto"/>
      <RowDefinition Height ="auto"/>
 </Grid.RowDefinitions>
<Grid.ColumnDefinitions>
     <ColumnDefinition Width="*"/>
     <ColumnDefinition Width="*"/>
      <ColumnDefinition Width="*"/>
     <ColumnDefinition Width="*"/>
</Grid.ColumnDefinitions>
<Rectangle Name ="dispRect" Stroke ="DarkGreen"
    StrokeThickness="5" Fill="Black" Grid.Column ="1"
    Grid.Row ="1" Grid.ColumnSpan ="2" Height ="100"
    Margin ="20,10,20,10"/>
 <StackPanel  Grid.Column ="0" Grid.Row ="0"
    Orientation ="Horizontal" >
    <TextBlock FontSize ="12" Margin ="10,30,5,20"
       Padding ="10">红:</TextBlock>
    <Slider Orientation ="Vertical"  Value="0"  Height ="80"
       Minimum="0" Maximum="255" TickPlacement="BottomRight"
       TickFrequency="10" IsSnapToTickEnabled="True" Margin
       ="0,0,5,0" AutoToolTipPlacement="BottomRight"
       AutoToolTipPrecision="2"
       ValueChanged="OnRedSliderValueChange"/>
 </StackPanel>
 <StackPanel  Grid.Column ="1" Grid.Row ="0"
       Orientation ="Horizontal" >
   <TextBlock FontSize ="12" Margin ="10,30,5,20"
       Padding ="10">绿:</TextBlock>
    <Slider Orientation ="Vertical"  Value="0"  Height ="80"
       Minimum="0" Maximum="255" TickPlacement="BottomRight"
       TickFrequency="10" IsSnapToTickEnabled="True" Margin
       ="0,0,5,0" AutoToolTipPlacement="BottomRight"
       AutoToolTipPrecision="2"
       ValueChanged="OnGreenSliderValueChange"/>
 </StackPanel>
 <StackPanel  Grid.Column ="2" Grid.Row ="0" Orientation
       ="Horizontal" >
       <TextBlock FontSize ="12" Margin ="10,30,5,20" Padding ="10">
           蓝:</TextBlock>
       <Slider Orientation ="Vertical" Height ="80" Minimum="0"
            Maximum="255" TickPlacement="BottomRight"
      TickFrequency="10" IsSnapToTickEnabled="True" Margin
      ="0,0,5,0" AutoToolTipPlacement="BottomRight"
      AutoToolTipPrecision="2"
            ValueChanged="OnBlueSliderValueChange"/>
</StackPanel>
<StackPanel  Grid.Column ="3" Grid.Row ="0" Orientation
      ="Horizontal" >
      <TextBlock FontSize ="12" Margin ="10,30,5,20" Padding ="10">
       透明度:</TextBlock>
      <Slider Orientation ="Vertical"  Height ="80" Minimum="0"
```

```
            Maximum="255" TickPlacement="BottomRight"
      TickFrequency="10" IsSnapToTickEnabled="True" Margin
      ="0,0,5,0" AutoToolTipPlacement="BottomRight"
      AutoToolTipPrecision="2"
      ValueChanged="OnOpacitySliderValueChange"
     Value="255"/>
</StackPanel>
     </Grid>
</Window>
```

上面这段程序使用表格（Grid）排版。笔者首先创建了2行4列的网格，在第2行的1，2两列，放一个矩形图形。这个矩形的填充色可以由第一行中的4个滑块控件来调整，这四个滑块控件依次调整红、绿、蓝及透明度的值。在该程序中滑块的滑动范围从0到255。TickPlacement设定显示刻度的位置，TickFrequency设置每个刻度的大小。这里TickFrequency设为10，所以在每个滑块的标尺上应有255/10=25个刻度。AutoToolTipPlacement用来设置显示滑块数值的位置，AutoToolTipPrecision用来设置显示滑块数值的精度，这里设为2，即显示小数点后两位（在第6章还要讨论Slider控件）。实际上，我们的ARGB为0~255之间的整数，所以精度对我们来说并不重要。

应当注意，我们必须把创建矩形的XAML语句放在创建滑块语句之前，原因是设定滑块值时，需要用到矩形对象，若没有创建矩形对象，C#会产生NullReference异常。处理滑块移动时的C#程序如下：

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Media.Imaging;
using System.Windows.Shapes;
namespace Yingbao.Chapter5
{
    public partial class TestColorProperty :System.Windows.Window
    {
        double lRed = 0;
        double lGreen = 0;
        double lBlue = 0;
        double lOpacity = 255;
        public TestColorProperty()
        {
            InitializeComponent();
        }

    private void OnRedSliderValueChange(object sender,
           RoutedPropertyChangedEventArgs<double> e)
        {
            lRed = e.NewValue;
            Color clr = new Color();
            clr.R = Convert.ToByte( lRed );
```

```
            clr.G = Convert.ToByte( lGreen );
            clr.B = Convert.ToByte( lBlue );
            clr.A = Convert.ToByte( lOpacity );
            dispRect.Fill = new SolidColorBrush(clr);
        }

   private void OnOpacitySliderValueChange(object sender,
              RoutedPropertyChangedEventArgs<double> e)
        {
            lOpacity = e.NewValue;
            Color clr = new Color();
            clr.R = Convert.ToByte( lRed );
            clr.G = Convert.ToByte( lGreen );
            clr.B = Convert.ToByte( lBlue );
            clr.A = Convert.ToByte( lOpacity );
            dispRect.Fill = new SolidColorBrush(clr);
        }

        private void OnGreenSliderValueChange(object sender,
              RoutedPropertyChangedEventArgs<double> e)
        {
            lGreen = e.NewValue;
            Color clr = new Color();
            clr.R = Convert.ToByte(lRed);
            clr.G = Convert.ToByte(lGreen);
            clr.B = Convert.ToByte(lBlue);
            clr.A = Convert.ToByte(lOpacity);
            dispRect.Fill = new SolidColorBrush(clr);
        }

        private void OnBlueSliderValueChange(object sender,
                   RoutedPropertyChangedEventArgs<double> e)
        {
            this.lBlue = e.NewValue;
            Color clr = new Color();
            clr.R = Convert.ToByte(lRed);
            clr.G = Convert.ToByte(lGreen);
            clr.B = Convert.ToByte(lBlue);
            clr.A = Convert.ToByte(lOpacity);
            dispRect.Fill = new SolidColorBrush(clr);
        }
    }
}
```

TextColorProperty类是从Window类中派生出来的，该类定义了四个类型为double的域：lRed、lGreen、lBlue、lOpacity，用来存放红、绿、蓝及透明度值。然后是四个滑块滑动事项的处理函数，每个函数读出相应于红、绿、蓝及透明度的值，并用这几个值来设定Color中的相应属性，最后创建该颜色的SolidColorBrush（本章后面会讲到这个类），并把该画刷作为矩形的填充画刷。

上面的这段程序并不简练，可以看到，在XAML里，使用的4个StackPanel其中的内容基本相同，而C#中的4个事项处理函数也很类似，有没有更简练的方法呢？答案是肯定的，但要用到模板和数据

绑定等技术，本书的第10章和第11章将讨论模板和数据绑定等相关技术。这段程序的运行结果如图5-1所示。

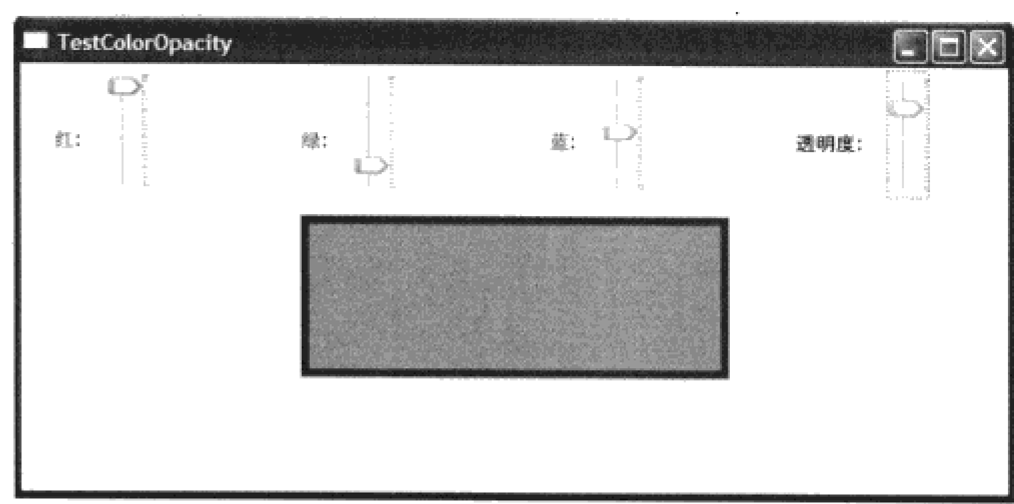


图5-1 自建调整颜色工具

这个程序可以作为调配颜色值的小工具。

在WPF的Color结构中，除了上面的红、绿、蓝及透明度属性之外，还有另外一套属性，即ScA、ScR、ScG和ScB的值。和标准的A、R、G、B只用一个字节不同，ScA、ScR、ScG和ScB是一个64位的浮点数。WPF的Color结构维护这两组属性间的关系。ScA取值在0和1之间相应于A在0到255之间的数值，ScR和R，ScG和G，ScB和B也有同样的关系，但这种对应关系并不是线性的。ScR等属性可以大于1或者小于0，由此可见，用Sc这套属性极大地扩充了过去可表示的颜色的范围。在更普遍的意义上来说，把一个模拟量数字化时，所取的采样点越多则所表示的模拟量就越精确。现代某些计算机的外围设备，如高档数字相机所展现出的自然色彩，已经不是过去把红色分为255种所能表示的了，这便是WPF采用ScA等64位属性的原因。

虽然WPF定义了64位的颜色表示方法，其可以表示的颜色已接近于实际意义上的无穷大了。但除了专门处理图像的软件（如Photoshop等），通常程序里用的色彩可能就几十种。为了方便大量的用户在WPF里使用颜色，WPF定义了一个颜色的枚举类型Colors，其中含有141个带名字的常用颜色值。这极大地方便了WPF程序员。然而，这些颜色的名字是用英文标记的，对于母语不是英文的程序员来说，并不直观。一种方法是使用微软提供的颜色表（http://msdn.microsoft. com/en-us/library/system.windows.
media.colors.aspx），另一种方法就是自己写一个小程序。下面的程序就是笔者写的一段小程序，这个程序可以把颜色的名字和其表示的颜色显示出来。

```
<Window x:Class="DisplayNamedColor.WNamedColor"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="DisplayNamedColor" Height="500" Width="300"
    ScrollViewer.VerticalScrollBarVisibility="Auto" >
  <ScrollViewer HorizontalScrollBarVisibility ="Auto ">
  <StackPanel Name ="TopPanel" Height ="Auto" />
  </ScrollViewer>
</Window>
```

这段XAML程序定义了显示窗口框架。主窗口下包含一个ScrollViewer控件，其中含有一个StackPanel控件，名字为TopPanel。当StackPanel中的内容大小超过窗口大小时，ScrollViewer会自动显示滚动条。注意笔者把ScrollViewer的VerticalScrollBarVisibility用在Window中，这是因为该属性是附加属性。在Window类中设置该属性，可以传递到其中的子UI元素中。

可以看到这段XAML非常简单，这是因为在这个程序中，笔者没有把定义表格（Grid）类放在XAML中，否则XAML会很长。由此也可以看出，XAML语言的局限：它没有循环语句，如for、while等。当UI元素重复出现的时候，使用XAML就会很累赘。在事先不知道UI元素的数目时，使用XAML就不仅是累赘，而是不可能了。

为此笔者把定义网格的任务交给C#:

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Data;
using System.Windows.Documents;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Media.Imaging;
using System.Windows.Shapes;
using System.Reflection;
namespace DisplayNamedColor
{

    public partial class WNamedColor : System.Windows.Window
    {
        public WNamedColor()
        {
            InitializeComponent();
            Init();
        }

        private void Init()
        {
            Grid  gmain= new Grid();
            TopPanel.Children .Add(gmain);
            ColumnDefinition coldef = new ColumnDefinition();
            coldef.Width = new
                        GridLength( 200,GridUnitType.Pixel  ) ;
            gmain.ColumnDefinitions.Add(coldef);
            coldef = new ColumnDefinition();
            coldef.Width = new GridLength (100,
                            GridUnitType.Pixel );
            gmain.ColumnDefinitions.Add(coldef);
            PropertyInfo[] pilist =
                          typeof(Colors).GetProperties();
            int i = 0;
            foreach (PropertyInfo pi in
```

```
                    typeof(Colors).GetProperties())
            {
                RowDefinition rowDef = new RowDefinition();
                rowDef.Height = new GridLength(100,
                         GridUnitType.Auto);
                gmain.RowDefinitions.Add(rowDef);
                Color mColor;
                mColor = (Color)pi.GetValue(null, null);
                Label lb = new Label();
                lb.Content = pi.Name;
                lb.HorizontalAlignment =
                             HorizontalAlignment.Center;
                gmain.Children.Add(lb);
                Grid.SetRow(lb, i);
                Grid.SetColumn(lb, 0);
                Rectangle rect = new Rectangle();
                rect.Fill = new SolidColorBrush(mColor );
                rect.Width = 100;
                rect.Height = 20;
                gmain.Children.Add(rect);
                Grid.SetRow(rect, i);
                Grid.SetColumn(rect, 1);
                ++i;
            }

            GridSplitter split = new GridSplitter();
            split.HorizontalAlignment =
                  HorizontalAlignment.Center;
            split.Width = 6;
            gmain.Children.Add(split);
            Grid.SetRow(split, 0);
            Grid.SetColumn(split, 1);
        }
    }
}
```

在这段C#程序中创建了一个Grid对象gmain，并把它作为TopPanel的子元素。然后为gmain定义了两列，其宽度分别为200和100。笔者使用.NET的reflection技术来获取Colors枚举类型中的属性数组：

```
PropertyInfo[] pilist = typeof(Colors).GetProperties();
```

再根据属性的个数来创建表格Grid的行数，其目的是要显示Colors枚举类型中所定义的颜色名及其对应的色彩。gmain中的第一列用来显示颜色的名字，第二列用来显示颜色。颜色只是画笔或者画刷的一个属性，要显示颜色的最简单办法是创建一个简单的图形，为此，笔者再次使用了矩形类，并把从Colors枚举类型中获取的对应于颜色名字的颜色用作矩形的填充色：

```
Rectangle rect = new Rectangle();
rect.Fill = new SolidColorBrush(mColor );
```

在获取mColor的值时，同样是使用.NET的reflection技术：

```
mColor = (Color)pi.GetValue(null, null);
```

有关.NET的reflection技术不是本书要讨论的范围，若读者不熟悉，可以参见.NET的基础书籍或文章。如Jeffrey Richter的《.NET Framework Programming》一书对这一技术有深入的讨论，这段程序的运行结果如图5-2所示。

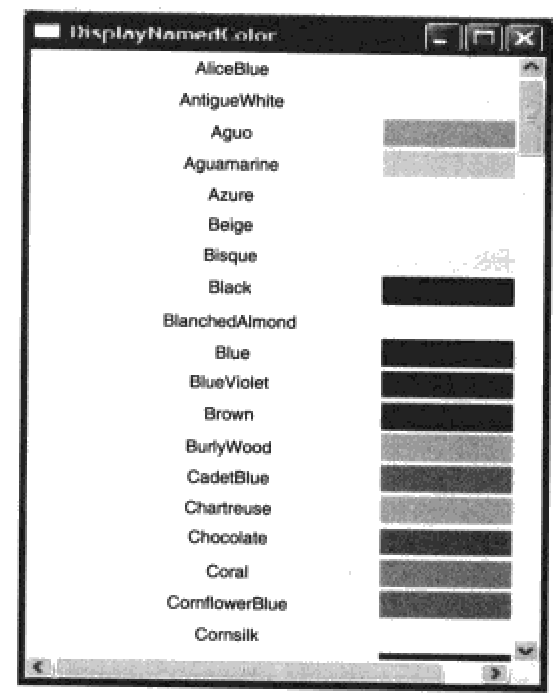


图5-2 WPF定义的颜色名字

5.2 画刷

从前面的例子可以看出在WPF中的颜色是在画刷中体现出来的，就像现实中的颜色必须在某个实体上体现出来一样，比如说红色的牡丹、蓝草莓，这里的颜色都是某个实体的属性。所谓“皮之不存，毛将焉附”，WPF中若没有画刷，颜色也就没有了着力点。

就像画家作画一样，在WPF中编程，程序员合理使用画刷可以产生不同的视觉效果。在软件中，画刷这个概念很早就有，OS2及Win16里就有支持画刷的函数。二十年前开发出的C++语言中就有Brush类，WPF对画刷类进行了重新设计，支持WPF的相关对象、动画和多线程等功能。WPF中画刷的类继承图如图5-3所示。

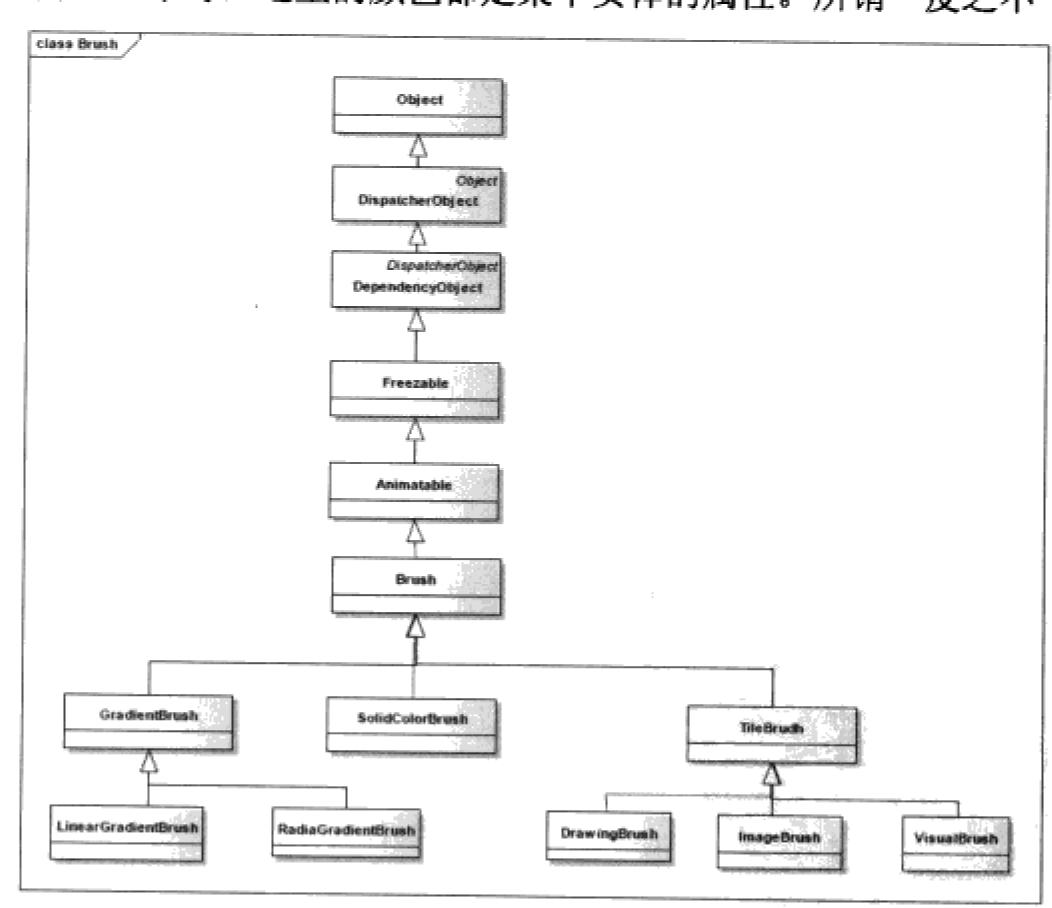


图5-3 WPF中的画刷类

画刷的继承树上有一个Freezable类，这个类提供的特性可以用来改善应用程序的性能。画刷和画笔都是从Freezable类中派生出来的。WPF的Freezable是对过去

C++和Java中的恒定（Immutable）概念的扩展。在C++或Java里，若一个对象声明为恒定变量，那么只能在创建该对象时给该对象赋值，在此之后你就不能再改变该对象的值。把对象声明为恒定变量的好处首先是可以改善应用程序的运行性能，C++和Java都对恒定变量进行了优化；其次恒定对象可以在多线程之间安全传递。

WPF对恒定概念进行了扩展，Freezable类中维护两个状态：只读（Frozen）和读写（Unfrozen）。当IsFrozen返回True时，该对象是只读的；当IsFrozen返回为False时，则该对象为可读写对象。若一个对象处在可读写状态，而其值发生改变时，Freezable类会发出Changed事项。用户可以把一个Freezable对象从读写状态转换为只读状态，方法为：

```
Button myButton = new Button();
SolidColorBrush myBrush = new SolidColorBrush(Colors.Yellow);
if (myBrush.CanFreeze){
     myBrush.Freeze();
}
```

一旦一个Freezable对象被设置为只读状态后，就不能把它再修改为读写状态了。但是你可以通过调用Freezable类的Clone()方法来获取该对象的一个副本：

```
if( myBrush.IsFrozon){
         SolidColorBrush myCloneBrush =  myBrush.Clone();
}
```

新复制的画刷myCloneBrush是可以读写的。

动画类Animatable将在第15章动画中讨论。Brsuh类是所有画刷的基类，它负责管理所有画刷的基本属性，如透明度及变换操作等，这是一个抽象类。

5.2.1 实心画刷（SolidColorBrush）

前面的程序中已经用了SolidColorBrush类，这是画刷类中最简单的一类，它在UI元素中填充指定的颜色。在XAML里使用画刷，有时候并不是很明显的，例如：

<Rectangle Width= "50" Height= "20"Fill= "Red"/>

这里并没有直接使用画刷，实际上上面的XAML语句相当于下面的C#语句：

```
Rectangle rect = new Rectangle();
rect.Fill = new SolidColorBrush(Colors.Red );
rect.Width = 50;
rect.Height = 20;
```

矩形的Fill属性为画刷Brush类型，当XAML遇到Fill= "…" 的语句时，XAML引擎自动把引号里的字符串作为颜色值转换成SolidColorBrush。当然，如果你不喜欢使用这种隐含转换的XAML语句，也可以用更明了的形式：

```
<Rectangle Width= "50" Height= "20">
   <Rectangle.Fill>
      <SolidColorBrush Color= "Red"/>
   </Rectangle.Fill>
 <Rectangle>
```

其效果是一样的。在本章的第一个有关颜色的例子里，还设置了透明度，在SolidColorBrush中，也可以设定透明度的值：

```
<Rectangle Width="50" Height="20" Fill="#FFFF0000" />
```

该语句把矩形中的填充画刷设为完全不透明的红色。符号“#”在XAML中表示十六进制数字。

5.2.2 梯度画刷（GradientBrush）

和SolidColorBrush只使用一种颜色来绘制某个区域不同，梯度画刷则是把颜色按梯度分布的方式来填充所要绘制的区域。根据颜色梯度分布方式的不同，WPF把这种梯度画刷分为两类，即线性梯度画刷（LinearGradientBrush）和圆弧梯度画刷(RadialGradientBrush)。

5.2.3 线性梯度画刷（LinearGradientBrush）

在线性梯度画刷中，颜色梯度是沿着一条直线分布的。这条直线就是LinearGradientBrush类中的StartPoint和EntPoint两点的连线，StartPoint的默认值是（0，0），EndPoint的默认值是（Cos α，Sin α）；其中α为射线和X坐标轴间的夹角，逆时针方向为正。垂直于这条直线的颜色是一样的，颜色沿着这条射线按梯度分布，可以通过改变角度来控制梯度分布的方式。直线梯度画刷中的StartPoint和EntPoint如图5-4所示。

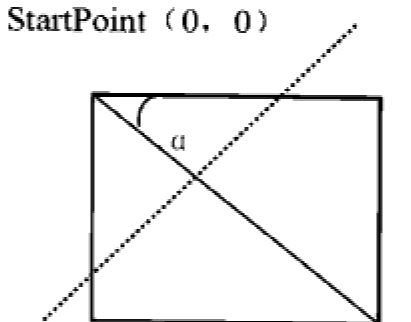


图5-4 直线梯度画刷中的StartPoint和EndPoint

一个通过改变角度α的值来控制直线梯度画刷的例子如下：

```
<Window x:Class="Yingbao.Chapter5.LinearGradientBrushWin"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Yingbao.Chapter5" Height="300" Width="400" >
  <Canvas>
    <Rectangle Width="200" Height="100" Name ="myRectangle"
       Canvas.Left ="20"  Canvas.Top ="20" Stroke ="Black "/>
    <TextBox Name ="txtAngleValue" Width ="40" Height ="20"
       Canvas.Left ="20" Canvas.Top ="130"/>
    <ScrollBar Orientation ="Horizontal " Width ="360"
       ValueChanged ="OnValueChanged"  Canvas.Left ="20"
       Canvas.Top ="150"  MinWidth ="0" Maximum ="360"/>
  </Canvas>
</Window>
```

笔者在这里使用了画布面板(Canvas)，其中有一个矩形，用来显示调整画刷的效果。TextBox用来显示所调整的角度α的值。ScrollBar用来调整角度α，其调整的范围为0°～360°，即整个圆周。

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Data;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Media.Imaging;
using System.Windows.Shapes;
namespace Yingbao.Chapter5
{
    public partial class LinearGradientBrushWin :
        System.Windows.Window
    {
        public LinearGradientBrushWin()
        {
            InitializeComponent();
        }
        void OnValueChanged(object sender,
            RoutedPropertyChangedEventArgs<double>  ea)
        {
            double angle = ea.NewValue;
            LinearGradientBrush mBrush = new LinearGradientBrush(
              Colors.Red, Colors.Blue, angle);
            this.myRectangle.Fill = mBrush;
            this.txtAngleValue.Text = angle.ToString();
        }
    }
}
```

在C#中处理移动ScrollBar滑块事件。把ScrollBar的值作为LinearGradientBrush的角度α的值。当旋转角度为0°时的两色直线梯度画刷如图5-5所示。

由图5-5可见，当角度为0°时，颜色的梯度是水平分布的；当角度旋转到90°时，则颜色梯度是垂直分布的，如图5-6所示。

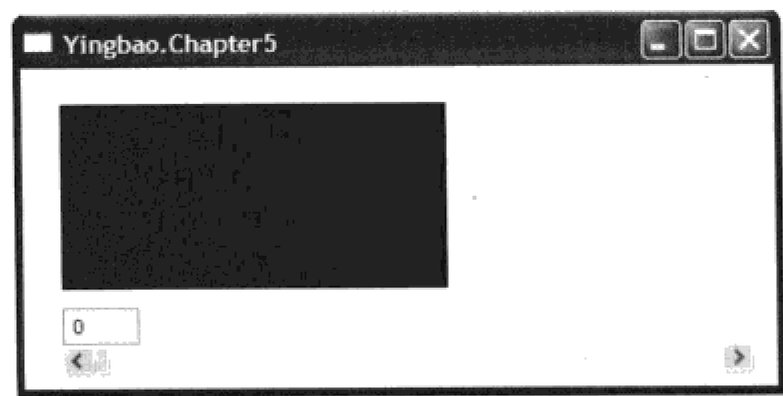


图5-5 旋转角度为0°时的两色直线梯度画刷

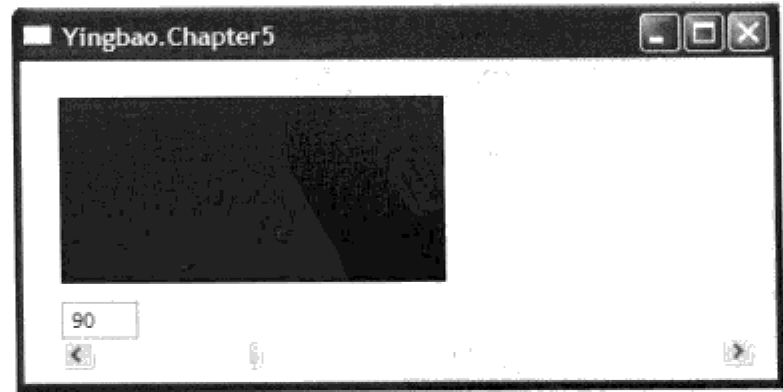


图5-6 当旋转角度为90°时的直线梯度画刷

还可以看到，当旋转角度在180°和270°之间时，整个矩形区域成了一片红色。造成这一结果的原因在于，起始点位置始终在(0,0)，而终点的位置已经在矩形的外面（左上角）。所以如果要取得整个圆周的直线梯度画刷，光调整角度是不够的，需要计算StartPoint和EndPoint坐标。

另外一个问题是：在上面的程序中我们并没有设定直线梯度画刷所使用的图形范围，WPF是根据什么来绘制整个矩形区域的呢？答案是LinearGradientBrush里面有一个SpreadMethod属性，这是一个枚举类型。其可能的值为Pad、Reflect和Repeat，Pad为其默认值。

当SpreadMethod为Repeat时，画刷会在达到EndPoint时，重复起始位置的颜色。例如图中α的值为59.25°，则EndPoint的值为（0.51，0.86）。由于颜色相同的直线和StartPoint及EndPoint连线垂直，所以相同颜色的终点在X方向约86%的位置，Y方向约51%的位置。颜色开始在该处重复（如图5-7所示）。在这里笔者用了解析几何中两条直线垂直，则其斜率的乘积为-1的结论。

若把SpreadMethod设为Reflect会怎么样呢？图5-8示出了把SpreadMethod设为Reflect的运行结果。

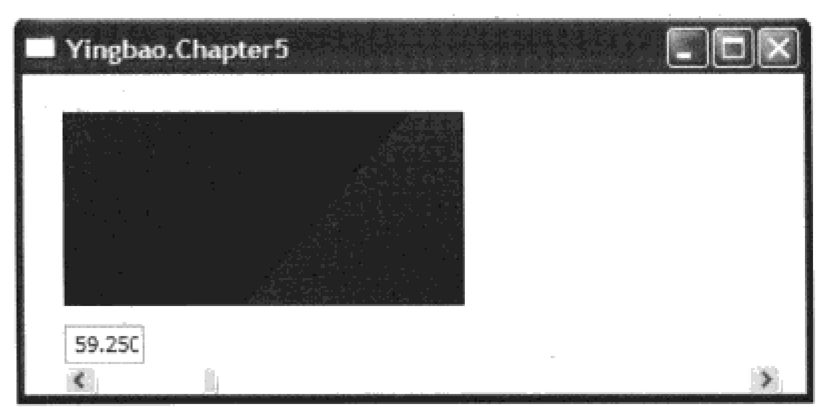


图5-7 SpreadMethod设为Repeat时的直线梯度画刷

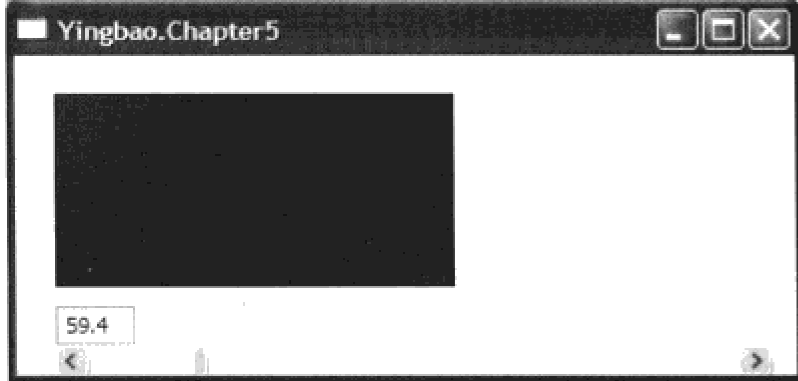


图5-8 SpreadMethod设为Reflect时的直线梯度画刷

比较图5-7和图5-8，可以看到，在相同位置处，图5-7用红色开始绘制（重复StartPoint处的颜色），而图5-8则用蓝色开始绘制（重复EndPoint处的颜色）。所以Reflect的绘制方法就像在EndPoint处立了一面镜子，后面的绘制是对前面的镜像。

WPF的梯度画刷还有一个重要的属性GradientStops，它是一个集合类型。允许在一个梯度画刷里使用多种颜色。下面的这段XAML就创建了一个七种颜色的直线梯度画刷：

```
<Rectangle Width="200" Height="100" Name ="myRectangle"
Canvas.Left ="20"  Canvas.Top ="20" Stroke ="Black ">
      <Rectangle.Fill >
        <LinearGradientBrush StartPoint="0,0" EndPoint="1,1">
          <GradientStop Color="Red" Offset="0.0" />
          <GradientStop Color="Orange"  Offset="0.17" />
          <GradientStop Color="Yellow " Offset="0.33" />
          <GradientStop Color="Green" Offset="0.5" />
          <GradientStop Color="Blue" Offset="0.67" />
          <GradientStop Color="Indigo" Offset="0.84" />
          <GradientStop Color="Violet" Offset="1" />
        </LinearGradientBrush >
      </Rectangle.Fill>
    </Rectangle >
```

其运行结果如图5-9所示。GradientStops的作用是可灵活控制颜色的转换点，及转换的颜色。

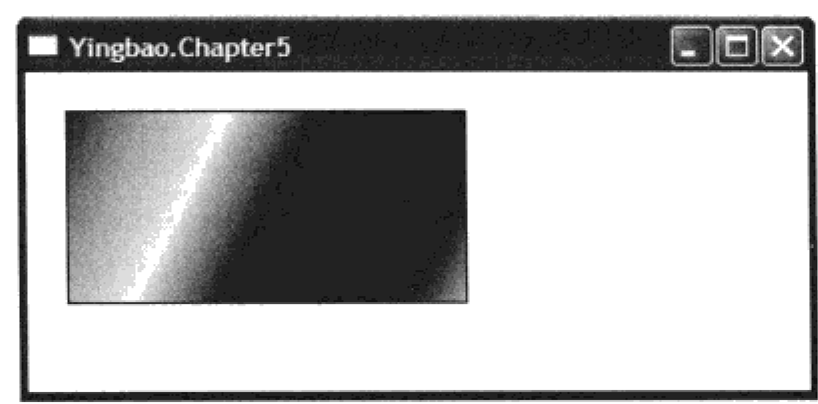


图5-9 多种颜色的直线梯度画刷

5.2.4 圆形梯度画刷（RadialGradientBrush）

上面讨论了直线梯度画刷，还有一种梯度画刷是圆形梯度画刷。由于LinearGradientBrush和RadialGradientBrush都是GradientBrush，故它们有很多特性是相同的。与LinearGradientBrush要说明StartPoint和EndPoint不同，RadialGadientBrush需要说明圆心坐标，及X方向和Y方向的半径：GradientOrigin、Center、RadiusX和RadiusY的值：

```
<Window x:Class="Yingbao.Chapter5.RadialGradientBrushWin"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Yingbao.Chapter5" Height="300" Width="400"  >
  <StackPanel>
    <Rectangle Width="200" Height="200">
      <Rectangle.Fill>
        <RadialGradientBrush GradientOrigin="0.5,0.5"
          Center="0.5,0.5" RadiusX="0.5" RadiusY="0.5">
          <RadialGradientBrush.GradientStops>
            <GradientStop Color="Yellow" Offset="0" />
            <GradientStop Color="Red" Offset="0.25" />
            <GradientStop Color="Blue" Offset="0.75" />
            <GradientStop Color="LimeGreen" Offset="1" />
          </RadialGradientBrush.GradientStops>
        </RadialGradientBrush>
      </Rectangle.Fill>
    </Rectangle>
  </StackPanel>
</Window>
```

上面的这段XAML，笔者创建了一个圆形梯度画刷，其圆形坐标和梯度原点都设为（0.5,0.5），即位于矩形的中心。X方向和Y方向的半径都设为0.5，即为圆形。与LinearGradientBrush一样，RadialGradientBrush也支持GradientStops属性，所以可以设置多种颜色。创建圆形梯度画刷程序的运行结果如图5-10所示。

圆形梯度画刷也含有SpreadMethod属性（从GradientBrush中继承而来），其用法和直线梯度画刷一样，这里不再赘述。

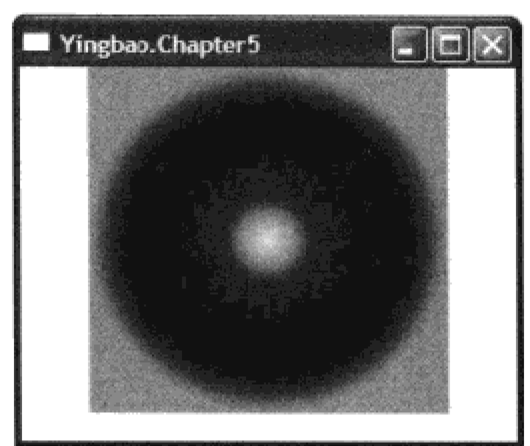


图5-10 圆形梯度画刷

5.2.5 自制画刷（DrawingBrush）

有了前面的两种梯度画刷，自然会想到：倘若能自己创建一种任意图形的画刷，那么用起来一定很方便。比如说，在电力系统中，常常遇到各种标准符号，如变压器、开关等。如我们能创建这些画刷，那么绘制电力系统接线图一定很方便，WPF中的DrawingBrush就支持这个概念。

```
<Window x:Class="Yingbao.Chapter5.TransformerDrawingBrush"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Yingbao.Chapter5" Height="300" Width="300">
    <Grid>
      <Rectangle Width="200" Height="200" Stroke="Blue"
                 StrokeThickness="2">
        <Rectangle.Fill>
          <DrawingBrush  Viewport ="0.1,0.1,0.5,0.5" >
            <DrawingBrush.Drawing>
              <DrawingGroup>
                <DrawingGroup.Children>
                  <GeometryDrawing Brush="White">
                    <GeometryDrawing.Geometry>
                      <GeometryGroup>
                        <EllipseGeometry Center="50,40"
                              RadiusX="45" RadiusY="45"/>
                        <EllipseGeometry Center="50,100"
                              RadiusX="45" RadiusY="45"/>
                        <LineGeometry StartPoint="50,20"
                              EndPoint="30,40" />
                        <LineGeometry StartPoint="50,20"
                              EndPoint="70,40" />
                        <LineGeometry StartPoint="30,40"
                              EndPoint="70,40" />
                        <LineGeometry StartPoint="50,100"
                              EndPoint="50,120" />
                        <LineGeometry StartPoint="50,100"
                              EndPoint="30,90" />
                        <LineGeometry StartPoint="50,100"
                              EndPoint="70,90" />
                      </GeometryGroup>
                    </GeometryDrawing.Geometry>
```

```
                    <GeometryDrawing.Pen>
                      <Pen Thickness="2" Brush="Black" />
                    </GeometryDrawing.Pen>
                  </GeometryDrawing>
                </DrawingGroup.Children>
              </DrawingGroup>
            </DrawingBrush.Drawing>
          </DrawingBrush>
        </Rectangle.Fill>
      </Rectangle>
    </Grid>
</Window>
```

在这段XAML里，笔者创建了一个变压器自制画刷，并把这个画刷作为矩形的填充属性(这和前面把SolidColorBrush作为矩形的填充属性是一样的)。自定义画刷的绘制属性（Drawing），程序中使用了DrawingGroup、GeometryDrawing、GeometryGroup等属性，将第13章讨论WPF图形类时讨论，现在只需要知道，在自制画刷（DrawingBrush）类中可以使用图形元素来绘制任意画刷就可以了。上述XAML的运行结果如图5-11所示。

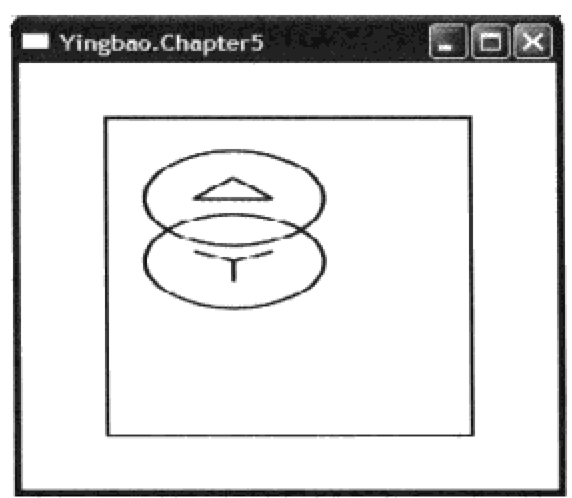


图5-11 用自制画刷（DrawingBrush）创建的变压器画刷

上述XAML代码，创建了一个一边是三角形接法、一边是星形接法的变压器符号。也许你觉得没有什么，需要指出的是由于这个变压器符号是一个画刷，故可以用这个画刷来绘制任何区域！

5.2.6 粘贴模式（TileMode）

自制画刷支持五种粘贴模式，这类似于装修房子时贴地板砖。这五种模式是：none、Tile、FlipX、FlipY、FlipXY。在XAML中设定粘贴模式很简单：

```
<DrawingBrush TileMode ="Tile">
```

用自制画刷设为Tile粘贴模式的结果如图5-12所示。

用自制画刷设为FlipY粘贴模式的结果如图5-13所示。

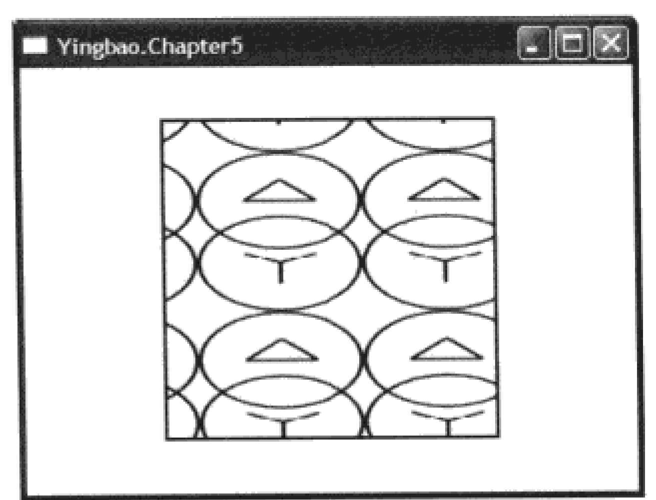


图5-12 用自制画刷设为Tile粘贴模式

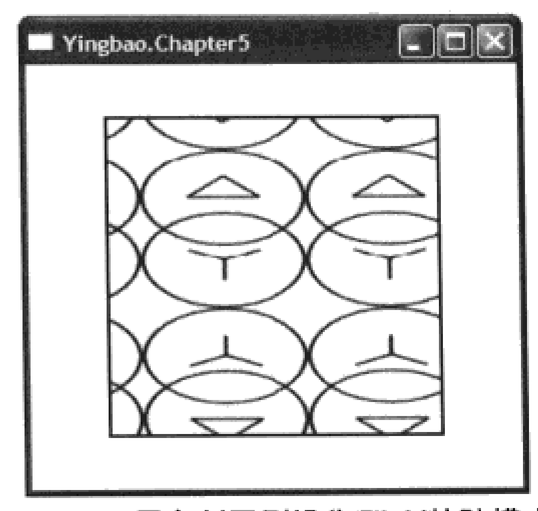


图5-13 用自制画刷设为FlipY粘贴模式

当把自制画刷设为FlipY粘贴模式时，在所绘制的矩形区域内，画刷在Y方向以镜像的方式重复出现；当把自制画刷设为FlipX粘贴模式时，在所绘制的矩形区域内，画刷在X方向以镜像的方式重复出现；当把自制画刷设为FlipXY粘贴模式时，在所绘制的矩形区域内，画刷在X，Y两个方向都以以镜像的方式重复出现；由于我们的变压器画刷在X方向相对于变压器的中线对称，所以我们看不出来；若把变压器画刷中三角形的一个腰去掉，在X方向造成不对称，便可以更容易观察粘贴模式的效果。用自制画刷设为FlipX粘贴模式和FlipXY粘贴模式的结果分别如图5-14和图5-15所示。

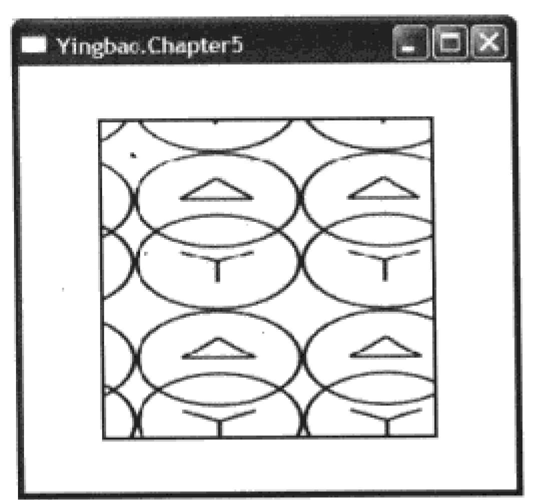


图5-14 用自制画刷设为FlipX粘贴模式

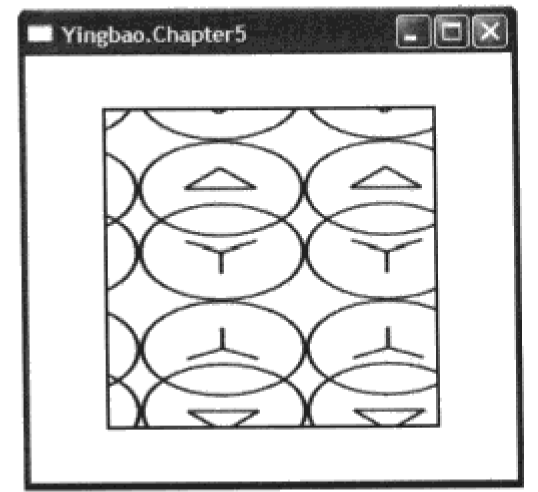


图5-15 用自制画刷设为FlipXY粘贴模式

5.2.7 伸展方式（Stretch）

自制画刷还有一个伸展方式的属性，其取值为none、Fill、Uniform及UniformToFill。它用来控制画刷用什么样的方式在所绘制的区域伸展，其语法如下：

```
<DrawingBrush  Stretch ="UniformToFill">
```

上述语句把自制画刷的伸展方式设为UniformToFill。

5.2.8 图像画刷（ImageBrush）

有了自制画刷以后，有时为了达到一定的效果，需要使用某种图像来绘制图形。这时就需要使用图形画刷。现实生活中最典型的例子是印花布料，可以把位图或数字相机找出的照片来作为画刷。需要注意的是，在这种情况下，使用的图像不再是矢量图形，所以内存的开销是巨大的。

图像画刷中一个重要的属性是ImageSource，其类型是ImageSource。ImageSource支持常用的图像文件格式如：JPG（数字相机的格式）、BMP（位图格式）、PNG（网络兼容的图像格式）等图像格式。

在C#里创建一个图像画刷的语法如下：

```
ImageBrush landBrush = new ImageBrush();
landBrush.ImageSource = new BitmapImage(
       new Uri(@"Image\land.jpg", UriKind.Relative)  );
```

等价的XAML语句：

```
<ImageBrush x:Name ="landBrush"  ImageSource ="Image\land.jpg"/>
```

与使用图像画刷的完整程序如下：

```
<Window x:Class="Yingbao.Chapter5.ImageBrushWin"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Yingbao.Chapter5" Height="200" Width="400" >
    <Grid>
      <Grid.RowDefinitions >
        <RowDefinition Height ="*" />
        <RowDefinition Height ="3*" />
      </Grid.RowDefinitions>
      <Grid.ColumnDefinitions >
        <ColumnDefinition Width ="*"/>
        <ColumnDefinition Width ="*"/>
        <ColumnDefinition Width ="*"/>
      </Grid.ColumnDefinitions>
      <Label Grid.Column ="0" Grid.Row ="0"
            Margin ="5">使用JPG格式</Label>
      <Label Grid.Column ="1" Grid.Row ="0"
            Margin ="5">使用BMP格式</Label>
      <Label Grid.Column ="2" Grid.Row ="0"
            Margin ="5" >使用PNG格式</Label>
      <Rectangle Grid.Column ="0" Grid.Row ="1"
            Stroke ="Blue" StrokeThickness ="3">
        <Rectangle.Fill >
          <ImageBrush x:Name ="landBrush"  ImageSource
             ="Image\land.jpg"/>
        </Rectangle.Fill>
      </Rectangle>
      <Rectangle Grid.Column ="1" Grid.Row ="1"
           Stroke ="Blue" StrokeThickness ="3">
        <Rectangle.Fill >
          <ImageBrush ImageSource ="Image\vase.bmp"/>
        </Rectangle.Fill>
      </Rectangle>
      <Rectangle Grid.Column ="2" Grid.Row ="1"
            Stroke ="Blue" StrokeThickness ="3">
        <Rectangle.Fill >
          <ImageBrush ImageSource ="Image\Simple_tux.png"/>
```

```
        </Rectangle.Fill>
      </Rectangle>
    </Grid>
</Window>
```

这段程序演示了在图像画刷里使用BMP、JPG及PNG文件格式，其运行结果如图5-16所示。

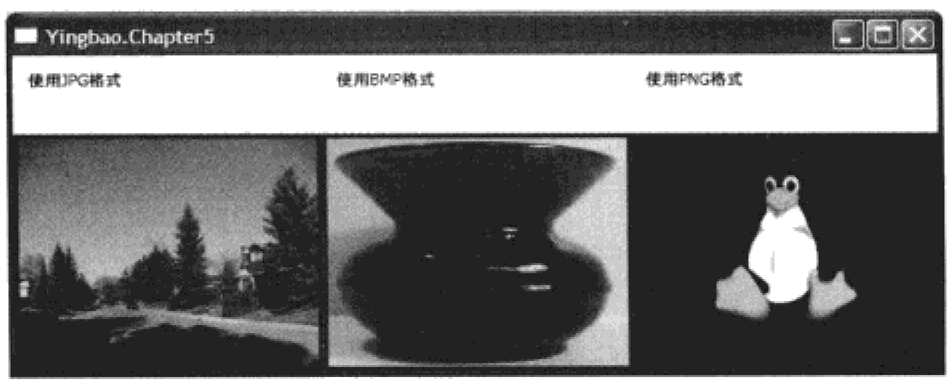


图5-16 图像画刷

其中的第一张图是笔者自己拍的家门前的小街，时间是2007年秋季，瞧：枫叶都红了！

由于画刷可以用在WPF的很多UI元素上，使用图像画刷，再加上设置伸展方式，粘贴模式、往往可以取得很美妙的效果。在字符框中使用图像画刷的效果如图5-17所示。

图5-17 在字符框中使用图像画刷

产生图5-16的XAML程序如下：

```
  <StackPanel >
  <TextBox  Text ="你好，加拿大！"  Height ="200" Width ="300"
FontSize ="25"
                FontWeight ="Bold" Foreground ="Red ">
    <TextBox.Background>
      <ImageBrush x:Name ="landBrush"  ImageSource
="Image\land.jpg"/>
    </TextBox.Background>
  </TextBox>
  </StackPanel>
```

在这里，我把图像画刷作为字符框的背景画刷。

5.2.9 控件画刷（VisualBrush）

有了用图形元素自制的画刷，又有了使用画面的图像画刷，还缺少什么？读者也许会想到，倘若

能用WPF控件来构建画刷，不就可以想用什么作画刷都行了吗？WPF支持这种想法，任何UI控件都可以用来构建一种新的画刷。

笔者在写有关画刷的时候，突然想到中国台湾地区诗人高阳的一组七绝，很有意思。虽然他写的是情诗，但是我觉得用在这里作为VisualBrush可能会有别样的效果。诗是这样的："水阔天长挥手时，待君相送竟迟迟，一朝缘证三生石，如影随行总不离。"到哪里去找"如影随行总不离"的爱情呢？而且还是三生呢！不过，如果在WPF里创建了这样的画刷，不管你用它来画什么，它都会如影随行地跟着你！

使用StackPanel及TextBlock创建了4个VisualBrush的程序如下。

```
<Window x:Class="Yingbao.Chapter5.VisualBrushWin"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Yingbao.Chapter5" Height="400" Width="600"  >
    <Grid>
      <Grid.Resources>
        <VisualBrush x:Key="PoemBrush" Viewport="0,0,1,1"
              TileMode="Tile" Opacity="0.8">
          <VisualBrush.Visual>
            <StackPanel >
              <TextBlock Foreground="Gold">
                    水阔天长挥手时</TextBlock>
              <TextBlock Foreground="LightBlue">
                    待君相送竟迟迟</TextBlock>
              <TextBlock Foreground="LightGray">
                    一朝缘证三生石</TextBlock>
              <TextBlock Foreground="Pink">
                    如影随行总不离</TextBlock>
            </StackPanel>
          </VisualBrush.Visual>
        </VisualBrush>
        <VisualBrush x:Key="PoemBrushFlipX" Viewport=
            "0,0,0.5,0.5" Opacity="0.8" TileMode="FlipX">
          <VisualBrush.Visual>
            <StackPanel >
              <TextBlock Foreground="Gold">
                    水阔天长挥手时</TextBlock>
              <TextBlock Foreground="LightBlue">
                    待君相送竟迟迟</TextBlock>
              <TextBlock Foreground="LightGray">
                    一朝缘证三生石</TextBlock>
              <TextBlock Foreground="Pink">
                    如影随行总不离</TextBlock>
            </StackPanel>
          </VisualBrush.Visual>
        </VisualBrush>
        <VisualBrush x:Key="PoemBrushFlipY" Viewport=
            "0,0,0.5,0.5" Opacity="0.8" TileMode="FlipY">
          <VisualBrush.Visual>
            <StackPanel >
```

```
            <TextBlock Foreground="Gold">
                  水阔天长挥手时</TextBlock>
            <TextBlock Foreground="LightBlue">
                  待君相送竟迟迟</TextBlock>
            <TextBlock Foreground="LightGray">
                  一朝缘证三生石</TextBlock>
            <TextBlock Foreground="Pink">
                  如影随行总不离</TextBlock>
          </StackPanel>
        </VisualBrush.Visual>
      </VisualBrush>
      <VisualBrush x:Key="PoemBrushFlipXY" Viewport=
          "0,0,0.5,0.5" Opacity="0.8" TileMode="FlipXY">
        <VisualBrush.Visual>
          <StackPanel >
            <TextBlock Foreground="Gold">
              水阔天长挥手时</TextBlock>
            <TextBlock Foreground="LightBlue">
              待君相送竟迟迟</TextBlock>
            <TextBlock Foreground="LightGray">
              一朝缘证三生石</TextBlock>
            <TextBlock Foreground="Pink">
              如影随行总不离</TextBlock>
          </StackPanel>
        </VisualBrush.Visual>
      </VisualBrush>
    </Grid.Resources >
      <Grid.ColumnDefinitions>
      <ColumnDefinition/>
      <ColumnDefinition/>
    </Grid.ColumnDefinitions>
    <Grid.RowDefinitions>
      <RowDefinition/>
      <RowDefinition/>
    </Grid.RowDefinitions>
    <Button  Background="{StaticResource  PoemBrush }" >
      <TextBlock Foreground="Blue" FontSize="50">
            三生石</TextBlock>
    </Button>
    <Button  Background="{StaticResource PoemBrushFlipX}"
           Grid.Column="1">
      <TextBlock Foreground="Blue" FontSize="50">
            横向对称</TextBlock>
    </Button>
    <Button  Background="{StaticResource PoemBrushFlipY}"
           Grid.Row="1">
      <TextBlock Foreground="Blue" FontSize="50">
            纵向对称</TextBlock>
    </Button>
    <Button  Background="{StaticResource PoemBrushFlipXY}"
           Grid.Row="1" Grid.Column="1">
```

```
        <TextBlock Foreground="Blue" FontSize="50">
              原点对称</TextBlock>
      </Button>
    </Grid>
</Window>
```

笔者在这里创建了4个VisualBrush。这4个VisualBrush的内容都是4个TextBlock，VisualBrush的粘贴模式（TileMode）是不同的，分别是Tile、FlipX、FlipY和FlipXY。在Tile模式下，不需要演示粘贴效果，所以，把ViewPort设为（0，0，1，1），即在整个绘制区域只显示一个画刷。在其他模式下，把ViewPort设为（0，0，0.5，0.5），这时所要绘制的区域面积是ViewPoint的4倍，所以其他三个区域用了4次画刷，每个区域的效果是不同的。

然后，把这些画刷用作按键Button的背景（Background），这样每个按键的背景就是三生石这首诗了。如果你试着改变窗口的大小，就可以看到按键和后面的背景也自动改变大小，是不是有点“如影随行总不离”的味道？

在上面的程序里，笔者把画刷放在Grid.Resource中，有关使用资源的技术细节，见本书第8章资源。

使用VisualBrush程序的运行结果如图5-18所示。

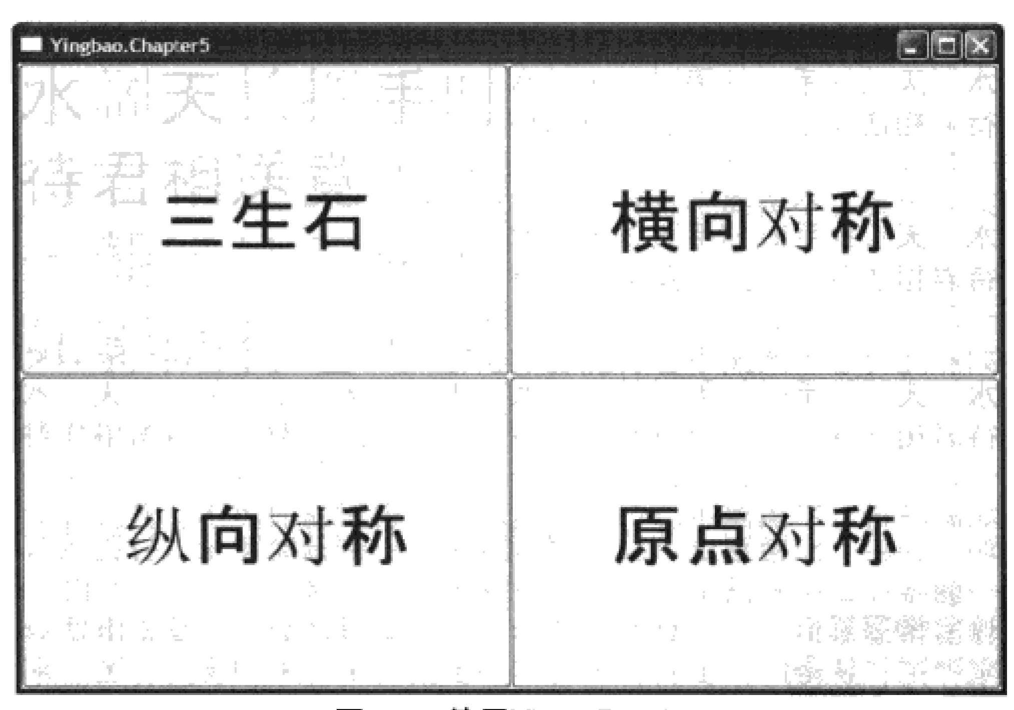


图5-18 使用VisualBrush

5.3 画笔

WPF中的画笔和画刷一样，使用的时候常常是“隐含”的，如图5-18中的按钮（Button）背景设为画刷。有的时候画笔的使用会隐含得更深一些，如在矩形Regtangle元素中，我们设置Stroke，和StrokeThickness，实际上设置的就是画笔。

和画刷一样，画笔也是从动画类Animatable中派生出来，这表明画笔具有动画和Freeze等画刷所具有的功能。图5-19示出了画笔类的继承图。

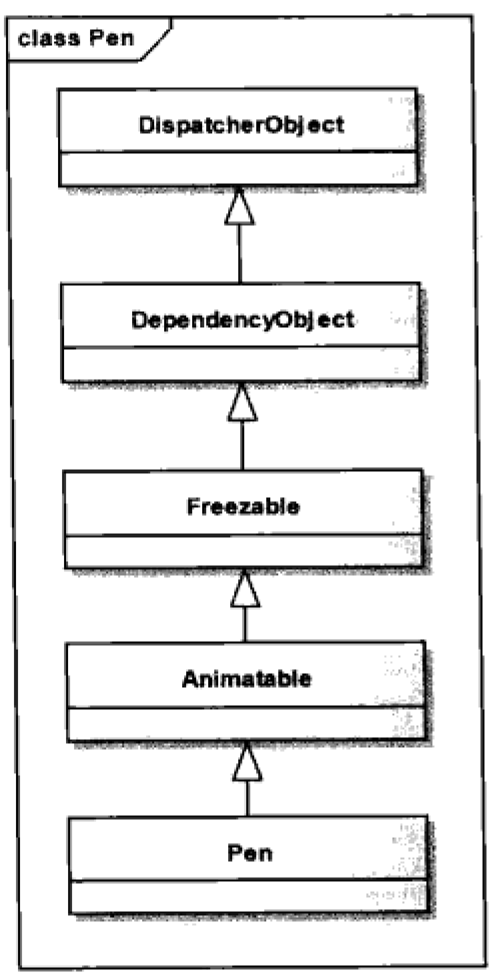


图5-19 画笔类继承图

有意思的是画刷是画笔的一个属性，这样前面所讨论的所有画刷都可以用到画笔中来，从而产生很有意思的效果。

表5-1示出了画笔的常用属性，Thichness、StartLineCap、DashStyle、DashCap、LineJoin和MitterLimit等属性是用来控制线形的。

表5-1 画笔的常用属性

画笔的属性名	功能描述
Brush	当用笔画图时所用的填充画刷
Thickness	笔的粗细，double类型
StartLineCap	规定启笔绘画时的线的形状
EndLineCap	规定终止绘画时的线的形状
DashStyle	规定虚线时的风格
DashCap	虚线的端头形状
LineJoin	规定两笔交接处的方式
MitterLimit	两条线接头长度对1/2笔宽的比率

笔者写了个测试程序，通过这个程序可以很容易地观察表5-1中各种属性的效果。

```
<Window x:Class="Yingbao.Chapter5.TestWPFPen"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Yingbao.Chapter5" Height="300" Width="600"    >
  <Grid>
    <Grid.RowDefinitions>
```

```
    <RowDefinition Height ="60"/>
    <RowDefinition Height ="60"/>
    <RowDefinition Height ="3*"/>
  </Grid.RowDefinitions>
  <Grid.ColumnDefinitions>
    <ColumnDefinition Width="*"/>
    <ColumnDefinition Width="*"/>
    <ColumnDefinition Width="*"/>
    <ColumnDefinition Width="*"/>
  </Grid.ColumnDefinitions>
  <Polyline Name ="myShape"
    Points="50,50 75,30 100,100 130,40"
    Stroke="Red"
    StrokeThickness="20"
    StrokeStartLineCap="Triangle"
    StrokeEndLineCap="Triangle"
    Grid.Row="2" Grid.Column="1" Grid.ColumnSpan ="3"/>
  <StackPanel Orientation ="Horizontal" Grid.Column ="0"
        Grid.Row ="0" Margin ="5,20,5,10" >
    <Label>颜色：</Label>
    <ComboBox  Width ="50" SelectionChanged ="OnColorChanged"
                SelectedIndex ="0">
      <ComboBoxItem>Red</ComboBoxItem>
      <ComboBoxItem>Blue</ComboBoxItem>
      <ComboBoxItem>Yellow</ComboBoxItem>
      <ComboBoxItem>Green</ComboBoxItem>
    </ComboBox>
  </StackPanel>
  <StackPanel Grid.Column ="1" Grid.Row ="0"
       Orientation ="Horizontal" Margin="5,20,5,10">
    <Label>线型：</Label>
    <TextBox LostFocus ="OnDashStyleChange" Width="80"  />
  </StackPanel>
  <StackPanel Grid.Column ="2" Grid.Row ="0"
       Orientation ="Horizontal" Margin  ="5,20,5,10">
    <Label>交接类型：</Label>
    <ComboBox  Width ="50"
     SelectionChanged ="OnLineJoinChanged" SelectedIndex ="0">
      <ComboBoxItem>Bevel</ComboBoxItem>
      <ComboBoxItem>Miter</ComboBoxItem>
      <ComboBoxItem>Round</ComboBoxItem>
    </ComboBox>
  </StackPanel>
  <StackPanel Grid.Column ="3" Grid.Row ="0"
              Orientation ="Horizontal" Margin="5,20,5,10">
    <Label>线宽：</Label>
    <TextBox  LostFocus ="OnLineThickNess" Width="80"
              Text ="20" />
  </StackPanel>
  <StackPanel Grid.Column ="0" Grid.Row ="1"
      Orientation ="Horizontal" Margin ="2,20,2,10">
    <Label>起始图形：</Label>
```

```
        <ComboBox  Width ="70" SelectionChanged
              ="OnStartLineCapChanged" SelectedIndex ="3">
          <ComboBoxItem>扁平</ComboBoxItem>
          <ComboBoxItem>圆形</ComboBoxItem>
          <ComboBoxItem>方形</ComboBoxItem>
          <ComboBoxItem>三角形</ComboBoxItem>
        </ComboBox>
      </StackPanel>
      <StackPanel Grid.Column ="1" Grid.Row ="1"
            Orientation ="Horizontal" Margin="2,20,2,10">
        <Label>终止图形：</Label>
        <ComboBox  Width ="70" SelectionChanged =
              "OnEndLineCapChanged"  SelectedIndex ="3">
          <ComboBoxItem>扁平</ComboBoxItem>
          <ComboBoxItem>圆形</ComboBoxItem>
          <ComboBoxItem>方形</ComboBoxItem>
          <ComboBoxItem>三角形</ComboBoxItem>
        </ComboBox>
      </StackPanel>
      <StackPanel Grid.Column ="2" Grid.Row ="1"
            Orientation ="Horizontal" Margin="5,20,5,10">
        <Label>交接长度：</Label>
        <TextBox  LostFocus ="OnMiterLimitChange" Width="60"
              Text ="1"/>
      </StackPanel>
    </Grid>
</Window>
```

相应的C#处理程序如下：

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Data;
using System.Windows.Documents;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Media.Imaging;
using System.Windows.Shapes;

namespace Yingbao.Chapter5
{
    public partial class TestWPFPen : System.Windows.Window
    {
        Int32 penColor;
        Pen lPen;
        public TestWPFPen()
        {
            penColor = 0xFF0000;
            lPen = new Pen();
```

```
        InitializeComponent();
        SetColor();
    }

    void OnColorChanged(object sender,
         SelectionChangedEventArgs e)
    {
       ComboBoxItem cbi =  ((sender as ComboBox).SelectedItem
                                as ComboBoxItem);
       string clrName= cbi.Content.ToString();
       penColor = GetColor(clrName);
       SetColor();
    }

    void SetColor()
    {
        byte a, r, g, b;
        a = 0xFF;
        r = (byte)(penColor >> 16);
        g = (byte)(penColor >> 8);
        b = (byte)(penColor);
        lPen.Brush = new SolidColorBrush(
                        Color.FromArgb(a,r,g,b) );
        if(myShape != null )
       myShape.Stroke =lPen.Brush;
    }

    void OnLineJoinChanged(object sender,
                     SelectionChangedEventArgs e)
    {
        ComboBoxItem cbi = ((sender as ComboBox).SelectedItem
                             as ComboBoxItem);
        string name = cbi.Content.ToString();

        lPen.LineJoin = (PenLineJoin)Enum.Parse(
                         typeof(PenLineJoin), name);
        myShape.StrokeLineJoin = lPen.LineJoin;
    }

    void OnDashStyleChange(object sender, EventArgs e)
    {
        string strValue = (sender as TextBox).Text;
        string[] styles =  strValue.Split(',');
        DashStyle ds = new DashStyle();
        for( int i =0; i <styles.Length; i++){
            ds.Dashes.Add( Convert.ToDouble( styles[i] ) );
        }
        lPen.DashStyle = ds;
        myShape.StrokeDashArray = lPen.DashStyle.Dashes;

    }

    void OnLineThickNess(object sender, EventArgs e)
```

```
{
    string strValue = (sender as TextBox).Text;
    lPen.Thickness = Convert.ToDouble(strValue);
    myShape.StrokeThickness = lPen.Thickness;
}

void OnEndLineCapChanged(object sender,
        SelectionChangedEventArgs e)
{
    ComboBoxItem cbi = ((sender as ComboBox).SelectedItem
                          as ComboBoxItem);
    string name = cbi.Content.ToString();
    lPen.EndLineCap = GetLineCap(name);
    myShape.StrokeEndLineCap = lPen.EndLineCap;
}

void OnStartLineCapChanged(object sender,
         SelectionChangedEventArgs e)
{
    ComboBoxItem cbi = ((sender as ComboBox).SelectedItem
                        as ComboBoxItem);
    string name = cbi.Content.ToString();
    lPen.StartLineCap  = GetLineCap(name);
    myShape.StrokeStartLineCap = lPen.StartLineCap;
}

void OnMiterLimitChange(object sender, EventArgs e)
{
    string strValue = (sender as TextBox).Text;
    lPen.MiterLimit  = Convert.ToDouble(strValue);
    myShape.StrokeMiterLimit = lPen.MiterLimit;
}

Int32 GetColor(string name)
{
    Int32 lColor = 0;
    switch (name)
    {
        case "Red":
            lColor= 0xFF0000;
            break;
        case "Yellow":
            lColor= 0xFFFF00;
            break;
        case "Blue":
            lColor= 0x0000FF;
            break;
        case "Green":
            return 0x00FF00;
            break;
        default:
            lColor= 0xFF0000;
            break;
```

```
            }
            return lColor;
        }

        PenLineCap GetLineCap(string strCap)
        {
            PenLineCap cap = PenLineCap.Flat;
            switch (strCap)
            {
                case  "扁平":
                    cap = PenLineCap.Flat;
                    break;
                case "圆形":
                    cap = PenLineCap.Round;
                    break;
                case  "方形":
                    cap = PenLineCap.Square;
                    break;
                case "三角形":
                    cap = PenLineCap.Triangle ;
                    break;
            }
            return cap;
        }
    }
}
```

笔者用PolyLine类实例myShape来画出一条之字形的线，使用PolyLine主要是便于观察画笔中StartLineCap、MitterLineCap，及两线交接处的属性LineJoint和MitterJoint对图形的影响。从程序中可以看出，myShape并没有一个画笔Pen的属性，而是直接定义了相应于画笔里的那些属性。笔者认为WPF这么做可能是基于简化XAML代码的角度考虑。PolyLine类中的Stroke相应于画笔中的画刷属性；PolyLine类中的StrokeThickness相应于画笔中的画笔粗细属性Thickness；PolyLine类中的StrokeStartLineCap 相应于画笔中的StartLineCap属性…… 所以说PolyLine类隐含地使用了画笔，前面对所有画笔类的讨论，都是有效的。运行调整画笔属性值程序的效果如图5-20所示。

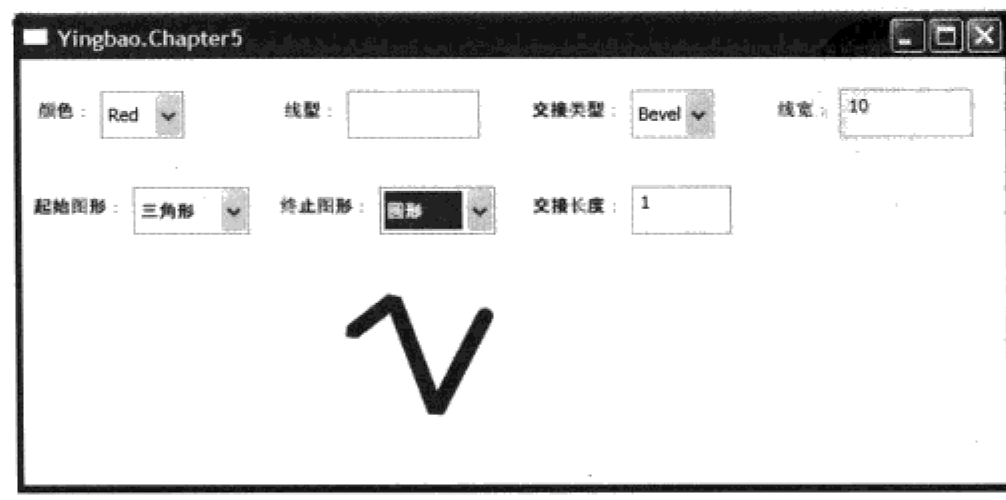


图5-20 调整画笔的属性值

还有一点需要注意，要测试画笔起始图形、终止图形，及两线交接处属性的影响，建议使用较粗

的笔（调整线宽到较大的值）。要测试线型，则要使用较细的笔（调整线宽到较小的值）。线性可以使用多个点画线，在线型的输入框里输入数字，中间用逗号隔开即可。

5.4 本章小结

本章详细地讨论了WPF对颜色的支持和扩展；实心画刷、梯度画刷（圆形梯度画刷和直线梯度画刷）、粘贴画刷（自制画刷、图像画刷和控件画刷），灵活使用这些画刷可以产生非常有趣的效果；本章最后讨论了画笔。

需要指出的是，WPF程序中有时在隐含地使用画笔和画刷，本章讨论的这些概念对于与画笔和画刷隐含的相关属性也是有效的。对于画笔和画刷，还有两大类的属性，转换和动画，将在本书第14章和第15章分别加以讨论。

第二篇　WPF专业程序员必备

第 6 章　WPF 控制

第 7 章　传递事件和传递命令系统

第 8 章　资源

第 9 章　风格

第 10 章　模板

第 11 章　数据绑定（Data Binding）

第 12 章　窗口对话框和打印

第6章 WPF控件

要说生活在北美和生活在中国内地有什么不同，其中之一就是装修。在中国内地，很少有人自己装修房子，原因是中国内地不仅人工便宜，而且一切都得自己从头干。在北美人工很贵，从事装修的工人的工资很高，所以很多装修活得自己做。比如我自己，就装修了一个地下室。自己做装修，想想可能很难，其实并不难。那些卖装修材料的商家，比如Home Depot和Rona等，备有专业人员提供装修咨询。而且装修也不必啥都自己做，商店里卖的大都是半成品。比如说隔一堵墙吧，你不需要自己一块砖一块砖地往上砌。商店里有标准尺寸的木料和标准化金属支撑材料，室内从地板到天花通常是八英尺或九英尺高，商店里的木料就提供八英尺或九英尺长的。可以很方便地用这些材料做一个架子，然后在这个架子上把房间里的电线布好。再去商店里买几块标准尺寸的石灰板，通常是4英尺乘8英尺大小。把石灰板用螺丝固定在架子上，再用商店里和好的石灰泥把石灰板间的缝隙填塞好，上上油漆就好了。

在WPF中构建人机界面，和在北美做装修一样，你也不必自己从头做。WPF提供了丰富的控件，.NET 1.0或Win32中的常用控件，如会话框、按钮、列表框、树形图等，WPF重写了一遍；WPF还提供了一些.NET 1.0中所没有的控件，如Frame、RepeatButton、Popup、GridSplitter等，所以，使用WPF比其他开发环境更方便。

笔者在前面几章已经用到了一些WPF控件，如Button、TextBox等，本章将系统介绍WPF控件。就像做装修一样，让我们从熟悉标准化装修材料开始吧。

6.1 WPF控件概述

WPF中有一个革命性的新概念，就是把控件的特性和控件的显示方式分开。控件在用户界面上的样子是由控件模板决定的，WPF为每个控件提供了默认的控件模板和相应的特性，但用户可以用自己的控件模板来替换WPF提供的控件模板，所以WPF中的每个控件都可以成为开发者自己个性化的控件。过去在Windows平台上开发的那种标准的控件样子（如 Visual Basic、MFC等），在WPF里开发者则可以有自己个性化的实现，比如过去方形的按钮，你可以换成圆的或椭圆的，或者其他任意的形状，甚至你自己的照片等。虽然用户改变了按钮的视觉效果，但并没有改变按钮的基本属性，比如当鼠标移过按钮，按钮边框会改变；当用户按下鼠标，按钮会产生单击事件等。有关控件模板，本书第9章会专门讨论写控件模板的方法。

很多人都知道，对于Win 16和Win 32应用程序，可以使用Visual Studio里的Spy程序来获取控件的信息，并跟踪该控件的运行（如显示该控件所发出的消息等）。这是一个很有用的工具，但该工具对WPF控件只能显示Windows句柄。幸运的是，WPF中类似于Spy工具的Spy++已经面世，可以在网站上免费下载（http: //blois.us/Snoop/）。

WPF的所有控件都是从System.Windows.Controls.Control类中派生出来，其命名空间是System.Windows.Controls。Control类中定义了一个重要的属性：Template，这个属性就是前面提到的控件模板。如果一个控件没有模板，那么这个控件在UI上是不可见的。Control中的一些相关属性，

如Background、Foreground等，只在有控件模板时才有意义。

WPF控件基类Control的类继承图如图6-1所示。由图6-1可见，所有的WPF控件都是UIElement（即是is-A），与UIElement相对应的另一个类是ContentElement。WPF有两个类似的类继承树，一个与界面（UI）相关，如UIElement类；另一个与内容（Content）相关，如ContentElement。FrameworkElement从UIElement中派生出来，与FrameworkElement相对的另一个类是FrameworkContentElement。需要注意的是ContentElement和下面要说的内容控件（Content Controls）不同，内容控件是一种控件，而ContentElement不是；ContentElement支持文本方式，而UIElement则支持图形方式。ContentElement类中没有OnRender方法，从ContentElement中派生出的类并不在屏幕上绘制自己，而是通过UIElement把自己展现出来。

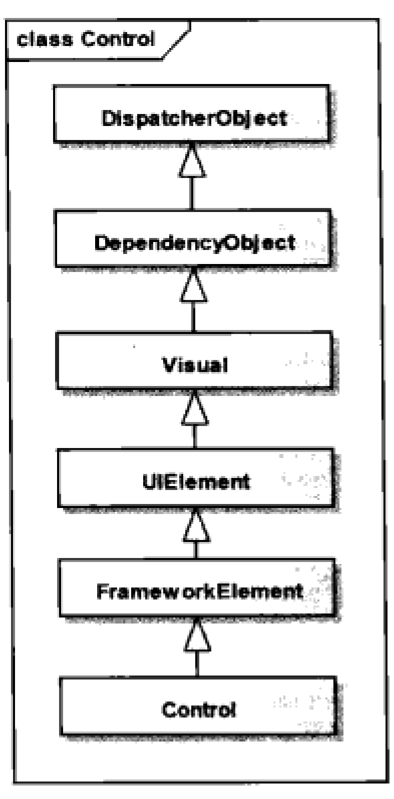


图6-1 WPF控件类继承图

那么什么是控件呢？笔者认为，控件要满足三个条件：一是它是相对独立的模块，这些模块可以通过一定的方式（通常是排版）像堆积木似的组合起来；二是它具有对用户输入反应的能力，用户输入经由Windows操作系统的消息机制而转换为控件事项；三是每个控件具有特定的图形表现形式，由于WPF把控件的特性和控件的显示方式分开，过去由显示方式区分控件的界限有时候不是很清晰的。

本章所要讨论的控件都是从Control类中派生出来的，基本上可以分为4类：

- 内容控件（Content Controls）
- 条目控件（Items Controls）
- 文本控件（Text Controls）
- 范围控件（Range Controls）

6.2 内容控件（Content Control）

内容控件是WPF控件中的一大类，ContentControl直接从Control类中派生出来。内容控件的最大特征是这类控件都含有一个Content属性，即它只能有一个直接的子控件，这个Content属性的类型为Object。众所周知.NET中所有的类都是从Object类中派生出来的，换句话说，内容控件可以是.NET的任何类。读者可能会有这样的问题：如果把一个内容控件设为没有用户界面的类，内容控件怎么在窗口上显示呢？在这种情况下，WPF会调用Object类的ToString方法，从而在控件中显示字符串；如果Content是UI元素，WPF会调用OnRender方法，从而让UI元素自己在控件中绘制出来。

ContentControl及其派生类如图6-2所示。Window类，前面已经多次用到，后面笔者还要专门讨论，UserControl要在第17章中加以讨论。

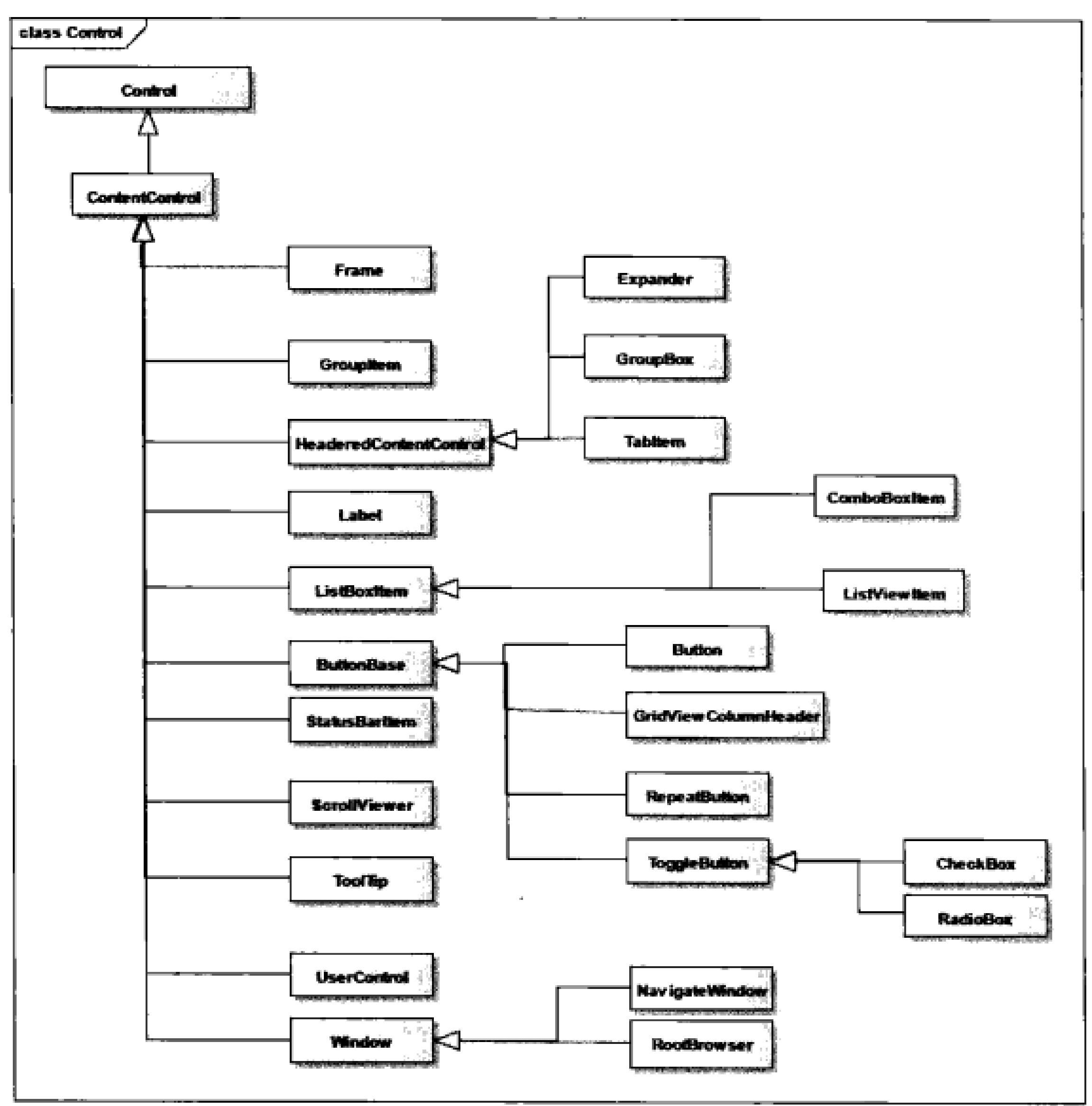


图6-2 内容控件

6.2.1 框架控件（Frame）

框架控件的主要作用是把其中的内容和界面上其他部分分开，它里面可以含有任何东西，但只能有一个直接的子控件。虽然Frame可以是其他控件的子控件，但是其宿主包容器的相关属性不会传递到Frame中，即起到隔离的作用。例如：

```
<Window x:Class="Yingbao.Chapter6.FrameDP"
xmlns=" http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:x= http://schemas.microsoft.com/winfx/2006/xaml
Title="Yingbao.Chapter6" Height="300" Width="300" FontSize ="20">
<StackPanel >
      <Button>在Frame的外面</Button>
      <Frame>
        <Frame.Content>
           <Button>在Frame里面</Button>
        </Frame.Content>
      </Frame>
    </StackPanel>
</Window>
```

在上面的XAML中，笔者把Window的字体大小FontSize属性设为20，用了两个同样的按钮控件Button。其中一个放在Frame控件的外面，另一个放在Frame控件的里面。图6-3示出了上述XAML的运行结果，由图6-3可见，位于Frame外面的按钮的字体大小为20，即Window控件的相关属性被传递到该按钮上；而位于Frame里面的按钮的字体大小仍然是默认值。由此可见Frame控件"阻断"了相关属性的传递。

图6-3 Frame对相关属性传递的影响

Frame这个控件是Win32及.NET2.0中所没有的，虽然Frame中可以包含任何.Net对象，但通常使用Frame控件的目的是要在其中嵌入HTML内容。在Frame中加入HTML非常简单，只要设定Source属性就可以了，例如下面的XAML显示搜狐网页：

```
<Window x:Class="Yingbao.Chapter6.UsingFrame"
xmlns= " http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
Title="Yingbao.Chapter6" Height="800" Width="300">
  <DockPanel DockPanel.Dock ="Top">
    <Frame Source="http://www.sohu.com" />
  </DockPanel>
</Window>
```

很多人都知道.NET中有一个WebBrowser控件，该控件在底层是使用Microsoft Web Browser Com组件，该组件具有强大的功能，笔者曾用该组件来显示PDF文件和Microsoft office软件（Work，Excel等）所产生的文档，但Frame却不具备打开这些文件的功能。与WebBrowserControl相比，Frame的长处在于可以显示XAML网页，而WebBrowser控件则不能。到目前为止，Microsoft还不能提供既有WebBrowser控件的功能、又能显示WPF控件的控件。

6.2.2 WPF按钮（Button）

WPF中的按钮都是从ButtonBase抽象类中派生出来的（如图6-2所示）。ButtonBase中定义了四个

属性，如表6-1所示。

表6-1 ButtonBase中的属性

属性名	功能描述
ClickMode	ClickMode为枚举类型，可取Hover、Press和Release三个值。设置这个属性指明按钮在什么情况下产生Click事件。若设为Press，则在按下鼠标左键时产生Click事件；若设为Hover，则在鼠标移到该按钮下时产生Click事件
IsFocused	若该值为true，则该按钮有输入焦点
IsMouseOver	若该值为true，则鼠标位于该按钮的上面
IsPressed	若该值为true，则鼠标在该按钮上按下

按钮中产生的事件是Click，WPF允许程序员设置这个事件在什么情况下产生，这是WPF的新特性。

以ButtonBase为基类的有四个类：Button、GridViewColumnHeader、RepeatButton和ToggleButton。其中GridViewColumnHeader控件，笔者将在讨论GridView控件时一并讨论；从ToggleButton中派生出两个著名的按钮控件：RadioButton和CheckBox。

按钮Button

按钮是Windows 用户界面程序中最常用的控件，在前面的章节中，已经多次用过。例如：

```
<Button Name="btn1" Background="Pink"    BorderBrush="Black"
BorderThickness="1" Click="OnClick1" ClickMode="Hover"> 按钮
</Button>
```

上面的XAML创建一个名为btn1的按钮，其背景色设为粉红色，边界设为黑色，当鼠标经过按钮时产生Click事件，并调用OnClick1事件处理程序。上面按钮的内容Content属性设为“按钮”字符串，也可以把该字符串改为其他形状，例如：

```
<Window x:Class="Yingbao.Chapter6.CircleButton"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
Title="Yingbao.Chapter6" Height="200" Width="200">
<StackPanel >
  <Button  Click ="OnClick" MouseEnter ="OnMouseEnter" MouseLeave
         ="OnMouseLeave">
   <StackPanel>
    <TextBlock FontSize ="16" Foreground ="Black">按钮</TextBlock>
<Ellipse Name ="btnEllipse"  Fill ="Cyan" Stroke ="Blue" Width
         ="100" Height ="50"/>
   </StackPanel>
  </Button>
  <Button IsDefault ="True" Click ="OnOK">确认</Button>
  <Button Click ="OnCancel">取消</Button>
  <TextBox Height ="50"></TextBox>
 </StackPanel>
</Window>
```

笔者在上面的这段程序中，创建了一个有点奇怪的按钮，它由一个TextBlock类和Ellipse类组成，为了使这个有点奇怪的按钮像一个按钮，需要处理三个相关事件：Click、MouseEnter和MouseLeave：

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Data;
using System.Windows.Documents;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Media.Imaging;
using System.Windows.Shapes;

namespace Yingbao.Chapter6
{
    public partial class CircleButton : System.Windows.Window
    {
        public CircleButton()
        {
            InitializeComponent();
        }
        void OnClick(object sender, RoutedEventArgs rea)
        {
           MessageBox.Show("你按了圆按钮！", "测试按钮",
                  MessageBoxButton.OK);
        }

        void OnMouseEnter(object sender, RoutedEventArgs rea)
        {
            btnEllipse.Fill = new SolidColorBrush(Colors.Cornsilk);
        }

        void OnMouseLeave(object sender, RoutedEventArgs rea)
        {
            btnEllipse.Fill = new SolidColorBrush(Colors.Cyan);
        }

        void OnOK(object sender, RoutedEventArgs rea)
        {
            MessageBox.Show("你按了确认按钮！", "测试按钮",
            MessageBoxButton.OK);
        }
        void OnCancel(object sender, RoutedEventArgs rea)
        {
            MessageBox.Show("你按了取消按钮！", "测试按钮",
            MessageBoxButton.OK);
        }
    }
}
```

在OnMouseEnter事件处理程序里，笔者把椭圆的填充色设为Cornsilk，这样，当把鼠标移到该按钮上，则所看到按钮的样子就不一样了。在OnMouseLeave事件处理程序里，笔者把椭圆的填充色设回原来的Cyan。在处理OnClick事件时，笔者简单地显示一个消息框："你按了圆按钮！"。而在

Win32会话框程序里，通常可以指定某个按钮为默认按钮，当用户按下键盘上"Enter"键时，触发该按钮的事件处理程序。在WPF里，你要做的是把Button中的IsDefault属性设为"True"，如上例中的"确认"按钮。需要注意的是，若一个窗口中有多个按钮，你只需把其中的一个按钮的IsDefault属性设为True即可。上述按钮实例的运行效果如图6-4所示。

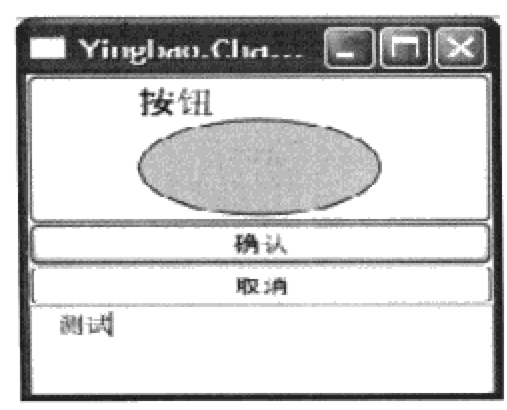


图6-4 按钮实例

6.2.3 拨动按钮（ToggleButton）

拨动按钮的特性就像拨动开关一样，把开关拨到一个位置，灯亮了；把开关拨到另一个位置，灯灭了。拨动按钮也是这样，按一下拨动按钮，其IsChecked属性变为True；再按一下拨动按钮，其IsChecked属性变为False。有时，我们需要拨动按钮维持三个状态，这时我们把IsThreeeState属性设为True。在IsThreeeState属性设为True时，IsChecked的属性值可能取三个值：True, False和Null，ToggleButton中的事件如表6-2所示。

表6-2 ToggleButton中的事件

事件名	发生条件
Checked	当IsChecked为True时，产生该事件。
UnChecked	当IsChecked为false时，产生该事件。
Indeterminate	当IsChecked为Null时，产生该事件。

一般并不直接创建ToggleButton实例，而是使用其派生类：CheckBox和RadioButton。

6.2.4 CheckBox控件

CheckBox控件用在可以有多个选项，而又可以同时选择两个或两个以上选项的情况下。比如说，某个专业一学期开了八门课，其中有两门是必修课，其余都是选修课，那么这个时候就可以把这六门选修课用CheckBox表现出来，供同学们选择。

CheckBox并没有增加任何属性，它为ToggleButton提供外观。

6.2.5 RadioButton控件

若在一组选项里面，你必须而且只能选择一个选项，需要使用RadioButton。比如，要调查某一类消费者使用你的产品，就需要知道消费者的性别、年龄组等，这时RadioButton就是最好的选择。因为一个人要么是男的，要么是女的。

就像前面所说的"你必须选择一个"，RadioButton并不提供"不选"操作，要去掉某个选项的方法是选择另一个选项。所以，需要把相关选项放到一个组里。WPF的RadioButton增加了一个属

性：GroupName，其类型为字符串string。例如：

```
<StackPanel>
    <RadioButton GroupName="gender">男</RadioButton>
    <RadioButton GroupName="gender">女</RadioButton>
    <RadioButton GroupName="agegrp">0-20</RadioButton>
<RadioButton GroupName="agegrp">21-40</RadioButton>
<RadioButton GroupName="agegrp">41-60</RadioButton>
<RadioButton GroupName="agegrp">>60</RadioButton>
</StackPanel>
```

男和女在“gender”组中；年龄分为0~20，21~40，41~60和60以上4个年龄段，在“agegrp”组中，任何一个人都必须而且只能在每个组中选一个。当可选择的选项很多时，虽然你也必须而且只能选一个，但一般不用RadioButton了，原因在于RadioButton在视窗上很占地方。这个时候，需要使用组合框ComboBox。有关组合框，本章后面会详细讨论。

6.2.6 重复按钮（RepeatButton）

重复按钮是WPF新增的一个控件，和一般的按钮不同，重复按钮可以在按下鼠标左键时，不停地发出单击Click事件。这种特性在某些情况下是很有用的，比如DVD机上的快进或快退键。在按下快进键的时候，DVD中的内容会快速地播放出来，还可以选择以2倍、4倍、8倍或更高速率播放，从而更快地找到所要找的内容。

重复按钮实现了这种功能，它增加了两个属性：Delay和Interval。

- Delay这个属性设置从用户按下鼠标左键到开始重复发送Click事件的延迟时间。单位为毫秒（ms），Delay的数值必须大于等于0。
- Interval这个属性设置当用户按下鼠标左键时，每次发出Click事件的时间间隔。单位为毫秒（ms），Interval的数值必须大于等于0。

下面的例子用了两个重复按钮，可以看到，当在“增加”按钮上按住鼠标不动时，TextBlock中的数值在不断增加；当在“减少”按钮上按住鼠标不动时，TextBlock中的数值在不断减少，即RepeatButton不断发出Click事件。

```
<Window x:Class="Yingbao.Chapter6.UseRepeatButton"
   xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
   xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
   Title="Yingbao.Chapter6" Height="100" Width="300">
  <DockPanel >
<RepeatButton Width="100" DockPanel.Dock="Top" Delay="500"
  Interval="100" Click="Increase" Content =" 增加"/>
    <TextBlock Name="valueText"       Width="100"
       DockPanel.Dock="Top" TextAlignment="Center" FontSize="16" >
           0  </TextBlock>
      <RepeatButton Width="100" DockPanel.Dock="Top"
       Delay="500" Interval="100" Click="Decrease" Content ="减少"/>
  </DockPanel>
</Window>
```

这种不断增加或减少的功能是在下列C#的事件处理程序中实现的，重复按钮程序运行的效果如图6-5所示。

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Data;
using System.Windows.Documents;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Media.Imaging;
using System.Windows.Shapes;

namespace Yingbao.Chapter6
{

    public partial class UseRepeatButton : System.Windows.Window
    {

        public UseRepeatButton()
        {
            InitializeComponent();
        }

        void Increase(object sender, RoutedEventArgs e)
        {
            Int32 Num = Convert.ToInt32(valueText.Text);

            valueText.Text = ((Num + 1).ToString());
        }

        void Decrease(object sender, RoutedEventArgs e)
        {
            Int32 Num = Convert.ToInt32(valueText.Text);

            valueText.Text = ((Num - 1).ToString());
        }
    }
}
```

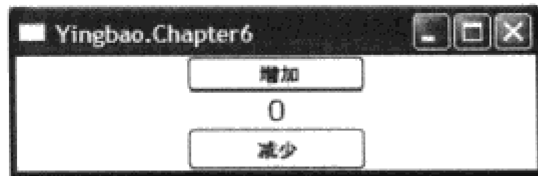


图6-5 重复按钮

6.2.7 带有标题栏的内容控件（HeaderedContentControl）

由图6-2可见，从HeaderedContentControl中派生出来三个类，即Expander、GroupBox和TabItem。TabItem控件和TabControl一起使用，笔者留到6.3.5节TabControl讨论。Expander是WPF引进的一个新

的控件，很多网页上都有这种控件，过去WinForm里没有这个控件，一些为.Net提供控件的第三方软件开发商，如DevExpress、Nevron等为WinForm提供了这一控件。

带有标题的内容控件（HeaderedContentControl）除了具有内容控件的Content属性之外，加入了一个新的标题属性（Header ）。Header的类型和Content一样，都是Object，换句话说，任何.NET对象都可以成为标题Header。当然最常用的标题是字符串，但也可以是任何其他控件或控件的组合。请看下面的例子：

```
<Window x:Class="Yingbao.Chapter6.usingHeaderedContentControl
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Yingbao.Chapter6" Height="300" Width="500">
  <HeaderedContentControl BorderThickness ="2" BorderBrush
="Chocolate" Background ="Cyan">
    <HeaderedContentControl.Header>
      <Label FontSize ="20" FontStyle ="Italic" FontWeight ="Bold">
                 第6章
      </Label>
    </HeaderedContentControl.Header>
    <HeaderedContentControl.Content>
      <StackPanel>
      <TextBox Height="140" Width="450"  >
        带有标题的内容控件（HeaderedContentControl）除了具有内容控件……
      </TextBox>
      </StackPanel>
    </HeaderedContentControl.Content>
  </HeaderedContentControl>
</Window>
```

在这段程序中，笔者把标题内容控件HeaderedContentControl的标题属性设为Label控件；把其中的内容属性设为StackPanel，其中含有一个TextBox控件，当然也可以把TextBox控件直接作为HeaderedConentControl的Content属性，其结果是一样的。

该程序的运行结果如图6-6所示：

图6-6 带有标题的内容控件

从上面的程序及其运行的结果，可以看到：若不用标题内容控件HeaderedContentControl，也可以很容易用其他控件的组合来达到同样的效果。确实，通常使用的不是HeaderedContentControl控件本身，而是它的两个派生类：分组框和伸展控件。

6.2.8 分组框（GroupBox）

分组框（GroupBox）是一个常用的控件，在Win32里就有这样的控件。其用法是把一些意义相近

的控件放在一起，同时用一个带标题的边框把它们组合起来。例如：

```
<Window x:Class="Yingbao.Chapter6.UsingGroupBox"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
   Title="Yingbao.Chapter6" Height="160" Width="300" FontSize ="12">
  <Grid>
    <Grid.ColumnDefinitions>
      <ColumnDefinition />
      <ColumnDefinition />
    </Grid.ColumnDefinitions>
<GroupBox  Width ="100" Height ="100" Header ="年龄组"
                Grid.Column ="0" Grid.Row ="0">
      <GroupBox.Content>
        <StackPanel>
          <RadioButton GroupName="agegrp" Margin="2">0-20
                        </RadioButton>
          <RadioButton GroupName="agegrp" Margin="2">21-40
                        </RadioButton>
          <RadioButton GroupName="agegrp" Margin="2">41-60
              </RadioButton>
          <RadioButton GroupName="agegrp" Margin="2">>60
                        </RadioButton>
        </StackPanel>
      </GroupBox.Content>
    </GroupBox>
<GroupBox  Width ="100" Height ="120" Header ="选修课"
              Grid.Column ="1" Grid.Row ="0">
      <GroupBox.Content>
        <StackPanel>
          <CheckBox  Margin="2">XML语言（1）</CheckBox>
          <CheckBox  Margin="2"> XSLT初步</CheckBox>
          <CheckBox  Margin="2">Java编程</CheckBox>
          <CheckBox  Margin="2">C++编程</CheckBox>
          <CheckBox  Margin="2">C#编程</CheckBox>
        </StackPanel>
      </GroupBox.Content>
    </GroupBox>
  </Grid>
</Window>
```

上面的这段程序，笔者创建了两个GroupBox。一个是年龄组，其中是一组RadioButton；另一个是选修课，其中是一组CheckBox。由于一个人只能取上述年龄组中的一个，所以用的是RadioButton；而选修课则可以根据各人的情况自由安排，是可以同时选择多个的，所以笔者用的是CheckBox。

图6-7是上述XAML的运行结果，从图6-7中，可以清楚地看到，分组框控件有一个矩形框，内容控件在矩形框内，其标题在矩形的上方。

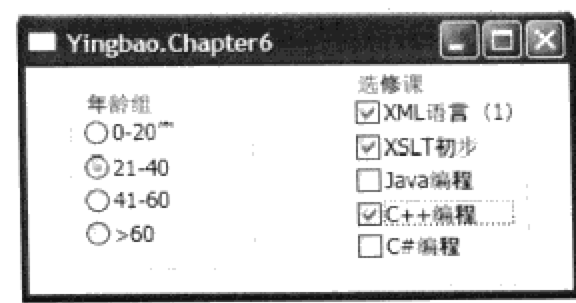


图6-7 分组框的应用

6.2.9 伸展控件（Expander）

伸展控件是Win 32中没有的一个控件，它在压缩时，只显示标题；在展开时才显示其中的内容。所以，使用这个控件可以在很小的版面上显示更多的内容。现在很多网页已经使用了这种控件，WPF对这一控件进行了标准化。

下面把前例中的GroupBox改为Expander：

```
<Window x:Class="Yingbao.Chapter6.usingExpander"
   xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Yingbao.Chapter6" Height="150" Width="300">
  <Grid>
    <Grid.ColumnDefinitions>
      <ColumnDefinition />
      <ColumnDefinition />
    </Grid.ColumnDefinitions>
<Expander  Width ="100" Height ="120" Header ="年龄组"
            Grid.Column ="0" Grid.Row ="0">
      <Expander.Content>
        <StackPanel>
          <RadioButton GroupName="agegrp" Margin="2">0-20
                     </RadioButton>
          <RadioButton GroupName="agegrp" Margin="2">21-40
                     </RadioButton>
          <RadioButton GroupName="agegrp" Margin="2">41-60
                     </RadioButton>
          <RadioButton GroupName="agegrp" Margin="2">>60
                     </RadioButton>
        </StackPanel>
      </Expander.Content>
    </Expander>
<Expander  Width ="100" Height ="120" Header ="选修课"
              Grid.Column ="1" Grid.Row ="0">
      <Expander.Content>
        <StackPanel>
          <CheckBox  Margin="2">XML语言（1）</CheckBox>
          <CheckBox  Margin="2"> XSLT初步</CheckBox>
          <CheckBox  Margin="2">Java编程</CheckBox>
          <CheckBox  Margin="2">C++编程</CheckBox>
          <CheckBox  Margin="2">C#编程</CheckBox>
        </StackPanel>
```

```
        </Expander.Content>
      </Expander>
  </Grid>
  </Window>
```

运行这段程序得到如图6-8所示的结果，当你用鼠标单击标题栏时，伸展控件在图6-8和图6-9之间切换。

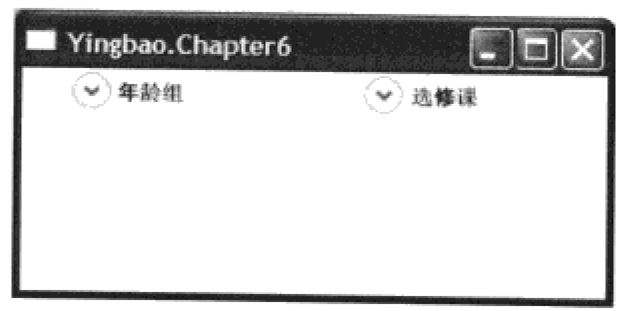


图6-8 伸展控件在压缩时只显示标题

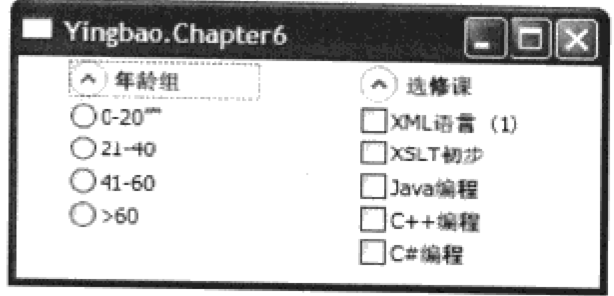


图6-9 伸展控件在展开时，显示标题和内容

伸展控件增加了两个相关属性，一个是ExpandDirection，其类型为枚举类型，可取Up、Down、Left、Right四个值；另一个是IsExpanded，其类型为布尔值，当该值为True时表明伸展控件处在展开状态。

6.2.10 标签控件（Label）

标签控件也是一个常用的控件，这个控件就是在窗口的某个地方显示字符串，这里的某个地方通常是由控制版面控件安排的。

标签控件是一个内容控件，与所有的内容控件一样，可以把它设为任何WPF控件及其组合，但我们最常用的标签控件的功能是显示字符串。在WPF里，Text-Block和TextBox等都可以用来显示字符串。标签控件区别于TextBlock和TextBox的功能主要有两点：

1. 在默认的情况下，标签控件是只读的，颜色是灰色的；
2. 标签控件内置了一个AccessText控件，可以直接设定热键到目标控件。

下面的例子笔者用标签(Label)控件来显示邮政编码，使用下画线字母（_P）把邮政编码的热键设置为P（Post Code），使用Target属性来设置热键按下时的输入焦点。这里笔者用了一个简单的数据绑定:{Binding ElelmentName=PostalCode},有关数据绑定的内容，将在第11章数据绑定中详细讨论。当程序运行时，若你同时按下“Alt”和“P”键，输入焦点会自动移到TextBox控件上。该程序的运行效果如图6-10所示。

```
<Window x:Class="Yingbao.Chapter6.UsingLabel"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
     xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
     Title="Yingbao.Chapter6" Height="80" Width="300">
  <StackPanel Orientation ="Horizontal ">
<Label Height ="30" Target ="{Binding ElementName=PostalCode}">邮
      政编码 (_P)</Label>
     <TextBox  Height="30" Width ="80" x:Name ="PostalCode"/>
  </StackPanel>
</Window>
```

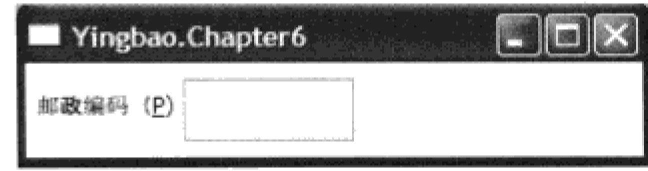


图6-10 带有设置热键功能的标签控件

笔者原来以为标签控件是一个最简单的控件，当用前面提到的snoop工具来检查Label控件时，这个控件竟然由Border、ContentPresenter、AccessText和TextBlock等几个控件组成（如图6-11所示）。原来标签（Label）控件具有设置热键功能的奥秘是其中包含了一个AccessText控件。AccessText虽然位于Control命名空间里，但严格地来说，它不是一个控件，因为它不是从FrameworkElement中派生出来的，它只能算作UI元素。

6.2.11 为按钮设置热键

用Snoop工具检查例子中标签控件的情况如图6-11所示。既然是AccessText使标签控件具有热键功能，那么，是否也可以用它来为其他控件设置热键呢？答案是肯定的。

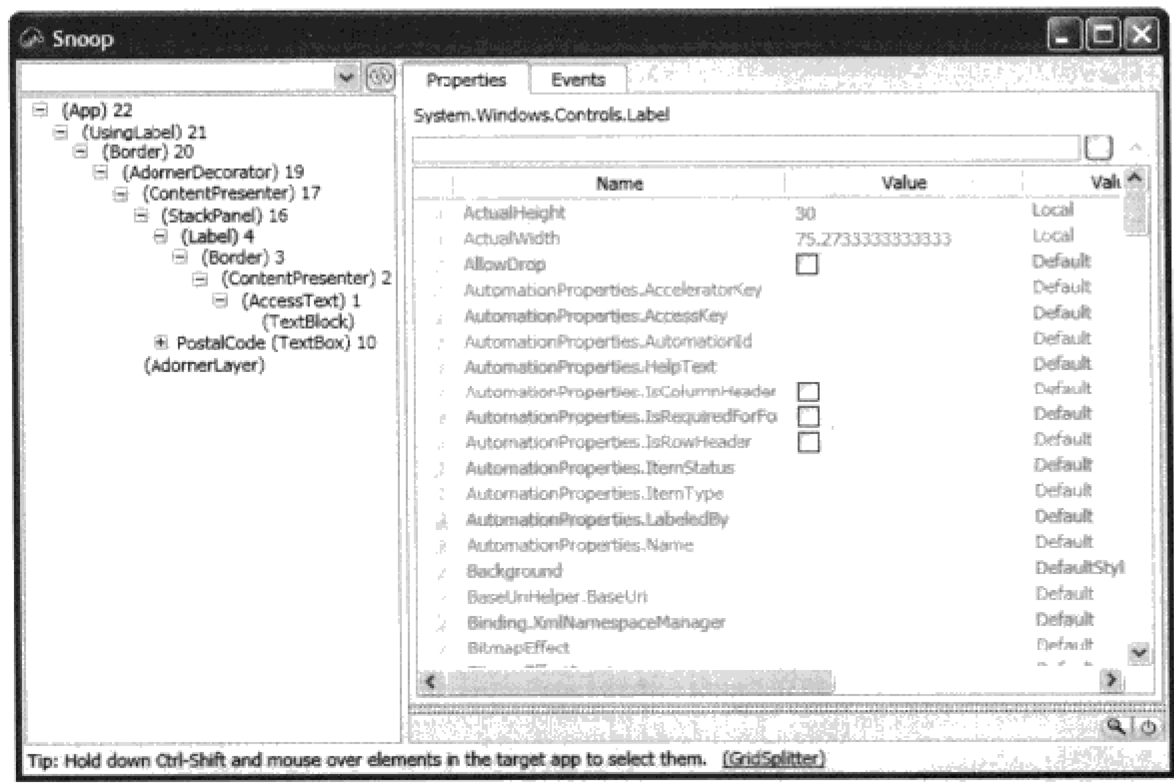


图6-11 用Snoop工具检查例子中的标签控件

下面的例子为按钮设置了热键：

```
<Window x:Class="Yingbao.Chapter6.SetHotKeyForButton"
   xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Yingbao.Chapter6" Height="80" Width="300">
  <StackPanel Orientation ="Horizontal " >
    <Button Height ="30" Width ="100" Background ="LightBlue"
     Click="OnClickHotKey">
      <AccessText >我有热键P(_P)</AccessText>
    </Button>
    <Button Height ="30" Width ="100" Background ="LightYellow"
```

```
     Click="OnClickNoHotKey">
          我没有热键
    </Button>
  </StackPanel>
</Window>
```

还可以在C#中捕获Click事件，从而考察热键的工作情况：

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Data;
using System.Windows.Documents;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Media.Imaging;
using System.Windows.Shapes;

namespace Yingbao.Chapter6
{
    public partial class SetHotKeyForButton : System.Windows.Window
    {

        public SetHotKeyForButton()
        {
            InitializeComponent();
        }

       void OnClickHotKey(object sender, RoutedEventArgs arg)
        {
            MessageBox.Show("你按下了有热键的按钮", "使用热键",
                MessageBoxButton.OK);
        }

       void OnClickNoHotKey(object sender, RoutedEventArgs arg)
        {
            MessageBox.Show("你按下了没有热键的按钮", "使用热键",
                MessageBoxButton.OK);
        }
    }
}
```

使用AccessText设置热键非常简单，如上例中：

<AccessText >我有热键P(_P)</AccessText>

即在字母前加一下画线即可。在使用中文时，一般不会遇到字母前用到下画线的情况，但在英文里会遇到字符串本身有下画线，而不是热键的情况。这时，我们需要加两个下画线：

<AccessText>This __is Really cool!</AccessText>

6.2.12 ToolTip

在图形界面程序中使用ToolTip已经有很长一段时间了，ToolTip通常是程序员在某些用户界面元素上加入的帮助信息，过去常用在工具条上。例如，在Microsoft Word软件中，当把鼠标移到工具条的某个按钮上时，在按钮的下面会显示一个矩形框，矩形框里显示该按钮的功能提示信息。当鼠标离开该按钮时，这个矩形框会自然消失，这就是我们常说的ToolTip。过去ToolTip控件相对简单，通常只能显示简单的文字信息，MFC里的CtoolTipCtrl就是这样的控件。而WPF中的ToolTip控件功能则强大灵活得多：首先，ToolTip属性是定义在FrameworkElement类和FrameworkContent中的，这意味着不仅WPF中的所有控件都可以使用ToolTip，而且图形和文档也可以有ToolTip。其次，ToolTip和其他WPF内容控件一样，其中可以含有其他任何WPF控件及其组合。

下面的XAML是使用ToolTip的例子：

```
<Window x:Class="Yingbao.Chapter6.UsingToolTip"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Yingbao.Chapter6" Height="300" Width="600">
  <StackPanel>
    <Button ToolTip="按此按钮会确认输入的信息"
         Height="20" Width="50">确认</Button>
    <Ellipse Stroke ="Blue" Fill ="DimGray" ToolTip ="这是一个椭圆"
Width ="100" Height ="80"/>
    <Label>出生国:</Label>
    <TextBox>
        <TextBox.ToolTip>
          <StackPanel>
            <Label Height ="30">你可以输入下属国家名: </Label>
            <ListBox Width ="200" Height ="100">
              <ListBoxItem>中国</ListBoxItem>
              <ListBoxItem>美国</ListBoxItem>
              <ListBoxItem>加拿大</ListBoxItem>
              <ListBoxItem>俄罗斯</ListBoxItem>
            </ListBox>
          </StackPanel>
        </TextBox.ToolTip>
    </TextBox>
  </StackPanel>
</Window>
```

在按钮控件Button和图形元素椭圆Ellipse中，笔者直接把字符串赋予ToopTip属性，而TextBox中的ToolTip则是由StackPanel等多种元素构成的。

图6-12是上述XAML程序的运行结果，当我们把鼠标移到图中的控件上时，就会显示相应的ToolTip信息。

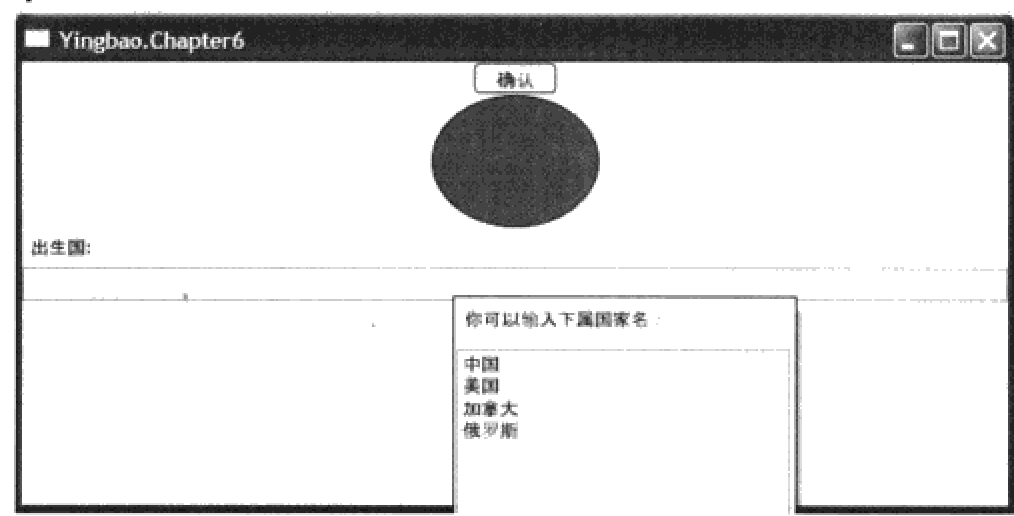


图6-12 WPF中的ToolTip

使用ToolTip需要注意下述两点：

1. ToolTip虽然是一个控件，但其本身不具备接收用户输入的功能。比如上面的例子中，在ToolTip中使用了ListBox，你就无法像通常ListBox那样选择其中的条目。

2. ToolTip虽然是一个控件，但它不能像其他控件那样作为其他控件的子控件，它必须和某个控件的ToolTip属性相连。

当鼠标离开控件时，其ToolTip会自动隐藏。有时候，你希望控制ToolTip的显示时间，ToolTipService类就是为此设计的，该类含有控制ToolTip的附加属性，如控制ToolTip水平或竖直方向位移的属性HorizontalOffset、VerticalOffset；控制ToolTip显示时间的ShowDuration属性；控制是否在未使能的控件上显示ToolTip的ShowOnDisabled属性等。

```
<Window x:Class="Yingbao.Chapter6.UsingToolTipService"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Yingbao.Chapter6" Height="200" Width="600">
  <StackPanel  ToolTipService.InitialShowDelay="1000"
           ToolTipService.ShowDuration="7000"
           ToolTipService.BetweenShowDelay="2000"
           ToolTipService.Placement="Right"
           ToolTipService.PlacementRectangle="50,0,0,0"
           ToolTipService.HorizontalOffset="10"
           ToolTipService.VerticalOffset="20"
           ToolTipService.HasDropShadow="false"
           ToolTipService.ShowOnDisabled="true"
           ToolTipService.IsEnabled="true">
  <Button>带ToolTip按钮
  <Button.ToolTip>
   <ToolTip>
    <BulletDecorator>
      <BulletDecorator.Bullet>
        <Ellipse Height="10" Width="20" Fill="Blue"/>
      </BulletDecorator.Bullet>
      <TextBlock>用了TooltipService类</TextBlock>
    </BulletDecorator>
  </ToolTip>
```

```
  </Button.ToolTip>
</Button>
  </StackPanel>
</Window>
```

需要指出的是，使用ToolTipService主要是为了保持多个控件的ToolTip属性的一致性，若只使用一个控件，则可以直接使用ToolTip中的相关属性。

6.2.13 ScrollViewer

当窗口中的内容比窗口的大小要大时，通常要在窗口中显示滚动条。在WPF中显示滚动条要用到ScrollViewer类，这是一个内容控件，和所有的内容控件一样，它只能含有一个子控件。

下面笔者用WPF的ScrollViewer来显示中国台湾地区作家三毛的一首诗——《七点钟》，这首诗由李宗盛先生谱曲，曾经在台湾很流行。据说在三毛自杀时，三毛迷们在大街小巷齐声唱这首歌，每每唱到“是我是我是我”时就放声大哭。不知怎的，我在北美工作十余年了，却也总有三毛的那么一种误把他乡作故乡的流浪的感觉。

```
<Window x:Class="Yingbao.Chapter6.ContentWithScrollViewer"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Yingbao.Chapter6" Height="200" Width="300">
    <ScrollViewer HorizontalScrollBarVisibility ="Auto"
    VerticalScrollBarVisibility ="Auto">
      <StackPanel>
        <TextBlock Width ="400" Height ="800"  FontSize ="12"
    TextWrapping ="NoWrap">
          <Bold FontSize ="20">七点钟</Bold>
          <LineBreak/>    <LineBreak/>
          <Italic> 三毛</Italic><LineBreak/>    <LineBreak/>
          今生就是那么地开始的<LineBreak/>
          走过操场的青草地  走到你的面前<LineBreak/>
          不能说一句话<LineBreak/>
          拿起钢笔  在你的掌心写下七个数字<LineBreak/>
          点一个头  然后<LineBreak/>
          狂奔而去<LineBreak/>
          <LineBreak/>    <LineBreak/>
          守住电话  就守住度日如年的狂盼<LineBreak/>
          铃声响的时候  自己的声音那么急迫    <LineBreak/>
          是我，是我，是我——是我是我是我    <LineBreak/>
          七点钟，你说七点钟？    <LineBreak/>
          好，好，好，我一定早点到    <LineBreak/>
          <LineBreak/>    <LineBreak/>
          啊明明站在你的面前  还是    <LineBreak/>
          害怕这是一场梦    <LineBreak/>
          <LineBreak/>    <LineBreak/>
          是真  是幻  是梦  是真是幻是梦    <LineBreak/>
          车厢里面对面坐着    <LineBreak/>
          你的眼底    <LineBreak/>
```

```
            一个惊慌少女的倒影    <LineBreak/>
            <LineBreak/>   <LineBreak/>
            火车一直往前去呀    <LineBreak/>
            我不愿意下车    <LineBreak/>
            <LineBreak/>   <LineBreak/>
            不管它要带我到什么地方    <LineBreak/>
            我的车站    <LineBreak/>
            就在你的身旁    <LineBreak/>
            是我    <LineBreak/>
            在你的身旁         <LineBreak/>
         </TextBlock>
      </StackPanel>
   </ScrollViewer>
</Window>
```

笔者把ScrollViewer类中的HorizontalScrollBarVisibility 和VerticalScrollBarVisibility两个属性设为"Auto"。为了显示滚动条，把窗口的大小设为300×200，而把ScrollViewer中所含的TextBlock大小设为400×800，这样当上面的程序运行时，就会自动显示滚动条了，用ScrollViewer滚动内容的情况如图6-13所示。

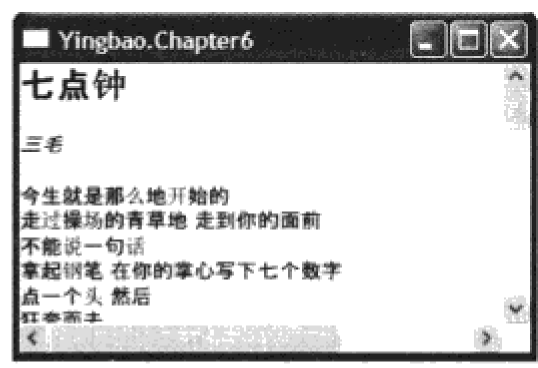


图6-13 用ScrollViewer滚动内容

笔者在上面的程序中使用Bold,Italic控制字符输出，使用LineBreak控制句子的分行。

6.3 条目控件（Items Controls）

WPF控件的另一大类就是ItemsControl，它是直接从Control类中派生出来的。与内容控件中只能含有一个控件不同，ItemsControl中含有Items属性，这个属性具有ItemCollection类型。在内容控件中，其内容可以是任何一个Object类型的对象；在Items属性中则可以加入任何一个Object类型的对象。WPF在显示Items属性中的对象时，如果这个对象是UI元素，就直接调用UI元素的OnRender方法；如果这个对象不是UI元素，WPF会创建一个TextBlock，并在TextBox中显示该对象的ToString方法所返回的结果（由于在.NET中所有的类都从Object类中派生出来，而Object类中含有ToString方法，所以，调用ToString方法是不会有什么问题的）。

除了可以加入任何.NET对象的Items属性之外，ItemsControl类中还有一个重要的属性ItemsSource。这个属性是用来作数据绑定的，我将在第11章详细讨论这个属性。需要注意的是，一旦我们使用了ItemsSource属性，Items属性就会被自动设置为null。这表明只能使用ItemsSource和Items两个属性当中的一个，而且ItemsSource具有更高的优先级。

图6-14示出了ItemsControl及其派生类的结构图。从对象组件的角度，基本上可以把这些类分成两大类，一类是可以放入条目的条目包容器；另一类是条目本身。条目又可以分为带有标题栏的条目和不带有标题栏的条目。有些条目类是内容控件，如ComboBoxItem、TabItem等，在内容控件中提到过（如图6-2所示）。

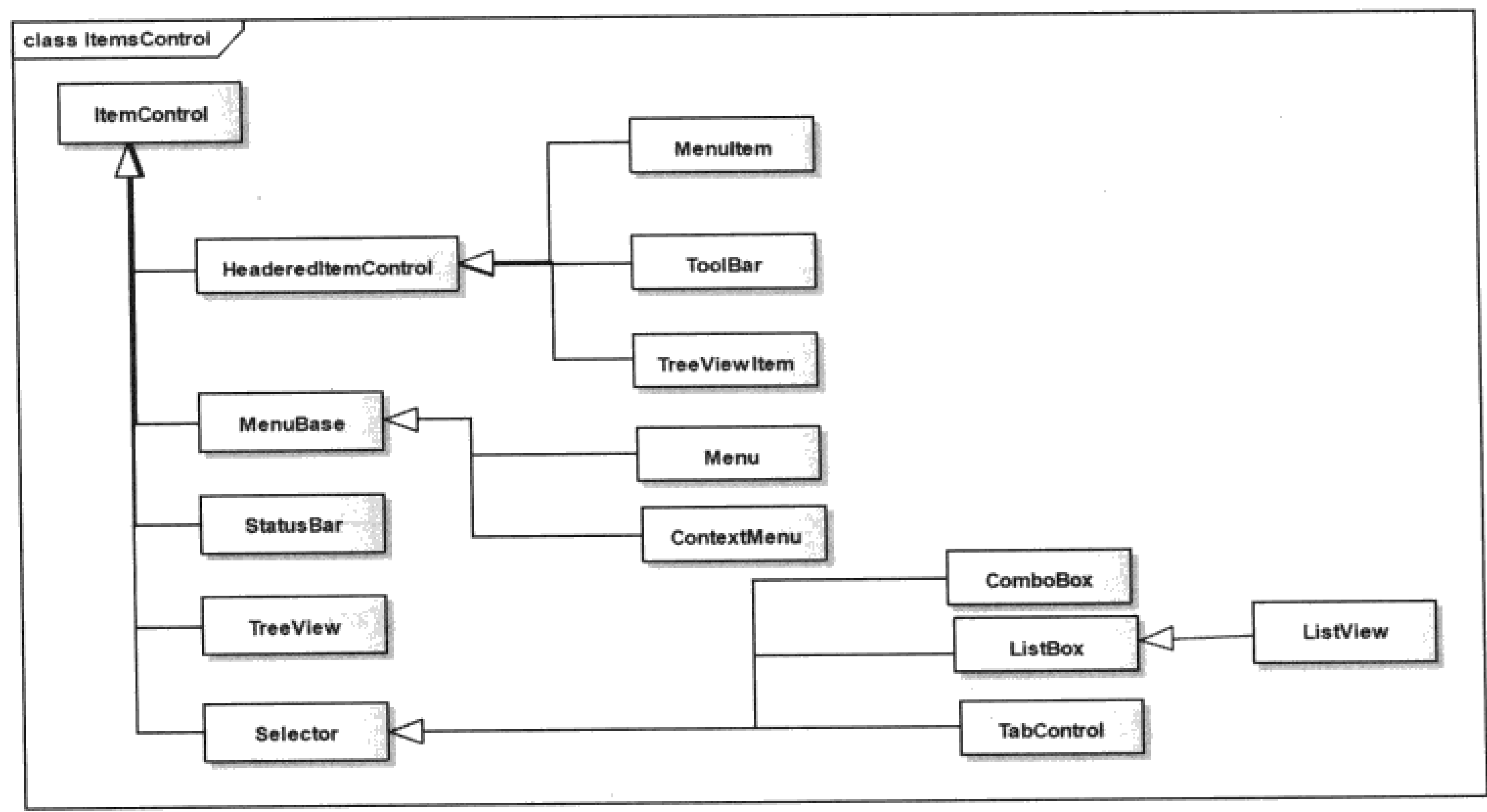


图6-14 ItemsControl类结构图

表6-3列出了条目包容器类及其相应的条目类，由表6-3可见，虽然ItemsControl类中的Items属性中可以加入任何.NET对象，但对某个具体的ItemsControl，其所含的条目类又是相对稳定的，比如表中的ComboBox和ComboBoxItem；ContextMenu和MenuItem等。这些条目为相关控件提供了相对稳定的条目。

表6-3 条目包容器类及相应的条目类

条目包容器类	条目类
ComboBox	ComboBoxItem
ContextMenu	MenuItem
ListBox	ListBoxItem
ListView	ListViewItem
Menu	MenuItem
StatusBar	StatusBarItem
TabControl	TabItem
TreeView	TreeViewItem

6.3.1 菜单（Menu）

菜单是一个常用的控件，常位于应用程序窗口的顶部。每个菜单里面含有菜单条目（MenuItem），MenuItem是一个带有标题的条目控件，它从HeaderedItemControl类中派生出来。HeaderedItemControl类中有一个类型为Object的属性：Header，在菜单条目中通常把它设为字符串。

有时候菜单条目下会有子菜单，子菜单的条目下还会有下一层的子菜单，等等。虽然技术上实现多级子菜单没有什么难度，但笔者不太喜欢多级子菜单，原因在于用户使用起来不方便，通常菜单再加一级子菜单就可以了。特别是引导界面潮流的Microsoft Office软件推出2007版之后，Ribbon控件取代了菜单，多级菜单就不值得提倡了。

菜单条目支持类似于CheckBox的选中标记，是布尔类型。IsCheckable属性表明该条目是否支持选中标记，若设为True，则表明可以设置选择标记。IsChecked表明该条目选择的状态，它也是布尔类型。当该值为True时，在该条目上显示一个选中标记。菜单条目的常用属性如表6-4所示。

表6-4 菜单条目常用属性（MenuItem）

属性名	功 能
Icon	用来显示与菜单条目相关的图像，类型为Object，可以显示任何图形
InputGestureText	用来显示热键，如Word里的复制功能菜单条目显示"Ctrl+C"，注意，该属性只显示字符串，并没有移植热键的功能
IsCheckable	表明该菜单条目是否可以设置IsChecked属性
IsChecked	若菜单条目的IsChecked属性为true，则显示选中标记
IsPressed	表明用户是否按下该条目
Command	菜单条目可以直接和命令相连，通常和WPF中与定义的命令ApplicationCommands相连， 也可以自定义Command

菜单条目里的Icon属性用来显示图像，这个属性的类型为Object，表明可以是任何.NET对象，如程序员自己绘制的图形。本节菜单的例子里，笔者使用了图标ICon，在XAML中引入菜单图标的语法为：

```
<MenuItem.Icon >
     <Image Source ="Image\open.png" />
</MenuItem.Icon>
```

这里的open.png位于Image目录下（在Visual Studio项目的下面），需要注意的是要把.png文件加入到Visual Studio的项目中，否则不工作。例外一种在XAML中引入图标的方法，是在资源中定义，详情可参见第8章。菜单有两种用法，一种是下拉式菜单，另一种是弹出菜单。下面我们来看一个下拉菜单的实例。

● 下拉菜单的例子

```
<Window x:Class="Yingbao.Chapter6.UseMenu"
   xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Yingbao.Chapter6" Height="300" Width="500">
  <DockPanel >
    <Menu DockPanel.Dock ="Top" >
      <MenuItem Header ="文件" >
        <MenuItem Header="打开" InputGestureText ="Ctrl+O">
          <MenuItem.Icon >
                  <Image Source ="Image\open.png" />
          </MenuItem.Icon>
        </MenuItem>
        <MenuItem Header="保存" InputGestureText ="Ctrl+S">
          <MenuItem.Icon >
            <Image Source ="Image\save.png" />
          </MenuItem.Icon>
```

```
          </MenuItem>
          <MenuItem Header="另存为...">
            <MenuItem.Icon >
              <Image Source ="Image\save_as.png" />
            </MenuItem.Icon>
        </MenuItem>
          <Separator />
          <MenuItem Header ="打印预览"/>
          <MenuItem Header ="打印" InputGestureText ="Ctrl+S"/>
          <MenuItem Header ="退出"/>
        </MenuItem>
        <Separator />
        <MenuItem Header ="编辑">
            <MenuItem Header ="恢复"/>
            <MenuItem Header ="重做"/>
            <Separator />
          <MenuItem Header ="拷贝" InputGestureText ="Ctrl+C" Command
              ="ApplicationCommands.Copy">
            <MenuItem.Icon >
              <Image Source ="Image\copy.png" />
            </MenuItem.Icon>
          </MenuItem>
          <MenuItem Header ="裁剪" InputGestureText ="Ctrl+X" Command
                ="ApplicationCommands.Cut">
            <MenuItem.Icon >
              <Image Source ="Image\cut.png" />
            </MenuItem.Icon>
          </MenuItem>
          <MenuItem Header ="粘贴" InputGestureText ="Ctrl+P" Command
                ="ApplicationCommands.Paste">
            <MenuItem.Icon >
              <Image Source ="Image\paste.png" />
            </MenuItem.Icon>
          </MenuItem>
          <MenuItem Header="字体">
            <MenuItem Header="黑体" IsCheckable="True"
                      Checked="Bold_Checked"
                      Unchecked="Bold_Unchecked"/>
            <MenuItem Header="斜体" IsCheckable="True"
                      Checked="Italic_Checked"
                      Unchecked="Italic_Unchecked"/>
            <Separator/>
            <MenuItem Header="增大字体"
                      Click="IncreaseFont_Click"/>
            <MenuItem Header="缩小字体"
                      Click="DecreaseFont_Click"/>
          </MenuItem>
        </MenuItem>
      </Menu>
      <TextBox Name ="textBox1" Width ="450" Height ="150"/>
    </DockPanel>
</Window>
```

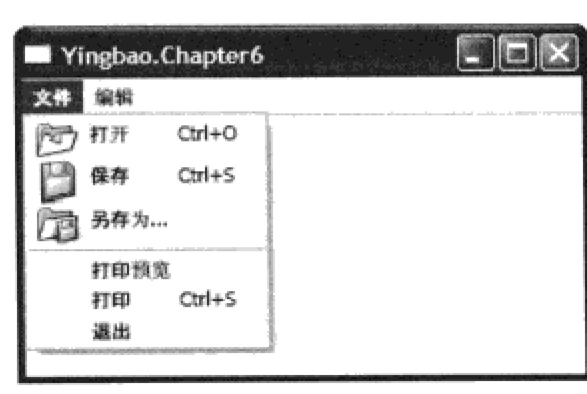
笔者在这个例子中使用了菜单里的常用功能：设置显示热键的InputGestureText，在某些菜单条目里显示图标，在某些菜单条目里使用子菜单，以及在某些菜单条目里使用了前面所说的选择功能等。

在后台C#的程序中处理UI相关事件：

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Data;
using System.Windows.Documents;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Media.Imaging;
using System.Windows.Shapes;

namespace Yingbao.Chapter6
{
    public partial class UseMenu : System.Windows.Window
    {
        public UseMenu()
        {
            InitializeComponent();
        }
   private void Bold_Checked(object sender, RoutedEventArgs e)
        {
            textBox1.FontWeight = FontWeights.Bold;}

   private void Bold_Unchecked(object sender, RoutedEventArgs e)
        {
            textBox1.FontWeight = FontWeights.Normal;
        }

   private void Italic_Checked(object sender, RoutedEventArgs e)
        {
            textBox1.FontStyle = FontStyles.Italic;
        }

   private void Italic_Unchecked(object sender, RoutedEventArgs e)
        {
            textBox1.FontStyle = FontStyles.Normal;
        }

   private void IncreaseFont_Click(object sender, RoutedEventArgs e)
        {
            if (textBox1.FontSize < 60)
            {
                textBox1.FontSize += 4;
            }
        }

   private void DecreaseFont_Click(object sender, RoutedEventArgs e)
```

```
        {
            if (textBox1.FontSize > 10)
            {
                textBox1.FontSize -= 4;
            }
        }
    }
}
```

运行WPF中下拉菜单的例子，如图6-15所示。



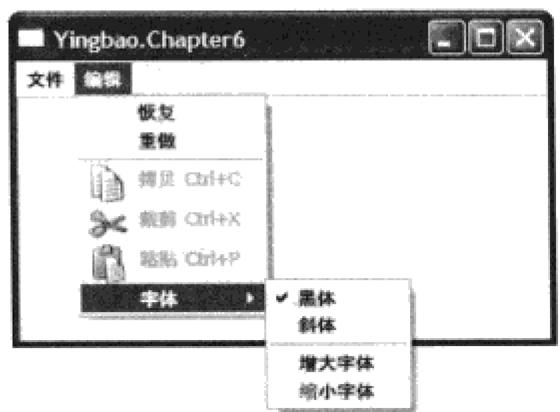


图6-15 WPF中的下拉菜单

若在上例中的字符框里输入字符，同时选择“黑体”、斜体、增大或缩小字体，便可以看到不同的效果。

在表6-4中，笔者提到了InputGestureText只负责显示热键信息，要是热键工作，需要用到命令Command属性。为方便程序员快速开发界面，WPF定义了一个ApplicationCommands的类。

ApplicationCommands类中定义了编辑软件常用的命令，这些命令有：Close、Copy、Cut、Delete、Find、Help、New、Open、Paste、Print、PrintPreview、Redo、Replace、Save、SaveAs, SelectAll、Stop、Undo等。当ApplicationCommands和菜单相连，它还具有自动Enable/Disable菜单条目的功能。

● 弹出菜单（ContextMenu）

与下拉菜单和排版类相连不同，弹出菜单则要附加在UI元素的ContextMenu属性上。FrameworkElement里面有一个ContextMenu属性，程序员所要做的就是构建弹出菜单，并在相应的UI元素上设置ContextMenu属性。由于FrameworkElement是所有UI元素的基类，所以，WPF在广泛的范围内支持弹出菜单。

使用弹出菜单和下拉菜单类似，为了使下面的例子更有趣，可以把一个.NET类实例作为MenuItem的条目。

```
using System;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Controls.Primitives;
using System.Windows.Data;
using System.Windows.Input;
```

```
using System.Windows.Media;
using System.Windows.Shapes;

namespace Yingbao.Chapter6
{
    class ColorBox : ListBox
    {
        string[] strColors =
        {
            "Black", "Brown", "DarkGreen", "MidnightBlue",
                "Navy", "DarkBlue", "Indigo", "DimGray",
            "DarkRed", "OrangeRed", "Olive", "Green",
                "Teal", "Blue", "SlateGray", "Gray",
            "Red", "Orange", "YellowGreen", "SeaGreen",
                "Aqua", "LightBlue", "Violet", "DarkGray",
            "Pink", "Gold", "Yellow", "Lime",
                "Turquoise", "SkyBlue", "Plum", "LightGray",
            "LightPink", "Tan", "LightYellow", "LightGreen",
                "LightCyan", "LightSkyBlue", "Lavender", "White"
        };

        public ColorBox()
        {
            FrameworkElementFactory factoryUnigrid =
                new FrameworkElementFactory(typeof(UniformGrid));
            factoryUnigrid.SetValue(UniformGrid.ColumnsProperty, 8);
            ItemsPanel = new ItemsPanelTemplate(factoryUnigrid);

            foreach (string strColor in strColors)
            {
                Rectangle rect = new Rectangle();
                rect.Width = 12;
                rect.Height = 12;
                rect.Margin = new Thickness(4);
                rect.Fill = (Brush)
                typeof(Brushes).GetProperty(strColor).GetValue(null,
                      null);
                Items.Add(rect);

                ToolTip tip = new ToolTip();
                tip.Content = strColor;
                rect.ToolTip = tip;
            }
            SelectedValuePath = "Fill";
        }
    }
}
```

ColorBox类从ListBox中派生出来，首先用UniformGrid作为ListBox的ItemsPanel。再在ListBox的Items属性中加入矩形，矩形的颜色是预定义的。这样，ColorBox 就会显示预定义的颜色供用户选择。

要在XAML中使用这个类，首先要引入这个类，方法是在前面的菜单程序中引入ColorBox类所在的命名空间：xmlns:src="clr-namespace: Yingbao. chapter6"：

```
<Window x:Class="Yingbao.Chapter6.UseMenu"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Yingbao.Chapter6" Height="300" Width="500"
     xmlns:src="clr-namespace:Yingbao.Chapter6">
```

然后，创建一个弹出菜单并且和TextBox相关联。笔者把有关TextBox的XAML修改如下：

```
<TextBox Name ="textBox1" Width ="450" Height ="150">
      <TextBox.ContextMenu>
        <ContextMenu Name ="Test" StaysOpen ="true">
          <MenuItem Name ="myColor" Header ="颜色" >
            <src:ColorBox HorizontalAlignment="Center"
                          VerticalAlignment="Center" Margin="24"/>
          </MenuItem>
        </ContextMenu>
      </TextBox.ContextMenu>
 </TextBox >
```

这样，就有了一个“颜色”的弹出菜单，当你在TextBox上，按下鼠标右键，就会出现颜色框供你选择。上面弹出菜单的效果如图6-16所示。

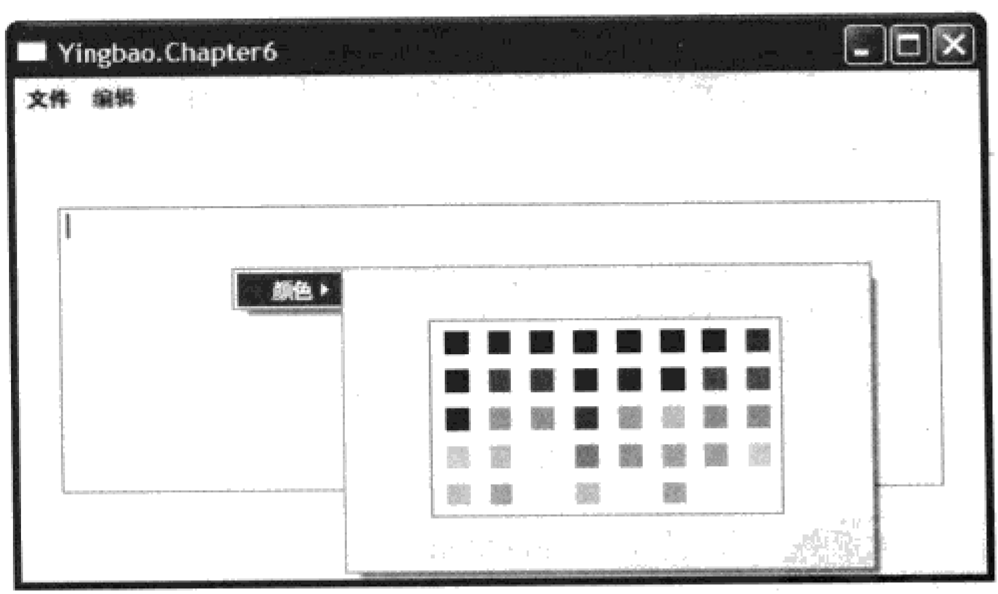


图6-16 自定义颜色框的弹出菜单

6.3.2 工具条（ToolBar）

过去工具条和菜单常常一起用在应用程序中，比如Microsoft Office软件，位于视窗标题栏下面的是菜单，菜单下面就是工具条。有时候，工具条位于一个浮动窗口内，这样用户可以把工具条停靠在屏幕的任何位置。与菜单相比，通常只要在工具条上单击鼠标一次就可以完成某个功能，而菜单通常要单击两次或以上；工具条使用图标加ToolTip的方式，在屏幕上要占据较大的位置，所以只适合把常用的菜单功能放在工具条上。笔者注意到用户常在熟悉某个软件后，更多地使用工具条，故设计软件时，应该把常用的功能放在工具条内。前面提过，Office 2007开始使用Ribbon控件来取代下拉菜单和工具条；AutoCAD 2009版也采用了Ribbon控件。遗憾的是，WPF并不提供Ribbon控件，本书第18

章，讨论了用WPF开发Ribbon控件的情况，笔者在通用电气做的软件，其主界面上使用的也是Ribbon控件。其实，设计Ribbon控件的想法就来源于工具条。

使用WPF工具条要涉及两个类：ToolBarTray和ToolBar。图6-17简单地示出了这两个类间的关系：ToolBarTray含有一个或多个ToolBar，ToolBar中含有一个或多个WPF控件。这里的控件通常是Button、CheckBox、ListBox、ComboBox和TextBox等。

ToolBarTray负责工具条的排版，Background属性用来设定工具条的背景色，Orientation用来确定工具条的方向，它可以取水平（Horizontal）或垂直（Vertical）两个值。

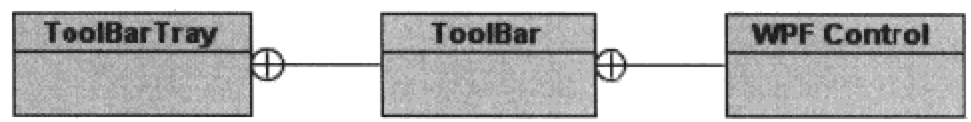


图6-17 工具调整所使用的类

当ToolBarTray里含有两个或两个以上工具条时，需要设置工具条ToolBar的Band属性，这个属性用来指出工具条在ToolBarTray里出现的行或列。用自然数1，2，3…来表示。如果在同一行或同一列中，有两个或两个以上工具条，这时候，我们还要设置BandIndex属性。BandIndex属性用来指出工具条在某个行或某个列中的位置。

有意思的是工具条中的另外两个属性：HasOverFlowItems和IsOverFlowOpen。这两个属性是用来表示工具条是否溢出，以及溢出的工具条是否可见。当工具条中控件的宽度或高度大于工具条所能显示的高度或宽度时，工具条就只能显示其中的部分控件，这时工具条处在“溢出”状态，HasOverFlowItems属性的值为True。若有的控件比较重要，希望该控件不被溢出时，可以设置工具条中的OverflowMode属性为Never，OverflowMode这个属性为附加属性，所以可以把它附加在工具条中的任一控件上。OverflowMode可能取的值为：Always、AsNeeded和Never。当把某个控件的ToolBar.OverflowMode属性设为AsNeeded的时候，该控件会在工具条显示的位置不够时，自动成为不可见，而在工具条的位置足够大时，自动地显示出来。

下面来看一个使用工具条的例子。在这个例子中，笔者使用了两个ToolBarTray，第一个为水平放置，位于窗口的上部；第二个为垂直放置，位于窗口的左边。本例中示出了在ToolBar中使用按钮和组合框的技术，当工具条在所占据的位置比窗口大时，工具条里的控件就自动进入溢出状态（如本例中的字体大小）。由于笔者并没有设置OverflowMode这个属性，当移动窗口边框时，ToolBar中的控件会自动依次从右边进入溢出状态。当工具条中有控件进入溢出状态时，工具条会在右下角显示一个小箭头。

```
<Window x:Class="Yingbao.Chapter6.UsingToolBar"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Yingbao.Chapter6" Height="200" Width="500" >
  <DockPanel>
    <ToolBarTray DockPanel.Dock="Top" Orientation ="Horizontal">
      <ToolBar Band ="1" BandIndex ="1">
        <Button>
          <Image Source ="Image\new.bmp"/>
        </Button>
```

