

.NET 开发经典名著

WPF 编程宝典

——使用 C# 2012 和.NET 4.5(第 4 版)

[美] Matthew MacDonald 著
王德才 译

清华大学出版社

北京

Matthew MacDonald

Pro WPF in C# 2012: Windows Presentation Foundation in .NET 4.5

EISBN: 978-1-4302-4365-6

Original English language edition published by Apress, 2855 Telegraph Avenue, #600, Berkeley, CA 94705 USA. Copyright © 2012 by Apress L.P. Simplified Chinese-Language edition copyright © 2013 by Tsinghua University Press. All rights reserved.

本书中文简体字版由 Apress 出版公司授权清华大学出版社出版。未经出版者书面许可，不得以任何方式复制或抄袭本书内容。

北京市版权局著作权合同登记号 图字：01-2013-5009

本书封面贴有清华大学出版社防伪标签，无标签者不得销售。

版权所有，侵权必究。侵权举报电话：010-62782989 13701121933

图书在版编目(CIP)数据

WPF 编程宝典——使用 C# 2012 和.NET 4.5(第 4 版) / (美)麦克唐纳(MacDonald, M.) 著; 王德才 译.
—北京：清华大学出版社，2013.8

(.NET 开发经典名著)

书名原文：Pro WPF in C# 2012: Windows Presentation Foundation in .NET 4.5

ISBN 978-7-302-32773-8

I. ①W… II. ①麦… ②王… III. ①Windows 操作系统—程序设计 ②C 语言—程序设计
IV. ①TP316.7 ②TP312

中国版本图书馆 CIP 数据核字(2013)第 136208 号

责任编辑：王军 李维杰

装帧设计：牛艳敏

责任校对：成凤进

责任印制：

出版发行：清华大学出版社

网 址：<http://www.tup.com.cn>, <http://www.wqbook.com>

地 址：北京清华大学学研大厦 A 座 邮 编：100084

社 总 机：010-62770175 邮 购：010-62786544

投稿与读者服务：010-62776969, c-service@tup.tsinghua.edu.cn

质 量 反 馈：010-62772015, zhiliang@tup.tsinghua.edu.cn

装 订 者：

经 销：全国新华书店

开 本：185mm×260mm 印 张：59.5 字 数：1600 千字

版 次：2009 年 8 月第 1 版 2013 年 8 月第 4 版 印 次：2013 年 8 月第 1 次印刷

印 数：1~4000

定 价：128.00 元

产品编号：

作者简介

Matthew MacDonald 是一位作家、教育家，曾三次荣膺微软 MVP。他迄今已经撰写了十多本有关.NET 编程的书籍，包括 *Pro Silverlight 5 in C#*(由 Apress 于 2012 年出版)和 *Beginning ASP.NET 4.5 in C#*(由 Apress 于 2012 年出版)。他还曾撰写 *Your Brain: The Missing Manual*(由 O'Reilly Media 于 2008 年出版)一书，该书讲述如何最大限度地激发大脑潜能，出版后受到广大读者的热烈欢迎。Matthew 目前与妻子和两个女儿居住在多伦多。

技术编辑简介

Fabio Claudio Ferracchiati 是一位前沿技术领域的多产作家。他已经撰写了十几本有关.NET、C#、Visual Basic 以及 ASP.NET 的编程书籍。他是.NET 领域微软认证的解决方案开发专家(MCSD)，现居意大利罗马。可以访问他的博客 Ferracchiati.com。

致 谢

如果没有他人相助，没有哪位作者能凭一己之力完成一本书籍。在撰写本书时，我极大地受惠于整个 Apress 团队，包括 Mark Powers，他全程负责本书的编辑工作；还有 Sharon Wilkey 和 Linda Seifert，他们加紧完成了审稿工作；还要感谢多位默默奉献的幕后工作者，他们负责编写索引页面、绘图以及最后的校对工作。

Fabio Claudio Ferracchiati 为本书提供了颇有见地且合乎时宜的技术分析评论，在此向他表示最诚挚的感谢。本书也吸收了来自各个 WPF 团队的众多博客的部分精髓，这些团队一直热衷于挖掘 WPF 最深层次的技术。我鼓励所有希望学习 WPF 未来版本的读者向他们学习。最后衷心感谢爱妻以及 Nora、Razia、Paul 和 Hamid，你们的支持给了我笔耕不辍的动力。感谢所有为本书作出贡献的人士！

前　　言

.NET 问世之初便引入了一些重要的新技术，包括编写 Web 应用程序的全新方法(ASP.NET)、连接数据库的全新方法(ADO.NET)、新的类型安全的语言(C#和 VB.NET)以及托管的运行时(CLR)。在这些新技术中，其中一项重要技术是 Windows 窗体，它是用于构建 Windows 应用程序的类库。

尽管 Windows 窗体是一个功能完备的工具包，但它绑定到旧式的核心 Windows 技术。最重要的是，Windows 窗体依靠 Windows API 创建标准用户界面元素的可视化外观，如按钮、文本框和复选框等。所以这些要素在本质上是不可定制的。例如，如果希望创建时髦的光晕按钮，就需要创建自定义控件，并使用低级的绘图模型为按钮(各种不同的状态)绘制各个方面的细节。更糟的是，普通窗口被切割成不同的区域，每个控件完全拥有自己的区域。所以没有较好的绘制方法可将一个控件的内容(如按钮背后的辉光效果)延伸到其他控件所占的区域中。更不要指望实现动画效果，如旋转文本、闪烁按钮、收缩窗口以及实时预览等，因为对于这些效果必须手工绘制每个细节。

WPF(Windows Presentation Foundation)通过引入一个使用完全不同技术的新模型改变了所有这一切。尽管 WPF 也提供了大家熟悉的标准控件，但它“自行”绘制每个文本、边框和背景填充。所以 WPF 的功能更强大，可以改变渲染屏幕上所有内容的方式。使用这些特性，可重新设置常见控件的样式(如按钮)，并且通常不需要编写任何代码。同样，可使用变换对象旋转、拉伸、缩放以及扭曲用户界面中的所有内容，甚至可使用 WPF 动画系统对用户界面中的内容进行变换。并且因为 WPF 引擎将在窗口上渲染的内容作为单独操作的一部分，所以能处理任意多层相互重叠的控件，即使这些控件具有不规则的形状且是半透明的也同样如此。

在 WPF 这些新特性的背后是基于 DirectX 的功能强大的基础结构，DirectX 是一套硬件加速的图形 API，通常用于开发最前沿的计算机游戏。这意味着可使用丰富的图形效果，而不会损失性能，而使用 Windows 窗体实现此类效果会严重影响程序运行的性能。实际上，甚至可使用更高级特性，例如对视频文件和 3D 内容的支持。使用这些特性以及优秀的设计工具，可创建出令人赏心悦目的用户界面和可视化效果，而使用 Windows 窗体技术是无法实现这些效果的。

还有必要指出，可使用 WPF 的标准控件和简单的可视化外观来构建普通 Windows 应用程序。实际上，在 WPF 中，可以像在旧式 Windows 窗体模型中那样方便地使用通用控件。更值得一提的是，WPF 增强了商业开发人员所需要的特性，包括大幅改进的数据绑定模型、一套用于打印以及管理打印队列的新类，以及用于显示大量格式化文本的文档特性。甚至提供了用于构建基于页面的应用程序的模型，这种应用程序可在 Internet Explorer 中流畅运行，

并能从 Web 站点启动，所有这些操作都不会出现常见的安全警告和令人讨厌的安装提示。总之，WPF 将以前 Windows 开发领域中的精华与当今的创新技术融为一体，得以构建现代化的富图形用户界面。

关于本书

本书深刻地介绍 WPF 技术，面向了解.NET 平台、C#语言以及 Visual Studio 开发环境的专业开发人员。在学习本书前，不需要具备使用以前版本 WPF 的经验，而使用过 WPF 的开发人员可以通过阅读每章开头的“新增功能”来了解 WPF 新特性。

本书全面描述 WPF 的所有主要特性，从 XAML(用于定义 WPF 用户界面的标记语言)到 3D 绘图和动画。本书个别之处会编写涉及.NET Framework 其他特性的代码，如用于查询数据库的 ADO.NET 类。本书中不讨论这些内容。但如果需要了解有关.NET(而非特定于 WPF)的特性的更多信息，请参阅 Apress 出版的许多专门介绍.NET 的书籍。

内容概览

本书共包括 33 章。如果刚开始学习 WPF，将发现按照章节顺序阅读本书是最容易的方法，因为后续章节常用到前面章节中演示的技术。

下面是本书每一章的主要内容：

第 1 章：WPF 概述 介绍 WPF 的体系结构，WPF 的 DirectX 基础设施，以及新的能自动改变用户界面尺寸的设备无关度量系统。

第 2 章：XAML 介绍用于定义用户界面的 XAML 标准。该章将讨论为什么创建 XAML 以及 XAML 的工作原理，并将用不同的编码方法创建基本的 WPF 窗口。

第 3 章：布局 深入研究在 WPF 窗口中用于组织元素的布局面板。该章将分析不同布局策略，并将构建一些普通类型的窗口。

第 4 章：依赖项属性 介绍 WPF 如何使用依赖项属性来支持重要特性，如数据绑定和动画。

第 5 章：路由事件 介绍 WPF 如何使用事件路由在用户界面元素中发送冒泡路由事件或隧道路由事件，还介绍所有 WPF 元素都支持的一组基本鼠标、键盘以及多点触控事件。

第 6 章：控件 分析所有 Windows 开发人员都十分熟悉的控件，如按钮、文本框和标签，还讨论它们在 WPF 中的区别。

第 7 章：Application 类 介绍 WPF 应用程序模型。在该章中您将看到如何创建单实例和基于文档的 WPF 应用程序。

第 8 章：元素绑定 介绍 WPF 数据绑定。在该章中您将看到如何将任意类型的对象绑定到用户界面。

第 9 章：命令 介绍 WPF 命令模型，使用 WPF 命令模型可将多个控件连接到同一个逻辑操作。

第 10 章：资源 介绍如何使用资源在程序集中嵌入二进制文件，以及如何在整个用户界面中重用重要的对象。

第 11 章：样式和行为 解释 WPF 样式系统，使用 WPF 样式可为一整组控件应用一套通用属性值。

第 12 章：形状、画刷和变换 介绍 WPF 中的 2D 绘图模型。在该章中您将学习如何创建形状、使用变换改变元素，以及使用渐变画刷、图像画刷和平铺图像画刷绘制特殊效果。

第 13 章：几何形状和图画 深入分析 2D 绘图。在该章中您将学习如何创建包含弧线和曲线的复杂路径，以及如何高效地使用复杂图形。

第 14 章：效果和可视化对象 介绍低级图形编程。在该章中您将使用像素着色器应用 Photoshop 风格的效果，手动构建位图，并为了优化绘图性能而使用 WPF 的可视化层。

第 15 章：动画基础 研究 WPF 的动画框架，通过 WPF 动画框架可使用简单的声明式标记将动态效果集成到应用程序中。

第 16 章：高级动画 研究更高级的动画技术，如关键帧动画、基于路径的动画以及基于帧的动画。该章还将列举一个详细示例，展示如何使用代码创建和管理动态的动画。

第 17 章：控件模板 介绍如何通过插入定制的模板来为任意 WPF 控件提供动态的新外观(以及新行为)，您还将看到如何使用模板构建能够换肤的应用程序。

第 18 章：自定义元素 研究如何扩展现有的 WPF 控件，以及如何创建自己的控件。在该章中您将看到几个示例，包括基于模板的颜色拾取器、可翻转的面板、自定义的布局容器，以及执行自定义绘图的装饰元素。

第 19 章：数据绑定 展示如何从数据库获取信息，将获取的信息插入到自定义的数据对象中，并将这些数据对象绑定到 WPF 控件。您还将学习如何借助虚拟化技术提高大型数据绑定列表的性能，以及如何使用验证方法捕获编辑错误。

第 20 章：格式化绑定的数据 展示将原始数据转换为包含图片、控件以及选择效果的富数据显示的一些技巧。

第 21 章：数据视图 分析如何在数据绑定窗口中使用视图在数据项列表中导航，以及应用过滤、分类和分组。

第 22 章：列表、树和网格 带您浏览 WPF 中的富数据控件，包括 ListView、TreeView 和 DataGrid。

第 23 章：窗口 分析 WPF 中窗口的工作原理。在该章中您还将学习如何创建不规则形状的窗口，以及如何使用 Vista 玻璃效果，您还将通过定制任务栏跳转列表、缩略图以及图标重叠实现大部分 Windows 7 特性。

第 24 章：页面和导航 介绍如何使用 WPF 构建页面，以及保持跟踪导航历史。该章还将介绍如何构建驻留于浏览器中的 WPF 应用程序，这种应用程序可从 Web 站点启动。

第 25 章：菜单、工具栏和功能区 分析面向命令的控件，如菜单和工具栏。在该章还将使用可免费下载的 Ribbon 控件尝试更富有现代气息的用户界面。

第 26 章：声音和视频 介绍 WPF 媒体支持。在该章中您将看到如何控制声音和视频的播放，以及如何合成功动画和生动鲜活的效果。

第 27 章：3D 绘图 研究 WPF 中对绘制 3D 图形的支持。在该章将学习如何创建和变换 3D 对象，以及如何为 3D 对象应用动画效果，甚至还会看到如何在 3D 表面上放置可交互的 2D 控件。

第 28 章：文档 介绍 WPF 的富文档支持。在该章中您将学习如何使用流文档以尽可能

便于阅读的方式呈现大量文本，并将学习如何用固定文档显示准备打印的页面，甚至还将学习如何使用 RichTextBox 控件提供文档编辑功能。

第 29 章：打印 演示 WPF 的打印模型，可通过打印模型在打印文档中绘制文本和图形。在该章中您还将学习如何管理页面设置和打印队列。

第 30 章：与 Windows 窗体进行交互 分析如何在同一个应用程序——甚至在同一个窗口中，结合使用 WPF 和 Windows 窗体内容。

第 31 章：多线程 介绍如何创建具有良好响应能力，在后台执行耗时任务的 WPF 应用程序。

第 32 章：插件模型 展示如何创建可扩展的、能动态发现和加载独立组件的应用程序。

第 33 章：ClickOnce 部署 展示如何使用 ClickOnce 安装模型部署 WPF 应用程序。

使用本书的前提条件

为运行 WPF 4.5 应用程序，计算机必须安装 Windows 7、Windows 8 或带有 Service Pack 2 的 Windows Vista，还需要.NET Framework 4.5。为创建 WPF 4.5 应用程序(并打开本书中提供的示例项目)，需要安装 Visual Studio 2012，Visual Studio 2012 中包含了.NET Framework 4.5。

还有一种选择。不使用任何版本的 Visual Studio，可使用 Expression Blend(一种面向图形的设计工具)来构建和测试 WPF 应用程序。总体而言，Expression Blend 是面向图形设计人员的工具，他们使用该工具创建绚丽夺目的内容；而对于编写大量代码的编程人员来说，Visual Studio 则是理想工具。本书假定使用的是 Visual Studio。如果准备使用 Expression Blend，务必选用明确支持 WPF 的版本(与某些 Visual Studio 版本绑定在一起的版本仅用于 Metro 开发，不支持 WPF)。到撰写本书时为止，支持 WPF 的 Expression Blend 版本是称为 Blend + Sketchflow Preview for Visual Studio 2012 的预览版本，网址是 <http://tinyurl.com/cgar5lz>。

代码示例和 URL

查看 Apress 网站或 www.prosetech.com 以下载最新的示例代码是个好主意。测试在本书中介绍的大部分更复杂的示例需要用到这些代码示例，因为在本书示例中那些较次要的细节通常被忽略了。本书关注最重要的部分，以免为阐明概念而无谓地占用过多篇幅。

为下载本书的源代码，可访问 Web 站点 <http://www.prosetech.com> 查找本书的页面，也可从 <http://www.tupwk.com.cn/downpage> 下载本书的源代码。您将发现在本书中提及的链接的列表，从而不需要键入任何内容就可以找到重要工具和例子。

反馈

本书力争成为 WPF 编程爱好者的最佳辅导和参考资料。为达到该目标，您的评论和建议对我们来说是非常有帮助的。您可将本书的缺点、优点及其他反馈信息直接发送到邮箱 wkservice@vip.163.com，我们将不胜感激。

目 录

第 I 部分 基础知识	
第 1 章 WPF 概述	3
1.1 Windows 图形演化	3
1.1.1 DirectX: 新的图形引擎	4
1.1.2 硬件加速与 WPF	4
1.2 WPF: 高级 API	4
1.3 分辨率无关性	5
1.3.1 WPF 单位	6
1.3.2 系统 DPI	7
1.3.3 位图和矢量图形	9
1.4 WPF 体系结构	10
1.5 WPF 4.5	13
1.5.1 WPF 工具包	14
1.5.2 Visual Studio 2012	14
1.6 小结	16
第 2 章 XAML	17
2.1 理解 XAML	17
2.1.1 WPF 之前的图形用户界面	17
2.1.2 XAML 变体	19
2.1.3 XAML 编译	19
2.2 XAML 基础	20
2.2.1 XAML 名称空间	21
2.2.2 代码隐藏类	22
2.3 XAML 中的属性和事件	24
2.3.1 简单属性与类型转换器	25
2.3.2 复杂属性	26
2.3.3 标记扩展	28
2.3.4 附加属性	29
2.3.5 嵌套元素	30
2.3.6 特殊字符与空白	32
2.3.7 事件	34
2.3.8 完整的 Eight Ball Answer 示例	35
2.4 使用其他名称空间中的类型	36
2.5 加载和编译 XAML	38
2.5.1 只使用代码	38
2.5.2 使用代码和未经编译的 XAML	40
2.5.3 使用代码和编译过的 XAML	42
2.5.4 只使用 XAML	44
2.6 小结	45
第 3 章 布局	47
3.1 理解 WPF 中的布局	47
3.1.1 WPF 布局原则	47
3.1.2 布局过程	48
3.1.3 布局容器	48
3.2 使用 StackPanel 面板进行简单布局	50
3.2.1 布局属性	52
3.2.2 对齐方式	52
3.2.3 边距	53
3.2.4 最小尺寸、最大尺寸以及显式地设置尺寸	54
3.2.5 Border 控件	56
3.3 WrapPanel 和 DockPanel 面板	57
3.3.1 WrapPanel 面板	57
3.3.2 DockPanel 面板	58
3.3.3 嵌套布局容器	59
3.4 Grid 面板	60
3.4.1 调整行和列	63
3.4.2 布局舍入	64
3.4.3 跨越行和列	65
3.4.4 分割窗口	66
3.4.5 共享尺寸组	69
3.4.6 UniformGrid 面板	72

3.5 使用 Canvas 面板进行基于坐标的布局	72	5.4.2 焦点	111
3.5.1 Z 顺序	73	5.4.3 获取键盘状态	112
3.5.2 InkCanvas 元素	74	5.5 鼠标输入	113
3.6 布局示例	76	5.5.1 鼠标单击	115
3.6.1 列设置	76	5.5.2 捕获鼠标	115
3.6.2 动态内容	77	5.5.3 鼠标拖放	116
3.6.3 组合式用户界面	79	5.6 多点触控输入	118
3.7 小结	80	5.6.1 多点触控的输入层次	119
第 4 章 依赖项属性	83	5.6.2 原始触控	119
4.1 理解依赖项属性	83	5.6.3 操作	122
4.1.1 定义依赖项属性	84	5.6.4 惯性	124
4.1.2 注册依赖项属性	84	5.7 小结	125
4.1.3 添加属性包装器	86		
4.1.4 WPF 使用依赖项属性的方式	87		
4.1.5 共享的依赖项属性	88		
4.1.6 附加的依赖项属性	88		
4.2 属性验证	90		
4.2.1 验证回调	90		
4.2.2 强制回调	91		
4.3 小结	93		
第 5 章 路由事件	95		
5.1 理解路由事件	95		
5.1.1 定义、注册和封装路由事件	95		
5.1.2 共享路由事件	96		
5.1.3 引发路由事件	96		
5.1.4 处理路由事件	97		
5.2 事件路由	99		
5.2.1 RoutedEventArgs 类	100		
5.2.2 冒泡路由事件	100		
5.2.3 处理挂起的事件	103		
5.2.4 附加事件	103		
5.2.5 隧道路由事件	105		
5.3 WPF 事件	106		
5.3.1 生命周期事件	106		
5.3.2 输入事件	108		
5.4 键盘输入	108		
5.4.1 处理按键事件	109		
5.5 鼠标输入	113		
5.5.1 鼠标单击	115		
5.5.2 捕获鼠标	115		
5.5.3 鼠标拖放	116		
5.6 多点触控输入	118		
5.6.1 多点触控的输入层次	119		
5.6.2 原始触控	119		
5.6.3 操作	122		
5.6.4 惯性	124		
5.7 小结	125		
第 II 部分 进一步研究 WPF			
第 6 章 控件	129		
6.1 控件类	129		
6.1.1 背景画刷和前景画刷	130		
6.1.2 字体	132		
6.1.3 鼠标光标	136		
6.2 内容控件	137		
6.2.1 Content 属性	138		
6.2.2 对齐内容	140		
6.2.3 WPF 内容原则	141		
6.2.4 标签	142		
6.2.5 按钮	142		
6.2.6 工具提示	145		
6.3 特殊容器	151		
6.3.1 ScrollViewer	152		
6.3.2 GroupBox	154		
6.3.3 TabItem	154		
6.3.4 Expander	155		
6.4 文本控件	158		
6.4.1 多行文本	158		
6.4.2 选择文本	159		
6.4.3 拼写检查	160		
6.4.4 PasswordBox	162		
6.5 列表控件	162		
6.5.1 ListBox	163		
6.5.2 ComboBox	166		

6.6 基于范围的控件	166	8.2.1 Source 属性	212
6.6.1 Slider	167	8.2.2 RelativeSource 属性	212
6.6.2 ProgressBar	168	8.2.3 DataContext 属性	213
6.7 日期控件	169	8.3 小结	214
6.8 小结	171	第 9 章 命令	215
第 7 章 Application 类	173	9.1 理解命令	215
7.1 应用程序的生命周期	173	9.2 WPF 命令模型	216
7.1.1 创建 Application 对象	173	9.2.1 ICommand 接口	217
7.1.2 派生自定义的 Application 类	174	9.2.2 RoutedCommand 类	217
7.1.3 应用程序的关闭方式	176	9.2.3 RoutedUICommand 类	218
7.1.4 应用程序事件	177	9.2.4 命令库	219
7.2 Application 类的任务	179	9.3 执行命令	220
7.2.1 显示初始界面	179	9.3.1 命令源	220
7.2.2 处理命令行参数	180	9.3.2 命令绑定	221
7.2.3 访问当前 Application 对象	181	9.3.3 使用多命令源	223
7.2.4 在窗口之间进行交互	182	9.3.4 微调命令文本	224
7.2.5 单实例应用程序	184	9.3.5 直接调用命令	224
7.3 程序集资源	189	9.3.6 禁用命令	225
7.3.1 添加资源	189	9.3.7 具有内置命令的控件	227
7.3.2 检索资源	190	9.4 高级命令	229
7.3.3 pack URI	192	9.4.1 自定义命令	229
7.3.4 内容文件	193	9.4.2 在不同位置使用相同的命令	230
7.4 本地化	193	9.4.3 使用命令参数	232
7.4.1 构建能够本地化的用户界面	194	9.4.4 跟踪和翻转命令	233
7.4.2 使应用程序为本地化做好准备	194	9.5 小结	237
7.4.3 管理翻译过程	195	第 10 章 资源	239
7.5 小结	200	10.1 资源基础	239
第 8 章 元素绑定	201	10.1.1 资源集合	239
8.1 将元素绑定到一起	201	10.1.2 资源的层次	241
8.1.1 绑定表达式	202	10.1.3 静态资源和动态资源	242
8.1.2 绑定错误	203	10.1.4 非共享资源	243
8.1.3 绑定模式	203	10.1.5 通过代码访问资源	244
8.1.4 使用代码创建绑定	205	10.1.6 应用程序资源	244
8.1.5 使用代码检索绑定	206	10.1.7 系统资源	245
8.1.6 多绑定	207	10.2 资源字典	246
8.1.7 绑定更新	210	10.2.1 创建资源字典	246
8.1.8 绑定延迟	211	10.2.2 使用资源字典	247
8.2 绑定到非元素对象	211	10.2.3 在程序集之间共享资源	248

10.3 小结	251	12.2.6 VisualBrush 画刷	297
第 11 章 样式和行为	253	12.2.7 BitmapCacheBrush 画刷	298
11.1 样式基础.....	253	12.3 变换.....	299
11.1.1 创建样式对象.....	256	12.3.1 变换形状	300
11.1.2 设置属性.....	257	12.3.2 变换元素	302
11.1.3 关联事件处理程序	258	12.4 透明.....	303
11.1.4 多层样式	259	12.4.1 使元素半透明	303
11.1.5 通过类型自动应用样式	261	12.4.2 透明掩码	304
11.2 触发器	262	12.5 小结	306
11.2.1 简单触发器.....	262	第 13 章 几何图形和图画	307
11.2.2 事件触发器.....	264	13.1 路径和几何图形	307
11.3 行为	266	13.1.1 直线、矩形和椭圆图形	308
11.3.1 获取行为支持.....	266	13.1.2 使用 GeometryGroup	
11.3.2 理解行为模型	267	组合形状	309
11.3.3 创建行为	268	13.1.3 使用 CombinedGeometry 融合	
11.3.4 使用行为	270	几何图形	311
11.3.5 Blend 中的设计时行为支持.....	271	13.1.4 使用 PathGeometry 绘制曲线和	
11.4 小结	271	直线	313
第III部分 图画和动画		13.1.5 微语言几何图形	318
第 12 章 形状、画刷和变换	275	13.1.6 使用几何图形进行剪裁	319
12.1 理解形状	275	13.2 图画	320
12.1.1 Shape 类	276	13.2.1 显示图画	322
12.1.2 矩形和椭圆	277	13.2.2 导出插图	324
12.1.3 改变形状的尺寸和放置形状	278	13.3 小结	326
12.1.4 使用 Viewbox 控件缩放形状	280	第 14 章 效果和可视化对象	327
12.1.5 直线	282	14.1 可视化对象	327
12.1.6 折线	283	14.1.1 绘制可视化对象	328
12.1.7 多边形	284	14.1.2 在元素中封装可视化对象	329
12.1.8 直线条帽和直线交点	286	14.1.3 命中测试	332
12.1.9 点划线	287	14.1.4 复杂的命中测试	334
12.1.10 像素对齐	288	14.2 效果	338
12.2 画刷	289	14.2.1 BlurEffect 类	338
12.2.1 SolidColorBrush 画刷	290	14.2.2 DropShadowEffect 类	339
12.2.2 LinearGradientBrush 画刷	290	14.2.3 ShaderEffect 类	340
12.2.3 RadialGradientBrush 画刷	292	14.3 WriteableBitmap 类	342
12.2.4 ImageBrush 画刷	294	14.3.1 生成位图	342
12.2.5 平铺的 ImageBrush 画刷	295	14.3.2 写入 WriteableBitmap 对象	343

14.4 小结	347	16.4 基于帧的动画.....	399
第 15 章 动画基础.....	349	16.5 使用代码创建故事板.....	402
15.1 理解 WPF 动画	349	16.5.1 创建主窗口	403
15.1.1 基于时间的动画	349	16.5.2 创建 Bomb 用户控件	405
15.1.2 基于属性的动画	350	16.5.3 投弹	406
15.2 基本动画	351	16.5.4 拦截炸弹	409
15.2.1 Animation 类	351	16.5.5 统计炸弹和清理工作	410
15.2.2 使用代码创建动画	353	16.6 小结.....	412
15.2.3 同时发生的动画	358		
15.2.4 动画的生命周期	358		
15.2.5 Timeline 类	359		
15.3 故事板	362	第 IV 部分 模板和自定义元素	
15.3.1 故事板.....	363	第 17 章 控件模板	417
15.3.2 事件触发器	363	17.1 理解逻辑树和可视化树	417
15.3.3 重叠动画	366	17.2 理解模板.....	422
15.3.4 同步的动画	367	17.2.1 修饰类	424
15.3.5 控制播放	367	17.2.2 剖析控件	426
15.3.6 监视动画进度	371	17.3 创建控件模板.....	428
15.4 动画缓动	373	17.3.1 简单按钮	429
15.4.1 使用缓动函数	373	17.3.2 模板绑定	430
15.4.2 在动画开始时应用缓动与在动画 结束时应用缓动	374	17.3.3 改变属性的触发器	431
15.4.3 缓动函数类	375	17.3.4 使用动画的触发器	434
15.4.4 创建自定义缓动函数	377	17.4 组织模板资源.....	435
15.5 动画性能	379	17.4.1 分解按钮控件模板	436
15.5.1 期望的帧率	380	17.4.2 通过样式应用模板	438
15.5.2 位图缓存	382	17.4.3 自动应用模板	439
15.6 小结	384	17.4.4 由用户选择的皮肤	440
第 16 章 高级动画.....	385	17.5 构建更复杂的模板.....	442
16.1 动画类型回顾	385	17.5.1 嵌套的模板	443
16.1.1 动态变换	386	17.5.2 修改滚动条	445
16.1.2 动态改变画刷	390	17.5.3 控件模板示例	450
16.1.3 动态改变像素着色器	392	17.6 可视化状态.....	451
16.2 关键帧动画	393	17.7 小结.....	452
16.2.1 离散的关键帧动画	395		
16.2.2 缓动关键帧	395		
16.2.3 样条关键帧动画	396		
16.3 基于路径的动画	397	第 18 章 自定义元素.....	453
		18.1 理解 WPF 中的自定义元素	454
		18.2 构建基本的用户控件.....	456
		18.2.1 定义依赖项属性	456
		18.2.2 定义路由事件	459
		18.2.3 添加标记	460
		18.2.4 使用控件	462

18.2.5 命令支持 462	19.3.2 项容器再循环 519
18.2.6 深入分析用户控件 465	19.3.3 缓存长度 519
18.3 创建无外观控件 466	19.3.4 延迟滚动 520
18.3.1 修改颜色拾取器的代码 466	19.4 验证 521
18.3.2 修改颜色拾取器的标记 467	19.4.1 在数据对象中进行验证 521
18.3.3 精简控件模板 469	19.4.2 自定义验证规则 526
18.4 支持可视化状态 472	19.4.3 响应验证错误 528
18.4.1 开始编写 FlipPanel 类 473	19.4.4 获取错误列表 529
18.4.2 选择部件和状态 475	19.4.5 显示不同的错误指示符号 530
18.4.3 默认控件模板 476	19.4.6 验证多个值 533
18.4.4 使用 FlipPanel 控件 482	19.5 数据提供者 535
18.4.5 使用不同的控件模板 483	19.5.1 ObjectDataProvider 536
18.5 创建自定义面板 485	19.5.2 XmlDataProvider 539
18.5.1 两步布局过程 485	19.6 小结 541
18.5.2 Canvas 面板的副本 488	第 20 章 格式化绑定的数据 543
18.5.3 更好的 WrapPanel 面板 489	20.1 数据绑定回顾 543
18.6 自定义绘图元素 492	20.2 数据转换 544
18.6.1 OnRender()方法 493	20.2.1 使用 StringFormat 属性 545
18.6.2 评估自定义绘图 494	20.2.2 值转换器简介 546
18.6.3 自定义绘图元素 495	20.2.3 使用值转换器设置字符串
18.6.4 创建自定义装饰元素 497	的格式 547
18.7 小结 498	20.2.4 使用值转换器创建对象 549
第 V 部分 数据	20.2.5 应用条件格式化 551
第 19 章 数据绑定 501	20.2.6 评估多个属性 552
19.1 使用自定义对象绑定到	20.3 列表控件 554
数据库 501	20.4 列表样式 555
19.1.1 构建数据访问组件 502	20.4.1 ItemContainerStyle 556
19.1.2 构建数据对象 504	20.4.2 包含复选框或单选按钮的 ListBox
19.1.3 显示绑定对象 505	控件 557
19.1.4 更新数据库 508	20.4.3 交替条目样式 559
19.1.5 更改通知 508	20.4.4 样式选择器 561
19.2 绑定到对象集合 510	20.5 数据模板 564
19.2.1 显示和编辑集合项 510	20.5.1 分离和重用模板 566
19.2.2 插入和移除集合项 513	20.5.2 使用更高级的模板 567
19.2.3 绑定到 ADO.NET 对象 514	20.5.3 改变模板 569
19.2.4 绑定到 LINQ 表达式 515	20.5.4 模板选择器 570
19.3 提高大列表的性能 518	20.5.5 模板与选择 573
19.3.1 虚拟化 518	20.5.6 改变项的布局 577