```
    <Button>
      <Image Source ="Image\open.bmp"/>
    </Button>
    <Separator />
    <Button>
      <Image Source ="Image\save.bmp"/>
    </Button>
    <Button>
      <Image Source ="Image\saveas.bmp"/>
    </Button>
  </ToolBar>
  <ToolBar Band ="1" BandIndex ="2">
    <Button>
      <Image Source ="Image\copy.bmp"/>
    </Button>
    <Button>
      <Image Source ="Image\paste.bmp"/>
    </Button>
    <Separator />
    <Button>
      <Image Source ="Image\cut.bmp"/>
    </Button>
  </ToolBar>
  <ToolBar Band ="2" BandIndex ="1">
    <ComboBox Width ="100">
      <ComboBoxItem  Background ="Yellow">200%</ComboBoxItem>
      <ComboBoxItem  Background ="Yellow" >150%</ComboBoxItem>
      <ComboBoxItem  Background ="Yellow" >120%</ComboBoxItem>
      <ComboBoxItem  Background ="Yellow" >100%</ComboBoxItem>
      <ComboBoxItem  Background ="Yellow" >80%</ComboBoxItem>
      <ComboBoxItem  Background ="Yellow" >20%</ComboBoxItem>
    </ComboBox>
    <ComboBox Width ="100">
      <ComboBoxItem  >Times New Roman</ComboBoxItem>
      <ComboBoxItem  >宋体</ComboBoxItem>
    </ComboBox>
    <ComboBox Width="50">
      <ComboBoxItem  >12</ComboBoxItem>
      <ComboBoxItem  >10</ComboBoxItem>
      <ComboBoxItem  >8</ComboBoxItem>
      <ComboBoxItem  >14</ComboBoxItem>
      <ComboBoxItem  >16</ComboBoxItem>
      <ComboBoxItem  >20</ComboBoxItem>
    </ComboBox>
  </ToolBar>
</ToolBarTray>
<ToolBarTray DockPanel.Dock ="Left " Orientation ="Vertical">
  <ToolBar >
    <Button>
      <Image Source ="Image\preview.bmp"/>
    </Button>
    <Button>
      <Image Source ="Image\print.bmp"/>
```

```
        </Button>
      </ToolBar>
    </ToolBarTray>
  </DockPanel>
</Window>
```

图6-18示出了上面这段程序的运行结果。

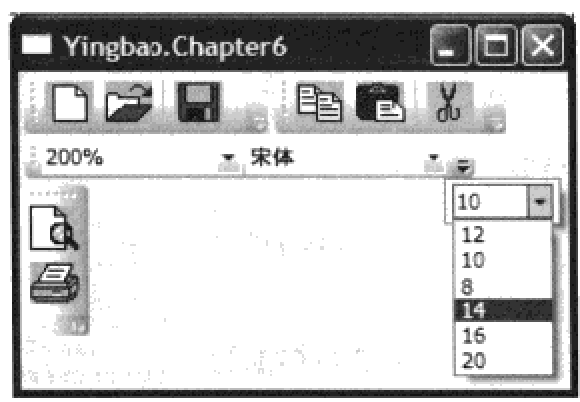


图6-18 工具条示例

6.3.3 Selector

Selector类是下面三个条目控件的基类：

- ComboBox
- ListBox
- TabControl

这三个控件共同的特点是：其中包含一个或多个条目供用户选择。而这个共同的特点，自然地就移植在它们的基类——Selector类中。ListBox控件还有一个派生类——ListView，它自然地继承了Selector类中所移植的特性。

Selector类是一个抽象类，不可以直接创建Selector对象。表6-5列出了Selector类属性，在ComboBox、ListBox、TabControl以及ListView中都会用到这些属性。

表6-5 Selector中的常用属性

属性名	功能
SelectedIndex	读写属性。设置或获取所选择条目的序号，若没有选择任何条目，则返回-1；若选择了多个条目，则返回第一个条目的序号。序号从0开始
SelectedItem	读写属性。设置或获取当前选择的条目对象，若没有选择条目，则返回为null, 为Object类型
SelectedValue	SelectedValue和SelectedPath这两个属性合起来使用。实现对条目控件的数据绑定
SelectedPath	

6.3.4 组合框（ComboBox）

组合框主要由字符框（TextBlock）、拨动按钮（ToggleButton）和Popup几个控件组成，如果使用前面我用过的Snoop工具，会发现其中还有其他的一些控件，但其基本功能是由这三个控件组成的。TextBlock用于显示所选中的条目，ToggleButton给用户提供按键操作，当用户按下ComboBox右边的按钮时，Popup控件显示ComboBox中的条目内容。

有时需要组合框支持用户在输入字符串时自动选择组合框中的条目的特性，用户输入的字符串可以不在组合框的条目中；有时候又需要限制用户输入字符串。当需要选择合适的组合框特性时，要设置IsEditable和IsReadOnly两个属性，表6-6列出了这两个属性的值及其对组合框属性的影响。

表6-6 组合框中IsEditable和IsReadOnly属性

IsEditable	IsReadOnly	功能描述
False	false	➤ 可以用输入字符的方法选择组合框中的条目； ➤ 不能输入组合框条目所没有的字符串； ➤ 不能选择条目中的部分字符； ➤ 不支持复制、粘贴功能
False	true	同上
True	false	➤ 不能用输入字符的方法选择组合框中的条目； ➤ 不能输入组合框条目所没有的字符串； ➤ 可以选择条目中的部分字符； ➤ 支持复制、粘贴功能
True	true	➤ 可以用输入字符的方法选择组合框中的条目； ➤ 可以输入组合框条目所没有的字符串； ➤ 可以选择条目中的部分字符； ➤ 可以复制但不能粘贴

为了测试上述属性，笔者设计了下面的简单程序。这个程序里，使用了三个组合框，一个是选择IsReadOnly的值，一个是选择IsEditable的值：

```
<Window x:Class="Yingbao.Chapter6.UseComboBox"
   xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
   xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
   Title="Yingbao.Chapter6" Height="200" Width="500" >
  <Grid>
    <Grid.RowDefinitions>
      <RowDefinition />
      <RowDefinition />
    </Grid.RowDefinitions>
    <Grid.ColumnDefinitions>
      <ColumnDefinition />
      <ColumnDefinition />
    </Grid.ColumnDefinitions>
<StackPanel Grid.Column ="0" Grid.Row ="0"
       Orientation ="Horizontal" Margin ="5">
      <Label Height ="30">IsEditable:</Label>
<ComboBox Name="cIsEditable" SelectedIndex ="0"
     SelectionChanged ="OnEditableChanged"  Height ="30">
        <ComboBoxItem>True</ComboBoxItem>
        <ComboBoxItem>False</ComboBoxItem>
      </ComboBox>
    </StackPanel>
<StackPanel Grid.Column ="1" Grid.Row ="0"
          Orientation ="Horizontal" Margin="5">
      <Label Height ="30">IsReadOnly:</Label>
<ComboBox Name="cIsReadOnly" SelectedIndex ="0"
       SelectionChanged ="OnReadOnlyChanged" Height ="30">
        <ComboBoxItem>True</ComboBoxItem>
```

```
        <ComboBoxItem>False</ComboBoxItem>
      </ComboBox>
    </StackPanel>
<ComboBox Name="cResult" SelectedIndex ="0" IsReadOnly="True"
     IsEditable ="True"     Grid.Row="1" Grid.ColumnSpan ="2"
     Height ="30" Width ="200" Margin ="10">
      <ComboBoxItem>C# Programming</ComboBoxItem>
      <ComboBoxItem>Java Programming</ComboBoxItem>
      <ComboBoxItem>Layer Architecture</ComboBoxItem>
      <ComboBoxItem>Service Oriented Architecture</ComboBoxItem>
      <ComboBoxItem>Data mining for Oracle</ComboBoxItem>
    </ComboBox>
  </Grid>
</Window>
```

与前面的做法一样，笔者在C#中处理cIsReadOnly和cIsEditable组合框中所选择的值发生变化时的事项：

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Data;
using System.Windows.Documents;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Media.Imaging;
using System.Windows.Shapes;

namespace Yingbao.Chapter6
{

    public partial class UseComboBox : System.Windows.Window
    {

        public UseComboBox()
        {
            InitializeComponent();
        }

        private void OnReadOnlyChanged(object sender,
                  RoutedEventArgs ea)
        {
bool isReadOnly = this.cIsReadOnly.SelectedIndex == 0 ?
    true : false;
            if( cResult != null )
            cResult.IsReadOnly = isReadOnly;
        }
        private void OnEditableChanged(object sender,
                  RoutedEventArgs ea)
        {
bool isEditable= this.cIsEditable.SelectedIndex
```

```
        ==0?true:false;
             if (cResult != null)
             cResult.IsEditable = isEditable;
         }
     }
}
```

这个程序很简单，即根据cIsEditable和cIsReadOnly中所选择的值，分别设置组合框cResult中的IsEditable和IsReadOnly两个属性。由于cIsEditable和cIsReadOnly组合框中只有“True”和“False”两个值可供选择，所以，笔者可以用

```
bool isReadOnly = this.cIsReadOnly.SelectedIndex == 0 ? true : false;
```

还要注意，在设置结果组合框的属性前，检查cResult是否为空：

```
if (cResult != null)  cResult.IsEditable = isEditable;
```

这是因为，在XAML中先创建了cIsEditable和cIsReadOnly组合框，而且在这两个组合框里，一开始就设定了SelectedIndex值。当设定这个属性时，会自动触发SelectionChanged事项，而这时cResult组合框还没有创建出来，若这时调用：

```
cResult.IsEditable = ….
```

就会出现“object reference not set to an instance of an object”（对象引用没有指向实例对象）的异常情况。

6.3.5 TabControl

TabControl在Win32中就有了，不过过去常用在会话框上。它就像一个文件夹，每个文件夹里含有特定的文件，用一个标签贴在文件夹的侧面，然后把文件夹叠起来。这样，比把文件铺开在桌面上要省地方，这就是设计TabControl的最初目的。在应用程序中是否使用TabControl过去是有争议的，主张使用TabControl的人主要理由是可以在会话框上把相关信息放在同一个TabControl里，这样可以在一个会话框里显示更多的信息；反对使用TabControl的人认为，会话框使得某些信息在用户界面上不太明显，使得某些重要的信息可能会被用户所遗漏；特别是在TabControl的某个条目不在最上面、而又有错误时，如何显示错误信息，常是一个难题。笔者过去曾经用在TabControl的顶端显示一个带有“X”的红色圆圈来表示该条目下有严重错误，用带有“！”的红色圆圈来表示该条目下有警告信息；但终究无法具体到在某个控件上直接显示出错信息（因为这时的控件在TabControl里面，不可见！）。

不过近来TabControl又有了流行的趋势，如为.NET提供控件的某些公司DevExpress等，开发了自动切换TabControl和MDI（Multiple Document Interface多文档界面)的控件。使得TabControl不再局限于用在会话框上了，它可以直接用于视窗的主程序上，而且，每个TabControl下面还可以有TabControl。也许你已经注意到了IE7、Chrome和FireFox浏览器，使用的都是基于TabControl的多文档（MDI）界面（它们不是WPF应用程序）。

如表6-3所示，TabControl里面可以含有一个或多个TabItem（至少含有一个）。TabItem是一个带有标题栏的内容控件（如图6-2所示），它可以显示一个标题，而其中的内容，就像所有内容控件一

样可以是任何对象！

TabControl有一个属性TabStripPlacement，这个属性可以取四个值，Top、Left、Botton和Right。它是用来指定放置TabControl在窗口中的位置的。

使用TabControl非常简单，看一个笔者用TabControl制作的诗签：

```
<Window x:Class="Yingbao.Chapter6.UseTabControl"
   xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Yingbao.Chapter6" Height="400" Width="500">
  <Grid>
    <Grid.RowDefinitions>
      <RowDefinition Height ="2*" />
      <RowDefinition Height ="*" />

    </Grid.RowDefinitions>
<TabControl  TabStripPlacement ="Top"  Margin ="5,2,10,10"
         Grid.Row ="0" Grid.Column ="0">
      <TabItem Header="李白" Background ="Cyan" FontSize ="12">
        <StackPanel >
        <TextBlock  FontWeight ="Bold" Margin ="0,0,0,20">观庐山瀑布
             </TextBlock>
        <TextBlock>
             日照香炉生紫烟,<LineBreak/>
             遥看瀑布挂前川。<LineBreak/>
             飞流直下三千尺,<LineBreak/>
             疑是银河落九天。<LineBreak/>
           <LineBreak/><LineBreak/>
      </TextBlock>
        </StackPanel>
      </TabItem>
      <TabItem Header="杜甫" Background ="LightPink" FontSize ="12">
        <StackPanel >
          <TextBlock  FontWeight ="Bold" Margin ="0,0,0,20">春夜喜雨
               </TextBlock>
          <TextBlock>
            好雨知时节，当春乃发生。<LineBreak/>
            随风潜入夜，润物细无声。<LineBreak/>
            野径云俱黑，江船火独明。<LineBreak/>
            晓看红湿处，花重锦官城。<LineBreak/><LineBreak/><LineBreak/>
            <LineBreak/><LineBreak/>
          </TextBlock>
        </StackPanel>
      </TabItem>
<TabItem Header="李应保" Background ="LightBlue"
           FontSize ="12">
        <ScrollViewer HorizontalScrollBarVisibility ="Auto"
                VerticalScrollBarVisibility ="Auto">
          <StackPanel >
            <TextBlock  FontWeight ="Bold" Margin ="0,0,0,20">
```

```
                听 雨
              </TextBlock>
              <TextBlock>
                黎明时分，空中穿梭的雪线<LineBreak/>
                迟疑地——<LineBreak/>
                化作年来第一场春雨<LineBreak/>
                敲打我的窗帘<LineBreak/><LineBreak/><LineBreak/>
                想你，就坐在我的对面<LineBreak/>
                双手捧着杯子<LineBreak/>
                小心翼翼地，把<LineBreak/>
                不曾表达的心事，又抚摸了一遍<LineBreak/>
                只有颔首 低眉<LineBreak/>
                于异国的音乐中<LineBreak/>
                感知那种迟来的危险——<LineBreak/>
                那个词，安睡了多年<LineBreak/>
                终究没有被你放到舌尖<LineBreak/><LineBreak/><LineBreak/>
                多像——<LineBreak/>
                一阵朦胧的春雨，拂过湖面<LineBreak/>
                我的目光，滑过你的指尖<LineBreak/>
                渗入你的双眼，凝结成<LineBreak/>
                无奈的怀念<LineBreak/><LineBreak/><LineBreak/>
黎明时分，你就坐在我的对面<LineBreak/>
       <LineBreak/><LineBreak/>
              </TextBlock>
            </StackPanel>
          </ScrollViewer>
       </TabItem>
     </TabControl>
<TabControl  TabStripPlacement ="Bottom"  Margin ="5,2,10,10"
       Grid.Row ="1" Grid.Column ="0">
       <TabItem Header="李商隐" Background ="Cyan" FontSize ="12">
         <StackPanel >
           <TextBlock  FontWeight ="Bold" Margin ="0,0,0,20">夜雨寄北
              </TextBlock>
           <TextBlock>
             君问归期未有期，巴山夜雨涨秋池。<LineBreak/>
何当共剪西窗烛，却话巴山夜雨时。
<LineBreak/><LineBreak/><LineBreak/>
           </TextBlock>
         </StackPanel>
       </TabItem>
    </TabControl >
    </Grid>
</Window>
```

在这个例子中，笔者把TabItem的标题设为诗人的名字，把条目的内容设为StackPanel，再在StackPanel里加入TextBlock，用TextBlock显示相应的诗歌。

运行该程序的结果如图6-19所示。

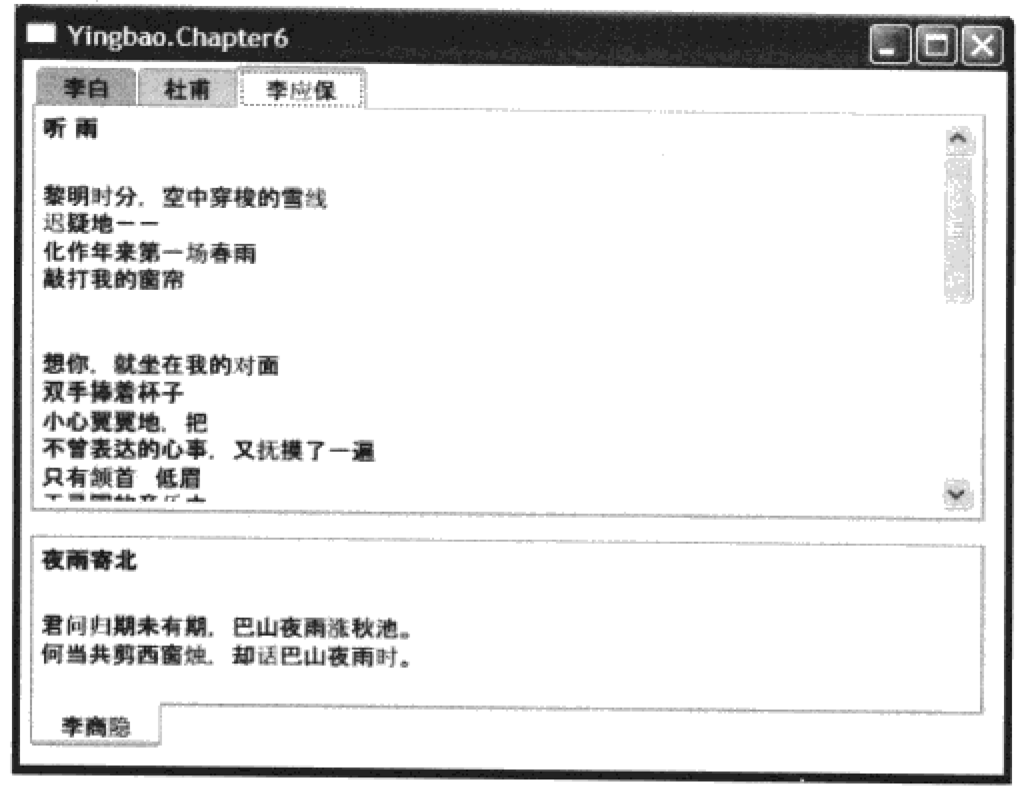


图6-19 用TabControl做的诗签

程序中用的《听雨》一诗，是笔者2003年写的。当时我在多伦多一家美国金融软件公司做系统软件分析，光阴荏苒，转眼5年过去了。今日重读，人生际遇，感慨良多。

6.3.6 列表框（ListBox）

ListBox这个控件早在OS2中就有，用于开发Win16的Borland公司的OWL库里也含有这个控件；所以，ListBox和按钮Button一样，是较早的Windows上的控件之一。

ListBox中含有至少一个ListBoxItem。如图6-2所示，ListBoxItem从内容控件(ContentControl)中派生出来，因此具有内容控件的特征。ListBox中有一个SelectionMode属性，该属性为枚举类型。可以取Single、Multiple和Extended三个值，当SelectionMode设为Single时，用户只能选择ListBox中一个条目；当SelectionMode设为Multiple时，用户可以选择ListBox中多个条目；当Selection Mode设为Extended时，用户需要按下“SHIFT”键，才能选择ListBox中的多个条目；由于ListBox允许用户选择多个ListBoxItem，所以它有一个SelectedItems。这是一个集合，其中含有用户选择的多个条目。

下面使用ListBox：

```
<Window x:Class="Yingbao.Chapter6.UseListBox"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Yingbao.Chapter6" Height="300" Width="500"  >
    <Grid>
      <Grid.RowDefinitions >
        <RowDefinition Height ="0.5*" />
        <RowDefinition Height ="*" />
        <RowDefinition  Height ="2*"/>
      </Grid.RowDefinitions>
      <Grid.ColumnDefinitions>
        <ColumnDefinition  Width ="*"/>
```

```
        <ColumnDefinition  Width ="2*"/>
      </Grid.ColumnDefinitions>
<Label Grid.Row ="0" Grid.Column ="0" Margin ="20,0,0,0">选择模
    式: </Label>
<ListBox  Name ="lMode" Grid.Row ="0" Grid.Column ="1"
    SelectionMode ="Single" SelectionChanged ="OnChangeMode">
        <ListBoxItem>Single</ListBoxItem>
        <ListBoxItem>Multiple</ListBoxItem>
        <ListBoxItem>Extended</ListBoxItem>
      </ListBox >
        <Label Grid.Column ="0" Grid.Row ="1" Margin ="20,0,0,0">请选
            择业余爱好: </Label>
<ListBox  Name="lHobby" Grid.Row ="1" Grid.Column ="1"
      SelectionChanged ="OnSelectHobby">
        <ListBoxItem> 钓鱼</ListBoxItem>
        <ListBoxItem>绘画</ListBoxItem>
        <ListBoxItem>书法</ListBoxItem>
        <ListBoxItem>旅游</ListBoxItem>
        <ListBoxItem>羽毛球</ListBoxItem>
        <ListBoxItem>乒乓球</ListBoxItem>
        <ListBoxItem>篮球</ListBoxItem>
        <ListBoxItem>网球</ListBoxItem>
      </ListBox>
<Label Grid.Row ="2" Grid.Column ="0" Margin ="20,0,0,0">你选择
    了: </Label>
<TextBlock Name ="tResult"  Grid.Row="2" Grid.Column ="1"
    Margin ="5" Background ="LightCoral"/>
    </Grid>
</Window>
```

在这个例子中，笔者使用了两个ListBox。第一个ListBox设定第二个ListBox的选择模式（SelectionMode），而它自己的选择模式设为Single，即只能在其条目中选择一个；其实，这里使用ComboBox可能更合适。第二个ListBox列出了个人爱好，可以根据第一个ListBox中设定的选择模式，允许用户选择一个或多个爱好。注意，当SelectionMode为Multiple时，可以任意选择多个选项；当SelectionMode为Extended的时候，必须按下“SHIFT”键，而且只能选择ListBox中的相邻的选项。

下面是处理事项的C#程序，该程序示出了如何处理SelectionMode为单个或多个选项的ListBox。

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Data;
using System.Windows.Documents;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Media.Imaging;
```

```
using System.Windows.Shapes;

namespace Yingbao.Chapter6
{
    public partial class UseListBox : System.Windows.Window
    {

        public UseListBox()
        {
            InitializeComponent();
        }
        private void OnChangeMode(object sender, RoutedEventArgs ea)
        {
            SelectionMode smode = SelectionMode.Single;
               switch( lMode.SelectedIndex ){
                    case 0:
                         smode=SelectionMode.Single;
                         break;
                    case 1:
                         smode=SelectionMode.Multiple;
                         break;
                    case 2:
                         smode=SelectionMode.Extended;
                         break;
               }
               lHobby.SelectionMode = smode;
        }


        private void OnSelectHobby(object sender,
                         RoutedEventArgs ea)
        {
            tResult.Text = string.Empty;
            StringBuilder sb = new StringBuilder();
            foreach (ListBoxItem  item in lHobby.SelectedItems)
            {
                 sb.Append(item.Content.ToString()).Append("\n");
            }
            tResult.Text = sb.ToString();
        }
    }
}
```

图6-20示出了上述程序的运行结果。

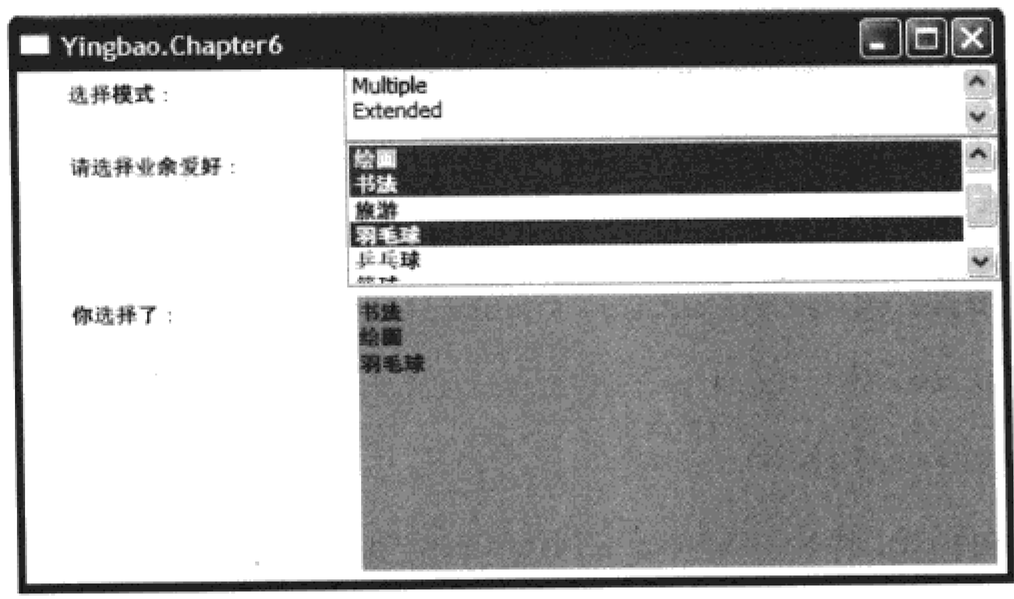


图6-20 ListBox示例

6.3.7 ListView

ListView也是一个老的控件，它和ListBox非常类似，所以它是ListBox的派生类。ListView和ListBox不同的地方在于，ListView有多列，而且每列都有头。ListView默认的SelectionMode为Extended。

在XAML中创建ListView头非常简单，下面的几行代码就创建了一个ListView的头：

```
<ListView>
  <ListView.View>
    <GridView>
      <GridViewColumn  Width ="100" Header ="姓名"/>
      <GridViewColumn  Width ="100" Header ="工作年限"/>
      <GridViewColumn  Width ="100" Header ="办公电话"/>
      <GridViewColumn  Width ="100" Header ="电邮"/>
    </GridView>
  </ListView.View>
</ListView>
```

若在上面的ListView里放入数据，通常ListView用于建立数据表。如直接和ADO的DataSet中的Table相连，或与XML数据进行绑定。有关数据绑定，我将在本书的第11章详细介绍。这里为简单起见，我在C#中创建一个PersonalInfoList：

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Data;
using System.Windows.Documents;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Media.Imaging;
using System.Windows.Shapes;
using System.Collections.ObjectModel;
```

```
namespace Yingbao.Chapter6
{
    public partial class UseListView : System.Windows.Window
    {
        private ObservableCollection<PersonalInfo> personalInfoList
          = new ObservableCollection<PersonalInfo>();
        public ObservableCollection<PersonalInfo> PersonalInfoList
        {
            get { return this.personalInfoList; }
        }
        public UseListView()
        {
           personalInfoList.Add(new PersonalInfo("程俊", 8,
                  "4338819", "chenJun@hotmail.com"));
           personalInfoList. Add(new PersonalInfo("张珊", 12,
                  "6566665", "zhangshang@hotmail.com"));
           personalInfoList. Add(new PersonalInfo("陶铸", 7,
                  "65765675", "taozhu@hotmail.com"));
           personalInfoList.Add(new PersonalInfo("李伟光", 25,
                  "342354325", "liweiguang@hotmail.com"));
           personalInfoList.Add(new PersonalInfo("刘表", 3,
                   "2342343", "liubiao@hotmail.com"));
            InitializeComponent();
        }

        private void AddRow_Click(object sender,
                  RoutedEventArgs rea)
        {
         personalInfoList.Add(new PersonalInfo("林晓霜", 1,
         "34523654", "xiaoshuanglin@hotmail.com"));
        }
    }

    public class PersonalInfo
    {
       private string name;
       public string Name
      {
            get{  return this.name;}
            set{ this.name = value;}
       }

       private int workYears;
       public int WorkYears
       {
            get{ return this.workYears;}
            set {this.workYears = value;}
       }

        private string workPhoneNumber;
        public string WorkPhoneNumber
```

```
        {
            get{ return this.workPhoneNumber; }
            set{ this.workPhoneNumber = value;}
        }

        private string email;
        public string Email
        {
            get{ return this.email;}
            set{ this.email = value;}
        }

        public PersonalInfo(string name, int year,
                string phoneno, string email)
        {
            this.name = name;
            this.workYears = year;
            this.workPhoneNumber = phoneno;
            this.email = email;
        }
    }
}
```

PersonalInfoList是一个ObservableCollection<T>，这个类是.NET 2.0引入的，它是一个模板类，其特点是它允许有一个观察者（Observer，有关Observer设计范例，请参考Gamma等4人合著的《Design Patterns》一书）。当把数据绑定到ListView上时，ListView就是PersonalInfoList的观察者。当PersonalInfoList中的数据发生变化时，ListView会自动反映这种变化。如本例中可以在PersonalInfoList中增加或减少一个记录。

PersonalInfoList里面存放的是PersonalInfo对象，这个类很简单，它存有姓名、工作年限、办公电话、电邮等信息。

下面是简单的XAML程序：

```
<Window x:Class="Yingbao.Chapter6.UseListView"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:x=http://schemas.microsoft.com/winfx/2006/xaml
Title="Yingbao.Chapter6" Height="300" Width="500"
DataContext="{Binding RelativeSource={RelativeSource Self}}">
<StackPanel>
    <ListView ItemsSource="{Binding PersonalInfoList}" >
        <ListView.View>
        <GridView>
          <GridViewColumn  Width ="100" Header ="姓名"
             DisplayMemberBinding="{Binding Path=Name }"  />
          <GridViewColumn  Width ="100" Header ="工作年限"
             DisplayMemberBinding="{Binding Path=WorkYears }"/>
          <GridViewColumn  Width ="100" Header ="办公电话"
             DisplayMemberBinding="{Binding Path=WorkPhoneNumber}"/>
          <GridViewColumn  Width ="100" Header ="电邮"
             DisplayMemberBinding="{Binding Path=Email}"/>
```

```
        </GridView>
      </ListView.View>
    </ListView>
    <Button HorizontalAlignment="Right" Margin="5,5,5,5"
     Content="Add Row" Click="AddRow_Click" />
  </StackPanel>
</Window>
```

图6-21是这个程序的运行结果。由于笔者在上面的程序中，使用的是StackPanel排版，所以在按“增加记录”按钮时，“增加记录”按键自己往下移动。解决这一问题的方法是用Grid或Canvas来代替 StackPanel，本书第3章已对这些排版类做过详细介绍，读者可以很容易解决这一问题。

GridView支持拖放功能，可以使用拖放功能任意改变ListView中列的位置。但遗憾的是，GridView并不支持排序功能。若要对记录进行排序，需要做进一步工作。

图6-21 ListView示例

6.3.8 状态条（StatusBar）

状态条用于显示应用程序的当前状态信息，一般放在窗口的底部。如Office Word主窗口下面的状态条，Word在其中显示文档的当前页、字数等相关信息。StatusBar是一个条目控件，它是ItemsControl的派生类。

使用状态条很简单，例如下面的XAML在窗口中加入一个简单的状态条：

```
<StatusBar VerticalAlignment="Bottom"/>
```

通常要在状态条中加入更多的内容，一般把状态条分为几个区域，在每个区域里动态显示应用程序相关信息。下面的这段XAML把状态条分为4 个区域：

```
<Window x:Class="Yingbao.Chapter6.UseStatusBar"
   xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="UseStatusBar" Height="300" Width="300">
    <Window.Resources>
        <Style TargetType="TextBlock" x:Key="StatusBarTextBlock">
            <Setter Property="TextWrapping" Value="NoWrap" />
            <Setter Property="TextTrimming"
                    Value="CharacterEllipsis" />
            <Setter Property="Padding" Value="2,0" />
```

```
    </Style>
    <Style TargetType="Separator" BasedOn="{StaticResource
            {x:Static ToolBar.SeparatorStyleKey}}">
        <Setter Property="Margin" Value="2,0" />
    </Style>
</Window.Resources>
<Grid>
    <StackPanel>
        <TextBlock FontSize="32">在WPF中使用状态条</TextBlock>
    </StackPanel>
    <StatusBar VerticalAlignment="Bottom">
        <Grid>
            <Grid.ColumnDefinitions>
                <ColumnDefinition Width="Auto" />
                <ColumnDefinition Width="Auto" />
                <ColumnDefinition Width="*" />
                <ColumnDefinition Width="*" />
            </Grid.ColumnDefinitions>
            <StackPanel Orientation="Horizontal"
                Grid.Column="0">
                <TextBlock Style="{StaticResource
                 StatusBarTextBlock}">行数: </TextBlock>
                <TextBlock Style="{StaticResource
                 StatusBarTextBlock}">50</TextBlock>
            </StackPanel>

            <StackPanel Orientation="Horizontal"
                 Grid.Column="1">
                <Separator />
                <TextBlock Style="{StaticResource
                     StatusBarTextBlock}">列数:</TextBlock>
                <TextBlock Style="{StaticResource
                       StatusBarTextBlock}">50</TextBlock>
            </StackPanel>

            <DockPanel LastChildFill="True" Grid.Column="2">
                <Separator DockPanel.Dock="Left" />
                <TextBlock Style="{StaticResource
                   StatusBarTextBlock}"
                   ToolTip="显示提示信息"
                   Text="显示提示信息" />
            </DockPanel>
            <DockPanel LastChildFill="True" Grid.Column="3">
                <Separator DockPanel.Dock="Left" />
                <Image DockPanel.Dock="Left" Margin="2,0"
                     Source="Image\Spell.bmp" Width="16" />
                <TextBlock Style="{StaticResource
                         StatusBarTextBlock}"
                   ToolTip="你的文章中有拼写错误"
                   Text="你的文章中有拼写错误" />
            </DockPanel>
        </Grid>
```

```
        </StatusBar>
    </Grid>
</Window>
```

在这段XAML中，笔者使用了资源及风格。有关资源和风格的技术细节，参见第8章和第9章，图6-22是这段XAML的运行结果。

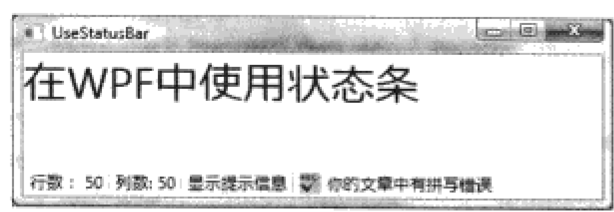


图6-22 StatusBar示例

6.3.9 树形控件TreeView和TreeViewItem

TreeView是一个非常重要的控件，微软在Windows 95 中首先引入树形控件。TreeView用来显示具有树形结构的信息，在现实生活中，有很多树形结构的例子。如家族树始于某个祖宗，随着一代代繁衍，家族树越来越庞大，但没个家族分支，都可以在树形图上找到其来源即“树根”。中国的家谱就是对家族树的描述，我们名字中的中间那个字表示辈分（这就是为什么三个字的名字要比两个字好，两个字的名字我们无法获知辈分信息），你可以很容易地从中识别你在家族书中的分支。树枝分叉的交接处叫做节点，节点有两个状态：展开和关闭，一般用两个不同的图标来表示这两种不同的状态。树形图可以直观地描述节点间的父与子和兄与弟这类关系。

TreeView 中的条目为TreeViewItem。TreeViewItem也是一个条目控件，其直接的父类是HeadedItemsControl。表6-7和表6-8分别列出了TreeView和TreeViewItem中的相关属性。

表6-7 TreeView中的相关属性

属 性	功能描述
SelectedItem	只读。表示当前选中的条目
SelectedValue	只读。读取当前选中的节点上的值
SelectedValuePath	读写属性。用于数据绑定，和SelectedValue联合使用，为选择的节点的路径

表6-8 TreeViewItem中的相关属性

属 性	功能描述
IsExpanded	读写属性。若该节点是展开的，返回True，否则返回False。该值的改变伴随着Expanded和Collapsed传递事件的发生
IsSelected	读写属性。表示该节点是否选中。这个值的改变伴随着Selected、Unselected传递事件的发生
IsSelectionActive	表示该节点是否有输入焦点

如果你有过使用MFC中的CTreeCtrl的经验，立刻就会体会到WPF中的TreeView和TreeViewItem非常好用。现在来看用TreeView来显示一个假想软件公司组织结构的例子：

```
<Window x:Class="Yingbao.Chapter6.UseTreeView"
   xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="UseTreeView" Height="300" Width="300">
    <Grid>
        <Grid.RowDefinitions>
            <RowDefinition Height="2*" />
            <RowDefinition Height="auto" />
```

```
        </Grid.RowDefinitions>
        <TreeView Grid.Row="0" Name="companyTree"
                  SelectedItemChanged="OnSelectItem" >
            <TreeViewItem Header="永华软件工程公司">
                <TreeViewItem Header="人事部">
                    <TreeViewItem Header="王刚"/>
                    <TreeViewItem Header="赵伟"/>
                </TreeViewItem>
                <TreeViewItem Header="销售部">
                    <TreeViewItem>
                        <TreeViewItem.Header>
                            <StackPanel Orientation="Horizontal">
                                 <TextBlock Text="姚新" Width="200"
                                     Margin=" 5,2,10,0"/>
                                 <TextBlock Text="出差" Margin="
                                     5,2,10,0"/>
                                 <CheckBox  IsChecked="True"/>
                            </StackPanel>
                        </TreeViewItem.Header>
                      </TreeViewItem>
                    <TreeViewItem>
                       <TreeViewItem.Header>
                            <StackPanel Orientation="Horizontal">
                                <TextBlock Text="赵勇明" Width="200"
                                    Margin=" 5,2,10,0"/>
                                <TextBlock Text="出差" Margin="
                                    5,2,10,0"/>
                                <CheckBox  IsChecked="False"/>
                            </StackPanel>
                       </TreeViewItem.Header>
                </TreeViewItem>
                </TreeViewItem>
            </TreeViewItem>
        </TreeView>
        <StackPanel Orientation="Horizontal" Grid.Row="1">
            <Label>选择的树节点值：</Label>
            <TextBlock Name="treeNodeValue" Width="300"/>
        </StackPanel>
    </Grid>
</Window>
```

永华软件工程公司树中包括人事部和销售部。人事部下面有两个工作人员，王刚和赵伟；销售部下面有两个销售人员，为了显示销售人员是否在外出差，笔者把销售部门下的TreeViewItem.Header设为StackPanel，其中含有两个TextBlock和一个CheckBox。当用户选择某个节点的时候，有C#对SelectedItemChanged事件进行处理，在treeNodeValue中显示所选择的节点。下面是简单的C#处理程序：

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Windows;
using System.Windows.Controls;
```

```
using System.Windows.Data;
using System.Windows.Documents;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Media.Imaging;
using System.Windows.Shapes;

namespace Yingbao.Chapter6
{
    public partial class UseTreeView : Window
    {
        public UseTreeView()
        {
            InitializeComponent();
        }

        void OnSelectItem(object sender,
                  RoutedPropertyChangedEventArgs<Object>  ea)
        {
            TreeViewItem selectedNode = ea.NewValue as TreeViewItem;
            if (selectedNode != null)
            {
                if (selectedNode.Header is string)
                {
                    treeNodeValue.Text =
                              selectedNode.Header.ToString();
                }
                else if (selectedNode.Header is StackPanel)
                {
                    treeNodeValue.Text = ((selectedNode.Header as
                       StackPanel).Children[0] as TextBlock).Text;
                }
            }
        }
    }
}
```

这一程序的运行结果如图6-23所示。

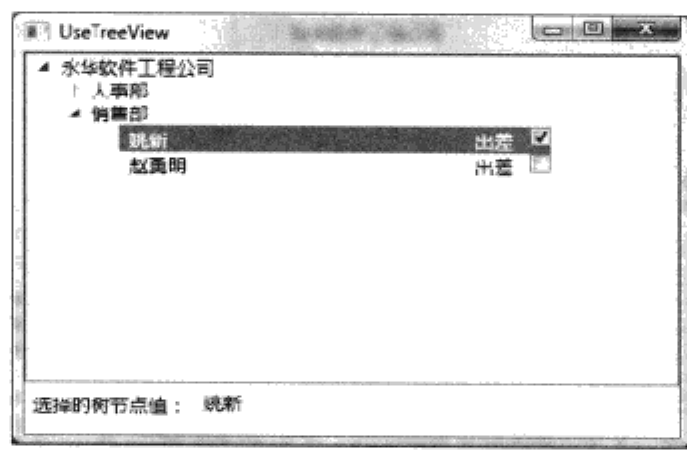


图6-23 TreeView和 TreeViewItem

树形控件特别容易和XML数据绑定起来，在SOA体系结构中，要大量使用XML。有关树形控件的数据绑定，要到第11章进行讨论。

6.4 文本控件（Text Controls）

文本控件共有4个类，这4个类是：用于用户输入口令的PasswordBox；提供基本字符输入的TextBox；用于增强字符输入的RichTextBox；TextBoxBase是TextBox和RichTextBox的基类，它是一个不能实例化的抽象类。图6-24示出了文本控件的类结构图。

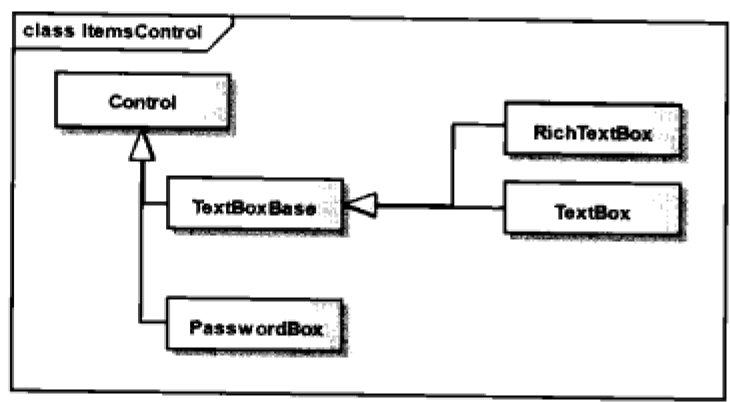


图6-24 文本控件的类结构图

6.4.1 口令输入框（PasswordBox）

在现代软件中，安全是一个重要的课题。为了避免软件系统不被不相关的人使用，通常需要对用户进行身份认证，并根据用户的级别赋予用户相应的功能。这个过程称作Authentication和Authorization。

最常用的Authentication方法就是要求用户输入用户名和口令。用户输入口令通常不能显示出来，以防站在用户边上的人偷看密码，所以需要把用户输入的字符掩藏起来。过去在MFC中，程序员要对TextBox做一些工作，以满足上述要求。WPF提供了用于输入用户口令的控件——口令输入框（PasswordBox）。

口令输入框中MaxLength属性用于设置用户可以输入的最多按键个数，PasswordChar用于设置输入口令的掩藏字符：

```
<Window x:Class="Yingbao.Chapter6.Login"
   xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Yingbao.Chapter6" Height="200" Width="500">
    <Grid>
     <Grid.RowDefinitions>
 <RowDefinition />
 <RowDefinition />
     </Grid.RowDefinitions>
     <StackPanel Grid.Row ="0" Orientation ="Horizontal"
            Margin ="30,20,20,20">
 <Label  Height ="30">用户名:</Label>
 <TextBox  TextBlock.TextAlignment ="Left" Width ="100"
            Height ="20"/>
     </StackPanel>
     <StackPanel Grid.Row ="1" Orientation ="Horizontal"
            Margin ="30,20,20,20">
        <Label  Height ="30">口令:</Label>
```

```
            <PasswordBox MaxLength ="10" Width ="200" Height ="20"
                Name="myPassword" PasswordChar="*"
                PasswordChanged ="OnChangePassword"/>
            </StackPanel>
        </Grid>
</Window>
```

在C#中可以对用户的输入及时处理（如果你希望的话）：

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Data;
using System.Windows.Documents;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Media.Imaging;
using System.Windows.Shapes;

namespace Yingbao.Chapter6
{
    public partial class Login : System.Windows.Window
    {
        public Login()
        {
            InitializeComponent();
        }
        void OnChangePassword(object sender, RoutedEventArgs rea)
        {
            //用户正在输入口令
        }
    }
}
```

运行上面这段程序，结果如图6-25所示。

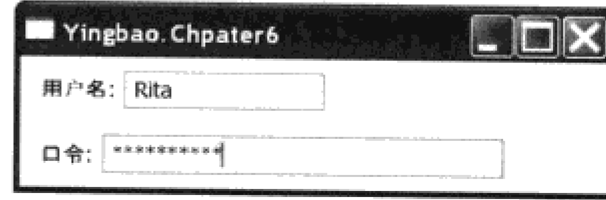


图6-25 使用口令输入框（PasswordBox）

6.4.2 文字输入框（TextBox）

文字输入框（TextBox）是一个非常常用的控件，在前面的程序中已经多次用过。TextBox看似简单，其实它是由多个控件组成的，如果你用Snoop工具检查TextBox，会看到其中含有Border、ScrollViewer、Grid、Rectangle、TextView、ScrollBar等一些控件。

表6-9列出了TextBoxBase中的相关属性，由于TextBox和RichTextBox都是以TextBoxBase为基类，所以这些属性对两种文本输入控件都有效。

表6-9 抽象类TextBoxBase中的常用相关属性

属性名	功能
AcceptsReturn	当该属性设为True，输入“Enter”键时，在光标位置插入换行符；若该属性设为False，忽略输入的“Enter”键
AcceptsTab	当该属性设为True，输入“Tab”键时，在光标位置插入tab控制字符
AutoWordSelection	当该属性设为True时，若选择一个词的部分字母，会自动选择整个词
CanRedo	这是一个只读属性，表示是否可以重做前一个动作
CanUndo	这是一个只读属性，表示是否可以恢复前一个动作
ExtentHeight	该值返回垂直方向可见区域的大小
ExtentWidth	该值返回水平方向可见区域的大小
HorizontalOffset	水平方向滚动位置
HorizontalScrollBarVisibility	设置水平方向滚动条是否可见
IsReadOnly	文字框是否只读
IsUndoEnable	表示是否可以进行恢复操作
SpellCheck	设置SpellCheck.IsEnabled属性，可自动检查文字输入框中的单词的拼写错误
UndoLimit	设置可以恢复动作的次数
VerticalOffset	垂直方向滚动位置
VerticalScrollBarVisibility	设置垂直方向滚动条是否可见
ViewportHeight	设定视窗的高度
ViewportWidth	设定视窗的宽度

若你没有设置TextBox的大小，则TextBox的大小会随着里面的内容变化而变化。在TextBox的宽度不变的情况下，设置TextWraping属性为Wrap，那么其中的文本字符会自动分行。若把TextWraping属性设为NoWrap，则TextBox中的字符不会自动分行。若把TextWraping属性设为WrapWithOverflow，则TextBox只在不同的词间分行，若有一个很长的词，那么就可能只能显示这个词的一部分。

请注意：这里很多的讨论并不适合中文，笔者认为，为了更好地支持中文，还有很多基础工作要做（比如如何检查中文的错别字等）。TextBox还支持某些编辑命令，例如剪辑、粘贴、复制、恢复、重做等。

下面的这段XAML在StackPanel中创建了一个TextBox，并设定了一些相关属性。

```
   <StackPanel  >
<TextBox  Name="myText" AcceptsReturn ="True"
   AcceptsTab ="True"  TextWrapping ="Wrap"
   HorizontalScrollBarVisibility ="Auto"
   VerticalScrollBarVisibility ="Visible"
   SpellCheck.IsEnabled ="True"/>
    </StackPanel>
```

当把SpellCheck.IsEnabled属性设为True时，TextBox会对其中的单词进行拼写检查，一旦发现错误，会在该单词下面显示红色的波浪线（和Microsoft 的Word功能一样）。

6.4.3 RichTextBox

TextBox中的字符只能有一种格式，若你需要用多种方式显示字符，或在字符中插入其他的图像、声音等对象时，就需要使用RichTextBox。

RichTextBox中的大部分属性和TextBox相同，都是继承TextBoxBase中的属性。RichTextBox中有

一个Document属性，其类型为FlowDocument。正是由于这个属性，使得RichTextBox中的内容丰富了起来。

6.5 范围控件（Range Controls）

ScrollBar、ProgressBar和SlideBar这三个控件统称为范围控件，它们都是从同一个基类RangeBase中派生出来的，图6-26示出了这些类之间的关系。

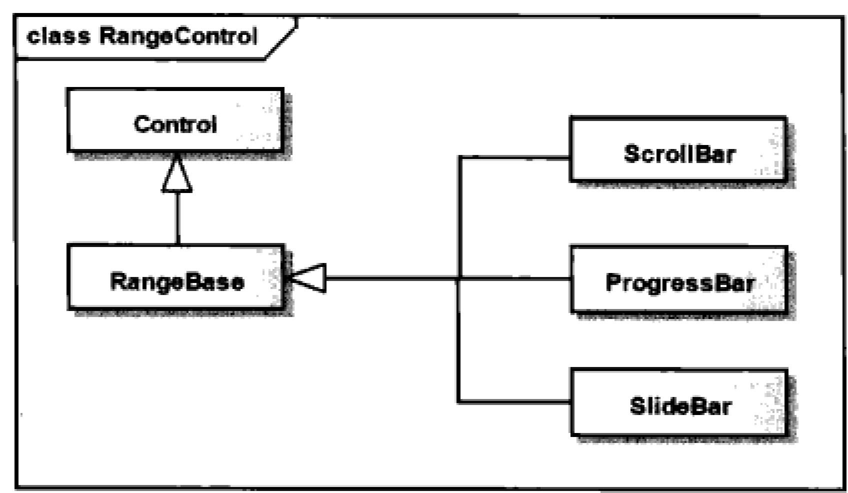


图6-26 范围控件类结构图

RangeBase是一个抽象类，它定义了最大(Maximum)，最小(Minimum)值；范围控件的当前值(Value)，SmallChange和LargeChange两个属性则定义了每次变化的单位。范围控件的取值一定在Minimum和Maximum之间，即：

Minimum < Value < Maximum

6.5.1 滚动条（ScrollBar）

滚动条是一个常用的控件，使用起来非常简单。下面的例子示出了一个水平放置的滚动条：

```
<Window x:Class="Yingbao.Chapter6.UseScrollBar"
   xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Yingbao.Chapter6" Height="80" Width="500">
    <StackPanel Orientation ="Horizontal"  Height ="40">
<ScrollBar Maximum ="100" Minimum ="5"
  Orientation ="Horizontal" SmallChange ="2"
        Scroll ="OnScroll" Width ="200"/>
      <Label Margin ="0,5,0,5">滑动块值: </Label>
      <TextBox Name="txtScrollValue" Margin ="0,5,0,5" Width ="40"/>
    </StackPanel>
</Window>
```

在上面的程序中，笔者设置了ScrollBar的最小值为5，最大值为100，SmallChange为2。为了观察滚动条值的变化，这段程序创建了一个文本控件（TextBox），用于显示滚动条的值。

处理滑动条的滚动事件非常简单，需要注意的是，在C#中需要引入Primitive命名空间。

```
using System;
using System.Collections.Generic;
```

```
using System.Text;
using System.Windows;
using System.Windows.Controls.Primitives;
using System.Windows.Controls;
using System.Windows.Data;
using System.Windows.Documents;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Media.Imaging;
using System.Windows.Shapes;

namespace Yingbao.Chapter6
{
    public partial class UseScrollBar : System.Windows.Window
    {
        public UseScrollBar()
        {
            InitializeComponent();
        }

        private void OnScroll(object sender, RoutedEventArgs rea)
        {
          txtScrollValue.Text = (sender as
                ScrollBar).Value.ToString();
        }
    }
}
```

图6-27示出了这段程序的运行结果，可以看到滚动条的活动范围为5在100 之间，当用鼠标单击滚动条的左右滚动键时，滚动条的值相应发生变化，每次变化的数值为2（SmallChange=2）。

图6-27 使用滚动条

6.5.2 滑动条（Slider）

滑动条是一个边上带有刻度的范围控件，在某些情况下使用滑动条很直观，如多媒体播放器，用户可以直接通过拉动滑动条上的滑块来选择要看的视频位置。

刻度的位置由TickPlacement属性确定，其类型为TickPlacement，是一个枚举类型。它可以取两个值：TopLeft和BottomRight。当滑动条是水平放置时，若TickPlacement为TopLeft，这是刻度位于滑动条的上方；若TickPlacement为BottomRight时，刻度位于滑动条的下方。当滑动条为垂直放置，TickPlacement控制刻度位于滑动条的左边或右边。设定滑动条的刻度有两种方法，方法之一是设置滑动条的Ticks属性，这是一个浮点数集合，滑动条会在你设定的浮点数的位置显示刻度。方法之二是设置TickFrequency,滑动条会在Maximum和Mimimum值之间，根据TickFrequency自动设置相应的刻度。例如，滑动条的最大值为100，最小值为0，TickFrequency为10，那么，活动条刻度的值会自动

设为0，10，20，……

有时需要设置滑动条的正常选择范围，例如变压器的输出电压，对于110kV的电压，可规定其在上下5%范围内波动是合理的，超过这一范围，就要发出警告。若用滑动条控件来调电压，就要设置调整电压的正常选择范围。滑动条中有三个属性与此相关：IsSelectRangeEnabled、SelectionStart和SelectionEnd。当设置IsSelectRangeEnabled为True，并设置SelectionStart和SelectionEnd的值后，滑动条上就会出现选择范围的标识。需要注意的是SelectionStart应小于SelectionEnd；并且这两个值应在Maximum和Minimum之间。

滑动条还可以在鼠标单击滑块时，用ToolTip的方式显示当前滑块的值。这时要用到AutoToolTipPlacement和AutoToolTipPrecision两个属性。AutoToolTipPlacement是一个枚举类型，可以取TopLeft和BottomRight两个值，其意义和TickPlacement相同。AutoToolTipPrecision用来定义显示的精度，其值为小数点位数。下面的例子可以用来考察滑动块的常用属性及其意义：

```
<Window x:Class="Yingbao.Chapter6.UseSlider"
   xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Yingbao.Chapter6" Height="300" Width="600">
    <Grid>
      <Grid.ColumnDefinitions>
        <ColumnDefinition />
        <ColumnDefinition />
      </Grid.ColumnDefinitions>
      <Grid.RowDefinitions >
        <RowDefinition />
        <RowDefinition />
        <RowDefinition />
        <RowDefinition />
        <RowDefinition />
      </Grid.RowDefinitions>
      <StackPanel  Grid.Column ="0" Grid.ColumnSpan ="2" Grid.Row
              ="4" Orientation ="Horizontal " >
        <Label>滑动条：</Label>
        <Slider Name="slider1" Width ="300" ValueChanged
          ="OnChangeValue" Orientation="Horizontal"
          HorizontalAlignment="Left"  Minimum="0" Maximum="100"
          Value ="50" IsSelectionRangeEnabled="true"
          IsDirectionReversed="False"/>
        <TextBox Name ="sliderValue" Width ="60" Height ="30"
                 Margin ="10,0,0,0"> 0</TextBox >
      </StackPanel>
      <StackPanel Grid.Column ="0" Grid.Row ="0"
            Orientation ="Horizontal ">
        <Label>AutoToolTipPlacement </Label>
        <ComboBox  Width ="100" Margin ="0,10,0,10" SelectedIndex
              ="1" SelectionChanged ="OnSelectAutoToolTipPlacement">
          <ComboBoxItem>TopLeft</ComboBoxItem>
          <ComboBoxItem>BottomRight</ComboBoxItem>
        </ComboBox>
      </StackPanel>
```

```
    <StackPanel Grid.Column ="1" Grid.Row ="0" Orientation
          ="Horizontal ">
      <Label>AutoToolTipPrecision: </Label>
      <TextBox  Width="60" Height ="30" TextChanged
                 ="OnSelectAutoToolTipPrecesion" Text ="2"/>
    </StackPanel>
    <StackPanel Grid.Column ="0" Grid.Row ="1" Orientation
               ="Horizontal">
      <Label>TickFrequency: </Label>
      <TextBox Width="60" Height ="30" TextChanged
           ="OnTickFrequencyChange" Text ="5.0"/>
    </StackPanel>
    <StackPanel Grid.Column ="1" Grid.Row ="1"
          Orientation ="Horizontal ">
      <Label>TickPlacement: </Label>
      <ComboBox Width="100" Height ="30" SelectedIndex ="0"
           SelectionChanged  ="OnSelectTickPlacement" >
        <ComboBoxItem >TopLeft</ComboBoxItem>
        <ComboBoxItem >BottomRight</ComboBoxItem>
      </ComboBox >
    </StackPanel>
    <StackPanel Grid.Column ="0" Grid.Row ="2"
        Orientation ="Horizontal ">
      <Label>IsSnapToTickEnabled: </Label>
      <ComboBox  Width="100" Height ="30" SelectedIndex ="0"
              SelectionChanged="OnSelectSnapToTick" >
        <ComboBoxItem >True</ComboBoxItem>
        <ComboBoxItem >False</ComboBoxItem>
      </ComboBox >
    </StackPanel>
    <StackPanel Grid.Column ="1" Grid.Row ="2"
          Orientation ="Horizontal ">
      <Label>IsMoveToPointEnabled: </Label>
      <ComboBox Width="100" Height ="30" SelectedIndex ="0"
                SelectionChanged  ="OnMoveToPointEnabled" >
        <ComboBoxItem >True</ComboBoxItem>
        <ComboBoxItem >False</ComboBoxItem>
      </ComboBox >
    </StackPanel>
    <StackPanel Grid.Column ="0" Grid.Row ="3"
     Orientation ="Horizontal ">
      <Label>SelectionStart: </Label>
      <TextBox   Width="60" Height ="30"
           TextChanged ="OnSelectionStartChange" Text ="10.0"/>
    </StackPanel>
    <StackPanel Grid.Column ="1" Grid.Row ="3"
     Orientation ="Horizontal ">
      <Label>SelectionEnd: </Label>
      <TextBox   Width="60" Height ="30"
           TextChanged ="OnSelectionEndChange" Text ="90.0"/>
    </StackPanel>
  </Grid>
</Window>
```

相关属性改变时的后台处理程序为：

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Controls.Primitives;
using System.Windows.Data;
using System.Windows.Documents;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Media.Imaging;
using System.Windows.Shapes;

namespace Yingbao.Chapter6
{
   public partial class UseSlider : System.Windows.Window
    {
        public UseSlider()
        {
            InitializeComponent();

        }

       private void OnChangeValue(object sender, RoutedEventArgs rea)
        {
            if( sliderValue !=null )
            sliderValue.Text = slider1.Value.ToString();
        }

       private void OnSelectAutoToolTipPlacement(object sender,
                 RoutedEventArgs rea)
        {
         string selectValue = GetComboBoxValue(sender);
         slider1.AutoToolTipPlacement = (AutoToolTipPlacement)
            Enum.Parse(typeof(AutoToolTipPlacement), selectValue);
        }

    private void OnSelectAutoToolTipPrecesion(object sender,
                    RoutedEventArgs rea)
        {
          if( (sender as TextBox ).Text.Length >0 )

           slider1.AutoToolTipPrecision = Convert.ToInt32( (sender
           as TextBox ).Text);
        }

    private void OnTickFrequencyChange(object sender,
                    RoutedEventArgs rea)
        {
            if ((sender as TextBox).Text.Length > 0)
            slider1.TickFrequency = Convert.ToDouble((sender
                as TextBox).Text);
        }
```

```
    private void OnSelectTickPlacement(object sender,
                RoutedEventArgs rea)
        {
            string selectValue = GetComboBoxValue(sender);
            slider1.TickPlacement  = (TickPlacement )
                Enum.Parse(typeof(TickPlacement), selectValue);
         }

    private void OnSelectSnapToTick(object sender,
                RoutedEventArgs rea)
        {
            string selectedValue = GetComboBoxValue(sender);
            slider1.IsSnapToTickEnabled =
                     Convert.ToBoolean(selectedValue);
        }

    private void OnMoveToPointEnabled(object sender,
                RoutedEventArgs rea)
        {
            string selectedValue = GetComboBoxValue(sender);
            slider1.IsMoveToPointEnabled=
                Convert.ToBoolean(selectedValue);
         }

    private void OnSelectionStartChange(object sender,
              RoutedEventArgs rea)
        {
            if ((sender as TextBox).Text.Length > 0)
           slider1.SelectionStart  = Convert.ToDouble((sender
               as TextBox).Text);
        }

    private void OnSelectionEndChange(object sender,
                     RoutedEventArgs rea)
        {
            if ((sender as TextBox).Text.Length > 0)
           slider1.SelectionEnd  = Convert.ToDouble((sender as
                   TextBox).Text);
        }

        private string GetComboBoxValue(object obj)
        {
          string selectValue = ((obj as ComboBox).SelectedItem as
             ComboBoxItem).Content.ToString();
          return selectValue;
        }
    }
}
```

使用上面这段程序可以观察滑动条属性的变化所引起的滑动条行为的变化（如图6-28所示）。

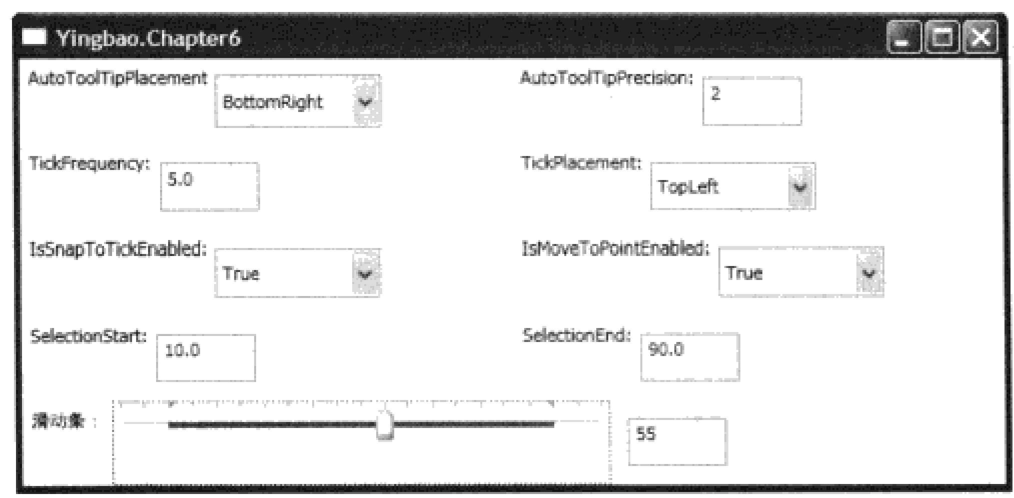


图6-28 滑动条实例

6.5.3 进展条（ProgressBar）

当某个任务需要较长时间才能完成时，需要在用户界面上显示一个任务进行到什么程度的控件，否则，若什么都不显示，用户会认为程序不能运行了。这个控件就是进展条（ProgressBar）。

进展条的默认Minimum值为0，默认Maximum的值为100。使用进展条通常要用到动画。若不想使用动画，而又不在乎什么时候任务会结束，可以把IsIndeterminate属性设为true，直到任务执行完时，再把它设为false。这样，你会看到进展条中的小矩形一直在动。下面的例子示出了这种简单的进展条：

```
<Window x:Class="Yingbao.Chapter6.UseProgressBar"
   xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Yingbao.Chapter6" Height="100" Width="300">
  <StackPanel >
    <ProgressBar Width="200" Height="20" IsIndeterminate ="True"
                 Name="progressBar1"/>
  </StackPanel>
</Window>
```

6.6 本章小结

本章详细地介绍了WPF中的控件，从功能的角度，这些控件可以分为四大类：内容控件、条目控件、文本控件和范围控件，并举例说明了各种控件的使用方法。控件是构建UI的重要内容，WPF围绕控件开发了一系列新技术：WPF中独特的传递事件系统和命令、数据绑定技术、控件模板、风格以及控件的动画，后面的章节将对这些技术进行探讨。

第7章 传递事件和传递命令系统

WPF中含有一个新的事件子系统—— 传递事件系统（Routed Event System）。传递事件是为WPF中的元素树设计的，当发生传递事件时，事件可以在WPF中的视觉树和逻辑树的元素间用一种简单的方式传递，而不必用户写任何代码。

本章将系统地介绍WPF中的传递事件系统、讨论鼠标和键盘输入事件在WPF元素中的传递，以及如何在程序中定义并移植传递事件，最后还将讨论WPF中的传递命令。

7.1 WPF中的元素树

众所周知XAML是以XML语言为基础、描述WPF人机界面的一种语言。XML是建立在DOM（Document of Model）之上的，而DOM实际上是一棵倒挂的树。所以，自然地，用XAML描述的WPF界面也有这样一棵倒挂的树。在桌面应用程序中，其树根通常是Window元素；在互联网应用程序中，其树根通常是Page元素。

若使用C#，Visual Basic，Managed C++或其他.NET语言来创建WPF应用程序，这种元素树同样存在。在人机界面上布置控件、图形元素以及排版元素时，元素树是自然而然形成的。

WPF中的元素树根据元素的性质可以分成两种：一种是视觉树（Visual Tree）；另一种是逻辑树（Logical Tree）。

视觉树由在界面上可见的元素构成，这些元素是从Visual 或Visual3D类中派生出来的类。逻辑树描述界面元素的实际结构，它通常由程序在XAML中所用的元素组成，不包括WPF为了在界面上表现逻辑元素所创建的一些附加视觉元素。这些附加视觉元素对一般的程序员来说，可能并不在意或根本不知道它们的存在。与视觉树仅由可见对象构成不同，逻辑树可以包括任何对象。

WPF使用元素树来实现WPF中的某些重要的功能，这些功能除了我在第4章WPF中的属性系统讨论的相关属性的传递之外，还有输入事件的传递及资源的定位等。所以，理解元素树对于理解WPF非常重要。

WPF提供了两个类来帮助我们遍历视觉树和逻辑树：VisualTreeHelper和LogicalTreeHelper。

LogicalTreeHelper中含有GetChildren、GetParent和FindLogicalNode等静态方法。

GetChildren返回IEnumrable接口，使用这个接口中的方法可以遍历当前逻辑节点上的子节点。实际上，LogicalTreeHelper提供了三个GetChildren方法的重载，其参数分别为DependencyObject、FrameworkElement 和FrameworkContentElement类型。由于FrameworkElement和FrameworkContentElement都是从DependencyObject中派生出来的，所以，若使用GetChildren（DependencyObject current），应该可以包含其他两种情况。WPF提供这个函数的其他两个重载，可能是基于实时性能的考虑。

由于在树形图中，每个节点只能有一个父节点，所以GetParent方法返回指向父节点的引用，其类型为DependencyObject。

FindLogicalNode方法可以根据逻辑节点的名字，来找到逻辑树中的某个节点。而在某个逻辑树

中，各个节点上的元素名字必须是唯一的，所以FindLogicalNode方法返回指向所找到的节点的引用，在未找到某个逻辑节点时返回 null。FindLogicalNode需要用户提供起始节点。

VisualTreeHelper类中提供了更多的方法，但对于遍历视觉树来说，我们关心的仍然是如何从当前的节点获取父节点或子节点的引用。有意思的是，VisualTreeHelper里遍历子节点所用的方法和逻辑树中有所不同，在逻辑树中，可以用下面的C#代码来遍历某个节点的子节点：

```
foreach( object logicalChild in
            LogicalTreeHelper.GetChildren( current)
{

  ...  //做你要做的工作
}
```

VisualTreeHelper提供的是GetChild和GetChildrenCount两个方法，所以，要遍历视觉树上某个节点的子节点，需要用下面的代码：

```
int  childNodesNum = VisualTreeHelper.GetChildrenCount( current);
for( int i =0; i< childNodesNum; i++)
{
     DependencyObject dObj = VisualTreeHelper.GetChild(current, i);
     ...//做你要做的工作
}
```

微软为什么不提供一致的遍历逻辑树和视觉树上子节点的方法呢？笔者认为这是一个缺陷，可能是不同的程序员写的吧？

VisualTreeHelper中所提供的GetParent方法返回指向父节点的引用。

需要指出的是：对于一般程序员来说，我们总是可以使用WPF中内容模型来获取某个元素的逻辑子或父节点。如Window对象中的Content就是Window的唯一子节点，而ListBox中的Items则允许你加入多个子节点。

视觉树由于内涵元素（Content Element）的加入而复杂起来，所谓内涵元素，是指WPF那些从ContentElement类中派生出来的元素。虽然内涵元素通过视觉元素在计算机屏幕上显示，但内涵元素本身并不在视觉树上，它们不是Visual或Visual3D的派生类。为了让传递事件在视觉树上按照同一种机制传播，WPF 通常把内涵元素用另一个视觉元素在用户界面上展现出来。这个视觉元素叫做内涵元素的宿主元素。可以把宿主元素和内涵元素之间的关系想象为互联网的浏览器和其中所显示的网页内容间的关系。当存在内涵元素的时候，视觉树不再是连续的，即视觉树即使在一个窗口中，也不是一棵，可能会存在多个视觉树。所以在遍历视觉树时，需要考虑这一情况：

```
private DependencyObject FindVisualTreeRoot(DependencyObject initial)
        {
            DependencyObject current = initial;
            DependencyObject result = initial;

            while (current != null)
            {
                result = current;
```

```
            if (current is Visual || current is Visual3D)
            {
                current = VisualTreeHelper.GetParent(current);
            }
            else
            {
                current = LogicalTreeHelper.GetParent(current);
            }
        }
        return result;
    }
```

上面的程序首先判断当前节点的元素是否是可视元素（Visual或Visual3D），若是，则使用VisualTreeHelper来得到视觉树的父节点；若不是，则使用LogicalTreeHelper来获取父节点。这时可能跨过了不同的视觉树。比如第6章WPF控件中使用Scroll View控件的例子，笔者在显示《七点钟》一诗时，用过Bold、Italic等元素，这些元素就不是可视元素。

在一个视窗Window或Page下，WPF可能创建多个视觉树；在同一个视窗Window或Page下，WPF也会创建多个逻辑树。对于某一个控件，往往只有一个视觉树，但却可以有多个逻辑树，这些逻辑树可以互不相关；所以，若只是用LogicalTreeHelper来遍历逻辑树，并不能保证能找出所有的逻辑树。由于WPF中的控件通过控件模板来显示其中的内容，控件和显示控件的模板间是相互独立的，从而视觉树的构成会非常复杂。控件模板的问题将在第10章加以讨论，现在，只需要认识到WPF中视觉树的复杂性就可以了。若需要从一个逻辑树跳到另一逻辑树时，则需要通过下列FrameworkElement或FrameworkContentElement的TempaletedParent属性：

```
public DependencyObject GetTemplatedParent(DependencyObject depObj)
    {
        FrameworkElement fe = depObj as FrameworkElement;
        FrameworkContentElement fce = depObj as
              FrameworkContentElement;
        DependencyObject result;
        if (fe != null)
            result = fe.TemplatedParent;
        else if (fce != null)
            result = fce.TemplatedParent;
        else
            result = null;

        return result;
    }
```

为了让读者对WPF中的元素树有一个直观的了解，笔者列举了一个显示逻辑树和视觉树的例子，这个例子可以显示任何控件的相关逻辑树或视觉树，它可以帮助我们直观地认识WPF中的元素树。

```
#region namespace
using System;
using System.Collections.Generic;
using System.Text;
using System.Windows;
```

```
using System.Windows.Controls;
using System.Windows.Data;
using System.Windows.Documents;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Media.Imaging;
using System.Windows.Shapes;
using System.Windows.Media.Media3D;

#endregion

namespace Yingbao.Chapter7
{
    public class DisplayTreeWindow : System.Windows.Window
    {
        TreeView tree = new TreeView() ;
        TextBlock tb1;
        TextBlock tb2;

        DependencyObject closestAncestor;

        public DisplayTreeWindow(): base()
        {
            ScrollViewer sv = new ScrollViewer();
            this.Content = sv;

            StackPanel sp = new StackPanel();
            sv.Content  = sp;
            sp.Children.Add(tree);
            tb1 = new TextBlock();
            tb1.Width = 400;
            tb1.Height = 200;
            sp.Children.Add(tb1);
            tb2 = new TextBlock();
            tb2.Width = 400;
            tb2.Height = 200;
            sp.Children.Add(tb2);

        }


      public void CreateLogicalTree( DependencyObject originalElement)
        {
            tree.Items.Clear();
            StringBuilder sb = new StringBuilder();
            closestAncestor =
                  FindClosestLogicalAncestor(originalElement);
            sb.Append("最近的逻辑树上的父节点：")
              .Append(closestAncestor.GetType().Name);
            tb1.Text = sb.ToString();
            sb.Remove(0, tb1.Text.Length);
            sb.Append( "你选择了：" )
              .Append (originalElement.GetType().Name);
```

```
            tb2.Text = sb.ToString();
            DependencyObject rootElement =
               FindLogicalTreeRoot(closestAncestor);
            PopulateLogicalTree(null, rootElement);
            this.Title = "显示逻辑树";
        }

    public void CreateVisualTree( DependencyObject originalElement)
        {
            DependencyObject closestVisualAncestor =
                 FindClosestVisualAncestor(originalElement);
            DependencyObject visualRoot =
                 FindVisualTreeRoot(originalElement);
            StringBuilder sb = new StringBuilder();
           string closestParen =  closestVisualAncestor ==

            originalElement ? "(self)" : closestVisualAncestor
                   .GetType().Name;
            sb.Append("最近视觉树上的父节点: ").Append( closestParen );
            sb.AppendLine();
            sb.Append("最近视觉树上父节点上的模板: ");
            DependencyObject templatedParent =
                GetTemplatedParent(closestVisualAncestor);
            string strTemplatedParent =
               templatedParent == null ? "(null)" :
               templatedParent.GetType().Name;
            sb.Append(strTemplatedParent);
            tb1.Text = sb.ToString();
            sb.Remove(0, tb1.Text.Length);
            sb.Append("你选择了: ").Append(originalElement
                    .GetType().Name);
            tb2.Text = sb.ToString();
            PopulateVisualTree(null, visualRoot);
            this.Title = "显示视觉树";

        }

         private void PopulateLogicalTree(TreeViewItem parentItem,
                      object currentElement)
        {
            TreeViewItem item = new TreeViewItem();
            item.IsExpanded = true;
            if (parentItem == null)
            {
                item.Header = currentElement.GetType().Name;
                tree.Items.Add(item);

            }
            else
            {
                item.Header = currentElement.GetType().Name;
```

```
            parentItem.Items.Add(item);
        }

        DependencyObject curObject = currentElement as
                DependencyObject ;
        if (curObject != null)
        {
            foreach (object chElement in
                    LogicalTreeHelper.GetChildren(curObject))
            {
                PopulateLogicalTree(item, chElement);
            }
        }
    }

    private  void PopulateVisualTree(TreeViewItem parentItem,
                DependencyObject currentElement )
    {
        TreeViewItem item = new TreeViewItem();
        item.IsExpanded = true;
        if (parentItem == null)
        {
            item.Header = currentElement.GetType().Name;
            tree.Items.Add(item);
        }
        else
        {
            item.Header = currentElement.GetType().Name;
            parentItem.Items.Add(item);
        }
        int visualChildrenCount = VisualTreeHelper
                .GetChildrenCount(currentElement);

        for (int i = 0; i < visualChildrenCount; ++i)
        {
            DependencyObject visualChild = VisualTreeHelper
                .GetChild(currentElement, i);
            PopulateVisualTree(item, visualChild);

        }
    }

    DependencyObject FindClosestLogicalAncestor(
                        DependencyObject initial)
    {
        DependencyObject current = initial;
        DependencyObject result = initial;

        while (current != null)
        {
            DependencyObject logicalParent = LogicalTreeHelper
                            .GetParent(current);
            if (logicalParent != null)
```

```
            {
                result = current;
                break;
            }
            current = VisualTreeHelper.GetParent(current);
        }

        return result;
    }

    DependencyObject FindLogicalTreeRoot(
            DependencyObject initial)
    {
        DependencyObject current = initial;
        DependencyObject result = initial;

        while (current != null)
        {
            result = current;
            current = LogicalTreeHelper.GetParent(current);
        }

        return result;
    }

    public DependencyObject GetTemplatedParent(
                    DependencyObject depObj)
    {
        FrameworkElement fe = depObj as FrameworkElement;
        FrameworkContentElement fce = depObj as
                            FrameworkContentElement;
        DependencyObject result;
        if (fe != null)
            result = fe.TemplatedParent;
        else if (fce != null)
            result = fce.TemplatedParent;
        else
            result = null;

        return result;
    }

    DependencyObject FindClosestVisualAncestor(
            DependencyObject initial)
    {
        if (initial is Visual || initial is Visual3D)
            return initial;
        DependencyObject current = initial;
        DependencyObject result = initial;
        while (current != null)
        {
            result = current;
            if (current is Visual || current is Visual3D)
```

```
                {
                    result = current;
                    break;
                }
                else
                {

                    current = LogicalTreeHelper.GetParent(current);
                }
            }
            return result;
        }

        private DependencyObject FindVisualTreeRoot(
                DependencyObject initial)
        {
            DependencyObject current = initial;
            DependencyObject result = initial;

            while (current != null)
            {
                result = current;
                if (current is Visual || current is Visual3D)
                {
                    current = VisualTreeHelper.GetParent(current);
                }
                else
                {

                    current = LogicalTreeHelper.GetParent(current);
                }
            }
            return result;
        }
    }
}
```

测试视觉树和逻辑树的应用程序如下：

```
<Window x:Class="Yingbao.Chapter7.TestElementTreeWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="视觉树和逻辑树" Height="416" Width="533">
  <Grid>
    <Grid.ColumnDefinitions >
      <ColumnDefinition Width ="*"/>
      <ColumnDefinition Width ="*"/>
    </Grid.ColumnDefinitions>
    <Grid.RowDefinitions>
      <RowDefinition  Height ="*"/>
      <RowDefinition  Height ="*"/>
    </Grid.RowDefinitions>
    <StackPanel Name ="LogicalPanel" Grid.Row ="0" Grid.Column ="0" >
```

```
      <TextBlock Margin="4" Background="LightBlue">
         按下"Ctrl"键，然后使用鼠标左键选择UI元素
      </TextBlock>
      <ListBox>
        <ListBoxItem >
          <CheckBox IsChecked ="true">旅游</CheckBox>
        </ListBoxItem>
        <ListBoxItem >
          <CheckBox >读书</CheckBox>
        </ListBoxItem>
        <ListBoxItem >
          <CheckBox >写作</CheckBox>
        </ListBoxItem>
      </ListBox>
      <TextBox> Test1</TextBox>
    </StackPanel >
    <StackPanel  Grid.Row="0"  Grid.Column ="1" >
      <ListBox Margin="4" xmlns:sys="clr-
            namespace:System;assembly=mscorlib">
        <sys:Int32>1</sys:Int32>
        <sys:Double>2.22</sys:Double>
        <sys:String>Three</sys:String>
        <Border BorderBrush="Black" BorderThickness="1">
          <CheckBox Padding="0,0,8,0">
            <TextBlock Background="LightBlue">
             位于ListBox中的元素
            </TextBlock>
          </CheckBox>
        </Border>
      </ListBox>
      <Button> 按钮1 </Button>
      <CheckBox>选择框</CheckBox>
      <RadioButton>Radio Button</RadioButton>
    </StackPanel>
    <StackPanel Grid.Row="1"  Grid.Column ="0"  >
    <Button FontSize ="14" Click ="OnDisplayLogicalTree">显示逻辑树
         </Button>
    <Button FontSize ="14" Click ="OnDisplayVisualTree">显示视觉树
         </Button>
    </StackPanel>
    <StackPanel Grid.Row="1"  Grid.Column ="1"  >
<TextBox Name ="DisplaySelectElement"
          Background ="AliceBlue" Foreground ="Red" FontSize ="16" />
      <TextBox Name ="DisplaySelectElementTemplateParent"
          Background ="AliceBlue" Foreground ="Red" FontSize ="16" />
    </StackPanel>
  </Grid>
</Window>
```

C#处理程序为：

```
using System;
using System.Collections.Generic;
using System.Text;
```

```
using System.Windows;
using System.Windows.Controls;
using System.Windows.Data;
using System.Windows.Documents;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Media.Imaging;
using System.Windows.Shapes;

namespace Yingbao.Chapter7
{
    public partial class TestElementTreeWindow :
                System.Windows.Window
    {
        DependencyObject  selectElement = null;
        public TestElementTreeWindow()
        {
            InitializeComponent();
            this.PreviewMouseLeftButtonDown += new
           MouseButtonEventHandler(HandlePreviewMouseLeftButtonDown);
        }

        void OnDisplayLogicalTree(object sender, RoutedEventArgs ea)
        {
            if (selectElement != null)
            {
                DisplayTreeWindow createWin = new
                        DisplayTreeWindow();
                createWin.CreateLogicalTree(selectElement);
                createWin.Show();
            }
        }

        void OnDisplayVisualTree( object sender, RoutedEventArgs ea)
        {
            if (selectElement != null)
            {
                DisplayTreeWindow createWin = new
                      DisplayTreeWindow();
                createWin.CreateVisualTree(selectElement);
                createWin.Show();
            }
        }

         void HandlePreviewMouseLeftButtonDown(object sender,
                  MouseButtonEventArgs e)
        {
            if (Keyboard.Modifiers == ModifierKeys.Control)
            {
                selectElement = e.OriginalSource as DependencyObject;
                DisplaySelectElement.Text =
                    " 你选择的控件: "+selectElement.GetType().Name;
                DependencyObject dobj =
```

```
                        GetTemplatedParent(selectElement);
                if (dobj != null)
                {
                    DisplaySelectElementTemplateParent.Text =
                        "你选择的控件模板: "+dobj.GetType().Name;
                }
                else
                {
                    DisplaySelectElementTemplateParent.Text = "";
                }
                 e.Handled = true;
            }
        }


  public DependencyObject GetTemplatedParent(DependencyObject depObj)
        {
            FrameworkElement fe = depObj as FrameworkElement;
            FrameworkContentElement fce = depObj as
                      FrameworkContentElement;
            DependencyObject result;
            if (fe != null)
                result = fe.TemplatedParent;
            else if (fce != null)
                result = fce.TemplatedParent;
            else
                result = null;
            return result;
        }
    }
}
```

当运行上面这段程序时，需要先选择一个控件。方法是按下“Ctrl”键，同时用鼠标左键单击一个控件。这时，事件处理程序HandlePreviewMouseLeftButtonDown被调用。设置HandlePreviewMouseLeftButtonDown处理程序是在TestElementTreeWindow类中进行的，TestElementTreeWindow是Window类的派生类，而真正发出事件的是该窗口类中的一些子控件，为什么父控件可以接收到子控件的消息？这就是7.2节要讨论的重点。在HandlePreviewMouseLeftButtonDown事件处理程序中，在DisplaySelectElement中显示所选择的控件，在DisplaySelectElementTemplateParent中显示该控件所使用的模板。

在选择了控件之后，就可以观察到该控件所处的视觉树及逻辑树。方法是单击“显示逻辑树”或“显示视觉树”按键，如图7-1所示。

图7-1 用于测试视觉树或逻辑树的对话框

图7-2和图7-3分别显示了所选控件的逻辑树和视觉树。

利用这个程序，可以观察到WPF中构建控件时用的各种元素，其中有许多视觉元素是WPF为在UI上表现某个控件自己加上去的。理解这一点，对于开发自己的控件非常重要，有关用户控件等相关技术在第17章定制控件和排版中加以讨论。

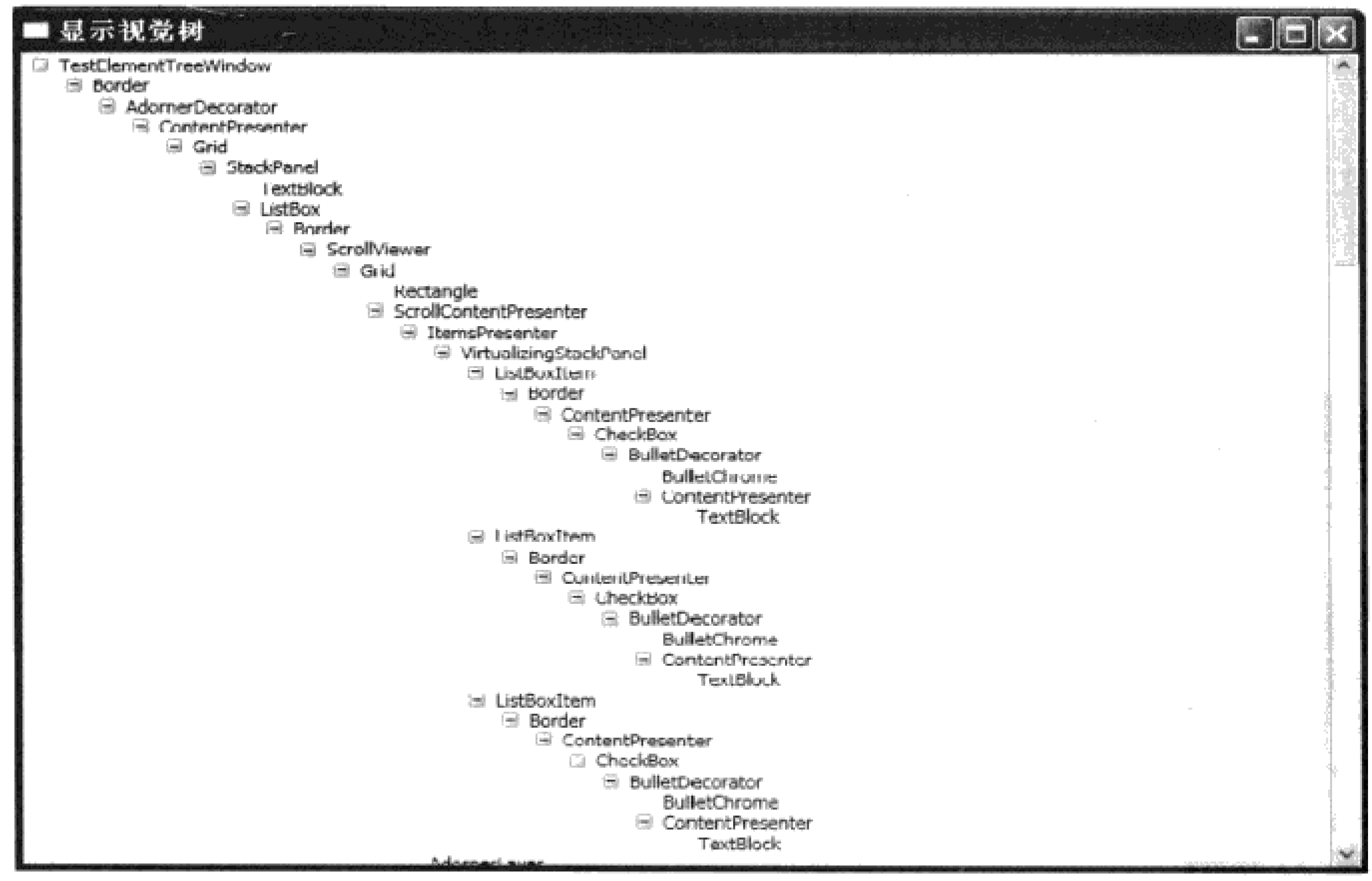


图7-2 当选择ListBox中的选择框时的视觉树

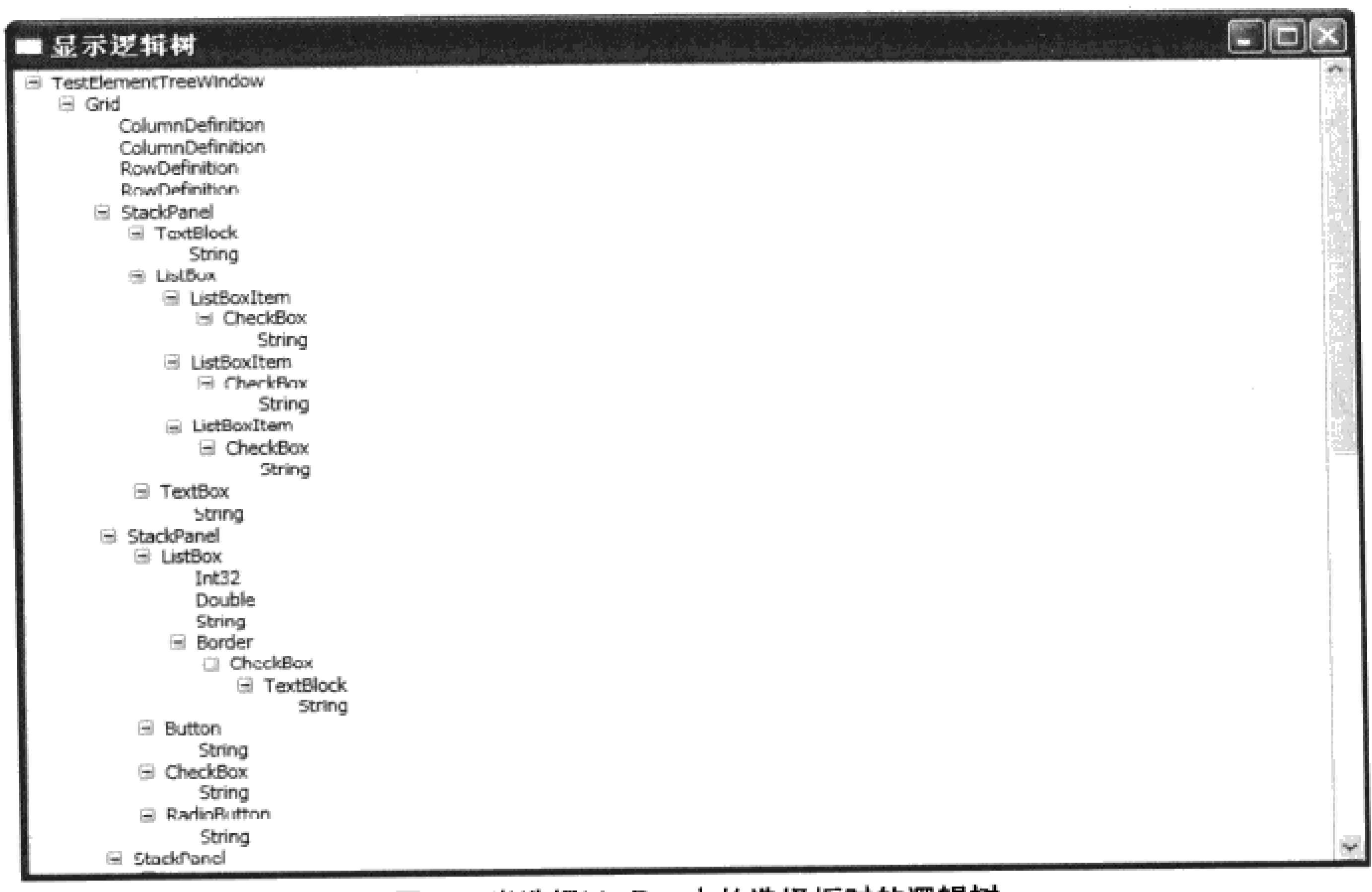


图7-3 当选择ListBox中的选择框时的逻辑树

7.2 传递事件（Routed Event）

7.1节讨论了WPF中的元素树，为了支持元素树，WPF开发了一个传递事件子系统。该子系统实际上是WPF接管了常规的Windows操作系统事件之后，把Windows事件变成WPF事件，并且让WPF事件在WPF元素树的节点上传播。这些事件基本上是在视觉树上传递的，有的时候也会跨过不同的视觉树。

和Windows操作系统一样，事件的产生往往是在具有输入焦点的控件上。在事件产生后，事件沿着元素树既可以向其包容类（父类）传播，也可以向其子类传播。这和在吉他上拨动琴弦时振动在琴弦上传播很类似，振动产生于手指的拨动处，然后在琴弦上上下传播，但这种振动只能在所拨动的琴弦上传播，而不会传播到其他的琴弦上。WPF中的事件也是这样，它可以沿着元素树向树根或树枝的元素上传播，但它不会传播到和它并列在同一个节点的分支上。传递事件的这两种传播方式有专门的名字，事件向树根的传递叫做冒泡（Bubbling），事件向树枝传递叫潜入（Tunneling）。

当用鼠标左键单击一个按键（Button）时，哪个元素最先得到了鼠标事件？过去可以非常明确地知道一定是按键接到了鼠标事件。但在WPF中，并不能够这么明确，因为按键（Button）是一个内容控件，和窗口类Window一样，其中可以放置任意的控件组合。比如说在Button中加入面板、在面板中加入各种图形元素，等等。究竟是哪个元素首先得到了鼠标事件？如果WPF中没有传递事件系统，那么Button类还会得到用户单击鼠标事件吗？

当你按下键盘上的键时，哪个元素得到了键盘输入事件？过去是具有输入焦点的控件获得键盘输入事件，但在WPF中，由于有输入焦点的元素可能不是我们真正想要获得键盘输入焦点的元素，比如说，菜单中的快捷键，那么当用户输入快捷键时，如果菜单处理程序获得键盘输入岂不是更好？

所有这一切，都需要WPF中有一个新的方式来管理用户输入事件。

7.2.1 RoutedEventArgs

处理常规.NET事件的委托函数一般有两个参数，一个是事件产生的对象，另一个是该事件所带的参数。例如，处理打开文件的菜单处理程序一般具有下面的形式：

```
void OnOpenFile (object sender, EventArgs ea )
{
    //处理该菜单
}
```

这里参数sender是object类型，即它可以是任何.NET类；EventArgs则是所有事件参数的基类。.NET程序员可以根据事件本身的要求在事件参数中加入自己的参数， 通常的做法是从EventArgs或其派生类中派生出自己的类。

传递事件也是采用这种处理方式，例如7.1节，笔者在处理按钮Click事件时用的处理函数：

```
void OnDisplayLogicalTree(object sender, RoutedEventArgs ea)
  {
       if (selectElement != null)
       {
            DisplayTreeWindow createWin = new
                     DisplayTreeWindow();
            createWin.CreateLogicalTree(selectElement);
            createWin.Show();
       }
  }
```

DisplayLogicalTree中也有两个参数，一个是sender，类型为object；另一个是ea，类型为RoutedEventArgs。有些传递事件直接使用RoutedEventArgs类，有些事件则要使用RoutedEventArgs的派生类。EventArgs是RoutedEventArgs的基类，表7-1示出了RoutedEventArgs类中的属性：

表7-1 RoutedEventArgs中的属性

属 性	意 义
Handled	True/false表示该事件是否已经处理, 可以用来控制是否进一步传播传递事件
OriginalSource	用来说明最初产生该事件的对象
RoutedEvent	传递事件实例
Source	发出该事件的对象

WPF根据传递事件的种类，从RoutedEventArgs中大概派生出了50个类，而且这些类的数目还在快速增长。例如处理键盘事件有KeyBoardEventArgs、处理鼠标事件有MouseEventArgs等，也可以定义自己的RoutedEventArgs。

7.2.2 终止事件传播

传递事件在元素树中传播，使得所有元素树中的元素都有机会对传递事件进行响应，然而，这种做法是以实时性能为代价的。有时候在对传递事件处理了之后，并不希望传递事件在元素中继续传播，即终止传递事件的传播。有时候则希望截获某个传递事件之后，根据自己的情况，产生一个新的传递事件，这时可能没有必要再传递原来的传递事件了。典型的例子如按钮，在截获了鼠标的

PreviewMouseButtonDown，PreviewMouseButtonUp事件之后产生了一个新的Click事件，这个时候若再让原来事件在元素树中传播，显然是对计算机资源的一种浪费。

RoutedEventArgs中的Handled属性就是为此设计的，如果把该属性设为True的话，那么WPF中的传递事件系统就自动终止了该传递事件的传播。例如：

```
void OnDisplayLogicalTree(object sender, RoutedEventArgs ea)
    {
          //…
          ea.Handled =  true;
    }
```

前面提到WPF中的传递事件是以冒泡和潜入这两种方式传播的，潜入总是在冒泡之前传播；所以，在某个类中处理传递事件的委托函数通常也是成对出现的，如处理鼠标按钮有MouseDown和PreviewMouseDown。作为命名规则，WPF潜入事件的委托函数，总是在前面加上“Preview”。若我们在处理潜入事件时，把RoutedEventArgs参数的Handled属性设为True，则WPF不仅会终止潜入事件的传播，也会同时终止冒泡事件的传播。

7.2.3 处理传递事件

WPF程序员需要对感兴趣的传递事件进行处理，对于键盘、鼠标这些输入设备所产生的输入事件，UIElement类中定义了大部分事件处理程序。WPF调用事件处理程序有两种方法：一种是利用虚函数的覆盖机制，另一种是利用.NET的委托函数把自己的处理函数连接到传递事件上。

- 虚函数覆盖方法

若类是从WPF类中派生出来的，这时可以通过虚函数覆盖来让WPF调用其事件处理方法。例如：

```
protected override void OnMouseDown(MouseEventArgs mea)
 {
    base.OnMouseDown(mea);
    ….
 }
```

其中MouseEventArgs是InputeEventArgs的派生类，而InputEventArgs则是以RoutedEventArgs为基类。

与过去基类中的虚函数通常带有事件处理代码不同（常称为默认移植），WPF基类中事件处理程序的虚函数通常并不提供默认移植，即基类中可被覆盖的虚函数常常是空的，这时调用base.OnMouseDown( )并不重要，但有时候WPF的基类确实提供了默认移植，若忘了调用基类函数，可能会有不良的结果，所以通常还是调用基类中被覆盖的函数。

- 使用委托函数

前面的例子中，基本上用的都是这种方法，可以在XAML中接入事件处理程序：

```
<Button  Click ="OnDisplayLogicalTree">显示逻辑树</Button>
```

这里Click= “….” 就是接入事件处理程序。

第二种方法是在C#或其他.NET语言中直接接入传递事件。例如，在7.1节的例子中，笔者在接入PreviewMouseLeftButtonDown事件时用了C#语句：

```
this.PreviewMouseLeftButtonDown += new
          MouseButtonEventHandler(HandlePreviewMouseLeftButtonDown);
```

7.2.4 附加传递事件（Attached Routed Event）

与相关属性可以附加到其他类上一样，WPF的附加传递事件也可以附加到其他类上。例如：

```
<Grid Mouse.PreviewMouseDown= "OnPreviewMouseDown"/>
```

Grid类中并没有定义PreviewMouseDown事件，但却可以把Mouse类中定义的附加传递事件引入到自己的类中。

在C#代码中接入附加传递事件有点不同，需要调用UIElement类中AddHandler：

```
public void AddHandler(
    RoutedEvent routedEvent,
    Delegate handler)
```

在Grid类中加入鼠标附加事件处理程序,可以调用AddEventHander：

```
 AddHandler(Mouse.PreviewMouseDownEvent, new
MouseButtonEventHandler( OnPreviewMouseDown) );
```

如果需要在某个类中动态去掉某个附加传递事件，则可以调用UIElement类中的RemoveHandler方法：

```
public void RemoveHandler(
    RoutedEvent routedEvent,
    Delegate handler)
```

到此为止，基本上讲述了WPF中的传递事件系统及其用法。要对这些概念进行深化，可以看7.3节所述的考察传递事件例子。

7.3 考察传递事件

为了考察传递事件在元素树中传播，可以从一个简单的元素树开始，图7-4示出了所要观察的逻辑树。

树根元素为Window，左边的树枝主要有一个Button；组成Button的元素为一个StackPanel中包含两个椭圆和一个TextBlock。右边的数值用来显示事项传递的结果。

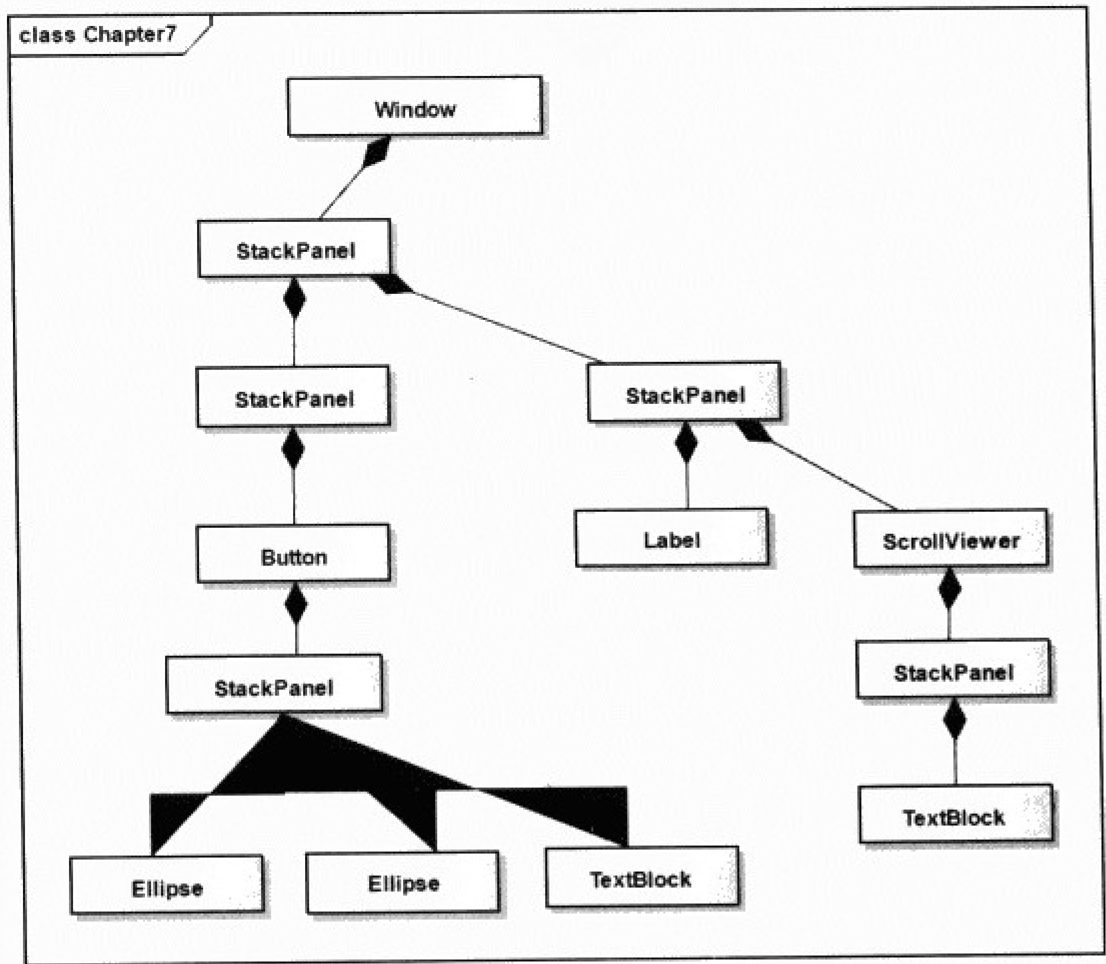


图7-4 考察传递事件程序中所用的逻辑树

下面是描述该UI的XAML：

```
<Window x:Class="Yingbao.Chapter7.ExaminRoutedEvent"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="考察传递事件" Height="300" Width="700"
        MouseDown ="RoutedEventHandler"
        PreviewMouseDown ="RoutedEventHandler"
        MouseUp ="RoutedEventHandler"
        PreviewMouseUp ="RoutedEventHandler"
        PreviewKeyDown ="RoutedEventHandler"
        PreviewKeyUp="RoutedEventHandler"
        KeyDown ="RoutedEventHandler"
        KeyUp ="RoutedEventHandler"
        Button.Click="RoutedEventHandler">
  <StackPanel Orientation ="Horizontal"
        MouseDown ="RoutedEventHandler"
        PreviewMouseDown ="RoutedEventHandler"
        MouseUp ="RoutedEventHandler"
        PreviewMouseUp ="RoutedEventHandler"
        PreviewKeyDown ="RoutedEventHandler"
        PreviewKeyUp="RoutedEventHandler"
        KeyDown ="RoutedEventHandler"
        KeyUp ="RoutedEventHandler"
       Button.Click="RoutedEventHandler">
    <StackPanel MouseDown ="RoutedEventHandler"
```

```
  PreviewMouseDown ="RoutedEventHandler"
  MouseUp ="RoutedEventHandler"
  PreviewMouseUp ="RoutedEventHandler"
  PreviewKeyDown ="RoutedEventHandler"
  PreviewKeyUp="RoutedEventHandler"
  KeyDown ="RoutedEventHandler"
  KeyUp ="RoutedEventHandler"
  Button.Click="RoutedEventHandler">
<Button MouseDown ="RoutedEventHandler"
  PreviewMouseDown ="RoutedEventHandler"
  MouseUp ="RoutedEventHandler"
  PreviewMouseUp ="RoutedEventHandler"
  PreviewKeyDown ="RoutedEventHandler"
  PreviewKeyUp="RoutedEventHandler"
  KeyDown ="RoutedEventHandler"
  KeyUp ="RoutedEventHandler">
  <StackPanel MouseDown ="RoutedEventHandler"
  PreviewMouseDown ="RoutedEventHandler"
  MouseUp ="RoutedEventHandler"
  PreviewMouseUp ="RoutedEventHandler"
  PreviewKeyDown ="RoutedEventHandler"
  PreviewKeyUp="RoutedEventHandler"
  KeyDown ="RoutedEventHandler"
  KeyUp ="RoutedEventHandler">
 <Ellipse Height ="50" Width ="100"  Fill ="Brown"
  MouseDown ="RoutedEventHandler"
  PreviewMouseDown ="RoutedEventHandler"
  MouseUp ="RoutedEventHandler"
  PreviewMouseUp ="RoutedEventHandler"
  PreviewKeyDown ="RoutedEventHandler"
  PreviewKeyUp="RoutedEventHandler"
  KeyDown ="RoutedEventHandler"
  KeyUp ="RoutedEventHandler"
  Button.Click="RoutedEventHandler"/>
  <Ellipse Height ="50" Width ="100"  Fill ="Green"
  MouseDown ="RoutedEventHandler"
  PreviewMouseDown ="RoutedEventHandler"
  MouseUp ="RoutedEventHandler"
  PreviewMouseUp ="RoutedEventHandler"
  PreviewKeyDown ="RoutedEventHandler"
  PreviewKeyUp="RoutedEventHandler"
  KeyDown ="RoutedEventHandler"
  KeyUp ="RoutedEventHandler"
  Button.Click="RoutedEventHandler"/>
  <TextBlock MouseDown ="RoutedEventHandler"
  PreviewMouseDown ="RoutedEventHandler"
  MouseUp ="RoutedEventHandler"
  PreviewMouseUp ="RoutedEventHandler"
  PreviewKeyDown ="RoutedEventHandler"
  PreviewKeyUp="RoutedEventHandler"
  KeyDown ="RoutedEventHandler"
  KeyUp ="RoutedEventHandler"
  Button.Click="RoutedEventHandler"
```

```
        >这是一个按键！</TextBlock>
        </StackPanel>
      </Button>
    </StackPanel>
    <StackPanel Width ="560"  Margin ="20" Height ="280">
      <Label  Name ="EventOutputHeader" FontSize ="12"> </Label>
      <ScrollViewer Height ="200">
        <StackPanel Name="EventOutput">
        </StackPanel>
      </ScrollViewer>
    </StackPanel>
  </StackPanel>
</Window>
```

笔者在逻辑树的每个节点上都接入了MouseDown、PreviewMouseDown、MouseUp、PreviewMouseUp、PreviewKeyDown、PreviewKeyUp、KeyDown、KeyUp等事件；同时，还在一些节点上接入了Button.Click的附加事件，所有的事件都是在RoutedEventHandler中处理的。

现在来看一下RoutedEventHandler函数：

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Data;
using System.Windows.Documents;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Media.Imaging;
using System.Windows.Shapes;

namespace Yingbao.Chapter7
{
    public partial class ExaminRoutedEvent : System.Windows.Window
    {
        string outputFormat = "{0,-40}{1,-30}{2,-30}{3,-30}";

        public ExaminRoutedEvent()
        {
            InitializeComponent();
            EventOutputHeader.Content = string.Format(outputFormat,
                "Routed Event ", "Sender", "Source", "Original
                 Source");

        }

        private void RoutedEventHandler(object sender,
                        RoutedEventArgs rea)
        {
            TextBlock text = new TextBlock();
            text.Text = string.Format(outputFormat,
                            rea.RoutedEvent.Name,
```

```
                    GetTypeName(sender),
                    GetTypeName(rea.Source),
                    GetTypeName(rea.OriginalSource));
            EventOutput.Children.Add(text);
            (EventOutput.Parent as ScrollViewer).ScrollToBottom();
            while(EventOutput.Children.Count > 20)
            {
                EventOutput.Children.RemoveAt(0);
            }

        }

        private string GetTypeName(object obj)
        {
            string[]  name = obj.GetType().ToString().Split('.');
            return name[name.Length - 1];
        }
    }
}
```

在RoutedEventHandler中，笔者创建了TextBlock对象，在其中显示传递事件的名字(Name)、发出传递事件的对象名字（Sender）、产生该传递事件的源对象名（Source）和最初源对象名（Original Source）。

运行RoutedEventHandler函数的结果如图7-5所示。

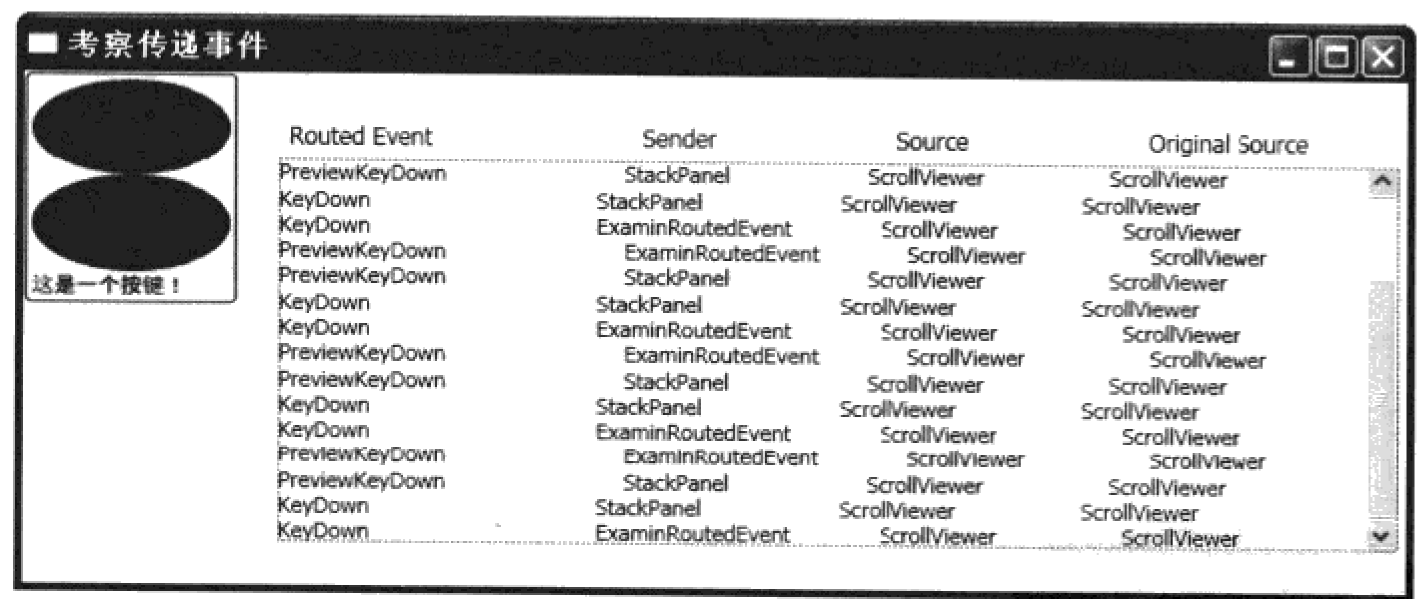


图7-5 考察WPF中的传递事件

当在图7-5左上方的“这是一个按键！”上按下鼠标右键时，可以看到PreviewMouseDown传递事件首先出现，事件源对象（Source）和事件的最初源对象（OriginalSource）均为TextBlock（按下鼠标的位置在TextBlock上！）。根据事件发出对象（sender）的顺序，可以看到PreviewMouseDown事件在元素树上的传递顺序如图7-6所示。

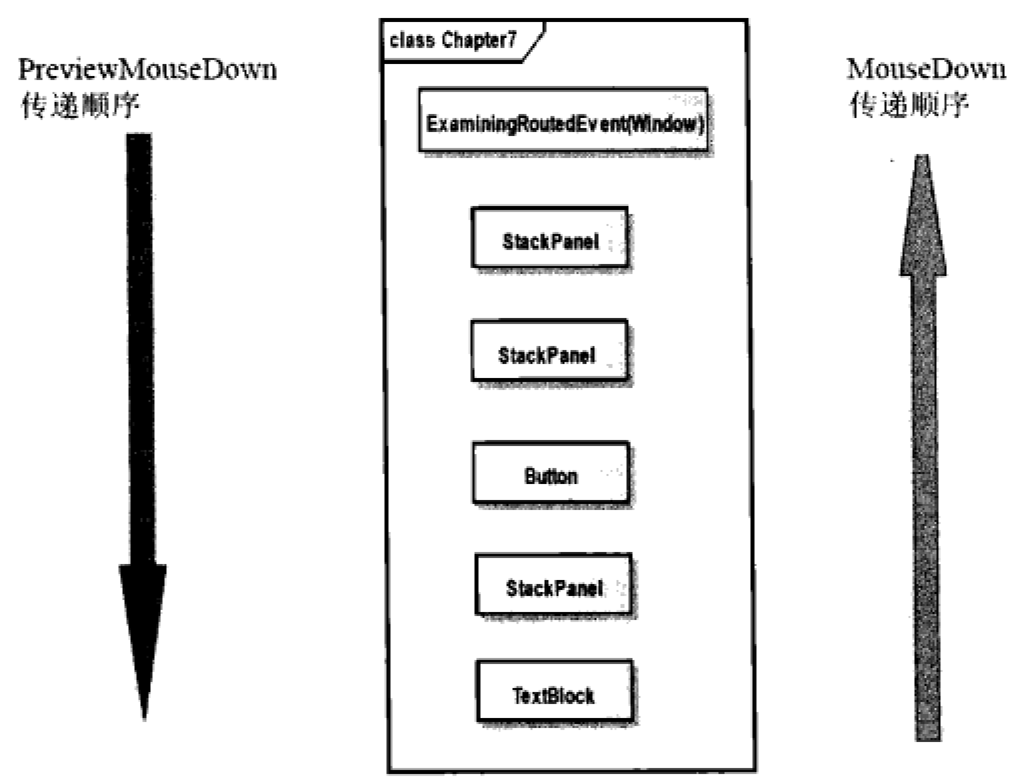


图7-6 PreviewMouseDown和MouseDown事件在元素树中的传递路径（鼠标右键）

从图7-6可以看出PreviewMouseDown传递事件是从树根向树枝传递的，即从Window对象传递到StackPanel，最后到达树枝末端（TextBlock）。在PreviewMouseDown事件之后，紧接着便是MouseDown事件。MouseDown在元素树中传递的顺序和PreviewMouseDown刚好相反。即从TextBlock传递到StackPanel，最后传递到树根（Window）。当松开鼠标右键，可以看到，PreviewMouseUp和MouseUp.事件按照同样的模式传递：即PreviewMouseUp传递事件先于MouseUp事件出现，PreviewMouseUp从树根向树枝传递，而MouseUp则是从事件发生处向树根传递。这就是传递事件的两种常见的传递方式——冒泡和潜入。

若在椭圆上按下鼠标右键，则可以看到Ellipse对象代替了TextBlock对象，其他一样。若在按键方框内，但既不在椭圆又不在TextBlock上按下鼠标右键，则PreviewMouseDown和PreviewMouseUp事件传递到Button对象为止；而MouseDown和MouseUp事件则由Button开始到Window结束。图7-6中的最后两个元素不再参加传递事件的传递。

现在要考察按下鼠标左键时，传递事件的传递过程。在“这是一个按键！”TextBlock上按下鼠标左键，图7-7示出了同样的MouseDown和PreviewMouseDown的传递过程。

比较图7-6和图7-7可以看出，虽然按下的仍然是TextBlock，PreviewMouseDown事件仍然从Window对象传递到TextBlock对象，但MouseDown事件却只从TextBlock传递到StackBlock为止。这是为什么呢？

要回答这个问题，先把鼠标松开。非常有意思的是，这时与松开鼠标右键的情况完全不同，看到PreviewMouseUp从树根Window开始传递，但传递到Button时截然而止。而MouseUp事件根本就没有出现，取而代之的是Click事件。Click事件沿着元素树按冒泡的方式传递，但是却没有潜入方式传递的Click事件。

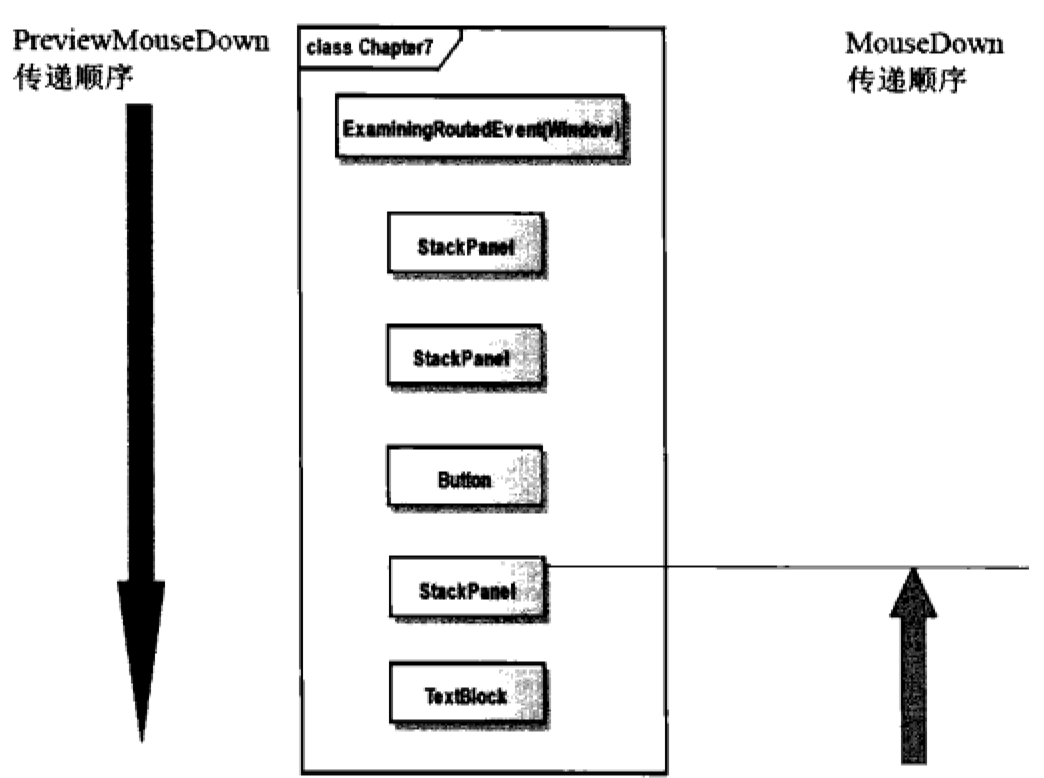


图7-7 PreviewMouseDown和MouseDown事件在元素树中的传递路径（鼠标左键）

原来Button对按下鼠标左键的传递事件进行了处理，它在截获MouseDown和PreviewMouseUp两个事件之后，产生了一个新的事件Click，这个新的事件只向树根传播。在7.4节自定义传递事件时，将介绍如何控制传递事件的传递方式。

7.3.1 键盘事件的产生和传递

上述程序也可用来考察键盘输入事件在元素树上的传播，只要先指定键盘的输入焦点，然后按下键盘上的一个键，就可以观察到PreviewKeyDown和KeyDown在元素树上的传播；请读者自行总结，当松开键盘上的键时，PreviewKeyUp 事件及Click事件的产生和传播。

7.4 自定义传递事件

笔者在第4章WPF中的属性系统中介绍了如何定义自己的相关属性；同样也可以定义自己的传递事件。下面是定义自己的传递事件的步骤：

- 声明一个传递事件 声明一个传递事件的语法为：

```
public static readonly RoutedEvent   YclickEvent;
public static readonly RoutedEvent   PreviewYclickEvent;
```

在这里笔者定义了YclickEvent和PreviewYclickEvent两个传递事件，其类型为RoutedEvent，它们都被说明为static和readonly的，这是必需的。

- 在WPF传递事件系统中注册传递事件 与相关属性需要在相关属性系统中注册一样，也需要向传递系统注册传递事件。

```
YclickEvent=EventManager.RegisterRoutedEvent ("Yclick",
RoutingStrategy.Bubble, typeof(RoutedEventHandler),
typeof( Ybutton)  );
```

```
PreviewYclickEvent =EventManager.RegisterRoutedEvent ("Yclick",
RoutingStrategy.Tunnel, typeof(RoutedEventHandler),
typeof( Ybutton)  ) ;
```

WPF用RoutingStrategy来定义传递事件的传递方式，它取Tunnel、Bubble和Direct三个值，分别对应于潜入传递、冒泡传递和直接传递（只传递到自身类）三种传递方式。在这里，我把YclickEvent的传递方式设为冒泡方式，把PreviewYclickEvent设为潜入方式。第三个参数为传递事件处理函数的类型，第四个参数说明传递事件是在哪个类中定义的。

● 定义传递事件的接入属性

```
public event RoutedEventHandler  Yclick
{
      add {AddHandler(YclickEvent,value);}
      remove{RemoveHandler(YclickEvent,value);}
}

public event RoutedEventHandler  PreviewYclick
{
      add {AddHandler(PreviewYclickEvent,value);}
      remove{RemoveHandler(PreviewYclickEvent,value);}
}
```

AddHandler和RemoveHandler是DependencyObject中的两个方法，所以为了使用自定义传递事件，Ybutton的继承树上需要有DependencyObject类。

● 产生传递事件

要在WPF传递事件系统中产生传递事件，需要调用UIElement类中的RaiseEvent方法：

```
RoutedEventArgs argsEvent = new RoutedEventArgs();
argsEvent.RoutedEvent = YButton.PreviewYclickEvent;
argsEvent.Source = this;
RaiseEvent(argsEvent);
```

好了，在完成了上面四个步骤之后，就可以使用自己定义的事项了。

前面的例子，我们创建的按钮是由一个StackPanel、两个Ellipse和一个TextBlock组成的，样子有点奇怪。笔者主要的目的是要展示传递事件如何在元素树中传播。为了演示如何定义并使用传递事件，在下面的例子中，笔者还是创建一个按钮，不过这个按钮不只是换一下视觉元素，而是做一点内部较深入的工作，让YButton 类从Control类中派生出来：

```
using System;
using System.Globalization;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Input;
using System.Windows.Media;
namespace Yingbao.Chapter7
{
    public class YButton : Control
    {
```

```
FormattedText formtxt;
bool isMouseReallyOver;
public static readonly DependencyProperty TextProperty;
public static readonly RoutedEvent YclickEvent;
public static readonly RoutedEvent PreviewYclickEvent;
static YButton()
{
    TextProperty =DependencyProperty.Register("Text",
          typeof(string),typeof(YButton),
          new FrameworkPropertyMetadata(" ",
           FrameworkPropertyMetadataOptions.AffectsMeasure));
    YclickEvent = EventManager.RegisterRoutedEvent(
                "Yclick", RoutingStrategy.Bubble,
                typeof(RoutedEventHandler), typeof(YButton));
    PreviewYclickEvent = EventManager.RegisterRoutedEvent(
                "PreviewYclick",RoutingStrategy.Tunnel,
                typeof(RoutedEventHandler), typeof(YButton));
}
public string Text
{
    set { SetValue(TextProperty, value == null ? " " :
          value); }
    get { return (string)GetValue(TextProperty); }
}

public event RoutedEventHandler Yclick
{
    add { AddHandler(YclickEvent, value); }
    remove { RemoveHandler(YclickEvent, value); }
}
public event RoutedEventHandler PreviewYclick
{
    add { AddHandler(PreviewYclickEvent, value); }
    remove { RemoveHandler(PreviewYclickEvent, value); }
}

protected override Size MeasureOverride(Size sizeAvailable)
{
    formtxt = new FormattedText(Text,
              CultureInfo.CurrentCulture, FlowDirection,
              new Typeface(FontFamily, FontStyle, FontWeight,
              FontStretch), FontSize, Foreground);

    Size sizeDesired = new Size(Math.Max(48, formtxt.Width)
              + 4,formtxt.Height + 4);
    sizeDesired.Width += Padding.Left + Padding.Right;
    sizeDesired.Height += Padding.Top + Padding.Bottom;

    return sizeDesired;
}

protected override void OnRender(DrawingContext dc)
{
```

```
        Brush brushBackground = SystemColors.ControlBrush;

        if (isMouseReallyOver && IsMouseCaptured)
            brushBackground = SystemColors.ControlDarkBrush;

        Pen pen = new Pen(Foreground, IsMouseOver ? 2 : 1);

        dc.DrawRoundedRectangle(brushBackground, pen,
                 new Rect(new Point(0, 0), RenderSize), 4, 4);

        formtxt.SetForegroundBrush( IsEnabled ? Foreground :
                SystemColors.ControlDarkBrush);

        Point ptText = new Point(2, 2);

        switch (HorizontalContentAlignment)
        {
            case HorizontalAlignment.Left:
                ptText.X += Padding.Left;
                break;

            case HorizontalAlignment.Right:
                ptText.X += RenderSize.Width - formtxt.Width -
                               Padding.Right;
                break;

            case HorizontalAlignment.Center:
            case HorizontalAlignment.Stretch:
                ptText.X += (RenderSize.Width - formtxt.Width -
                         Padding.Left - Padding.Right) / 2;
                break;
        }
        switch (VerticalContentAlignment)
        {
            case VerticalAlignment.Top:
                ptText.Y += Padding.Top;
                break;

            case VerticalAlignment.Bottom:
                ptText.Y += RenderSize.Height - formtxt.Height -
                               Padding.Bottom;
                break;

            case VerticalAlignment.Center:
            case VerticalAlignment.Stretch:
                ptText.Y += (RenderSize.Height - formtxt.Height -
                         Padding.Top - Padding.Bottom) / 2;
                break;
        }

        dc.DrawText(formtxt, ptText);
    }
```

```
    protected override void OnMouseEnter(MouseEventArgs args)
    {
        base.OnMouseEnter(args);
        InvalidateVisual();
    }
    protected override void OnMouseLeave(MouseEventArgs args)
    {
        base.OnMouseLeave(args);
        InvalidateVisual();
    }
    protected override void OnMouseMove(MouseEventArgs args)
    {
        base.OnMouseMove(args);
        Point pt = args.GetPosition(this);
        bool isReallyOverNow = (pt.X >= 0 && pt.X < ActualWidth
                  && pt.Y >= 0 && pt.Y < ActualHeight);
        if (isReallyOverNow != isMouseReallyOver)
        {
            isMouseReallyOver = isReallyOverNow;
            InvalidateVisual();
        }
    }

  protected override void OnMouseLeftButtonDown(
                   MouseButtonEventArgs args)
    {
        base.OnMouseLeftButtonDown(args);
        CaptureMouse();
        InvalidateVisual();
        args.Handled = true;
    }

    protected override void OnMouseLeftButtonUp(
                   MouseButtonEventArgs args)
    {
        base.OnMouseLeftButtonUp(args);

        if (IsMouseCaptured)
        {
            if (isMouseReallyOver)
            {
                OnPreviewYclick();
                OnYclick();
            }
            args.Handled = true;
            Mouse.Capture(null);
        }
    }

    protected override void OnLostMouseCapture(
                     MouseEventArgs args)
    {
```

```
            base.OnLostMouseCapture(args);
            InvalidateVisual();
        }

        protected override void OnKeyDown(KeyEventArgs args)
        {
            base.OnKeyDown(args);
            if (args.Key == Key.Space || args.Key == Key.Enter)
                args.Handled = true;
        }

        protected override void OnKeyUp(KeyEventArgs args)
        {
            base.OnKeyUp(args);
            if (args.Key == Key.Space || args.Key == Key.Enter)
            {
                OnPreviewYclick();
                OnYclick();
                args.Handled = true;
            }
        }

        protected virtual void OnYclick()
        {
            RoutedEventArgs argsEvent = new RoutedEventArgs();
            argsEvent.RoutedEvent = YButton.YclickEvent;
            argsEvent.Source = this;
            RaiseEvent(argsEvent);
        }

        protected virtual void OnPreviewYclick()
        {
            RoutedEventArgs argsEvent = new RoutedEventArgs();
            argsEvent.RoutedEvent = YButton.PreviewYclickEvent;
            argsEvent.Source = this;
            RaiseEvent(argsEvent);
        }
    }
}
```

在YButton中，笔者定义了两个传递事件：一个是Yclick，另一个是PreviewYclick；一个相关属性TextProperty。为了产生传递事件，在YButton中截获了按下鼠标左键的传递事件，并把RoutedEventArgs类中的Handled的属性设为true，以阻止传递事件的传播。在释放鼠标左键的时候，首先产生的是PreviewYclick事件，再产生Yclick事件。类似地，还截获了空格键和回车键，在用户按下或释放这两个键的过程中产生Yclick 和PreviewYclick传递事件。

使用YButton的程序如下：

```
<Window x:Class="Yingbao.Chapter7.UsingYButton"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:src="clr-namespace:Yingbao.Chapter7"
    Title="使用自己的按钮" Height="500" Width="700"
```

```
    src:YButton.PreviewYclick="OnPreviewYclick"
    src:YButton.Yclick="OnYclick">
  <StackPanel Orientation ="Horizontal"
    src:YButton.PreviewYclick="OnPreviewYclick"
    src:YButton.Yclick="OnYclick">
  <StackPanel  src:YButton.PreviewYclick="OnPreviewYclick"
    src:YButton.Yclick="OnYclick">
    <src:YButton x:Name="myButton1" Text=" 使用自定义按钮1"/>
    <src:YButton x:Name="myButton2" Text=" 使用自定义按钮2"/>
    <src:YButton x:Name="myButton3" Text=" 使用自定义按钮3"/>
  </StackPanel>
    <StackPanel Width ="560"  Margin ="20" Height ="450">
      <Label  Name ="EventOutputHeader" FontSize ="12"> </Label>
      <ScrollViewer Height ="400">
        <StackPanel Name="EventOutput">
        </StackPanel>
      </ScrollViewer>
    </StackPanel>
  </StackPanel>
</Window>
```

后台C#程序如下：

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Data;
using System.Windows.Documents;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Media.Imaging;
using System.Windows.Shapes;

namespace Yingbao.Chapter7
{
        public partial class UsingYButton : System.Windows.Window
    {
        string outputFormat = "{0,-40}{1,-30}{2,-30}{3,-30}";
        public UsingYButton()
        {
            InitializeComponent();
            EventOutputHeader.Content = string.Format(outputFormat,
        "Routed Event ", "Sender", "Source", "Original Source");

        }

        private void OnPreviewYclick(object sender,
                      RoutedEventArgs rea)
        {
            RoutedEventHandler(sender, rea);
        }
        private void OnYclick(object sender, RoutedEventArgs rea)
        {
```

```
            RoutedEventHandler(sender, rea);
        }

        private void RoutedEventHandler(object sender,
                RoutedEventArgs rea)
        {
            TextBlock text = new TextBlock();
            text.Text = string.Format(outputFormat,
                    rea.RoutedEvent.Name, GetTypeName(sender),
                    GetTypeName(rea.Source),
                    GetTypeName(rea.OriginalSource));
            EventOutput.Children.Add(text);
            (EventOutput.Parent as ScrollViewer).ScrollToBottom();
            while (EventOutput.Children.Count > 40)
            {
                EventOutput.Children.RemoveAt(0);
            }
        }
        private string GetTypeName(object obj)
        {
            string[] name = obj.GetType().ToString().Split('.');
            return name[name.Length - 1];
        }
    }
}
```

与7.3节考察传递事件在元素树中传播的例子一样，这个例子可以考察自定义的传递事件，也可以按照冒泡和潜入两种方式传播。按下按钮，我们可以看到PreviewYclick和Yclick事件的传递过程。图7-8示出了上面程序的运行结果，在XAML中使用YButton按钮，和使用WPF本身所提供的控件几乎没什么区别，这样，就可以根据自己的需要，很方便地对WPF控件进行自己的扩展。

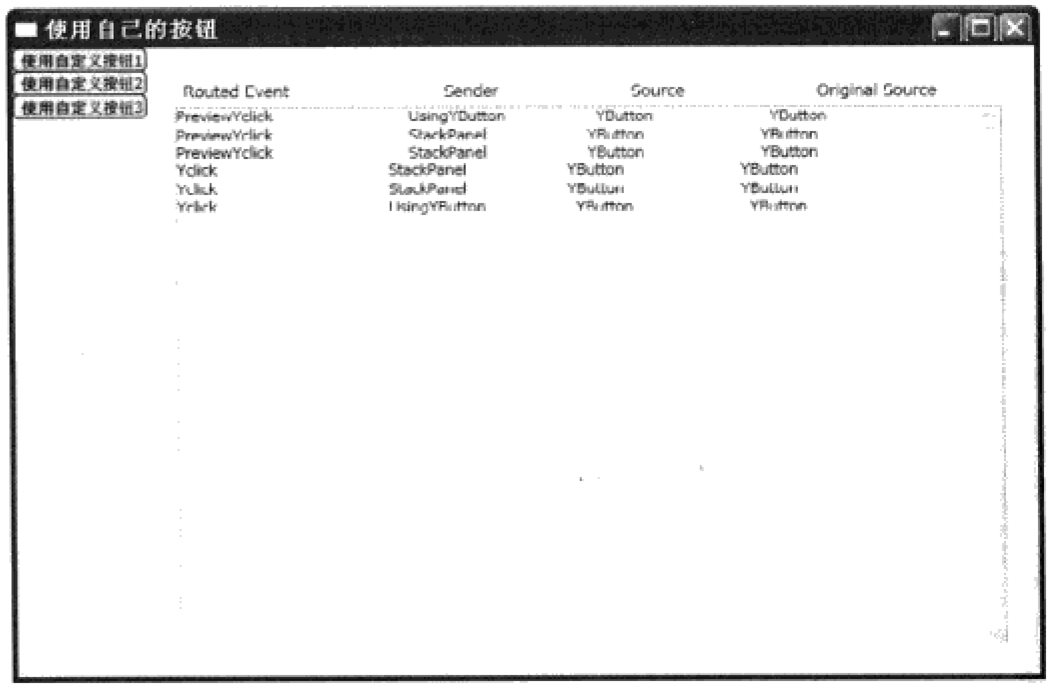


图7-8 使用自定义传递事件

7.5 管理键盘和鼠标输入事件

人主要通过三种途径向个人计算机发出指令：键盘、鼠标和手写设备。键盘首先出现，然后是鼠标，现在手写设备开始普及。在可见的将来，我比较看好语音输入和虚拟输入，这两种输入方式将会极大地提高人机交换的速度和信息量。但由于技术上的复杂性，目前还不能成为主流。

对应于这三种输入设备，WPF有三个类与之对应：Keyboard、Mouse和Stylus。当用户使用这三种输入设备时，WPF产生相应的传递事件在WPF元素树上传播。除了这三个专门的输入类之外，输入事件集中在WPF中的四大基类：UIElement、FrameworkElement、ContentElement、FrameworkContentElement中进行处理。

7.5.1 键盘输入

键盘是较早的输入设备，从前面的例子，可以看出，键盘事件是一种传递事件，它可以沿着WPF中的元素树向树根（Bubble）或树枝传递（Tunnel）。

WPF中的传递事件采用一种开放式架构，在UIElement和ContentElement中都可以附加传递事件。这两个类是WPF大多UI元素的基类，所以传递事件可以附加到任何WPF的控件中。所有与键盘相关的事件都定义在Keyboard类中；所有与鼠标相关的事件都定义在Mouse类中；所有与手写输入相关的事件都定于在Stylus类中。这些传递事件都是以附加事件的形式存在，将来一旦有新的输入设备出现，WPF只要创建一个新的类，并在这个类中定义相应的附加事件，然后就可以附加到所有控件上，这样可以很容易地支持新的输入设备。

Keyboard类中所定义的传递事件如表7-2所示。

表7-2 Keyboard类中所移植的附加事件

传递事件名	传递方式	功能描述
GotKeyboardFocus	冒泡	当元素获得输入焦点时产生。
KeyDown	冒泡	当按下键盘时产生
KeyUp	冒泡	当释放键盘时产生
LostKeyboardFocus	冒泡	当元素失去输入焦点时产生，与GotKeyboardFocus事件相对
PreviewGotKeyboardFocus	潜入	当元素获得输入焦点时产生
PreviewKeyDown	潜入	当按下键盘时产生
PreviewKeyUp	潜入	当释放键盘时产生
PreviewLostKeyboardFocus	潜入	当元素失去输入焦点时产生，与PreviewGotKeyboardFocus事件相对

当在键盘上按下一个键时，总有某个应用程序中的某个元素获得输入消息，这个消息被WPF翻译成传递事件，然后在元素树中传播。当按下Tab键，或者用鼠标单击某个图形元素时，键盘的输入焦点在图形元素间移动。当同时按下“Alt”和Tab键时，可以在不同的应用程序间切换。幸运的是，你切换不同的应用程序，应用程序会自动记住你的输入焦点，这样，当你切换回原来的应用程序时，自动回到原来的输入焦点。

7.5.2 鼠标输入

就像键盘输入传递事件是在Keyboard类中移植的一样，鼠标输入也有一个专门的类Mouse。表7-3列出了Mouse类中移植的传递事件，这些传递事件都可以附加到任何UI元素上。

- 俘获鼠标 所谓俘获鼠标是指在应用程序中设置鼠标的一种状态，这时哪怕是光标离开了应用程序的窗口，应用程序仍然可以接收到鼠标的信息。与之相反的操作就是释放鼠标，这种功能对于开发拖放功能非常有用。

在WPF中俘获和释放鼠标非常简单，一种方法是调用UIElement类中的CaptureMouse()和ReleaseCaptureMouse()方法（如我在前面的例子中用的）。另一种方法是调用Mouse类中的静态Capture方法。使用第一种方法的前提是UIElement类要在你的类的继承树上，多数需要捕获鼠标的元素都是从UIElement的派生类。

表7-3 Mouse类中所支持的附加事件

传递事件名	传递方式	功能描述
GotMouseCapture	冒泡	当元素捕获到鼠标时，发出的消息。当元素捕获到鼠标时，所有的鼠标事件都发给该元素。即使输入焦点不在元素所在的应用程序上时，鼠标的事件仍然被发送到该元素上
LostMouseCapture	冒泡	当元素不再捕获鼠标时，产生这一事件
MouseDown	冒泡	当鼠标在元素上按下时，产生该事件
MouseEnter	冒泡	当鼠标进入元素的几何范围时，产生该事件
MouseLeave	冒泡	当鼠标离开元素的几何范围时，产生该事件
MouseMove	冒泡	当移动鼠标时，产生该事件
MouseUp	冒泡	当释放鼠标时，产生该事件
MouseWheel	冒泡	当移动鼠标上的小轮子时，产生该事件
PreviewMouseDown	潜入	当按下鼠标时，产生该事件
PreviewMouseDownOutsideCapturedElement	潜入	当鼠标在捕获鼠标的元素几何界限外的区域内，按下鼠标时，产生该事件
PreviewMouseMove	潜入	但鼠标移动时，产生该事件
PreviewMouseUp	潜入	当释放鼠标时，产生该事件
PreviewMouseUpOutsideCapturedElement	潜入	当鼠标在捕获鼠标的元素几何界限外的区域内，释放鼠标时，产生该事件
PreviewMouseWheel	潜入	当移动鼠标上的小轮子时，产生该事件
QueryCursor	冒泡	当询问当前鼠标位置时，产生该事件

- 输入焦点 WPF支持丰富的用户界面，某个呈现在用户面前的图形可能是由多个元素组合而成的。当这些元素前后叠加时，究竟是哪个元素得到了输入焦点？在二维图形元素当中通常是在最上面的元素获得输入焦点，然而，有时候希望某个图形的下面的元素获得输入焦点。比如，要在界面上展现一个盒子，盒子里放着红色、白色和黑色的小球，希望当用户把盒子的盖子打开的时候，用户可以用鼠标操作把其中的小球一个一个地取出来。在这种情况下，小球在界面上不是处在最上面的位置，要用鼠标把小球取出来的第一项工作就是小球要能够获得输入焦点。WPF的解决方法是若在绘制元素时的画刷不是空的（null ）,那么图形元素就可以获得输入焦点。若绘制图形时所用的画刷是空的，那么该图形就不能获得输入焦点，而是他下面的其他元素获得了输入焦点。对于上面的问题，只要把位于小球前方的图形元素的画刷设为空即可。

- 鼠标经过元素 当多个图形元素叠加在界面上，当鼠标在界面上移动的时候，可能多个元素都会产生MouseOver的传递事件。有时候希望知道鼠标是否直接在某个元素的上面，例如，按钮可能由多个图形、TextBlock等控件组合起来，鼠标的光标可能在图形元素上，也可能在TextBlock上，但都在按键的几何范围内。这个时候要想知道是否在某个元素上，需要调用DirectlyOver方法，该方法返回IInputElement。

- 获取鼠标状态　鼠标键的状态用MouseButtonState来描述，这是一个枚举类型。它有两个值：一个是Pressed，另一个是Released。一般的鼠标有左右两个键，再加上一个小轮子；WPF支持五个键的鼠标，即LeftButton、RightButton、MiddleButton和XButton1、XButton2。XButton和XButton2是位于鼠标侧面的两个键。要获取某个键的状态，直接读取Mouse类中的相应属性就行，非常方便。
- 获取鼠标的位置 WPF中鼠标的位置是一个相对坐标，相对与某个控件的左上角（0，0）。若鼠标位于该元素的左边，那么X值为负值；若鼠标位于该元素的上面边，那么Y值为负值；GetPosition（IInputElement）返回的是Point类型，为鼠标的光标所在点的位置。
- 设置鼠标的光标，鼠标的光标有时可以用来直观地表示应用程序所处的状态。Mouse类中的SetCursor方法可以改变鼠标的光标。

7.6 传递命令

命令这个设计范例最早见于Henry Lieberman在1985年发表的一篇论文中，后来Ganma等在《Design Partterns》一书中对这一设计范例进行了系统的总结。命令这种设计范例用在这样的场合：发布命令的对象并不知道该命令怎么执行。有意思的是，笔者在写这一段的时候，刚好是中国神舟7号载人航天卫星（2008年9月25号）发射的时间，笔者一边写，一边在听中央电视台的实况转播。命令这个设计范例就好像神舟发射现场那个发令员对着话筒喊出：

“2分钟准备！”

“1分钟准备！”

“20秒准备！”……

这一系列指令一样，你觉得那个发出指令的发令员知道2分钟该准备什么吗？笔者虽然没有参加航天工程，但可以放心地告诉你：他不知道该准备啥，这是一个大的系统工程，有些人在现场，有些人在千里之外！他要是知道哪个地方该准备啥，那他就神了！他的工作只要在预订的时间点上把相关的命令发出去，至于怎么执行这个命令，这不是他的职责。

在大型软件工程中，一个软件往往由多个模块组成，通常希望模块和模块之间的相关性越少越好，用专门的术语叫decouple，从而把对一个模块的修改对其他模块的影响降到最低。使用命令这种设计范例就可以达到目的：发布命令的模块对于命令如何执行一无所知，接收命令的模块对命令加以解释并具体执行。例如，Visual Studio、Eclipse、Office等软件，允许插入第三方模块，菜单上的选项事先根本不可能知道要对哪个控件进行操作。比如说你在Visual Studio里使用DevExpress控件，当你在一个Form上选择一个DevExpress控件，然后按下菜单中的复制选项时，Microsoft的Visual Studio 根本不知道该如何复制DevExpress的控件，这是DevExpress控件的责任，它把自己复制到剪贴板上；同样，当选择粘贴命令的时候，也是DexExpress控件完成把自己粘贴到目标对象的任务。

命令范例的实现方式如图7-9所示。

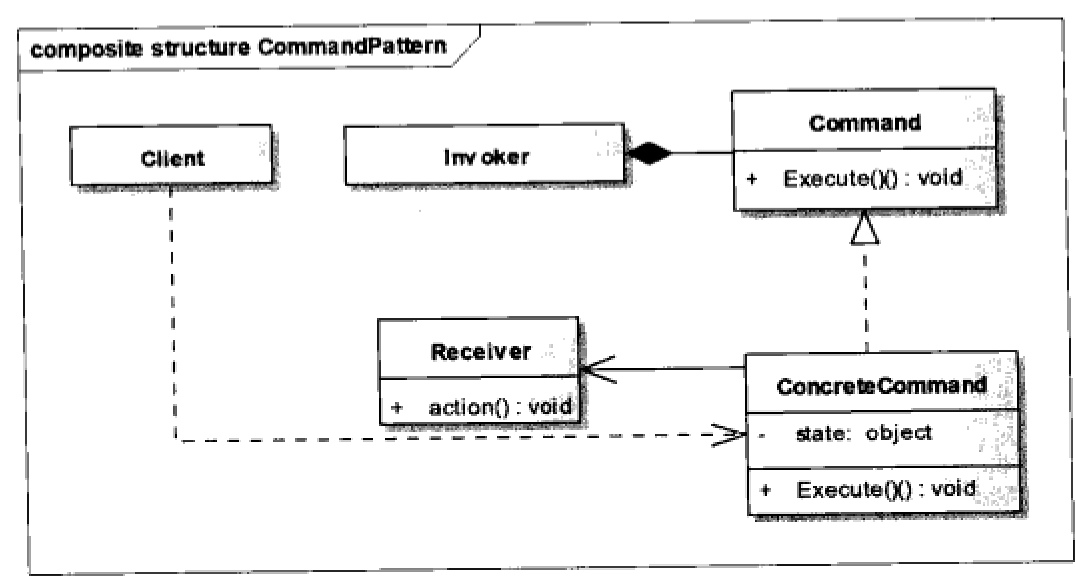


图7-9 命令范例的UML图示

在图7-9中Invoker类是命令的发出者，Command通常是抽象类（Absctract class in C++/C#）或Interface（C#或Java），ConcreteCommand实现到具体对象上的操作（Receiver类），它应知道要调用目标类上的方法。

WPF中不仅支持命令范例，而且实现了命令和传递事件一样在元素树上的传递，WPF更进一步开发了常用命令的功能。WPF传递命令的作用机制如图7-10所示。

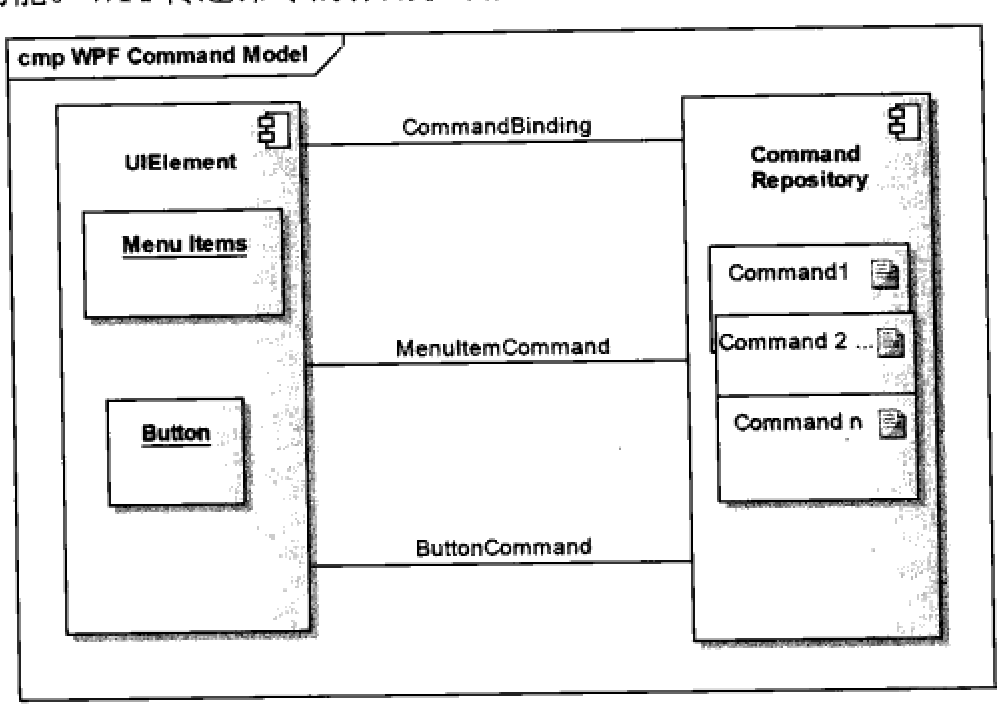


图7-10 WPF中的传递命令

通常菜单条目或按钮发出相关命令，WPF的命令仓库中（Command Repository）支持一系列命令，然后把命令绑定到用户界面上的UI元素上。例如，在菜单条目里使用命令：

```
<StackPanel>
  <Menu>
    <MenuItem Command="ApplicationCommands.Paste" />
  </Menu>
  <TextBox />
</StackPanel>
```

其中ApplicationCommands.Paste就是命令仓库中的粘贴命令。

7.6.1 ICommand接口

WPF中的命令至少要移植ICommand接口，在ICommand接口中，定义了两个方法和一个事件：

- CanExecute() 返回true或false，表示命令是否可以执行；
- Execute(object) 执行命令；
- 一个事件是CanExecuteChanged，当命令从不能执行转变为可以执行时，或者从不能执行转变为可以执行时，产生这一事件。例如，在剪贴板中没有对象时，粘贴（Paste）命令是不能执行的，这时菜单“粘贴”条目应该变灰（disable）。当用户将某个对象复制到剪贴板中，粘贴命令就可以执行了，但需要通知命令源使得菜单“粘贴”条目使能（enable），通知菜单条目使能实际上就是通过这一事件完成的。

7.6.2 ICommandSource接口

要成为可以发出命令的对象，需要移植ICommandSource接口。WPF中的ButtonBase、MenuItem、ListBoxItem和HyperLink类都移植了ICommandSource接口。

ICommandSource定义了三个属性，即Command（要发出的命令）、CommandParameter（发出命令时的参数）和CommandTarget（命令作用的对象）。

7.6.3 CommandTarget

CommandTarget是指要执行命令的元素。比如，粘贴命令要在TextBox上执行，那么TextBox就是粘贴命令的对象元素。

7.6.4 命令绑定（CommandBinding）

命令绑定就是把处理该命令的事件联系起来的一种机制。如下面的XAML把ApplicationCommands的打开文件命令（Open）和OpenCmdExecuted事件联系起来：

```
<Window.CommandBindings>
  <CommandBinding Command="ApplicationCommands.Open"
                  Executed="OpenCmdExecuted"
                  CanExecute="OpenCmdCanExecute"/>
</Window.CommandBindings>
```

在window类中，要定义OpenCmdExecuted事件处理程序，如：

```
void OpenCmdExecuted(object target, ExecutedRoutedEventArgs e)
{
    String command, targetobj;
    command = ((RoutedCommand)e.Command).Name;
    targetobj = ((FrameworkElement)target).Name;
    MessageBox.Show("The " + command +  " command has been invoked on
target object " + targetobj);
}
```

当发出ApplicationCommands.Open命令时，上述传递事件处理程序就被调用。

7.6.5 传递命令（Routed Command）

只要移植了ICommand的对象就是传递命令，传递命令像传递事项一样可以沿着WPF的元素树传递。实际上，它与通常的命令不同的是，当调用CanExecute方法时，传递命令产生CanExecute（冒泡）和PreviewCanExecute（潜入）两个事项；当调用Execute方法时，传递命令产生Executed（冒泡）和PreviewExecuted(潜入)两个传递事件。所以传递命令，实际上真正传递的依然是传递事件。理解了事件在元素树中传播，理解传递命令就是举手之劳。

7.6.6 WPF命令仓库（Command Repository）

WPF的命令仓库移植了五大类命令，它们分别是：ApplicationCommands、NavigationCommands、MediaCommands、EditingCommands和 ComponentCommands。这些命令都是传递命令，在应用程序中可以直接拿来使用，而且WPF中某些元素移植了这些命令的标准操作，如TextBox中的剪辑（Cut）、复制（Copy）和粘贴（Paste）的功能，使用起来非常方便。

你也可以根据自己的需要，开发出自己的命令仓库，比如笔者在通用电气的变电站应用软件中，就开发了规约属性、规约通道、节点映射等一系列命令。

好了，到目前为止我们对传递命令的方方面面有了一个概念上的认识。现在来看一个应用命令的实例。

```
<Window x:Class="Yingbao.Chapter7.CommandWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="WPF 命令" Height="150" Width="300">
    <Window.CommandBindings>
        <CommandBinding Command="Help"
       CanExecute="HelpCanExecute"
       Executed="HelpExecuted" />
    </Window.CommandBindings>
    <Window.InputBindings>
        <KeyBinding Command="Help" Key="H" Modifiers="Ctrl"/>
        <MouseBinding Command="Help" MouseAction="LeftDoubleClick" />
    </Window.InputBindings>
    <StackPanel>
        <Button Command="Help" Content="帮助" />
        <Button Content="自定义命令" x:Name="myCommandButton" />
    </StackPanel>
</Window>
```

这是一个简单视窗程序，在视窗中放入了两个按钮。第一个“帮助”按钮，其Command属性设为“Help”。Help为WPF内置命令。如前所述，ButtonBase移植了ICommandSource接口，所以我们可以把Button的Command属性链接到“Help”命令上：

```
Command = "Help"
```

Help命令是一个传递命令，当按下“帮助”命令，该命令沿着WPF的视觉树传播，若不处理该命令，那么这个命令就是什么都不做。就像气泡从水底浮上来，直到水面，然后破了消失了……这里，笔者在Window类里，使用命令绑定截获这个命令：

```
<Window.CommandBindings>
     <CommandBinding Command="Help"
    CanExecute="HelpCanExecute"
    Executed="HelpExecuted" />
 </Window.CommandBindings>
```

命令绑定很简单，使用**CommandBinding**类，并提供**CanExecute**和**Executed**两个方法，在后台**C#**程序里，需要移植这两个方法：

```
private void HelpCanExecute(object sender,
                      CanExecuteRoutedEventArgs e)
        {
            e.CanExecute = true;
            e.Handled = true;
        }

private void HelpExecuted(object sender,
                      ExecutedRoutedEventArgs e)
        {
            MessageBox.Show("有什么可以帮你？");
        }
```

笔者在这里只是示范，所以在**HelpExecuted**方法里，仅仅显示一个消息框，不过还可以把快捷键和某个命令联系起来，方法是使用**KeyBinding**，把命令和鼠标联系起来则要使用**MouseBinding**：

```
<Window.InputBindings>
        <KeyBinding Command="Help" Key="H" Modifiers="Ctrl"/>
        <MouseBinding Command="Help" MouseAction="LeftDoubleClick" />
</Window.InputBindings>
```

这里我把**Help**命令分别和“**Ctrl**”+“**H**”按键及双击鼠标左键联系了起来。

作为完整的示例，第二个按钮“自定义命令”和自定义命令的**myCommand**联系起来：

```
myCommandButton.Command = MyCommand;
```

同样可以在**CommandWindow**类中处理该命令。下面是完整的**C#**程序：

```
using System;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Input;
namespace Yingbao.Chapter7
{
    partial class CommandWindow
    {
        public static RoutedCommand MyCommand = new RoutedCommand();
        public CommandWindow()
        {
            InitializeComponent();

            CommandBinding cb = new CommandBinding(MyCommand,
                ExecuteMyCommand, CanExecuteMyCommand);
            this.CommandBindings.Add(cb);
```

```
            myCommandButton.Command = MyCommand;

            KeyGesture kg = new KeyGesture(Key.M,
                                ModifierKeys.Control);
            InputBinding ib = new InputBinding(MyCommand, kg);
            this.InputBindings.Add(ib);
        }

        private void HelpCanExecute(object sender,
                       CanExecuteRoutedEventArgs e)
        {
            e.CanExecute = true;
            e.Handled = true;
        }

        private void HelpExecuted(object sender,
                         ExecutedRoutedEventArgs e)
        {
            MessageBox.Show("有什么可以帮你? ");
        }

        private void CanExecuteMyCommand(object sender,
                         CanExecuteRoutedEventArgs e)
        {
            e.CanExecute = true;
            e.Handled = true;
        }

        private void ExecuteMyCommand(object sender,
                    ExecutedRoutedEventArgs e)
        {
            MessageBox.Show("这是你自己的命令。");
        }
    }
}
```

运行该程序，用鼠标单击“帮助”按钮或双击鼠标左键或按下“Ctrl”+“H”，会弹出一个“有什么可以帮助你?”的消息框。同样，若用鼠标单击“自定义命令”按钮，或“Ctrl”+“M”键，会弹出一个“这是你自己的命令。”的消息框，如图7-11所示。

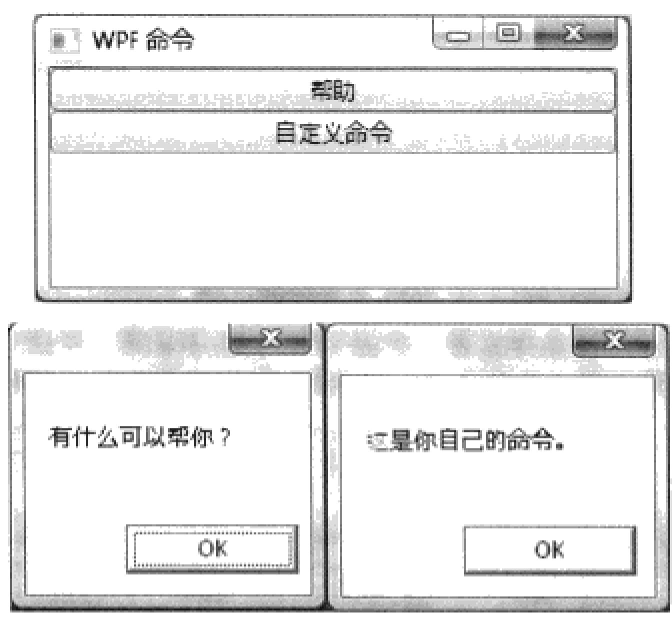


图7-11 WPF命令示例

若你的应用程序需要扩张WPF命令系统，可以从WPF的RoutedCommand派生出自己的命令，也可以开发自己的类并移植ICommand接口，只要遵守前面所说的规则即可。

7.7 本章小结

本章从研究WPF中的元素树开始，详细介绍了传递事件在元素树中的传播，最后总结了WPF中的传递命令。WPF中的元素树可以分为逻辑树和视觉树，这些树远比我们想象得要复杂。传递事件可以按照冒泡（Bubble）和潜入（Tunnel）两种方式在元素树上传播，虽然还有一种传递方式叫直接传递（Direct），但在笔者看来，这时事件根本没有在WPF的元素间流动，所以没有讨论这种情形。命令是一种使模块接口简单化的设计范例，WPF对命令的支持有两个方面，一是有了传递事件的支持，使得命令可以在WPF元素间“流动起来”；二是WPF提供了5大类命令的默认移植，使得应用程序使用起来更加方便。

第8章 资 源

几乎所有的计算机软件都要使用资源，这里的资源不是指使用CPU、内存和硬盘这样的硬件资源，而是程序中那些不能执行的部分。本章讨论WPF中的资源，从资源的定义和创建，到资源的访问和使用。需要指出的是，WPF资源包括风格和模板，但由于这两个概念在WPF中非常重要，将用专门的篇幅来介绍风格（第9章）和模板（第10章）。

8.1 资源定义及其类型

资源这个概念很早就有，恐怕可以追溯到在程序中使用常量。比如我们在C语言中定义：

```
const float pai = 3.14;
```

这里的常数pai其实就是最简单的一种资源，在以后的程序中，在遇到用圆周率的时候，就可以使用pai来替代：

```
float r = 1.2;
float d = 2*pai;
float area = pai* r * r;
```

这样做的好处是：当需要修改参数pai值时（如提高计算精度），只要在一个地方修改pai的值，比如把pai改为3.1415，而不必在程序里逐一修改3.14。当程序很大时，需要使用很多这样的常量，这个时候，就把它们放到一个单独的头文件中。凡是要用到这些常数的程序，只要在相应的程序中引用该头文件即可，这恐怕是资源出现的最早的形式。

后来在开发OS2和Window程序时，正式引入了资源的概念。如Borland的C++，Delphi把会话框分成两部分：一部分是程序的逻辑，另一部分则是对话框及其上面的控件的大小、位置、所用的字体、颜色等等信息。这些信息就是资源，它在程序中就是不变的常量。在Window程序中使用字符串或使用某个对话框，通常都把它们和常数联系起来，一旦要用到这些字符串或对话框时，只要引用相应的常数即可。早期的资源文件一般都嵌入到程序中，我们可以使用一些工具在dll或exe中读出这些资源。

所以，可以把资源看做是程序中可以和代码分开的部分。

随着计算机技术的飞速发展，资源的类型越来越丰富。如现在有图形SVG文件、Word文件、位图文件、PDF文件、png文件、jpg文件、html文件等。从某种意义上来说，整个WPF的XAML语言，其实就是一种资源描述语言，除非你在XAML中直接使用C#或Visual Basic语句，XAML主要用作描述UI元素的大小、位置、颜色等等信息用的这些东西不就是资源吗？

从软件国际化（Globalization/Localization）的角度，可以把资源分为两类:一类是需要随使用者所用的语言而改变的资源，如某个软件的英文版和中文版，其用户界面上的所有信息都要翻译成相应的语言；另一类是不需要随使用者所用的语言而改变的资源，如照片等。

从资源出现的方式来看，资源可以分为三类：第一类是嵌入在.NET的聚合块（Assembly）中；第二类是以单独文件出现在本地的存储介质上（通常是硬盘），如.bmp，SVG等文件；第三类是以单独文件存储在异地的存储介质上，如网络盘或网站的服务器上。

在WPF中，也可以根据定义资源的层次把资源分成聚合块资源（Assmebly资源）和XAML资源两大类。这种分类着眼于把XAML资源和其他类型的资源区分开来。

资源又可以分为在Visual Studio项目中管理的资源和不在项目中管理的资源两种。在项目中管理的资源，最终可以被编译到聚合块中，也可以不被编译到聚合块中，从而以独立的形式存在。不在Visual Studio项目中管理的资源，将会以独立的文件的形式存在，这种资源由程序员自己管理。

若资源文件在Visual Studio中管理，可以选择三种方式来编译项目中的资源，这三种方式是：内容（Content）、嵌入资源（Embedded Resource）和资源（Resource）。方法是：在Visual Studio的项目下，选择某个资源文件，按下鼠标右键，在弹出的菜单中选择"属性（Properties）"，Visual Studio就显示如图8-1所示的界面。

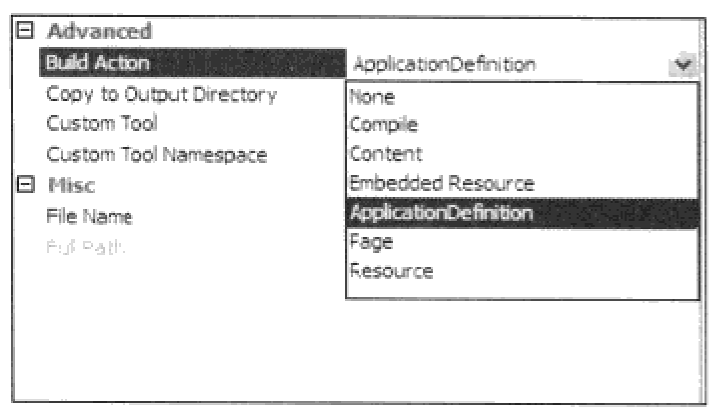


图8-1 在Visual Studio里选择Build Action

- 内容（Content）选择这种编译方式，其结果是资源文件仍然以独立的文件存在，Visual Studio会在所产生的聚合块中插入资源文件在介质（通常是硬盘）上的位置信息。
- 嵌入资源（Embedded Resource ） 选择这种编译方式会把资源嵌入到目标聚合块中，程序员可以使用ResourceManager来读出资源信息。嵌入资源是WPF前.NET程序中常用的编译方式，但这种编译方式在目标聚合块中无法用统一资源标识（Uri）来访问。在WPF中，应该使用Uri来访问资源，所以，应当尽量避免使用嵌入资源这种方式编译。
- 资源（Resource）这是Visual Studio 专门为WPF开发的功能。Visual Studio也会把资源编译到目标聚合块中，但资源可以用标准的Uri访问。

8.2 统一资源标识（Unified Resource Identifier）

WPF引入了统一资源标识（Unified Resource Identifier或Uri），Uri是基于OPC（Open Packaging Conventions）标准的一种识别资源的方式。使用Uri的优点是，可以使用统一模型来访问聚合块中、可知聚合块外、不可知聚合块外中的资源。

在Visual Studio中，XAML文件和资源文件一样会被编译到目标块中或所引用的目标块中。Uri可以访问下面基类XAML文件：

- Window类；
- Page类；
- PageFunction类；

- ResourceDictionary类（本章后面会用到）；
- FlowDocument类；
- UserControl类。

Uri还可以用统一的方式来加载数据文件、图像文件、.Net的聚合块以及聚合块里引用的其他聚合块等。

开放式封装规则（OPC）对软件包进行了高度抽象，它首先把软件包分成一个或多个逻辑部件（如图8-2所示）。一个软件包的Uri一般用下面的形式：

pack://authority/path

授权（authority）说明软件包的类型，路径（path）说明某个逻辑部件的位置。WPF支持两种authority：一种是application，另一种是siteoforigin：

pack://application:///

pack://siteoforigin:///

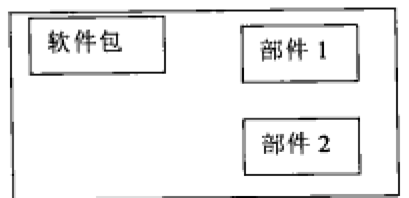


图8-2 OPC把软件包分成多个逻辑部件

授权application:///用作访问在编译时已经知道的文件，包括Visual Studio中管理的文件，这些文件可以是用内容（Content）编译的，也可以是用资源（Resource）编译的，但不能是嵌入资源（Embedded Resource）编译的。

授权siteoforigin:/// 用作访问在编译不知道的文件，前面提到，应用程序需要访问不在Visual Studio中管理的文件。siteoforigin容易和访问互联网相混淆，虽然它确实支持访问互联网上的资源，但其主要的目的是支持访问在编译时不知道的资源文件，标识应用程序所在的位置。如你的应用程序在：

C:\Myfolder\subfolder1

那么授权siteoforigin就是C:\Myfolder\subfolder1。若你的应用程序在：

http://www.microsoft.com/something/

那么授权siteoforigin就是http://www.microsoft.com/something/。

在WPF里面需要把位于application:和siteoforigin后面的三个///变换为三个逗号,,,。

构建路径较容易理解，下面笔者结合实例来说明如何构建Uri。

- 资源在应用程序的聚合块中，这是最简单的情况。
  - ➢ 授权：application:///
  - ➢ 路径：资源的名字，包括相对于应用程序的路径。

如：application:,,,/grass.jpg

application:,,,/myfolder/grass.jpg

第二个grass.jpg在Visual Studio的项目中位于/myfolder的子目录下。

- 资源位于应用程序所引用的聚合块中。
  - 授权：application:///
  - 路径：资源被编译到集合块中，要应用其他集合块中的资源，需要给出集合块的信息。这时路径应具有下面的形式：

集合块的名字[;version][;publicKey]；component/相对路径

其中括号中的内容是可选项。例如：

pack://application:,,,/myphoto.dll; v1.0.0.0 component/fm/mapple.jpg

- 资源不在聚合块中，但在Visual Studio项目管理器中管理。
  - 授权：application:///
- 路径：资源在编译时选择的是“content”，路径要包括资源文件名及相对于执行聚合块的路径。

例如：pack://application:,,,/mycontent.xaml

pack://application:,,,/subfolder/mycontent.xaml

- 资源不在聚合块中，也不在Visual Studio的项目管理器中管理。
  - 授权： siteoforigin:///
  - 路径： 包括资源的文件名和相对于执行聚合块的路径。

例如：pack://siteoforigin:,,,/Subfolder/myPhoto.jpg

如何在XAML中使用Uri的例子如下。

```
<Window x:Class="UriResourceTest.ResourceTestWin"
   xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="UriResourceTest" Height="600" Width="300">
  <StackPanel >
    <Button Height ="100">
      <Image Source ="pack://application:,,,/flower1.jpg" />
    </Button>
    <Button Height ="100">
      <Image Source ="pack://application:,,,/Resource/snail.jpg" />
    </Button>
    <Button Height ="100">
      <Image Source ="pack://application:,,,/WPFResourceLib;
                     component/wpfflower.jpg"/>
    </Button>
    <Button Height ="100">
      <Image Source ="pack://siteoforigin:,,,/Resource/
```

```
                    ResFlower.jpg"/>
    </Button>
  </StackPanel>
</Window>
```

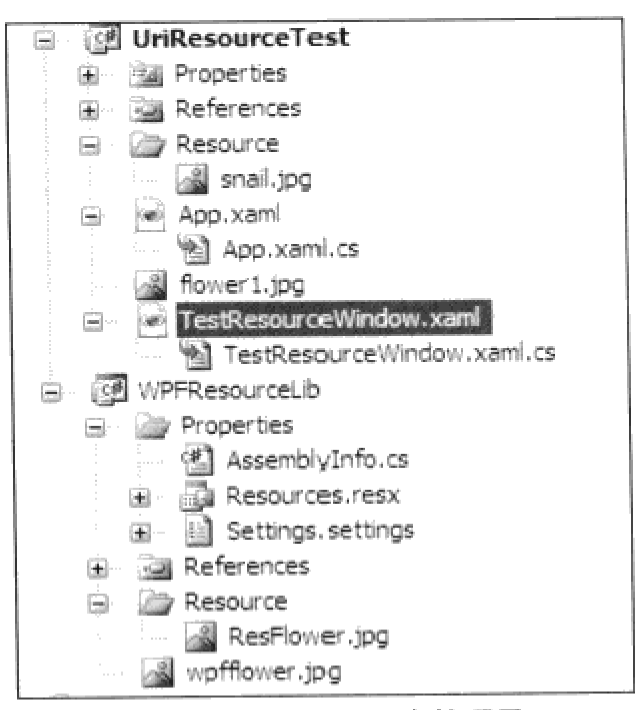


图8-3 Visual Studio中的项目

在上面的例子中，我创建了两个项目：一个是WPFResourceLib，这是一个Dll；另一个是UriResourceTest，这是一个可执行文件，如图8-3所示。

在UriResourceTest聚合块中，snail.jpg放在“Resource”子目录下，笔者用“Content”进行编译，结果snail.jpg被复制到UriResourceTest下面Resource子目录中。flow1.jpg放在UriResourceTest目录下，用“Resource”编译，结果嵌在UriResourceTest.exe聚合块中。

在WPFResourceLib聚合块中，wpfflower.jpg放在WPFResourceLib的根目录下，使用“Resource”进行编译。ResFlower.jpg放在“Resource”子目录下，使用“Content”进行编译。

从这个例子可以看出，Uri对各种资源的配置进行了统一处理。由于.NET支持xcopy方式的安装，所以资源文件在硬盘上的绝对位置并不重要，重要的是其相对于运行聚合块的相对位置。Uri正是基于这一情况，强调这种相对位置。当然，如果你一定要用绝对路径，Uri也是支持的。不过，如果要把你的应用程序从C盘复制到D盘，那就要修改程序了！

图8-4示出了上面的程序运行的结果。

顺便提一下，上面的照片是笔者2007年夏天在温哥华的Buchart花园自己拍摄的（http://www.butchartgardens.com）。这个花园由于所处的自然环境优越，加上精心设计和护理，里面的鲜花娇艳欲滴。Buchart花园的美是一种让你震撼的美，一种梦中也不曾有过的美，一种怀疑自己是否在地球上的美！

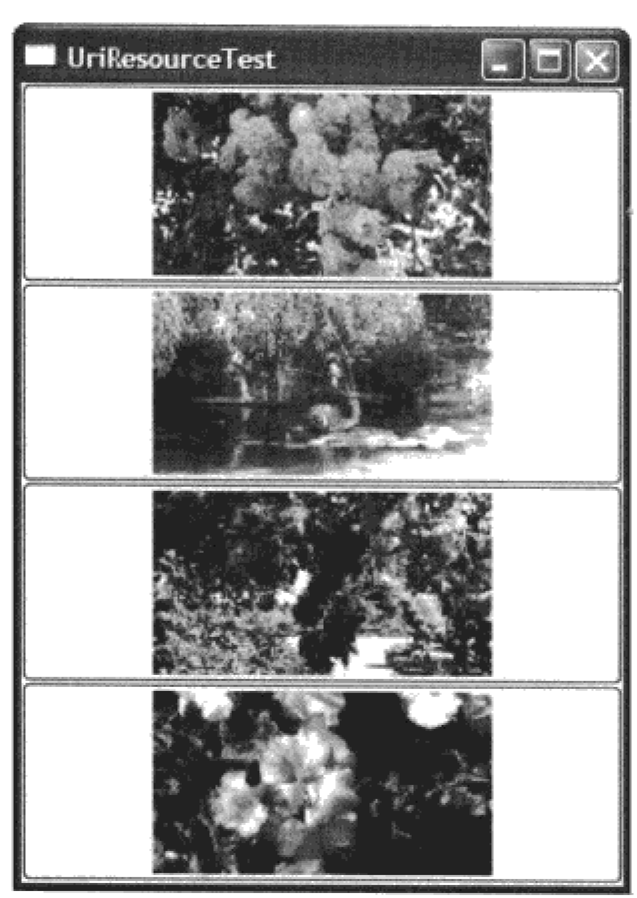


图8-4 使用Uri访问资源文件

8.3 .NET开发平台对资源国际化的支持

随着软件在越来越多的国家使用，应用程序从设计时就要考虑对多种语言的支持。.NET开发平台从一开始就支持应用软件的国际化，其中引入了一个文化（Culture）的概念。文化可以唯一标识应用程序用在某个区域时所要做的改变，这个概念比过去的Locale和语言要精确。比如，你的软件用在加拿大，就需要支持两种文化：en-CA和fr-CA。en表示英语，CA表示加拿大，fr表示法语。这是因为加拿大的英语和美国英语有所差异，加拿大的法语和法国的法语差异更大。再比如中国内地的中文使用的是简体字，而中国台湾和香港等地区使用的则是繁体字，其中香港又由于发音不同而区分于台湾等地区的中文。所以，在.NET里中文有三个文化：zh-CN、zh-HK、zh-TW。所以，所谓的文化实际上是语言和地区的组合。.NET目前所支持的文化，可参见微软提供的文化列表：

http://msdn.microsoft.com/en-us/library/system.globalization.cultureinfo.aspx

.NET的线程类Thread中有两个属性：CurrentCulture和CurrentUICulture. 这两个属性的类型都是CultureInfo，通过设置这两个属性，可以告诉.NET开发平台应用程序当前所处的文化环境。通常应用程序的做法有两种，一是根据操作系统的设置来确定文化；二是当用户登录系统时，确认用户身份，然后设定文化，在这种情况下，需要事先把用户的文化信息设定在数据库中；第二种情况尤其适合互联网应用程序，因为在服务器端的程序无法预先知道客户端操作系统的设置（当然你可以从Http的请求中获取用户浏览器的信息，但需要考虑不同的浏览器和不同的版本等多种情况）。

若使用操作系统的设置来确定文化，通常不必做什么特殊的工作。.NET开发平台会自动根据操作系统的设置来确定线程中的CurrentCulture和CurrentUICulture。

前面笔者在资源的分类中已提到，资源按照是否要随着文化的改变而改变而分成两类资源，本节

我要讨论如何创建并使用随文化的改变而改变的资源。

8.3.1 WinForm下的资源管理

现在来看一个简单的例子，在这个例子中，笔者演示了三种文化：en-CA、en-US和zh-CN，即加拿大英语、美国英语和中文。通过这个例子，可以看到使用.NET的ResourceManager管理与文化相关的资源的方式。

在这个简单的实例中，笔者创建了一个用于用户登录的窗口程序。如图8-5所示。当用户选择某个文化时，希望本视窗上所显示的文字也发生相应改变。为此，笔者在这个视窗上加了一个组合框控件，用于选择文化。在实际的应用程序中，可能由系统管理员来管理用户的信息，如把用户名、密码、文化等信息存储在数据库中。

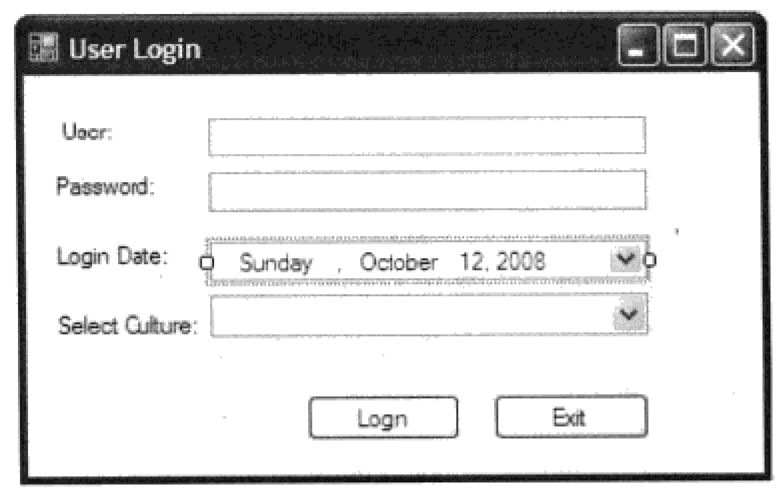
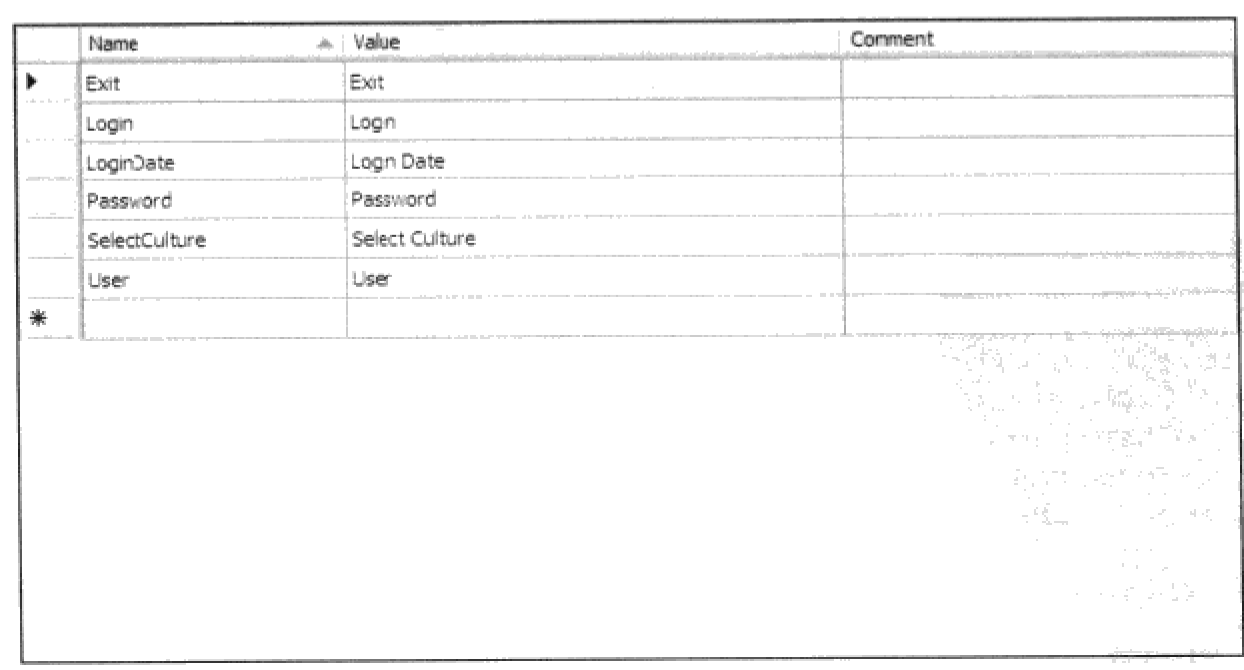


图8-5 用户登录的视窗

然后，笔者创建一个文化为中性的资源文件AppSprings.resx。方法是在visual studio的项目管理器中，选择“Add New Item...”菜单条目，然后选择“Resources File”模板，如图8-6所示。



Name	Value	Comment
Exit	Exit	
Login	Login	
LoginDate	Login Date	
Password	Password	
SelectCulture	Select Culture	
User	User	

图8-6 在VisualStudio中创建中性资源（AppStrings.resx）

用同样的方式，可以创建中国内地中文环境下的资源文件。其文件名的命名规则是在中性资源文

件名后加上文化标识符，由于中国内地的中文文化标识符是：zh-CN，所以，笔者把该资源文件命名为AppStrings.zh-CN.resx，在Visual Studio中创建中文资源的情况如图8-7所示。

Name	Value	Comment
Exit	退出	
Login	登录	
LoginDate	登录日期	
Password	口令	
SelectCulture	选择文化	
User	用户名	

图8-7　在Visual Studio中创建中文资源（AppStrings.zh-CN.resx）

在选择文化的组合框中，笔者加入了en-CA、en-US和zh-CN三种文化，当用户选择其中的一个文化时，在组合框的事件处理程序中设定当前线程的CurrentCulture和CurrentUICulture两个属性：

```
string selectCulture = this.cmbCulture.SelectedItem.ToString();
Thread.CurrentThread.CurrentCulture =

CultureInfo.CreateSpecificCulture(selectCulture);
Thread.CurrentThread.CurrentUICulture = new
CultureInfo(selectCulture);
 SetUIText();
```

在SetUIText（）函数里，笔者使用ResourceManager的GetString方法来获取当前文化下的字符串值，然后把它放到相应的控件中。下面是完整的处理程序：

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Text;
using System.Windows.Forms;
using System.Globalization;
using System.Threading;
using System.Resources;
namespace Yingbao.Chapter8.LocalLizeResource
{
    public partial class LoginForm : Form
    {
        public LoginForm()
        {
            InitializeComponent();
        }
```

```
        private void btnLogin_Click(object sender, EventArgs e)
        {
            this.Close();
        }

        private void cmbCulture_SelectedIndexChanged(object sender,
                 EventArgs e)
        {
         string selectCulture =
                this.cmbCulture.SelectedItem.ToString();
         Thread.CurrentThread.CurrentCulture =
         CultureInfo.CreateSpecificCulture(selectCulture);
         Thread.CurrentThread.CurrentUICulture = new
                CultureInfo(selectCulture);
         SetUIText();
        }

        private void SetUIText()
        {
            ResourceManager rm = new
                  ResourceManager("LocalLizeResource.AppStrings",
                  Assembly.GetExecutingAssembly() );
            lblUser.Text = rm.GetString("User");
            lblPassword.Text = rm.GetString("Password");
            lblLoginDate.Text = rm.GetString("LoginDate");
            lblSelectCulture.Text = rm.GetString("SelectCulture");
            btnExit.Text = rm.GetString("Exit");
            btnLogin.Text = rm.GetString("Login");
        }
    }
}
```

当选择了zh-CN文化时，视窗便自动成为中文的了（如图8-8所示）。

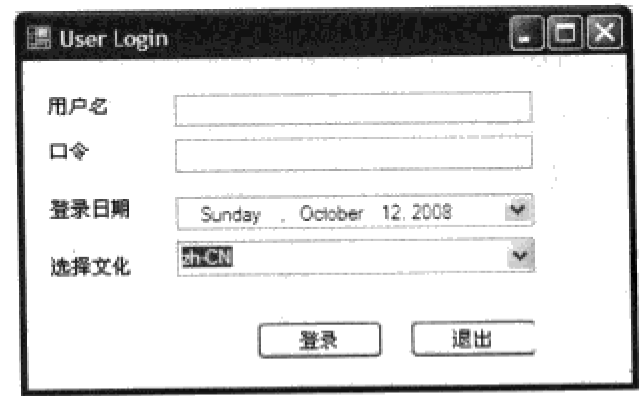


图8-8 当选择zh-CN时，视窗显示出相应的中文

.NET资源管理器可以自动根据文化选择相应的资源。资源管理器（ResourceManager）选择资源的规则是：

- 首先看当前文化下，是否存在资源文件，若有，在该资源文件中获取相应的资源；
- 若没有，则看是否存在该语言的资源文件，若有则使用该资源文件；
- 若没有，则使用中性资源文件。

以上面的例子为例，从理论上来讲，应该创建下面几个资源文件：

- AppStrings.resx（中性资源文件）；
- AppStrings.en.resx（英文资源文件）；
- AppStrings.en-US.resx（美国英文资源文件）；
- AppStrings.en-CA.resx（加拿大英文资源文件）；
- AppStrings.zh.resx（中文资源文件）；
- AppStrings.zh-CN.resx（中华人民共和国资源文件）。

实际上，当en-CA、en-US、en和中性资源差不多时，可以不用建立这些文化或语言的资源文件（如本例就没有创建这些资源）。

最后需要指出：.NET的DateTimePicker控件，并不自动随着文化的改变而改变，它实际上与老的区域设置相关，这一点实在是一个遗憾。希望微软将来能解决这个问题。

8.3.2 用XAML创建本地资源

笔者在前面提过，XAML实际上是一种描述资源的语言（除了在XAML中直接加入C#或VB语句之外），所以，同样有把XAML创建的资源国际化的要求。

XAML编译器把XAML文件编译成BAML资源，.NET的资源管理器(ResourceManager)用和8.3.1节读取字符串资源一样的规则来读取XAML资源。和使用字符串资源的规则一样，资源管理器使用BAML资源的顺序为：文化BAML→语言BAML→中性BAML。

遗憾的是，Visual Studio并没有提供创建本地BAML资源的工具。但在SDK中，微软提供了LocBaml工具（http://msdn.microsoft.com/en-us/library/ms771568.aspx）。LocBaml是以源程序的方式提供的，需要先编译才能使用。此外，还有第三方为BAML资源本地化提供的工具，这些工具不是免费的，如Sisulizer(www.sisulizer . com)。下面的讨论基于使用免费的LocBaml。

- 第一步，在VisualStudio里创建一个WPF视窗应用程序项目:“ResourceUseBaml”。

我们在Window元素里加入两个元素：StackPanel和Button。然后给每个元素加上x:Uid属性，Uid的值必须在整个应用程序中不能重复（唯一性），这个简单的XAML程序如下：

```
<Window x:Uid="Window_1"
    x:Class="Yingbao.Chapter8.ResourceUseBaml.MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="ResourceUseBaml" Height="150" Width="300" >
  <StackPanel x:Uid="StackPanel_1">
       <Button  x:Uid="button1" Margin="30,20,35,26" Name="button1">
         Hello World
       </Button>
     </StackPanel>
</Window>
```

如果不想自己加这个Uid，可以在DOS窗口中运行下面的命令：

```
msbuild /t:updateuid ResourceUseBaml.csproj
```

运行这个命令后，msbuild自动给XAML中的每一个元素加上x:Uid。

- 第二步，在文本编辑模式下打开ResourceUseBaml.csproj文件，在该文件中加入UICulture：

```
<Project DefaultTargets="Build"
     xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
  <PropertyGroup>
    <Configuration Condition=" '$(Configuration)' == ''
">Debug</Configuration>
    <Platform Condition=" '$(Platform)' == '' ">AnyCPU</Platform>
    <UICulture>en-CA</UICulture>
    <ProjectGuid>{C768606D-5E68-41A7-A60F-70D17C8FA4F0}</ProjectGuid>
......
</Project>
```

然后在visual studio中重新加载ResourceUseBaml.csproj。

- 第三步，把项目的NeutralResourcesLanguage属性设为你的应用程序主要支持的语言，比如笔者把ResourceUseBaml.csproj设为“en-CA”。在项目里打开AssemblyInfo.cs文件，

```
[assembly: NeutralResourcesLanguage("en-CA",
UltimateResourceFallbackLocation.Satellite)]
```

实际上，VisualStudio在AsemblyInfo.cs文件中已经含有这句话，要做的只是把原来的注释符号“//”拿掉，把原来的“en-US”改为“en-CA”。

NeutralResourcesKanguage属性告诉CLR寻找当前文化下的资源DLL的方法，以及在找不到特定文化资源时所要采取的措施。当笔者把该属性设为“en-CA”时，相当于告诉CLR，如找不到特定文化相对应的资源，就使用en-CA文化资源。

现在，把项目重新编译一遍，可以看到Visual Studio在Bin目录下产生了一个新的目录“en-CA”，其中含有一个资源文件：

ResourceUseBaml.resources.dll

- 第四步，在Application.Resource中加入要本地化的XAML文件：

```
<Application x:Uid="Application_1" x:Class="ResourceUseBaml.App"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    StartupUri="MainWindow.xaml">
    <Application.Resources>
      <ResourceDictionary>
        <ResourceDictionary.MergedDictionaries>
          <ResourceDictionary Source="AppString.xaml" />
        </ResourceDictionary.MergedDictionaries>
      </ResourceDictionary>
    </Application.Resources>
</Application>
```

笔者在这里加入了AppString.xaml文件，把它加在ResourceDictionary.MergedDictionaries里面目的

是告诉WPF，把AppString.xaml和其他资源整合到一块。

- 第五步，使用LocBaml.exe工具把ResourceUseBaml.resources.dll中的资源转换成.csv文件：

LocBaml ResourceUseBaml.resources.dll /parse /out：resTrans.csv

这个命令产生resTrans.csv文件。这个文件可以在微软的电子表格程序中打开并编辑。打开该文件，把Window.Title改为“使用LocBaml本地化资源”，把Button$Content改为“同胞们，你们好”。

- 第六步，产生中文环境下的资源Dll。运行下面的命令：

LocBaml /generate ResourceUseBaml.resources.dll /cul:en-CA /trans:ResTrans.csv /out：/cul:zh-CN

就会在zh-CN子目录下产生另一个资源聚合块（ResourceUseBaml.resources-.dll），这个聚合块中含有中文信息。

- 第七步，切换到中文文化下。方法和8.3.1节所做的一样，我们把当前线程的CurrentCulture属性设为“zh-CH”：

```
   string culture = "zh-CN";
   Thread.CurrentThread.CurrentCulture =
CultureInfo.CreateSpecificCulture(culture);
  Thread.CurrentThread.CurrentUICulture = CultureInfo(culture);
```

- 第八步，把按钮的内容和资源联系起来：

```
<Button  x:Uid="button1" Margin="30,20,35,26" Name="button1"
                                   Content="{DynamicResource
buttonText}"/>
```

这就是XAML的本土化/国际化过程。在经过这些步骤后，笔者认为使用这么多的步骤来完成XAML的本土化太麻烦！觉得微软有义务提供更好的工具。虽然在这里提供了常用的步骤，但笔者不会这么去用。这里掩藏了太多的东西，一旦哪个步骤除了问题，哪怕是一点点小问题，都不是很容易找出根源的。

说实话，笔者宁愿使用WinForm下的资源管理方式，来用C#代码对XAML进行控制，一切都在自己的把握中……

8.4 WPF元素中定义的资源

前面讨论的资源都是以独立的文件的方式存在，现在要讨论WPF中的另一类资源。这类资源在WPF元素中定义，元素通常是Window、Page、Control或Application。这些资源只能在所定义的元素或该元素的内涵元素中才能使用，它们在应用程序中只创建一次。

这类资源存储在ResourceDictionary中，ResourceDictionary含有哈希表，可以使用关键字key来存储和访问资源。

在三大基类元素：FrameworkElement、FrameworkContentElement和Applic-ation中都含有一个Resources属性，这个属性的类型为ResourceDictionary。

所有从FrameworkElement或FrameworkContentElement中派生出来的元素都可以在Resources属性中加上资源，其语法如下：

```
<Window>
   <Window.Resources>
        <ElementType x:key="myElement">
                              资源值
        </ElementType>
   </Window.Resources>
</Window>
```

在C#中定义资源的语法为：

```
Window.Resources.Add("myElement",资源);
```

WPF使用扩展标识（Markup Extension）方法来访问资源，WPF定义了四大类扩展标识，即：静态资源扩展标识（StaticResource markup extension）、动态扩展标识（DynamicResource markup extension）、数据绑定扩展标识和模板扩展标识。笔者在第2章XAML语言中曾经提到过XAML的静态资源和动态资源扩展，本章介绍静态资源扩张标识和动态资源扩张标识的实际应用，数据绑定和模板分别在第10章模板和第11章数据绑定中介绍。

8.4.1 静态资源（StaticResource）

静态资源扩张标识是XAML对XML语法的扩展，WPF内部使用StaticResourceExtension类对XMAL进行解释，资源关键字是其唯一属性。当程序员在XAML中设定资源关键字时，StaticResourceExtension类负责访问资源值。

访问静态资源值有两种方法，一种是使用属性：

<StaticResource ResourceKey="LargeFont"/>

另一种是使用{}:

FontSize= "{StaticResource LargeFont}"

下面的例子分别示出了这两种方法：

```
<Window x:Class="Yingbao.Chapter8.StaticResourceExample.MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:sys="clr-namespace:System;assembly=mscorlib"
        Title="使用静态资源扩张的例子" Height="100" Width="400">
  <Window.Resources>
    <SolidColorBrush x:Key ="YellowBrush">Yellow</SolidColorBrush>
    <SolidColorBrush x:Key ="RedBrush">Red</SolidColorBrush>
    <sys:Double x:Key ="LargeFont">20</sys:Double>
    <sys:Double x:Key ="SmallFont">10</sys:Double>
  </Window.Resources>
  <StackPanel >
    <Button Height ="30">
      <Button.Background>
        <StaticResource ResourceKey ="RedBrush"/>
      </Button.Background>
      <Button.FontSize>
        <StaticResource ResourceKey ="LargeFont"/>
```

```
        </Button.FontSize>
        红色大字按钮
    </Button>
    <Button Height ="30" Background ="{StaticResource YellowBrush}"
         FontSize ="{StaticResource SmallFont}">黄色小字按钮
    </Button>
  </StackPanel>
</Window>
```

在这段程序中，笔者首先在Window.Resource中定义了红色和黄色画刷，还有大小为20的大字体和大小为10的小字体。然后，在这些资源在按钮中使用两种方法加以应用。

上面程序运行的结果如图8-9所示。

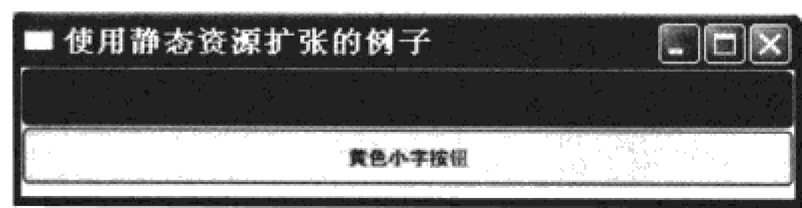


图8-9 使用静态资源扩展来访问静态资源

XAML用大括号{}来表示其中的字符串是扩展标识，那么如果如何显示带有字符串的字符呢？比如说：

<TextBlock Text ="{带大括号的字符串}"></TextBlock>

这个语句会产生下面的错误：

"error MC3074: The tag '带大括号的字符串' does not exist in XML namespace 'http://schemas.microsoft.com/winfx/2006/xaml/presentation'."

解决的方法是在大括号前再加上一对大括号，我在第2章XAML语言中曾经提过：

<TextBlock Text ="{}{带大括号的字符串}"></TextBlock>

这和C++或C#中处理"\"是类似的原理。

8.4.2 资源的作用范围

前面提到，FramewrkElement和FrameworkContentElement中都含有Resources属性，而WPF中的UI元素都是从这两个类中派生出来的；换句话说，可以在任何UI元素内加入资源。WPF对其中的唯一要求是哈希表中的关键字Key必须是唯一的，在UI元素中使用关键字Key来引用资源。虽然在同一个元素的资源里，不能使用同一个关键字来定义不同的资源，但在不同的元素里，却可以使用其他元素资源中已经使用过的关键字。那么，在用关键字引用资源时，究竟是引用哪个资源呢？

把前面的例子稍微修改一下，在StackPanel中加入资源：

```
    <StackPanel.Resources>
            <SolidColorBrush x:Key =
"YellowBrush">Blue</SolidColorBrush>
    </StackPanel.Resources>
```

这里笔者定义了关键字为"YellowBrush"的填充画刷，与Window中的一个画刷的关键字是相同

的，但是所用的颜色不同，由于YellowBrush位于两个不同的UI元素中，编译程序并不产生任何错误。请看下面的XAML：

```
<Window x:Class="Yingbao.Chapter8.StaticResourceExample.MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:sys="clr-namespace:System;assembly=mscorlib"
        Title="使用静态资源扩张的例子" Height="100" Width="400">
  <Window.Resources>
    <SolidColorBrush x:Key ="YellowBrush">Yellow</SolidColorBrush>
    <SolidColorBrush x:Key ="RedBrush">Red</SolidColorBrush>
    <sys:Double x:Key ="LargeFont">20</sys:Double>
    <sys:Double x:Key ="SmallFont">10</sys:Double>
  </Window.Resources>
  <StackPanel >
    <StackPanel.Resources>
      <SolidColorBrush x:Key ="YellowBrush">Blue</SolidColorBrush>
    </StackPanel.Resources>
    <Button Height ="30">
      <Button.Background>
        <StaticResource ResourceKey ="RedBrush"/>
      </Button.Background>
      <Button.FontSize>
        <StaticResource ResourceKey ="LargeFont"/>
      </Button.FontSize>
      红色大字按钮
    </Button>
    <Button Height ="30" Background ="{StaticResource YellowBrush}"
         FontSize ="{StaticResource SmallFont}">黄色小字按钮
    </Button>
  </StackPanel>
</Window>
```

这段程序的运行结果如204页中的图8-9所示。由图8-9可见，第二个按钮现在使用的是StackPanel中定义的资源，而不再是Window中定义的资源了。

资源在元素树中的传递和覆盖的规则与相关属性的传递和覆盖的规则一样：

- 资源从树根元素向树枝元素传递；
- 若资源在传递路径中被重新定义，则原来的资源就不再向下面的树杈传递，转而传递新定义的元素。

需要注意的是，我这里说的"传递"是一种通俗的说法。与传递事件真的在元素树中动态传递不同，资源的传递实际上发生在引用该资源的地方，WPF从该元素沿着元素树向树根寻找某个资源的过程。一旦找到了某个资源，WPF就使用该资源，而不再寻找了，虽然表面上看似乎是在"传递"和"覆盖"，但其内部的机理却大不相同。

8.4.3 静态扩展标识（Static markup extension）

有时候在XAML中需要访问C#类中的静态属性（static property）或静态域（Static Field），这

时，需要用到静态扩展标识。在XAML中使用静态扩展标识的语法为：

```
<object>
  <object.property>
    <x:Static Member="prefix:typeName.staticMemberName" .../>
  </object.property>
</object>
```

需要注意的是，等号两边属性的类型必须一致，或者在WPF中存在默认的转化。如CaptionHeight类型和Button.Height（从FrameworkElement中继承的）的类型都为double。

下面是一个使用静态扩展标识的例子：

```
<Window x:Class="Yingbao.Chapter8.UseStaticExtension.MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:s="clr-namespace:System;assembly=mscorlib"
    xmlns:src="clr-namespace:Yingbao.Chapter8.UseStaticExtension"
    Title="使用静态扩展标识" Height="150" Width="600">
  <StackPanel>
    <TextBlock >
      <Label>你的机器名：</Label>
      <Label Content ="{x:Static s:Environment.MachineName}" />
      <LineBreak />
      <Label FontSize ="{x:Static src:MainWindow.largeFont}">用户名：
             </Label>
      <Label FontSize ="{x:Static src:MainWindow.largeFont}"
                  Content ="{x:Static s:Environment.UserName}" />
      <LineBreak />
      <Label Foreground ="Purple" FontSize ="{x:Static
          src:MainWindow.largeFont}">现在时间：</Label>
      <Label Foreground ="Purple" FontSize ="{x:Static
                  src:MainWindow.largeFont}"
                  Content ="{x:Static s:DateTime.Now}"/>
    </TextBlock>
  </StackPanel>
</Window>
```

在这个例子中，笔者示出了两种典型的静态扩展标识用法：一种是使用.NET平台所提供的静态属性；另一种是使用自己定义的静态域。

- 使用.NET平台所提供的静态属性

我们知道.NET类Environment中提供了应用程序所在的机器的基本信息：如操作系统的版本、当前目录、命令行等。笔者在上面的这段XAML中显示了本机的机器名。要使用Environment中的属性，首先需要在项目中引用mscorlib，然后在XAML里引入Environment所处的命名空间：

```
xmlns:s="clr-namespace:System;assembly=mscorlib"
```

- 使用自定义的静态域（静态属性）

首先笔者在MainWindow类里定义一个静态域：

```
public static readonly double largeFont = 24;
```

这是一个只读浮点数。注意，一定要把这个域设为public，否则静态扩展标识类无法访问静态域。

然后在XAML中应用该域：

```
<Label FontSize ="{x:Static src:MainWindow.largeFont}">用户名：</Label>
<Label FontSize ="{x:Static src:MainWindow.largeFont}"
                  Content ="{x:Static s:Environment.UserName}" />
```

有意思的是，笔者原来以为在该XAML自己的类中定义的域largeFont，在XAML中应该可以直接使用域名访问了：

```
<Label FontSize ="{x:Static largeFont}">用户名：</Label>
```

但在编译时产生类型错误信息。通过研究，发现编译器实际上是根据WPF的Schema进行校验的，WPF的Schema当然不知道笔者定义的域在XAML自己的类中了。所以，我们不仅要在largeFont域前加上类名字，还要在XAML中引入自定义类所处的命名空间。

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Data;
using System.Windows.Documents;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Media.Imaging;
using System.Windows.Shapes;
namespace Yingbao.Chapter8.UseStaticExtension
{
    public partial class MainWindow : System.Windows.Window
    {
        public static readonly double largeFont = 24;

        public MainWindow()
        {
            InitializeComponent();
        }

    }
}
```

最后笔者在XAML中加入显示当前时间：

```
<Label Foreground ="Purple" FontSize ="{x:Static
       src:MainWindow.largeFont}"
       Content ="{x:Static s:DateTime.Now}"/>
```

上面这段程序的运行结果如图8-10所示。

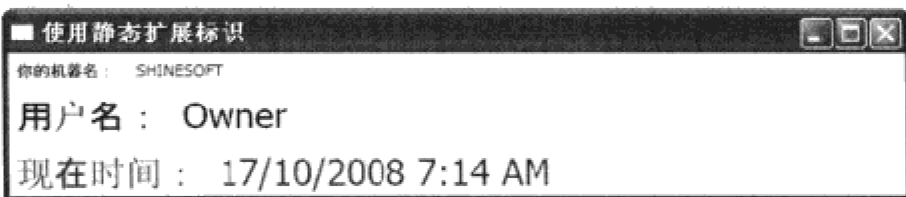


图8-10 使用静态扩展

可以看到，上面的时间并不变化。其原因在于用的是静态扩展，静态扩展的最大特点是只在程序加载时读取，在此之后与所读取的属性间没有关系。

8.4.4 动态资源扩展标识（DynamicResource Markup Extension）

WPF中的静态资源只在应用程序加载的时候创建一次，以后就不再改变。有的时候希望资源可以随着某些条件的变化而变化，比如窗口的标题栏，希望在用户改变设置的时候（如在xp中，从ControlPanel->Display中改变视窗的主题/皮肤（Theme/Skin）），应用程序也能随之改变，这时就要用到动态资源标识。

在XAML中使用动态资源扩展标识的语法与静态资源扩展标识一样有两种，一种是使用DynamicResource标识：

```
<object>
  <object.property>
    <DynamicResource ResourceKey="key" .../>
  </object.property>
</object>
```

另一种是使用大括号{}:

```
<object property="{DynamicResource ResourceKey=key}" .../>
```

两者的效果是一样的。只需要记住一点：访问动态资源与访问静态资源一样，需要用资源关键字——ResourceKey，这其实是访问哈希表的最大特征。

和静态资源扩展不同，动态资源在编译时并不把资源赋予相关属性，而是在运行时刻根据需要连接资源。

下面是一个使用动态资源的例子，从而体会当所连接的属性发生改变时所产生的影响。

```
<Window x:Class="Yingbao.Chapter8.UseDynamicResource.MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:s="clr-namespace:System;assembly=mscorlib"
    Title="使用动态资源扩展" Height="300" Width="300"
    FontSize ="14" >
  <Window.Resources>
    <LinearGradientBrush x:Key="myBrush1" StartPoint ="0  0" EndPoint
="1  1">
        <GradientStop Color="{DynamicResource {x:Static
SystemColors.ActiveCaptionColorKey}}" Offset="0" />
        <GradientStop Color="{DynamicResource {x:Static
SystemColors.InactiveCaptionColorKey}}"  Offset="1" />
    </LinearGradientBrush>
```

```
    <SolidColorBrush x:Key="myBrush2"  Color ="{DynamicResource
{x:Static SystemColors.InactiveCaptionColorKey}}"/>
  </Window.Resources>
 <StackPanel>
   <TextBlock Foreground ="{StaticResource myBrush1}">
     <Bold FontSize ="20">窗下</Bold>
     <LineBreak/>   <LineBreak/>
     <Italic> 洛夫</Italic><LineBreak/>   <LineBreak/>
     当暮色装饰着雨后的窗子<LineBreak/>
     我便从这里探测出远山的深度<LineBreak/>
     </TextBlock>
     <TextBlock Foreground ="{StaticResource myBrush2}">
     在窗玻璃上呵一口气<LineBreak/>
     再用手指画一条长长的小路<LineBreak/>
     以及小路尽头的<LineBreak/>
     一个背影<LineBreak/><LineBreak/><LineBreak/>
     有人从雨中而去
   </TextBlock>
  </StackPanel>
</Window>
```

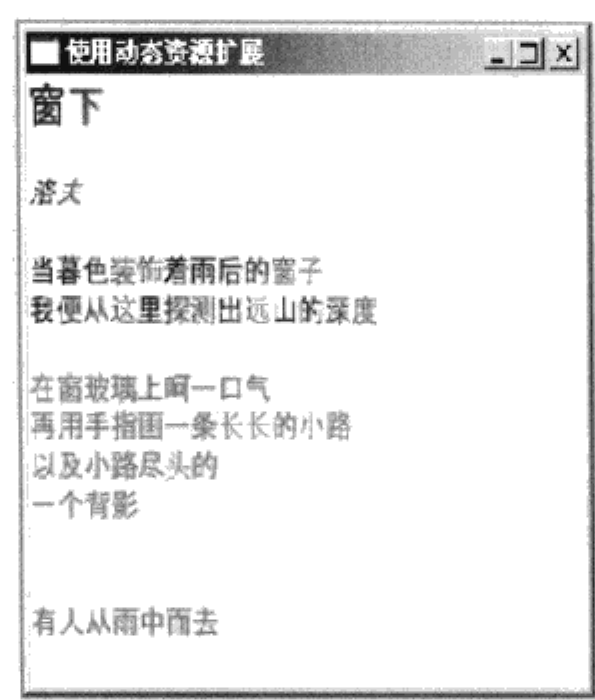
在这段程序中，笔者在Window.Resources下面定义了两个画刷，一个是线性梯度画刷（LinearGradientBrush第4章），另一个是实心填充画刷。与第4章WPF中的属性系统不同，这里使用动态资源扩展标识来设置颜色。在使用画刷的时候，笔者用的是静态资源扩展：

```
<TextBlock Foreground ="{StaticResource myBrush1}">
<TextBlock Foreground ="{StaticResource myBrush2}">
```

现在运行这段程序，在应用程序运行的情况下，通过操作系统的控制面板（Control Panel）->显示（Display）->选择外观（Appearance），在窗口和按键的组合框中选择“Windows XP Style”时，这段程序的显示结果如图8-11 a）所示；在窗口和按键的组合框中选择“Windows Classic Style”时，这段程序的显示结果如图8-11 b）所示。由此可见，在使用动态资源扩展的情况下，应用程序中所用的相应的属性发生了相应的变化。

笔者在这段程序中使用了中国台湾地区诗人洛夫的《窗下》一诗，在此表示感谢。这首诗颇有印象派的味道，与洛夫的大部分诗的风格有所不同。

a）在控制面板中把窗口和按键设为"Windows XP Style"



b）在控制面板中把窗口和按键设为"Windows Classic Style"

图8-11 使用动态资源

8.5 本章小结

本章系统地讨论了WPF中的资源及其用法，资源可以嵌入到聚合块中，也可以以独立的文件的形式存在。第一部分的资源实际上在.NET 1.0中就有，但WPF增加了新的访问资源的机制——Uri。然后讨论了WPF的资源扩展，本章讨论了三种扩展：StaticResource、Static和DynamicResource。在WPF应用程序中，资源中通常含有风格（Style）和模板（Template），有关这两部分的内容将在第9章风格和第10章模板中详细讨论。

第9章 风 格

WPF中的风格实际上是集中设定元素属性的一种机制，这里的属性是指FrameworkElement和FrameworkContentElement及其派生类所支持的相关属性或附加属性。它可以部分弥补XAML语言不支持循环语句的不足，由于在风格中引入触发器，在XAML里可以方便地实现事项处理，甚至动画，而不必写任何C#或VB代码。

例如，网页或窗口中有很多按钮，而希望所有的按钮看起来都一样，这时候最好的解决方案就是使用风格。从这个角度上来说，WPF中的风格和HTML网页中的CSS（Cascading Style Sheets）相似；但由于WPF支持传递事件。相关属性和触发器（Trigger），所以WPF风格的功能更为强大。

本章详细讨论了WPF中的风格类，并用实例说明在XAML中使用风格类中属性的方法。

9.1 Style类

Style类位于System.Windows命名空间中，它定义了7个属性，这7个属性及其功能如表9-1所示。

表9-1 Style属性

属性名	功 能
Resources	读写属性
Setters	Setter和EventSetter类的集合
Triggers	TriggerBase的集合
TargetType	说明风格的施加对象
BasedOn	说明当前风格所基于的对象
IsSealed	只读属性，若为True，不能“派生”出其他风格
Dispatcher	读写属性，说明和本风格相关的DispatchObject类。

下面将对这些属性进行详细介绍。

9.2 Setters

Setters是Style类中重要的属性，它的类型是SetterBaseCollection。SetterBaseCollection是一个可放入SetterBase类型对象的包容器，所以只要是从SetterBase中派生出来的对象都可以放入到SetterBaseCollection中。实际上从SetterBase中派生出来的有两个类：一个是Setter，另一个是EventSetter。

SetterBase类中只定义了一个属性IsSealed和一个方法CheckSealed。Setter类中定义了三个属性：Property、Value、TargetName。在XAML中使用Setter的语法如下：

```
<Setter Property= "Control.FontSize", Value= "24"/>   (最常用)
Setter>                                                (有时用)
   ….
</Setter>
<Setter Property= "…", Value= "…", TargetName= "…" />
```

在C#中，就是创建Setter对象：

```
Setter mySetter = new Setter();
Setter mySetter = new Setter(Control.FontSize ,24     );
Setter mySetter = new Setter(Control.FontSize ,24 , "Button"    );
```

Setter类中的Perperty一定要是相关属性，在说明该相关属性时还要加上属性所在的类（可以是基类也可以是具体的派生类）。比如上例中我用的Control，实际上是WPF中的一个基类。

现在来看看使用Setter最简单的例子：

```
<Window
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="SimpleTextStyle" Height="300" Width="300">
    <TextBlock Height ="100">
      <TextBlock.Style>
        <Style >
          <Setter Property ="TextBlock.Background" Value ="Yellow" />
          <Setter Property ="TextBlock.FontSize" Value ="24" />
          <Setter Property ="TextBlock.FontFamily"
                Value ="Times New Roman" />
          <Setter Property ="TextBlock.FontWeight" Value ="Bold" />
        </Style>
      </TextBlock.Style>
        Test WPF  Style
    </TextBlock>
</Window>
```

在这里笔者共设置了TextBlock控件的4个属性，即Background、FontSize、Font-Family和FontWeight。查一下.NET文档，很容易看出这几个属性都是相关属性，其运行结果如图9-1所示。

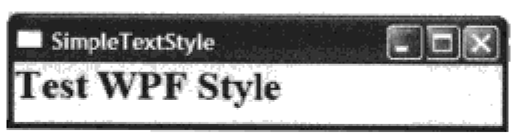


图9-1 设置TextBlock控件风格

在上面的例子中，笔者设置了TextBlock的背景色为黄色，字体的大小为24，字体为“Times New Roman”，字体可以方便地在办公软件里找。FontWeight可以设定下面的值：Black、Bold、DemiBold、ExtraBold、Heavy、Medium、SemiBold、UltraBold、ExtraLight、Light、Normal、Regular、Thin、UltraLight等。

在实际应用中，我们很少对某一个控件使用风格。使用风格的目的是：当改变某个风格时，希望所有使用该风格的控件都会改变它们的表现形式，从而不必对某控件逐一进行修改。怎样达到这一目标呢？显然，我们不能只在控件中使用风格；应该把风格作为资源（Resource）的一部分 。我们知道，在WPF中，FrameWorkElement、FrameWorkContentElement和Application 三大类里都含有Resources属性，即在这三个类及其派生类中，都可以使用资源。

一旦在资源中定义了风格，就可以在具体的控件中引用相应的风格，从而达到多个控件具有统一风格的视觉效果。通常我们把风格定义在Application的Resource中，这样整个应用程序都可以共享同

一风格。

在下面的例子中，笔者把风格定义在Window类的资源部分（FrameWorkElement是Window类继承树上的基类之一）。

```
<Window
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="StyleInResourceSection" Height="300" Width="400">
  <Window.Resources >
    <Style x:Key ="largeText">
      <Setter Property ="TextBlock.Background" Value ="Yellow" />
      <Setter Property ="TextBlock.FontSize" Value ="24" />
      <Setter Property ="TextBlock.FontFamily"
              Value ="Times New Roman" />
      <Setter Property ="TextBlock.FontWeight" Value ="Bold" />
    </Style>
  </Window.Resources>
  <StackPanel >
    <TextBlock Style ="{StaticResource largeText}">旅望因高尽</TextBlock>
    <TextBlock Style ="{StaticResource largeText}">乡心遇物悲</TextBlock>
  </StackPanel>
</Window>
```

首先，笔者在XAML中加入<Window.Resources>一节，然后在其中定义风格（Style）。在定义风格时，要使用x:Key，在引用风格时要使用“{StaticResource keyValue}”，这在为同一控件定义不同的风格时，非常方便。上面的XAML程序的运行结果如图9-2所示，可以看到两个TextBlock控件具有同一种风格。

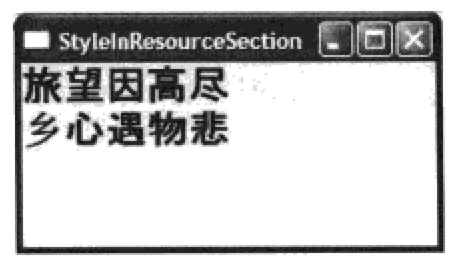


图9-2 在资源中定义风格

由于在WPF中控件类间存在继承关系，某些派生类中的属性是从基类中继承的，若设置基类的属性，那么这些属性会自动在派生类中起作用。因此可以利用这个特点来灵活地使用属性。比如，笔者把上例中的风格中Property的值改为Control：

```
<Window
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="StyleInResourceSection" Height="300" Width="400">
  <Window.Resources >
    <Style  x:Key ="largeText">
      <Setter Property ="Control.Foreground" Value ="Blue" />
      <Setter Property ="Control.FontSize" Value ="24" />
      <Setter Property ="Control.FontFamily" Value="Times New Roman"/>
      <Setter Property ="Control.FontWeight" Value ="Bold" />
```

```
    </Style>
  </Window.Resources>
  <StackPanel >
    <TextBlock  Style="{StaticResource  largeText}">旅望因高尽
    </TextBlock>
    <TextBlock  Style="{StaticResource  largeText}">乡心遇物悲
    </TextBlock>
    <Button  Style ="{StaticResource  largeText}">旅思</Button>
  </StackPanel>
</Window>
```

然后在StackPanel中加入Button元素，则largeText风格就同时对TextBlock和Button控件起作用了。原因是TextBlock和Button控件都是从Control类中派生出来的，这段XAML的运行结果如图9-3所示。

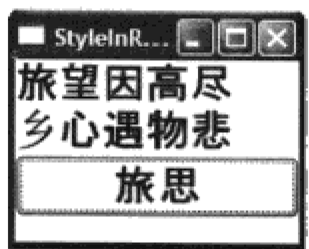


图9-3 风格用在Control的不同派生类中

再来看看如何对同一控件使用不同的风格，下面的这段程序中，笔者定义了两种风格，一个是蓝色的大字体，另一个是黑色的小字体。把蓝色的大字体用在“旅望因高尽，乡心遇物悲”上，而“故林归宿处，一叶下梧桐”两句则使用黑色的小字体。方法是在TextBlock控件中指定不同的风格，下面XAML的运行结果如图9-4所示：

```
<Window
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="StyleInResourceSection" Height="300" Width="400">
  <Window.Resources >
    <Style  x:Key ="largeText">
      <Setter Property ="Control.Foreground" Value ="Blue" />
      <Setter Property ="Control.FontSize" Value ="24" />
      <Setter Property ="Control.FontFamily" Value ="Times New Roman"/>
      <Setter Property ="Control.FontWeight" Value ="Bold" />
    </Style>
    <Style  x:Key ="smallText">
      <Setter Property ="Control.Foreground" Value ="Black" />
      <Setter Property ="Control.FontSize" Value ="12" />
      <Setter Property ="Control.FontFamily" Value ="Times New Roman"/>
      <Setter Property ="Control.FontWeight" Value ="Bold" />
    </Style>
  </Window.Resources>
  <StackPanel >
    <TextBlock  Style="{StaticResource  largeText}">旅望因高尽
                </TextBlock>
    <TextBlock  Style="{StaticResource  largeText}">乡心遇物悲
                </TextBlock>
    <TextBlock  Style="{StaticResource  smallText}">故林归宿处
```

```
            </TextBlock>
    <TextBlock  Style="{StaticResource  smallText}">一叶下梧桐
            </TextBlock>
    <Button  Style ="{StaticResource  largeText}">旅思</Button>
  </StackPanel>
</Window>
```

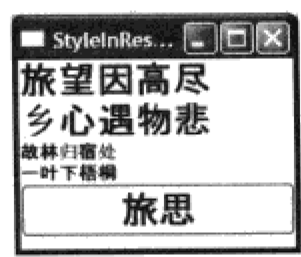


图9-4 TextBlock选择不同的风格

9.3 TargetType

和Setter一样，TargetType也是Style类中的一个属性。这个属性用来说明所定义的风格要施加的对象，使用TargetType属性和HTML或ASP网页中定义CSS非常类似。比如，笔者把上面的XAML改写为：

```
<Window
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="StyleWithTargetType" Height="300" Width="300">
  <Window.Resources >
    <Style  TargetType="{x:Type Button}">
      <Setter Property ="Control.Foreground" Value ="Blue" />
      <Setter Property ="Control.FontSize" Value ="24" />
      <Setter Property ="Control.FontFamily" Value ="Times New Roman"/>
      <Setter Property ="Control.FontWeight" Value ="Bold" />
    </Style>
    <Style  TargetType="{x:Type TextBlock}">
      <Setter Property ="Control.Foreground" Value ="Black" />
      <Setter Property ="Control.FontSize" Value ="12" />
      <Setter Property ="Control.FontFamily" Value ="Times New Roman"/>
      <Setter Property ="Control.FontWeight" Value ="Bold" />
    </Style>
  </Window.Resources>
  <StackPanel >
    <TextBlock  >旅望因高尽</TextBlock>
    <TextBlock  >乡心遇物悲</TextBlock>
    <TextBlock  >故林归宿处</TextBlock>
    <TextBlock  >一叶下梧桐</TextBlock>
    <Button>旅思</Button>
  </StackPanel>
</Window>
```

其运行的结果如图9-5所示：

图9-5 风格用于不同的目标类

由图9-5可见“旅望因高尽”等四句用的是黑色的小字体，而“旅思”按钮使用的是蓝色的大字体。在TextBlock和Button控件中笔者并没有指定风格，为什么TextBlock和Button控件使用不同的风格呢？这是因为在风格中使用了TargetType属性，其语法是：

```
TargetType= "{x:Type 控件的类}"
```

XAML中的x:Type扩展，相当于C#中的typeof。在这里应该指出，由于使用了TargetType，故可以省略Setter中Property所指定的类，例如，可以把：

```
<Setter Property ="Control.Foreground" Value ="Black" />
```

简写作：

```
<Setter Property ="Foreground" Value ="Black" />
```

其效果是一样的。

WPF还支持视觉树上风格的覆盖，其原则是控件将采用离其距离最近的风格。请看下面的例子：

```
<Window
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="StyleWithTargetType" Height="300" Width="300">
  <Window.Resources >
    <Style  TargetType="{x:Type Button}">
      <Setter Property ="Foreground" Value ="Blue" />
      <Setter Property ="Control.FontSize" Value ="24" />
      <Setter Property ="Control.FontFamily" Value ="Times New Roman"/>
      <Setter Property ="Control.FontWeight" Value ="Bold" />
    </Style>
    <Style  TargetType="{x:Type TextBlock}">
      <Setter Property ="Control.Foreground" Value ="Black" />
      <Setter Property ="Control.FontSize" Value ="12" />
      <Setter Property ="Control.FontFamily" Value ="Times New Roman"/>
      <Setter Property ="Control.FontWeight" Value ="Bold" />
    </Style>
  </Window.Resources>
  <StackPanel >
    <StackPanel.Resources>
      <Style  TargetType="{x:Type TextBlock}">
            <Setter Property ="Control.Foreground" Value ="Red" />
        </Style>
    </StackPanel.Resources>
    <TextBlock  >旅望因高尽</TextBlock>
```

```
    <TextBlock   >乡心遇物悲</TextBlock>
    <TextBlock   >故林归宿处</TextBlock>
    <TextBlock   >一叶下梧桐</TextBlock>
    <Button>旅思</Button>
  </StackPanel>
</Window>
```

笔者在Window.Resource中定义了一个风格，其目标对象为TextBlock，在StackPanel.Resources中定义了另一个风格，其目标对象还是TextBlock。这段XAML的运行结果如图9-6所示：

图9-6 风格在继承树上的覆盖

由图9-6可见“旅望因高尽”等四句现在不再是黑色的小字体，而是红色的小字体。这说明TextBlock不再用Window.Resources中的风格，而是用StackPanel.Resources中定义的红色前景色。

上述的XAML的视觉树可大概如图9-7所示，这里说的“大概”是因为真正的视觉树比图9-7要复杂得多，可以从第7章传递事件和传递命令系统开发的显示WPF视觉树程序中获得更多的视觉树的信息。对我们来说，使用这种“大概”视觉树并不影响所得出的结论。在这个视觉树中StackPanel离TextBlock比Window离TextBlock要近，所以在两种风格都适合的时候，WPF采用近水楼台先得月的原则，控件优先采用离自己近的风格，这和相关属性的应用法则是一致的。

当使用TargetType的时候，其目标对象必须是具体的控件，WPF不允许把WPF中的基类作为TargetType，即不允许有任何歧义。在上面的例子中，为了简单说明TargetType属性，笔者在Style中省去了x:Key。实际上，WPF允许同时使用TargetType和x:Key。这样，我们就可以对同一个目标对象，定义多个风格，然后再在控件中使用Key来引用不同的风格。WPF在对控件施加风格的时候，将寻找既满足TargetType、又满足x:Key属性的风格。

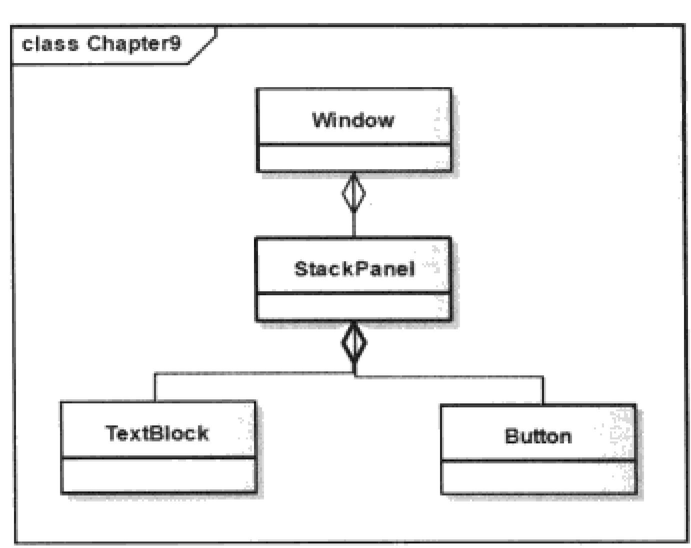


图9-7 XAML例中的大概的视觉树

9.4 BasedOn

我们知道，面向对象编程最大的优势之一就是支持继承和覆盖，在9.3节，笔者简要地说明了视觉树中风格的继承和覆盖。本节，要讨论风格本身的继承和覆盖，这种特性和C#或C++中的派生类和基类间的关系类似，就是在Style里使用BasedOn属性。

下面列出了一个使用BasedOn来扩展风格的例子：

```
<Window
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="BasedOnStyle" Height="300" Width="300">
  <Window.Resources >
    <Style  x:Key ="normal">
      <Setter Property ="Control.FontSize" Value ="24" />
      <Setter Property ="Control.FontFamily" Value ="Times New Roman"/>
      <Setter Property ="Control.FontWeight" Value ="Normal" />
    </Style>
    <Style  x:Key ="RedNormal" BasedOn ="{StaticResource  normal}">
      <Setter Property ="Control.Foreground" Value ="Red" />
     </Style>
    <Style  x:Key ="BlueSmall" BasedOn ="{StaticResource  normal}">
      <Setter Property ="Control.Foreground" Value ="Blue" />
      <Setter Property ="Control.FontSize" Value ="12" />
    </Style>
  </Window.Resources>
  <StackPanel >
    <StackPanel.Resources>
      <Style  TargetType="{x:Type Button}">
        <Setter Property ="Control.Foreground" Value ="Red" />
      </Style>
    </StackPanel.Resources>
    <TextBlock  Style ="{StaticResource normal}" >旅望因高尽</TextBlock>
    <TextBlock Style ="{StaticResource RedNormal}" >乡心遇物悲
                  </TextBlock>
    <TextBlock  Style ="{StaticResource BlueSmall}">故林归宿处
                  </TextBlock>
    <TextBlock  Style ="{StaticResource normal}">一叶下梧桐</TextBlock>
    <Button>旅思</Button>
  </StackPanel>
</Window>
```

在Window.Resources中，笔者先用x:Key定义了一个风格“normal”，其中字体的大小为24，字形为“Time New Roman”，字体的粗细为“Normal”。然后再定义一个新的风格“RedNormal”，它对风格“normal”进行了扩展。RedNormal中除了具有“normal”中所定义的字体大小、字型等风格外，还把控件的前景色设为红色。这里所使用的技术就是使用BasedOn属性，使用BasedOn的语法和引用风格的语法一样：

```
BaseOn= "{StaticResource 基类风格key }"
```

风格“BlueSmall”对风格“normal”进行了扩充和覆盖，其中前景色是normal风格中所没有的，

所以是对normal风格的扩充，而把字体的大小设为12，则是对normal风格中字体的大小24的覆盖。定义了这些风格之后，可在TextBlock中引用这些风格，这时我们需要使用key引用相应的风格，这和前面所应用的技术一致。上述XAML的运行结果如图9-8所示。

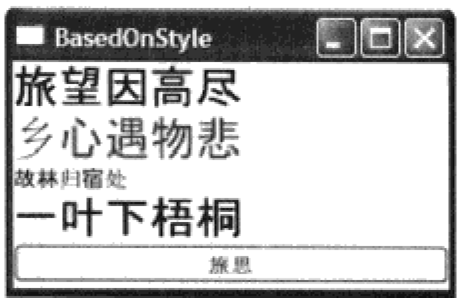


图9-8 使用BasedOn对风格进行扩展和覆盖(使用x:Key）

BasedOn属性，不仅可以利用x:Key对风格进行扩展和覆盖，而且可以对用TargetType定义的风格进行扩展和覆盖。其语法为：

```
BasedOn= "{StaticResource {x:Type 控件风格类型}}"
```

让我们来看一看这种用法：

```
<Window
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="BasedOnTargetType" Height="300" Width="300"
    >
  <Window.Resources >
    <Style TargetType="{x:Type Control}">
      <Setter Property="Control.FontSize" Value="24" />
      <Setter Property="Control.Foreground" Value="Blue" />
      <Setter Property="Control.HorizontalAlignment" Value="Center" />
    </Style>
    <Style TargetType="{x:Type Button}"   BasedOn="{StaticResource
           {x:Type Control}}">
       <Setter Property="Control.Foreground" Value="Red" />
    </Style>
    <Style TargetType="{x:Type Label}"  BasedOn="{StaticResource
           {x:Type Control}}">
       <Setter Property="Control.Foreground" Value="LightGreen" />
    </Style>
  </Window.Resources>
  <StackPanel >
    <Label >旅望因高尽</Label>
    <Label >乡心遇物悲</Label>
    <Label >故林归宿处</Label>
    <Label >一叶下梧桐</Label>
    <Button>旅思</Button>
  </StackPanel>
</Window>
```

首先，笔者定义了TargetType为Control的风格，然后在此基础上扩展了两个风格，一个TargetType是Button，另一个是Label。注意，这里把TextBlock改成了Label，否则WPF会产生“Can

only base on a Style with target type that is base type 'TextBlock'”，经过研究，笔者发现可从使用TargetType风格中扩展新的风格。WPF严格要求TargetType的控件间有严格的继承关系。在WPF中，TextBlock并不是从Control中派生出来的，而是从FrameWorkElement类中派生出来，所以若从TargetType为Control的风格中扩展TargetType为TextBlock的风格时会产生错误。而把“TextBlock”改为“Label”后，一切正常。这是因为Label和Button都是从Control中派生出来的，当然“旅望因高尽”等四句所用控件也要改为“Label”。图9-9示出了这段XAML的运行结果。

图9-9 使用BasedOn对风格进行扩展和覆盖(使用x:Type）

9.5 触发器（Triggers）

触发器（Triggers）是风格中的另一个重要的属性。这里Triggers用的是复数，意思是触发器的集合。顾名思义，触发器是在一定条件下发生的某些事件。

WPF中定义了5个触发器类：DataTrigger、MultiDataTrigger、Trigger、MultiTrigger、EventTrigger。这5个类都是从TriggerBase类中派生出来的，图9-10示出了这些触发器之间相互的关系。DataTrigger和MultiDataTrigger是一对数据触发器，两者的区别是在DataTrigger中只能说明一个条件，而MultiDataTrigger中则可以说明多个条件。Trigger和MultiTrigger也是一对触发器，和DataTrigger相似，Trigger中只能说明一个条件，而MultiTrigger里可以说明多个条件。DataTrigger和Trigger的不同在于，DataTrigger中带有Banding属性，即DataTrigger支持数据绑定。下面在第11章讨论数据绑定和第15章讨论动画时要用到这些触发器。

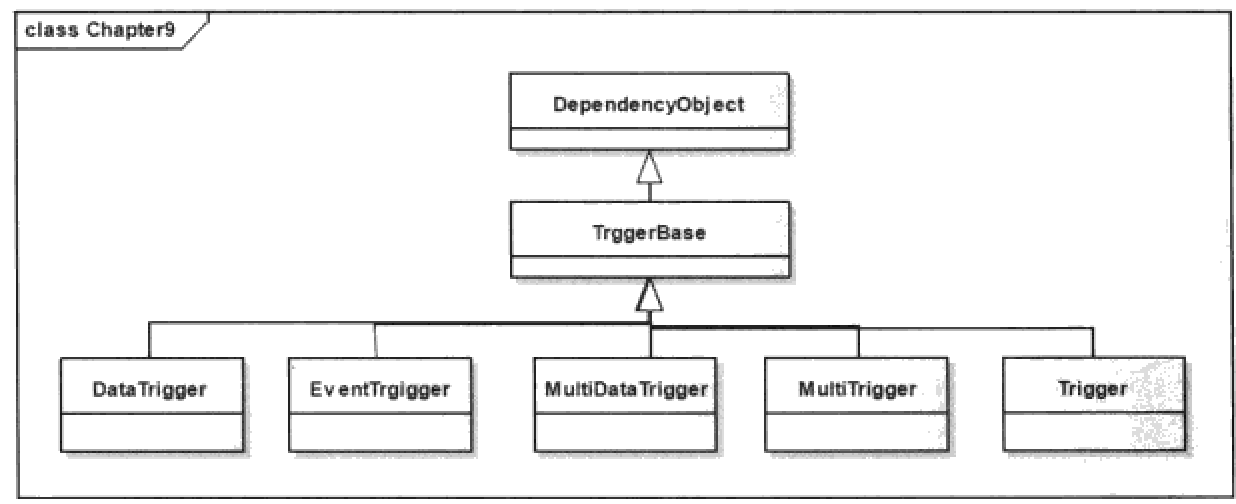


图9-10 WPF中的触发器类

9.5.1 使用单一条件的触发器

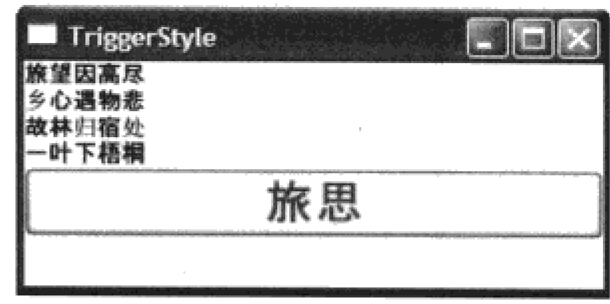
在XAML中使用触发器的语法如下：

```
<Trigger  相关属性  属性值>
      使用Setter来修改风格。
</Trigger>
```

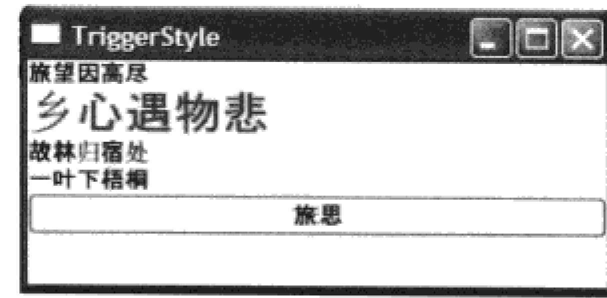
当相关属性的值发生变化时，我们希望风格应该做出反应。这种反应是通过使用Setter设定控件中其他相关属性来实现的。注意在使用Trigger时要避免死循环，换句话说，Trigger中设定的相关属性，不能作为触发器的条件，即：改变相关属性A引起相关属性B发生改变，而相关属性B改变又引发相关属性A改变的情况。

```
<Window
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="TriggerStyle" Height="300" Width="300"
    >
  <Window.Resources >
    <Style  x:Key ="smallText">
      <Setter Property ="Control.Foreground" Value ="Black" />
      <Setter Property ="Control.FontSize" Value ="12" />
      <Setter Property ="Control.FontFamily" Value ="Times New Roman"/>
      <Setter Property ="Control.FontWeight" Value ="Bold" />
      <Style.Triggers>
        <Trigger Property ="Control.IsMouseOver" Value ="true">
          <Setter Property ="Control.Foreground" Value ="Red"/>
          <Setter Property ="Control.FontSize" Value ="24" />
        </Trigger>
      </Style.Triggers>
    </Style>
  </Window.Resources>
  <StackPanel >
    <TextBlock  Style="{StaticResource  smallText}">旅望因高尽
                </TextBlock>
    <TextBlock  Style="{StaticResource  smallText}">乡心遇物悲
                </TextBlock>
    <TextBlock  Style="{StaticResource  smallText}">故林归宿处
                </TextBlock>
    <TextBlock  Style="{StaticResource  smallText}">一叶下梧桐
                </TextBlock>
    <Button  Style ="{StaticResource  smallText}">旅思</Button>
  </StackPanel>
</Window>
```

在这个例子中，笔者定义了一个可以为smallText的风格，其中字体的前景色设为黑色，字体的大小设为12等。接着笔者在风格中加入触发器，当IsMouseOver为True时，把控件的前景色改为红色，把字体的大小改为24。最后在TextBlock和Button中引用这个风格。图9-11示出了这段XAML的运行结果，从图9-11 a）和图9-11 b）可以看出，当鼠标移过字符框和按钮时，其IsMouseOver为True，从而触发了控件中的字体和颜色的改变。



a）鼠标在“乡心遇物悲”TextBlock控件上



b）鼠标在“旅思”按钮控件上

图9-11 在风格中使用Trigger

由此可以看出WPF中风格的强大功能：过去要实现这一功能只能通过事件处理程序来实现，而且要对每个控件的实例来进行处理。现在，由于WPF中定义了Trigger，所有的处理都移到了资源中，同样的处理不仅可以同一类型的控件起作用（如我们这里的4个TextBlock），而且对不同的控件也可以起作用（如这里的TextBlock和Button），因此使用WPF风格使你的代码更简练。

9.5.2 使用多个条件的触发器

上面的例子是在一个鼠标移过控件时，触发控件的风格发生改变，这时我们用单一条件触发器Trigger就可以了。当希望多个条件同时满足时，才改变控件风格的话，那么就要用MultiTrigger。使用MultiTrigger的语法如下：

```
<MultiTrigger>
      <Multitrigger.Conditions>
            <Condition Perperty= "相关属性" Value= "相关属性的值"/>
            ......
            <Condition Perperty= "相关属性" Value= "相关属性的值"/>
      </Multitrigger.Conditions>
           <Setter Property= "相关属性" Value= "相关属性的值"/>
</MultiTrigger>
```

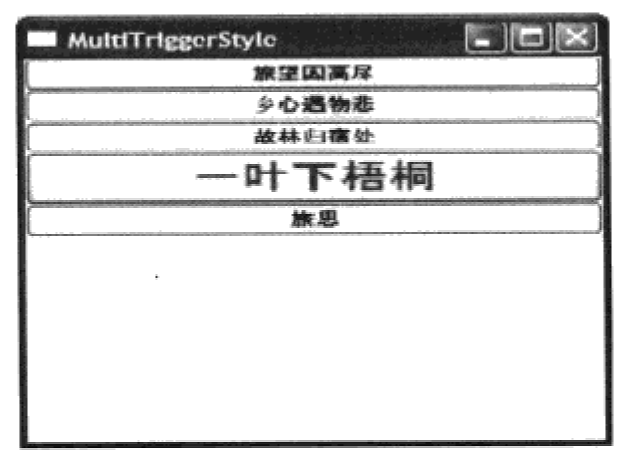
MultiTrigger和Trigger的主要区别在于MultiTrigger类中含有Conditions属性，Conditions是Condition的集合，用来定义多个触发的条件。

下面是使用MultiTrigger的例子，首先笔者把“旅望因高尽”等4句的控件改为Button；然后，在风格中加入Trigger和MultiTrigger。在MultiTrigger中，加了两个条件，其一是鼠标要在控件上，其二是按下按钮。其实在这里使用IsPressed一个条件就够了，加上两个条件纯粹是为了简单演示使用MultiTrigger。

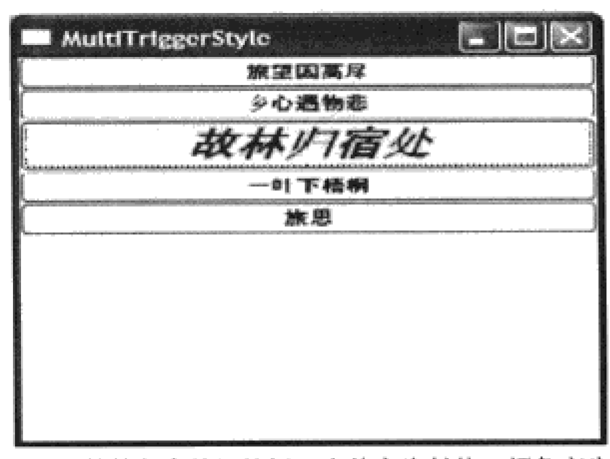
```
<Window
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="MultiTriggerStyle" Height="300" Width="300">
  <Window.Resources >
    <Style  x:Key ="smallText">
      <Setter Property ="Control.Foreground" Value ="Black" />
      <Setter Property ="Control.FontSize" Value ="12" />
      <Setter Property ="Control.FontFamily" Value ="Times New Roman"/>
      <Setter Property ="Control.FontWeight" Value ="Bold" />
      <Style.Triggers>
        <Trigger Property ="Control.IsMouseOver" Value ="true">
```

```
        <Setter Property ="Control.Foreground" Value ="Red"/>
        <Setter Property ="Control.FontSize" Value ="24" />
      </Trigger>
      <MultiTrigger >
        <MultiTrigger.Conditions >
          <Condition Property ="Control.IsMouseOver" Value ="True"/>
          <Condition Property ="Button.IsPressed" Value ="True"/>
        </MultiTrigger.Conditions>
        <Setter Property ="Control.Foreground" Value ="Blue"/>
        <Setter Property ="Control.FontStyle" Value ="Italic" />
      </MultiTrigger>
    </Style.Triggers>
  </Style>
</Window.Resources>
<StackPanel >
  <Button  Style="{StaticResource  smallText}">旅望因高尽</Button>
  <Button  Style="{StaticResource  smallText}">乡心遇物悲</Button>
  <Button  Style="{StaticResource  smallText}">故林归宿处</Button>
  <Button  Style="{StaticResource  smallText}">一叶下梧桐</Button>
  <Button  Style ="{StaticResource  smallText}">旅思</Button>
</StackPanel>
</Window>
```

这段XAML的运行结果如图9-12 a）和图9-12 b）所示。



a）鼠标经过“一叶下梧桐”按钮，字体的大小和颜色都发生了改变



b）按下“故林归宿处”按钮，字体变为斜体，颜色变为蓝色

图9-12 使用MultiTrgger

9.5.3 使用数据触发器（DataTrigger）

前面简单地提了一下DataTrigger和MultiDataTrigger这一对触发器和Trigger和MultiTrigger非常类似。但是DataTrigger多了一个Binding属性，其语法如下：

```
<DataTrigger Binding= "{Binding ElementName=控件名，Path=控件中的相关属性}"
             Value= "相关属性的值">
```

关于数据绑定将在第11章进行详细讨论。下面的XAML给出了使用DataTrigger的例子：

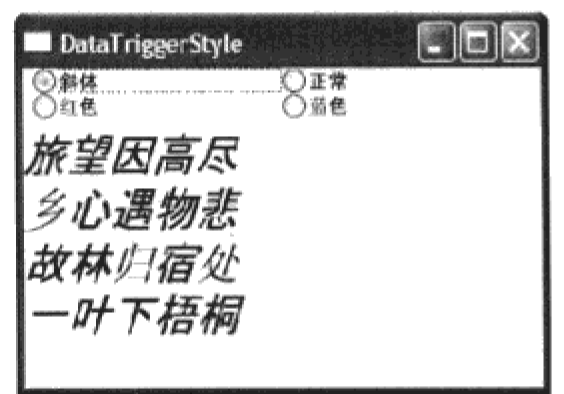
```
<Window
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
```

```
    Title="DataTriggerStyle" Height="300" Width="300" >
  <Window.Resources >
    <Style  x:Key ="smallText">
      <Setter Property ="Control.Foreground" Value ="Black" />
      <Setter Property ="Control.FontSize" Value ="24" />
      <Setter Property ="Control.FontFamily" Value="Times New Roman" />
      <Setter Property ="Control.FontWeight" Value ="Regular" />
      <Style.Triggers>
        <DataTrigger Binding ="{Binding  ElementName=italicFont,
                Path=IsChecked}" Value="True">
              <Setter Property ="Control.FontStyle" Value ="Italic"/>
        </DataTrigger>
        <DataTrigger Binding ="{Binding  ElementName=boldFont,
                Path=IsChecked}" Value="True">
            <Setter Property ="Control.FontStyle" Value ="Normal"/>
        </DataTrigger>
        <DataTrigger Binding ="{Binding  ElementName=redFont,
                Path=IsChecked}" Value="True">
          <Setter Property ="Control.Foreground" Value ="Red"/>
        </DataTrigger>
        <DataTrigger Binding ="{Binding  ElementName=blueFont,
                Path=IsChecked}" Value="True">
          <Setter Property ="Control.Foreground" Value ="Blue"/>
        </DataTrigger>
      </Style.Triggers>
    </Style>
  </Window.Resources>
  <StackPanel >
    <GroupBox >
      <Grid >
        <Grid.RowDefinitions>
          <RowDefinition />
          <RowDefinition />
        </Grid.RowDefinitions>
        <Grid.ColumnDefinitions>
          <ColumnDefinition />
          <ColumnDefinition />
        </Grid.ColumnDefinitions>
      <RadioButton Name="italicFont" Grid.Row="0" Grid.Column="0"> 斜体
                  </RadioButton>
      <RadioButton Name="regularFont" Grid.Row="0" Grid.Column="1">正常
                 </RadioButton>
      <RadioButton Name="redFont"  Grid.Row="1" Grid.Column="0">红色
                  </RadioButton>
      <RadioButton Name ="blueFont" Grid.Row ="1" Grid.Column ="1">蓝色
                  </RadioButton>
     </Grid>
    </GroupBox>
    <TextBlock  Style="{StaticResource  smallText}">旅望因高尽</TextBlock>
    <TextBlock  Style="{StaticResource  smallText}">乡心遇物悲
</TextBlock>
    <TextBlock  Style="{StaticResource  smallText}">故林归宿处</TextBlock>
```

```
    <TextBlock  Style="{StaticResource  smallText}">一叶下梧桐</TextBlock>
  </StackPanel>
</Window>
```

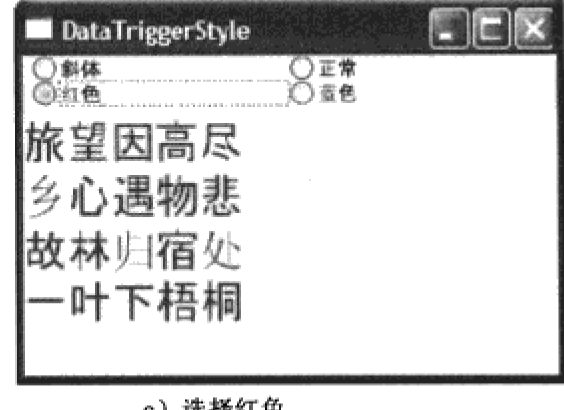
StackPanel里面加入了GroupBox控件，GroupBox含有一个2x2的Grid面板，Grid面板中有4个RadioButton控件。使用GroupBox可以自动实现在4个RadioButton中自动选择一个的功能，Grid用来控制RadioButton的排版。

接着笔者在Window.Resources中的风格部分定义了4个数据触发器，这4个触发器和4个RadioButton控件相连。当用户选择相关的RadioButton按钮时，相应的触发器开始启动，设置控件的字体、颜色等。选择不同字体属性时的结果分别如图9-13 a）、图9-13 b）、图9-13 c）和图9-13 d）所示。

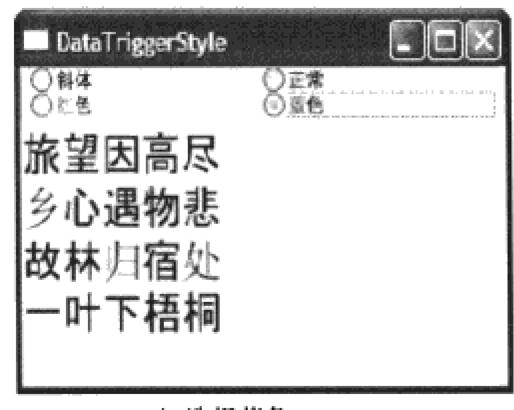


a）选择斜体

b）选择常规字体



c）选择红色



d）选择蓝色

图9-13 选择不同的字体属性

在上面的例子中，笔者讨论了使用DataTrigger的方法，和Trigger一样，数据触发器也有一个MultiDataTrigger类。MultiDataTrigger允许用户设置多个触发的条件，假如把上面例子中的4个RadioButton改为CheckBox控件，然后对这些CheckBox的值进行相关组合并对这些组合进行处理，那么就需要使用MultiDataTrigger。请读者自己完成，应该不太困难。这一节详细讨论了WPF中Trigger、MultiTrigger、DataTrigger和MultiDataTrigger的触发器，还剩下一个事件触发器EventTrigger，EventTrigger和动画密切相关，将在第15章动画中讨论EventTrigger这一特殊的触发器。

9.6 风格中的资源

第8章讨论了WPF中的各种资源，风格本身作为一种资源，可以很方便地嵌入到Application、Window、Page及各种控件中，同时风格本身也是资源的容器，Style类本身含有Resources属性，可以在其中放入资源。和WPF中其他复数属性一样，这里的资源也是复数，表示风格里可以含有多个资

源的集合。使用资源的语法很简单：

```
<Style.Resources>
       嵌入资源
</Style.Resources>
```

请看在风格中使用资源的例子：

```
<Window
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="ResourceInStyle" Height="300" Width="300">
  <Window.Resources >
    <Style x:Key ="SpecialFill">
      <Style.Resources>
        <LinearGradientBrush x:Key ="lnbrush" EndPoint="0.5,1"
                      StartPoint="0.5,0">
          <GradientStop Color="White" Offset="0"/>
          <GradientStop Color="Blue" Offset="1"/>
        </LinearGradientBrush>
      </Style.Resources>
      <Setter Property ="Rectangle.Fill" Value ="{StaticResource
                        lnbrush}"/>
    </Style>
  </Window.Resources>
  <StackPanel >
    <Rectangle Style ="{StaticResource SpecialFill}" Width="200"
               Height="100" />
    <Rectangle Style ="{StaticResource SpecialFill}" Width="200"
               Height="100" />
    <Rectangle Style ="{StaticResource SpecialFill}" Width="200"
               Height="100" />
    <Rectangle Style ="{StaticResource SpecialFill}" Width="200"
               Height="100" />
  </StackPanel>
</Window>
```

在这个例子中，笔者在风格里定义了lnbrush作为风格的资源。然后把该LinearGradientBrush用作绘制填充矩形的填充属性，从而得到特殊效果的矩形。这段XAML的运行结果如图9-14所示：

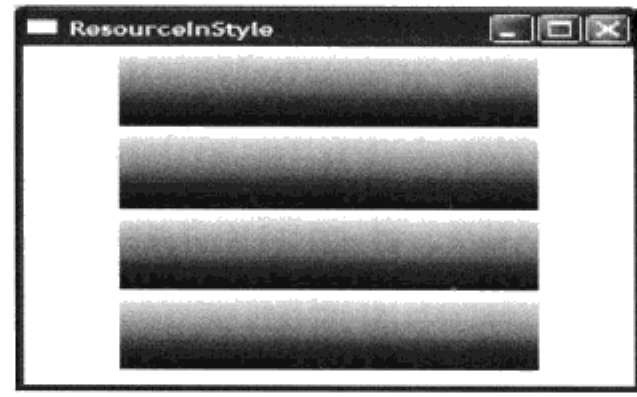


图9-14 风格中使用资源

9.7 IsSealed

在C#中，当不希望其他程序员从某个类派生出子类时，在该类的前面要加上sealed关键字。在Style中IsSealed属性表示该风格不能再改变，这是一个只读属性。WPF把风格装入内存时，还可以对风格进行修改，如加入新的Setter或修改相关属性的值等。但一旦风格施加到控件后，WPF就不再允许修改风格了，这时IsSealed属性为True。若你的程序需要改变风格，注意检查这个属性。Style类中的Seal方法，是把该属性设为True。

9.8 把风格定格定义在单独的文件中

前面举的例子，风格只施加到一两个控件上，即使不用风格，而直接在控件里设定相关属性值，也不是什么大不了的事。风格真正强大的功能是在大型软件工程中，这时你要开发多个WPF应用程序或DLL，如何让你的应用软件有一致的外观？近年骨架和皮肤（Themes/Skin）之所以流行，正是为满足这一要求。WPF并不支持Themes和Skin，但通过风格，可以很容易地实现Skin。

这时，需要把风格定义在单独的文件中。本书第18章定义了多个风格文件，如OfficeBlueSkin.xaml，在该文件中定义了各种控件的外观，使得控件和Office 2007的蓝皮肤一样：

```
<ResourceDictionary
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:RibbonLib="clr-namespace:Yingbao.Chapter18.RibbonLib"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
   ....

    <SolidColorBrush x:Key="{ComponentResourceKey RibbonLib:Skins,
PopupContainerBgBrush}" Color="White"/>

    <!--Specifies the chrome for the expanded RibbonGroup:-->
    <Style TargetType="{x:Type RibbonLib:RibbonChrome}"
x:Key="{ComponentResourceKey RibbonLib:Skins,
ExpandedRibbonGroupChromeStyle}">
        <Setter Property="InnerBorderThickness" Value="0.5"/>
        <Setter Property="NoAnimation" Value="False"/>
        <Setter Property="RenderFlat" Value="False"/>
        <Setter Property="Background" Value="{DynamicResource
{ComponentResourceKey RibbonLib:Skins, RibbonPanelBgBrush}}"/>
        <Setter Property="CornerRadius" Value="3"/>
        <Setter Property="BorderBrush" Value="#80CADDF1"/>
        <Setter Property="MouseOverBackground" Value="{DynamicResource
{ComponentResourceKey RibbonLib:Skins, RibbonPanelMouseOverBg}}"/>
    </Style>

    <!--Specifies the chrome for the collapsed RibbonGroup:-->
    <Style TargetType="{x:Type RibbonLib:RibbonChrome}"
x:Key="{ComponentResourceKey RibbonLib:Skins,
CollapsedRibbonGroupChromeStyle}">
        <Setter Property="RenderFlat " Value="False"/>
        <Setter Property="NoAnimation " Value="True"/>
        <Setter Property="BorderBrush" Value="{x:Null}"/>
        <Setter Property="CornerRadius" Value="4"/>
```

```
        <Setter Property="BorderBrush" Value="#80CADDF1"/>
        <Setter Property="Background" Value="{DynamicResource
{ComponentResourceKey RibbonLib:Skins, MinimizedRibbonPanelBgBrush}}"/>
        <Setter Property="MouseOverBackground" Value="{DynamicResource
{ComponentResourceKey RibbonLib:Skins, MinimizedMouseOverBrush}}"/>
        <Setter Property="MousePressedBackground"
Value="{DynamicResource     {ComponentResourceKey RibbonLib:Skins,
DefaultMousePressedBtnBrush}}"/>
    </Style>

......

</ResourceDictionary>
```

当需要使用Office 2007蓝皮肤的时候，只要在ResourceDictionary里加入该皮肤文件即可：

```
<ResourceDictionary
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
   xmlns:RibbonLib="clr-namespace:Yingbao.Chapter18.RibbonLib"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
    <ResourceDictionary.MergedDictionaries>
        <ResourceDictionary
Source="pack://application:,,,/YBRibbonLib;Component/Themes/OfficeBlueS
kin.xaml"/>
    </ResourceDictionary.MergedDictionaries>
...
</ResourceDictionary>
```

这是组织大型软件Style的基本方法。

9.9 在FrameworkContentElement中使用风格

风格不仅可以用在UI元素上，而且可以用在文档上。FlowDocument、FixedDocument、TextElement等都是从FrameworkContentElement中派生出来的，FrameworkContentElement中有一个Style属性，可以在这个属性中引入风格。下面的XAML中在Paragraph中使用了风格：

```
<Window
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="StyleWithTargetType" Height="300" Width="300">
    <Window.Resources>
        <Style TargetType="{x:Type Paragraph}" x:Key="BluePara">
            <Setter Property ="Control.Foreground" Value ="Blue" />
            <Setter Property ="Control.FontSize" Value ="24" />
            <Setter Property ="Control.FontFamily" Value ="Times New
                 Roman" />
            <Setter Property ="Control.FontWeight" Value ="Regular" />
        </Style>
        <Style TargetType="{x:Type Paragraph}" x:Key="RedPara">
            <Setter Property ="Control.Foreground" Value ="Red" />
            <Setter Property ="Control.FontSize" Value ="24" />
            <Setter Property ="Control.FontFamily" Value ="Times New
              Roman" />
```

```
            <Setter Property ="Control.FontWeight" Value ="Regular" />
        </Style>
    </Window.Resources>
    <StackPanel>
        <FlowDocumentPageViewer>
            <FlowDocument>
              <Paragraph Style="{StaticResource BluePara}">旅望因高尽
                 </Paragraph>
              <Paragraph Style="{StaticResource RedPara}">乡心遇物悲
                  </Paragraph>
            </FlowDocument>
        </FlowDocumentPageViewer>
      </StackPanel>
</Window>
```

这里“旅望因高尽”、“乡心遇物悲”两句分别使用两个不同的风格。风格的实际效果如图9-15所示。

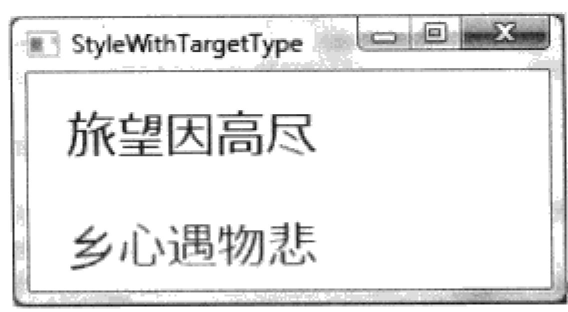


图9-15 在Document中使用风格

9.10 再谈Setter属性

在创建风格的例子中，我们一直都在使用Setter来设定相关属性的值。Setter是一个类，它以SetterBase为基类。从前面的例子你也许已经有些疑惑，比如9.9节的例子：

```
<Style TargetType="{x:Type Paragraph}" x:Key="BluePara">
     <Setter Property ="Control.Foreground" Value ="Blue" />
     <Setter Property ="Control.FontSize" Value ="24" />
     <Setter Property ="Control.FontFamily" Value ="Times New Roman" />
     <Setter Property ="Control.FontWeight" Value ="Regular" />
</Style>
```

这里风格的TargetType设为“Paragraph”，但其中的Setter设置的相关属性为Control.Forground等。我们知道Paragraph的类继承树上并没有Control类，为什么用Setter所设定的风格仍然可以正确地起作用呢？更进一步，若省略设置Style里TargetType属性，结果仍然是一样的，即：

```
<Style  x:Key="BluePara">
     <Setter Property ="Control.Foreground" Value ="Blue" />
     <Setter Property ="Control.FontSize" Value ="24" />
     <Setter Property ="Control.FontFamily" Value ="Times New Roman" />
     <Setter Property ="Control.FontWeight" Value ="Regular" />
</Style>
```

这里的奥秘在于WPF中的相关属性系统，在第4章WPF中的属性系统讨论相关属性和附加属性

时，我们曾经提过在WPF相关属性系统中注册相关属性和附加属性的方法。WPF中有两类相关属性，一类是某个类中独有的相关属性，另一类是共享相关属性。当相关属性是共享相关属性时，相关属性和其所在的类没有什么关系，只是在该相关属性上增加了一个Owner，如第4章的CustomeTextBlock类中，我们对TimerWindow中的定义的相关属性TimeProperty增加了一个Owner：

```
TimeProperty =TimerWindow.TimeProperty.
              AddOwner(typeof(CustomeTextBlock));
```

当TimeProperty在TimerWindow中改变时，该值在CustomeTextBlock中同时改变。使用Setter设定相关属性时，其作用的机制是一样的。然而，若某个相关属性是某个类中独有的，那么在风格中使用另一类型设置相关属性就不起作用。很多程序员在遇到这种情况时，会迷惑不解。为了避免这种混淆，最好更清晰地设定相关属性，例如，把上面的风格改为：

```
<Style  x:Key="BluePara">
     <Setter Property ="Paragraph.Foreground" Value ="Blue" />
     <Setter Property ="Paragraph.FontSize" Value ="24" />
     <Setter Property ="Paragraph.FontFamily" Value ="Times New
                       Roman" />
     <Setter Property ="Paragraph.FontWeight" Value ="Regular" />
</Style>
```

这时就不会有什么歧义了。

我们不仅可以在风格中使用Setter设置相关属性的值，也可以用Setter设定控件模板的中相关属性的值，第10章讨论控件模板时，还将遇到这个问题。

9.11 本章小结

本章讨论了WPF中的风格，设定风格就是使用Setter类来设定相关属性的值。虽然Setter可以设定任何相关属性，但一般来说，在风格里设定的是与界面有关的相关属性。在概念上WPF的风格和HTML中的CSS类似，但由于WPF中相关属性的强大功能，使得WPF中的风格可以完成CSS所不具备的功能。

---

*附注：我这里用的4句诗："旅望因高尽，乡心遇物悲"是唐代诗人崔曙《途中晓发》中的名句，"故林归宿处，一叶下梧桐"则出自崔曙《山下晚晴》一诗中。

第10章 模 板

模板是WPF控件中独特的技术，在前面的章节中，曾提到WPF中控件的显示和其内部逻辑是分开的。过去，在Windows操作系统中，控件的外观是固定的，程序员能改变控件在界面上的一些有限参数，如位置（Location）、尺寸（Size）等。常常听到业内人士对Windows中这种呆板的控件外观所作的批评，在苹果的Mac 操作系统中，其控件的外观比Windows 中的控件更吸引人。为此，WPF发明了另外一套解决方案：首先把控件的外观和控件的逻辑分开，控件的外观由控件模板提供；其次，WPF让程序员随时使用自己的控件模板。WPF在设计时，考虑到两类不同的人的需求，一类是程序员，他们的特长是开发控制程序的逻辑；另一类是专门进行界面设计的设计人员。界面设计人员专注于设计控件的外观，程序员则专注于控件的逻辑。

这种解决方案非常灵活，在实际应用中也是非常有用的。例如：在电力系统中断路器和隔离开关都是状态量，它一般有两种状态：合上或断开。若用ToggleButton或CheckBox来显示，该是多么的呆板！在电力系统的主接线图中，断路器和隔离开关是不一样的。过去我们要自己开发这些符号，需要做大量的工作，而在WPF中，只要开发不同的模板，插入选择按钮控件即可！

可以说，WPF控件的外观只有你想不到的，没有你做不到的。这实际上是WPF设计时提出的基本思想——一切交给应用程序开发者。界面设计人员更像一个艺术家，WPF给你提供了画板、画布、颜料、画笔和画刷，最后你创造出什么样的作品，全凭你的功力！准备创造你独特的，哪怕是奇怪的控件呢！如果你觉得没有艺术天赋，不要着急，WPF为所有的控件提供了基本的模板，这些模板就是在前面章节中所显示的控件的默认外观。这有点像练习书法或绘画，你可以从临摹开始。

10.1 模板概述

所有的模板类都是从FrameworkTemplate类中派生出来的，FrameworkTemplate是一个抽象类，它管理模板的一些基本属性，如视觉树、资源等。FrameworkTemplate 是相关对象（DependencyObject），所以它具有相关对象的所有功能。

从FrameworkTemplate中派生出三个类：ControlTemplate、ItemsPanelTemplate和DataTemplate，从DataTemplate类中又派生出了ContentPresenter和HierachicalDataTemplate两个类。图10-1示出了WPF中管理模板的类结构。

我们知道，所有的WPF控件都是从 Control类中派生出来的，Control类中有一个属性Template，其类型是ControlTemplate。设置Control类中的Template属性就可以改变控件在界面上的外观。对于条目控件（ItemsControl），通常并不需要替换整个控件的模板，而只需要改变显示条目的方式，ItemsPanelTemplate类就是为这类控件设计的，当然也可以设定条目控件的模板，从而彻底改变条目控件的外观。ItemsControl除了继承了Control类中的Template属性之外，增加了三个与模板相关的属性：ItemsPanel、ItemTemplate和ItemTemplateSelector。ItemsPanel的类型为ItemsPanelTemplate，用来设定条目面板的视觉树。ItemTemplate的类型为DataTemplate，通常条目控件中每个具体的条目是一个.Net 数据对象。设定DataTemplate可以把特定的数据对象以特有的方式展现出来，

ItemTemplateSelector属性用来灵活地用不同模板来显示不同条目的目的。

图10-1 WPF中管理模板的类结构

对于内容控件，我们通常要在替换控件外观的同时显示控件的内容。比如说按钮控件，既要能改变按钮的外观，又能让控件灵活改变其中的内容。ContentPresenter类就是为此而设计的，它可以灵活地把控件中的内容，粘贴到模板类中。

条目控件中派生出来的另外三个类Menu、ContextMenu和TreeView通常具有层次结构，HierarchicalDataTemplate就是专门为这种结构设计的，HierarchicalDataTemplate中含有ItemSource、ItemTemplate和ItemTemplateSelector属性，这些属性用来设置条目的相应模板。

和控件的风格一样，模板也可以直接嵌入到控件中，或控件的资源中，或者应用程序的视觉树上任何一个节点的资源中。与位于资源中的风格一样，模板也是沿着视觉树从树根向树枝传递的。

在模板中构建视觉树时，常常需要用到控件中的相关属性。为了在模板中方便地访问控件的相关属性，需要用到数据绑定。TemplateBindingExtension类就是为此设计的，这个类比BindingExtension所用的计算机的资源要少。

风格用来改变某个或某类控件的属性，模板比风格有更大的自由度，有时候需要结合使用这两种技术。比如在风格里根据实际情形设置控件的模板，这样可以使得控件更加多样化。总而言之，在WPF里，只有你想不到的、没有你做不到的。

10.2 控件模板

10.2.1 在控件中使用模板

在Control类中，有一个Template属性，其类型为ControlTemplate。可以通过直接设置ControlTemplate，从而改变控件的视觉树。由于WPF控件类都是从Control类中派生出来的，因此，可以用这个技术设置所有控件的外观。

在XAML中，设置Template的语法如下：

```
<Control.Template>
      <ControlTemplat>
      ....
      ...
      </ControlTemplate>
</Control.Template>
```

让我们来构建按钮的模板，笔者用一个填充矩形来构建按钮的外观，在矩形内，用一个TextBlock来显示按钮上的字符串：

```
<Page
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
      xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
  <StackPanel>
    <Button HorizontalAlignment="Center"
       VerticalAlignment="Center"
       FontSize="48" Padding="20" Height ="100" Width ="200">
      <Button.Template>
        <ControlTemplate >
        <Rectangle RadiusX="10" RadiusY="10" Stroke="BlueViolet"
           StrokeThickness="3" Name="myRectangle">
          <Rectangle.Fill>
            <VisualBrush Opacity="0.7">
              <VisualBrush.Visual>
                <TextBlock Name="myTextBlock"
                   Foreground="LightYellow" Background="DarkBlue"
                   Padding="10">模板按钮</TextBlock>
              </VisualBrush.Visual>
            </VisualBrush>
          </Rectangle.Fill>
        </Rectangle>
        </ControlTemplate>
      </Button.Template >
    </Button>
  </StackPanel>
</Page>
```

由于上面的XAML的根元素为Page，可以直接在浏览器中显示该XAML。上述XAML的显示结果如图10-2所示：

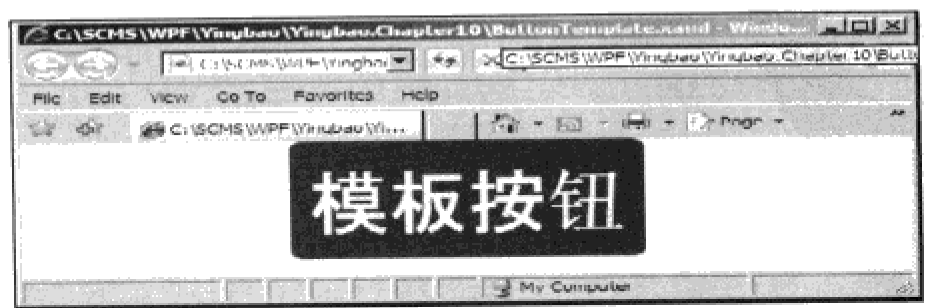


图10-2 模板按钮

对于按钮控件，还需要给模板加入一些功能，使得我们的模板和通常按钮表现一样：当鼠标在按钮控件上拖过时，需要改变按钮的显示；当用户在按钮上按下鼠标时，按钮需要作出相应的反应，使

用户知道按钮已被按下。为此，需要在模板中加入触发器。

```
<ControlTemplate.Triggers>
   <Trigger  Property ="UIElement.IsMouseOver" Value ="True">
     <Setter TargetName="myRectangle" Property="Stroke"
           Value="Green"/>
     <Setter TargetName="myTextBlock" Property="Foreground"
            Value="Red"/>
  </Trigger>
  <Trigger Property="Button.IsPressed" Value="True">
     <Setter TargetName="myRectangle" Property="Stroke"
            Value="LightGreen"/>
     <Setter TargetName="myTextBlock" Property="Background">
       <Setter.Value>
         <LinearGradientBrush>
             <GradientStop Offset="1" Color="DarkBlue"/>
             <GradientStop Offset="0.5" Color="Blue"/>
             <GradientStop Offset="0" Color="DarkBlue"/>
         </LinearGradientBrush>
       </Setter.Value>
     </Setter>
  </Trigger>
</ControlTemplate.Triggers>
```

在控件模板中使用触发器和在风格中使用触发器的原理是一样的，这里我加入了两个触发器。一个是当IsMouseOver属性为真时，把按钮外面的矩形框的颜色改为绿色，把按钮的字体颜色改为红色。另一个是当IsPressed属性为真时，把按钮外面的矩形框的颜色改为浅绿色，把字符框的背景画刷改为线性梯度画刷。

10.2.2 在资源中使用模板

和风格一样，一般很少在控件中直接定义控件的模板，通常控件的模板要放在某个视窗（Window）、网页（Page）或应用程序（Application）的资源部分。这样同一类别的控件可以共享控件模板。

要在控件中引用特定模板，先要给控件模板定义一个关键字Key：

```
<ControlTemplate x:Key ="myButtonTemplate" >
```

用静态资源绑定在控件中引用该模板：

```
<Button  Template ="{StaticResource myButtonTemplate}"/>
```

下面是完整的XAML：

```
<Page
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
      xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
  <Page.Resources>
    <ControlTemplate x:Key ="myButtonTemplate"   >
      <Rectangle RadiusX="10" RadiusY="10" Stroke="BlueViolet"
         StrokeThickness="3" Name="myRectangle">
        <Rectangle.Fill>
```

```
        <VisualBrush Opacity="0.7">
          <VisualBrush.Visual>
            <TextBlock Name="myTextBlock"
               Foreground="LightYellow" Background="DarkBlue"
               Padding="10">模板按钮</TextBlock>
          </VisualBrush.Visual>
        </VisualBrush>
      </Rectangle.Fill>
    </Rectangle>
    <ControlTemplate.Triggers>
      <Trigger  Property ="UIElement.IsMouseOver" Value ="True">
        <Setter TargetName="myRectangle" Property="Stroke"
                Value="Green"/>
        <Setter TargetName="myTextBlock" Property="Foreground"
                Value="Red"/>
      </Trigger>
      <Trigger Property="Button.IsPressed" Value="True">
        <Setter TargetName="myRectangle" Property="Stroke"
                Value="LightGreen"/>
        <Setter TargetName="myTextBlock" Property="Background">
          <Setter.Value>
            <LinearGradientBrush>
              <GradientStop Offset="1" Color="DarkBlue"/>
              <GradientStop Offset="0.5" Color="Blue"/>
              <GradientStop Offset="0" Color="DarkBlue"/>
            </LinearGradientBrush>
          </Setter.Value>
        </Setter>
      </Trigger>
    </ControlTemplate.Triggers>
  </ControlTemplate>
</Page.Resources>
<StackPanel>
  <Button HorizontalAlignment="Center"
          VerticalAlignment="Center" FontSize="48" Padding="20"
          Height ="100" Width ="200"
          Template ="{StaticResource myButtonTemplate}"/>
  <Button HorizontalAlignment="Center"
          VerticalAlignment="Center"
          FontSize="48" Padding="20" Height ="100" Width ="200"
          Template ="{StaticResource myButtonTemplate}"/>
</StackPanel>
</Page>
```

10.2.3 在控件模板中使用TargetType

在控件模板中使用控件的相关属性，需要使用控件类名+属性名的方式，如上面的XAML中的：

```
<Trigger  Property ="UIElement.IsMouseOver" Value ="True">
```

这里我们用UIElement.IsMouseOver来引用，若在控件模板中加上TargetType，可以直接使用属性，而不必加上控件类名字。

```
<ControlTemplate x:Key ="myButtonTemplate"  TargetType ="{x:Type
Button}" >
```

这时，可以把UIElement.IsMouseOver改为：

```
<Trigger  Property ="IsMouseOver" Value ="True">
```

和风格不同，在控件模板中使用TargetType，并不能让你省略设置x:Key属性，在控件中引用模板的唯一方式是使用x:Key。

在没有设置TargetType的控件模板中，可以在任何控件中应用该模板，如可以用下面的语句设置TextBox的模板：

```
<TextBox HorizontalAlignment="Center" VerticalAlignment="Center"
            FontSize="48" Padding="20" Height ="100" Width ="200"
            Template ="{StaticResource
            myButtonTemplate}"/>
```

虽然TextBox和Button两个控件的属性，以及它们运行时的特性是不同的，但WPF会自动挑出可以用到TextBox上的模板属性和运行时的特性，同时自动忽略那些不合适的属性或运行时的特性。

一旦设置了模板中的TargetType属性，引用该模板的控件必须和模板的类型一致。

10.2.4 在模板中显示控件的内容

在上例的按钮模板中，当在多个按钮中引用该模板时，按钮中显示的都是“模板按钮”字符串，这显然不能满足实际应用的要求。我们知道按钮是一种内容控件，其中可以嵌入任何对象。

要在控件模板中显示控件的内容需要用到数据绑定，第11章将详细讨论各种数据绑定，好在模板的数据绑定是一种最简单的情形。在XAML的控件模板中实现数据绑定，需要用到TemplateBindingExtension的XAML扩展类。

模板绑定的数据源总是目标元素，“path”总是目标元素的相关属性。在TemplateBinding中，我们使用属性而不是路径“path”。例如，要在按钮模板中显示控件的内容，可以用下面的XAML：

```
<TextBlock Name="myTextBlock"  Text=  "{TemplateBinding
          Property=Button.Content}" / >
```

由于TemplateBinding的构造函数接受相关属性，上面的XAML可以简写为：

```
<TextBlock Name="myTextBlock" Text=  "{TemplateBinding
          Button.Content}" / >
```

若在ControlTemplate中设置了TargetType属性，上面的TemplateBinding更可简化为：

```
<TextBlock Name="myTextBlock" Text="{TemplateBinding Content}" />
```

最后，可把按钮修改如下：

```
<StackPanel>
    <Button HorizontalAlignment="Center"
           VerticalAlignment="Center"
            FontSize="48" Padding="20" Height ="100" Width ="200"
             Template ="{StaticResource
```

```
            myButtonTemplate}" Content ="按钮一"/>
    <Button HorizontalAlignment="Center"
            VerticalAlignment="Center"
            FontSize="48" Padding="20" Height ="100" Width ="200"
            Template ="{StaticResource myButtonTemplate}"
            Content ="按钮二"/>
  </StackPanel>
```

在IE中打开该XAML，结果如图10-3所示：

图10-3 利用TemplateBinding显示按钮中的Content属性

10.2.5 在模板中使用ContentPresenter

在上面的例子中，我们利用TextBlock和TemplateBinding在模板中显示控件的内容，使用TextBlock只能显示字符串。一般情况下，需要显示内容控件中的任何内容。为此，WPF专门设计了一个类ContentPresenter。

使用ContentPresenter的语法如下：

```
<ContentPresenter Margin ="10"  Content ="{TemplateBinding
Content}"/>
```

由于ContentPresenter类的唯一作用就是把模板中的内容绑定到目标控件的内容上，所以，可以用更简洁的形式：

```
<ContentPresenter Margin ="10" />
```

为了测试ContentPresenter可以绑定任何内容，笔者把第二个Button的内容设为：

```
<Button HorizontalAlignment="Center" VerticalAlignment="Center"
        FontSize="24" Padding="5" Height ="60" Width ="100"
        Template ="{StaticResource  myButtonTemplate}" >
  <StackPanel>
      <TextBlock FontSize ="16" Foreground ="Black">按钮和圆
           </TextBlock>
      <Ellipse Name ="btnEllipse"  Fill ="Cyan" Stroke ="Blue"
         Width="100" Height ="50"/>
      </StackPanel>
   </Button >
```

这个按钮在第6章WPF控件中用过，笔者把page改成Window，下面是完整的程序：

```
<Window
 xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="TemplateContentPresenter" Height="300" Width="300">
  <Window.Resources>
    <ControlTemplate x:Key ="myButtonTemplate"
         TargetType ="{x:Type Button}" >
      <Rectangle RadiusX="10" RadiusY="10" Stroke="BlueViolet"
         StrokeThickness="3" Name="myRectangle">
        <Rectangle.Fill>
          <VisualBrush Opacity="0.7">
            <VisualBrush.Visual>
              <ContentPresenter Margin ="10" Content
                 ="{TemplateBinding Content}"/>
            </VisualBrush.Visual>
          </VisualBrush>
        </Rectangle.Fill>
      </Rectangle>
      <ControlTemplate.Triggers>
        <Trigger  Property ="UIElement.IsMouseOver" Value ="True">
          <Setter TargetName="myRectangle" Property="Stroke"
                Value="Green"/>
        </Trigger>
        <Trigger Property="Button.IsPressed" Value="True">
          <Setter TargetName="myRectangle" Property="Stroke"
                Value="LightGreen"/>
          <Setter TargetName="myRectangle" Property="Fill">
            <Setter.Value>
              <LinearGradientBrush >
                <GradientStop Offset="1" Color="DarkBlue"/>
                <GradientStop Offset="0.5" Color="Blue"/>
                <GradientStop Offset="0" Color="DarkBlue"/>
              </LinearGradientBrush>
            </Setter.Value>
          </Setter>
        </Trigger>
      </ControlTemplate.Triggers>
    </ControlTemplate>
  </Window.Resources>
  <StackPanel>
    <Button HorizontalAlignment="Center"
       VerticalAlignment="Center" FontSize="24" Padding="5"
       Height ="60" Width ="100"
       Template ="{StaticResource myButtonTemplate}"
       Content ="按钮一"/>
    <Button HorizontalAlignment="Center"
       VerticalAlignment="Center" FontSize="24" Padding="5"
       Height ="60" Width ="100"
       Template ="{StaticResource myButtonTemplate}" >
      <StackPanel>
        <TextBlock FontSize ="16" Foreground ="Black">按钮和圆
             </TextBlock>
        <Ellipse Name ="btnEllipse"  Fill ="Cyan" Stroke ="Blue"
```

```
            Width="100" Height ="50"/>
      </StackPanel>
    </Button >
  </StackPanel>
</Window>
```

图10-4示出了这段程序的运行结果。由图10-4可见，在ControlTemplate中使用ContentPresenter后，可以显示按钮中的任何内容（如第二个按钮中构建的整个视觉树）。

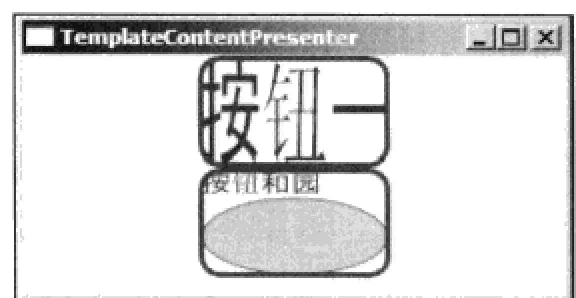


图10-4 在ControlTemplate中使用ContentPresenter

10.2.6 模板中元素名Name属性

我们知道当给控件设定Name属性时，WPF会创建该控件类型的变量，如：

```
<TextBox Name="myText1"> 测试</ TextBox>     (XAML)
```

相当于：

```
TextBox myText1= new TextBox();                    (C#)
myText1.Text="测试";
```

但在控件模板中，当同样给控件设定Name属性时，WPF不会创建给控件类型的变量，如上例中的：

<Rectangle Name="myRectangle" >

但不能在C#中使用 myRectangle 来设置矩形的其他属性，WPF之所以不为控件模板中的元素创建同名域，原因在于：控件模板可以用到多个控件上，这时名字并不能唯一地标识相应控件的实例。模板中元素的名字只在控件模板内有效。

10.2.7 在模板中绑定控件的其他属性

由于模板通常要被多个控件使用，通常不希望在模板中固定死某些属性的值，而希望这些属性可以根据控件的情况进行调整。例如在上面的例子中，笔者把Margin设为10，其实若把它绑定到控件的Padding属性，则该Margin属性对每个控件可以取不同的值。

因此，可以将下面的语句：

```
<ContentPresenter Margin ="10"/>
```

改为：

```
<ContentPresenter   Margin ="{TemplateBinding Padding}"/>
```

那么，当设定按钮的Padding值时：

```
<Button  Padding="20"  ...../>
```

ContentPresenter的Margin的值就是20，而：

```
<Button  Padding="5"  ...../>
```

ContentPresenter的Margin的值就是5。可以用这种方式在控件模板中使用控件的其他相关属性。

10.2.8 使用模板显示电力系统的断路器和刀闸开关

本章一开始，曾经提到在电力系统主接线图上，需要显示不同的符号。如断路器，用方块表示。断路器有两种状态：闭合和断开。当断路器处在闭合状态时，方块是实心的；当断路器处在断开状态时，方块是空心的。另外一种符号是隔离开关，虽然隔离开关和断路器一样有闭合和断开两种状态，但其符号是不同的。

在下面的XAML中，笔者写了两个控件模板，两个模板的TargetType都是CheckBox。

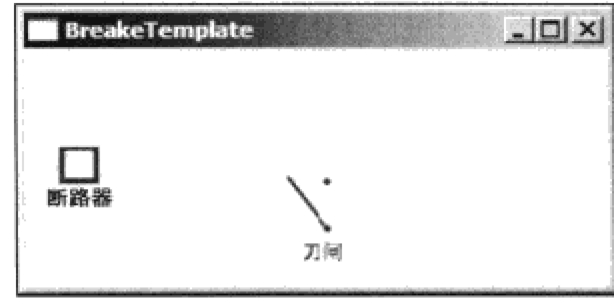
```
<Window
x:Class="Yingbao.Chapter11.BreakeTemplate.BreakTemplateWindow"
 xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="BreakeTemplate" Height="300" Width="300">
  <Window.Resources>
    <ControlTemplate x:Key="break"
            TargetType ="{x:Type CheckBox}">
      <ControlTemplate.Resources>
        <SolidColorBrush x:Key="RedBrush" Color ="Red"/>
      </ControlTemplate.Resources>
      <StackPanel>
      <Rectangle Name ="breakRectangle"  Stroke ="Red"
        StrokeThickness="3" Width ="20" Height ="20">
        <Rectangle.Fill>
                 <SolidColorBrush Color="White"/>
        </Rectangle.Fill>
      </Rectangle>
        <ContentPresenter />
      </StackPanel>
        <ControlTemplate.Triggers>
          <Trigger Property="IsChecked" Value="True">
            <Setter TargetName="breakRectangle" Property="Fill"
                    Value="{StaticResource RedBrush}" />
             </Trigger>
        </ControlTemplate.Triggers>
    </ControlTemplate>
    <ControlTemplate x:Key="switch"
            TargetType="{x:Type CheckBox}">
      <Grid>
        <Grid.RowDefinitions>
          <RowDefinition Height="Auto" />
          <RowDefinition Height="Auto" />
        </Grid.RowDefinitions>
```

```
        <Border Width="96" Height="48" Grid.Row="0"
                BorderBrush="Black" BorderThickness="1">
          <Canvas Background="LightGray">
            <Ellipse  Stroke ="Green" StrokeThickness ="2"
                      Canvas.Left ="48" Canvas.Top ="16"
                      Width="4" Height ="4"/>
            <Ellipse  Stroke ="Green" StrokeThickness ="2"
                      Canvas.Left ="48" Canvas.Top ="40"
                      Width="4" Height ="4"/>
            <Line Name="lineOff"   StrokeThickness="2"
                  Stroke="Red" X1="48" Y1="40" X2="30" Y2="16"
                  StrokeStartLineCap="Round"
                  StrokeEndLineCap="Round"  />
            <Line Name="lineOn"   StrokeThickness="2" Stroke="Red"
                  X1="48" Y1="40" X2="48" Y2="16"
                  StrokeStartLineCap="Round"
                  StrokeEndLineCap="Round"  Visibility="Hidden"/>
          </Canvas>
        </Border>
        <ContentPresenter Grid.Row="1"
                 Content="{TemplateBinding Content}"
                 HorizontalAlignment="Center" />
      </Grid>
      <ControlTemplate.Triggers>
        <Trigger Property="IsChecked" Value="True">
          <Setter TargetName="lineOff" Property="Visibility"
                  Value="Hidden" />
          <Setter TargetName="lineOn" Property="Visibility"
                  Value="Visible" />
        </Trigger>
      </ControlTemplate.Triggers>
    </ControlTemplate>
  </Window.Resources>
  <Canvas>
    <CheckBox Template="{StaticResource break}"  Content="断路器"
       Canvas.Top="50" Canvas.Left="10"
       HorizontalAlignment="Center" VerticalAlignment="Center" />
    <CheckBox Template="{StaticResource switch}"
              Content="刀闸" Canvas.Top="50" Canvas.Left="100"
              HorizontalAlignment="Center"
              VerticalAlignment="Center" />
  </Canvas>
</Window>
```

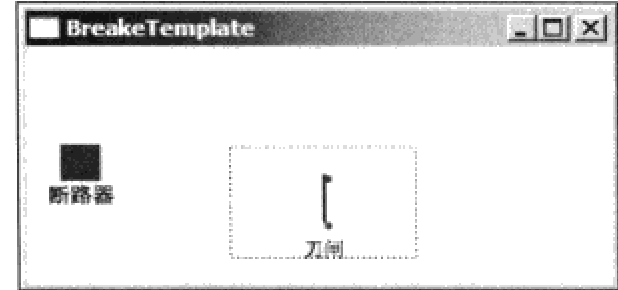
图10-5 a）和图10-5 b）示出了这段XAML的运行结果，当用鼠标单击断路器时，随着CheckBox状态的改变，断路器和开关的符号相应地发生变化。在上面的例子中，笔者把红色的画刷RedBrush作为资源放在ControlTemplate的资源部分：

```
<ControlTemplate.Resources>
     <SolidColorBrush x:Key="RedBrush" Color ="Red"/>
</ControlTemplate.Resources>
```

上面的例子中，笔者利用控件模板彻底改变了CheckBox的外观，读者可以仔细体会WPF把控件外观和控件逻辑分开的优势。



a）处在断开状态的断路器和隔离刀闸



b）处在闭合状态的断路器和隔离刀闸

图10-5 使用控件模板产生的断路器和隔离开关

10.2.9 在风格中使用模板

把风格和模板合起来使用，可以取得独特的效果，这时我们在风格里设置控件的模板，其语法如下：

```
<Style>
  ….
  <Setter Property="Template">
    <Setter.Value>
      <ControlTemplate >
       ….
       </ControlTemplate>
    </Setter.Value>
    </Setter>
</Style>
```

即在风格中设置控件的模板，下面的例子设置了ToolTip风格：

```
<Window x:Class="TemplateControlEmbededInStyle.Window1"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="TemplateControlEmbededInStyle" Height="200"
Width="300">
  <Window.Resources>
    <SolidColorBrush x:Key="SolidRedBrush" Color ="Red"/>
    <SolidColorBrush x:Key="SolidBorderBrush" Color="#888" />
    <Style x:Key="{x:Type ToolTip}" TargetType="ToolTip">
    <Setter Property="OverridesDefaultStyle" Value="true"/>
    <Setter Property="HasDropShadow" Value="True"/>
    <Setter Property="Template">
      <Setter.Value>
        <ControlTemplate TargetType="ToolTip">
          <Border Name="Border"
                Background="{StaticResource SolidRedBrush}"
                BorderBrush="{StaticResource SolidBorderBrush}"
                BorderThickness="1"
                Width="{TemplateBinding Width}"
                Height="{TemplateBinding Height}">
            <ContentPresenter
                TextElement.Foreground="White"
```

```
                Margin="4"
                HorizontalAlignment="Left"
                VerticalAlignment="Top" />
          </Border>
          <ControlTemplate.Triggers>
            <Trigger Property="HasDropShadow" Value="true">
              <Setter TargetName="Border"
              Property="CornerRadius" Value="4"/>
              <Setter TargetName="Border"
              Property="SnapsToDevicePixels" Value="true"/>
            </Trigger>
          </ControlTemplate.Triggers>
        </ControlTemplate>
      </Setter.Value>
    </Setter>
  </Style>
  </Window.Resources>
  <StackPanel>
    <TextBlock Height ="30" Text ="王为"   Background ="Yellow" >
      <TextBlock.ToolTip>
        请输入你的名字!
      </TextBlock.ToolTip>
    </TextBlock >
    <Button Height ="20" Width ="40" Background ="Aqua"
       Content ="添加">
      <Button.ToolTip>添加一个记录</Button.ToolTip>
    </Button>
    </StackPanel>
</Window>
```

这样，所有控件的ToolTip都具有红底白字的风格。上面XAML的显示结果如图10-6所示：

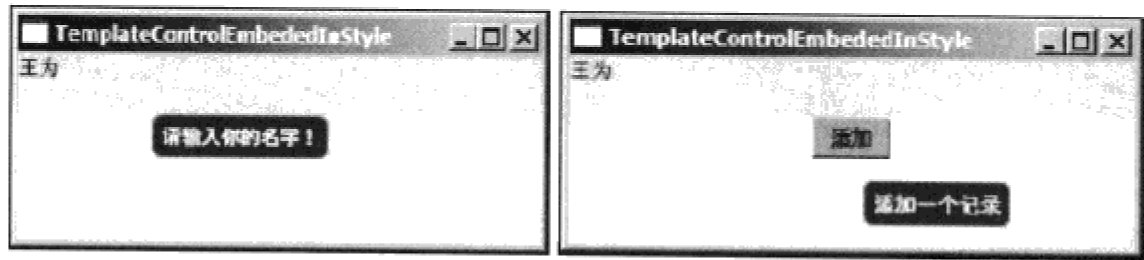


图10-6 在风格中使用控件模板

10.2.10 获取WPF控件的模板

在开发某类控件模板之前，我们往往需要研究已有的控件的模板和风格。这时可以读取WPF控件的默认模板，做法非常简单：

```
XmlWriterSettings xmlSetting = new XmlWriterSettings();
xmlSetting.Indent = true;
xmlSetting.IndentChars = new string( ' ', 5 );
xmlSetting.NewLineOnAttributes = true;

XmlWriter xw =
   XmlWriter.Create("C:\\SCMS\\Control.xaml",xmlSetting);
```

```
System.Windows.Controls.Button btn =new
    System.Windows.Controls.Button();
btn.Content = "Test";
XamlWriter.Save(btn.Template, xw);
```

上面这段程序，把控件的默认模板存到Control.Xaml文件中。在调试上面的程序时，笔者发现只有设置按钮的Content属性之后，按钮的Template才不为null，这说明控件的模板只在需要的时候才加上的。

10.3 数据模板（DataTemplate）

数据模板可能是程序员最常用的一种模板，与控件模板替代整个控件的外观不同，数据模板可用来按自己的需要显示相应的数据。

10.3.1 我们所面临的问题

假设有一个应用程序，需要显示客户的地址，客户的信息一般存在数据库中。这里为简化起见，我们用.NET的对象表示。客户地址中包含国家、省、城市、街道、邮政编码等信息，我们用AddressInfo和AddressList两个类来表示：

```
namespace Yingbao.Chapter10
{
    using System;
    using System.Collections.Generic;
    using System.Text;
    using System.Collections.ObjectModel;
    using System.ComponentModel;
    using System.Collections.Specialized;
    public class AddressInfo
    {
        private string streetNo;
        private string streetName;
        private string city;
        private string province;
        private string country;
        private string postCode;
        private bool chineseFormat;

        public bool ChineseFormat
        {
            get { return this.chineseFormat; }
            set { this.chineseFormat = value; }
        }

        public string StreetNo
        {
            get { return this.streetNo; }
            set { this.streetNo = value; }
        }

        public string StreetName
```

```
    {
        get { return this.streetName; }
        set { this.streetName = value; }
    }

    public string City
    {
        get { return this.city; }
        set { this.city = value; }
    }

    public string Province
    {
        get { return this.province; }
        set { this.province = value; }
    }

    public string Country
    {
        get { return this.country; }
        set { this.country = value; }
    }

    public string PostCode
    {
        get { return this.postCode; }
        set { this.postCode = value; }
    }

    public AddressInfo()
    {
    }

    public AddressInfo( string country, string province,
        string city, string streetName, string streetNo,
        string postCode, bool chineseFormat )
    {
        this.country = country;
        this.province = province;
        this.city = city;
        this.streetName = streetName;
        this.streetNo = streetNo;
        this.postCode = postCode;
        this.chineseFormat = chineseFormat;
    }
}

public class AddressList : ObservableCollection<AddressInfo>
{
    public AddressList():base()
    {
        this.Add(new AddressInfo(
            "中国","","北京","学院路","178","100080",true ));
```