	第 VI 窗口、页面和富控件
	第 23 章 窗口 639
20.6 ComboBox 控件 578	23.1 Window 类 639
20.7 小结 581	23.1.1 显示窗口 641
第 21 章 数据视图 583	23.1.2 定位窗口 642
21.1 View 对象 583	23.1.3 保存和还原窗口位置 642
21.1.1 检索视图对象 584	23.2 窗口交互 644
21.1.2 视图导航 584	23.2.1 窗口所有权 646
21.1.3 以声明方式创建视图 587	23.2.2 对话框模型 647
21.2 过滤、排序与分组 588	23.2.3 通用对话框 648
21.2.1 过滤集合 588	23.3 非矩形窗口 649
21.2.2 过滤 DataTable 对象 591	23.3.1 简单形状窗口 649
21.2.3 排序 592	23.3.2 具有形状内容的透明窗口 651
21.2.4 分组 593	23.3.3 移动形状窗口 653
21.2.5 实时成型 598	23.3.4 改变形状窗口的尺寸 653
21.3 小结 599	23.3.5 组合到一起：窗口的自定义控件 模板 655
第 22 章 列表、树和网格 601	23.4 Windows 7 任务栏编程 658
22.1 ListView 控件 601	23.4.1 使用跳转列表 659
22.1.1 使用 GirdView 创建列 602	23.4.2 改变任务栏图标和预览 663
22.1.2 创建自定义视图 606	23.5 小结 667
22.2 TreeView 控件 613	第 24 章 页面和导航 669
22.2.1 创建数据绑定的 TreeView 控件 614	24.1 基于页面的导航 669
22.2.2 将 DataSet 对象绑定到 TreeView 控件 617	24.2 基于页面的界面 670
22.2.3 即时创建节点 618	24.2.1 创建一个具有导航窗口的基子 页面的简单应用程序 670
22.3 DataGridView 控件 621	24.2.2 Page 类 672
22.3.1 改变列的尺寸与重新安排列 622	24.2.3 超链接 673
22.3.2 定义列 623	24.2.4 在框架中驻留页面 675
22.3.3 设置列的格式和样式 628	24.2.5 在另一个页面中驻留页面 677
22.3.4 设置行的格式 629	24.2.6 在 Web 浏览器中驻留页面 678
22.3.5 显示行细节 630	24.3 页面历史 678
22.3.6 冻结列 631	24.3.1 深入分析 WPF 中的 URI 678
22.3.7 选择 632	24.3.2 导航历史 679
22.3.8 排序 632	24.3.3 维护自定义的属性 681
22.3.9 编辑 633	24.4 导航服务 682
22.4 小结 635	24.4.1 通过编程进行导航 682
	24.4.2 导航事件 683

24.4.3 管理日志 684 24.4.4 向日志添加自定义项 685 24.4.5 使用页函数 689 24.5 XAML 浏览器应用程序 692 24.5.1 创建 XBAP 应用程序 692 24.5.2 部署 XBAP 应用程序 693 24.5.3 更新 XBAP 应用程序 695 24.5.4 XBAP 应用程序的安全性 695 24.5.5 完全信任的 XBAP 应用程序 696 24.5.6 组合 XBAP/独立应用程序 697 24.5.7 为不同的安全级别编写代码 697 24.5.8 在网页中嵌入 XBAP 应用程序 702 24.6 WebBrowser 控件 702 24.6.1 导航到页面 703 24.6.2 构建 DOM 树 704 24.6.3 使用.NET 代码为网页添加 脚本 706 24.7 小结 708 第 25 章 菜单、工具栏和功能区 709 25.1 菜单 709 25.1.1 Menu 类 710 25.1.2 菜单项 710 25.1.3 ContextMenu 类 712 25.1.4 菜单分隔条 713 25.2 工具栏和状态栏 714 25.2.1ToolBar 控件 714 25.2.2 StatusBar 控件 717 25.3 功能区 718 25.3.1 添加功能区 719 25.3.2 应用程序菜单 720 25.3.3 选项卡、组与按钮 722 25.3.4 富工具提示 724 25.3.5 带有快捷键提示的键盘访问 725 25.3.6 改变功能区的大小 726 25.3.7 快速访问工具栏 729 25.4 小结 730	第 26 章 声音和视频 731 26.1 播放 WAV 音频 731 26.1.1 SoundPlayer 类 731 26.1.2 SoundPlayerAction 类 733 26.1.3 系统声音 733 26.2 MediaPlayer 类 734 26.3 MediaElement 类 735 26.3.1 使用代码播放音频 736 26.3.2 处理错误 737 26.3.3 使用触发器播放音频 737 26.3.4 播放多个声音 739 26.3.5 改变音量、平衡、速度以及 位置 740 26.3.6 将动画同步到音频 742 26.3.7 播放视频 744 26.3.8 视频效果 744 26.4 语音 747 26.4.1 语音合成 747 26.4.2 语音识别 749 26.5 小结 751 第 27 章 3D 绘图 753 27.1 3D 绘图基础 753 27.1.1 视口 754 27.1.2 3D 对象 754 27.1.3 摄像机 761 27.2 深入研究 3D 绘图 765 27.2.1 着色和法线 766 27.2.2 更复杂的形状 769 27.2.3 Model3DGroup 集合 769 27.2.4 使用材质 771 27.2.5 纹理映射 773 27.3 交互和动画 777 27.3.1 变换 777 27.3.2 旋转 778 27.3.3 飞过 779 27.3.4 跟踪球 781 27.3.5 命中测试 782 27.3.6 3D 表面上的 2D 元素 786
---	--

<p>27.4 小结 789</p> <p>第VII部分 文档和打印</p> <p>第 28 章 文档 793</p> <p>28.1 理解文档 793</p> <p>28.2 流文档 794</p> <p> 28.2.1 流内容元素 795</p> <p> 28.2.2 设置内容元素的格式 796</p> <p> 28.2.3 创建简单的流文档 797</p> <p> 28.2.4 块元素 799</p> <p> 28.2.5 内联元素 804</p> <p> 28.2.6 通过代码与元素进行交互 809</p> <p> 28.2.7 文本对齐 812</p> <p>28.3 只读流文档容器 813</p> <p> 28.3.1 缩放 814</p> <p> 28.3.2 创建页面和列 815</p> <p> 28.3.3 从文件加载文档 817</p> <p> 28.3.4 打印 818</p> <p>28.4 编辑流文档 818</p> <p> 28.4.1 加载文件 819</p> <p> 28.4.2 保存文件 821</p> <p> 28.4.3 设置所选文本的格式 822</p> <p> 28.4.4 获取单个单词 824</p> <p>28.5 固定文档 825</p> <p>28.6 批注 826</p> <p> 28.6.1 批注类 827</p> <p> 28.6.2 启用批注服务 828</p> <p> 28.6.3 创建批注 829</p> <p> 28.6.4 检查批注 832</p> <p> 28.6.5 响应批注更改 835</p> <p> 28.6.6 在固定文档中保存批注 835</p> <p> 28.6.7 自定义便签的外观 836</p> <p>28.7 小结 837</p> <p>第 29 章 打印 839</p> <p>29.1 基本打印 839</p> <p> 29.1.1 打印元素 840</p> <p> 29.1.2 变换打印输出 842</p> <p> 29.1.3 打印不显示的元素 844</p>	<p>29.1.4 打印文档 845</p> <p>29.1.5 在文档打印输出中控制页面 848</p> <p>29.2 自定义打印 851</p> <p> 29.2.1 使用可视化层中的类 进行打印 851</p> <p> 29.2.2 自定义多页打印 854</p> <p>29.3 打印设置和管理 859</p> <p> 29.3.1 保存打印设置 859</p> <p> 29.3.2 打印页面范围 859</p> <p> 29.3.3 管理打印队列 860</p> <p>29.4 通过 XPS 进行打印 863</p> <p> 29.4.1 为打印预览创建 XPS 文档 863</p> <p> 29.4.2 写入内存的 XPS 文档 864</p> <p> 29.4.3 通过 XPS 直接打印到 打印机 865</p> <p> 29.4.4 异步打印 866</p> <p>29.5 小结 866</p> <p>第VIII部分 其他主题</p> <p>第 30 章 与 Windows 窗体进行交互 869</p> <p>30.1 访问互操作性 869</p> <p>30.2 混合窗口和窗体 870</p> <p> 30.2.1 为 WPF 应用程序添加窗体 870</p> <p> 30.2.2 为 Windows 窗体应用程序 添加 WPF 窗口 870</p> <p> 30.2.3 显示模态窗口和窗体 871</p> <p> 30.2.4 显示非模态窗口和窗体 871</p> <p> 30.2.5 启用 Windows 窗体控件的 可视化风格 872</p> <p>30.3 创建具有混合内容的窗口 872</p> <p> 30.3.1 WPF 和 Windows 窗体 “空域” 873</p> <p> 30.3.2 在 WPF 中驻留 Windows 窗体控件 874</p> <p> 30.3.3 使用 WPF 和 Windows 窗体 用户控件 876</p> <p> 30.3.4 在 Windows 窗体中驻留 WPF 控件 877</p> <p> 30.3.5 访问键、助记码和焦点 879</p>
--	--

30.3.6 属性映射	880	32.3.4 插件适配器	902
30.4 小结	882	32.3.5 宿主视图	903
第 31 章 多线程	883	32.3.6 宿主适配器	904
31.1 了解多线程模型	883	32.3.7 宿主	904
31.1.1 Dispatcher 类	884	32.3.8 更多插件	907
31.1.2 DispatcherObject 类	884	32.4 与宿主进行交互	908
31.2 BackgroundWorker 类	887	32.5 可视化插件	912
31.2.1 简单的异步操作	887	32.6 小结	915
31.2.2 创建 BackgroundWorker 对象	888	第 33 章 ClickOnce 部署	917
31.2.3 运行 BackgroundWorker 对象	889	33.1 理解应用程序部署	917
31.2.4 跟踪进度	891	33.1.1 ClickOnce 安装模型	918
31.2.5 支持取消	893	33.1.2 ClickOnce 部署的局限性	919
31.3 小结	894	33.2 简单的 ClickOnce 发布	920
第 32 章 插件模型	895	33.2.1 设置发布者和产品	920
32.1 在 MAF 和 MEF 两者间 进行选择	895	33.2.2 启动发布向导	922
32.2 了解插件管道	896	33.2.3 理解部署文件的结构	926
32.2.1 管道的工作原理	897	33.2.4 安装 ClickOnce 应用程序	926
32.2.2 插件文件夹结构	898	33.2.5 更新 ClickOnce 应用程序	928
32.2.3 为使用插件模型准备 解决方案	899	33.3 ClickOnce 附加选项	928
32.3 创建使用插件模型的 应用程序	900	33.3.1 发布版本	928
32.3.1 协定	901	33.3.2 更新	929
32.3.2 插件视图	901	33.3.3 文件关联	930
32.3.3 插件	902	33.3.4 发布选项	931

第 I 部分



基础 知识

第 1 章 WPF 概述

第 2 章 XAML

第 3 章 布局

第 4 章 依赖项属性

第 5 章 路由事件

第 1 章



WPF 概述

WPF(Windows Presentation Foundation)是用于 Windows 的现代图形显示系统。与之前出现的其他技术相比，WPF 发生了根本性变化，引入了“内置硬件加速”和“分辨率无关”等创新功能；本章将介绍这两项功能。

如要构建运行在 Windows Vista、Windows 7 和 Windows 8 桌面模式(以及对应的 Windows Server 版本)下的富桌面应用程序，WPF 无疑是最适用的工具包。事实上，WPF 是针对这些 Windows 版本的唯一通用工具包。比较起来，Microsoft 新推出的 Metro 工具包虽然令人感到激动，但 Metro 的使用范围仅限于 Windows 8 系统。WPF 的应用范围却广泛得多，它甚至可运行在仍在很多企业中使用的已经过时的 Windows XP 计算机上；唯一的局限性在于您必须对 Visual Studio 进行配置，使其将较为陈旧的.NET 4.0 Framework(而非.NET 4.5)作为目标。

本章将首先介绍 WPF 的体系结构，然后讨论 WPF 如何处理可变屏幕分辨率，将概述 WPF 的核心程序集和类，并将介绍 WPF 如何从初始版本演变为 WPF 4.5。

1.1 Windows 图形演化

在 WPF 问世之前的近 15 个年头，Windows 开发人员一直在使用本质上相同的显示技术。究其原因，是由于此前的每个传统 Windows 应用程序都依靠 Windows 操作系统的如下两个由来已久的部分来创建用户界面：

- **User32：**该部分为许多元素(如窗口、按钮和文本框等)提供了熟悉的 Windows 外观。
- **GDI/GDI+：**该部分为渲染简单形状、文本以及图像提供了绘图支持，但增加了复杂程度(而且通常性能较差)。

历经多年发展，这两种技术都得到了改进，而且开发人员使用的与其交互的 API 也已发生了巨大变化。但在构建应用程序时，不管使用.NET 和 Windows 窗体，还是使用过去的 Visual Basic 6 或基于 MFC 的 C++代码，底层都是使用 Windows 操作系统的相同部分来工作的。不同框架工具只是为与 User32 和 GDI/GDI+进行交互提供了不同的封装器而已。这些框架工具能提高效率，降低复杂性，并提供了更多预置特性，从而使开发人员不必再自行编写底层代码，但这些框架工具不可能消除在 10 多年前设计的系统组件的基本限制。

注意：

在超过 15 年前，在 Windows 3.0 中完备地建立了 User32 和 GDI/GDI+的基本分工。当然，User32 在那时简化了用户操作，因为那时软件尚未进入 32 位的世界。

1.1.1 DirectX: 新的图形引擎

Microsoft 曾针对 User32 和 GDI/GDI+库的限制提供了一个解决方案: DirectX。DirectX 起初是一个易于出错的组合性质的工具包, 用于在 Windows 平台上开发游戏。DirectX 在设计上关注的重点是速度, 为此, Microsoft 和显卡供应商密切合作, 以便为 DirectX 提供复杂的纹理映射、特殊效果(如半透明)以及三维图形所需的硬件加速功能。

在首次发布 DirectX(在 Windows 95 发布后不久发布)后历经数年的发展, DirectX 已趋成熟。现在的 DirectX 已成为 Windows 的基本组成部分, 可支持所有现代显卡。然而, DirectX 编程 API 一直未背离其设计初衷, 仍主要作为游戏开发人员的工具包。因为 DirectX 固有的复杂性, 它几乎从未用于开发传统类型的 Windows 应用程序(如商业软件)。

WPF 彻底扭转了这种局面。在 WPF 中, 底层的图形技术不再是 GDI/GDI+, 而是 DirectX。事实上, 不管创建哪种用户界面, WPF 应用程序在底层都是使用 DirectX。这意味着, 无论设计复杂的三维图形(这是 DirectX 的特长), 还是仅绘制几个按钮和纯文本, 所有绘图工作都是通过 DirectX 管线完成的。因此, 即使是最普通的商业应用程序也能使用丰富的效果, 如半透明和反锯齿。在硬件加速方面也带来了好处, DirectX 在渲染图形时会将尽可能多的工作递交给图形处理单元(GPU)去处理, GPU 是显卡专用的处理器。

注意:

因为 DirectX 能理解可由显卡直接渲染的高层元素, 如纹理和渐变, 所以 DirectX 效率更高。而 GDI/GDI+不理解这些高层元素, 因此必须将它们转换成逐像素指令, 而通过现代显卡渲染这些指令更慢。

不过, 仍有一个 User32 组件得以保留, 该组件只用于有限的范围。因为对于特定服务, WPF 仍依赖于 User32, 如处理和路由输入信息以及区分哪个应用程序实际拥有屏幕的哪一部分。但所有绘图操作都是由 DirectX 完成的。

1.1.2 硬件加速与 WPF

显卡在支持特定渲染特性和优化方面是有区别的。令人感到庆幸的是, 这并不是什么问题, 原因有两点。首先, 当今大多数计算机配备的显卡硬件都足以支持 3D 绘图和动画等 WPF 功能。即使是使用集成图形处理器(图形处理器集成到主板中, 而非独立的卡)的便携式电脑和桌面计算机也同样如此。其次, WPF 为要完成的所有工作都预备了软件处理方式。这意味着, WPF 的智能程度足够高, 会尽量采用硬件优化方式, 但如有必要, 它也可采用软件计算方式来完成同样的工作。因此, 如果在配备旧式显卡的计算机上运行 WPF 应用程序, 界面仍将按其设计方式显示。当然, 采用软件计算方式时, 速度自然会慢很多, 而且配备旧式显卡的计算机不能十分顺畅地运行富 WPF 应用程序。如果富 WPF 应用程序包含复杂动画或其他密集图形效果, 这表现得尤为明显。

1.2 WPF: 高级 API

如果 WPF 仅通过 DirectX 提供硬件加速功能, 那么它只能算是一项重要改进, 而不是革命性的变化。实际上, WPF 包含了一整套面向应用程序编程人员的高级服务。

下面列出 WPF 引入到 Windows 编程领域中的一些最重要变化：

- **类似 Web 的布局模型。**与通过特定的坐标将控件固定在具体位置不同，WPF 十分注重灵活的流式布局，根据控件的内容灵活地排列控件，从而使用户界面能适应变化幅度大的内容以及不同的语言。
- **丰富的绘图模型。**与逐像素进行绘制不同，在 WPF 中可直接处理图元——基本形状、文本块以及其他图形元素。也可使用其他新特性，如真正的透明控件、放置多层并具有不同透明度内容的功能以及本地 3D 支持。
- **丰富的文本模型。**WPF 为 Windows 应用程序提供了在用户界面的任何位置显示丰富的样式化文本的功能。甚至可将文本和列表、浮动的图形以及其他用户界面元素结合起来。并且如果需要显示大量文本，还可使用高级的文档显示特性，例如换行、分列和对齐，以提高可读性。
- **作为首要编程概念的动画。**在 WPF 中，不必再用计时器来强制窗体重绘自身。与此相反，动画成为 WPF 框架的固有部分。在 WPF 中可使用声明式标签定义动画，WPF 会自动让它们运动起来。
- **支持音频和视频媒体。**以前的用户界面开发工具包(如 Windows 窗体)对多媒体的处理有很大的限制。但 WPF 支持播放任何 Windows 媒体播放器所支持的音频和视频文件，并允许同时播放多个媒体文件。更引人注目的是，WPF 提供了允许在用户界面的其他部分集成视频内容的工具，还允许添加特效技巧，比如在一个旋转的 3D 立方体上放置视频窗口。
- **样式和模板。**通过样式可实现显示格式的标准化，并可在整个应用程序中反复使用。通过模板可改变元素的渲染方式，甚至改变核心控件(如按钮)的渲染方式。在创建现代的具有皮肤的用户界面时，从来都不像现在这样方便。
- **命令。**大多数用户已认识到，通过菜单或工具栏触发 Open 命令并没什么区别，最终结果是相同的。现在通过代码抽象，可在特定位置定义应用程序命令并将其链接到多个控件上。
- **声明式用户界面。**尽管可编写代码来创建 WPF 窗口，但 Visual Studio 提供了另一种方式。它将每个窗口的内容串行化到 XAML 文档中的一组 XML 标签中。其优点是用户界面和代码完全分离，并且图形设计人员可使用专业工具编辑 XAML 文件，并最终润色应用程序的前端界面。XAML 是 Extensible Application Markup Language(可扩展应用程序标记语言)的缩写，第 2 章将详细介绍 XAML 的相关内容。
- **基于页面的应用程序。**可使用 WPF 创建类似于浏览器的应用程序，此类应用程序可通过“前进”和“后退”导航按钮在一组页面中移动。由 WPF 来处理那些纷繁的细节，如页面历史。甚至可将项目部署为运行于 IE 中的基于浏览器的应用程序。

1.3 分辨率无关性

传统的 Windows 应用程序都会受特定的假定屏幕分辨率的限制。在设计窗口时，开发人员通常假定标准的显示器分辨率(如 1366×768 像素)，并针对更小或更大的分辨率尽量保证窗口能够合理地改变尺寸。

问题是传统 Windows 应用程序的用户界面是不可伸缩的。因此，如果使用更高的显示器分

分辨率，将会更紧密地排列像素，应用程序窗口将变得更小并更难以阅读。特别是对于使用像素排列更加紧密的新式显示器，当以较高分辨率运行时，问题更趋严重。例如，通常可发现用户使用的某些显示器(特别是便携式电脑的显示器)的像素排列密度是 120 dpi(dot per inch, 每英寸像素点数)或 144 dpi，超过更常见的 96 dpi。当这些显示器使用它们默认的分辨率时，像素会以更紧密的方式显示，使控件和文本变得更小。

理想情况下，应用程序应使用更高的像素密度显示更多细节。例如，高分辨率显示器可显示相同大小的工具栏图标，但使用更多像素显示更清晰的图形。这样可保持相同的基本布局，但增加了清晰度和细节。出于多种原因，这种解决方法在过去是无法实现的。尽管可改变用 GDI/GDI+绘制的图形内容的大小，但 User32(负责为通用控件生成可视化外观)不支持真正的缩放。

这个问题在 WPF 中不复存在，因为 WPF 自行渲染所有用户界面元素，从简单的形状到通用控件(如按钮)。所以，如果在计算机显示器上创建一个 1 英寸宽的按钮，在更高分辨率的显示器上它仍能保持 1 英寸的宽度——WPF 只是使用更多像素更详细地渲染这个按钮罢了。

这里做了总体性描述，并通过几个细节进行了解释。最重要的是要认识到 WPF 根据系统 DPI 设置进行缩放，并不根据物理显示设备的 DPI 进行缩放。这是十分合理的——毕竟在 100 英寸的投影仪上显示应用程序，您可能会站在投影仪后面几步远的地方，并希望看到特大版本的窗口。不希望 WPF 骤然间将应用程序缩至“正常”大小。同样，如果使用具有更高分辨率显示器的便携式电脑，您可能希望窗口稍小些——这是在更小屏幕上显示信息必须付出的代价。更进一步讲，不同用户有不同的偏好。有些用户可能希望显示更丰富的细节，而另一些用户可能希望显示更多内容。

那么 WPF 如何确定应用程序窗口的大小呢？简单来讲，就是当 WPF 计算窗口尺寸时使用系统 DPI 设置。但要想理解底层工作原理，进一步探讨 WPF 度量系统是很有帮助的。

1.3.1 WPF 单位

WPF 窗口以及其中的所有元素都使用与设备无关的单位进行度量。一个与设备无关的单位被定义为 1/96 英寸。为了理解其实际含义，下面将分析一个例子。

设想用 WPF 创建一个尺寸为 96×96 单位的小按钮。如果使用标准的 Windows DPI 设置(96 dpi)，每个设备无关单位实际上对应一个物理像素。因为对于这种情况，WPF 用以下公式进行计算：

$$\begin{aligned} [\text{物理单位尺寸}] &= [\text{设备无关单位尺寸}] \times [\text{系统 DPI}] \\ &= 1/96 \text{ 英寸} \times 96 \text{ dpi} \\ &= 1 \text{ 像素} \end{aligned}$$

本质上，WPF 假定使用 96 个像素构成 1 英寸，因为这是 Windows 操作系统通过系统 DPI 设置告诉 WPF 的。但实际上依赖于显示设备。

例如，考虑一个最大分辨率为 1600×1200 像素的 19 英寸 LCD 显示器。可用勾股定理算出这个显示器的像素密度，如下所示：

$$\begin{aligned} [\text{屏幕DPI}] &= \frac{\sqrt{1600^2 + 1200^2} \text{ 像素}}{19 \text{ 英寸}} \\ &= 100 \text{ dpi} \end{aligned}$$

在这种情况下，像素密度达到 100 dpi，稍高于 Windows 假定的数值。因此在该显示器上，一个 96×96 像素的按钮将略小于 1 英寸。

另一方面，考虑分辨率为 1024×768 像素的 15 英寸 LCD 显示器。对于这种情况，像素密度降至约 85 dpi，因此 96×96 像素的按钮看起来比 1 英寸稍大。

在这两种情况下，如果减小屏幕尺寸(比如将分辨率切换到 800×600 像素)，那么按钮(以及屏幕上的其他内容)将相应放大。这是因为系统 DPI 仍使用 96 dpi。换句话说，Windows 仍假定 96 像素代表 1 英寸，尽管在更低的分辨率下像素更少。

提示：

正如您了解的，LCD 显示器被设计成在特定分辨率下的效果最佳，该分辨率称为自然分辨率(native resolution)。如果降低分辨率，显示器必须使用插值来填充额外像素(这会导致模糊)。为获得最佳显示效果，最好始终使用自然分辨率。如果希望显示出更大的窗口、按钮和文本，应考虑修改系统 DPI 设置。

1.3.2 系统 DPI

到目前为止，WPF 按钮示例和其他类型 Windows 应用程序中的其他任意用户界面元素完全相同。如果改变系统 DPI 设置，结果就不同了。在上一代 Windows 中，该特性有时称为大字体。因为那时系统 DPI 会影响系统字体的大小，但其他细节通常不变。

注意：

许多 Windows 应用程序不完全支持更高的 DPI 设置。在最糟糕的情形下，增加系统 DPI 可能会使窗口中的一些内容被缩放，但其他内容则未被缩放，这可能导致有些内容被隐藏起来，甚至窗口无法使用。

这正是 WPF 的不同之处。WPF 本身就可以十分轻松地支持系统 DPI 设置。例如，将系统 DPI 设置改为 120 dpi(高分辨率显示器的用户常选择这么做)，WPF 假定需要 120 个像素来填满 1 英寸的空间。WPF 使用以下公式计算如何将逻辑单位变换为物理设备像素：

$$\begin{aligned} [\text{物理单位尺寸}] &= [\text{设备无关单位尺寸}] \times [\text{系统 DPI}] \\ &= 1/96 \text{ 英寸} \times 120 \text{ dpi} \\ &= 1.25 \text{ 像素} \end{aligned}$$

换句话说，将系统 DPI 设为 120 dpi 时，WPF 渲染引擎假定设备无关单位等于 1.25 个像素。如果显示 96×96 像素大小的按钮，那么物理尺寸实际为 120×120 像素(因为 $96 \times 1.25 = 120$)。这正是你所期望的结果——在标准显示器上大小为 1 英寸的按钮，在像素密度更高的显示器上仍保持 1 英寸的大小。

如果只用于按钮，这种自动缩放的意义不大。但 WPF 对它所显示的任何内容都使用设备无关单位，包括形状、控件、文本以及其他放在窗口中的内容。所以可将系统 DPI 改为任何所希望的数值，WPF 将无缝地调整应用程序的尺寸。

注意：

根据系统 DPI 计算出的像素尺寸可能是小数。可假定 WPF 简单地将度量尺寸舍入为最接近的像素。然而，默认情况下，WPF 的处理方式与此不同。如果元素的一条边落在两个像素之间，WPF 将使用反锯齿特性将这条边混合到相邻的像素。这看起来可能是多余的选择，但的确可改进视觉效果。如果为给控件增加皮肤效果而使用自定义绘制图形，那么就未必会整齐、清晰地定义边缘，从而需要进行一定程度的反锯齿处理。

调整系统 DPI 的步骤取决于操作系统。下面解释如何根据使用的操作系统调整系统 DPI。

Windows Vista

(1) 右击桌面并从上下文菜单中选择 Personalize 菜单项。

(2) 在左边的链接列表中，选择 Adjust Font Size (DPI)。

(3) 选择 96 或 120 dpi。或单击 Custom DPI 按钮，从而使用自定义的 DPI 设置。还可指定一个百分比值，如图 1-1 所示(例如，175%会将标准的 96 dpi 放大为 168 dpi)。此外，当使用自定义 DPI 设置时，还可使用 Use Windows XP style DPI scaling 选项，后面的“DPI 缩放”补充说明将对该选项进行说明。

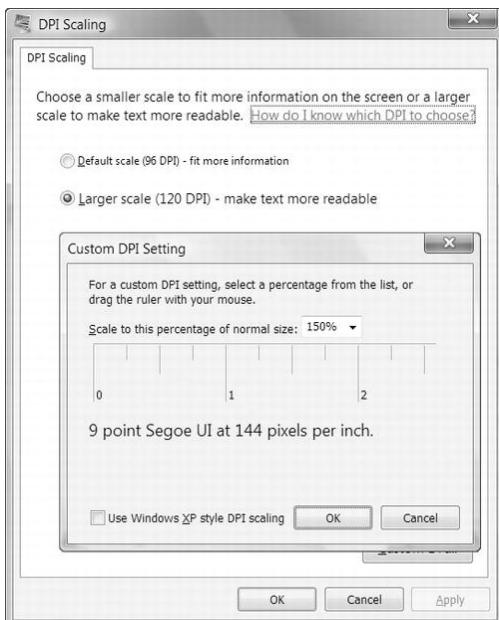


图 1-1 改变系统 DPI

Windows 7 和 Windows 8

(1) 右击桌面并从上下文菜单中选择 Personalize 菜单项。

(2) 在窗口左下角的链接列表中，选择 Dispaly。

(3) 在 Smaller(默认选项)、Medium 以及 Larger 选项之间进行选择。尽管这些选项用缩放百分比(100%、125%以及 150%)加以描述，但它们实际上对应于 DPI 值 96、120 和 144。您可能注意到前两个选项与 Windows Vista 和 Windows XP 中的标准设置是相同的，而第三个选项更大些。此外，为使用自定义的 DPI 百分比，可单击 Set Custom Text Size，如图 1-1 所示(例如，175%)

将标准的 96 dpi 放大为 168 dpi)。当使用自定义 DPI 设置时, 还可使用 Use Windows XP style DPI scaling 选项, 下面的“DPI 缩放”补充说明描述了这个选项。

DPI 缩放

众所周知, 旧式应用程序不支持较高的 DPI 设置。为此, Windows Vista 引入了一种称为位图缩放的新技术。Windows 较新版本也支持该特性。

通过缩放位图, 当运行不支持高 DPI 设置的应用程序时, Windows 会改变其尺寸, 就像它是一幅图像一样。该方法的优点在于应用程序仍认为它运行在标准的 96 dpi 设置下。Windows 无缝地变换输入(如鼠标单击), 并将输入传递到应用程序的“真正”坐标系统下的正确位置。

Windows 使用的是一种极好的缩放算法——该算法会考虑像素边界以避免模糊边缘, 并尽可能使用显卡硬件来提高速度——但该方法不可避免地会导致一定的显示模糊。它也存在严重的局限性, 因为 Windows 不能识别其实支持较高 DPI 设置的旧式应用程序。这是因为应用程序需要包含清单(manifest)或调用 SetProcessDPIAware(在 User32 中)来公布它们对高 DPI 设置的支持。尽管 WPF 应用程序正确地处理了该步骤, 但在 Windows Vista 之前创建的应用程序没有使用任何方法, 从而即使它们支持更高的 DPI 设置, 也不能使用位图缩放。

有两个可能的解决方案。如果有少数几个特定的应用程序支持高 DPI 设置, 但并不明确, 那么可以手动进行详细配置。为此, 在 Start 菜单中右击启动应用程序的快捷方式, 从上下文菜单中选择 Properties 菜单项。在 Compatibility 选项卡中选择 Disable Display Scaling on High DPI Settings 选项。如果有许多应用程序需要配置, 那么很快就会令人感到厌烦。

另一个可能的解决方法是完全禁用位图缩放。为此, 在图 1-1 中显示的 Custom DPI Setting 对话框中选中 Use Windows XP style DPI scaling 复选框。该方法的唯一限制是在高 DPI 设置下, 可能有些应用程序不能正确地显示(并且可能不能使用)。默认情况下, 当 DPI 设置为 120 dpi 或更小时, 选中 Use Windows XP style DPI scaling 复选框; 当 DPI 设置大于 120 dpi 时, 不选中 Use Windows XP style DPI scaling。

1.3.3 位图和矢量图形

当使用普通控件时, 自然可利用 WPF 的分辨率无关性。WPF 会负责确保任何显示内容都能自动地具有正确的尺寸。但是, 如果准备在应用程序中包含图像, 偶尔可能出现问题。例如, 在传统 Windows 应用程序中, 开发人员为工具栏命令按钮使用非常小的位图, 但在 WPF 应用程序中这并非一种理想方法, 因为当根据系统 DPI 进行放大或缩小时, 位图可能出现伪影(变得模糊)。反而, 当设计 WPF 用户界面时, 即使是最小的图标, 通常也使用矢量图形来实现。矢量图形被定义为一系列的形状, 并且它们能够很容易地缩放为任何尺寸。

注意:

当然, 相对于绘制一幅基本的位图, 绘制矢量图形需要耗费更长的时间, 但 WPF 包含可减少开销的优化措施, 以确保性能始终处于合理范围之内。

分辨率无关性的重要性无论如何强调都不过分。因为乍一看, 对于这个由来已久的问题(该问题确实如此), 它看起来像是简单的、优美的解决方法。但为了设计完全可缩放的用户界面, 开发人员需要接受一种新的思想。

1.4 WPF 体系结构

WPF 使用多层次体系结构。在顶层，应用程序与完全由托管 C#代码编写的一组高层服务进行交互。至于将.NET 对象转换为 Direct3D 纹理和三角形的实际工作，是在后台由一个名为 milcore.dll 的低级非托管组件完成的。milcore.dll 是使用非托管代码实现的，因为它需要和 Direct3D 紧密集成，并且它对性能极其敏感。

图 1-2 显示了 WPF 应用程序中各层的工作情况。

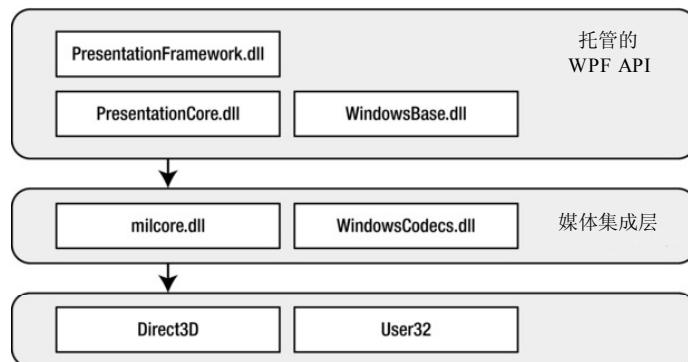


图 1-2 WPF 体系结构

以下列出图 1-2 中包含的一些重要组件：

- **PresentationFramework.dll** 包含 WPF 顶层的类型，包括那些表示窗口、面板以及其他类型控件的类型。它还实现了高层编程抽象，如样式。开发人员直接使用的大部分类都来自这个程序集。
- **PresentationCore.dll** 包含了基础类型，如 UIElement 类和 Visual 类，所有形状类和控件类都继承自这两个类。如果不需要窗口和控件抽象层的全部特征，可使用这一层，而且仍能利用 WPF 的渲染引擎。
- **WindowsBase.dll** 包含了更多基本要素，这些要素具有在 WPF 之外重用的潜能，如 DispatcherObject 类和 DependencyObject 类，这两个类引入了依赖项属性(详见第 4 章)。
- **milcore.dll** 是 WPF 渲染系统的核心，也是媒体集成层(Media Integration Layer, MIL)的基础。其合成引擎将可视化元素转换为 Direct3D 所期望的三角形和纹理。尽管将 milcore.dll 视为 WPF 的一部分，但它也是 Windows Vista 和 Windows 7 的核心系统组件之一。实际上，桌面窗口管理器/Desktop Window Manager, DWM)使用 milcore.dll 渲染桌面。
- **WindowsCodecs.dll** 是一套提供图像支持的低级 API(例如处理、显示以及缩放位图和 JPEG 图像)。
- **Direct3D** 是一套低级 API，WPF 应用程序中的所有图形都由它进行渲染。
- **User32** 用于决定哪些程序实际占有桌面的哪一部分。所以它仍被包含在 WPF 中，但不再负责渲染通用控件。

注意：

milcore.dll 有时称为“托管图形”引擎。与公共语言运行库(CLR)管理.NET 应用程序的生命期非常类似，milcore.dll 管理显示状态。而且正如有了 CLR，开发人员不再为释放对象和回收内存而感到烦恼一样，milcore.dll 让开发人员不必再考虑使窗口无效和重绘窗口。只需使用希望显示的内容创建对象即可，当拖动窗口、窗口被覆盖和显露、最小化窗口和还原窗口时，由 milcore.dll 负责绘制窗口的恰当部分。

需要认识到的最重要事实是，在 WPF 中所有绘图内容都由 Direct3D 渲染。不管使用普通显卡还是使用功能更强大的显卡，不管使用基本控件还是绘制更复杂的内容，也不管是在 Windows XP、Windows Vista 还是在 Windows 7 上运行应用程序，情况都是如此。甚至二维图形和普通文本也被转换为三角形并被传送到 Direct3D 管线，而不使用 GDI+或 User32 渲染图形。

类层次结构

本书将占用大量篇幅讲述 WPF 的名称空间和类。但在此之前，首先分析一下构成 WPF 基本控件集合的类的层次结构是很有帮助的。

图 1-3 简要显示了类层次结构中的几个重要分支。本书将详细深入地分析这些类以及它们之间的关系。

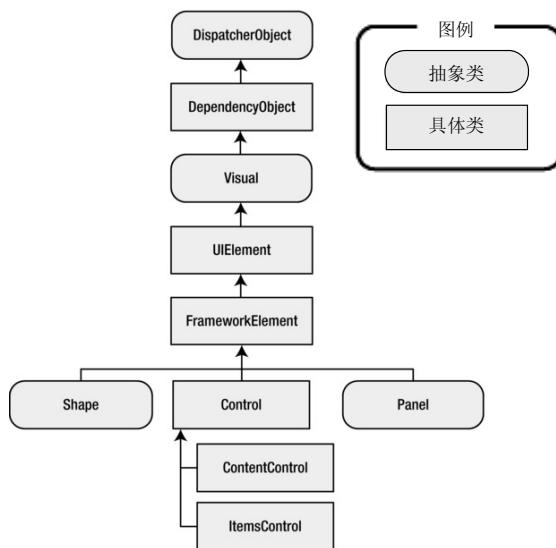


图 1-3 WPF 的主要类

下面将简要介绍图 1-3 中呈现的核心类。这些类中的许多类构成了完整的元素分支(如形状、面板以及控件)。

注意：

WPF 核心名称空间以 System.Windows 开头(如 System.Windows、System.Windows.Controls 以及 System.Windows.Media)。唯一例外是由 System.Windows.Forms 开头的名称空间，它们是 Windows 窗体工具包的一部分。

1. System.Threading.DispatcherObject 类

WPF 应用程序使用为人熟知的单线程亲和(Single-Thread Affinity, STA)模型，这意味着整个用户界面由单个线程拥有。从另一个线程与用户界面元素进行交互是不安全的。为方便使用此模型，每个 WPF 应用程序由协调消息(键盘输入、鼠标移动乃至框架处理，如布局)的调度程序管理。通过继承自 DispatcherObject 类，用户界面中的每个元素都可以检查代码是否在正确的线程上运行，并能通过访问调度程序为用户界面线程封送代码。在第 31 章将介绍有关 WPF 线程模型的更多内容。

2. System.Windows.DependencyObject 类

在 WPF 中，主要通过属性与屏幕上的元素进行交互。在早期设计阶段，WPF 的设计者决定创建一个更加强大的属性模型，该模型支持许多特性，例如更改通知、默认值继承以及减少属性存储空间。最终结果就是依赖项属性(dependency property)特性，第 4 章将分析该特性。通过继承自 DependencyObject 类，WPF 类可获得对依赖项属性的支持。

3. System.Windows.Media.Visual 类

在 WPF 窗口中显示的每个元素本质上都是 Visual 对象。可将 Visual 类视为绘图对象，其中封装了绘图指令、如何执行绘图的附加细节(如剪裁、透明度以及变换设置)以及基本功能(如命中测试)。Visual 类还在托管的 WPF 库和渲染桌面的 milcore.dll 程序集之间提供了链接。任何继承自 Visual 的类都能在窗口上显示出来。如果更愿意使用轻量级的 API 创建用户界面，而不想使用 WPF 的高级框架特征，可使用第 14 章中描述的方法，直接对 Visual 对象进行编程。

4. System.Windows.UIElement 类

UIElement 类增加了对 WPF 本质特征的支持，如布局、输入、焦点和事件(WPF 团队使用首字母缩写词 LIFE 来表示)。例如，这里定义两个步骤的测量和排列布局过程，这些内容将在第 18 章中介绍。在该类中，原始的鼠标单击和按键操作被转换为更有用的事件，如 MouseEnter 事件。与属性类似，WPF 实现了增强的称为路由事件(routed event)的事件路由系统。第 5 章将讲述路由事件的工作原理。最后，UIElement 类中还添加了对命令的支持(详见第 9 章)。

5. System.Windows.FrameworkElement 类

FrameworkElement 类是 WPF 核心继承树中的最后一站。该类实现了一些全部由 UIElement 类定义的成员。例如，UIElement 类为 WPF 布局系统设置了基础，但 FrameworkElement 类提供了支持它的重要属性(如 HorizontalAlignment 和 Margin 属性)。UIElement 类还添加了对数据绑定、动画以及样式等核心特性的支持。

6. System.Windows.Shapes.Shape 类

基本的形状类(如 Rectangle 类、Polygon 类、Ellipse 类、Line 类以及 Path 类)都继承自该类。可将这些形状类与更传统的 Windows 小组件(如按钮和文本框)结合使用。在第 12 章将开始介绍如何构建形状。

7. System.Windows.Controls.Control 类

控件(control)是可与用户进行交互的元素。控件显然包括 TextBox 类、Button 类和 ListBox 类等。Control 类为设置字体以及前景色与背景色提供了附加属性。但最令人感兴趣的细节是模板支持，通过模板支持，可使用自定义风格的绘图替换控件的标准外观。第 17 章将介绍控件模板。

注意：

在 Windows 窗体编程中，窗体中的每个可视化项都称为控件。在 WPF 中，情况不再如此。可视化内容被称为元素(element)，只有部分元素是控件(控件是那些能够接收焦点并能与用户进行交互的元素)。更令人费解之处在于，许多元素是在 System.Windows.Controls 名称空间中定义的，但它们不是继承自 System.Windows.Controls.Control 类，并且不被认为是控件。Panel 类便是其中一例。

8. System.Windows.Controls.ContentControl 类

ContentControl 类是所有具有单一内容的控件的基类，包括简单的标签乃至窗口的所有内容。该模型给人印象最深刻的部分是：控件中的单一内容可以是普通字符串乃至具有其他形状和控件组合的布局面板(详见第 6 章)。

9. System.Windows.Controls.ItemsControl 类

ItemsControl 类是所有显示选项集合的控件的基类，如 ListBox 和 TreeView 控件。列表控件十分灵活——例如，使用 ItemsControl 类的内置特征，可将简单的 ListBox 控件转换成单选按钮列表、复选框控件列表、平铺的图像或是您所选择的完全不同的元素的组合。实际上，WPF 中的菜单、工具栏以及状态栏都是特定的列表，并且实现它们的类都继承自 ItemsControl 类。在第 19 章中学习数据绑定时，将开始使用列表控件。在第 20 章中将进一步学习列表控件，第 22 章将介绍最专业的列表控件。

10. System.Windows.Controls.Panel 类

Panel 类是所有布局容器的基类，布局容器是可包含一个或多个子元素、并按特定规则对子元素进行排列的元素。这些容器是 WPF 布局系统的基础，要以最富有吸引力、最灵活的方式安排内容，使用这些容器是关键所在。在第 3 章将详述 WPF 布局系统。

1.5 WPF 4.5

WPF 是一种成熟的技术。它是几个已经发布的.NET 平台的一部分，并通过以下版本不断地进行完善：

- **WPF 3.0。**这是 WPF 的第一个版本，它与另两种新技术一并发布：WCF(Windows Communication Foundation)和 Windows WF(Workflow Foundation)。这三种新技术合称为.NET Framework 3.0。

- **WPF 3.5。**一年后，一个新的 WPF 版本作为.NET Framework 3.5 的一部分发布。新版本 WPF 的新特性主要是一些小的改进，包括错误修复和性能改进。
- **WPF 3.5 SP1。**当发布.NET Framework Service Pack 1(SP1)时，WPF 设计人员抓住这个机会增添了一些新功能，例如平滑图形效果(通过像素着色器实现)以及高级的 DataGrid 控件。
- **WPF 4。**该 WPF 版本做了大量改进，包括更好地渲染文本、动画更自然流畅以及支持多点触控。
- **WPF 4.5。**相对于上述版本更新，迄今为止，这一最新 WPF 版本对 WPF 4 所做的更新是最少的，这也表明 WPF 技术已经走向成熟。除纠正一些一般性错误并对性能做了调整外，WPF 4.5 还对数据绑定系统做了大量完善工作，比如完善了数据绑定表达式、可视化，并可以支持 INotifyDataError 接口以及数据视图同步。第 8 章、第 19 章和第 22 章将介绍这些功能。

1.5.1 WPF 工具包

在将新控件集成到.NET 平台的 WPF 库之前，通常首先将新控件放入被称为 WPF 工具包(WPF Toolkit)的 Microsoft 下载中。WPF 工具包不仅是预览 WPF 未来方向的场所，还是非常好的实用组件和控件的源，使得在正常的 WPF 发布周期外可使用这些控件和组件。例如，WPF 没有提供任何类型的制图工具，但 WPF 工具包提供了一套控件用于创建柱状图、饼图、气泡图、散点图以及线图。

本书偶尔会引用 WPF 工具包，以指出在核心.NET 运行库中未提供但非常有用的部分功能。为下载 WPF 工具包、查看其代码或阅读其文档，可导航到 <http://wpf.codeplex.com>。还可以在那里找到指向由 Microsoft 托管的其他 WPF 项目的链接，包括 WPF Futures(该项目提供了更多实验性 WPF 特性和 WPF 测试工具)。

1.5.2 Visual Studio 2012

尽管可以手工或使用面向图形设计的工具 Expression Blend 构建 WPF 用户界面，但大多数开发人员将首先使用 Visual Studio 进行开发，而且大部分(或全部)时间用于使用 Visual Studio 进行开发。本书假定您使用 Visual Studio，并且偶尔会介绍如何使用 Visual Studio 用户界面执行一项重要任务，如添加资源、配置项目属性以及创建控件库程序集。但不会花费大量时间研究 Visual Studio 的设计时技巧，而是重点研究创建专业应用程序所需的底层标记和代码。

注意：

您可能已经知道了如何使用 Visual Studio 创建 WPF 项目，但这里仍要简单概括一下。首先选择 File | New | TRA Project。然后在左边的树中选择 Visual C# | Windows 组，在右边的列表中选择 WPF Application 模板。第 24 章将介绍更专用的 WPF Browser Application 模板。一旦选择好路径，就输入项目名，并单击 OK 按钮，最终将得到 WPF 应用程序的基本框架。

1. 多目标

在过去，Visual Studio 的每个版本和特定版本的.NET 紧密耦合在一起。Visual Studio 2012 没有这一限制，允许您设计针对 2.0 到 4.5 之间的任何.NET 版本的应用程序。

显然，并不能使用.NET 2.0 创建 WPF 应用程序，但所有更新版本都支持 WPF。为了获得最广泛的兼容性，可选择将较旧版本(如.NET 3.5 或.NET 4)作为目标。例如，.NET 3.5 应用程序可运行于.NET 3.5、4 以及 4.5 运行库上。或者，也可选择.NET 4.5 作为目标，以使用 WPF 或.NET 平台中的更新特性。但是，如果需要支持旧式 Windows XP 计算机，那么无法将.NET 4 作为目标，因为这是支持 Windows XP 的最新.NET 版本。

当使用 Visual Studio 创建新项目时，可在 New Project 对话框，从项目模板列表上方的位于顶部的下拉列表中选择.NET Framework 的目标版本，如图 1-4 所示。

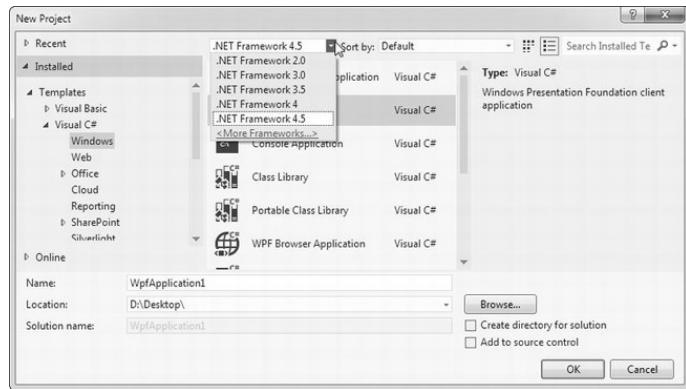


图 1-4 选择.NET Framework 的目标版本

以后还可以随时通过在 Solution Explorer 中双击 Properties 节点，并改变 Target Framework 列表中的选项来改变目标版本。

为提供准确的多目标特征，Visual Studio 为每个.NET 版本提供了参考程序集。这些程序集包括所有类型的元数据，但不包括需要的实现代码。这意味着 Visual Studio 可使用参考程序集修补智能感知和错误检查，确保您不能使用在设定的.NET 版本中未提供的控件、类以及成员。Visual Studio 还使用元数据确定在工具箱中应当显示哪些控件、在 Properties 窗口和 Object Browser 中应当显示哪些成员等，保证整个 IDE 被限制于您选择的版本。

2. Visual Studio 设计器

Visual Studio 提供了功能丰富的设计器用于创建 WPF 用户界面。虽然 Visual Studio 2012 允许拖放 WPF 窗口，但这并不意味着应当立即(或完全)使用 Visual Studio 创建用户界面。正如第 3 章中介绍的，WPF 使用灵活的、具有细微差别的布局模型，从而允许您在用户界面中为确定元素的尺寸和位置使用不同的策略。为得到需要的结果，需要选择恰当的布局容器组合，对其进行合理安排，并配置属性。Visual Studio 可帮您完成该任务，但如果首先学习基本的 XAML 标记以及 WPF 布局，该任务会变得更容易。此后就能查看 Visual Studio 可视化设计器生成的标记，并酌情手动修改这些标记。

一旦熟练掌握了 XAML 语法(详见第 2 章)，并学习了 WPF 的布局控件系列(详见第 3 章)，对于选择如何创建窗口就足够了。有些专业开发人员使用 Visual Studio，有些使用 Expression Blend，有些手动编写 XAML，而有些组合使用这些方法(例如手动创建基本布局结构，然后使用 Visual Studio 设计器以对其进行配置)。

1.6 小结

本章简要介绍了 WPF 及其作用，分析了 WPF 的底层体系结构，并简要介绍了核心类。

显然，WPF 引入了许多重要变化。然而，有 5 条重要的准则更加突出，因为它们和以前的 Windows 用户界面工具包(如 Windows 窗体)的区别很大。这些准则如下：

- **硬件加速。**通过 DirectX 执行所有 WPF 绘图操作，以便充分利用现代显卡的最新功能。
- **分辨率无关性。**WPF 能够根据系统 DPI 设置，非常灵活地放大和缩小显示的内容，以使其适合所用的显示器和显示选择。
- **控件无固定外观。**在传统的 Windows 开发中，在定制的符合需求的控件(此类控件是指自绘制的控件)和由操作系统渲染的本质上外观固定的控件之间存在很大的差别。在 WPF 中，从基本的 Rectangle 形状到标准的 Button 控件或更复杂的 Toolbar 控件，都是使用相同的渲染引擎绘制的，并且都是完全可定制的。因此，WPF 控件经常被称为无外观控件——它们为控件定义了功能，但没有固定“外观”。
- **声明式用户界面。**第 2 章将介绍 XAML，XAML 是用于定义 WPF 用户界面的标记标准。通过 XAML，不必编写代码即可创建窗口。特别是 XAML 的能力不局限于创建一成不变的用户界面。可以使用许多工具，如数据绑定和触发器等自动运行基本的用户界面行为(例如，当页面通过记录源时文本框更新自身，当鼠标移动到标签上时标签变亮)，所有这些都不需要编写 C# 代码。
- **基于对象的绘图。**即使准备在更低级的可视化层(而非高级元素层)上工作，也不需要使用绘图和像素进行工作，而是创建图形对象并让 WPF 尽可能最优化地显示出来。

全书都会运用这些原则。但在进一步分析这些原则之前，应当首先学习相关的补充标准。

下一章将介绍用于定义 WPF 用户界面的标记语言 XAML。

第 2 章



XAML

XAML(Extensible Application Markup Language 的简写，发音为“zammel”)是用于实例化.NET 对象的标记语言。尽管 XAML 是一种可应用于诸多不同问题领域的技术，但其主要作用是构造 WPF 用户界面。换言之，XAML 文档定义了在 WPF 应用程序中组成窗口的面板、按钮以及各种控件的布局。

不必再手动编写 XAML，您将使用工具生成所需的 XAML。如果您是一位图形设计人员，该工具可能是图形设计程序，如 Expression Blend。如果您是一名开发人员，您开始时可能使用 Microsoft Visual Studio。这两个工具在生成 XAML 时本质上是相同的，因此可使用 Visual Studio 创建一个基本用户界面，然后将该界面移交给一个出色的设计团队，由设计团队在 Expression Blend 中使用自定义图形润色这个界面。实际上，将开发人员和设计人员的工作流程集成起来的能力，是 Microsoft 推出 XAML 的重要原因之一。

本章将详细介绍 XAML，分析 XAML 的作用、宏观体系结构以及语法。一旦理解了 XAML 的一般性规则，就可以了解在 WPF 用户界面中什么是可能的、什么是不可能的，并了解在必要时如何手动修改用户界面。更重要的是，通过分析 WPF XAML 文档中的标签，可学习一些支持 WPF 用户界面的对象模型，从而为进一步深入分析 WPF 用户界面做好准备。

新增功能：

WPF 4.5 并未为 XAML 标准添加新内容。事实上，即使是 XAML 2009 较小的完善之处也未得到完全实现。只在松散的 XAML 文件中支持它们，在编译的 XAML 资源(几乎每个 WPF 应用程序都使用这些资源)却不可以。实际上，XAML 2009 可能永远都不会成为完全集成到 WPF 中的一部分，原因是 XAML 2009 的改进没那么重要，而且对 XAML 编译器的任何更改都会引起安全和性能问题。因此，本书不会介绍 XAML 2009。

2.1 理解 XAML

开发人员很久前就已经意识到，要处理图形丰富的复杂应用程序，最有效的方式是将图形部分从底层的代码中分离出来。这样一来，美工人员可独立地设计图形，而开发人员可独立地编写代码。这两部分工作可单独地进行设计和修改，而不会有任何版本问题。

2.1.1 WPF 之前的图形用户界面

使用传统的显示技术，从代码中分离出图形内容并不容易。对于 Windows 窗体应用程序而言，关键问题是创建的每个窗体完全都是由 C# 代码定义的。在将控件拖动到设计视图上并配置

控件时, Visual Studio 将在相应的窗体类中自动调整代码。但图形设计人员没有任何可以使用 C# 代码的工具。

相反, 美工人员必须将他们的工作内容导出为位图。然后可使用这些位图确定窗体、按钮以及其他控件的外观。对于简单的固定用户界面而言, 这种方法效果不错, 但在其他一些情况下会受到很大的限制。这种方法存在以下几个问题:

- 每个图形元素(背景和按钮等)需要导出为单独的位图。这限制了组合位图的能力和使用动态效果的能力, 如反锯齿、透明和阴影效果。
- 相当多的用户界面逻辑都需要开发人员嵌入到代码中, 包括按钮的大小、位置、鼠标悬停效果以及动画。图形设计人员无法控制其中的任何细节。
- 在不同的图形元素之间没有固有的连接, 所以最后经常会使用不匹配的图像集合。跟踪所有这些项会增加复杂性。
- 在调整图形大小时必然会损失质量。因此, 一个基于位图的用户界面是依赖于分辨率的。这意味着它不能适应大显示器以及高分辨率显示设置, 而这严重背离了 WPF 的设计初衷。

如果曾经有过在一个团队中使用自定义图形来设计 Windows 窗体应用程序的经历, 肯定遇到过不少挫折。即使用户界面是由图形设计人员从头开始设计的, 也需要使用 C# 代码重新创建它。通常, 图形设计人员只是准备一个模拟界面, 然后需要开发人员再不辞辛劳地将它转换到应用程序中。

WPF 通过 XAML 解决了该问题。当在 Visual Studio 中设计 WPF 应用程序时, 当前设计的窗口不被转换为代码。相反, 它被串行化到一系列 XAML 标签中。当运行应用程序时, 这些标签用于生成构成用户界面的对象。

注意:

XAML 对于 WPF 不是必需的, 理解这一点是很重要的。Visual Studio 当然可使用 Windows 窗体方法, 通过语句代码来构造 WPF 窗口。但如果这样的话, 窗口将被限制在 Visual Studio 开发环境之内, 只能由编程人员使用。

换句话说, WPF 不见得使用 XAML。但 XAML 为协作提供了可能, 因为其他设计工具理解 XAML 格式。例如, 聪明的设计人员可使用 Microsoft Expression Design 等工具精细修改 WPF 应用程序的图形界面, 或使用 Expression Blend 等工具为 WPF 应用程序构建精美动画。当学完本章后, 您可能希望阅读位于 <http://windowsclient.net/wpf/white-papers/thenevaporation.aspx> 的 Microsoft 白皮书, 该白皮书对 XAML 进行了评论, 并且分析了开发人员和设计人员协作开发 WPF 应用程序的一些方法。

提示:

XAML 在 Windows 应用程序中扮演的角色, 与控件标签在 ASP.NET Web 应用程序中扮演的角色类似。区别是 ASP.NET 标签语法设计得看起来像 HTML, 所以设计人员能使用普通的 Web 设计应用程序设计 Web 页面, 如 Microsoft Expression 和 Adobe Dreamweaver。与 WPF 一样, 为便于设计, 用于 ASP.NET Web 页面的实际代码通常单独放在一个文件中。

2.1.2 XAML 变体

实际上术语“XAML”有多种含义。到目前为止，我们使用 XAML 表示整个 XAML 语言，它是一种基于通用 XML 语法、专门用于表示一棵.NET 对象树的语言(这些对象可以是窗口中的按钮、文本框，或是您已经定义好的自定义类。实际上，XAML 甚至可用于其他平台来表示非.NET 对象)。

XAML 还包含如下几个子集：

- **WPF XAML** 包含描述 WPF 内容的元素，如矢量图形、控件以及文档。目前，它是最重要的 XAML 应用，也是本书将要分析的一个子集。
- **XPS XAML** 是 WPF XAML 的一部分，它为格式化的电子文档定义了一种 XML 表示方式。XPS XAML 已作为单独的 XML 页面规范(XML Paper Specification, XPS)标准发布。第 28 章将分析 XPS。
- **Silverlight XAML** 是一个用于 Microsoft Silverlight 应用程序的 WPF XAML 子集。Silverlight 是一个跨平台的浏览器插件，通过它可创建具有二维图形、动画、音频和视频的富 Web 内容。第 1 章介绍了关于 Silverlight 的更多内容，您也可以访问 <http://silverlight.net> 来了解详情。
- **WF XAML** 包括描述 WF(Work Flow, 工作流)内容的元素，可访问 <http://tinyurl.com/d9xr2nv> 来了解有关 WF 的更多内容。

2.1.3 XAML 编译

WPF 的创建者知道，XAML 不仅要能够解决设计协作问题，它还需要快速运行。尽管基于 XML 的格式(如 XAML)可以很灵活并且很容易地迁移到其他工具和平台，但它们未必是最有效的选择。XML 的设计目标是具有逻辑性、易读而且简单，没有被压缩。

WPF 使用 BAML(Binary Application Markup Language, 二进制应用程序标记语言)来克服这个缺点。BAML 并非新事物，它实际上就是 XAML 的二进制表示。当在 Visual Studio 中编译 WPF 应用程序时，所有 XAML 文件都被转换为 BAML，这些 BAML 然后作为资源被嵌入到最终的 DLL 或 EXE 程序集中。BAML 是标记化的，这意味着较长的 XAML 被较短的标记替代。BAML 不仅明显小一些，还对其进行优化，从而使它在运行时能够更快地解析。

大多数开发人员不必考虑 XAML 向 BAML 的转换，因为编译器会在后台执行这项工作。但也可以使用未经编译的 XAML，这对于需要即时提供一些用户界面的情况可能是有意义的(例如，将从某个数据库中提取的内容作为一块 XAML 标签)。本章稍后的 2.5 节“加载和编译 XAML”将介绍工作原理。

使用 Visual Studio 创建 XAML

本章将介绍 XAML 标记的所有细节。当然，在设计应用程序时，不必手动编写所有 XAML。反而，将使用一个能够拖放用户界面元素的工具，例如 Visual Studio。鉴于这种情况，您可能会好奇是否值得花费大量时间来学习 XAML 语法。

答案是肯定的。理解 XAML 对于设计 WPF 应用程序是至关重要的。这将有助于学习 WPF 的重要概念，例如附加属性(本章)、布局(第 3 章)、路由事件(第 4 章)和内容模型(第 6 章)等。更重要的是，有许多任务只能通过手动编写 XAML 来完成，或者通过手动编写 XAML 来完成

更加容易。

大多数 WPF 开发人员会结合使用多种技术，使用设计工具(Visual Studio 或 Expression Blend)设置用户界面的布局，然后通过手动编辑 XAML 标记对其进行精细调整。不过，您可能会发现在第 3 章学习布局容器之前，手动编写所有的 XAML 是最容易的，这是因为需要使用布局容器在窗口中合理地布置多个控件。

2.2 XAML 基础

一旦理解了一些基本规则，XAML 标准是非常简单的：

- XAML 文档中的每个元素都映射为.NET 类的一个实例。元素的名称也完全对应于类名。例如，元素<Button>指示 WPF 创建 Button 对象。
- 与所有 XML 文档一样，可在一个元素中嵌套另一个元素。您在后面将看到，XAML 让每个类灵活地决定如何处理嵌套。但嵌套通常是一种表示“包含”的方法——换句话说，如果在一个 Grid 元素中发现一个 Button 元素，那么用户界面可能包括一个在其内部包含一个 Button 元素的 Grid 元素。
- 可通过特性(attribute)设置每个类的属性(property)。但在某些情况下，特性不足以完成这项工作。对于此类情况，需要通过特殊的语法使用嵌套的标签(tag)。