```
            this.Add(new AddressInfo(
                "中国","","北京","望京南路","10","100089",true ));
            this.Add(new AddressInfo(
                "中国","安徽","合肥","长江路","285","241000",true ));
            this.Add(new AddressInfo(
                "中国","安徽","芜湖","学院路","44","242000",true ));
            this.Add(new AddressInfo(
                "中国","陕西","西安","咸宁路","79","710049",true ));
            this.Add(new AddressInfo("Canada", "Ontario",
                "Toronto", "Salem", "62",
                "M2M 3C1", false));
            this.Add(new AddressInfo("Canada", "Alberta",
               "Calgary", "Hopewell", "2728", "T1J 3Y3", false));
        }
    }
}
```

AddressList表示由多个地址组成的地址表。在这个地址表中分别存有中国和加拿大两种格式的地址，现在我们要在界面上显示这个地址表。显示地址表的最简单的方法是使用ListBox，现在从下面XAML开始：

```
<Window x:Class="Yingbao.Chapter10.MainWindow"
 xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="使用数据模板显示地址" Height="300" Width="300"
       xmlns:src="clr-namespace:Yingbao.Chapter10">
  <Window.Resources>
    <ObjectDataProvider x:Key="addressList"
                ObjectType="{x:Type src:AddressList}"/>
  </Window.Resources >
  <StackPanel >
    <ListBox  ItemsSource="{Binding Source={StaticResource
              addressList}}"/>
  </StackPanel>
  </Window>
```

我们用ObjectDataProvider引入AddressList，有关ObjectDataProvider和数据绑定，将在第11章数据绑定中详细讨论。运行上面的程序，得到如图10-7所示的结果：

图10-7 在不用数据模板时，ListBox中所显示的结果

对于AddressList中的每一项，在没有指示ListBox如何显示的情况下，List-Box会调用AddressInfo的ToString()方法来显示AddressInfo。显然这一结果不是我们希望的。当然可以使用虚函数覆盖技术

在AddressInfo中覆盖ToString方法，但结果是只能显示一个字符串。如果要达到更好的显示效果，需要用到数据模板DataTemplate。

10.3.2 定义数据模板

在ListBox中有一个属性：ItemTemplate，WPF用这个属性来显示ListBox中的每个条目，若这个属性为null，WPF就调用每个对象的ToString()方法。这个属性的类型为DataTemplate，我们可用自己的数据模板来显示所要的信息。

下面的XAML，笔者设计了显示地址的数据模板：

```
<ListBox ItemsSource=
          "{Binding Source={StaticResource addressList}}">
      <ListBox.ItemTemplate>
        <DataTemplate>
          <Border Name="border" BorderBrush="Aqua"
                  BorderThickness="1"
              Padding="5" Margin="5">
            <StackPanel>
              <StackPanel Orientation="Horizontal">
                <TextBlock Text="{Binding Path=PostCode}"
                   Margin="5,0,0,0" />
                <TextBlock Text="{Binding Path=Country}"
                   Margin="5,0,0,0"/>
              </StackPanel>
              <StackPanel Orientation="Horizontal">
                <TextBlock Text="{Binding Path=Province}"
                   Margin="5,0,0,0"/>
                <TextBlock Text="{Binding Path=City}"
                   Margin="5,0,0,0"/>
                <TextBlock Text="{Binding Path=StreetName}"
                   Margin="5,0,0,0"/>
                <TextBlock Text="{Binding Path=StreetNo}"
                   Margin="5,0,0,0"/>
              </StackPanel>
            </StackPanel>
          </Border >
        </DataTemplate>
      </ListBox.ItemTemplate >
    </ListBox >
```

由于WPF自动把AddressInfo绑定到ItemTemplate，所以，在数据模板中，可以直接绑定AddressInfo中的属性。

使用数据模板显示地址信息的结果如图10-8所示，有关中国的地址很好地在ListBox中显示了出来。

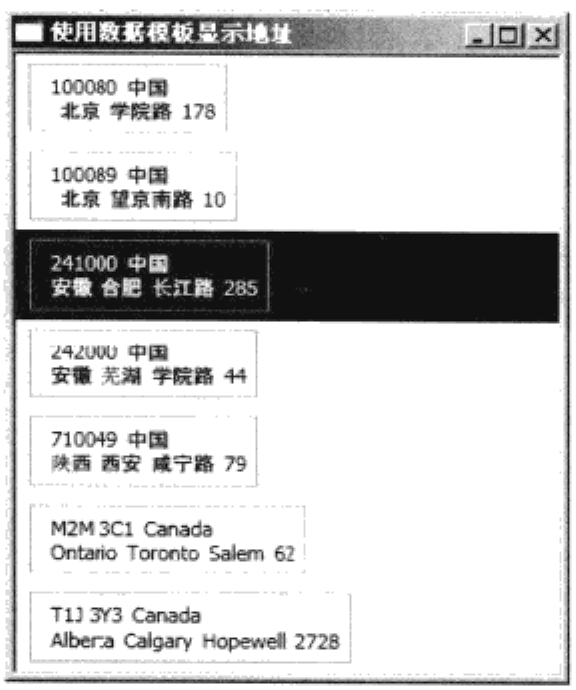


图10-8 使用数据模板显示地址

10.3.3 在资源中使用数据模板

有时需要在多个控件中共享数据模板，那么可以把数据模板放在资源中，其语法如下：

```
<DataTemplate x:Key="myAddressTemplate">
….
</DataTemplate >
```

现在，笔者把上面的XAML改写为：

```
<Window x:Class="Yingbao.Chapter10.MainWindow"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="使用数据模板显示地址" Height="300" Width="300"
       xmlns:src="clr-namespace:Yingbao.Chapter10">
  <Window.Resources>
    <ObjectDataProvider x:Key="addressList"
          ObjectType="{x:Type src:AddressList}"/>
    <DataTemplate x:Key ="myAddressTemplate">
      <Border Name="border" BorderBrush="Aqua"
          BorderThickness="1" Padding="5" Margin="5">
        <StackPanel>
          <StackPanel Orientation="Horizontal">
            <TextBlock Text="{Binding Path=PostCode}"
                Margin="5,0,0,0"/>
            <TextBlock Text="{Binding Path=Country}"
                Margin="5,0,0,0"/>
          </StackPanel>
          <StackPanel Orientation="Horizontal">
            <TextBlock Text="{Binding Path=Province}"
                Margin="5,0,0,0"/>
            <TextBlock Text="{Binding Path=City}"
                Margin="5,0,0,0"/>
            <TextBlock Text="{Binding Path=StreetName}"
```

```
                Margin="5,0,0,0"/>
            <TextBlock Text="{Binding Path=StreetNo}"
                Margin="5,0,0,0"/>
          </StackPanel>
        </StackPanel>
      </Border >
    </DataTemplate>
  </Window.Resources >
  <StackPanel  Orientation ="Horizontal">
    <ListBox
      ItemsSource="{Binding Source={StaticResource addressList}}"
      ItemTemplate="{StaticResource myAddressTemplate}"/>
    <ListBox
      ItemsSource="{Binding Source={StaticResource addressList}}"
      ItemTemplate="{StaticResource myAddressTemplate}"/>
  </StackPanel>
</Window>
```

注意：在窗口中用了两个ListBox，这两个ListBox中的ItemTemplet都使用同一个数据模板。当然，在实际应用程序中，不可能在同一个视窗中，使用两个一样的ListBox，这么做是为了突出"共享"数据模板的技术。这段XAML改写的程序的运行结果如图10-9所示。

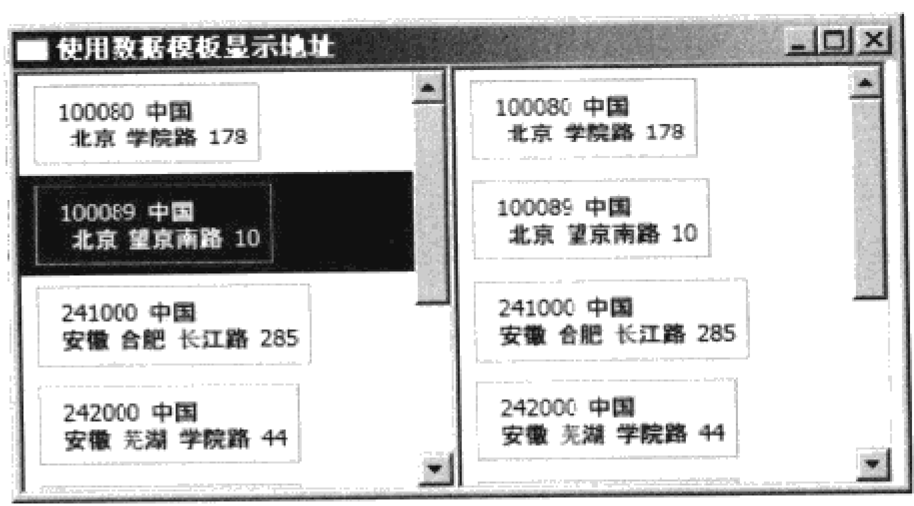


图10-9 使用资源共享数据模板

10.3.4 数据模板触发器

数据模板支持触发器，使用触发器，我们可以根据一定的条件来设定控件的属性。例如，可以把地址在中国的条目加上红框：

```
<DataTemplate>
    .....
   <DataTemplate.Triggers>
        <DataTrigger Binding="{Binding Path=ChineseFormat}"
              Value ="True">
          <Setter TargetName="border" Property="BorderBrush"
              Value="Red"/>
        </DataTrigger>
      </DataTemplate.Triggers>
</DataTemplate>
```

使用数据触发器后的结果如图10-10所示。

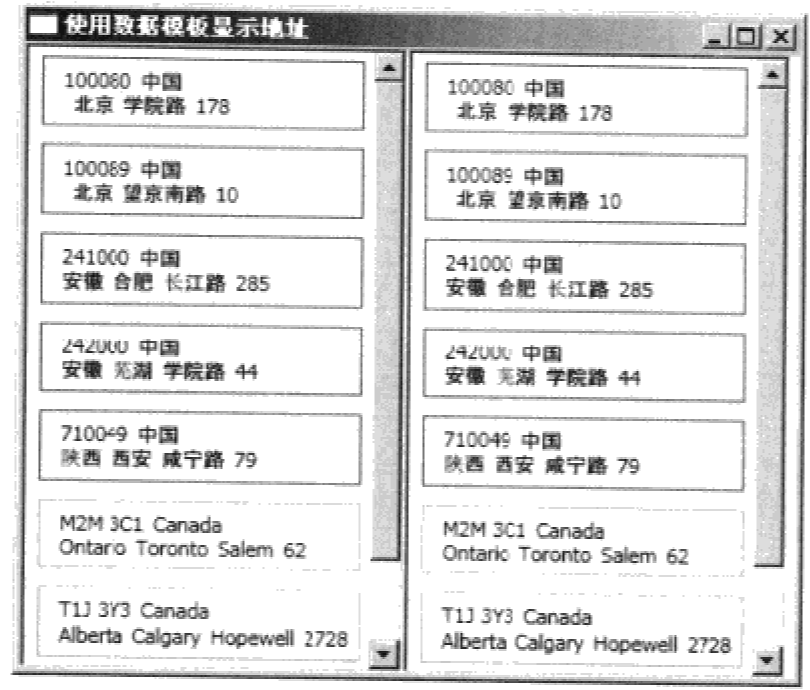


图10-10 使用数据触发器

10.3.5 根据数据属性选择相应的模板

在这个例子中，加拿大的地址格式是先写街道、再写城市、接着是省份、然后是邮编，最后才是国家的。即加拿大的地址应写为：

62 Salem Toronto

Ontario M2M 3C1 Canada

为此，需要写两个DataTemplate，一个用来显示中国地址，一个用来显示加拿大地址。

```
<DataTemplate x:Key ="chineseAddressFormat">
  ......
</DataTemplate>
<DataTemplate x:Key ="canadianAddressFormat">
 ......
</DataTemplate>
```

然后，要写一个AddressDataTemplateSelector类，这个类从DataTemplator派生出来：

```
public class AddressDataTemplateSelector : DataTemplateSelector
{
   public override DataTemplate
      SelectTemplate(object item, DependencyObject container)
   {
     if (item != null && item is AddressInfo)
        {
          AddressInfo addrInfo = item as AddressInfo;
          Window window = Application.Current.MainWindow;
          if (addrInfo.ChineseFormat)
             return window.FindResource("chineseAddressFormat")
             as DataTemplate;
          else
```

```
            return window.FindResource("canadianAddressFormat")
            as DataTemplate;
        }
      return null;
   }
}
```

对于数据模板选择类，唯一需要做的就是覆盖SelectTemplate方法。当我们把该类和ListBox中ItemTemplateSelector属性相连之后，ListBox在显示每个条目之前都要调用SelectTemplate方法。

同样，要把AddressDataTemplateSelector放到资源中，以便引用：

```
<Window.Resources>
…
<src:AddressDataTemplateSelector
x:Key="addressDataTemplateSelector"/>
….
</Window.Resources>
```

最后，设置ListBox中ItemTemplateSelector属性：

```
<ListBox  HorizontalContentAlignment="Stretch"
  ItemTemplateSelector="{StaticResource
  addressDataTemplateSelector}"
  ItemsSource="{Binding Source={StaticResource addressList}}" />
```

注意：这时候不需要设置ItemTemplate，因为，这时ItemTemplate由AddressDataTemplateSelector根据条目来进行设置。

下面是完整的XAML程序，运行的结果如图10-11所示。由图10-11可见，ListBox中的加拿大地址和中国地址都正确地显示了出来。

```
<Window x:Class="Yingbao.Chapter10.MainWindow"
 xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="使用数据模板显示地址" Height="300" Width="300"
       xmlns:src="clr-namespace:Yingbao.Chapter10">
  <Window.Resources>
    <ObjectDataProvider x:Key="addressList"
           ObjectType="{x:Type src:AddressList}"/>
    <src:AddressDataTemplateSelector
           x:Key="addressDataTemplateSelector"/>
    <DataTemplate x:Key ="chineseAddressFormat">
      <Border Name="border" BorderBrush="Aqua" BorderThickness="1"
              Padding="5" Margin="5">
        <StackPanel>
          <StackPanel Orientation="Horizontal">
            <TextBlock Text="{Binding Path=PostCode}"
              Margin="5,0,0,0" />
            <TextBlock Text="{Binding Path=Country}"
              Margin="5,0,0,0"/>
          </StackPanel>
          <StackPanel Orientation="Horizontal">
            <TextBlock Text="{Binding Path=Province}"
```

```
            Margin="5,0,0,0"/>
          <TextBlock Text="{Binding Path=City}"
            Margin="5,0,0,0"/>
          <TextBlock Text="{Binding Path=StreetName}"
            Margin="5,0,0,0"/>
          <TextBlock Text="{Binding Path=StreetNo}"
            Margin="5,0,0,0"/>
        </StackPanel>
      </StackPanel>
    </Border >
    <DataTemplate.Triggers>
      <DataTrigger Binding="{Binding Path=ChineseFormat}"
            Value ="True">
        <Setter TargetName="border" Property="BorderBrush"
            Value="Red"/>
      </DataTrigger>
    </DataTemplate.Triggers>
  </DataTemplate>
  <DataTemplate x:Key ="canadianAddressFormat">
    <Border Name="border" BorderBrush="Aqua" BorderThickness="1"
            Padding="5" Margin="5">
      <StackPanel>
        <StackPanel Orientation="Horizontal">
          <TextBlock Text="{Binding Path=StreetNo}"
            Margin="5,0,0,0"/>
          <TextBlock Text="{Binding Path=StreetName}"
            Margin="5,0,0,0"/>
          <TextBlock Text="{Binding Path=City}"
            Margin="10,0,0,0"/>
        </StackPanel>
        <StackPanel Orientation="Horizontal">
          <TextBlock Text="{Binding Path=Province}"
            Margin="5,0,0,0"/>
          <TextBlock Text="{Binding Path=PostCode}"
            Margin="5,0,0,0" />
          <TextBlock Text="{Binding Path=Country}"
            Margin="5,0,0,0"/>
        </StackPanel>
      </StackPanel>
    </Border >
  </DataTemplate>
 </Window.Resources >
 <StackPanel  Orientation ="Horizontal">
<ListBox  HorizontalContentAlignment="Stretch"
     ItemTemplateSelector="{StaticResource
      addressDataTemplateSelector}" ItemsSource="{Binding
      Source={StaticResource addressList}}" />
<ListBox  HorizontalContentAlignment="Stretch"
      ItemTemplateSelector="{StaticResource
      addressDataTemplateSelector}" ItemsSource="{Binding
      Source={StaticResource addressList}}" />
 </StackPanel>
 </Window>
```

图10-11 区别显示中国格式和加拿大格式的地址

10.3.6 在数据模板中使用类型转换技术

在使用模板时，我们会用到数据间的类型转换。下面笔者用实例说明在显示时间的模板中如何进行类型转换。

在这个例子中我们要做的是利用数据模板显示卡尔加里、北京、东京和多伦多的时间。笔者在给国内和多伦多的朋友打电话之前，要先换算一下当地的时间，所以写一个显示各地时间的程序是一个不错的主意。预先设定要做到的结果如图10-12所示。

和前面的地址类似，首先，我们需要一个描述时间的C#类LocalTime：

```
namespace Yingbao.Chapter10.UseDataTemplate
{

   public class LocalTime
    {
        private string place;
        private DateTime time;
        public LocalTime(string place, DateTime time)
        {
           this.place = place;
            this.time = time;
        }
        public DateTime Time
        {
            get { return time; }
            set { time = value; }
        }
        public string Place
        {
            get { return place; }
        }
    }
}
```

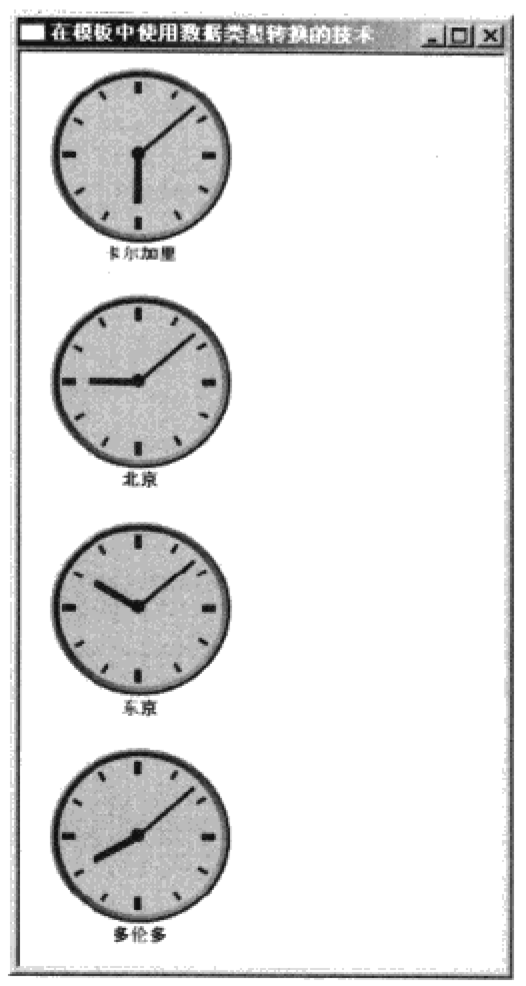


图10-12 用数据模板显示时区

这个类中包括时间和地点信息。然后，设定包括上面几个地点的时间集合：

```
public class LocalTimes : ObservableCollection<LocalTime>
    {
        public LocalTimes()
        {
            this.Add(new LocalTime("卡尔加里", DateTime.Now));
            this.Add(new LocalTime("北京",
                DateTime.Now.AddHours(+15)));
            this.Add(new LocalTime("东京",
                DateTime.Now.AddHours(+16)));
            this.Add(new LocalTime("多伦多",
                DateTime.Now.AddHours(+2)));
        }
    }
```

由于笔者住在卡尔加里，所以把卡尔加里的时间设为现在，其他城市的时间加上相应的时差。

下面的两个类是把小时的值和分钟值转换成角度的转换器类：

```
public class HourToAngle : IValueConverter
 {
        #region IValueConverter Members
    public object Convert(object value, Type targetType,
      object parameter, System.Globalization.CultureInfo culture)
    {
      DateTime time = System.Convert.ToDateTime(value);
```

```
        double Angle = time.Hour * 30;
        Angle += 12 * time.Minute / 60;
        return Angle;
    }

    public object ConvertBack(object value, Type targetType,
       object parameter, System.Globalization.CultureInfo culture)
        {
            return null;
        }
        #endregion
    }

    public class MinuteToAngle : IValueConverter
    {
        #region IValueConverter Members
       public object Convert(object value, Type targetType,
       object parameter, System.Globalization.CultureInfo culture)
        {
            DateTime time = System.Convert.ToDateTime(value);
            double Angle = time.Minute * 6;
            return Angle;
        }

       public object ConvertBack(object value, Type targetType,
       object parameter, System.Globalization.CultureInfo culture)
        {
            return null;
        }
        #endregion
    }
```

写数据类型转换器类，需要做的是移植IValueConverter接口。在这个例子中，不必把时针或分针的位置转换为相应的时分值，所以上面两个类中的ConvertBack均返回空。

在XAML中，需要先在资源中创建这两个转化器类：

```
<src:HourToAngle x:Key="hourToAngle"/>
<src:MinuteToAngle x:Key="minuteToAngle"/>
```

在模板中，使用数据绑定时，设置相应的转换器，如画时针时用的绑定：

```
<RotateTransform x:Name="HourHand2" CenterX="2" CenterY="30"
Angle="{Binding Mode=OneWay, Converter={StaticResource
hourToAngle}}">
```

有关图形转换类RotateTransfor，将在第13章二维图形中进行详细介绍。到时候可回过头来看这个例子。下面是完整的XAML：

```
<Window x:Class="Yingbao.Chapter10.UseDataTemplate.MainWindow"
 xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="在模板中使用数据类型转换的技术" Height="300" Width="300"
```

```
    xmlns:src="clr-namespace:Yingbao.Chapter10.UseDataTemplate">
<Window.Resources>
  <ObjectDataProvider x:Key="localTimes"
     ObjectType="{x:Type src:LocalTimes}"/>
  <ControlTemplate x:Key="clockTemplate">
    <Grid>
      <Grid.Resources>
        <src:HourToAngle x:Key="hourToAngle"/>
        <src:MinuteToAngle x:Key="minuteToAngle"/>
      </Grid.Resources>
      <Ellipse Width="108" Height="108" StrokeThickness="3">
        <Ellipse.Stroke>
          <LinearGradientBrush>
            <GradientStop Color="LightBlue" Offset="0" />
            <GradientStop Color="DarkBlue" Offset="1" />
          </LinearGradientBrush>
        </Ellipse.Stroke>
      </Ellipse>
      <Ellipse VerticalAlignment="Center"
          HorizontalAlignment="Center" Width="104"
          Height="104" Fill="LightBlue" StrokeThickness="3">
        <Ellipse.Stroke>
          <LinearGradientBrush>
            <GradientStop Color="DarkBlue" Offset="0" />
            <GradientStop Color="LightBlue" Offset="1" />
          </LinearGradientBrush>
        </Ellipse.Stroke>
      </Ellipse>
      <Canvas Width="102" Height="102">
        <Ellipse Width="8" Height="8" Fill="Black"
              Canvas.Top="46" Canvas.Left="46" />
        <Rectangle Canvas.Top="5" Canvas.Left="48" Fill="Black"
              Width="4" Height="8">
          <Rectangle.RenderTransform>
          <RotateTransform CenterX="2" CenterY="46" Angle="0" />
          </Rectangle.RenderTransform>
        </Rectangle>
        <Rectangle Canvas.Top="5" Canvas.Left="49" Fill="Black"
               Width="2" Height="6">
          <Rectangle.RenderTransform>
            <RotateTransform CenterX="2" CenterY="46"
               Angle="30" />
          </Rectangle.RenderTransform>
        </Rectangle>
        <Rectangle Canvas.Top="5" Canvas.Left="49" Fill="Black"
               Width="2" Height="6">
          <Rectangle.RenderTransform>
            <RotateTransform CenterX="2" CenterY="46"
               Angle="60" />
          </Rectangle.RenderTransform>
        </Rectangle>
        <Rectangle Canvas.Top="5" Canvas.Left="48" Fill="Black"
               Width="4" Height="8">
```

```
  <Rectangle.RenderTransform>
    <RotateTransform CenterX="2" CenterY="46"
       Angle="90" />
  </Rectangle.RenderTransform>
</Rectangle>
<Rectangle Canvas.Top="5" Canvas.Left="49" Fill="Black"
       Width="2" Height="6">
  <Rectangle.RenderTransform>
    <RotateTransform CenterX="2" CenterY="46"
       Angle="120" />
  </Rectangle.RenderTransform>
</Rectangle>
<Rectangle Canvas.Top="5" Canvas.Left="49" Fill="Black"
       Width="2" Height="6">
  <Rectangle.RenderTransform>
    <RotateTransform CenterX="2" CenterY="46"
       Angle="150" />
  </Rectangle.RenderTransform>
</Rectangle>
<Rectangle Canvas.Top="5" Canvas.Left="48" Fill="Black"
       Width="4" Height="8">
  <Rectangle.RenderTransform>
    <RotateTransform CenterX="2" CenterY="46"
       Angle="180" />
  </Rectangle.RenderTransform>
</Rectangle>
<Rectangle Canvas.Top="5" Canvas.Left="49" Fill="Black"
       Width="2" Height="6">
  <Rectangle.RenderTransform>
    <RotateTransform CenterX="2" CenterY="46"
       Angle="210" />
  </Rectangle.RenderTransform>
</Rectangle>
<Rectangle Canvas.Top="5" Canvas.Left="49" Fill="Black"
       Width="2" Height="6">
  <Rectangle.RenderTransform>
    <RotateTransform CenterX="2" CenterY="46"
       Angle="240" />
  </Rectangle.RenderTransform>
</Rectangle>
<Rectangle Canvas.Top="5" Canvas.Left="48" Fill="Black"
     Width="4" Height="8">
  <Rectangle.RenderTransform>
    <RotateTransform CenterX="2" CenterY="46"
     Angle="270" />
  </Rectangle.RenderTransform>
</Rectangle>
<Rectangle Canvas.Top="5" Canvas.Left="49" Fill="Black"
     Width="2" Height="6">
  <Rectangle.RenderTransform>
    <RotateTransform CenterX="2" CenterY="46"
     Angle="300" />
  </Rectangle.RenderTransform>
```

```
          </Rectangle>
          <Rectangle Canvas.Top="5" Canvas.Left="49" Fill="Black"
               Width="2" Height="6">
            <Rectangle.RenderTransform>
              <RotateTransform CenterX="2" CenterY="46"
                Angle="330" />
            </Rectangle.RenderTransform>
          </Rectangle>
         <Rectangle x:Name="HourHand" Canvas.Top="21"
                Canvas.Left="48" Fill="Black"
                Width="4" Height="30">
            <Rectangle.RenderTransform>
              <RotateTransform x:Name="HourHand2" CenterX="2"
                CenterY="30" Angle="{Binding Mode=OneWay,
                   Converter={StaticResource hourToAngle}}">
              </RotateTransform>
            </Rectangle.RenderTransform>
          </Rectangle>
          <Rectangle x:Name="MinuteHand" Canvas.Top="6"
              Canvas.Left="49" Fill="Black"
              Width="2" Height="45">
            <Rectangle.RenderTransform>
              <RotateTransform CenterX="1" CenterY="45"
                 Angle="{Binding Mode=OneWay,
                 Converter={StaticResource minuteToAngle}}">
              </RotateTransform>
            </Rectangle.RenderTransform>
          </Rectangle>
        </Canvas>
      </Grid>
    </ControlTemplate>
    <DataTemplate x:Key="ShowTime"
      DataType="Yingbao.Chapter10.UseDataTemplate.LocalTime">
      <StackPanel Margin="10">
        <Control Template="{StaticResource clockTemplate}"
           Width="120" Height="108"
           DataContext="{Binding Path=Time}"/>
        <TextBlock Text="{Binding Path=Place}"
           HorizontalAlignment="Center"/>
      </StackPanel>
    </DataTemplate>
  </Window.Resources>
  <Grid>
    <ListBox ItemsSource="{Binding Source={StaticResource
        localTimes}}" ItemTemplate="{StaticResource ShowTime}"/>
  </Grid>
</Window>
```

10.4 ItemsPanelTemplate

在条目控件ItemsControl中，有一个属性ItemsPanel。这个属性的类型是ItemsPanelTemplate，设置这个属性是改变条目控件中的条目在控件里面的排版。若研究条目控件的模板，可以发现，ListBox

中的ItemsPanel为VisualizingStackPanel；Menu的ItemsPanel设为WrapPanel；而StatusBar中的ItemsPanel则是使用StackPanel。

比如对10.3.6节的显示各地时间的程序，若想把时钟由竖排改为横排，可以在资源中加入下面的风格：

```
<Style TargetType="{x:Type ListBox}">
  <Setter Property="ItemsPanel">
    <Setter.Value>
      <ItemsPanelTemplate>
        <StackPanel Orientation="Horizontal"
                    VerticalAlignment="Center"
                    HorizontalAlignment="Center"/>
      </ItemsPanelTemplate>
    </Setter.Value>
  </Setter>
</Style>
```

在原程序中加入这一风格后的运行结果如图10-13所示，很简单，笔者只是改变了StackPanel的排版方向。

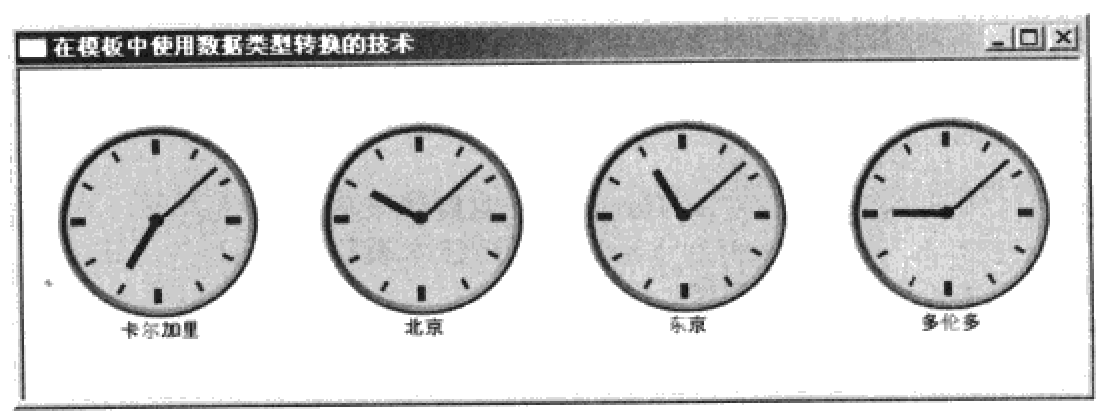


图10-13 横排时钟

10.5 层次结构数据模板（HierarchicalDataTemplate）

从图10-1可见，HierarchicalDataTemplate类是从DataTemplate中派生出来的，这个类是为具有分层结构的数据模型设计的。在现实生活中分层数据到处都是，比如说，大学里面设有学院和研究所，学院下面设有系，系下面设有教研室。再比如说，电力系统中的通信模型中，调度中心管理各个变电站，变电站里有数据收集器，数据收集器管理变电中的微机保护、数据采集、故障录播等智能仪器，等等。

下面，笔者以西安交通大学的院系设置为例，来用HierarchicalDataTemplate显示分层数据。这里的数据取自西安交通大学的网站，笔者曾经分别在电气工程系和信息与控制工程系学习过，只是，去交大的网站上一查，过去的教授们都已退休了，那些留校的同学也都成了博导，成为交大的骨干。在下面的XAML中，首先用XML引入西安交通大学电气学院和电子和信息工程学院的机构设置。

```
<Window x:Class="Yingbao.Chapter10.JiaotongUniversityWindow"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
     xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
```

```
    Title="西安交通大学院系设置" Height="300" Width="300">
<Window.Resources>
    <XmlDataProvider x:Key ="JiaotongUniversity" XPath
      ="Universities">
     <x:XData>
       <Universities xmlns="">
         <University Name="西安交通大学" >
           <Schools>
             <School Name="电气学院" startYear="1908">
               <Departments>
                 <Department  Name="电机电器及控制系">
                   <Institutes>
                     <Institute Name="电机教研室"/>
                     <Institute Name="电器教研室"/>
                   </Institutes>
                 </Department>
                 <Department  Name="高电压与绝缘技术系">
                   <Institutes>
                     <Institute Name="高电压技术教研室"/>
                     <Institute Name="绝缘教研室"/>
                   </Institutes>
                 </Department >
                 <Department  Name="工业自动化系">
                   <Institutes>
                     <Institute Name="工业自动化教研室"/>
                     <Institute Name="测控技术教研室"/>
                   </Institutes>
                 </Department>
                 <Department  Name="电力工程系"/>
               </Departments>
             </School>
             <School Name="电子与信息工程学院" startYear="1957">
               <Departments>
                 <Department  Name="电子科学与技术系">
                   <Institutes>
                     <Institute Name="电子物理与器件"/>
                     <Institute Name="电子材料与器件"/>
                     <Institute Name="信息材料与元器件"/>
                   </Institutes>
                 </Department>
                 <Department  Name="信息与通信工程系">
                   <Institutes>
                     <Institute Name="3COM"/>
                     <Institute Name="信息工程研究所"/>
                     <Institute Name="信息处理与识别研究所"/>
                     <Institute Name="微波工程与光通讯研究所"/>
                     <Institute Name="数据广播研究中心"/>
                     <Institute Name="数据广播研究中心"/>
                   </Institutes>
```

```
                </Department >
                <Department  Name="计算机科学与技术系">
                  <Institutes>
                    <Institute Name="计算机系统结构与网络研究所"/>
                    <Institute Name="计算机软件与理论研究所"/>
                    <Institute Name="计算机应用技术研究所"/>
                    <Institute Name="新型机研究所"/>
                    <Institute Name="数据通信网络与工程研究所"/>
                  </Institutes>
                </Department>
                <Department  Name="自动化科学与技术系">
                  <Institutes>
                    <Institute Name="自动控制研究所"/>
                    <Institute Name="人工智能与机器人研究所"/>
                    <Institute Name="综合自动化研究所"/>
                    <Institute Name="系统工程研究所"/>
                  </Institutes>
                </Department>
              </Departments>
            </School>
          </Schools>
        </University>
      </Universities>
    </x:XData>
  </XmlDataProvider>
  <HierarchicalDataTemplate DataType ="University"
      ItemsSource ="{Binding XPath=Schools/School}">
    <StackPanel  Background ="AliceBlue">
      <TextBlock FontSize ="20" Text="{Binding XPath=@Name}"/>
    </StackPanel>
  </HierarchicalDataTemplate>
  <HierarchicalDataTemplate DataType ="School"
      ItemsSource="{Binding XPath=Departments/Department}">
    <StackPanel  Background ="LightBlue">
      <TextBlock FontSize ="18"  Text="{Binding XPath=@Name}"/>
    </StackPanel>
  </HierarchicalDataTemplate>
  <HierarchicalDataTemplate DataType ="Department"
       ItemsSource="{Binding XPath=Institutes/Institute}">
    <StackPanel  Background ="LightSlateGray">
      <TextBlock FontSize ="18"  Text="{Binding XPath=@Name}"/>
    </StackPanel>
  </HierarchicalDataTemplate>
  <HierarchicalDataTemplate DataType ="Institute">
    <StackPanel  Background ="LightSalmon">
      <TextBlock FontSize ="16" Text="{Binding XPath=@Name}"/>
    </StackPanel>
  </HierarchicalDataTemplate>
</Window.Resources>
<TreeView ItemsSource ="{Binding Source={StaticResource
        JiaotongUniversity}, XPath=University}"/>
```

```
</Window>
```

然后用HierarchicalDataTemplate在界面上显示XML数据，由于HierarchicalDataTemplate是DataTemplate的派生类，所以，WPF可根据数据的类型选择合适的DataTemplate。在各个HierarchicalDataTemplate中，使用不同大小的字体和不同的背景色来分别显示大学、学院、系、教研室的名字。上述XAML的运行结果如图10-14所示。

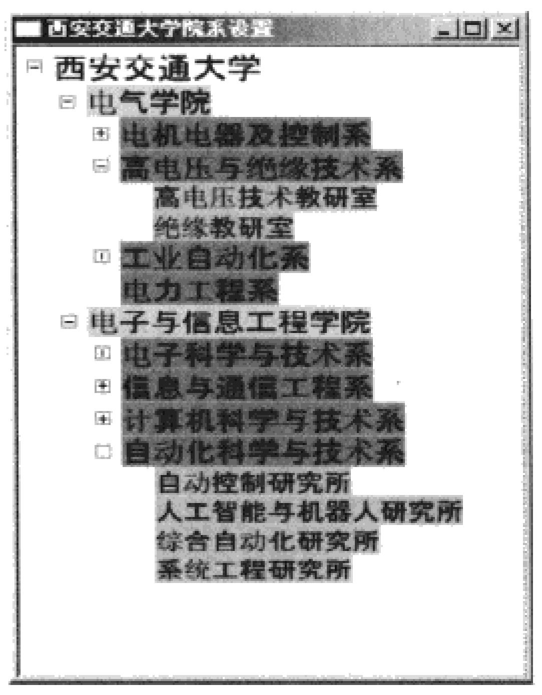


图10-14 用层次结构数据模板显示的院系设置图

10.6 本章小结

本章详细介绍了WPF中的各种模板类型，这些类型都是针对不同的控件或数据结构设计的。可以用控件模板完全替换控件原有的“外观”；也可以部分地替换原有控件的“外观”；还可以根据不同的数据类型，使用不同的模板；也可以根据数据中的某个属性来选择不同的显示模板。作为应用程序的开发人员，过去我们常常要开发自定义控件（或用户化控件），现在，只要开发不同的模板就可以达到目的。模板常常和风格结合起来使用，使得WPF控件的表现力极为丰富。

第11章 数据绑定（Data Binding）

数据绑定是把WPF中的UI控件和数据元素连接在一起的技术。这一定义有点抽象，但实际上却反映了数据绑定技术的多样性。数据绑定的范围非常广泛，它可以是简单地把选择按钮（Radio）连接到一个逻辑变量上，也可以复杂到把数据库和输入面板相连。

控件通常起两个作用：一是把数据用某种方式呈现在用户面前；二是接受用户的输入，并把用户输入读到相关的数据对象中。在应用程序中，这一过程常常是自动完成的，其中就是使用数据绑定技术。

在WPF中使用数据绑定技术不仅可以使程序员写出的代码更加简洁明了，而且这种绑定可以在XAML中通过声明的方式来完成。无疑，这是一个实用的技术。但另一方面，WPF掩藏了大量的内部处理过程，所以一旦数据绑定出现问题，初学者往往不知道从何着手。

本章将系统地介绍WPF数据绑定的原理及其内部机制，源对象和目标对象之间的类型转换，在绑定中对用户输入的数据进行校验并在UI元素上显示错误信息，对XML数据对象及集合对象的绑定等一系列相关问题。

在笔者所接触的数据绑定技术中，WPF的数据绑定最为灵活，功能也是最强大的。

11.1 数据绑定概述

数据绑定发生在源对象和目标对象之间，当源对象或目标对象的属性值发生改变时，所绑定的对象也会跟着发生改变。数据绑定的目标对象一定是相关对象（Dependency Object），所绑定的目标对象的属性一定要是相关属性（Dependency Property），而源对象则既可以是相关对象（或属性），也可以是一般.NET对象（或属性）。

当我们把源对象和目标对象绑定到一起的时候，WPF就会把数据在目标对象和源对象之间进行传递，图11-1用UML语言简单显示了这种机制，遗憾的是，笔者所使用的UML工具不支持中文。伴随着数据的传递，有时候需要对数据的类型进行转换，甚至需要在其中插入对数据的限制条件（校验），用来防止不恰当的数据进入系统。在某些情况下，还需要控制数据交换发生的时刻（同步或异步）。

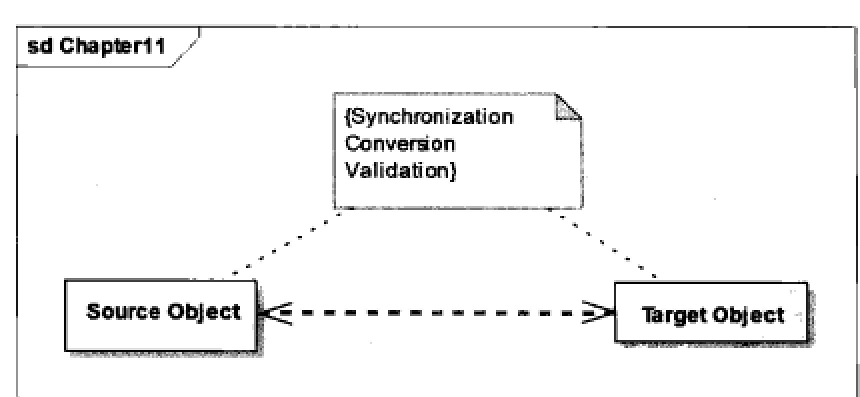


图11-1 数据绑定的UML图示。Source Object：源对象，Target Object：目标对象，Synchronization：同步，Conversion：类型转化，Validation：校验

WPF支持OneWay、TwoWay、OneTime、Default和OneWayToSource等多种绑定模式。

（1）OneWay是从源对象到目标对象的绑定，当源对象发生变化时，目标对象也跟着改变；但当目标对象发生变化时，源对象不会跟着变化。在这种情况下，目标对象是只读的。

（2）TwoWay，这种绑定模式下，不仅目标对象会随着源对象的变化而变化，而且源对象也会随着目标对象的变化而变化，换句话说，这种绑定是双向的。即目标对象是可读写对象。

（3）OneTime，在这种绑定模式下，源对象和目标对象只在初始化时发生作用，数据从源对象传递到目标对象。

（4）OneWayToSource，它与OneWay相似，不同的是源对象随着目标对象的变化而变化。

（5）Default, 在这种模式下，若源对象的属性是只读的，WPF会把绑定模式设为OneWay，若源对象的属性是读写属性，WPF会把绑定模式设为TwoWay，所以这种默认模式一般可以包括大多数情况。

当数据在源对象和目标对象之间流动时，有时目标对象和源对象数据的类型是不同的，需要进行类型转换。WPF提供了一些类型转换类，如BooleanToVisibilityConverter等，有时候需要程序员提供类型转换对象。这时，需要根据情况移植IValueConverter或IMultiValueConverter接口。

处理用户输入数据的用户界面的一个重要的功能是要能防止垃圾数据进入系统，比如：用户的支票账户的存款余额不能是负数；又比如小学生的年龄不能超过15岁等。在使用数据绑定的情况下，数据在源对象和目标对象之间的传递是自动的，我们必须在数据发生改变之前对数据进行校验。WPF提供了插入校验规则的方法，即用户为数据绑定提供ValidationRule对象。

在源对象是集合对象，如List、哈希表等的情况下，WPF 通过CollectionViewSource来创建不同的呈现方式。从而可以很方便地实现对同一数据源进行过滤、分组、查询、排序等功能，WPF中的数据绑定中包含了一个MVC（Model、View和Control）设计范例。

WPF直接支持XML数据的绑定，XML数据通过XmlDataProvider把XML数据直接和用户界面上的控件相连，在第10章模板中，我们已经遇到过XML数据的情况。

在某些特殊的情况下，WPF需要把来自不同对象的数据绑定到用户界面元素上，例如，你的网页上要显示天气、外汇牌价等信息，天气状况可能来自气象台的Web Service，外汇牌价可能来自银行等。WPF支持这种复杂的数据绑定，你可以通过CompositeCollection、MultiBinding、PriorityBinding来对数据源进行重新整合。

数据绑定技术和事件处理密切相关。数据绑定是在设计时刻（design time）完成的，当程序员设定了源对象和目标对象后，一切交给WPF了；而事件处理常发生在运行时刻（run time）。使用数据绑定技术的优点是一切都在初始化时完成（事件处理往往在后台），可以避免一些错误。缺点是一旦数据结构发生变化，则需要对整个初始化过程进行修改。

11.2 最简单的数据绑定——从界面元素到界面元素

11.2.1 一对一数据绑定

让我们从最简单的数据绑定开始，笔者在视窗上创建了三个元素，ScrollBar、Label和TextBox：

```
<Window
 xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="绑定到界面元素" Height="200" Width="300">
    <StackPanel>
      <ScrollBar Name="scroll"  Orientation ="Horizontal "
         Margin ="24" Maximum ="100" LargeChange ="10"
         SmallChange ="1"/>
        <StackPanel Orientation ="Horizontal">
            <Label  Height ="30">滚动块值: </Label>
            <TextBox   Name="txtScrollvalue" Height ="30"
              Width="100" HorizontalAlignment="Center"
             Text ="{Binding ElementName=scroll, Path=Value}"/>
        </StackPanel>
    </StackPanel>
</Window>
```

把TextBox中的Text属性绑定到ScrollBar的值上，当用鼠标单击ScrollBar时，我们可以看到TextBox中显示的数字在变化。这里ScrollBar就是源对象， TextBox 中的Text就是目标对象，目标对象和源对象都是WPF控件。如果不用数据绑定，而用事件处理程序来实现在TextBox 中显示滑动块的值，就可以在Window1类的C#代码中对ValueChanged事件进行处理，读出ScrollBar的当前值，并把该数值输出到TextBox中：

```
public void OnValueChange(object sender, EventArgs args)
        {
            string value = scroll.Value.ToString();
            txtScrollValue.Text = value;
        }
```

而使用数据绑定技术，只要在<TextBox>标记里加入Binding扩展：

```
Text ="{Binding ElementName=scroll, Path=Value}"
```

显然XAML代码比C#代码更为简洁。

注意：一定要把Binding扩展加在目标对象上，用ElementName来识别源对象，它就是ScrollBar的Name属性值，使用Path来表示要绑定的源对象的属性，这里为ScrollBar的Value属性。

但也可以采用例外一种写法：

```
<TextBox Name="txtScrollValue" Height ="30" Width ="200"
          HorizontalAlignment ="Center">
      <TextBox.Text>
         <Binding ElementName="scroll"  Path="Value"/>
      </TextBox.Text>
</TextBox>
```

这两种写法的效果是一样的，使用哪一种则完全是个人的风格。

11.2.2 在C#中，实现数据绑定

若你在一家大公司工作，界面设计和软件开发分属于两个不同的团队。通常界面设计团队负责编

写XAML，他们一般使用工具产生XAML，而软件开发团队负责写后台逻辑，这个时候你可能要在C#中加入数据绑定。在C#中加入数据绑定的语法如下：

```
Binding bd = new Binding();
bd.Source = scroll;
bd.Path = new PropertyPath(ScrollBar.ValueProperty);
txtScrollvalue.SetBinding(TextBox.TextProperty, bd);
```

注意：笔者这里用的ValueProperty和TextProperty。由于ScrollBar位于System.Windows.Controls.Primitive命名空间中，故需要在C#中应用该命名空间。

11.2.3 对不是FrameworkElement和FrameworkContentElement元素实现数据绑定

SetBinding方法是在FrameworkElement和FrameworkContentElement中定义的，对于大多数UI元素来说，实现数据绑定非常方便。有时候我们需要对非FrameworkElement和FrameworkContentElement进行绑定，这时，需要用到BindingOperations类：

```
Binding bd = new Binding();
bd.Source = scroll;
bd.Path = new PropertyPath(ScrollBar.ValueProperty);
BindingOperations. SetBinding (txtScrollValue,TextBox.TextProperty, bd);
```

11.3 使用不同的绑定模式

前面的章节提到过数据绑定的五种模式，为了验证这些绑定模式的效果，笔者把上面的XAML改为：

```
<Window
   mlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
   xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="各种模式下的绑定" Height="373" Width="337">
    <StackPanel>
      <ScrollBar Name="scroll"  Orientation ="Horizontal "
        Margin ="24" Maximum ="100" LargeChange ="10"
        SmallChange ="1"/>
      <Label Height ="30">OneWay:</Label>
      <TextBox Name="scrollValue1" Height ="20" Width ="200"
        HorizontalAlignment="Center"  Text ="{Binding
        ElementName=scroll, Path=Value, Mode=OneWay}"/>
      <Label Height ="30">TwoWay:</Label>
      <TextBox Name="scrollValue2" Height ="20" Width ="200"
        HorizontalAlignment="Center"  Text ="{Binding
        ElementName=scroll, Path=Value, Mode=TwoWay}"/>
      <Label Height ="30">OneTime:</Label>
      <TextBlock Name="scrollValue3" Height ="20" Width ="200"
        HorizontalAlignment="Center"  Text ="{Binding
        ElementName=scroll, Path=Value, Mode=OneTime}"/>
      <Label  Height ="30">OneWayToSource:</Label>
      <TextBox Name="scrollValue4" Height ="20" Width ="200"
        HorizontalAlignment="Center"  Text ="{Binding
        ElementName=scroll, Path=Value, Mode=OneWayToSource}"/>
      <Label  Height ="30">Default:</Label>
```

```
    <TextBlock Name="scrollValue5" Height ="20" Width ="200"
      HorizontalAlignment="Center"  Text ="{Binding
      ElementName=scroll, Path=Value,Mode=Default}"/>
  </StackPanel>
</Window>
```

笔者用5种模式来绑定滑动条的值，其结果如图11-2所示。当用鼠标单击滑动条时，可以看到在OneWay、TwoWay和Default绑定模式下，TextBox中的数值一起随滑块位置而变化。而在OneTime绑定模式下，TextBox的值总是0，这是因为ScrollBar初始化值为0；OneTime绑定模式下，TextBox的值仅决定于ScrollBar的初始值。

图11-2 分别用五种绑定模式绑定滑动条的值

现在我们在OneWayToSource绑定模式的TextBox中输入数值50，便可以看到滑动条中的滑块也跟着变化到中间的位置，这说明在OneWayToSource绑定模式下，其源对象的值随着目标对象的变化而变化。而且OneWay、TwoWay和Default绑定模式下的TextBox中值也一起变化。需要注意的是，你输入的数值一定要在ScrollBar的允许范围内。

在TwoWay绑定模式下，在TextBox中输入数据，可以看到与OneWayToSource一样的效果。

而在OneWay绑定模式下的TextBox中输入数据，则ScrollBar的滑块位置并不改变。

11.4 动态绑定

数据绑定把两个元素连接起来，从而一个元素发生变化会以某种方式影响另一个变化。有时候需要动态地进行数据绑定，比如你要做一个银行间转账的程序。在用户界面上，你可以设计一个包含用户账号、姓名、存款余额等控件的界面，然后你可以用一个组合框来显示用户在各个银行里的开户情况。这个时候你需要动态地把用户账号、姓名、存款余额等控件绑定到不同的对象上。

WPF支持这种数据绑定，你可以用ClearBinding方法清除老的数据绑定，再重新建立新的绑定：

```
BindingOperations.ClearBinding
(txtScrollValue,TextBox.TextProperty);
```

11.5 最简单的数据绑定——从.NET对象到界面元素

现在来看另一种简单的数据绑定情形——从.NET对象到界面元素。首先来创建一个简单的类Book：

```
using System;
using System.Collections.Generic;
using System.Text;

namespace Yingbao.Chapter11.BookBanding
{
    public class Book
    {
        public Book(string title, string publisher, string isbn,
           params string[] authors)
        {
            this.title = title;
            this.publisher = publisher;
            this.isbn = isbn;
            foreach (string author in authors)
            {
                this.authors.Add(author);
            }
        }
        public Book()
            : this("unknown", "unknown", "unknown")
        {
        }
        private string title;
        public string Title
        {
            get { return title; }
            set { title = value; }
        }
        private string publisher;
        public string Publisher
        {
            get { return publisher; }
            set { publisher = value; }
        }
        private string isbn;
        public string Isbn
        {
            get { return isbn; }
            set { isbn = value; }
        }

        public override string ToString()
        {
            return title;
        }
        private readonly List<string> authors = new List<string>();
        public string[] Authors
```

```
        {
            get { return authors.ToArray(); }
        }
    }
}
```

在Book这个类中，笔者定义了书名、书号、作者和出版商这样的一些属性。由于Book类将作为数据源出现在绑定中，所以，它可以是一般的.Net对象。同样，这些属性也不是相关属性。

要在XAML中使用自定义类，需要引入该类所在的命名空间，即在XAML中加入Yingbao.Chapter11.BookBanding：

```
xmlns:src="clr-namespace:Yingbao.Chapter11.BookBanding
```

然后，把Book类中定义的属性绑定到UI元素上，如：

```
<TextBox Width="200" Height="30" Grid.Column="1" Grid.Row="0"
         Text="{Binding Path=Title, Mode=OneWay}" />
```

这次笔者在绑定时没有使用ElementName来指定源对象，而是直接使用Path。也许你会问，WPF怎么知道这个TextBox的Text要绑定到那个对象的Title呢？

```
<Window x:Class="Yingbao.Chapter11.BookWindow"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
      xmlns:src="clr-namespace:Yingbao.Chapter11.BookBanding"
    Title=".NET对象绑定到界面元素" Height="300" Width="340"
    >
  <Grid Name="bookGrid" Margin="5,5,5,5" >
    <Grid.ColumnDefinitions>
      <ColumnDefinition Width="30*" />
      <ColumnDefinition Width="70*" />
    </Grid.ColumnDefinitions>
    <Grid.RowDefinitions>
      <RowDefinition Height="50" />
      <RowDefinition Height="50" />
      <RowDefinition Height="50" />
      <RowDefinition Height="50" />
      <RowDefinition Height="50" />
    </Grid.RowDefinitions>
    <Label Grid.Column="0" Grid.Row="0">书名:</Label>
    <TextBox Width="200" Height="30" Grid.Column="1" Grid.Row="0"
          Text="{Binding Path=Title, Mode=OneWay}" />
    <Label Grid.Column="0" Grid.Row="1">出版社:</Label>
    <TextBox Width="200" Height="30" Grid.Column="1" Grid.Row="1"
          Text="{Binding Publisher}" />
    <Label Grid.Column="0" Grid.Row="2">书号:</Label>
    <TextBox Width="200" Height="30" Grid.Column="1" Grid.Row="2"
          Text="{Binding Isbn}" />
    <Button Margin="5,5,5,5" Grid.Column="1" Grid.Row="4"
           Click="OnOpenBookDialog"  Name="bookButton">显示图书信息</Button>
  </Grid>
</Window>
```

WPF能够绑定该属性的奥秘在于在C#的代码中，设定了bookGrid的DataCon-text属性：

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Data;
using System.Windows.Documents;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Media.Imaging;
using System.Windows.Shapes;


namespace Yingbao.Chapter11.BookBanding
{

    public partial class BookWindow : System.Windows.Window
    {
        private Book book1;

        public BookWindow()
        {
            InitializeComponent();
            book1 = new Book("红楼梦","中国文学出版社","88568823", new
string[]{"曹雪芹","高鹗"});
            bookGrid.DataContext = book1;
        }

        void OnOpenBookDialog(object obj, RoutedEventArgs rea)
        {
            StringBuilder  message= new StringBuilder();
            message.AppendLine("作者: ");
            for( int i=0; i<book1.Authors.Length;i++)
            {
                message.AppendLine(book1.Authors[i]);
            }
            message.Append("书号: ").AppendLine(book1.Isbn)  ;

            string caption = book1.Title ;
            MessageBox.Show(message.ToString(), caption);
        }
    }
}
```

在BookWindow类的C#代码中，笔者创建了Book类的实例，并把它赋予bookGrid的DataContext属性。这个例子的视觉树上，bookGrid是其他UI元素的父类。若没有设定UI元素的ElementName，WPF会沿着视觉树从树枝向树根寻找DataContext属性，直到找到该属性为止。所以，在这个例子中，若不设定bookGrid的DataContext，而设定BookWindow的DataContext属性，其效果是一样的：

this.DataContext = book1;

这是不是有点悬？这就是WPF中的相关属性继承的技术优势。11.6节，还要详细讨论DataContext的应用，现在看一下这个程序的运行结果（如图11-3所示）。

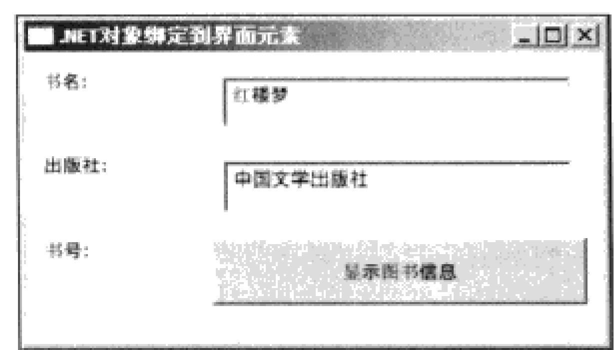


图11-3 绑定Book类的运行结果

11.6 DataContext

上面的例子，笔者使用了DataContext。DataContext是.NET 1.0就有的机制，WPF不仅保留了这一概念，而且由于相关对象（Dependency Object）和相关属性的引入，使得DataContext的功能更加强大。DataContext是FrameworkElement 类中的一个相关属性，如果有大量数据来源于同一种数据结构，如数据库的同一个表（database table），或同一个类（class）等情况下，那么使用DataContext就非常方便；如果源对象之间没啥关系，DataContext就不会有什么优势。下面是一个在XAML中设定DataContext属性的例子：

```
<Window
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="DataBinding" Height="300" Width="300" >
  <StackPanel>
<ScrollBar Name="scroll"  Orientation ="Horizontal " Margin ="24"
    Maximum ="100"  LargeChange ="10" SmallChange ="1"/>
<TextBox Name="scrollValue1" Height ="20" Width ="200"
    HorizontalAlignment ="Center"  DataContext="{Binding
   ElementName=scroll}" Text ="{Binding Path=Value, Mode=TwoWay}"/>
  </StackPanel >
</Window>
```

我们设定的只是scrollValue1这一个控件的DataContext，可以看出设定DataContext的语法:

DataContext="{Binding ElementName=scroll}"

即使用XAML的绑定扩展。

在上面这个例子中，我直接设定TextBox的DataContext属性。对于单个UI元素，使用DataContext不仅没有优势，而且还有点烦琐。原来我们只要用一个绑定扩展，现在却要用两个！如果有多个控件绑定到同一个数据结构或一个控件上，就可以看到使用DataContext的方便之处：

```
<Window x:Class="DataBinding.MultipleContextBinding"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="MultipleContextBinding"  DataContext="{Binding
    ElementName=scroll}" Height="373" Width="337">
  <StackPanel>
```

```
    <ScrollBar Name="scroll"  Orientation ="Horizontal "
          Margin ="24" Maximum ="100"
          LargeChange ="10" SmallChange ="1"/>
    <Label Height ="30">One way binding:</Label>
    <TextBox Name="scrollValue1" Height ="20" Width ="200"
       HorizontalAlignment ="Center"  Text ="{Binding Path=Value,
       Mode=OneWay}"/>
    <Label Height ="30">Two way binding:</Label>
    <TextBox Name="scrollValue2" Height ="20" Width ="200"
       HorizontalAlignment="Center"  Text ="{Binding Path=Value,
       Mode=TwoWay}"/>
    <Label Height ="30">One time binding:</Label>
    <TextBlock Name="scrollValue3" Height ="20" Width ="200"
      HorizontalAlignment ="Center"  Text~="{Binding Path=Value,
      Mode=OneTime}"/>
    <Label  Height ="30">One way to source:</Label>
    <TextBox Name="scrollValue4" Height ="20" Width ="200"
      HorizontalAlignment ="Center"  Text ="{Binding Path=Value,
      Mode=OneWayToSource}"/>
    <Label  Height ="30">Default:</Label>
    <TextBlock Name="scrollValue5" Height ="20" Width ="200"
      HorizontalAlignment ="Center"  Text ="{Binding Path=Value,
      Mode=Default}"/>
  </StackPanel>
</Window>
```

这个例子是对本章开始时例子的改写，笔者把Window的相关属性DataContext设为scroll1，这样，所有绑定到ScrollBar的控件（TextBlock）都可以在Binding的大括号中省掉ElementName=scroll。这里所用的机制是WPF中相关属性的传递：包容器类的相关属性被传递到其中的子类。由于在视觉树的任何一个父节点上设定DataContext，都可以传递到其中的子类，所以我们设定StackPanel的DataContext属性，其结果是一样的：

```
<StackPanel DataContext="{Binding ElementName=scroll}">
```

当然如果你不使用DataContext，则也可以在每个元素中加入ElementName。

11.7 控制绑定时刻

当用户在界面上输入数据的时候，你可能希望自己能够控制用户的输入传递到源对象的时间。比如，当用户单击CheckBox时，希望CheckBox的状态变化立即传递到所绑定的对象；当用户在TextBox中输入字符串时，往往不必要立即把用户的输入传递到所绑定的对象，用户可能对输入的字符串进行修改，这时我们希望等到输入焦点移到下一个控件上时，再把用户的输入传递到所绑定的源对象。换句话说，数据的传递发生在这个TextBox失去输入焦点的那一刻。

WPF数据绑定允许你自由地控制数据传递发生的时刻，控制绑定对象间数据传递发生的时刻是由UpdateSourceTrigger来完成的。表11-1列出了UpdateSourceTrigger的可能取值及其功能。

表11-1 UpdateSourceTrigger的值

UpdateSourceTrigger	功能
Default	对大多数相关属性而言,默认的绑定时刻是在属性改变时发生的(PropertyChanged)，而Text属性则是在失去输入焦点时发生的(LostFocus)
PropertyChanged	当绑定对象的属性改变时,立即改变所绑定源对象的值
LostFocus	当目标对象失去输入焦点时立即改变源对象的值
Explicit	仅在调用UpdateSource方法时才更新源对象的数值

可以在XAML中设置UpdateSourceTrigger，例如：

```
<TextBlock Name="scrollValue" Height ="20" Width ="200"
   HorizontalAlignment ="Center"  Text ="{Binding Path=Value,
   Mode=Default UpdateSourceTrigger=LostFocus}"/>
```

11.8 开发自己的IValueConverter

在第10章模板中，我们曾接触到移植IValueConverter接口的类，在那里，我们需要把时间转换成时针和分针在时钟上的旋转角度。与控件模板需要转换数据类型一样，在进行数据绑定时，有时候目标对象和源对象间的数据类型是不同的，需要对数据类型进行转化。WPF提供了一些默认的转换，例如，颜色值和画刷间的转换。但任何系统都没法包括现实中所有的类型转换，解决方法是让程序员根据需要提供自己的类型转换。

程序员为Binding类提供类型转换时需要做两件事：一是开发一个支持IValueConverter接口的类；二是把这个类的实例赋给Binding类中的Converter。WPF在传递数据时自动调用IValueConverter的Convert和ConvertBack方法：

```
public class  MyConverter: IValueConverter
{
    public object Convert( object value, Type typeTarget,
                     object param, CultureInfo culture)
          {
              ...
           }

      public object ConvertBack( object value Type  typeTarget,
                                 object param, CultureInfo culture)
           {
              ...
           }
 }
```

其中，参数value是要转换的值，typeTarget是要转换后的值的类型。如果无法转换该Value的值，则Convert和ConvertBack应返回null。参数param是为Binding类中的ConvertParameter准备的，CultureInfo是某些情况下，转换可能要涉及语言环境，比如说100.35，在德语和法语里应为“100，35”。

在C#里使用自定义Converter类，只要把自定义的Convert值连接到Bingding类中的Convert属性即可：

```
Binding myBinding = new Binding();
myBinding. Convert = new MyConverter();
```

在XAML中使用自定义类型转换，你需要：

1. 在Resources中加入自定义Converter类：

<srcLMyConverter x: key="conv" />

2. 在Binding中，加上Converter属性：

<.... "{Binding.Converter= {StaticResource conv}...}"

现在，让我们来看一个完整使用自定义类型转化的例子。AmountMoneyConvert类移植了IValueConverter接口，把用户输入的数字转换成两位小数的字符串，这个数字的物理意义是用户在银行户头上的存款余额，对于普通储户来说，存款余额只需要精确到两位小数（即几分钱“）。笔者在AmountMoneyConvert类前面加上了ValueConversion属性，ValueConversion属性的作用是告诉WPF AmountMoneyConvert类可以转换的源数据和目标数据的类型。在Convert方法中，首先把value转换成字符串，若该字符串为空，则返回0.0。这是为了支持TextBox中的值为空Null的情形。

```
namespace Yingbao.Chapter11.BindingWithConverter
{
    using System;
    using System.Globalization;
    using System.Windows;
    using System.Windows.Data;
    [ValueConversion(typeof(string), typeof(string))]
    public class AmountMoneyConvert : IValueConverter
    {
        public object Convert(object value, Type typeTarget,
                 object param, CultureInfo culture)
        {
            string inputValue = value.ToString();
            if (inputValue.Length == 0) return 0.00;
            Decimal num = System.Convert.ToDecimal(value);
            if (param != null)
            {
                num = Decimal.Round(num,
                          Int32.Parse(param as string));
            }
            return num;
        }

        public object ConvertBack(object value,
                  Type typeTarget, object param, CultureInfo culture)
        {
            return value;  // we don't need to do anything
        }
    }
}
```

这里用param来表示在进行字符串转换时所要保留的小数位数，下面这段XAML程序使用了AmountMoneyConvert类：

```
<Window x:Class="Yingbao.Chapter11.BindingWithConverter.MainWindow"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
```

```
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:src=
            "clr-namespace:Yingbao.Chapter11.BindingWithConverter"
    Title="自定义转换的数据绑定" Height="150" Width="300">
  <Window.Resources>
    <src:AmountMoneyConvert  x:Key="amtConv"/>
  </Window.Resources>
  <Grid>
    <Grid.ColumnDefinitions >
      <ColumnDefinition Width ="120"/>
      <ColumnDefinition Width ="2*"/>
    </Grid.ColumnDefinitions>
    <Grid.RowDefinitions>
      <RowDefinition  Height ="25" />
      <RowDefinition  Height ="25" />
    </Grid.RowDefinitions>
    <Label Content ="输入存款: " FontSize ="14" Grid.Row="0"
       Grid.Column ="0"/>
    <TextBox  Name="InputAmt"  HorizontalAlignment ="Center"
        FontSize ="10" Width ="130"
        Height ="18" Grid.Row ="0" Grid.Column ="1"/>
    <Label Content ="转换后的存款: " FontSize ="14" Grid.Row="1"
        Grid.Column ="0"/>
    <TextBox HorizontalAlignment ="Center" FontSize ="10"
        Width ="130" Height ="18" Grid.Row ="1" Grid.Column ="1"
        Text ="{Binding ElementName=InputAmt, Path=Text,
        Mode=OneWay, Converter={StaticResource
        amtConv },ConverterParameter=2}" />
  </Grid>
</Window>
```

笔者使用了两个字符输入框TextBox，第一个字符输入框是源对象，第二个字符输入框被绑定到第一个字符输入框上，绑定模式为OneWay，即第一个TextBox输入的值InputAmt会影响第二个TextBox。若第一个TextBox中输入的值为多个小数点的小数，则第二个TextBox中只显示带有两位小数点的小数值。其结果如图11-4所示：

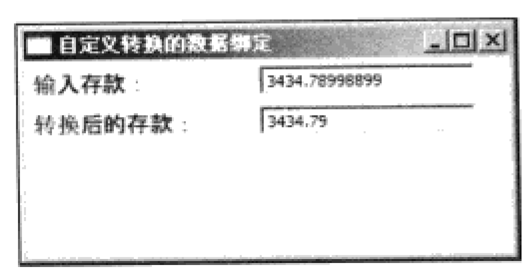


图11-4 使用自定义转换

正如我们所希望的，转换后的钱数精确到分，而且是四舍五入的。

11.9 在数据绑定中加入校验

几乎所有人机界面都要对用户输入进行校验，否则，一些无效的输入就会进入系统。当我们使用数据绑定后，数据就在目标对象和源对象间自动传递。我们希望在数据开始传递之前能对数据进行校验，若用户的输入为无效输入，就可以通知用户该操作是无效的。

例如，银行的客户存取款程序。在客户输入账号时，需要校验账号的格式是否正确，比如说，账号只能是整数，而且第一位不能是0；还要询问数据库，这个账号是否存在。在用户取款时，要保证用户不能透支等。过去，程序员根据自己的情况，开发出各种各样的校验程序，这些校验程序可以在不同的层次上出现。

WPF的数据绑定提供了一个标准的解决方案，其思路和类型转换一样，即在数据绑定中允许用户插入一定的业务规则。

11.9.1 开发业务规则类

下面通过一个例子来说明如何在数据绑定中插入业务规则。首先，从简单的银行账户类开始：

```
namespace Yingbao.Chapter11.BindingWithBusinessRules
{
    using System;
    using System.Collections.Generic;
    using System.Text;
    public class Account
    {
        string accountNumber;
        double totalAmount;
        public double TotalAmount
        {
            get { return this.totalAmount; }
            set { this.totalAmount = value; }
        }
        public double Deposit
        {
            set { this.totalAmount += value; }
        }

        public double WithDraw
        {
            set { this.totalAmount -= value; }
        }

        public string AccountNumber
        {
            get { return this.accountNumber; }
            set { this.accountNumber = value; }
        }

        public Account()
        {
        }

        public Account(double amount)
        {
            this.totalAmount = amount;
        }
    }
}
```

类Account包括用户的账号和账号里结余的钱款，定义了AccountNumber（账号）、TotalAmount（现有的存款余额）、Deposit（存款）和WithDraw（取款）这几个属性，Account用于存储用户的账户信息。我们要把这个类绑定到用户界面元素上，在用户输入账号时，要确保用户输入的账号是正确的账号。在实际应用程序中，还要用户输入密码等安全信息，这里为简单起见，我们只校验用户账户，其道理是一样的。为此，我们需要写一个校验账号的类AccountNumberRule：

```
namespace Yingbao.Chapter11.BindingWithBusinessRules
{
    using System;
    using System.Collections.Generic;
    using System.Text;
    using System.Text.RegularExpressions;
    using System.Windows.Controls;

    public class AccountNumberRule : ValidationRule
    {
        List<string> accountList = new List<string>();
        public AccountNumberRule()
        {
            accountList.Add("1234567890");
            accountList.Add("2234567890");
            accountList.Add("3234567890");
            accountList.Add("4234567890");
         }

     public override ValidationResult Validate(object value,
              System.Globalization.CultureInfo cultureInfo)
        {
            string accountNumber =
                value.ToString().TrimStart(' ').TrimEnd(' ')  ;
            bool existAccount = false;
            foreach( string account in this.accountList )
            {
                if( account.Equals(  accountNumber ) )
                {
                    existAccount= true;
                    break;
                }
            }
            if( !existAccount )
            {
                return new ValidationResult(false,"输入的账号错误！");
            }
            return ValidationResult.ValidResult;
        }
    }
}
```

为了把AccountNumberRule用在Binding类中，AccountNumberRule需要从ValidationRule类中派生出来，ValidationRule类位于System.Windows.Controls命名空间。在具体的业务规则类中，需要覆盖基类中的Validate方法。笔者在这个方法中，对用户输入的账号和预先设定好的4个账号进行比较（在实

际应用程序中，应该询问数据库用户输入的账户是否存在），看看用户输入的账号是否在这几个账号中。若用户输入的账号在这四个账号中，返回ValidationResult.ValidResult；否则返回ValidationResult(false, “输入账号错误！”)。ValidationResult类中有三个属性：ValidResult表示校验结果为有效；IsValid属性为布尔类型，当其值为false时，表示用户输入的值是无效的；ErrorContent属性表示具体的错误，通常是错误信息。笔者认为微软应该把ValidResult和IsValid这两个属性合为一个属性，并使用枚举类型，这个枚举类型应该包括“严重错误”、“警告错误”、“提示信息”和“结果有效”等值。而目前的移植，IsValid为true时，与ValidResult表示的是一个意思。

为了强调用户输入的账号为以1到9数字开头的数字，笔者写了NumericRule类。这个类同样是从ValidationRule类中派生出来，可用正则表达式（Regular Expression）对用户输入的账号进行校验。也可以把NumericRule和Account-Number合为一个类，这么做是为了演示XAML组合多个业务规则的能力。

```
namespace Yingbao.Chapter11.BindingWithBusinessRules
{
    using System;
    using System.Collections.Generic;
    using System.Text;
    using System.Text.RegularExpressions;
    using System.Windows.Controls;

    class NumericRule: ValidationRule
    {
        string numericEx = "^([1-9]([0-9]+))$";
        public override ValidationResult Validate(object value,
               System.Globalization.CultureInfo cultureInfo)
        {
             Regex rg = new Regex(numericEx);
             if( !rg.IsMatch( value.ToString() )  )
             {
                 return new ValidationResult(false,
                     "输入的字符中含有非数字键！");
             }
             return ValidationResult.ValidResult;
        }
    }
}
```

最后一个规则是用来对用户取款进行校验的。WithDrawRule用来强制两个规则：

- 用户必须输入有效的浮点数；
- 用户不能透支。

```
namespace Yingbao.Chapter11.BindingWithBusinessRules
{
    using System;
    using System.Collections.Generic;
    using System.Text;
    using System.Text.RegularExpressions;
    using System.Windows.Controls;
```

```
    public class WithDrawRule : ValidationRule
    {
        double accountBalance;
        public double AccountBalance
        {
            get { return this.accountBalance; }
            set { this.accountBalance = value; }
        }

        public WithDrawRule()
        {
        }

        public override ValidationResult Validate(object value,
             System.Globalization.CultureInfo
                                 cultureInfo)
        {
            double amount = 0;
            if (!Double.TryParse(value.ToString(), out amount))
            {
                return new ValidationResult(false, "输入无效字符！");
            }
            if (amount > this.accountBalance)
            {
                return new ValidationResult(false, "你无权透支！");
            }
            return ValidationResult.ValidResult;
        }
    }
}
```

11.9.2 在绑定中添加任意多个业务规则

在XAML中把上面的规则和用户界面上的输入控件联系起来，要用到Binding类中的ValidationRules属性，其语法是：

```
<Binding.ValidationRules>
     <src:NumericRule/>
     <src:AccountNumberRule/>
</Binding.ValidationRules>
```

在Binding.ValidationRules标识中，可以根据需要添加任意多个业务规则。

11.9.3 在控件上显示校验信息

最后要做的是在界面控件上显示校验结果，笔者希望若用户输入有错，我们的程序应该在正确的时间、正确的地方通知用户出错的信息。显然，最好的地方是用户当前输入的控件上。为此，需要在C#代码中加入事项处理程序：

```
Validation.AddErrorHandler(txtAccount, OnAccountError);
Validation.AddErrorHandler(txtWithdraw, OnWithDrawError);
```

我们要用Validation类中的AddErrorHandler这个静态方法。

在控件上显示出错信息可以有两种方法，一种是使用模板，当错误出现的时候，用专用的错误模板显示错误信息，这时WPF允许你显示任何形式的信息。第二中方式是设置控件的ToolTip属性，当用户在控件上拖过鼠标时显示相应的错误信息，这种简单的机制，可以包括大多数的情况。笔者这里采用的是第二个方法：

```
void OnAccountError(object sender, ValidationErrorEventArgs vea)
        {
            txtAccount.ToolTip = vea.Error.ErrorContent.ToString();
        }

        void OnWithDrawError(object sender,
                   ValidationErrorEventArgs vea)
        {
            txtWithdraw.ToolTip = vea.Error.ErrorContent.ToString();
        }
```

11.9.4 触发错误处理事件

要在控件上显示错误信息，别忘了最后一步：在XAML中设置Binding类中的NotifyOnValidationError属性，否则，即使校验有错，WPF也不会调用C#中的错误处理程序：

```
            NotifyOnValidationError="true"
```

下面是完整的主窗口XAML程序：

```
<Window
x:Class="Yingbao.Chapter11.BindingWithBusinessRules.MainWindow"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:src=
       "clr-namespace:Yingbao.Chapter11.BindingWithBusinessRules"
    Title="带校验的数据绑定" Height="150" Width="300">
  <Window.Resources>
    <src:Account  x:Key="MyAccount" TotalAmount="10000.00"/>
  </Window.Resources>
  <Grid DataContext="{StaticResource MyAccount}" >
  <Grid.ColumnDefinitions >
    <ColumnDefinition Width ="120"/>
    <ColumnDefinition Width ="2*"/>
  </Grid.ColumnDefinitions>
  <Grid.RowDefinitions>
    <RowDefinition  Height ="25" />
    <RowDefinition  Height ="25" />
    <RowDefinition  Height ="25" />
  </Grid.RowDefinitions>
  <Label Content ="用户账号：" FontSize ="14" Grid.Row="0"
      Grid.Column ="0"/>
     <TextBox  Name="txtAccount"  HorizontalAlignment ="Center"
       FontSize ="10" Width ="130"
```

```
      Height ="18" Grid.Row ="0" Grid.Column ="1">
    <TextBox.Text>
      <Binding Path="AccountNumber"
        NotifyOnValidationError="true">
        <Binding.ValidationRules>
          <src:NumericRule/>
          <src:AccountNumberRule/>
        </Binding.ValidationRules>
      </Binding>
    </TextBox.Text>
  </TextBox>
  <Label Content ="请输入取款金额: " FontSize ="14" Grid.Row="1"
       Grid.Column ="0"/>
  <TextBox Name="txtWithdraw" HorizontalAlignment ="Center"
       FontSize ="10" Width ="130" Height ="18"  Grid.Row ="1"
       Grid.Column ="1" >
    <TextBox.Text>
      <Binding Path="WithDraw" NotifyOnValidationError="true"
       Mode="OneWayToSource">
        <Binding.ValidationRules>
          <src:WithDrawRule AccountBalance="1000.00"/>
        </Binding.ValidationRules>
      </Binding>
    </TextBox.Text>
  </TextBox>
  <Button Name="btnOK"  Width ="50" Margin ="2" Click="OnOk"
      Grid.Row="2" Grid.Column="1">确认</Button>
</Grid>
</Window>
```

相应的C#处理程序如下：

```
namespace Yingbao.Chapter11.BindingWithBusinessRules
{
    using System;
    using System.Collections.Generic;
    using System.Text;
    using System.Windows;
    using System.Windows.Controls;
    using System.Windows.Input;
    using System.Windows.Media;
    using System.Windows.Media.Imaging;
    using System.Windows.Shapes;

    public partial class MainWindow: System.Windows.Window
    {

        public MainWindow()
        {
            InitializeComponent();
            Validation.AddErrorHandler(txtAccount, OnAccountError);
            Validation.AddErrorHandler(txtWithdraw,
                        OnWithDrawError);
        }
```

```
        void OnAccountError(object sender,
             ValidationErrorEventArgs vea)
        {
            txtAccount.ToolTip = vea.Error.ErrorContent.ToString();
        }

        void OnWithDrawError(object sender,
             ValidationErrorEventArgs vea)
        {
            txtWithdraw.ToolTip = vea.Error.ErrorContent.ToString();
        }

        void OnOk(object sender, EventArgs ea)
        {
            this.Close();
        }
    }
}
```

上面的程序在几种错误情况下的运行结果如图11-5所示。由图11-5可以看到，当出现错误时，WPF会自动在控件上加上红色的边框，同时Tooltip会显示出错信息。

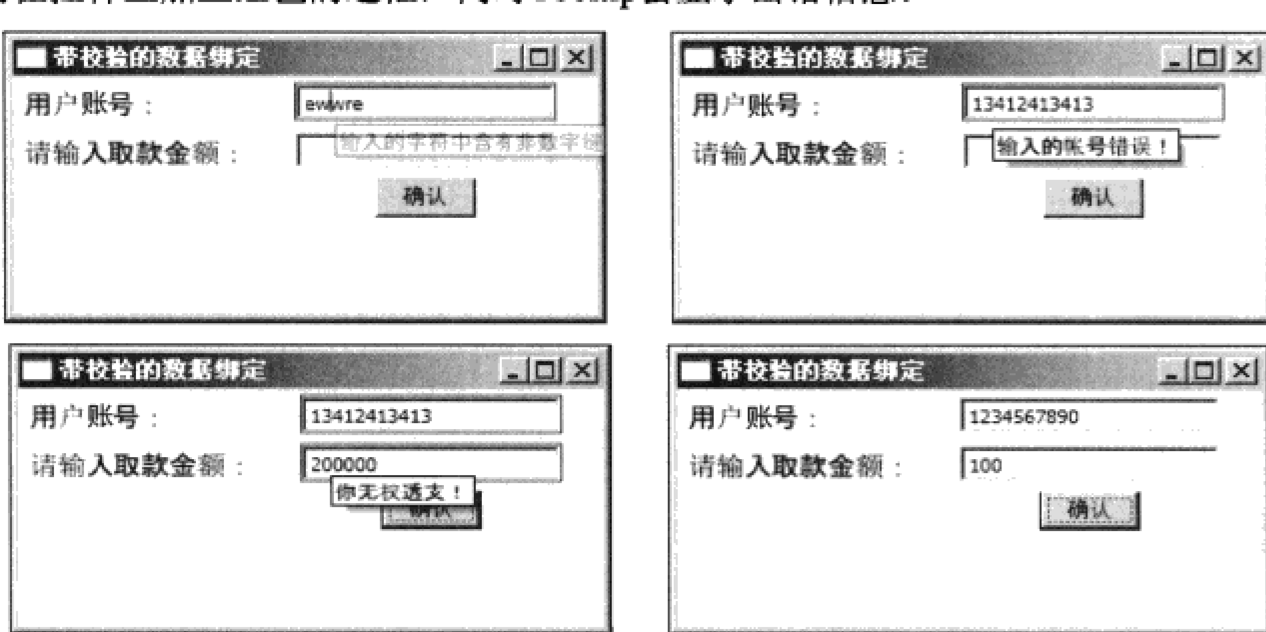


图11-5 具有校验功能的数据绑定结果

11.9.5 清除控件上的错误信息

前面的程序有一个问题，就是在用户改正了输入错误后，原来的错误信息还在相应的控件的提示条上。这是因为，当校验正确时我们并没有清除控件上的控件信息。遗憾的是，ValidationRule里面并没有通知校验成功的事件（笔者认为应该提供），为了清除控件上的错误信息，需要用到Validation.Errors这个附加属性：

```
<TextBox.ToolTip>
    <Binding ElementName="txtAccount"
          Path="(Validation.Errors)[0].ErrorContent"/>
 </TextBox.ToolTip>
```

在上面的XAML语句中，把ToolTip属性绑定到字符输入框自己的（Validation.Errors）

[0].ErrorContent 附加属性上。也可以不用 ElementName，而使用 XAML 的相对资源扩展(RelativeSource)，其语法是：

```
<TextBox.ToolTip>
    <Binding RelativeSource="{RelativeSource Self}"
        Path="(Validation.Errors)[0].ErrorContent"/>
</TextBox.ToolTip>
```

注意：在对ToolTip进行上述绑定后，我们就不需要设置Binding类的NotifyOnValidationError了，也不必在C#中加入事项处理程序。如果不去掉这些代码，程序仍然可以正确运行。

11.10 对集合对象的绑定

到此为止，笔者介绍了WPF中数据绑定的一些基本机制，本节将讨论对数据集合的绑定。在实际应用中，我们常常面对的是集合对象。比如，11.9节对银行账户的讨论，应用程序要管理的常常是成千上万个账户。这些账户的特点是每个账户的数据结构是相同的，在关系数据库中，这些数据可能位于同一个表内。

应用程序在管理客户的地址时，情况也是类似的。每个客户都有至少一个地址（笔者在State Street银行工作时，我们的软件支持每个客户的地址多达16个，客户可以对不同的地址设置不同的用途），这些地址的数据结构也是相同的。由此可见，具有相同数据结构的数据表非常广泛。

下面我以地址为例来讨论聚合对象的绑定。为此，笔者写了一个简单的类AddressInfo：

```
namespace Yingbao.Chapter11.AddressListBinding
{
    using System;
    using System.Collections.Generic;
    using System.Text;
    using System.Collections.ObjectModel;
    public class AddressInfo
    {
        private string streetNo;
        private string streetName;
        private string city;
        private string province;
        private string country;
        private string postCode;

        public string StreetNo
        {
            get { return this.streetNo; }
            set { this.streetNo = value; }
        }

        public string StreetName
        {
            get { return this.streetName; }
            set { this.streetName = value; }
        }
```

```
        public string City
        {
            get { return this.city; }
            set { this.city = value; }
        }

        public string Province
        {
            get { return this.province; }
            set { this.province = value; }
        }

        public string Country
        {
            get { return this.country; }
            set { this.country = value; }
        }

        public string PostCode
        {
            get { return this.postCode; }
            set { this.postCode = value; }
        }

        public override string ToString()
        {
           StringBuilder sb = new StringBuilder();
           sb.Append(Country)
             .Append("   ").Append(Province).Append("  ")
             .Append(City).Append("  ").Append(StreetName)
             .Append("  ").Append(streetNo);
           return sb.ToString();
        }

        public AddressInfo()
        {
        }
    }
```

在第10章模板中笔者曾以这个类为例示范数据模板，这里笔者要示范创建AddressList的不同方法。这个类描述用户的地址信息，包括国家、省、市、街道和邮政编码等信息。在C#中，要管理这个地址数据的集合，最简单的方法是使用List（在第10章，我们用ObservableCollection)，例如可以用AddressList来管理多个地址：

```
public class AddressList :List<AddressInfo>
 {
     public AddressList()
     {
     }
 }
```

现在我们把AddressList放到XAML的资源中（第10章我们在C#中创建AddressList数据），下面的

XAML创建了三个地址，并放在AddressList中：

```
<Window.Resources>
    <src:AddressList  x:Key="addressList">
    <src:AddressInfo  Country="中国" City="北京"
        StreetName="学院路" StreetNo="178"/>
    <src:AddressInfo  Country="中国" Province="安徽"
        City="合肥" StreetName="长江路" StreetNo="355"/>
    <src:AddressInfo  Country="中国" Province="安徽" City="芜湖"
        StreetName="九华山路"   StreetNo="23"/>
    </src:AddressList>
  </Window.Resources>
```

我们可以使用ListBox来显示AddressList中的信息，可以直接把ListBox中的ItemsSource绑定到AddressInfo上，把ListBox的DataContext设为AddressList：

```
<ListBox Name="lbAddress" Grid.Column ="0" Grid.Row ="0"
   Grid.ColumnSpan ="2" ItemsSource="{Binding}"
   DataContext ="{StaticResource addressList}" />
```

这样，ListBox就会显示AddressInfo的信息。在未指定显示AddressInfo的特定属性的情况下，ListBox会调用AddressInfo的ToString()方法来作为每个条目的默认显示。下面是完整的XAML：

```
<Window x:Class="Yingbao.Chapter11.AddressListBinding.MainWindow"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:src="clr-namespace:Yingbao.Chapter11.AddressListBinding"
    Title="绑定地址列表" Height="350" Width="400">
    <Window.Resources>
      <src:AddressList  x:Key="addressList">
        <src:AddressInfo  Country="中国" City="北京"
            StreetName="学院路" StreetNo="178"/>
        <src:AddressInfo  Country="中国" Province="安徽" City="合肥"
            StreetName="长江路" StreetNo="355"/>
         <src:AddressInfo  Country="中国" Province="安徽" City="芜湖"
            StreetName="九华山路" StreetNo="23"/>
      </src:AddressList>
    </Window.Resources>
    <Grid  >
      <Grid.ColumnDefinitions >
        <ColumnDefinition Width ="120"/>
        <ColumnDefinition Width ="2*"/>
      </Grid.ColumnDefinitions>
      <Grid.RowDefinitions>
        <RowDefinition  Height ="100" />
        <RowDefinition  Height ="25" />
        <RowDefinition  Height ="25" />
        <RowDefinition  Height ="25" />
        <RowDefinition  Height ="25" />
        <RowDefinition  Height ="25" />
        <RowDefinition  Height ="25" />
```

```
            <RowDefinition  Height ="25" />
        </Grid.RowDefinitions>
        <ListBox Name="lbAddress" Grid.Column ="0" Grid.Row ="0"
            Grid.ColumnSpan ="2" ItemsSource="{Binding}"
            DataContext ="{StaticResource addressList}" />
</Grid>
</Window>
```

由于笔者在AddressInfo中，移植了ToString() 方法，其返回的字符中包括国家、省、市、街道和街道号等信息，所以上面的程序的运行结果为：

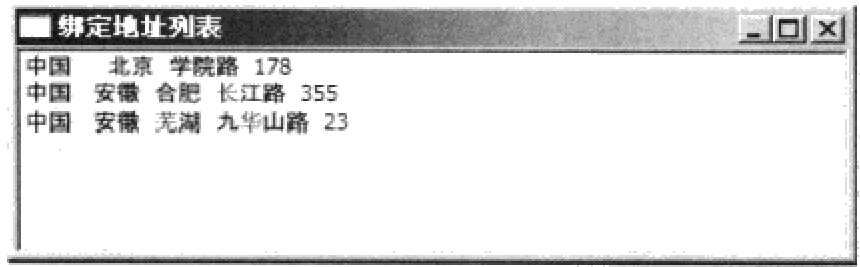


图11-6 ListBox的绑定

这个程序比第10章所用的数据模板技术要简单，其运行结果如图11-6所示。

11.10.1 使用DisplayMemberPath属性

ListBox的数据绑定允许你在每个条目中显示任意属性。例如，只关心客户所在的城市，那么可以在ListBox中只显示城市名。方法是设置DisplayMemberPath属性：

```
<ListBox Name="lbAddress" Grid.Column ="0" Grid.Row ="0"
   Grid.ColumnSpan ="2"  ItemsSource="{Binding}"
   DisplayMemberPath ="City"  DataContext ="{StaticResource
              addressList}" />
```

修改后的结果如图11-7所示。使用DisplayMemberPath，可以选择性地显示AddressInfo中的任何属性。

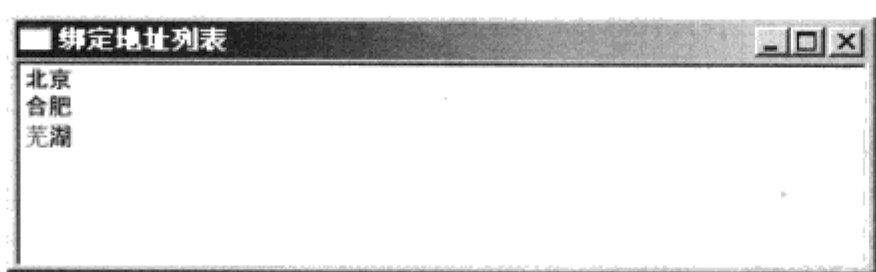


图11-7 在ListBox中显示城市

11.10.2 显示当前条目

应用程序需要为客户提供修改地址的功能，为此，需要把当前条目下的AddressInfo对象中的属性显示在字符输入框中，当用户在ListBox中选择某个条目时，我们希望字符输入框中显示当前条目的信息。由于这时ListBox和TextBox控件都要显示AddressInfo中的内容，故需要把DataContext属性设置到ListBox和TextBox所在的视觉树的父节点中，即Grid或Window。笔者把它放到Grid中：

```
<Grid DataContext ="{StaticResource addressList}" >
```

我们需要设置ListBox的IsSynchronizedWithCurrentItem属性，以保证在用户选择ListBox中不同的条目时，字符输入框显示的是同一个条目的值:

```
IsSynchronizedWithCurrentItem ="True"
```

修改后的完整XAML程序如下：

```
<Window x:Class="Yingbao.Chapter11.AddressListBinding.MainWindow"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:src="clr-namespace:Yingbao.Chapter11.AddressListBinding"
    Title="绑定地址列表" Height="350" Width="400">
    <Window.Resources>
      <src:AddressList  x:Key="addressList">
         <src:AddressInfo  Country="中国" City="北京"
            StreetName="学院路" StreetNo="178"/>
         <src:AddressInfo  Country="中国" Province="安徽"
            City="合肥" StreetName="长江路" StreetNo="355"/>
         <src:AddressInfo  Country="中国" Province="安徽"
            City="芜湖" StreetName="九华山路" StreetNo="23"/>
      </src:AddressList>
    </Window.Resources>
    <Grid DataContext ="{StaticResource addressList}" >
      <Grid.ColumnDefinitions >
        <ColumnDefinition Width ="120"/>
        <ColumnDefinition Width ="2*"/>
      </Grid.ColumnDefinitions>
      <Grid.RowDefinitions>
        <RowDefinition  Height ="100" />
        <RowDefinition  Height ="25" />
        <RowDefinition  Height ="25" />
        <RowDefinition  Height ="25" />
        <RowDefinition  Height ="25" />
        <RowDefinition  Height ="25" />
        <RowDefinition  Height ="25" />
        <RowDefinition  Height ="25" />
      </Grid.RowDefinitions>
      <ListBox Name="lbAddress" Grid.Column ="0" Grid.Row ="0"
         Grid.ColumnSpan ="2" ItemsSource="{Binding}"  D
         DisplayMemberPath ="City"
         IsSynchronizedWithCurrentItem ="True" />
     <Label Content ="国家: " FontSize ="14" Grid.Row="2"
         Grid.Column ="0"/>
      <TextBox  Name="txtCountry"  HorizontalAlignment ="Center"
         FontSize ="10" Width ="130" Height ="18" Grid.Row ="2"
         Grid.Column ="1" Text="{Binding Path=Country}"/>
      <Label Content ="省份: " FontSize ="14" Grid.Row="3"
         Grid.Column ="0"/>
      <TextBox  Name="txtProvince"  HorizontalAlignment ="Center"
         FontSize ="10" Width ="130" Height ="18" Grid.Row ="3"
         Grid.Column ="1" Text="{Binding Path=Province}"/>
      <Label Content ="城市: " FontSize ="14" Grid.Row="4"
```

```
        Grid.Column ="0"/>
    <TextBox  Name="txtCity"  HorizontalAlignment ="Center"
        FontSize ="10" Width ="130"
        Height ="18" Grid.Row ="4" Grid.Column ="1"
        Text="{Binding Path=City}"/>
    <Label Content ="街道: " FontSize ="14" Grid.Row="5"
        Grid.Column ="0"/>
    <TextBox  Name="txtStreet"  HorizontalAlignment ="Center"
        FontSize ="10" Width ="130" Height ="18" Grid.Row ="5"
        Grid.Column ="1" Text="{Binding Path=StreetName}"/>
    <Label Content ="街道号: " FontSize ="14" Grid.Row="6"
          Grid.Column ="0"/>
    <TextBox  Name="txtStreetNo"  HorizontalAlignment ="Center"
          FontSize ="10" Width ="130" Height ="18" Grid.Row ="6"
          Grid.Column ="1" Text="{Binding Path=StreetNo}"/>
</Grid>
</Window>
```

这段程序的运行结果如图11-8所示。若你用鼠标选择ListBox中的不同条目时，便可以看到ListBox下面的字符输入框中的内容也在跟着变化：

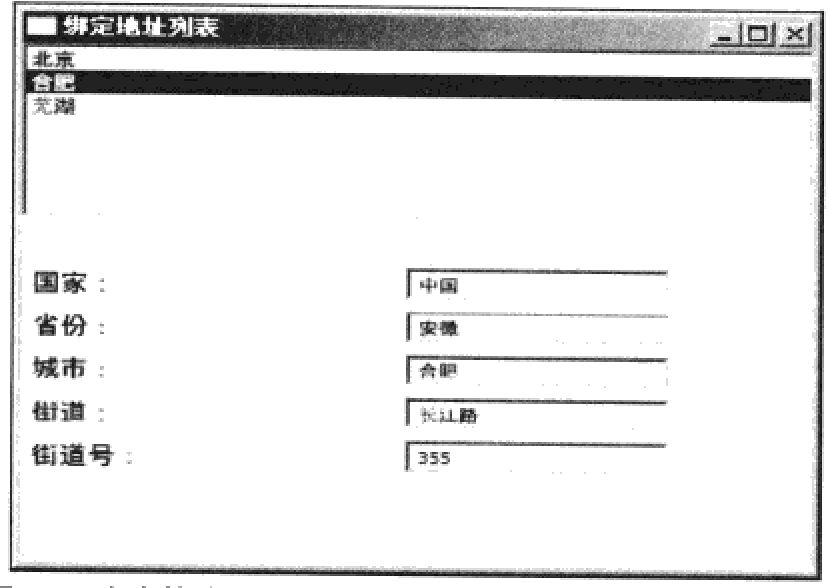


图11-8 在字符输入框中显示当前条目下的AddressInfo属性值

11.10.3 遍历集合中的记录

在图11-8中，若用鼠标选择ListBox中的条目，ListBox下面的字符输入框中的字符也跟着变化，这实际上在遍历集合AddressList中的不同的记录，这是数据绑定自动完成的。若在界面上没有ListBox时，如何遍历AddressList中的不同记录呢？我们需要对数据集合AddressList创建一个视图View，通过View来遍历数据集合中的数据。

```
ICollectionView view = CollectionViewSource.GetDefaultView(addressList);
```

CollectionViewSource提供了在数据集合上面创建视图的方法。CollectionViewSource 类从CollectionView中派生出来，专门用作XAML中的数据绑定。CollectionViewSource类可以看做是在数据源上又建立了一层接口，这层接口的功能是把数据源和表示层隔开。通过这一层，可以实现在UI上对数据的分组、排序和过滤等功能，同时并没有改变数据源。CollectionViewSource类中含有Source

属性，这个属性为指向数据源的引用；CollectionViewSource类中的View属性则指向数据在UI上的表现（若大家熟悉MVC设计范例或数据库里的Data/View模型，就很容易理解CollectionViewSource的功能）。若数据源移植了INotifyCollectionChanged接口，那么，在数据源发生改变时，会产生CollectionChanged事件，这个事件会自动传递到数据的视图（View）上。由于数据的视图和数据源间相互独立，一个数据源可以有多个视图。

当ListBox中IsSynchronizedWithCurrentItem设为True时，WPF自动把ListBox中的SelectedItem和数据源的默认视图中的CurrentItem属性联系起来，所以我们只要操作数据源的默认视图中的CurrentItem就可以实现对ListBox 的操作，而不必写自己的代码。

让我们在前面XAML的Grid中加入“上一记录”和“下一记录”两个按键：

```
<Button Name="btnPreviousRecord"  Width ="50" Margin ="2" Click="OnPreviouse"
    Grid.Row="7" Grid.Column="0">上一记录</Button>
<Button Name="btnNextRecord"  Width ="50" Margin ="2" Click="OnNext"
    Grid.Row="7" Grid.Column="1">下一记录</Button>
```

当用户单击“上一记录”和“下一记录”按键时，我们来移动数据源的默认视图中的CurrentItem属性：

```
void OnPreviouse(object sender, EventArgs ea)
{
        AddressList addressList =
            this.FindResource("addressList") as AddressList;
        ICollectionView view =
            CollectionViewSource.GetDefaultView(addressList);
        view.MoveCurrentToPrevious();
        if (view.IsCurrentBeforeFirst) {
            view.MoveCurrentToFirst();
        }
}

void OnNext(object sender, EventArgs ea)
{
        AddressList addressList =
              this.FindResource("addressList") as AddressList;
        ICollectionView view =
              CollectionViewSource.GetDefaultView(addressList);
        view.MoveCurrentToNext() ;
        if (view.IsCurrentAfterLast)
        {
             view.MoveCurrentToLast() ;
        }
}
```

用按键遍历数据源中的记录如图11-9所示。

图11-9 使用按键遍历数据源记录

11.10.4 增加或删除记录

数据表的另外两个常用操作是增加或删除记录。通常，我们在ListBox的下面加上“添加”和“删除”两个按键：

```
<Button Name="btnAdd"  Width ="50" Margin ="2" Click="OnAdd"
        Grid.Row="1"  Grid.Column="0">添加</Button>
<Button Name="btnDelete"  Width ="50" Margin ="2" Click="OnDelete"
        Grid.Row="1" Grid.Column="1">删除</Button>
```

其在界面上的排版如图11-10所示。我们需要在MainWindow 类中增加两个方法：OnAdd和OnDelete：

```
void OnAdd(object sender, EventArgs ea)
        {
            AddressList addressList =
                  this.FindResource("addressList") as AddressList;
            addressList.Add(new AddressInfo());
            ICollectionView view =
                  CollectionViewSource.GetDefaultView(addressList);
            view.MoveCurrentToLast();
        }

        void OnDelete(object sender, EventArgs ea)
        {
            AddressList addressList =
              this.FindResource("addressList") as AddressList;
            ICollectionView view =
               CollectionViewSource.GetDefaultView(addressList);
            addressList.Remove(view.CurrentItem as AddressInfo );
            view.MoveCurrentToNext();
```

```
            if (view.IsCurrentAfterLast)
            {
                view.MoveCurrentToLast();
            }
        }
```

图11-10 在数据表中增加、删除记录

在OnAdd的方法中，笔者在数据源AddressList中加入一个AddressInfo对象，然后通过修改CurrentItem来在界面上显示新加入的记录：

```
ICollectionView view =
            CollectionViewSource.GetDefaultView(addressList);
view.MoveCurrentToLast();
```

在OnDelete方法中，在数据源AddressList中删除当前选中的AddressInfo对象，然后把CurrentItem指向下一条地址：

```
view.MoveCurrentToNext();
  if (view.IsCurrentAfterLast)
  {
      view.MoveCurrentToLast();
  }
```

然而，在运行这个程序时，我们发现界面上并没有增加或删除一个记录，为什么呢？这是因为数据绑定的是数据源的视图（View），而我们增加或删除的是数据源。换句话说，虽然视图是在数据源上创建的，一旦视图创建之后，视图和数据源之间就是独立的。若要在数据源改变时视图能发生相应的变化，就需要在数据源发生改变时能够“通知”相应视图的机制。WPF中的INotifyCollectionChanged接口就是这样的机制。所以，解决的方法是为AddressList移植INotifyCollectionChanged接口。

```
public class AddressList :
       List<AddressInfo>,INotifyCollectionChanged
    {
        public AddressList()
        {

        }

        public event NotifyCollectionChangedEventHandler
             CollectionChanged;

        protected virtual void
          OnCollectionChanged(NotifyCollectionChangedEventArgs e)
        {
            if (CollectionChanged != null)
            {
                CollectionChanged(this, e);
            }
        }

        public new void Add(AddressInfo o)
        {
            base.Add(o);
            OnCollectionChanged(new
                  NotifyCollectionChangedEventArgs(
                  NotifyCollectionChangedAction.Add, o));
        }

        public new void Remove(AddressInfo o)
        {
            base.Remove(o);
            OnCollectionChanged(new
                 NotifyCollectionChangedEventArgs(
                 NotifyCollectionChangedAction.Remove, o));
        }

        public void Move(AddressInfo o, Int32 newIndex)
        {
           Int32 oldIndex = 0;
           OnCollectionChanged(new
                NotifyCollectionChangedEventArgs(
                NotifyCollectionChangedAction.Move,
                o, newIndex, oldIndex));
        }

        public new AddressInfo  this[Int32 index]
        {
            get
            {
                return null;
            }
            set
            {
                AddressInfo oldValue = null;
                OnCollectionChanged(new
```

```
                    NotifyCollectionChangedEventArgs(
                    NotifyCollectionChangedAction.Replace,
                    value, oldValue));
            }
        }
    }
```

上面的程序是改写的AddressList对象支持INotifyCollectionChanged接口。当在AddressList中添加或删除AddressInfo对象时，AddressList会发出CollectionChanged，基于AddressList的视图会自动订阅该事件，从而绑定到默认视图的列表框就会反应数据源的变化。

移植INotifyCollectionChanged接口有点复杂，为了简化应用程序的工作，WPF提供了ObservableCollection类，这个类移植了INotifyCollectionChanged和INotifyPropertyChanged两个接口。若我们使用ObservableCollection类，则AddressList类变得非常简单：

```
public class AddressList :  ObservableCollection<AddressInfo>
    {
        public AddressList()
        {

        }
    }
```

即AddressList作为ObservableCollection的派生类，第10章模板中的AddressList就是从ObservableCollection中派生出来的，当时，笔者并没有解释为什么使用ObservableCollection的问题。

11.10.5 对集合对象分组

前面提到在数据源上创建视图（View），这些视图在创建之后和数据源间是一种非常松散的关系。对同一个数据源，我们可以创建多个视图，这些视图之间互相独立。我们可以根据需要对同一个数据源创建多种视图，从而得到不同的显示效果。

对集合对象进行分组显示是一个很有用的功能，比如说，前面的AddressList例子。在你的数据库中可能有成千上万个地址，若能按地址所在的城市进行分组显示，在界面上就可以直观地看出你的客户在城市间的分布。

在视图里对数据源分组非常简单，只要设置分组的属性即可。比如，要把地址信息按城市分组，可以在MainWindow的构造函数中在视图的GroupDescripton中加入AddressInfo的City属性即可：

```
public MainWindow()
 {
    InitializeComponent();
    AddressList addressList = this.FindResource("addressList")
          as AddressList;
    ICollectionView view =
          CollectionViewSource.GetDefaultView(addressList);
           view.GroupDescriptions.Clear();
           view.GroupDescriptions.Add(new
                 PropertyGroupDescription("City"));
       }
```

为了验证分组效果，笔者在AddressList 中加入5个地址：

```
<src:AddressList  x:Key="addressList">
     <src:AddressInfo  Country="中国" City="北京"
          StreetName="学院路" StreetNo="178"/>
     <src:AddressInfo  Country="中国" Province="安徽"
          City="合肥" StreetName="长江路" StreetNo="355"/>
     <src:AddressInfo  Country="中国" Province="安徽" City="芜湖"
          StreetName="九华山路" StreetNo="23"/>
     <src:AddressInfo  Country="中国" Province="安徽"
          City="合肥" StreetName="长江路" StreetNo="21"/>
     <src:AddressInfo  Country="中国"  City="北京"
          StreetName="望京南路" StreetNo="10"/>
</src:AddressList>
```

上面程序的运行结果如图11-11所示，由图11-11可见，列表框中，位于北京市的地址列在一块。

图11-11 集合数据按城市分组

有意思的是View中的GroupDescriptions中可以加入多个PropertyGroupDescription，这说明可以用多个属性来分组，比如，可以按照城市和街道来分组等。

11.10.6 对集合对象排序

视图还具有对集合对象排序的功能，与分组功能类似，只要在视图的SortDescriptions中加入排序的方式即可。比如说，可以对地址按照省份进行排序：

```
view.SortDescriptions.Add(new SortDescription("Province",
ListSortDirection.Descending ));
```

排序后的结果如图11-12所示。

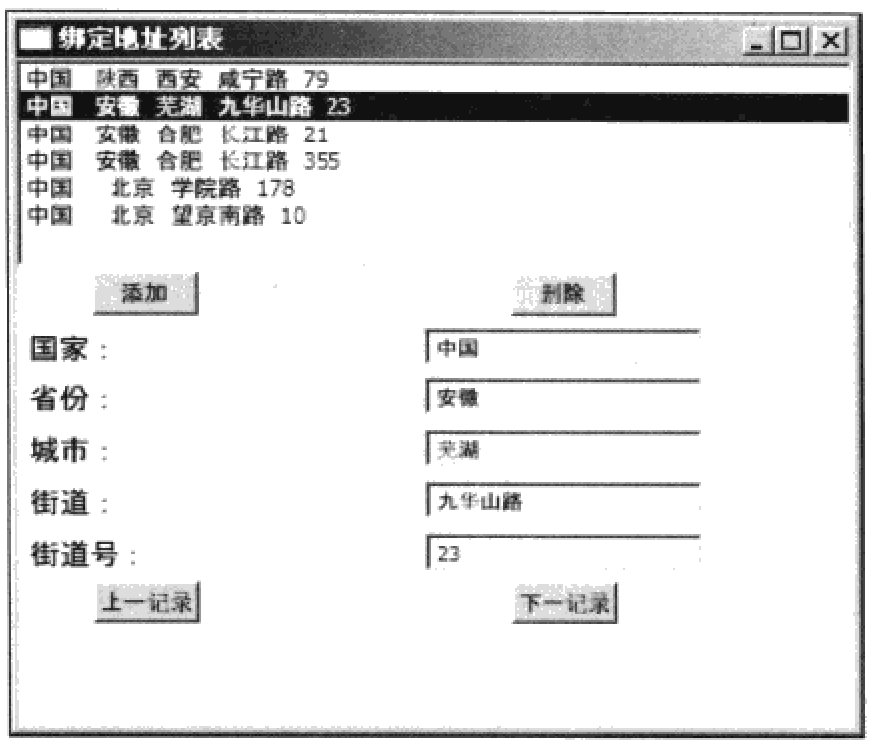


图11-12 集合数据按城市分组，按省降序排列

从图11-12中可以看出，WPF对省份的排序是按照汉语拼音的顺序进行的（注意在北京市的地址中，因没有省份的信息而排在最后）。如果你对这种默认的排序方式不满意，可以开发自己的排序方式。通过设置ListCollectionView中的CustomSort属性来控制排序方式，CustomSort的类型是IComparer，需要移植Compare方法，提供比较两个对象的结果。

11.10.7 对集合对象过滤

对集合对象的过滤要定义一个回调函数，在回调函数中提供过滤逻辑，然后把回调函数和视图（view）的Filter属性联系起来：

```
view.Filter = new Predicate<object>(Contains);
```

这里Contains为回调函数，例如，我们要在界面上只显示芜湖市，可以在Contains中加入下面的逻辑：

```
public bool Contains(object obj)
 {
     AddressInfo addrInfo = obj as AddressInfo;
     if( addrInfo.City.Equals("芜湖") )
     {
         return true;
     }
     return false;
 }
```

在加入上面的逻辑后，界面上的显示结果如图11-13所示：

图11-13 在视图中加入过滤逻辑

可以根据情况动态地设置视图中的Filter，从而在界面上显示我们需要的结果。比如，可以在界面上增加一个组合框，其中含有各大城市的列表，根据用户选择的城市在界面上只显示该城市的用户地址。

11.11 数据源

为了更好地支持数据绑定，WPF对XML数据源和.NET对象数据源进行包装，本节我将讨论这两个包装类：XmlDataProvider和ObjectDataProvider，这两个类都是从DataSourceProvider类中派生出来的，DataSourceProvider是一个抽象类。

11.11.1 XML数据源

XML数据在应用程序中使用得非常普遍，从用户界面到事务处理，再到通信接口、数据库管理，都大量使用XML。如摩根思坦利（Morgan Stanley）制定的银行间的XML通信规范，IEC推出的电力系统61850通信规约等都是以XML技术为基础的。WPF使用XmlDataProvider把XML数据直接绑定到控件上。

在XAML中使用XML数据有两种方式：一种是在XAML中嵌入XML数据，另一种是XML作为文件独立存在。

- XML岛屿

把在XAML中嵌入XML数据叫做XML岛屿（Xml Island），在下面的XAML中笔者把AddressList作为XML数据嵌入到Window的资源中：

```
<Window.Resources>
    <XmlDataProvider x:Key ="xmlAddress"
           XPath ="AddressList/AddressInfo">
      <x:XData >
        <AddressList xmlns="">
          <AddressInfo  Country="中国" City="北京"
```

```
                StreetName="学院路" StreetNo="178"/>
         <AddressInfo  Country="中国" Province="安徽"
                City="合肥" StreetName="长江路" StreetNo="355"/>
         <AddressInfo  Country="中国"  City="北京"
                StreetName="望京南路" StreetNo="10"/>
         <AddressInfo  Country="中国" Province="安徽"
                City="芜湖" StreetName="九华山路" StreetNo="23"/>
         <AddressInfo  Country="中国" Province="安徽"
                City="合肥" StreetName="长江路"  StreetNo="21"/>
         <AddressInfo  Country="中国" Province="陕西" City="西安"
                StreetName="咸宁路" StreetNo="79"/>
       </AddressList>
     </x:XData>
   </XmlDataProvider>
 </Window.Resources>
```

我们看到XmlDataProvider把AddressList“包装”了起来，使用x:Kay来标识AddressList资源，与所有资源对Key的要求一样，这里的Key必须在Window.Resource范围内是唯一的。Xpath属性用来指向XML的树根。

在<x:XData>和</x;XData>两个标识中间，放入我们的AddressList数据。注意，在<AddressList>中加入了xmlns属性：

```
 xmlns=" "
```

这是必须的，否则，这个元素会自动继承Window中的名称空间，即在AddressList中加入了：

```
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
```

两个名称空间，这时XPath就不会像预期那样地工作。然后我定义一个DataTemplate用于显示列表框中的每一个条目：

```
<DataTemplate x:Key="addressTemplate">
      <Label Content="{Binding XPath=@City}" Foreground ="Coral"
 Background ="LightPink"/>
 </DataTemplate>
```

把上面的XML岛屿中的数据绑定到列表框ListBox上，非常简单：

```
<ListBox Name="lbAddress" Grid.Column ="0" Grid.Row ="0"
Grid.ColumnSpan ="2"
IsSynchronizedWithCurrentItem="True"
ItemsSource="{Binding}"
ItemTemplate="{StaticResource addressTemplate}" />
```

与前面对ListBox的绑定的唯一区别就是使用了XPath。XAML中的XPath遵守W3C标准，若读者对XPath不熟，Michael Key《XPath 2.0 programmer 's Reference》是一本不错的参考书。

同样，我们用XPath代替Path来绑定原来的字符输入框TextBox，整个程序如下：

```
<Window x:Class="Yingbao.Chapter11.Island.IslandWindow"
```

```
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="XML 数据源" Height="300" Width="500">
  <Window.Resources>
    <XmlDataProvider x:Key ="xmlAddress"
        XPath ="AddressList/AddressInfo">
      <x:XData >
        <AddressList xmlns="">
          <AddressInfo  Country="中国" City="北京"
              StreetName="学院路" StreetNo="178"/>
          <AddressInfo  Country="中国" Province="安徽"
             City="合肥" StreetName="长江路" StreetNo="355"/>
          <AddressInfo  Country="中国"  City="北京"
             StreetName="望京南路" StreetNo="10"/>
          <AddressInfo  Country="中国" Province="安徽" City="芜湖"
             StreetName="九华山路" StreetNo="23"/>
          <AddressInfo  Country="中国" Province="安徽" City="合肥"
             StreetName="长江路" StreetNo="21"/>
          <AddressInfo  Country="中国" Province="陕西" City="西安"
             StreetName="咸宁路" StreetNo="79"/>
        </AddressList>
      </x:XData>
    </XmlDataProvider>
    <DataTemplate x:Key="addressTemplate">
      <Label Content="{Binding XPath=@City}" Foreground ="Coral"
      Background ="LightPink"/>
    </DataTemplate>
  </Window.Resources>
  <Grid DataContext ="{StaticResource xmlAddress}" >
    <Grid.ColumnDefinitions >
      <ColumnDefinition Width ="120"/>
      <ColumnDefinition Width ="2*"/>
    </Grid.ColumnDefinitions>
    <Grid.RowDefinitions>
      <RowDefinition  Height ="100" />
      <RowDefinition  Height ="25" />
      <RowDefinition  Height ="25" />
      <RowDefinition  Height ="25" />
      <RowDefinition  Height ="25" />
      <RowDefinition  Height ="25" />
      <RowDefinition  Height ="25" />
      <RowDefinition  Height ="25" />
    </Grid.RowDefinitions>
    <ListBox Name="lbAddress" Grid.Column ="0" Grid.Row ="0"
       Grid.ColumnSpan ="2" IsSynchronizedWithCurrentItem="True"
       ItemsSource="{Binding}"
       ItemTemplate="{StaticResource addressTemplate}" />
    <Label Content ="国家: " FontSize ="14" Grid.Row="2"
       Grid.Column ="0"/>
    <TextBox  Name="txtCountry"  HorizontalAlignment ="Center"
       FontSize ="10" Width ="130" Height ="18" Grid.Row ="2"
```

```
        Grid.Column ="1" Text="{Binding XPath=@Country}"/>
        <Label Content ="省份：" FontSize ="14" Grid.Row="3"
          Grid.Column ="0"/>
    <TextBox  Name="txtProvince"  HorizontalAlignment ="Center"
          FontSize ="10" Width ="130" Height ="18" Grid.Row ="3"
          Grid.Column ="1" Text="{Binding XPath=@Province}"/>
    <Label Content ="城市：" FontSize ="14" Grid.Row="4"
          Grid.Column ="0"/>
    <TextBox  Name="txtCity"  HorizontalAlignment ="Center"
          FontSize ="10" Width ="130"  Height ="18" Grid.Row ="4"
          Grid.Column ="1" Text="{Binding XPath=@City}"/>
    <Label Content ="街道：" FontSize ="14" Grid.Row="5"
          Grid.Column ="0"/>
    <TextBox  Name="txtStreet"  HorizontalAlignment ="Center"
          FontSize ="10" Width ="130" Height ="18" Grid.Row ="5"
          Grid.Column ="1" Text="{Binding XPath=@StreetName}"/>
    <Label Content ="街道号：" FontSize ="14" Grid.Row="6"
          Grid.Column ="0"/>
    <TextBox  Name="txtStreetNo"  HorizontalAlignment ="Center"
          FontSize ="10" Width ="130" Height ="18" Grid.Row ="6"
          Grid.Column ="1" Text="{Binding XPath=@StreetNo}"/>
</Grid>
</Window>
```

上面的XAML运行结果如图11-14所示。

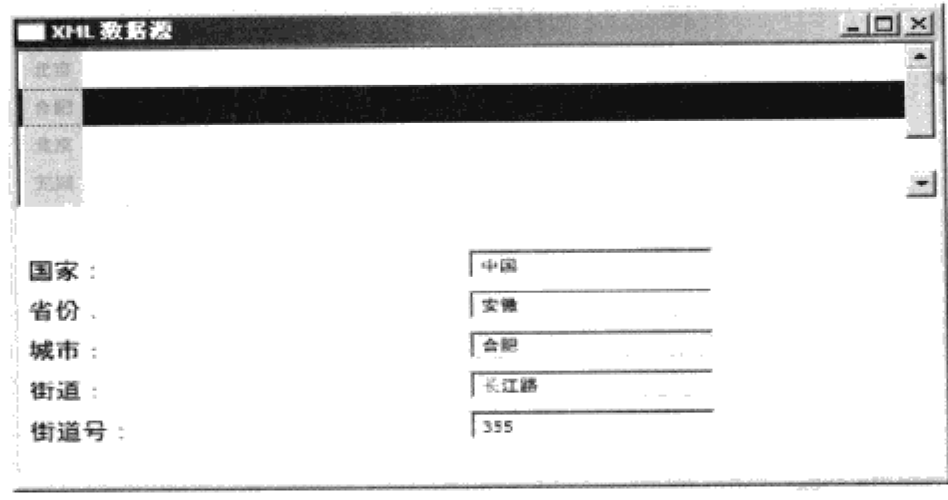


图11-14 使用XML岛屿绑定地址

- 绑定XML文件

XML岛屿一般用于静态XML数据场合，当用户需要修改地址时，XML岛屿显然无法把用户修改的结果保留起来。若用户需要对XML进行修改，就必须使用XML文件。在XAML中利用XmlDataProvider，可以引入XML文件。

笔者把上面XML岛屿中的地址信息放到一个XML文件AddressList.xml中的程序如下：

```
<?xml version="1.0" encoding="utf-8" ?>
<AddressList >
  <AddressInfo  Country="中国" City="北京" StreetName="学院路"
        StreetNo="178"/>
```

```
  <AddressInfo  Country="中国" Province="安徽" City="合肥"
       StreetName="长江路" StreetNo="355"/>
  <AddressInfo  Country="中国"  City="北京" StreetName="望京南路"
       StreetNo="10"/>
  <AddressInfo  Country="中国" Province="安徽" City="芜湖"
       StreetName="九华山路" StreetNo="23"/>
  <AddressInfo  Country="中国" Province="安徽" City="合肥"
       StreetName="长江路" StreetNo="21"/>
  <AddressInfo  Country="中国" Province="陕西" City="西安"
       StreetName="咸宁路" StreetNo="79"/>
</AddressList>
```

然后，把该XML文件引入到Window的资源中：

```
<XmlDataProvider x:Key="xmlAddress" Source="AddressList.xml"
XPath="AddressList/AddressInfo" />
```

笔者使用xmlDataProvider的Source属性引入AddressList.xml文件。注意我们的XML和可执行的聚合体在同一个目录下，否则要用到第8章资源中所描述的Uri。

最后，在界面上加上“存盘”按钮：

```
<Button Name="btnSave"  Width ="50" Margin ="2" Click="OnSave"
     Grid.Row="7"  Grid.Column="0">存盘</Button>
```

并在OnSave事件处理程序中，存储修改的结果：

```
public void OnSave(object sender, RoutedEventArgs rea)
{
     XmlDataProvider addressData =
           this.FindResource("xmlAddress") as XmlDataProvider;
     addressData.Document.Save("AddressList.xml");
}
```

从上面的程序可以看出，利用XmlDataProvider类，可以很方便地读取、修改并存储XML数据。也可以添加、删除XML记录，有关在XML中添加、删除XML记录，读者可以自行完成。

- 声明XML命名空间

XML数据文件常常含有名字空间，例如，可以在AddressList中加上命名空间：

```
<AddressList
xmlns:ChinaAddress="http://yingbao.com/China/Address">
```

要在XAML中使用正确的XPath，要设置XmlNamespaceManager属性：

```
<XmlDataProvider x:Key="xmlAddress" Source="AddressList.xml"
XmlNamespaceManager="{StaticResource namespaceMapping}"
XPath="AddressList/AddressInfo" />
```

其中namespaceMapping定义如下：

```
<XmlNamespaceMappingCollection x:Key ="namespaceMapping">
      <XmlNamespaceMapping Uri="http://yingbao.com/China/Address"
```

```
        Prefix="ChinaAddress"/>
</XmlNamespaceMappingCollection>
```

之后，在XPath中，就可以使用该命名空间。

11.11.2 .NET 对象数据源

与XmlDataProvider用来包装XML数据一样，ObjectDataProvider用来包装一般.NET对象。也许你会问，我们不是已经可以对.NET对象进行绑定了吗？是的，通常我们并不需要对.NET对象进行包装，只是在某些特殊的情况下，使用ObjectDataProvider：

- 在XAML中使用带参数的构造函数；
- 在慢处理程序中（有大量数据在后台），需要用到异步处理时；
- 在XAML中把对象中的方法绑定到源对象上。

1. 在XAML中使用带参数的构造函数

有时候，.NET对象提供的构造函数带有参数，这时候就要在XAML中用到ObjectDataProvider。例如本章前面的地址表，假定AddressList中能够放入的地址元素AddressInfo数目是在其构造函数中说明的：

```
public AddressList(int capacity)
  {
            this.Capacity = capacity;
  }
```

要在XAML中构造这个AddressList，就需要使用ObjectDataProvider对AddressList进行包装，其语法如下：

```
<ObjectDataProvider x:Key ="addressListWithParam" ObjectType ="{x:Type
   src:AddressList}">
  <ObjectDataProvider.ConstructorParameters >
   <sys:Int32>50</sys:Int32>
  </ObjectDataProvider.ConstructorParameters>
</ObjectDataProvider>
```

在这里把AddressList中可以放入AddressInfo对象的个数设为50。使用ObjectType属性来说明所要包装的对象类型，这里是AddressList。

2. 绑定到对象中的方法上

使用ObjectDataProvider可以把目标对象直接绑定源对象中的方法，这在某些情况下非常有用。下面是一个简单的在组合框中显示常用颜色的例子：

```
namespace Yingbao.Chapter11.BindingWithObjectDataProvider
{
    using System;
    using System.Collections.Generic;
    using System.Text;
    using System.Collections.ObjectModel;
```

```
    using System.ComponentModel;
    using System.Collections.Specialized;
    using System.Windows.Media;
    using System.Reflection;
    public static class ColorHelper
    {
        public static IEnumerable<string> GetColorNames()
        {
            foreach (PropertyInfo p
              in typeof(Colors).GetProperties(
              BindingFlags.Public | BindingFlags.Static))
            {
                yield return p.Name;
            }
        }
    }
}
```

ColorHelper类中含有一个静态方法：GetColorNames()，这个方法返回Colors类中定义的各种颜色名。和第5章画笔和画刷中的有关颜色的例子一样，这里也用了.NET的Reflection技术从元数据（Metadata）中获取颜色名。

接着，使用ObjectDataProvider对GetColorNames()方法进行包装：

```
<ObjectDataProvider x:Key="colors"  ObjectType="{x:Type src:ColorHelper}"
           MethodName="GetColorNames"/>
```

然后把GetColorNames绑定到组合框的ItemsSource上：

```
<ComboBox  ItemsSource="{Binding Source={StaticResource colors}}"/>
```

下面是完整的程序：

```
<Window x:Class=
  "Yingbao.Chapter11.BindingWithObjectDataProvider.MainWindow"
 xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:src=
    "clr-namespace:Yingbao.Chapter11.BindingWithObjectDataProvider"
    xmlns:sys="clr-namespace:System;assembly=mscorlib"
    Title="使用ObjectDataProvider" Height="200" Width="400">
  <Window.Resources>
    <ObjectDataProvider x:Key="colors"
     ObjectType="{x:Type src:ColorHelper}"
     MethodName="GetColorNames"/>
  </Window.Resources>
  <StackPanel>
    <ComboBox
     ItemsSource="{Binding Source={StaticResource colors}}">
      <ComboBox.ItemTemplate>
        <DataTemplate>
          <StackPanel Margin="1" Orientation="Horizontal">
            <Rectangle Fill="{Binding}" Height="10" Width="10"
```

```
                Margin="2"/>
            <TextBlock Text="{Binding}" Margin="2 0 0 0"/>
          </StackPanel>
        </DataTemplate>
      </ComboBox.ItemTemplate>
    </ComboBox>
  </StackPanel>
</Window>
```

在这里为了在组合框形象地显示颜色及颜色名字，笔者使用了ItemTemplate。使用ObjectDataProvider绑定GetColorNames方法的程序运行结果如图11-15所示。

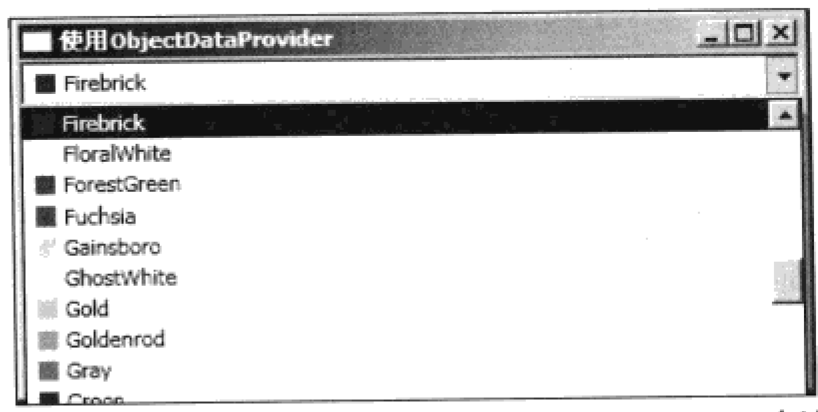


图11-15 使用ObjectDataProvider绑定GetColorNames方法

11.12 层次结构数据的绑定

树形图（TreeView）可以显示具有层次结构的数据，在第10章模板中，我们用TreeView显示了西安交通大学电气学院和电子与信息工程学院的组织架构。读者可以结合本章的内容，重新温习第10章中的例子。

11.13 本章小结

本章系统地讨论了WPF中数据绑定的各个方面，使用WPF的数据绑定，可以使应用程序的代码更加简洁。若你的应用程序界面和数据层的对象间的关系和WPF所设计的数据绑定情形类似，那么使用数据绑定是非常方便的。若你的数据对象间的关系非常复杂，那么你可能要为数据绑定写大量附加代码，这个时候是否该使用数据绑定技术，就是一个很大的问号了。

数据绑定常常要用到第10章模板中的控件模板技术，读者需要把这两个技术结合使用。

第12章 窗口对话框和打印

本章讨论WPF视觉树中的树根对象——窗口（Window）、网页（Page）和NavigateWindow。在前面的章节中，已经多次用到了Window和Page对象。Window主要用在桌面应用程序中，而Page则主要用在浏览器中。实际上WPF的设计目标是建立桌面应用程序和互联网应用程序的统一编程。

Window和Page的主要功能是给WPF中的各种控件提供展现自己的平台，再使用数据绑定把控件和后台数据相连。

12.1 窗口（Window）

WPF窗口是一个内容控件（见第6章6.2节内容控件），与MFC（Microsoft Foundation Class）的窗口一样，视窗通常由下面几个部分组成：

- 边框；
- 标题栏；
- 图标（通常显示公司标识）；
- 最小化、最大化和恢复按钮；
- 系统菜单（通常位于左上角，当按下图标时显示）；
- 关闭按钮（位于窗口右上角的X按钮）；
- 用户区（这里是放置其他控件的地方）。

窗口的主要作用是为特定的内容提供显示平台。WPF视窗是由类Window管理的，我们在前面的章节中已经大量用到了Window类，Window是从ContentControl中派生出来，它是一个内容控件（如第6章图6-2所示），和所有的内容控件一样，它只能含有唯一的一个子控件。笔者在第6章中介绍内容控件时，曾经提到Window类和NavigateWindow类，和其他内容控件类相比，这两个类比较特殊，它们通常为视窗程序提供宿主。

12.1.1 窗口的状态变化和事件

在Windows操作系统中，桌面应用程序是在一个视窗里运行的，这个视窗称为主窗口。应用程序除了一个主窗口之外，还可以创建任意多个视窗。每个视窗都有一个生命周期，就像生物的生命周期一样，视窗在某个时间点，一定处在某种状态下。图12-1 用UML语言示出了视窗的状态图：

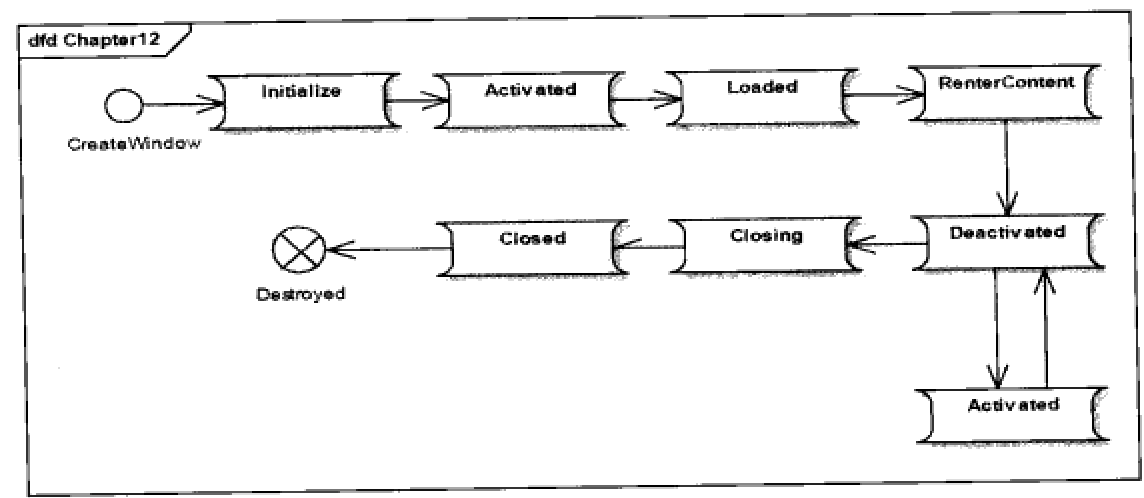


图12-1 Window对象在应用程序中的状态变化图

一个视窗有创建(Create)、初始化（Initialize）、激活（Activated）、装载（Loaded）、提交内容（Renter Content）、失活（Deactivated）、关闭中（Closing）、已关闭（Closed）这样一些状态。伴随着状态的变化，视窗会发出一些传递事件，传递事件沿着视觉树传播。当视窗处于激活状态时，它可以和人进行交互，即可以接受键盘或鼠标输入；但视窗处于失活状态，它就不再能和人进行交互。激活和失活这两种状态可以相互转化，如我们在Windows操作系统里同时运行多个程序，那么只有一个程序的主视窗是激活的，其他应用程序都处在失活的状态。当我们在任务管理器或状态栏上选择相应的应用程序，从而激活相应的应用程序的主窗口，这时候原来处在激活状态的视窗就转变为失活状态；而原处在失活状态的主窗口就由失活而转变为激活状态。在整个操作系统中，在某一个特定时刻，永远只有一个视窗处在激活状态。

当用户单击“关闭”按钮、“确认”按钮、“取消”按钮、或菜单上的相关条目时，视窗进入退出程序，这时会经过关闭中、和已关闭状态、并进而消亡。当视窗发出“关闭中”事件时，这是应用程序可以阻止视窗关闭的最后的机会。视窗一旦进入已关闭的状态，我们就无法把它“救活”了，这个时候我们要做的是清理视窗所用的某些资源（如关闭视窗打开的文件等），即开始处理后事。

当视窗进入析构（Destroged）状态时，其中的子元素会自动进入析构状态，.NET的废品回收机制（Garbage Collection）会在适当的时候回收其所占用的资源。

为了研究窗口的生命周期，笔者创建了一个类，这个类用于记录视窗的状态：

```
using System;
using System.Collections.Generic;
using System.Text;
using System.IO;
namespace Yingbao.Chapter12.WindowLifeTime
{
    public class LogFile
    {
       public static void LogMessage(string message)
        {
            FileStream fs = GetFileStream();
            StreamWriter writer= new StreamWriter(fs);
            writer.WriteLine(message);
            writer.Close();
            fs.Close();
```

```
        }

        static FileStream GetFileStream()
        {
            FileStream fs;
            string logFileName =
             AppDomain.CurrentDomain.BaseDirectory + "LogFile.txt";
            if (!File.Exists(logFileName))
            {
                fs = File.Create(logFileName);
            }
            else
            {
                fs= File.Open(logFileName, FileMode.Append);
            }
            return fs;
        }
    }
}
```

这个类的作用是在运行程序的目录下，产生一个LogFile.txt文件，当LogMessage 方法被调用时，在LogFile.txt中写入一行信息。这一功能和Console.WriteLine很类似，只是Console是向DOS窗口写字符，LogFile是向文件写字符串。

```
<Window x:Class="Yingbao.Chapter12.WindowLifeTime.MainWindow"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="视窗的生命周期" Height="300" Width="300">
  <StackPanel >
    <Button>视窗的生命周期</Button>
  </StackPanel>
  </Window>
```

上面这段XAML很简单，窗口里含有一个StackPanel，StackPanel里含有一个按钮Button控件。这里的重点是C#中MainWindow，它从Window类里派生出来，在其构造函数中，笔者加入了对激活（Activated）、正关闭（Closing）、已关闭、显示内容等事件的处理。

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Data;
using System.Windows.Documents;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Media.Imaging;
using System.Windows.Shapes;

namespace Yingbao.Chapter12.WindowLifeTime
{
     public partial class MainWindow : System.Windows.Window
```

```
{
    public MainWindow()
    {
        InitializeComponent();
        this.Activated += new EventHandler(OnWindowActivated);
        this.Closing +=new
                System.ComponentModel.CancelEventHandler(
                OnWindowClosing);
        this.Closed += new EventHandler(OnWindowClosed);
        this.ContentRendered +=new
                EventHandler(OnWindowContentRendered);
        this.Deactivated +=new
                EventHandler(OnWindowDeactivated);
        this.FocusableChanged +=new
                DependencyPropertyChangedEventHandler(
                OnWindowFocusableChanged);
        this.GotFocus +=new
                RoutedEventHandler(OnWindowGetFocus);
        this.GotKeyboardFocus += new
                KeyboardFocusChangedEventHandler(
                        OnWindowGotKeyboardFocus);
        this.Initialized += new
                EventHandler(OnWindowInitialized);
        this.Loaded +=new RoutedEventHandler(OnWindowLoaded);
        this.LostFocus +=new
                RoutedEventHandler(OnWindowLostFocus);
        this.LostKeyboardFocus +=new
                KeyboardFocusChangedEventHandler(
                        OnWindowLostKeyboardFocus);
    }

    void OnWindowActivated(object sender, EventArgs ea)
    {
        LogFile.LogMessage("OnWindowActivated");
    }

    void OnWindowClosing(object sender,
            System.ComponentModel.CancelEventArgs cea)
    {
        LogFile.LogMessage("OnWindowClosing");
    }

    void OnWindowClosed(object sender, EventArgs ea)
    {
        LogFile.LogMessage("OnWindowClosed");
    }


    void OnWindowContentRendered(object sender, EventArgs ea)
    {
        LogFile.LogMessage("OnWindowContentRendered");
    }
```

```
        void OnWindowDeactivated(object sender, EventArgs ea)
        {
            LogFile.LogMessage("OnWindowDeactivated");
        }

        void OnWindowFocusableChanged(object sender,
                DependencyPropertyChangedEventArgs  ea)
        {
            LogFile.LogMessage("OnWindowFocusableChanged");
        }

        void OnWindowGetFocus(object sender, RoutedEventArgs  ea)
        {
            LogFile.LogMessage("OnWindowGetFocus");
        }

        void OnWindowGotKeyboardFocus(object sender,
            KeyboardFocusChangedEventArgs kea)
        {
            LogFile.LogMessage("OnWindowGetFocus");
        }

        void OnWindowInitialized(object sender, EventArgs ea)
        {
            LogFile.LogMessage("OnWindowInitialized");
        }

        void OnWindowLoaded(object sender, RoutedEventArgs rea)
        {
            LogFile.LogMessage("OnWindowLoaded");
        }

        void OnWindowLostFocus(object sender, RoutedEventArgs rea)
        {
            LogFile.LogMessage("OnWindowLostFocus");
        }

        void OnWindowLostKeyboardFocus(object sender,
            KeyboardFocusChangedEventArgs  rea)
        {
            LogFile.LogMessage("OnWindowLostKeyboardFocus");
        }
    }
}
```

在创建视窗时，我们看到LogFile里的记录：

```
OnWindowActivated
OnWindowGetFocus
OnWindowLoaded
OnWindowContentRendered
```

即窗口在创建之后立刻就进入了激活态、接着是获得了输入焦点、跟着是窗口加载、最后是内容提交（显示）。

当关闭窗口时，视窗产生下面的记录：

```
OnWindowClosing
OnWindowDeactivated
OnWindowLostKeyboardFocus
OnWindowClosed
```

即窗口先发出要关闭的信息、接着是失活、然后是键盘失去输入焦点、最后是窗口关闭事件。注意，事件产生的顺序非常重要，比如要关闭（Closing）必须在失活之前发出，这样，应用程序才有机会去阻止窗口关闭。

当显示某个窗口的时候，我们是不是希望窗口一定要激活呢？不一定。比如说你要开发一个类似于Visual Studio的软件，其中要管理多个工具窗口，这些工具窗口在一开始创建并显示时，我们并不希望它们处在激活的状态。为此，WPF在3.5中引入了一个属性ShowActivated，若把这个属性设为false，那么，在显示窗口时视窗就处在失活状态，处在失活状态下的视窗不会发出Activated和GetFocus事件，原来处在激活状态的视窗仍然处在激活状态。在默认的情况下ShowActivated属性设为true，因此，新创建的视窗在显示时处在激活状态。

当视窗处在激活状态时，它可以激活同一应用程序的不同窗口，或不同的应用程序的窗口。当其他窗口被激活时，原来的窗口就自动转到失活状态。总而言之，在Windows操作系统中只有一个窗口处在激活状态，只有一个窗口具有输入焦点，属性IsActive表示视窗是否处在激活状态。

Closing事件在视窗要关闭前发出，有的时候我们需要阻止窗口关闭。比如说，视窗用来接收用户的输入，在用户的输入有错、而按下“确认”时，应该显示错误信息，而不关闭窗口。再比如说，文字输入窗口，在用户关闭窗口前，需要提示用户是否需要存盘，若用户选择取消退出窗口，则不应该关闭该窗口。阻止窗口关闭很简单，只要把CancelEventArgs中的Cancel属性设为true：

```
void OnWindowClosing(object sender,
         System.ComponentModel.CancelEventArgs cea)
   {
        cea.Cancel = true;
        LogFile.LogMessage("OnWindowClosing");
   }
```

12.1.2 确定视窗的位置

视窗的位置用Left和Top两个属性表示，它是相对于屏幕左上角的坐标。视窗的位置还受到另外一个属性的控制，这个属性是WindowStartupLocation，这是一个枚举类型，其取值是：

- Manual（默认设置） 这时视窗的位置由Left、Top两个坐标确定；
- CenterScreen 这时视窗自动被放在屏幕的中间的位置；
- CenterOwner 这时视窗被放在拥有该窗口的中间的位置。若一个视窗的Owner设为null，而WindowStartupLocation设为CenterOwner，其行为和Manual一样。

12.1.3 确定视窗的大小

视窗的大小由多个属性共同作用：Width、Height、MinWidth、MaxWidth、MinHeight、MaxHeight和SizeToContent。而视窗的真实大小由ActualWidth和ActualHeight两个只读属性确定。

若视窗设置了MinWidth和MaxWidth，那么视窗的宽度只能在这两个数值之间，比如在我们设置MinWidth和MaxWidth之后：

MinWidth=200；MaxWidth=500；

若把Width设为100，我们看到视窗的ActualWidth的值为200，而不是100，因为视窗宽度的最小值为MinWidth；若把Width设为300，视窗的ActualWidth的值为300，即视窗的实际宽度和所设的宽度相等；若把Width设为600，视窗的ActualWidth的值为500，即视窗的实际宽度等于最大宽度值。

类似的结论也适用于高度。

其实视窗的上述和大小相关的属性是从FrameworkElement类中继承下来的，FrameworkElement中之所以要支持上述相关属性，是由于排版的需要。我们知道，当我们改变视窗的大小时，所有的控件的大小也会相应地发生改变，有时候我们需要告诉WPF该元素在一定的尺寸范围，其显示的效果最好。

SizeToContent属性是一个枚举类型，设置这个属性可以让视窗的大小随着其中内容的变化而变化。

- Manual（默认值），这时，视窗的大小由Width、Height、MinWidth、Max-Width、MinHeight和MaxHeight共同确定。
- Width，视窗的宽度随着其中界面元素的宽度而变化。
- Height，视窗的高度随着其中界面元素的高度而变化。
- WidthAndHeight，视窗的宽度和高度随着其中界面元素的宽度和高度而变化。

在设置了SizeToContent属性后，视窗的大小随着其中内容的变化而变化，但是，无论宽度还是高度，都必须受到MinWidth、MaxWidth和MinHeight，MaxHeight的约束。

12.1.4 视窗状态属性（WindowState）

对于可改变大小的视窗，它可以有三个状态：Normal、Minimized和Maximized。所有使用Windows操作系统的人都熟悉这三种状态，在视窗的右上角，一般有三个按键，其中的两个是用来控制WindowState的。当WindowState为Maximized的时候，它占据整个计算机屏幕。当WindowState为Minimized的时候，若其ShowIntaskBar属性设为True，那么你能在视窗的任务条上找到该窗口的图标；否则只有在Windows操作系统的任务管理器中去找了。

12.1.5 视窗大小模式（ResizeMode）

可以设定允许用户改变视窗大小的方式，ResizeMode可以取下面的一些值：

- NoResize 用户无法改变视窗的大小，这时视窗标题栏上的按钮不再显示。
- CanMinimize 用户可以最小化视窗，视窗标题栏上的最大化按钮为失活状态。

- CanResize 这是一种默认设置，用户可以改变视窗的大小。
- CanResizeWithGrip 这时窗口的右下角显示三条斜线，表示视窗的大小是可以改变的，用鼠标拖动视窗的右下角，窗口的大小也随之改变（如图12-2所示）。

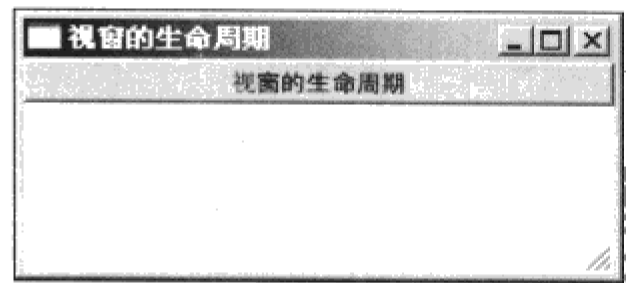


图12-2 在视窗的右下角显示的斜线

12.1.6 视窗风格（WindowStyle）

视窗的风格WindowStyle用来设定视窗的边框，若为枚举类型，可以取下面的一些值：

- None
- SingleBoardWindow
- ThreeDBoardWindow
- ToolWindow

图12-3示出了设置不同WindowStyle值时，窗口边框的样子。由图12-3可见，当WindowStyle为none时，窗口上没有最大最小和X按钮，只有标题；SingleBoardWindow和ThreeDBoardWindow类似，TreeDBoardWindow边框有立体效果；ToolWindow没有最大、最小按钮但有X按钮。

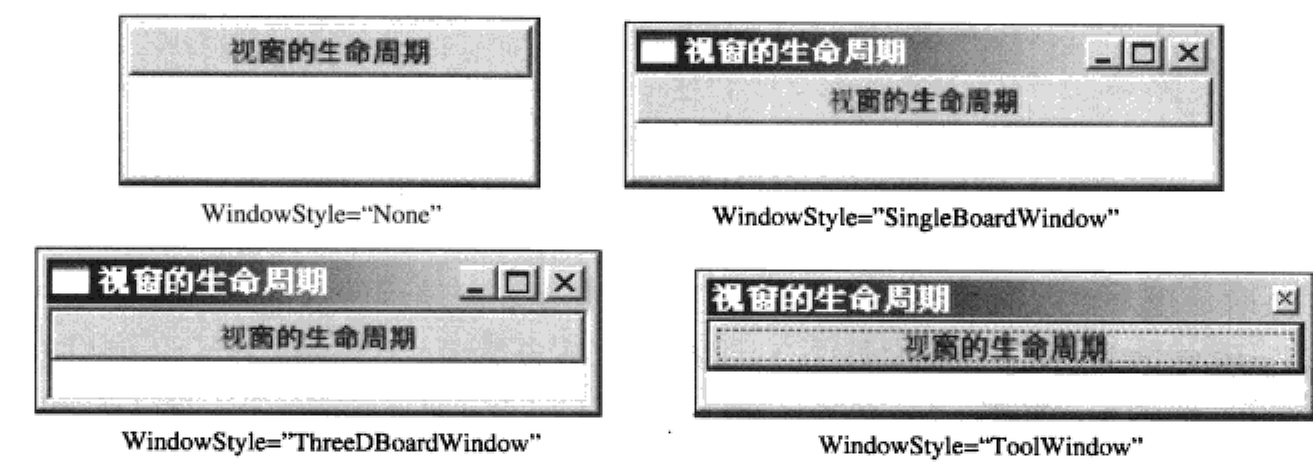


图12-3 不同的风格下，视窗边框的样子

12.2 网页（Page）

XAML浏览器应用程序（XBAPS）基于XAML网页，就像Window类为桌面程序中控件提供宿主（Container）一样，Page类是XAML程序中控件的宿主。

Page类从UIElement中直接派生出来，它比控件所消耗的资源要少，通常在NavigateWindow（见12.3节）中显示，微软的互联网浏览器IE 7及以上的版本支持XAML网页，在浏览器上的插件叫SilverLight，现已可在FireFox及Chrome浏览器上运行，IE使用NavigateWindow和Page相互作用。

Page类中含有一个Content属性，与Window类中的Content属性一样，它可以是任何.NET对象。但

一般情况下，我们把这个属性和排版对象类相连，WPF中的任何控件都可以加入到Page实例中，即Page起到其他控件宿主的作用。

12.2.1 创建网页

在XAML中创建Page类实例和创建Window类似：