提示：

如果对 XML 一无所知，那么有必要在处理 XAML 之前学习 XML 基础知识。为了快速了解 XML，可参阅 <http://www.w3schools.com/xml> 网址上的基于 Web 的免费辅导。

在继续学习前，先看一看下面的 XAML 文档基本框架，该文档表示一个新的空白窗口(与使用 Visual Studio 创建的一样)。为了便于说明，对每行代码都使用数字进行了编号：

```

1 <Window x:Class="WindowsApplication1.Window1"
2     xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
3     xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
4     Title="Window1" Height="300" Width="300">
5
6     <Grid>
7     </Grid>
8 </Window>
```

该文档仅含两个元素——顶级的 Window 元素以及一个 Grid 元素，Window 元素代表整个窗口，在 Grid 元素中可以放置所有控件。尽管可使用任何顶级元素，但是 WPF 应用程序只使用以下几个元素作为顶级元素：

- Window 元素
- Page 元素(该元素和 Window 元素类似，但它用于可导航的应用程序)
- Application 元素(该元素定义应用程序资源和启动设置)

与在所有 XML 文档中一样，在 XAML 文档中只能有一个顶级元素。在上例中，这意味着只要使用</Window>标签关闭了 Window 元素，文档就结束了。在后面不能再有任何内容了。

查看 Window 元素的开始标签，将发现几个有趣的特性，包括一个类名和两个 XML 名称空间(将在 2.2.1 节介绍)。还会发现三个属性，如下所示：

```
4 Title="Window1" Height="300" Width="300">>
```

每个特性对应 Window 类的一个单独属性。总之，这告诉 WPF 创建标题为“Window1”的窗口，并使窗口的大小为 300×300 单位。

注意：

第 1 章已经提到，WPF 使用相对度量系统，这不是大多数 Windows 开发人员所期望的。WPF 不是使用物理像素设置尺寸，而是使用可进行缩放以适应不同显示器分辨率的设备无关单位，设备无关单位被定义为 1/96 英寸。这意味着，如果系统 DPI 设置为标准的 96 dpi，那么在上例中，300×300 单位大小的窗口将被渲染为 300×300 像素大小。但在一个使用更高系统 DPI 设置的系统中，将使用更多的像素来渲染这个窗口。在第 1 章中已对此进行了完整介绍。

2.2.1 XAML 名称空间

显然，只提供类名是不够的。XAML 解析器还需要知道类位于哪个.NET 名称空间。例如，在许多名称空间中可能都有 Window 类——Window 类可能是指 System.Windows.Window 类，也可能是指位于第三方组件中的 Window 类，或您自己在应用程序中定义的 Window 类等。为了弄清实际上希望使用哪个类，XAML 解析器会检查应用于元素的 XML 名称空间。

下面是该机制的工作原理。上面显示的示例文档定义了两个名称空间：

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

注意：

使用特性声明 XML 名称空间。这些特性能被放入任何元素的开始标签中。但约定要求，在文档中需要使用的所有名称空间应在第一个标签中声明，正如在这个示例中所做的那样。一旦声明一个名称空间，在文档中的任何地方都可以使用该名称空间。

xmlns 特性是 XML 中的一个特殊特性，它专门用来声明名称空间。这段标记声明了两个名称空间，在创建的所有 WPF XAML 文档中都会使用这两个名称空间：

- `http://schemas.microsoft.com/winfx/2006/xaml/presentation` 是 WPF 核心名称空间。它包含了所有 WPF 类，包括用来构建用户界面的控件。在该例中，该名称空间的声明没有使用名称空间前缀，所以它成为整个文档的默认名称空间。换句话说，除非另行指明，每个元素自动位于这个名称空间。
- `http://schemas.microsoft.com/winfx/2006/xaml` 是 XAML 名称空间。它包含各种 XAML 实用特性，这些特性可影响文档的解释方式。该名称空间被映射为前缀 x。这意味着可通过在元素名称之前放置名称空间前缀 x 来使用该名称空间(例如`<x:ElementName>`)。

正如在前面看到的，XML 名称空间的名称和任何特定的.NET 名称空间都不匹配。XAML 的创建者选择这种设计的原因有两个。按照约定，XML 名称空间通常是 URI(如上面所示)。这些 URI 看起来像是在指明 Web 上的位置，但实际上不是。通过使用 URI 格式的名称空间，不同组织就基本不会无意中使用相同的名称空间创建不同的基于 XML 的语言。因为 schemas.com 域归 Microsoft 所有，只有 Microsoft 会在 XML 名称空间的名称中使用它。

另一个原因是 XAML 中使用的 XML 名称空间和.NET 名称空间不是一一对应的，如果一一对应的话，会显著增加 XAML 文档的复杂程度。此处的问题在于，WPF 包含了十几种名称空间(所有这些名称空间都以 System.Windows 开头)。如果每个.NET 名称空间都有不同的 XML 名称空间，那就需要为使用的每个控件指定确切的 XML 名称空间，这很快就会使 XAML 文档变得混乱不堪。所以，WPF 创建人员选择了这种方法，将所有这些.NET 名称空间组合到单个 XML 名称空间中。因为在不同的.NET 名称空间中都有一部分 WPF 类，并且所有这些类的名称都不相同，所以这种设计是可行的。

名称空间信息使得 XAML 解析器可找到正确的类。例如，当查找 Window 和 Grid 元素时，首先会查找默认情况下它们所在的 WPF 名称空间，然后查找相应的.NET 名称空间，直至找到 System.Windows.Window 类和 System.Windows.Controls.Grid 类。

2.2.2 代码隐藏类

可通过 XAML 构造用户界面，但为了使应用程序具有一定的功能，就需要用于连接包含应用程序代码的事件处理程序的方法。XAML 通过使用如下所示的 Class 特性简化了这个问题：

```
1 <Window x:Class="WindowsApplication1.Window1"
```

在 XAML 名称空间的 Class 特性之前放置了名称空间前缀 x，这意味着这是 XAML 语言中更通用的部分。实际上，Class 特性告诉 XAML 解析器用指定的名称生成一个新类。该类继承自由 XML 元素命名的类。换句话说，该例创建了一个名为 Window1 的新类，该类继承自 Window 基类。

Window1 类是编译时自动生成的。这正是令人感兴趣之处。您可以提供 Window1 的部分类，该部分类会与自动生成的那部分合并在一起。您提供的部分类正是包含事件处理程序代码的理想容器。

注意:

这个过程是使用 C# 语言的部分类(partial class)特征实现的。部分类允许在开发阶段把一个类分成两个或更多独立的部分，并在编译过的程序集中把这些独立的部分融合到一起。部分类可用于各种代码管理情形，但在此类情况下是最有用的，在此编写的代码需要和设计工具生成的文件融合到一起。

Visual Studio 会自动帮助您创建可以放置事件处理代码的部分类。例如，如果创建一个名为 WindowsApplication1 的应用程序，该应用程序包含名为 Window1 的窗口(就像上面的示例那样)，Visual Studio 将首先提供基本的类框架：

```
namespace WindowsApplication1
{
    /// <summary>
    /// Interaction logic for Window1.xaml
    /// </summary>
    public partial class Window1 : Window
    {
        public Window1()
        {
            InitializeComponent();
        }
    }
}
```

```

    }
}

```

在编译应用程序时，定义用户界面的 XAML(如 Window1.xaml)被转换为 CLR 类型声明，这些类型声明与代码隐藏类文件(如 Window1.xaml.cs)中的逻辑代码融合到一起，形成单一的单元。

1. InitializeComponent()方法

现在，Window1 类尚不具有任何真正的功能。然而，它确实包含了一个非常重要的细节——默认构造函数，当创建类的一个实例时，该构造函数调用 InitializeComponent()方法。

注意：

InitializeComponent()方法在 WPF 应用程序中扮演着重要角色。因此，永远不要删除窗口构造函数中的 InitializeComponent()调用。同样，如果为窗口类添加另一个构造函数，也要确保调用 InitializeComponent()方法。

InitializeComponent()方法在源代码中不可见，因为它是在编译应用程序时自动生成的。本质上，InitializeComponent() 方法的所有工作就是调用 System.Windows.Application 类的 LoadComponent()方法。LoadComponent()方法从程序集中提取 BAML(编译过的 XAML)，并用它来构建用户界面。当解析 BAML 时，它会创建每个控件对象，设置其属性，并关联所有事件处理程序。

注意：

如果仍然不甚明了，可跳过前面的部分转到本章的结尾。在 2.5.3 节“使用代码和编译过的 XAML”中，可看到自动为 InitializeComponent()方法生成的代码。

2. 命名元素

还有一个需要考虑的细节。在代码隐藏类中，经常希望通过代码来操作控件。例如，可能需要读取或修改属性，或自由地关联以及断开事件处理程序。为达到此目的，控件必须包含 XAML Name 特性。在上面的示例中，Grid 控件没有包含 Name 特性，所以不能在代码隐藏文件中对其进行操作。

下面的标记演示了如何为 Grid 控件关联名称：

```

6   <Grid x:Name="grid1">
7   </Grid>

```

可在 XAML 文档中手动执行这个修改，也可在 Visual Studio 设计器中选择该网格，并通过 Properties 窗口设置其 Name 属性。

无论使用哪种方法，Name 特性都会告诉 XAML 解析器将这样一个字段添加到为 Window1 类自动生成的部分：

```
private System.Windows.Controls.Grid grid1;
```

现在可以在 Window1 类的代码中，通过 grid1 名称与网格元素进行交互了：

```
MessageBox.Show(String.Format("The grid is {0}x{1} units in size.",
```

```
grid1.ActualWidth, grid1.ActualHeight));
```

该技术没有为这个简单的网格示例添加更多内容，但当需要从输入控件(如文本框和列表框)中读取数值时它将变得更重要。

上面显示的 Name 属性是 XAML 语言的一部分，用于帮助集成代码隐藏类。让人感到有些困惑的是，许多类定义了自己的 Name 属性(FrameworkElement 基类就是一例，所有 WPF 元素都继承自该类)。XAML 解析器使用一种更聪明的方法来处理这一问题。可设置 XAML Name 属性(使用 x:前缀)，也可设置属于实际元素的 Name 属性(通过删除前缀)。对于这两种方式，结果都是相同的——指定的名称在自动生成的代码文件中使用，并且用于设置 Name 属性。

这意味着下面的标记和前面的标记是等价的：

```
<Grid Name="grid1">
</Grid>
```

只有当包含 Name 属性的类使用 RuntimeNameProperty 特性修饰之后，这才是可行的。RuntimeNameProperty 特性指示哪个属性的值将作为该类型的实例的名称(显然，通常是使用 Name 属性)。FrameworkElement 类使用 RuntimeNameProperty 特性进行了修饰，所以上面的标记是没有问题的。

提示：

在传统的 Windows 窗体应用程序中，每个控件都有名称。而在 WPF 应用程序中，没有这一要求。在本书的示例中，当不需要元素名称时通常会省略，这样可以使标记更加简洁。

到现在为止，应当对如何解释定义窗口的 XAML 文档，以及 XAML 文档是如何被转换为最终编译过的类(包括编写的其他所有代码)有了基本的理解。下一节将介绍有关属性语法的更多细节，并将学习如何关联事件处理程序。

2.3 XAML 中的属性和事件

到目前为止，只考虑了一个较为单调乏味的示例——包含一个空 Grid 控件的空白窗口。在继续学习之前，有必要首先介绍一个更贴近实际的包含几个控件的窗口。图 2-1 显示了这样一个具有自动问答功能的示例。

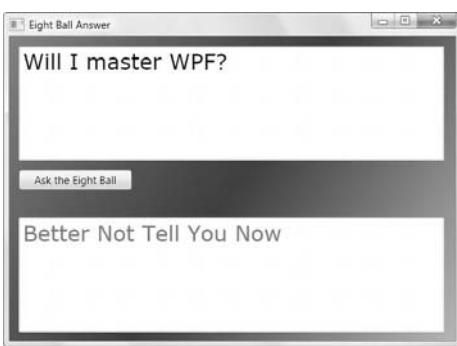


图 2-1 提出一个问题并且会显示答案

图 2-1 中显示的 Eight Ball Answer 窗口包含 4 个控件：一个 Grid 控件(在 WPF 中最常见的用于安排布局的工具)、两个 TextBox 控件和一个 Button 控件。安排和配置这些控件所需的标记比前面例子中的标记长得多。下面简要列出这些标记，其中一些细节用省略号(...)代替，以便于描述整个结构：

```
<Window x:Class="EightBall.Window1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Eight Ball Answer" Height="328" Width="412">
<Grid Name="grid1">
    <Grid.Background>
        ...
    </Grid.Background>
    <Grid.RowDefinitions>
        ...
    </Grid.RowDefinitions>
    <TextBox Name="txtQuestion" ... >
        ...
    </TextBox>
    <Button Name="cmdAnswer" ... >
        ...
    </Button>

    <TextBox Name="txtAnswer" ... >
        ...
    </TextBox>
</Grid>
</Window>
```

后面几节将分析该文档中的各个部分，并学习 XAML 的语法。

注意：

XAML 并不局限于只能用于属于 WPF 部分的那些类。只要符合几条基本规则，就可以使用 XAML 创建任何类的实例。本章后面将介绍如何使用 XAML 创建自定义类。

2.3.1 简单属性与类型转换器

前面已经介绍过，元素的特性设置相应用对象的属性。例如，我们为上面示例中的文本框设置了对齐方式、页边距和字体：

```
<TextBox Name="txtQuestion"
    VerticalAlignment="Stretch" HorizontalAlignment="Stretch"
    FontFamily="Verdana" FontSize="24" Foreground="Green" ... >
```

为使上面的设置起作用，`System.Windows.Controls.TextBox` 类必须提供以下属性：`VerticalAlignment`、`HorizontalAlignment`、`FontFamily`、`FontSize` 和 `Foreground`。后面几章将介绍这些属性的具体含义。

为使这个系统能够工作，XAML 解析器需要执行比表面上看起来更多的工作。XML 特性中的值总是纯文本字符串。但对象的属性可以是任何.NET 类型。在上面的示例中，有两个属性

为枚举类型(VerticalAlignment 属性和 HorizontalAlignment 属性)、一个为字符串类型(FontFamily 属性)、一个为整型(FontSize 属性)，还有一个为 Brush 对象(Foreground 属性)。

为了关联字符串值和非字符串属性，XAML 解析器需要执行转换。由类型转换器执行转换，类型转换器是从.NET 1.0 起就已经引入的.NET 基础结构的一个基本组成部分。

实际上，类型转换器在这个过程中扮演着重要角色——提供了实用的方法，这些方法可将特定的.NET 数据类型转换为任何其他.NET 类型，或将其他任何.NET 类型转换为特定的数据类型，比如这种情况下的字符串类型。XAML 解析器通过以下两个步骤来查找类型转换器：

(1) 检查属性声明，查找 TypeConverter 特性(如果提供了 TypeConverter 特性，该特性将指定哪个类可执行转换)。例如，当使用诸如 Foreground 这样的属性时，.NET 将检查 Foreground 属性的声明。

(2) 如果在属性声明中没有 TypeConverter 特性，XAML 解析器将检查对应数据类型的类声明。例如，Foreground 属性使用一个 Brush 对象。由于 Brush 类使用 TypeConverter(typeof(BrushConverter)) 特性声明进行了修饰，因此 Brush 类及其子类使用 BrushConverter 类型转换器。

如果属性声明或类声明都没有与其关联的类型转换器，XAML 解析器会生成错误。

这个系统简单灵活。如果在类层次上设置一个类型转换器，该转换器将应用到所有使用这个类的属性上。另一方面，如果希望为某个特定属性微调类型转换方式，那么可以在属性声明中改用 TypeConverter 特性。

在代码中使用类型转换器从技术角度看也是可行的，但语法有些复杂。直接设置属性几乎总是更好一些——不仅速度更快，而且可以避免一些因为键入非法字符串而产生的错误，这些错误只有在运行时才会被发现(这个问题不影响 XAML，因为 XAML 是在编译期间进行解析和验证的)。当然，在为 WPF 元素设置属性前，需要了解更多关于基本的 WPF 属性和数据类型方面的内容——这些内容将在后续几章中逐渐学习。

注意:

与所有基于 XML 的语言一样，XAML 也区分大小写。这意味着不能用<button>替代<Button>。然而，类型转换器通常不区分大小写，这意味着 Foreground="White" 和 Foreground="white"具有相同的效果。

2.3.2 复杂属性

虽然类型转换器便于使用，但它们不能解决所有的实际问题。例如，有些属性是完备的对象，这些对象具有自己的一组属性。尽管创建供类型转换器使用的字符串表示形式是可能的，但使用这种方法时语法可能十分复杂，并且容易出错。

幸运的是，XAML 提供了另一种选择：属性元素语法(property-element syntax)。使用属性元素语法，可添加名称形式为 Parent.PropertyName 的子元素。例如，Grid 控件有一个 Background 属性，该属性允许提供用于绘制控件背景区域的画刷。如果希望使用更复杂的画刷——比单一固定颜色填充更高级的画刷——就需要添加名为 Grid.Background 的子标签，如下所示：

```
<Grid Name="grid1">
  <Grid.Background>
    ...
  </Grid.Background>
  ...

```

```
</Grid>
```

真正起作用的重要细节是元素名中的句点(.)。这个句点把该属性和其他类型的嵌套内容区分开来。

还有一个细节，即一旦识别出想要配置的复杂属性，该如何设置呢？这里有一个技巧：可在嵌套元素内部添加其他标签来实例化特定的类。在 Eight Ball Answer 示例中(如图 2-1 所示)，用渐变颜色填充背景。为了定义所需的渐变颜色，需要创建 LinearGradientBrush 对象。

根据 XAML 规则，可使用名为 LinearGradientBrush 的元素创建 LinearGradientBrush 对象：

```
<Grid Name="grid1">
<Grid.Background>
<LinearGradientBrush>
</LinearGradientBrush>
</Grid.Background>
...
</Grid>
```

LinearGradientBrush 类是 WPF 名称空间集合中的一部分，所以可为标签继续使用默认的 XML 名称空间。

但是，只是创建 LinearGradientBrush 对象还不够——还需要为其指定渐变的颜色。通过使用 GradientStop 对象的集合填充 LinearGradientBrush.GradientStops 属性可完成这一工作。同样，由于 GradientStops 属性太复杂，因此不能通过一个简单的特性值设置该属性。需要改用属性元素语法：

```
<Grid Name="grid1">
<Grid.Background>
<LinearGradientBrush>
<LinearGradientBrush.GradientStops>
</LinearGradientBrush.GradientStops>
</LinearGradientBrush>
</Grid.Background>
...
</Grid>
```

最后，可使用一系列 GradientStop 对象填充 GradientStops 集合。每个 GradientStop 对象都有 Offset 和 Color 属性。可使用普通的属性-特性语法提供这两个值：

```
<Grid Name="grid1">
<Grid.Background>
<LinearGradientBrush>
<LinearGradientBrush.GradientStops>
<GradientStop Offset="0.00" Color="Red" />
<GradientStop Offset="0.50" Color="Indigo" />
<GradientStop Offset="1.00" Color="Violet" />
</LinearGradientBrush.GradientStops>
</LinearGradientBrush>
</Grid.Background>
...
</Grid>
```

注意:

可为任何属性使用属性元素语法。但如果属性具有合适的类型转换器，通常使用更简单的属性-特性方式，这样代码会更加简洁。

任何 XAML 标签集合都可以用一系列执行相同任务的代码语句代替。上面显示的使用所选的渐变颜色填充背景的标签，与以下代码是等价的：

```
LinearGradientBrush brush = new LinearGradientBrush();

GradientStop gradientStop1 = new GradientStop();
gradientStop1.Offset = 0;
gradientStop1.Color = Colors.Red;
brush.GradientStops.Add(gradientStop1);

GradientStop gradientStop2 = new GradientStop();
gradientStop2.Offset = 0.5;
gradientStop2.Color = Colors.Indigo;
brush.GradientStops.Add(gradientStop2);

GradientStop gradientStop3 = new GradientStop();
gradientStop3.Offset = 1;
gradientStop3.Color = Colors.Violet;
brush.GradientStops.Add(gradientStop3);

grid1.Background = brush;
```

2.3.3 标记扩展

对大多数属性而言，XAML 属性语法可以工作得非常好。但有些情况下，不可能硬编码属性值。例如，可能希望将属性值设置为一个已经存在的对象，或者可能希望通过将一个属性绑定到另一个控件来动态地设置属性值。这两种情况都需要使用标记扩展——一种以非常规的方式设置属性的专门语法。

标记扩展可用于嵌套标签或 XML 特性中(用于 XML 特性的情况更常见)。当用在特性中时，它们总是被花括号 {} 包围起来。例如，下面的标记演示了如何使用标记扩展，它允许引用另一个类中的静态属性：

```
<Button ... Foreground="{x:Static SystemColors.ActiveCaptionBrush}" >
```

标记扩展使用 {标记扩展类 参数} 语法。在上面的示例中，标记扩展是 StaticExtension 类(根据约定，在引用扩展类时可以省略最后一个单词 Extension)。x 前缀指示在 XAML 名称空间中查找 StaticExtension 类。还有一些标记扩展是 WPF 名称空间的一部分，它们不需要 x 前缀。

所有标记扩展都由继承自 System.Windows.Markup.MarkupExtension 基类的类实现。 MarkupExtension 基类十分简单——它提供了一个简单的 ProvideValue() 方法来获取所期望的数值。换句话说，当 XAML 解析器遇到上述语句时，它将创建 StaticExtension 类的一个实例(传递字符串 SystemColors.ActiveCaptionBrush 作为构造函数的参数)，然后调用 ProvideValue() 方法获取

`SystemColors.ActiveCaptionBrush` 静态属性返回的对象。最后使用检索的对象设置 `cmdAnswer` 按钮的 `Foreground` 属性。

这段 XAML 的最终结果与下面的相同：

```
cmdAnswer.Foreground = SystemColors.ActiveCaptionBrush;
```

因为标记扩展映射为类，所以它们也可用作嵌套属性，与上一节中学过的一样。例如，可以像下面这样为 `Button.Foreground` 属性使用 `StaticExtension` 标记扩展：

```
<Button ... >
  <Button.Foreground>
    <x:Static Member="SystemColors.ActiveCaptionBrush"></x:Static>
  </Button.Foreground>
</Button>
```

根据标记扩展的复杂程度，以及想要设置的属性数量，这种语法有时更简单。

和大多数标记扩展一样，`StaticExtension` 需要在运行时赋值，因为只有在运行时才能确定当前的系统颜色。一些标记扩展可在编译时评估。这些扩展包括 `NullExtension`(该扩展构造表示.NET 类型的对象)。在本书中，您将看到许多使用标记扩展的例子，特别是在使用资源和数据绑定时。

2.3.4 附加属性

除普通属性外，XAML 还包括附加属性(attached property)的概念——附加属性是可用于多个控件但在另一个类中定义的属性。在 WPF 中，附加属性常用于控件布局。

下面解释附加属性的工作原理。每个控件都有各自固有的属性(例如，文本框有其特定的字体、文本颜色和文本内容，这些是通过 `Fontfamily`、`Foreground` 和 `Text` 属性指定的)。当在容器中放置控件时，根据容器的类型控件会获得额外特征(例如，如果在网格中放置一个文本框，就需要选择文本框放在网格控件中的哪个单元格中)。使用附加属性设置这些附加的细节。

附加属性始终使用包含两个部分的命名形式：定义类型.属性名。这种包含两个部分的命名语法使 XAML 解析器能够区分开普通属性和附加属性。

在 Eight Ball Answer 示例中，通过附加属性在(不可见)网格的每一行中放置各个控件：

```
<TextBox ... Grid.Row="0">
  [Place question here.]
</TextBox>

<Button ... Grid.Row="1">
  Ask the Eight Ball
</Button>

<TextBox ... Grid.Row="2">
  [Answer will appear here.]
</TextBox>
```

附加属性根本不是真正的属性。它们实际上被转换为方法调用。XAML 解析器采用以下形式调用静态方法：`DefiningType.SetPropertyName()`。例如，在上面的 XAML 代码段中，定义类型是 `Grid` 类，并且属性是 `Row`，所以解析器调用 `Grid.SetRow()` 方法。

当调用 `SetPropertyName()` 方法时，解析器传递两个参数：被修改的对象以及指定的属性值。

例如，当为 TextBox 控件设置 Grid.Row 属性时，XAML 解析器执行以下代码：

```
Grid.SetRow(txtQuestion, 0);
```

这种方式(调用定义类型的一个静态方法)隐藏了实际发生的操作，使用起来非常方便。乍一看，这些代码好像将行号保存在 Grid 对象中。但行号实际上保存在应用它的对象中——对于上面的示例，就是 TextBox 对象。

这种技巧之所以能够奏效，是因为与其他所有 WPF 控件一样，TextBox 控件继承自 DependencyObject 基类。从第 4 章将可以了解到，DependencyObject 类旨在存储实际上没有限制的依赖项属性的集合(前面讨论的附加属性是特殊类型的依赖项属性)。

实际上，Grid.SetRow()方法是和 DependencyObject.SetValue()方法调用等价的简化操作，如下所示：

```
txtQuestion.SetValue(Grid.RowProperty, 0);
```

附加属性是 WPF 的核心要素。它们充当通用的可扩展系统。例如，通过将 Row 属性定义为附加属性，可确保任何控件都可以使用它。另一个选择是将该属性作为基类的一部分，例如，作为 FrameworkElement 类的一部分，但这样做很复杂。因为只有在特定情况下(在这个示例中，是当在 Grid 内部使用元素的时候)有些属性才有意义，如果将它们作为基类的一部分，不仅会使公共接口变得十分杂乱，而且也不能添加需要新属性的新类型的容器。

2.3.5 嵌套元素

正如您所看到的，XAML 文档被排列成一棵巨大的嵌套的元素树。在当前示例中，Window 元素包含 Grid 元素，Grid 元素又包含 TextBox 元素和 Button 元素。

XAML 让每个元素决定如何处理嵌套的元素。这种交互使用下面三种机制中的一种进行中转，而且求值的顺序也是下面列出这三种机制的顺序：

- 如果父元素实现了 IList 接口，解析器将调用 IList.Add()方法，并且为该方法传入子元素作为参数。
- 如果父元素实现了 IDictionary 接口，解析器将调用 IDictionary.Add()方法，并且为该方法传递子元素作为参数。当使用字典集合时，还必须设置 x:Key 特性以便为每个条目指定键名。
- 如果父元素使用 ContentProperty 特性进行修饰，解析器将使用子元素设置对应的属性。

例如，您已经在本章前面的示例中看到过 LinearGradientBrush 画刷如何使用如下所示的语法，从而包含 GradientStop 对象集合：

```
<LinearGradientBrush>
  <LinearGradientBrush.GradientStops>
    <GradientStop Offset="0.00" Color="Red" />
    <GradientStop Offset="0.50" Color="Indigo" />
    <GradientStop Offset="1.00" Color="Violet" />
  </LinearGradientBrush.GradientStops>
</LinearGradientBrush>
```

因为包含一个句点，所以 XAML 解析器知道 LinearGradientBrush.GradientStops 是复杂属性。但它需要以稍有不同的方式处理内部的标签(即三个 GradientStop 元素)。在这个示例中，解析器知道 GradientStops 属性返回一个 GradientStopCollection 对象，而且 GradientStopCollection 类实现了 IList 接口。因此，解析器假定(也正是如此)应当使用 IList.Add()方法将每个 GradientStop 对象添加

到集合中：

```
GradientStop gradientStop1 = new GradientStop();
gradientStop1.Offset = 0;
gradientStop1.Color = Colors.Red;
IList list = brush.GradientStops;
list.Add(gradientStop1);
```

有些属性可支持多种类型的集合。在这种情况下，需要添加一个标签来指定集合类，如下所示：

```
<LinearGradientBrush>
  <LinearGradientBrush.GradientStops>
    <GradientStopCollection>
      <GradientStop Offset="0.00" Color="Red" />
      <GradientStop Offset="0.50" Color="Indigo" />
      <GradientStop Offset="1.00" Color="Violet" />
    </GradientStopCollection>
  </LinearGradientBrush.GradientStops>
</LinearGradientBrush>
```

注意：

如果集合默认为 null，那么需要包含用于指定集合类的标签，以便创建集合对象。如果有一个默认的集合实例而且只需要为它填充元素，那么可以忽略这一部分。

嵌套的内容并非总是指定为集合。例如，分析以下包含其他几个控件的 Grid 元素：

```
<Grid Name="grid1">
  ...
  <TextBox Name="txtQuestion" ... >
  ...
</TextBox>
<Button Name="cmdAnswer" ... >
  ...
</Button>
<TextBox Name="txtAnswer" ... >
  ...
</TextBox>
</Grid>
```

这些嵌套的标签没有包含句点，因此并未对应于复杂属性。而且，Grid 控件也不是集合，所以它也就没有实现 IList 或 IDictionary 接口。Grid 控件支持 ContentProperty 特性，该特性指出应当接收任意嵌套内容的属性。从技术角度看，ContentProperty 特性被应用于 Panel 类，而 Grid 类继承自 Panel 类，如下所示：

```
[ContentPropertyAttribute("Children")]
public abstract class Panel
```

这表明应使用任何嵌套元素来设置 Children 属性。XAML 解析器根据是否是集合属性(集合属性实现了 IList 或 IDictionary 接口)，采用不同方式处理内容属性。因为 Panel.Children 属性返回一个 UIElementCollection 对象，而且 UIElementCollection 类实现了 IList 接口，所以解析器使用 IList.Add()方法将嵌套的内容添加到网格中。

换句话说，当 XAML 解析器遇到上面的标记时，会为每个嵌套的元素创建实例，并使用 Grid.Children.Add()方法将创建的实例传递给 Grid 控件。

```

txtQuestion = new TextBox();
...
grid1.Children.Add(txtQuestion);

cmdAnswer = new Button();
...
grid1.Children.Add(cmdAnswer);

txtAnswer = new TextBox();
...
grid1.Children.Add(txtAnswer);

```

下一步的具体操作完全取决于控件实现内容属性的方式。Grid 控件在不可见的行和列布局中显示它所包含的所有控件，详见第 3 章。

WPF 中经常使用 ContentProperty 特性。该特性不仅用于容器控件(如 Grid 控件)和那些包含可视化条目集合的控件(如 ListBox 和 TreeView 控件)，也用于包含单一内容的控件。例如，TextBox 和 Button 控件只能包含一个元素或一段文本，但它们都使用内容属性来处理嵌套的内容，如下所示：

```

<TextBox Name="txtQuestion" ... >
    [Place question here.]
</TextBox>
<Button Name="cmdAnswer" ... >
    Ask the Eight Ball
</Button>
<TextBox Name="txtAnswer" ... >
    [Answer will appear here.]
</TextBox>

```

TextBox 类使用 ContentProperty 特性来标识 TextBox.Text 属性。Button 类使用 ContentProperty 特性来标识 Button.Content 属性。XAML 解析器使用提供的文本来设置这些属性。

TextBox.Text 属性只接受字符串。但 Button.Content 属性可使用更多有趣的内容。正如第 6 章中介绍的那样，Content 属性可接受任何元素。例如，下面的按钮包含一个图形对象：

```

<Button Name="cmdAnswer" ... >
    <Rectangle Fill="Blue" Height="10" Width="100" />
</Button>

```

Text 和 Content 属性没有使用集合，因此只能包含一段内容。例如，如果试图在一个按钮中嵌套多个元素，XAML 解析器将抛出异常。如果提供非文本内容(比如一个 Rectangle 对象)，解析器也会抛出异常。

注意：

作为一条经验法则，所有继承自 ContentControl 类的控件只允许包含单一的嵌套元素。所有继承自 ItemsControl 类的控件都允许包含一个条目集合，该集合映射到控件的某些部分(例如条目列表或节点树)。所有继承自 Panel 类的控件都是用来组织多组控件的容器。ContentControl、ItemsControl 和 Panel 基类都使用 ContentProperty 特性。