```
<Page x:Class="Yingbao.Chapter12.PageNavigation.StartPage"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="XAML 网页"/>
```

X:Class和后面的C#类相连，这个类的名字是StartPage，它在Yingbao.Chapter12. PageNavigation名称空间中。这个类是Page的派生类：

```
namespace Yingbao.Chapter12.PageNavigation
{
    public partial class StartPage: System.Windows.Controls.Page
    {
        public StartPage()
        {
            InitializeComponent();
        }
    }
}
```

就像桌面应用程序一般不直接创建Window类实例，而创建Window的派生类一样；在XBAP应用程序中，一般也不创建Page类实例，而是创建其派生类。

12.2.2 KeepAlive属性

当用户浏览到某个网页时，WPF就创建该网页的一个实例。当用户离开该网页时，WPF的浏览器窗口（NavigationWindow）可以把该网页的URI放到日志（Journal）中，然后销毁该网页。这是默认的管理网页的方式，这种管理的好处是节省计算机的内存资源，这种方式带来的问题是当用户返回到上次的网页时，用户输入的数据会被丢失。这是所有互联网编程的共同特点，在ASP或JSP环境中，我们一般把用户的数据和一次会话联系在一起保存在服务器端。有时候，为了改善实时性能，我们需要使用近年来发展起来的Ajax技术。在客户端，在数据安全不太重要的情况下，可以使用cookies。

WPF提供了一些新的保存网页状态的机制，其中一个就是设置KeepAlive属性为true，这样当离开某个网页时，WPF会自动保存该网页Page实例。

这种做法是有运行时的开销的，因此，除非有运行性能方面的考虑，一般是不用的。

12.2.3 NavigationService属性

NavigationService类的主要功能是管理浏览器的内容，负责把Page和HTML下载到本机上。Page利用NavigationService所提供的导航服务，切换到其他Page。

Page总是在某个宿主中显示的，但Page本身并不知道自己在NavigationWindow类中，还是在Frame中。当Page实例在某个宿主中创建时，宿主就为Page提供这个属性，其类型是NavigationService

类。Page实例可以利用这个属性跳转到其他页面。例如，在Page实例中调用：

this.NavigationService.GoBack();

会从该网页跳转到曾经访问过的网页。

12.2.4 ShowsNavigationUI属性

这个属性用来显示NavigateWindow的前进和后退按钮，由于IE7已经集成了WPF，所以这个属性的值在IE7中没有作用。NavigationWindow中的浏览条如图12-4所示。

图12-4 NavigationWindow中的浏览条

12.3 浏览窗口（NavigationWindow）

说到浏览，大家都会想到互联网的浏览器，它具有从一个网页转到另一个网页、向前翻页和向后翻页的功能。在现实生活中，有很多工作流程具有这样的特点。比如工厂里加工某个零件的工序、物资流通中的管理、机关文件传递等，实际上都可以抽象成工作流程。.NET 中的工作流程平台（Windows Workflow Foundation简称WF）就是为这种应用程序设计的，不过WPF技术主要用在服务和集成，在UI部分，使用的就是NavigateWindow。

在微软的互联网浏览器IE 7及以上的版本中，对显示XAML网页的支持就是基于NavigateWindow上的；换句话说，IE 7使用NavigateWindow来显示XAML网页。.NET技术一直致力于桌面程序和互联网程序统一编程，NavigateWindow就是一个架设在浏览器和桌面程序间的桥梁。

在准备这一节时，笔者创建了一个自己的喜爱的诗人作品集程序，这个程序里可以添加任意多的诗人及其作品，相当于免费在自己的计算机上放了多本诗集，所创建的浏览器窗口如下：

```
<NavigationWindow
x:Class="Yingbao.Chapter12.PoemNavigation.PoemMainWindow"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="PoemNavigation" Height="500" Width="600" Source
="MainPage.xaml"/>
```

在这个浏览窗口中，笔者直接把其中的Source属性和MainPage.xaml相连。MainPage.xaml 是本程序要收集的诗人的名字，相当于一本书的目录，使用NavigationWindow创建的诗人及其作品的主界面如图12-5所示。

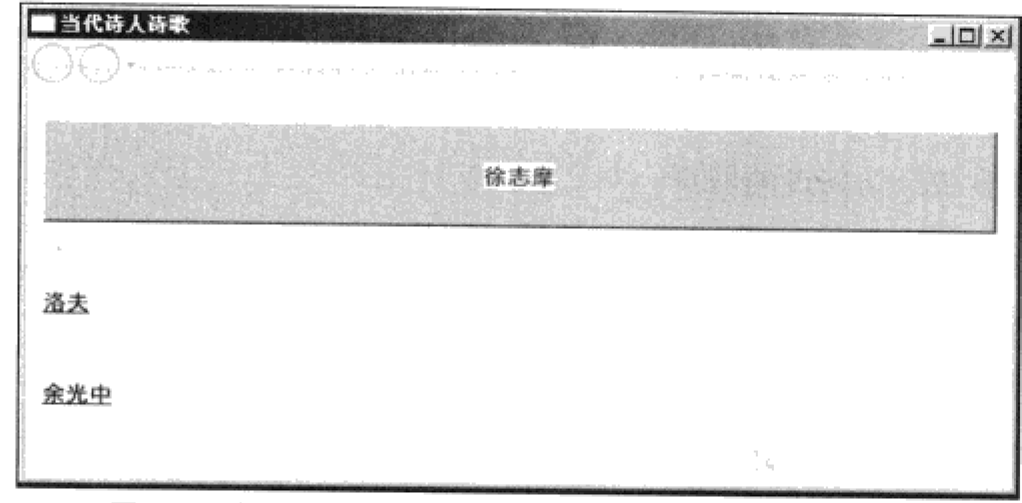


图12-5 使用NavigationWindow创建的诗人及其作品主界面

12.3.1 使用统一风格

为了在多个网页中显示一致的风格，笔者在Application类中加入了简单的风格：

```
<Application x:Class="Yingbao.Chapter12.PoemNavigation.App"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    StartupUri="PoemMainWindow.xaml" >
    <Application.Resources>
        <Style TargetType ="TextBlock">
          <Setter Property ="Margin" Value ="10,20,10,20"/>
          <Setter Property ="FontSize" Value ="14"/>
          <Setter Property ="Foreground" Value ="Brown"/>
          <Setter Property ="Background" Value ="FloralWhite"/>
        </Style>
      <Style TargetType ="Button">
        <Setter Property ="Margin" Value ="10,20,10,20"/>
        <Setter Property ="FontSize" Value ="14"/>
        <Setter Property ="Foreground" Value ="Brown"/>
        <Setter Property ="Background" Value ="LightBlue"/>
      </Style>
    </Application.Resources>
</Application>
```

这里分别对TextBlock和Button定义了风格。

12.3.2 设置NavigationWindow的标题

我们希望每个网页都有自己的标题，所以需要设置NavigationWindow的标题，方法是使用Page中的WindowTitle属性，如在MainPage.xaml中：

```
<Page x:Class="Yingbao.Chapter12.PoemNavigation.MainPage"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    WindowTitle="当代诗人诗歌">
```

这样在显示该页面的时候，NavigationWindow就显示为“当代诗人诗歌”了。

12.3.3 浏览网页

NavigationWindow的特点是可以在其中浏览不同的网页，这些网页可以是XAML写的，也可以是老的HTML页面。主要有三种方式浏览不同的页面：

- 调用NavigationService中的Navigate方法；
- 使用HyperLink类;
- 使用日志。

12.3.4 使用HyperLink类

在当代诗人诗歌主页面中，笔者在洛夫和余光中的条目下，使用的就是这种技术：

```
<Page x:Class="Yingbao.Chapter12.PoemNavigation.MainPage"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    WindowTitle="当代诗人诗歌">
  <StackPanel>
    <Button  Click ="OnZhimo">徐志摩</Button>
    <TextBlock>
      <Hyperlink  NavigateUri ="Luofu.xaml">洛夫</Hyperlink >
    </TextBlock>
    <TextBlock >
      <Hyperlink  NavigateUri="yuguangzhong.xaml">余光中</Hyperlink >
    </TextBlock>
  </StackPanel >
</Page>
```

WPF中的Hyperlink和HTML中的Hyperlink类似，只要设置NavigateUri到相应的XAML网页即可，如本例中的Luofu.xaml和yuguangzhong.xaml：

Luofu.xaml:

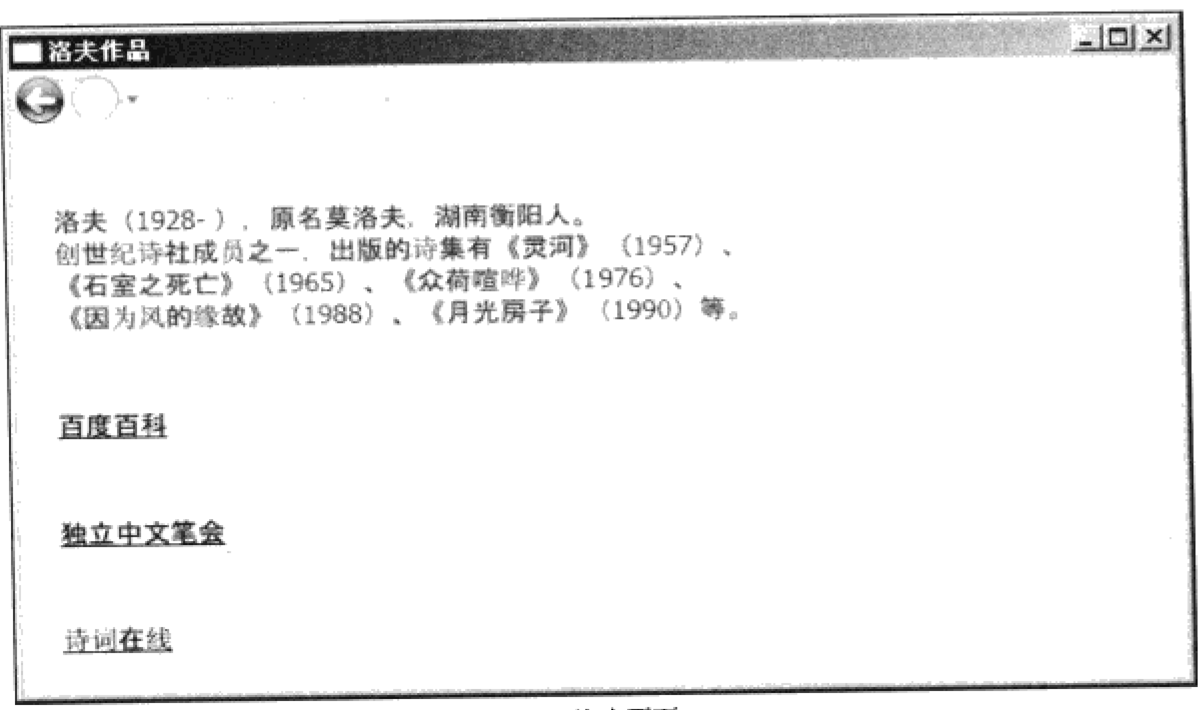
```
<Page x:Class="Yingbao.Chapter12.PoemNavigation.Luofu"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    WindowTitle="洛夫作品">
  <StackPanel  Margin="10, 20,20,10">
  <TextBlock >
        洛夫（1928- ），原名莫洛夫，湖南衡阳人。<LineBreak/>创世纪诗社成员
之一，出版的诗集有《灵河》（1957）、<LineBreak/>《石室之死亡》（1965）、《众
荷喧哗》（1976）、<LineBreak/>《因为风的缘故》（1988）、《月光房子》（1990）
等。
      </TextBlock>
      <TextBlock>
         <Hyperlink NavigateUri=
            "http://baike.baidu.com/view/285602.htm">百度百科
         </Hyperlink >
      </TextBlock>
      <TextBlock >
        <Hyperlink  NavigateUri
```

```
            ="http://www.boxun.com/hero/luofu/2_1.shtml"> 独立中文笔会
          </Hyperlink >
        </TextBlock>
        <TextBlock >
          <Hyperlink  NavigateUri
="http://www.chinapoesy.com/XianDaiAuthora1d7f4df-eca6-4869-aae8-
8b24ef4200f3.html">诗词在线</Hyperlink >
        </TextBlock>
    </StackPanel>
</Page>
```

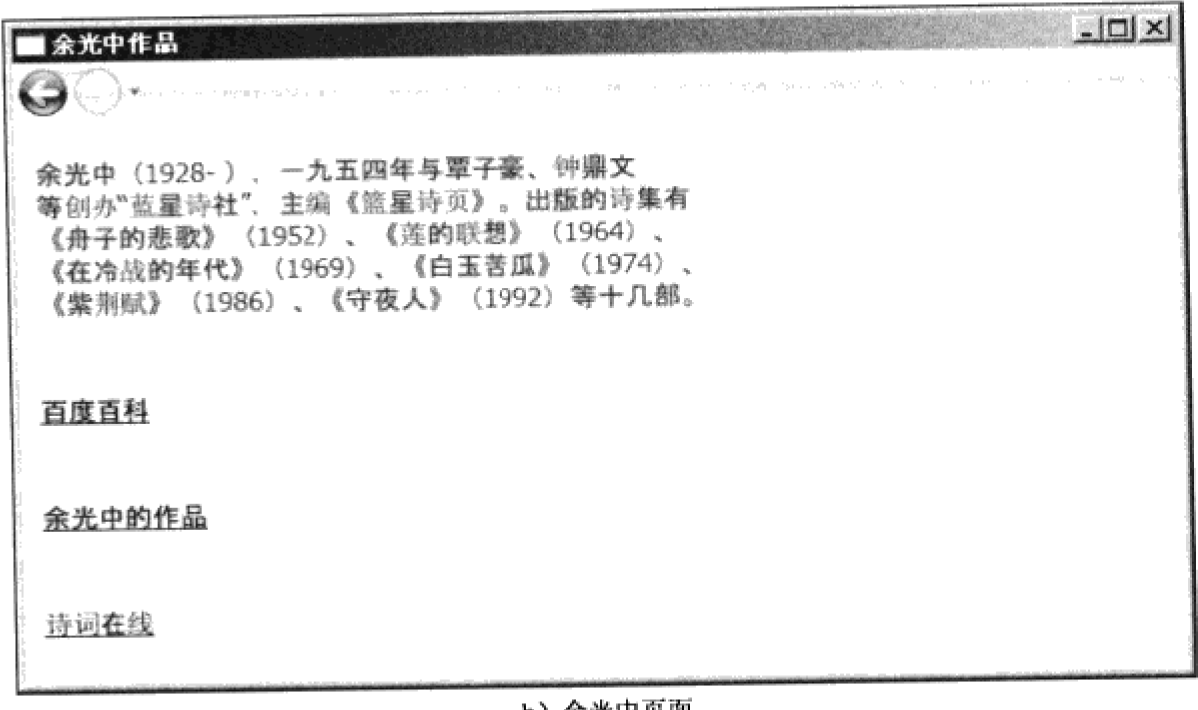
yuguangzhong.xaml:

```
<Page x:Class="Yingbao.Chapter12.PoemNevigation.yuguangzhong"
 xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    WindowTitle="余光中作品">
  <StackPanel >
    <TextBlock >
      余光中（1928- ），一九五四年与覃子豪、钟鼎文<LineBreak/>
      等创办“蓝星诗社”，主编《篮星诗页》。出版的诗集有<LineBreak/>
      《舟子的悲歌》（1952）、《莲的联想》（1964）、<LineBreak/>
      《在冷战的年代》（1969）、《白玉苦瓜》（1974）、<LineBreak/>
      《紫荆赋》（1986）、《守夜人》（1992）等十几部。
      </TextBlock>
    <TextBlock>
      <Hyperlink  NavigateUri
="http://baike.baidu.com/view/5363.htm">百度百科</Hyperlink>
    </TextBlock>
    <TextBlock >
      <Hyperlink  NavigateUri
="http://www.tianyabook.com/yuguangzhong/sanwen.htm"> 余光中的作品
</Hyperlink >
    </TextBlock>
    <TextBlock >
      <Hyperlink  NavigateUri
="http://www.chinapoesy.com/XianDaiAuthor8e114e09-322c-4a67-b2a5-
6584e0696e9d.html">诗词在线</Hyperlink >
    </TextBlock>
  </StackPanel>
</Page>
```

当单击洛夫和余光中时，得到如图12-6 a）和图12-6 b）所示的结果：



a）洛夫页面



b）余光中页面

图12-6 切换到洛夫和余光中页面

在这两个页面中，主要收集两位诗人在互联网上的作品，也可以添加任意多个连接，如：

```
NavigateUri ="http://www.chinapoesy.com/ XianDaiAuthor8e114e09-322c-
4a67-b2a5-6584e0696e9d.html"
```

当我们单击诗词在线时，我们的应用程序自动转到诗词在线网页有关余光中项（如图12-7所示）。

我们在浏览不同的网页时，可以看到浏览工具条上的“后退”按钮能自动工作了；当单击后退按钮时，自动回到前面的网页，同时，“前进”按钮也能自动工作了。

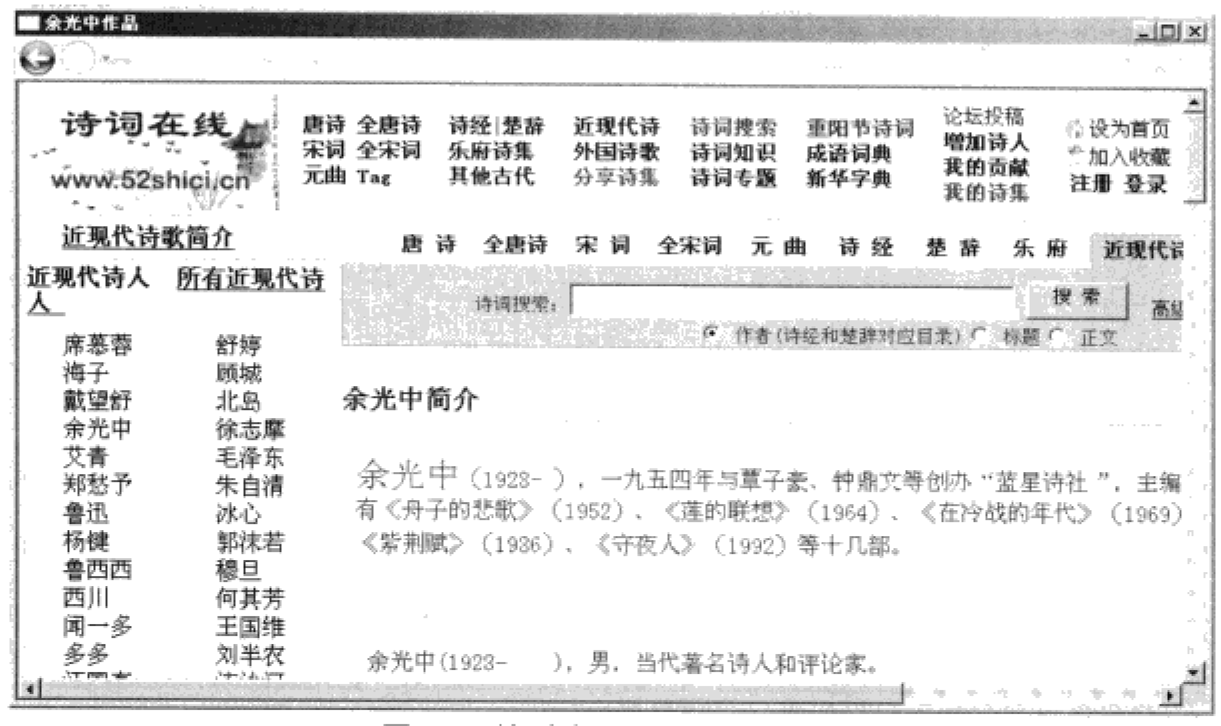


图12-7 转到诗词在线余光中网页

12.3.5 使用NavigationService转到不同的网页

第二种切换网页的方法是调用NavigationService中的Navigate方法。为了演示这一方法，笔者把徐志摩条目用Button表示（这纯粹是从演示使用Navigate方法的角度来设计的，并不是徐志摩条目和其他条目真有什么不同）。在按钮单击事件中，使用Navigate转到xuzhimo.xaml网页：

```
NavigationService.Navigate(new Uri("Xuzhimo.xaml",UriKind.Relative ) );

namespace Yingbao.Chapter12.PoemNavigation
{
    public partial class MainPage : System.Windows.Controls.Page
    {
        public MainPage()
        {
            InitializeComponent();
        }
        void OnZhimo(object sender, RoutedEventArgs rea)
        {
           NavigationService.Navigate(new Uri("Xuzhimo.xaml",
                  UriKind.Relative ) );
         }
    }
}
```

也可以直接设置NavigationService中的Content来转到xuzhimo.xaml网页：

```
void OnZhimo(object sender, RoutedEventArgs rea)
  {
    Xuzhimo xpage = new Xuzhimo();
    NavigationService.Content = xpage;
  }
```

还可以直接设置NavigationService中的Source来达到同样的效果：

```
void OnZhimo(object sender, RoutedEventArgs rea)
 {
    NavigationService.Source = new Uri("Xuzhimo.xaml",
           UriKind.Relative);
 }
```

12.3.6 使用浏览日志转换到不同的网页

NavigationWindow中维护两个堆栈，一个为前向堆栈，保留可以“前进”的页面；一个为后向堆栈，保留可以“后退”的页面。当用户按下“后退”按钮时，当前页面被推入前向堆栈，同时从后退堆栈中取出最上面的页面作为当前页面；当用户按下“前进”按钮时，当前页面被推入后向堆栈，同时从前向堆栈中取出最上面的页面作为当前页面。

若需要在程序中对访问过的页面进行顺序切换，可以调用NavigationService中的GoBack和GoForward方法。

12.3.7 浏览窗口的浏览事件

不管以何种方式加载网页，WPF中网页的加载方式都是以异步的方式进行的。在加载的过程中，WPF会产生一系列事件，这些事件是按照一定的次序出现的，应用程序可以在这些事件出现的时候做相应的处理。浏览窗口加载网页时所产生的事件及其顺序如图12-8所示。

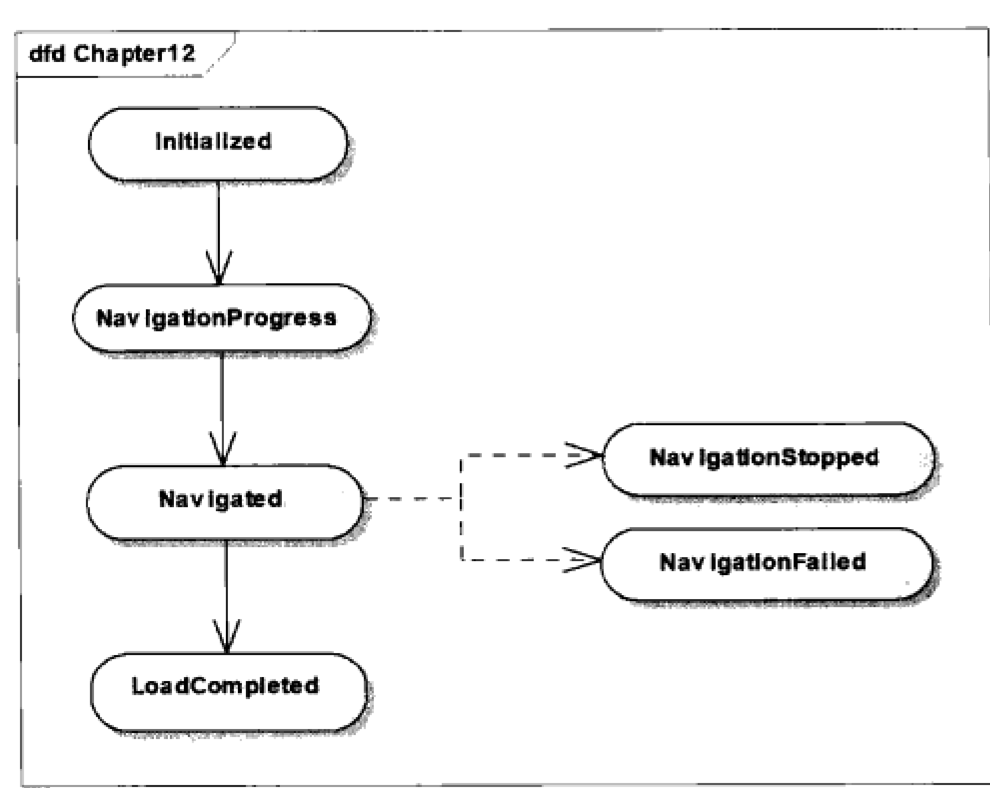


图12-8 浏览窗口加载网页时所产生的事件及其顺序

在图12-8中，当加载页面时，首先发出Initialized事件，接着是NavigationProgress，根据网页的中的内容，NavigationProgress事件会发出多次。若遇到某种情况，终止了浏览过程，WPF会发出NavigationStopped或NavigationFailed事件。

12.4 对话框（DialogBox）

对话框和视窗之间的区别越来越小，在MFC中，对话框有专门的类；在WPF中，对话框实际上就是视窗。通常桌面应用程序有一个主要的窗口，上面有菜单工具条、状态行等界面元素。但这也不一定，比如笔者曾经为摩根斯坦利（J.P.Morgan）和华尔街银行（State Street Bank）做的基金交易系统，其整个人机界面用的都是对话框，在对话框上也做了主菜单、状态行等。

对话框主要用在以下三方面：（1）显示信息；（2）接受用户输入；（3）两者都有。

对话框有两种类型，一种是模式（Model）对话框和无模式（Modelless）对话框。模式对话框，不可以在主窗口和对话框之间切换，当显示模式对话框的时候，用户必须关闭对话框才能回到原来的窗口；无模式对话框则允许用户在对话框和主窗口间自由切换。比如说，Visual studio里的查找对话框，就是一个无模式对话框，可以随时在编辑窗口和查找对话框之间进行切换。

12.4.1 消息框（MessageBox）

WPF的消息框位于System.Window名称空间，与过去MFC中的消息框的功能相同。 使用消息框非常简单，例如图12-9所示的C#代码显示的消息框。

```
string messageBoxText = "文件有错继续存档吗？";
string caption = "图文编辑器";
MessageBoxButton button = MessageBoxButton.YesNoCancel;
MessageBoxImage icon = MessageBoxImage.Warning;
MessageBoxResult result = MessageBox.Show(messageBoxText, caption,
button, icon);
```

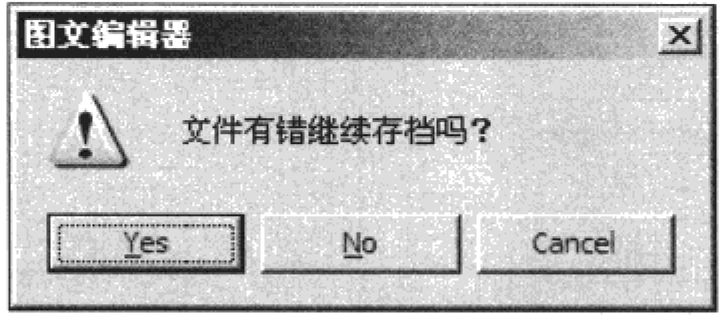


图12-9 WPF消息框

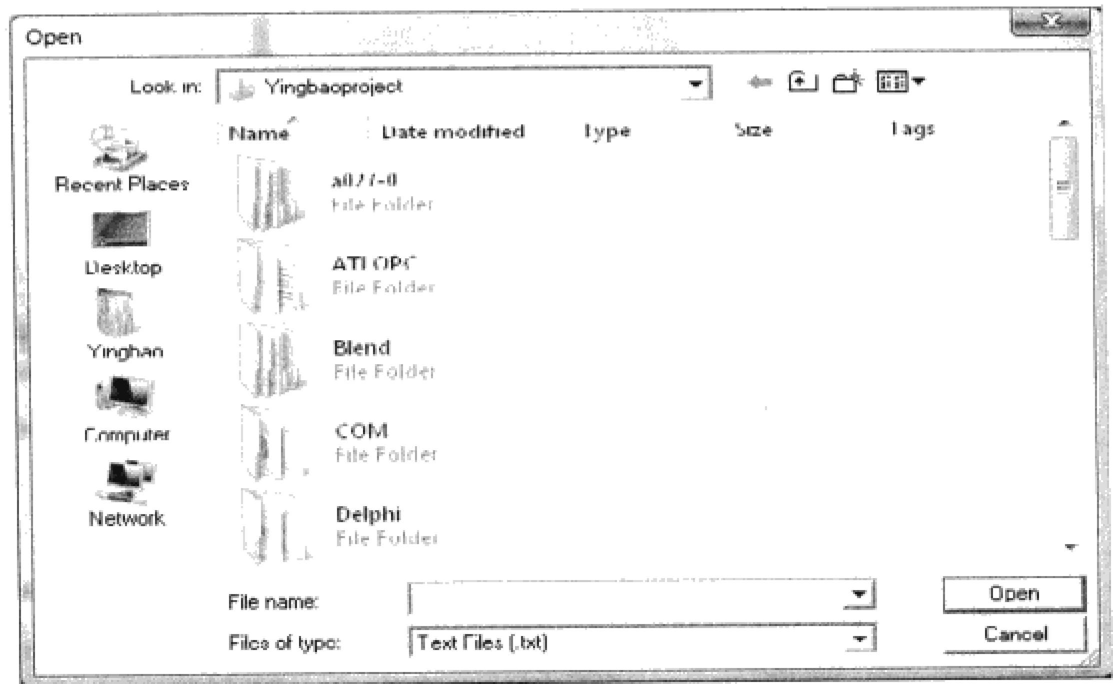
MessageBox中的show方法有十二个重载，可以根据需要选择使用。笔者对于上面的结果不太满意，原来想提供全中文的界面，可是MessageBox中的按钮，无法汉化，这给直接在中文版软件中使用WPF内置的消息框带来麻烦。同样下面的这些通用对话框也有这样的问题，微软应该解决这个问题。

12.4.2 通用对话框

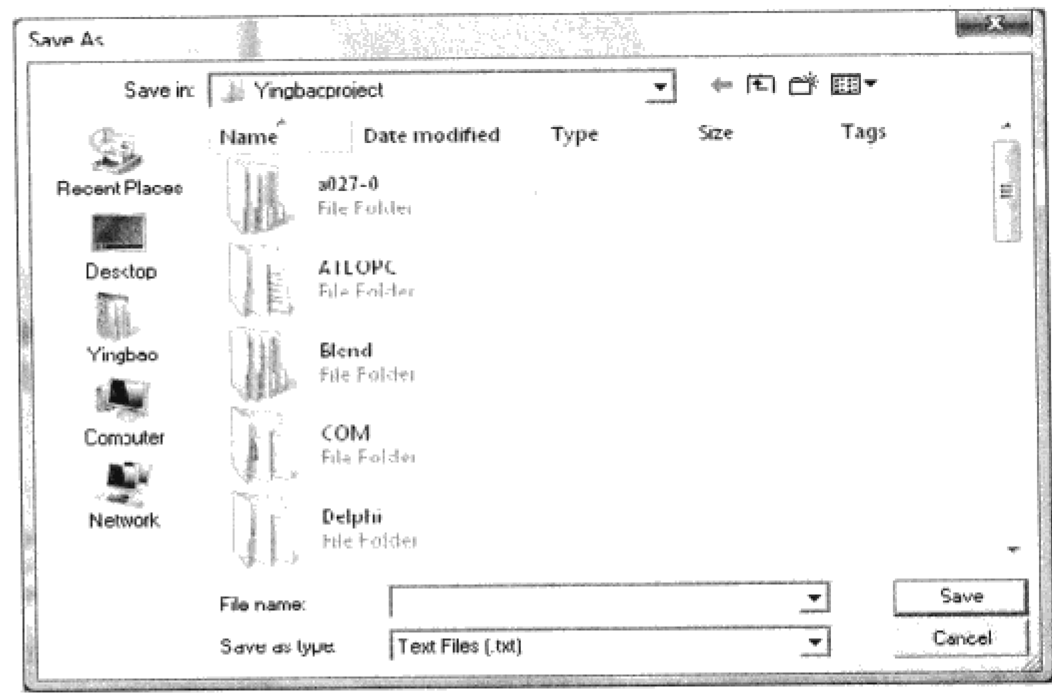
WPF中的通用对话框如打开文件对话框OpenFileDialog、存储文件对话框SaveFileDialog位于Microsoft.Win32命名空间，这些对话框实际上就是Win32的对话框。注意不要把WPF中的通用对话框和System.Window.Forms命名空间中的同名对话框相混淆，虽然其完成的功能和类名字一样（msdn提供了Window、Forms和Window.Controls下控件的比较：http:// msdn.microsoft.com/en-us/library/ms750559. aspx）。

打开文件对话框OpenFileDialog和存储文件对话框对话框。

一般用后台C#显示打开文件对话框，InitialDirectory属性用来指出打开对话框中的“Look in”组合框中的初始目录或存储对话框中的“Save in”组合框中的初始目录（如图12-10所示）。



a）打开文件对话框



b）存储文件对话框

图12-10 打开文件对话框的用户界面

Filter属性用来设定选择的文件后缀，下面的C#示出了OpenFileDialog和存储对话框的用法：

```
void OnOpenFile(object sender, RoutedEventArgs rea)
      {
         OpenFileDialog ofd = new OpenFileDialog();
         ofd.InitialDirectory = @"c:\yingbaoproject";
         ofd.Filter = "Text Files (.txt)|*.txt|All Files
                    (*.*)|*.*";
         ofd.FilterIndex = 1;
         if (ofd.ShowDialog(this) == true)
```

```
            {
                //放入其他逻辑
            }
        }

void OnSaveFile(object sender, RoutedEventArgs rea)
        {
            SaveFileDialog sfd = new SaveFileDialog();
            sfd.InitialDirectory = @"c:\yingbaoproject";
            sfd.Filter = "Text Files (.txt)|*.txt|All Files
                          (*.*)|*.*";
            sfd.FilterIndex = 1;
            if (sfd.ShowDialog(this) == true)
            {
                //放入其他逻辑
            }
    }
```

12.4.3 自定义对话框

更多的时候，需要开发自己的对话框。如前所述，对话框和窗口的界限越来越模糊了。简单来说，使用自定义对话框需要注意下面几点。

- 显示模式（Model）对话框　调用Windows的ShowDialog() 方法，用户只有关闭模式对话，才能回到父窗口。
- 显示无模式（Modelless）对话框　调用Windows的Show()方法，用户可以自由地在父窗口间切换。
- 设置对话框的owner，维护父窗口和对话框间的关系。当父窗口最小化时，字窗口也跟着最小化；当子窗口最小化时，父窗口保持其原来的状态；当父窗口最大化时，父窗口和子窗口均恢复正常的状态；关闭父窗口时，自窗口自动关闭。
- 设置显示对话框的初始位置WindowStartupLocation时，该属性为枚举类型，可以取Menual、CenterScreen和CenterOwner三个值。
- 设置ShowInTaskBar属性，若该属性为True，则窗口会在Windows操作系统的任务栏上显示。对于对话框，要把该属性设为False，即在任务栏上不显示。
- 设置起始具有输入焦点控件，在XAML中设置输入焦点的语法为：

```
FocusManager.FocusedElement="{Binding ElementName=button2}"
```

- 设置默认按钮　默认按钮是当我们按回车键时，产生click事件的按钮。通常情况下，我们把默认按钮设在“确认”按钮上，设置默认按钮的语法为：

```
<Button Name="btnOK" IsDefault="True">确认</Button>
```

- 使用第11章介绍的对话框中用户输入的结果进行校验，并显示出错信息。

若程序员注意上面的技术要点，就可以开发出任何自定义对话框了。

12.5 打印输出

12.5.1 XPS 文档简介

长期以来，软件界一直在寻找一种文件格式，这种格式可用来显示、编辑、打印文件。Adobe公司的ADF文件就是这样的尝试，PDF文件不仅可以在ADF编辑器上显示，而且可以在浏览器上显示，Word文件可以转换成PDF文件；好一点的复印机还可以把纸上的文档通过扫描，转换为PDF文件。

为了支持WPF的打印，微软开发了基于XML的XPS文档（XPS是XML Paper Specification format的速写）。使用XPS来描述文档，并说明如何在打印机上显示出来。XPS采用Zip的压缩方式，按照一定的结构组织XAML。XPS描述的文档和非辨率无关，由于采用矢量技术，图形在各种打印机上输出时不会失真。XSP Viewer 可以免费下载（http://www.microsoft.com/whdc/xps/viewxps.mspx），IE 7支持XSP文件格式，图12-11示出了XPS文档的基本组成部分。

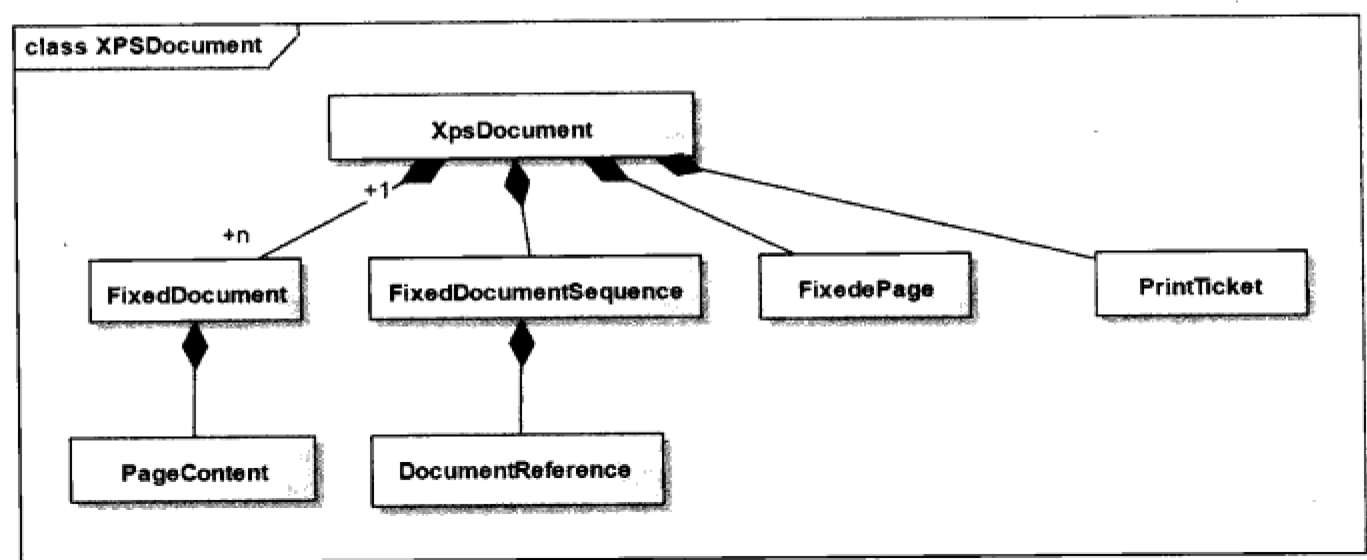


图12-11 XPS文档的组成

XPS文档中含有一个或多个FixedDocuemnt，FixedDocument中含有一个或多个PageContent，PageContent指向具体的xaml 文件。

```
 <FixedDocument xmlns="http://schemas.microsoft.com/xps/2005/06">
    <PageContent Source="../FixedPages/FixedPage_1.xaml" />
    <PageContent Source="../FixedPages/FixedPage_2.xaml" />
   ....
</FixedDocument>
```

FixedDocument 在XPS文档中的次序有FixedDocumentSequence确定：

```
<FixedDocumentSequence
     xmlns="http://schemas.microsoft.com/xps/2005/06">
  <DocumentReference
    Source="../FixedDocuments/FixedDocument_1.xaml" />
       ...
</FixedDocumentSequence>
```

XPS文档中可以含有一个或多个FixedPage，FixedPage里含有真正的内容，FixedPage的格式为：

```
<FixedPage xmlns="http://schemas.microsoft.com/xps/2005/06"
```

```
xmlns:x="http://schemas.microsoft.com/xps/2005/06/resourcedictionary-key"
xml:lang="en-us" Width="816" Height="1056">
<FixedPage.Resources>
     <ResourceDictionary>
          .....
    </ResourceDictionary>
</FixedPage.Resources>
    页面中的内容

</FixedPage>
```

即FixedPage是FrameworkElement的派生类，其中可以含有资源和Visual元素。

XPS文档还可以包括Resources等其他元素，Resources含有字体及图像。PrintTicket中含有打印机的配置信息，如单面打印或双面打印、打印时是否打孔或装订等。

12.5.2 创建XPS文档

在打印XPS文档之前，让我们来看看如何创建XPS文档。创建XPS文档和创建一般文件类似，只是要用到XpsDocumentWriter，下面的GetSaveXpsDocumentWriter方法返回XpsDocumentWriter：

```
private XpsDocumentWriter GetSaveXpsDocumentWriter(string filename)
 {
     File.Delete(filename);
     _xpsDocument = new
          System.Windows.Xps.Packaging.XpsDocument(filename,
          FileAccess.ReadWrite);
     XpsDocumentWriter xpsdw =
         System.Windows.Xps.Packaging.XpsDocument.
         CreateXpsDocumentWriter(_xpsDocument);
     return xpsdw;
 }
```

注意XpsDocumentWriter实例是在XpsDocument类中创建的。XpsDocuemntWriter支持同步和异步两种方法写如XpsDocument，同步Writer()方法有十二个重载，可以写字符串、Visual元素、FixedPage、FixedDocument等；异步Writer-Async方法有二十个重载，一般来说，对于比较大的XPS文档，或打印XPS文档，我们使用异步写方法。

- 在XPS文档中写入Visual对象

让我们先来看用同步写一个Visual对象，即使用XpsDocumentWriter. Writer(Visual)方法来创建XPS文件。由于WPF中UI元素都是从Visual类中派生出来的，因此Writer(Visual)包括大多数实际情形。笔者用唐代诗人李商隐的诗来创建XPS文档：

```
public Visual GetLiShanyinRain()
 {
    string[] poem = { "凄凉宝剑篇，  ", "羁泊欲穷年。  ", "黄叶仍风雨，  ",
      "青楼自管弦。  ", "新知遭薄俗，  ", "旧好隔良缘。  ", "心断新丰酒，  ",
      "消愁斗几千？ " };
    return CreatePoemVisual("风雨",poem,"李商隐");
 }
```

CreatePoemVisual()方法返回一个Canvas对象，其中创建《风雨》一诗的排版：

```
private Visual CreatePoemVisual(string title, string[] poem,
         string author )
{
             Canvas cnv =  new Canvas();
             //在Canvas中加入诗歌
 ....
             return cnv;
}
```

然后就可以调用XpsDocumentWriter中的Writer方法来创建XPS文件了。

```
Visual v = fpc.GetLiShanyinRain();
XpsDocumentWriter xdwSave = GetSaveXpsDocumentWriter(filename);
xdwSave.Write(v);
```

很简单。

- 在XPS文档中写入FixedDocument

由图12-11可见，XPS文档中更常见的是含有FixedDocument。GetLishanyinPoems方法，创建一个FixedDocument，FixedDocument中加入了两个PageContent，每个PageContent中含有一个FixedPage。FixedPage中含有如下的实际内容：

```
public FixedDocument GetLiShanyinPoems()
{
   FixedDocument fixedDocument = GetEmptyFixedDocument();
   PageContent pageContent = new PageContent();
   FixedPage fixedPage = CreateFixedPage(GetLiShanyinRain() );
   ((IAddChild)pageContent).AddChild(fixedPage);
   fixedDocument.Pages.Add(pageContent);

   PageContent pageContent1 = new PageContent();
   FixedPage fixedPage1 = CreateFixedPage(GetLiShanyinNoTitle());
   ((IAddChild)pageContent1).AddChild(fixedPage1);
   fixedDocument.Pages.Add(pageContent1);
   return fixedDocument;
}
```

有了FixedDocuemnt之后，就可以把FixedDocument写入XPS文档中：

```
FixedDocument fd = fpc.GetLiShanyinPoems() ;
XpsDocumentWriter xdwSave=GetSaveXpsDocumentWriter(containerName);
xdwSave.Write(fd);
```

同样，这里用的是XpsDocumentWriter的同步Writer。笔者把有关创建Visual和FixedDocuemnt对象放到了类FixedPageContent类中：

```
using System;
using System.IO;
using System.IO.Packaging;
using System.Collections.Generic;
using System.Windows.Media;
```

```
using System.Windows.Media.Imaging;
using System.Windows;
using System.Windows.Documents;
using System.Windows.Xps.Packaging;
using System.Windows.Controls;
using System.Windows.Shapes;
using System.Windows.Markup;
using System.Printing;
using System.Windows.Documents.Serialization;
using System.Windows.Xps;
namespace Yingbao.Chapter12.XpsDocument
{
    public class FixPageContent
    {
        public Visual GetLiShanyinRain()
        {
            string[] poem = { "凄凉宝剑篇，  ", "羁泊欲穷年。  ", "黄叶仍
风雨，  ", "青楼自管弦。  ", "新知遭薄俗，  ", "旧好隔良缘。  ", "心断新丰酒，  ",
"消愁斗几千？ " };
            return CreatePoemVisual("风雨",poem,"李商隐");
        }

        public Visual GetLiShanyinNoTitle()
        {
            string[] poem = { "昨夜星辰昨夜风，  ", "画楼西畔桂堂东。  ",
"身无彩凤双飞翼，  ","心有灵犀一点通。  ","隔座送钩春酒暖，  ", "分曹射覆蜡灯红。
", "嗟余听鼓应官去，  ", "走马兰台类转蓬？ " };
            return CreatePoemVisual("无题", poem, "李商隐");
        }

        private Visual CreatePoemVisual(string title,
                  string[] poem, string author )
        {
            Canvas cnv =  new Canvas();
            TextBlock tb = new TextBlock();
            tb.Width = 66;
            tb.Height = 37;
            tb.SetValue(Canvas.LeftProperty, 213.0);
            tb.SetValue(Canvas.TopProperty, 27.0);
            tb.FontFamily = new FontFamily("SimHei");
            tb.FontSize = 24;
            ScaleTransform st = new ScaleTransform(2, 1.5);
            tb.RenderTransform = st;
            tb.Text = title;
            cnv.Children.Add(tb);
        //author
            TextBlock tb2 = new TextBlock();
            tb2.Width = 66;
            tb2.Height = 22;
            tb2.SetValue(Canvas.LeftProperty, 300.0);
            tb2.SetValue(Canvas.TopProperty, 80.0);
            tb2.FontSize = 18;
```

```
        tb2.Text = author;
        cnv.Children.Add(tb2);
        // poem
        TextBlock tb1 = new TextBlock();
        tb1.Width = 463;
        tb1.Height = 500;
        tb1.SetValue(Canvas.LeftProperty, 100.0);
        tb1.SetValue(Canvas.TopProperty, 130.0);
        tb1.FontFamily = new FontFamily("SimSun-ExtB");

        tb1.FontSize = 22;
        foreach (string pline in poem)
        {
            tb1.Inlines.Add(pline);
            tb1.Inlines.Add(new LineBreak());
            tb1.Inlines.Add(new LineBreak());
        }
        cnv.Children.Add(tb1);
        Size sz = new Size(8.5 * 96, 11 * 96);
        cnv.Measure(sz);
        cnv.Arrange(new Rect(new Point(), sz));
        cnv.UpdateLayout();
        return cnv;
    }

    private FixedPage CreateFixedPage(Visual vis)
    {
        UIElement content = vis as UIElement;
        FixedPage fixedPage = new FixedPage();
        fixedPage.Background = Brushes.LightYellow;
        FixedPage.SetLeft(content, 0);
        FixedPage.SetTop(content, 0);
        fixedPage.Width = 96 * 8.5;
        fixedPage.Height = 96*11;
        fixedPage.Children.Add( content);
        Size sz = new Size(8.5 * 96, 11 * 96);
        fixedPage.Measure(sz);
        fixedPage.Arrange(new Rect(new Point(), sz));
        fixedPage.UpdateLayout();
        return fixedPage;
    }

    private FixedDocument GetEmptyFixedDocument()
    {
        FixedDocument fixedDocument = new FixedDocument();
        fixedDocument.DocumentPaginator.PageSize =
                new Size(96 * 8.5, 96 * 11);
        return fixedDocument;
    }

    public FixedDocument GetLiShanyinPoems()
    {
        FixedDocument fixedDocument = GetEmptyFixedDocument();
```

```
            PageContent pageContent = new PageContent();
            FixedPage fixedPage =
                    CreateFixedPage(GetLiShanyinRain() );
            ((IAddChild)pageContent).AddChild(fixedPage);
            fixedDocument.Pages.Add(pageContent);
            PageContent pageContent1 = new PageContent();
            FixedPage fixedPage1 =
                    CreateFixedPage(GetLiShanyinNoTitle());
            ((IAddChild)pageContent1).AddChild(fixedPage1);
            fixedDocument.Pages.Add(pageContent1);
            return fixedDocument;
        }
    }
}
```

12.5.3 显示XPS文档

前面提到，IE7可以显示XPS文档，图12-12示出了前面创建的文档在IE7中显示的样子。我们也可以自己创建一个显示XPS文档的程序，使用DocumentViewer类可以很容易地显示XPS文档：

```
private void LoadDocumentViewer(string xpsFilename)
        {
            System.Windows.Xps.Packaging.XpsDocument oldXpsPackage
= _xpsPackage;

            _xpsPackage =     new
System.Windows.Xps.Packaging.XpsDocument(xpsFilename,
                FileAccess.Read, CompressionOption.NotCompressed);

            FixedDocumentSequence fixedDocumentSequence =
                _xpsPackage.GetFixedDocumentSequence();

            docViewer.Document =
                fixedDocumentSequence as IDocumentPaginatorSource;

            if (oldXpsPackage != null)
                oldXpsPackage.Close();

        }
    }
```

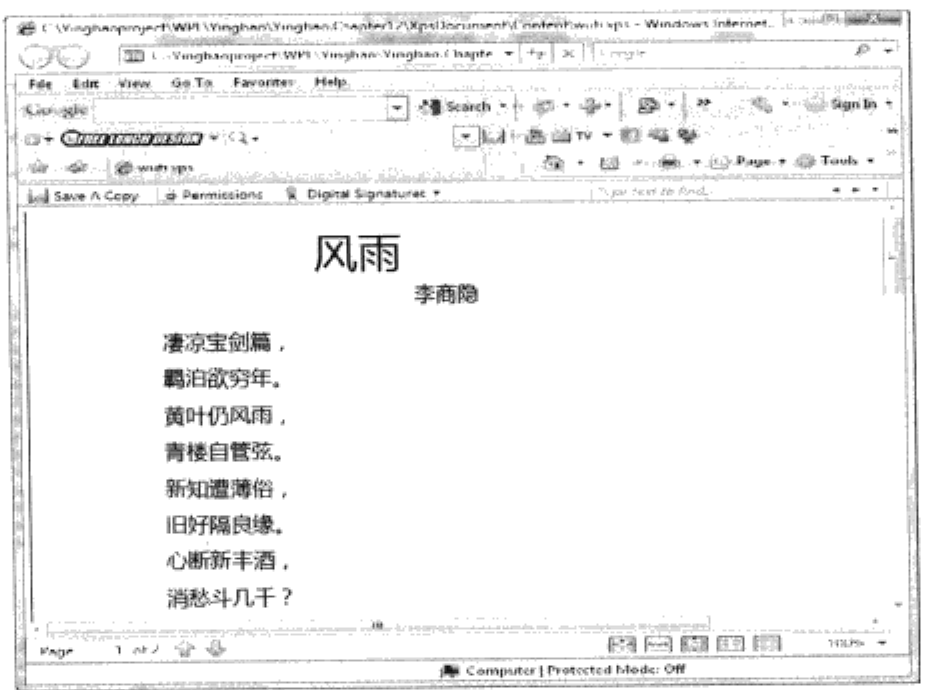


图12-12 在IE7中显示XPS文档

下面的XAML是显示并创建XPS文档的界面：

```
<Window x:Class="Yingbao.Chapter12.XpsDocument.AppWin"
 xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
 xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
 Title="应用XPS 文档" Height="536" Width="814">
    <Grid>
        <Grid.RowDefinitions >
            <RowDefinition Height="2*"/>
            <RowDefinition Height="auto"/>
            <RowDefinition Height="auto"/>
         </Grid.RowDefinitions>
        <Grid.ColumnDefinitions>
            <ColumnDefinition />
            <ColumnDefinition />
            <ColumnDefinition Width="2*" />
        </Grid.ColumnDefinitions>
        <DocumentViewer Name="docViewer" Grid.Row ="0"
             Grid.ColumnSpan="3"/>
        <StackPanel Orientation="Horizontal" Grid.Row="1"
             Grid.Column="2">
           <Button Name="btnLoadDocument" Height="20"
             Margin="0,3,5,2" Click="OnLoadDocument"
             Content="读入XPS文件" />
           <Button Name="btnFixedDocument" Height="20"
             Margin="5,3,5,2" Click="OnCreateFixedDocument"
             Content="创建FixedDocuemnt"/>
           <Button Name="btnVisualDocument" Height="20"
             Margin="5,3,5,2" Click="OnCreateVisual"
             Content="创建SingleVisual"/>
        </StackPanel>
    </Grid>
</Window>
```

后台C#的程序为：

```
using System;
using System.IO;
using System.IO.Packaging;
using System.Collections.Generic;
using System.Windows.Media;
using System.Windows.Media.Imaging;
using System.Windows;
using System.Windows.Documents;
using System.Windows.Xps.Packaging;
using System.Windows.Controls;
using System.Windows.Shapes;
using System.Windows.Markup;
using System.Printing;
using System.Windows.Documents.Serialization;
using System.Windows.Xps;
using Microsoft.Win32;
namespace Yingbao.Chapter12.XpsDocument
{
    public partial class AppWin : Window
    {
        System.Windows.Xps.Packaging.XpsDocument _xpsDocument;
        System.Windows.Xps.Packaging.XpsDocument _xpsPackage;

        public AppWin()
        {
            InitializeComponent();
        }

        void OnLoadDocument(object sender, RoutedEventArgs rea)
        {
            string filename = GetOpenFileName();
            if (filename != string.Empty)
            {
                LoadDocumentViewer(filename);
            }
        }

        void OnCreateFixedDocument(object sender,
              RoutedEventArgs rea)
        {
            string filename = GetSaveFileName();
            if (filename != string.Empty)
            {
                CreateFixedContentDocument(filename);
            }

        }

        void OnCreateVisual(object sender, RoutedEventArgs rea)
        {
            string filename = GetSaveFileName();
```

```
            if (filename != string.Empty)
            {
                FixPageContent fpc = new FixPageContent();
                Visual v = fpc.GetLiShanyinRain();
                XpsDocumentWriter xdwSave =
                        GetSaveXpsDocumentWriter(filename);
                xdwSave.Write(v);
                _xpsDocument.Close();
            }
        }

        private XpsDocumentWriter GetSaveXpsDocumentWriter(
                string filename)
        {
            File.Delete(filename);
            _xpsDocument = new
                System.Windows.Xps.Packaging.XpsDocument(
                 filename, FileAccess.ReadWrite);
            XpsDocumentWriter xpsdw =
                   System.Windows.Xps.Packaging.XpsDocument.
                   CreateXpsDocumentWriter(_xpsDocument);
            return xpsdw;
        }

        private string GetSaveFileName()
        {
            SaveFileDialog sfd = new SaveFileDialog();
            sfd.InitialDirectory =
            @"c:\yingbaoproject\wpf\yingbao\yingbao.Chapter12";
            sfd.Filter = "XPS Files (.xps)|*.xps|All Files
(*.*)|*.*";
            sfd.FilterIndex = 1;
            if (sfd.ShowDialog(this) == true)
            {
                return sfd.FileName;
            }
            return string.Empty;
        }

        private string GetOpenFileName()
        {
            OpenFileDialog sfd = new OpenFileDialog();
            sfd.InitialDirectory =
            @"c:\yingbaoproject\wpf\yingbao\yingbao.Chapter12";
            sfd.Filter = "XPS Files (.xps)|*.xps|All Files
(*.*)|*.*";
            sfd.FilterIndex = 1;
            if (sfd.ShowDialog(this) == true)
            {
                return sfd.FileName;
            }
            return string.Empty;
        }
```

```
        private  void CreateFixedContentDocument(
                string containerName)
         {
             FixPageContent fpc = new FixPageContent();
             FixedDocument fd = fpc.GetLiShanyinPoems() ;
             XpsDocumentWriter xdwSave =
                    GetSaveXpsDocumentWriter(containerName);
             xdwSave.Write(fd);
             _xpsDocument.Close();

         }

         private void LoadDocumentViewer(string xpsFilename)
         {
             System.Windows.Xps.Packaging.XpsDocument
                 oldXpsPackage = _xpsPackage;

             _xpsPackage = new
System.Windows.Xps.Packaging.XpsDocument(xpsFilename,
                FileAccess.Read, CompressionOption.NotCompressed);

             FixedDocumentSequence fixedDocumentSequence =
                _xpsPackage.GetFixedDocumentSequence();

             docViewer.Document =
                fixedDocumentSequence as IDocumentPaginatorSource;

             if (oldXpsPackage != null)
                oldXpsPackage.Close();
         }
     }
}
```

在这段程序里，笔者使用了前面提过的打开文件和存储文件对话框。注意，我们要在项目中引用ReachFramework模块。图12-13示出了使用DocumentViewer显示我们创建的XPS文档。

图12-13 使用DocumentViewer显示XPS文档

我们不仅可以使用XpsDocumentWriter来产生XPS文档，还可以利用XMLWriter直接操作FixedPage中的内容。

12.5.4 打印

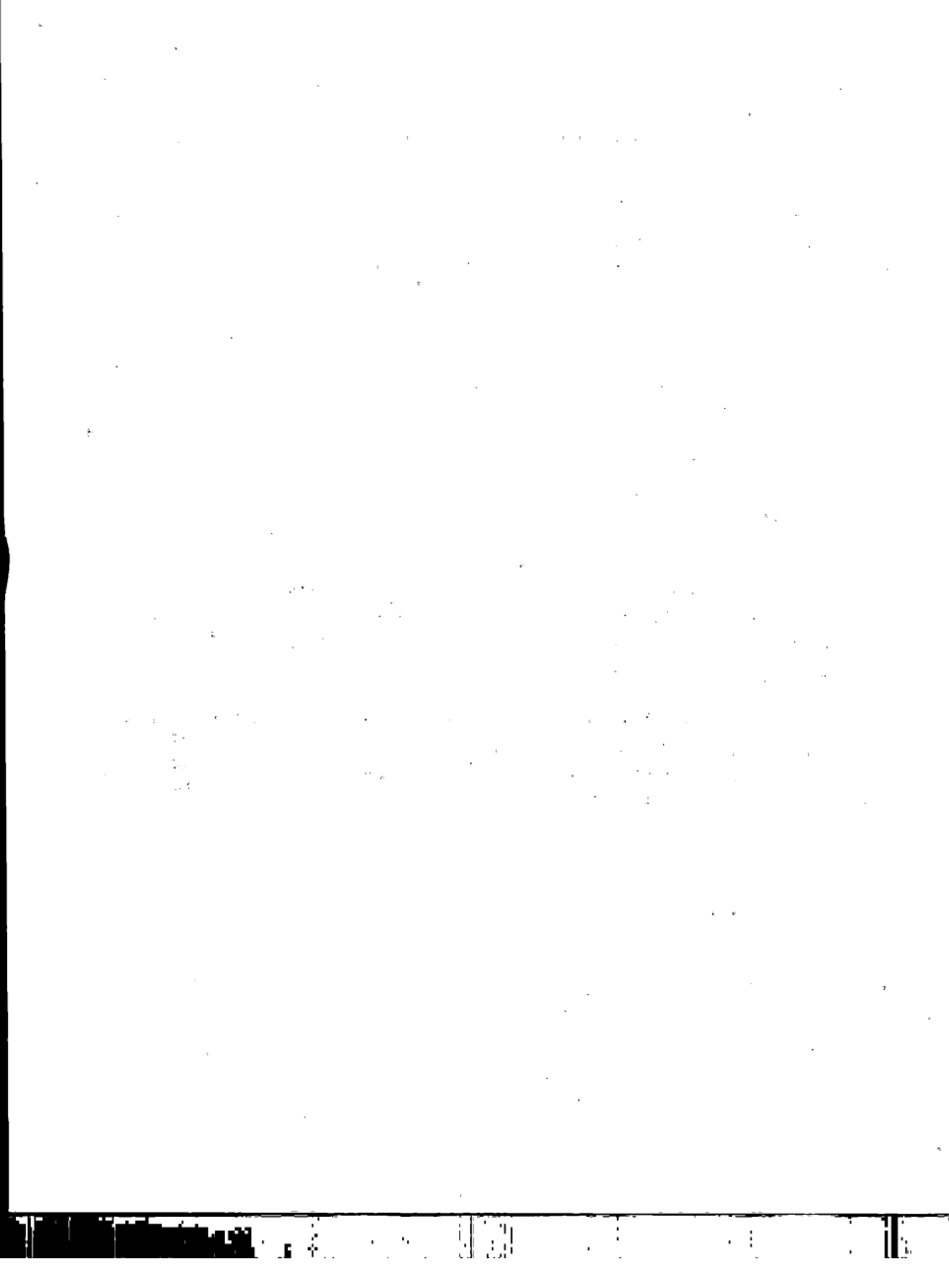
理解了XPS规范及生成XPS文档过程后，打印XPS文档就非常容易。在生成XPS文档时，从XpsDocument类中获得XpsDocumentWriter，在打印时，从PrintQueue类中获取XpsDocumentWriter实例，方法如下：

```
private void PrintDocuemnt(IDocumentPaginatorSource dps)
 {
    PrintDocumentImageableArea area = null;
    XpsDocumentWriter writer =
          PrintQueue.CreateXpsDocumentWriter(ref area);
    if (writer != null)
    {
       DocumentPaginator paginator = dps.DocumentPaginator;
       paginator.PageSize = new Size(area.ExtentWidth,
              area.ExtentHeight);
       writer.Write(paginator);
    }
}
```

由于FixedDocumentSequence和FixedDocument都移植了IDocumentPaginatorSource接口，所以可以通过遍历XpsDocument中的FixedDocuemnt或FixedDocu-mentSequence在打印机上输出XpsDocuemnt文档。在调用PrintQueue.CreateXpsDocumentWriter的时候，会自动弹出打印对话框。

12.6 本章小结

本章介绍了WPF应用程序的两个重要的类：Window和Page，这两个类分别是桌面应用程序和互联网应用程序的最初包容器类。本章还讨论了Window的主要属性、模式及风格，以及模式和无模式对话框。最后本章讨论了XPS文档，XAML的许多功能特别是Silver Light许多思想来源于Flash，所以XPS文档在不远的将来会成为PDF文档的强有力的竞争者。



第三篇　图形与动画

第 13 章　二维图形

第 14 章　图形转换

第 15 章　动画

第13章 二维图形

从本章开始，我们要讨论WPF的图形系统。WPF的图形系统是建立在微软的DirectX技术之上的，而DirectX是专门用来为游戏程序开发人员准备的。对于一般程序员来说，使用DirectX过于繁重；故WPF试图在DirectX和.NET编程模型之间找到一种平衡，一方面充分利用DirectX的图形能力，另一方面方便在.NET平台上编程的程序员，当然需要付出一点运行时速度的代价。

本书的第5章讨论了画刷和画笔，第3章讨论了排版，其中一个排版方式就是画板（Canvas）。本章要讨论WPF中基本的图素。理解这些图素不仅对程序员写程序有用，对于使用Blend等工具的图形界面设计人员来说，也是有用的。

本章集中讨论二维图形的三个方面：一是基本图形（Shape）；二是几何图形（Geometry）；三是几何图形在界面上的展示（Visual）。

13.1 WPF图形系统概述

13.1.1 统一编程模型

过去，程序员在开发用户界面时，通常要面对两个不同的编程模型。一个是控件：这是以视窗为基础的编程模型，视窗中的控件，如按钮、列表框等实际上是一个个小的视窗，对这些小的视窗加上一定的限制，再截获视窗的某些消息——有时候会根据情况产生新的消息——从而形成控件， 这是一个编程模型。另外一个编程模型是：当我们在视窗中画直线、园、矩形等几何图形时，需要自己截获视窗的消息，再调用操作系统提供的绘图函数在视窗上绘制相应的图素；需要自己管理每个图形元素在视窗上的位置、自己管理图形的刷新、选择、移动等等操作。

这两个编程模型过去一直是分开的，我们没法在按钮上加上图形元素（位图除外）。WPF把这两种编程模型统一了起来，前面已经多次出现在WPF控件中加入任意图形的例子，图形元素和控件一样，可以加入到视觉树中。

当把控件放在视窗上时，可以对Windows操作系统发出这样的指令：“把这个控件放在x，y坐标处。”那就不用管了，操作系统会管理控件的刷新、滚动等一系列事件。现在，也可以对图形元素发出相同的指令：“把这个园放在x，y坐标处。”也就不用自己管理刷新、滚动这些操作，WPF会自动完成。当我们把一个图形从一个位置移动到另一个位置时，只要重新设置图形的坐标即可，就不必像过去那样先要擦掉原来的图形，然后再在新的位置绘出相应的图形。在多种图形叠加的时候，管理这种操作并不是一般程序员所能做到的。

下面是一个移动图形的例子。在这个例子中，笔者把一个矩形加入到视窗中：

```
<Window x:Class="Yingbao.Chapter13.MovingShape.MainWindow"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="移动图形" Height="300" Width="300">
  <Canvas Name ="mCanvas">
    <Rectangle Name="moveRect" Canvas.Left ="100" Canvas.Top ="100"
```

```
                Width ="50" Height ="50" Fill ="LightBlue"/>
  </Canvas>
</Window>
```

矩形的初始坐标x=100、y=100，结果如图13-1所示：

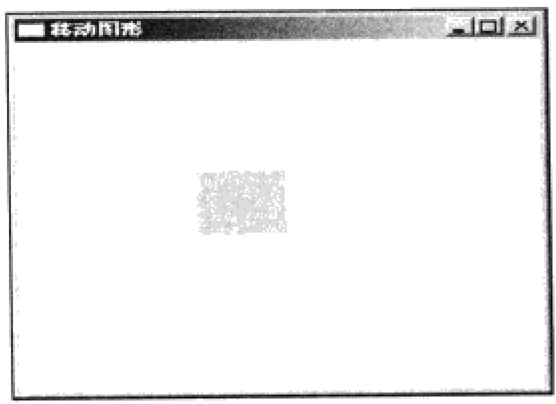


图13-1 在窗口中显示图形

现在要在视窗中移动这个矩形，我们希望使用上下左右四个箭头来向上向下向左向右移动矩形，为此，需要截获键盘消息KeyDown：

```
namespace Yingbao.Chapter13.MovingShape
{
    public partial class MainWindow : System.Windows.Window
    {
        public MainWindow()
        {
            InitializeComponent();
            this.KeyDown += new KeyEventHandler(OnKeyDown);
        }

        void OnKeyDown(object sender, KeyEventArgs e)
        {
            if (e.Key != Key.Down && e.Key != Key.Up &&
                  e.Key != Key.Left  && e.Key != Key.Right)
            {
                return;
            }

            double  x = Convert.ToDouble (moveRect.GetValue(
                           Canvas.LeftProperty) );
            double y = Convert.ToDouble (moveRect.GetValue(
                           Canvas.TopProperty));
            switch (e.Key)
            {
                case Key.Left:
                    if (x > 50)
                    {
                        x -= 50;
                    }
                    break;
                case Key.Right:
                    if (x < 1000)
```

```
                    {
                        x += 50;
                    }
                    break;
                case Key.Up:
                    if (y > 50)
                    {
                        y -= 50;
                    }
                    break;
                case Key.Down:
                    if (y < 800)
                    {
                        y += 50;
                    }
                    break;
            }
            moveRect.SetValue(Canvas.LeftProperty, x);
            moveRect.SetValue(Canvas.TopProperty, y);
        }
    }
}
```

操作时主要调整矩形在Canvas中的左上角坐标即可，非常方便。

13.1.2 坐标系统

在Windows操作系统中进行图形操作时，需要面对多种坐标。现在，只需要用一种坐标，这个坐标的原点为屏幕的左上角。

坐标的单位和显示器的分辨率及显示卡无关，其大小为1/96英寸（一英寸为25.4毫米）。如把矩形的宽度Width设为96，那么矩形在屏幕上占1英寸宽。无论你所用的分辨率是640×480还是1024×768，矩形的宽度总是1英寸。分辨率高低的区别在于显示1英寸需要用多少个点，分辨率越高，显示1英寸所用的点数就越多。

13.1.3 Shape 和Geometry

WPF有两套独立的图形系统，一套是以Shape类为基类；另一套是以Goemetry类为基类。例如绘制矩形，可以用从Shape类派生出来的Rectangle类；也可以用从Geometry类中派生出来的RectangleGeometry类。类似地，Shape类派生出来的其他类，也可以在Geometry下面找到相应的派生类。

为什么WPF要提供两套独立的图形系统呢？这是因为在设计图形系统时，需要兼顾方便编程和提高实时性能两个因素。以Shape类为基类的图形是UIElement，你可以不用做什么工作，就可以像使用控件一样地使用Shape，其代价是实时性能。从Shape类中派生出来的图形元素要使用较多的资源，速度较慢。从Geometry类中派生出来的图形则使用较少的资源，适合组成复杂的图形，如地图等。

下面是这两套图形系统的比较：

- Shape可以在界面上显示自己，而Geometry不能；

- Shape可以直接参与排版，Geometry则不能；
- Shape里含有画刷和画笔，而Geometry内则没有；
- 可以对Shape进行动画，而Geometry不能；
- Geometry可以通过Shape或其他类在界面上显示出来；
- Geometry可以组成复杂的图形。

13.2 Shape及其派生类

图13-2示出了Shape及其派生类的关系。从Shape类中派生出了六个类：Line、Ellipse、Path、Polygon、Polyline和Rectangle。Line用来画直线，Rectangle用来画矩形，Ellipse用来绘制椭圆，Path用来绘制多点连线，Polygon用来绘制多边形，Polyline和Polygon的唯一区别是Polygon的第一点和最后一点是自动连接的。

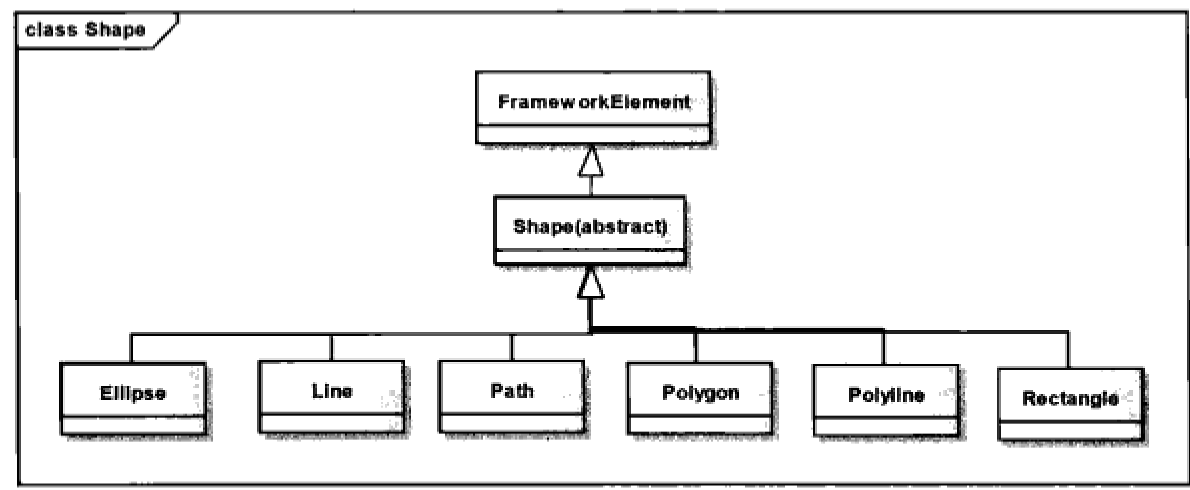


图13-2 Shape及其派生类

Shape类中定义了两个类型为Brush的属性：Stroke和Fill，Stroke用来画线，Fill用来填充图形的内部区域。在默认的情况下，Stroke和Fill的值为null，这时，你在屏幕上看不到所绘制的图形。

- StrokeThickness用来说明线的粗细；Data属性用来说明线所经过的坐标。
- StrokeLineJoin用来说明两条线交接处的方式，与画笔的LineJoin意义一样。
- StrokeMiterLimit，与画笔的MiterLimit意义一样。
- StrokeDashArray，用来说明点画线的方式，其类型为双精度小数，其值相对于画笔的粗细。数组中的第一个值说明点画线的线的长度，第二个值说明点画线中间隔的长度，若取值为1，则说明线的长度或间隔的长度和笔一样粗。
- StrokeDashCap，虚线的端头形状，与画笔的DashCap意义一样。
- StrokeDashOffset，表示点画线的起始位置。
- StrokeStartLineCap，与画笔的StartLineCap相同。
- StrokeEndLineCap，与画笔的EndLineCap相同。

上述和Stroke相关的属性实际上就是画笔的属性，为什么Shape要用这么多的属性，而不用一个

类型为Pen的属性呢？这是为XAML设计的，显然在XAML中设定上述和Stroke相关的属性比设定Pen要方便得多。

Shape内还有一个属性Stretch，这个属性说明在Shape的大小和屏幕上可用的大小不一致时，如何显示图形。这个属性可取的值为：

- None：保持其固有的形状；
- Fill：图形填满可用的区域，不管长宽比例；
- Uniform：图形填满可用区域，但保留原有的长宽比例，填充以长边为准；
- UniformToFill：图形填满可用区域，保留长宽比例，以短边为准，长边会被裁掉。

显然上面所列的这些属性，并不是都适合于派生类的，比如Stretch和Fill属性就不适合直线Line。WPF所采用的策略是若基类中的属性不适合派生类，就自动舍弃。有点“有则改之，无则加勉”的意思。

13.2.1 直线（Line）

直线是最简单的图形，两点决定一条直线，所以只要设定两点，就可以在屏幕上画一条直线：

```
<StackPanel>
   <Line Stroke="Yellow" X1="100" Y1="100" X2="200"  Y2="200"/>
   <Line Stroke="Yellow" X1="120" Y1="500" X2="300" Y2= "250"/>
</StackPanel>
```

设置Stroke相关的属性可以画出各种直线，还可以设置Stretch属性让直线根据窗口的大小自动延伸。

13.2.2 矩形（Rectangle）

矩形的大小是由Height和Width决定的，可以使用各种画刷来填充矩形，其边框则有Stroke属性确定；也可以用Rectangle来画圆角矩形，RadiusX和RadiusY属性就是用来确定矩形的圆角的：

```
<Window x:Class="Yingbao.Chapter13.MovingShapge.Rectangles"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="矩形示例" Height="300" Width="350">
  <StackPanel>
    <Rectangle Width ="300" Height ="50" Fill ="Yellow" Stroke
="Blue" StrokeThickness ="5"/>
    <Rectangle Width ="300" Height ="50" Fill ="Yellow" Stroke
="Blue" StrokeThickness ="5" RadiusX ="10" RadiusY ="10"/>
    <Rectangle Width ="300" Height ="50" Fill ="Yellow" Stroke
="Blue" StrokeThickness ="5" RadiusX ="50" RadiusY ="20"/>
    <Rectangle Width ="300" Height ="50" Fill ="Yellow" Stroke
="Blue" StrokeThickness ="5" RadiusX ="150" RadiusY ="25"/>
  </StackPanel>
</Window>
```

矩形的圆角RadiusX最大可取矩形宽度的一半；RadiusY最大可取高度的一半。当RadiusX和RadiusY取最大值时，矩形自动过度到椭圆（如图12-3所示）

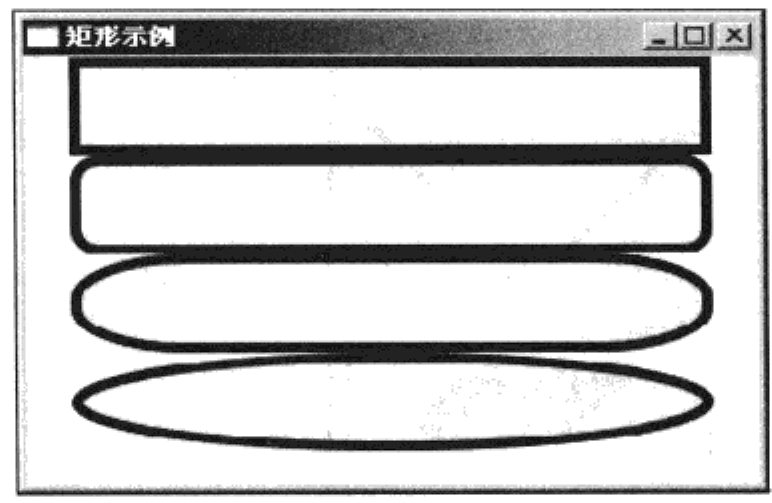


图13-3 各种矩形

13.2.3 椭圆（Ellipse）

椭圆可看成矩形的特例，从上面的例子可以看出，当设置矩形的RadiusX和RadiusY为最大值时，矩形就变成了椭圆。WPF提供Ellipse只是为了编程方便：

```
<Ellipse Width ="300" Height ="50" Fill ="Yellow"
   Stroke ="Blue" StrokeThickness ="5"/>
```

这句XAML画出的图形和上例中最后一个矩形完全相同。

13.2.4 折线（Polyline）

折线由多条直线组成，上一条直线的终点成为下一条直线的起点。直线是折线的特例：

```
<Window x:Class="Yingbao.Chapter13.PolyLines"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="折线和多边形" Height="300" Width="300">
    <StackPanel>
      <Polyline Points ="0,0  100,100" Stroke ="DarkGreen"
                StrokeThickness ="4" />
      <Polyline Points ="0,0  100,100  200,0" Stroke ="DarkGreen"
                StrokeThickness ="4" />
      <Polyline Points ="0,0  100,100  120,50 130,40  180,100  "
                Stroke ="DarkGreen"
                StrokeThickness ="4"  Fill ="Red"/>
    </StackPanel>
</Window>
```

由于需要使用多点来绘制折线，所以我们需要设置Points属性。X和Y之间用逗号隔开，点和点之间用空格隔开。我们可以设置与Stroke有关的属性来画出不同的折线。若我们设置Fill属性，虽然终点没有回到起点，但好像在终点和起点之间有一条直线，然后以所绘折线为边界来填充（如图12-4所示）。

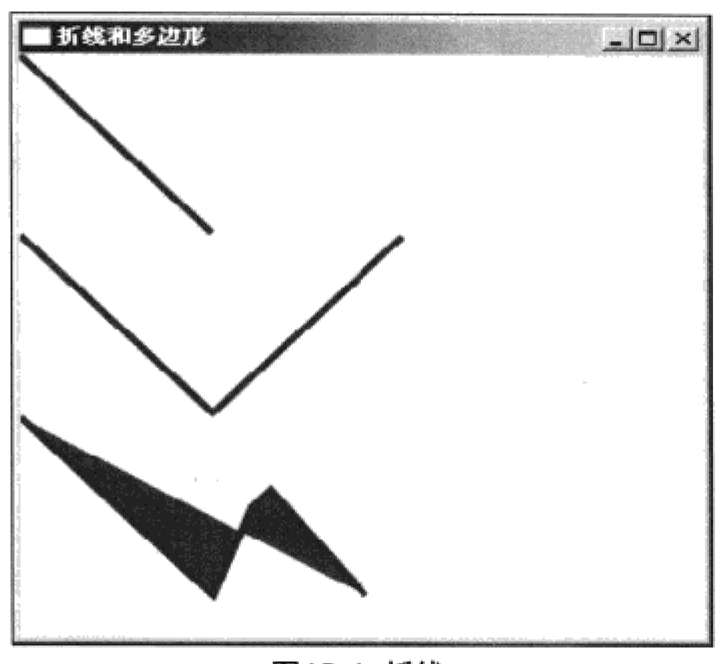


图13-4 折线

13.2.5 多边形（Polygon）

多边形是折线的特例，它自动把折线的终点和起点相连。

13.2.6 填充规则（FillRule）

多重折线和多边形可能会形成多重重叠的情形，WPF使用FillRule属性来确定某个区域是否应该填充。FillRule有两个值：

- EvenOdd（默认值）：从某个区域开始，从里向外画一条直线，如果和偶数个线段相交，那么就不填充该区域；如果和奇数个线段相交，那么就填充该区域。
- NonZero：填充算法比较复杂，需要考虑线段的方向。从某个区域开始，从里向外画一条直线，如果和奇数个线段相交，那么就填充该区域，这和EvenOdd一样。如果和偶数个线段相交，那么要看和假想直线相交的线段的方向；如果同一方向线段的数目和另一方向线段的数目不同，那么就填充该区域。通常情况下，FillRule为NonZero时，某个区域是填充的。

下面的例子示出了两个重叠三角形在两种填充规则下的填充结果：

```
<Window x:Class="Yingbao.Chapter13.FillRule"
 xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="MovingShapge" Height="300" Width="500">
    <Canvas>
      <Canvas.Resources>
        <Style x:Key ="myPolygon">
          <Setter Property ="Polygon.Points"
           Value="0,0   0,100  200,100  100,0 0,100 200,100 0,0 "/>
          <Setter Property ="Polygon.Fill" Value ="Red"/>
          <Setter Property ="Polygon.Stroke" Value ="Yellow" />
          <Setter Property ="Polygon.StrokeThickness" Value ="4" />
        </Style>
      </Canvas.Resources>
      <TextBlock Canvas.Left ="10" Canvas.Top ="5">
```

```
            FillRule=NonZero</TextBlock>
      <TextBlock Canvas.Left ="250" Canvas.Top ="5">
            FillRule=EvenOdd</TextBlock>
      <Polygon Style ="{StaticResource myPolygon}"
         FillRule ="Nonzero" Canvas.Left ="10" Canvas.Top ="50" />
      <Polygon Style ="{StaticResource myPolygon}"
         FillRule ="EvenOdd" Canvas.Left ="250" Canvas.Top ="50" />
   </Canvas>
</Window>
```

由Shape是UIElement，所以可以在资源中定义多边形的风格。图13-5是上面这段XAML的运行结果。

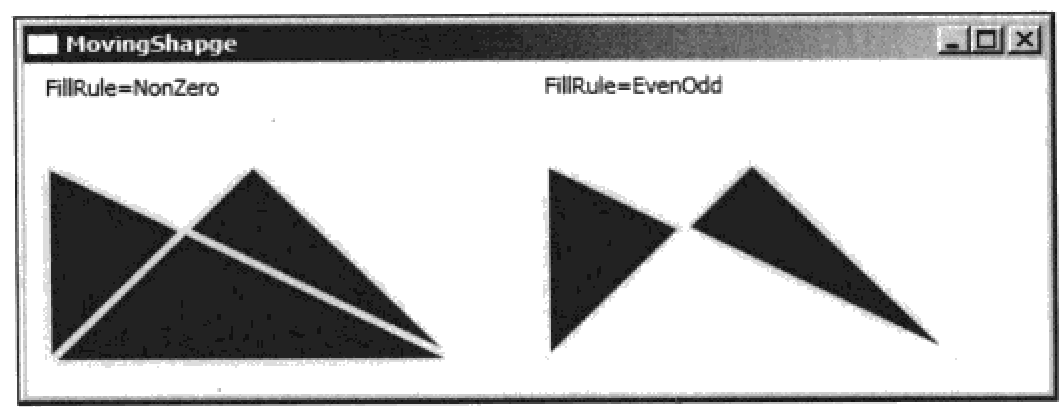


图13-5 NoneZero和EvenOdd的两种填充结果

13.2.7 路径（Path）

Path增加了一个Data属性，其类型为Geometry。由于它和WPF中的另一套图形系统（Geometry）相连，所以功能非常强大。它为Geometry提供了一种嵌入到UIElement中的方式，从而可以对其进行排版，我将结合WPF的几何图形系统对路径加以讨论。

13.3 Geometry及其派生类

WPF提供了另外一套几何图形系统，它不再是UIElement，它所使用的系统资源比Shape要少得多。它只描述几何形状本身，并不负责在界面上显示。它可以通过和路径（Path）中的数据（Data）相连而在界面上展示出来，也可以用WPF的绘制类（13.4节Drawing class）在界面上展示出来。WPF设计这一套图形系统主要是为了提高运行时的速度，像游戏软件、地图软件（GIS)等都需要强大的图形系统，以UIElement为基类的Shape则有失轻灵。

WPF中的几何图形系统以Geometry为基类，派生出七大类（如图13-6所示）：LineGeometry、RectangleGeometry、EllipseGeometry、GeometryGroup、PathGeometry、StreamGeometry和CombinedGeometry。Geomtry是从Animatable里派生出来的，我们可以方便地对几何图形系统实施动画（动画将在第15章讨论）。

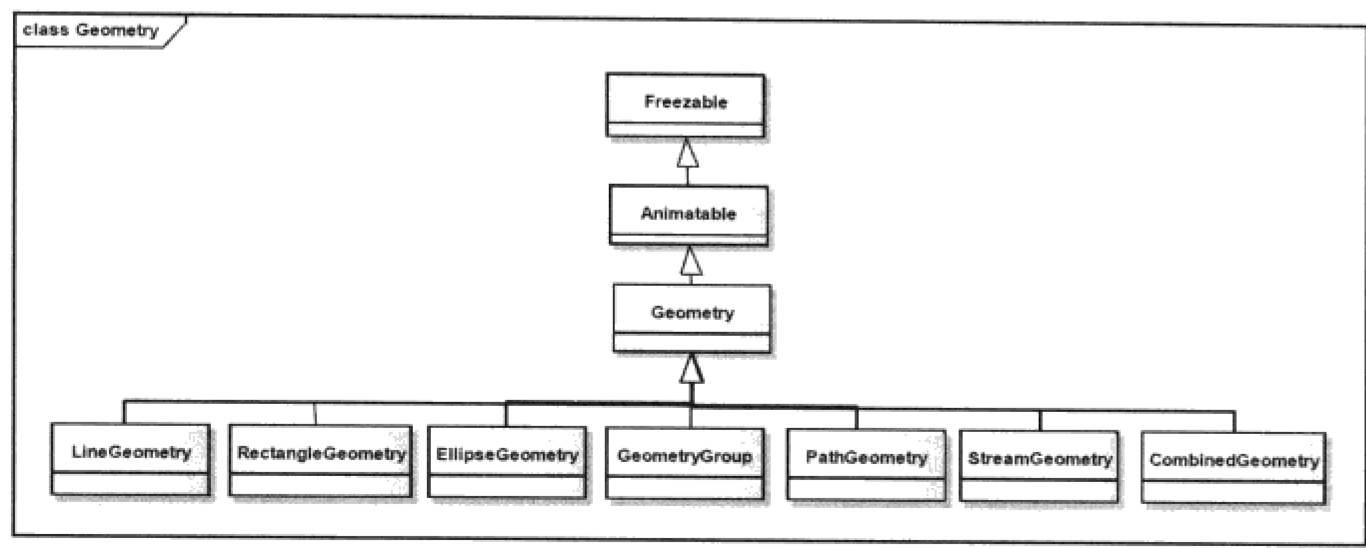


图13-6 WPF的几何图形系统

Geometry类管理图形所占据的区域（Area），Bounds属性类型为Rect、是只读的，其值为图形所占据的最大区域。Geometry中的图形属性，如X、Y坐标，半径、长宽、点等等都是相关属性，其类型为双精度浮点数（Double）。

13.3.1 直线（LineGeometry）

直线几何图形是最简单的，该类定义了StartPoint和EndPoint两个属性。在XAML中定义StartPoint和EndPoint坐标，可以用空格“ ”，也可以用逗号“，”分开，效果是一样的。

StartPoint= “10,10” （使用逗号）

或 StartPoint = “ 10 10 ” （使用空格）

13.3.2 矩形（RectangleGeometry）

矩形几何图形定义了一个Rect属性，定义了矩形的左上角坐标和矩形的大小（宽和高），与Shape里的矩形一样，这里还定义了RadiusX和RadiusY属性，用来说明矩形的圆角。与直线一样，在XAML中定义这些属性可以用逗号也可以用空格分开。

13.3.3 椭圆（EllipseGeometry）

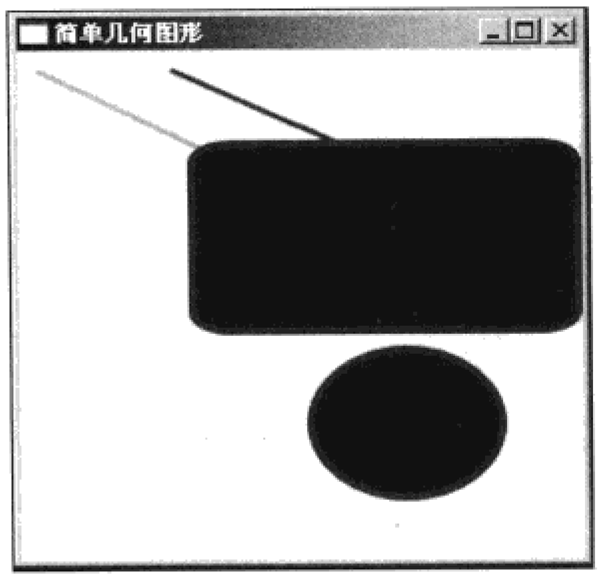
椭圆几何图形和Shape里的椭圆不同，需要给出椭圆的圆心坐标和椭圆在X轴和Y轴方向的半径。同样使用矩形几何图形，也可以得到任意椭圆图形，WPF提供椭圆类，只是为了方便。

下面的这段XAML使用Shape中的路径Path来显示几何图形，直线、矩形和椭圆则作为路径Path的Data：

```
<Window x:Class="Yingbao.Chapter13.SimpleGeometry"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="MovingShapge" Height="300" Width="300">
  <Canvas>
    <Path Stroke ="Cyan" StrokeThickness ="3">
      <Path.Data>
        <LineGeometry StartPoint ="10,10" EndPoint ="150, 80"/>
      </Path.Data>
```

```
    </Path>
    <Path Stroke ="Red" StrokeThickness ="3">
      <Path.Data>
        <LineGeometry StartPoint ="80,10" EndPoint ="230, 80"/>
      </Path.Data>
    </Path>
    <Path Fill ="Blue" Stroke ="Red" StrokeThickness ="4">
      <Path.Data>
        <RectangleGeometry Rect="90,50,200,100" RadiusX ="20"
              RadiusY ="10"/>
      </Path.Data>
    </Path>
    <Path Fill ="Blue" Stroke ="Red" StrokeThickness ="4">
      <Path.Data>
        <EllipseGeometry Center ="200, 200" RadiusX ="50"
             RadiusY ="40"/>
      </Path.Data>
    </Path>
  </Canvas>
</Window>
```

上面这段XAML的运行结果如图13-7所示。



使用Path.Data的局限：

Path.Data只能和一个Geometry相连

图13-7 直线、矩形和椭圆的几何图形

13.3.4 几何图形组（GeometryGroup）

我们知道，使用Path.Data类显示几何图形有一个局限，即Data属性只能和一个几何图形Geometry相连。克服这一局限的方法就是使用GeomtryGroup，GeometryGroup从Geometry中派生出来，其中含有一个属性：Children，这个属性的类型为GeometryCollection, 即为几何图形的集合，其中可以加入任意多个Geometry，举个例子如下：

```
<Window x:Class="Yingbao.Chapter13.UsingGeometryGroup"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="使用GeometryGroup" Height="300" Width="300">
    <Canvas>
```

```
        <Path Fill ="Blue" Stroke ="Red" StrokeThickness ="5">
          <Path.Data>
              <GeometryGroup FillRule ="Nonzero">
                <EllipseGeometry Center ="150,150" RadiusX ="100"
                    RadiusY ="100"/>
                <RectangleGeometry Rect="10,10, 100,200"/>
            </GeometryGroup>
          </Path.Data>
        </Path>
    </Canvas>
</Window>
```

在这个例子中，Path.Data不再和单个几何图形Geometry相连，而是和GeometryGroup相连。GeometryGroup中可以加入任意多个几何图形，笔者在这里加入了椭圆和矩形两个图形。

当多个图形叠加时，GeometryGroup中的FillRule定义填充规则，其意义和Shape中的多边形填充规则是一样的，默认为EvenOdd，笔者这里把FillRule设为NonZero，所以，两个图形的叠加部分是填充的（如图13-8所示）。

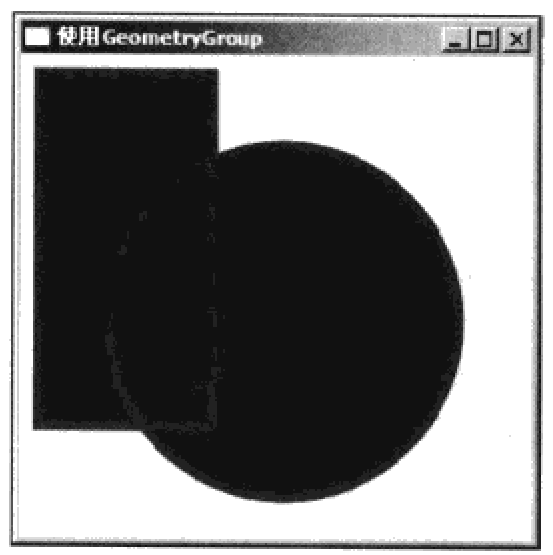


图13-8 使用GeometryGroup显示多个几何图形

13.3.5 合并图形（CombinedGeometry）

有时候我们需要把两个几何图形按照一定的方式合并起来，以达到特殊的效果。合并图形（CombinedGeometry）就是为此设计的。这个类中定义了下面三个属性：

- Geometry1：用于合并的第一个图形，类型为Geometry；
- Geometry2：用于合并的第二个图形，类型为Geometry；
- GeometryCombineMode：这个属性说明两个几何图形的合并方式，可取4个值：Union、Intersect、Xor和Exclude。Union取两个几何图形的最大区域；Intersect取两个几何图形所共有的部分；Xor取两个几何图形的非共有部分；Exclude去第一个图形中未被第二个几何图形覆盖的部分。

下面是不同合并模式下，两个几何图形的合并结果，这段XAML的运行结果如图13-9所示：

```
<Window x:Class="Yingbao.Chapter13.CombineGeometryMode"
```

```
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
   xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
   Title="不同模式下合并几何图形" Height="300" Width="300">
 <Canvas>
   <TextBlock Canvas.Left ="10"  Canvas.Top="10">Union</TextBlock>
   <TextBlock Canvas.Left ="120"
        Canvas.Top="10">Intersect</TextBlock>
   <TextBlock Canvas.Left ="250"  Canvas.Top="10">Xor</TextBlock>
   <TextBlock Canvas.Left ="340"
        Canvas.Top="10">Exclude</TextBlock>
   <Path Stroke="Red" StrokeThickness="3" Fill="Blue"
        Canvas.Left="10" Canvas.Top="60">
     <Path.Data>
       <CombinedGeometry  GeometryCombineMode="Union" >
         <CombinedGeometry.Geometry1>
              <EllipseGeometry Center="45,80" RadiusX="30"
                   RadiusY="40"/>
         </CombinedGeometry.Geometry1>
         <CombinedGeometry.Geometry2>
           <EllipseGeometry Center="55,80" RadiusX="30"
                   RadiusY="40"/>
         </CombinedGeometry.Geometry2>
       </CombinedGeometry>
     </Path.Data>
   </Path>
   <Path Stroke="Red" StrokeThickness="3" Fill="Blue"
                 Canvas.Left="110" Canvas.Top="60">
     <Path.Data>
       <CombinedGeometry  GeometryCombineMode="Intersect" >
         <CombinedGeometry.Geometry1>
           <EllipseGeometry Center="45,80" RadiusX="30"
                  RadiusY="40"/>
         </CombinedGeometry.Geometry1>
         <CombinedGeometry.Geometry2>
           <EllipseGeometry Center="55,80" RadiusX="30"
                 RadiusY="40"/>
         </CombinedGeometry.Geometry2>
       </CombinedGeometry>
     </Path.Data>
   </Path>
   <Path Stroke="Red" StrokeThickness="3" Fill="Blue"
                 Canvas.Left="210" Canvas.Top="60">
     <Path.Data>
       <CombinedGeometry  GeometryCombineMode="Xor" >
         <CombinedGeometry.Geometry1>
           <EllipseGeometry Center="45,80" RadiusX="30"
                 RadiusY="40"/>
         </CombinedGeometry.Geometry1>
         <CombinedGeometry.Geometry2>
           <EllipseGeometry Center="55,80" RadiusX="30"
                 RadiusY="40"/>
         </CombinedGeometry.Geometry2>
       </CombinedGeometry>
```

```
      </Path.Data>
    </Path>
    <Path Stroke="Red" StrokeThickness="3" Fill="Blue"
          Canvas.Left="310" Canvas.Top="60">
      <Path.Data>
        <CombinedGeometry  GeometryCombineMode="Exclude" >
          <CombinedGeometry.Geometry1>
            <EllipseGeometry Center="45,80" RadiusX="30"
                RadiusY="40"/>
          </CombinedGeometry.Geometry1>
          <CombinedGeometry.Geometry2>
            <EllipseGeometry Center="55,80" RadiusX="30"
                RadiusY="40"/>
          </CombinedGeometry.Geometry2>
        </CombinedGeometry>
      </Path.Data>
    </Path>
  </Canvas>
</Window>
```

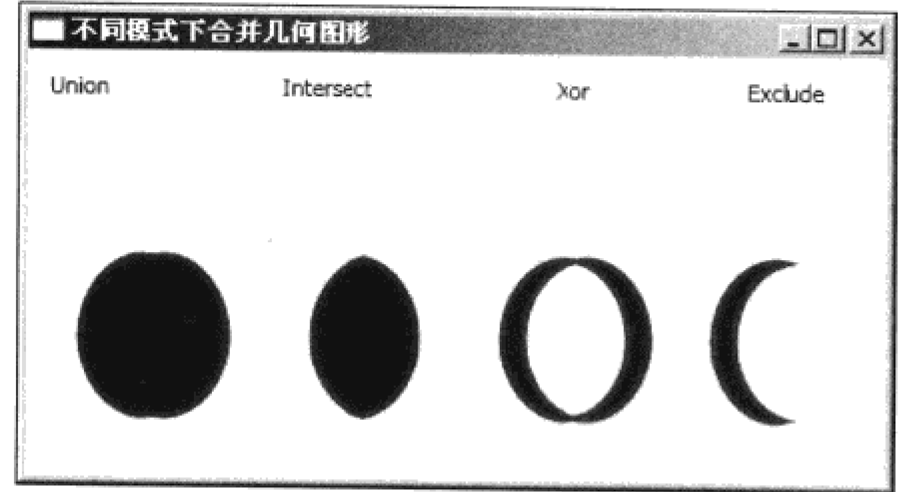


图13-9 两个椭圆在不同合并模式下的合并结果

虽然CombinedGeometry只能合并两个几何图形，但实际上，每个几何图形又可以是由另外两个几何图形合并而来，如此循环，便可以使用多重合并来达到各种效果。

13.3.6 几何路径（PathGeometry）

WPF图形最强大的功能在于几何路径，我们可以利用几何路径产生任意图形。PathGeometry类里定义了两个属性，一个是Figures，其类型为PathFigureCollection；另一个是FillRule，定义图形叠加时的填充规则。

图13-10示出了几何路径（PathGeometry）中所使用的类UML继承图，从Animatable类中派生出4个类：PathFigure、PathFigureCollection、PathSegment和PathSegmentCollection。其中PathFigureCollection和PathSegmentCollection为集合类。

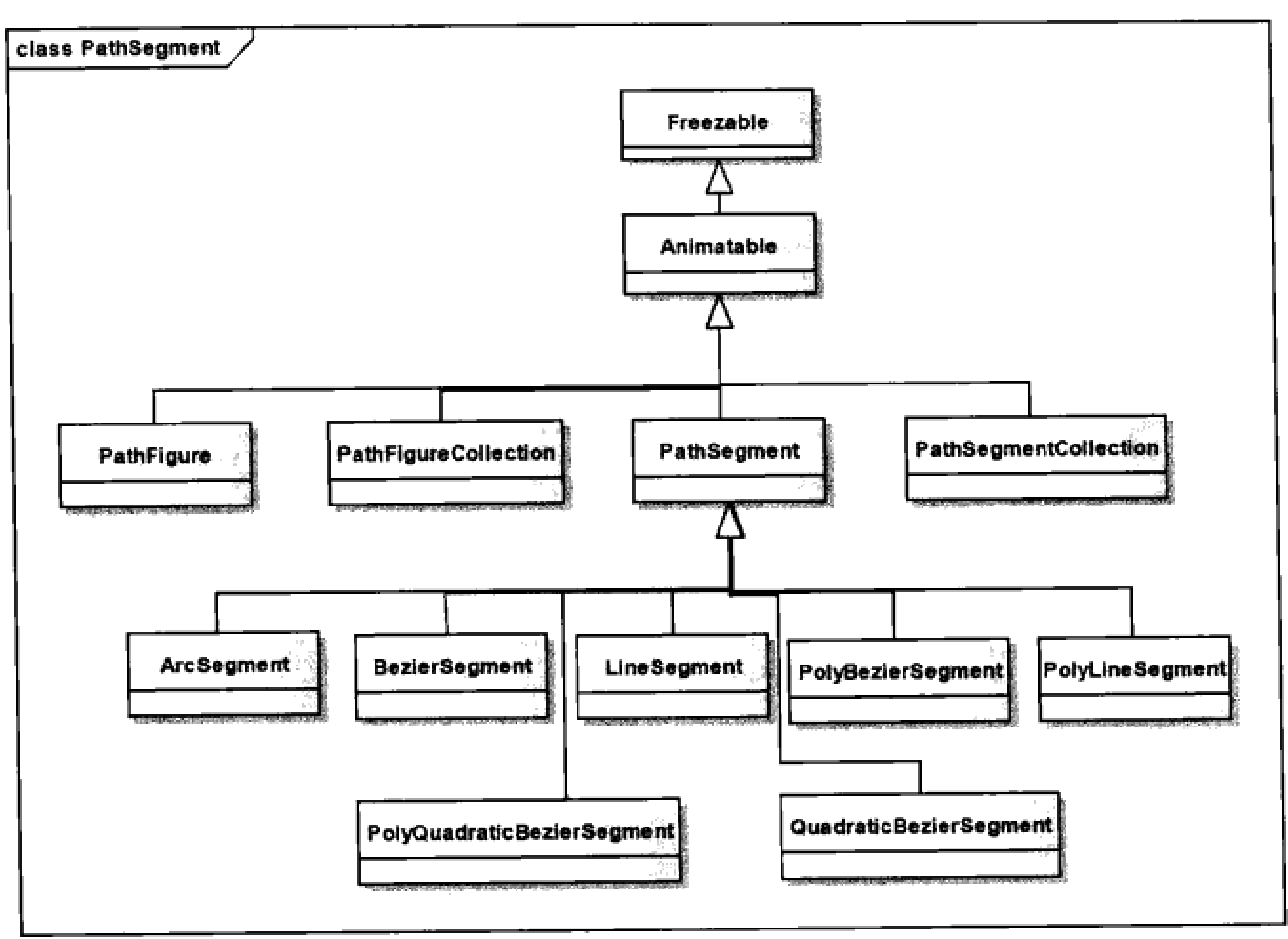


图13-10 几何路径（PathGemotry）中所用的类继承图

PathFigure类是封闭（Sealed）的，即不能从该类中派生出新的类。PathFigure中定义了两个布尔属性：IsClosed和IsFilled。若IsClosed设为True，PathFigure自动把其中的线段封闭起来，默认值为False；若IsFilled设为True，表示PathFigure所围的内部区域要用画刷填充，默认值为True。需要说明IsFilled和IsClosed不相关，换句话说，即使图形是不封闭的，仍然可以用画刷填充内部区域，只是有一个边是开放的，这有点像数学里的开区间的概念。如果以个图形首尾相接，就可以说这个图形是封闭图形。

PathFigure中的Segments属性为PathSegmentCollection类型，在其中存放PathSegment。StartPoint属性表示图形开始的位置。

图13-11示出了几何路径中使用的各个类间的包容关系。由图13-11可见，PathGeometry包含PathFigureCollection（通过属性Figures），PathFigureCollection包含多个PathFigure，PathFigure包含PathSegmentCollection（通过属性Segments)，其中又可以放入多个PathSegment。弄清楚这种包容关系，便可以很容易地写出XAML。

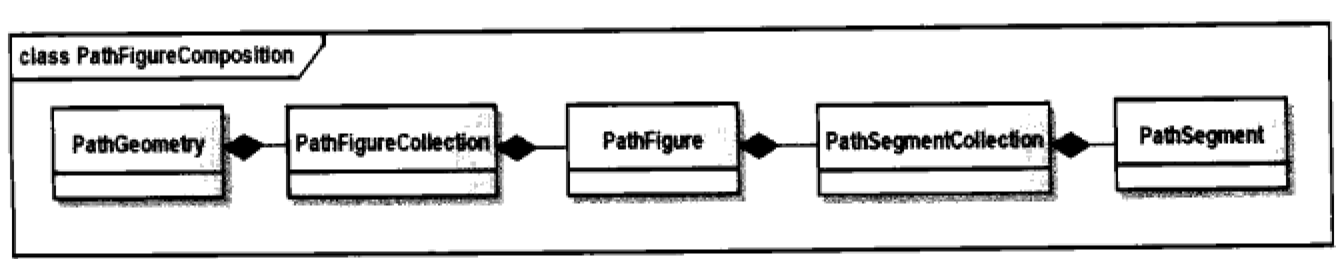


图13-11 几何路径中类间的包容关系

13.3.7 分段路径（PathSegment）

分段路径是一个抽象类，为其他各种分段路径的基类。PathSegment中定义了两个布尔变量：

- IsSmoothJoin：当设为True时，在两个分段路径的交接处要用画笔画出交角。这样当画笔较粗时，两个分段在路径间可平滑过渡；
- IsStroked：当设为True时，在绘制分段路径时，会加上画笔所占的区域，画笔所增加的区域对测试鼠标单击（Hittesting）区域有影响。

13.3.8 弧线（ArcSegment）

顾名思义，ArcSegment是用来画圆弧的。要在平面上画一条弧线，需要给定弧线的起点和终点，X方向和Y方向的半径（Size），即画的是大圆弧还是小圆弧。

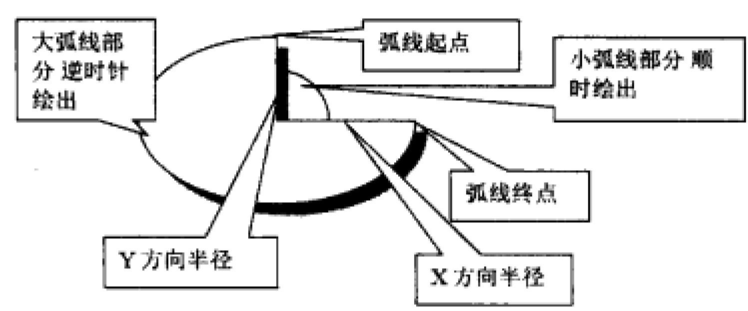


图13-12 弧线

图13-12给出了圆弧的起点、终点、X方向半径和Y方向半径，我们可以画出四条弧线。这是因为给定一条弦，可以画出两个圆，一个圆可分出两条弧线，所以有四条弧线。

为了唯一确定一条弧线，ArcSegment增加了下面几个相关属性：

- Point：弧线的终点。ArcSegment中只定义终点，其起点在PathFigure中（即上一条分段路径结束的位置）。
- IsLargeArc：若设为True，是指较长的那条弧线，否则，是指较短的那条弧线。设定这个属性后，四条弧线变成了两条。
- SweepDirection：可取ClockWise（顺时针）和CounterClockWise（逆时针）两个值。
- RotateAngle：表示圆弧沿着X方向旋转的角度（逆时针旋转时，角度为正，反之为负）。

```
<Window x:Class="Yingbao.Chapter13.ArcsSegment"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="弧线" Height="300" Width="328">
  <StackPanel>
    <StackPanel  Orientation ="Horizontal" Margin ="20"
          Height ="100">
    <Path Stroke ="Blue" StrokeThickness ="3">
      <Path.Data>
        <PathGeometry>
          <PathFigure StartPoint ="50,0">
            <ArcSegment Point ="80,50"  Size ="50,30"
                 IsLargeArc ="True" SweepDirection ="Clockwise"/>
```

```
        </PathFigure>
      </PathGeometry>
    </Path.Data>
  </Path>
  <Path Stroke ="Blue" StrokeThickness ="3">
    <Path.Data>
      <PathGeometry>
        <PathFigure StartPoint ="50,0">
          <ArcSegment Point ="80,50"  Size ="50,30"
          IsLargeArc ="True" SweepDirection ="Counterclockwise"/>
        </PathFigure>
      </PathGeometry>
    </Path.Data>
  </Path>
  <Path Stroke ="Blue" StrokeThickness ="3">
    <Path.Data>
      <PathGeometry>
        <PathFigure StartPoint ="50,0">
          <ArcSegment Point ="80,50"  Size ="50,30"
              IsLargeArc ="False" SweepDirection ="Clockwise"/>
        </PathFigure>
      </PathGeometry>
    </Path.Data>
  </Path>
  <Path Stroke ="Blue" StrokeThickness ="3">
    <Path.Data>
      <PathGeometry>
        <PathFigure StartPoint ="50,0">
          <ArcSegment Point ="80,50"  Size ="50,30"
         IsLargeArc ="False" SweepDirection ="Counterclockwise"/>
        </PathFigure>
      </PathGeometry>
    </Path.Data>
  </Path>
</StackPanel>
    <StackPanel  Orientation ="Horizontal"  Margin ="20"
            Height ="100" >
      <Path Stroke ="Blue" StrokeThickness ="3">
        <Path.Data>
          <PathGeometry>
            <PathFigure StartPoint ="50,0">
              <ArcSegment Point ="80,50"  Size ="50,30"
                  IsLargeArc ="False" SweepDirection ="Clockwise"
                  RotationAngle ="30"/>
            </PathFigure>
          </PathGeometry>
        </Path.Data>
      </Path>
      <Path Stroke ="Blue" StrokeThickness ="3">
        <Path.Data>
          <PathGeometry>
            <PathFigure StartPoint ="50,0">
              <ArcSegment Point ="80,50"  Size ="50,30"
```

```
                  IsLargeArc ="False" SweepDirection ="Clockwise"
                  RotationAngle ="-30"/>
            </PathFigure>
          </PathGeometry>
        </Path.Data>
      </Path>
      <Path Stroke ="Blue" StrokeThickness ="3">
        <Path.Data>
          <PathGeometry>
            <PathFigure StartPoint ="50,0">
              <ArcSegment Point ="80,50"  Size ="50,30"
                 IsLargeArc ="False" SweepDirection ="Clockwise"
                 RotationAngle ="120"/>
            </PathFigure>
          </PathGeometry>
        </Path.Data>
      </Path>
    </StackPanel>
   </StackPanel>
</Window>
```

上面这段XAML示出了通过两点的弦所作的四条弧线，程序的后面是对弧线旋转后的效果（如图13-13所示）。

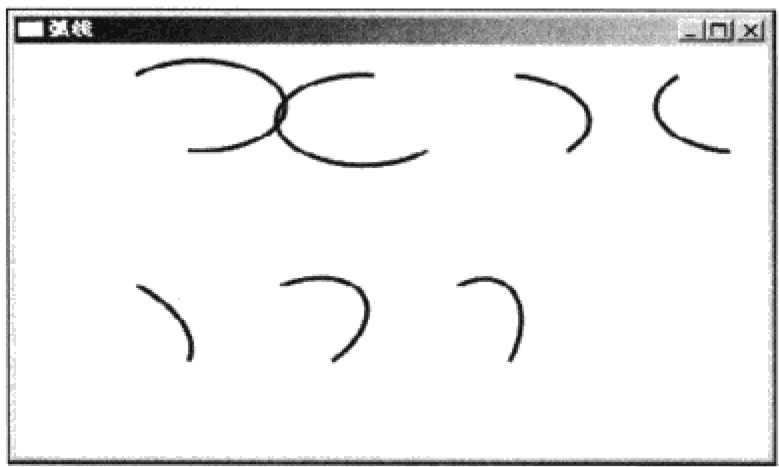


图13-13 过相同起点和终点画出的四条弧线（上半部分）和旋转后的效果

13.3.9 直线段（LineSegment）

直线段（LineSegment)中定义了一个相关属性：终点（Point）。直线的起点在PathFigure中定义。所以下面这段XAML在屏幕上绘制一条从点（10，10）到点（200，200）的一条线段：

```
<Path Stroke ="Red" StrokeThickness ="3">
  <Path.Data>
    <PathGeometry>
      <PathFigure StartPoint ="10,10">
        <LineSegment Point ="200,200" />
      </PathFigure>
    </PathGeometry>
  </Path.Data>
</Path>
```

13.3.10 折线段（PolyLineSegment）

折线段（PolyLineSegment）中有一个属性Points，它是一个点的集合，其中可以含有任意多个点，每个点的位置都是折线的转折处。起点可以是PathFigure中的StartPoint，也可以是上一个图形的终点。

```
<Path Stroke ="Red" StrokeThickness ="3">
  <Path.Data>
    <PathGeometry>
      <PathFigure StartPoint ="10,10">
        <PolyLineSegment Points ="200,200  100,180" />
      </PathFigure>
    </PathGeometry>
  </Path.Data>
</Path>
```

13.3.11 柏之线（BezierSegment）

柏之线是由法国工程师Pierre Etienne Bezier（参见维基百科有关柏之的条目http://en.wikipedia.org/wiki）首先发明的。在20世纪60年代，柏之开始研究CAD/ CAM系统，该系统需要对各种曲线进行数学表达，他所开发的表达曲线的数学方法，后来广泛应用在计算机的各种图形系统中，故称之为柏之方法。

作为程序员来说，一般不需要弄明白柏之曲线的数学表达，只需要知道柏之线有四个几何上的点确定，其中两个点是起点和终点；例外两个点是控制点，控制点用来控制曲线的弯曲程度及弯曲的方向。

```
<Window x:Class="Yingbao.Chapter13.BezierSegmentWindow"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="柏之线" Height="200" Width="500">
  <Canvas >
   <Path Stroke ="Green" StrokeThickness ="1">
     <Path.Data>
       <PathGeometry  >
         <PathFigure StartPoint ="10,100">
           <BezierSegment Point1 ="10,50" Point2 ="90,150"
             Point3 ="100, 100"/>
           <BezierSegment Point1 ="90,50" Point2 ="200,50"
             Point3 ="200, 100"/>
           <BezierSegment Point1 ="250,100" Point2 ="280,100"
             Point3 ="300, 100"/>
           <BezierSegment Point1 ="450,50" Point2 ="280,30"
             Point3 ="400, 100"/>
         </PathFigure>
       </PathGeometry>
     </Path.Data>
   </Path>
  </Canvas>
</Window>
```

柏之线中的Point1和Point2为控制点，Point3为终点。上面的四段柏之线如图13-14所示。

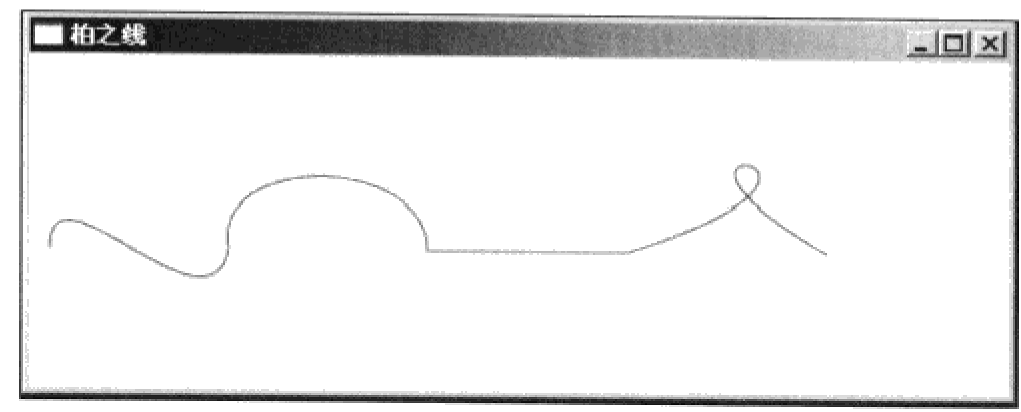


图13-14 柏之线

柏之线不仅可以产生圆、椭圆，而且可以产生直线。

13.3.12 多段柏之线（PolyBezierSegment）

在图13-14所示的例子中，笔者用了四个柏之线段，其实对于多个在一起的柏之线，可以用多段柏之线来构图，使得XAML更加简单。

```
<Path Stroke ="Green" StrokeThickness ="1" Fill ="Red">
  <Path.Data>
    <PathGeometry  >
      <PathFigure StartPoint ="10,100" IsClosed ="True"
           IsFilled ="true">
        <PolyBezierSegment Points="10,50 90,150 100,100  90,50
           200,50 200,100  250,100 280,100
          300,100 450,50 280,30 400, 100"/>
      </PathFigure>
    </PathGeometry>
  </Path.Data>
</Path>
```

起点由PathFigure中的StartPoint确定，接着的两点是控制点，然后是第一段柏之线的终点，该终点成为下一段柏之线的起点……如此循环下去，用多段柏之线产生的填充图形如图13-15所示。

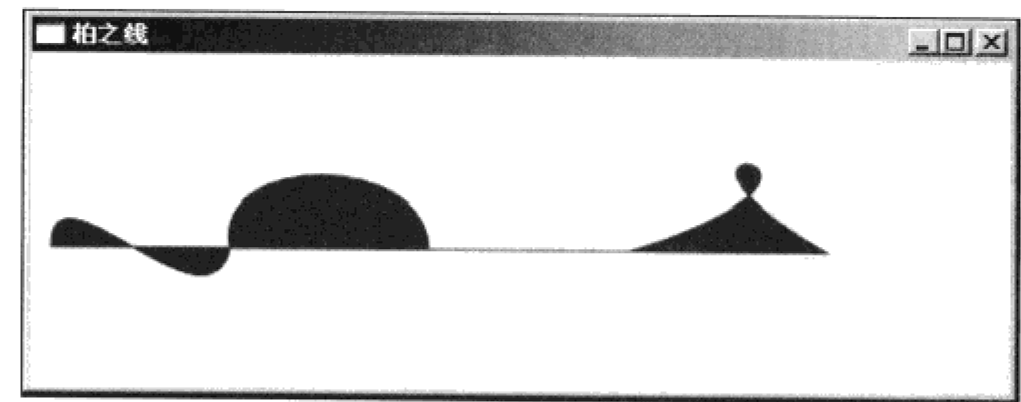


图13-15 用多段柏之线产生的填充图形

13.3.13 二次柏之线（QuadraticBezierSegment）

二次柏之线是一种简化的柏之线，控制点从两点缩略为一点。当然，二次柏之线不能产生柏之线的那种交叉互纽的效果：

```
<Canvas>
  <Path Stroke ="Green" StrokeThickness ="1" Fill ="Red">
    <Path.Data>
      <PathGeometry  >
        <PathFigure StartPoint ="10,100" IsClosed ="True"
               IsFilled ="True">
          <QuadraticBezierSegment Point1 ="45,50"
               Point2 ="100,100"/>
          <QuadraticBezierSegment Point1 ="145,150"
               Point2 ="200,100"/>
          <QuadraticBezierSegment Point1 ="45,30"
               Point2 ="300,100"/>
        </PathFigure>
      </PathGeometry>
    </Path.Data>
  </Path>
</Canvas>
```

这段XAML的运行结果如图13-16所示。

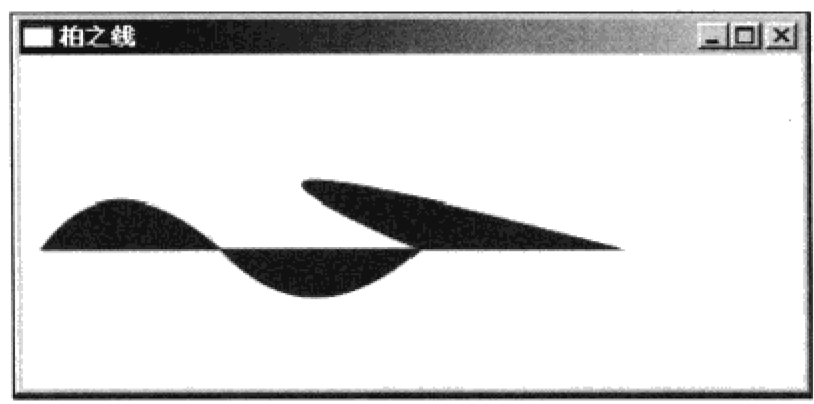


图13-16 二次柏之线

13.3.14 多段二次柏之线（PolyQuadraticBezierSegment）

我们自然会想到，对于图13-16中所用的多个QuadraticBezierSegement，是否能够和柏之线一样进行合并？合并的结果就是多段二次柏之线了，所以可以把上面的XAML用PolyQuadraticBezierSegment改写如下：

```
<Path Stroke ="Green" StrokeThickness ="1" Fill ="Red">
   <Path.Data>
     <PathGeometry  >
       <PathFigure StartPoint ="10,100" IsClosed ="True"
           IsFilled ="True">
         <PolyQuadraticBezierSegment Points ="45,50 100,100
           145,150 200,100 45,30  300,100"/>
       </PathFigure>
     </PathGeometry>
   </Path.Data>
</Path>
```

其产生的图形和图13-16完全相同。

13.3.15 迷你绘图语言

在绘制复杂的图形时，我们希望有一种更简洁的写法，这就是描述路径的迷你绘图语言，迷你语言的命令集如表13-1所示。

表13-1 几何路径命令集

命 令	名 称	功 能
F n	FillRule	n=0: EvenOdd; n=1: NonZero; 必须位于字符串的起始处
M x, y	移动（Move）	把PathFigure的StartPoint设到（x,y）处
Z	关闭图形	结束一个PathFigure，并把IsClosed属性设为true。如果不希望所绘的几何图形是封闭的，可以略去这一命令
H x	水平线	保持Y坐标不变
V y	垂直线	保持X坐标不变
L x,y	任意直线	从某点画到（x,y)处
A rx,ry d f1 f2 x,y	弧线	rx: X方向半径 ry: Y方向半径 (x,y) 弦的一个端点 d: 旋转的角度 f1: IsLargeArc设为True f2: ClockWise顺时针
Cx1,y1,x2,y2,x,y	柏之线	画柏之线到（x,y）处，（x1,y1）和（x2,y2）为两个控制点
S x2,y2 x, y	平滑柏之线	画柏之线到（x,y）处，(x1,y1)和（x2,y2)为控制点，其中（x1,y1)是自动计算出来的。S是平滑（Smooth）的缩写
Qx1,y1 x,y	二次柏之线	画二次柏之线到（x,y）处，（x1,y1）为控制点
小写命令	上面的所有命令都有小写字母版本，当使用小写字母时，所有的坐标都相对于当前的坐标	

下面以电力系统真实的电压曲线图举例，说明如何使用迷你绘图语言来生成坐标图。在电力系统中变压器进线电压或出线电压是一个重要的指标，因此，调度人员常常要实时监控电压的变化。如果电压过低，可能需要拉闸限电，如果电压过高，则要进行调压。监控电压是在一天24小时内进行的，所以，我们取横坐标为时间（小时），纵坐标为电压。本例中假定监视的电压为220V。下面是XAML：

```
<Window x:Class="Yingbao.Chapter13.VoltageWindow"
 xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="VoltageCurve" Height="300" Width="700">
    <Canvas>
      <Path Stroke ="Blue" StrokeThickness ="1" Data="M30,10
        L630,10 M30,30  L630,30  M30,50  L630,50 M30,70  L630,70
        M30,90  L630,90 M30,110  L630,110 M30,130  L630,130
        M30,150  L630,150 M30,170  L630,170 M30,190  L630,190
        M30,210  L630,210 M30,10  L30,210  M55,10 L55,210 M80,10
        L80,210 M105,10 L105,210 M130,10 L130,210 M155,10 L155,210
        M180,10 L180,210 M205,10 L205,210 M230,10 L230,210 M255,10
        L255,210 M280,10 L280,210 M305,10 L305,210 M330,10
        L330,210 M355,10 L355,210 M380,10 L380,210 M405,10
        L405,210 M430,10 L430,210 M455,10 L455,210 M480,10
        L480,210 M505,10 L505,210 M530,10 L530,210 M555,10
```

```
            L555,210 M580,10 L580,210 M605,10 L605,210 M630,10
            L630,210"/>
        <TextBlock Canvas.Left ="10" Canvas.Top ="210"
            Foreground ="Red">150 </TextBlock>
        <TextBlock Canvas.Left ="10" Canvas.Top ="170"
            Foreground ="Red">170 </TextBlock>
        <TextBlock Canvas.Left ="10" Canvas.Top ="130"
            Foreground ="Red">190 </TextBlock>
        <TextBlock Canvas.Left ="10" Canvas.Top ="90"
            Foreground ="Red">210 </TextBlock>
        <TextBlock Canvas.Left ="10" Canvas.Top ="50"
            Foreground ="Red">230 </TextBlock>
        <TextBlock Canvas.Left ="10" Canvas.Top ="10"
            Foreground ="Red">250 </TextBlock>
        <TextBlock Canvas.Left ="630" Canvas.Top ="10"
            Foreground ="Red">电压（V） </TextBlock>
        <TextBlock Canvas.Left ="80" Canvas.Top ="210"
            Foreground ="Red">2:00 </TextBlock>
        <TextBlock Canvas.Left ="130" Canvas.Top ="210"
            Foreground ="Red">4:00 </TextBlock>
        <TextBlock Canvas.Left ="180" Canvas.Top ="210"
            Foreground ="Red">6:00 </TextBlock>
        <TextBlock Canvas.Left ="230" Canvas.Top ="210"
            Foreground ="Red">8:00 </TextBlock>
        <TextBlock Canvas.Left ="280" Canvas.Top ="210"
            Foreground ="Red">10:00 </TextBlock>
        <TextBlock Canvas.Left ="330" Canvas.Top ="210"
            Foreground ="Red">12:00 </TextBlock>
        <TextBlock Canvas.Left ="380" Canvas.Top ="210"
            Foreground ="Red">14:00 </TextBlock>
        <TextBlock Canvas.Left ="430" Canvas.Top ="210"
            Foreground ="Red">16:00 </TextBlock>
        <TextBlock Canvas.Left ="480" Canvas.Top ="210"
            Foreground ="Red">18:00 </TextBlock>
        <TextBlock Canvas.Left ="530" Canvas.Top ="210"
            Foreground ="Red">20:00 </TextBlock>
        <TextBlock Canvas.Left ="580" Canvas.Top ="210"
            Foreground ="Red">22:00 </TextBlock>
        <TextBlock Canvas.Left ="630" Canvas.Top ="210"
            Foreground ="Red">24:00 </TextBlock>
        <TextBlock Canvas.Left ="630" Canvas.Top ="230"
            Foreground ="Red">时间（小时）</TextBlock>
        <Path Name ="voltageCurvve" Stroke ="Red"
            StrokeThickness ="1"/>
    </Canvas>
</Window>
```

在上面的XAML中，笔者使用了简单的迷你绘图命令M（移动到某个位置）和L（画直线）。电压曲线是要实时生成的，在实际应用程序中，电压数据是由现场数据采集设备送来的，通常需要存在数据库中。为简单起见，笔者在C#中定义了一个电压数组voltage。假定每5分钟取一个点，这样一天需要288个点。

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Data;
using System.Windows.Documents;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Media.Imaging;
using System.Windows.Shapes;
using System.Windows.Threading;

namespace Yingbao.Chapter13
{
    public partial class VoltageWindow : System.Windows.Window
    {
        double[] voltage ={230.00, 230.01, 231.22, 231.00, 231.88 ,
231.85, 232.66, 231.08, 231.00, 231.01, 230.85, 230.83, 231.88 ,
231.33, 231.66, 231.08, 231.00, 230.88, 230.99, 231.00, 230.88 ,
232.33, 231.66, 232.08, 231.88, 231.56, 230.93, 230.00, 230.88 ,
229.33, 230.66, 231.08, 229.88, 229.08, 230.22, 230.00, 230.88 ,
230.33, 230.66, 231.08, 230.00, 230.01, 230.22, 230.44, 231.23 ,
230.65, 230.77, 229.95, 229.83, 229.85, 229.44, 229.24, 228.78 ,
228.75, 228.66, 229.08, 228.98, 229.01, 228.22, 228.56, 228.88 ,
228.45, 228.37, 228.08, 228.00, 228.01, 228.34, 228.00, 227.96 ,
227.87, 227.66, 227.54, 227.00, 227.01, 227.22, 227.00, 226.67 ,
226.33, 226.66, 226.05, 226.08, 226.01, 225.97, 225.83, 225.35 ,
225.01, 224.96, 224.85, 224.86, 224.65, 224.37, 224.00, 223.88 ,
223.44, 223.66, 223.08, 223.00, 222.96, 222.76, 222.76, 222.65 ,
222.33, 221.66, 221.97, 221.85, 221.92, 221.67, 221.48, 220.85 ,
221.04, 220.66, 220.56, 220.34, 220.02, 220.23, 220.00, 219.92 ,
219.33, 219.66, 220.08, 219.66, 219.01, 219.22, 219.00, 218.76 ,
218.33, 218.66, 218.08, 217.86, 216.95, 217.22, 216.88, 216.45 ,
215.97, 215.66, 215.87, 214.87, 214.65, 214.22, 214.00, 213.96 ,
213.33, 213.22, 212.56, 212.87, 212.88, 212.22, 212.00, 211.88 ,
211.23, 210.66, 210.08, 210.00, 211.01, 210.56, 209.90, 209.87 ,
209.77, 208.66, 209.08, 209.00, 209.01, 208.87, 208.60, 208.55 ,
208.34, 208.45, 208.08, 207.88, 207.33, 208.88, 207.00, 209.88 ,
208.33, 207.66, 208.08, 208.00, 207.01, 206.22, 206.00, 207.88 ,
205.33, 205.66, 206.08, 205.65, 206.01, 207.22, 206.00, 207.88 ,
206.33, 205.66, 206.08, 206.00, 204.01, 205.22, 203.00, 202.88 ,
205.33, 205.66, 206.08, 202.00, 202.01, 203.22, 203.00, 203.88 ,
202.33, 202.66, 202.08, 200.00, 201.01, 201.22, 203.00, 202.88 ,
205.33, 205.66, 206.08, 200.00, 201.01, 201.22, 203.00, 202.88 ,
205.33, 205.66, 206.08, 200.00, 201.01, 201.22, 203.00, 202.88 ,
205.33, 205.66, 206.08, 200.00, 201.01, 201.22, 203.00, 202.88 ,
205.33, 205.66, 206.08, 200.00, 201.01, 201.22, 203.00, 202.88 ,
205.33, 205.66, 206.08, 200.00, 201.01, 201.22, 203.00, 202.88 ,
205.33, 205.66, 206.08, 200.00, 201.01, 201.22, 203.00, 202.88 ,
205.33, 205.66, 206.08, 200.00, 201.01, 201.22, 203.00, 202.88 ,
205.33, 205.66, 206.08, 200.00, 201.01, 201.22, 203.00, 202.88 ,
205.33, 205.66, 206.08, 200.00, 201.01, 201.22, 203.00, 202.88 ,
```

```
205.33, 205.66, 206.08};

        PolyLineSegment pls = new PolyLineSegment();
        public VoltageWindow()
        {
            InitializeComponent();
            InitVoltageCurve();
        }
        void InitVoltageCurve()
        {
            Int32 xPos = Convert.ToInt32(System.DateTime.Now.Hour *
                        12 + System.DateTime.Now.Minute / 5 );
            PathGeometry gm = new PathGeometry();
            voltageCurvve.Data = gm;
            PathFigure pf = new PathFigure();
            gm.Figures.Add(pf);
            double x, y;
            x=30; y=210-(voltage[0]-150) *2;
            pf.StartPoint = new Point(x, y);
            pf.Segments.Add(pls);

            for (int i = 1; i < xPos; i++)
            {
                x += 2.0833333;
                y = 210 - (voltage[i] - 150) * 2;
                pls.Points.Add(new Point(x,y) );
            }

            DispatcherTimer timer = new DispatcherTimer();
            timer.Interval = TimeSpan.FromMinutes (1.000);
                // every 5 mitues
            timer.Tick +=new EventHandler(OnTimer);
            timer.Start();
        }

        void OnTimer(object sender, EventArgs ea)
        {
            if (System.DateTime.Now.Minute % 5 != 0)
            {
                return;
            }
            Int32 xPos = Convert.ToInt32(System.DateTime.Now.Hour *
                        12 + System.DateTime.Now.Minute / 5);
            double  x = 30 + xPos * 2.0833333;
            double y = 210 - (voltage[xPos] - 150) * 2;
            pls.Points.Add(new Point(x, y));
        }
    }
}
```

随着时间的变化，电压曲线需要不停地刷新。为此，在C#中使用了一个定时器。注意：在WPF中使用定时器，要使用System.Windows.Threading名称空间里的DispatchTimer，这个定时器是专门为

WPF设计的。笔者把定时器设置为每分钟产生一次事件，这样OnTimer每分钟会调用一次。若时间刚好在5分钟的整数倍，则可算出相应的电压值所对应的坐标，再刷新曲线。

由于电压数据是每5分钟一个点，所以每小时有12个点。对应于某一个时刻的电压，其在数组中的位置为：

```
Int32 xPos = Convert.ToInt32(System.DateTime.Now.Hour * 12 +
                             System.DateTime.Now.Minute / 5);
```

X方向的坐标：

```
double  x = 30 + xPos * 2.0833333;
```

在上述例子中5分钟在屏幕上占据2.0833333个距离。类似地，可以根据电压值算出某一时刻的y坐标：

```
          double y = 210 - (voltage[xPos] - 150) * 2;
```

图13-17是这一程序的运行结果，若让它运行着，便可以看到曲线在延伸。

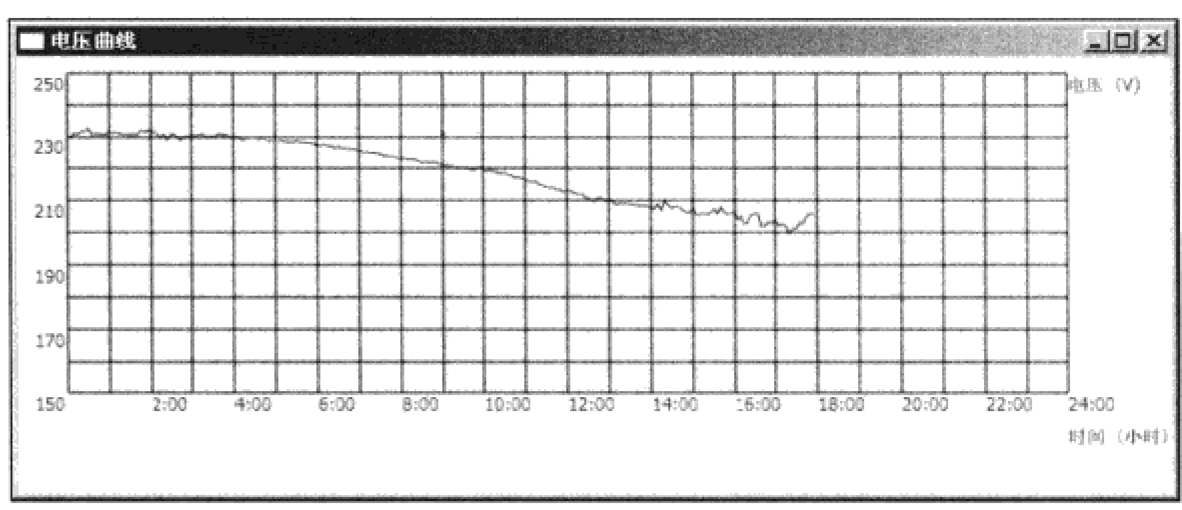


图13-17 使用WPF迷你语言产生的坐标及电压曲线图

13.3.16 流几何图形（StreamGeometry）

在有大量的图形数据的时候，如Google Earth等，需要进一步改善运行性能。WPF使用流几何图形来提高运行速度，流几何图形是把大量的矢量数据读入内层，这些数据是不需要修改的，换句话说，流几何图形只显示在计算机的屏幕上。

流几何图形不支持动画和数据绑定，它一般使用StreamGeometryContext来绘制，下面的例子是典型的用法：

```
StreamGeometry geometry = new StreamGeometry();

using (StreamGeometryContext ctx = geometry.Open())
{
 ctx.BeginFigure(new Point(), true, true);
 double step = 2 * Math.PI / Math.Max(numSides, 3);
 Point cur = c;
 double a = Math.PI * offsetDegree / 180.0;
 for (int i = 0; i < numSides; i++, a += step)
```

```
    {
        cur.X = c.X + r * Math.Cos(a);
        cur.Y = c.Y + r * Math.Sin(a);
        ctx.LineTo(cur, true , false );
    }
}
```

13.4 绘制（Drawing）

显示几何图形的一种方式是通过Shape中的路径，另外一种方式就是经由绘制（Drawing）由DrawingImage、DrawingBrush和DrawingVisual在屏幕上展现出来。在第5章画笔和画刷中，笔者创建的一个变压器画刷就是通过DrawingBrush把几何图形显示出来的。

DrawingImage、DrawingBrush和DrawingVisual这三个类中都含有一个Drawing属性，其类型为Drawing。类Drawing是一个抽象类，从Drawing类中派生出五个类：DrawingGroup、GeometryDrawing、ImageDrawing、GlyphRunDrawing和VideoDrawing（如图13-18所示）。

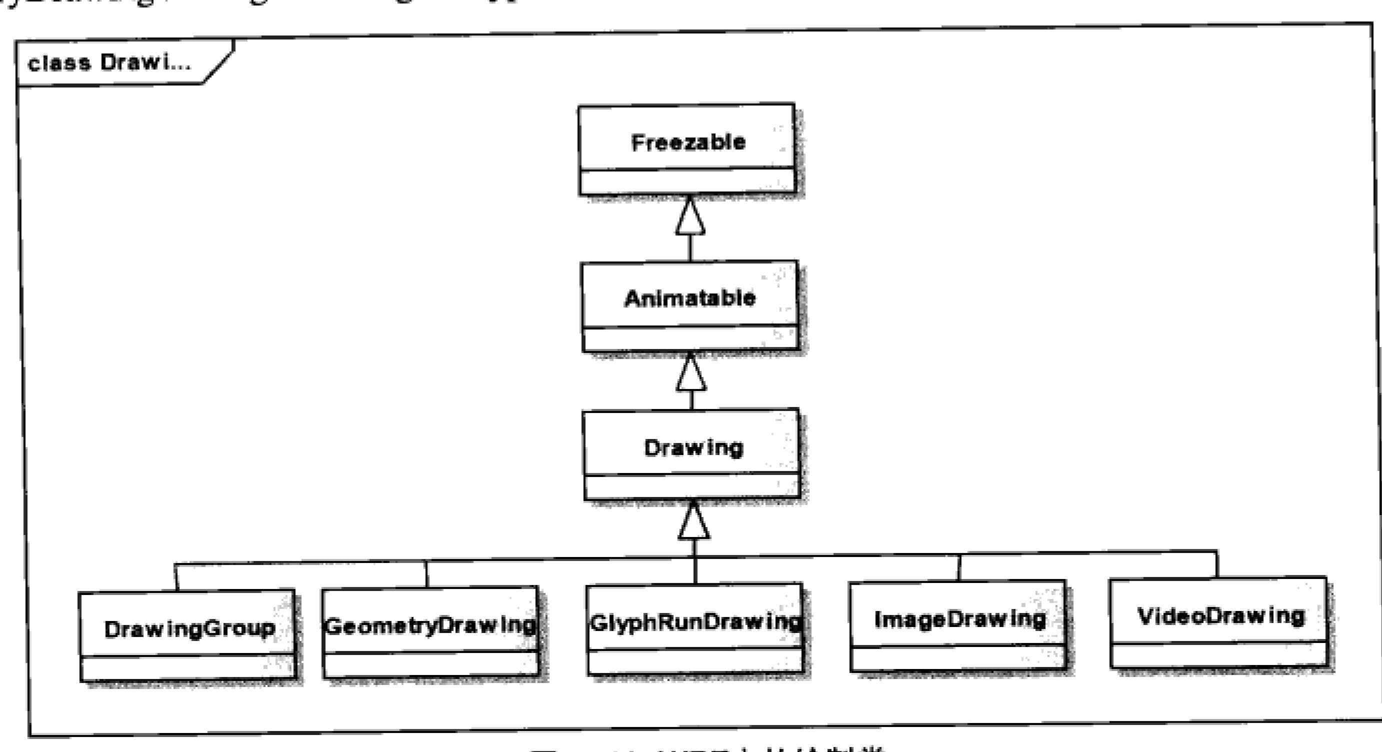


图13-18 WPF中的绘制类

- DrawingGroup的功能是把多个绘制组合到一块，其中含有透明度（Opacity）、变换等属性。这些属性可施加到其中所包含的任何绘制上。
- GeometryDrawing：含有画刷和画笔，其中的Geometry属性就是用来与前面讨论的几何图形相连。
- ImageDrawing：WPF对于光栅图形和矢量图形的界限不是很明晰，光栅图形（位图）中可以插入矢量图形；同样，矢量图形中可以含有位图。ImageDrawing就是这样一个在位图中绘制矢量图形的类。
- GlyphRunDrawing：与GlyphRun一起使用，可以创建特殊效果的字形。
- VideoDrawing：在图形中插入多媒体插件，第16章多媒体技术及其应用中将对此进行详细讨论。

13.4.1 使用DrawingImage显示几何图形

我们可以利用DrawingImage里的Drawing属性来合成位图和矢量图形，下面的例子展示了这一技术：

```
<Window x:Class="Yingbao.Chapter13.HouseWindow"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="HouseImage" Height="500" Width="500">
  <Image Stretch ="UniformToFill">
    <Image.Source>
      <DrawingImage>
        <DrawingImage.Drawing >
          <DrawingGroup>
            <ImageDrawing Rect ="10,10,400,400"
             ImageSource ="pack://application:,,,/myhouse.jpg"/>
            <GeometryDrawing>
              <GeometryDrawing.Pen>
                <Pen Brush ="Red" Thickness ="3"/>
              </GeometryDrawing.Pen>
              <GeometryDrawing.Geometry>
                <GeometryGroup>
                  <EllipseGeometry Center ="100,300" RadiusX ="30"
                       RadiusY ="20"/>
                  <EllipseGeometry Center ="145,300"
                       RadiusX ="30" RadiusY ="20"/>
                  <EllipseGeometry Center ="190,300"
                       RadiusX ="30" RadiusY ="20"/>
                  <EllipseGeometry Center ="120,270"
                       RadiusX ="30" RadiusY ="20"/>
                 <EllipseGeometry Center ="165,270"
                       RadiusX ="30" RadiusY ="20"/>
                </GeometryGroup>
              </GeometryDrawing.Geometry>
            </GeometryDrawing>
          </DrawingGroup>
        </DrawingImage.Drawing>
      </DrawingImage>
    </Image.Source>
  </Image>
</Window>
```

在DrawingGroup中，笔者放入了两个绘制（Drawing），一个是ImageDrawing，这是一张位图形式的照片；另一个为矢量图形的几何绘制。在这个简单的几何绘制里，画出了是奥运会的五环标志。WPF允许把两种不同形式的图像混合到一起，从而产生特定的效果，如图13-19所示。

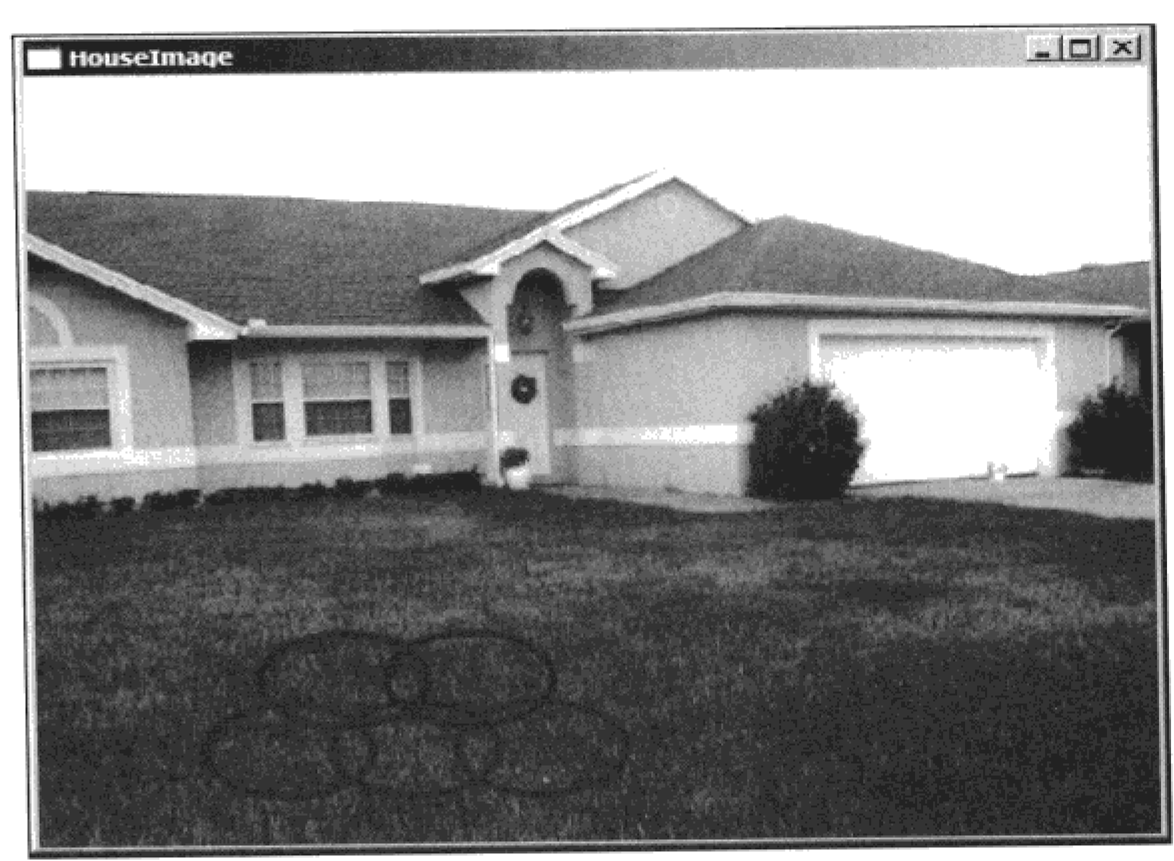


图13-19 几何图形和照片合成的图像

在上例中，使用了两个名字相似的类：ImageDrawing和DrawingImage，这两个类都和图像有关。ImageDrawing是一个绘制，用来显示位图形式的图像；DrawingImage是一个图像（ImageSource），其中的内容是绘制（Drawing）。

13.4.2 使用DrawingVisual来显示几何绘制

WPF中所有的显示操作都是通过类Visual来进行的，从这个类派生出了UIElement、ContainerVisual和ViewPort3DVisual这三大基类。前面的章节谈到的所有控件及Shape类，在其继承树上都可以找到UIElement。ViewPort3DVisual是显示三维图形的基类。DrawingVisual是从ContainerVisual中派生出来的一个重要的类，与UIElement相比，这个类不具有排版、风格和数据绑定这样一些控件的功能，因此它消耗的资源少、速度快。适合于使用大量图形元素的应用程序，绘制可以通过DrawingVisual在计算机屏幕上显示出来。

使用DrawingVisual的代价是不再能直接用XAML了——这个限制很容易理解：XAML比C#要慢。由于DrawingVisual没有排版功能，所以，在运行时刻DrawingVisual不知道放在屏幕上的位置。为此首先要为DrawingVisual创建一个宿主。

13.4.3 创建DrawingVisual宿主

WPF所有的控件都具有排版的功能，因此，从理论上来说，这些控件都可以作为DrawingVisual的宿主。从运行开销的角度出发，通常用FrameworkElement作为DrawingVisual的宿主。

例如，我们可以从FramworkElement中派生出HappyFace类作为宿主：

```
public class HappyFace : FrameworkElement
{
    public HappyFace()
```

```
    {
        this.Loaded += new RoutedEventHandler( OnLoaded );
    }
  .....
}
```

由于FrameworkElement具有排版、处理传递事件的功能，所以可以像放置控件一样地放置HappyFace。

13.4.4 绘制几何图形

通过DrawingVisual获取几何图形，需要用到DrawingContext类。DrawingContext类提供了十大类绘图函数及其重载，从简单的直线到在屏幕上显示复杂的几何图形，再到位图，都可以用DrawingContext绘制出来。

我们无法直接创建DrawingContext类，而需要通过DrawingVisual中的RenderOpen方法来获取DrawingContext对象：

```
DrawingVisual lVisual = new DrawingVisual();
using (DrawingContext dc = lVisual.RenderOpen())
   {
      .....
       //开始绘图
   }
```

13.4.5 把DrawingVisual对象加到FrameworkElement中的视觉树和逻辑树中

在创建了DrawingVisual对象之后，需要把该对象加入到FrameworkElement的视觉树和逻辑树中，由WPF平台管理：

```
AddVisualChild(face);
AddLogicalChild(face);
```

当把DrawingVisual加入到视觉树之后，还需要覆盖FrameworkElement中的VisualChildrenCount属性和GetVisualChild虚函数。

VisualChildrenCount必须返回HappyFace中的视觉元素的数目。这个简单的程序值含有一个DrawingVisual，所以VisualChildrenCount总是返回1。GetVisualChild 必须返回可视化元素，其产生从0开始的视觉元素数组下标index。在HappyFace例子中，只有一个元素，所以下标Index必须为0。当Index不为0 时出错；当Index为0时，返回DrawingVisual对象。若你的程序里含有多个视觉元素，就要根据这个下标Index返回不同的视觉元素。

下面是HappyFace的完整的程序：

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Windows;
using System.Windows.Media;
namespace Yingbao.Chapter13
{
```

```
    public class HappyFace : FrameworkElement
    {
      DrawingVisual    face;
      public HappyFace()
      {
        this.Loaded += new RoutedEventHandler( OnLoaded );
      }

      void OnLoaded( object sender, RoutedEventArgs e )
      {
          face = GetHappyFace();
          AddVisualChild(face);
          AddLogicalChild(face);
      }

      private DrawingVisual GetHappyFace()
      {
          DrawingVisual lVisual = new DrawingVisual();
          using (DrawingContext dc = lVisual.RenderOpen())
          {
              Pen p = new Pen(Brushes.BlueViolet,1);
              dc.DrawGeometry(null,p,Geometry.Parse(@"M150,100
              A30  50  0  1  1  250 120  A30 50  0  0 1 150 100
              M160,80 S165,60,190,80   M210,80 S225,60,240,80
              M165,100 S170,80,180,100  S170,120,165,100
              M215,100 S220,80,230,100  S220,120,215,100
              M195,135 A4 4  0  1  1  200 135 M170,140
              S200,220,220,140  M170,140 S200,180,220,140 " ) );
          }
          return lVisual;
      }

      protected override int VisualChildrenCount
      {
          get { return 1; }
      }

      protected override Visual GetVisualChild(int index)
      {
          if (index < 0 || index >= 1)
          {
              throw new IndexOutOfRangeException("index");
          }
          return face;
      }
    }
}
```

在Windows中显示快乐表情绘制和显示控件类似，下面的XAML是测试程序：

```
<Window x:Class="Yingbao.Chapter13.VisualWindow"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
   xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
```

```
    xmlns:local="clr-namespace:Yingbao.Chapter13"
   Title="快乐表情"
   Height="300" Width="400">
  <Canvas>
    <local:HappyFace Width="200" Height="300"  Canvas.Top="10"
          Canvas.Left="0"/>
    <local:HappyFace Width="200" Height="300"  Canvas.Top="10"
          Canvas.Left="100"/>
  </Canvas >
</Window>
```

图13-20是这个程序的运行结果：

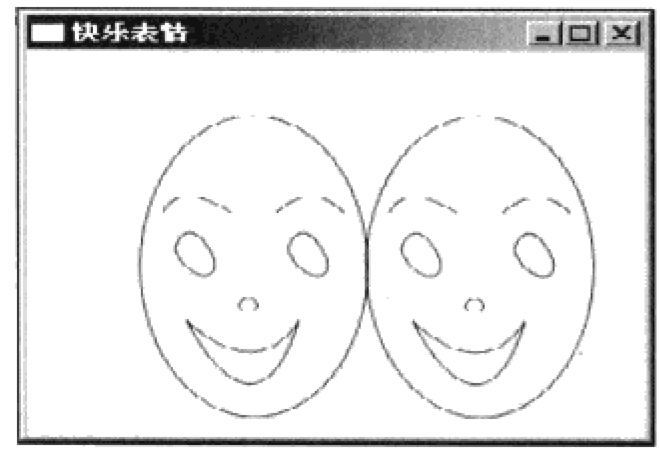


图13-20 使用DrawingVisual画出的快乐表情

13.4.6 选择视觉元素（Visual Hit Testing）

当用户在视窗下单击时，需要知道某个图形元素是否被选中。换句话说，对某个图形元素而言，当用户单击某个点时，需要知道这个图形是否包含该坐标？

单击某个UIElement元素，该元素会得到输入焦点。对于Drawing图形元素来说，其宿主是FrameworkElement或UIElement。当用户移动、单击、释放鼠标时，WPF传递事件系统会发出相应的传递事件。我们需要处理这些传递事件，并调用VisualTreeHelper中的HitTest方法来判定图形是否被选中。

13.4.7 简单选择判断

最简单的选择判断情形是只有一个图形，以我们的快乐表情为例，需要处理MouseLeftDown传递事件：

```
public HappyFace()
{
 //….
  this.MouseLeftButtonDown += new System.Windows.Input.
                      MouseButtonEventHandler(OnMouseLeftButtonDown);
 //…..
}

void OnMouseLeftButtonDown(object sender,
                     System.Windows.Input.MouseButtonEventArgs e)
{
```

```
    Point location = e.GetPosition(this);
    HitTestResult result = VisualTreeHelper.HitTest(this, location);
    if (result.VisualHit == face)
    {
     MessageBox.Show( "你选择了快乐表情！","快乐",MessageBoxButton.OK );
    }
}
```

这里的技术关键是使用VisualTreeHelper中的HitTest函数，其返回结果在HitTestResult中。HistTestResult的VisualHit属性就是鼠标单击处的视觉对象。

13.4.8 多个视觉元素的选择判断

若我们的HappyFace类中有多个视觉元素，仍然可以使用相同的技术，HitTestResult里的VisualHit属性返回鼠标单击处的视觉对象，总能够找出我们感兴趣的视觉元素。

13.4.9 视觉元素重叠时的选择判断

真正麻烦的是视觉元素重叠时的选择判断，当元素A处在元素B的后面，若整个元素A被元素B遮住了，我们这时一般不会关心元素A。但有时元素A被元素B部分遮住，或元素B上有一个空洞，从空洞里露出元素A。这时候，便需要知道鼠标单击时选择的是元素A还是元素B。

WPF提供一种回调机制，VisualTreeHelper里的HitTest允许用户提供一个委托函数，该函数中有一个HitTestResult参数。视觉树上有多少个元素被选中，VisualTreeHelper就调用多少次这个委托函数，这样，便可以得到鼠标单击处的所有元素。使用这一技术的代码如下：

```
void OnMouseLeftButtonDown(object sender,
              System.Windows.Input.MouseButtonEventArgs e)
  {
     base.OnMouseLeftButtonDown(e);
     Point location = e.GetPosition(this);
     VisualTreeHelper.HitTest(this, null, new HitTestResultCallback(HitTestCallBack),
       new PointHitTestParameters(location));
  }
```

有关选择的处理实际上是在委托函数中进行的：

```
public HitTestResultBehavior HitTestCallBack(HitTestResult result)
  {
      if (result.VisualHit == face)
      {
          // 进行有关快乐表情的处理
      }
      else {
             //进行其他处理
      }
      return HitTestResultBehavior.Continue;
  }
```

委托函数需要返回一个参数，其类型为HitTestResultBehavior，它有两个值：Continue和Stop。当函数返回Stop时，VisualTreeHelp就停止了HitTest测试。可以想象，当绘制图形复杂时，选择视觉元

素可能是一个非常复杂的工作。

13.5 本章小结

本章详细介绍了WPF中的二维图形系统，主要有形状（Shape）和几何图形两大类，可以直接对Shape进行排版、设定风格和数据绑定；而几何图形则需要通过视觉元素才能在屏幕上显示出来。如果你的应用程序所用到的视觉元素较少，使用Shape非常简便；如果你的应用程序要用到大量的图形元素，那么就要考虑使用几何图形。第14章将讨论图形变换技术。

第14章 图形转换

13章介绍了WPF中的二维图形，但是要完成复杂的图形，需要对图形进行转换。图形转换技术在动画中尤其有用，所谓的动画，无非是在某个事件发生时，需要对某个UI对象进行图形移动、放大、缩小，旋转等操作，而这一切正是图形转换所要完成的工作。我将在第15章讨论动画技术，本章我们集中讨论基本的图形转换技术。

14.1 图形转换概述

由于在WPF中控件和图形使用统一的编程模型，因此，转换技术不仅适用于图形，也适用于控件。WPF的转换可以对单个图形或控件进行，而不影响周围其他的控件或图形。

与WPF中很多操作是在相关属性的基础上进行的一样，转换技术也是以相关属性为基础的。转换技术实际上是对某个界面元素中类型为Transform的相关属性进行操作，例如在UIElement类中，这个属性是RenderTransform。对于FrameworkElement来说，可以进行两种转换，一种是对单个元素进行的，所操作的相关属性是RenderTransform，它是从UIElement继承来的相关属性；另外一种是对排版进行的转换，如果该元素含有其他界面元素，那么这种转换对其中的所有元素都有影响，这个相关属性是LayoutTransform。

表14-1 可转换的基类及相关属性

类名	用于转换的相关属性
Brush	Transform，RelativeTransform
ContainerVisual	Transform
DrawingGroup	Transform
FrameworkElement	RenderTransform，LayoutTransform
Geometry	Transform
TextEffect	Transform
UIElement	RenderTransform

表14-1列出了可转换的基类及其相关属性，由表14-1，我们可以发现：

- 转换是在WPF类结构中靠近树根处进行的，所以，虽然表14-1只列出了七大基类，但这七大类涵盖了WPF所有的控件、画刷、图形元素和文字元素。
- 用于转换的相关属性具有类型Transform，而Transform是所有转换类的基类，这表明所有的具体转换操作都可以施加到这些相关属性上（如图14-1所示）。

由于WPF中的图形是矢量图形，所以，在进行转换的过程中不会丢失细节。

图14-1示出了用于图形转换的类。实际上使用矩阵转换（MatrixTransform）可以达到缩放（ScaleTransform）、位移（TranslateTransform)、扭曲（SkewTransform）、旋转（RotateTransform）和组合转换（TranformGroup）所有的效果。就像第13章中，使用几何路径（path）可以产生矩形和椭圆一样，但WPF仍然提供Rectangle和Ellipse类，其基本考虑是使用方便。对于大多数程序员来说，可能不会用到MatrixTransform。

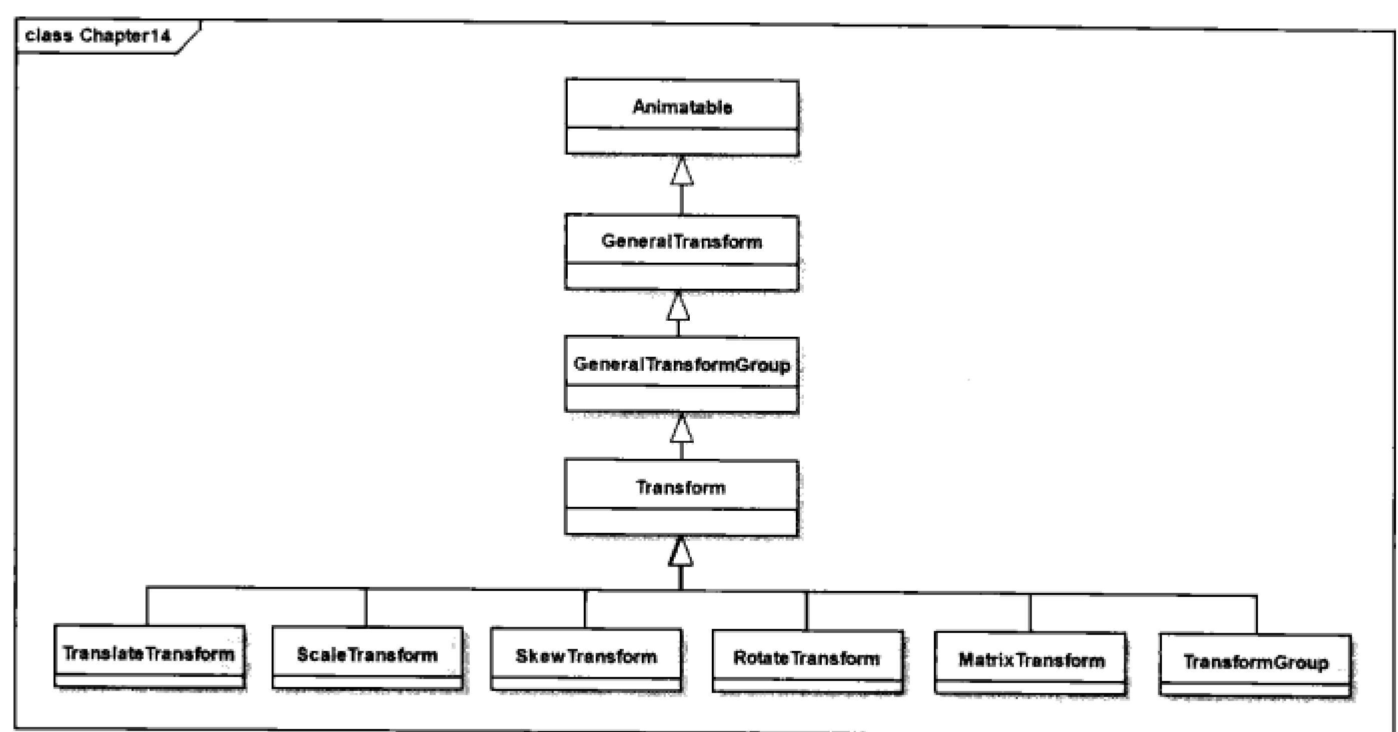


图14-1 用于图形转换的基本类

14.2 项目管理器

现在，我们掌握的技术越来越多，本章及以后的章节笔者试图逐步把所学过的技术结合起来。作为第一步，要把本章所要列举的例子组成一个项目，这个项目的名字就叫“Chapter14”。在Visual Studio里创建一个WPF项目，Visual Studio自动创建两个类，一个类是App，它是从Application类中派生出来的；另一个类是Window1，笔者把这个类的名字改为MainWindow，这个类从Window中派生出来，将作为本项目的主窗口。

让一个应用程序共享资源的方法是把资源放在Applicaion.Resource。本章要用到一个自制画刷，为此可把这个自制画刷放在Application.Resources中，并把它和关键字“YingbaoBlueGridBrush”联系起来：

```
<Application x:Class="Yingbao.Chapter14.App"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
StartupUri="MainWindow.xaml">
  <Application.Resources>
      <DrawingBrush x:Key="YingbaoBlueGridBrush"
        Viewport="0,0,10,10" ViewportUnits="Absolute"
        TileMode="Tile">
        <DrawingBrush.Drawing>
          <DrawingGroup>
            <DrawingGroup.Children>
              <GeometryDrawing Brush="White">
                <GeometryDrawing.Geometry>
                  <RectangleGeometry Rect="0,0,1,1" />
                </GeometryDrawing.Geometry>
              </GeometryDrawing>
              <GeometryDrawing Geometry="M0,0 L1,0 1,0.1, 0,0.1Z"
                  Brush="#CCCCFF" />
```

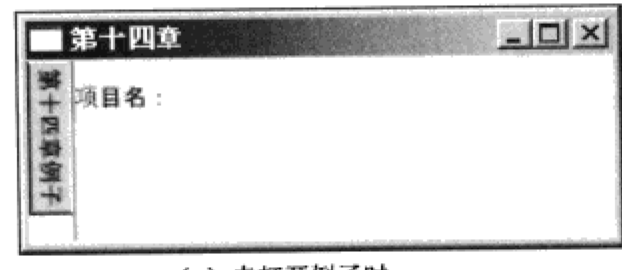
```
            <GeometryDrawing Geometry="M0,0 L0,1 0.1,1, 0.1,0Z"
                Brush="#CCCCFF" />
          </DrawingGroup.Children>
        </DrawingGroup>
      </DrawingBrush.Drawing>
    </DrawingBrush>
  </Application.Resources>
</Application>
```

在C# 代码中，App类是Visual Studio自动产生的，只是把名称控件改为Yingbao. Chapter14：

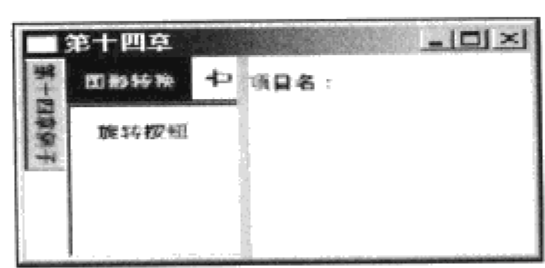
```
namespace Yingbao.Chapter14
{
    public partial class App : System.Windows.Application
    {
    }
}
```

在主窗口中，要能够产生类似于Visual Studio工具条那样的停靠效果，可把本章的例子放在一个可以停靠的窗口中（如图14-2（a））所示，当鼠标拖过"第14章例子"按钮时，要自动展开本章例子"窗口"（如图14-2（b）所示）。若按下"第14章例子"窗口上的图钉，主窗口自动分成两个部分（如图14-2（c）所示）。若选择某个例子，将在主窗口的右边显示例子的运行结果（如图14-2（d）所示）。最后，若觉得右边的窗口不够大，则可以单击左边窗口的图钉，主窗口的左半部自动隐藏起来（如图14-2（e）所示）。

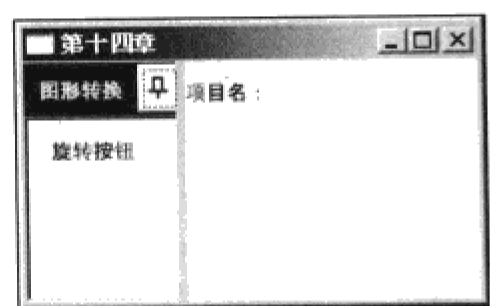
过去，要实现这样的功能，我们需要写数千行代码，而现在使用WPF则简单得多。图（14-2（a））中的"第14章例子"为一个按钮，在排版时，我把它旋转了90°：



（a）未打开例子时

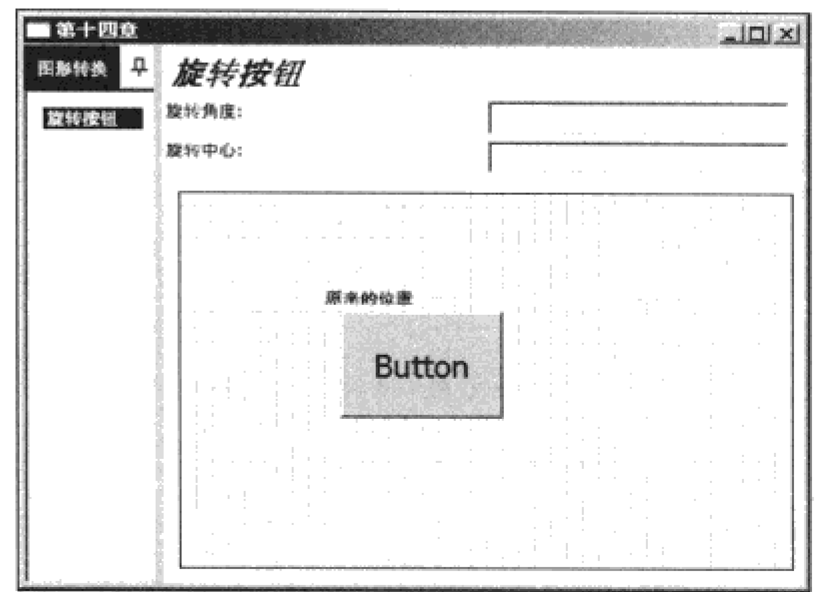


（b）鼠标拖过"第14章例子"，未按下图钉

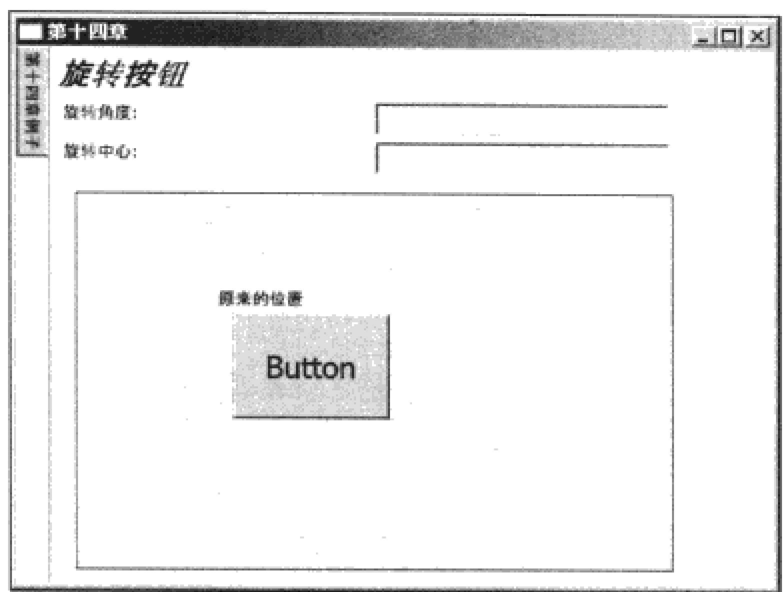


（c）按下图钉，主窗口自动分为两部分

图14-2 第14章项目的主窗口



(d) 双击左边的例子，主窗口的右边显示相应的例子



(e) 再单击图钉，主窗口由左右两个部分变成一个部分

图14-2 第14章项目的主窗口（续）

```
<StackPanel Name="exampleBar" Orientation="Horizontal"
       DockPanel.Dock="Left">
      <StackPanel.LayoutTransform>
        <RotateTransform Angle="90"/>
      </StackPanel.LayoutTransform>
      <Button Name="exampleButton"
            MouseEnter="ExampleButton_MouseEnter">
        第14章例子
      </Button>
    </StackPanel >
```

要实现图14-2所示的功能，首先要设计主窗口的排版。笔者用DockPanel视窗的子元素，然后把“第14章例子”这个按钮经过旋转90°后停靠在窗口的左边。然后在DockPanel里面放置一个名为parentGrid的网格排版控件(Grid），这个网格排版控件占据了整个窗口：

```
<Grid Name ="parentGrid" Grid.IsSharedSizeScope="True">
```

这里把IsSharedSizeScope设为True，目的是在不同的子网格控件中自动把两列的宽度设为相等。在ParentGrid中，放了两个Grid面板，mainGrid和exampleGrid。exampleGrid的Visibility属性设为“Collapsed”，即不可见。mainGrid和exampleGrid都有两列，其中第一列的宽度在显示exampleGrid的时候设为相等；在不显示exampleGrid的时候，把mainGrid的第一列的宽度设为1.0个单位（相当于把第一列隐藏起来，这是因为Grid中没有动态显示/隐藏行或列的功能）。

MainGrid中第一列的作用就是给exampleGrid预留地方的；MainGrid中的第二列中含有一个标题栏控件（HeaderedContentControl），其标题Header将用来显示例子的名字，其内容将是每个例子中的排版控件（Panel）。

exampleGrid的第一列中含有一个两行的子Grid，第二列包含一个Gridsplitter控件，用来调整exampleGrid的宽度。子Grid的第一行用于显现图钉和名称，第二行中包含一个列表框，我们将把所有的例子都放在这个列表框中。下面是完整的XAML：

```
<Window x:Class="Yingbao.Chapter14.MainWindow"
```

```
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="第14章" Height="400" Width="600">
  <DockPanel >
    <StackPanel Name="exampleBar" Orientation="Horizontal"
                DockPanel.Dock="Left">
      <StackPanel.LayoutTransform>
        <RotateTransform Angle="90"/>
      </StackPanel.LayoutTransform>
      <Button Name="exampleButton"
          MouseEnter="ExampleButton_MouseEnter">第14章例子
      </Button>
    </StackPanel >
    <Grid Name ="parentGrid" Grid.IsSharedSizeScope="True">
      <Grid Name="mainGrid"   MouseDown ="MainGrid_MouseEnter">
        <Grid.RowDefinitions>
          <RowDefinition/>
        </Grid.RowDefinitions>
        <Grid.ColumnDefinitions>
          <ColumnDefinition Name ="mainGridColumn0"
              SharedSizeGroup ="column0" Width ="auto" />
          <ColumnDefinition />
        </Grid.ColumnDefinitions>
          <HeaderedContentControl  Name ="projects"
             BorderThickness ="2" BorderBrush="Chocolate"
             Background ="Cyan" Grid.Column ="1">
            <HeaderedContentControl.Header>
              <Label FontSize ="20" FontStyle ="Italic"
                FontWeight ="Bold" Name="lblProjectName"/>
            </HeaderedContentControl.Header>
            <HeaderedContentControl.Content>
              项目名:
            </HeaderedContentControl.Content >
          </HeaderedContentControl>
      </Grid>
      <Grid Name ="exmpleGrid" Visibility ="Collapsed">
          <Grid.ColumnDefinitions>
            <ColumnDefinition SharedSizeGroup="column0"
               Width ="auto"/>
            <ColumnDefinition />
          </Grid.ColumnDefinitions>
          <Grid Grid.Column="0"
             MouseEnter="ExampleGridMouseEnter"
             Background="{DynamicResource {x:Static
             SystemColors.ActiveCaptionBrushKey}}" >
            <Grid.RowDefinitions>
              <RowDefinition Height="auto"/>
              <RowDefinition/>
            </Grid.RowDefinitions>
            <DockPanel Grid.Row="0">
              <Button Width="26" Name="examplePin"
                DockPanel.Dock="Right"
                Click="ExPinClick" Background="White">
```

```
                <Image Name="ExPanelPinImage"
                Source="Image/pinHorizontal.gif"/>
              </Button>
              <TextBlock Padding="8"
                 TextTrimming="CharacterEllipsis"
                 Foreground = "{DynamicResource {x:Static
                 SystemColors.ActiveCaptionTextBrushKey}}"
                 DockPanel.Dock="Left">图形转换</TextBlock>
            </DockPanel>
            <ListBox  Name ="listExample" Padding="10"
                 Grid.Row="1" SelectionChanged
                 ="OnExampleChanged" SelectionMode ="Single">
              <ListBoxItem Tag ="RotatedButton" >旋转按钮
                 </ListBoxItem>
              <!--     我们将加入更多的例子--   >
            </ListBox>
          </Grid>
          <GridSplitter Width="5" Grid.Column="0"
                        HorizontalAlignment="Right"/>
        </Grid>
    </Grid >
  </DockPanel>
</Window>
```

当我们选择列表框里的条目时，将产生OnExampleChanged事件。在这个事件处理程序中，笔者动态创建相应的类：

```
private void OnExampleChanged(object sender,
               RoutedEventArgs ea)
{
    ListBoxItem item = listExample.SelectedItem
          as ListBoxItem;
    string className = item.Tag as string;
    Type type = this.GetType();
    Assembly assembly = type.Assembly;
    Panel panel = (Panel)assembly.CreateInstance(
        type.Namespace + "." + className);
    lblProjectName.Content = item.Content;
    projects.Content = panel;
}
```

注意，这里className取自item.Tag，在XAML里，笔者把Tag设为相应例子的类名：

```
<ListBoxItem Tag ="RotatedButton" >旋转按钮</ListBoxItem>
```

完整的C#后台处理程序如下所示：

```
namespace Yingbao.Chapter14
{
    public partial class MainWindow : System.Windows.Window
    {
        GridLength col0Width = new GridLength(1.0);
        public MainWindow()
```

```
    {
        InitializeComponent();
        ShowMainGridColumn0(false);
    }
    void ExampleButton_MouseEnter(object sender,
                RoutedEventArgs rea)
    {
        exmpleGrid.Visibility = Visibility.Visible;
        ShowMainGridColumn0(true);
    }

    void MainGrid_MouseEnter(object sender,
            RoutedEventArgs rea)
    {
        if (exampleButton.Visibility == Visibility.Visible)
        {
            exmpleGrid.Visibility = Visibility.Collapsed;
            ShowMainGridColumn0(false);
        }
    }

    void ExampleGridMouseEnter(object sender,
            RoutedEventArgs rea)
    {
    }

    void ExPinClick(object sender, RoutedEventArgs rea)
    {
        if (exampleButton.Visibility == Visibility.Collapsed)
            UnDockPanel( );
        else
            DockPanel();
    }

public void DockPanel( )
{
    exampleButton.Visibility = Visibility.Collapsed;
    ExPanelPinImage.Source = new BitmapImage(new
          Uri("Image/pin.gif", UriKind.Relative));
    ShowMainGridColumn0(true);
}

    public void UnDockPanel()
    {
        exampleButton.Visibility = Visibility.Visible;
        ExPanelPinImage.Source = new BitmapImage(new
          Uri("Image/pinHorizontal.gif", UriKind.Relative));
        ShowMainGridColumn0(false);

    }
    private void ShowMainGridColumn0(bool show)
    {
        if (show)
```

```
            {
                mainGridColumn0.Width = GridLength.Auto;
                mainGridColumn0.SharedSizeGroup = "column0";
            }
            else {
                mainGridColumn0.Width = col0Width;
                mainGridColumn0.SharedSizeGroup = "noGroup";
            }
        }
        private void OnExampleChanged(object sender,
                       RoutedEventArgs ea)
        {
            ListBoxItem item = listExample.SelectedItem
                                 as ListBoxItem;
            string className = item.Tag as string;
            Type type = this.GetType();
            Assembly assembly = type.Assembly;
            Panel panel = (Panel)assembly.CreateInstance(
                type.Namespace + "." + className);
            lblProjectName.Content = item.Content;
            projects.Content = panel;
        }
    }
}
```

有了这样一个管理例子的程序框架之后，就应在其中加入本章要讨论的所有例子。

14.3 旋转转换（RotateTransform）

在平面内控制图形旋转，需要设置两个参数：一个是旋转的中心，另一个是旋转的角度。WPF中的旋转中心坐标，不在坐标轴的原点，而是相对于每个图形或控件本身。

如图14-3所示，当旋转矩形区域时，其默认的旋转中心的原点在矩形的左上角(0,0)，以矩形的长度为1，所以右上角的坐标为（1，0），右下角的坐标为（1，1），矩形的中心坐标为（0.5，0.5)……可以让矩形围绕着旋转中心（2，2）旋转，该点在(0，0)、(1，1)连线右下角的两倍处；也可以让矩形围绕着旋转中心(-2，-2)旋转，该点位于同一连线左上角的两倍处。

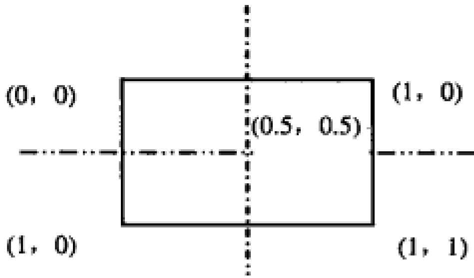


图14-3 旋转中心坐标

下面的XAML可让你任意旋转一个按钮：

```
<StackPanel x:Class="Yingbao.Chapter14.RotatedButton"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
   xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
   Height ="340" Width ="450">
```

```
  <StackPanel.Resources>
    <Style TargetType="{x:Type Button}">
      <Setter Property="FontSize" Value="20" />
      <Setter Property="Content" Value="Button" />
      <Setter Property="HorizontalAlignment" Value="Center" />
      <Setter Property="VerticalAlignment" Value="Center" />
    </Style>
  </StackPanel.Resources>
  <Grid Width="400" Height="52"
        HorizontalAlignment="Left" VerticalAlignment="Top">
    <Grid.ColumnDefinitions>
      <ColumnDefinition/>
      <ColumnDefinition/>
    </Grid.ColumnDefinitions>
    <Grid.RowDefinitions>
      <RowDefinition/>
      <RowDefinition/>
    </Grid.RowDefinitions>
     <Label Padding ="2,2,10,2" Grid.Column="0">旋转角度:</Label>
    <TextBox Name="rotateAngle" Margin="2,2,10,2"
             Grid.Column="1" />
    <Label Padding ="2,2,10,2" Grid.Column="0"
             Grid.Row="1">旋转中心:</Label>
    <TextBox Name ="rotateCenter"    Margin ="2,2,10,2"
             Grid.Column="1"  Grid.Row="1"/>
  </Grid>
  <Border Margin="10" BorderBrush="Black" BorderThickness="1"
             Background="{StaticResource YingbaoBlueGridBrush}"
             HorizontalAlignment="Left">
    <Canvas ClipToBounds="True" Width="380" Height="250">
    <TextBlock Canvas.Top="63" Canvas.Left="90" Text="原来的位置"/>
      <Rectangle Canvas.Top="80" Canvas.Left="100"
          Width="100" Height="70" Stroke="Black"
          StrokeThickness="1"  StrokeDashArray="4,2,1"/>
      <Button RenderTransformOrigin="{Binding ElementName=
          rotateCenter,Path=Text}" Canvas.Left="100"
          Canvas.Top="80" Width="100" Height="70">
        <Button.RenderTransform>
          <RotateTransform  Angle="{Binding ElementName=
             rotateAngle,Path=Text}" />
        </Button.RenderTransform>
      </Button>
    </Canvas>
  </Border>
</StackPanel >
```

在旋转角度和旋转中心中输入相应的值，如旋转角度输入90°，旋转中心输入1,1，便可以立即看出如图14-4所示的按钮围绕右下角旋转90°的效果。

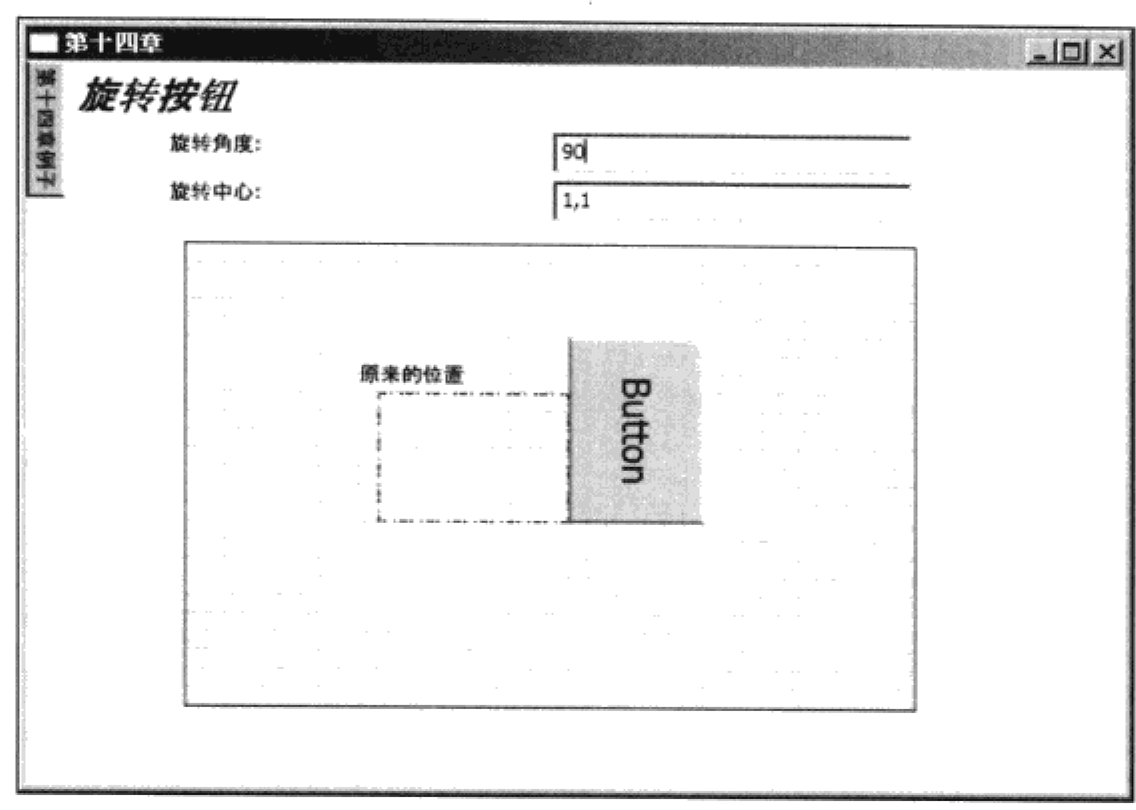


图14-4 按钮围绕右下角旋转90° 的效果

14.4 位移转换（TranslateTransform）

位移是一个简单的转换，在平面上进行位移转换，实际上是改变图形元素的位置坐标（x, y）。如图形元素原来在（x_0, y_0)处，那么，当在x方向移动dx，在y方向移动dy时，图形将在新的位置：

$x = x_0 + dx$;

$y = y_0 + dy$;

这里dx和dy可以是负数，当dx和dy为负数时，图形对象向左上方移动。

下面的XAML是使用位移转换移动矩形：

```
<StackPanel x:Class="Yingbao.Chapter14.TranslatesRectangle"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
Height ="340" Width ="450">
  <Grid Width="400" Height="52" HorizontalAlignment="Left"
        VerticalAlignment="Top">
    <Grid.ColumnDefinitions>
      <ColumnDefinition Width="*"/>
      <ColumnDefinition Width="3*"/>
      <ColumnDefinition Width="*"/>
    </Grid.ColumnDefinitions>
    <Grid.RowDefinitions>
      <RowDefinition/>
      <RowDefinition/>
    </Grid.RowDefinitions>
    <Label Padding ="2,2,10,2" Grid.Column="0">X方向移动:</Label>
    <ScrollBar Name="xScroll" Orientation="Horizontal"
        Minimum="-200" Maximum="300" Grid.Column="1"/>
    <TextBox  Margin ="2,2,10,2" Grid.Column="2"
```

```
        Text="{Binding ElementName=xScroll, Path=Value}" />
    <Label Padding ="2,2,10,2" Grid.Column="0"
        Grid.Row="1">Y方向移动:</Label>
    <ScrollBar Name="yScroll" Orientation="Horizontal"
        Minimum="-200" Maximum="300" Grid.Column="1"
        Grid.Row="1"/>
    <TextBox Name ="rotateCenter"    Margin ="2,2,10,2"
        Text="{Binding  ElementName=yScroll, Path=Value}"
        Grid.Column="2"  Grid.Row="1"/>
  </Grid>
  <Border Margin="10" BorderBrush="Black" BorderThickness="1"
            Background="{StaticResource YingbaoBlueGridBrush}"
            HorizontalAlignment="Left">
    <Canvas ClipToBounds="True" Width="380" Height="250">
    <TextBlock Canvas.Top="63" Canvas.Left="90" Text="开始位置:"/>
        <Rectangle Canvas.Top="80" Canvas.Left="100"
           Width="100" Height="70" Stroke="Black"
           StrokeThickness="1" StrokeDashArray="4,2,1"/>
      <Rectangle Canvas.Top="80" Canvas.Left="100"
           Width="100" Height="70" Stroke="Red"
           StrokeThickness="3" Fill="LightBlue">
        <Rectangle.RenderTransform>
          <TranslateTransform
             X="{Binding ElementName=xScroll, Path=Value}"
             Y="{Binding ElementName=yScroll, Path=Value}"/>
        </Rectangle.RenderTransform>
      </Rectangle>
      </Canvas>
  </Border>
</StackPanel >
```

图14-5是使用位移转换移动矩形后得到的结果。

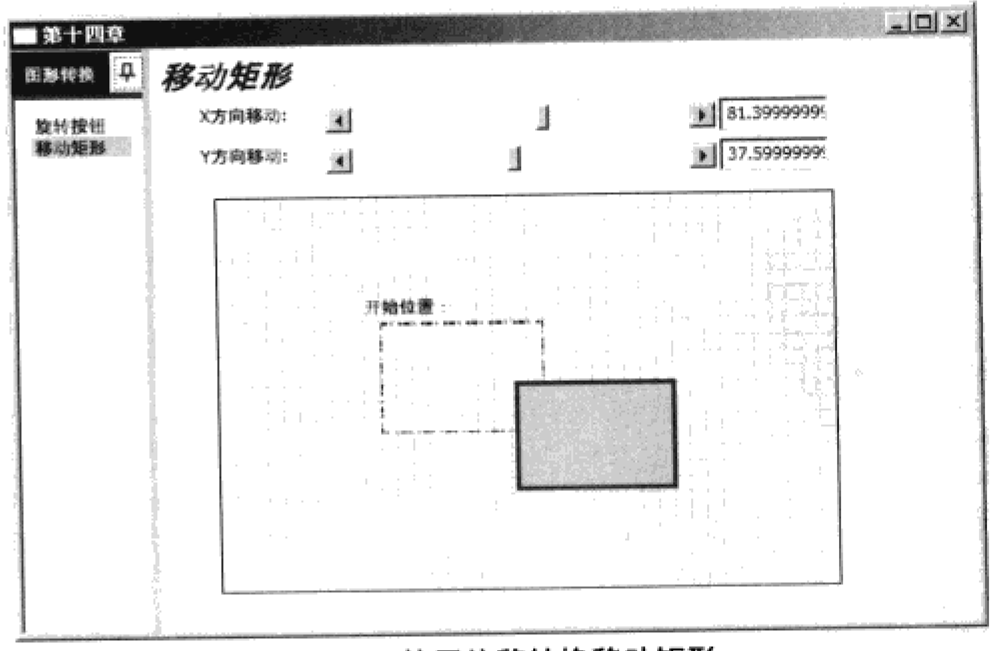


图14-5 使用位移转换移动矩形

14.5 缩放转换（ScaleTransform）

放大缩小是图形软件中常用的操作，我们可以单独放大或缩小视窗中某个元素，而不必管理视窗中的其他元素。

缩放操作使用两个参数：放大/缩小倍数和缩放中心位置。若用P_x表示X轴方向的缩放倍数，用P_y表示Y轴方向的缩放倍数，用（C_x,C_y）表示缩放中心，那么图形上的任何一点（x,y)转换后的坐标(Xt,Yt)为：

$Xt=P_x(x-C_x)+C_x$；

$Yt=P_y(y-C_y)+C_y$;

若缩放倍数大于1，图形是放大的；若小于1，图形是缩小的。缩放倍数也可以是负数。当缩放倍数为负数时，所生成的图形称为原图形的镜像。

下面的XAML可让我们观察缩放转换的效果：

```
<StackPanel
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
   xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    x:Class="Yingbao.Chapter14.ScaleRectangle" Height ="500"
        Width ="450">
  <Grid Width="400" Height="104"
        HorizontalAlignment="Left" VerticalAlignment="Top">
    <Grid.ColumnDefinitions>
      <ColumnDefinition Width="*"/>
      <ColumnDefinition Width="3*"/>
      <ColumnDefinition Width="*"/>
    </Grid.ColumnDefinitions>
    <Grid.RowDefinitions>
      <RowDefinition/>
      <RowDefinition/>
      <RowDefinition/>
      <RowDefinition/>
    </Grid.RowDefinitions>
    <Label Padding ="2,2,10,2" Grid.Column="0">X方向缩放:</Label>
    <ScrollBar Name="xScroll" Orientation="Horizontal"
        Minimum="0.1" Maximum="2" Grid.Column="1"/>
    <TextBox  Margin ="2,2,10,2" Grid.Column="2"
        Text="{Binding ElementName=xScroll, Path=Value}" />
    <Label Padding ="2,2,10,2" Grid.Column="0"
        Grid.Row="1">Y方向缩放:</Label>
    <ScrollBar Name="yScroll" Orientation="Horizontal"
        Minimum="0.1" Maximum="2" Grid.Column="1" Grid.Row="1"/>
    <TextBox    Margin ="2,2,10,2"
        Text="{Binding ElementName=yScroll, Path=Value}"
        Grid.Column="2"  Grid.Row="1"/>
    <Label Padding ="2,2,10,2" Grid.Column="0"
        Grid.Row="2">X中心:</Label>
    <ScrollBar Name="xcenter" Orientation="Horizontal"
      Minimum="-100" Maximum="100" Grid.Column="1" Grid.Row="2"/>
    <TextBox Margin ="2,2,10,2"
        Text="{Binding ElementName=xcenter, Path=Value}"
```

```
        Grid.Column="2"  Grid.Row="2"/>
    <Label Padding ="2,2,10,2" Grid.Column="0"
        Grid.Row="3">Y中心:</Label>
    <ScrollBar Name="ycenter" Orientation="Horizontal"
     Minimum="-100" Maximum="100" Grid.Column="1" Grid.Row="3"/>
    <TextBox Margin ="2,2,10,2" Grid.Column="2"  Grid.Row="3"
       Text="{Binding ElementName=ycenter, Path=Value}"  />
  </Grid>
  <Border Margin="10" BorderBrush="Black" BorderThickness="1"
            Background="{StaticResource YingbaoBlueGridBrush}"
            HorizontalAlignment="Left">
    <Canvas ClipToBounds="True" Width="380" Height="250">
      <TextBlock Canvas.Top="63" Canvas.Left="90"
            Text="开始位置: "/>
        <Rectangle Canvas.Top="80" Canvas.Left="100"
            Width="100" Height="70" Stroke="Black"
            StrokeThickness="1" StrokeDashArray="4,2,1"/>

      <Rectangle Canvas.Top="80" Canvas.Left="100"
           Width="100" Height="70" Stroke="Red"
           StrokeThickness="3" Fill="LightBlue">
        <Rectangle.RenderTransform>
          <ScaleTransform
            ScaleX="{Binding ElementName=xScroll, Path=Value}"
            ScaleY="{Binding ElementName=yScroll, Path=Value}"
            CenterX="{Binding ElementName=xcenter, Path=Value}"
            CenterY="{Binding ElementName=ycenter, Path=Value}"/>
        </Rectangle.RenderTransform>
      </Rectangle>
      </Canvas>
  </Border>
</StackPanel >
```

我们可以分别调整*X*和*Y*方向缩放倍数，以及*X*和*Y*中心坐标。上面的XAML在我们项目管理器中的运行情况如图14-6所示。

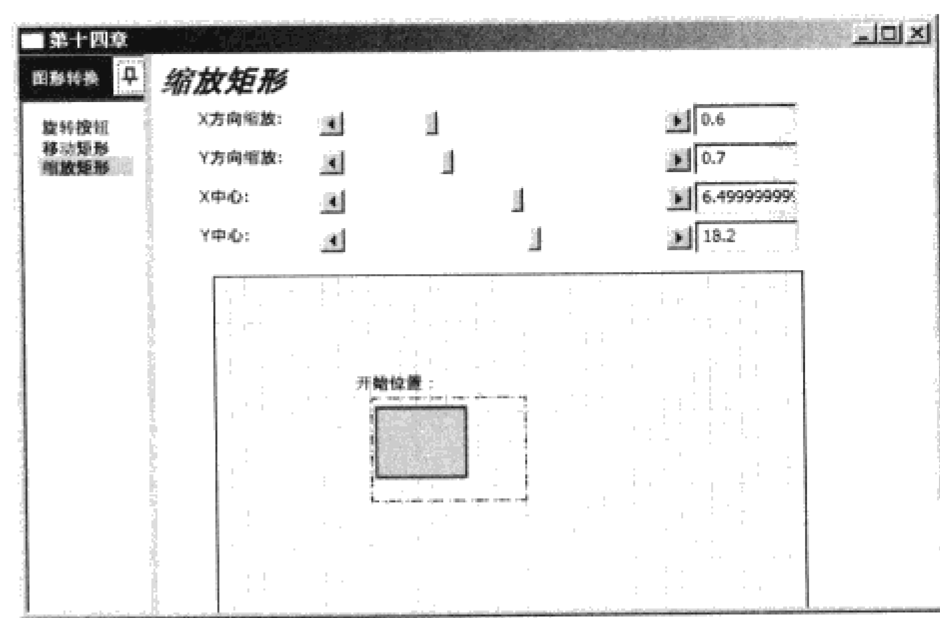


图14-6 缩放转换矩形

14.6 扭曲转换（SkewTransform）

在平面内使用扭曲转换（SkewTransform）可以达到简单的立体效果，而没有三维图形运行时的开销。

扭曲操作使用两个参数：扭曲度数和扭曲中心位置。若用A_x表示X轴方向的扭曲度数，用A_y表示Y轴方向的扭曲度数，用（C_x, C_y）表示扭曲中心，那么图形上的任何一点（x,y)转换后的坐标(Xt,Yt)为：

$Xt=x + \tan(A_x) (y-C_x)$;

$Yt=y + \tan(A_y) (x-C_y)$;

通过调整X方向和Y方向的扭曲度数，可以产生简单的立体图形。调整下例中的参数，可以看出扭曲的效果。

```
<StackPanel Height ="500" Width ="450"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        x:Class="Yingbao.Chapter14.SkewRectangle" >
  <Grid Width="400" Height="104"
        HorizontalAlignment="Left"  VerticalAlignment="Top">
    <Grid.ColumnDefinitions>
      <ColumnDefinition Width="*"/>
      <ColumnDefinition Width="3*"/>
      <ColumnDefinition Width="*"/>
    </Grid.ColumnDefinitions>
    <Grid.RowDefinitions>
      <RowDefinition/>
      <RowDefinition/>
      <RowDefinition/>
      <RowDefinition/>
    </Grid.RowDefinitions>
    <Label Padding ="2,2,10,2" Grid.Column="0">X方向角度:</Label>
    <ScrollBar Name="xScroll" Orientation="Horizontal" Value ="0"
           Minimum="-90"  Maximum="90" Grid.Column="1"/>
    <TextBox  Margin ="2,2,10,2" Grid.Column="2"
           Text="{Binding ElementName=xScroll, Path=Value}" />
    <Label Padding ="2,2,10,2" Grid.Column="0"
           Grid.Row="1">Y方向角度:</Label>
    <ScrollBar Name="yScroll" Orientation="Horizontal"
           Value ="0" Minimum="-90"
           Maximum="90" Grid.Column="1" Grid.Row="1"/>
    <TextBox Margin ="2,2,10,2" Grid.Column="2"  Grid.Row="1"
           Text="{Binding ElementName=yScroll, Path=Value}"/>
    <Label Padding ="2,2,10,2" Grid.Column="0"
           Grid.Row="2">X中心:</Label>
    <ScrollBar Name="xcenter" Orientation="Horizontal"
           Minimum="-100" Maximum="100" Grid.Column="1"
           Grid.Row="2"/>
    <TextBox Margin ="2,2,10,2" Grid.Column="2"  Grid.Row="2"
        Text="{Binding ElementName=xcenter, Path=Value}"/>
```

```
    <Label Padding ="2,2,10,2" Grid.Column="0"
        Grid.Row="3">Y中心:</Label>
    <ScrollBar Name="ycenter" Orientation="Horizontal"
        Minimum="-100" Maximum="100" Grid.Column="1"
        Grid.Row="3"/>
    <TextBox Margin ="2,2,10,2" Grid.Column="2"  Grid.Row="3"
        Text="{Binding ElementName=ycenter, Path=Value}"/>
  </Grid>
  <Border Margin="10" BorderBrush="Black" BorderThickness="1"
          Background="{StaticResource YingbaoBlueGridBrush}"
          HorizontalAlignment="Left">
    <Canvas ClipToBounds="True" Width="380" Height="250">
      <TextBlock Canvas.Top="63" Canvas.Left="90"
          Text="开始位置: "/>
      <Rectangle Canvas.Top="80" Canvas.Left="100"
          Width="100" Height="70" Stroke="Black"
          StrokeThickness="1" StrokeDashArray="4,2,1"/>
      <Rectangle Canvas.Top="80" Canvas.Left="100"
           Width="100" Height="70" Stroke="Red"
           StrokeThickness="3"  Fill="LightBlue">
        <Rectangle.RenderTransform>
          <SkewTransform
            AngleX ="{Binding ElementName=xScroll, Path=Value}"
            AngleY ="{Binding ElementName=yScroll, Path=Value}"
            CenterX="{Binding ElementName=xcenter, Path=Value}"
            CenterY="{Binding ElementName=ycenter, Path=Value}"/>
        </Rectangle.RenderTransform>
      </Rectangle>
    </Canvas>
  </Border>
</StackPanel >
```

上面这段XAML的运行结果如图14-7所示。

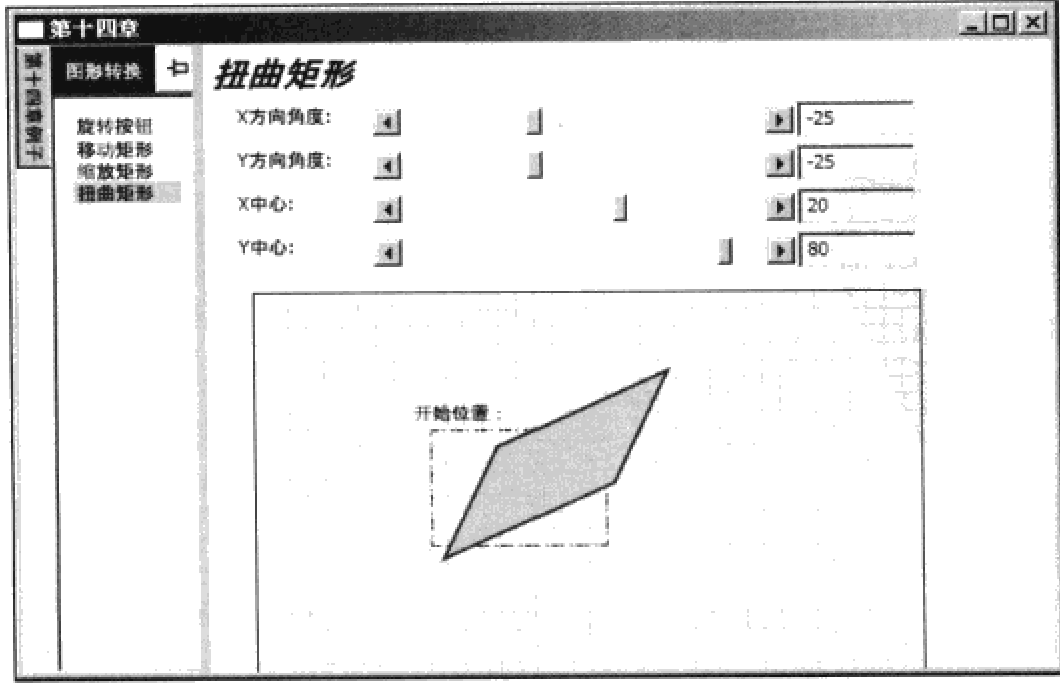


图14-7 扭曲转换

14.7 组合转换（TransformGroup）

组合转换是把上述四种转换结合起来的一种技术。其基本的语法如下：

```
<TransformGroup>
   <ScaleTransform  ....../>
   <RotateTransform ....../>
            ....
</TransformGroup />
```

由于TransformGroup是从Transform中派生出来的，所以，所有可以使用Transform的地方，都可以使用TransformGroup。在组合转换中的单个转换的次序非常重要，例如，先做位移再做旋转，得到的结果和先做旋转再做位移是不一样的。这是因为在这两次旋转中，对象的旋转中心不一样了。

```
<StackPanel Height ="340" Width ="450"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    x:Class="Yingbao.Chapter14.TransformGroupTextBox">
  <Border Margin="10" BorderBrush="Black" BorderThickness="1"
         Background="{StaticResource YingbaoBlueGridBrush}"
         HorizontalAlignment="Left">
    <Canvas ClipToBounds="True" Width="380" Height="250">
      <TextBlock Canvas.Top="80" Canvas.Left="100"
         Width="300" Height="70" Foreground ="Red"
         Background ="LightBlue" Text ="转换字符串1"
         FontSize ="30" Opacity ="0.7">
        <TextBlock.RenderTransform>
          <TransformGroup >
            <ScaleTransform ScaleY ="1.5"  ScaleX="0.8"
               CenterX="20" CenterY ="20"/>
            <SkewTransform AngleX ="-45" CenterX ="20"/>
          </TransformGroup>
        </TextBlock.RenderTransform>
      </TextBlock>
      <TextBlock Canvas.Top="80" Canvas.Left="100"  Width="300"
         Height="70" Foreground ="Red" Background ="Blue"
         Text ="转换字符串2"  FontSize ="30" Opacity ="0.4">
        <TextBlock.RenderTransform>
          <TransformGroup >
            <SkewTransform AngleX ="-45" CenterX ="20"/>
            <ScaleTransform ScaleY ="1.5"  ScaleX="0.8"
                          CenterX="20" CenterY ="20"/>
          </TransformGroup>
        </TextBlock.RenderTransform>
      </TextBlock>
    </Canvas>
  </Border>
</StackPanel >
```

上面的XAML，笔者对同一个TextBlock进行了两次不同的转换组合。这两个组合转换的唯一不同是调换了SkewTransform和ScaleTransform在组合转换中出现的次序。在矩阵转换中，调换次序后，转换出来的结果是不一样的，如图14-8所示。

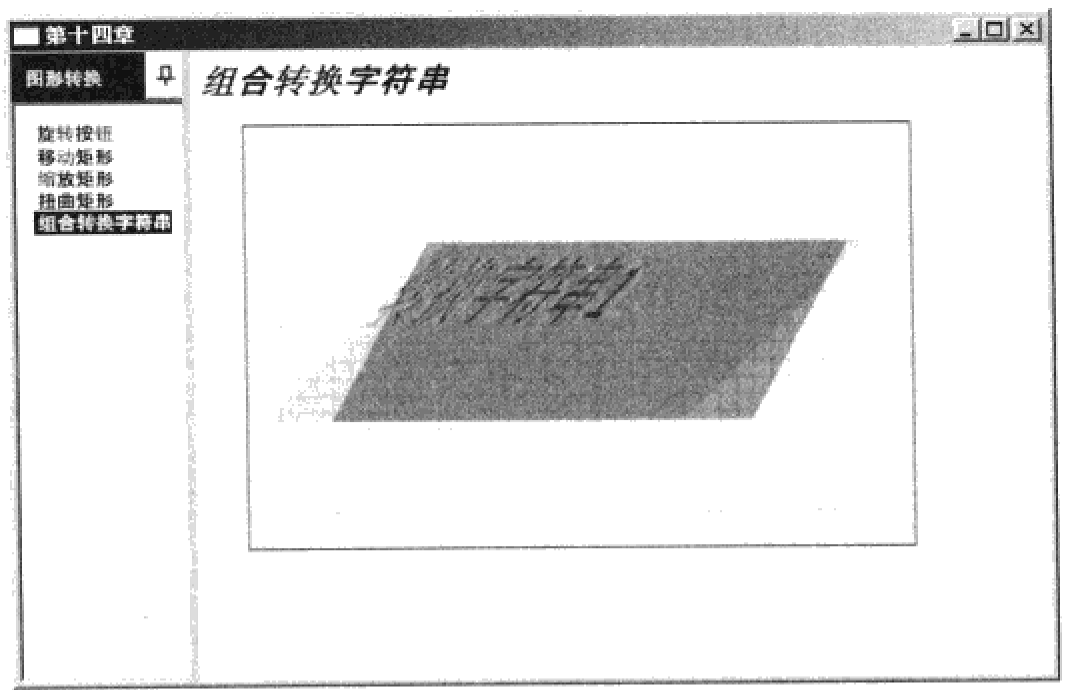


图14-8 改变组合转换中的次序，得出的结果是不同的

14.8 矩阵转换（MatrixTransform）

前面讨论的任何一种图形转换，都可以用矩阵变换来进行。实际上，图形的旋转、位移、缩放、扭曲和组合等转换在WPF中都是通过矩阵转换来实现的。如果直接使用矩阵转换，不仅可以实现复杂的变换，而且可以提高运行速度。本节需要用到线性代数里的内容，读者若不需要深入理解WPF的图形转换，或在实际编程中，前面的几种简单变换就可以满足要求，可以略过本节。

14.8.1 矢量操作

平面中的点$P1(x1,y1)$，可以用一个矢量来表示（如图14-9所示）：

Vector *V1* = new Vector(*x1*,*y1*);

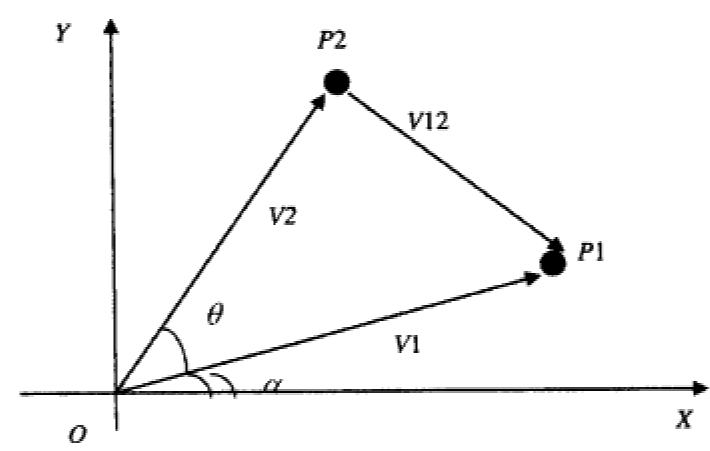


图14-9 平面中的点和距离

其中$(x1,y1)$是点$P1$的坐标。

Vector *V2*= new Vector(*x2*, *y2*);

Vector *V12*= *V1*- *V2*;

矢量$V12$表示图14-9中连接$P1$和$P2$两点的矢量。

矢量在极坐标中可以用矢量的模和相对于极轴的角度来表示。.NET中的Vector类提供了这些基本操作：

- Length属性：返回矢量的模；
- LengthSquared属性：返回矢量模的平方。
- Add 方法：把一个矢量和一个点或另一个矢量相加。
- Substract方法：用一个矢量减去另一个矢量，得到的是一个矢量。
- Multiply方法：把一个矢量和浮点数、矩阵或另一个矢量相乘，得到的是一个矢量。WPF提供了四个重载。
- Divide方法：把矢量除以一个浮点数，得到一个矢量。
- AngleBetween：返回两个矢量见到的角度。
- Normalize：返回单位矢量（模为1）。
- CrossProduct：返回两个矢量交叉相乘的结果，得到一个浮点数。

例如：

```
Vector V1= new Vector( 10, 20 );
Vector V2 = new Vector( 15, 40 );
Double crossPruduct = Vector.CrossProduct( V1, V2 );
```

结果为：

CrossProduct = $V1.X * V2.Y - V1.Y * V2.X = 10*40 - 20*15 = 100$

14.8.2 H坐标系

在图形系统中，使用H坐标系（Homogeneous Coordinates)。这个坐标系首先由August Ferdinand Möbius（见维基百科有关Möbius的条目：http://en . wikipedia.org/wiki/August_Ferdinand_M%C3%B6bius）在1827年引入的。使用这个坐标系，可以很方便地对图形进行任意矩阵转换，这种转换方法实际上适用于任意维数的控件图形，这里仅限于讨论二维图形，即在平面中的情形。

平面中的任意一点$P(x, y)$在H坐标系中用(X, Y, W)来表示，即增加了一W维。若W不为零，把这个点写作（X/W, Y/W, 1)。H坐标系中的点（X/W, Y/W, 1)是直角坐标系中的点（x,y）在H坐标系中的映射。显然这种映射是一一对应的。

为什么要引入H坐标系呢？让我们来看平面中的任意一点$P1$（$x1$, $y1$），现在要把该点移动到$P2(x2, y2)$：

$$x2 = x1 + a \quad \text{(即在}X\text{方向移动}a\text{)} \tag{14-1}$$

$$y2 = y1 + b \quad \text{(即在}Y\text{方向移动}b\text{)} \tag{14-2}$$

那么，在H坐标系中，这个移动操作可以用一个矩阵来表示：

$$(x2\ \ y2\ \ 1) = (x1\ \ y1\ \ 1)\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ a & b & 1 \end{pmatrix} \qquad (14\text{-}3)$$

式（14-3）和式（14-1）及式（14-2）是等价的。我们还可以用另一种方式来做同样的变换：

$$\begin{pmatrix} x2 \\ y2 \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & a \\ 0 & 1 & b \\ 0 & 0 & 1 \end{pmatrix}\begin{pmatrix} x1 \\ y1 \\ 1 \end{pmatrix} \qquad (14\text{-}4)$$

式（14-4）和式（14-3）也是等价的，所以从实用的角度来说，只要支持一种即可。WPF采用式（14-3）形式的变换。我们把式（14-3）和式（14-4）中的矩阵叫位移变换矩阵。

14.8.3 位移转换矩阵

WPF中的位移变换矩阵具有下面的形式：

$$T(a,b) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ a & b & 1 \end{pmatrix} \qquad (14\text{-}5)$$

这样把点$P1(x1,y1)$在H坐标系中变换到点$P2(x2,y2)$，可以简记作：

$$P2(x2,y2) = P1(x1,y1)*\ T(a, b) \qquad (14\text{-}6)$$

其中a和b分别是在X轴和Y轴方向移动的距离。在需要连续位移变换时，可以先计算出位移变换矩阵，再调用WPF的矩阵变换，这样可以提高实时性能。例如，把点$P2(x2,y2)$在X轴方向移动$a1$，在Y轴方向移动$b1$，从而变换到点$P3$（$x3,y3$），我们可以先算出位移变换矩阵：

$$\begin{aligned} P3(x3,y3) &= P2(x2,y2)*T(a1,b1) \\ &= P1(x1,y1)*T(a,b)*T(a1,b1) \\ &= P1(x1,y1)\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ a & b & 1 \end{pmatrix}\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ a & b & 1 \end{pmatrix} \\ &= P1(x1,y1)\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ a+a1 & b+b1 & 1 \end{pmatrix} \\ &= P1(x1,y1)*T(a+a1,\ b+b1) \end{aligned}$$

14.8.4 旋转转换矩阵

图14-9示出了矢量$V1$逆时针旋转θ度变换为$V2$的情况。若矢量的模为r，那么矢量$V1$的坐标为：

$x1 = r \cos x$

$y1 = r \sin\alpha$

矢量$V2$的坐标为：

$x2 = r \cos(\alpha+\theta) = r \cos x \cos\theta - r \sin\alpha \sin\theta = x1\cos\theta - y1 \sin\theta$

$y2 = r \sin(\alpha+\theta) = r \sin\alpha \cos\theta + r \cos x \sin\theta = x1 \sin\theta + y1 \cos\theta$

如果旋转中心不在原点，而在（cx,cy）处，则上述变换应写为：

$x2=(x1-cx)\cos\theta - (y1-cy) \sin\theta + cx = x1 \cos\theta - y1\sin\theta - cx \cos\theta + cy \sin\theta + cx$

$y2 = (x1-cx) \sin\theta + (y1-cy) \cos\theta + cy = x1 \sin\theta + y1 \cos\theta - cx \sin\theta - cy \cos\theta + cy$

上面的式子实际上，是先把点（cx,cy）到点（$x1,y1$）的向量平移到原点，再旋转，旋转完了之后，再把该矢量平移到（cx,cy）处。

由此，我们可以写出旋转变换矩阵：

$$R(\theta) = \begin{pmatrix} \cos\theta & \sin\theta & 0 \\ -\sin\theta & \cos\theta & 0 \\ -cx\cos\theta + cy\sin\theta + cx & -cx\sin\theta - cy\cos\theta + cy & 1 \end{pmatrix} \qquad (14\text{-}7)$$

在H坐标系中，矢量$V2$可以用旋转变换矩阵表示为：

$$(x2, y2, 1) = (x1, y1, 1)\begin{pmatrix} \cos\theta & \sin\theta & 0 \\ -\sin\theta & \cos\theta & 0 \\ -cx\cos\theta + cy\sin\theta + cx & -cx\sin\theta - cy\cos\theta + cy & 1 \end{pmatrix}$$

或简单地记为：

$P2(x2,y2)=P1(x1,y1) * R(\theta)$

这样便可以容易地推出连续作旋转变换的变换矩阵。

14.8.5 缩放转换矩阵

类似地，可以推导出缩放变换的变换矩阵为：

$$S(px,py) = \begin{pmatrix} px & 0 & 0 \\ 0 & py & 0 \\ (1-px)cx & (1-py)cy & 1 \end{pmatrix} \qquad (14\text{-}8)$$

其中，px和py分别为X方向和Y方向的缩放倍数，(cx, cy)为缩放相对的中心点。

14.8.6 扭曲转换矩阵

同样，可以写出扭曲转换的转换矩阵：

$$Sk = \begin{pmatrix} 1 & \tan\alpha y & 0 \\ \tan\alpha x & 1 & 0 \\ -\tan\alpha x\ Cx & -\tan\alpha y\ Cy & 1 \end{pmatrix} \qquad (14\text{-}9)$$

其中αx是X方向的旋转角度，αy是Y方向的旋转角度，Cx和Cy为扭曲转换的中心点。这样，点$P1(x1,y1)$通过扭曲转换到点$P2(x2,y2)$可以写为：

$$(x2, y2, 1) = (x1, y1, 1)\begin{pmatrix} 1 & \tan\alpha y & 0 \\ \tan\alpha x & 1 & 0 \\ -\tan\alpha x\ Cx & -\tan\alpha y\ Cy & 1 \end{pmatrix}$$

由此可见，前面讨论的多种转换实际上都可以用矩阵转换来表示。换句话说，矩阵转换是图形转换的一般形式。

有了H坐标系中的矩阵转换，就可以很容易地证明先缩放再旋转和先旋转再缩放所得到的结果为什么是不一样的。

先缩放再旋转的转换矩阵为：

$$R(\theta) * S(px, py) \qquad (14\text{-}10)$$

先旋转再缩放的转换矩阵为：

$$S(px, py) * R(\theta) \qquad (14\text{-}11)$$

若我们把式（14-7）和式（14-8）带入式（14-10）和式（14-11），得出的结果是不一样的。这就是为什么在组合转换的例子中，转换的结果和转换的次序有关。更一般地，在线性代数中，矩阵相乘没有交换律。

14.8.7 矩阵操作

WPF提供了基于H坐标系的矩阵类Matrix。由于H坐标系中，矩阵的最后一列总是（0 0 1），为简化起见，WPF的Matrix类定义一个3×2矩阵，这个矩阵的六个元素表示为（M11、M12、M21、M22、Offset*X*、Offset*Y*）。M的第一个下标为矩阵中的行号，第二个下标为矩阵中的列号。Offset*X*处在第三行第一列；Offset*Y*处在第三行的第二列。之所以把这两个元素叫Offset*X*和Offset*Y*是因为这两个元素的值总是把图形向*X*方向移动Offset*X*个单位，向*Y*方向移动Offser*Y*个单位（参见式14-5中 a和b的位置）。

WPF中的Matrix类提供下属矩阵操作：

- 求一个矩阵的逆矩阵，Invert。

我们知道，对于矩阵A：若：

$$A*B = 1$$

则矩阵*B*称为矩阵*A*的逆矩阵，显然，*B*是*A*的逆矩阵，*A*不一定是*B*的逆矩阵。从线性代数我们知道，不是所有的矩阵都有逆矩阵，所以在调用Invert之前，需要判断矩阵是否可逆，如：

```
Matrix m1 = new Matrix(10, 5, 20, 30, 8, 40);
  if (m1.HasInverse)
  {
      m1.Invert();
  }
```

我们得到m1的逆矩阵为：(0.15,-0.025,-0.1,0.05,2.8,-1.8)。

- 两个矩阵相乘：Multiply(*m*1, *m*2)。该方法把两个矩阵*m*1和*m*2相乘，得到一个新的矩阵。
- 旋转矩阵：Rotate(angle)。该方法把一个矩阵旋转一个角度angle（相对于该矩阵的原点）。
- 相对某个点旋转矩阵：Rotate*At*(angle,center*X*,center*Y*)。该方法把矩阵相对于点（center*X*,center*Y*)旋转一个角度(angle)。
- 放大/缩小：Scale(Scale*X*,Scale*Y*)，把一个矩阵在*X*方向和*Y*方向分别放大Scale*X*和Scale*Y*倍。相对于原点。
- 相对某个点放大/缩小矩阵：Scale*At*。
- 位移操作：Translate（a, b)。在*X*方向移动*a*，在*Y*方向移动*b*。
- 扭曲操作：Skew(Skew*X*，Skew*Y*)，其中Skew*X*和Skew*Y*分别为在*X*方向和*Y*方向扭曲的角度。
- 转换：Transform，该方法有四个重载，可用于转换一点、多点、一个矢量和多个矢量。

我们知道，矩阵运算的次序很重要。上面的旋转、缩放、位移、扭曲操作都是把转换矩阵右乘所要变换的矩阵，Matrix还提供了左乘操作，这就是Prepend、RotatePrepend，RetateAtPrepend、ScalePrepend、ScaleAtPrepend和SkewPrepend方法。在多重转换中，有时需要对前一转换的结果实行左乘操作。

现在让我们来看一个简单的例子，这个例子直接改变矩阵转换中的元素。

```
<DockPanel xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        x:Class="Yingbao.Chapter14.MatrixTransform" Height ="500"
Width ="450">
  <Grid DockPanel.Dock="Top">
    <Grid.ColumnDefinitions>
      <ColumnDefinition Width="Auto"/>
      <ColumnDefinition Width="Auto"/>
      <ColumnDefinition Width="Auto"/>
      <ColumnDefinition Width="Auto" />
      <ColumnDefinition Width="Auto" />
    </Grid.ColumnDefinitions>
    <Grid.RowDefinitions>
      <RowDefinition Height="Auto"/>
      <RowDefinition Height="Auto"/>
      <RowDefinition Height="Auto"/>
      <RowDefinition Height="Auto"/>
      <RowDefinition Height="Auto"/>
```

```
</Grid.RowDefinitions>
<Label Grid.Row="1" Grid.Column="0">M11</Label>
<TextBox Grid.Row="1" Grid.Column="1"
   Name="M11TextBox">1.0</TextBox>
<Label Grid.Row="2" Grid.Column="0">M12</Label>
<TextBox Grid.Row="2" Grid.Column="1"
   Name="M12TextBox">0.0</TextBox>
<Label Grid.Row="1" Grid.Column="2">M21</Label>
<TextBox Grid.Row="1" Grid.Column="3"
   Name="M21TextBox">0.0</TextBox>
<Label Grid.Row="2" Grid.Column="2">M22</Label>
<TextBox  Grid.Row="2" Grid.Column="3"
   Name="M22TextBox">1.0</TextBox>
<Label Grid.Row="3" Grid.Column="0">OffsetX</Label>
<TextBox Grid.Row="3" Grid.Column="1"
   Name="OffsetXTextBox">0.0</TextBox>
<Label Grid.Row="3" Grid.Column="2">OffsetY</Label>
<TextBox Grid.Row="3" Grid.Column="3"
   Name="OffsetYTextBox">0.0</TextBox>
<Button Grid.Row="4" Grid.Column="0"
  Click="applyButtonClicked">转换</Button>
<Grid Grid.Column="4" Grid.Row="1" Grid.RowSpan="4"
  Margin="10,0,10,0">
  <Grid.Resources>
    <Style TargetType="{x:Type TextBlock}">
      <Setter Property="FontFamily" Value="Courier New"/>
      <Setter Property="FontWeight" Value="Bold"/>
      <Setter Property="FontSize" Value="12"/>
      <Setter Property="HorizontalAlignment"
        Value="Stretch"/>
    </Style>
  </Grid.Resources>
  <Grid.ColumnDefinitions>
    <ColumnDefinition Width="Auto" />
    <ColumnDefinition Width="Auto" />
    <ColumnDefinition Width="Auto" />
    <ColumnDefinition Width="Auto" />
    <ColumnDefinition Width="Auto" />
  </Grid.ColumnDefinitions>
  <Grid.RowDefinitions>
    <RowDefinition Height="Auto" />
    <RowDefinition Height="Auto" />
    <RowDefinition Height="Auto" />
    <RowDefinition Height="Auto" />
    <RowDefinition Height="Auto" />
    <RowDefinition Height="Auto" />
  </Grid.RowDefinitions>
  <Rectangle
    Grid.Row="0" Grid.Column="5"
    Grid.RowSpan="6" Width="1" Fill="Black" />
  <Rectangle Grid.Row="0" Grid.Column="0"
    Grid.RowSpan="6" Width="1" Fill="Black" />
  <Rectangle Grid.Row="0" Grid.Column="0"
```

```
        Grid.ColumnSpan="2" Height="1"
        Width="10" Fill="Black"
        HorizontalAlignment="Left"/>
      <Rectangle
        Grid.Row="0" Grid.Column="3" Width="10"
        Grid.ColumnSpan="2" Height="1" Fill="Black"
        HorizontalAlignment="Right"/>
      <Rectangle Grid.Row="5" Grid.Column="0" Width="10"
        Grid.ColumnSpan="2" Height="1" Fill="Black"
        HorizontalAlignment="Left" />
      <Rectangle Grid.Row="5" Grid.Column="3"
        Width="10"
        Grid.ColumnSpan="2" Height="1" Fill="Black"
        HorizontalAlignment="Right"/>
      <TextBlock   Grid.Row="1" Grid.Column="1" Margin="10">
        <TextBlock.Text>
          <Binding ElementName="M11TextBox" Path="Text"/>
        </TextBlock.Text>
      </TextBlock>
      <TextBlock  Grid.Row="2" Grid.Column="1" Margin="10">
        <TextBlock.Text>
          <Binding ElementName="M12TextBox" Path="Text"/>
        </TextBlock.Text>
      </TextBlock>
      <TextBlock   Grid.Row="1" Grid.Column="2" Margin="10">
        <TextBlock.Text>
          <Binding ElementName="M21TextBox" Path="Text"/>
        </TextBlock.Text>
      </TextBlock>
      <TextBlock     Grid.Row="2" Grid.Column="2" Margin="10">
        <TextBlock.Text>
          <Binding ElementName="M22TextBox" Path="Text"/>
        </TextBlock.Text>
      </TextBlock>
      <TextBlock  Grid.Row="3" Grid.Column="1" Margin="10">
        <TextBlock.Text>
          <Binding ElementName="OffsetXTextBox" Path="Text"/>
        </TextBlock.Text>
      </TextBlock>
      <TextBlock   Grid.Row="3" Grid.Column="2" Margin="10">
        <TextBlock.Text>
          <Binding ElementName="OffsetYTextBox" Path="Text"/>
        </TextBlock.Text>
      </TextBlock>
      <TextBlock  Grid.Row="1" Grid.Column="3"
           Margin="10">0.0</TextBlock>
      <TextBlock  Grid.Row="2" Grid.Column="3"
           Margin="10">0.0</TextBlock>
      <TextBlock  Grid.Row="3" Grid.Column="3"
           Margin="10">1.0</TextBlock>
    </Grid>
  </Grid>
  <Border BorderBrush="Black" BorderThickness="1"
```

```
    HorizontalAlignment="Center"
    Margin="0,10,0,10">
    <Canvas Height="360" Width="400"
      Background="{StaticResource YingbaoBlueGridBrush}">
      <Rectangle Canvas.Top="100" Canvas.Left="100"
        Fill="LightBlue" Stroke="Black" StrokeThickness="2"
        StrokeDashArray="2,1"
        Height="100" Width="100">
      </Rectangle>
      <TextBlock Canvas.Top="80" Canvas.Left="80">初始位置
          </TextBlock>
      <Rectangle Name="transformedRectangle"
        Canvas.Top="100" Canvas.Left="100"
        Fill="Red" Stroke="{StaticResource YingbaoBlueGridBrush}"
        StrokeThickness="2"
        Height="100" Width="100" Opacity="0.50">
        <Rectangle.RenderTransform>
          <MatrixTransform x:Name="myMatrixTransform"/>
        </Rectangle.RenderTransform>
      </Rectangle>
    </Canvas>
  </Border>
</DockPanel >
```

下面是转换按钮的事件处理程序：

```
namespace Yingbao.Chapter14
{
     public partial class MatrixTransform :
                    System.Windows.Controls.DockPanel
    {
        public MatrixTransform()
        {
            InitializeComponent();
        }
        private void applyButtonClicked(object sender,
                 EventArgs args)
        {
            updateMatrixTransform();
        }
        private void updateMatrixTransform()
        {
            Matrix myMatrix = new Matrix();
            myMatrix.M11 = Double.Parse(M11TextBox.Text);
            myMatrix.M12 = Double.Parse(M12TextBox.Text);
            myMatrix.M21 = Double.Parse(M21TextBox.Text);
            myMatrix.M22 = Double.Parse(M22TextBox.Text);
            myMatrix.OffsetX = Double.Parse(OffsetXTextBox.Text);
            myMatrix.OffsetY = Double.Parse(OffsetYTextBox.Text);
            myMatrixTransform.Matrix = myMatrix;
        }
    }
}
```

当按下“转换”按钮时，updateMatrixTransform方法首先读取界面上用户输入的数值，然后把这些数值放到矩阵的相应位置，最后对矩形进行转换。当在TextBlock中输入数值时，右边矩阵中相应位置的元素也会跟着变化。这样可以直观地调整矩阵中的元素值，再观察这些值对转换的影响。图14-10 是使用矩阵转换的结果。这个转换矩阵中含有缩放、位移和旋转等转换。

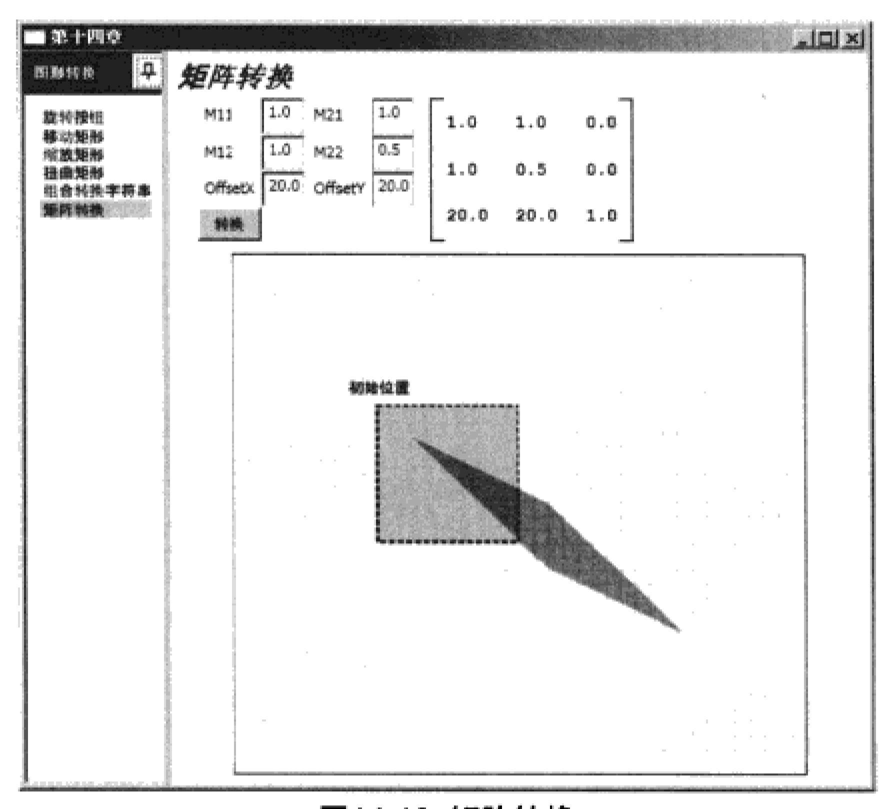


图14-10 矩阵转换

使用矩阵转换，再结合调用Matrix类所提供的方法，可对界面元素进行任意转换。我们不能在XAML中使用Matrix类，所以要用Matrix类所提供的功能，C#或Visual Basic语言是必须的。

14.9 本章小结

本章系统地讨论了WPF中界面元素的转换技术，RenderTransform图形转换是对界面中的单个元素进行的：对某个元素进行的转换，不会对其他元素有任何影响。LayoutTransform对排版元素中所包含的所有元素都有影响。简单的转换包括位移、旋转、缩放、扭曲及这些转换的组合，转换的结果和转换的次序有关。

所有的图形转换最后都归结于H坐标系下的矩阵运算，使用矩阵转换速度快，灵活但必须在C#或Visual Basic 中进行。图形转换技术是实现动画的基础，第15章将详细讨论WPF中的动画技术。

第15章 动 画

本章将讨论WPF中的动画技术，提起动画，大部分人会联想到卡通片，我们知道卡通片的制作是把相关图片放在特定的时间点上，当这些离散的图片快速从我们眼前通过时，在我们头脑中产生的印象就是连贯的动作。一般来说，可以认为动画实际上是把连续变量离散化，再把离散的信息连续播放的过程。

动画的对象（变量）具有可连续变化的特性时，我们才会看到连贯的渐变效果。若动画对象本身是离散的，如开关量，那么我们看到的只是在某个时间点上该变量的值。

好莱坞制作电影时，为了让整个团队了解所要拍摄的电影，导演常常把电影中的故事情节分成一个个片段写在卡片上，然后把卡片按照情节出现的时间顺序贴在墙板上。这个用于张贴故事卡片的面板就叫做故事版（Storyboard），现在软件工程中常用的Agile方法，也使用故事版，WPF中的Storyboard类就来源于此。

15.1 WPF中的动画

WPF中的动画不一定是位置的变化，它可以是任意相关属性的值随时间的变化。总的说来，可进行动画的相关属性需要满足下面两个条件：

- 该相关属性所在的类必须是 DependencyObject的派生类，并且移植了IAnimatable接口；
- 该相关属性的数据类型是WPF动画所支持的数据类型。

WPF中许多类的相关属性都满足上面这两个条件，WPF Framework对动画的支持从控件到几何图形、从排版到风格和模板、几乎所有的对象都可以进行动画。

有意思的是满足上述两个条件的相关属性在用户界面上不一定是可见的，这给动画对象提供了广阔的空间。

WPF中用于支持动画的类大多位于System.Windows.Media.Animation名称空间中，共有200多个类。这些类可以分为几大类别，对每一种数据类型的动画WPF提供了专门的类，表15-1列出了22种数据类型及其相关动画类。从表15-1可以看出，这些类的命名遵循严格的规范。 动画类的名字是在数据类型的后面加上Animation后缀；KeyFrame类的名字则在数据类型的后面加上AnimationUsingKeyFrame。例如Int16数据类型，其动画类的类型为Int16Anima-tion；其相应的KeyFrame类的类型为Int16AnimationUsingKeyFrame。

WPF有两种方式制作动画，一种是直接使用Animation类，一种是使用AnimationUsingKeyFrame类。AnimationUsingKeyFrame可以精确地指定相关属性在某个时刻的值，Expression blend只使用这种方式。若数据类型不能连续变化，则不存在动画类，只有KeyFrame类，如Bool、Char等。数据类型和动画类的具体情况如表15-1所示。

表15-1 数据类型和动画类

数据类型	动画类	KeyFrame类
Bool		BooleanAnimationUsingKeyFrame
Byte	ByteAnimation	ByteAnimationUsingKeyFrame
Char		CharAnimationUsingKeyFrame
Decimal	DecimalAnimation	DecimalAnimationUsingKeyFrame
Int16	Int16Animation	Int16AnimationUsingKeyFrame
Int32	Int32Animation	Int32AnimationUsingKeyFrame
Int64	Int64Animation	Int64AnimationUsingKeyFrame
Single	SingleAnimation	SingleAnimationUsingKeyFrame
Double	DoubleAnimation	DoubleAnimationUsingKeyFrame
String		StringAnimationUsingKeyFrame
Object		ObjectAnimationUsingKeyFrame
Thickness	ThicknessAnimation	ThicknessAnimationUsingKeyFrame
Color	ColorAnimation	ColorAnimationUsingKeyFrame
Size	SizeAnimation	SizeAnimationUsingKeyFrame
Rect	RectAnimation	RectAnimationUsingKeyFrame
Point	PointAnimation	PointAnimationUsingKeyFrame
Point3D	Point3DAnimation	PointAnimationUsingKeyFrame
Vector	VectorAnimation	VectorAnimationUsingKeyFrame
Vector3D	Vector3DAnimation	Vector3DAnimationUsingKeyFrame
Rotation3D	Rotation3DAnimation	Rotation3DAnimationUsingKeyFrame
Matrix		MatrixAnimationUsingKeyFrame
Quaternion	QuaternionAnimation	QuaternionAnimationUsingKeyFrame

15.2 动画类继承树

表15-1列出的动画类，在继承树上都有共同的特点：它们都是从Animation-TimeLine中派生出来的。AnimationTimeLine的直接派生类以Base为后缀，如Int16数据类型，Int16AnimatonBase从AnimationTimeLine中直接派生出来，Int16AnimationBase派生出两个类，即Int16Animation和Int16AnimationUsingKeyFrame。有些数据类型支持沿着某一几何轨迹的动画，这个时候需要用第三个类：AnimationUsingPath。WPF中共有3个类型支持AnimationUsingPath，如DoubleAnimationUsingPath。

动画类的典型类继承树结构如图15-1所示。在这里笔者用的例子是Double数据类型，对于其他的数据类型，其继承树的结构是类似的。

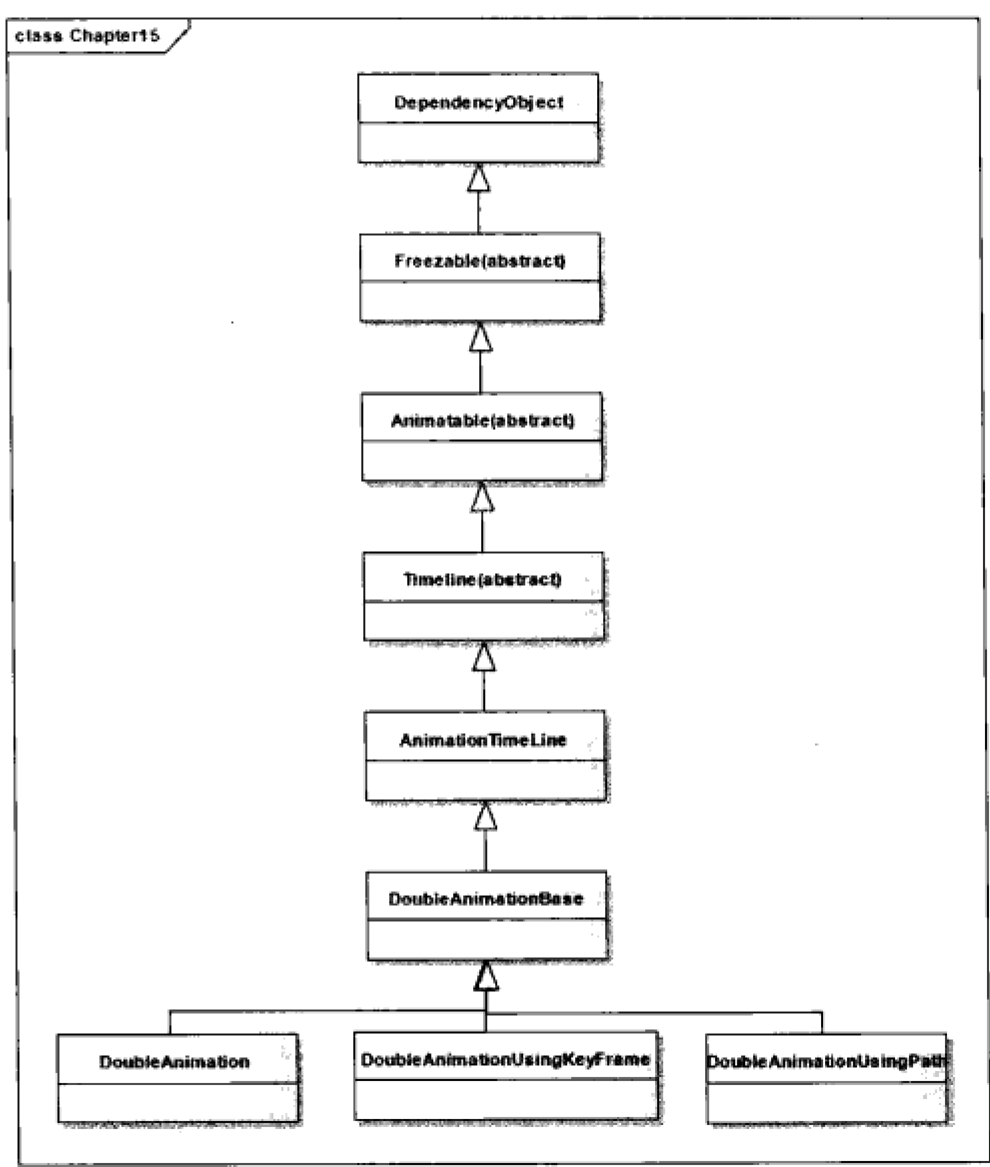


图15-1 典型动画类结构

15.3 一个简单的动画

下面是一个简单动画的例子，Window对象里面含有一个Canvas，Canvas中有一个按钮控件：

```
<Window x:Class="Yingbao.Chapter15.UseAnimation.Window1"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="视窗" Height="136" Width="242">
    <Canvas Height="84" Name="canvas1" Width="213">
        <Button Canvas.Left="88" Canvas.Top="12" Height="43"
                Name="myButton" Width="112">动画按钮</Button>
    </Canvas>
</Window>
```

笔者选按钮的宽度Width属性进行动画，由于Width为double类型，所以要用DoubleAnimation类。把宽度的动画范围设为50到100之间，所需的时间设为30秒。为说明触发动画的条件，为此，笔者在C#中对Window类中的OnMouseDoubleClick虚函数进行了覆盖，即当在窗口内双击鼠标时开始动画。

```
namespace Yingbao.Chapter15.UseAnimation
{
```

```
public partial class Window1 : Window
    {
        public Window1()
        {
            InitializeComponent();
        }
        protected override void OnMouseDoubleClick(
                             MouseButtonEventArgs e)
        {
            base.OnMouseDoubleClick(e);
            DoubleAnimation da = new DoubleAnimation();
            da.From = 50;
            da.To = 100;
            da.Duration = new Duration(TimeSpan.Parse("0:0:30"));
            myButton.BeginAnimation(Button.WidthProperty, da);
        }
    }
}
```

这个例子虽然简单，但说明了WPF写动画程序常用的下列步骤：

- 选定动画对象；
- 选定动画对象中的相关属性；
- 根据相关属性的数据类型，选定相应的动画类，如本例中的Width为double类型，我们用DoubleAnimation；
- 设置动画属性，如本例中的From、to等；
- 选择触发动画的条件，如本例中用双击鼠标开始动画。
- 调用所要动画对象中的BeginAnimation方法，该方法把DoubleAnimation和按钮的WidthProperty联系了起来

运行该程序，可以使你在窗口内双击鼠标的按钮时，按钮的宽度在30秒的时间内从50变到100。除了From和To属性之外，动画类还提供了其他的一些属性，使用这些属性，我们可以更好地控制动画的进程。

15.4 控制动画

动画的很多特性都是由TimeLine类控制的，它处在动画类继承树靠近树根的位置。所谓TimeLine是指一个时间片，这个类提供了动画常用的属性，如动画时间、开始时刻、动画重复次数、动画终止的状态等。TimeLine类中的属性如表15-2所示：

表15-2 TimeLine类中的属性

属性名	功能描述
AccelerationRatio	加速比率
AutoReverse	自动回到初始状态
BeginTime	指定动画开始的时间
DecelerationRatio	减速比率，与加速比率相对
Duration	动画经过的时间

续表

属性名	功能描述
FillBehavior	设定动画最终状态
RepeatBehavior	动画重复特性
SpeedRadio	加速比例

15.4.1 动画所用的时间（duration）

Duration用来指定某个相关属性动画从开始到结束所用的时间，如前面的例子：

```
da.Duration = new Duration(TimeSpan.Parse("0:0:5"));
```

一般使用TimeSpan来说明动画的时间，其格式如下：

```
days.hours:minutes:seconds.fraction.
```

注意在日和小时之间使用的是点“.”，在时分秒之间使用的是“:”，秒以下又是点“.”。常用动画通常在分秒之间，可以使用简化的形式，如5秒钟可以写为：

“0：0：5”

但不能把3.5秒写成“3.5”而要写成“0：0：3.5”，因为“3.5”表示3天零5小时。

虽然在大多情况下，Duration用TimeSpan来表达，但是Duration和TimeSpan并不是一个类型。Duration除了用TimeSpan表示具体的动画时间之外，还有两个特殊值：Automatic和Forever。Automatic是Duration的默认值，其值为1秒，所以若不设定Duration，则WPF为动画提供的默认Duration为1 秒。Forever主要用在故事板Storyboard中，笔者后面还会用到。

15.4.2 设定动画开始时间BeginTime

在前面的例子中，我们在双击鼠标事件发生的时候立即开始动画，即BeginTime设为为0（默认值）。实际上，也可以设定动画在某个时间开始，例如可以让动画在双击鼠标事件后两秒开始：

```
da.BeginTime = TimeSpan.Parse("0:0:2");
```

BeginTime的类型为TimeSpan，和前面所述的用法完全相同。

我们可以把BeginTime设为负数，这时动画相当于提前到某个时刻开始，如

```
da.BeginTime = TimeSpan.Parse("-0:0:1");
```

相当于双击鼠标事件前1秒动画已经开始。

15.4.3 设定自动返回（AutoReverse）

WPF动画提供了前进和后退功能。如果AutoReverse设为False，那么动画只向前进行，如果AutoReverse设为True，那么动画在完成前进的过程后，自动以相同的方式后退到初始状态。

比如前面的例子:

```
DoubleAnimation da = new DoubleAnimation();
da.From = 50;
da.To = 100;
da.Duration = new Duration(TimeSpan.Parse("0:0:30"));
```

```
da.AutoReverse = true;
myButton.BeginAnimation(Button.WidthProperty, da);
```

按钮宽度从50变到100要用30秒，笔者把da.AutoReverse之后，经过30秒，按钮的宽度又回到50，这就是AutoReserve的动画效果。

15.4.4 设定动画速度（SpeedRatio）

当需要对两个以上对象进行类似动画时，可以微调同一个动画类，并把它们用在不同的对象上。例如，可以用同一个DoubleAnimation对椭圆的宽度进行动画，但把动画的速度增加一倍，那么，可以简单地用下面的C#来完成：

```
da.SpeedRatio = 2;
ellipse1.BeginAnimation(Ellipse.WidthProperty, da);
```

在这里笔者先把SpeedRatio设为2，然后把该动画加到椭圆上，运行该程序，可以看到，椭圆宽度动画的速度比按钮宽度快一倍。

当SpeedRatio>1时，动画的速度加快；当SpeedRatio<1时，动画的速度变慢。

15.4.5 加快和减慢动画（AccelarationRatio和DecelarationRatio）

在默认的情况下，WPF对某个相关属性所进行的动画是均匀的，即在某个时刻该相关属性的值，随着时间的变化而线性地变化。比如按钮的宽度从50变到100，Duration设为5秒，那么我们可以算出按钮的宽度在任意时刻的值为：

$$Width = [(100-50)/5]\, t + 50$$

其中t为时间，其取值在0~5秒之间。

为了更好地模拟现实，WPF提供了非线性变化相关属性的动画，对于更复杂的情况，要等到讲述KeyFrame技术时讨论。对于简单的情况，可以使用AccelarationRatio和DecelarationRatio来调整相关属性的变化速度。例如，我们开车时，开始的时候有一个加速过程，中间行驶的时候是一个匀速的过程，到达终点时，我们需要经过先减速到停车的过程。

图15-2是简单动画加速和减速示意图，AccelarationRatio在开始到动画的某个时点作用，DecelarationRatio则是从某个时点到结束作用。假定整个动画所用的全部时间为1，那么，AccelarationRatio和DeccelarationRatio加起来的时间应小于等于1。如果AccelarationRatio为1，那么DecelarationRatio必须为0，这是整个动画过程都在均匀加速的情况；如果DecelarationRatio为1，那么AccelarationRatio的值必须为0，即整个动画过程都在减速。所以AccelarationRatio和DecelarationRatio的值应在0到1之间。

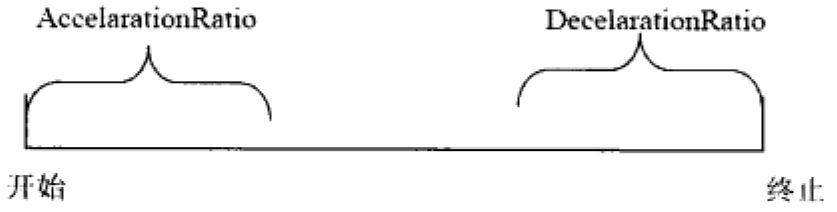


图15-2 简单动画加速和减速

为了有一个直观的认识，一个对汽车动画使用AccelarationRatio和DecelarationRatio来改变加速或减速过程的例子如下。

```
<Window x:Class="Yingbao.Chapter15.UseAnimation.SolarCarAnimation"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="SolarCarAnimation" Height="306" Width="813"
    Background="Crimson">
    <Canvas Width="768">
        <Image Margin="61,75,74,71" Name="image1" Stretch="Fill"
               Source="/Chapter15;component/Image/SolarCar.jpg"
               RenderTransformOrigin="10,0" Height="114"
               Width="163" Canvas.Left="-19" Canvas.Top="-3" />
    </Canvas>
</Window>

namespace Yingbao.Chapter15.UseAnimation
{

    public partial class SolarCarAnimation : Window
    {
        public SolarCarAnimation()
        {
            InitializeComponent();
        }

        protected override void OnMouseDoubleClick(
                        MouseButtonEventArgs e)
        {
            base.OnMouseDoubleClick(e);
            DoubleAnimation da = new DoubleAnimation();
            da.From = 0;
            da.To = 500;
            da.AccelerationRatio = 0.2;
            da.DecelerationRatio = 0.2;
            da.RepeatBehavior = RepeatBehavior.Forever;
            da.AutoReverse = true;
            da.Duration = new Duration(TimeSpan.Parse("0:0:5") );
            image1.BeginAnimation(Canvas.LeftProperty, da);
        }

    }
}
```

这段简单程序的运行结果如图15-3所示。

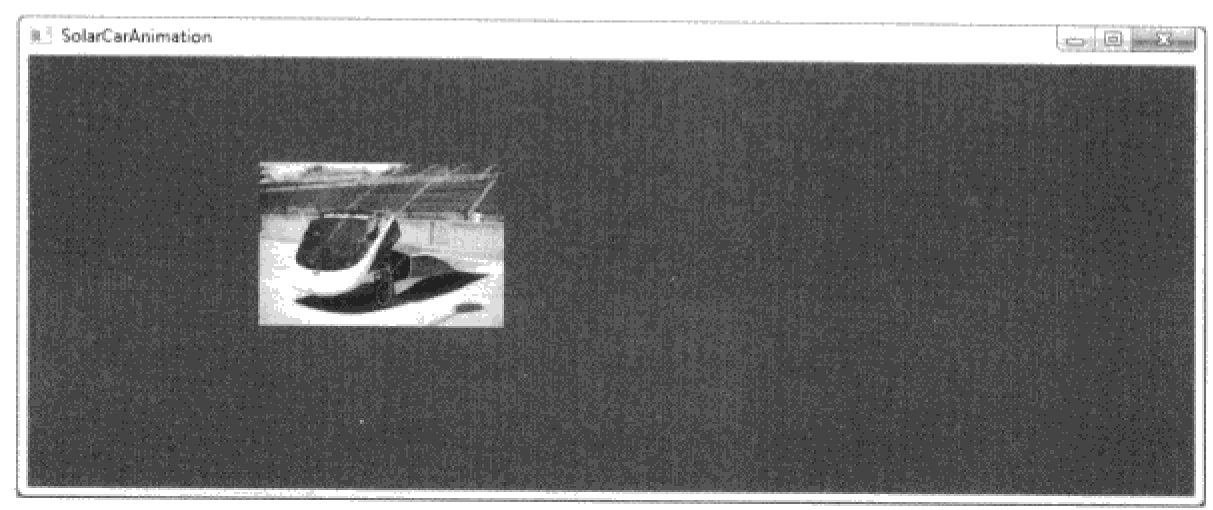


图15-3 太阳能新型汽车动画

15.4.6 设定动画的重复特性（RepeatBehavior）

我们可以指定动画重复的次数，方法是设定RepeatBehavior。有两种方法来设定动画的重复特性，一种是设定重复的次数，例如：

```
da.RepeadBehavior = new RepeatBehavior(4);
```

该语句设定动画重复4次。

第二种方法是设定动画重复到一定的时间，如：

```
da.RepeadBehavior = new RepeatBehavior(TimeSpan.Parse("0:0:10"));
```

上述语句的意思是，把动画一直重复到10秒。如果动画的持续时间是1秒，上面这句C#实际上把动画重复了10次；如果动画持续时间是2秒，那么，同样的这句C#只把动画重复5次。可以想象，如果动画持续的时间大于10秒，那么这句C#使得动画不能做完。

15.4.7 设定动画的终止状态（FillBehavior）

那么动画结束的时候，相应的相关属性的值是保持动画结束时的值还是回到最初的状态？WPF使得这两种结果都有可能，它取决于FillBehavior的值。FillBehavior有两个值：HoldEnd和Stop。若FillBehavior设为HoldEnd（默认值），动画结束时，相关属性的值为动画结束时的值；若FillBehavior设为Stop，动画结束时，相关属性的值为动画开始时的值。

15.4.8 设定相关属性的动画范围（From和To）

WPF中的动画可以分为两类，一类是相关属性可以连续变化的，如Int16Animation、Int32Animation和DoubleAnimation等，另一类是不能连续变化的，如Bool、Char和String这些类型。对于可以连续变化的相关属性的动画，WPF提供设置动画的开始（From）和终止值（To），设定From和To后，WPF自动算出在某个特定的时刻，该相关属性的值。在没有加速或加速因素的情况下，WPF使用简单的线性插值法。

在前面汽车动画的例子中，我们设置了动画的From和To属性。

```
DoubleAnimation da = new DoubleAnimation();
da.From = 0;
```

```
da.To = 500;
image1.BeginAnimation(Canvas.LeftProperty, da);
```

有时候可以省略设置From或To，在这种情况下，WPF自动使用当前相关属性值来作为相应的From或To的值。比如，在汽车动画的例子中，若汽车原来在画布上的位置Left为100，那么，省去From：

```
DoubleAnimation da = new DoubleAnimation();
 //da.From = 0;
da.To = 500;
image1.BeginAnimation(Canvas.LeftProperty, da);
```

汽车动画Left的范围在100~500。若省去To，汽车动画Left的范围在0~100：

```
DoubleAnimation da = new DoubleAnimation();
da.From = 0;
//da.To = 500;
image1.BeginAnimation(Canvas.LeftProperty, da);
```

由此可见，如果动画某个相关属性具有初始值，省略设置From或To时，就把该初始值作为动画的From和To的值。但是，若相关属性并没有初值，动画的From和To是不能省略的！

15.4.9 设定相关属性的动画范围（By）

设定相关属性动画范围的另一个方法是使用By属性，By的意思是在所动画原有值的基础上加（若By为正）或减（若By为负）By值。例如，汽车在画布上的初始位置Left为100，若把By设为500，那么，汽车的动画范围就在100~600：

```
DoubleAnimation da = new DoubleAnimation();
da.By = 500;
image1.BeginAnimation(Canvas.LeftProperty, da);
```

15.4.10 设定IsAdditive和IsCumulative 属性

对于可以连续变化的相关属性，若IsAdditive设为True时，相关属性的动画从From+相关属性的当前值。

当连续变化的相关属性进行多次动画时，若把IsCumulative设为true，则第二次动画的from和To在原有的from和To上加上原来的动画范围。以汽车动画为例：汽车第一次动画Left在0~100内移动， 第二次动画就在100~200内移动，第三次就在200~300内移动，依次类推。IsCumulative有累加效果，这种累加仅在设置动画的RepeatBehavior时才有效果。

15.4.11 WPF动画的时间片类

控制动画的进程主要是安排某个事件在某个时刻发生，安排时间序列就是TimeLine类的工作。WPF提供了丰富的TimeLine类，前面已介绍了Timeline类，这是一个基类，从TimeLine类中派生出三个类，分别为：AnimationTimeLine、TimelineGroup和MediaTimeLine。而TimelineGroup类又派生出ParrellelTime和Storyboard，如图15-4所示。

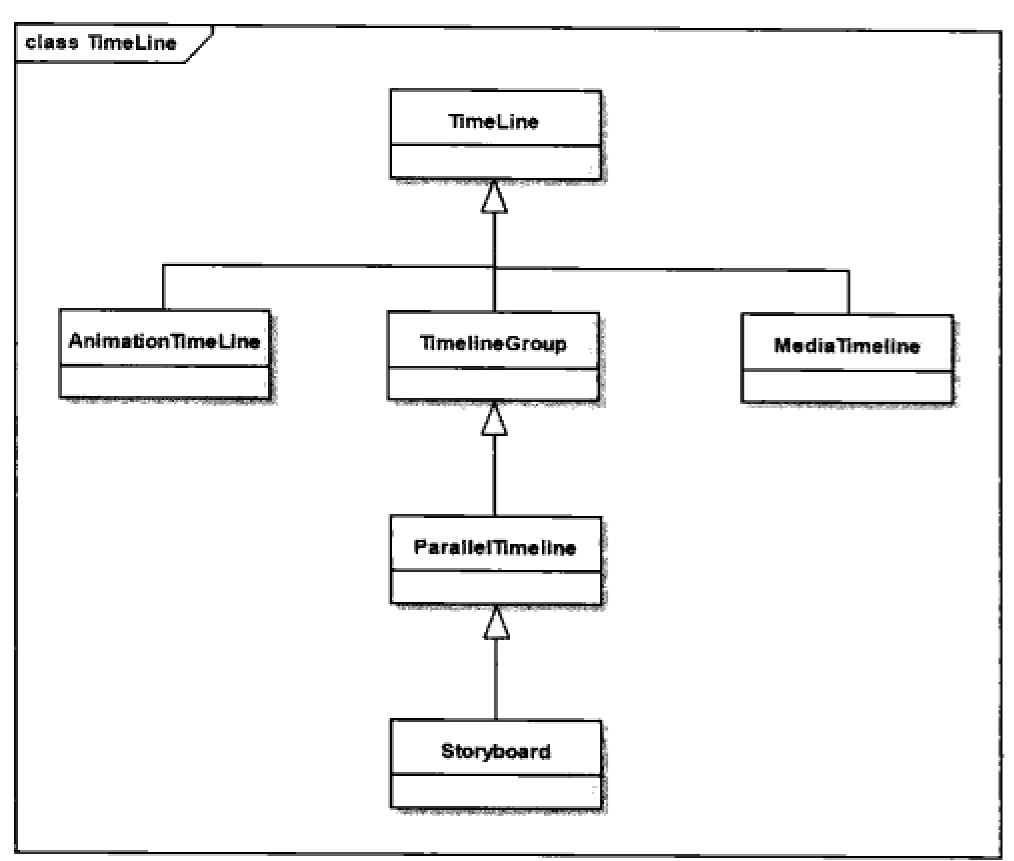


图15-4 控制动画的时间片类

从AnimationTimeLine中派生出22个动画基类，这22个动画基类对应于表15-1数据类型和动画类列出的22个数据类型，它们是类似的，若理解了其中的一个，就不难理解其余的21个。

TimelineGroup是一个抽象类，它是一个包容器类，其唯一增加的属性是Children，在其中可以加入任意多个TimeLine。

MediaTimeline支持录像和声音的动画，其中的Source属性在MediaTimeline中引入录像和声音文件。由于MediaTimeline通常要在Storyboard中使用，笔者将在15.5节Storyboard中再举例说明。

15.5 故事板（Storyboard）

故事版是Timeline，它在WPF动画里起着重要的作用，由于它是ParallelTimeline的派生类，它支持多个动画并发进行。所有动画都需要在某个事件触发下开始，WPF专门设计了BeginStoryBoard类。BeginStoryBoard是一个触发器类，它为Storyboard类提供了宿主，每个BeginStoryBoard中能够且仅能够含有一个Storyboard。

15.5.1 使用故事板的一般格式

让我们先来看一个使用故事板的简单动画：

```
<Window
x:Class="Yingbaosoft.Chapter15.UseAnimation.StoryBoardRectangle"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="故事版---矩形动画" Height="303" Width="444">
    <Canvas>
        <Rectangle Canvas.Left="40" Canvas.Top="59" Height="100"
```

```
            Name="myRectangle" Stroke="Black" Width="200"
            Fill="Violet" RadiusX="10" RadiusY="10" >
             <Rectangle.Triggers>
                 <EventTrigger
                     RoutedEvent="Rectangle.MouseRightButtonUp">
                     <BeginStoryboard>
                         <Storyboard >
                           <DoubleAnimation
                           Storyboard.TargetProperty="(Canvas.Left)"
                           From="0" To="300" Duration="0:0:0.5"
                           AutoReverse="True" RepeatBehavior="5x"/>
                           <DoubleAnimation
                            Storyboard.TargetProperty="(Canvas.Top)"
                            From="0" To="400" Duration="0:0:1"
                            AutoReverse="True" RepeatBehavior="2x"/>
                         </Storyboard>
                     </BeginStoryboard>
                 </EventTrigger>
             </Rectangle.Triggers>
         </Rectangle>
     </Canvas>
</Window>
```

在这个小动画里，全部使用XAML。WPF中的动画基本上都可以完全用XAML来表示。

FrameworkElement中有一个Triggers属性，其类型为TriggerCollection。Triggers中可以放入任意多个EventTrigger。EventTrigger可以捕获传递事件，其中的Actions属性可以放入任意一个触发器。图15-5示出了EventTrigger和ActionTrigger之间的关系。

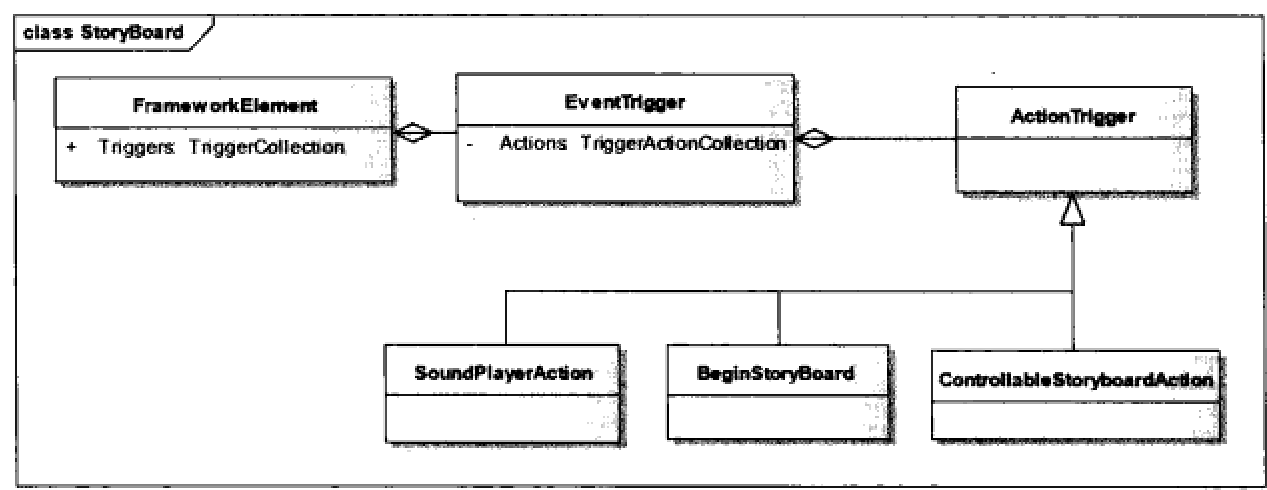


图15-5 WPF中的事件触发器及其相互关系

EventTrigger用来捕获传递事件，ActionTrigger则是在捕获传递事件后所要做的工作。从图15-5可以看到，BeginStoryboard是一种ActionTrigger，即BeginStoryboard应移植动画。BeginStoryboard中含有一个Storyboard属性，真正的动画从Storyboard开始。所以，在XAML中使用Storyboard具有下面的通用结构：

```
<EventTrigger  RoutedEvent="" >
    <BeginStoryboard>
       <Storyboard>
```

```
            //  具体动画类
          </Storyboard>
     </ BeginStoryboard>
</EventTrigger>
```

Storyboard类中含有三个重要的附加属性：Target，类型为相关对象（DependencyObject)；TargetName，其类型为string，必须是FrameworkElement、FrameworkContentElement或Freezable对象的名字；TargetProperty，其类型为PropertyPath，必须指向相关属性。

15.5.2 设定Target和TargetName

Storyboard中的Target属性是指动画所要施加的对象。例如在前面的例子中：

```
image1.BeginAnimation(Canvas.LeftProperty, da);
```

image1就是动画的Target。在Storyboard的例子中，我们并没有设定Target或TargetName。在不设定Target或TargetName的情况下，为BeginStory提供宿主的DependencyObject就是Storyboard的Target。例如在上面的例子中，Rectangle就是其中的Storyboard的Target。如果把storyboard放到资源中，则设定target或TargetName是必须的。

15.5.3 操作Storyboard

在EventTrigger里，我们使用BeginStoryborad来开始“讲述”一个故事。实际上除了开始“讲述”故事之外，还需要对故事板进行控制。这就是图15-6中ControllableStoryboard及其派生类的工作。从ControllableStoryboard中派生出9个类：BeginStoryboard、PauseStoryboard、RemoveStoryboard、ResumeStoryboard、SeekStoryboard、SkipStoryboardToFill、SoundPlayerAction、StopStoryboard和SetStoryboardSpeedRatio。

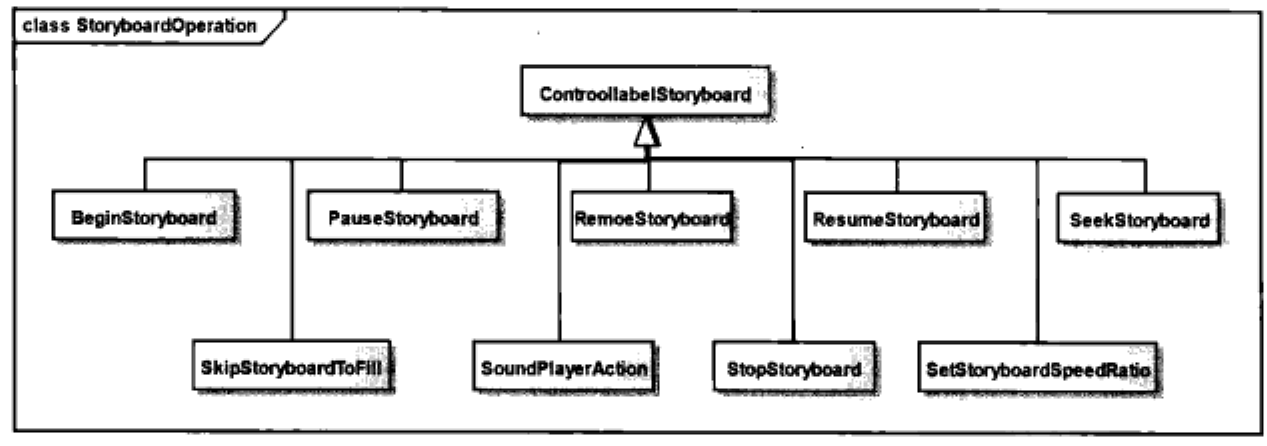


图15-6 对Storyboard的操作

BeginStoryboard类开始一个动画，上述9个类则是在Storyboard开始后对其进行操作。对Storyboard的操作听起来复杂，其实比较简单。首先需要给Storyboard设置名字（Name）属性，然后可以在后面引用该Storyboard请看下面的例子：

```
<Window x:Class="Yingbao.Chapter15.UseAnimation.ControlStoryboard"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="操作故事板" Height="213" Width="390">
```

```
    <Canvas Width="273" Height="163">
        <Ellipse Canvas.Left="43" Canvas.Top="51" Height="90"
               Name="ellipse1" Stroke="Green" StrokeThickness="5"
               Width="140" >
            <Ellipse.Fill>
                <SolidColorBrush x:Name="ybAnimatedBrush"
                    Color="White" />
            </Ellipse.Fill>
            <Ellipse.Triggers>
                <EventTrigger RoutedEvent="Ellipse.MouseEnter">
                    <BeginStoryboard Name="AnimateColor">
                        <Storyboard >
                            <ColorAnimation
                    Storyboard.TargetName="ybAnimatedBrush"
                    Storyboard.TargetProperty="Color" To="Red"
                    Duration=" 0:0:5"/>
                        </Storyboard>
                    </BeginStoryboard>
                </EventTrigger>
                <EventTrigger RoutedEvent=
                            "Ellipse.MouseLeftButtonDown">
                    <PauseStoryboard BeginStoryboardName=
                             "AnimateColor"/>
                </EventTrigger>

                <EventTrigger RoutedEvent=
                             "Ellipse.MouseLeftButtonUp">
                    <ResumeStoryboard BeginStoryboardName
                             ="AnimateColor"/>
                </EventTrigger>

                <EventTrigger RoutedEvent="Ellipse.MouseLeave">
                    <StopStoryboard BeginStoryboardName
                             ="AnimateColor"/>
                </EventTrigger>
            </Ellipse.Triggers>
        </Ellipse>
    </Canvas>
</Window>
```

在这个例子中，笔者对椭圆内的填充色进行动画。动画对象是ybAnimatedBrush中的颜色，这是SolidColorBrush类中的一个属性，可用该画刷填充椭圆。由于对颜色进行动画，笔者在Storyboard中选用ColorAnimation类进行。从白色变化到红色，历经5秒钟的时间，下面是ColorAnimation的语法：

```
  <ColorAnimation   Storyboard.TargetName="ybAnimatedBrush"
            Storyboard.TargetProperty="Color" To="Red"
              Duration=" 0:0:5"/>
```

给BeginStory对象起名为“AnimateColor”，它在鼠标进入椭圆区时开始，有MouseEnter事件触发。当鼠标左键按下时（MouseLeftButtonDown），要暂停该动画：

```
<EventTrigger RoutedEvent="Ellipse.MouseLeftButtonDown">
      <PauseStoryboard BeginStoryboardName="AnimateColor"/>
```

```
</EventTrigger>
```

使用PauseStoryboard并指明暂停动画的名称即可。

当鼠标左键释放时（MouseLeftButtonUp），要让该动画接着进行：

```
<EventTrigger RoutedEvent="Ellipse.MouseLeftButtonUp">
     <ResumeStoryboard BeginStoryboardName="AnimateColor"/>
</EventTrigger>
```

在这里笔者使用ResumeStoryboard类，并设置BeginStoryboardName属性。

最后，在鼠标离开椭圆时（MouseLeave），要停止该动画：

```
<EventTrigger RoutedEvent="Ellipse.MouseLeave">
       <StopStoryboard BeginStoryboardName="AnimateColor"/>
</EventTrigger>
```

笔者在这里使用的是StopStoryboard类，同样设置BeginStoryboardName属性。

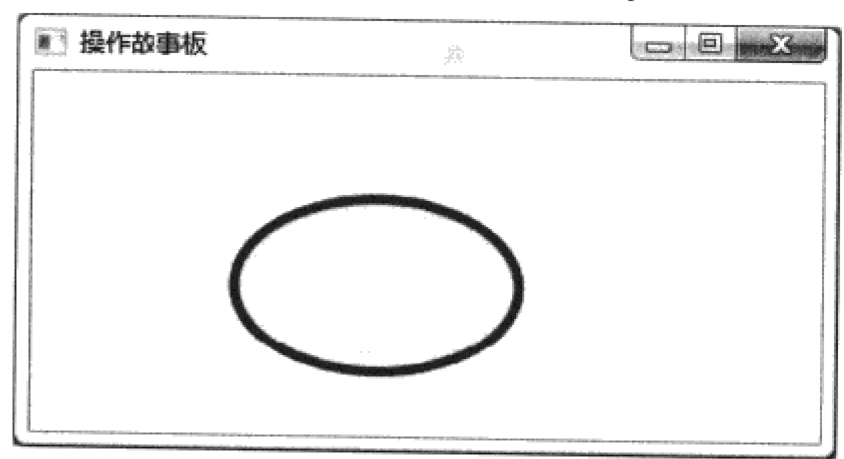


图15-7 操作故事板示例

SoundPlayerAction允许在动画中插入声音。例如，可以在上面的动画中加上《渔舟唱晚》古筝曲：

```
<EventTrigger RoutedEvent="Ellipse.MouseEnter">
      <SoundPlayerAction
Source="C:\Yingbaoproject\WPF\Yingbao\Chapter15\Chapter15\Music\yjc
wlc.wav"/>
</EventTrigger>
```

这会使你的动画更加生动。需要注意的是SoundPlayerAction只接受wav格式的声音文件，有关SoundPlayerAction将在第16章详细讨论。

15.6 KeyFrame

WPF动画最重要的一类是KeyFrame类，表15-1列出的22种数据类型，若某个数据类型的值是可以连续变化的，则WPF提供两种类型的动画。如Int16，有Int16Animation类和Int16AnimationUsingKeyFrame，本章前面所讲述的动画一概没有涉及KeyFrame。若某个数据类型不能连续变化，WPF只提供一种动画，即使用KeyFrame。Bool、Char、String和Matrix是这种类型。所谓KeyFrame动画，程序员需要为某个相关属性在特定的时间给出特定的值。 对于连续变化的相关属性，WPF根据设定的动画属性，

计算出在某个时刻的相应值（在没有设置加速或减速的情况下，通常是线性插值）。使用KeyFrame可以更精确、更灵活地控制动画。

使用KeyFrame可以完成前面所述的动画的所有功能，Microsoft所开发的WPF工具Express Blend只支持KeyFrame动画，由此可知KeyFrame在动画中的突出地位。

xxxAnimationKeyFrames是一个包容器类，其中可以含有任意多个xxxKeyFrame类，这里xxx为22种数据类型中的一个。

×××KeyFrame为抽象类，对于可以连续变化的数据类型，从xxxKeyFrame中派生出三个类：DiscreteXXXKeyFrame、LinearXXXKeyFrame和SplineXXXKeyFrame，其中xxx为数据类型。例如颜色的动画，共有ColorAnimationUsingKeyFrames、ColorKeyFrame、DiscreteColorKeyFrame、LinearColorKeyFrame和SplineColorKeyFrame。图15-8示出了这5个类间关系的UML表示。

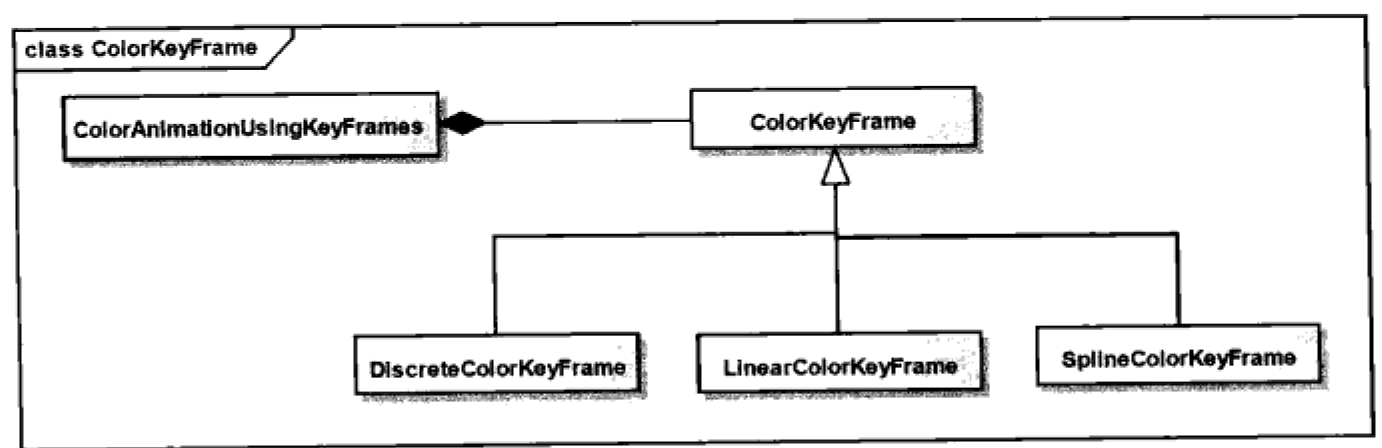


图15-8 ColorAnimationUsingKeyFrames、ColorKeyFrame、DiscreteColorKeyFrame、LinearColorKeyFrame 和SplineColorKeyFrame间的关系

对于不可以连续变化的数据类型，xxxKeyFrame只有一个派生类，即DiscreteXXXKeyFrame。使用KeyFrame时，启动故事板（BeginStoryboard）的机制和15.5.3节的XXXAnimation相同，即UsingKeyFrames是在Storyboard这个级别。

15.6.1 线性KeyFrame

下面是一个拍球的动画例子。在这个例子中，笔者对球（椭圆）的两个属性进行了动画，一个是其填充色，另一个是其在平面上的位置。完整的XAML如下：

```
<Window x:Class="Yingbao.Chapter15.UseAnimation.LinearKeyFrame"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
x:Name="Window" Title="使用KeyFrame1" Width="287" Height="480">
    <Window.Resources>
        <Storyboard x:Key="bouncing" RepeatBehavior="Forever" >
            <ColorAnimationUsingKeyFrames BeginTime="00:00:00"
                Storyboard.TargetName="ellipse"
                Storyboard.TargetProperty
                ="(Shape.Fill).(SolidColorBrush.Color)">
                <LinearColorKeyFrame KeyTime="00:00:00"
                    Value="#FF8C1010"/>
                <LinearColorKeyFrame KeyTime="00:00:03"
```

```
                        Value="#FF8B8C10"/>
                    <LinearColorKeyFrame KeyTime="00:00:06"
                        Value="#FF8C3D10"/>
                </ColorAnimationUsingKeyFrames>
                <DoubleAnimationUsingKeyFrames BeginTime="00:00:00"
                    Storyboard.TargetName="ellipse"
                    Storyboard.TargetProperty="(UIElement.RenderTransform).
                    (TransformGroup.Children)[3].(TranslateTransform.Y)">
                    <LinearDoubleKeyFrame KeyTime="00:00:00" Value="0"/>
                    <LinearDoubleKeyFrame KeyTime="00:00:03" Value="293"/>
                    <LinearDoubleKeyFrame KeyTime="00:00:06" Value="0"/>
                </DoubleAnimationUsingKeyFrames>
            </Storyboard>
    </Window.Resources>
    <Window.Triggers>
        <EventTrigger RoutedEvent="FrameworkElement.Loaded">
            <BeginStoryboard Storyboard="{StaticResource bouncing}"/>
        </EventTrigger>
    </Window.Triggers>
    <Grid x:Name="LayoutRoot" Width="264">
        <Ellipse Fill="#FFFFFFFF" Stroke="#FF000000"
              HorizontalAlignment="Left"  Margin="92,70,0,0"
              VerticalAlignment="Top" Width="64" Height="56"
              x:Name="ellipse" RenderTransformOrigin="0.5,0.5">
            <Ellipse.RenderTransform>
                <TransformGroup>
                    <ScaleTransform ScaleX="1" ScaleY="1"/>
                    <SkewTransform AngleX="0" AngleY="0"/>
                    <RotateTransform Angle="0"/>
                    <TranslateTransform X="0" Y="0"/>
                </TransformGroup>
            </Ellipse.RenderTransform>
        </Ellipse>
    </Grid>
</Window>
```

首先，笔者把Storyboard放到资源中，并把x:Key设为“bouncing”以便后面使用，并将其RepeatBehavior属性设为“Forever”，即无穷多遍地重复其中的动画。启动该动画的条件为窗口Loaded事件，该事件在创建视窗后立即产生：

```
<Window.Triggers>
  <EventTrigger RoutedEvent="FrameworkElement.Loaded">
      <BeginStoryboard Storyboard="{StaticResource bouncing}"/>
  </EventTrigger>
</Window.Triggers>
```

先来看看对球的颜色的动画：

```
<ColorAnimationUsingKeyFrames BeginTime="00:00:00"
           Storyboard.TargetName="ellipse"
           Storyboard.TargetProperty=
           "(Shape.Fill).(SolidColorBrush.Color)">
   <LinearColorKeyFrame KeyTime="00:00:00" Value="#FF8C1010"/>
```

```
   <LinearColorKeyFrame KeyTime="00:00:03" Value="#FF8B8C10"/>
   <LinearColorKeyFrame KeyTime="00:00:06" Value="#FF8C3D10"/>
</ColorAnimationUsingKeyFrames>
```

笔者把Storyboard.TargetName设为“ellipse”（即球）；把Storyboard.TargetProperty设为(Shape.Fill).(SolidColorBrush.Color)，即球的颜色。注意，在XAML中设置Color语法的问题。由于Fill属性是Ellipse从Shape中继承的，故可以使用基类的名字。Color是SolidColorBrush的属性，所以，要用SolidColorBrush.Color，在XMAL中使用类名及其属性，使用括弧是必须的。因此要用（SolidColorBrush. Color），而不能直接用SolidColorBrush.Color。

在ColorAnimationUsingKeyFrames中，使用了三个线性颜色KeyFrame。分别在时刻0秒、3秒和6秒，在0~3这个时间内，球的填充色从"#FF8C1010"变到"#FF8B8C10"；在3~6秒这个时间内，球的填充色又从"#FF8B8C10"变回到"#FF8C1010"。使用LinearColorKeyFrame告诉WPF，使用线性插值的方法在起始值和终止值之间插入相应的值。

再来看一下球在视窗中的位置动画，在这里笔者选用的动画对象是球的TranslanteTransForm.Y，即球显示时在Y方向的坐标变换值：

```
<DoubleAnimationUsingKeyFrames BeginTime="00:00:00"
Storyboard.TargetName="ellipse"
Storyboard.TargetProperty="(UIElement.RenderTransform).
(TransformGroup.Children)[3].(TranslateTransform.Y)">
```

这里下标[3]（即（TransformGroup.Children)[3]），是指RenderTranform：

```
<Ellipse Fill="#FFFFFFFF" Stroke="#FF000000"
    HorizontalAlignment="Left" Margin="92,70,0,0"
    VerticalAlignment="Top" Width="64" Height="56" x:Name="ellipse"
    RenderTransformOrigin="0.5,0.5">
    <Ellipse.RenderTransform>
       <TransformGroup>
            <ScaleTransform ScaleX="1" ScaleY="1"/>
            <SkewTransform AngleX="0" AngleY="0"/>
            <RotateTransform Angle="0"/>
            <TranslateTransform X="0" Y="0"/>
        </TransformGroup>
      </Ellipse.RenderTransform>
</Ellipse>
```

TranslateTransform是TransformGroup中的第4个子元素，即下标为3。在DoubleAnimationUsingKeyFrame中，笔者加入了3个LinearDoubleKeyFrame：

```
<LinearDoubleKeyFrame KeyTime="00:00:00" Value="0"/>
<LinearDoubleKeyFrame KeyTime="00:00:03" Value="293"/>
<LinearDoubleKeyFrame KeyTime="00:00:06" Value="0"/>
```

一个是在0秒，对应球的变换位置为0；一个是在3秒，对应球的变换位置为293；最后一个是在6秒，对应球的变换位置为0。即球在0~3秒内落下，在3~6秒内，球有反弹到最初的位置。

图15-9示出了动画结果。从这个例子可以看出，使用线性KeyFrame，要做的是设定两个属性，

一个是时间，另一个是所动画的相关属性在该时刻的值。在上面的例子中，使用的是绝对时间，实际上也可以用相对时间来代替。比如把上面的动画DoubleAnimationUsingKeyFrames改为：

```
<DoubleAnimationUsingKeyFrames BeginTime="00:00:00"
Storyboard.TargetName="ellipse"
Storyboard.TargetProperty="(UIElement.RenderTransform).(TransformGroup.
Children)[3].(TranslateTransform.Y)" Duration=" 0:0:6">
            <LinearDoubleKeyFrame KeyTime="0%" Value="0"/>
            <LinearDoubleKeyFrame KeyTime="50%" Value="293"/>
            <LinearDoubleKeyFrame KeyTime="100%" Value="0"/>
</DoubleAnimationUsingKeyFrames>
```

所得到的效果是一样的。

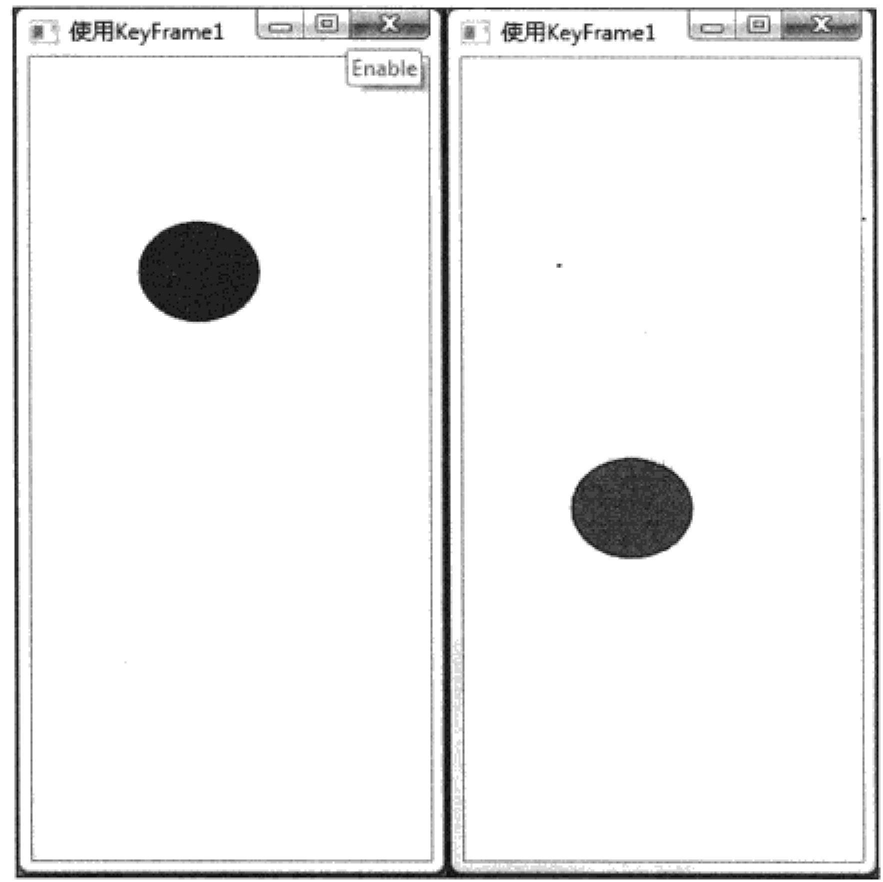


图15-9 拍皮球（1）球的颜色和位置发生变化

15.6.2 非线性KeyFrame（Spline KeyFrame）

有时候需要使用非线性KeyFrame来使动画达到更逼真的效果。例如，对上面的拍球例子来说，球向下运动时，由于重力的作用，球的速度应该是加快的；当球从地上弹起时，球的速度是减慢的。前面的例子中，用线性KeyFrame时，球的速度总是匀速变化的。

为此，需要对上面的例子进行改进，改进的方法是使用SplineColorKeyFrame来代替LinearColorKeyFrame；使用SplineDoubleKeyFrame来代替LinearDoubleKeyFrame：

```
<SplineColorKeyFrame KeyTime="00:00:00" Value="#FF8C1010"/>
<SplineColorKeyFrame KeyTime="00:00:03" Value="#FF8B8C10"
                     KeySpline="0.9,0.06,1,1"/>
<SplineColorKeyFrame KeyTime="00:00:06" Value="#FF8C3D10"
                     KeySpline="0.06,0.9,1,1"/>
```

```
<SplineDoubleKeyFrame KeyTime="00:00:00" Value="0"/>
<SplineDoubleKeyFrame KeyTime="00:00:03" Value="293"
                      KeySpline="0.9,0.06,1,1"/>
<SplineDoubleKeyFrame KeyTime="00:00:06" Value="0"
                      KeySpline="0.06,0.9,1,1"/>
```

使用SplineXXXKeyFrame的关键是设定KeySpline这个属性。KeySpline是一个特殊的柏之线，第13章曾对柏之线进行过深入的讨论。通常的柏之线由4点构成，即起点、终点和两个控制点。KeySpline使用一个控制点，起点的坐标为（0，0），终点的坐标为（1，1），KeySpline参数的示意图如图15-10所示。

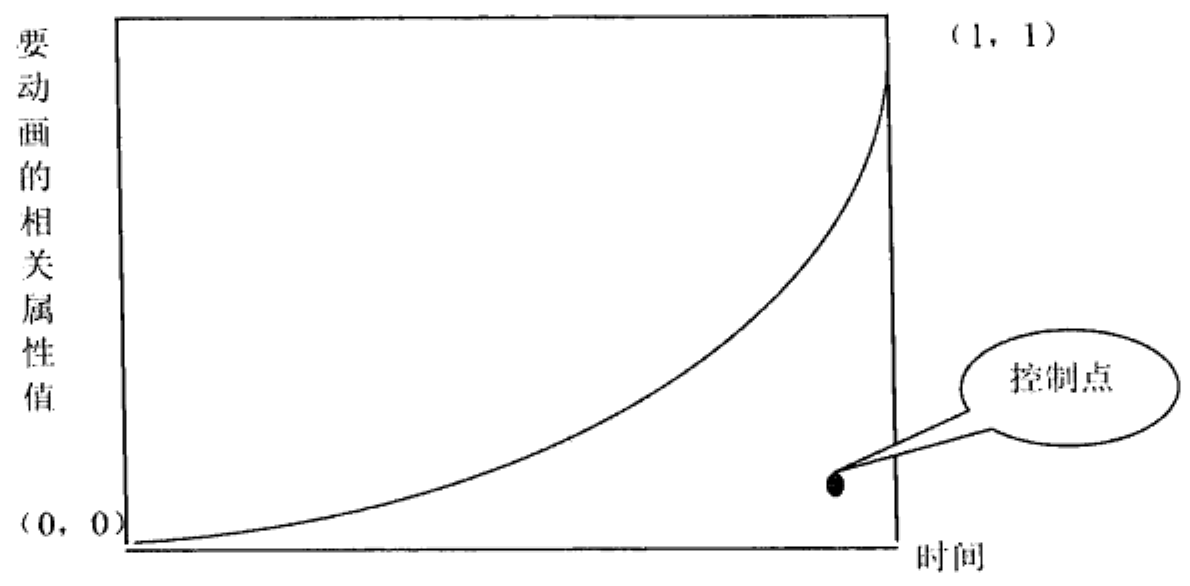


图15-10 KeySpline参数示意图

实现上面所述动画效果的XAML如下：

```
<Window
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
      xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
      x:Class="Yingbao.Chapter15.UseAnimation.ImpovedBoucingball"
      x:Name="Window"
      Title="改进的拍球例子"
      Width="204" Height="480">
    <Window.Resources>
        <Storyboard x:Key="bouncing" RepeatBehavior="Forever">
            <ColorAnimationUsingKeyFrames BeginTime="00:00:00"
             Storyboard.TargetName="ellipse" Storyboard.
             TargetProperty="(Shape.Fill).(SolidColorBrush.Color)">
                <SplineColorKeyFrame KeyTime="00:00:00"
                    Value="#FF8C1010"/>
                <SplineColorKeyFrame KeyTime="00:00:03"
                    Value="#FF8B8C10" KeySpline="0.9,0.06,1,1"/>
                <SplineColorKeyFrame KeyTime="00:00:06"
                    Value="#FF8C3D10" KeySpline="0.06,0.9,1,1"/>
            </ColorAnimationUsingKeyFrames>
            <DoubleAnimationUsingKeyFrames BeginTime="00:00:00"
               Storyboard.TargetName="ellipse"
              Storyboard.TargetProperty="(UIElement.RenderTransform)
```