2.3.6 特殊字符与空白

XAML 受到 XML 规则的限制。例如，XML 特别关注一些特殊字符，如&、<和>。如果

试图使用这些字符设置元素的内容，将遇到麻烦，因为 XAML 解析器认为您正在处理其他事情——例如，创建嵌套的元素。

例如，假设需要创建一个包含<Click Me>文本的按钮。下面的标记是无法奏效的：

```
<Button ... >
  <Click Me>
</Button>
```

此处的问题在于，上面的标记看起来像正在试图创建一个名为 Click，并且带有<Click>文本的元素。解决问题的方法是用实体引用代替那些特殊字符，实体引用是 XAML 解析器能够正确解释的特定字符编码。表 2-1 列出了可能选用的字符实体。注意，只有当使用特性设置属性值时，才需要使用引号字符实体，因为引号用于指示特性值的开始和结束。

表 2-1 XAML 字符实体

特 殊 字 符	字 符 实 体
小于号(<)	<
大于号(>)	>
&符号(&)	&
引号(")	"

下面是使用恰当字符实体的正确标记：

```
<Button ... >
  &lt;Click Me&gt;
</Button>
```

当 XAML 解析器遇到这些标记时，它能正确地理解到您希望添加<Click Me>文本，而且解析器为 Button.Content 属性传递具有相应内容的字符串，字符串内容将包含完整的尖括号。

注意：

这一限制只是 XAML 的细节，如果希望在代码中设置 Button.Content 属性，那么不受影响。当然，C# 有它自己的特殊字符(反斜杠)，因为相同的原因，在字符串字面量中该特殊字符必须被转义。

特殊字符并非使用 XAML 的唯一障碍。另一个问题是空白的处理。默认情况下，XAML 折叠所有空白，这意味着包含空格、Tab 键以及硬回车的长字符串将被转换为单个空格。而且，如果在元素内容之前或之后添加空白，将完全忽略这个空格。在 Eight Ball Answer 示例中您将看到这种情形。在按钮和两个文本框中的文本，使用硬回车字符从 XAML 标签中分离出来，并使用 Tab 字符使标记更加清晰易读。但多余的空格不会再显示在用户界面中。

有时这并不是所期望的结果。例如，可能希望在按钮文本中包含一系列空格。在这种情况下，需要为元素使用 xml:space="preserve" 特性。

xml:space 特性是 XML 标准的一部分，是一个要么包括全部、要么什么都不包括的设置。一旦使用了该设置，元素内的所有空白字符都将被保留。比如下面的标记：

```
<TextBox Name="txtQuestion" xml:space="preserve" ...>
  [There is a lot of space inside these quotation marks "      ".]
```

```
</TextBox>
```

在这个示例中，文本框中的文本在实际文本之前将包含硬回车和 Tab 字符等。在显示的文本中也将包含一系列空格并且在文本之后还跟有一个硬回车字符。

如果只想保留内部的空格，那么需要使用不很清晰的标记：

```
<TextBox Name="txtQuestion" xml:space="preserve" ...>[There is a lot of space inside these quotation marks " .]</TextBox>
```

上面这个技巧是为了确保在开始符号>和具体内容之间，以及具体内容和结束符号<之间没有空白。

同样，该问题只存在于 XAML 标记中。如果通过代码设置文本框中的文本，所有空格都将被使用。

2.3.7 事件

到目前为止介绍的所有特性都被映射为属性。然而，特性也可用于关联事件处理程序。用于关联事件处理程序的语法为：事件名="事件处理方法名"。

例如，Button 控件提供了 Click 事件。可使用如下所示的标记关联事件处理程序：

```
<Button ... Click="cmdAnswer_Click">
```

上面的标记假定在代码隐藏类中有名为 cmdAnswer_Click 的方法。事件处理程序必须具有正确的签名(也就是说，必须匹配 Click 事件的委托)。下面是一个符合要求的方法：

```
private void cmdAnswer_Click(object sender, RoutedEventArgs e)
{
    this.Cursor = Cursors.Wait;

    // Dramatic delay...
    System.Threading.Thread.Sleep(TimeSpan.FromSeconds(3));

    AnswerGenerator generator = new AnswerGenerator();
    txtAnswer.Text = generator.GetRandomAnswer(txtQuestion.Text);
    this.Cursor = null;
}
```

WPF 中的事件模型和其他类型的.NET 应用程序的事件模型不同。WPF 事件模型依赖于事件。详见第 5 章。

许多情况下，将使用特性为同一元素设置属性和关联事件处理程序。WPF 总是遵循以下顺序：首先设置 Name 属性(如果设置的话)，然后关联任意事件处理程序，最后设置其他属性。这意味着，所有对属性变化做出响应的事件处理程序在第一次设置属性时都会被触发。

注意：

也可以使用 Code 元素直接在 XAML 文档中嵌入代码(如事件处理程序)。然而，这是一项令人灰心的技术，在任何实际的 WPF 应用程序中不需要使用这种技术。Visual Studio 不支持该技术，在本书中也不对其予以讨论。

当添加事件处理程序特性时，Visual Studio 的智能感知功能可提供极大的帮助。一旦输入

等号(例如，在<Button>元素中输入“Click=”之后)，Visual Studio 会显示一个包含在代码隐藏类中的所有合适的事件处理程序的下拉列表，如图 2-2 所示。如果需要创建一个新的事件处理程序来处理这一事件，只需从列表顶部选择<New Event Handler>选项。此外，也可以使用 Properties 窗口的 Events 选项卡来关联和创建事件处理程序。



图 2-2 使用 Visual Studio 的智能感知功能关联事件

2.3.8 完整的 Eight Ball Answer 示例

现在已经学习了 XAML 的基本内容，应该可以完全理解在图 2-1 中显示的窗口的定义。下面是完整的 XAML 标记：

```
<Window x:Class="EightBall.Window1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Eight Ball Answer" Height="328" Width="412" >
    <Grid Name="grid1">
        <Grid.RowDefinitions>
            <RowDefinition Height="*" />
            <RowDefinition Height="Auto" />
            <RowDefinition Height="*" />
        </Grid.RowDefinitions>
        <TextBox VerticalAlignment="Stretch" HorizontalAlignment="Stretch"
            Margin="10,10,13,10" Name="txtQuestion"
            TextWrapping="Wrap" FontFamily="Verdana" FontSize="24"
            Grid.Row="0">
            [Place question here.]
        </TextBox>
        <Button VerticalAlignment="Top" HorizontalAlignment="Left"
            Margin="10,0,0,20" Width="127" Height="23" Name="cmdAnswer"
            Click="cmdAnswer_Click" Grid.Row="1">
            Ask the Eight Ball
        </Button>
        <TextBox VerticalAlignment="Stretch" HorizontalAlignment="Stretch"
            Margin="10,10,13,10" Name="txtAnswer" TextWrapping="Wrap"
            IsReadOnly="True" FontFamily="Verdana" FontSize="24" Foreground="Green"
            Grid.Row="2">
    
```

```

[Answer will appear here.]
</TextBox>

<Grid.Background>
    <LinearGradientBrush>
        <LinearGradientBrush.GradientStops>
            <GradientStop Offset="0.00" Color="Red" />
            <GradientStop Offset="0.50" Color="Indigo" />
            <GradientStop Offset="1.00" Color="Violet" />
        </LinearGradientBrush.GradientStops>
    </LinearGradientBrush>
</Grid.Background>
</Grid>
</Window>

```

请记住，可能不会为整个用户界面手动编写 XAML——这样做将是非常单调乏味的。但可能会编辑 XAML 标记，对界面进行某些修改，而在设计器中完成这些修改可能是很笨拙的。您可能还会发现通过分析 XAML 可以很好地理解窗口的工作原理。

2.4 使用其他名称空间中的类型

前面已经介绍了如何在 XAML 中使用 WPF 中的类来创建基本的用户界面。但 XAML 是实例化.NET 对象的通用方法，包括那些位于其他非 WPF 名称空间以及自己创建的名称空间中的对象。

创建那些不是用于在 XAML 窗口中显示的对象听起来像是多余的，但在很多情况下这是需要的。一个例子是，当使用数据绑定并希望在某个控件上显示从其他对象提取的信息时。另一个例子是希望使用非 WPF 对象为 WPF 对象设置属性时。

例如，可使用数据对象填充 WPF 的 ListBox 控件。ListBox 控件将调用 ToString()方法来获取文本，以便在列表中显示每个条目(或者使用更好的列表，可创建数据模板来提取多段信息，并将它们设置成合适的格式。这一技术将在第 20 章中介绍)。

为使用未在 WPF 名称空间中定义的类，需要将.NET 名称空间映射到 XML 名称空间。XAML 有一种特殊的语法可用于完成这一工作，该语法如下所示：

```
xmlns:Prefix="clr-namespace:Namespace;assembly=AssemblyName"
```

通常，在 XAML 文档的根元素中，在紧随声明 WPF 和 XAML 名称空间的特性之后放置这个名称空间。还需要使用适当的信息填充三个斜体部分，这三部分的含义如下：

- **Prefix** 是希望在 XAML 标记中用于指示名称空间的 XML 前缀。例如，XAML 语言使用 x 前缀。
- **Namespace** 是完全限定的.NET 名称空间的名称。
- **AssemblyName** 是声明类型的程序集，没有.dll 扩展名。这个程序集必须在项目中引用。如果希望使用项目程序集，可忽略这一部分。

例如，下面的标记演示了如何访问 System 名称空间中的基本类型，并将其映射为前缀 sys：

```
xmlns:sys="clr-namespace:System;assembly=mscorlib"
```

下面的标记演示了如何访问当前项目在 MyProject 名称空间中声明的类型，并将它们映射为前缀 local：

```
xmlns:local="clr-namespace:MyNamespace"
```

现在，为了创建其中一个名称空间中的类的实例，可使用名称空间前缀：

```
<local:MyObject ...></local:MyObject>
```

提示：

请记住，可使用任何想要使用的名称空间前缀，只要在整个 XAML 文档中保持一致即可。但 sys 和 local 前缀通常在导入 System 名称空间和当前项目的名称空间时使用。您可以在本书中看到使用这两个前缀的情形。

理想情况是，希望在 XAML 中使用的每个类都有无参构造函数。如果具有无参构造函数，XAML 解析器就可创建对应的对象，设置其属性，并关联所提供的任何事件处理程序。XAML 不支持有参构造函数，而且 WPF 中的所有元素都包含无参构造函数。此外，需要能够使用公共属性设置您所期望的所有细节。XAML 不允许设置公共字段或调用方法。

如果想要使用的类没有无参构造函数，就有一些限制。如果试图创建简单的基本类型(如字符串、日期或数字类型)，可提供数据的字符串表示形式作为标签中的内容。XAML 解析器接着将使用类型转换器将字符串转换为合适的对象。下面列举一个使用 DateTime 结构的例子：

```
<sys:DateTime>10/30/2010 4:30 PM</sys:DateTime>
```

因为 DateTime 类使用 TypeConverter 特性将自身关联到 DateTimeConverter 类，所以上面的标记可以奏效。DateTimeConverter 类知道这个字符串是合法的 DateTime 对象，并对其进行转换。当使用该技术时，不能使用特性为您的对象设置任何属性。

如果想创建没有无参构造函数的类，但没有可供使用的适当类型转换器，那将是很不幸的。

注意：

一些开发人员通过创建自定义的封装器类来克服这些限制。例如，FileStream 类没有包含无参构造函数。然而，您可以创建具有无参构造函数的封装器类。封装器类在其构造函数中创建所期望的 FileStream 对象，检索所需的信息，然后关闭 FileStream 对象。此类解决方案通常并不理想，因为是在类的构造函数中硬编码信息，并使异常处理变得复杂。在大多数情况下，更好的方法是使用少许事件处理代码来控制对象，而完全不使用 XAML。

下面的示例将所有这些概念融合在一起。将 sys 前缀映射到 System 名称空间，并使用 System 名称空间创建三个 DateTime 对象，然后用这三个 DataTeime 对象填充一个列表：

```
<Window x:Class="WindowsApplication1.Window1"
       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"
       Width="300" Height="300"
       >
```

```

<ListBox>
  <ListBoxItem>
    <sys:DateTime>10/13/2013 4:30 PM</sys:DateTime>
  </ListBoxItem>
  <ListBoxItem>
    <sys:DateTime>10/29/2013 12:30 PM</sys:DateTime>
  </ListBoxItem>
  <ListBoxItem>
    <sys:DateTime>10/30/2013 2:30 PM</sys:DateTime>
  </ListBoxItem>
</ListBox>
</Window>

```

2.5 加载和编译 XAML

前面已经介绍过，尽管 XAML 和 WPF 这两种技术具有相互补充的作用，但它们也是相互独立的。因此，完全可以创建不使用 XAML 的 WPF 应用程序。

总之，可使用三种不同的编码方式来创建 WPF 应用程序：

- **只使用代码。**这是在 Visual Studio 中为 Windows 窗体应用程序使用的传统方法。它通过代码语句生成用户界面。
- **使用代码和未经编译的标记(XAML)。**这种具体方式对于某些特殊情况是很有意义的，例如创建高度动态化的用户界面。这种方式在运行时使用 System.Windows.Markup 名称空间中的 XamlReader 类，从 XAML 文件中加载部分用户界面。
- **使用代码和编译过的标记(BAML)。**对于 WPF 而言这是一种更好的方式，也是 Visual Studio 支持的一种方式。这种方式为每个窗口创建一个 XAML 模板，这个 XAML 模板被编译为 BAML，并嵌入到最终的程序集中。编译过的 BAML 在运行时被提取出来，用于重新生成用户界面。

接下来的几节将深入分析这三种方式及其工作原理。您将看到如何在浏览器中打开松散的、没有使用任何代码的 XAML 文件。

2.5.1 只使用代码

对于编写 WPF 应用程序，只使用代码进行开发而不使用任何 XAML 的做法并不常见(但是仍然完全支持)。只使用代码进行开发的明显缺点在于，可能会使编写 WPF 应用程序成为极端乏味的工作。WPF 控件没有包含参数化的构造函数，因此即使为窗口添加一个简单按钮也需要编写几行代码。

只使用代码进行开发的一个潜在的优点是可以随意定制应用程序。例如，可根据数据库记录中的信息生成充满输入控件的窗体，或可根据当前的用户酌情添加或替换控件。需要的所有内容只不过是少量的条件逻辑。相比之下，如果使用 XAML 文档，它们只能作为固定不变的资源嵌入到程序集中。

注意：

尽管不太可能创建只使用代码的 WPF 应用程序，但当需要一个自适应的用户界面模块时，您可能会通过只使用代码的方式来创建 WPF 控件。

以下代码用于生成一个普通窗口，该窗口包含一个按钮和一个事件处理程序(见图 2-3)。在创建窗口时，构造函数调用 InitializeComponent()方法，该方法实例化并配置这个按钮和窗体，并连接(hook up)事件处理程序。



图 2-3 包含一个按钮的窗口

注意：

要创建该示例，必须从头编写 Window1 类(右击 Solution Explorer 中的项目，然后从上下文菜单中选择 Add | Class 菜单项)。不能选择 Add | Window 菜单项，因为这将为窗口添加一个代码文件和一个 XAML 模板，并带有自动生成的 InitializeComponent()方法。

```
using System.Windows;
using System.Windows.Controls;
using System.Windows.Markup;

public class Window1 : Window
{
    private Button button1;

    public Window1()
    {
        InitializeComponent();
    }

    private void InitializeComponent()
    {
        // Configure the form.
        this.Width = this.Height = 285;
        this.Left = this.Top = 100;
        this.Title = "Code-Only Window";

        // Create a container to hold a button.
        DockPanel panel = new DockPanel();

        // Create the button.
        button1 = new Button();
    }
}
```

```

button1.Content = "Please click me.";
button1.Margin = new Thickness(30);

// Attach the event handler.
button1.Click += button1_Click;

// Place the button in the panel.
IAddChild container = panel;
container.AddChild(button1);

// Place the panel in the form.
container = this;
container.AddChild(panel);
}

private void button1_Click(object sender, RoutedEventArgs e)
{
    button1.Content = "Thank you.";
}
}

```

从概念上讲，本例中的 Window1 类更像传统的 Windows 窗体应用程序中的窗体。它继承自 Window 基类，并为每个控件添加一个私有成员变量。为清晰起见，该类在专门的 InitializeComponent()方法中执行初始化操作。

为启动该应用程序，可在 Main()方法中添加如下代码：

```

public class Program : Application
{
    [STAThread()]
    static void Main()
    {
        Program app = new Program();
        app.MainWindow = new Window1();
        app.MainWindow.ShowDialog();
    }
}

```

2.5.2 使用代码和未经编译的 XAML

使用 XAML 最有趣的方式之一是使用 XamlReader 类随时解析它。例如，假设开始时在一个名为 Window1.xaml 的文件中使用下面的 XAML 内容：

```

<DockPanel xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation">
    <Button Name="button1" Margin="30">Please click me.</Button>
</DockPanel>

```

在运行时，可将上面的内容加载到一个已经存在的窗口中，以便创建一个与图 2-3 中显示的窗口相同的窗口。下面是完成这一工作的代码：

```

using System.Windows;
using System.Windows.Controls;
using System.Windows.Markup;

```

```

using System.IO;

public class Window1 : Window
{
    private Button button1;

    public Window1()
    {
        InitializeComponent();
    }

    public Window1(string xamlFile)
    {
        // Configure the form.
        this.Width = this.Height = 285;
        this.Left = this.Top = 100;
        this.Title = "Dynamically Loaded XAML";

        // Get the XAML content from an external file.
        DependencyObject rootElement;
        using (FileStream fs = new FileStream(xamlFile, FileMode.Open))
        {
            rootElement = (DependencyObject)XamlReader.Load(fs);
        }

        // Insert the markup into this window.
        this.Content = rootElement;

        // Find the control with the appropriate name.
        button1 = (Button)LogicalTreeHelper.FindLogicalNode(rootElement, "button1");

        // Wire up the event handler.
        button1.Click += button1_Click;
    }

    private void button1_Click(object sender, RoutedEventArgs e)
    {
        button1.Content = "Thank you.";
    }
}

```

在此，构造函数接收 XAML 文件名作为参数(在这个示例中是 Window1.xaml)。然后构造函数打开一个 FileStream 对象，并使用 XamlReader.Load()方法将这个文件中的内容转换成 DependencyObject 对象，DependencyObject 是所有 WPF 控件继承的基类。DependencyObject 对象可放在任意类型的容器中(如面板)，但在这个示例中它被用作整个窗口的内容。

注意:

在这个示例中，从 XAML 文件中加载了一个元素——DockPanel 对象。同样可加载整个 XAML 窗口。在这种情况下，必须将 XamlReader.Load()方法返回的对象转换为 Window 类型，然后为了显示加载的窗口，调用它的 Show()方法或 ShowDialog()方法。

为操纵元素——如 Windows1.xaml 文件中的按钮，需要在动态加载的内容中查找相应的控件对象。LogicalTreeHelper 类可达到该目的，因为它具有查找一棵完整控件对象树的能力，它可以查找所需的许多层，直至找到具有指定名称的对象。然后将一个事件处理程序关联到 Button.Click 事件。

另一种方法是使用 FrameworkElement.FindName()方法。在这个示例中，根元素是 DockPanel 对象。与 WPF 窗口中的所有控件一样，DockPanel 类继承自 FrameworkElement 类，这意味着可使用如下等效的方法：

```
FrameworkElement frameworkElement = (FrameworkElement)rootElement;
button1 = (Button)frameworkElement.FindName("button1");
```

代替下面这行代码：

```
button1 = (Button)LogicalTreeHelper.FindLogicalNode(rootElement, "button1");
```

在这个示例中，Window1.xaml 文件和可执行的应用程序位于同一文件夹中，并一同发布。然而，尽管该文件没有被编译为应用程序的一部分，但仍可以将其添加到 Visual Studio 项目中。这样可以更方便地管理文件，并使用 Visual Studio 设计用户界面(假定使用.xaml 文件扩展名，从而使 Visual Studio 能够识别出该文档是 XAML 文档)。

如果使用这种方法，确保松散的 XAML 文件不会像传统的 XAML 文件那样被编译或嵌入到项目中。将文件添加到项目后，在 Solution Explorer 中选中该文件，然后使用 Properties 窗口将 Build Action 设置为 None，并将 Copy to Output Directory 设置为 Copy Always。

显然，先将 XAML 编译为 BAML，再在运行时加载 BAML，比动态加载 XAML 的效率高，当用户界面比较复杂时尤其如此。然而，这种编码模式为构建动态的用户界面提供了多种可能。例如，可创建通用的检测应用程序，从 Web 服务中读取窗体文件，然后显示相应的检测控件(标签、文本框和复选框等)。窗体文件可以是具有 WPF 标签的普通 XML 文档，使用 XamlReader 类将该文档加载到一个已经存在的窗体中。检测之后，为了收集结果，只需要枚举所有输入控件并提取它们的内容即可。

2.5.3 使用代码和编译过的 XAML

通过在图 2-1 中显示的 Eight Ball Answer 示例，您已经看到了 XAML 的最常用方式，并在本章中通篇进行了分析。这是 Visual Studio 使用的方法，它具有几个在本章中已经介绍过的优点：

- 有些内容可以自动生成。不必使用 LogicalTreeHelper 类进行 ID 查找，也不需要在代码中关联事件处理程序。
- 在运行时读取 BAML 比读取 XAML 的速度要快。
- 部署更简单。因为 BAML 作为一个或多个资源嵌入到程序集中，不会丢失。
- 可在其他程序中编辑 XAML 文件，例如设计工具。这为程序编程人员和设计人员之间更好地开展协作提供了可能(当使用未编译的 XAML 时，也能获得这个好处，如上一节所述)。

当编译 WPF 应用程序时，Visual Studio 使用分为两个阶段的编译处理过程。第一阶段将 XAML 文件编译为 BAML。例如，如果项目中包含名为 Window1.xaml 的文件，编译器将创建名为 Window1.baml 的临时文件，并将该文件放在项目文件夹的 obj/Debug 子文件夹中。同时，

使用选择的语言为窗口创建部分类。例如，如果使用 C#语言，编译器将在 obj\Debug 文件夹中创建名为 Window1.g.cs 的文件。g 代表生成的(generated)。

部分类包括如下三部分内容：

- 窗口中所有控件的字段。
- 从程序集中加载 BAML 的代码，由此创建对象树。当构造函数调用 InitializeComponent() 方法时将发生这种情况。
- 将恰当的控件对象指定给各个字段以及连接所有事件处理程序的代码。该过程是在名为 Connect() 的方法中完成的，BAML 解析器在每次发现一个已经命名的对象时调用该方法一次。

部分类不包含实例化和初始化控件的代码，因为这项任务由 WPF 引擎在使用 Application.LoadComponent() 方法处理 BAML 时执行。

注意：

在 XAML 编译期间，XAML 编译器需要创建部分类。只有当使用的编程语言支持.NET CodeDOM 模型时，才可能出现这个过程。C# 和 VB 支持 CodeDOM 模型，但如果使用一种第三方语言，那么在创建编译的 XAML 应用程序之前需确保该语言支持 CodeDOM 模型。

下面的 Window1.g.cs 文件(稍有删减)来自图 2-1 中显示的 Eight Ball Answer 示例：

```
public partial class Window1 : System.Windows.Window,
    System.Windows.Markup.IComponentConnector
{
    // The control fields.
    internal System.Windows.Controls.TextBox txtQuestion;
    internal System.Windows.Controls.Button cmdAnswer;
    internal System.Windows.Controls.TextBox txtAnswer;

    private bool _contentLoaded;

    // Load the BAML.
    public void InitializeComponent()
    {
        if (_contentLoaded)
        {
            return;
        }
        _contentLoaded = true;

        System.Uri resourceLocater = new System.Uri("window1.baml",
            System.UriKind.RelativeOrAbsolute);
        System.Windows.Application.LoadComponent(this, resourceLocater);
    }

    // Hook up each control.
    void System.Windows.Markup.IComponentConnector.Connect(int connectionId,
        object target)
    {
        switch (connectionId)
```

```

{
    case 1:
        txtQuestion = ((System.Windows.Controls.TextBox)(target));
        return;
    case 2:
        cmdAnswer = ((System.Windows.Controls.Button)(target));
        cmdAnswer.Click += new System.Windows.RoutedEventHandler(
            cmdAnswer_Click);
        return;
    case 3:
        txtAnswer = ((System.Windows.Controls.TextBox)(target));
        return;
}
this._contentLoaded = true;
}

}

```

当“从 XAML 到 BAML”的编译阶段结束后，Visual Studio 使用合适的语言编译器来编译代码和生成的部分类文件。对于 C# 应用程序而言，使用 csc.exe 编译器处理这一任务。编译过的代码会变成单个程序集(对于 Eight Ball Answer 示例，是 EightBall.exe 程序集)，而且每个窗口的 BAML 都作为独立资源被嵌入到程序集中。

2.5.4 只使用 XAML

前几节介绍了如何在基于代码的应用程序中使用 XAML。.NET 开发人员的大部分工作时间都将花费在这个方面。但也可能使用 XAML 文件而不创建任何代码，这称为松散的 XAML 文件。可直接在 Internet Explorer 浏览器中打开松散的 XAML 文件。

注意：

如果 XAML 文件使用了代码，就不能在 Internet Explorer 浏览器中打开它。但可以通过构建称为 XBAP 的基于浏览器的应用程序来突破这一限制。第 24 章将介绍如何创建基于浏览器的应用程序。

到目前为止，创建松散的 XAML 看起来好像没什么用处——毕竟，没有代码驱动的用户界面并无意义。但当浏览 XAML 时将发现几个完全声明的特性。这些特性包括动画、触发器、数据绑定和链接(链接可指向其他松散的 XAML 文件)等。使用这些特性，可构建一些非常简单的没有代码的 XAML 文件。它们看起来不像完整的应用程序，但可以完成比静态的 HTML 页面更多的工作。

为了测试松散的 XAML 页面，对.xaml 文件做如下修改：

- 删除根元素的 Class 特性。
- 删除关联事件处理程序的任意特性(如 Button.Click 特性)。
- 将打开和关闭标签的名称由 Window 改为 Page。IE 只能显示驻留的页面，不能显示单独窗口。

此后可双击 XAML 文件在 Internet Explorer 浏览器中加载。图 2-4 显示了一个修改过的 EightBall.xaml 页面，本章的下载代码中包含了该页面。可在顶部的文本框中输入内容，但由于

应用程序缺少代码隐藏文件，因此当单击按钮时什么也不会发生。如果希望创建功能更强大的、可包含代码的基于浏览器的应用程序，就需要使用将在第 24 章介绍的 XBAP 模型。

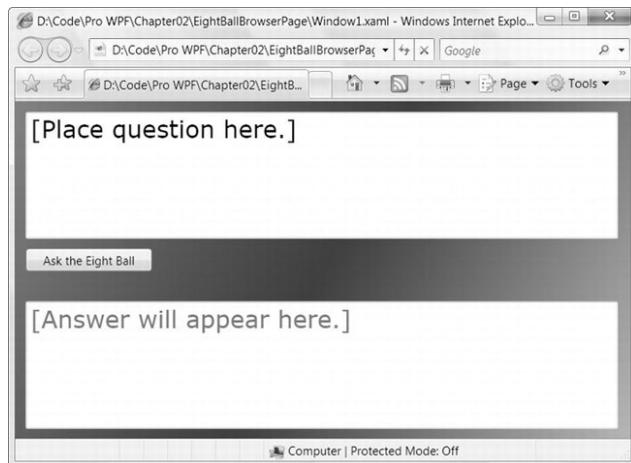


图 2-4 浏览器中的 XAML 页面

2.6 小结

本章分析了一个简单的 XAML 文件，同时分析了 XAML 的语法。下面列出本章中介绍的内容：

- 介绍了 XAML 的主要组成部分，如类型转换器、标记扩展和附加属性。
- 学习了如何连接可以处理由控件触发的事件的代码隐藏类。
- 介绍了将标准的 WPF 应用程序编译成可执行文件的编译过程，同时介绍了其他三种方式：只使用代码创建 WPF 应用程序、只使用 XAML 创建 WPF 页面以及在运行时手动加载 XAML。

尽管本章未能涵盖 XAML 标记的所有细节，但通过已经介绍的内容您足以理解 XAML 的所有优点。现在，请把注意力转移到 WPF 技术本身上，WPF 技术包含一些非常有趣而且令人惊奇的内容。下一章将介绍如何使用 WPF 布局面板将控件布置到真实的窗口中。

第 3 章



布 局

在任意用户界面设计中，有一半的工作是以富有吸引力、灵活实用的方式组织内容。但真正的挑战是确保界面布局能够恰到好处地适应不同的窗口尺寸。

WPF 用不同的容器(container)安排布局。每个容器有各自的布局逻辑——有些容器以堆栈方式布置元素，另一些容器在网格中不可见的单元格中排列元素，等等。在 WPF 中非常抵制基于坐标的布局，而是注重创建更灵活的布局，使布局能够适应内容的变化、不同的语言以及各种窗口尺寸。迁移到 WPF 的许多开发人员会觉得新布局系统令自己倍感惊奇——这也是开发人员面临的一个真正挑战。

本章将介绍 WPF 布局模型的工作原理，并且将开始使用基本的布局容器。为了学习 WPF 布局的基础知识，本章还将介绍几个通用的布局示例——从基本的对话框乃至可改变尺寸的拆分窗口。

3.1 理解 WPF 中的布局

在 Windows 开发人员设计用户界面的方式上，WPF 布局模型是一个重大改进。在 WPF 问世之前，Windows 开发人员使用刻板的基于坐标的布局将控件放到正确位置。在 WPF 中，这种方式虽然可行，但已经极少使用。大多数应用程序将使用类似于 Web 的流(flow)布局；在使用流布局模型时，控件可以扩大，并将其他控件挤到其他位置，开发人员能创建与显示分辨率和窗口大小无关的、在不同的显示器上正确缩放的用户界面；当窗口内容发生变化时，界面可调整自身，并且可以自如地处理语言的切换。要利用该系统的优势，首先需要进一步理解 WPF 布局模型的基本概念和假设。

3.1.1 WPF 布局原则

WPF 窗口只能包含单个元素。为在 WPF 窗口中放置多个元素并创建更贴近实用的用户界面，需要在窗口上放置一个容器，然后在这个容器中添加其他元素。

注意：

造成这一限制的原因是 Window 类继承自 ContentControl 类，在第 6 章中将进一步分析 ContentControl 类。

在 WPF 中，布局由您使用的容器来确定。尽管有多个容器可供选择，但“理想的”WPF 窗口需要遵循以下几条重要原则：

- 不应显式设定元素(如控件)的尺寸。元素应当可以改变尺寸以适合它们的内容。例如，当添加更多的文本时按钮应当能够扩展。可通过设置最大和最小尺寸来限制可以接受的控件尺寸范围。
- 不应使用屏幕坐标指定元素的位置。元素应当由它们的容器根据它们的尺寸、顺序以及(可选的)其他特定于具体布局容器的信息进行排列。如果需要在元素之间添加空白空间，可使用 Margin 属性。

提示：

以硬编码方式设定尺寸和位置是极其不当的处理方式，因为这会限制本地化界面的能力，并且会使界面更难处理动态内容。

- 布局容器的子元素“共享”可用的空间。如果空间允许，布局容器会根据每个元素的内容尽可能为元素设置更合理的尺寸。它们还会向一个或多个子元素分配多余的空间。
- 可嵌套的布局容器。典型的用户界面使用 Grid 面板作为开始，Grid 面板是 WPF 中功能最强大的容器，Grid 面板可包含其他布局容器，包含的这些容器以更小的分组排列元素，比如带有标题的文本框、列表框中的项、工具栏上的图标以及一列按钮等。