```
            .(TransformGroup.Children)[3].(TranslateTransform.Y)">
                <SplineDoubleKeyFrame KeyTime="00:00:00"
                     Value="0"/>
                <SplineDoubleKeyFrame KeyTime="00:00:03"
                     Value="293" KeySpline="0.9,0.06,1,1"/>
                <SplineDoubleKeyFrame KeyTime="00:00:06" Value="0"
                     KeySpline="0.06,0.9,1,1"/>
            </DoubleAnimationUsingKeyFrames>
        </Storyboard>
    </Window.Resources>
    <Window.Triggers>
        <EventTrigger RoutedEvent="FrameworkElement.Loaded">
            <BeginStoryboard Storyboard="{StaticResource
                bouncing}"/>
        </EventTrigger>
    </Window.Triggers>
    <Grid x:Name="LayoutRoot" Width="170">
        <Ellipse Fill="#FFFFFFFF" Stroke="#FF000000" Margin
             ="30,56,76,0" VerticalAlignment="Top" Height="56"
             x:Name="ellipse" RenderTransformOrigin="0.5,0.5">
            <Ellipse.RenderTransform>
                <TransformGroup>
                    <ScaleTransform ScaleX="1" ScaleY="1"/>
                    <SkewTransform AngleX="0" AngleY="0"/>
                    <RotateTransform Angle="0"/>
                    <TranslateTransform X="0" Y="0"/>
                </TransformGroup>
            </Ellipse.RenderTransform>
        </Ellipse>
    </Grid>
</Window>
```

15.6.3 离散KeyFrame（Discrete KeyFrame）

与LinearXXXKeyFrame及SplineXXXKeyFrame不同，DiscreteXXXKeyFrame不存在插值问题。DiscreteXXXKeyFrame设定某个时刻所动画的相关属性的值，在下一个时刻，设定另外一个值。所有连续变化的相关属性的动画，都有相应的DiscreteXXXKeyFrame类。但表15-1中的bool、Char、string、Matrix和Object这五种类型只有离散KeyFrame动画类。

DiscreteXXXKeyFrame的用法为：

```
<DiscreteColorKeyFrame KeyTime="00:00:00" Value="Black"/>
```

设定其中的两个参数，一个是KeyTime（给定时刻），另一个是相关属性的值。

让我们来看一看如何用WPF动画来实现十字路口的交通灯。交通灯由三个信号灯组成，这三个信号灯在给定的时间只有一个灯亮，即在任意给定时间，红黄绿三个信号灯中只有一个亮。

交通灯的时序如图15-11所示。我们把交通灯的故事板的Duration属性设为9秒。

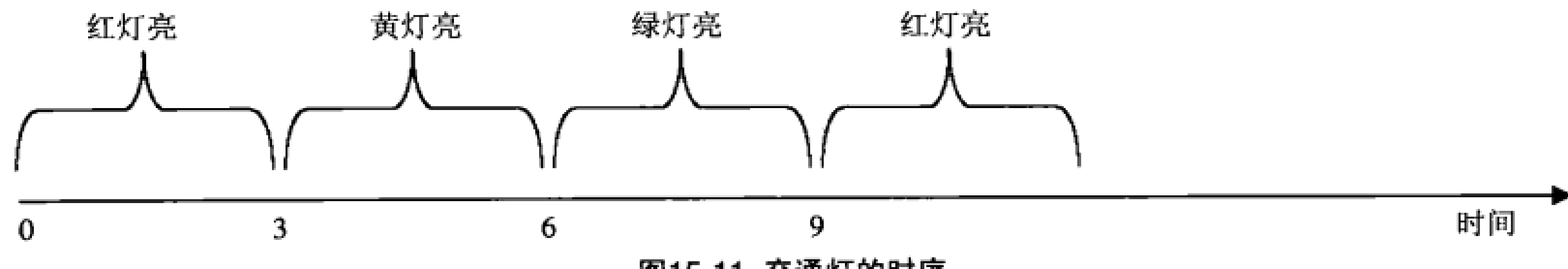


图15-11 交通灯的时序

交通灯的XAML实现如下：

```
<Window x:Class="Yingbao.Chapter15.UseAnimation.TraficSignal"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="TraficSignal" Height="300" Width="142">
    <Window.Resources>
      <Storyboard x:Key="SignalStory" RepeatBehavior="Forever"
          Duration=" 0:0:09">
          <ColorAnimationUsingKeyFrames BeginTime="00:00:00"
               Storyboard.TargetName="redSignal"
               Storyboard.TargetProperty=
               "(Shape.Fill).(SolidColorBrush.Color)">
              <DiscreteColorKeyFrame KeyTime="00:00:00"
                  Value="Red" />
               <DiscreteColorKeyFrame KeyTime="00:00:03"
                  Value="Black"  />
           </ColorAnimationUsingKeyFrames>
           <ColorAnimationUsingKeyFrames BeginTime="00:00:00"
                Storyboard.TargetName="yellowSignal"
                Storyboard.TargetProperty=
                "(Shape.Fill).(SolidColorBrush.Color)">
               <DiscreteColorKeyFrame KeyTime="00:00:03"
                Value="Yellow"  />
              <DiscreteColorKeyFrame KeyTime="00:00:06"
                Value="Black"/>
           </ColorAnimationUsingKeyFrames>
           <ColorAnimationUsingKeyFrames BeginTime="00:00:00"
                   Storyboard.TargetName="greenSignal"
                   Storyboard.TargetProperty=
                   "(Shape.Fill).(SolidColorBrush.Color)">
              <DiscreteColorKeyFrame KeyTime="00:00:00"
                  Value="Black"/>
              <DiscreteColorKeyFrame KeyTime="00:00:06"
                 Value="Green" />
           </ColorAnimationUsingKeyFrames>
       </Storyboard>
    </Window.Resources>
    <Window.Triggers>
        <EventTrigger RoutedEvent="FrameworkElement.Loaded">
            <BeginStoryboard Storyboard="{StaticResource
                 SignalStory}"/>
        </EventTrigger>
    </Window.Triggers>
    <Canvas>
```

```
        <Ellipse Fill="Red" Name="redSignal" Width="64" Height="50"
             Canvas.Left="20" Canvas.Top="20" Stroke="Black"/>
        <Ellipse Canvas.Left="20" Canvas.Top="76" Fill="Black"
             Height="50" Name="yellowSignal" Stroke="Black"
             Width="64" />
        <Ellipse Canvas.Left="20" Canvas.Top="132" Fill="Black"
             Height="50" Name="greenSignal" Stroke="Black"
             Width="64" />
    </Canvas>
</Window>
```

这段程序的运行结果如图15-12所示。

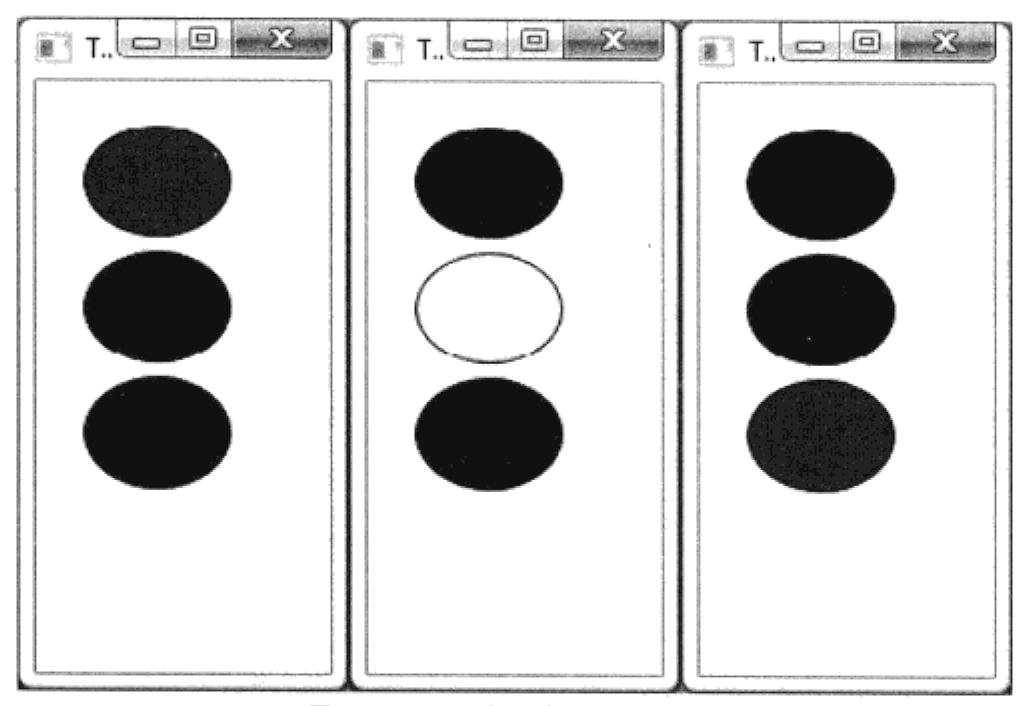


图15-12 用动画实现的交通灯

15.7 本章小结

本章比较详细地讨论了WPF中的动画。WPF的动画类主要集中在System.Windows.Media.Animation命名空间中，其他命名空间的大量的类对动画提供支持。WPF的动画是对相关属性进行的，WPF动画的目标对象一定要是相关对象。在CLR支持的22个数据类型中，其中有5个是离散类型。动画要有开始动画的条件（Trigger），动画的主要任务是管理时间片。

第四篇　开发WPF产品

第 16 章　多媒体技术及其应用

第 17 章　定制控件和排版

第 18 章　综合应用

第16章 多媒体技术及其应用

多媒体技术是一个非常激动人心的领域，其研究的领域包括语音识别、音乐及声音播放、录像播放、图形动画等。WPF对图形及文字的动画做了很多工作，但严格说来，WPF对语音识别、音乐播放、文字朗读和录像播放等几乎没做什么特殊的工作。 大部分工作实际上是使用已有的库函数，WPF只是在其上面加了一层，引入相关属性及传递事件，从而我们可以在XAML中使用这些多媒体技术。

16.1 播放.wav声音格式的SoudPlayer和SoundPlayerAction

类SoundPlayer是.NET 2.0中就有的，它位于System.Media命名空间中；类SoundPlayerAction是WPF引入的，它位于System.Windows.Controls命名空间中。SoundPlayer的功能是播放.wav格式的声音文件，要使用SoundPlayer类，必须在后台代码C#、VB等语言中创建SoundPlayer类实例，并调用其相应的方法。 SoundPlayerAction是为XAML开发的类，其功能和SoundPlayer完全相同，SoundPlayerAction内部使用SoundPlayer类。所以，要了解SoundPlayerAction的功能，必先从SoundPlayer开始。

16.1.1 装载.wav文件

SoundPlayer提供两种装载.wav文件的方法：一种是同步方式，另一种是异步方式。方法Load是在同步模式下的装载操作，Load和调用Load的方法是在同一个线程中。若.wav文件较小或计算机的速度快，使用这个方法非常简便。相反，若.wav文件较大、不在本机或计算机速度较慢时，调用Load时可能会阻断UI线程，出现停顿。装载.wav文件的方法如下。

```
try
{
    SoundPlayer player = new SoundPlayer();
    player.SoundLocation = @" C:\Yingbaoproject\mp3\zheng -
yjcw1c.wav"
    player.Load();
}
catch (Exception ex)
{
  //...show error
}
```

从上面这段代码中，可以看出，Load方法本身并不带有参数，在调用之前，可以先设定SoundLocation，或调用带有参数的构造函数，Location采用URL格式。

与同步装载相对的是异步装载，这时需要调用LoadAsync方法，调用LoadAsync时，LoadAsync会创建一个独立的线程，并在该线程中加载.wav文件，LoadAsync不会阻断UI线程。通常，若.wav不在本机，应该使用LoadAsync以改善用户性能。LoadTimeout属性用来设定加载超时，LoadCompleted事件用来通知调用线程加载完毕，调用线程可以通过查询IsLoadCompleted属性来获取装载的状态。

16.1.2 播放.wav文件

SoundPlayer支持同步播放和异步播放两种模式。调用Play()方法为异步播放，即播放有自己独立的线程；调用PlaySync()方法为同步播放，一般使用Play。

PlayLooping()方法可以重复播发同一.wav文件。

16.1.3 停止播放

Stop方法终止播放声音。SoundPlayer不支持调整音量、多声道平衡等高级功能。

16.1.4 在XAML中使用SoundPlayerAction

SoundPlayer虽然有一些局限，但在应用程序使用简单的声音提示时，使用SoundPlayer是一个不错的选择（使用较少的资源）。为了在XAML中支持.wav文件播放，WPF提供了SoundPlayerAction类。

图 16-1 为 SoundPlayer 类继承图，它是一个触发器类，可以在任何事件触发器中使用SoundPlayer：

```
<Window x:Class="Yingbao.Chapter16.AppWin"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="使用SoundPlayerAction" Height="100" Width="200">
    <Grid>
       <Button Height="23" HorizontalAlignment="Left"
          Margin="12,12,0,0" Name="button1"
          VerticalAlignment="Top" Width="37" Content="0">
            <Button.Triggers>
                <EventTrigger RoutedEvent="Button.Click">
                    <EventTrigger.Actions>
                        <SoundPlayerAction
                             Source="Sound/Play0.wav"/>
                    </EventTrigger.Actions>
                </EventTrigger>
                <EventTrigger RoutedEvent="Button.MouseEnter">
                    <EventTrigger.Actions>
                        <SoundPlayerAction
                             Source="Sound/hover.wav"/>
                    </EventTrigger.Actions>
                </EventTrigger>
            </Button.Triggers>
        </Button>
    </Grid>
</Window>
```

在这段程序中，SoundPlayerAction直接加入到EventTriggerActrion中。图16-2显示这段XAML的运行结果，当把鼠标拖过按钮“0”时，可以听到hover.wav播放出的声音；当单击按钮“0”时，我们可以听见读出“0”的声。

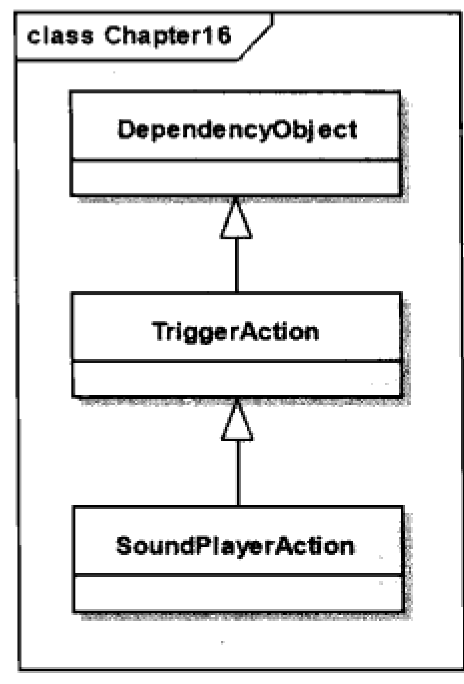


图16-1 SoundPlayer类继承图

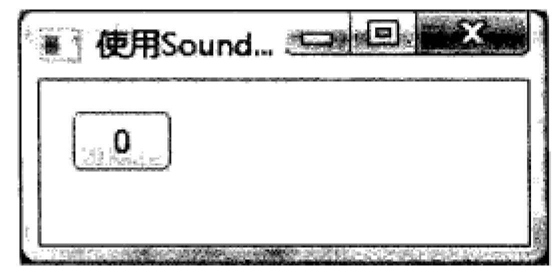


图16-2 在XAML中使用SoundPlayerAction

使用SoundPlayerAction很简单，几乎没什么变化，若需要在应用程序中加入简单的语音帮助功能，使用SoundPlayerAction是一个不错的选择。

16.2 播放多种格式的声音和图像

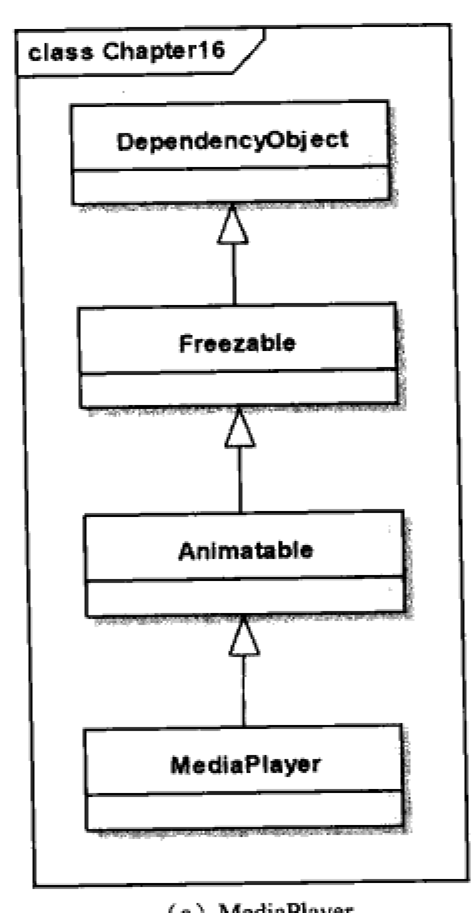
当简单的.wav格式声音及SoundPlayer的功能不能满足你的要求时，WPF中的MediaPlayer、MediaElement和MediaTimeLine三个类提供了播放各种声音和图像及控制播放的更多功能。

MediaPlayer和MediaElement都是构建在微软的Media Player 10.0之上的，所以要使用MediaPlayer和MediaElement，必须在机器上安装版本10及其以上的Media Player应用程序。

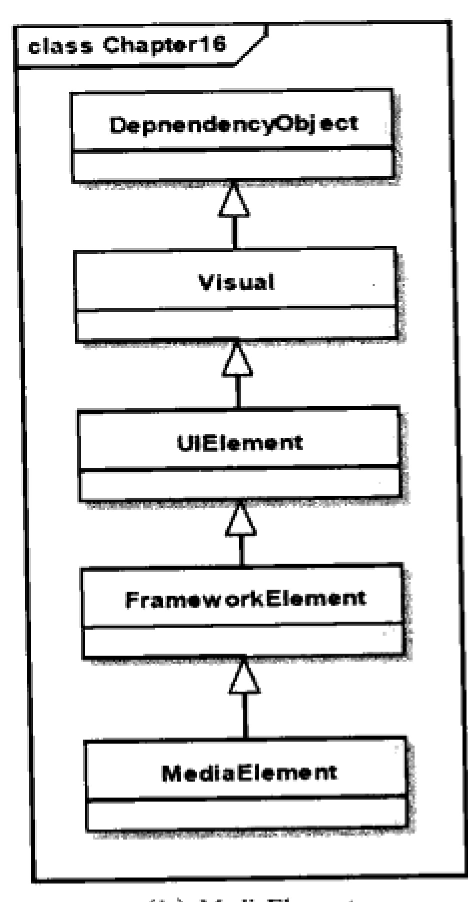
微软的Media Player应用程序包含一个ActiveX组件，这个组件是播放多媒体核心部件，它提供了丰富的功能（http://msdn.microsoft.com/en-us/library/dd7580 70(VS.85).aspx ）。WPF中的MediaPlayer和MediaElement提供的只是该ActiveX组件功能的一个子集，如果你的应用程序需要开发更丰富的录像播放功能，需要直接使用SDK。

对于大多数WPF应用程序来说，使用MediaPlayer和MediaElement类就已经足够了。这两个类支持Media Player应用程序所有声音格式（.wav、.wma、mp3等）和图像格式（.wmv、.avi、.mpg等）的文件播放。

图16-3示出了MediaPlayer和MediaElement的类结构图。从图16-3中可以看出，MediaElement是WPF的界面元素，而MediaPlayer不是。所以MediaElement可以直接用在XAML中，并在界面上显示出来。而MediaPlayer是为Drawing对象设计的，当需要在界面上显示录像时，需要通过VideoDrawing或使用DrawingContext，总之，使用MediaPlayer需要更多的C#或VB代码的干预。



(a) MediaPlayer



(b) MediaElement

图16-3 MediaPlayer和MediaElement 的类结构图

16.2.1 播放模式

MediaPlayer 和 MediaElement 支持两种播放模式：独立模式和时钟模式。MediaPlayer 和 MediaElemen这两个类中都有一个属性Clock，简单地说，若Clock属性为null，则播放模式为独立模式；反之，播放模式为时钟模式。在时钟模式下，所有的播放操作交给时钟控制。

在独立播放模式下，可以调用Open、Play、Stop、Pause等方法来对播放过程进行控制；还可以通过设置下列的属性来调整媒体播放参数。

- Balance：左右声道的比率，0表示送到左右声道的比率是一样的；-1表示所有的声音都送到左声道；1表示所有的声音都送到右声道。
- BufferProgress：表示媒体装载到内存的百分数，其值在0～1之间。我们可以利用这个属性在界面上显示装载的进度。
- DownloadProgress：表示下载媒体的进度，通常媒体文件在远距离服务器上。
- IsMuted：读写属性，若为true，则不再播放声音。
- Position：设置这个属性可以控制播放位置。
- SpeedRatio：设置这个属性可以放慢或加快媒体的播放速度，正常播放时SpeedRatio为1。
- Volume：控制音量，其值在0～1之间，默认值为0.5。当该值为1时，播出最大音量。

在时钟模式下，MediaTimeLine控制播放。当MediaElement装载媒体时，MediaTimeLine创建相应的时钟实例，并赋给MediaElement的Clock属性。在时钟模式下，控制播放的方法如Stop、Pause等都

不能使用。LoadedBehavior属性在装载媒体时起作用；UnloadedBehavior在卸载媒体时起作用。

16.2.2 使用MediaPlayer实例

一个MediaPlayer在独立模式下的应用实例如下：

```
<Window x:Class="Yingbao.Chapter16.UsingVideoPlayer.AppWin"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="使用VideoPlayer" Height="591" Width="716">
    <StackPanel Height="527">
        <Border  Height="400" Width="645" Margin=" 5,10,5,10"
          BorderBrush="AliceBlue"
          BorderThickness="10" CornerRadius=" 5,5,5,5">
          <Rectangle Name="videoRectangle" Height="350"
              Width="600"/>
        </Border>
        <StackPanel Orientation="Horizontal">
            <TextBox Name="txtFileName" Width="400" Height="25"
                    Margin=" 5,2,20,2"/>
            <Button Height="25" HorizontalAlignment="Right"
                  Name="btnSelectFile" VerticalAlignment="Bottom"
                  Width="75" Click="OnSelectFile">选择文件</Button>
        </StackPanel>
        <StackPanel Orientation="Horizontal" Margin=" 20,0,20,0">
            <Button Name="btnPlay" Height="25" Width="30"
                Click="OnPlay">
                <Image Source="Images/play.gif"/>
            </Button>
            <Button Name="btnPause" Height="25" Width="30"
                Click="OnPause">
                <Image Source="Images/pause.gif"/>
            </Button>
            <Button Name="btnStop" Height="25" Width="30"
                Click=" OnStop">
                <Image Source="Images/stop.gif"/>
            </Button>
        </StackPanel>
        <StackPanel Orientation="Horizontal" >
            <Label>音量</Label>
            <Slider Name="volumeSlider" Minimum="0" Maximum="1"
              Value="0.5" Width="100" ValueChanged="OnVolume"/>
            <Label>播放速度</Label>
            <Slider Name="speedSlider" Minimum="0" Maximum="10"
                   Value="1" Width="100"
     ValueChanged="OnSpeed"/>
            <Label>播放位置</Label>
            <Slider Name="positionSlider" Minimum="0" Maximum="100"
                   Value="0" Width="300"
     ValueChanged="OnPosition"/>
        </StackPanel>
    </StackPanel>
</Window>
```

在XAML里，我们定义了几个控件，其中videoRectangle是一个矩形，可利用MediaPlayer中的HasVideo属性在该矩形中显示一个静止的图或录像。HasVideo为True时，表示该文件为图像文件。

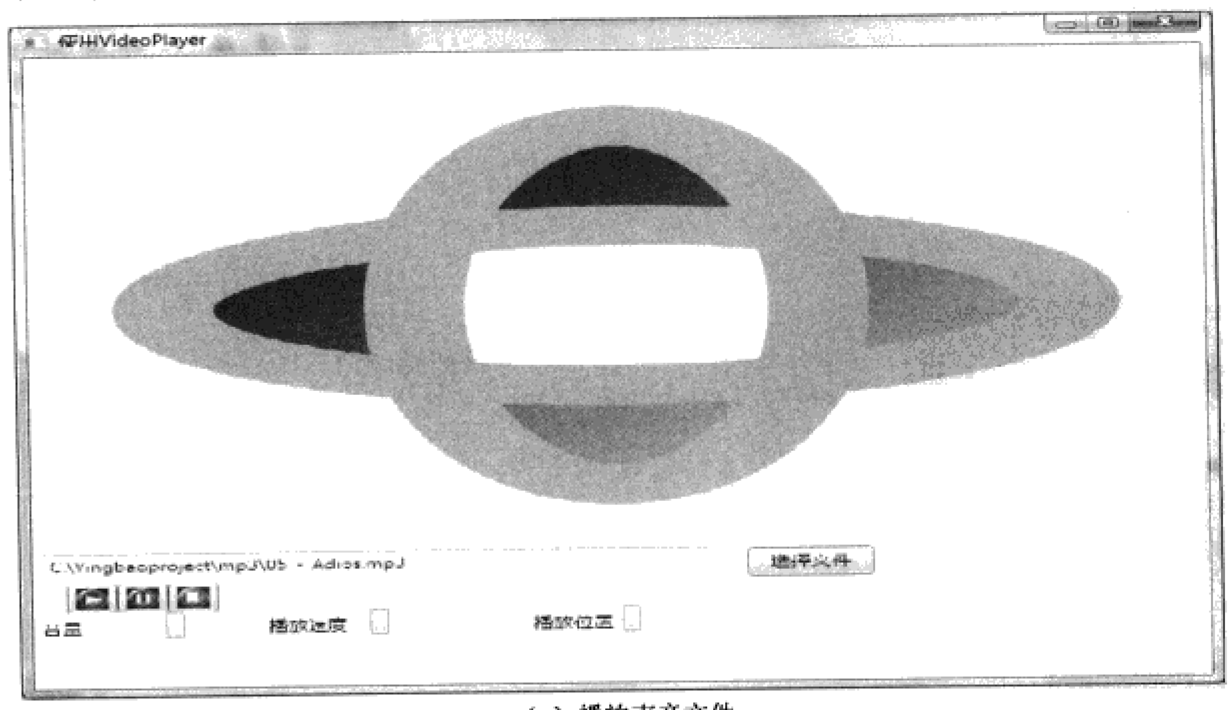
在C#后台程序里，需要用到两个类，一个是MediaPlayer，这是本节的重点，另一个是VideoDrawing。需要注意：若在调用MediaPlayer的Open方法之后，立即检查HasVideo属性，其值是不正确的。这是因为：Open方法是在另一个独立线程中进行（即异步读入）的，当读入文件完成后，MediaPlayer才产生MediaOpened事件，同时设置HasVideo属性，因此，可以在MediaPlayer类的MediaOpened传递事件中处理播放图像等相关问题：

```
namespace Yingbao.Chapter16.UsingVideoPlayer
{
    public partial class AppWin : Window
    {
        MediaPlayer player = new MediaPlayer();
        VideoDrawing vDrawing = new VideoDrawing();

        public AppWin()
        {
            InitializeComponent();
            player.MediaOpened += new
                        EventHandler(player_MediaOpened);
        }

        void OnSelectFile(object sender, RoutedEventArgs rea)
        {
            OpenFileDialog ofd = new OpenFileDialog();

            ofd.InitialDirectory = @"c:\yingbaoproject\mp3\" ;
            ofd.Filter = "wav files (*.wav)|*.wav| mp3
            files(*.mp3)|*.mp3|All files (*.*)|*.*" ;
            ofd.FilterIndex = 2 ;
            ofd.RestoreDirectory = true ;
            if (ofd.ShowDialog() == System.Windows.Forms.
                                    DialogResult.OK)
            {
                txtFileName.Text = ofd.FileName;
                if (File.Exists(ofd.FileName))
                {
                     player.Open(new Uri(ofd.FileName,
                        UriKind.Absolute ));
                }
            }
        }

        void player_MediaOpened(object sender, EventArgs e)
        {

            if (player.HasVideo)
            {
                vDrawing.Rect = new Rect(0, 0, 100, 100);
                DrawingBrush dBrush = new DrawingBrush(vDrawing);
```

```
            videoRectangle.Fill = dBrush;
            vDrawing.Player = player;
        }
        else
        {
            GeometryGroup ellipses = new GeometryGroup();
            ellipses.Children.Add(
                new EllipseGeometry(new Point(50, 50), 45, 20)
                );
            ellipses.Children.Add(
                new EllipseGeometry(new Point(50, 50), 20, 45)
                );

            GeometryDrawing aGeometryDrawing = new
                                GeometryDrawing();
            aGeometryDrawing.Geometry = ellipses;

            aGeometryDrawing.Brush =
                new LinearGradientBrush(
                    Colors.Blue,
                    Color.FromRgb(204, 204, 255),
                    new Point(0, 0),
                    new Point(1, 1));
            aGeometryDrawing.Pen = new Pen(Brushes.Cyan, 10);
            DrawingBrush dBrush = new
            DrawingBrush(aGeometryDrawing);
            videoRectangle.Fill = dBrush;

        }
        positionSlider.Maximum =
        player.NaturalDuration.TimeSpan.TotalMilliseconds;
        player.Play();
    }

    void OnPlay(object sender, RoutedEventArgs rea)
    {
        player.Play();
    }

    void OnPause(object sender, RoutedEventArgs rea)
    {
        player.Pause();
    }

    void OnStop(object sender, RoutedEventArgs rea)
    {
        player.Stop();
    }

    void OnVolume(object sender,
              RoutedPropertyChangedEventArgs<double> e)
    {
```

```
            player.Volume = e.NewValue;
        }

        void OnSpeed(object sender,
                RoutedPropertyChangedEventArgs<double> e)
        {
            player.SpeedRatio = e.NewValue;

        }

        void OnPosition(object sender,
                RoutedPropertyChangedEventArgs<double> e)
        {
            int SliderValue = (int)positionSlider.Value;

            TimeSpan ts = new TimeSpan(0, 0, 0, 0, SliderValue);
            player.Position = ts;

        }
    }
}
```

在这段程序中，笔者还移植了暂停播放、改变音量、改变播放速度、停止播放等相关功能，使用MediaPlayer移植这些功能相当简单。图16-4（a）示出了用上面的程序打开.mp3文件的显示结果，图16-4（b）是打开.avi文件时的显示结果。例子中所播放的录像是笔者在多伦多天穹体育场（Skydome）观看NBA比赛时现场拍摄的，这是一场激动人心的高水平比赛，多伦多猛龙队对休斯顿火箭队，天穹体育场是多伦多猛龙队（Toronto Rapters）的主场，他们在数万观众的欢呼声中赢得了胜利，遗憾的是，无法在书本上展示录像和声音的播放。



（a）播放声音文件

（b）播放图像文件（NBA多伦多猛龙对火箭队比赛实况——李应保摄）

图16-4 使用MediaPlayer和VideoDrawing来播放录像

16.2.3 使用MediaElement和MediaTimeline实例

MediaElement和MediaTimeline是为了支持XAML而开发的类，MediaElement直接参与WPF的版面控制，MediaTimeline是从TimeLine类中派生出来的，因此，其控制媒体播放的方式和动画中的时间流程是一样的。可以用两种方式把时间流程和MediaElement联系起来：

- 使用故事版（StoryBoard），当故事版的目标对象设为MediaElement时，MediaClock会接管媒体的播放。下面这段XAML示出了使用故事版时，MediaElement和MediaTimeline的用法：

```
<MediaElement Name="myMediaElement" Width="260" Height="150"
Stretch="Fill" />
<Button>
  <Button.Triggers>
    <EventTrigger RoutedEvent="Button.Click">
      <EventTrigger.Actions>
        <BeginStoryboard>
          <Storyboard>
            <MediaTimeline Source="media\numbers.wmv"
              Storyboard.TargetName="myMediaElement"
              BeginTime="0:0:0"/>
          </Storyboard>
        </BeginStoryboard>
      </EventTrigger.Actions>
    </EventTrigger>
  </Button.Triggers>
<Button>
```

- 调用MediaTimeline中的AllocateClock方法（该方法返回一个时钟Clock实例），并用该实例设置MediaElement中的Clock属性。虽然，可以这样设置时钟，但通常使用前面的方法。

下面把前面的例子用MediaElement和MediaTimeline来重写一遍。我们知道，在MediaTimeline控

制播放时，不能改变播放速度及设置播放的位置，因此，把OnSpeed和OnPosition两个传递事件处理程序设为空：

```
namespace Yingbao.Chapter16.UsingMediaElement
{
    public partial class AppWin : Window
    {
        public AppWin()
        {
            InitializeComponent();
            DataContext = this;
        }

        void OnSelectFile(object sender, RoutedEventArgs rea)
        {
            OpenFileDialog ofd = new OpenFileDialog();

            ofd.InitialDirectory = @"c:\yingbaoproject\mp3\";
            ofd.Filter = "wav files (*.wav)|*.wav| mp3
                 files(*.mp3)|*.mp3|All files (*.*)|*.*";
            ofd.FilterIndex = 2;
            ofd.RestoreDirectory = true;
            if (ofd.ShowDialog() ==
               System.Windows.Forms.DialogResult.OK)
            {
                txtFileName.Text = ofd.FileName;
                if (File.Exists(ofd.FileName))
                {
                    MediaSource = new Uri(ofd.FileName,
                               UriKind.Absolute);
                }
            }
        }

        public Uri MediaSource
        {
            get { return (Uri)GetValue(MediaSourceProperty); }
            set { SetValue(MediaSourceProperty, value); }
        }

        public static readonly DependencyProperty
        MediaSourceProperty =
            DependencyProperty.Register("MediaSource", typeof(Uri),
            typeof(AppWin), new FrameworkPropertyMetadata(new
            Uri("http://www.microsoft.com"),null));

        void OnVolume(object sender,
                     RoutedPropertyChangedEventArgs<double> e)
        {
            videoRectangle.Volume =  e.NewValue;
        }

        void OnSpeed(object sender,
```

```
                    RoutedPropertyChangedEventArgs<double> e)
        {
        }

        void OnPosition(object sender,
                    RoutedPropertyChangedEventArgs<double> e)
        {
        }

    }
}
```

在AppWin类中，我们定义了相关属性MediaSource，该属性为媒体文件Uri。在构造函数中，笔者把DataContext设为AppWin实例，以支持数据绑定。在XAML中，使用Storyboard、BeginStoryboard、PauseStoryboard和StopStoryboard来控制媒体的播放：

```
<Window x:Class="Yingbao.Chapter16.UsingMediaElement.AppWin"
 xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:src="clr-namespace:Yingbao.Chapter16.UsingMediaElement"
    Title="使用VideoPlayer" Height="591" Width="716">
    <Window.Resources >
        <Storyboard x:Key="videoStory" >
            <MediaTimeline Storyboard.TargetName ="videoRectangle"
             Source="{Binding Path=MediaSource}"/>
        </Storyboard>
    </Window.Resources>
    <Window.Triggers>
        <EventTrigger RoutedEvent="Button.Click"
             SourceName="btnPlay">
            <EventTrigger.Actions>
                <BeginStoryboard Name="clickPlay"
                    Storyboard="{StaticResource videoStory}"/>
            </EventTrigger.Actions>
        </EventTrigger>
        <EventTrigger RoutedEvent="Button.Click"
                    SourceName="btnPause">
            <EventTrigger.Actions>
              <PauseStoryboard   BeginStoryboardName="clickPlay"/>
            </EventTrigger.Actions>
        </EventTrigger>
        <EventTrigger RoutedEvent="Button.Click"
               SourceName="btnStop">
            <EventTrigger.Actions>
                <StopStoryboard BeginStoryboardName="clickPlay"/>
            </EventTrigger.Actions>
        </EventTrigger>
    </Window.Triggers>
    <StackPanel Height="527">
        <Border  Height="400" Width="645" Margin=" 5,10,5,10"
             BorderBrush="AliceBlue" BorderThickness="10"
             CornerRadius=" 5,5,5,5">
```

```
            <MediaElement  Name="videoRectangle"
              Height="350" Width="600"/>
        </Border>
        <StackPanel Orientation="Horizontal">
            <TextBox Name="txtFileName" Width="400"
              Height="25" Margin=" 5,2,20,2"/>
            <Button Height="25" HorizontalAlignment="Right"
              Name="btnSelectFile" VerticalAlignment="Bottom"
              Width="75" Click="OnSelectFile">选择文件</Button>
        </StackPanel>
          <StackPanel Orientation="Horizontal"
                Margin=" 20,0,20,0">
            <Button Name="btnPlay" Height="25" Width="30" >
                <Image Source="Images/play.gif"/>
            </Button>
            <Button Name="btnPause" Height="25" Width="30" >
                 <Image Source="Images/pause.gif"/>
            </Button>
            <Button Name="btnStop" Height="25" Width="30" >
                <Image Source="Images/stop.gif"/>
            </Button>
        </StackPanel>
        <StackPanel Orientation="Horizontal" >
            <Label>音量</Label>
            <Slider Name="volumeSlider" Minimum="0" Maximum="1"
                  Value="0.5" Width="100" ValueChanged="OnVolume"/>
            <Label>播放速度</Label>
            <Slider Name="speedSlider" Minimum="0" Maximum="10"
                  Value="1" Width="100" ValueChanged="OnSpeed"/>
            <Label>播放位置</Label>
            <Slider Name="positionSlider" Minimum="0" Maximum="100"
                  Value="0" Width="300" ValueChanged="OnPosition"/>
        </StackPanel>
    </StackPanel>
</Window>
```

MediaElement的Source属性是所要播放媒体文件Uri，可以直接设置该属性：

```
<MediaElement  Name="videoRectangle"
Source="c:\yingbaoproject\mp3\nba.avi" Height="350"
                              Width="600"/>
```

若运行上面的程序，可以马上看到nba.avi这段录像。MediaTimeline的Source属性被绑定到MediaSource上，当MediaTimeline接管媒体播放时，MediaTImeline的Source属性被用作MediaElement的Source，从而实现录像的播放：

```
<MediaTimeline Storyboard.TargetName ="videoRectangle"
Source="{Binding Path=MediaSource}"/>
```

- 控制播放次数

MediaTimeline是一个时间流程，RepeatBehavior属性用来设定播放次数：

```
<MediaTimeline Storyboard.TargetName ="videoRectangle"
Source="{Binding Path=MediaSource}" RepeatBehavior="2"/>
```

上面的XAML把RepeatBehavior设为2，即播放2次。

- 控制播放速度

前面已提到，在使用MediaTimeline后，不能直接设定MediaElement的播放速度，但却可以用设定MediaTimeline中的SpeedRatio来达到类似的效果：

```
<MediaTimeline Storyboard.TargetName ="videoRectangle"
      Source="{Binding Path=MediaSource}"  RepeatBehavior="2"
      SpeedRatio="{Binding ElementName=speedSlider,
      Path=Value,Mode=OneWay}"/>
```

SpeedRatio只在下次动画时起作用。

注意：由于WPF的媒体播放器是建立在Windows媒体播放器上面的，而后者是32位应用程序，所以WPF的媒体播放器也是32位的，哪怕你的机器上安装了64位.NET framework!

16.3 语音合成和语音识别

语音合成和语音识别是一个重大的基础课题，在过去四十多年里，软件界对这个课题的研究从来没有中断过，希望有一天能够不需要键盘和鼠标，而直接和计算机对话。就像我们和朋友聊天一样，比如需要了解今天的天气，我们可以问计算机：“今天北京天气怎么样？”计算机会自动打开互联网，搜索有关北京的天气，然后回答：“2009年6月30号，北京，晴转多云，最高气温摄氏30°。”再比如，你早上要去上班，但又想多睡一会，可以问计算机：“现在几点啦？”计算机马上告诉你几点几分。你不需要开灯去找你的手表了！

目前计算机的语音合成和语音识别技术虽然还达不到我描述的程度，但是已经取得了重大进展。体会一下Windows Vista中的语音功能，可以看到我们在逐步接近目标。

微软的Speech SDK 5.1是微软在语音合成和语音识别领域的最新成果，它可以免费下载（http://www.microsoft.com/downloads/details.aspx?displaylang=en&FamilyID=5e86ec97-40a7-453f-b0ee-6583171b4530#filelist）。这个SDK支持市场上几乎所有的Windows操作系统，.NET平台3.0中含有一个System.Speech.Synthesis命名空间，其中含有26个类，这些类提供了SSDK5.1的接口，其语音合成功能是由SSDK5.1提供的。就像Media player支持非.NET语言及开发环境一样，SSDK5.1也支持其他非.NET开发环境。.NET 语音识别和语音合成的层次结构如图16-5所示。

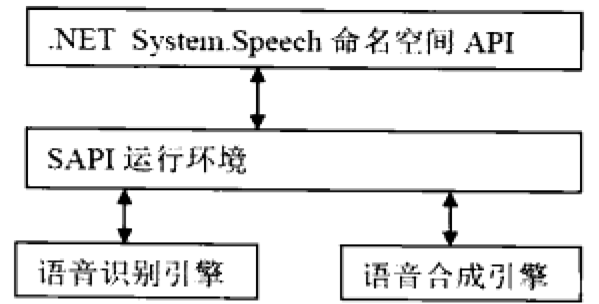


图16-5 .NET语音识别和语音合成的层次结构

语音合成是把文本转变为声音的过程，前台软件对文本进行分析，根据语法分析出段落、句子和

关键词以及词性和时态等，然后把它们提交给后面的语音合成引擎，语音合成引擎把前台处理的结果合成为声音。由于自然语言非常复杂，同一个词在不同的语义环境下，其发声是不同的，此外，还要支持儿童、老人、男女及多种语言，整个工作难度非常大。

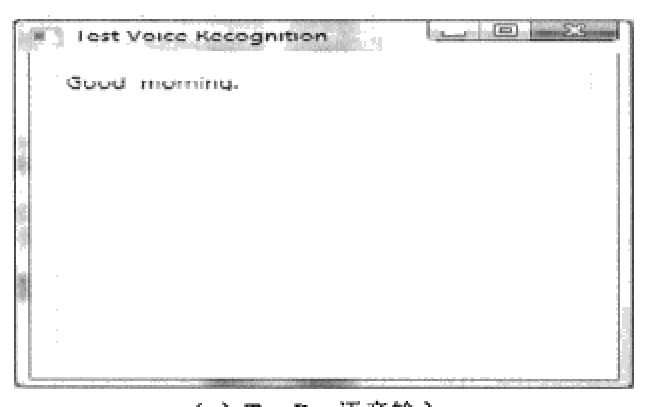
语音识别的技术难度更大，它不仅要支持不同语言，还要支持同一种语言的不同发音。我们知道即使是使用同一种语言，表达同一个意思，完全可以用不同的说法。这就是为什么虽然计算机界对这个问题研究了几十年，还没有真正实用化的原因。微软从1994年发布SAPI（Speech应用程序开发接口）第1版以来，做了大量的研究工作，并计划在其未来的主流产品中集成语音功能。

16.3.1 尝试Windows Vista的语音功能

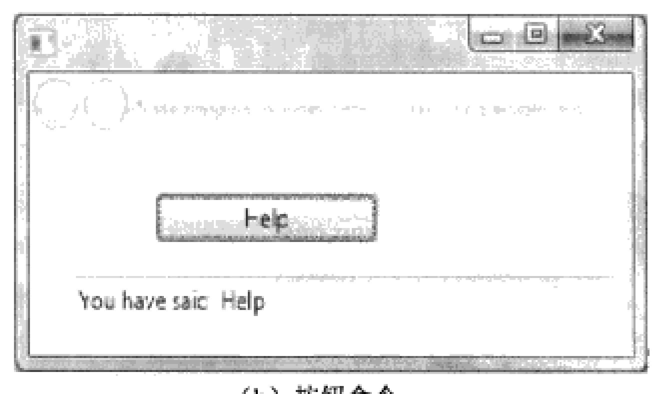
Windows Vista配有语音功能，在目录C:\ProgramData\Microsoft\Windows\Start Menu\Programs\Accessories\Accessibility下运行Windows Speech Recognition.exe程序，可以看到图16-6所示的界面。在黑色区域显示的是语音识别程序的状态，用鼠标右击话筒图标，它会弹出一个菜单，你可以通过该菜单设置语音识别的一些功能。一般在使用该应用程序之前，需要对其进行训练，这样会提高其识别的准确度。然后就可以对着话筒下命令了，比如可以说："Open Internet Explore"看看它是否启动浏览器，等等。

图16-6 Windows Vista中的语音识别程序界面

你可以创建一个简单的应用程序，在主视窗中放一个TextBox，单击TextBox使其具有输入焦点，然后我们对着话筒说话，比如说"Good morning period"，TextBox就显示"Good morning."字符串（如图16-7（a）所示）。



（a）TextBox语音输入



（b）按钮命令

图16-7 尝试语音输入

让我们来看另外一个小程序，这是一个网页，其中含有一个按钮和一个文字输入框：

```
<Page x:Class="Yingbao.Chapter16.SpeechRecognitionEx1.ButtonPage"
 xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
   xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
   Title="Voice button">
   <Grid Height="146">
       <Button Height="23" Margin="71,27,143,0" Name="button1"
```

```
            VerticalAlignment="Top"
            Click="OnClickButton">Help</Button>
        <TextBox Height="23" Name="txtDisplay" Margin="23,66,12,0"
            VerticalAlignment="Top"/>
    </Grid>
</Page>
```

在C#中，移植一个传递事件处理函数OnClickButton，在触发单击按钮事件时，在文字输入框中显示“You have said 'Help'”：

```
namespace Yingbao.Chapter16.SpeechRecognitionEx1
{

    public partial class ButtonPage : Page
    {
        public ButtonPage()
        {
            InitializeComponent();
            button1.Focus();

        }
        void OnClickButton(object sender, RoutedEventArgs rea)
        {
            txtDisplay.Text = "You have said 'Help'";
            button1.Focus();
        }
    }
}
```

运行该程序，与通常我们用鼠标单击按钮不同，这次你对着话筒说“Help”，会看到图16-7（b）中所显示出的结果。你不需要做任何特别的工作，一切都是操作系统和WPF完成的。Windows Vista语音识别系统和WPF界面元素之间通过COM的自动化界面联系起来，若某个界面元素具有输入焦点，该元素自动接收语音命令。读到这里，你也许会觉得这个功能真棒——是的，如果一切正常的话！

笔者在后面加了一个“如果”，实际上，笔者最近一直在尝试语音功能，结果并不理想。原因在于太多的错误失去了进一步使用的信心。原来以为这可能是笔者本人的口音问题，为此，专门请本地土生土长的加拿大人和美国人使用，结果大同小异！这可能就是目前语音识别的水平吧。

16.3.2 使你的程序发音

在WPF中使用应用程序发音非常简单，只要使用SpeechSynthesizer类即可：

```
SpeechSynthesizer synth = new SpeechSynthesizer();
synth.Speak("Hello everyone! ");
 synth.Speak("I am a computer, but I can speak!");
```

也可以使用异步发音，这时要用SpeakAsync方法代替Speak方法。Speak、SpeakAsync方法有三个重载：

- Speak(string)
- Speak(Prompt)

- Speak(PromptBuilder)

前面我们使用了第一个重载，即调用Speak。调用第二个重载方法需要用到Prompt，或FilePrompt类作为参数。使用FilePrompt的好处是，可以把要计算机语音输出的文字放到一个文件中，将来在需要改变语音输出内容时，只需修改该文件即可，而不需要修改源程序。如：

```
synth.SpeakAsync( new
   FilePrompt(@"c:\yingbaoproject\WPF\Test\content.dat",
                                        SynthesisMediaType.Text );
```

调用第三个方法，需要用到PromptBuilder。16.3.3节将介绍这个类。

16.3.3 PromptBuilder和SSML

为了使计算机更好地发音，W3C于2004年制定了一个新的标准——SSML语言（http://www.w3.org/TR/speech-synthesis/）。SSML是Speech Synthesis Markup Language的速写，可译为语音合成标记语言。这个语言是建立在XML上的，其中定义了语速、强调、输出语言（如英语或中文）等的语法，用来指导计算机“说话”。如：

```
<?xml version="1.0"?>
<speak version="1.0" xmlns="http://www.w3.org/2001/10/synthesis"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://www.w3.org/2001/10/synthesis
                   http://www.w3.org/TR/speech-
synthesis/synthesis.xsd"
         xml:lang="en-US">
  That is a <emphasis> big </emphasis> car!
  That is a <emphasis level="strong"> huge </emphasis>
  bank account!
</speak>
```

若你编写好了SSML文件，可以调用SpeechSynthesizer类中的SpeakSsml或SpeakSsmlAsync方法来让计算机输出语音。如不熟悉SSML语言，.NET提供的PromptBuilder类，可以让你像使用StringBuilder构建字符串一样方便地构建SSML。例如：

```
PromptBuilder BuildSSML()
        {
            PromptBuilder pb = new PromptBuilder();
            pb.StartVoice("Microsoft Lili");
            pb.AppendText("现在是北京时间早上");
            pb.AppendTextWithHint(DateTime.Now.ToString("hh:mm"),
SayAs.Time);
            pb.AppendBreak(new TimeSpan(0,0,3));
            pb.EndVoice();

            pb.StartVoice("Microsoft Sam");
            pb.AppendText("Beijing Time, it is ");
            pb.AppendTextWithHint(DateTime.Now.ToString("hh:mm"),
SayAs.Time);
            pb.AppendBreak(new TimeSpan(0,0,3));
            pb.EndVoice();
```

```
            return pb;
        }
```

在上面这段小程序中，笔者使用了两种语言让计算机报时。Microsoft Lili是中文合成器，可以免费下载，但在Vista上安装需要自己做点事（参见http://www. speaktext.com/faq.htm#InstallMSLili中安装Lili的部分），Windows 7对多语言语音功能有更好的支持。

不仅可以用PromptBuilder构造SSML，还可以用PromptBuilder读入SSML文件，如：

```
bp.AppendSsml(@"c:\yingbaoproject\wpf\test\content.ssml")
```

这句话直接把描述语音合成的ssml文件读入到PromptBuilder中。PromptBuilder的另一个功能是把其中的内容转化为XML输出（即产生SSML）：

```
string ssmlResult = bp.ToXml ()
```

这样，你可以在一个程序中用PromptBuilder产生SSML文件，而在其他程序中使用该文件。

16.3.4 语音识别中的语法

语音识别中最困难的部分是辨别语音输入的语法，使用语法可以提高语音识别的精度。W3C Voice Browser Activity工作组于2004 年发布了Speech Recognition Grammar Specification 1.0版（一般简称为SRGS，见http://www.w3.org/ TR/ speechgrammar/）。.NET中的System.Speech.Recognition.SrgsGrammer命名空间就是支持SRGS标准的API，SRGS标准也是基于XML的。对于简单的语法，可以直接使用System.Speech.Recognition命名空间中的Grammar、DictationGrammer和GrammarBuilder类。比如要捕获颜色的关键词，可以用：

```
Choices cColor = GetColorChoices();
GrammarBuilder gb = new GrammarBuilder(cColor);
Grammar grammarColors = new Grammar(gb);
SpeechRecognizer sr = new SpeechRecognizer();
sr.LoadGrammar(grammarColors);
```

还可以用第5章中用过的Reflection方法来获取系统中的颜色名：

```
private Choices GetColorChoices()
{
     Choices cColor = new Choices();

    Type t = typeof(Colors);
    MemberInfo[] mia = t.GetMembers(BindingFlags.Public |
       BindingFlags.Static);
    foreach (MemberInfo mi in mia)
    {
        if (mi.Name.StartsWith("get_") == true)
            continue;
        cColor.Add(mi.Name);
    }

    return cColor;
}
```

Choices管理字符串数组，说明语音识别引擎需要捕获的关键词。

SrgsDocument类和XMLDocument类很相似，它可以读入GrammarBuilder、SrgsRules或XML文件，也可以产生.srgs文件。SRGS文件是XML文件，以Grammar为根元素，其中含有一个或多个Rule元素，请看下面的YesNo规则：

```
<grammar xml:lang="en-US" version="1.0"
xmlns="http://www.w3.org/2001/06/grammar" tag-format="semantics-
ms/1.0">
<rule id="YesNo" scope="public">
      <example> yes </example>
      <example> sure </example>
      <example> no </example>
      <example> I don't think so </example>
      <one-of>
              <item>
                  <ruleref uri="#Yes" />
              </item>
              <item>
                  <ruleref uri="#No" />
              </item>
      </one-of>
      <tag> $ = $$ </tag>
</rule>
     ...
</grammar>
```

SrgsDocument类中包含SrgsRule类集合，SrgsRule类中可以包含下述几种元素或元素的集合：

- SrgsItem;
- SrgsNameValueTag;
- SrgsOneOf;
- SrgsRuleRef;
- SrgsSemanticInterpretationTag;
- SrgsSubSet;
- SrgsText;
- SrgsToken;

有了.srgs文件，就可以把该文件加载到SrgsDocument中来。然后把相应的语法规则和SpeechRecognizer实例联系起来：

```
public  class SpeechRecongNizerTest
{
   SpeechRecognizer sr = new SpeechRecognizer();
       ...
       void LoadSRGSDocument()
        {
           SrgsDocument sdocumnet = new
```

```
            SrgsDocument(@"C:\Yingbaoproject\WPF\Test\Sample.srgs");
              sr.LoadGrammar(new Grammar(sdocumnet, "YesNo"));
        }
}
```

使用SRGS标准，可以创建比较实用的应用程序。

16.4 本章小结

本章介绍了在WPF中使用音像、语音合成和语音播放技术，这些技术的底层都是使用微软的COM技术，因此，它们并不仅限于用在WPF软件中，所有的COM客户软件都可以通过COM接口使用COM所提供的服务。语音技术还在不断发展的过程中，这是一个活跃的基础研究领域。

第17章 定制控件和排版

第6章WPF控件中详细地介绍了WPF中的各种控件，第9章讨论了通过设定风格，来使应用程序中的控件有相同的视觉效果。第10章，比较深入地涉及控件模板这一课题，通过修改控件模板，可以完全改变一个控件的样子。虽然我们有比Win32及.NET WinForm中的控件更灵活有效的技术手段来使用WPF控件，但是在某些情况下，仍需定制自己的控件，本章讨论WPF中定制控件和排版技术。

在定制控件之前，我们一定要对WPF中现有的控件进行评估，如果现有的控件通过设置风格、模板等技术可以达到要求，那么，应该使用现有的控件。如果现有的控件不能满足要求（通常不只是控件的外观），那么需要定制自己的控件。

17.1 用户控件和自定义控件

定制控件有两种方法：一是创建用户控件（User Control）；二是创建自定义控件（Custom Control）。

一个用户控件（User Control）和一个视窗或网页在本质上没什么区别。在Visual Studio里的选择一个项目，单击鼠标右键，Visual Studio会弹出一个菜单。选择“Add”条目，可以看到Visual Studio允许我们创建Window、Page和User Control，如图17-1所示。这说明Visual Studio把用户控件和Window即Page并列在同一层次上。Visual Studio 会自动创建一个XAML文件和一个C#文件（若你使用C#），这一点也和加入Window和Page相同。

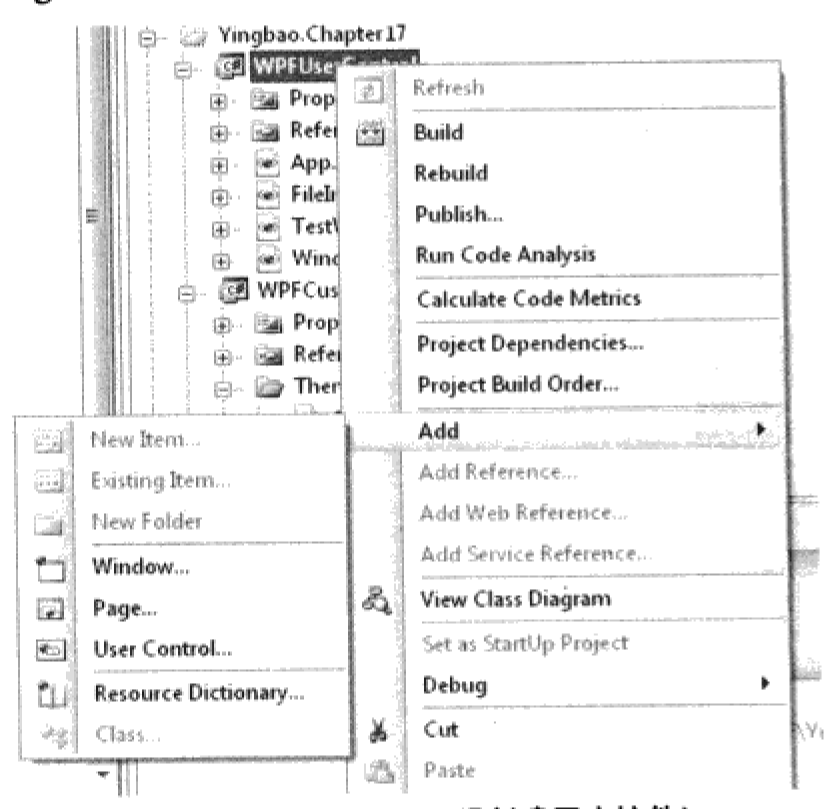


图17-1 （在Visual Studio里创建用户控件）

在创建自定义控件（Custom Control)时，情况则有所不同，需要选择“Add/ New Item”，然后选

择Custom Control 模板。有意思的是Visual Studio没有像创建Window或Page那样，自动产生同名的XAML和C#文件；而是创建了一个自定义控件的C#文件和一个Themes文件夹，同时在该文件夹下产生一个Generic.xaml的文件（笔者会在后面考察Generic.xaml文件中的内容）。如果再在该项目下添加新的自定义控件，Visual Studio会自动修改Generic.xaml文件。

用户控件通常用在自己的项目里，自定义控件一般有范围更大的用户群。如果你准备开发WPF控件并在市场上出售，那么应该选择自定义控件。

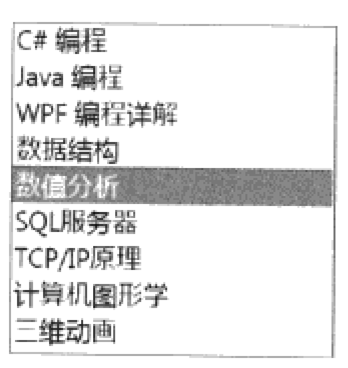
用户控件是UserControl的派生类，而UserControl的直接基类为ContentControl，这是固定的，ContentControl只能有一个直接子元素。开发自定义控件首先是要选择一个WPF控件作为自定义控件的基类，其原则是选择功能上最接近所要开发的自定义控件的类为基类。有时候，选择开发自定义控件还是用户控件并不那么明显，比如，你要开发的控件最接近内容控件，那么是选择用户控件还是自定义控件？

这个时候，我们需要进一步考察所要开发控件的功能。自定义控件允许用户彻底取代其控件模板（ControlTemplate），而用户控件一般是最终使用者自己创建的，通常不允许用户完全取代控件模板。

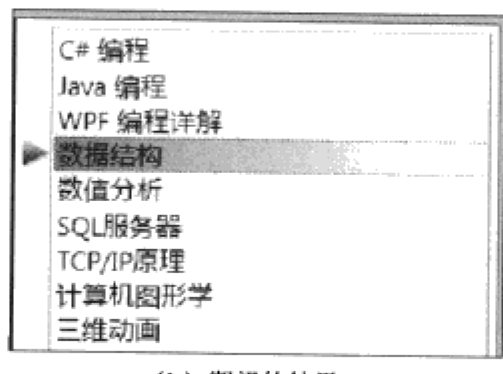
众所周知在软件开发过程中，说明一个概念的最好方法是解决实际问题。下面就用一个实例来说明如何创建用户控件和自定义控件。

假定要用WPF创建一个程序，你的任务是让计算机系的同学在课程表中选课。你可以用WPF的ListBox来显示学校里的课程列表，如图17-2（a）所示。

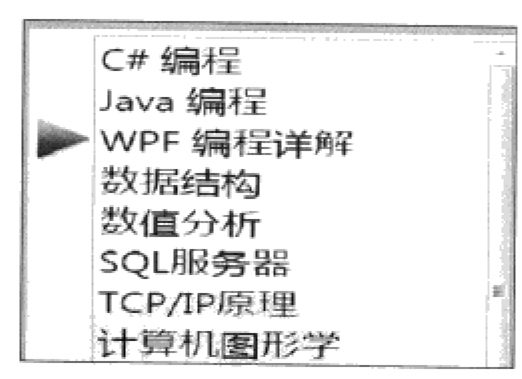
我们希望当学生选课时，在ListBox左边显示一个三角形符号来表示课程选中。如图17-2(b)或图17-2(c)所示。这个三角符号只是表示课程被选中，它并不参加和用户的交互，所以，若单击三角符号，并不影响条目的选中与否。如果移动滚动条，三角符号也应该和相应的条目一块移动。



（a）使用WPF ListBox控件选择课程列表



（b）期望的结果一



（c）期望的结果二

图17-2 选课问题

有多种方案来实现上面的结果，首先把课程表放到一个XML文件中（CourseList.xml），这样可以随时加入更多的课程：

```
<?xml version="1.0" encoding="utf-8" ?>
<CourseList  xmlns:ChinaAddress="http://yingbao.com/Computer/Courses">
        <Course>C# 编程 </Course>
        <Course>Java 编程</Course>
        <Course>WPF 编程详解</Course>
        <Course>数据结构</Course>
        <Course>数值分析</Course>
```

```
        <Course>SQL服务器</Course>
        <Course>TCP/IP原理</Course>
        <Course>计算机图形学</Course>
        <Course>三维动画</Course>
        <Course>电路原理</Course>
        <Course>电子技术（1）</Course>
        <Course>电子技术（2）</Course>
        <Course>半导体物理（1）</Course>
        <Course>半导体物理（2）</Course>
        <Course>固体物理（1）</Course>
        <Course>固体物理（2）</Course>
        <Course>计算机操作系统</Course>
        <Course>计算方法</Course>
</CourseList>
```

为了在XAML中使用这一XML文件，需要把该XML文件作为资源引入到Window的资源中：

```
<XmlDataProvider x:Key="computerCouse" Source="CourseList.xml"
XPath="CourseList/Course" />
```

17.2 创建用户控件（User Control）

现在我们用用户控件来实现上面的构想，其想法是在ListBox的外面构建一个显示三角形的地方，当ListBox中的条目被选中时，显示该三角形。

让我们在Visual Studio中，加入一个名为CourseListUserControl。可以看到，Visual Studio自动产生CouseListUserControl.xaml和CourseListUserControl.xaml.cs两个文件。在CourseListUserControl.xaml.cs中，Visual Studio产生下面的代码：

```
public partial class CourseListUserControl : UserControl
{
      public CourseListUserControl()
       {
          InitializeComponent();
       }
}
```

这几行代码的意义只是表示我们的用户控件CourseListUserControl是从UserControl中派生出来的。

17.2.1 设计用户控件UI

CourseListUserControl的UI由两部分组成，右边为WPF的ListBox，左边为显示三角形的区域。可以应用下面的XAML来控制版面：

```
<UserControl>
  <DockPanel>
    <Grid DockPanel.Dock="Left" />
    <ListBox />
  </DockPanel>
```

```
</UserControl>
```

左边为Grid控件，将在其中显示三角形，右边为ListBox控件。由于显示选中条目的区域在ListBox的外面，故需要解决当某个条目选中时，自动在该条目的位置显示一个三角形，当在ListBox中增加或减少一个条目时，左边的Grid中也会增加或减少一行，当用户滚动ListBox时，左边的Grid也要跟着一起滚动。ItemsControl默认排版是使用StackPanel，它把其中的Item沿着垂直方向排列起来，而我们需要画图，所以在其中使用了Canvas：

```
<Grid DockPanel.Dock="Left" >
  <ItemsControl x:Name="_indicatorList" >
      <ItemsControl.ItemsPanel>
          <ItemsPanelTemplate>
              <Canvas />
          </ItemsPanelTemplate>
      </ItemsControl.ItemsPanel>
    ...
 </ItemsControl>
</Grid>
```

我们的目的是在Canvas里面画一个三角形表示ListBox中的条目被选中。由于Canvas在ItemsPanelTemplate中，其左右大小是固定的，故需要计算的只是三角形在垂直方向的相应于选中条目的位置。先假定该位置已经算好，要显示三角形，然后使用DataTemplate：

```
  <ItemsControl.ItemTemplate>
     <DataTemplate>
       <Grid Width="16" Height="16">
          <Polygon Fill="LightGray">
              <Polygon.Points>
                  <Point X="4" Y="4" />
                  <Point X="16" Y="10" />
                  <Point X="4" Y="16" />
              </Polygon.Points>
          </Polygon>
          <Polygon Fill="{Binding ElementName=courseListUserControl,
                 Path=IndicatorBrush}">
                 <Polygon.Points>
                      <Point X="2" Y="2" />
                      <Point X="14" Y="8" />
                      <Point X="2" Y="14" />
                 </Polygon.Points>
          </Polygon>
        </Grid>
      </DataTemplate>
  </ItemsControl.ItemTemplate>
```

我们在这里用两个Polygon来画三角形，第一个用LightGray，这是用来绘制三角形的投影部分，从而使三角形具有立体效果。第二个Polygon绘制所要的三角形，注意我把其填充色绑定到courseListUserControl的IndicatorBrush上，用户可以根据需要来设置三角形的填充色。IndicatorBrush是我们为CourseListUserControl定义的一个相关属性，稍后再加以说明。

CourseListUserControl的完整XAML如下：

```
<UserControl
x:Class="Yingbao.Chapter17.WPFUserControl.CourseListUserControl"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    x:Name="courseListUserControl">
    <DockPanel ClipToBounds="True">
        <Grid DockPanel.Dock="Left" Width="16">
            <ItemsControl x:Name="_indicatorList" Focusable="False" >
                <ItemsControl.ItemsPanel>
                    <ItemsPanelTemplate>
                        <Canvas />
                    </ItemsPanelTemplate>
                </ItemsControl.ItemsPanel>
                <ItemsControl.ItemContainerStyle>
                    <Style TargetType="ContentPresenter">
                     <Setter Property="Canvas.Top"
                            Value="{Binding Path=.}" />
                    </Style>
                </ItemsControl.ItemContainerStyle>
                <ItemsControl.ItemTemplate>
                    <DataTemplate>
                        <Grid Width="16" Height="16">
                            <Polygon Fill="LightGray">
                                <Polygon.Points>
                                    <Point X="4" Y="4" />
                                    <Point X="16" Y="10" />
                                    <Point X="4" Y="16" />
                                </Polygon.Points>
                            </Polygon>
                            <Polygon Fill="{Binding
                               ElementName=courseListUserControl,
                               Path=IndicatorBrush}">
                                <Polygon.Points>
                                    <Point X="2" Y="2" />
                                    <Point X="14" Y="8" />
                                    <Point X="2" Y="14" />
                                </Polygon.Points>
                            </Polygon>
                        </Grid>
                    </DataTemplate>
                </ItemsControl.ItemTemplate>
            </ItemsControl>
        </Grid>
         <ListBox x:Name="_listBox"  Style="{Binding
                  ElementName=courseListUserControl,
                  Path=ListBoxStyle}"/>
    </DockPanel>
</UserControl>
```

17.2.2 开发支持用户控件UI的逻辑

前面提到，我们需要计算出三角形在垂直方向的位置。这个位置和ListBox中所选择的条目有关，因此，一旦所选择的条目发生了变化，便需要重新计算；若用户滚动ListBox，也需要重新计算三角形的位置。当ListBox的选择模式（SelectionMode)为Multiple或Extend模式时，用户可以同时选择ListBox中的多个条目，为此，我们定义一个_indicatorOffsets:

```
readonly ObservableCollection<double> _indicatorOffsets;
```

计算三角形的位置，我们需要获取ListBox中某个条目的位置，方法是使用GeneralTransform类：

```
GeneralTransform trans = lbItem.TransformToAncestor(_listBox);
Point location = trans.Transform(new Point(0, 0));
```

一旦有了ListBox中的相应条目位置，就很容易得到三角形在垂直方向的位置：

```
double offset = location.Y + (lbItem.ActualHeight / 2) -
(INDICATOR_HEIGHT / 2);
```

其中lbItem为ListBox中的条目。

除了计算三角形在垂直方向的位置之外，还需要支持设定三角形的填充色，为此我们定义一个相关属性IndicatorBrush：

```
public Brush IndicatorBrush
        {
                get { return (Brush)GetValue(IndicatorBrushProperty); }
                set { SetValue(IndicatorBrushProperty, value); }
        }
        public static readonly DependencyProperty IndicatorBrushProperty
=
                DependencyProperty.
Register("IndicatorBrush",typeof(Brush),
      typeof(CourseListUserControl),new PropertyMetadata(
       SystemColors.HighlightBrush));
```

最后，我们定义相关属性ListBoxStyle，这个相关属性主要用于支持数据绑定。

```
public static readonly DependencyProperty ListBoxStyleProperty =
            DependencyProperty.Register("ListBoxStyle", typeof(Style),
            typeof(CourseListUserControl));
 public ListBox ListBox
          {
                get { return _listBox; }
          }
  public Style ListBoxStyle
           {
                get { return (Style)GetValue(ListBoxStyleProperty); }
                set { SetValue(ListBoxStyleProperty, value); }
           }
```

完整的C#程序如下：

```
using System;
```

```
using System.Collections.Generic;
using System.Collections.ObjectModel;
using System.Text;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Controls.Primitives;
using System.Windows.Data;
using System.Windows.Documents;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Media.Imaging;
using System.Windows.Navigation;
using System.Windows.Shapes;
namespace Yingbao.Chapter17.WPFUserControl
{
    public partial class CourseListUserControl : UserControl
    {
        const int INDICATOR_HEIGHT = 16;
        readonly ObservableCollection<double> _indicatorOffsets;
        public CourseListUserControl()
        {
            InitializeComponent();
            _indicatorOffsets = new ObservableCollection<double>();
            _indicatorList.ItemsSource = _indicatorOffsets;
            _listBox.SelectionChanged += delegate
            {
                this.UpdateIndicators();
            };
          _listBox.AddHandler(ScrollViewer.ScrollChangedEvent,
                new ScrollChangedEventHandler(delegate
             { this.UpdateIndicators(); }
                 ));
               }
        public Brush IndicatorBrush
        {
               get { return (Brush)GetValue(IndicatorBrushProperty); }
               set { SetValue(IndicatorBrushProperty, value); }
        }

        public static readonly DependencyProperty IndicatorBrushProperty
            = DependencyProperty.Register("IndicatorBrush",
              typeof(Brush), typeof(CourseListUserControl),new
              PropertyMetadata(SystemColors.HighlightBrush));

       public static readonly DependencyProperty ListBoxStyleProperty =
              DependencyProperty.Register("ListBoxStyle",
              typeof(Style), typeof(CourseListUserControl));

        public ListBox ListBox
        {
               get { return _listBox; }
        }
       public Style ListBoxStyle
```

```
        {
           get { return (Style)GetValue(ListBoxStyleProperty); }
                       set { SetValue(ListBoxStyleProperty, value); }
        }

        void UpdateIndicators()
        {
           if (_indicatorOffsets.Count > 0)
           _indicatorOffsets.Clear();
           if (_listBox.SelectedItems.Count == 0) return;
           ItemContainerGenerator gen = _listBox.ItemContainerGenerator;
           if (gen.Status != GeneratorStatus.ContainersGenerated)
           return;
           foreach (object selectedItem in _listBox.SelectedItems)
           {
               ListBoxItem lbItem = gen.ContainerFromItem(selectedItem)
                  as  ListBoxItem;
               if (lbItem == null) continue;
               GeneralTransform trans =
                         lbItem.TransformToAncestor(_listBox);
               Point location = trans.Transform(new Point(0, 0));
               double offset = location.Y + (lbItem.ActualHeight / 2)
                    - (INDICATOR_HEIGHT / 2);
               _indicatorOffsets.Add(offset);
           }
        }
    }
}
```

现在，我们来使用CourseListUserControl，为此创建一个TestWindow窗口。

```
<Window x:Class="Yingbao.Chapter17.WPFUserControl.TestWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:sys="clr-namespace:System;assembly=mscorlib"
    xmlns:local="clr-namespace:Yingbao.Chapter17.WPFUserControl"
    Title="使用用户控件" Height="300" Width="336">
    <Window.Resources>
        <XmlDataProvider x:Key="computerCouse" Source="CourseList.xml"
                         XPath="CourseList/Course" />
        <ObjectDataProvider x:Key="SelectionModeValues"
              MethodName="GetValues" ObjectType="{x:Type sys:Enum}">
            <ObjectDataProvider.MethodParameters>
                <x:Type TypeName="SelectionMode" />
            </ObjectDataProvider.MethodParameters>
        </ObjectDataProvider>
        <LinearGradientBrush x:Key="IndicatorBrush" StartPoint="0.5,0"
             EndPoint="0.5,1">
            <GradientStop Color="LightBlue" Offset="0.1" />
            <GradientStop Color="Blue" Offset="1" />
        </LinearGradientBrush>
        <Style x:Key="BlankSelectionStyle">
            <Style.Resources>
```

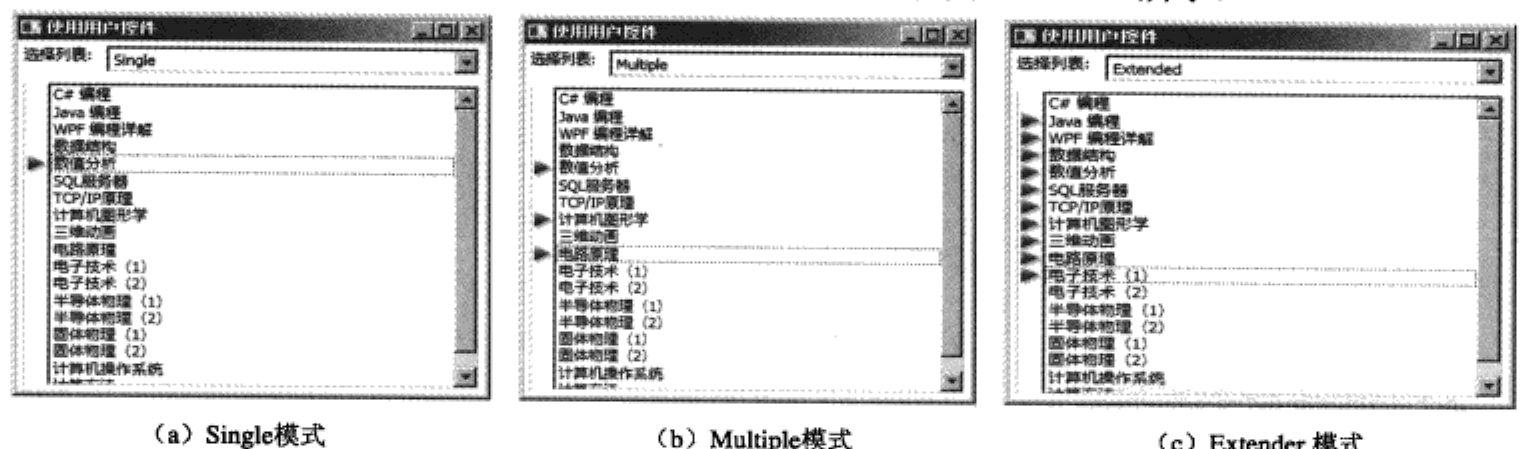
```
            <SolidColorBrush x:Key="{x:Static
             SystemColors.HighlightBrushKey}"  Color="Transparent"/>
             <SolidColorBrush x:Key="{x:Static
                SystemColors.ControlBrushKey}" Color="Transparent"/>
              <SolidColorBrush
               x:Key="{x:Static SystemColors.HighlightTextBrushKey}"
               Color="{x:Static SystemColors.ControlTextColor}"/>
          </Style.Resources>
      </Style>
      <Style x:Key="ListBoxStyle" TargetType="ListBox">
          <Setter Property="DataContext"
             Value="{StaticResource computerCouse}"/>
          <Setter Property="ItemsSource" Value="{Binding Path=.}" />
          <Setter Property="ItemContainerStyle"
                  Value="{StaticResource BlankSelectionStyle}" />
       </Style>
      <Style TargetType="{x:Type local:CourseListUserControl}">
          <Setter Property="IndicatorBrush"
                 Value="{StaticResource IndicatorBrush}" />
          <Setter Property="ListBoxStyle"
                 Value="{StaticResource ListBoxStyle}" />
      </Style>
  </Window.Resources>
  <Grid>
      <Grid.RowDefinitions>
          <RowDefinition Height="Auto" />
          <RowDefinition Height="*" />
      </Grid.RowDefinitions>
      <DockPanel Grid.Row="0" Margin="4">
          <TextBlock DockPanel.Dock="Left">选择列表:</TextBlock>
          <Rectangle DockPanel.Dock="Left" Width="10" />
          <ComboBox  x:Name="_selectionModeCombo"
              DataContext="{StaticResource SelectionModeValues}"
              IsReadOnly="True" ItemsSource="{Binding Path=.}"
              SelectedIndex="0"/>
      </DockPanel>
     <Border BorderBrush="{x:Static SystemColors.ActiveBorderBrush}"
        BorderThickness="1,0,0,0" Grid.Row="1" Margin="4">
          <local:CourseListUserControl
                 x:Name="_listBoxWithIndicator" />
     </Border>
  </Grid>
</Window>
```

C#后台程序如下：

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Data;
```

```
using System.Windows.Documents;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Media.Imaging;
using System.Windows.Shapes;
namespace Yingbao.Chapter17.WPFUserControl
{
    public partial class TestWindow : Window
    {
        public TestWindow()
        {
            InitializeComponent();
                this._selectionModeCombo.SelectionChanged +=
            this.OnSelectionModeChanged;
            }
           void OnSelectionModeChanged(object sender,
             SelectionChangedEventArgs e)
            {
            if (_selectionModeCombo.SelectedIndex < 0)
                          return;
            SelectionMode mode =
                 (SelectionMode)_selectionModeCombo.SelectedItem;
            _listBoxWithIndicator.ListBox.SelectionMode = mode;
            }
    }
}
```

上面这段程序的运行结果如图17-3（a）、图17-3（b）和图17-3（c）所示：



（a）Single模式　　（b）Multiple模式　　（c）Extender 模式

图17-3 使用CourseListUserControl的结果

17.3 创建自定义控件（Custom Control）

在17.2节中我们利用用户控件开发了CourseListUserControl，如果你觉得这个控件会有更多人感兴趣，甚至你想开发一套WPF的第三方控件并在市场上出售。那么，一个做法就是把这个控件开发成一个自定义控件。

我们在Visual Studio里加入一个新项目WPFCustomControl，并把这个项目设置为类库，使其编译后成为独立的DLL。然后，我们在该项目下添加一个自定义控件，由于这个控件可能不仅仅用在显示课程表上，所以给这个控件取一个更一般的名字：ListBoxWithIndicator。注意到VisualStudio为我们

创建了一个名为ListBoxWithIndicator类：

```
public class ListBoxWithIndicator : ContentControl
    {
      static ListBoxWithIndicator()
      {
          DefaultStyleKeyProperty.OverrideMetadata(
              typeof(ListBoxWithIndicator), new
              FrameworkPropertyMetadata(typeof(ListBoxWithIndicator)));
        }
    }
```

Visual Studio 在创建这个类时，是从Control类中派生出来的。从前面开发CourseListUserControl时，我们知道UserControl是从ContentControl中派生出来的，所以可选用ContentControl作为ListBoxWithIndicator的基类。Visual Studio在ListBoxWithIndicator的静态构造函数里调用了DefaultStyleKeyProperty. OverrideMetaData。相关属性DefaultStyleKeyProperty是FrameworkElement以及FrameworkContentElement类用来指示控件的默认风格键值的属性，利用这个属性来获取控件的所有风格。例如：

```
object  defaultStyleKey = someElement.GetValue(
     FrameworkElement.DefaultStyleKeyProperty ) ;
Style style =
           (Style) Application.Current.FindResource(defaultStyleKey) ;
string xaml = System.Windows.Markup.Xamlwriter.Save( Style );
```

无论你把某个元素的风格放在Application，Window或其他元素的资源里，DefaultStyleKeyProperty 表示某个FrameworkElement或FrameworkContentElement的全部风格的综合。调用OverrideMetadata表示自定义类将使用自己的风格，所以，有人又把调用该函数叫做使自己的UI元素风格化。

那么我们的ListBoxWithIndicator的风格在哪里呢？前面提到，Visual Studio为我们创建了另外一个文件：Themes\Generic.xaml。

```
<Style TargetType="{x:Type local: ListBoxWithIndicator}">
  <Setter Property="Template">
      <Setter.Value>
          <ControlTemplate
              TargetType="{x:Type local: ListBoxWithIndicator}">
              <Border Background="{TemplateBinding Background}"
                BorderBrush="{TemplateBinding BorderBrush}"
                BorderThickness="{TemplateBinding
                BorderThickness}">
              </Border>
          </ControlTemplate>
      </Setter.Value>
  </Setter>
</Style>
```

每添加一个自定义控件，Visual Studio会自动在Generic.xaml文件中添加一节Style。我们将在ListBoxWithIndicator的风格中加入更多的内容，所以接着讨论ListBoxWithIndicator类。我们采用

CourseListUserControl一样的排版方式，即左边为ItemsControl右边为ListBox：

```
private ItemsControl _indicatorList;
private ListBox _listBox;
```

在_indicatorOffsets中放入三角形的位置：

```
private ObservableCollection<Double> _indicatorOffsets;
```

我们还需要支持在XAML中修改三角形的颜色和大小的功能，为此，定义了两个相关属性：

```
public static readonly DependencyProperty IndicatorBrushProperty
       = DependencyProperty.Register( "IndicatorBrush",
         typeof(Brush), typeof (ListBoxWithIndicator),
         new PropertyMetadata( new
                LinearGradientBrush(Colors.LightBlue, Colors.Blue,
                 new Point(0.5, 0), new Point(0.5, 1))));

public static readonly DependencyProperty
         IndicatorHeightWidthProperty =
          DependencyProperty.Register("IndicatorHeightWidth",
                typeof(Double), typeof(ListBoxWithIndicator), new
                PropertyMetadata(16.0));

[Description("Brush used to paint the indicator.  Defaults to a nice blue
gradient brush"), Category("Custom")]
public Brush IndicatorBrush
{
    get {
          return GetValue(IndicatorBrushProperty) as Brush;
        }
    set{
         SetValue( IndicatorBrushProperty,value );
       }
}
[Description("Size of indictor.  Indicator is rendered in a square so
this value is the height and width of the indicator.  Default value is
16."), Category("Custom")]
public Double  IndicatorHeightWidth
{
   get {
      return (Double)GetValue(IndicatorHeightWidthProperty);
   }
   set{
      SetValue( IndicatorHeightWidthProperty,value );
      }
}
```

我们使用Loaded事项来初始化ListBoxWithSelectedIndicator：

```
public ListBoxWithIndicator()
  {
     this.Loaded += new
          RoutedEventHandler(ListBoxWithSelectedItemIndicator_Loaded);
```

```
}

private void ListBoxWithSelectedItemIndicator_Loaded(object sender,
                RoutedEventArgs e)
  {
    if( _indicatorList != null )
    {
      this._indicatorOffsets = new ObservableCollection<double>();
      this._indicatorList.ItemsSource = this._indicatorOffsets;
      this.AddHandler(ListBox.SelectionChangedEvent, new
           SelectionChangedEventHandler(ListBox_SelectionChanged));
      this.AddHandler(ScrollViewer.ScrollChangedEvent, new
     ScrollChangedEventHandler( ListBox_ScrollViewer_ScrollChanged));
      UpdateIndicators();
    }
}
```

我们在UpdateIndicators()方法里，计算三角形在垂直方向的位置，其方法和上节相同。在这里不再赘述，下面是完整的ListBoxWithSelectedIndicator：

```
namespace Yingbao.Chapter17.WPFCustomControl
{
    [TemplatePart(Name ="PART_IndicatorList",
      Type=typeof(ItemsControl))]
    public class ListBoxWithIndicator : ContentControl
    {
        static ListBoxWithIndicator()
        {
            DefaultStyleKeyProperty.OverrideMetadata(
               typeof(ListBoxWithIndicator), new
               FrameworkPropertyMetadata(typeof(ListBoxWithIndicator)));
        }

        public ListBoxWithIndicator()
        {
             this.Loaded += new RoutedEventHandler(
                   ListBoxWithSelectedItemIndicator_Loaded);
        }

       private ItemsControl _indicatorList;
       private ObservableCollection<Double> _indicatorOffsets;
       private ListBox _listBox;

       public static readonly DependencyProperty
           IndicatorBrushProperty = DependencyProperty.Register(
            "IndicatorBrush", typeof(Brush),
            typeof (ListBoxWithIndicator),new PropertyMetadata(
             new LinearGradientBrush(Colors.LightBlue, Colors.Blue,
             new Point(0.5, 0), new Point(0.5, 1))));
       public static readonly DependencyProperty
            IndicatorHeightWidthProperty =
                  DependencyProperty.Register( "IndicatorHeightWidth",
                  typeof(Double), typeof(ListBoxWithIndicator), new
```

```
                PropertyMetadata(16.0));
    #region Properties
    [Description("Brush used to paint the indicator.  Defaults to
         a nice blue gradient brush"), Category("Custom")]
    public Brush IndicatorBrush
    {
       get {
           return GetValue(IndicatorBrushProperty) as Brush;
             }
       set{
           SetValue( IndicatorBrushProperty,value );
       }
    }
    [Description("Size of indictor.  Indicator is rendered in a
    square so this value is the height and width of the indicator.
    Default value is 16."), Category("Custom")]
    public Double  IndicatorHeightWidth
    {
       get {
           return (Double)GetValue(IndicatorHeightWidthProperty);
             }
       set{
           SetValue( IndicatorHeightWidthProperty,value );
           }
    }
 #endregion
 #region Methods
protected override void  OnContentChanged(object oldContent,
            object newContent)
{
    base.OnContentChanged(oldContent, newContent);
    if( newContent == null   || newContent is  ListBox)
    {
         _listBox= newContent as ListBox;
        if( _indicatorOffsets != null &&
              _indicatorOffsets.Count >0 )
        {
          _indicatorOffsets.Clear();
        }
    }
    else{
        throw new NotSupportedException(
         "Invalid content. istBoxWithSelectedItemIndicator
          only accepts a ListBox control as its content.");
    }
}

 public override void  OnApplyTemplate()
    {
        base.OnApplyTemplate();
       _indicatorList = GetTemplateChild("PART_IndicatorList")
              as ItemsControl;
       if( _indicatorList == null )
```

```
            {
              throw new   Exception("Hey!  The PART_IndicatorList is
missing from the template or is  not an ItemsControl.  Sorry but this
ItemsControl is required.");
            }
        }

    private void ListBoxWithSelectedItemIndicator_Loaded(object sender,
RoutedEventArgs e)
    {
      if( _indicatorList != null )
      {
        this._indicatorOffsets = new ObservableCollection<double>();
        this._indicatorList.ItemsSource = this._indicatorOffsets;
        this.AddHandler(ListBox.SelectionChangedEvent, new
            SelectionChangedEventHandler(ListBox_SelectionChanged));
        this.AddHandler(ScrollViewer.ScrollChangedEvent, new
       ScrollChangedEventHandler( ListBox_ScrollViewer_ScrollChanged));
        UpdateIndicators();
      }
     }
    private void UpdateIndicators()
    {
        if( this._indicatorOffsets != null && this._listBox != null )
        {
            if( this._indicatorOffsets != null &&
                  this._indicatorOffsets.Count > 0 )
            {
                this._indicatorOffsets.Clear();
            }
            if( this._listBox.SelectedItems.Count > 0)
            {
               ItemContainerGenerator icGen =
                   this._listBox.ItemContainerGenerator;
               if( icGen.Status !=
                    System.Windows.Controls.Primitives.GeneratorStatus
                    .ContainersGenerated)
               {
                    return;
               }
               foreach( object selectedItem in
                         this._listBox.SelectedItems )
               {
                  ListBoxItem lbItem = icGen.ContainerFromItem
                        ( selectedItem) as  ListBoxItem;
                  if( lbItem != null )
                  {
                    GeneralTransform gt = lbItem.
                                 TransformToAncestor(this._listBox );
                    Point pt = gt.Transform( new Point(0,0));
                    Double dblOffset = pt.Y + lbItem.ActualHeight / 2
                                - this.IndicatorHeightWidth/2;
                      this._indicatorOffsets.Add(dblOffset);
```

```
                    }
                }
            }
        }
    }

    private void ListBox_ScrollViewer_ScrollChanged(object sender,
                        ScrollChangedEventArgs e)
    {
        if( e.VerticalChange != 0 )
        {
          UpdateIndicators();
        }
    }
    private void ListBox_SelectionChanged(object sender ,
                    SelectionChangedEventArgs e )
    {
        UpdateIndicators();
    }
    private void ListBoxWithSelectedItemIndicator_Unloaded(object
                    sender, RoutedEventArgs e)
    {
        this.RemoveHandler(ListBox.SelectionChangedEvent, new
             SelectionChangedEventHandler( ListBox_SelectionChanged));
        this.RemoveHandler(ScrollViewer.ScrollChangedEvent, new
        ScrollChangedEventHandler(ListBox_ScrollViewer_ScrollChanged));
    }
    #endregion
    }
}
```

注意：笔者在ListBoxWithIndicator的前面加了TemplatePart(Name = "PART_Indicator List", Type=typeof(ItemsControl))属性，对这个自定义控件来说，非常重要。WPF使用PART_xxx这个惯例来标记控件模板中的某个部件（Part）。我们知道WPF中控件的样子和其控制逻辑是分开的，在模板中定义控件的视觉树，通常称WPF控件为无外观控件就是这个意思。有时候控件模板和控件逻辑之间需要协同工作，就需要一种在控件模板和控件之间交互机制。PART_xxx用来标记视觉树上的节点，我们在后台C#类中寻找这样的节点，如果能找到相应的视觉树上的节点，就说明我们的控件模板没有被用户替换掉，因此，可以对控件施加相应的逻辑。笔者在ListBoxWithIndicator类中覆盖了基类的OnApplyTemplate方法，当WPF把控件的模板和控件联系起来的时候调用该方法。在该方法里使用GetTemplateChild来获取我们的条目控件ItemsControl，如果该条目控件不存在，我们的控件模板一定被别人给替换了，而该条目控件是显示三角形所必需的。如果没有这个条目控件，我们的自定义控件就不能工作，所以要产生一个异常：

```
public override void  OnApplyTemplate()
{
     base.OnApplyTemplate();
     _indicatorList = GetTemplateChild("PART_IndicatorList")
                    as ItemsControl;
    if( _indicatorList == null )
```

```
    {
      throw new   Exception("Hey!  The PART_IndicatorList is
       missing from the template or is not an ItemsControl.
       Sorry but this ItemsControl is required.");
    }
  }
```

我们在Generic.xaml中定义ItemsControl视觉树，并命名为“Part_IndicatorList”：

```
<ItemsControl x:Name="PART_IndicatorList" Focusable="False">
     <ItemsControl.ItemsPanel>
       <ItemsPanelTemplate>
           <Canvas />
       </ItemsPanelTemplate>
    </ItemsControl.ItemsPanel>
   <ItemsControl.ItemContainerStyle>
       <Style TargetType="ContentPresenter">
         <Setter Property="Canvas.Top" Value="{Binding Path=.}" />
       </Style>
       </ItemsControl.ItemContainerStyle>
          <ItemsControl.ItemTemplate>
            <DataTemplate>
              <Grid Width="{Binding Path=IndicatorHeightWidth,
                 RelativeSource={RelativeSource FindAncestor,
                 AncestorType={x:Type
                 local:ListBoxWithIndicator}}}" Height="{Binding
                 Path=IndicatorHeightWidth,
                 RelativeSource={RelativeSource FindAncestor,
                 AncestorType={x:Type
                 local:ListBoxWithIndicator}}}">
                 <Path Fill="LightGray" Stretch="Uniform"
                       Data="M4,4 L16,10 L4,16 z"
                       RenderTransformOrigin="0.5,0.5"
                       SnapsToDevicePixels="True">
                    <Path.RenderTransform>
                    <TransformGroup>
                      <TranslateTransform X="2" Y="2"/>
                    </TransformGroup>
                 </Path.RenderTransform>
                </Path>

                <Path Fill="{Binding Path=IndicatorBrush,
                      RelativeSource={RelativeSource FindAncestor,
                      AncestorType={x:Type
                      local:ListBoxWithIndicator}}}"
                      Stretch="Uniform" Data="M2,2 L14,8 L2,14 z"
                      SnapsToDevicePixels="True"/>
                </Grid>
        </DataTemplate>
    </ItemsControl.ItemTemplate>
</ItemsControl>
```

和 17.2 节 CourseListUserControl 一样，这里的视觉树用作显示三角形。下面是完整的

Generic.xaml：

```
<ResourceDictionary
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="clr-namespace:Yingbao.Chapter17.WPFCustomControl"
    xmlns:System="clr-namespace:System;assembly=mscorlib">
    <Style TargetType="{x:Type local:ListBoxWithIndicator}">
        <Setter Property="Template">
            <Setter.Value>
                <ControlTemplate TargetType="{x:Type
                    local:ListBoxWithIndicator}">
                    <ControlTemplate.Resources>
                        SolidColorBrush x:Key="{x:Static
                             SystemColors.HighlightBrushKey}"
                                        Color="Transparent" />
                        <SolidColorBrush x:Key="{x:Static
                              SystemColors.ControlBrushKey}"
                                        Color="Transparent" />
                        <SolidColorBrush x:Key="{x:Static
                              SystemColors.HighlightTextBrushKey}"
                              Color="{x:Static
                               SystemColors.ControlTextColor}" />
                    </ControlTemplate.Resources>
                    <Border Background="{TemplateBinding Background}"
                          BorderBrush="{TemplateBinding BorderBrush}"
                          BorderThickness="{TemplateBinding
                          BorderThickness}"
                          Padding="{TemplateBinding Padding}">
                        <DockPanel>
                            <Grid ClipToBounds="True"
                         DockPanel.Dock="Left" Width="{TemplateBinding
                         IndicatorHeightWidth}" Margin="0,0,1,0">
                         <ItemsControl x:Name="PART_IndicatorList"
                                 Focusable="False">
                                        <ItemsControl.ItemsPanel>
                                        <ItemsPanelTemplate>
                                            <Canvas />
                                        </ItemsPanelTemplate>
                                    </ItemsControl.ItemsPanel>
                                    <ItemsControl.ItemContainerStyle>
                                       <Style TargetType
                                          ="ContentPresenter">
                                          <Setter Property=
                                          "Canvas.Top"
                                          Value="{Binding Path=.}" />
                                        </Style>
                                    </ItemsControl.ItemContainerStyle>
                                    <ItemsControl.ItemTemplate>
                                      <DataTemplate>
                                        <Grid Width="{Binding
                                           Path=IndicatorHeightWidth,
                                           RelativeSource={RelativeSource
```

```
                                        FindAncestor, AncestorType=
                                    {x:Type local:ListBoxWithIndicator}}}"
                                Height="{Binding Path=IndicatorHeightWidth,
                                RelativeSource={RelativeSource
                                FindAncestor, AncestorType={x:Type
                                local:ListBoxWithIndicator}}}"><Path
                                Fill="LightGray" Stretch="Uniform"
                                Data="M4,4 L16,10 L4,16 z"
                                RenderTransformOrigin="0.5,0.5"
                                SnapsToDevicePixels="True">
                                <Path.RenderTransform>
                                   <TransformGroup>
                                    <TranslateTransform X="2" Y="2"/>
                                  </TransformGroup>
                                </Path.RenderTransform>
                              </Path>
                              <Path Fill="{Binding Path=IndicatorBrush,
                                RelativeSource={RelativeSource
                                FindAncestor, AncestorType={x:Type
                                 local:ListBoxWithIndicator}}}"
                                  Stretch="Uniform" Data="M2,2 L14,8
                                  L2,14 z" SnapsToDevicePixels="True"/>
                              </Grid>
                            </DataTemplate>
                          </ItemsControl.ItemTemplate>
                        </ItemsControl>
                      </Grid>
                    <ContentPresenter Content="{TemplateBinding
                            Content}"/>
                  </DockPanel>
                </Border>
            </ControlTemplate>
        </Setter.Value>
    </Setter>
  </Style>
</ResourceDictionary>
```

现在使用我们刚刚创建的ListBoxWithIndicator自定义控件，只要用下面的XAML即可：

```
<YBCustomControls:ListBoxWithIndicator HorizontalAlignment="Stretch"
                    IndicatorHeightWidth="32" >
   <ListBox  x:Name="lbDemoTwo"  Width="Auto" Height="Auto"
        DataContext="{DynamicResource computerCouse}"
        ItemsSource="{Binding}" FontSize="24" SelectionMode="{Binding
        Path=Text, ElementName=SelectionModeCombo, Mode=OneWay}"
        ScrollViewer.CanContentScroll="False"/>
</YBCustomControls:ListBoxWithIndicator>
```

下面是一个完整的测试程序：

```
<Window x:Class="Yingbao.Chapter17.TestApp.AppWin"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:sys="clr-namespace:System;assembly=mscorlib"
```

```
xmlns:YBCustomControls="clr-namespace:Yingbao.Chapter17.
     WPFCustomControl;assembly=WPFCustomControl"
xmlns:SystemDrawingPrinting="clr-namespace:System.Drawing.
     Printing;assembly=System.Drawing"
Title="自定义列表框"    Height="Auto"
Width="Auto"
ResizeMode="CanResizeWithGrip"
SizeToContent="Manual">
<Window.Resources>
  <XmlDataProvider x:Key="computerCouse" Source="CourseList.xml"
       XPath="CourseList/Course" />
    <ObjectDataProvider x:Key="SelectionModeValues"
       MethodName="GetValues"
          ObjectType="{x:Type sys:Enum}">
        <ObjectDataProvider.MethodParameters>
            <x:Type TypeName="SelectionMode" />
        </ObjectDataProvider.MethodParameters>
    </ObjectDataProvider>
    <ObjectDataProvider x:Key="SystemDrawingPrintingValues"
        MethodName="GetValues" ObjectType="{x:Type sys:Enum}">
        <ObjectDataProvider.MethodParameters>
            <x:Type TypeName="SystemDrawingPrinting:PaperKind" />
        </ObjectDataProvider.MethodParameters>
    </ObjectDataProvider>

    <ObjectDataProvider x:Key="SystemDrawingPrintingValues2"
         MethodName="GetValues"  ObjectType="{x:Type sys:Enum}">
        <ObjectDataProvider.MethodParameters>
            <x:Type TypeName="SystemDrawingPrinting:PaperKind" />
        </ObjectDataProvider.MethodParameters>
    </ObjectDataProvider>

    <ObjectDataProvider x:Key="SystemDrawingPrintingValues3"
         MethodName="GetValues"  ObjectType="{x:Type sys:Enum}">
        <ObjectDataProvider.MethodParameters>
            <x:Type TypeName="SystemDrawingPrinting:PaperKind" />
        </ObjectDataProvider.MethodParameters>
    </ObjectDataProvider>
    <ObjectDataProvider x:Key="SystemDrawingPrintingValues4"
        MethodName="GetValues" ObjectType="{x:Type sys:Enum}">
        <ObjectDataProvider.MethodParameters>
            <x:Type TypeName="SystemDrawingPrinting:PaperKind" />
        </ObjectDataProvider.MethodParameters>
    </ObjectDataProvider>
</Window.Resources>
<Grid>
    <Grid.RowDefinitions>
        <RowDefinition Height="Auto"/>
        <RowDefinition Height="Auto"/>
        <RowDefinition/>
    </Grid.RowDefinitions>
    <TextBlock Width="Auto" Height="Auto" FontFamily="Consolas"
        FontSize="24" FontWeight="Bold" Foreground="#FF004BC9"
```

```
        Text="使用定制列表框"  TextAlignment="Center" x:Name="tbTitle"
        TextWrapping="Wrap" Background="#00000000"
        VerticalAlignment="Top"/>
    <Border HorizontalAlignment="Stretch" Margin="10,10,10,10"
        VerticalAlignment="Stretch" Width="Auto" Height="Auto"
        Grid.Row="2" BorderBrush="#FF222222" Padding="10,10,10,10"
        CornerRadius="20,20,20,20" BorderThickness="1,1,1,1" >
        <Border.Background>
            <LinearGradientBrush EndPoint="0.526,0.968"
               StartPoint="0.526,0.021">
                <GradientStop Color="#FF98B3FF" Offset="0"/>
                <GradientStop Color="#FF85C1C4" Offset="1"/>
            </LinearGradientBrush>
        </Border.Background>
        <Grid Width="Auto" Height="Auto" Background="{x:Null}">
            <Grid.ColumnDefinitions>
                <ColumnDefinition/>
                <ColumnDefinition/>
                <ColumnDefinition/>
            </Grid.ColumnDefinitions>
            <Grid.RowDefinitions>
                <RowDefinition Height="Auto"/>
                <RowDefinition Height="Auto"/>
                <RowDefinition Height="Auto"/>
                <RowDefinition />
             </Grid.RowDefinitions>
            <StackPanel Orientation="Horizontal"
               HorizontalAlignment="Center" Margin="10,10,10,10"
               VerticalAlignment="Top" Height="Auto"
               Grid.ColumnSpan="3">
                <TextBlock Margin="0,0,10,0" VerticalAlignment
                   ="Center">选择列表框模式:</TextBlock>
                <ComboBox x:Name="SelectionModeCombo"
                   DataContext="{StaticResource SelectionModeValues}"
                   IsReadOnly="True" ItemsSource="{Binding}"
                   SelectedIndex="0"/>
             </StackPanel>
             <TextBlock HorizontalAlignment="Center" Margin="0,0,0,0"
                    VerticalAlignment="Top" TextWrapping="Wrap"
                    Height="Auto" Grid.Row="1" >为设定列表框属性
                        <LineBreak/>  使用资源画刷设置选择条目的颜色
             </TextBlock>
             <TextBlock HorizontalAlignment="Center" Margin="0,0,0,0"
                 VerticalAlignment="Top" Height="Auto" Grid.Column="1"
                 Grid.Row="1" TextWrapping="Wrap">设置指示牌大小为32
                 <LineBreak/>使用滑动块</TextBlock>
             <TextBlock HorizontalAlignment="Center"
                 Margin="10,0,10,0" VerticalAlignment="Top"
                 Height="Auto" Grid.Column="2" Grid.Row="1"
                 TextWrapping="Wrap">设置指示牌的颜色<LineBreak/>图像条目
             </TextBlock>
            <YBCustomControls:ListBoxWithIndicator
```

```
                HorizontalAlignment="Stretch"
                VerticalAlignment="Stretch" Width="Auto" Height="Auto"
                Margin="10,10,10,10" FontSize="12" ClipToBounds="True"
                Background="#FFFFFFFF" BorderBrush="#FF000000"
                BorderThickness="1,1,1,1" Padding="5,5,5,5"
                Grid.Row="3" IndicatorHeightWidth="16">
                 <YBCustomControls:ListBoxWithIndicator.Resources>
                    <LinearGradientBrush x:Key="{x:Static
                      SystemColors.HighlightBrushKey}" EndPoint="1,0.5"
                      StartPoint="0,0.5">
                           <GradientStop Color="#FF3FBA00" Offset="0"/>
                           <GradientStop Color="#FFFFFFFF" Offset="1"/>
                       </LinearGradientBrush>
                       <SolidColorBrush x:Key="{x:Static
                        SystemColors.ControlBrushKey}"
                              Color="#FFCDFFC9"/>
                   </YBCustomControls:ListBoxWithIndicator.Resources>
                    <ListBox x:Name="lbDemoOne" Width="Auto"
                       Height="Auto"  DataContext="{StaticResource
                       computerCouse}" ItemsSource="{Binding}"
                       FontSize="16" SelectionMode="{Binding Path=Text,
                       ElementName=SelectionModeCombo, Mode=OneWay}"/>
              </YBCustomControls:ListBoxWithIndicator>
               <YBCustomControls:ListBoxWithIndicator
                 HorizontalAlignment="Stretch" Width="Auto"
                 Height="Auto" VerticalAlignment="Stretch"
                 Margin="10,10,10,10" FontSize="12" ClipToBounds="True"
                 Background="#FFDEFDFF" BorderBrush="#FF00A7B9"
                 BorderThickness="1,1,1,1" Padding="5,5,5,5"
                 Grid.Row="3" Grid.Column="1"
                 IndicatorHeightWidth="32" >
                 <ListBox  x:Name="lbDemoTwo"  Width="Auto"
                   Height="Auto" DataContext="{DynamicResource
                   computerCouse}" ItemsSource="{Binding}"
                   FontSize="24" SelectionMode="{Binding Path=Text,
                   ElementName=SelectionModeCombo, Mode=OneWay}"
                   ScrollViewer.CanContentScroll="False"/>
               </YBCustomControls:ListBoxWithIndicator>
               <ListBox Grid.Column="2" Grid.Row="3"
                  x:Name="lbDemoThree"  Width="Auto" Height="Auto"
                 DataContext="{DynamicResource computerCouse}"
                 ItemsSource="{Binding}" FontSize="24"
                 SelectionMode="{Binding Path=Text,
                 ElementName=SelectionModeCombo, Mode=OneWay}"
                 ScrollViewer.CanContentScroll="False"/>
            </Grid>
        </Border>
    </Grid>
</Window>
```

下面是C#后台程序：

```
using System;
```

```
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Data;
using System.Windows.Documents;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Media.Imaging;
using System.Windows.Navigation;
using System.Windows.Shapes;
namespace Yingbao.Chapter17.TestApp
{
      public partial class AppWin : Window
    {
        public AppWin()
        {
            InitializeComponent();
        }
    }
}
```

使用自定义控件ListBoxWithIndicator，我们不用写什么C#代码。这段程序的运行结果如图17-4所示。

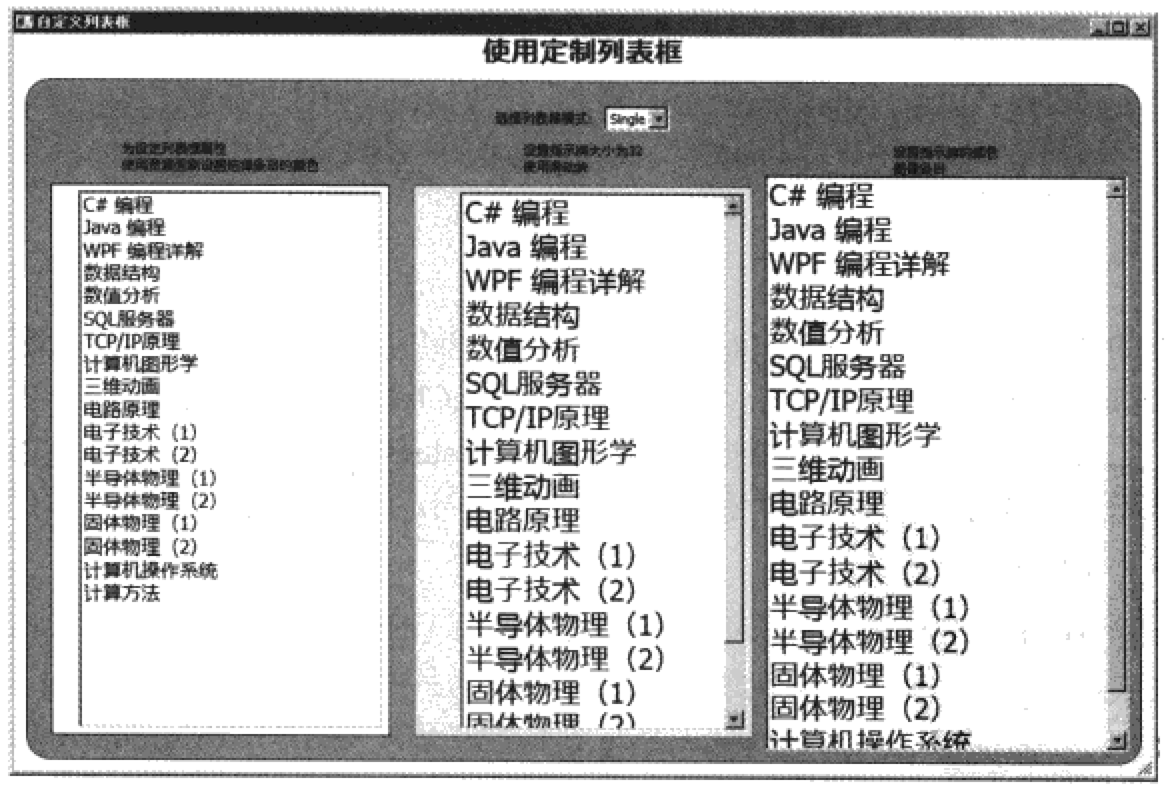


图17-4 使用自定义列表框（ListBoxWithIndicator）

17.4 创建自定义排版（Custom Panel）

第3章WPF排版讨论了WPF中的排版系统，和Win3及.NET Form对排版的支持相比，WPF要丰富得多。WPF类库本身带有DockPanel、StackPanel、WrapPanel、Grid、Canvas和UniformGrid，而且可

以随时组合这些排版类，所以，利用这些排版所产生的排版效果可以说是无限的。在你的软件里，是否真的需要开发自己的排版类？你可能要在创建排版类和利用已有的排版类间认真选择。如果你的项目里需要多次使用同一式样排版，可能要开发一个自己的排版类；如果你是WPF软件开发商，为市场上提供WPF控件，可能也需要考虑提供独特的排版类。

自定义排版类可以从Panel类或其他排版类中派生出来，但也可以从其他控件中派生出来。由于Panel类中含有相关属性Children，它是UIElementCollection类型，具有管理逻辑树和视觉树的功能，所以开发自定义排版，从Panel中派生出排版类是最方便的。

创建自定义排版类主要工作是确定其子元素在排版类上的位置和大小，由于排版类自己通常不是顶层UI元素，它也需要告诉其父元素它的大小和位置。

WPF中的排版是经由两个过程完成的，第一个过程叫做测量（Measure）；第二个过程叫做排列（Arrange）。第一个过程是父元素问子元素：我有这么大地方，你需要多大地方啊？这个过程遍历所有的子元素，从而WPF知道应该给每个元素分配多大地方。第二个过程是排列的过程，作为排版类，它需要告诉其中的子元素其位置和大小。

对于第一个过程，WPF要求自定义排版类覆盖MeasureOverride方法，其原型为：

```
protected override Size MeasureOverride( Size sizeAvailable )
{
      Size  desiredSize;
      Size availableChild; //计算出可以分配给每个子元素的大小
     foreach( UIElement child in Children )
     {
         child.Measure(availableChild);  //询问其中的子元素
      }
      .... //计算出所需的大小
      return desiredSize;
}
```

排版类中的所有子元素都在Children这个属性内，所以上面这个方法实际上遍历了所有的子元素。当调用ChildMeasure()方法时，子元素根据availableChild大小，计算出其所希望的大小并设置DesiredSize属性，自定义排版类可以根据这个值调整MeasureOverride的返回值。注意，自定义排版类一定要调用子元素的Measure函数，即使自定义排版类并不打算考虑子元素所希望的大小，否则，某些子元素可能不能正常工作。

在MessureOvverride方法里，参数availableSize可以取无穷大，意思是在大小没有限制的情况下，希望用多大的地方展现自己？一般情况下，即使是带滚动条的窗口，也是有大小限制的。无穷大只是一种极端的情形，但在WPF里合法。DesiredSize是不能为无穷大的，即DesiredSize取无穷大为非法。

对于第二个过程，WPF有求自定义排版类覆盖ArrangeOverride方法。WPF排版系统根据每个子元素希望的大小，计算出每个子元素最终该占据的位置和大小，并把这个值作为参数传递到Arrange方法中。ArrangeOverride具有下面的原型：

```
protected override Size ArrangeOverride(Size sizeFinal )
{
```

```
        Size sizeChild; //计算出其中子元素的最终大小
            foreach( UIElement child in Children )
        {
              child. Arrange( new Rect( ...) );  //通知所有的子元素，其位置和
大小。
                  ...
            }
    return sizeFinal;
}
```

在调用child. Arrange方法后，子元素设置了ActualHeight和ActualWidth属性。

17.4.1 照片浏览器

下面用一个照片浏览器的例子来说明如何创建自定义排版类。要创建一个GalleryPanel，这个排版类从Panel类中派生出来。如前所述，第一个需要移植的函数是MeasureOverride：

```
public class GalleryPanel:Panel
    {
       ...
        protected override Size MeasureOverride(Size availableSize)
        {
            Size idealSize = new Size(0, 0);
            Size size = new Size(Double.PositiveInfinity,
                 Double.PositiveInfinity);
            foreach (UIElement child in Children)
            {
                child.Measure(size);
                idealSize.Width += child.DesiredSize.Width;
                idealSize.Height = Math.Max(idealSize.Height,
                child.DesiredSize.Height);
            }
            if (double.IsInfinity(availableSize.Height) ||
               double.IsInfinity(availableSize.Width))
                return idealSize;
            else
                return availableSize;
        }
}
```

这里的逻辑很简单，我们用无穷大尺寸轮询每个子元素，然后根据每个子元素的DesiredSize计算出总的Gallery的大小，由于GalleryPanel只有一行，所以Gallery理想的宽度为每个子元素希望宽度之和；其高度为子元素的最大高度。返回值要由WPF传进来的参数availableSize确定，若availableSize为无穷大，就返回计算出的理想大小。如果WPF传进来的参数availableSize不是无穷大，显然这没法达到自定义排版类GalleryPanel的理想，那么退而求其次，就给我你所有的地方吧， 即返回availableSize。

第二个需要移植的是ArrangeOverride函数：

```
public class GalleryPanel:Panel
{
```

```
.....
protected override Size ArrangeOverride(Size finalSize)
    {
        if (this.Children == null || this.Children.Count == 0)
            return finalSize;
        ourSize = finalSize;
        totalChildWidth = 0;
        foreach (UIElement child in this.Children)
        {
            if (child.RenderTransform as TransformGroup == null)
            {
                child.RenderTransformOrigin = new Point(0, 0.5);
                TransformGroup group = new TransformGroup();
                child.RenderTransform = group;
                group.Children.Add(new ScaleTransform());
                group.Children.Add(new TranslateTransform());
            }

            child.Arrange(new Rect(0, 0, child.DesiredSize.Width,
             child.DesiredSize.Height));
            totalChildWidth += child.DesiredSize.Width;
        }
        AnimateAll();
        return finalSize;
    }
}
```

由于我们要在一行内显示所有的照片，而无论怎样照片都不可能在一行中显示，我们采用的方法是根据总照片数，计算出照片的总宽度，然后按比例缩小。所以在调用**Child.Arrage(..)**方法的时候，使用的是**child.DesiredSize**。下面是完整的程序：

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Data;
using System.Windows.Documents;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Media.Imaging;
using System.Windows.Navigation;
using System.Windows.Shapes;
using System.Windows.Media.Animation;
namespace Yingbao.Chapter17.PhotoGallery
{
    public class GalleryPanel:Panel
    {
        enum AnimateState { None, Up, Down };
        public GalleryPanel()
        {
            this.Background = Brushes.Transparent;
```

```
        this.MouseMove += new MouseEventHandler(OnMouseMove);
        this.MouseEnter += new MouseEventHandler(OnMouseEnter);
        this.MouseLeave += new MouseEventHandler(OnMouseLeave);
    }
    public double Magnification
    {
        get { return (double)GetValue(MagnificationProperty); }
        set { SetValue(MagnificationProperty, value); }
    }
    public static readonly DependencyProperty
        MagnificationProperty = DependencyProperty.Register(
        "Magnification", typeof(double), typeof(GalleryPanel),
        new UIPropertyMetadata(2d));

    public int AnimationMilliseconds
    {
        get { return (int)GetValue(
                AnimationMillisecondsProperty); }
        set { SetValue(AnimationMillisecondsProperty, value); }
    }

    public static readonly DependencyProperty
            AnimationMillisecondsProperty =
        DependencyProperty.Register("AnimationMilliseconds",
      typeof(int), typeof(GalleryPanel),new UIPropertyMetadata(125));
    public bool ScaleToFit
    {
        get { return (bool)GetValue(ScaleToFitProperty); }
        set { SetValue(ScaleToFitProperty, value); }
    }
    public static readonly DependencyProperty ScaleToFitProperty =
        DependencyProperty.Register("ScaleToFit", typeof(bool),
        typeof(GalleryPanel), new UIPropertyMetadata(true));
    private bool animating = false;
    private Size ourSize;
    private double totalChildWidth = 0;
    private bool wasMouseOver = false;
    void OnMouseMove(object sender, MouseEventArgs e)
    {
        if (!animating)  this.InvalidateArrange();
    }
    void OnMouseEnter(object sender, MouseEventArgs e)
    {
        this.InvalidateArrange();
    }
    void OnMouseLeave(object sender, MouseEventArgs e)
    {
        this.InvalidateArrange();
    }
    protected override Size MeasureOverride(Size availableSize)
    {
        Size idealSize = new Size(0, 0);
        Size size = new Size(Double.PositiveInfinity,
```

```
                Double.PositiveInfinity);
        foreach (UIElement child in Children)
        {
            child.Measure(size);
            idealSize.Width += child.DesiredSize.Width;
            idealSize.Height = Math.Max(idealSize.Height,
               child.DesiredSize.Height);
        }

        if (double.IsInfinity(availableSize.Height) ||
              double.IsInfinity(availableSize.Width))
            return idealSize;
        else
            return availableSize;
    }
    protected override Size ArrangeOverride(Size finalSize)
    {
        if (this.Children == null || this.Children.Count == 0)
            return finalSize;
        ourSize = finalSize;
        totalChildWidth = 0;
        foreach (UIElement child in this.Children)
        {
            if (child.RenderTransform as TransformGroup == null)
            {
                child.RenderTransformOrigin = new Point(0, 0.5);
                TransformGroup group = new TransformGroup();
                child.RenderTransform = group;
                group.Children.Add(new ScaleTransform());
                group.Children.Add(new TranslateTransform());
            }
            child.Arrange(new Rect(0, 0, child.DesiredSize.Width,
                  child.DesiredSize.Height));
            totalChildWidth += child.DesiredSize.Width;
        }
        AnimateAll();
        return finalSize;
    }
    void AnimateAll()
    {
        if (this.Children == null || this.Children.Count == 0)
            return;
        animating = true;
        double childWidth = ourSize.Width / this.Children.Count;
        double overallScaleFactor = ourSize.Width / totalChildWidth;
        UIElement prevChild = null;
        UIElement theChild = null;
        UIElement nextChild = null;
        double widthSoFar = 0;
        double theChildX = 0;
        double ratio = 0;
        if (this.IsMouseOver)
        {
```

```
        double x = Mouse.GetPosition(this).X;
        foreach (UIElement child in this.Children)
        {
            if (theChild == null)
                theChildX = widthSoFar;
            widthSoFar += (ScaleToFit ? childWidth :
                   child.DesiredSize.Width * overallScaleFactor);
            if (x < widthSoFar && theChild == null)
                theChild = child;
            if (theChild == null)
                prevChild = child;
            if (nextChild == null && theChild != child &&
                theChild != null)
            {
                nextChild = child;
                break;
            }
        }
        if (theChild != null)
            ratio = (x - theChildX) / (ScaleToFit ? childWidth :
           (theChild.DesiredSize.Width * overallScaleFactor));
    }
    double mag = Magnification;
    double extra = 0;
    if (theChild != null)
        extra += (mag - 1);

    if (prevChild == null)
        extra += (ratio * (mag - 1));
    else if (nextChild == null)
        extra += ((mag - 1) * (1 - ratio));
    else
        extra += (mag - 1);
    double prevScale = this.Children.Count * (1 + ((mag - 1) * (1
      - ratio))) / (this.Children.Count + extra);
    double theScale = (mag * this.Children.Count) /
      (this.Children.Count + extra);
    double nextScale = this.Children.Count * (1 + ((mag - 1) *
       ratio)) / (this.Children.Count + extra);
    double otherScale = this.Children.Count /
       (this.Children.Count + extra);
    if (!ScaleToFit && this.IsMouseOver)
    {
        double bigWidth = 0;
        double actualWidth = 0;
        if (prevChild != null)
        {
            bigWidth += prevScale * prevChild.DesiredSize.Width *
              overallScaleFactor;
            actualWidth += prevChild.DesiredSize.Width;
        }
        if (theChild != null)
        {
```

```
                bigWidth += theScale * theChild.DesiredSize.Width *
                    overallScaleFactor;
                actualWidth += theChild.DesiredSize.Width;
            }
            if (nextChild != null)
            {
                bigWidth += nextScale * nextChild.DesiredSize.Width *
                    overallScaleFactor;
                actualWidth += nextChild.DesiredSize.Width;
            }
            double w = (totalChildWidth - actualWidth) *
                         overallScaleFactor * otherScale;
            otherScale *= (ourSize.Width - bigWidth) / w;
        }
        widthSoFar = 0;
        double duration = 0;
        if (wasMouseOver != this.IsMouseOver)
            duration = AnimationMilliseconds;

        foreach (UIElement child in this.Children)
        {
            double scale = otherScale;
            if (child == prevChild)
            {
                scale = prevScale;
            }
            else if (child == theChild)
            {
                scale = theScale;
            }
            else if (child == nextChild)
            {
                scale = nextScale;
            }

            if (ScaleToFit)
            {
                scale *= childWidth / child.DesiredSize.Width;
            }
            else
            {
                scale *= overallScaleFactor;
            }
            AnimateTo(child, 0, widthSoFar, (ourSize.Height -
               child.DesiredSize.Height) / 2,
               scale, duration);
            widthSoFar += child.DesiredSize.Width * scale;
        }
        wasMouseOver = this.IsMouseOver;
    }

    private void AnimateTo(UIElement child, double r, double x,
          double y, double s, double duration)
```

```
        {
            TransformGroup group = (TransformGroup)child.RenderTransform;
            ScaleTransform scale = (ScaleTransform)group.Children[0];
            TranslateTransform trans =
                        (TranslateTransform)group.Children[1];
            if (duration == 0)
            {
                trans.BeginAnimation(TranslateTransform.XProperty, null);
                trans.BeginAnimation(TranslateTransform.YProperty, null);
                scale.BeginAnimation(ScaleTransform.ScaleXProperty, null);
                scale.BeginAnimation(ScaleTransform.ScaleYProperty, null);
                trans.X = x;
                trans.Y = y;
                scale.ScaleX = s;
                scale.ScaleY = s;
                animation_Completed(null, null);
            }
            else
            {
                trans.BeginAnimation(TranslateTransform.XProperty,
                      MakeAnimation(x, duration, animation_Completed));
                trans.BeginAnimation(TranslateTransform.YProperty,
                      MakeAnimation(y, duration));
                scale.BeginAnimation(ScaleTransform.ScaleXProperty,
                      MakeAnimation(s, duration));
                scale.BeginAnimation(ScaleTransform.ScaleYProperty,
                      MakeAnimation(s, duration));
            }
        }
        private DoubleAnimation MakeAnimation(double to, double duration)
        {
            return MakeAnimation(to, duration, null);
        }
        private DoubleAnimation MakeAnimation(double to, double duration,
                  EventHandler endEvent)
        {
            DoubleAnimation anim = new DoubleAnimation(to,
                        TimeSpan.FromMilliseconds(duration));
            anim.AccelerationRatio = 0.2;
            anim.DecelerationRatio = 0.7;
            if (endEvent != null)
                anim.Completed += endEvent;
            return anim;
        }
        void animation_Completed(object sender, EventArgs e)
        {
            animating = false;
        }
    }
}
```

为了测试GalleryPanel的效果，列举一个应用实例：

```
<Window x:Class="Yingbao.Chapter17.PhotoGallery.AppWin"
```

```
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="clr-namespace:Yingbao.Chapter17.PhotoGallery"
    Title="Window1" Height="452" Width="710">
    <Window.Background>
        <ImageBrush ImageSource="Photos\Waterloo1.jpg"/>
    </Window.Background>
    <Window.Resources>
        <local:PathConverter x:Key="PathConverter" />
        <XmlDataProvider x:Key="photoList" XPath="PhotoList/Photo">
            <x:XData>
                <PhotoList xmlns="">
                    <Photo Image="Waterloo2.jpg" Width="50"/>
                    <Photo Image="Waterloo3.jpg" Width="50"/>
                    <Photo Image="Waterloo4.jpg" Width="50"/>
                    <Photo Image="Waterloo5.jpg" Width="50"/>
                    <Photo Image="Waterloo6.jpg" Width="50"/>
                    <Photo Image="Waterloo7.jpg" Width="50"/>
                    <Photo Image="Waterloo8.jpg" Width="50"/>
                    <Photo Image="Waterloo9.jpg" Width="50"/>
                    <Photo Image="Waterloo10.jpg" Width="50"/>
                    <Photo Image="Waterloo11.jpg" Width="100"/>
                    <Photo Image="Waterloo12.jpg" Width="150"/>
                    <Photo Image="Waterloo13.jpg" Width="100"/>
                </PhotoList>
            </x:XData>
        </XmlDataProvider>
    </Window.Resources>
    <Grid>
        <Grid.RowDefinitions>
            <RowDefinition Height="*"/>
            <RowDefinition Height="2*"/>
        </Grid.RowDefinitions>
        <Border CornerRadius="10" Background="#99ffffff" Grid.Row="1">
            <ItemsControl DataContext="{Binding
                Source={StaticResource photoList}}"
                ItemsSource="{Binding}" Margin="0">
                <ItemsControl.ItemsPanel>
                    <ItemsPanelTemplate>
                      <local:GalleryPanel  Magnification="16"
                          AnimationMilliseconds="150"
                          ScaleToFit="true"/>
                    </ItemsPanelTemplate>
                </ItemsControl.ItemsPanel>
                <ItemsControl.ItemContainerStyle>
                    <Style TargetType="{x:Type ContentPresenter}">
                        <Setter Property="ContentTemplate">
                            <Setter.Value>
                                <DataTemplate>
                                    <Image Source="{Binding Converter=
                                         {StaticResource PathConverter},
                                         XPath=@Image}" Width="{Binding
                                         XPath=@Width}" Margin="5"/>
```

```
                                        </DataTemplate>
                                    </Setter.Value>
                                </Setter>
                            </Style>
                        </ItemsControl.ItemContainerStyle>
                    </ItemsControl>
                </Border>
    </Grid>
</Window>
```

在这段程序中，首先利用XmlDataProvider引入12张照片文件名。这是笔者在加拿大安大略省滑铁卢小镇上拍的照片，滑铁卢是一个人口约10万人的城镇，这里有著名的滑铁卢大学和一些高科技公司。注意，本例的照片文件放在运行程序所在目录Photos\子目录下。后台C#代码为VisualStudio产生的，我们不需要写什么逻辑：

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Data;
using System.Windows.Documents;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Media.Imaging;
using System.Windows.Navigation;
using System.Windows.Shapes;
namespace Yingbao.Chapter17.PhotoGallery
{
    public partial class AppWin : Window
    {
       public AppWin()
        {
            InitializeComponent();
        }
    }
}
```

在数据绑定中，我们使用了PathConverter， 这个类把XmlDataProvide中文件名转化为带整个路径的文件名。

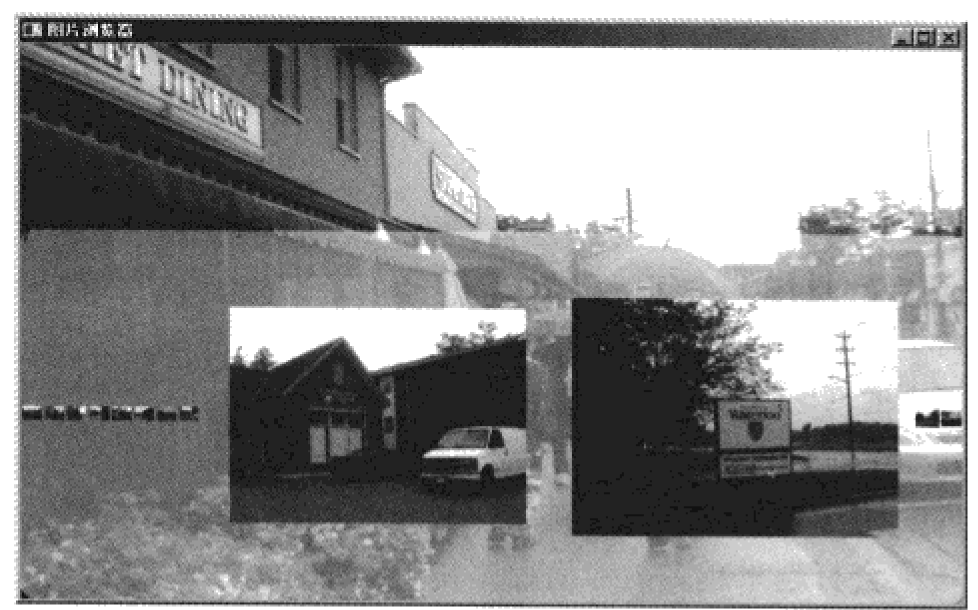
```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.IO;
using System.Reflection;
using System.Windows.Data;
namespace Yingbao.Chapter17.PhotoGallery
{
    public class PathConverter : IValueConverter
```

```
    {
        #region IValueConverter Members
        private static string path;
        public object Convert(object value, Type targetType,
             object parameter, System.Globalization.CultureInfo culture)
        {
            if (path == null)
            {
               path = ath.GetDirectoryName(Path.GetDirectoryName(
                      Path.GetDirectoryName(Assembly.
                      GetExecutingAssembly().Location))) + "\\Images";
                if (!Directory.Exists(path))
                {
                    path = Path.GetDirectoryName(Assembly.
                           GetExecutingAssembly().Location) + "\\Photos";
                    if (!Directory.Exists(path))
                        throw new FileNotFoundException("找不到照片", path);
                }
                path += "\\";
            }
            return string.Format("{0}{1}", path, (string)value);
        }
        public object ConvertBack(object value, Type targetType,
               object parameter,
               System.Globalization.CultureInfo culture)
        {
            throw new Exception("未移植！");
        }
        #endregion
    }
}
```

整个程序的运行结果如图17-5（a）和图17-5（b）所示。

（a）图中的一行多个照片为使用GalleryPanel的结果

图17-5 照片浏览器的运行结果（Waterloo Town——李应保摄）



（b）当鼠标拖过GalleryPanel中的照片上时，放大该照片，并按一定比例放大其前后照片

图17-5 照片浏览器的运行结果（Waterloo Town——李应保摄）（续）

17.5 本章小结

本章从开发用户控件开始，讨论了用户控件和自定义控件的区别，进而提供了自定义控件的实例。WPF中的排版系统非常丰富，通常情况下，不必开发自定义排版，在某些特殊情况下，需要提供特殊的排版功能，这时就需要开发自定义排版类。自定义排版类一般从Panel中派生出来，需要做的是覆盖两个基类方法：一个是MeasureOverride，另一个是ArrangeOverride。

第18章 综合应用

本章是本书的最后一章，将讨论一个实用项目。我发现很多计算机书籍都注重讨论技术细节，而如何组织一个完整的应用项目讨论得很少，读者读完一本书后，如何在实际工程中应用书中所讨论的技术，往往需要付出较大的努力。而且，笔者认为对于计算机软件技术，如果你不能用到实际的工程中，那么你对于某些概念的理解也没法深入。

Microsoft办公软件Office 2007一改常规视窗应用软件中使用菜单工具条的做法，推出了Ribbon界面。笔者在GE工作期间，开发了基于DevExpress Ribbon控件的图形界面系统，AutoCAD 2009 业已转向Ribbon界面，所以将来会有越来越多的应用程序使用Ribbon。本章我将用WPF创建Ribbon控件。Microsoft提供了开发Ribbon的技术指南，并提供了类库（http://msdn.microsoft.com/en-us/library/aa338202.aspx，http://msdn.microsoft.com/en-us/office/aa973809.aspx），感兴趣的读者可以参考。本章的重点在于如何应用WPF技术组织一个完整的项目，很多实现的具体细节留给读者自己体会（否则本章的篇幅将过长）。笔者在写本书的同时，微软业已宣布，.NET Framework 4.0将会包括Ribbon控件。

18.1 Ribbon界面概览

图18-1示出了Word 2007中的主界面。Ribbon界面由下面几个部分组成：

- 主菜单 也叫应用程序菜单，位于窗口的左上角，通常在这里显示公司的图标。当用户单击该图标时，应用程序弹出一个菜单（如图18-2所示），该菜单中通常放入应用程序最常用的功能，如Word中对文档的操作，右边则放入最近访问的文档。提供用户快速访问这些文档的功能。
- 快速访问工具条（Quick Access Tool Bar） 把最常用的功能放在快速访问工具条中。快速访问工具条可以由用户根据自己的需要随时加入或去除，你可以在Ribbon子控件上右击鼠标，选择“Custom Quick Access Tool Bar”来定制自己的快速访问工具条，也可以选择“Add to Quick access Tool Bar”来把子控件直接加入到快速访问工具条中。
- Ribbon Tab Ribbon 主体由三部分组成，即RibbonTabItem、Ribbon Group和控件。Ribbon Tab位于应用程序主窗口的下面，如图18-1中的“Home”、“Insert”、“Page Layout”等都是Ribbon Tab。Tab为应用程序的功能提供第一个大块的分类。
- Ribbon Group Ribbon Group为应用程序功能提供了二级分类，一般把相近的 功能放在同一个Group中，这样方便用户查找，如图18-1中的Clipboard和Font等。
- Ribbon控件 这是对应用程序功能的最后一层分类，其中可以含有各种控件，如按钮、组合框、列表框等。如图18-1中的“Paste”为按钮，字体为组合框等。

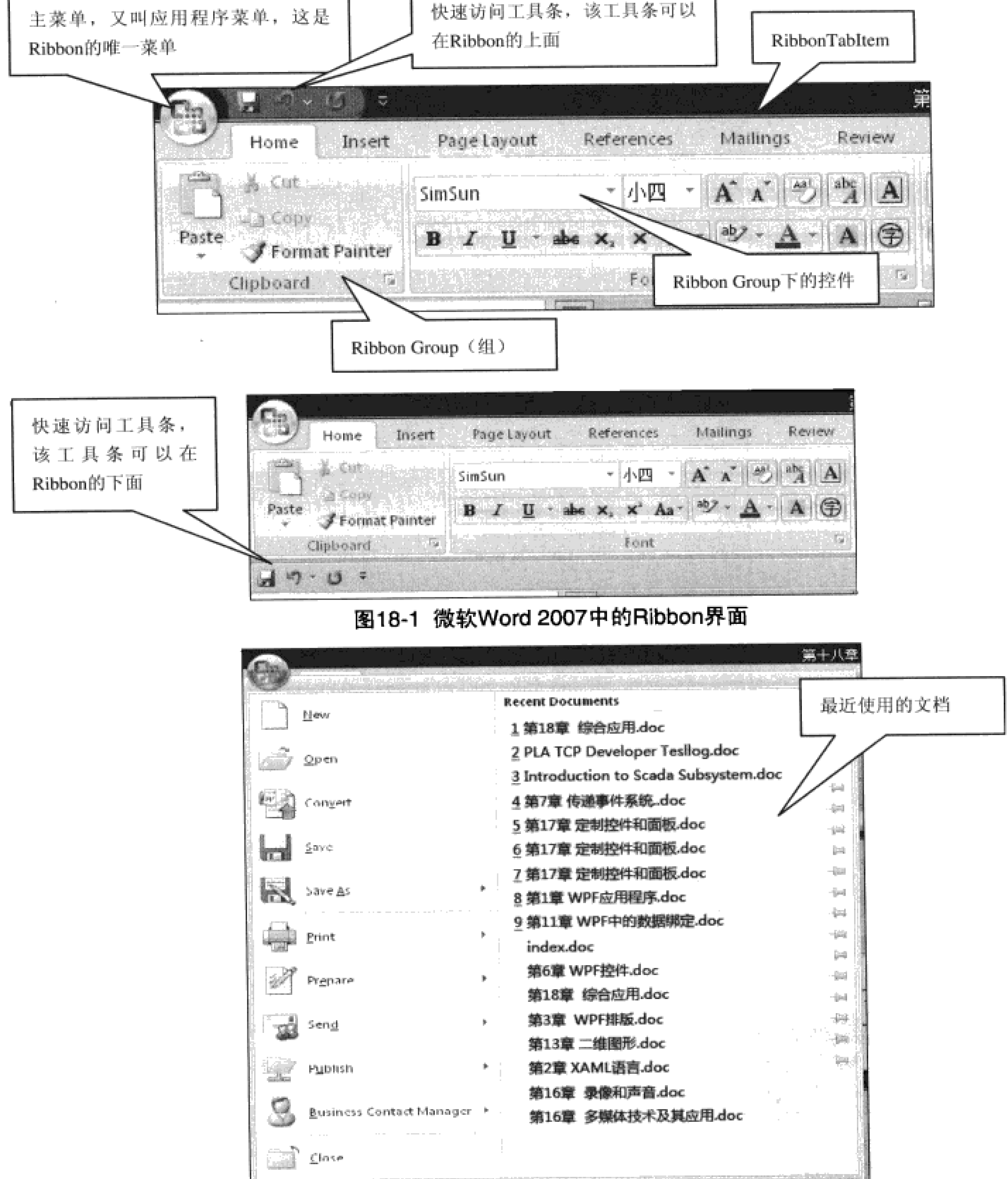


图18-1 微软Word 2007中的Ribbon界面

图18-2 Word中Ribbon界面的主菜单

18.2 项目的组织

作为练习，我们要用WPF来开发Ribbon界面。首先在YingbaoWPFExa-mple solution中加入Yingbao.Chapter18文件夹，在该文件夹中，加入两个项目。一个是支持Ribbon功能的WPF类作为一个类库，命名为YBRibbonLib；另一个是用于测试该类库的应用程序RibbonTest，如图18-3所示。

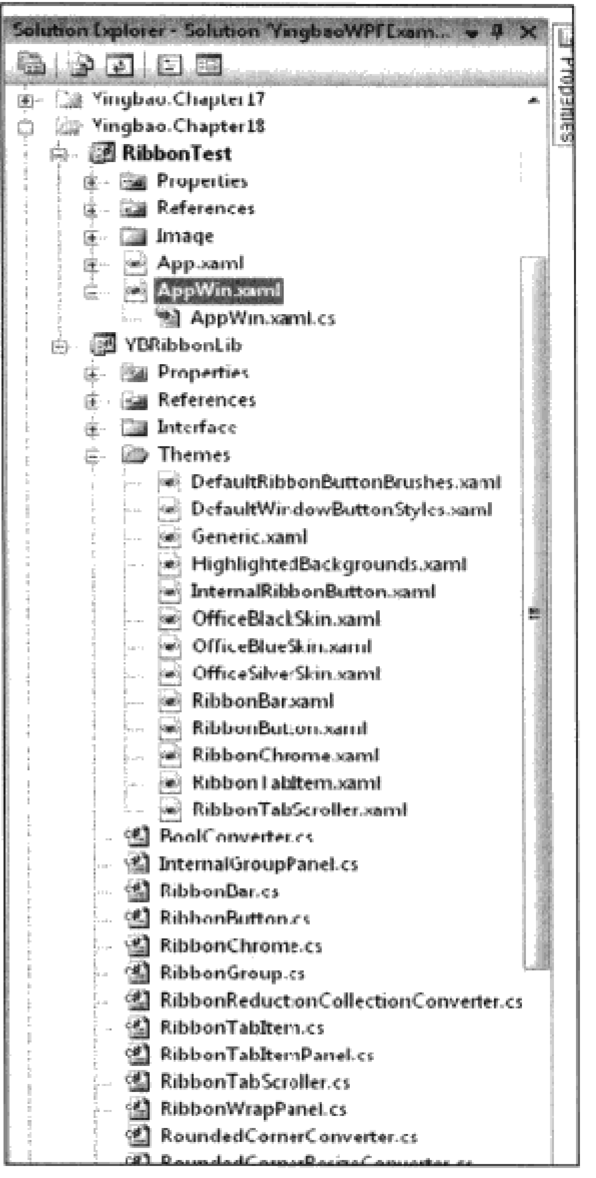


图18-3 Visual Studio中的YBRibbonLib

选择YBRibbonLib项目，右击鼠标，选择“Property”，就可以看到Visual Studio显示图18-4所示的对话框。我们需要把Default namespace改为“Yingbao. Chapter18.YBRibbonLib”，这样做的好处是以后在该项目中加入任何类，Visual Studio会自动把该类放入同一命名空间中。如你在该项目下加入类A，Visual studio会产生下面的代码：

```
namespace Yingbao.Chapter18.RibbonLib
{
    public clas A
    {
        public A()
        {
              .....
        }
    }
}
```

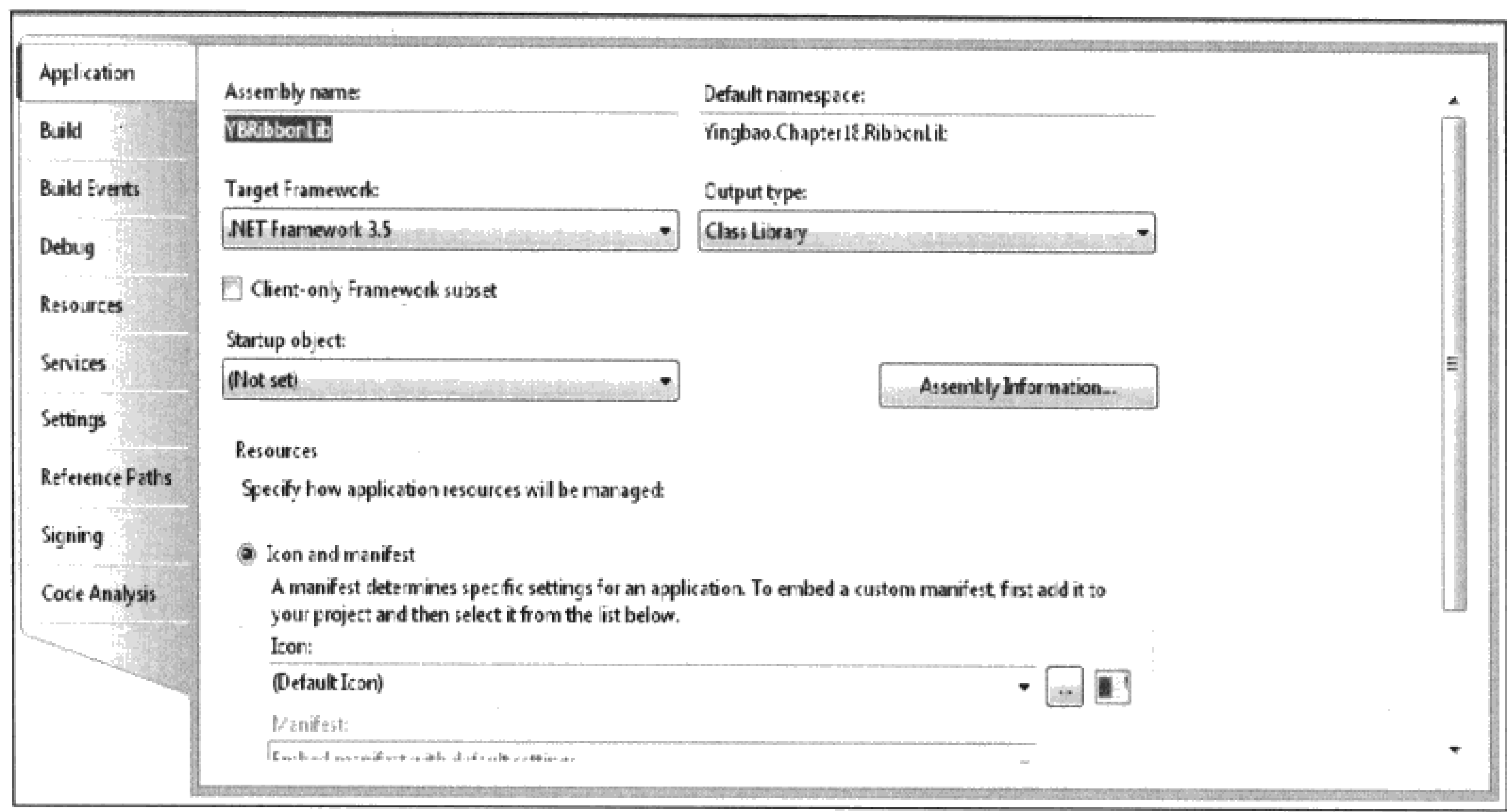


图18-4 YBRibbonLib项目的设置

注意：需要把YBRibbonLib的“Output type”设置为“Class Library”，从而这个Assembly产生的是一个DLL而不是Exe。Target Framework为.NET Frame work 3.5.

18.3 管理Generic.XAML文件

我们需要开发一系列自定义控件，从第17章我们知道，当在项目中加入自定义控件时，Visual Studio自动在项目中创建一个“Themes”文件夹，并在该文件夹里生成一个generic.xaml 文件， 该文件具有下面的格式：

```
<ResourceDictionary
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="clr-namespace:Yingbao.Chapter18.RibbonLib">
     <Style TargetType="{x:Type local:RibbonButton}">
        ....
    </Style>

    <Style TargetType="{x:Type local: RibbonGroup}">
        ....
    </Style>
     ....

</ResourceDictionary>
```

每增加一个自定义控件，Visual Studio会在 ResourceDictionary下面加入一节<Style Target Type=....> XAML，当自定义控件多时，管理起来很不方便，为此，我们采用ResourceDictionary融合技术，把每个控件的Style放到独立的文件中，同时把这些文件引入到Generic.Xaml里来：

```
<ResourceDictionary

    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="clr-namespace:Yingbao.Chapter18.RibbonLib">
    <ResourceDictionary.MergedDictionaries>
        <ResourceDictionary
           Source="pack://application:,,,/YBRibbonLib;
           Component/Themes/RibbonChrome.xaml"/>
        <ResourceDictionary
           Source="pack://application:,,,/YBRibbonLib;
           Component/Themes/RibbonTabItem.xaml"/>
        <ResourceDictionary
           Source="pack://application:,,,/YBRibbonLib;
           Component/Themes/RibbonBar.xaml"/>
        <ResourceDictionary
           Source="pack://application:,,,/YBRibbonLib;
           Component/Themes/RibbonButton.xaml"/>
       ...
    </ResourceDictionary.MergedDictionaries>
</ResourceDictionary>
```

然后，就可以在控件自己的XAML中写有关该控件的模板及风格。这样做的好处是，当我们需要对控件的XAML进行修改时，就不必担心由于改错了地方而影响其他控件。

18.4 开发自定义控件

笔者多年来的经验是阅读别人的代码，可以看到某个对象的当前状态，即现在这个“样子”。至于为什么是这个“样子”，代码本身并不能告诉我们有用的信息。作为一名优秀的程序员，不仅仅要读懂别人的代码，而且要搞清楚别人在写该代码的时候是怎么想的，是为了满足什么样的要求，等等。这些东西往往是没有文档的，需要自己体会。这一点有点像阅读文学作品，为了理解某句诗歌，我们需要阅读诗人写的相关札记或评论。下面将讨论在支持Ribbon功能的RibbonLib中，为什么要开发自定义控件，以及如何调试这些控件。

18.4.1 自定义控件间的关系

图18-5示出了本章要开发的自定义控件类间的关系。RibbonWindow为自定义窗口，其中含有唯一的UI元素RibbonBar。RibbonBar要管理快速访问工具条（RibbonQAToolBar）、应用程序菜单和RibbonTabItem。RibbonTabItem管理RibbonGroup，最后RibbonGroup里包含各种Ribbon控件。

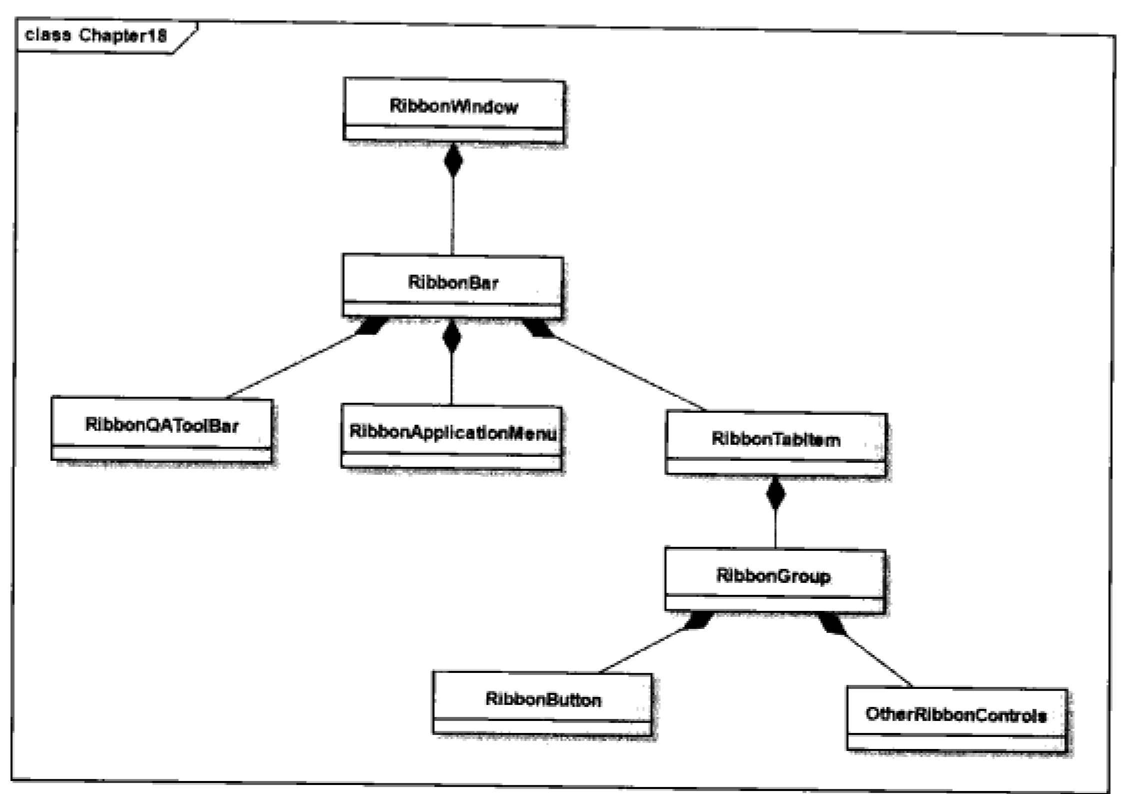
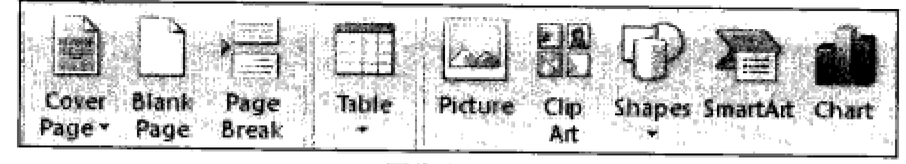


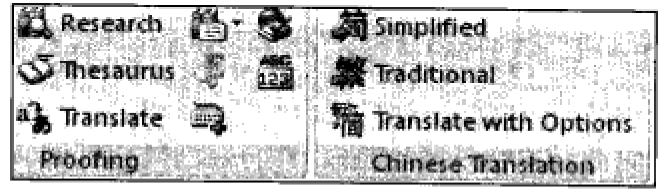
图18-5 自定义控件间的关系

18.4.2 Ribbon按钮

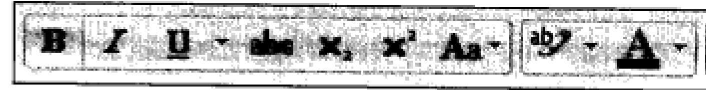
让我们从Ribbon按钮开始。通常Ribbon按钮有图像和文字两部分，或者光有图像，没有文字。典型的Ribbon按钮如图18-6（a）、图18-6（b）和图18-6（c）所示：



（a）图像在文字的上方



（b）图像在文字的左边



（c）没有文字的Ribbon按钮

图18-6 典型的Ribbon按钮

仔细研究上面的Ribbon按钮，还可以发现：

- 当按钮比较大时，文字位于图像的下面，显示两行文字；
- 当按钮比较小时，文字位于图像的右边；
- 当没有足够地方显示按钮时，按钮只显示小的图标而没有文字。

显然，WPF本身所带的按钮不能满足上面的这些要求。为此需要自定义按钮。

对于小按钮，可以在DockPanel中加入Image和ContentPresenter表示：

```
<DockPanel >
   <Image x:Name="image" Source="{TemplateBinding SmallImage}"
      Height="16" Width="16" DockPanel.Dock="Left"/>
```

```
   <ContentPresenter x:Name="content"
      Content="{TemplateBinding Content}"
      Visibility="Visible"  />
 </DockPanel>
```

Content通常是文字。

若没有足够的地方，需要隐藏 content，为此我们在ControlTemplate中加入触发器：

```
<ControlTemplate.Triggers>
      ....
  <Trigger Property="RibbonLib:RibbonBar.Size" Value="Small">
     <Setter Property="Visibility" Value="Collapsed"
             TargetName="content"/>
     <Setter Property="Width" Value="0" TargetName="content"/>
  </Trigger>
        .....
 </ControlTemplate.Triggers>
```

RibbonBar.Size相关属性将在RibbonBar中详述。

对于大按钮，需要创建以下结构：

```
<Grid Height="72" Margin="0,-2,0,-2">
     <Grid.RowDefinitions>
            <RowDefinition Height="44"/>
            <RowDefinition Height="4"/>
            <RowDefinition Height="12"/>
            <RowDefinition Height="12"/>
     </Grid.RowDefinitions>
     <Image Source="{TemplateBinding LargeImage}"  Grid.Row="0"
      Stretch="{Binding
       Path=(RibbonLib:RibbonButton.ImageStretch) ... />
     <TextBlock x:Name="content" Grid.Row="2" .../>
     <TextBlock x:Name="content2" Grid.Row="3" .../>
 </Grid>
```

这是一个4行的Grid，第一行放入按钮图像，第二行留出4个图素空格，第三行和第四行为两个文字行。

下面是完整XAML：

```
<ResourceDictionary
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:mwt="clr-namespace:Microsoft.Windows.Themes;assembly=
             PresentationFramework.Aero"
  xmlns:RibbonLib="clr-namespace:Yingbao.Chapter18.RibbonLib">
  <RibbonLib:TwoLineConverter x:Key="twoLineConverter"/>
  <RibbonLib:TwoLineTextConverter x:Key="twoLineTextConverter"/>
  <RibbonLib:RoundedCornerConverter
            x:Key="roundedCornerConverter"/>
  <ControlTemplate x:Key="SmallRibbonButtonControlTemplate"
            TargetType="{x:Type RibbonLib:RibbonButton}">
     <RibbonLib:RibbonChrome x:Name="chrome"
```

```
    SnapsToDevicePixels="True"
    RenderEnabled="{TemplateBinding IsEnabled}"
    CornerRadius="{TemplateBinding CornerRadius}"
    BorderBrush="{TemplateBinding BorderBrush}"
    Background="{TemplateBinding Background}"
    RenderFlat="{TemplateBinding IsFlat}"
    RenderMouseOver="{Binding IsMouseOver,
    RelativeSource={RelativeSource Self}}"
    HorizontalContentAlignment="{TemplateBinding
         HorizontalContentAlignment}"
    RenderPressed="{TemplateBinding IsPressed}">
    <RibbonLib:RibbonChrome.Content>
      <DockPanel  Margin="2,0,2,0" x:Name="dock1"
           SnapsToDevicePixels="True"
           HorizontalAlignment="Stretch">
           <Image Margin="0" x:Name="image"
              Source="{TemplateBinding SmallImage}"
              Height="16" Width="16"
              VerticalAlignment="Center"
              DockPanel.Dock="Left"
              SnapsToDevicePixels="True" ClipToBounds="True"
              Stretch="{Binding
              Path=(RibbonLib:RibbonButton.ImageStretch),
              RelativeSource={RelativeSource
              TemplatedParent}}"/>
           <ContentPresenter x:Name="content"
              Content="{TemplateBinding Content}"
              VerticalAlignment="Center" Margin="2,0,0,0"
              Visibility="Visible"/>
       </DockPanel>
      </RibbonLib:RibbonChrome.Content>
  </RibbonLib:RibbonChrome>
  <ControlTemplate.Triggers>
     <Trigger Property="SmallImage" Value="{x:Null}">
        <Setter Property="Visibility" Value="Collapsed"
               TargetName="image"/>
        <Setter Property="Margin" Value="0"
               TargetName="content"/>
        <Setter Property="HorizontalAlignment" Value="Center"
               TargetName="content"/>
      </Trigger>
      <Trigger Property="RibbonLib:RibbonBar.Size"
              Value="Small">
        <Setter Property="Visibility" Value="Collapsed"
              TargetName="content"/>
        <Setter Property="Width" Value="0"
              TargetName="content"/>
      </Trigger>
      <Trigger Property="RibbonLib:RibbonBar.Size"
              Value="Minimized">
        <Setter Property="Visibility" Value="Collapsed" />
      </Trigger>
  </ControlTemplate.Triggers>
```

```
</ControlTemplate>

<ControlTemplate x:Key="LargeRibbonButtonControlTemplate"
   TargetType="{x:Type RibbonLib:RibbonButton}">
   <RibbonLib:RibbonChrome x:Name="chrome" Height="72"
    RenderEnabled="{TemplateBinding IsEnabled}"
    HorizontalContentAlignment="Center"
    VerticalContentAlignment="Top"
    BorderBrush="{TemplateBinding BorderBrush}"
    CornerRadius="{TemplateBinding CornerRadius}"
    RenderMouseOver="{Binding IsMouseOver,
    RelativeSource={RelativeSource Self}}"
    RenderPressed="{TemplateBinding IsPressed}">
    <RibbonLib:RibbonChrome.Content>
      <Grid Height="72" Margin="0,-2,0,-2">
      <Grid.RowDefinitions>
          <RowDefinition Height="44"/>
          <RowDefinition Height="4"/>
          <RowDefinition Height="12"/>
          <RowDefinition Height="12"/>
      </Grid.RowDefinitions>
      <Image Source="{TemplateBinding LargeImage}"
           VerticalAlignment="Center"
           Margin="4,0,0,0" Grid.Row="0"
           SnapsToDevicePixels="True" Width="32" Height="32"
           Stretch="{Binding
           Path=(RibbonLib:RibbonButton.ImageStretch),
           RelativeSource={RelativeSource
           TemplatedParent}}"    />
      <TextBlock x:Name="content" Grid.Row="2"
           Text="{Binding Content, VerticalAlignment="Bottom"
           RelativeSource={RelativeSource TemplatedParent},
           Converter={StaticResource twoLineTextConverter},
           ConverterParameter=1}"
           HorizontalAlignment="Center" />
      <TextBlock x:Name="content2" Grid.Row="3"
           Text="{Binding Content, VerticalAlignment="Bottom"
           HorizontalAlignment="Center"
           RelativeSource={RelativeSource TemplatedParent},
           Converter={StaticResource twoLineTextConverter},
           ConverterParameter=2}"/>
            </Grid>
        </RibbonLib:RibbonChrome.Content>
    </RibbonLib:RibbonChrome>
 <ControlTemplate.Triggers>
   <Trigger Property="RibbonLib:RibbonBar.Size" Value="Small">
     <Setter Property="Visibility" Value="Collapsed"
          TargetName="content"/>
   </Trigger>
   <Trigger Property="RibbonLib:RibbonBar.Size"
         Value="Minimized">
     <Setter Property="Visibility" Value="Collapsed" />
   </Trigger>
```

```
    </ControlTemplate.Triggers>
  </ControlTemplate>

   <Style TargetType="{x:Type RibbonLib:RibbonButton}">
      <Setter Property="HorizontalContentAlignment" Value="Left"/>
      <Setter Property="BorderBrush" Value="{DynamicResource
          {ComponentResourceKey RibbonLib:Skins,
         HighlightedRibbonBorderBrush}}"/>
     <Setter Property="Background" Value="{DynamicResource
          {ComponentResourceKey   RibbonLib:Skins,
             RibbonButtonGroupBgBrush}}"/>
     <Setter Property="Template" Value="{StaticResource
                    SmallRibbonButtonControlTemplate}"/>

      <Style.Triggers>
        <Trigger Property="RibbonLib:RibbonBar.Size" Value="Large">
             <Setter Property="Template" Value="{StaticResource
                      LargeRibbonButtonControlTemplate}"/>
        </Trigger>
     </Style.Triggers>
   </Style>
</ResourceDictionary>
```

在后台C#中，需要定义LargeImage、SmallImage、IsFlat等相关属性：

```
namespace Yingbao.Chapter18.RibbonLib
{
    [ContentProperty("Content")]
    public class RibbonButton : Button, IRibbonButton
    {
        static RibbonButton()
        {
            DefaultStyleKeyProperty.OverrideMetadata(
             typeof(RibbonButton), new
               FrameworkPropertyMetadata(
               typeof(RibbonButton)));
        }

        public CornerRadius CornerRadius
        {
            get { return
                 (CornerRadius)GetValue(CornerRadiusProperty); }
            set { SetValue(CornerRadiusProperty, value); }
        }

        public static readonly DependencyProperty
         CornerRadiusProperty =
            DependencyProperty.Register("CornerRadius",
            typeof(CornerRadius), typeof(RibbonButton),
            new UIPropertyMetadata(new CornerRadius(3)));

        public ImageSource LargeImage
        {
```

```
        get {
         return(ImageSource)GetValue(LargeImageProperty); }
        set { SetValue(LargeImageProperty, value); }
    }

    public static readonly DependencyProperty
     LargeImageProperty =
         DependencyProperty.Register("LargeImage",
         typeof(ImageSource), typeof(RibbonButton),
         new FrameworkPropertyMetadata(null));
    public ImageSource SmallImage
    {
        get {
           return (ImageSource)GetValue(SmallImageProperty); }
        set { SetValue(SmallImageProperty, value); }
    }

   public static readonly DependencyProperty
      SmallImageProperty =
        DependencyProperty.Register("SmallImage",
        typeof(ImageSource), typeof(RibbonButton),
        new FrameworkPropertyMetadata(null));

    public bool IsFlat
    {
        get { return (bool)GetValue(IsFlatProperty); }
        set { SetValue(IsFlatProperty, value); }
    }

   public static readonly DependencyProperty IsFlatProperty =
        DependencyProperty.Register("IsFlat", typeof(bool),
        typeof(RibbonButton), new
        UIPropertyMetadata(true));

    public static Stretch GetImageStretch(DependencyObject obj)
    {
        return (Stretch)obj.GetValue(ImageStretchProperty);
    }

    public static void SetImageStretch(DependencyObject obj,
          Stretch value)
    {
        obj.SetValue(ImageStretchProperty, value);
    }

    public static readonly DependencyProperty
       ImageStretchProperty =
       DependencyProperty.RegisterAttached("ImageStretch",
       typeof(Stretch), typeof(RibbonButton), new
       UIPropertyMetadata(Stretch.Uniform));
  }
}
```

在写完Ribbon按钮后，我们应该对其进行简单测试，下面是测试程序：

```
<Window x:Class="Yingbao.Chapter18.RibbonTest.AppWin"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:RibbonLib=
   "clr-namespace:Yingbao.Chapter18.RibbonLib;assembly=YBRibbonLib"
    Title="Ribbon测试程序" Height="300" Width="300">
    <Grid>
        <Grid.RowDefinitions>
            <RowDefinition/>
            <RowDefinition/>
            <RowDefinition/>
        </Grid.RowDefinitions>
        <Grid.ColumnDefinitions>
            <ColumnDefinition />
            <ColumnDefinition />
            <ColumnDefinition />
            <ColumnDefinition />
        </Grid.ColumnDefinitions>
     <RibbonLib:RibbonButton Grid.Row="1" Grid.Column="0"
      IsFlat="False"  CornerRadius="10,10,10,10"   Content="Test"
      SmallImage ="Image/cut16.png"  LargeImage="Image/cut32.png"/>
    </Grid>
</Window>
```

笔者认为写好一个自定义控件就应该立即测试，而不必等到写完整个Ribbon再测试所有的控件，这样我们可以随时修改设计。上面这段XAML的运行结果如图18-7所示。

图18-7 测试Ribbon按钮

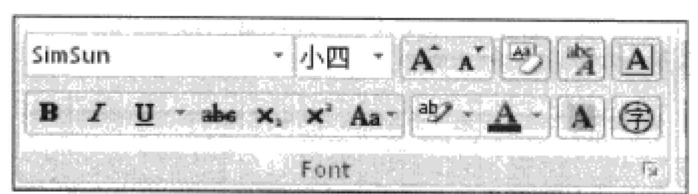
18.4.3 Ribbon分组（Group）

Ribbon分组中可以放入Ribbon按钮或其他Ribbon控件（如组合框等），典型的Ribbon分组如图18-8（a）、图18-8（b）和图18-8（c）所示。Ribbon分组控件在正常情况下，显示其中所有的控件，它由三部分组成：第一部分为子元素提供宿主，第二部分为分组标题，第三部分为显示在右下角的小箭头，单击该小箭头，可以弹出和该分组相应的对话框（如图18-9所示）。

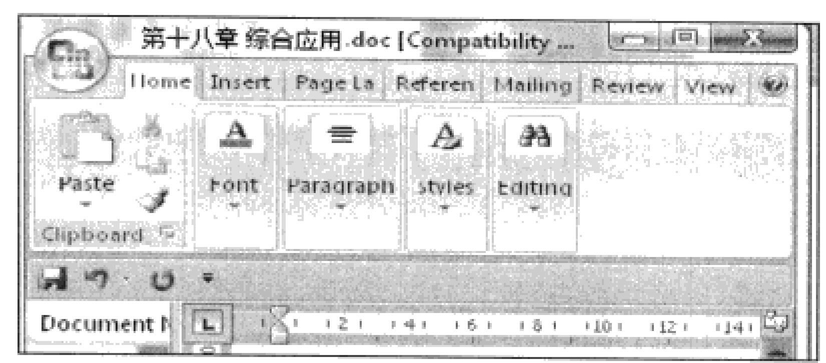
当我们把Word窗口缩小，每个Ribbon分组会缩小成一个小按钮，并在按钮的下面显示一个向下的箭头（如图18-8（b）所示）。可以看出，在正常情况下这个小按钮是不显示的。

当我们单击图18-8（b）中的“Font”按钮时，Word 2007会弹出一个窗口，其中的内容和图18-8（a）

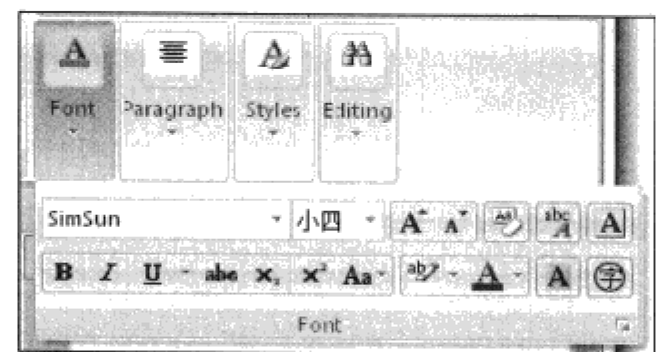
完全相同。



（a）Word 2007中字体分组



（b）当窗口缩小时，字体等分组只显示图像，下面带一个向下的小箭头



（c）单击Font，在下面显示一个弹出窗口，其内容和（a）图中完全相同

图18-8 Ribbon分组

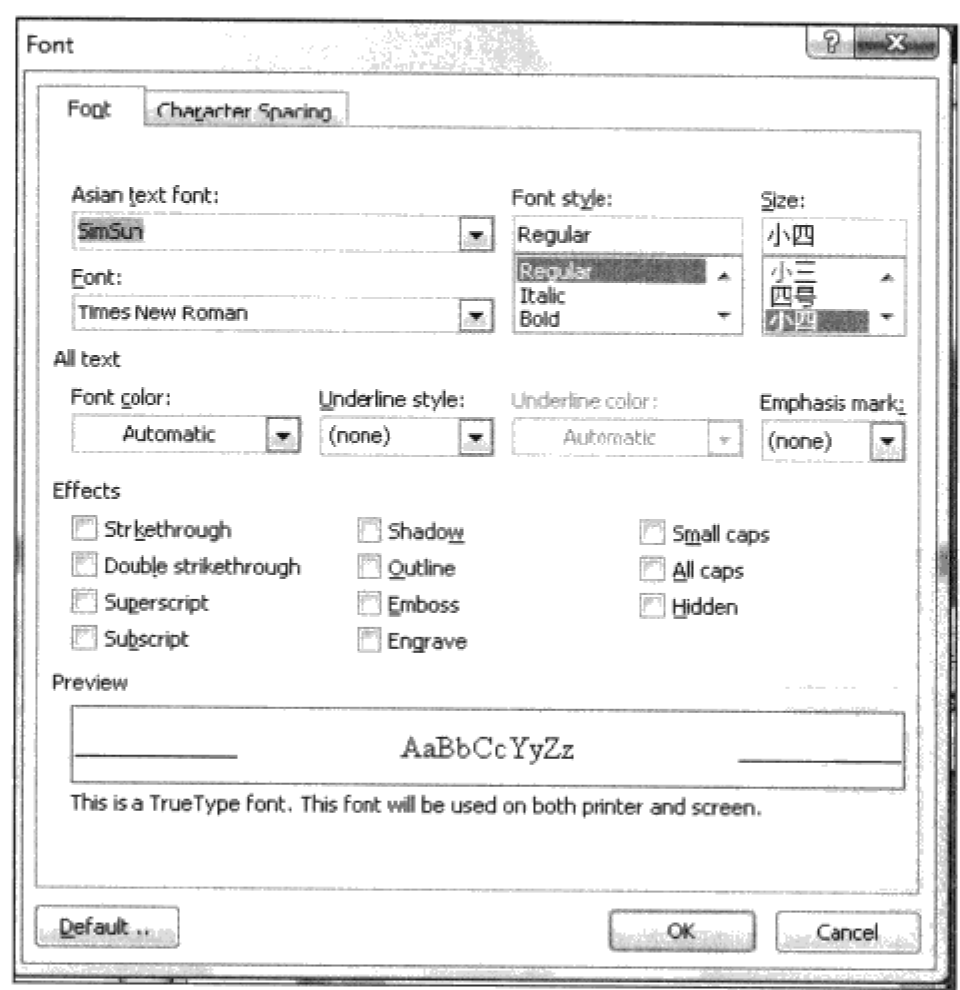


图18-9 单击图18-8（a）中右下角小箭头，Word 2007弹出一个对话框

从上面的分析，可以看到我们应该开发自定义Ribbon分组控件RibbonGroup。其中含有三大部分。

- 第一部分：为子元素提供宿主。

```
<StackPanel >
    <Border  x:Name="PART_ItemsPanelHost"/>  <!--在其中显示子元素-->
       <DockPanel >
         <Button Width="15" Height="14" DockPanel.Dock="Right" />
              <!--右下角按钮-->
         <TextBlock x:Name="groupTitle" />  <!-- 显示分组标题 -->
    </DockPanel>
 </StackPanel>
```

其中PART_ItemsPanelHost为子元素的宿主。

- 第二部分：构建窗口缩小时的小按钮，它是一个ToggleButton，其中含有RibbonChrome和向下的小箭头（symbol），注意ToggleButton的Visibility设为“Hidden”，即在通常情况下该按钮不可见：

```
<ToggleButton x:Name="PART_Button"  Visibility="Hidden">
   <StackPanel >
     <RibbonLib:RibbonChrome>
       <Image Source="{TemplateBinding Image}" />  显示按钮图像
     </RibbonLib:RibbonChrome>
     <Image x:Name="symbol" />     显示向下箭头
   </StackPanel>
</ToggleButton>
```

- 第三部分：构建弹出窗口。

弹出窗口需要显示分组中的所有子元素，Border为其在视觉树上占一个地方，在C#代码中，要动态加入子元素。

```
<Popup x:Name="PART_Popup" >
 <Border>
   <RibbonLib:RibbonChrome x:Name="popupchrome">
     <StackPanel>
        <Border x:Name="PART_PopupItemsPanelHost"/>显示子控件
        <DockPanel >
            <Button DockPanel.Dock="Right"/> 显示右下角按钮
            <ContentControl />  显示分组标题
        </DockPanel>
     </StackPanel>
    </RibbonLib:RibbonChrome>
  </Border>
</Popup>
```

我们知道，Popup在界面上显示是由IsOpen相关属性控制的，我们需要在C#代码中加入一个相关属性IsDropDownOpen作为过渡，把ToggleButton的IsChecked相关属性绑定到IsDropDownOpen上，从而当用户单击ToggleButton时，自动显示弹出窗口。

```
<ResourceDictionary
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:RibbonLib="clr-namespace:Yingbao.Chapter18.RibbonLib"
```

```
xmlns:mwt="clr-
namespace:Microsoft.Windows.Themes;assembly=PresentationFramework.A
ero"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
    <ResourceDictionary.MergedDictionaries>
     <ResourceDictionary
        Source="pack://application:,,,/YBRibbonLib;Component/
                  Themes/HighlightedBackgrounds.xaml"/>
      <ResourceDictionary
          Source="pack://application:,,,/YBRibbonLib;Component/
                Themes/RibbonSeparator.xaml"/>
    </ResourceDictionary.MergedDictionaries>
    <RibbonLib:TwoLineConverter x:Key="twoLineConverter"/>
    <ControlTemplate TargetType="{x:Type RibbonLib:RibbonGroup}"
                              x:Key="RibbonGroupTemplate">
     <Grid Focusable="False">
        <ToggleButton x:Name="PART_Button"
             SnapsToDevicePixels="True" Focusable="True"
             IsChecked="{Binding IsDropDownOpen,
             RelativeSource={RelativeSource
             TemplatedParent},Mode=TwoWay}"
             ClickMode="Press" Visibility="Hidden"
             BorderBrush="{TemplateBinding BorderBrush}"
             Foreground="{TemplateBinding Foreground}"
             Width="{TemplateBinding Width}"
             Height="{TemplateBinding Height}">
            <ToggleButton.Template>
              <ControlTemplate TargetType="{x:Type ToggleButton}" >
                <RibbonLib:RibbonChrome  x:Name="chrome3"
                    SnapsToDevicePixels="True"
                    Focusable="False"
                    BorderBrush="{TemplateBinding BorderBrush}"
                    RenderPressed="{Binding IsChecked,
                    ElementName=PART_Button}"
                    RenderMouseOver="{TemplateBinding IsMouseOver}"
                    Style="{DynamicResource {ComponentResourceKey
                    RibbonLib:Skins,
                    CollapsedRibbonGroupChromeStyle}}">
                <ContentPresenter  Content="{TemplateBinding
                    ToggleButton.Content}" VerticalAlignment="Top"
                    HorizontalAlignment="Center"
                    TextBlock.TextAlignment="Center" Margin="4"/>
                 </RibbonLib:RibbonChrome>
                    <ControlTemplate.Triggers>
                      <Trigger Property="IsPressed" Value="True">
                          <Setter Property="RenderPressed"
                            Value="True" TargetName="chrome3"/>
                      </Trigger>
                    </ControlTemplate.Triggers>
                </ControlTemplate>
            </ToggleButton.Template>
            <StackPanel Height="84" SnapsToDevicePixels="True"
                Focusable="False">
                <RibbonLib:RibbonChrome RenderFlat="False"
                     SnapsToDevicePixels="True"
```

```
            Focusable="False"  Width="31"
            Height="31"  CornerRadius="4"
            BorderBrush="{TemplateBinding BorderBrush}"
            Background="{DynamicResource
            {ComponentResourceKey RibbonLib:Skins,
            SmallIconBgBrush}}"
            Margin="5,2,5,4"  HorizontalAlignment="Center"
            VerticalAlignment="Top">
         <Image Source="{TemplateBinding Image}"
            SnapsToDevicePixels="True"
            ClipToBounds="True" Focusable="False"
         RenderOptions.BitmapScalingMode="{TemplateBinding
         RenderOptions.BitmapScalingMode}"
         Width="16" Height="16"  Stretch="Fill"
         Margin="4,4,0,4" VerticalAlignment="Top"
         HorizontalAlignment="Center" />
       </RibbonLib:RibbonChrome>
       <ContentPresenter Focusable="False"
          Content="{Binding Title,
          RelativeSource={RelativeSource TemplatedParent},
          Converter={StaticResource twoLineConverter}}"
          x:Name="title" Margin="2,0,2,0"
          VerticalAlignment="Center"
          HorizontalAlignment="Center" MaxWidth="140" />
       <Image x:Name="symbol" Source="{DynamicResource
          {ComponentResourceKey
          RibbonLib:Skins, DownArrowImage}}"
          Margin="1,4,0,0" Stretch="None"
          SnapsToDevicePixels="True"/>
       </StackPanel>
   </ToggleButton>
   <RibbonLib:RibbonChrome x:Name="chrome1"
       SnapsToDevicePixels="True" Focusable="False"
       Style="{DynamicResource {ComponentResourceKey
       RibbonLib:Skins, ExpandedRibbonGroupChromeStyle}}"
       Width="{TemplateBinding Width}"
       Height="{TemplateBinding Height}"
       BorderBrush="{TemplateBinding BorderBrush}"
       RenderMouseOver="{TemplateBinding IsMouseOver}">
       <RibbonLib:RibbonChrome.Content>
         <StackPanel SnapsToDevicePixels="True"
             Margin="{TemplateBinding Padding}"
             Focusable="False">
           <Border Height="74" Padding="1,1,1,0"
             x:Name="PART_ItemsPanelHost"
             SnapsToDevicePixels="True"
             Focusable="False"/>
             <DockPanel SnapsToDevicePixels="True"
                  Focusable="False">
               <Button Width="15" Height="14"
                  Margin="0,2,1,1" DockPanel.Dock="Right"
                  VerticalAlignment="Bottom"
                  x:Name="launcher" Command="{x:Static
```

```
                    RibbonLib:RibbonGroup.
                    LaunchDialogCommand}">
                    <Button.Template>
                     <ControlTemplate TargetType="{x:Type
                       Button}">
                      <RibbonLib:RibbonChrome
                       RenderPressed="{TemplateBinding
                       IsPressed}" RenderMouseOver=
                       "{TemplateBinding IsMouseOver}"
                       CornerRadius="0">
                       <Image Source="{DynamicResource
                       {ComponentResourceKey RibbonLib:Skins,
                       LauncherButtonImage}}"
                       Margin="1,1,0,0" Stretch="None"

                       SnapsToDevicePixels="True"/>
                     </RibbonLib:RibbonChrome>
                    </ControlTemplate>
                  </Button.Template>
                </Button>
              <TextBlock x:Name="groupTitle" Focusable="False"
                 TextTrimming="CharacterEllipsis"
                 Foreground="{DynamicResource
                 {ComponentResourceKey RibbonLib:Skins,
                 GroupTitleForegroundBrush}}"
                 Text="{TemplateBinding Title}"
                 VerticalAlignment="Center"
                 HorizontalAlignment="Center" FontSize="11"
                 Height="16" Visibility="Visible"
                 Margin="0,2,0,0"/>
          </DockPanel>
      </StackPanel>
    </RibbonLib:RibbonChrome.Content>
  </RibbonLib:RibbonChrome>
  <Popup x:Name="PART_Popup"  IsOpen="{Binding IsDropDownOpen,
      RelativeSource={RelativeSource Focusable="False"
      TemplatedParent},Mode=TwoWay}" PopupAnimation="Fade"
      AllowsTransparency="True" StaysOpen="False">
      <mwt:SystemDropShadowChrome Margin="0,0,5,5"
      Focusable="False" Color="#41000000" Name="Shdw"
      MinWidth="{TemplateBinding
      FrameworkElement.ActualWidth}"  CornerRadius="4"  >
      <Border Background="{DynamicResource
        {ComponentResourceKey RibbonLib:Skins,
        RibbonTabBrush}}" CornerRadius="3"
        BorderThickness="0" Focusable="False">
        <RibbonLib:RibbonChrome x:Name="popupchrome"
          Focusable="False"
          BorderBrush="{TemplateBinding BorderBrush}"
          Style="{DynamicResource {ComponentResourceKey
          RibbonLib:Skins, ExpandedRibbonGroupChromeStyle}}"
          RenderMouseOver="{Binding IsMouseOver,
          ElementName=popupchrome}">
```

```
                <StackPanel Margin="0" Focusable="False">
                  <Border x:Name="PART_PopupItemsPanelHost"
                    Padding="1,1,1,0" Focusable="False"/>
                    <DockPanel Focusable="False">
                     <Button Width="15" Height="14"
                       DockPanel.Dock="Right"
                       VerticalAlignment="Bottom" Margin="0,2,1,1"
                       x:Name="launcher2" Command="{x:Static
                       RibbonLib:RibbonGroup.LaunchDialogCommand}">
                     <Button.Template>
                     <ControlTemplate TargetType="{x:Type Button}">
                      <RibbonLib:RibbonChrome
                        RenderPressed="{TemplateBinding IsPressed}"
                        RenderMouseOver="{TemplateBinding
                        IsMouseOver}" CornerRadius="0">
                      <Image Source="{DynamicResource
                        {ComponentResourceKey RibbonLib:Skins,
                        LauncherButtonImage}}" Margin="1,1,0,0"

                      Stretch="None" SnapsToDevicePixels="True"/>
                    </RibbonLib:RibbonChrome>
                   </ControlTemplate>
                 </Button.Template>
               </Button>
        <ContentControl  Foreground="{DynamicResource
           {ComponentResourceKey RibbonLib:Skins,
           GroupTitleForegroundBrush}}" Content="{TemplateBinding
           Title}" VerticalAlignment="Center"
           HorizontalAlignment="Center" Height="16"/>
      </DockPanel>
      </StackPanel>
      </RibbonLib:RibbonChrome>
     </Border>
   </mwt:SystemDropShadowChrome>
  </Popup>
</Grid>
<ControlTemplate.Triggers>
  <Trigger Property="IsDialogLauncherVisible" Value="False">
   <Setter Property="Visibility" Value="Collapsed"
     TargetName="launcher"/>
  </Trigger>
  <Trigger Property="IsDialogLauncherVisible" Value="False">
    <Setter Property="Visibility" Value="Collapsed"
          TargetName="launcher2"/>
  </Trigger>
  <Trigger Property="IsMinimized" Value="True">
    <Setter Property="Visibility" Value="Collapsed"
          TargetName="PART_ItemsPanelHost"/>
    <Setter Property="Visibility" Value="Collapsed"
          TargetName="chrome1"/>
    <Setter Property="Visibility" Value="Visible"
          TargetName="PART_Button"/>
    <Setter Property="Visibility" Value="Collapsed"
```

```
            TargetName="groupTitle"/>
  </Trigger>
  </ControlTemplate.Triggers>
  <ControlTemplate.Resources>
    <Style TargetType="{x:Type Separator}"
           BasedOn="{StaticResource RibbonSeparator}"/>
   <Style TargetType="{x:Type RibbonLib:RibbonChrome}"
           BasedOn="{StaticResource RibbonChromeStyle}"/>
  </ControlTemplate.Resources>
 </ControlTemplate>
 <Style TargetType="{x:Type RibbonLib:RibbonGroup}">
    <Setter Property="BorderBrush" Value="{DynamicResource
    {ComponentResourceKey RibbonLib:Skins, DefaultBorderBrush}}"/>
    <Setter Property="Foreground"  Value="{DynamicResource
      {ComponentResourceKey RibbonLib:Skins,
       RibbonForegroundBrush}}"/>
    <Setter Property="MinWidth" Value="48"/>
    <Setter Property="MinHeight" Value="88"/>
    <Setter Property="Margin" Value="2,2,2,2"/>
    <Setter Property="Template" Value="{StaticResource
            RibbonGroupTemplate}"/>
    <Setter Property="Focusable" Value="False"/>
  </Style>
</ResourceDictionary>
```

- C#代码

我们从Control中派生出RibbonGroup，所以首先要做的就是为RibbonGroup中的其他元素提供宿主。为此，在RibbonGroup中加入Controls属性：

```
private ObservableCollection<UIElement> controls = new
ObservableCollection<UIElement>();
 public Collection<UIElement> Controls { get { return controls; } }
 protected override System.Collections.IEnumerator LogicalChildren
 {
    get
    {
     return controls.GetEnumerator();
    }
 }
```

RibbonGroup在正常和缩小模式下，其中的子元素需要在两个宿主间进行切换。

为此，RibbonGroup定义了ItemPanelInstance

```
 private Panel itemPanelInstance = null;
 private Panel ItemPanelInstance
 {
     get
     {
         if (itemPanelInstance == null)
         {
             if (ItemsPanel != null)
             {
```

```
                itemPanelInstance = ItemsPanel;
            }
            else itemPanelInstance = new RibbonWrapPanel();
        }
        return itemPanelInstance;
    }
}
```

这是一个自定义排版类：RibbonWrapPanel。在初始化视觉树时，我们把RibbonGroup中的子元素放到IntemPanelInstance中：

```
public override void OnApplyTemplate()
{
    ...
   if (ItemPanelInstance != null)
       {
           foreach (UIElement e in Controls)
           {
               ItemPanelInstance.Children.Add(e);
           }
       }
    ...
}
```

然后，根据不同的情况，把这个排版在正常显示宿主和弹出窗口中的宿主间进行切换：

```
private void AttachControlsToItemsPanel()
{
    if (ItemPanelInstance != null)
     {
      Decorator parent = ItemPanelInstance.Parent as Decorator;
      if (parent != null) parent.Child = null;
      Decorator host = IsMinimized ? popupItemsPanelHost :
                       itemsPanelHost;
      if (host != null)
      {
         host.Child = ItemPanelInstance;
      }
   }
}
```

● 使用WPF命令

Ribbon分组的右下角按钮为显示分组对话框按钮，我们把它和WPF命令连续起来。你可以使用WPF传递事件来实现，然而，若要在Ribbon模板中实现这一功能的话，则可以绑定WPF命令(因在模板中，无法绑定相关事件)：

```
<Button Width="15" Height="14" DockPanel.Dock="Right"
    VerticalAlignment="Bottom" Margin="0,2,1,1" x:Name="launcher"
    Command="{x:Static RibbonLib:RibbonGroup.LaunchDialogCommand}">
```

发出命令时，产生传递事件：

```
partial class RibbonGroup
```

```
{
   private static RoutedUICommand launchDialogCommand = new
       RoutedUICommand("Launch", "LaunchDialogCommand",
       typeof(RibbonGroup));
   private static void RegisterCommands()
   {
    CommandManager.RegisterClassCommandBinding(typeof(RibbonGroup),
           new CommandBinding(launchDialogCommand, launchDialog));
   }
   private static void launchDialog(object sender,
                 ExecutedRoutedEventArgs e)
   {
       RibbonGroup group = (RibbonGroup)sender;
       RoutedEventArgs args = new RoutedEventArgs(
                               ExecuteLauncherEvent);
     group.RaiseEvent(args);
   }

  public static RoutedUICommand LaunchDialogCommand
  {
       get { return launchDialogCommand; }
  }
}
```

由于RibbonGroup.cs 代码太长，可参见本书光盘中的内容或与笔者联系。

18.4.4 RibbonTabItem

图18-1中示出了Word 2007中的RibbonTabItem，它位于窗口标题栏的下方，RibbonGroup是其中的子元素。

- UI部分

RibbonTabItem有两种状态，一种是选中状态，另一种是未选中状态。RibbonTabItem在选中状态时，需要绘制标签部分，以示区别。在未选中状态时，只需要显示其标题。

在未选中时，其视觉树很简单：

```
 <RibbonLib:RibbonChrome >
     <TextBlock />
<!--     显示标题  -->
 </RibbonLib:RibbonChrome>
```

在选中RibbonTabItem时，需要绘制Tab的标签部分：

```
 <Grid Name="grid" >
  <Grid.ColumnDefinitions>
      <ColumnDefinition Width="Auto" />
      <ColumnDefinition Width="*" />
      <ColumnDefinition Width="Auto" />
  </Grid.ColumnDefinitions>
  <Path StrokeThickness="1"  Grid.Column="0"
     x:Name="Left" Data="M 0,23 L 3,20 L3,3 A 3,3 45 0,1 6,0 L 7,0
     L 7,23" /> <!--绘制标签的左边 -->
  <Path StrokeThickness="1" Margin="-1.5,0,0,0"
```

```
        SnapsToDevicePixels="False"  x:Name="Right"
        Grid.Column="2" Focusable="False"
        Data="M0,23 L 0,0 L 1,0 A 3,3 45 0,1 4,3 L 4,20 L7,23"/>
        <!--绘制标签的右边-->
  <Border BorderThickness="0,1,0,0" >
       ...
    <!-- 绘制标签中间 -->
    </Border>
  <Border Name="MouseOverBorder"  ">  在鼠标拖过时画出区别
    <Border.Effect>
     <BlurEffect KernelType="Gaussian" Radius="4"
           RenderingBias="Performance"/>
     </Border.Effect>
   </Border>
</Grid>
```

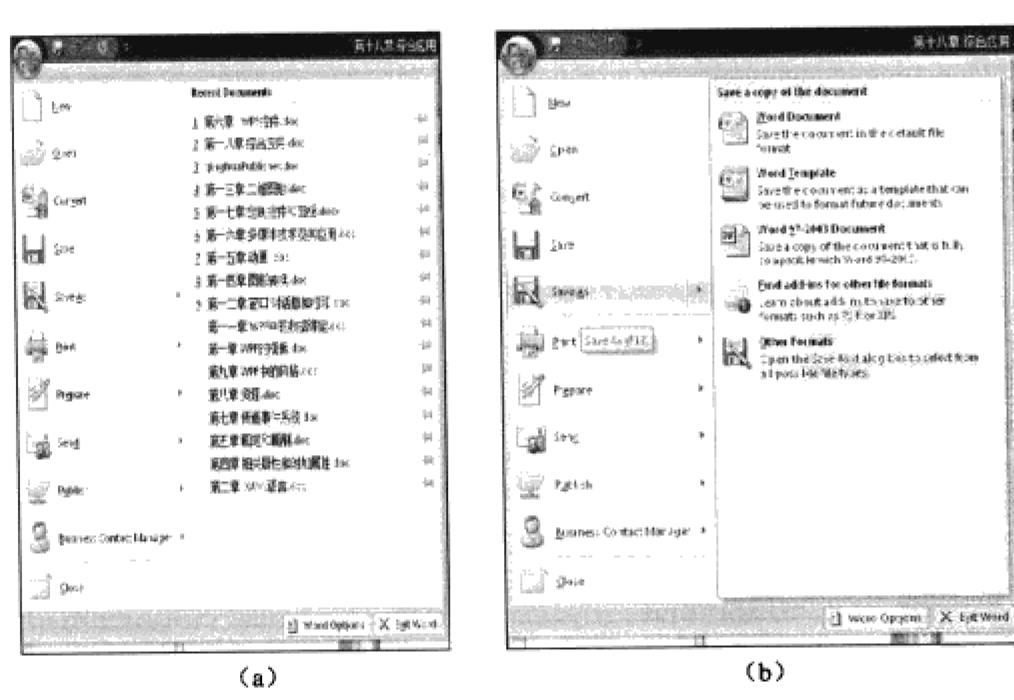
● 后台代码

RibbonTabItem也是从Control中派生出来的，其中可加入多个RibbonGroup，所以，我们定义一个Groups属性：

```
private ObservableCollection<RibbonGroup> groups = new
         ObservableCollection<RibbonGroup>();
public Collection<RibbonGroup> Groups { get { return groups; } }
```

18.4.5 RibbonApplicationMenuItem

图18-10示出了Word 2007 中的菜单条目。RibbonApplicationMenuItem需要处理两种不同的情况。一种是菜单条目不带有子菜单（如图18-10（a）所示），另一种是菜单条目带有子菜单（如图18-10（b）所示）。



（a）　　（b）

图18-10 Word 2007中的菜单条目

第一种情况比较简单，和通常菜单条目没什么区别，在菜单条目的左边显示图标（可选），在菜单条目的右边显示文字。第二种情况比较复杂，当选择带有子菜单的条目时，该条目实际上由三个部

分组成：左边的图标，中间的文字，右边的向右的箭头图标，右边弹出框里显示子菜单。

```
<Grid>
 <RibbonLib:RibbonChrome>
  <DockPanel HorizontalAlignment="Stretch" >
    <ToggleButton x:Name="PART_DropDown" Width="24"
         DockPanel.Dock="Right">
     <!--菜单条目右边带箭头的图标    -->
    </ToggleButton>
    <RibbonLib:RibbonChrome x:Name="item" CornerRadius="3,0,0,3">
      <Grid>
        <Grid.ColumnDefinitions>
          <ColumnDefinition Width="36"/>
          <ColumnDefinition Width="*"/>
        </Grid.ColumnDefinitions>
        <Border Name="GlyphPanel" Grid.Column="0">
              <!--显示选中标记    --->
        </Border>
              <!--显示条目图标    -->
        <Image  Name="Image" Grid.Column="0"/>
           <!--显示菜单文字  -->
        <ContentPresenter x:Name="content"  Grid.Column="1" />
      </Grid>
     </RibbonLib:RibbonChrome>
        </DockPanel>
    </RibbonLib:RibbonChrome>
      <Popup x:Name="PART_Popup" Placement="Right" >
      <!-- 显示弹出子菜单 -->
  </Popup>
 </Grid>
```

在使用数据绑定时，需要用到一些小技巧。首先，用触发器确定是否显示菜单条目右边的小箭头：

```
<Trigger Property="HasItems" Value="False">
     <Setter Property="Visibility" TargetName="PART_DropDown"
          Value="Collapsed"/>
</Trigger>
```

即当该条目没有子菜单时，我们不显示条目右边的箭头。其次显示图18-10（b）右边的子菜单，我们把弹出菜单的IsOpen绑定到IsSubmenuOpen相关属性上：

```
<Popup x:Name="PART_Popup" IsOpen="{Binding IsSubmenuOpen,
RelativeSource={RelativeSource TemplatedParent},Mode=TwoWay}">
```

同时，我们把条目右边ToggleButton的IsChecked相关属性绑定到IsSubmenuOpen相关属性上：

```
<ToggleButton x:Name="PART_DropDown"
  IsChecked="{Binding IsSubmenuOpen, RelativeSource={RelativeSource
TemplatedParent},Mode=TwoWay}">
```

这样，当ToggleButton的相关属性IsChecked为True时，IsSubmenuOpen也为True，当

IsSubmenuItemOpen为True时，自动设定PART_Pop up的相关属性IsOpen，从而自动显示右边的弹出窗口及其中的菜单。

18.4.6 RibbonApplicationMenu

RibbonApplicationMenu是一个自定义控件，它从MenuItem中派生出来。这个自定义控件的功能是移植类似于图18-10（a）中的菜单，该菜单分为四个部分：最下面的矩形区域中可以放入按钮控件，如图18-10（a）中的“Word Opentions”和“Exit word”。左边的矩形区域为显示菜单条目的地方；右边的矩形显示“RecentDocument”；左上角显示一个图标按钮，当单击该按钮时，弹出RibbonApplication菜单。

```
<Grid Margin="0">
<!-- 显示左上角图标按钮  -->
 <RibbonLib:RibbonToggleButton x:Name="PART_AppButton" >
  <Image Width="24" Height="24" Source="{TemplateBinding
         MenuButtonImage}"/>
  </RibbonLib:RibbonToggleButton>
<!--显示下拉菜单-->
  <Popup    x:Name="PART_AppMenuPopup">
    <Grid Focusable="False">
      <StackPanel Focusable="False">
       <Border BorderBrush="White" BorderThickness="1"
           Focusable="False">
          <Border   Width="480"">
           <Grid Margin="-1,0,-1,0" >
            <Grid.ColumnDefinitions>
              <ColumnDefinition Width="Auto"/>
              <ColumnDefinition Width="*"/>
            </Grid.ColumnDefinitions>
            <!--左边显示菜单条目  -->
           <Border Grid.Row="0" Grid.Column="0">
             <ItemsPresenter/>
           </Border>
    <!--右边显示最近访问的文件-->
       <StackPanel Grid.Column="1" Grid.Row="0"
           x:Name="PART_RecentItemsList">
                   ...
       </StackPanel>
      </Grid>
     </Border>
    </Border>
       <!--显示最下面的矩形区-->
    <ContentPresenter Content="{TemplateBinding Footer}"
        ContentTemplate="{TemplateBinding FooterTemplate}"/>
  </StackPanel>
  </Border>
 </Grid>
</mwt:SystemDropShadowChrome>
</Popup>
</Grid>
```

注意：在显示最下面的矩形区域时，我们用的是ContentPresenter，它绑定到相关属性Footer上，

我们在RibbonApplicationMenu.cs中定义Footer为object：

```
public object Footer
{
    get { return (object)GetValue(FooterProperty); }
    set { SetValue(FooterProperty, value); }
}

public static readonly DependencyProperty FooterProperty =
    DependencyProperty.Register("Footer", typeof(object),
   typeof(RibbonApplicationMenu), new UIPropertyMetadata(null));
```

即它可以是任何CLR元素。例如可以用下面的XAML来设定Footer：

```
<RibbonApplicationMenu.Footer>
  <StackPanel HorizontalAlignment="Stretch" Margin="0,4,0,0">
       <Button>退出</Button>
  </StackPanel>
</RibbonApplicationMenu.Footer>
```

18.4.7 RibbonQAToolBar

从图18-1可以看出，Ribbon的快速访问工具条可以出现在RibbonTabItem的上方也可以出现在RibbonTabItem的下方。Ribbon快速访问工具条中可以含有按钮或下拉式菜单，下拉式菜单总是在快速访问工具条的最后。

管理Ribbon快速访问工具条的位置，我们用ToolBarPlacement 相关属性来表示：

```
public QAPlacement ToolBarPlacement
 {
   get { return (QAPlacement)GetValue(ToolBarPlacementProperty); }
      set { SetValue(ToolBarPlacementProperty, value); }
 }

 public static readonly DependencyProperty ToolBarPlacementProperty
   =DependencyProperty.Register("ToolBarPlacement",
    typeof(QAPlacement), typeof(RibbonQAToolBar),
    new FrameworkPropertyMetadata(QAPlacement.Top,
        FrameworkPropertyMetadataOptions.AffectsMeasure |
        FrameworkPropertyMetadataOptions.AffectsArrange |
        FrameworkPropertyMetadataOptions.AffectsRender));
```

它可以取Top和Bottom两个值。

自定义RibbonQAToolBar是从ItemsControl中派生出来的，第6章WPF控件讨论过ItemControl，其中含有一个相关属性Items。当我们在XAML里写下面的语句：

```
<ItemsControl>
   <Button>Test1</Button>
   <TextBox>Test2</TextBox>
    ....
</ItemsControl>
```

这里的按钮和TextBox实际上被加到Items集合中。在开发RibbonQAToolBar的时候，用了一个小技巧。在模板中定义两个宿主排版类，一个用来放按钮，另一个用来放下拉式菜单：

```
<!--用来放按钮-->
<odc:RibbonQAToolbarPanel Orientation="Horizontal"
x:Name="PART_ToolBarPanel" />

<!--用来放下拉式菜单-->

<RibbonLib:RibbonQAToolbarPanel Orientation="Vertical"
x:Name="PART_MenuItemHost" Margin="2"/>
```

当RibbonQAToolBar中的按钮足够多，以至于工具条放不下时，需要把那些按钮放到另外一个排版类PART_OverflowHost中：

```
<RibbonLib:RibbonQAToolbarPanel Orientation="Horizontal"
x:Name="PART_OverflowHost"/>
```

类似地，在PART_ToolBarPanel放不下按钮时，需要把菜单放到PART_MemuItemOverflowHost中：

```
< RibbonLib:RibbonQAToolbarPanel Orientation="Vertical"
x:Name="PART_MenuItemOverflowHost" Margin="2"/>
```

18.4.8 RibbonBar

RibboBar是一个非常重要的自定义控件，它是从ContentControl中派生出来的。RibbonBar需要管理RibbonApplicationMenu、RibbonQATollBar、RibbonTabItem等。

图18-11示出了RibbonApplicationMenu、RibbonQAToolBar、窗口标题、PART_GroupPanel和PART_TabItemContainer在RibbonBar中的相对位置。PART_TabItemContainer用来显示TabItem的标题，PART_ GroupPanel是RibbonGroup的宿主。

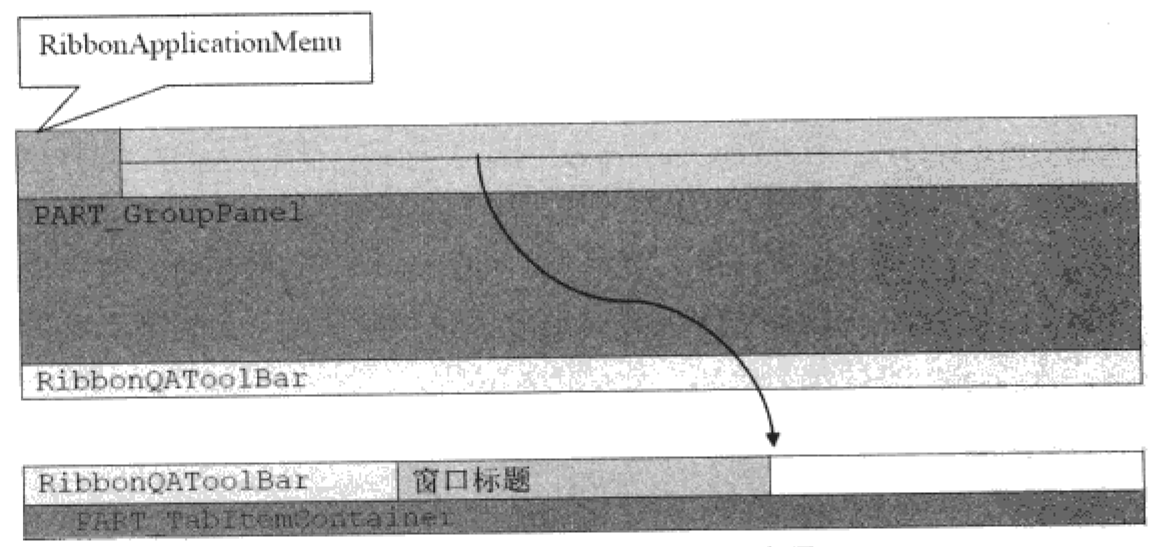


图18-11 RibbonBar中控件的布局

RibbonBar中定义了下属相关属性：

- QAToolBar用来支持快速访问工具条；

- ToolbarPlacement确定快速访问工具条的位置，它可取Top和Bottom两个值；
- ApplicationMenu用来支持应用程序的RibbonApplicationMenu；
- Tabs用来显示RibbonTabItem。

18.4.9 RibbonWindow

图18-12示出了在Windows Vista中的WPF默认窗口风格；图18-13示出了Word 2007 Ribbon窗口风格。在Ribbon窗口中，需要在左上角显示应用程序主菜单的图标，接着是快速访问工具条。显然，我们不能直接使用WPF所提供的Window类。

要在WPF中取得图18-13的效果，需要用到Win32中的函数，WPF开发小组成员Joe Castro就此写了篇博客（http://blogs.msdn.com/wpfsdk/archive/2008 /09/08/custom-window-chrome-in-wpf.aspx），在这篇文章里，他提出了解决这个问题的基本思路。

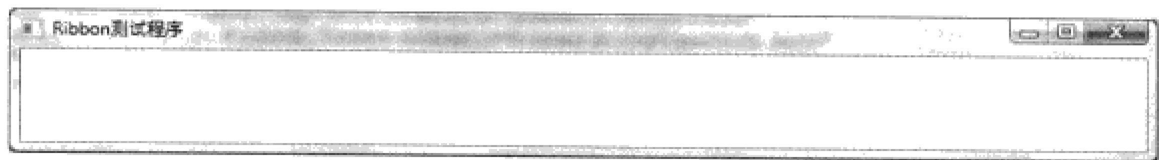


图18-12 WPF默认的窗口风格

- 自定义RibbonWindow界面

根据Joe的建议，自定义窗口RibbonWindow的界面可以在正常窗口中的客户区进行，我们需要自定义窗口的边框，最大化按钮、最小化按钮和关闭按钮。

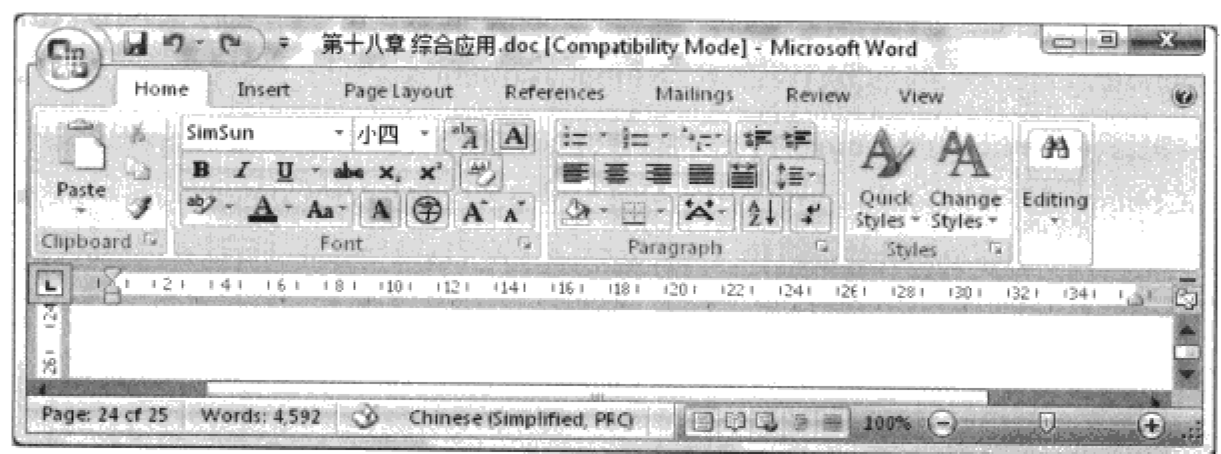


图18-13 Word 2007 Ribbon窗口风格

先创建一个窗口，并命名为TextWindow，并在其中加入一个3x3的Grid。第一行作为显示自定义窗口的标题及最大化、最小化、关闭窗口等按钮的地方；第二行第二列为自定义窗口的客户区；其他地方作为自定义窗口的边框。下面是实现这一想法的XAML：

```
<Window x:Class="RibbonTest.TestWindow"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
xmlns:RibbonLib=
    "clr-namespace:Yingbao.Chapter18.RibbonLib;assembly=YBRibbonLib"
    Title="TestWindow" Height="300" Width="300">
    <Border  SnapsToDevicePixels="True" ClipToBounds="True"
       Background="WhiteSmoke" BorderThickness="0"
```

```
        IsHitTestVisible="True" x:Name="PART_OuterBorder"
        CornerRadius="7">
    <Grid>
        <Grid.RowDefinitions>
         <RowDefinition Height="31"/>
          <RowDefinition Height="*"/>
          <RowDefinition Height="7"/>
        </Grid.RowDefinitions>
        <Grid.ColumnDefinitions>
          <ColumnDefinition Width="7"/>
           <ColumnDefinition Width="*"/>
           <ColumnDefinition Width="7"/>
        </Grid.ColumnDefinitions>
        <Border x:Name="border"  Grid.RowSpan="3" Grid.ColumnSpan="3"
           BorderThickness="8" BorderBrush="AliceBlue"
           CornerRadius="7"/>
      <Border x:Name="PART_Title" Background="LightBlue" Margin="1"
           ClipToBounds="True" Grid.ColumnSpan="3"
           CornerRadius="6,6,0,0"/>
      <Border BorderThickness="1" x:Name="innerBorder" Margin="1"
           Grid.RowSpan="3" Grid.ColumnSpan="3"   BorderBrush="Red"
           CornerRadius="6"/>
        <StackPanel Grid.ColumnSpan="3" Margin="0,2,10,2"
           TextBlock.Foreground="Red" Orientation="Horizontal"
           HorizontalAlignment="Right" VerticalAlignment="Center">
         <RibbonLib:RibbonButton x:Name="Minimize" Width="30"
            Height="17" Style="{DynamicResource {ComponentResourceKey
            RibbonLib:Skins, MinimizeButtonStyle}}"/>
         <RibbonLib:RibbonButton x:Name="Maximize" Width="30"
            Height="17" Style="{DynamicResource {ComponentResourceKey
            RibbonLib:Skins,MaximizeButtonStyle}}"/>
         <RibbonLib:RibbonButton x:Name="Close" Width="30"
            Height="17" Style="{DynamicResource {ComponentResourceKey
            RibbonLib:Skins, CloseButtonStyle}}"/>
        </StackPanel>
      <Border x:Name="contentBorder" Grid.IsSharedSizeScope="True"
           Margin="0,-26,0,0" Grid.Row="1" Grid.Column="1"
           IsHitTestVisible="True" >
      </Border>
     </Grid>
   </Border>
</Window>
```

笔者在上面的XAML里使用了本项目自定义的风格，运行该XAML，我们可以得到如图18-14所示的结果：

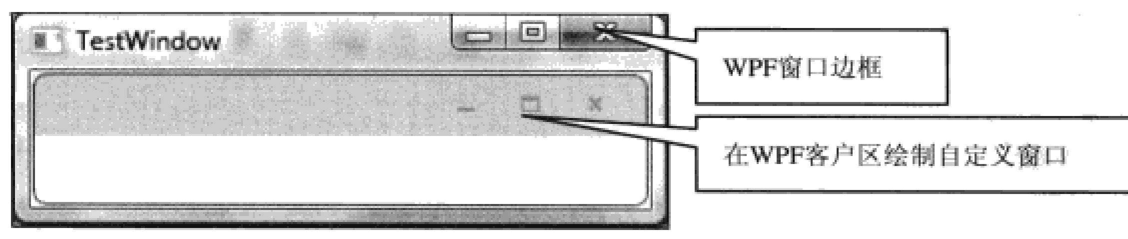


图18-14 自定义窗口

如果我们能去掉WPF窗口边框，而在最外层直接显示我们在原窗口客户区绘制的窗口，那么就达到了目的，我们需要通过调用Win32函数来实现这一功能。

● 去掉WPF窗口边框

前面提到，要去掉WPF窗口的边框，我们需要调用Win32函数来实现。在软件工程中，我们希望把某类功能集中在一块，以便于使用和维护。本项目中调用Win32函数放到一个类Win32Wrapper的类中：

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Runtime.InteropServices;
using System.Windows;

namespace Yingbao.Chapter18.RibbonLib
{
  public static class Win32Wrapper
   {
     public const int NOREPOSITION = 0x200;

     [DllImport("gdi32.dll", SetLastError = true)]
     public static extern IntPtr CreateRectRgn(int left, int top,
         int right, int bottom);

     [DllImport("user32.dll")]
     public static extern int SetWindowRgn(IntPtr hWnd, IntPtr hRgn,
          [MarshalAs(UnmanagedType.Bool)] bool bRedraw);

     [return: MarshalAs(UnmanagedType.Bool)]
     [DllImport("user32.dll")]
     public static extern bool GetWindowRect(IntPtr hWnd,
     .      out RECT lpRect);

     [return: MarshalAs(UnmanagedType.Bool)]
     [DllImport("user32.dll")]
     public static extern bool IsWindowVisible(IntPtr hwnd);

     [DllImport("gdi32.dll")]
     public static extern IntPtr CreateRoundRectRgn(int x1, int y1,
          int x2, int y2, int cx, int cy);

     [DllImport("gdi32.dll", SetLastError = true)]
     public static extern int CombineRgn(IntPtr hrgnDest,
          IntPtr hrgnSrc1, IntPtr hrgnSrc2, int fnCombineMode);

    [return: MarshalAs(UnmanagedType.Bool)]
    [DllImport("user32.dll", SetLastError = true)]
    public static extern bool SetWindowPos(IntPtr hWnd,
        IntPtr hWndInsertAfter, int x, int y, int cx, int cy,
        SWP uFlags);
```

```
[DllImport("user32.dll", EntryPoint = "GetWindowLongPtr")]
private static extern IntPtr GetWindowLongPtr64(IntPtr hWnd,
        GWL nIndex);

[DllImport("user32.dll", EntryPoint = "GetWindowLong")]
private static extern IntPtr GetWindowLongPtr32(IntPtr hWnd,
    GWL nIndex);

[DllImport("user32.dll", EntryPoint = "DefWindowProcW",
    CharSet = CharSet.Unicode)]
public static extern IntPtr DefWindowProc(IntPtr hWnd, WM Msg,
    IntPtr wParam, IntPtr lParam);

[DllImport("user32.dll", EntryPoint = "GetWindowLongPtr",
     SetLastError = true)]
private static extern IntPtr SetWindowLongPtr64(IntPtr hWnd,
     GWL nIndex, IntPtr dwNewLong);

[DllImport("user32.dll", EntryPoint = "SetWindowLong",
     SetLastError = true)]
private static extern IntPtr SetWindowLongPtr32(IntPtr hWnd,
     GWL nIndex, IntPtr dwNewLong);

[return: MarshalAs(UnmanagedType.Bool)]
[DllImport("dwmapi.dll")]
public static extern bool DwmDefWindowProc(IntPtr hwnd, int msg,
      IntPtr wParam, IntPtr  lParam, out IntPtr plResult);

[DllImport("dwmapi.dll", PreserveSig = false)]
public static extern void DwmExtendFrameIntoClientArea(
      IntPtr hwnd, ref MARGINS   pMarInset);

[DllImport("dwmapi.dll", PreserveSig = false)]
public static extern bool DwmIsCompositionEnabled();
[StructLayout(LayoutKind.Sequential)]
public struct MARGINS
 {
     public int cxLeftWidth;
     public int cxRightWidth;
     public int cyTopHeight;
     public int cyBottomHeight;
 }

 [StructLayout(LayoutKind.Sequential)]
 public struct NCCALCSIZE_PARAMS
 {
     public RECT rect0, rect1, rect2;
     public IntPtr lppos;
 }

 [Flags]
 public enum SWP
 {
```

```
        ASYNCWINDOWPOS = 0x4000,
        DEFERERASE = 0x2000,
        DRAWFRAME = 0x20,
        FRAMECHANGED = 0x20,
        HIDEWINDOW = 0x80,
        NOACTIVATE = 0x10,
        NOCOPYBITS = 0x100,
        NOMOVE = 2,
        NOOWNERZORDER = 0x200,
        NOREDRAW = 8,
        NOREPOSITION = 0x200,
        NOSENDCHANGING = 0x400,
        NOSIZE = 1,
        NOZORDER = 4,
        SHOWWINDOW = 0x40
    }

    public enum WM
    {
        ACTIVATE = 6,
        APP = 0x8000,
        CLOSE = 0x10,
        CREATE = 1,
        DESTROY = 2,
        DWMCOMPOSITIONCHANGED = 0x31e,
        ENABLE = 10,
        ERASEBKGND = 20,
        GETDLGCODE = 0x87,
        GETTEXT = 13,
        GETTEXTLENGTH = 14,
        KILLFOCUS = 8,
        MOVE = 3,
        NCACTIVATE = 0x86,
        NCCALCSIZE = 0x83,
        NCCREATE = 0x81,
        NCDESTROY = 130,
        NCHITTEST = 0x84,
        NCLBUTTONDBLCLK = 0xa3,
        NCLBUTTONDOWN = 0xa1,
        NCLBUTTONUP = 0xa2,
        NCMBUTTONDBLCLK = 0xa9,
        NCMBUTTONDOWN = 0xa7,
        NCMBUTTONUP = 0xa8,
        NCMOUSEMOVE = 160,
        NCPAINT = 0x85,
        NCRBUTTONDBLCLK = 0xa6,
        NCRBUTTONDOWN = 0xa4,
        NCRBUTTONUP = 0xa5,
        NULL = 0,
        PAINT = 15,
        QUERYENDSESSION = 0x11,
        QUERYOPEN = 0x13,
        QUIT = 0x12,
```

```
        SETFOCUS = 7,
        SETICON = 0x80,
        SETREDRAW = 11,
        SETTEXT = 12,
        SIZE = 5,
        SYNCPAINT = 0x88,
        SYSCHAR = 0x106,
        SYSCOLORCHANGE = 0x15,
        SYSCOMMAND = 0x112,
        SYSDEADCHAR = 0x107,
        SYSKEYDOWN = 260,
        SYSKEYUP = 0x105,
        USER = 0x400,
        WINDOWPOSCHANGED = 0x47,
          WINDOWPOSCHANGING = 70,
          MSG794 = 794
    }

  public enum HT
    {
        BORDER = 0x12,
        BOTTOM = 15,
        BOTTOMLEFT = 0x10,
        BOTTOMRIGHT = 0x11,
        CAPTION = 2,
        CLIENT = 1,
        CLOSE = 20,
        ERROR = -2,
        GROWBOX = 4,
        HELP = 0x15,
        HSCROLL = 6,
        LEFT = 10,
        MAXBUTTON = 9,
        MENU = 5,
        MINBUTTON = 8,
        NOWHERE = 0,
        OBJECT = 0x13,
        RIGHT = 11,
        SYSMENU = 3,
        TOP = 12,
        TOPLEFT = 13,
        TOPRIGHT = 14,
        TRANSPARENT = -1,
        VSCROLL = 7
    }

    public static IntPtr GetWindowLongPtr(IntPtr hwnd, GWL nIndex)
     {
        if (8 == IntPtr.Size)
        {
            return GetWindowLongPtr64(hwnd, nIndex);
        }
        return GetWindowLongPtr32(hwnd, nIndex);
```

```
   }

  public enum GWL
  {
     EXSTYLE = -20,
     HINSTANCE = -6,
     HWNDPARENT = -8,
     ID = -12,
     STYLE = -16,
     USERDATA = -21,
     WNDPROC = -4
  }

  [Flags]
  public enum WS : uint
  {
     BORDER = 0x800000,
     CAPTION = 0xc00000,
      CHILD = 0x40000000,
      CHILDWINDOW = 0x40000000,
      CLIPCHILDREN = 0x2000000,
      CLIPSIBLINGS = 0x4000000,
      DISABLED = 0x8000000,
      DLGFRAME = 0x400000,
      GROUP = 0x20000,
      HSCROLL = 0x100000,
      ICONIC = 0x20000000,
      MAXIMIZE = 0x1000000,
      MAXIMIZEBOX = 0x10000,
      MINIMIZE = 0x20000000,
      MINIMIZEBOX = 0x20000,
      OVERLAPPED = 0,
      OVERLAPPEDWINDOW = 0xcf0000,
      POPUP = 0x80000000,
      POPUPWINDOW = 0x80880000,
      SIZEBOX = 0x40000,
      SYSMENU = 0x80000,
      TABSTOP = 0x10000,
      THICKFRAME = 0x40000,
      TILED = 0,
      TILEDWINDOW = 0xcf0000,
      VISIBLE = 0x10000000,
      VSCROLL = 0x200000
    }

   public static IntPtr SetWindowLongPtr(IntPtr hwnd, GWL nIndex,
        IntPtr dwNewLong)
   {
      if (8 == IntPtr.Size)
      {
        return SetWindowLongPtr64(hwnd, nIndex, dwNewLong);
      }
      return SetWindowLongPtr32(hwnd, nIndex, dwNewLong);
```

```
        }

        public static int SignedLoWord(int i)
        {
            return (short)(i & 0xffff);
        }

        public static int SignedHiWord(int i)
        {
            return (short)(i >> 0x10);
        }

      [StructLayout(LayoutKind.Sequential)]
      public struct RECT
        {
          public int Left;
          public int Top;
          public int Right;
          public int Bottom;
          public int Width { get { return Right - Left; } }
          public int Height { get { return Bottom - Top; } }
        }
    }
}
```

去掉WPF窗口边框需要做两方面的工作。首先是调用SetWindowPos函数（http://msdn.microsoft.com/en-us/library/bb688195(VS.85).aspx）或SetWindowRgn函数（http://www.planet-source-code.com/vb/scripts/ShowCode. asp?txtCodeId=5620&lngWId=3）。这里，我们采用第二种方法。

其次，我们需要处理WM_NCCALCSIZE消息。由于Windows操作系统是消息驱动的，应用程序可以任意截取操作系统的消息并加以处理，应用程序没有处理的消息，会自动回到操作系统，操作系统对其进行默认处理。自定义窗口需要截获Win32消息，并对其进行特殊的处理。为了截获Win32消息，首先需要获得WPF窗口的句柄：

```
IntPtr hwnd = new WindowInteropHelper(_window).Handle;
```

然后，我们链接上自己的消息处理方法：

```
HwndSource.FromHwnd(hwnd).AddHook(_WndProc);

protected virtual IntPtr WndProc(IntPtr hwnd, int msg,
                    IntPtr wParam, IntPtr lParam, ref bool handled)
  {
      bool IsWindowActive;
      IntPtr result = IntPtr.Zero;
      Win32Wrapper.WM wm = (Win32Wrapper.WM)msg;
      switch (wm)
      {
          case Win32Wrapper.WM.SIZE:
          ...
              break;
          case Win32Wrapper.WM.NCCALCSIZE:
```

```
            ...
        break;
      }
}
```

在处理完WM_NCCALCSIZE消息之后，可以得到如图18-15 所示的窗口，这时不再显示WPF窗口边框。

● 移动窗口

当我们在窗口标题栏处按下鼠标左键，并移动鼠标时，窗口应跟着鼠标一起移动。对于自定义窗口，需要处理NTHITTEST消息。

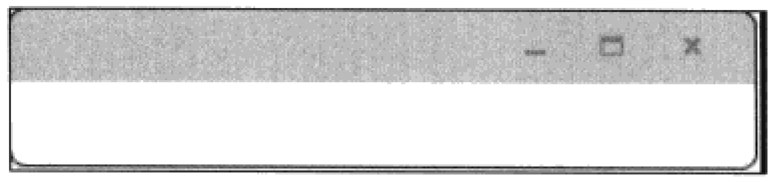


图18-15 去掉WPF窗口的边框

```
public partial class RibbonWindow : Window
  {
    const string partOuterBorder = "PART_OuterBorder";
    static RibbonWindow()
    {
      DefaultStyleKeyProperty.OverrideMetadata(
       typeof(RibbonWindow), new FrameworkPropertyMetadata(
       typeof(RibbonWindow)));
    }

public RibbonWindow(): base()
 {
    SizeChanged += new SizeChangedEventHandler(OnSizeChanged);
    HookWndProc();
    RegisterCommands();
 }

protected override void OnSourceInitialized(EventArgs e)
 {
     base.OnSourceInitialized(e);
     SetGlassOn();
 }

public bool IsGlassAvailable
 {
    get{
      if (Environment.OSVersion.Version.Major >= 6) {
        if (Win32Wrapper.DwmIsCompositionEnabled()) return true;
      }
      return false;
    }
 }

private void SetGlassOn()
```

```
{
  if (IsGlassAvailable)
   {
     IntPtr hwnd = new WindowInteropHelper(this).Handle;
     HwndSource src = HwndSource.FromHwnd(hwnd);
     src.CompositionTarget.BackgroundColor=
     Color.FromArgb(0, 0, 0, 0);
     const int GlassBorderSize = 8;
     const int GlassTitleBorderHeight = 31;

     Win32Wrapper.MARGINS margins = new Win32Wrapper.MARGINS();
     margins.cxLeftWidth = GlassBorderSize;
     margins.cxRightWidth = GlassBorderSize;
     margins.cyTopHeight = GlassTitleBorderHeight;
     margins.cyBottomHeight = GlassBorderSize;
     Win32Wrapper.DwmExtendFrameIntoClientArea(hwnd, ref margins);
    }
  }

private void HookWndProc()
{
  EventHandler handler = delegate(object sender, EventArgs e)
  {
   ((HwndSource)PresentationSource.FromVisual(this)).AddHook(new
              HwndSourceHook(this.WndProc));
   };
  base.SourceInitialized += handler;
}

 protected override Size ArrangeOverride(Size arrangeBounds)
 {
   UIElement child = this.VisualChildrenCount > 0 ?
            GetVisualChild(0) as UIElement : null;
   if (child != null)
   {
     child.Arrange(new Rect(arrangeBounds));
   }
   return arrangeBounds;
 }

 public bool IsGlassEnabled
 {
     get { return (bool)GetValue(IsGlassEnabledProperty); }
     set { SetValue(IsGlassEnabledProperty, value); }
 }

 public static readonly DependencyProperty IsGlassEnabledProperty =
           DependencyProperty.Register("IsGlassEnabled",
           typeof(bool), typeof(RibbonWindow),
           new FrameworkPropertyMetadata(false,
           GlassEnabledPropertyChanged));

public bool IsGlassOn
```

```
 {
     get { return (bool)GetValue(IsGlassOnProperty); }
     private set { SetValue(IsGlassOnPropertyKey, value); }
 }

private static readonly DependencyPropertyKey
    IsGlassOnPropertyKey = DependencyProperty.RegisterReadOnly(
         "IsGlassOn", typeof(bool), typeof(RibbonWindow), new
         UIPropertyMetadata(false, GlassOnPropertyChanged));

public static DependencyProperty IsGlassOnProperty =
    IsGlassOnPropertyKey.DependencyProperty;

static void GlassOnPropertyChanged(DependencyObject o,
           DependencyPropertyChangedEventArgs e)
 {
     RibbonWindow w = (RibbonWindow)o;
     w.SetWindowTitleBrush();
 }

static void GlassEnabledPropertyChanged(DependencyObject o,
           DependencyPropertyChangedEventArgs e)
 {
     RibbonWindow w = (RibbonWindow)o;
     w.SetIsGlassOnState();
 }

 private void SetIsGlassOnState()
 {
  IsGlassOn=IsGlassEnabled &&(Environment.OSVersion.Version.Major
     >= 6) && Win32Wrapper.DwmIsCompositionEnabled();
  AttachRegion();
 }

 protected virtual void OnSizeChanged(object sender,
                        SizeChangedEventArgs e)
 {
     AttachRegion();
     SetWindowTitleBrush();
 }

private void AttachRegion()
{
  if (!IsGlassOn)
  {
     Win32Wrapper.RECT rect;
     IntPtr hwnd = new WindowInteropHelper(this).Handle;
     Win32Wrapper.GetWindowRect(hwnd, out rect);
     int w = rect.Width + 1;
     int h = rect.Height + 1;
     if (WindowState != WindowState.Maximized &&
                RoundedCornerMode != RibbonWindowCornerMode.None)
     {
```

```
// note: the last two parameters are the diameter, not the radius:
      IntPtr rgn=Win32Wrapper.CreateRoundRectRgn(0,0,w,h,12,12);
      if (RoundedCornerMode == RibbonWindowCornerMode.Top)
      {
        IntPtr rgn2 = Win32Wrapper.CreateRectRgn(0, 6, w, h);
        Win32Wrapper.CombineRgn(rgn, rgn2, rgn, 2);
      }
      Win32Wrapper.SetWindowRgn(hwnd, rgn,
                     Win32Wrapper.IsWindowVisible(hwnd));
     }
     else
     {
       IntPtr rgn = Win32Wrapper.CreateRectRgn(0, 0, w, h);
       Win32Wrapper.SetWindowRgn(hwnd, rgn,
                   Win32Wrapper.IsWindowVisible(hwnd));
     }
   }
   else{
     IntPtr hwnd = new WindowInteropHelper(this).Handle;
     Win32Wrapper.SetWindowRgn(hwnd, IntPtr.Zero,
     Win32Wrapper.IsWindowVisible(hwnd));
   }
    }

   protected virtual IntPtr WndProc(IntPtr hwnd, int msg, IntPtr
           wParam, IntPtr lParam, ref bool handled)
   {
    IntPtr result = IntPtr.Zero;
    Win32Wrapper.WM wm = (Win32Wrapper.WM)msg;
    if (Environment.OSVersion.Version.Major >= 6)
    {
      handled = Win32Wrapper.DwmDefWindowProc(hwnd, msg, wParam,
                          lParam, out result);
     }
     if (!handled)
     switch (wm)
     {
         case Win32Wrapper.WM.SIZE:
             handled = false;
             break;
         case Win32Wrapper.WM.NCCALCSIZE:
             handled = true;
             break;
         case Win32Wrapper.WM.SETICON:
             handled = true;
             return IntPtr.Zero;
         case Win32Wrapper.WM.SETTEXT:
             handled = true;
             InvalidateArrange();
             break;
         case Win32Wrapper.WM.NCACTIVATE:
             IsWindowActive = wParam.ToInt32() == 1;
             handled = true;
```

```
                result = Win32Wrapper.DefWindowProc(hwnd,
                 Win32Wrapper.WM.NCACTIVATE, wParam, new IntPtr(-1));
                break;

            case Win32Wrapper.WM.NCHITTEST:
                if (result == IntPtr.Zero)
                {
                WndProcHitTest(hwnd, lParam, ref handled, ref result);
                }
                break;

            case Win32Wrapper.WM.DWMCOMPOSITIONCHANGED:
                 SetIsGlassOnState();
                 AttachRegion();
                 InvalidateVisual();
                 UpdateLayout();
                 handled = true;
                 result = IntPtr.Zero;
                 break;
        }

        return result;
    }

private void WndProcHitTest(IntPtr hwnd, IntPtr lParam, ref bool
        handled, ref IntPtr result)
 {
     int xy = lParam.ToInt32();
     Point p = new Point(Win32Wrapper.SignedLoWord(xy),
                    Win32Wrapper.SignedHiWord(xy));
     Win32Wrapper.RECT rect = new Win32Wrapper.RECT();
     Win32Wrapper.GetWindowRect(hwnd, out rect);
     Rect windowRect = new Rect(rect.Left, rect.Top,
          rect.Right - rect.Left, rect.Bottom - rect.Top);
     Win32Wrapper.HT ht = NCHitTest(p, windowRect);
     result = new IntPtr((int)ht);
     handled = ht != Win32Wrapper.HT.NOWHERE;
 }


private Win32Wrapper.HT NCHitTest(Point p, Rect rect)
{
    const double borderSize = 6.0;
    const double titleHeight = 34.0;
    if (p.Y < rect.Top + borderSize)
    {
       if (p.X < rect.Left + borderSize) return
                Win32Wrapper.HT.TOPLEFT;
       if (p.X > rect.Right - borderSize) return
                Win32Wrapper.HT.TOPRIGHT;
       return Win32Wrapper.HT.TOP;
    }
    if (p.Y > rect.Bottom - borderSize)
```

```
    {
      if (p.X < rect.Left + borderSize) return
                Win32Wrapper.HT.BOTTOMLEFT;
      if (p.X > rect.Right - borderSize) return
                Win32Wrapper.HT.BOTTOMRIGHT;
      return Win32Wrapper.HT.BOTTOM;
    }
  if (p.X < rect.Left + borderSize) return Win32Wrapper.HT.LEFT;
  if (p.X > rect.Right - borderSize) return Win32Wrapper.HT.RIGHT;
  if (p.Y < rect.Top + titleHeight)
  {
    Point localPoint = ScreenToLocal(p);
    IInputElement e = InputHitTest(localPoint);
    if (e == null)
    {
        return Win32Wrapper.HT.CAPTION;
    }
    else
    {
      if (e == outerBorder) return Win32Wrapper.HT.CAPTION;
      UIElement ue = e as UIElement;
      if (ue == null || !ue.IsHitTestVisible) return
            Win32Wrapper.HT.CAPTION;
     }
   }
   return Win32Wrapper.HT.NOWHERE;
}

private Point ScreenToLocal(Point point)
{
  Win32Wrapper.RECT rect;
  Win32Wrapper.GetWindowRect(new WindowInteropHelper(this).Handle,
          out rect);
  Matrix matrix =resentationSource.FromVisual(this)
                      .CompositionTarget.TransformFromDevice;
  point.Offset((double)(-1 * rect.Left), (double)(-1 * rect.Top));
  point.X *= matrix.M11;
  point.Y *= matrix.M22;
  return point;
}

private DependencyObject outerBorder;

public override void OnApplyTemplate()
{
   base.OnApplyTemplate();
   outerBorder = GetTemplateChild(partOuterBorder);
}

public bool IsWindowActive
{
  get { return (bool)GetValue(IsWindowActiveProperty); }
  private set { SetValue(IsWindowActivePropertyKey, value); }
```

```
}

private static readonly DependencyPropertyKey
    IsWindowActivePropertyKey=DependencyProperty.RegisterReadOnly(
      "IsWindowActive", typeof(bool),typeof(RibbonWindow),
      new FrameworkPropertyMetadata(false,
         FrameworkPropertyMetadataOptions.AffectsRender,
         WindowActivePropertyChanged));

public static readonly DependencyProperty IsWindowActiveProperty =
     IsWindowActivePropertyKey.DependencyProperty;

static void WindowActivePropertyChanged(DependencyObject o,
     DependencyPropertyChangedEventArgs e)
{
      RibbonWindow w = (RibbonWindow)o;
      w.SetWindowTitleBrush();
}

void SetWindowTitleBrush()
{
  if (WindowState == WindowState.Maximized && IsGlassOn)
   {
      WindowTitleBrush = Brushes.White;
   }
  else
   {
     WindowTitleBrush = IsWindowActive?
      SystemColors.ActiveCaptionTextBrush :
      SystemColors.InactiveCaptionTextBrush;
   }
}

public RibbonWindowCornerMode RoundedCornerMode
{
   get { return (RibbonWindowCornerMode)GetValue(
              RoundedCornerModeProperty); }
   set { SetValue(RoundedCornerModeProperty, value); }
}

public static readonly DependencyProperty RoundedCornerModeProperty
     = DependencyProperty.Register("RoundedCornerMode",
       typeof(RibbonWindowCornerMode), typeof(RibbonWindow),
     new FrameworkPropertyMetadata(RibbonWindowCornerMode.Top,
         FrameworkPropertyMetadataOptions.AffectsRender,
         RoundedCornerModePropertyChanged));

public static void RoundedCornerModePropertyChanged(
     DependencyObject o,DependencyPropertyChangedEventArgs e)
  {
     RibbonWindow window = (RibbonWindow)o;
     window.AttachRegion();
  }
```

```
 public Brush WindowTitleBrush
  {
    get { return (Brush)GetValue(WindowTitleBrushProperty); }
    private set { SetValue(WindowTitleBrushPropertyKey, value); }
  }

private static readonly DependencyPropertyKey
    WindowTitleBrushPropertyKey
       DependencyProperty.RegisterReadOnly("WindowTitleBrush",
         typeof(Brush),typeof(RibbonWindow),
         new UIPropertyMetadata(null));

public static DependencyProperty WindowTitleBrushProperty =
      WindowTitleBrushPropertyKey.DependencyProperty;


private static RoutedUICommand closeCommand = new
     RoutedUICommand("Close","CloseCommand",typeof(RibbonWindow));

private static RoutedUICommand minimizeCommand = new
RoutedUICommand("Minimize","MinimizeCommand",typeof(RibbonWindow));

private static RoutedUICommand maximizeCommand = new
RoutedUICommand("Maximize","MaximizeCommand",typeof(RibbonWindow));

private static void RegisterCommands()
{
 CommandManager.RegisterClassCommandBinding(typeof(RibbonWindow),
     new CommandBinding(closeCommand, PerformClose));

 CommandManager.RegisterClassCommandBinding(typeof(RibbonWindow),
     new CommandBinding(minimizeCommand, PerformMinimize));

  CommandManager.RegisterClassCommandBinding(typeof(RibbonWindow),
     new CommandBinding(maximizeCommand, PerformMaximize));
}
private static void PerformClose(object sender,
                 ExecutedRoutedEventArgs e)
{
   RibbonWindow window = (RibbonWindow)sender;
   window.Close();
}

private static void PerformMinimize(object sender,
                 ExecutedRoutedEventArgs e)
 {
     RibbonWindow window = (RibbonWindow)sender;
     window.WindowState = WindowState.Minimized;
 }

private static void PerformMaximize(object sender,
          ExecutedRoutedEventArgs e)
{
```

```
        RibbonWindow window = (RibbonWindow)sender;
        window.WindowState = window.WindowState ==
                WindowState.Maximized ? WindowState.Normal :
                WindowState.Maximized;
    }

    public static RoutedUICommand CloseCommand
    {
        get { return closeCommand; }
    }

    public static RoutedUICommand MinimizeCommand
    {
        get { return minimizeCommand; }
    }
    public static RoutedUICommand MaximizeCommand
    {
        get { return maximizeCommand; }
    }
}
```

18.4.10 支持不同皮肤

在互联网及桌面编程中，为了使网页和网页间或窗口和窗口间看起来相似，通常要用到骨架和皮肤（Themes/Skins）技术。所谓皮肤，是指控件的背景色、前景色、字体等。严格说来WPF并没有骨架和皮肤的概念，但是使用WPF中的Style，我们可以很容易达到给应用程序换皮肤的效果。

为了支持皮肤，通常在资源中加入控件的风格，把同一风格的设定放到一个文件中，从而形成一种皮肤。把另外一种风格放到另外一个文件中，从而形成不同的皮肤。要给应用程序换上某个皮肤，只要在应用程序的ResourceDictionary里引入该风格文件即可。

首先在OfficeBlueSkin.XAML中创建一个应用程序的基本风格，比如，其中定义CloseButton的样子为：

```
<ResourceDictionary>
...
<Style TargetType="{x:Type RibbonLib:RibbonButton}"
x:Key="DefaultCloseButtonStyle">
  <Setter Property="Foreground"
    Value="{DynamicResource {ComponentResourceKey RibbonLib:Skins,
           WindowButtonPenColor}}"/>
  <Setter Property="Template">
    <Setter.Value>
      <ControlTemplate TargetType="{x:Type RibbonLib:RibbonButton}">
       <RibbonLib:RibbonChrome
          RenderMouseOver="{TemplateBinding IsMouseOver}"
          RenderPressed="{TemplateBinding IsPressed}"
          HorizontalContentAlignment="Center"
          CornerRadius="3">
       <Path Data="M0,0L6,6M6,0L0,6"
          RenderOptions.BitmapScalingMode="HighQuality"
          StrokeThickness="2" SnapsToDevicePixels="True"
```

```
            Stroke="{TemplateBinding Foreground}"
            StrokeStartLineCap="Flat" StrokeEndLineCap="Flat"
            Stretch="None"/>
         </RibbonLib:RibbonChrome>
        </ControlTemplate>
      </Setter.Value>
    </Setter>
</Style>

<Style TargetType="{x:Type RibbonLib:RibbonButton}"
x:Key="{ComponentResourceKey RibbonLib:Skins, CloseButtonStyle}"
          BasedOn="{StaticResource DefaultCloseButtonStyle}"/>
      ...
</ResourceDictionary>
```

我们可以在OfficeBlackSkin.xaml文件中对CloseButton的某些属性进行修改，形成CloseButton的黑皮肤：

```
<Style TargetType="{x:Type RibbonLib:RibbonButton}"
   x:Key="{ComponentResourceKey RibbonLib:Skins, CloseButtonStyle}"
   BasedOn="{StaticResource DefaultCloseButtonStyle}">
   <Style.Resources>
     <Style TargetType="{x:Type RibbonLib:RibbonChrome}">
       <Setter Property="InnerBorderThickness"
               Value="0.2,0.5,0.2,0.5"/>
       <Setter Property="BorderBrush" Value="Black"/>
       <Setter Property="MouseOverBackground"
               Value="{StaticResource MouseOverWndButtonBrush}" />
       <Setter Property="MousePressedBackground"
              Value="{StaticResource MousePressedWndButtonBrush}"/>
     </Style>
   </Style.Resources>
   <Style.Triggers>
     <Trigger Property="IsMouseOver" Value="True">
       <Setter Property="Foreground" Value="Black"/>
      </Trigger>
      <Trigger Property="IsPressed" Value="True">
        <Setter Property="Foreground" Value="#FF7F8894"/>
      </Trigger>
    </Style.Triggers>
 </Style>
```

然后，在应用程序的资源文件中对控件的皮肤进行置换：

```
private static void ApplySkin(SkinId skin)
{
    var dict = Application.Current.Resources.MergedDictionaries;
       dict.Remove(OfficeBlack);
       dict.Remove(OfficeSilver);
       dict.Remove(WindowsSeven);
       switch (skin)
       {
           case SkinId.OfficeBlack:
```

```
                dict.Add(OfficeBlack);
                break;

            case SkinId.OfficeSilver:
                dict.Add(OfficeSilver);
                break;

            case SkinId.Windows7:
                dict.Add(WindowsSeven);
                break;
        }
}
```

整个用户界面可以实时改变皮肤。

18.5 使用Ribbon自定义控件实例

最后，让我们来看一下使用Ribbon自定义控件的效果，为此，我们创建一个简单的桌面应用程序：

```
<RibbonLib:RibbonWindow x:Class="Yingbao.Chapter18.RibbonTest.AppWin"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    x:Name="window"
    xmlns:local="clr-namespace:Yingbao.Chapter18.RibbonTest"
    xmlns:RibbonLib="clr-namespace:Yingbao.Chapter18.RibbonLib;
         assembly=YBRibbonLib"
    IsGlassEnabled="True" Title="使用Ribbon" Height="400" Width="850">
    <RibbonLib:RibbonWindow.Resources>
        <local:ThumbnailConverter x:Key="ThumbnailConverter"/>
    </RibbonLib:RibbonWindow.Resources>
    <DockPanel  RenderOptions.BitmapScalingMode="HighQuality" >
      <RibbonLib:RibbonBar x:Name="ribbonBar" Title="RibbonBar Demo"
                 DockPanel.Dock="Top" >
        <RibbonLib:RibbonBar.QAToolBar>
          <RibbonLib:RibbonQAToolBar>
            <RibbonLib:RibbonButton SmallImage="Images/save16.png"/>
            <RibbonLib:RibbonButton SmallImage="Images/undo16.png"/>
            <RibbonLib:RibbonButton SmallImage="Images/delete16.png"/>
            <RibbonLib:RibbonButton SmallImage="Images/folder16.png"/>
            <RibbonLib:RibbonToggleButton Content="Enable Glass"
               RibbonLib:RibbonBar.MinSize="Large"
               SmallImage="Images/search32.png"
               IsChecked="{Binding IsGlassEnabled,ElementName=window}"/>
            <RibbonLib:RibbonButton SmallImage="Images/props16.png"/>
            <RibbonLib:RibbonMenuItem Image="Images/props16.png"
               Header="显示在Ribbon的下面" Click="ShowBelowClick"/>
            <RibbonLib:RibbonMenuItem Image="Images/props16.png"
               Header="显示在Ribbon的上面" Click="ShowAboveClick"/>
            <Separator/>
            RibbonLib:RibbonMenuItem Header="Minimize Ribbon"
              IsCheckable="True" IsChecked="{Binding CanMinimize,
```

```
            ElementName=ribbonBar,Mode=TwoWay}"/>
        </RibbonLib:RibbonQAToolBar>
      </RibbonLib:RibbonBar.QAToolBar>
      <RibbonLib:RibbonBar.ApplicationMenu>
        <RibbonLib:RibbonApplicationMenu
            MenuButtonImage="Images/favorites16.png">
          <RibbonLib:RibbonApplicationMenuItem Header="打开文件..."/>
          <RibbonLib:RibbonApplicationMenuItem Header="存储文件"
              Image="Images/save32.png"/>
          <RibbonLib:RibbonApplicationMenuItem Header="另存为..."
              Image="Images/save32.png"/>
          <Separator/>
          <RibbonLib:RibbonApplicationMenuItem Header="最小 Ribbon"
           IsCheckable="True"
           IsChecked="{Binding CanMinimize, ElementName=ribbonBar}"/>
          <RibbonLib:RibbonApplicationMenuItem Header="打印...">
          <RibbonLib:RibbonApplicationMenuItem Header="子菜单1" />
          <RibbonLib:RibbonApplicationMenuItem Header="子菜单2" />
          <RibbonLib:RibbonApplicationMenuItem Header="子菜单3" />
          <RibbonLib:RibbonApplicationMenuItem Header="子菜单4" />
          <RibbonLib:RibbonApplicationMenuItem Header="子菜单5" />
          <RibbonLib:RibbonApplicationMenuItem Header="子菜单6" />
          <RibbonLib:RibbonApplicationMenuItem>
              <RibbonLib:RibbonMenuItem Header="第二级1"/>
              <RibbonLib:RibbonMenuItem Header="第二级2"/>
              <RibbonLib:RibbonMenuItem Header="第二级3"/>
          </RibbonLib:RibbonApplicationMenuItem>
          <RibbonLib:RibbonApplicationMenuItem Header="子菜单7" />
          <RibbonLib:RibbonApplicationMenuItem Header="子菜单8" />
          <RibbonLib:RibbonApplicationMenuItem Header="子菜单9" />
        </RibbonLib:RibbonApplicationMenuItem>
        <Separator/>
        <RibbonLib:RibbonApplicationMenuItem Header=" 退出"
            Command="{x:Static
            RibbonLib:RibbonWindow.CloseCommand}"/>
       <RibbonLib:RibbonApplicationMenu.RecentItemsList>
        <StackPanel>
        <TextBlock Text="最近打开的文档" Margin="3" MinHeight="20"
          HorizontalAlignment="Center"
          VerticalAlignment="Top" TextWrapping="Wrap"/>
        <TextBlock Text="第18章 综合应用.doc" Margin="3"
            HorizontalAlignment="Center" VerticalAlignment="Top"
            MinHeight="20" TextWrapping="Wrap"/>
         </StackPanel>
       </RibbonLib:RibbonApplicationMenu.RecentItemsList>
       <RibbonLib:RibbonApplicationMenu.Footer>
        <StackPanel HorizontalAlignment="Stretch" Margin="0,4,0,0">
          <Button HorizontalAlignment="Right" Command="{x:Static
             RibbonLib:RibbonWindow.CloseCommand}">Close Demo
          </Button>
```

```
      </StackPanel>
     </RibbonLib:RibbonApplicationMenu.Footer>
    </RibbonLib:RibbonApplicationMenu>
   </RibbonLib:RibbonBar.ApplicationMenu>
   <RibbonLib:RibbonTabItem Title="常用" >
     <RibbonLib:RibbonGroup Title="皮肤" Image="Images/home16.png"
          IsDialogLauncherVisible="True"
          ExecuteLauncher="RibbonGroup_LaunchDialog">
       <RibbonLib:RibbonToggleButton Content="Enable Glass"
          RibbonLib:RibbonBar.MinSize="Large"
          LargeImage="Images/search32.png"
          IsChecked="{Binding IsGlassEnabled,
          ElementName=window}"/>
       <RibbonLib:RibbonSeparator/>
       <RibbonLib:RibbonButton Content="Windows 7"
          RibbonLib:RibbonBar.MinSize
            ="Large" SmallImage="Images/save16.png"
          LargeImage="Images/Save32.png"
          Click="Win7Click"/>
       <RibbonLib:RibbonButton Content="Office Blue"
          RibbonLib:RibbonBar.MinSize="Medium"
          SmallImage="Images/home16.png"
          LargeImage="Images/home32.png" Click="OfficeBlueClick"/>
       <RibbonLib:RibbonButton Content="Office Silver"
          RibbonLib:RibbonBar.MinSize="Medium"
          SmallImage="Images/paste16.png"
          LargeImage="Images/paste32.png"
          Click="OfficeSilverClick"/>
       <RibbonLib:RibbonButton Content="Office Black"
          RibbonLib:RibbonBar.MinSize="Medium"
          SmallImage="Images/search16.png"
          LargeImage="Images/search32.png"
          Click="OfficeBlackClick"/>
    </RibbonLib:RibbonGroup>
    <RibbonLib:RibbonGroup Title="控件" Image="Images/undo16.png"
         IsDialogLauncherVisible="True" ExecuteLauncher=
         "RibbonGroup_LaunchDialog">
      <RibbonLib:RibbonGallery x:Name="gallery"
        RibbonLib:RibbonGallery.Stretch="None" DropDownColumns="5">
       <RibbonLib:RibbonGallery.ThumbnailSize>
          <Size Width="36" Height="36"/>
       </RibbonLib:RibbonGallery.ThumbnailSize>
       <RibbonLib:RibbonThumbnail x:Name="thumb1"
            ImageSource="Images/cut16.png"
            RibbonLib:RibbonGallery.Stretch="None"/>
       <RibbonLib:RibbonThumbnail x:Name="thumb2"
            ImageSource="Images/delete16.png"/>
       <RibbonLib:RibbonThumbnail
            ImageSource="Images/favorites16.png"/>
       <RibbonLib:RibbonThumbnail
            ImageSource="Images/folder16.png"/>
       <RibbonLib:RibbonThumbnail
            ImageSource="Images/history16.png"/>
```

```
      <RibbonLib:RibbonThumbnail ImageSource="Images/home16.png"/>
      <RibbonLib:RibbonThumbnail ImageSource="Images/mail16.png"/>
      <RibbonLib:RibbonThumbnail
           ImageSource="Images/paste16.png"/>
      <RibbonLib:RibbonThumbnail
           ImageSource="Images/props16.png"/>
      <RibbonLib:RibbonThumbnail ImageSource="Images/save16.png"/>
      <RibbonLib:RibbonThumbnail
           ImageSource="Images/search16.png"/>
      <RibbonLib:RibbonThumbnail ImageSource="Images/undo16.png"/>
    </RibbonLib:RibbonGallery>
    <RibbonLib:RibbonButton RibbonLib:RibbonBar.MaxSize="Medium"
        Content="Home" SmallImage="Images/home16.png"
        LargeImage="Images/home32.png"/>
    <RibbonLib:RibbonButton RibbonLib:RibbonBar.MaxSize="Medium"
        Content="Paste" SmallImage="Images/paste16.png"
        LargeImage="Images/paste32.png"/>
    <RibbonLib:RibbonButton RibbonLib:RibbonBar.MaxSize="Medium"
        Content="Search" SmallImage="Images/search16.png"
        LargeImage="Images/search32.png"/>
  </RibbonLib:RibbonGroup>
  <RibbonLib:RibbonGroup Title="组合框" Image="Images/mail16.png">
    <RibbonLib:RibbonComboBox Title="组合框"
        Image="Images/history16.png" ContentWidth="100">
      <RibbonLib:RibbonComboBoxItem Content="条目 1"
        Image="Images/mail16.png" />
      <RibbonLib:RibbonComboBoxItem Content="条目 2"
        Image="Images/props16.png"/>
      <RibbonLib:RibbonComboBoxItem Content="条目 3"/>
      <RibbonLib:RibbonComboBoxItem Content="条目 4"/>
    </RibbonLib:RibbonComboBox>
    <RibbonLib:RibbonTextBox x:Name="rbox" Title="文字"
      Image="Images/search16.png" Text="{Binding HotItem,
      ElementName=gallery,Converter={StaticResource
      ThumbnailConverter},Mode=OneWay}" ContentWidth="100"/>
    <RibbonLib:RibbonComboBox Title="可输入组合框"
      Image="Images/save16.png"
      ContentWidth="100" IsEditable="True" Text="Edit">
     <RibbonLib:RibbonComboBoxItem Content="条目 a" />
     <RibbonLib:RibbonComboBoxItem Content="条目 b"/>
     <RibbonLib:RibbonComboBoxItem Content="条目 c"/>
     <RibbonLib:RibbonComboBoxItem Content="条目 d"/>
  </RibbonLib:RibbonComboBox>
  </RibbonLib:RibbonGroup>
  <RibbonLib:RibbonGroup Title="按钮" Image="Images/props16.png">
    <RibbonLib:RibbonButton RibbonLib:RibbonBar.MinSize="Medium"
        Content="按钮"
        SmallImage="Images/undo16.png"
        LargeImage="Images/undo32.png" Click="ContextOffClick"/>
    <RibbonLib:RibbonToggleButton Content="拨动按钮"
        RibbonLib:RibbonBar.MinSize="Medium"
```

```
            SmallImage="Images/search16.png"
            LargeImage="Images/search32.png"/>
         <RibbonLib:RibbonDropDownButton Content="下拉按钮"
            RibbonLib:RibbonBar.MinSize="Medium"
            SmallImage="Images/folder16.png"
            LargeImage="Images/folder32.png">
          <RibbonLib:RibbonMenuItem Header="Vista"
               Image="Images/search16.png"
               IsCheckable="True" IsChecked="{Binding
               IsGlassEnabled,ElementName=window}"/>
          <RibbonLib:RibbonMenuItem Header="条目 2"
               Image="Images/cut16.png"/>
          <RibbonLib:RibbonMenuItem Header="条目 3"
               Image="Images/cut16.png"/>
          <RibbonLib:RibbonDropDownButton.DropDownHeader>
             <TextBlock Text="标题" Background="Orange"/>
          </RibbonLib:RibbonDropDownButton.DropDownHeader>
          <RibbonLib:RibbonDropDownButton.DropDownFooter>
             <TextBlock Text="脚注" Background="Lime"/>
          </RibbonLib:RibbonDropDownButton.DropDownFooter>
        </RibbonLib:RibbonDropDownButton>
        <RibbonLib:RibbonSplitButton
           RibbonLib:RibbonBar.MinSize="Medium"
           Content="分层按钮"  Click="Context1Click"
           SmallImage="Images/delete16.png"
           LargeImage="Images/delete32.png">
           <RibbonLib:RibbonMenuItem Header="条目 1"
             Image="Images/cut16.png"/>
           <RibbonLib:RibbonMenuItem Header="条目 2"
             Image="Images/cut16.png"/>
           <RibbonLib:RibbonMenuItem Header="条目 3"
             Image="Images/cut16.png"/>
           <RibbonLib:RibbonSplitButton.DropDownFooter>
             <Border Background="Lime" Padding="8,4,4,4">
               <CheckBox HorizontalAlignment="Left">选择</CheckBox>
             </Border>
           </RibbonLib:RibbonSplitButton.DropDownFooter>
          </RibbonLib:RibbonSplitButton>
        </RibbonLib:RibbonGroup>
        <RibbonLib:RibbonGroup Title="各种按钮"
         Image="Images/favorites16.png"
         RibbonLib:RibbonBar.Reduction="Large,Large,Minimized" >
       <RibbonLib:RibbonFlowGroup>
       <RibbonLib:RibbonButtonGroup>
     <RibbonLib:RibbonToggleButton SmallImage="Images/cut16.png"/>
     <RibbonLib:RibbonToggleButton SmallImage="Images/delete16.png"/>
     <RibbonLib:RibbonToggleButton SmallImage="Images/paste16.png"/>
     </RibbonLib:RibbonButtonGroup>
     <RibbonLib:RibbonButtonGroup>
       <RibbonLib:RibbonButton SmallImage="Images/home16.png"/>
       <RibbonLib:RibbonButton SmallImage="Images/history16.png"/>
```

```
      <RibbonLib:RibbonButton SmallImage="Images/favorites16.png"/>
      <RibbonLib:RibbonButton SmallImage="Images/mail16.png"/>
     </RibbonLib:RibbonButtonGroup>
     <RibbonLib:RibbonButtonGroup>
      <RibbonLib:RibbonButton SmallImage="Images/search16.png"/>
      <RibbonLib:RibbonDropDownButton
           SmallImage="Images/undo16.png"/>
      <RibbonLib:RibbonButton SmallImage="Images/folder16.png"/>
      <RibbonLib:RibbonSplitButton SmallImage="Images/props16.png"/>
      <RibbonLib:RibbonButton SmallImage="Images/save16.png"/>
    </RibbonLib:RibbonButtonGroup>
    <RibbonLib:RibbonButtonGroup>
       <RibbonLib:RibbonButton SmallImage="Images/search16.png"/>
    </RibbonLib:RibbonButtonGroup>
   </RibbonLib:RibbonFlowGroup>
  </RibbonLib:RibbonGroup>
 </RibbonLib:RibbonTabItem>
 <RibbonLib:RibbonTabItem Title="编辑">
  <RibbonLib:RibbonGroup Title="CheckBoxes"
     Image="Images/favorites16.png" Padding="10,2,10,2">
     <CheckBox Content="Check 1" Margin="0,0,16,0"/>
     <CheckBox Content="Check 2"/>
     <CheckBox Content="Check 3"/>
     <Separator Width="16"/>
     <CheckBox Content="Check 4"/>
     <CheckBox Content="Check 5"/>
      <CheckBox Content="Check 6"/>
    </RibbonLib:RibbonGroup>
  <RibbonLib:RibbonGroup Title="Radio" >
    <RibbonLib:RibbonGroup.Resources>
     <ResourceDictionary>
        <Style TargetType="{x:Type RadioButton}">
           <Setter Property="Margin" Value="0,0,10,0"/>
        </Style>
     </ResourceDictionary>
    </RibbonLib:RibbonGroup.Resources>
    <RadioButton>Radio 1</RadioButton>
       <RadioButton>Radio 2</RadioButton>
       <RadioButton>Radio 3</RadioButton>
       <RadioButton>Radio 4</RadioButton>
       <RadioButton>Radio 5</RadioButton>
       <RadioButton>Radio 6</RadioButton>
     </RibbonLib:RibbonGroup>
  </RibbonLib:RibbonTabItem>

 <RibbonLib:RibbonTabItem Title="实例1">
   <RibbonLib:RibbonGroup Title="9 Buttons">
    <RibbonLib:RibbonButton Content="按钮 1"
         LargeImage="Images/paste32.png"
         SmallImage="Images/paste16.png"/>
    <RibbonLib:RibbonButton Content="按钮 2"
         LargeImage="Images/paste32.png"
        SmallImage="Images/paste16.png"/>
```

```
      <RibbonLib:RibbonButton Content="按钮 3"
          LargeImage="Images/paste32.png"
          SmallImage="Images/paste16.png"/>
      <RibbonLib:RibbonButton Content="按钮 4"
          LargeImage="Images/paste32.png"
          SmallImage="Images/paste16.png"/>
      <RibbonLib:RibbonButton Content="按钮 5"
          LargeImage="Images/paste32.png"
          SmallImage="Images/paste16.png"/>
      <RibbonLib:RibbonButton Content="按钮 6"
          LargeImage="Images/paste32.png"
          SmallImage="Images/paste16.png"/>
      <RibbonLib:RibbonButton Content="按钮 7"
          LargeImage="Images/paste32.png"
          SmallImage="Images/paste16.png"/>
      <RibbonLib:RibbonButton Content="按钮 8"
          LargeImage="Images/paste32.png"
          SmallImage="Images/paste16.png"/>
      <RibbonLib:RibbonButton Content="按钮 9"
          LargeImage="Images/paste32.png"
          SmallImage="Images/paste16.png"/>
    </RibbonLib:RibbonGroup>
    <RibbonLib:RibbonGroup Title="8 Buttons">
      <RibbonLib:RibbonButton Content="按钮 1"
          LargeImage="Images/paste32.png"
          SmallImage="Images/paste16.png"/>
      <RibbonLib:RibbonButton Content="按钮 2"
          LargeImage="Images/paste32.png"
          SmallImage="Images/paste16.png"/>
      <RibbonLib:RibbonButton Content="按钮 3"
          LargeImage="Images/paste32.png"
          SmallImage="Images/paste16.png"/>
      <RibbonLib:RibbonButton Content="按钮 4"
          LargeImage="Images/paste32.png"
          SmallImage="Images/paste16.png"/>
      <RibbonLib:RibbonButton Content="按钮 5"
          LargeImage="Images/paste32.png"
          SmallImage="Images/paste16.png"/>
      <RibbonLib:RibbonButton Content="按钮 6"
          LargeImage="Images/paste32.png"
          SmallImage="Images/paste16.png"/>
      <RibbonLib:RibbonButton Content="按钮 7"
          LargeImage="Images/paste32.png"
          SmallImage="Images/paste16.png"/>
      <RibbonLib:RibbonButton Content="按钮 8"
          LargeImage="Images/paste32.png"
          SmallImage="Images/paste16.png"/>
    </RibbonLib:RibbonGroup>
    <RibbonLib:RibbonGroup Title="4 Buttons">

        ....
    </RibbonLib:RibbonGroup>
  </RibbonLib:RibbonTabItem>
 </RibbonLib:RibbonBar>
 </DockPanel>
```

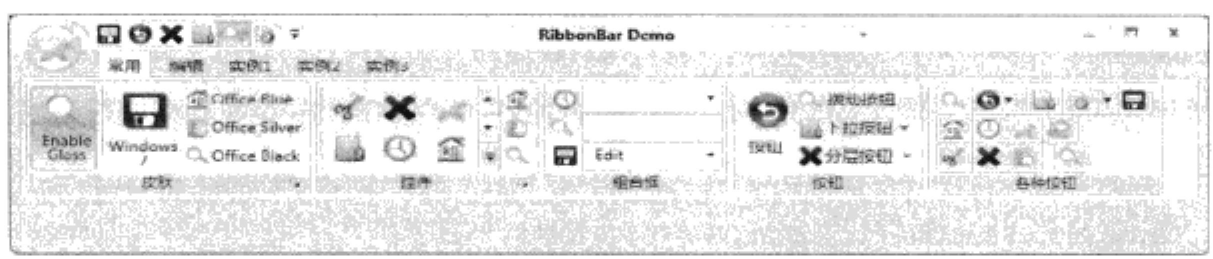
```
</RibbonLib:RibbonWindow>
```

C#代码：

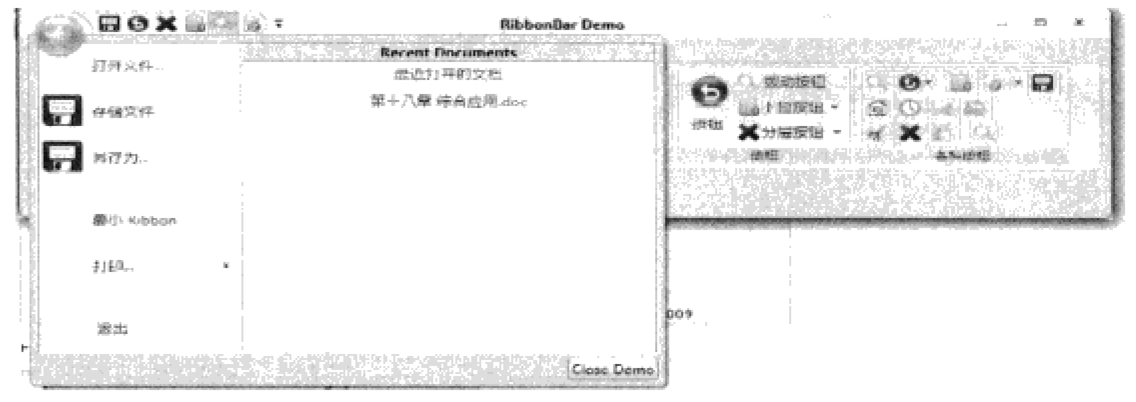
```
namespace Yingbao.Chapter18.RibbonTest
{

 public partial class AppWin :RibbonWindow
 {
   public AppWin()
    {
      InitializeComponent();
    }

   private void Win7Click(object sender, RoutedEventArgs e)
    {
      SkinManager.SkinId = SkinId.Windows7;
    }

   private void OfficeBlueClick(object sender, RoutedEventArgs e)
   {
      SkinManager.SkinId = SkinId.OfficeBlue;
   }

   private void OfficeSilverClick(object sender, RoutedEventArgs e)
   {
       SkinManager.SkinId = SkinId.OfficeSilver;
   }

   private void OfficeBlackClick(object sender, RoutedEventArgs e)
   {
       SkinManager.SkinId = SkinId.OfficeBlack;
   }

   private void Context1Click(object sender, RoutedEventArgs e)
   {
       ribbonBar.ContextualTabSet = ribbonBar.ContextualTabSets[0];
   }

   private void Context2Click(object sender, RoutedEventArgs e)
    {
        ribbonBar.ContextualTabSet = ribbonBar.ContextualTabSets[1];
    }

  private void ContextOffClick(object sender, RoutedEventArgs e)
   {
        ribbonBar.ContextualTabSet = null;
   }

  private void ShowBelowClick(object sender, RoutedEventArgs e)
  {
       ribbonBar.ToolbarPlacement = QAPlacement.Bottom;
  }

 private void ShowAboveClick(object sender, RoutedEventArgs e)
```

```
    {
        ribbonBar.ToolbarPlacement = QAPlacement.Top;
    }

private void CloseDemoClick(object sender, RoutedEventArgs e)
  {
        Close();
  }

 private void RibbonGroup_LaunchDialog(object sender,
          RoutedEventArgs e)
 {
            MessageBox.Show("Launcher");
  }
 }
}
```

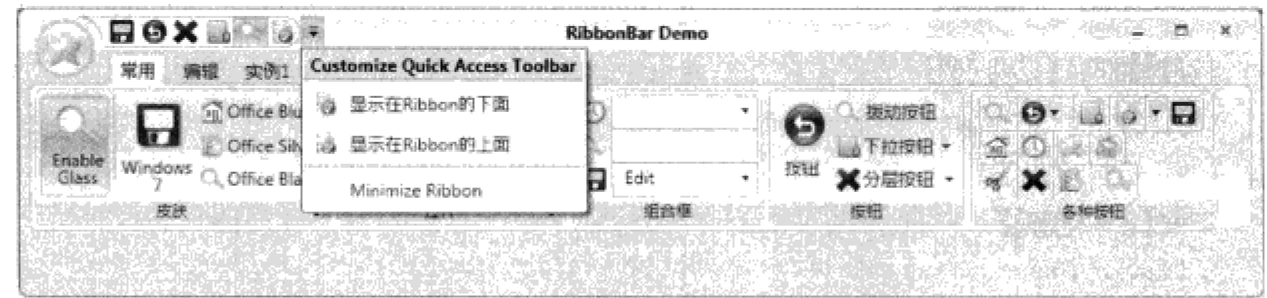
这段程序的运行结果如图18-16（a）、图18-16（b）、图18-16（c）和图18-16（d）所示。切换到Office Silver和Office Black皮肤的结果如图18-17（a）和图18-17（b）所示。



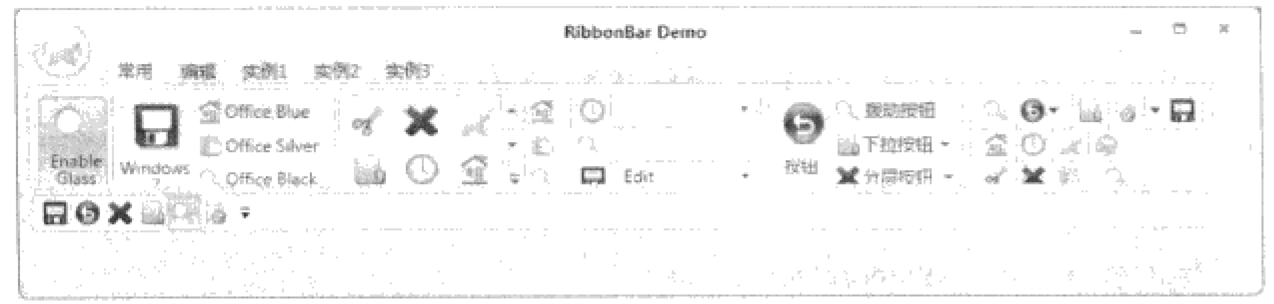
（a）使用Ribbon库里的自定义控件



（b）Ribbon 菜单



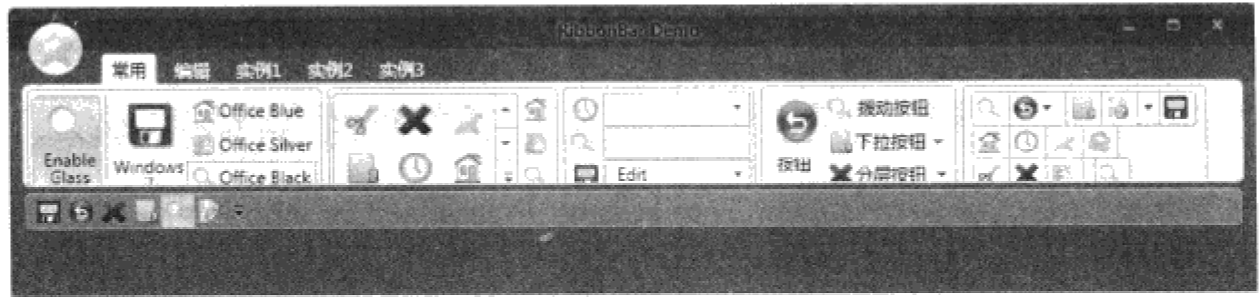
（c）Ribbon快速访问工具条，显示在Ribbon的上方



（d）Ribbon快速访问条显示在Ribbon的下方

图18-16 使用Ribbon自定义控件实例

（a）Office Silver皮肤



（b）Office Black皮肤

图18-17 换成Office Silver 和Office Black 皮肤

18.6 本章小结

本章通过对开源代码Ribbon类库的改写（http://www.codeproject.com/KB/ toolbars/WinFormsRibbon.aspx），总结了如何组织WPF项目，以及在面对实际工程问题时，使用WPF的思路。读完本书的读者应该可以开发任何WPF项目，恭喜你，你可以上路了！

参考文献

[1] http://msdn.microsoft.com/en-ca/default.aspx.

[2] http://windowsclient.net/.

[3] http://silverlight.net/.

[4] Charles Petzold, Applications = Code + Markup, Microsoft Press, 2006.

[5] Adam Nathan, Windows Presentation Foundation Unleashed, Sams Publishing, 2007.

[6] Chris Sells & Ian Griffiths, Programming Windows Presention Foundation, O'Reilly Media Inc., 2005.

[7] Jesse Liberty and Alex Horovitz, Programming .NET 3.5, O'Reilly Media Inc., 2008.

[8] Guy Eddon and Henry Eddon, Inside COM+ Base Services, Microsoft Press, 1999.

[9] Chris Peiris, Dennis Mulder Shawn Cicoria, Amit Bahree and Nishith Pathak, Pro WCF Practical Microsoft SOA Implementation, Apress, 2007.

[10] Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, Design Patterns, Addison Wesley Longman, Inc. 1985.

[11] Grady Booch, James Rumbaugh and Ivar Jacobson, The Unified Modeling Language User Guide, Addison Wesley Longman Inc. 1999.

[12] Various developers' blogs.

［Ｇｅｎｅｒａｌ　Ｉｎｆｏｒｍａｔｉｏｎ］
书名＝ＷＰＦ专业编程指南
作者＝李应保著
页数＝５２６
出版社＝电子工业出版社
出版日期＝２０１０
ＳＳ号＝１２４５５４９１
ＤＸ号＝
ＵＲＬ＝ｈｔｔｐ：／／ｂｏｏｋ２．ｄｕｘｉｕ．ｃｏｍ／ｂｏｏｋＤｅｔａｉｌ．ｊｓｐ？ｄｘＮｕｍｂｅｒ＝＆ｄ＝２０４０１８３５０Ｅ１５３ＡＢＢ９Ｂ７３ＦＥＢＣ８２Ｂ５６７１５

封面
书名
版权
前言
目录
第一篇WPF编程基础
第1章WPF应用程序
1．1　WPF应用程序的创建
1．2　Dos窗口
1．3　WPF应用程序的启动和终止
1．4输入参数
1．5在Xaml中创建Application
1．6窗口大小
1．7互联网应用程序
1．8应用程序的异常处理
1．9应用程序中的资源
1．10应用程序的发布
1．11　WPF开发环境
1．12本章小结
第2章XAML语言
2．1　XAML是一种界面描述语言
2．2　XAML的根元素
2．3　XAML命名空间（NameSpace ）
2．4　XAML和代码分离技术（code behind）
2．5子元素
2．6相关属性（Dependency Property）
2．7附加属性（Attached Property）
2．8　XAML标记扩展
2．8．1静态资源扩展（StaticResourceExtension）
2．8．2动态资源扩展（DynamicResourceExtension）
2．8．3数据绑定扩展（Binding）
2．8．4相对数据源扩展（RelativeSource）
2．8．5模板绑定（TemplateBinding）
2．8．6　x：Type扩展
2．8．7　x：Static扩展
2．8．8　x：nuII扩展
2．8．9　x：Array扩展
2．9本章小结
第3章WPF排版
3．1排版基础
3．2堆积面板（StackPanel）
3．3 WrapPanel
3．4停靠面板（DockPanel）
3．5表格式面板（Grid）
3．5．1设定UI元素在Grid中的位置
3．5．2设定　Grid行或列的尺寸
3．5．3元素横跨多个行列时的设定

３．５．４在Ｇｒｉｄ中保持多行或多列大小的一致性
３．６　ＵｎｉｆｏｒｍＧｒｉｄ
３．７画布面板（Ｃａｎｖａｓ）
３．８本章小结
第４章ＷＰＦ中的属性系统
４．１　ＣＬＲ属性
４．２相关属性的概念
４．２．１相关属性的传递
４．２．２　ＷＰＦ对相关属性的支持
４．３自定义相关属性
４．４附加属性
４．５本章小结
第５章　画笔和画刷
５．１　ＷＰＦ中的颜色
５．２画刷
５．２．１实心画刷（　ＳｏｌｉｄＣｏｌｏｒＢｒｕｓｈ　）
５．２．２梯度画刷（　ＧｒａｄｉｅｎｔＢｒｕｓｈ）
５．２．３线性梯度画刷（ＬｉｎｅａｒＧｒａｄｉｅｎｔＢｒｕｓｈ）
５．２．４圆形梯度画刷（ＲａｄｉａｌＧｒａｄｉｅｎｔＢｒｕｓｈ）
５．２．５自制画刷（　ＤｒａｗｉｎｇＢｒｕｓｈ）
５．２．６粘贴模式（ＴｉｌｅＭｏｄｅ）
５．２．７伸展方式（Ｓｔｒｅｔｃｈ）
５．２．８图像画刷（Ｉｍ　ａｇｅＢｒｕｓｈ）
５．２．９控件画刷（　ＶｉｓｕａｌＢｒｕｓｈ）
５．３画笔
５．４本章小结
第二篇ＷＰＦ专业程序员必备
第６章ＷＰＦ控制
６．１　Ｗ　ＰＦ控件概述
６．２内容控件（Ｃｏｎｔｅｎｔ　Ｃｏｎｔｒｏｌ　）
６．２．１框架控件（Ｆｒａｍｅ）
６．２．２　Ｗ　ＰＦ按钮（Ｂｕｔｔｏｎ）
６．２．３拨动按钮（ＴｏｇｇｌｅＢｕｔｔｏｎ）
６．２．４　ＣｈｅｃｋＢｏｘ控件
６．２．５　ＲａｄｉｏＢｕｔｔｏｎ控件
６．２．６重复按钮（ＲｅｐｅａｔＢｕｔｔｏｎ）
６．２．７带有标题栏的内容控件（ＨｅａｄｅｒｅｄＣｏｎｔｅｎｔＣｏｎｔｒｏｌ）
６．２．８分组框（Ｇ　ｒｏｕｐＢｏｘ）
６．２．９伸展控件（Ｅｘｐａｎｄｅｒ）
６．２．１０标签控件（Ｌａｂｅｌ）
６．２．１１为按钮设置热键
６．２.１２　ＴｏｏｌＴｉｐ
６．２.１３　ＳｃｒｏｌｌＶｉｅｗｅｒ
６．３条目控件（　Ｉｔｅｍｓ　Ｃｏｎｔｒｏｌｓ）
６．３．１菜单（Ｍｅｎｕ）
６．３．２工具条（ＴｏｏｌＢａｒ）

６．３．３　Ｓｅｌｅｃｔｏｒ
６．３．４组合框（ＣｏｍｂｏＢｏｘ）
６．３．５　ＴａｂＣｏｎｔｒｏｌ
６．３．６列表框（ＬｉｓｔＢｏｘ　）
６．３．７　ＬｉｓｔＶｉｅｗ
６．３．８状态条（ＳｔａｔｕｓＢａｒ）
６．３．９树形控件ＴｒｅｅＶ　ｉｅｗ和ＴｒｅｅＶｉｅｗｉｔｅｍ
６．４文本控件（Ｔｅｘｔ　Ｃｏｎｔｒｏｌｓ）
６．４．１口令输入框（ＰａｓｓｗｏｒｄＢｏｘ　）
６．４．２文字输入框（ＴｅｘｔＢｏｘ）
６．４．３　ＲｉｃｈＴｅｘｔＢｏｘ
６．５范围控件（Ｒａｎｇｅ　Ｃｏｎｔｒｏｌｓ）
６．５．１滚动条（ＳｃｒｏｌｌＢａｒ　）
６．５．２滑动条（Ｓｌｉｄｅｒ　）
６．５．３进展条（ＰｒｏｇｒｅｓｓＢａｒ）
６．６本章小结
第７章　传递事件和传递命令系统
７．１　ＷＰＦ中的元素树
７．２传递事件（Ｒｏｕｔｅｄ　Ｅｖｅｎｔ）
７．２．１　ＲｏｕｔｅｄＥｖｅｎｔＡｒｇｓ
７．２．２终止事件传播
７．２．３处理传递事件
７．２．４附加传递事件（Ａｔｔａｃｈｅｄ　Ｒｏｕｔｅｄ　Ｅｖｅｎｔ）
７．３考察传递事件
７．３．１键盘事件的产生和传递
７．４自定义传递事件
７．５管理键盘和鼠标输入事件
７．５．１键盘输入
７．５．２鼠标输入
７．６传递命令
７．６．１　ＩＣｏｍｍａｎｄ接口
７．６．２　ＩＣｏｍｍａｎｄＳｏｕｒｃｅ接口
７．６．３　ＣｏｍｍａｎｄＴａｒｇｅｔ
７．６．４命令绑定（ＣｏｍｍａｎｄＢｉｎｄｉｎｇ）
７．６．５传递命令（Ｒｏｕｔｅｄ　Ｃｏｍｍａｎｄ）
７．６．６　ＷＰＦ命令仓库（Ｃｏｍｍａｎｄ　Ｒｅｐｏｓｉｔｏｒｙ　）
７．７本章小结
第８章　资源
８．１资源定义及其类型
８．２统一资源标识（Ｕｎｉｆｉｅｄ　Ｒｅｓｏｕｒｃｅ　Ｉｄｅｎｔｉｆｉｅｒ）
８．３．ＮＥＴ开发平台对资源国际化的支持
８．３．１　ＷｉｎＦｏｒｍ下的资源管理
８．３．２用ＸＡＭ　Ｌ创建本地资源
８．４　ＷＰＦ元素中定义的资源
８．４．１静态资源（　ＳｔａｔｉｃＲｅｓｏｕｒｃｅ　）

８．４．２资源的作用范围
８．４．３静态扩展标识（Ｓｔａｔｉｃ　ｍａｒｋｕｐ　ｅｘｔｅｎｓｉｏｎ）
８．４．４动态资源扩展标识（ＤｙｎａｍｉｃＲｅｓｏｕｒｃｅ　Ｍａｒｋｕｐ　Ｅｘｔｅｎｓｉｏｎ　）
８．５本章小结
第９章　风格
９．１　Ｓｔｙｌｅ类
９．２　Ｓｅｔｔｅｒｓ
９．３　ＴａｒｇｅｔＴｙｐｅ
９．４　ＢａｓｅｄＯｎ
９．５触发器（Ｔｒｉｇｇｅｒｓ）
９．５．１使用单一条件的触发器
９．５．２使用多个条件的触发器
９．５．３使用数据触发器（ＤａｔａＴｒｉｇｇｅｒ）
９．６风格中的资源
９．７　ＩｓＳｅａｌｅｄ
９．８把风格定格定义在单独的文件中
９．９在ＦｒａｍｅｗｏｒｋＣｏｎｔｅｎｔＥｌｅｍｅｎｔ中使用风格
９．１０再谈Ｓｅｔｔｅｒ属性
９．１１本章小结
第１０章　模板
１０．１模板概述
１０．２控件模板
１０．２．１在控件中使用模板
１０．２．２在资源中使用模板
１０．２．３在控件模板中使用ＴａｒｇｅｔＴｙｐｅ
１０．２．４在模板中显示控件的内容
１０．２．５在模板中使用ＣｏｎｔｅｎｔＰｒｅｓｅｎｔｅｒ
１０．２．６模板中元素名Ｎａｍｅ属性
１０．２．７在模板中绑定控件的其他属性
１０．２．８使用模板显示电力系统的断路器和刀闸开关
１０．２．９在风格中使用模板
１０．２．１０获取ＷＰＦ控件的模板
１０．３数据模板（ＤａｔａＴｅｍｐｌａｔｅ）
１０．３．１我们所面临的问题
１０．３．２定义数据模板
１０．３．３在资源中使用数据模板
１０．３．４数据模板触发器
１０．３．５根据数据属性选择相应的模板
１０．３．６在数据模板中使用类型转换技术
１０．４　ＩｔｅｍｓＰａｎｅｌＴｅｍｐｌａｔｅ
１０．５层次结构数据模板（ＨｉｅｒａｒｃｈｉｃａｌＤａｔａＴｅｍｐｌａｔｅ　）
１０．６本章小结
第１１章　数据绑定（Ｄａｔａ　Ｂｉｎｄｉｎｇ）
１１．１数据绑定概述
１１．２最简单的数据绑定——从界面元素到界面元素
１１．２．１一对一数据绑定

１１．２．２在Ｃ＃中，实现数据绑定
１１．２．３对不是ＦｒａｍｅｗｏｒｋＥｌｅｍｅｎｔ和ＦｒａｍｅｗｏｒｋＣｏｎｔｅｎｔＥｌｅｍｅｎｔ元素实现数据绑定
１１．３使用不同的绑定模式
１１．４动态绑定
１１．５最简单的数据绑定——从．ＮＥＴ对象到界面元素
１１．６　ＤａｔａＣｏｎｔｅｘｔ
１１．７控制绑定时刻
１１．８开发自己的ＩＶａｌｕｅＣｏｎｖｅｒｔｅｒ
１１．９在数据绑定中加入校验
１１．９．１开发业务规则类
１１．９．２在绑定中添加任意多个业务规则
１１．９．３在控件上显示校验信息
１１．９．４触发错误处理事件
１１．９．５清除控件上的错误信息
１１．１０对集合对象的绑定
１１．１０．１使用ＤｉｓｐｌａｙＭｅｍｂｅｒＰａｔｈ属性
１１．１０．２显示当前条目
１１．１０．３遍历集合中的记录
１１．１０．４增加或删除记录
１１．１０．５对集合对象分组
１１．１０．６对集合对象排序
１１．１０．７对集合对象过滤
１１．１１数据源
１１．１１．１　ＸＭＬ数据源
１１．１１．２．ＮＥＴ对象数据源
１１．１２层次结构数据的绑定
１１．１３本章小结
第１２章　窗口对话框和打印
１２．１窗口（Ｗｉｎｄｏｗ）
１２．１．１窗口的状态变化和事件
１２．１．２确定视窗的位置
１２．１．３确定视窗的大小
１２．１．４视窗状态属性（ＷｉｎｄｏｗＳｔａｔｅ）
１２．１．５视窗大小模式（ＲｅｓｉｚｅＭｏｄｅ）
１２．１．６视窗风格（ＷｉｎｄｏｗＳｔｙｌｅ）
１２．２网页（Ｐａｇｅ）
１２．２．１创建网页
１２．２．２　ＫｅｅｐＡｌｉｖｅ属性
１２．２．３　Ｎａｖｉｇａｔｉｏｎ Ｓｅｒｖｉｃｅ属性
１２．２．４　ＳｈｏｗｓＮａｖｉｇａｔｉｏｎＵｌ属性
１２．３浏览窗口（ＮａｖｉｇａｔｉｏｎＷ ｉｎｄｏｗ ）
１２．３．１使用统一风格
１２．３．２设置ＮａｖｉｇａｔｉｏｎＷｉｎｄｏｗ的标题
１２．３．３浏览网页
１２．３．４使用ＨｙｐｅｒＬｉｎｋ类
１２．３．５使用ＮａｖｉｇａｔｉｏｎＳｅｒｖｉｃｅ转到不同的网页
１２．３．６使用浏览日志转换到不同的网页

１２．３．７浏览窗口的浏览事件
１２．４对话框（ＤｉａｌｏｇＢｏｘ）
１２．４．１消息框（ＭｅｓｓａｇｅＢｏｘ）
１２．４．２通用对话框
１２．４．３自定义对话框
１２．５打印输出
１２．５．１　ＸＰＳ文档简介
１２．５．２创建ＸＰＳ文档
１２．５．３显示ＸＰＳ文档
１２．５．４打印
１２．６本章小结
第三篇　图形和动画
第１３章　二维图形
１３．１　ＷＰＦ图形系统概述
１３．１．１统一编程模型
１３．１．２坐标系统
１３．１．３　Ｓｈａｐｅ和Ｇｅｏｍｅｔｒｙ
１３．２　Ｓｈａｐｅ及其派生类
１３．２．１直线（Ｌｉｎｅ）
１３．２．２矩形（Ｒｅｃｔａｎｇｌｅ）
１３．２．３椭圆（Ｅｌｌｉｐｓｅ）
１３．２．４折线（Ｐｏｌｙｌｉｎｅ）
１３．２．５多边形（Ｐｏｌｙｇｏｎ）
１３．２．６填充规则（ＦｉｌｌＲｕｌｅ）
１３．２．７路径（Ｐａｔｈ）
１３．３　Ｇｅｏｍｅｔｒｙ及其派生类
１３．３．１直线（ＬｉｎｅＧｅｏｍｅｔｒｙ）
１３．３．２矩形（ＲｅｃｔａｎｇｌｅＧｅｏｍｅｔｒｙ）
１３．３．３椭圆（ＥｌｌｉｐｓｅＧｅｏｍｅｔｒｙ）
１３．３．４几何图形组（ＧｅｏｍｅｔｒｙＧｒｏｕｐ）
１３．３．５合并图形（ＣｏｍｂｉｎｅｄＧｅｏｍｅｔｒｙ　）
１３．３．６几何路径（ＰａｔｈＧｅｏｍｅｔｒｙ）
１３．３．７分段路径（ＰａｔｈＳｅｇｍｅｎｔ）
１３．３．８弧线（ＡｒｃＳｅｇｍｅｎｔ）
１３．３．９直线段（ＬｉｎｅＳｅｇｍｅｎｔ）
１３．３．１０折线段（ＰｏｌｙＬｉｎｅＳｅｇｍｅｎｔ）
１３．３．１１柏之线（ＢｅｚｉｅｒＳｅｇｍｅｎｔ）
１３．３．１２多段柏之线（ＰｏｌｙＢｅｚｉｅｒＳｅｇｍｅｎｔ　）
１３．３．１３二次柏之线（Ｑｕａｄ　ｒａｔｉｃＢｅｚｉｅｒＳｅｇｍｅｎｔ　）
１３．３．１４多段二次柏之线（ＰｏｌｙＱｕａｄｒａｔｉｃＢｅｚｉｅｒＳｅｇｍｅｎｔ　）
１３．３．１５　迷你绘图语言
１３．３．１６流几何图形（ＳｔｒｅａｍＧｅｏｍｅｔｒｙ　）
１３．４绘制（Ｄｒａｗｉｎｇ）
１３．４．１使用ＤｒａｗｉｎｇＩｍａｇｅ显示几何图形
１３．４．２使用ＤｒａｗｉｎｇＶｉｓｕａｌ来显示几何绘制
１３．４．３创建ＤｒａｗｉｎｇＶｉｓｕａｌ宿主

１３．４．４绘制几何图形
１３．４．５把ＤｒａｗｉｎｇＶ ｉｓｕａｌ对象加到ＦｒａｍｅｗｏｒｋＥｌｅｍｅｎｔ中的视觉树和逻辑树中
１３．４．６选择视觉元素（Ｖｉｓｕａｌ Ｈｉｔ Ｔｅｓｔｉｎｇ）
１３．４．７简单选择判断
１３．４．８多个视觉元素的选择判断
１３．４．９视觉元素重叠时的选择判断
１３．５本章小结
第１４章　图形转换
１４．１图形转换概述
１４．２项目管理器
１４．３旋转转换（ＲｏｔａｔｅＴｒａｎｓｆｏｒｍ ）
１４．４位移转换（ＴｒａｎｓｌａｔｅＴｒａｎｓｆｏｒｍ ）
１４．５缩放转换（Ｓｃａ ｌ ｅＴｒａ ｎｓｆｏｒｍ ）
１４．６扭曲转换（ＳｋｅｗＴｒａｎｓｆｏｒｍ ）
１４．７组合转换（ＴｒａｎｓｆｏｒｍＧｒｏｕｐ ）
１４．８矩阵转换（ＭａｔｒｉｘＴｒａｎｓｆｏｒｍ ）
１４．８．１矢量操作
１４．８．２　Ｈ坐标系
１４．８．３位移转换矩阵
１４．８．４旋转转换矩阵
１４．８．５缩放转换矩阵
１４．８．６扭曲转换矩阵
１４．８．７　矩阵操作
１４．９本章小结
第１５章　动画
１５．１　ＷＰＦ中的动画
１５．２动画类继承树
１５．３一个简单的动画
１５．４控制动画
１５．４．１动画所用的时间　（ｄｕｒａｔｉｏｎ ）
１５．４．２设定动画开始时间ＢｅｇｉｎＴｉｍｅ
１５．４．３设定自动返回（ＡｕｔｏＲｅｖｅｒｓｅ ）
１５．４．４设定动画速度（ＳｐｅｅｄＲａｔｉｏ ）
１５．４．５加快和减慢动画（Ａｃｃｅｌａｒａｔｉｏｎ Ｒａｔｉｏ和ＤｅｃｅｌａｒａｔｉｏｎＲａｔｉｏ ）
１５．４．６设定动画的重复特性（ＲｅｐｅａｔＢｅｈａｖｉｏｒ ）
１５．４．７设定动画的终止状态（ＦｉｌｌＢｅｈａｖｉｏｒ ）
１５．４．８设定相关属性的动画范围（Ｆｒｏｍ和Ｔｏ ）
１５．４．９设定相关属性的动画范围（Ｂｙ ）
１５．４．１０设定ＩｓＡｄｄｉｔｉｖｅ和ＩｓＣｕｍｕｌａｔｉｖｅ属性
１５．４．１１　ＷＰＦ动画的时间片类
１５．５故事板（Ｓｔｏｒｙｂｏａｒｄ）
１５．５．１使用故事板的一般格式
１５．５．２设定ＴａｒｇｅｔＮａｍｅ和ＴａｒｇｅｔＮａｍｅ

１５．５．３操作Ｓｔｏｒｙｂｏａｒｄ
１５．６　ＫｅｙＦｒａｍｅ
１５．６．１线性ＫｅｙＦｒａｍｅ
１５．６．２非线性ＫｅｙＦｒａｍｅ　（Ｓｐｌｉｎｅ　ＫｅｙＦｒａｍｅ　）
１５．６．３离散ＫｅｙＦｒａｍｅ　（　ＤｉｓｃｒｅｔｅＫｅｙＦｒａｍｅ　）
１５．７本章小结
第四篇　开发ＷＰＦ产品
第１６章　多媒体技术及其应用
１６．１播放．ｗａｖ声音格式的ＳｏｕｄＰｌａｙｅｒ和Ｓｏｕｎｄ　ＰｌａｙｅｒＡｃｔｉｏｎ
１６．１．１装载．ｗａｖ文件
１６．１．２播放．ｗａｖ文件
１６．１．３停止播放
１６．１．４在ＸＡＭＬ中使用Ｓｏｕｎｄ　ＰｌａｙｅｒＡｃｔｉｏｎ
１６．２播放多种格式的声音和图像
１６．２．１播放模式
１６．２．２使用ＭｅｄｉａＰｌａｙｅｒ实例
１６．２．３使用ＭｅｄｉａＥｌｅｍｅｎｔ和ＭｅｄｉａＴｉｍｅｌｉｎｅ实例
１６．３语音合成和语音识别
１６．３．１尝试Ｗｉｎｄｏｗｓ　Ｖｉｓｔａ的语音功能
１６．３．２使你的程序发音
１６．３．３　ＰｒｏｍｐｔＢｕｉｌｄｅｒ和ＳＳ　Ｍ　Ｌ
１６．３．４语音识别中的语法
１６．４本章小结
第１７章　定制控件和排版
１７．１用户控件和自定义控件
１７．２创建用户控件（Ｕｓｅｒ　Ｃｏｎｔｒｏｌ）
１７．２．１设计用户控件Ｕ　Ｉ
１７．２．２开发支持用户控件ＵＩ的逻辑
１７．３创建自定义控件（Ｃｕｓｔｏｍ　Ｃｏｎｔｒｏｌ）
１７．４创建自定义排版（Ｃｕｓｔｏｍ　Ｐａｎｅｌ）
１７．４．１照片浏览器
１７．５本章小结
第１８章　综合应用
１８．１　Ｒｉｂｂｏｎ界面概览
１８．２项目的组织
１８．３管理Ｇｅｎｅｒｉｃ．ＸＡＭＬ文件
１８．４开发自定义控件
１８．４．１自定义控件间的关系
１８．４．２　Ｒｉｂｂｏｎ按钮
１８．４．３　Ｒｉｂｂｏｎ分组（Ｇｒｏｕｐ）
１８．４．４　ＲｉｂｂｏｎＴａｂｌｔｅｍ
１８．４．５　ＲｉｂｂｏｎＡｐｐｌｉｃａｔｉｏｎＭｅｎｕｌｔｅｍ
１８．４．６　ＲｉｂｂｏｎＡｐｐｌｉｃａｔｉｏｎＭｅｎｕ

１８．４．７　ＲｉｂｂｏｎＱＡＴｏｏｌＢａｒ
１８．４．８　ＲｉｂｂｏｎＢａｒ
１８．４．９　ＲｉｂｂｏｎＷｉｎｄｏｗ
１８．４．１０支持不同皮肤
１８．５使用Ｒｉｂｂｏｎ自定义控件实例
１８．６本章小结
参考文献