尽管对于这几条原则而言也有一些例外，但它们反映了 WPF 的总体设计目标。换句话说，如果创建 WPF 应用程序时遵循了这些原则，将会创建出更好的、更灵活的用户界面。如果不遵循这些原则，最终将得到不是很适合 WPF 的并且难以维护的用户界面。

3.1.2 布局过程

WPF 布局包括两个阶段：测量(measure)阶段和排列(arrange)阶段。在测量阶段，容器遍历所有子元素，并询问子元素它们所期望的尺寸。在排列阶段，容器在合适的位置放置子元素。

当然，元素未必总能得到最合适尺寸——有时容器没有足够大的空间以适应所含的元素。在这种情况下，容器为了适应可视化区域的尺寸，就必须剪裁不能满足要求的元素。在后面可以看到，通常可通过设置最小窗口尺寸来避免这种情况。

注意：

布局容器不能提供任何滚动支持。相反，滚动是由特定的内容控件——*ScrollViewer*——提供的，*ScrollViewer* 控件几乎可用于任何地方。在第 6 章中将学习 *ScrollViewer* 控件的相关内容。

3.1.3 布局容器

所有 WPF 布局容器都是派生自 System.Windows.Controls.Panel 抽象类的面板(见图 3-1)。Panel 类添加了少量成员，包括三个公有属性，表 3-1 列出了这三个公有属性的详情。

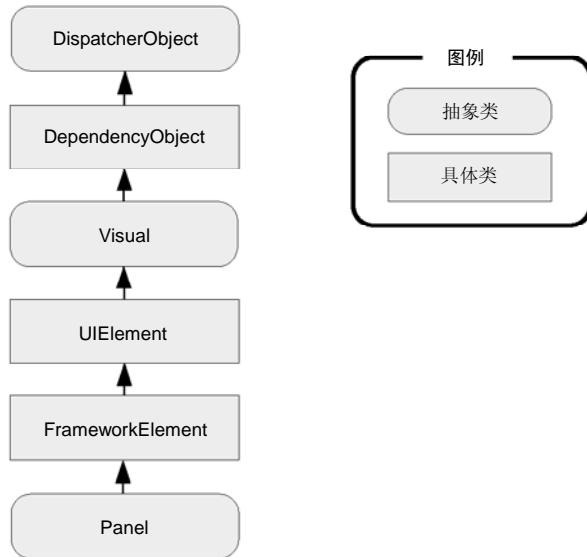


图 3-1 Panel 类的层次结构

表 3-1 Panel 类的公有属性

名 称	说 明
Background	该属性是用于为面板背景着色的画刷。如果想接收鼠标事件，就必须将该属性设置为非空值(如果想接收鼠标事件，又不希望显示固定颜色的背景，那么只需要将背景色设置为透明即可)。在第 6 章中将学习基本画刷的更多内容(并将在第 12 章中学习更多高级的画刷)
Children	该属性是在面板中存储的条目集合。这是第一级条目——换句话说，这些条目自身也可以包含更多的条目
IsItemsHost	该属性是一个布尔值，如果面板用于显示与 ItemsControl 控件关联的项(例如，TreeView 控件中的节点或列表框中的列表项)，该属性值为 true。在大多数情况下，甚至不需要知道列表控件使用后台面板来管理它所包含的条目的布局。但如果希望创建自定义的列表，以不同方式放置子元素(例如，以平铺方式显示图像的 ListBox 控件)，该细节就变得很重要了。在第 20 章中将使用这种技术

注意：

Panel 类还包含几个内部属性，如果希望创建自己的容器，就可以使用它们。最特别的是，可重写继承自 FrameworkElement 类的 MeasureOverride() 和 ArrangeOverride() 方法，以修改当组织子元素时面板处理测量阶段和排列阶段的方式。第 18 章将介绍如何创建自定义面板。

就 Panel 基类本身而言没有什么特别的，但它是其他更多特殊类的起点。WPF 提供了大量可用于安排布局的继承自 Panel 的类，表 3-2 中列出了其中几个最基本的类。与所有 WPF 控件和大多数可视化元素一样，这些类位于 System.Windows.Controls 名称空间中。

表 3-2 核心布局面板

名 称	说 明
StackPanel	在水平或垂直的堆栈中放置元素。这个布局容器通常用于更大、更复杂窗口中的一些小区域
WrapPanel	在一系列可换行的行中放置元素。在水平方向上，WrapPanel 面板从左向右放置条目，然后在随后的行中放置元素。在垂直方向上，WrapPanel 面板在自上而下的列中放置元素，并使用附加的列放置剩余的条目
DockPanel	根据容器的整个边界调整元素
Grid	根据不可见的表格在行和列中排列元素，这是最灵活、最常用的容器之一
UniformGrid	在不可见但是强制所有单元格具有相同尺寸的表中放置元素，这个布局容器不常用
Canvas	使用固定坐标绝对定位元素。这个布局容器与传统 Windows 窗体应用程序最相似，但没有提供锚定或停靠功能。因此，对于尺寸可变的窗口，该布局容器不是合适的选择。如果选择的话，需要另外做一些工作

除这些核心容器外，还有几个更专业的面板，在各种控件中都可能遇到它们。这些容器包括专门用于包含特定控件子元素的面板——如 TabPanel 面板(在 TabPanel 面板中包含多个选项卡)、ToolbarPanel 面板(工具栏中的多个按钮)以及 ToolbarOverflowPanel 面板(Toolbar 控件的溢出菜单中的多个命令)。还有 VirtualizingStackPanel 面板，数据绑定列表控件使用该面板以大幅降低开销；还有 InkCanvas 控件，该控件和 Canvas 控件类似，但该控件支持处理平板电脑(TabletPC)上的手写笔(stylus)输入(例如，根据选择的模式，InkCanvas 控件支持使用指针绘制范围，以选择屏幕上的元素。也可通过普通计算机和鼠标使用 InkCanvas 控件，尽管这有点违反直觉)。本章将介绍 InkCanvas，第 19 章将详细介绍 VirtualizingStackPanel，在本书其他地方谈到相关控件时，将介绍其他专门的面板。

3.2 使用 StackPanel 面板进行简单布局

StackPanel 面板是最简单的布局容器之一。该面板简单地在单行或单列中以堆栈形式放置其子元素。

例如，分析下面的窗口，该窗口包含 4 个按钮：

```
<Window x:Class="Layout.SimpleStack"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Layout" Height="223" Width="354"
    >
<StackPanel>
    <Label>A Button Stack</Label>
    <Button>Button 1</Button>
    <Button>Button 2</Button>
    <Button>Button 3</Button>
    <Button>Button 4</Button>
</StackPanel>
</Window>
```

图 3-2 显示了最终得到的窗口。

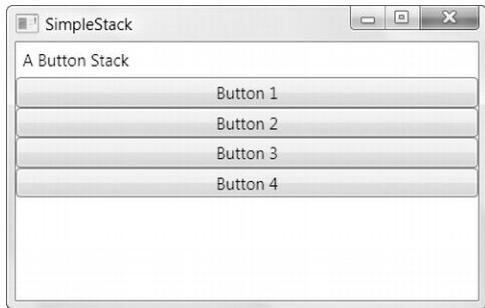


图 3-2 使用 StackPanel 面板

在 Visual Studio 中添加布局容器

在 Visual Studio 中使用设计器创建这个示例要比较容易。首先删除 Grid 根元素(如果有的话)。然后将一个 StackPanel 面板拖动到窗口上。接下来将其他元素以所希望的自上而下的顺序(标签和 4 个按钮)拖放到窗口上。如果想重新排列 StackPanel 面板中的元素，可以简单地将它们拖动到新的位置。

虽然本书不会占用大量篇幅来讨论 Visual Studio 的设计时支持特性，但实际上，自从推出首个 WPF 版本以来，Visual Studio 已经做了很大的改进。例如，Visual Studio 不再为添加到设计器中的每个新控件指定名称；而且除非您手动调整控件大小，Visual Studio 不再添加硬编码的 Width 值和 Height 值。

默认情况下，StackPanel 面板按自上而下的顺序排列元素，使每个元素的高度适合它的内容。在这个示例中，这意味着标签和按钮的大小刚好足够适应它们内部包含的文本。所有元素都被拉伸到 StackPanel 面板的整个宽度，这也是窗口的宽度。如果加宽窗口，StackPanel 面板也会变宽，并且按钮也会拉伸自身以适应变化。

通过设置 Orientation 属性，StackPanel 面板也可用于水平排列元素：

```
<StackPanel Orientation="Horizontal">
```

现在，元素指定它们的最小宽度(足以适合它们所包含的文本)并拉伸至容器面板的整个高度。根据窗口的当前大小，这可能导致一些元素不适应，如图 3-3 所示。

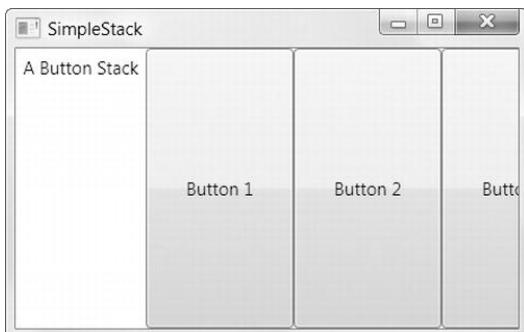


图 3-3 水平方向的 StackPanel 面板

显然，这并未提供实际应用程序所需的灵活性。幸运的是，可使用布局属性对 StackPanel 面板和其他布局容器的工作方式进行精细调整，如稍后所述。

3.2.1 布局属性

尽管布局由容器决定，但子元素仍有一定的决定权。实际上，布局面板支持一小组布局属性，以便与子元素结合使用，在表 3-3 中列出了这些布局属性。

表 3-3 布局属性

名 称	说 明
HorizontalAlignment	当水平方向上有额外的空间时，该属性决定了子元素在布局容器中如何定位。可选用 Center、Left、Right 或 Stretch 等属性值
VerticalAlignment	当垂直方向上有额外的空间时，该属性决定了子元素在布局容器控件中如何定位。可选用 Center、Top、Bottom 或 Stretch 等属性值
Margin	该属性用于在元素的周围添加一定的空间。Margin 属性是 System.Windows.Thickness 结构的一个实例，该结构具有分别用于为顶部、底部、左边和右边添加空间的独立组件
MinWidth 和 MinHeight	这两个属性用于设置元素的最小尺寸。如果一个元素对于其他布局容器来说太大，该元素将被剪裁以适合容器
MaxWidth 和 MaxHeight	这两个属性用于设置元素的最大尺寸。如果有更多可以使用的空间，那么在扩展子元素时就不会超出这一限制，即使将 HorizontalAlignment 和 VerticalAlignment 属性设置为 Stretch 也同样如此
Width 和 Height	这两个属性用于显式地设置元素的尺寸。这一设置会重写为 HorizontalAlignment 和 VerticalAlignment 属性设置的 Stretch 值。但不能超出 MinWidth、MinHeight、MaxWidth 和 MaxHeight 属性设置的范围

所有这些属性都从 FrameworkElement 基类继承而来，所以在 WPF 窗口中可使用的所有图形小组件都支持这些属性。

注意：

您在第 2 章中已学习过，不同的布局容器可以为它们的子元素提供附加属性。例如，Grid 对象的所有子元素可以获得 Row 和 Column 属性，以便选择容纳它们的单元格。通过附加属性可为特定的布局容器设置其特有的信息。然而，在表 3-3 中列出的布局属性是可以应用于许多布局面板的通用属性。因此，这些属性被定义为 FrameworkElement 基类的一部分。

这个属性列表就像它所没有包含的属性一样值得注意。如果查找熟悉的与位置相关的属性，例如 Top 属性、Right 属性以及 Location 属性，是不会找到它们的。这是因为大多数布局容器（Canvas 控件除外）都使用自动布局，并未提供显式定位元素的能力。

3.2.2 对齐方式

为理解这些属性的工作原理，可进一步分析图 3-2 中显示的简单 StackPanel 面板。在这个

示例中——有一个垂直方向的 StackPanel 面板——VerticalAlignment 属性不起作用，因为所有元素的高度都自动地调整为刚好满足各自需要。但 HorizontalAlignment 属性非常重要，它决定了各个元素在行的什么位置。

通常，对于 Label 控件，HorizontalAlignment 属性的值默认为 Left；对于 Button 控件，HorizontalAlignment 属性的值默认为 Stretch。这也是为什么每个按钮的宽度被调整为整列的宽度的原因所在。但可以改变这些细节：

```
<StackPanel>
    <Label HorizontalAlignment="Center">A Button Stack</Label>
    <Button HorizontalAlignment="Left">Button 1</Button>
    <Button HorizontalAlignment="Right">Button 2</Button>
    <Button>Button 3</Button>
    <Button>Button 4</Button>
</StackPanel>
```

图 3-4 显示了最终结果。现在前两个按钮的尺寸是它们应当具有的最小尺寸，并进行了对齐，而底部两个按钮被拉伸至整个 StackPanel 面板的宽度。如果改变窗口的尺寸，就会发现标签保持在中间位置，而前两个按钮分别被粘贴到两边。

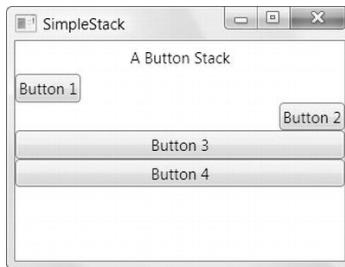


图 3-4 包含对齐按钮的 StackPanel 面板

注意：

StackPanel 面板也有自己的 HorizontalAlignment 和 VerticalAlignment 属性。这两个属性默认都被设置为 Stretch，所以 StackPanel 面板完全充满它的容器。在这个示例中，这意味着 StackPanel 面板充满整个窗口。如果使用不同设置，StackPanel 面板的尺寸将足够宽以容纳最宽的控件。

3.2.3 边距

在 StackPanel 示例中，在当前情况下存在一个明显的问题。设计良好的窗口不只是包含元素——还应当在元素之间包含一定的额外空间。为了添加额外的空间并使 StackPanel 面板示例中的按钮不那么紧密，可为控件设置边距。

当设置边距时，可为所有边设置相同的宽度，如下所示：

```
<Button Margin="5">Button 3</Button>
```

相应地，也可为控件的每个边以左、上、右、下的顺序设置不同的边距：

```
<Button Margin="5,10,5,10">Button 3</Button>
```

在代码中，使用 Thickness 结构来设置边距：

```
cmd.Margin = new Thickness(5);
```

为得到正确的控件边距，需要采用一些艺术手段，因为需要考虑相邻控件边距设置的相互影响。例如，如果两个按钮堆在一起，位于最高处的按钮的底部边距设置为 5，而下面按钮的顶部边距也设置为 5，那么在这两个按钮之间就有 10 个单位的空间。

理想情况是，能尽可能始终如一地保持不同的边距设置，避免为不同的边设置不同的值。例如，在 StackPanel 示例中，为按钮和面板本身使用相同的边距是比较合适的，如下所示：

```
<StackPanel Margin="3">
    <Label Margin="3" HorizontalAlignment="Center">
        A Button Stack</Label>
    <Button Margin="3" HorizontalAlignment="Left">Button 1</Button>
    <Button Margin="3" HorizontalAlignment="Right">Button 2</Button>
    <Button Margin="3">Button 3</Button>
    <Button Margin="3">Button 4</Button>
</StackPanel>
```

这种设置使得两个按钮之间的总空间(两个按钮的边距之和)和按钮与窗口之间的总空间(按钮边距和 StackPanel 边距之和)是相同的。图 3-5 显示了这个更合理的窗口，图 3-6 是边距设置的分解图。

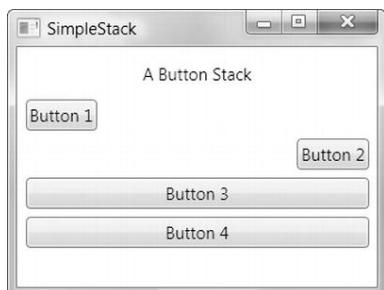


图 3-5 在元素之间添加边距

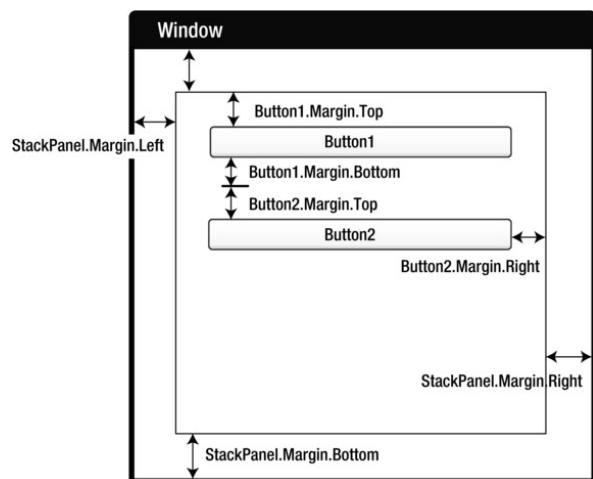


图 3-6 边距的结合方式

3.2.4 最小尺寸、最大尺寸以及显式地设置尺寸

最后，每个元素都提供了 Height 和 Width 属性，用于显式地指定元素大小。但这种设置一般不是一个好主意。相反，如有必要，应当使用最大尺寸和最小尺寸属性，将控件限制在正确范围内。

提示：

在 WPF 中显式地设置尺寸之前一定要三思。在良好的布局设计中，不必显式地设置尺寸。如果确实添加了尺寸信息，那就冒险创建了一种更不稳定的布局，这种布局不能适应变化(例如，不能适应不同的语言和不同的窗口尺寸)，而且可能剪裁您的内容。

例如，您可能决定拉伸 StackPanel 容器中的按钮，使其适合 StackPanel，但其宽度不能超过 200 单位，也不能小于 100 单位(默认情况下，最初按钮的最小宽度是 75 单位)。下面是所需

的标记：

```
<StackPanel Margin="3">
    <Label Margin="3" HorizontalAlignment="Center">
        A Button Stack</Label>
    <Button Margin="3" MaxWidth="200" MinWidth="100">Button 1</Button>
    <Button Margin="3" MaxWidth="200" MinWidth="100">Button 2</Button>
    <Button Margin="3" MaxWidth="200" MinWidth="100">Button 3</Button>
    <Button Margin="3" MaxWidth="200" MinWidth="100">Button 4</Button>
</StackPanel>
```

提示：

现在，您可能会好奇是否有更简便的方法设置那些在多个元素中是标准化的属性，如这个示例中的按钮边距。答案是使用样式——一种允许重复使用(甚至自动应用)属性设置的特性。第 11 章将介绍样式的相关内容。

当 StackPanel 调整按钮的尺寸时，需要考虑以下几部分信息：

- **最小尺寸**。每个按钮的尺寸始终不能小于最小尺寸。
- **最大尺寸**。每个按钮的尺寸始终不能超过最大尺寸(除非执行错误操作，使最大尺寸比最小尺寸还小)。
- **内容**。如果按钮中的内容需要更大的宽度，StackPanel 容器会尝试扩展按钮(可以通过检查 DesiredSize 属性确定所需的按钮大小，该属性返回最小宽度或内容的宽度，返回两者中较大的那个)。
- **容器尺寸**。如果最小宽度大于 StackPanel 面板的宽度，按钮的一部分将被剪裁掉。否则，不允许按钮比 StackPanel 面板更宽，即使不能适合按钮表面的所有文本也同样如此。
- **水平对齐方式**。因为默认情况下按钮的 HorizontalAlignment 属性值设置为 Stretch，所以 StackPanel 面板将尝试放大按钮以占满 StackPanel 面板的整个宽度。

理解这个过程的关键在于，要认识到最小尺寸和最大尺寸设置了绝对界限。在这些界限内，StackPanel 面板尝试反映按钮所期望的尺寸(以适合其内容)以及对齐方式的设置。

图 3-7 显示了 StackPanel 面板工作方式的几个例子。在左图中，窗口的尺寸缩到最小。每个按钮是 100 单位宽，窗口不能变得更窄。如果再收缩窗口，每个按钮的右边将被剪裁掉(可通过设置窗口本身的 MinWidth 属性，使窗口不能小于最小宽度，以免发生这种情况)。

当放大窗口时，会增加按钮的宽度直到它们达到 200 单位的宽度上限。此时如果继续放大窗口，会在按钮的两边添加额外空间(如图 3-7 中右图所示)。

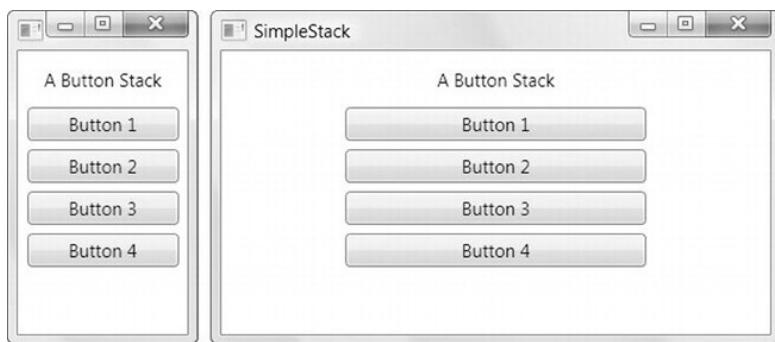


图 3-7 限制按钮尺寸的变化

注意：

某些情况下，可能希望使用代码检查窗口中某个元素的尺寸。这时使用 Height 和 Width 属性是没有用的，因为这两个属性指示的是您所期望的尺寸设置，可能和实际的渲染尺寸不同。在理想情况下，应让元素的尺寸适应它们的内容，根本不用设置 Height 和 Width 属性。但是，可以通过读取 ActualHeight 和 ActualWidth 属性得到用于渲染元素的实际尺寸。需要记住的是，当窗口大小发生变化或其中的内容改变时，这些值可能会改变。

自动改变大小的窗口

在本例中，还有一个元素具有硬编码的尺寸：包含 StackPanel 面板(以及该面板中的所有内容)的顶级窗口。出于很多原由，可认为使用硬编码的窗口尺寸仍然有意义。

但可以自动改变窗口大小，如果使用动态内容构造简单窗口，这还是有意义的。为使窗口能自动改变大小，需要删除 Height 和 Width 属性，并将 Window.SizeToContent 属性设置为 WidthAndHeight。这时窗口就会扩大自身的尺寸，从而足以容纳包含的所有内容。通过将 SizeToContent 属性设置为 Width 或 Height，还可使窗口只能在一个方向上改变自身的尺寸。

3.2.5 Border 控件

Border 控件不是布局面板，而是非常便于使用的元素，经常与布局面板一起使用。所以，在继续介绍其他布局面板之前，现在先介绍一下 Border 控件是有意义的。

Border 类非常简单。它只能包含一段嵌套内容(通常是布局面板)，并为其添加背景或在其周围添加边框。为了深入地理解 Border 控件，只需要掌握表 3-4 中列出的属性就可以了。

表 3-4 Border 类的属性

名 称	说 明
Background	使用 Brush 对象设置边框中所有内容后面的背景。可使用固定颜色背景，也可使用其他更特殊的背景
BorderBrush 和 BorderThickness	使用 Brush 对象设置位于 Border 对象边缘的边框的颜色，并设置边框的宽度。为显示边框，必须设置这两个属性
CornerRadius	该属性可使边框具有雅致的圆角。CornerRadius 的值越大，圆角效果就越明显
Padding	该属性在边框和内部的内容之间添加空间(与此相对，Margin 属性在边框之外添加空间)

下面是一个具有轻微圆角效果的简单边框，该边框位于一组按钮的周围，这组按钮包含在一个 StackPanel 面板中：

```
<Border Margin="5" Padding="5" Background="LightYellow"
BorderBrush="SteelBlue" BorderThickness="3,5,3,5" CornerRadius="3"
VerticalAlignment="Top">
<StackPanel>
<Button Margin="3">One</Button>
<Button Margin="3">Two</Button>
<Button Margin="3">Three</Button>
```

```
</StackPanel>
</Border>
```

图 3-8 显示了该例的结果。



图 3-8 基本边框

第 6 章将介绍有关画刷和颜色的详情，它们可用于设置 BorderBrush 和 Background 属性。

注意：

从技术角度看，Border 是装饰元素(decorator)，装饰元素是特定类型的元素，通常用于在对象周围添加某些种类的图形装饰。所有装饰元素都继承自 System.Windows.Controls.Decorator 类。大多数装饰元素设计用于特定控件。例如，Button 控件使用 ButtonChrome 装饰元素，以获取其特有的圆角和阴影背景效果；而 ListBox 控件使用 ListBoxChrome 装饰元素。还有两个更通用的装饰元素，当构造用户界面时它们非常有用：在此讨论的 Border 元素以及将在第 12 章中研究的 Viewbox 元素。

3.3 WrapPanel 和 DockPanel 面板

显然，只使用 StackPanel 面板还不能帮助您创建出实用的用户界面。要设计出最终使用的用户界面，StackPanel 面板还需要与其他更强大的布局容器协作。只有这样才能组装成完整的窗口。

最复杂的布局容器是 Grid 面板，稍后将分析该面板。在介绍 Grid 面板之前，有必要首先看一下 WrapPanel 和 DockPanel 面板，它们是 WPF 提供的两个更简单的布局容器。这两个布局容器通过不同的布局行为对 StackPanel 面板进行补充。

3.3.1 WrapPanel 面板

WrapPanel 面板在可能的空间中，以一次一行或一列的方式布置控件。默认情况下，WrapPanel.Orientation 属性设置为 Horizontal；控件从左向右进行排列，再在下一行中排列。但可将 WrapPanel.Orientation 属性设置为 Vertical，从而在多个列中放置元素。

提示：

与 StackPanel 面板类似，WrapPanel 面板实际上主要用来控制用户界面中一小部分的布局细节，并非用于控制整个窗口布局。例如，可能使用 WrapPanel 面板以类似工具栏控件的方式将所有按钮保持在一起。

下面的示例中定义了一系列具有不同对齐方式的按钮，并将这些按钮放到一个 WrapPanel 面板中：

```
<WrapPanel Margin="3">
<Button VerticalAlignment="Top">Top Button</Button>
<Button MinHeight="60">Tall Button 2</Button>
<Button VerticalAlignment="Bottom">Bottom Button</Button>
<Button>Stretch Button</Button>
<Button VerticalAlignment="Center">Centered Button</Button>
</WrapPanel>
```

图 3-9 显示了如何对这些按钮进行换行以适应 WrapPanel 面板的当前尺寸(WrapPanel 面板的当前尺寸是由包含它的窗口的尺寸决定的)。正如这个示例所演示的，WrapPanel 面板水平地创建了一系列假想的行，每一行的高度都被设置为所包含元素中最高元素的高度。其他控件可能被拉伸以适应这一高度，或根据 VerticalAlignment 属性的设置进行对齐。在图 3-9 的左图中，所有按钮都在位于较高的行中，并被拉伸或对齐以适应该行的高度。在右图中，有几个按钮被挤到第二行中。因为第二行没有包含特别高的按钮，所以第二行的高度保持为最小按钮的高度。因此，在该行中不必关心各按钮的 VerticalAlignment 属性的设置。



图 3-9 对按钮进行换行

注意：

WrapPanel 面板是唯一一个不能通过灵活使用 Grid 面板代替的面板。

3.3.2 DockPanel 面板

DockPanel 面板是更有趣的布局选项。它沿着一条外边缘来拉伸所包含的控件。理解该面板最简便的方法是，考虑一下位于许多 Windows 应用程序窗口顶部的工具栏。这些工具栏停靠到窗口顶部。与 StackPanel 面板类似，被停靠的元素选择它们布局的一个方面。例如，如果将一个按钮停靠在 DockPanel 面板的顶部，该按钮会被拉伸至 DockPanel 面板的整个宽度，但根据内容和 MinHeight 属性为其设置所需的高度。而如果将一个按钮停靠到容器左边，该按钮的高度将被拉伸以适应容器的高度，而其宽度可以根据需要自由增加。

这里很明显的问题是：子元素如何选择停靠的边？答案是通过 Dock 附加属性，可将该属性设置为 Left、Right、Top 或 Bottom。放在 DockPanel 面板中的每个元素都会自动捕获该属性。

下面的示例在 DockPanel 面板的每条边上都停靠一个按钮：

```
<DockPanel LastChildFill="True">
<Button DockPanel.Dock="Top">Top Button</Button>
<Button DockPanel.Dock="Bottom">Bottom Button</Button>
<Button DockPanel.Dock="Left">Left Button</Button>
<Button DockPanel.Dock="Right">Right Button</Button>
<Button>Remaining Space</Button>
```

```
</DockPanel>
```

该例还将 DockPanel 面板的 LastChildFill 属性设置为 true，该设置告诉 DockPanel 面板使最后一个元素占满剩余空间。图 3-10 显示了结果。

显然，当停靠控件时，停靠顺序很重要。在这个示例中，顶部和底部按钮充满了 DockPanel 面板的整个边缘，这是因为这两个按钮首先被停靠。接着停靠左边和右边的按钮时，这两个按钮将位于顶部按钮和底部按钮之间。如果改变这一顺序，那么左边和右边的按钮将充满整个面板的边缘，而顶部和底部的按钮则变窄一些，因为它们将在左边和右边的两个按钮之间进行停靠。

可将多个元素停靠到同一边缘。这种情况下，元素按标记中声明的顺序停靠到边缘。而且，如果不希望空间分割或拉伸行为，可修改 Margin 属性、HorizontalAlignment 属性以及 VerticalAlignment 属性，就像使用 StackPanel 面板进行布局时所介绍的那样。下面是前面演示的程序的修改版本：

```
<DockPanel LastChildFill="True">
    <Button DockPanel.Dock="Top">A Stretched Top Button</Button>
    <Button DockPanel.Dock="Top" HorizontalAlignment="Center">
        A Centered Top Button</Button>
    <Button DockPanel.Dock="Top" HorizontalAlignment="Left">
        A Left-Aligned Top Button</Button>
    <Button DockPanel.Dock="Bottom">Bottom Button</Button>
    <Button DockPanel.Dock="Left">Left Button</Button>
    <Button DockPanel.Dock="Right">Right Button</Button>
    <Button>Remaining Space</Button>
</DockPanel>
```

停靠行为保持不变。首先停靠顶部按钮，然后是底部按钮，顶部和底部按钮之间剩余的空间会被分割，并且最后一个按钮在中间。图 3-11 显示了最终窗口。

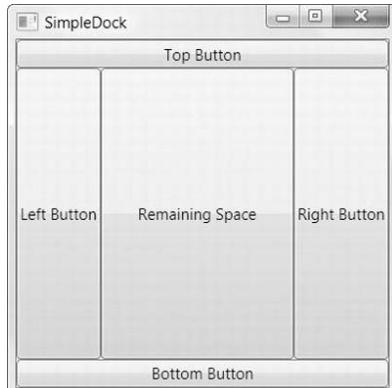


图 3-10 停靠到每个边缘

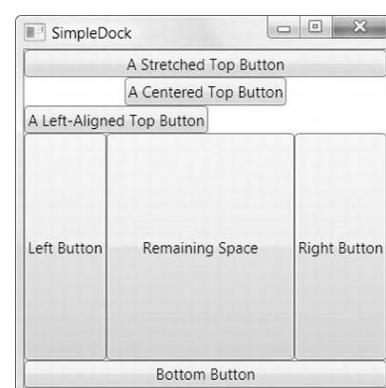


图 3-11 在顶部停靠多个元素

3.3.3 嵌套布局容器

很少单独使用 StackPanel、WrapPanel 和 DockPanel 面板。相反，它们通常用来设置一部分用户界面的布局。例如，可使用 DockPanel 面板在窗口的合适区域放置不同的 StackPanel 和 WrapPanel 面板容器。

例如，假设希望创建一个标准对话框，在其右下角具有 OK 按钮和 Cancel 按钮，并且在窗口的剩余部分是一块较大的内容区域。在 WPF 中可采用几种方法完成这一布局，但最简单的

方法如下，该方法使用前面介绍过的各种面板：

- (1) 创建水平 StackPanel 面板，用于将 OK 按钮和 Cancel 按钮放置在一起。
- (2) 在 DockPanel 面板中放置 StackPanel 面板，将其停靠到窗口底部。
- (3) 将 DockPanel.LastChildFill 属性设置为 true，以使用窗口剩余的部分填充其他内容。在此可以添加另一个布局控件，或者只添加一个普通的 TextBox 控件(本例中使用的是 TextBox 控件)。
- (4) 设置边距属性，提供一定的空白空间。

下面是最终的标记：

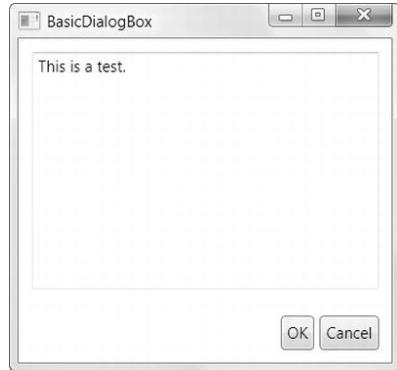
```
<DockPanel LastChildFill="True">
  <StackPanel DockPanel.Dock="Bottom" HorizontalAlignment="Right"
    Orientation="Horizontal">
    <Button Margin="10,10,2,10" Padding="3">OK</Button>
    <Button Margin="2,10,10,10" Padding="3">Cancel</Button>
  </StackPanel>
  <TextBox DockPanel.Dock="Top" Margin="10">This is a test.</TextBox>
</DockPanel>
```

在这个示例中，Padding 属性在按钮边框与内部的内容(单词 OK 或 Cancel)之间添加了尽量少的空间。图 3-12 显示了这个示例创建的相对流行的对话框。

乍一看，相对于使用坐标精确地放置控件而言，这有些多余。在许多情况下，确实如此。不过，设置时间固然较长，但这样做的好处是在将来可以很方便地修改用户界面。例如，如果决定让 OK 按钮和 Cancel 按钮位于窗口底部的中间，只需要修改包含这两个按钮的 StackPanel 面板的对齐方式即可：

```
<StackPanel DockPanel.Dock="Bottom"
  HorizontalAlignment="Center" ... >
```

图 3-12 基本对话框



与诸如 Windows 窗体的旧式用户界面框架相比，这里使用的标记更整洁、更简单也更紧凑。如果为这个窗口添加一些样式(详见第 11 章)，还可对该窗口进行进一步的改进，并移除其他不必要的细节(如边距设置)，从而创建真正的自适应用户界面。

提示：

如果有一棵茂密的嵌套元素树，很可能看不到整个结构。Visual Studio 提供了一个方便的功能，用于显示一棵表示各个元素的树，并允许您通过逐步单击进入希望查看(或修改)的元素。这一功能是指 Document Outline 窗口，可通过选择 View | Other Windows | Document Outline 菜单项来显示该窗口。

3.4 Grid 面板

Grid 面板是 WPF 中功能最强大的布局容器。很多使用其他布局控件能完成的功能，用 Grid 面板也能实现。Grid 面板也是将窗口分割成(可使用其他面板进行管理的)更小区域的理想工具。实际上，由于 Grid 面板十分有用，因此在 Visual Studio 中为窗口添加新的 XAML 文档时，会自

动添加 Grid 标签作为顶级容器，并嵌套在 Window 根元素中。

Grid 面板将元素分隔到不可见的行列网格中。尽管可在每一个单元格中放置多个元素(这时这些元素会相互重叠)，但在每个单元格中只放置一个元素通常更合理。当然，在 Grid 单元格中的元素本身也可能是另一个容器，该容器组织它所包含的一组控件。

提示：

尽管 Grid 面板被设计成不可见的，但可将 Grid.ShowGridLines 属性设置为 true，从而更清晰地观察 Grid 面板。这一特性并不是真正试图美化窗口，反而是为了方便调试，设计该特性旨在帮助理解 Grid 面板如何将其自身分割成多个较小的区域。这一特性十分重要，因为通过该特性可准确控制 Grid 面板如何选择列宽和行高。

需要两个步骤来创建基于 Grid 面板的布局。首先，选择希望使用的行和列的数量。然后，为每个包含的元素指定恰当的行和列，从而在合适的位置放置元素。

Grid 面板通过使用对象填充 Grid.ColumnDefinitions 和 Grid.RowDefinitions 集合来创建网格和行。例如，如果确定需要两行和三列，可添加以下标签：

```
<Grid ShowGridLines="True">
  <Grid.RowDefinitions>
    <RowDefinition></RowDefinition>
    <RowDefinition></RowDefinition>
  </Grid.RowDefinitions>
  <Grid.ColumnDefinitions>
    <ColumnDefinition></ColumnDefinition>
    <ColumnDefinition></ColumnDefinition>
    <ColumnDefinition></ColumnDefinition>
  </Grid.ColumnDefinitions>

  ...
</Grid>
```

正如本例所演示的，在 RowDefinition 或 ColumnDefinition 元素中不必提供任何信息。如果保持它们为空(本例正是如此)，Grid 面板将在所有行和列之间平均分配空间。在本例中，每个单元格的尺寸完全相同，具体取决于包含窗口的尺寸。

为在单元格中放置各个元素，需要使用 Row 和 Column 附加属性。这两个属性的值都是从 0 开始的索引数。例如，以下标记演示了如何创建 Grid 面板，并使用按钮填充 Grid 面板的部分单元格。

```
<Grid ShowGridLines="True">
  ...

  <Button Grid.Row="0" Grid.Column="0">Top Left</Button>
  <Button Grid.Row="0" Grid.Column="1">Middle Left</Button>
  <Button Grid.Row="1" Grid.Column="2">Bottom Right</Button>
  <Button Grid.Row="1" Grid.Column="1">Bottom Middle</Button>
</Grid>
```

每个元素必须被明确地放在对应的单元格中。可在单元格中放置多个元素(通常这没什么意义)，或让单元格保持为空(这通常是有用的)。也可以不按顺序声明元素，正如本例中的最后两个按钮那样。但如果逐行(并在每行中按从右向左的顺序)定义控件，可使标记更清晰。

此处存在例外情况。如果不指定 Grid.Row 属性，Grid 面板会假定该属性的值为 0。对于 Grid.Column 属性也是如此。因此，在 Grid 面板的第一个单元格中放置元素时可不指定这两个属性。

注意：

Grid 面板在预定义的行和列中放置元素。这与 WrapPanel 和 StackPanel 面板(当它们布置子元素时，会隐式地创建行或列)等布局容器不同。如果希望创建具有多行和多列的网格，就必须使用 RowDefinitions 和 ColumnDefinitions 对象显式地定义行和列。

图 3-13 显示了这个简单网格在两种不同尺寸的情形下是如何显示的。注意，ShowGridLines 属性被设置为 true，从而可以看到每列和每行之间的分割线。

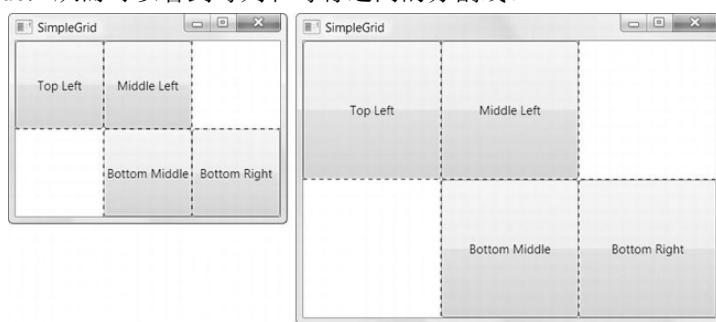


图 3-13 简单的网格

正如您所期望的，Grid 面板具有在表 3-3 中列出的基本布局属性集。这意味着可在单元格内容的周围添加边距，从而改变元素尺寸的变化方式，使其不充满整个单元格，并且可以沿着单元格的一条边缘对齐元素。如果强制一个元素的尺寸大于单元格允许的范围，那么这个元素的部分内容会被剪裁掉。

在 Visual Studio 中使用 Grid 面板

在 Visual Studio 设计视图中使用 Grid 面板时，将发现它和其他布局容器有些不同。当把一个元素拖动到 Grid 面板中时，Visual Studio 允许将该元素放置到精确位置。Visual Studio 通过设置元素的 Margin 属性完成这一工作。

在设置边距时，Visual Studio 使用最近的角。例如，如果元素距离网格的左上角最近，那么 Visual Studio 通过上边和左边的边距来定位元素(并且右边和下边的边距保持为 0)。如果拖动元素使其距左下角较近，那么 Visual Studio 设置下边和左边的边距，并将 VerticalAlignment 属性设置为 Bottom。当网格大小发生变化时，这显然会影响元素的移动方式。

Visual Studio 中的边距设置过程看起来非常直接，但在大多数情况下得不到所期望的结果。您通常希望得到更灵活的流式布局，以允许一些元素可动态扩展，并将其他元素推挤到其他位置。在这种情形下，将发现通过 Margin 属性的硬编码定位是非常不灵活的。当添加多个元素时问题会更加严重，因为 Visual Studio 不能自动地添加新的单元格。因此，所有元素将被放到同一单元格中。不同元素可与 Grid 面板的不同拐角对齐，当窗口大小发生变化时，又会导致这些元素彼此之间相对移动(甚至互相重叠)。

一旦理解 Grid 面板的工作原理，就可以正确地解决这些问题。第一个技巧是在添加元素之前定义 Grid 面板的行和列(可通过 Properties 窗口编辑 RowDefinitions 和 ColumnDefinitions 集合)

并对 Grid 面板进行配置。一旦设置好 Grid 面板，即可将元素拖放到 Grid 面板中，并使用 Properties 窗口或通过手动编辑 XAML 来配置它们的边距和对齐方式设置。

3.4.1 调整行和列

如果 Grid 面板只是按比例分配尺寸的行和列的集合，它也就没什么用处了。幸运的是，情况并非如此。为了充分发挥 Grid 面板的潜能，可更改每一行和每一列的尺寸设置方式。

Grid 面板支持以下三种设置尺寸的方式：

- **绝对设置尺寸方式**。使用设备无关单位准确地设置尺寸。这是最无用的策略，因为这种策略不够灵活，难以适应内容大小和容器大小的改变，而且难以处理本地化。
- **自动设置尺寸方式**。每行和每列的尺寸刚好满足需要。这是最有用的尺寸设置方式。
- **按比例设置尺寸方式**。按比例将空间分割到一组行和列中。这是对所有行和列的标准设置。例如，从图 3-13 中可看到当扩展 Grid 面板时，所有单元格都按比例增加尺寸。

为了获得最大的灵活性，可混合使用这三种尺寸设置方式。例如，创建几个自动设置尺寸的行，然后通过按比例设置尺寸的方式让最后的一行或两行充满剩余的空间，这通常是很常用的。

可通过将 ColumnDefinition 对象的 Width 属性或 RowDefinition 对象的 Height 属性设置为数值来确定尺寸设置方式。例如，下面的代码显示了如何设置 100 设备无关单位的绝对宽度：

```
<ColumnDefinition Width="100"></ColumnDefinition>
```

为使用自动尺寸设置方式，可使用 Auto 值：

```
<ColumnDefinition Width="Auto"></ColumnDefinition>
```

最后，为了使用按比例尺寸设置方式，需要使用星号(*)：

```
<ColumnDefinition Width="*"></ColumnDefinition>
```

如果混合使用按比例尺寸设置方式和其他尺寸设置方式，就可以在剩余的任意空间按比例改变行或列的尺寸。

如果希望不均匀地分割剩余空间，可指定权重，权重必须放在星号之前。例如，如果有两行是按比例设置尺寸，并希望第一行的高度是第二行高度的一半，那么可以使用如下设置来分配剩余空间：

```
<RowDefinition Height="* "></RowDefinition>
<RowDefinition Height="2* "></RowDefinition>
```

上面的代码告诉 Grid 面板，第二行的高度应是第一行高度的两倍。可使用您喜欢的任何数字来划分剩余空间。

注意：

通过代码可以很方便地与 ColumnDefinition 和 RowDefinition 对象进行交互。只需要知道 Width 和 Height 属性是 GridLength 类型的对象即可。为创建表示特定尺寸的 GridLength 对象，只需要为 GridLength 类的构造函数传递一个合适的数值即可。为了创建一个表示按比例设置尺寸(*)的 GridLength 对象，可为 GridLength 类的构造函数传递数值作为第一个参数，并传递 GridUnitType.Start 作为第二个参数。要指定使用自动设置尺寸方式，可使用静态属性 GridLength.Auto。

使用这些尺寸设置方式，可重现图 3-12 中所示的简单示例对话框。使用顶级的 Grid 容器将窗口分成两行，而不是使用 DockPanel 面板。下面是所需要的标记：

```
<Grid ShowGridLines="True">
<Grid.RowDefinitions>
<RowDefinition Height="*"/></RowDefinition>
<RowDefinition Height="Auto"/></RowDefinition>
</Grid.RowDefinitions>
<TextBox Margin="10" Grid.Row="0">This is a test.</TextBox>
<StackPanel Grid.Row="1" HorizontalAlignment="Right" Orientation="Horizontal">
<Button Margin="10,10,2,10" Padding="3">OK</Button>
<Button Margin="2,10,10,10" Padding="3">Cancel</Button>
</StackPanel>
</Grid>
```

提示：

这个 Grid 面板未声明任何列，如果 Grid 面板只有一列并且该列是按比例设置尺寸的(从而该列充满整个 Grid 面板的宽度)，那么这是一种可使用的快捷方式。

上面的标记稍长些，但按显示顺序声明控件具有一项优点，可使标记更容易理解。在该例中使用的方法仅仅是一种选择，如果愿意，也可使用一行两列的 Grid 面板来代替嵌套的 StackPanel 面板。

注意：

使用嵌套的 Grid 面板容器，几乎可以创建任何用户界面(例外是使用 WrapPanel 面板换行或换列)。但当处理用户界面中的一小部分或布置少量元素时，通常使用更特殊的 StackPanel 和 DockPanel 面板容器以便简化操作。

3.4.2 布局舍入

如第 1 章所述，WPF 使用分辨率无关的测量系统。尽管该测量系统为使用各种不同的硬件提供了灵活性，但有时也会引入一些问题。其中一个问题时元素可能被对齐到子像素(subpixel)边界——换句话说，使用没有和物理像素准确对齐的小数坐标定位元素。可通过为相邻的布局容器提供非整数尺寸强制发生这个问题。但是当不希望发生这个问题时，在某些情况下该问题也可能会出现，例如当创建按比例设置尺寸的 Grid 面板时就可能会发生该问题。

例如，假设使用一个包含两列且具有 200 像素的 Grid 面板。如果将该面板均匀分成两个按比例设置尺寸的列，那么意味着每列为 100 像素宽。但是如果这个 Grid 面板的宽度为 175 像素，就不能很清晰地分割成两列，并且每列为 87.5 像素。这意味着第二列会和原始的像素边界稍有些错位。这通常不是问题，但是如果该列包含一个形状元素、一个边框或一幅图像，那么该内容的显示可能是模糊的，因为 WPF 会使用反锯齿功能“混合”原本清晰的像素边界边缘。图 3-14 显示了这一问题。该图放大了窗口的一部分，该窗口包含两个 Grid 面板容器。最上面的 Grid 面板没有使用布局舍入(layout rounding)，所以矩形的清晰边缘在特定的窗口尺寸下变得模糊了。

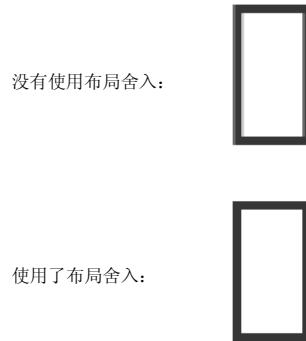


图 3-14 按比例设置尺寸导致的模糊问题

如果这个问题影响到布局，可以采用一种方法很方便地解决该问题。只需要将布局容器的 `UseLayoutRounding` 属性设置为 `true`:

```
<Grid UseLayoutRounding="True">
```

现在，WPF 会确保布局容器中的所有内容对齐到最近的像素边界，从而消除了所有模糊问题。

3.4.3 跨越行和列

您已经看到如何使用 `Row` 和 `Column` 附加属性在单元格中放置元素。还可以使用另外两个附加属性使元素跨越多个单元格，这两个附加属性是 `RowSpan` 和 `ColumnSpan`。这两个属性使用元素将会占有的行数和列数进行设置。

例如，下面的按钮将占据第一行中的第一个和第二个单元格的所有空间：

```
<Button Grid.Row="0" Grid.Column="0" Grid.RowSpan="2">Span Button</Button>
```

下面的代码通过跨越两列和两行，拉伸按钮使其占据所有 4 个单元格：

```
<Button Grid.Row="0" Grid.Column="0" Grid.RowSpan="2" Grid.ColumnSpan="2">  
Span Button</Button>
```

通过跨越行和列可得到更有趣的效果，当需要在由分割器或更长的内容区域分开的表格结构中放置元素时，这是非常方便的。

使用列跨越特征，可以只使用 `Grid` 面板重新编写图 3-12 中的简单示例对话框。`Grid` 面板将窗口分割成三列，展开文本框使其占据所有的三列，并使用最后两列对齐 `OK` 按钮和 `Cancel` 按钮：

```
<Grid ShowGridLines="True">  
  <Grid.RowDefinitions>  
    <RowDefinition Height="*"/></RowDefinition>  
    <RowDefinition Height="Auto"/></RowDefinition>  
  </Grid.RowDefinitions>  
  <Grid.ColumnDefinitions>  
    <ColumnDefinition Width="*"/></ColumnDefinition>  
    <ColumnDefinition Width="Auto"/></ColumnDefinition>  
    <ColumnDefinition Width="Auto"/></ColumnDefinition>  
  </Grid.ColumnDefinitions>  
  <TextBox Margin="10" Grid.Row="0" Grid.Column="0" Grid.ColumnSpan="3">
```

```

This is a test.</TextBox>
<Button Margin="10,10,2,10" Padding="3"
        Grid.Row="1" Grid.Column="1">OK</Button>
<Button Margin="2,10,10,10" Padding="3"
        Grid.Row="1" Grid.Column="2">Cancel</Button>
</Grid>

```

大多数开发人员认为这种布局不清晰也不明智。列宽由窗口底部的两个按钮的尺寸决定，这使得难以向已经存在的 Grid 结构中添加新内容。即使向这个窗口增加很少的内容，也必须创建新的列集合。

正如上面所显示的，当为窗口选择布局容器时，不仅关心能否得到正确的布局行为——还希望构建便于在未来维护和增强的布局结构。一条正确的经验法则是，对于一次性的布局任务，例如排列一组按钮，使用更小的布局容器(如 StackPanel)。但如果需要为窗口中的多个区域使用一致的结构(比如稍后在图 3-22 中演示的一列文本框)，对于标准化布局而言，Grid 面板是必不可少的工具。

3.4.4 分割窗口

每个 Windows 用户都见过分割条——能将窗口的一部分与另一部分分离的可拖动分割器。例如，当使用 Windows 资源管理器时，会看到一系列文件夹(在左边)和一系列文件(在右边)。可拖动它们之间的分割条来确定每部分占据窗口的比例。

在 WPF 中，分割条由 GridSplitter 类表示，它是 Grid 面板的功能之一。通过为 Grid 面板添加 GridSplitter 对象，用户就可以改变行和列的尺寸。图 3-15 显示了一个窗口，在该窗口中有一个 GridSplitter 对象，它位于 Grid 面板的两列之间。通过拖动分割条，用户可以改变两列的相对宽度。

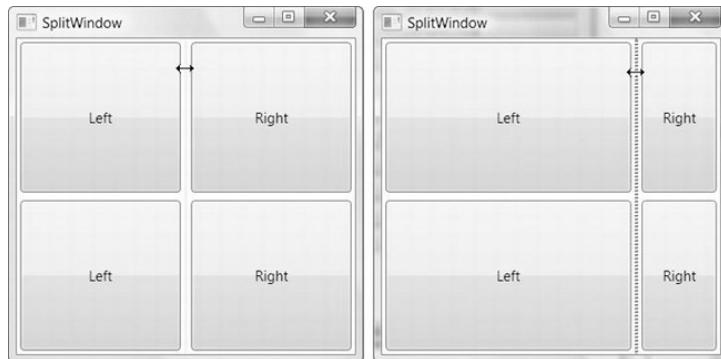


图 3-15 移动分割条

大多数开发人员认为 WPF 中的 GridSplitter 类不是最直观的。理解如何使用 GridSplitter 类，从而得到所期望的效果需要一定的经验。下面列出几条指导原则：

- GridSplitter 对象必须放在 Grid 单元格中。可与已经存在的内容一并放到单元格中，这时需要调整边距设置，使它们不相互重叠。更好的方法是预留一列或一行专门用于放置 GridSplitter 对象，并将预留行或列的 Height 或 Width 属性的值设置为 Auto。
- GridSplitter 对象总是改变整行或整列的尺寸(而非改变单个单元格的尺寸)。为使 GridSplitter 对象的外观和行为保持一致，需要拉伸 GridSplitter 对象使其穿越整行或整列，而不是将其限制在单元格中。为此，可使用前面介绍过的 RowSpan 或 ColumnSpan

属性。例如，图 3-15 中 GridSplitter 对象的 RowSpan 属性被设置为 2，因此被拉伸充满整列。如果不使用该设置，GridSplitter 对象会显示在顶行(放置它的行)中，即使这样，拖动分割条时也会改变整列的尺寸。

- 最初，GridSplitter 对象很小不易看见。为了使其更可用，需要为其设置最小尺寸。对于竖直分割条(图 3-15 中显示的分割条)，需要将 VerticalAlignment 属性设置为 Stretch(使分割条填满区域的整个高度)，并将 Width 设置为固定值(如 10 个设备无关单位)。对于水平分割条，需要设置 HorizontalAlignment 属性来拉伸，并将 Height 属性设置为固定值。
- GridSplitter 对齐方式还决定了分割条是水平的(用于改变行的尺寸)还是竖直的(用于改变列的尺寸)。对于水平分割条，需要将 VerticalAlignment 属性设置为 Center(这也是默认值)，以指明拖动分割条改变上面行和下面行的尺寸。对于竖直分割条(图 3-15 中显示的分割条)，需要将 HorizontalAlignment 属性设置为 Center，以改变分割条两侧列的尺寸。

注意：

可使用 GridSplitter 对象的 ResizeDirection 和 ResizeBehavior 属性修改其尺寸调整行为。然而，使这一行为完全取决于对齐方式将更简单，这也是默认设置。

您是不是觉得茫然了？为了进一步强化这些规则，分析一下在图 3-15 中所演示程序的实际标记是有帮助的。下面的程序清单以加粗形式显示了 GridSplitter 对象的细节：

```
<Grid>
<Grid.RowDefinitions>
<RowDefinition></RowDefinition>
<RowDefinition></RowDefinition>
</Grid.RowDefinitions>
<Grid.ColumnDefinitions>
<ColumnDefinition MinWidth="100"></ColumnDefinition>
<ColumnDefinition Width="Auto"></ColumnDefinition>
<ColumnDefinition MinWidth="50"></ColumnDefinition>
</Grid.ColumnDefinitions>

<Button Grid.Row="0" Grid.Column="0" Margin="3">Left</Button>
<Button Grid.Row="0" Grid.Column="2" Margin="3">Right</Button>
<Button Grid.Row="1" Grid.Column="0" Margin="3">Left</Button>
<Button Grid.Row="1" Grid.Column="2" Margin="3">Right</Button>

<GridSplitter Grid.Row="0" Grid.Column="1" Grid.RowSpan="2"
Width="3" VerticalAlignment="Stretch" HorizontalAlignment="Center"
ShowsPreview="False"></GridSplitter>
</Grid>
```

提示：

为了成功地创建 GridSplitter 对象，务必为 VerticalAlignment、HorizontalAlignment 以及 Width 属性(或 Height 属性)提供相应的属性值。

上面的标记还包含了一处额外的细节。在声明 GridSplitter 对象时，将 ShowsPreview 属性

设置为 `false`。因此，当把分割条从一边拖到另一边时，会立即改变列的尺寸。但是如果将 `ShowsPreview` 属性设置为 `true`，当拖动分割条时就会看到一个灰色的阴影跟随鼠标指针，用于显示将在何处进行分割。并且直到释放了鼠标键之后列的尺寸才改变。如果 `GridSplitter` 对象获得了焦点，也可以使用箭头键改变相应的尺寸。

`ShowsPreview` 不是唯一可设置的 `GridSplitter` 属性。如果希望分割条以更大的幅度(如每次 10 个单位)进行移动，可调整 `DragIncrement` 属性。如果希望控制列的最大尺寸和最小尺寸，只需要在 `ColumnDefinitions` 部分设置合适的属性，如上面的示例所演示的那样。

提示：

可以改变 `GridSplitter` 对象的填充方式，使其不只是具有阴影的灰色矩形。技巧是使用 `Background` 属性应用填充，该属性接受简单的颜色或更复杂的画刷。

Grid 面板通常包含多个 `GridSplitter` 对象。然而，可以在一个 Grid 面板中嵌套另一个 Grid 面板；而且，如果确实在 Grid 面板中嵌套了 Grid 面板，那么每个 Grid 面板可以有自己的 `GridSplitter` 对象。这样就可以创建被分割成两部分(如左边窗格和右边窗格)的窗口，然后将这些区域(如右边的窗格)进一步分成更多的部分(例如，可调整大小的上下两部分)。图 3-16 演示了这个示例。

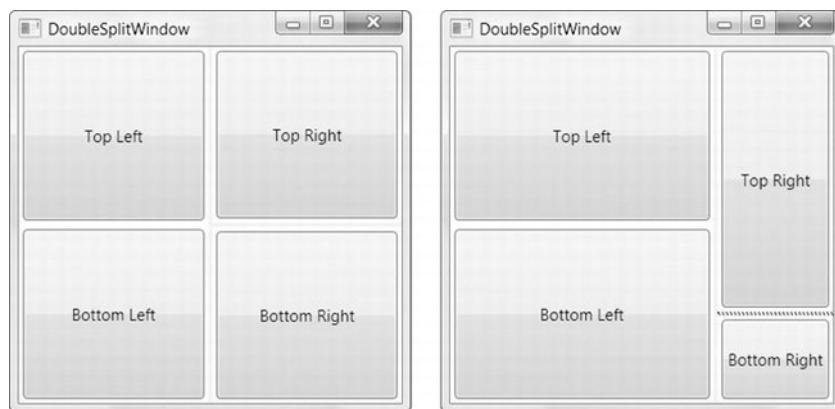


图 3-16 使用两个分割条调整窗口大小

这个窗口创建起来非常简单，尽管为了保持跟踪涉及的三个 Grid 容器有些繁杂：整体 Grid 面板、在左边嵌套的 Grid 面板和在右边嵌套的 Grid 面板。唯一的技巧是确保将 `GridSplitter` 放到正确的单元格中，并设置正确的对齐方式。下面是完整的标记：

```
<!-- This is the Grid for the entire window. -->
<Grid>
    <Grid.ColumnDefinitions>
        <ColumnDefinition></ColumnDefinition>
        <ColumnDefinition Width="Auto"></ColumnDefinition>
        <ColumnDefinition></ColumnDefinition>
    </Grid.ColumnDefinitions>

    <!-- This is the nested Grid on the left. -->
    <Grid>
        <Grid.RowDefinitions>
            <RowDefinition Height="Auto"></RowDefinition>
            <RowDefinition Height="Auto"></RowDefinition>
        </Grid.RowDefinitions>
        <GridSplitter HorizontalAlignment="Right" VerticalAlignment="Stretch" Width="10" />
        <Grid>
            <Grid.ColumnDefinitions>
                <ColumnDefinition Width="*"/>
                <ColumnDefinition Width="*"/>
            </Grid.ColumnDefinitions>
            <Text>Top Left</Text>
            <Text>Top Right</Text>
        </Grid>
        <GridSplitter HorizontalAlignment="Left" VerticalAlignment="Bottom" Width="10" />
        <Grid>
            <Grid.ColumnDefinitions>
                <ColumnDefinition Width="*"/>
                <ColumnDefinition Width="*"/>
            </Grid.ColumnDefinitions>
            <Text>Bottom Left</Text>
            <Text>Bottom Right</Text>
        </Grid>
    </Grid>
</Grid>
```

```

It isn't subdivided further with a splitter. -->
<Grid Grid.Column="0" VerticalAlignment="Stretch">
  <Grid.RowDefinitions>
    <RowDefinition></RowDefinition>
    <RowDefinition></RowDefinition>
  </Grid.RowDefinitions>
  <Button Margin="3" Grid.Row="0">Top Left</Button>
  <Button Margin="3" Grid.Row="1">Bottom Left</Button>
</Grid>

<!-- This is the vertical splitter that sits between the two nested
(left and right) grids. -->
<GridSplitter Grid.Column="1"
  Width="3" HorizontalAlignment="Center" VerticalAlignment="Stretch"
  ShowsPreview="False"></GridSplitter>

<!-- This is the nested Grid on the right. -->
<Grid Grid.Column="2">
  <Grid.RowDefinitions>
    <RowDefinition></RowDefinition>
    <RowDefinition Height="Auto"></RowDefinition>
    <RowDefinition></RowDefinition>
  </Grid.RowDefinitions>

  <Button Grid.Row="0" Margin="3">Top Right</Button>
  <Button Grid.Row="2" Margin="3">Bottom Right</Button>

  <!-- This is the horizontal splitter that subdivides it into
  a top and bottom region.. -->
  <GridSplitter Grid.Row="1"
    Height="3" VerticalAlignment="Center" HorizontalAlignment="Stretch"
    ShowsPreview="False"></GridSplitter>
</Grid>
</Grid>
```

提示：

请记住，如果 Grid 面板只有一行或一列，可忽略 RowDefinitions 部分。而且，对于没有明确设置行位置的元素，假定其 Grid.Row 属性值为 0，并被放到第一行中。没有提供 Grid.Column 属性值的元素被放置到第一列中。

3.4.5 共享尺寸组

正如在前面看到的，Grid 面板包含一个行列集合，可以明确地按比例确定行和列的尺寸，或根据其子元素的尺寸确定行和列的尺寸。还有另一种确定一行或一列尺寸的方法——与其他行或列的尺寸相匹配。这是通过称为“共享尺寸组”(shared size groups)的特性实现的。

共享尺寸组的目标是保持用户界面独立部分的一致性。例如，可能希望改变一列的尺寸以适应其内容，并改变另一列的尺寸使其与前面一列改变后的尺寸相匹配。然而，共享尺寸组的真正优点是使独立的 Grid 控件具有相同的比例。

为了理解共享尺寸组的工作原理，分析一下图 3-17 所演示的示例。该窗口具有两个 Grid 对象——一个位于窗口顶部(有三列)，另一个位于窗口底部(有两列)。第一个 Grid 面板的最左边一列按比例地改变其尺寸，以适应其包含的内容(一个较长的文本字符串)。第二个 Grid 面板的最左边一列和第一个 Grid 面板的最左边一列的宽度完全相同，但包含的内容较少。这是因为它们共享相同的尺寸分组。不管在第一个 Grid 面板的第一列中放置了多少内容，第二个 Grid 面板中的第一列总是和第一个 Grid 面板中的第一列保持同步。

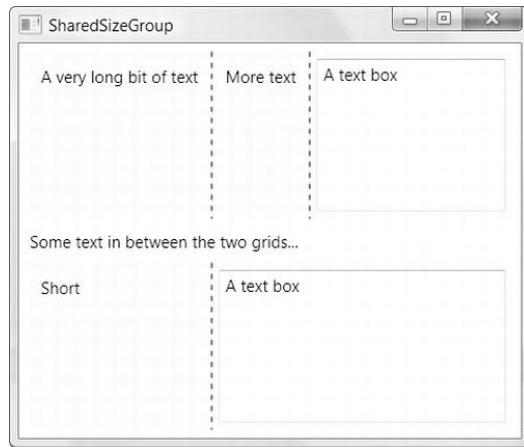


图 3-17 两个网格共享同一列定义

正如本例所演示的，共享的列可用于不同的网格中。在这个示例中，顶部的 Grid 面板有一个特别的列，从而剩余的空间以不同方式进行分割。同样，共享的列可占据不同的位置，从而可以在一个 Grid 面板中的第一列和另一个 Grid 面板中的第二列之间创建一种联系。显然，这些列可包含完全不同的内容。

当使用共享尺寸组时，就像创建了一列(或一行)的定义，列定义(或行定义)在多个地方被重复使用。这不是简单地将一列(或一行)复制到另外一个地方。可以使用前面的示例，通过改变第二个 Grid 面板中共享列的内容来对此进行测试。现在，第一个 Grid 面板中的列会被拉长以进行匹配(见图 3-18)。

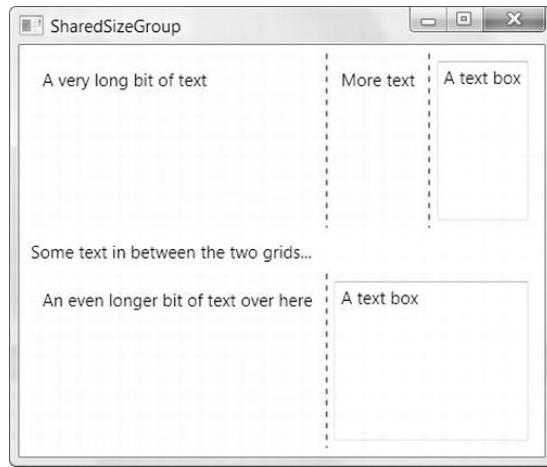


图 3-18 共享尺寸的列保持同步

甚至可为其中一个 Grid 对象添加 GridSplitter。当用户改变一个 Grid 面板中列的尺寸时，另一个 Grid 面板中的共享列会同时相应地改变尺寸。

可轻而易举地创建共享组。只需要使用对应的字符串设置两列的 SharedSizeGroup 属性即可。在当前示例中，两列都使用了名为 TextLabel 的分组：

```
<Grid Margin="3" Background="LightYellow" ShowGridLines="True">
  <Grid.ColumnDefinitions>
    <ColumnDefinition Width="Auto" SharedSizeGroup="TextLabel"></ColumnDefinition>
    <ColumnDefinition Width="Auto"></ColumnDefinition>
    <ColumnDefinition></ColumnDefinition>
  </Grid.ColumnDefinitions>

  <Label Margin="5">A very long bit of text</Label>
  <Label Grid.Column="1" Margin="5">More text</Label>
  <TextBox Grid.Column="2" Margin="5">A text box</TextBox>
</Grid>
...
<Grid Margin="3" Background="LightYellow" ShowGridLines="True">
  <Grid.ColumnDefinitions>
    <ColumnDefinition Width="Auto" SharedSizeGroup="TextLabel"></ColumnDefinition>
    <ColumnDefinition></ColumnDefinition>
  </Grid.ColumnDefinitions>

  <Label Margin="5">Short</Label>
  <TextBox Grid.Column="1" Margin="5">A text box</TextBox>
</Grid>
```

还有一个细节，对于整个应用程序来说，共享尺寸组并不是全局的，因为多个窗口可能在无意间使用相同名称。可以假定共享尺寸组被限制在当前窗口，但是 WPF 甚至更加严格。为了共享一个组，需要在包含具有共享列的 Grid 对象的容器中，在包含 Grid 对象之前明确地将 Grid.IsSharedSizeScope 附加属性设置为 true。为了实现该目标，在当前这个示例中，将顶部和底部的 Grid 面板包含在另一个 Grid 面板中。当然，也可以很方便地使用不同的容器，如 DockPanel 或 StackPanel。

下面是顶级 Grid 面板的标记：

```
<Grid Grid.IsSharedSizeScope="True" Margin="3">
  <Grid.RowDefinitions>
    <RowDefinition></RowDefinition>
    <RowDefinition Height="Auto"></RowDefinition>
    <RowDefinition></RowDefinition>
  </Grid.RowDefinitions>

  <Grid Grid.Row="0" Margin="3" Background="LightYellow" ShowGridLines="True">
    ...
  </Grid>
  <Label Grid.Row="1" >Some text in between the two grids...</Label>
  <Grid Grid.Row="2" Margin="3" Background="LightYellow" ShowGridLines="True">
    ...
  </Grid>
```

```
</Grid>
```

提示:

可使用共享尺寸组来同步具有列标题的不同网格。每列的宽度可由列中的内容决定，列标题将共享这一宽度。甚至可在列标题中放置一个 GridSplitter 对象，用户可通过拖动该 GridSplitter 对象来改变列标题以及列标题下的整个列。

3.4.6 UniformGrid 面板

有一种网格不遵循前面讨论的所有原则——UniformGrid 面板。与 Grid 面板不同，UniformGrid 面板不需要(甚至不支持)预先定义的列和行。相反，通过简单地设置 Rows 和 Columns 属性来设置其尺寸。每个单元格始终具有相同的大小，因为可用的空间被均分。最后，元素根据定义的顺序被放置到适当的单元格中。UniformGrid 面板中没有 Row 和 Column 附加属性，也没有空白单元格。

下面列举一个示例，该例使用 4 个按钮填充 UniformGrid 面板：

```
<UniformGrid Rows="2" Columns="2">
  <Button>Top Left</Button>
  <Button>Top Right</Button>
  <Button>Bottom Left</Button>
  <Button>Bottom Right</Button>
</UniformGrid>
```

与 Grid 面板相比，UniformGrid 面板很少使用。Grid 面板是用于创建简单乃至复杂窗口布局的通用工具。UniformGrid 面板是一种更特殊的布局容器，主要用于在刻板的网格中快速地布局元素(例如，为特定游戏构建播放面板)。许多 WPF 开发人员可能永远不会使用 UniformGrid 面板。

3.5 使用 Canvas 面板进行基于坐标的布局

到目前为止唯一尚未介绍的布局容器是 Canvas 面板。Canvas 面板允许使用精确的坐标放置元素，如果设计数据驱动的富窗体和标准对话框，这并非好的选择；但如果需要构建其他一些不同的内容(例如，为图形工具创建绘图表面)，Canvas 面板可能是个有用的工具。Canvas 面板还是最轻量级的布局容器。这是因为 Canvas 面板没有包含任何复杂的布局逻辑，用以改变其子元素的首选尺寸。Canvas 面板只是在指定的位置放置其子元素，并且其子元素具有所希望的精确尺寸。

为在 Canvas 面板中定位元素，需要设置 Canvas.Left 和 Canvas.Top 附加属性。Canvas.Left 属性设置元素左边和 Canvas 面板左边之间的单位数，Canvas.Top 属性设置子元素顶边和 Canvas 面板顶边之间的单位数。同样，这些数值也是以设备无关单位设置的，当将系统 DPI 设置为 96 dpi 时，设备无关单位恰好等于通常的像素。

注意:

另外，可使用 Canvas.Right 属性而不是 Canvas.Left 属性来确定元素和 Canvas 面板右边缘间的距离；可使用 Canvas.Bottom 属性而不是 Canvas.Top 属性来确定元素和 Canvas 面板底部边缘的距离。不能同时使用 Canvas.Right 和 Canvas.Left 属性，也不能同时使用 Canvas.Top 和 Canvas.Bottom 属性。

可使用 `Width` 和 `Height` 属性明确设置子元素的尺寸。与使用其他面板相比，使用 `Canvas` 面板时这种设置更普遍，因为 `Canvas` 面板没有自己的布局逻辑(并且当需要精确控制组合元素如何排列时，经常会使用 `Canvas` 面板)。如果没有设置 `Width` 和 `Height` 属性，元素会获取它所期望的尺寸——换句话说，它将变得足够大以适应其内容。

下面是一个包含 4 个按钮的简单 `Canvas` 面板示例：

```
<Canvas>
    <Button Canvas.Left="10" Canvas.Top="10">(10,10)</Button>
    <Button Canvas.Left="120" Canvas.Top="30">(120,30)</Button>
    <Button Canvas.Left="60" Canvas.Top="80" Width="50" Height="50">
        (60,80)</Button>
    <Button Canvas.Left="70" Canvas.Top="120" Width="100" Height="50">
        (70,120)</Button>
</Canvas>
```

图 3-19 显示了结果。

如果改变窗口的大小，`Canvas` 面板就会拉伸以填满可用空间，但 `Canvas` 面板上的控件不会改变其尺寸和位置。`Canvas` 面板不包含任何锚定和停靠功能，这两个功能是在 Windows 窗体中使用坐标布局提供的。造成该问题的部分原因是为了保持 `Canvas` 面板的轻量级，另一个原因是防止以不当目的使用 `Canvas` 面板(例如，确定标准用户界面的布局)。

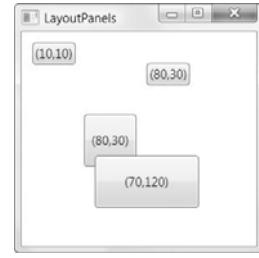


图 3-19 在 `Canvas` 面板中明确地定位按钮

与其他所有布局容器一样，可在用户界面中嵌套 `Canvas` 面板。这意味着可使用 `Canvas` 面板在窗口的一部分绘制一些细节内容，而在窗口的其余部分使用更合乎标准的 WPF 面板。

提示：

如果与其他元素一起使用 `Canvas` 面板，可能希望将它的 `ClipToBounds` 属性设置为 `true`。这样，如果 `Canvas` 面板中的元素被拉伸超出 `Canvas` 面板的边界，将在 `Canvas` 面板的边缘处剪裁这些子元素(这样可以阻止它们与窗口中的其他元素重叠)。其他所有布局容器总是剪裁它们的子元素以适应其尺寸，而不考虑 `ClipToBounds` 的设置。

3.5.1 Z 顺序

如果 `Canvas` 面板中有多个互相重叠的元素，可通过设置 `Canvas.ZIndex` 附加属性来控制它们的层叠方式。

添加的所有元素通常都具有相同的 `ZIndex` 值——0。如果元素具有相同的 `ZIndex` 值，就按它们在 `Canvas.Children` 集合中的顺序进行显示，这个顺序依赖于元素在 XAML 标记中定义的顺序。在标记中靠后位置声明的元素(如按钮(70, 120))会显示在前面声明的元素(如按钮(120, 30))的上面。

然而，可通过增加任何子元素的 `ZIndex` 值来提高层次级别。因为具有更高 `ZIndex` 值的元素始终显示在较低 `ZIndex` 值的元素的上面。使用这一技术，可改变前一示例中的分层情况：

```
<Button Canvas.Left="60" Canvas.Top="80" Canvas.ZIndex="1" Width="50" Height="50">
    (60,80)</Button>
<Button Canvas.Left="70" Canvas.Top="120" Width="100" Height="50">
    (70,120)</Button>
```

注意：

应用于 Canvas.ZIndex 属性的实际值并无意义。重要细节的是一个元素的 ZIndex 值和另一个元素的 ZIndex 值相比较如何。可将 ZIndex 属性设置为任何正整数或负整数。

如果需要通过代码来改变元素的位置，ZIndex 属性是非常有用的。只需要调用 Canvas.SetZIndex()方法，并传递希望修改的元素和希望使用的新 ZIndex 值即可。遗憾的是，并不存在 BringToFront()或 SendToBack()方法——要实现这一行为，需要跟踪最高和最低的 ZIndex 值。

3.5.2 InkCanvas 元素

WPF 还提供了 InkCanvas 元素，它与 Canvas 面板在某些方面是类似的(而在其他方面却完全不同)。和 Canvas 面板一样，InkCanvas 元素定义了 4 个附加属性(Top、Left、Bottom 和 Right)，可将这 4 个附加属性应用于子元素，以根据坐标进行定位。然而，基本的内容区别很大——实际上，InkCanvas 类不是派生自 Canvas 类，甚至也不是派生自 Panel 基类，而是直接派生自 FrameworkElement 类。

InkCanvas 元素的主要目的是用于接收手写笔输入。手写笔是一种在平板 PC 中使用的类似于钢笔的输入设备，然而，InkCanvas 元素同时也可使用鼠标进行工作，就像使用手写笔一样。因此，用户可使用鼠标在 InkCanvas 元素上绘制线条，或者选择以及操作 InkCanvas 中的元素。

InkCanvas 元素实际上包含两个子内容集合。一个是为人熟知的 Children 集合，它保存任意元素，就像 Canvas 面板一样。每个子元素可根据 Top、Left、Bottom 和 Right 属性进行定位。另一个是 Strokes 集合，它保存 System.Windows.Ink.Stroke 对象，该对象表示用户在 InkCanvas 元素上绘制的图形输入。用户绘制的每条直线或曲线都变成独立的 Stroke 对象。得益于这两个集合，可使用 InkCanvas 让用户使用存储在 Strokes 集合中的笔画(stroke)为保存在 Children 集合中的内容添加注释。

例如，图 3-20 显示了包含一幅图片的 InkCanvas 元素，这幅图片已经使用附加的笔画注释过。下面是这个示例中定义 InkCanvas 的标记，这些标记定义了图像。



图 3-20 在 InkCanvas 元素中添加笔画

```
<InkCanvas Name="inkCanvas" Background="LightYellow"
    EditingMode="Ink">
```

```
<Image Source="office.jpg" InkCanvas.Top="10" InkCanvas.Left="10"
Width="287" Height="319"></Image>
</InkCanvas>
```

笔画是用户在运行时绘制的。

根据为 InkCanvas.EditingMode 属性设置的值，可以采用截然不同的方式使用 InkCanvas 元素。表 3-5 列出了所有选项。

表 3-5 InkCanvasEditingStyle 枚举值

名 称	说 明
Ink	InkCanvas 元素允许用户绘制批注，这是默认模式。当用户用鼠标或手写笔绘图时，会绘制笔画
GestureOnly	InkCanvas 元素不允许用户绘制笔画批注，但会关注预先定义的特定姿势(例如在某个方向拖动手写笔或涂画内容)。能识别的姿势的完整列表由 System.Windows.Ink.ApplicationGesture 枚举给出
InkAndGesture	InkCanvas 元素允许用户绘制笔画批注，也可以识别预先定义的姿势
EraseByStroke	当单击笔画时，InkCanvas 元素会擦除笔画。如果用户使用手写笔，可使用手写笔的底端切换到该模式(可使用只读的 ActiveEditingStyle 属性确定当前编辑模式，也可通过改变 EditingModeInverted 属性来改变手写笔的底端使用的工作模式)
EraseByPoint	当单击笔画时，InkCanvas 元素会擦除笔画中被单击的部分(笔画上的一个点)
Select	InkCanvas 面板允许用户选择保存在 Children 集合中的元素。要选择一个元素，用户必须单击该元素或拖动“套索”选择该元素。一旦选择一个元素，就可以移动该元素、改变其尺寸或将其删除
None	InkCanvas 元素忽略鼠标和手写笔输入

InkCanvas 元素会引发多种事件，当编辑模式改变时会引发 ActiveEditingStyleChanged 事件，在 GestureOnly 或 InkAndGesture 模式下删除姿势时会引发 Gesture 事件，绘制完笔画时会引发 StrokeCollected 事件，擦除笔画时会引发 StrokeErasing 事件和 StrokeErased 事件，在 Select 模式下选择元素或改变元素时会引发 SelectionChanging 事件、SelectionChanged 事件、SelectionMoving 事件、SelectionMoved 事件、SelectionResizing 事件和 SelectionResized 事件。其中，名称以“ing”结尾的事件表示动作将要发生，但可以通过设置 EventArgs 对象的 Cancel 属性取消事件。

在 Select 模式下，InkCanvas 元素可为拖动以及操作内容提供功能强大的设计界面。图 3-21 显示了 InkCanvas 元素中的一个按钮控件，左图中显示的是该按钮被选中的情况，而右图中显示的是选中该按钮后，改变其位置和尺寸的情况。

虽然 Select 模式十分有趣，但并不适合用于构建绘图工具。第 14 章将列举创建自定义绘图界面的更好示例。

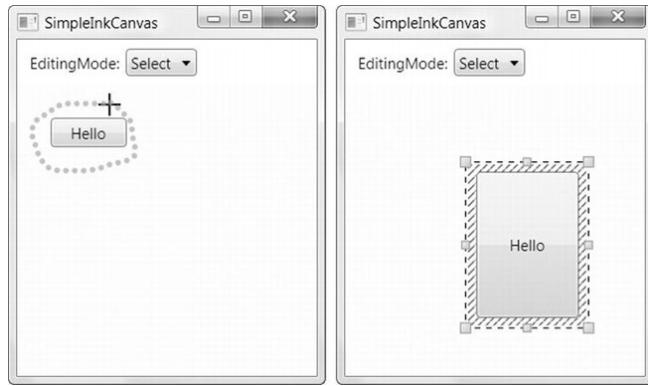


图 3-21 在 InkCanvas 中移动元素并调整其尺寸

3.6 布局示例

前面已经占用了相当大的篇幅研究有关 WPF 布局容器的复杂内容。在掌握了这些基础知识后，就可以研究几个完整的布局示例了。通过研究完整的布局示例，可更好地理解各种 WPF 布局概念(例如，根据内容改变尺寸、拉伸以及嵌套等)在实际窗口中的工作方式。

3.6.1 列设置

布局容器(如 Grid 面板)使得为窗口创建整个布局结构变得非常容易。例如，分析图 3-22 中显示的窗口及设置。该窗口在一个表格结构中排列各个组件——标签、文本框以及按钮。

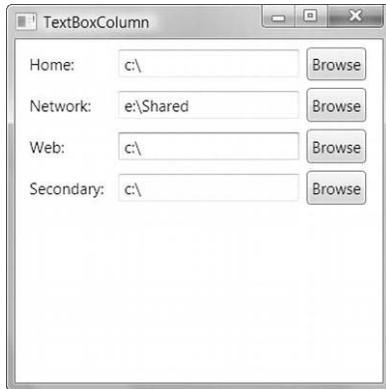


图 3-22 列中的文件夹设置

为创建这一表格，首先定义网格的行和列。行定义足够简单——只需要将每行的尺寸设置为所含内容的高度。这意味着所有行都将使用最大元素的高度，在该例中，最大的元素是第三列中的 Browse 按钮。

```
<Grid Margin="3,3,10,3">
<Grid.RowDefinitions>
<RowDefinition Height="Auto"/></RowDefinition>
<RowDefinition Height="Auto"/></RowDefinition>
<RowDefinition Height="Auto"/></RowDefinition>
<RowDefinition Height="Auto"/></RowDefinition>
```

```

</Grid.RowDefinitions>
...

```

接下来需要创建列。第一列和最后一列的尺寸要适合其内容(分别是标签文本和 Browse 按钮)。中间列占用所有剩余空间，这意味着当窗口变大时，该列的尺寸会增加，这样可有更大的空间显示选择的文件夹(如果希望拉伸不超过一定的最大宽度，在定义列时可使用 MaxWidth 属性，就像对单个元素使用 MaxWidth 属性一样)。

```

...
<Grid.ColumnDefinitions>
<ColumnDefinition Width="Auto"></ColumnDefinition>
<ColumnDefinition Width="*"></ColumnDefinition>
<ColumnDefinition Width="Auto"></ColumnDefinition>
</Grid.ColumnDefinitions>
...

```

提示：

Grid 面板需要一定的最小空间——要足以容纳整个标签文本、浏览按钮以及在中间列中有一定的像素以显示文本框。如果缩小包含窗口使其小于这一最小空间，就会将一些内容剪裁掉。与通常的情形一样，使用窗口的 MinWidth 和 MinHeight 属性防止这种情况的发生是有意义的。

现在已经具备了基本结构，接下来只需要在恰当的单元格中放置元素。然而，还需要仔细考虑边距和对齐方式。每个元素需要基本的边距(3 个单位较恰当)以在其周围添加一些空间。此外，标签和文本框在垂直方向上需要居中，因为它们没有 Browse 按钮高。最后，文本框需要使用自动设置尺寸模式，这样它会被拉伸以充满整列。

下面是定义网格的第一行所需要的标记：

```

...
<Label Grid.Row="0" Grid.Column="0" Margin="3"
VerticalAlignment="Center">Home:</Label>
<TextBox Grid.Row="0" Grid.Column="1" Margin="3"
Height="Auto" VerticalAlignment="Center"></TextBox>
<Button Grid.Row="0" Grid.Column="2" Margin="3" Padding="2">Browse</Button>
...
</Grid>

```

可重复上面的标记，并简单地递增 Grid.Row 特性的值来添加所有行。

一个不是非常明显的事是，因为使用了 Grid 控件，所以该窗口是非常灵活的。没有任何一个元素——标签、文本框以及按钮——是通过硬编码来定位和设置尺寸的。因此，可通过简单地修改 ColumnDefinition 元素来快速改变整个网格。甚至，如果添加了包含更长标签文本的行(迫使第一列更宽)，就会调整整个网格使其保持一致，包括已经添加的行。如果希望在两行之间添加元素——例如，添加分割线以区分窗口的不同部分——可保持网格的列定义不变，但使用 ColumnSpan 属性拉伸某个元素，使其覆盖更大的区域。

3.6.2 动态内容

与上面演示的列设置一样，当修订应用程序时，可方便地修改使用 WPF 布局容器的窗口，并且可以很容易地使窗口适应对应用程序的修订。这样的灵活性不仅能使开发人员在设计时受益，而且如果需要显示在运行时变化很大的内容，这也是非常有用的。

一个例子是本地化文本——对于不同的地域，在用户界面中显示的文本需要翻译成不同的语言。在老式的基于坐标的应用程序中，改变窗口中的文本会造成混乱，部分原因是少量英语文本翻译成许多语言后会变得特别大。尽管允许改变元素的尺寸以适应更大的文本，但这样做经常使整个窗口失去平衡。

图 3-23 演示了 WPF 布局控件是如何聪明地解决这一问题的。在这个示例中，用户界面可选择短文本和长文本。当使用长文本时，包含文本的按钮会自动改变其尺寸，而其他内容也会相应地调整位置。并且因为改变了尺寸的按钮共享同一布局容器(在该例中是一个表格列)，所以整个用户界面都会改变尺寸。最终结果是所有按钮保持一致的尺寸——最大按钮的尺寸。

为实现这个效果，窗口使用一个具有两行两列的表格进行分割。左边的列包含可改变大小的按钮，而右边的列包含文本框。底行用于放置 Close 按钮，底行和顶行位于同一表格中，从而可以根据顶行改变尺寸。

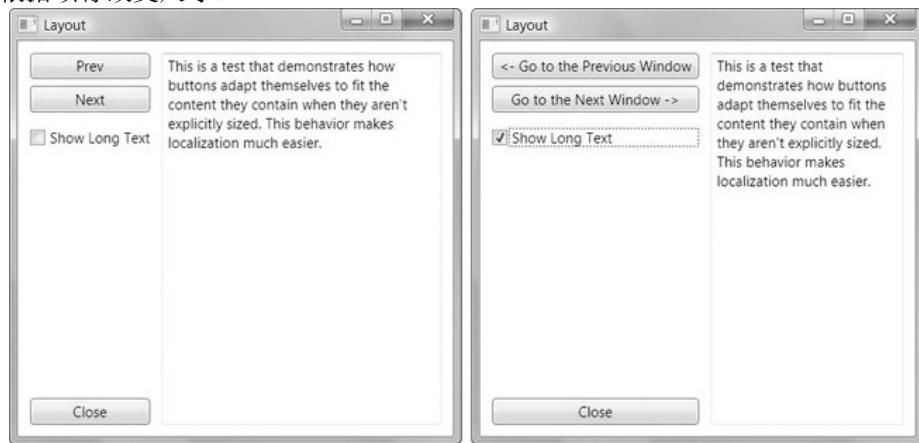


图 3-23 自适应窗口

下面是完整的标记：

```

<Grid>
    <Grid.RowDefinitions>
        <RowDefinition Height="*" />
        <RowDefinition Height="Auto" />
    </Grid.RowDefinitions>
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="Auto" />
        <ColumnDefinition Width="*" />
    </Grid.ColumnDefinitions>

    <StackPanel Grid.Row="0" Grid.Column="0">
        <Button Name="cmdPrev" Margin="10,10,10,3">Prev</Button>
        <Button Name="cmdNext" Margin="10,3,10,3">Next</Button>
        <CheckBox Name="chkLongText" Margin="10,10,10,10"
            Checked="chkLongText_Checked" Unchecked="chkLongText_Unchecked">
            Show Long Text</CheckBox>
    </StackPanel>
    <TextBlock Grid.Row="0" Grid.Column="1" Text="This is a test that demonstrates how buttons adapt themselves to fit the content they contain when they aren't explicitly sized. This behavior makes localization much easier." />
    <TextBlock Grid.Row="1" Grid.Column="1" Text="This is a test that demonstrates how buttons adapt themselves to fit the content they contain when they aren't explicitly sized. This behavior makes localization much easier." />

```

```

<TextBox Grid.Row="0" Grid.Column="1" Margin="0,10,10,10"
    TextWrapping="WrapWithOverflow" Grid.RowSpan="2">This is a test that
demonstrates
how buttons adapt themselves to fit the content they contain when they aren't
explicitly sized. This behavior makes localization much easier.</TextBox>
<Button Grid.Row="1" Grid.Column="0" Name="cmdClose"
    Margin="10,3,10,10">Close</Button>
</Grid>

```

此处没有给出 CheckBox 控件的事件处理程序，它们只是改变两个按钮中的文本。

3.6.3 组合式用户界面

许多布局容器(如 StackPanel 面板、DockPanel 面板以及 WrapPanel 面板)可以采用灵活多变的柔性方式非常得体地将内容安排到可用窗口空间中。该方法的优点是，它允许创建真正的组合式界面。换句话说，可在用户界面中希望显示的恰当部分插入不同的面板，而保留用户界面的其他部分。整个应用程序本身可以相应地改变界面，这与 Web 门户站点有类似之处。

图 3-24 演示了一个组合式用户界面。在一个 WrapPanel 面板中放置几个独立的面板。用户可以通过窗口顶部的复选框，选择显示这些面板中的哪些面板。

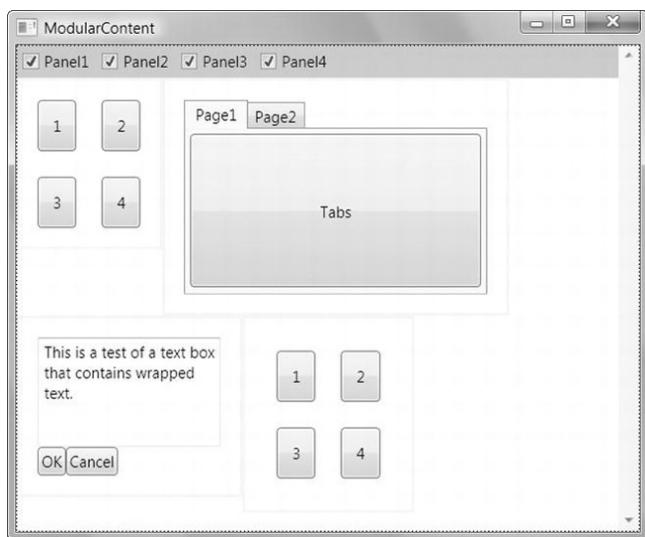


图 3-24 位于 WrapPanel 面板中的一系列面板

注意：

尽管可以设置布局面板的背景色，但不能在其周围设置边框。该例通过使用 Border 元素包含每个面板来突破这一限制，Border 元素恰好描述了面板的范围。

隐藏不同的面板后，剩余面板会重新改变自身以适应可用空间(以及声明它们的顺序)。图 3-25 显示了面板的不同排列方式。

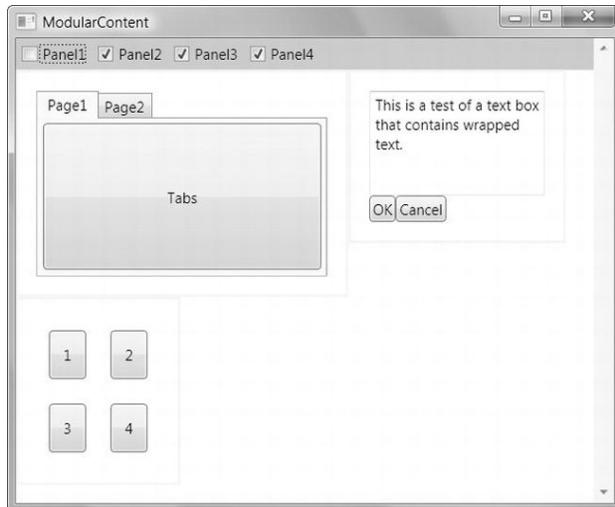


图 3-25 隐藏几个面板

为了隐藏和显示单个面板，需要使用一些代码处理复选框的单击事件。尽管尚未考虑 WPF 事件处理模型的任何细节(在第 5 章将完整介绍这些内容)，技巧是设置 Visibility 属性：

```
panel.Visibility = Visibility.Collapsed;
```

Visibility 属性是 UIElement 基类的一部分，因此放置于 WPF 窗口中的任何内容都支持该属性。该属性可使用三个值，它们来自 System.Windows.Visibility 枚举，如表 3-6 所示。

表 3-6 Visibility 枚举值

值	说 明
Visible	元素在窗口中正常显示
Collapsed	元素不显示，也不占用任何空间
Hidden	元素不显示，但仍为其保留空间(换句话说，会在元素可能显示的地方保留空白空间)。如果需要隐藏和显示元素，又不希望改变窗口布局和窗口中剩余元素的相对位置，使用该设置是非常方便的

提示：

可使用 Visibility 属性动态调整各种界面。例如，可制作在窗口一边显示的可折叠窗格。需要完成的全部工作就是在几种布局容器中包含窗格的所有内容，并恰当地设置其 Visibility 属性。剩余的内容会重新排列以适应余下的空间。

3.7 小结

本章详细介绍了 WPF 布局模型，并讨论了如何以堆栈、网格以及其他排列方式放置元素。可使用嵌套的布局容器组合创建更复杂的布局，可结合使用 GridSplitter 对象创建可变的分割窗口。本章一直非常关注这一巨大变化的原因——WPF 布局模型在保持、加强以及本地化用户界面方面所具有的优点。

布局内容远不止这些。接下来的几章还将列举更多使用布局容器组织元素分组的示例，还将学习允许在窗口中排列内容的几个附加功能：

- **特殊容器。**可以使用 ScrollViewer、TabItem 以及 Expander 控件滚动内容、将内容放到单独的选项卡中以及折叠内容。与布局面板不同，这些容器只能包含单一内容。不过，可以很容易地组合使用这些容器和布局面板，以便准确实现所需的效果。第 6 章将尝试使用这些容器。
- **Viewbox。**需要一种方法来改变图形内容(如图像和矢量图形)的尺寸吗？Viewbox 是另一种特殊容器，可帮助您解决这一问题，而且 Viewbox 控件内置了缩放功能。在第 12 章中，您将首次接触到 Viewbox 容器。
- **文本布局。**WPF 新增了用于确定大块格式化文本布局的新工具。可使用浮动图形和列表，并且可以使用分页、分列以及更复杂、更智能的换行功能来获得非常完美的结果。第 28 章将介绍文本布局的方式。